バージョン13

スクリプトガイド

「真の発見の旅とは、新しい風景を探すことではなく、新たな視点を持つことである。」
マルセル・プルースト

JMP, A Business Unit of SAS
SAS Campus Drive
Cary, NC 27513

13.1

このマニュアルを引用する場合は、次の正式表記を使用してください: SAS Institute Inc. 2017. 『JMP® 13 スクリプトガイド』Cary, NC: SAS Institute Inc.

JMP® 13 スクリプトガイド

SAS Institute Inc., SAS Campus Drive, Cary, North Carolina 27513-2414.

2016年9月

2017年2月

SASソフトウェアは、オープンソースのソフトウェアを含むがそれに限らない、特定のサードパーティ製ソフトウェアと共に提供される場合があります。かかるソフトウェアは、適用されるサードパーティソフトウェアライセンス契約に基づいてライセンスを得たものです。SASソフトウェアと共に配布されるサードパーティ製ソフトウェアに関する情報は、http://support.sas.com/thirdpartylicenses を参照してください。

テクノロジーライセンスに関する通知

- Scintilla - Copyright © 1998-2014 by Neil Hodgson <neilh@scintilla.org>.

  All Rights Reserved.

  何らかの目的でこのソフトウェアとそのマニュアルを手数料なしで使用、コピー、変更および配布することは、これをもって許可されます。ただし、すべてのコピーに上記の著作権に関する通知が記載されていること、および補助的なマニュアルに著作権に関する通知とこの許可に関する通知の両方が記載されていることを条件とします。

  NEIL HODGSONは、商業性および適合性の黙示的な保証を含め、このソフトウェアに関するすべての保証を放棄します。NEIL HODGSONは、いかなる場合においても、それが契約、過失、もしくは他の不法行為のどれであれ、このソフトウェアの使用もしくは性能から生じた、もしくはそれに関連して生じた使用、データ、もしくは利益の損失の結果として生じる特別損害、間接損害、もしくは付随的損害を始めとするいかなる損害に対しても責任を負いません。

- Telerik RadControls: Copyright © 2002-2012, Telerik. 含まれているTelerik RadControlsをJMP以外で使用することは許可されていません。
- ZLIB圧縮ライブラリ - Copyright © 1995-2005, Jean-Loup Gailly and Mark Adler.
- Natural Earthを使用して作成。無料のベクトルおよびラスター地図データ @ naturalearthdata.com.
- パッケージ - Copyright © 2009-2010, Stéphane Sudre (s.sudre.free.fr). All rights reserved.

  ソースおよびバイナリの形で、そのまま、もしくは変更を加えて再配布および使用することは、次のような条件を満たす限り、許可されます。

  再配布するソースコードには、上記の著作権に関する通知、この条件リスト、これに続く放棄声明が記載されていなければなりません。

  バイナリ形式で再配布する場合は、共に提供されるマニュアルなどの資料に上記の著作権に関する通知、この条件リスト、これに続く放棄声明が記載されていなければなりません。

  事前に書面による許可を得ることなく、このソフトウェアから派生した製品の推奨または宣伝のためにWhiteBoxの名前やその貢献者の名前を使用することはできません。

  このソフトウェアは、著作権保有者および貢献者によって「現状のままで」提供され、商業性および特定の目的に対する適合性に関する黙示的な保証を含むがそれに限らない、いかなる明示的もしくは黙示的な保証も行われません。いかなる場合においても、著作権保有者または貢献者は、損害の原因が何であれ、そして法的責任の根拠が何であれ、つまり、契約、厳格責任、不法行為（過失その他を含む）の

どれであれ、かかる損害の発生する可能性を事前に知らされていたとしても、このソフトウェアをどのように使用して生じた損害であれ、いかなる直接損害、間接損害、付随的損害、特別損害、懲罰的損害、もしくは結果損害（代替品または代替サービスの調達、使用機会、データもしくは利益の損失、業務の中断を含むがそれに限らない）に対しても責任を負いません。

- iODBC ソフトウェア - Copyright © 1995-2006, OpenLink Software Inc and Ke Jin (www.iodbc.org). All rights reserved.

  ソースおよびバイナリの形で、そのまま、もしくは変更を加えて再配布および使用することは、次のような条件を満たす限り、許可されます。

  – 再配布するソースコードには、上記の著作権に関する通知、この条件リスト、これに続く放棄声明が記載されていなければなりません。

  – バイナリ形式で再配布する場合は、共に提供されるマニュアルなどの資料に上記の著作権に関する通知、この条件リスト、これに続く放棄声明が記載されていなければなりません。

  – 事前に書面による許可を得ることなく、このソフトウェアから派生した製品の推奨または宣伝のために OpenLink Software Inc. の名前やその貢献者の名前を使用することはできません。

  このソフトウェアは、著作権保有者および貢献者によって「現状のままで」提供され、商業性および特定の目的に対する適合性に関する黙示的な保証を含むがそれに限らない、いかなる明示的もしくは黙示的な保証も行われません。いかなる場合においても、OPENLINK または貢献者は、損害の原因が何であれ、そして法的責任の根拠が何であれ、つまり、契約、厳格責任、不法行為（過失その他を含む）のどれであれ、かかる損害の発生する可能性を事前に知らされていたとしても、このソフトウェアをどのように使用して生じた損害であれ、いかなる直接損害、間接損害、付随的損害、特別損害、懲罰的損害、もしくは結果損害（代替品または代替サービスの調達、使用機会、データもしくは利益の損失、業務の中断を含むがそれに限らない）に対しても責任を負いません。

- bzip2、関連ライブラリの「libbzip2」、およびすべてのマニュアル: Copyright © 1996-2010, Julian R Seward. All rights reserved.

  ソースおよびバイナリの形で、そのまま、もしくは変更を加えて再配布および使用することは、次のような条件を満たす限り、許可されます。

  再配布するソースコードには、上記の著作権に関する通知、この条件リスト、これに続く放棄声明が記載されていなければなりません。

  このソフトウェアの供給源は正しく表記しなければならず、使用者が元のソフトウェアを記述したと主張することはできません。ある製品の中でこのソフトウェアを使用する場合は、その製品のマニュアルに謝辞を記載してもらえるとありがたいですが、必須ではありません。

  ソースに変更を加えたバージョンには、その旨を明記しなければならず、元のソフトウェアとは違うものであることを明確にしてください。

事前に書面による許可を得ることなく、このソフトウェアから派生した製品の推奨または宣伝のために作成者の名前を使用することはできません。

このソフトウェアは、作成者によって「現状のままで」提供され、商業性および特定の目的に対する適合性に関する黙示的な保証を含むがそれに限らない、いかなる明示的もしくは黙示的な保証も行われません。いかなる場合においても、作成者は、損害の原因が何であれ、そして法的責任の根拠が何であれ、つまり、契約、厳格責任、不法行為（過失その他を含む）のどれであれ、かかる損害の発生する可能性を事前に知らされていたとしても、このソフトウェアをどのように使用して生じた損害であれ、いかなる直接損害、間接損害、付随的損害、特別損害、懲罰的損害、もしくは結果損害（代替品または代替サービスの調達、使用機会、データもしくは利益の損失、業務の中断を含むがそれに限らない）に対しても責任を負いません。

- R ソフトウェア : Copyright © 1999-2012, R Foundation for Statistical Computing.
- MATLAB ソフトウェア : Copyright © 1984-2012, The MathWorks, Inc. 米国特許法および国際特許法によって保護されています。www.mathworks.com/patents を参照してください。MATLAB および Simulink は、The MathWorks, Inc. の登録商標です。他の商標は、www.mathworks.com/trademarks に一覧されています。他の製品名やブランド名は、それぞれの所有者の商標または登録商標である可能性があります。
- libopc: Copyright © 2011, Florian Reuter. All rights reserved.

  ソースおよびバイナリの形で、そのまま、もしくは変更を加えて再配布および使用することは、次のような条件を満たす限り、許可されます。

  – 再配布するソースコードには、上記の著作権に関する通知、この条件リスト、これに続く放棄声明が記載されていなければなりません。

  – バイナリ形式で再配布する場合は、共に提供されるマニュアルなどの資料に上記の著作権に関する通知、この条件リスト、これに続く放棄声明が記載されていなければなりません。

  – 事前に書面による許可を得ることなく、このソフトウェアから派生した製品の推奨または宣伝のために Florian Reuter の名前やその貢献者の名前を使用することはできません。

  このソフトウェアは、著作権保有者および貢献者によって「現状のままで」提供され、商業性および特定の目的に対する適合性に関する黙示的な保証を含むがそれに限らない、いかなる明示的もしくは黙示的な保証も行われません。いかなる場合においても、著作権保有者または貢献者は、損害の原因が何であれ、そして法的責任の根拠が何であれ、つまり、契約、厳格責任、不法行為（過失その他を含む）のどれであれ、かかる損害の発生する可能性を事前に知らされていたとしても、このソフトウェアをどのように使用して生じた損害であれ、いかなる直接損害、間接損害、付随的損害、特別損害、懲罰的損害、もしくは結果損害（代替品または代替サービスの調達、使用機会、データもしくは利益の損失、業務の中断を含むがそれに限らない）に対しても責任を負いません。

- libxml2 - ソースコードに特に記載がある場合を除く（たとえば、使用しているライセンスは類似しているが、著作権の通知が異なる hash.c、list.c ファイルや trio ファイル）、すべてのファイル :

Copyright © 1998 - 2003 Daniel Veillard. All Rights Reserved.

これをもって、このソフトウェアのコピーと関連する文書ファイル（「本ソフトウェア」）を入手した人すべてに対し、無料で本ソフトウェアを使用、コピー、変更、マージ、パブリッシュ、配布、サブライセンスする、もしくはコピーを販売する権利を含むがそれに限定せず、本ソフトウェアを制限なく取り扱う権利、および本ソフトウェアの供給相手に対してそうすることを許可する権利が付与されます。ただし、以下の条件を満たさなければなりません。

上記の著作権に関する通知とこの許可に関する通知が、本ソフトウェアのコピーのすべてまたは大部分に記載されていること。

このソフトウェアは、 現状のままで」提供され、商業性および特定の目的に対する適合性、および非侵害の保証を含むがそれに限らない、いかなる明示的もしくは黙示的な保証も行われません。DANIEL VEILLARDは、いかなる場合においても、それが契約、過失、もしくは他の不法行為のどれであれ、本ソフトウェアから、もしくは本ソフトウェアに関連して、または本ソフトウェアの使用もしくは他の取り扱いに関連して生じた申し立て、損害賠償もしくは他の義務に対し、責任を負いません。

この通知に含まれているものを除き、Daniel Veillardから事前に書面による許可を得ることなく、本ソフトウェアの広告、またはその他の手段による本ソフトウェアの販売、使用もしくは他の取り扱いの宣伝にDaniel Veillardの名前を使用することはできません。

- UNIXファイルに使用された解凍アルゴリズムについて:

  Copyright © 1985, 1986, 1992, 1993

  カリフォルニア大学評議員。All rights reserved.

  このソフトウェアは、評議員および貢献者によって「現状のままで」提供され、商業性および特定の目的に対する適合性に関する黙示的な保証を含むがそれに限らない、いかなる明示的もしくは黙示的な保証も行われません。いかなる場合においても、評議員または貢献者は、損害の原因が何であれ、そして法的責任の根拠が何であれ、つまり、契約、厳格責任、不法行為（過失その他を含む）のどれであれ、かかる損害の発生する可能性を事前に知らされていたとしても、このソフトウェアをどのように使用して生じた損害であれ、いかなる直接損害、間接損害、付随的損害、特別損害、懲罰的損害、もしくは結果損害（代替品または代替サービスの調達、使用機会、データもしくは利益の損失、業務の中断を含むがそれに限らない）に対しても責任を負いません。

  1. 再配布するソースコードには、上記の著作権に関する通知、この条件リスト、これに続く放棄声明が記載されていなければなりません。

  2. バイナリ形式で再配布する場合は、共に提供されるマニュアルなどの資料に上記の著作権に関する通知、この条件リスト、これに続く放棄声明が記載されていなければなりません。

  3. 事前に書面による許可を得ることなく、このソフトウェアから派生した製品の推奨または宣伝のために大学の名前や貢献者の名前を使用することはできません。

- Snowball - Copyright © 2001, Dr Martin Porter, Copyright © 2002, Richard Boulton.

  All rights reserved.

ソースおよびバイナリの形で、そのまま、もしくは変更を加えて再配布および使用することは、次のような条件を満たす限り、許可されます。

1. 再配布するソースコードには、上記の著作権に関する通知、この条件リスト、これに続く放棄声明が記載されていなければなりません。

2. バイナリ形式で再配布する場合は、共に提供されるマニュアルなどの資料に上記の著作権に関する通知、この条件リスト、これに続く放棄声明が記載されていなければなりません。

3. 事前に書面による許可を得ることなく、このソフトウェアから派生した製品の推奨または宣伝のために著作権保有者の名前や貢献者の名前を使用することはできません。

このソフトウェアは、著作権保有者および貢献者によって「現状のままで」提供され、商業性および特定の目的に対する適合性に関する黙示的な保証を含むがそれに限らない、いかなる明示的もしくは黙示的な保証も行われません。いかなる場合においても、著作権保有者または貢献者は、損害の原因が何であれ、そして法的責任の根拠が何であれ、つまり、契約、厳格責任、不法行為（過失その他を含む）のどれであれ、かかる損害の発生する可能性を事前に知らされていたとしても、このソフトウェアをどのように使用して生じた損害であれ、いかなる直接損害、間接損害、付随的損害、特別損害、懲罰的損害、もしくは結果損害（代替品または代替サービスの調達、使用機会、データもしくは利益の損失、業務の中断を含むがそれに限らない）に対しても責任を負いません。

目次
スクリプトガイド

4 スクリプト作成のツール

5 JSLの構成要素

第 1 章

JMP の概要
マニュアルとその他のリソース

この章には以下の情報が記載されています。

- 本書の表記法
- JMP のマニュアル
- JMP ヘルプ
- その他のリソース
  - その他の JMP のドキュメンテーション
  - チュートリアル
  - 索引
  - Web リソース
  - テクニカルサポートのオプション

表記規則

マニュアルの内容と画面に表示される情報を対応付けるために、次のような表記規則を使っています。

- サンプルデータ名、列名、パス名、ファイル名、ファイル拡張子、およびフォルダ名は「」で囲んで表記しています。
- スクリプトのコードは`Lucida Sans Typewriter`フォントで表記しています。
- スクリプトコードの結果（ログに表示されるもの）は*`Lucida Sans Typewriter`*（斜体）フォントで表記し、先に示すコードよりインデントされています。
- クリックまたは選択する項目は [] で囲んで太字で表記しています。これには以下の項目があります。
  - ボタン
  - チェックボックス
  - コマンド
  - 選択可能なリスト項目
  - メニュー
  - オプション
  - タブ名
  - テキストボックス
- 次の項目の表記規則は下記のとおりです。
  - 重要な単語や句、JMPに固有の定義を持つ単語や句は太字または「」で囲んで表記
  - マニュアルのタイトルは『』で囲んで表記
  - 変数名は斜体で表記
  - スクリプトの出力は斜体で表記
- JMP Proのみの機能にはJMP Proアイコン JMP PRO がついています。JMP Proの機能の概要については https://www.jmp.com/ja_jp/software/predictive-analytics-software.html をご覧ください。

メモ: 特別な情報および制限事項には、この文のように「メモ」という見出しがついています。

ヒント: 役に立つ情報には「ヒント」という見出しがついています。

JMPのマニュアル

JMP では、PDF 形式のマニュアルが用意されています。

- PDF版は［**ヘルプ**］>［**ドキュメンテーション**］メニューまたはJMPオンラインヘルプのフッタから開くことができます。
- 検索しやすいようにすべてのドキュメンテーションが1つのPDFファイルにまとめられた『JMP ドキュメンテーションライブラリ』と呼ばれるファイルがあります。『JMP ドキュメンテーションライブラリ』のPDFファイルは［**ヘルプ**］>［**ドキュメンテーション**］メニューから開くことができます。

JMP ドキュメンテーションライブラリ

以下の表は、JMP ライブラリに含まれている各ドキュメンテーションの目的および内容をまとめたものです。

マニュアル	目的	内容
『はじめてのJMP』	JMPをあまりご存知ない方を対象とした入門ガイド	JMPの紹介と、データを作成および分析し始めるための情報
『JMPの使用法』	JMPのデータテーブルと、基本操作を理解する	一般的なJMPの概念と、データの読み込み、列プロパティの変更、データの並べ替え、SASへの接続など、JMP全体にわたる機能の説明
『基本的な統計分析』	このマニュアルを見ながら、基本的な分析を行う	［分析］メニューからアクセスできる以下のプラットフォームの説明： • 一変量の分布 • 二変量の関係 • 表の作成 • テキストエクスプローラ ［分析］>［二変量の関係］で二変量、一元配置分散分析、分割表に対する分析を実行する方法の説明。ブートストラップを使用した標本分布の近似方法やシミュレーションの機能を使用したパラメトリックな標本再抽出の実行方法の説明も含まれています。

マニュアル	目的	内容
『グラフ機能』	データに合った理想的なグラフを見つける	［グラフ］メニューからアクセスできる以下のプラットフォームの説明： • グラフビルダー • 重ね合わせプロット • 三次元散布図 • 等高線図 • バブルプロット • パラレルプロット • セルプロット • ツリーマップ • 散布図行列 • 三角図 • チャート このマニュアルには背景マップやカスタムマップの作成方法も記載されています。
『プロファイル機能』	対話式のプロファイルツールの使い方を学ぶ。任意の応答曲面の断面を表示できるようになります。	［グラフ］メニューに表示されるすべてのプロファイルについて。誤差因子の分析が、ランダム入力を使用したシミュレーションの実行とともに含まれています。
『実験計画(DOE)』	実験の計画方法と適切な標本サイズの決定方法を学ぶ	［実験計画（DOE）］メニューと［分析］>［発展的なモデル］メニューの「発展的な実験計画モデル」に関するすべてのトピックについて。

マニュアル	目的	内容
『基本的な回帰モデル』	「モデルのあてはめ」プラットフォームとその多くの手法について学ぶ	［分析］メニューの「モデルのあてはめ」プラットフォームで使用できる、以下の手法の説明： • 標準最小2乗 • ステップワイズ • 一般化回帰 • 混合モデル • MANOVA • 対数線形-分散 • 名義ロジスティック • 順序ロジスティック • 一般化線形モデル

マニュアル	目的	内容
『予測モデルおよび発展的なモデル』	さらなるモデリング手法について学ぶ	［分析］>［予測モデル］メニューで使用できる以下のプラットフォームの説明： • モデル化ユーティリティ • ニューラル • パーティション • ブートストラップ森 • ブースティングツリー • K近傍法 • 単純Bayes • モデルの比較 • 計算式デポ ［分析］>［発展的なモデル］メニューで使用できる以下のプラットフォームの説明： • 曲線のあてはめ • 非線形回帰 • Gauss過程 • 時系列分析 • 対応のあるペア ［分析］>［スクリーニング］メニューで使用できる以下のプラットフォームの説明： • 応答のスクリーニング • 工程のスクリーニング • 説明変数のスクリーニング • アソシエーション分析 ［分析］>［発展的なモデル］>［発展的な実験計画モデル］で使用できるプラットフォームについては、『実験計画(DOE)』に説明があります。

マニュアル	目的	内容
『多変量分析』	複数の変数を同時に分析するための手法について理解を深める	［分析］>［多変量］メニューで使用できる以下のプラットフォームの説明： • 多変量の相関 • 主成分分析 • 判別分析 • PLS ［分析］>［クラスター分析］メニューで使用できる以下のプラットフォームの説明： • 階層型クラスター分析 • K Means クラスター分析 • 正規混合 • 潜在クラス分析 • 変数のクラスタリング
『品質と工程』	工程を評価し、向上させるためのツールについて理解を深める	［分析］>［品質と工程］メニューで使用できる以下のプラットフォームの説明： • 管理図ビルダーと個々の管理図 • 測定システム分析 • 計量値/計数値ゲージチャート • 工程能力 • パレート図 • 特性要因図

マニュアル	目的	内容
『信頼性/生存時間分析』	製品やシステムにおける信頼性を評価し、向上させる方法、および人や製品の生存時間データを分析する方法について学ぶ	［分析］>［信頼性/生存時間分析］メニューで使用できる以下のプラットフォームの説明： • 寿命の一変量 • 寿命の二変量 • 累積損傷 • 再生モデルによる分析 • 劣化分析と破壊劣化 • 信頼性予測 • 信頼性成長 • 信頼性ブロック図 • 修理可能システムのシミュレーション • 生存時間分析 • 生存時間(パラメトリック)のあてはめ • 比例ハザードのあてはめ
『消費者調査』	消費者選好を調査し、その洞察を使用してより良い製品やサービスを作成するための方法を学ぶ	［分析］>［消費者調査］メニューで使用できる以下のプラットフォームの説明： • カテゴリカル • 多重対応分析 • 多次元尺度構成 • 因子分析 • 選択モデル • MaxDiff • アップリフト • 項目分析
『スクリプトガイド』	パワフルなJMPスクリプト言語（JSL）の活用方法について学ぶ	スクリプトの作成やデバッグ、データテーブルの操作、ディスプレイボックスの構築、JMPアプリケーションの作成など。
『スクリプト構文リファレンス』	JSL関数、その引数、およびオブジェクトやディスプレイボックスに送信するメッセージについて理解を深める	JSLコマンドの構文、例、および注意書き。

メモ：［**ドキュメンテーション**］メニューでは、印刷可能な2つのリファレンスカードも用意されています。『**メニューカード**』はJMPのメニューをまとめた表で、『**クイックリファレンス**』はJMPのショートカットキーをまとめた表です。

JMPヘルプ

JMPヘルプは、一連のマニュアルの簡易版です。JMPのヘルプは、次のいくつかの方法で開くことができます。

- Windowsでは、F1キーを押すとヘルプシステムウィンドウが開きます。
- データテーブルまたはレポートウィンドウの特定の部分のヘルプを表示します。［**ツール**］メニューからヘルプツールを選択した後、データテーブルやレポートウィンドウの任意の位置でクリックすると、その部分に関するヘルプが表示されます。
- JMPウィンドウ内で［**ヘルプ**］ボタンをクリックします。
- Windowsの場合、［ヘルプ］メニューの［**ヘルプの目次**］、［**ヘルプの検索**］、［**ヘルプの索引**］の各オプションを使用して、JMPヘルプ内を検索し、目的の内容を表示します。Macの場合、［**ヘルプ**］>［**JMPヘルプ**］を選択します。

JMPを習得するためのその他のリソース

JMPのマニュアルとJMPヘルプの他、次のリソースもJMPの学習に役立ちます。

- チュートリアル（「チュートリアル」（29ページ）を参照）
- サンプルデータ（「サンプルデータテーブル」（30ページ）を参照）
- 索引（「統計用語とJSL用語の習得」（30ページ）を参照）
- 使い方ヒント（「JMPを使用するためのヒント」（30ページ）を参照）
- Webリソース（「JMP User Community」（31ページ）を参照）
- 専門誌『JMPer Cable』（「JMPer Cable」（31ページ）を参照）
- JMPに関する書籍（「JMP関連書籍」（31ページ）を参照）
- JMPスターター（「「JMPスターター」ウィンドウ」（32ページ）を参照）
- 教育用リソース（「サンプルデータテーブル」（30ページ）を参照）

チュートリアル

［**ヘルプ**］>［**チュートリアル**］を選択して、JMPのチュートリアルを表示できます。［**チュートリアル**］メニューの最初の項目は［**チュートリアルディレクトリ**］です。この項目を選択すると、すべてのチュートリアルをカテゴリ別に整理した新しいウィンドウが開きます。

JMPに慣れていない方は、まず［**初心者用チュートリアル**］を試してみてください。JMPのインターフェースおよび基本的な使用方法を学ぶことができます。

他のチュートリアルでは、実験の計画、標本平均と定数の比較など、JMPの具体的な活用法を学習できます。

サンプルデータテーブル

JMPのマニュアルで取り上げる例は、すべてサンプルデータを使用しています。サンプルデータディレクトリを開くには、［**ヘルプ**］>［**サンプルデータライブラリ**］を選択します。

サンプルデータテーブルを文字コード順に並べた一覧を表示する、またはカテゴリごとにサンプルデータを表示するには、［**ヘルプ**］>［**サンプルデータ**］を選択します。

サンプルデータテーブルは次のディレクトリにインストールされています。

Windowsの場合: C:¥Program Files¥SAS¥JMP¥13¥Samples¥Data

Macintoshの場合: \Library\Application Support\JMP\13\Samples\Data

JMP Proでは、サンプルデータが（JMPではなく）JMPPROディレクトリにインストールされています。シングルユーザーライセンス版のJMP（JMP シュリンクラップ）では、サンプルデータがJMPSWディレクトリにインストールされています。

サンプルデータの使用例を参照するには、［**ヘルプ**］>［**サンプルデータ**］を選択し、教育用セクションから検索してください。教育用リソースについては、http://jmp.com/tools にも情報があります。

統計用語とJSL用語の習得

［**ヘルプ**］メニューには、次の索引が用意されています。

統計の索引　統計用語が説明されています。

スクリプトの索引　JSL関数、オブジェクト、ディスプレイボックスに関する情報を検索できます。スクリプトの索引からサンプルスクリプトを編集して実行することもできます。

JMPを使用するためのヒント

JMPを最初に起動すると、「使い方ヒント」ウィンドウが表示されます。このウィンドウには、JMPを使う上でのヒントが表示されます。

「使い方ヒント」ウィンドウを表示しないようにするには、［**起動時にヒントを表示する**］のチェックを外します。再表示するには、［**ヘルプ**］>［**使い方ヒント**］を選択します。または、「環境設定」ウィンドウで非表示に設定することもできます。詳細については、『JMPの使用法』を参照してください。

ツールヒント

次のような項目の上にカーソルを置くと、その項目を説明するツールヒントが表示されます。

- メニューまたはツールバーのオプション
- グラフ内のラベル
- レポートウィンドウ内の結果（テキスト）（カーソルで円を描くと表示される）
- 「ホームウィンドウ」内のファイル名またはウィンドウ名
- スクリプトエディタ内のコード

ヒント： Windowsでは、JMP環境設定でツールヒントを表示しないよう設定できます。［**ファイル**］>［**環境設定**］>［**一般**］を選択し、［**メニューのヒントを表示**］の選択を解除します。このオプションは、Macintoshでは使用できません。

JMP User Community

JMP User Communityでは、さまざまな方法でJMPをさらに習得したり、他のSASユーザとのコミュニケーションを図ったりできます。ラーニングライブラリには1ページガイド、チュートリアル、デモなどが用意されており、JMPを使い始める上でとても便利です。また、JMPのさまざまなトレーニングコースに登録して、自己教育を進めることも可能です。

その他のリソースとして、ディスカッションフォーラム、サンプルデータやスクリプトファイルの交換、Webcastセミナー、ソーシャルネットワークグループなども利用できます。

WebサイトのJMPリソースにアクセスするには、［**ヘルプ**］>［**JMP User Community**］を選択するか、https://community.jmp.com/ をご覧ください。

JMPer Cable

JMPer Cable は、JMP ユーザを対象とした年刊の専門誌です。JMPer Cable は次の JMP Web サイトで閲覧可能です。

http://www.jmp.com/about/newsletters/jmpercable/（英語）

JMP関連書籍

JMP 関連書籍は、次の JMP Web ページで紹介されています。

https://www.jmp.com/ja_jp/academic/books-for-jmp-users.html

「JMP スターター」ウィンドウ

JMP またはデータ分析にあまり慣れていないユーザは、「JMP スターター」ウィンドウから開始するとよいでしょう。カテゴリ分けされた項目には説明がついており、ボタンをクリックするだけで該当の機能を起動できます。「JMP スターター」ウィンドウには、［分析］、［グラフ］、［テーブル］、および［ファイル］メニュー内の多くのオプションがあります。また、JMP Pro の機能やプラットフォームのリストも含まれています。

- 「JMP スターター」ウィンドウを開くには、［**表示**］（Macintosh では［**ウィンドウ**］）**>**［**JMP スターター**］を選択します。
- Windows で JMP の起動時に自動的に「JMP スターター」を表示するには、［**ファイル**］**>**［**環境設定**］**>**［**一般**］を選び、「開始時の JMP ウィンドウ」リストから［**JMP スターター**］を選択します。Macintosh では、［**JMP**］**>**［**環境設定**］**>**［**起動時に JMP スターターウィンドウを表示する**］を選択します。

テクニカルサポート

JMP のテクニカルサポートは、JMP のエンジニアが担当し、その多くは、統計学などの技術的な分野の知識を有しています。

http://www.jmp.com/japan/support には、テクニカルサポートへの連絡方法などが記載されています。

第2章

概要
JMPスクリプト言語にようこそ

JMPスクリプト言語（JSL）でスクリプトを記述すると、JMPの分析結果を再現することができます。パワーユーザの多くは、JMPの機能を拡張するスクリプトを作成し、製造に関する設定など定期的に行う分析を自動化しています。JSLの習得が面倒な人でも、スクリプトを自動生成する機能を利用できます。

JSLによっていろいろなことを行えます。

- 列の計算式を実行する
- プラットフォームを起動する
- プラットフォームをインタラクティブに変更する
- グラフを作成する

JSL でできること

JMP では、データテーブルや分析の現在の状態を再現するスクリプトを自動的に生成し、保存できます。分析の途中でスクリプトをスクリプトウィンドウ（または**スクリプトエディタ**）、データテーブル、または分析レポートに保存しておけば、後で修正して別のプロジェクトに利用できます。分析が終わった時点でスクリプトを保存すれば、最終結果を再現することができます。

以下は、JSL スクリプトが役に立つ例です。

- 分析プロセスを最初から最後まで詳細に記述しなければならない場合。たとえば、管理当局や、学術論文をレビューする人のために、追跡調査に役立つ記録を作成できます。
- 研究所の技術者が、一連の分析を定期的に実施する場合。
- 毎日、新しいデータに同じ手順で同じモデルをあてはめる場合。

JMP を通常どおりインタラクティブに使用して、作業を再現するスクリプトを保存しておけば、スクリプトを実行するだけで結果が再現できるようになります。

JSL は、設計上、次のような目的には使用できません。

- JMP での操作そのものをスクリプトとして記録することはできません。スクリプト言語の中にはスクリプトレコーダーによる記録が便利なものもありますが、JMP のように結果が大切なソフトウェアではあまり重要ではありません。スクリプトレコーダーを使って、インタラクティブな操作が実行される様子を観察することはできません。
- JSL は、JMP を操作するための代替手段として用意されたコマンドラインインターフェースではありません。

JSL 習得のサポート

JMP には、JSL スクリプトを書いたり理解したりする上で役立つツールがいくつか用意されています。

JMP スクリプト言語（JSL）ガイド

『スクリプトガイド』では、スクリプト言語にあまり慣れていない JMP ユーザ向けの基本的な情報（用語や構文など）から、より高度な情報まで説明しています。

第2章～第4章	JSL の習得、基本的なスクリプトを生成させる方法、また JSL スクリプトを作成する環境についての情報を含む。
第5章～第8章	JSL の基本要素、数値や文字列などの基本的なデータタイプ、リストや行列、連想配列の記述、名前空間、JSL でのプログラミングの基礎について紹介する。

第9章～第13章	データテーブル、プラットフォーム、ウィンドウ、グラフィックなどのJMPオブジェクトに対するJSLの使用方法を取り上げる。
第14章	SAS、R、Microsoft Excelなどの外部プログラムと連携するためのスクリプトの記述方法について説明する。
第15章	ボタン、リスト、グラフ、その他のオブジェクトを含むウィンドウをドラッグ＆ドロップで作成できる、アプリケーションビルダーでのJMPアプリケーションの作成について紹介する。また、アドインビルダーを使ってスクリプトをコンパイルし、簡単に共有できるファイルを作成する方法についても説明する。
第16章	スクリプトのいろいろなサンプルや例の紹介。コピーして、必要なところだけ変更を加えて利用できる。
付録Aおよび付録B	前バージョンのJMPとの互換性に関する情報を提供。また、JSLの概念と用語について説明する。

スクリプトの索引

［**ヘルプ**］メニューにあるスクリプトの索引には、JSL関数、オブジェクト、およびディスプレイボックスの簡単な説明と構文が記載されています。各エントリに用意されている例を実行したり、修正を加えてコードのテストに利用することができます。例を実行すると、下に埋め込まれたログウィンドウにメッセージが表示されます。

「スクリプトの索引」ウィンドウには、次のボタンがあります。


検索を開始するには、［**検索**］ボタンをクリックします。


［**クリア**］ボタンをクリックすると、検索テキストボックスがクリアされ、新たに検索を開始することができます。


検索方法とパラメータを設定するには、［**設定**］ボタンをクリックします。

［**設定**］ボタンには複数の検索方法が用意されています。

部分一致　少なくとも一部が検索文字列と一致するすべての項目を戻します。たとえば、「leas」で検索すると、「Release Zoom」や「Partial Least Squares」といったメッセージが戻されます。デフォルトでは、この方法により検索されます。

フレーズ検索　検索文字列と完全に一致するテキストを含む項目を戻します。たとえば、「text」で検索すると、「text」という文字列を含むすべての要素が戻されます。

すべての項　いずれかの文字列またはすべての文字列を含む項目を戻します。たとえば、「t test」で検索すると、「Pat Test」、「Shortest Edit Script」、「Paired t test」が戻されます。

いずれかの項　検索文字列のいずれかを含む項目を戻します。たとえば、「text string」で検索すると、「Context Box」、「Drag Text」、「Is String」などが戻されます。

正規表現 検索ボックスにワイルドカード（*）とピリオド（.）を使用できます。たとえば、「get *name」で検索すると、「Get Name Info」、「Get Namespace」などのメッセージが戻されます。「get.*name」で検索すると、「Get Color Theme Names」、「Get Name Info」、「Get Effect Names」などの項目が戻されます。

［**設定**］ボタンにはいくつかの検索パラメータも用意されています。

すべてのフィールド 索引内のすべてのフィールドを対象に検索文字列を検索するよう指定します。

タイトルのみ 索引のタイトルのみを対象に検索文字列を検索するよう指定します。

例のみ 索引の例のみを対象に検索文字列を検索するよう指定します。

例を除く 例以外を対象に検索文字列を検索するよう指定します。

JMPのオンラインヘルプで項目の詳細を確認するには、その項目の［**トピックのヘルプ**］ボタンをクリックします。

JMPからJSLを学ぶ

JMP自身が、実は最高のJSLプログラマーです。JMPでインタラクティブに作業し、その結果をスクリプトで保存して、後から再使用できます。スクリプトに簡単な編集を加え、テンプレートとして使えば、毎日の作業がスピードアップします。

JSLは非常に柔軟性のある言語で、同じ目的をさまざまな方法で達成できます。以下はその例です。JMPで分析を行うと、大部分の処理をデフォルトで自動的に行った場合でも、分析のあらゆる詳細が保存されます。ということは、ユーザが書くスクリプトも同様に完全で詳細でなければならないのでしょうか。そんなことはありません。通常は、グラフィカルユーザインターフェース（GUI）で選択する項目だけで問題ありません。たとえば、サンプルデータのフォルダにある「Big Class.jmp」を開き、「**身長(インチ)**」、「**体重(ポンド)**」、「**性別**」の一変量の分布を起動したいときは、次のようなスクリプトだけで大丈夫です。

```
Distribution( Y (:Name(" 身長 ( インチ )"), :Name(" 体重 ( ポンド )"), : 性別 ) );
```

GUIで「一変量の分布」プラットフォームを実行し、レポートの赤い三角ボタンのメニューから［**スクリプトの保存**］>［**スクリプトウィンドウへ**］を選択すると、次のようなスクリプトが表示されます。

```
Distribution(
   Nominal Distribution( Column( :sex ) ),
   Continuous Distribution( Column(:Name(" 身長 ( インチ )") ) ),
   Continuous Distribution( Column( :weight ) )
);
```

どちらのスクリプトも結果は同じです。

JSLをいろいろと試してみてください。こんなこともできるのでは、と思ったなら、おそらくできるはずです。どうなるか試してみましょう。

用語

スクリプトの作成を始める前に、このマニュアル全体で使用されている基本的なJSL用語を知っておく必要があります。

演算子と関数

演算子は、一般的な演算に使用される1文字または2文字の記号（+や=など）です。

関数はコマンドで、追加情報として引数を指定する場合もあります。

一部のJSL関数は演算子と同様に機能しますが、より複雑なアクションを実行できます。たとえば、次の2つのスクリプトは同じ処理を行います。

```
2 + 3; // 5を戻す
Add( 2, 3 ); // 5を戻す
```

最初の行は+演算子を使用しています。2行目は等価な関数であるAdd()を使用しています。

JSLの演算子のすべてに、等価な関数がありますが、関数の中には等価な演算子がないものもあります。たとえば、Sqrt(a)はSqrt()関数でしか表現できません。

メモ: 以前のバージョンのJMPおよびそれらのマニュアルでは、**演算子**と**関数**という2つの用語があまり区別されていませんでした。今バージョンでは、より厳密に区別して記載しています。

オブジェクトとメッセージ

オブジェクトとは、JMPの動的なエンティティで、たとえばデータテーブル、データ列、プラットフォーム結果ウィンドウ、グラフなどがこれに当たります。ほとんどのオブジェクトは、何らかのアクションを実行するメッセージを受け取ることができます。

メッセージとは、オブジェクトに送られるJSL式を指します。オブジェクトは、送られたメッセージを評価し、何らかのアクションを実行します。次の例で、dtはデータテーブルオブジェクトです。<<はメッセージが続くことを意味します。次のメッセージは、指定の変数を使って要約テーブルを作成するよう指示しています。

```
dt << Summary( Group( :年齢 ), Mean( :Name("身長(インチ)") ) )
```

この式で、dtはデータテーブルへの参照を含む変数の名前です。この変数には任意の名前を使用できます。このマニュアルでは、データテーブルの参照をdtと表記します。以下に、このマニュアルでよく使用されているオブジェクトの表記方法を示します。

略語	オブジェクト
dt	データテーブル
col	データテーブルの列
colname	データテーブルの列の名前

略語	オブジェクト
obj	オブジェクト
db	ディスプレイボックス

これらの変数には参照が事前に割り当てられているわけではありません。どの変数も、使用する前に割り当てておく必要があります。次の例では、Aという名前のグローバル変数に“世界のみなさん、こんにちは”という値が割り当てられています。Show(A)コマンドが処理されると、Aの値が表示されます。

```
A = "Hello, World";
Show( A );
  A = "世界のみなさん、こんにちは";
```

引数とパラメータ

引数は、関数またはメッセージに送ることのできる追加の情報です。たとえば、Root(25)の場合、25はRoot()関数に対する引数です。Root()は指定された引数に対して実行され、結果（5）を戻します。

プログラミングやスクリプティングのマニュアルを読むと、パラメータも登場するのが普通です。**パラメータ**は、関数が受け入れる引数についての説明です。たとえば、Root()は一般的にRoot(number)のように指定され、*number*がパラメータです。

パラメータと**引数**は、関数が必要とする情報をそれぞれ異なる視点から表現したものです。

このマニュアルでは、簡単にするために両方のケースで**引数**という用語を使用します。

名前付き引数は、事前に決められた引数のセットの中から、引数の名前を記述して明示的に指定するものです。たとえば、Graph Box()関数内のtitle("折れ線グラフ")は、タイトルを明示的に定義する名前付き引数です。

```
Graph Box( title( "折れ線グラフ" ),
      Frame Size( 300, 500 ),
      Marker( Marker State( 3 ), [11 44 77], [75 25 50] );
      Pen Color( "Blue" );
      Line( [10 30 70], [88 22 44] ));
```

Frame Size()の引数の300と500は名前付き引数ではありません。これらの引数は位置に重要な意味があります。最初の引数は幅、2番目の引数は高さを表します。

オプションの引数

関数とメッセージには、必須の引数とオプションの引数があります。オプションの引数は、指定しなくてもかまいません。表記上、オプションの引数は鍵括弧で囲んであります。例:

```
Root( x, <n> )
```

で、引数*x*は必須です。引数*n*はオプションです。

オプションの引数の多くに、デフォルトの値があります。たとえばRoot()の場合、*n*のデフォルト値は2です。

コード	出力	説明
Root(25)	5	25の平方根を戻す。
Root(25, 2)	5	25の平方根を戻す。
Root(25, 3)	2.92401773821287	25の三乗根を戻す。

式

式は、タスクを実行するJSLコードの一部分です。JSL式は、データの保持、データの処理、およびオブジェクトへのコマンドの送信を行います。たとえば、次の式は「**Big Class.jmp**」サンプルデータテーブルを開き、二変量のグラフを作成します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Bivariate( Y( :Name("体重(ポンド)") ), X( :Name("身長(インチ)") ) );
```

Orおよび縦棒の記号

1本の縦棒（|）は論理ORを表します。引数を表すとき、**or**を単に | で表記します。

たとえば、パス名を指定する引数として`absolute|relative`のような表記があった場合、引数には次の2つのオプションの**どちらか**を指定できることを示しています。

- `absolute`は絶対パス名を示します。
- `relative`は相対パス名を示します。

同様に、縦棒を使って3つ以上のオプションを示す場合もあります。

スクリプトのフォーマット

空白（スペース、タブ、改行など）と大文字、小文字の違いはJSLでは無視されます。つまり、次の2つの式は等価です。

```
// 式1
sum = 0;
For( i = 1, i <= 10, i++,
   sum += i;
   Show( i, sum );
);

// 式2
Sum = 0;
For( i = 1, i <= 10, i++,
   Sum += i;
   Show( i, Sum );
);
```

スクリプトはお好みのフォーマットで記述できますが、スクリプトエディタを使えば、自動的にフォーマットすることも可能です。このマニュアルでは、大文字／小文字、スペース、改行、タブなどの使い方に、スクリプトエディタのデフォルトのフォーマットを使用します。スクリプトエディタの使用方法については、「スクリプト作成のツール」章の「スクリプトエディタの使用」（52 ページ）を参照してください。

メモ: 空白に関連する唯一の例外は、2 文字の演算子（<= または ++）です。これらの演算子はスペースで分離することができません。

基本的な JSL 構文

JSL スクリプトは一連の式です。各式は、タスクを実行する JSL コードの一部分です。JSL 式は、データの保持、データの処理、およびオブジェクトへのコマンドの送信を行います。

多くの式は、メッセージ名の後ろに内容が括弧で囲まれた構造になっています。

```
Message Name ( argument 1, argument 2, ... );
```

JSL 名の意味はコンテキストによって異なります。同じ名前でも、データテーブルのコンテキストと、関数のコンテキストでは全く別の意味を持つことがあります。詳細については、「JSL の構成要素」章の「名前解決のルール」（91 ページ）を参照してください。

括弧の一致など特定の記述ルールに従うものはほとんどが JSL 式として有効です。例:

```
win = New Window( "Window の例 ",
   <<Modal,
   Text Box( " 世界のみなさん、こんにちは " ),
   Text Box( "-----" ),
   Button Box( "OK" )
);
```

次の点を念頭に置いてください。

- 名前にはスペースを埋め込むことができます。詳細については、「JSL の構成要素」章の「名前」（83 ページ）を参照してください。
- メッセージの内容は、必ず一致した括弧で囲みます。詳細については、「JSL の構成要素」章の「括弧」（80 ページ）を参照してください。
- 項目はカンマで区切ります。詳細については、「JSL の構成要素」章の「カンマ」（80 ページ）を参照してください。
- JSL には大文字と小文字の区別がありません。“text box();” と “Text Box()” は等価です。
- メッセージが別のメッセージの入れ子になるケースも多くあります。

本書の表記法

本書では、示されているとおりの表記で使用する必要がある関数については、関数名の頭文字を大文字で示しています。また、実際に選択したものを入力する引数は小文字で示しています。次の例で説明すると、Connect Colorは、そのとおりに入力する必要がある関数であり、colorの部分は、ユーザが好みで入力する色を指します。

```
Connect Color(color);
```

この場合、括弧内の引数は色の値でなければなりません。色の値とは、JMPの色番号、redやblueなどのサポートされている色名、または{.75, .50, .50}のようにリストで指定するRGB値です。このように、複数の指定方法がある場合は、次のように、「または」を意味する「|」文字で示します。

```
Connect Color( number | "color name" | {r,g,b} );
```

スクリプトエディタに貼り付けて実行できるスクリプトについては、その構文が色分けされています。

第3章

始めましょう
JMPによるスクリプトの作成

同じデータを使い、同じレポートを定期的に作成するケースはよくあります。この章では、テキストデータを読み込む、Microsoft Excelファイルを開く、レポートを作成するといった一般的なタスクを実行するスクリプトを、JMPで自動生成させる方法について説明します。最後のチュートリアルでは、それらをまとめ、1つのスクリプトでMicrosoft Excelファイルを開き、3つのレポートを自動的に作成できるようにします。

このマニュアルは、JMPの使用には慣れているものの、JSLにはあまり慣れていないユーザを対象としています。一般的なタスクの実行については、『JMPの使用法』を参照してください。また、『はじめてのJMP』も、基本概念やJMPでの操作の流れを理解するのに役立ちます。

分析レポートのスクリプトを取得する

分析を再現するためのスクリプトは、次のような手順で取得できます。

1. 一変量の分布など、プラットフォームを起動します。
2. 必要に応じて変更を加えます。たとえば、検定や他のグラフを追加します。
3. 結果を再現するスクリプトを生成させます。

スクリプトはデータテーブルに保存できるので、他のユーザにデータテーブルを送るだけで、そのユーザはスクリプトを実行し、レポートを再現することができます。

例

次の手順に従って、一変量の分布のレポートを作成し、レポートを再現するスクリプトを生成させ、スクリプトをデータテーブルに保存してみましょう。

メモ: 例で使用するデータテーブルはJMPの「Samples¥Data」フォルダにあります。

1. ［**ヘルプ**］>［**サンプルデータライブラリ**］を選択し、「Companies.jmp」を開きます。
2. ［**分析**］>［**一変量の分布**］を選択して、「一変量の分布」起動ウィンドウを開きます。
3. 「列の選択」ボックスで**「利益 ($M)」**を選択し、［**Y, 列**］ボタンをクリックします。
4. ［**OK**］をクリックします。

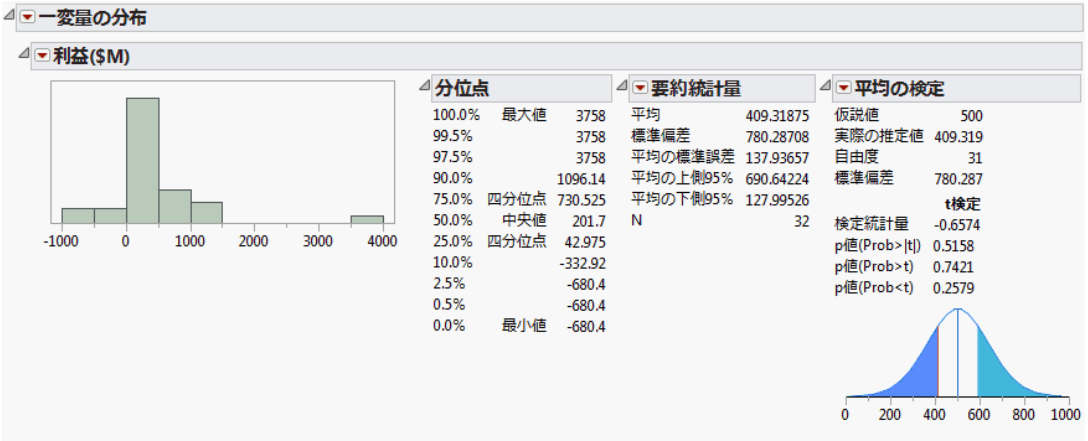
   一変量の分布のレポートウィンドウが表示されます。
5. 「一変量の分布」の赤い三角ボタンのメニューから［**積み重ねて表示**］を選択し、レポートを横に表示します。
6. 「利益 ($M)」の赤い三角ボタンのメニューで［**外れ値の箱ひげ図**］の選択を解除し、このオプションをオフにします。
7. 「利益 ($M)」の赤い三角ボタンのメニューから［**平均の検定**］を選択します。

   「平均の検定」ウィンドウが表示されます。
8. ［**仮説平均を指定**］ボックスに500と入力します。
9. ［**OK**］をクリックします。

   レポートウィンドウに平均の検定が追加されます。

   これで、カスタマイズしたレポートが作成できました。

図3.1 カスタマイズした一変量の分布レポート



10.「一変量の分布」の赤い三角ボタンのメニューから［**スクリプトの保存**］>［**データテーブルへ**］を選択します。

これで、データテーブルに「利益 ($M) の一変量の分布」という名前のスクリプトが保存されました。そのスクリプトの赤い三角ボタンのメニューから［**編集**］を選択すると、スクリプトが表示されます。

図3.2 データテーブルに保存された一変量の分布のスクリプト

11. このスクリプトを実行して、最終的なレポートをそのまま再現するには、スクリプトの赤い三角ボタンのメニューから［**編集**］>［**スクリプトの実行**］を選択します。

データテーブルのスクリプトを取得する

データテーブルを再現するためのスクリプトは、次のような手順で取得します。

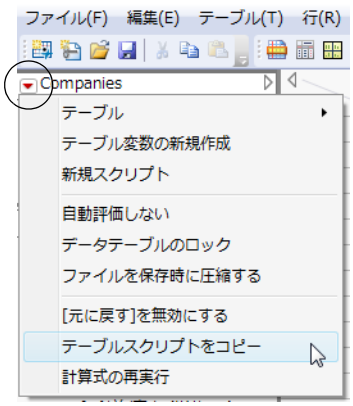
1. データテーブルを開きます。
2. 必要に応じて変更を加えます。たとえば、スクリプトを追加したり、値を修正したり、新しい列を追加したりします。
3. データテーブルを再現するスクリプトを取得します。

例

先ほどの例で取得した、スクリプトを保存したデータテーブルを使用します。

1. データテーブル名の隣にある赤い三角ボタンをクリックします。
2. ［**テーブルスクリプトをコピー**］を選択します。

図3.3 テーブルスクリプトをコピー



3. ［**ファイル**］>［**新規作成**］>［**スクリプト**］を選択してスクリプトウィンドウを開きます。
4. ［**編集**］>［**貼り付け**］を選択します。

これで、データテーブルを再現するスクリプトができました。このスクリプトを保存しておけば、後でいつでもデータテーブルを再現することができます。データテーブルにはスクリプトもそのまま含まれます。

ファイルを読み込むスクリプトを取得する

ファイルを読み込むスクリプトを取得するには、JMP で該当するファイルを開きます。JMP は、ファイルを開いた際の手順を自動的に記録します。

テキストファイルの読み込み

1. ［**ファイル**］>［**開く**］を選びます。

   「データファイルを開く」ウィンドウが表示されます。

2. 「**ファイル名**」の横のリストから［**テキストファイル**］を選択します。
3. 「開く方法」セクションで、［**データ、形式を識別する**］を選択します。

   JMP は、テキストファイルのタブ、カンマ、空白、その他の区切り文字基づいてデータをフォーマットします。

4. ファイルを選択し、［**開く**］を選択します。

   ファイルはデータテーブルとして開きます。データテーブルには「ソース」という名前のスクリプトが含まれます。これは、ユーザが使用したテキスト読み込みルールを使ってテキストファイルを読み込む JSL スクリプトです。

5. 「ソース」の赤い三角ボタンのメニューから［**編集**］を選択します。

   スクリプトをコピーして新しいスクリプトウィンドウに貼り付け、保存します。こうして保存したスクリプトを後で実行すれば、同じテキストファイルを再度読み込むことができます。

ヒント： このスクリプトは、読み込むテキストファイルの指定と、それを JMP に正しく読み込むためのオプションを含む Open() 式で構成されています。式の最初の部分は、読み込むファイルのパス名です。このスクリプトを保存し、別の PC などで実行する際は、テキストファイルを指すパス名の編集が必要となる場合があります。パス名については、「データタイプ」章の「パス変数」（119 ページ）で詳しく説明しています。

スクリプトを 1 つにまとめる

週に 1 回、新しいデータが Microsoft Excel ファイルに保存され、それを基に同じ内容のレポートを作成する必要があるとしましょう。ファイルを開いて毎週同じ手順を実行することもできますが、新しい Microsoft Excel ファイルを JMP に読み込み、分析するところまでをすべて自動で行う方がずっと効率的です。次の例では、スクリプトを準備し、毎週実行する方法を示しています。

Microsoft Excel ファイルの読み込み

1. 新しいスクリプトウィンドウを開きます（［**ファイル**］>［**新規作成**］>［**スクリプト**］）。
2. スクリプトウィンドウに、「Solubil.xls」サンプルデータファイルを開く Open() 式を入力します。ファイルは JMP の「Samples¥Import Data」フォルダにあります。

```
dt = Open( "$SAMPLE_IMPORT_DATA/Solubil.xls" );
```

後で式を追加するので、式の最後に必ずセミコロンを付けてください。セミコロンには式をつなぐ役目があります。

3. ［**編集**］>［**スクリプトの実行**］を選択し、Microsoft Excelファイルを読み込むスクリプトを実行します。

   Microsoft Excelファイルがデータテーブルとして開きます。

メモ:

- `Open()`式に`Excel Wizard`引数を含めて、ファイルを読み込む前にワークシートをプレビューすることもできます。詳細は、「データテーブル」章の「Microsoft Excelファイルからのデータの読み込み」（282ページ）の節を参照してください。
- パス変数を使わず、ファイルの絶対パスまたは相対パスを指定することもできます。相対パスを指定した場合は、スクリプトを実行するときに、スクリプトとファイルが相対的に同じ場所になければなりません。絶対パスの場合は、そのスクリプトを使用する他のユーザが指定のパスでファイルにアクセスできることを確認してください。パス名の使用について詳しくは、「データタイプ」章の「パス変数」（119ページ）を参照してください。

レポートを実行してスクリプトを取得する

一変量の分布、三次元散布図、多変量の3つのレポートを作成するとしましょう。それぞれをGUIを使って作成し、スクリプトをスクリプトウィンドウに追加します。

1. 新しいデータテーブルを開いたまま、［**分析**］>［**一変量の分布**］を選択します。
2. 「**Labels**」以外の列をすべて選択し、［**Y, 列**］を選択します。
3. ［**OK**］をクリックします。
4. 「**eth**」の赤い三角ボタンのメニューから、Ctrl キーを押しながら［**ヒストグラムオプション**］>［**度数の表示**］を選択します。

   6つすべてのヒストグラムに度数が表示されます。
5. 「一変量の分布」ウィンドウで、一変量の分布の赤い三角ボタンのメニューから［**スクリプト**］>［**スクリプトのコピー**］を選択します。
6. スクリプトウィンドウの`Open()`式の1、2行あとにカーソルを置き、［**編集**］>［**貼り付け**］を選択します。
7. 最後の閉じ括弧のあとにセミコロンを入力します。
8. ［**グラフ**］>［**三次元散布図**］を選択します。
9. 「**Labels**」以外の列をすべて選択し、［**Y, 列**］を選択します。
10. ［**OK**］をクリックします。
11. 一変量の分布レポートで行ったように、三次元散布図のスクリプトをコピーしてスクリプトウィンドウに貼り付けます。最後に必ずセミコロンを加えます。
12. ［**分析**］>［**多変量**］>［**多変量の相関**］を選択します。
13. 「**Labels**」以外の列をすべて選択し、［**Y, 列**］を選択します。
14. ［**OK**］をクリックします。

15. 一変量の分布および三次元散布図で行ったように、多変量の相関のスクリプトをコピーしてスクリプトウィンドウに貼り付けます。

    スクリプトは次のようになります。

```
dt = Open( "$SAMPLE_IMPORT_DATA/Solubil.xls" );
Distribution(
   Continuous Distribution( Column( :eth ), Show Counts( 1 ) ),
   Continuous Distribution( Column( :oct ), Show Counts( 1 ) ),
   Continuous Distribution( Column( :cc14 ), Show Counts( 1 ) ),
   Continuous Distribution( Column( :c6c6 ), Show Counts( 1 ) ),
   Continuous Distribution( Column( :hex ), Show Counts( 1 ) ),
   Continuous Distribution( Column( :chc13 ), Show Counts( 1 ) ),
);
Scatterplot 3D(
      Y( :eth, :oct, :cc14, :c6c6, :hex, :chc13 ),
      Frame3D( Set Grab Handles( 0 ), Set Rotation( -54, 0, 38) )
);
Multivariate(
   Y( :eth, :oct, :cc14, :c6c6, :hex, :chc13 ),
   Estimation Method( "Row-wise" ),
   Scatterplot Matrix(
      Density Ellipses( 1 ),
      Shaded Ellipses( 0 ),
      Ellipse Color( 3 )
   )
);
```

スクリプトの保存

これで、手動で行ったすべての手順を再現するスクリプトを取得できました。スクリプトを保存し、データテーブルおよびすべてのレポートウィンドウを閉じます。

1. スクリプトが表示されたスクリプトウィンドウで、[**ファイル**] > [**上書き保存**] または [**ファイル**] > [**名前を付けて保存**] を選択します。
2. ファイル名を指定します（たとえば、「週間レポート」）。
3. [**保存**] をクリックします。

スクリプトの実行

更新される Microsoft Excel ファイルが毎週同じ場所に保存され、かつ同じ列を含んでいる限り、スクリプトを実行するだけでレポートを自動的に作成できます。

1. 保存したスクリプトを開きます。
2. [**編集**] > [**スクリプトの実行**] を選択します。

   JMP で Microsoft Excel ファイルが開き、3つのレポートが表示されます。

このスクリプトを他の人に送ることができます。同じ場所の同じMicrosoft Excelファイルにアクセスできる人であれば、JMPでスクリプトを実行し、レポートを見ることができます。

上級者用メモ : Auto-Submit

特定のスクリプトを、スクリプトウィンドウで開くのではなく、直接実行したい場合は、スクリプトの最初の行に次のコマンドを入力します。

```
//!
```

このコマンドの入力が最初の行でなかったり、同じ行に別の文字も入力されている場合、このコマンドは効力を持ちません。

このコマンドは、ファイルを開く際に、無効にすることができます。

1. ［**ファイル**］>［**開く**］を選びます。
2. 呼び出されたダイアログにて、JSLファイルを選択し、Ctrlキーを押しながら［**開く**］をクリックします。

   これにより、スクリプトは実行されずに、スクリプトウィンドウに表示されます。

また、ホームウィンドウでファイルを右クリックし、［**スクリプトの編集**］を選択することにより、スクリプトを実行せずスクリプトウィンドウに表示させることができます。

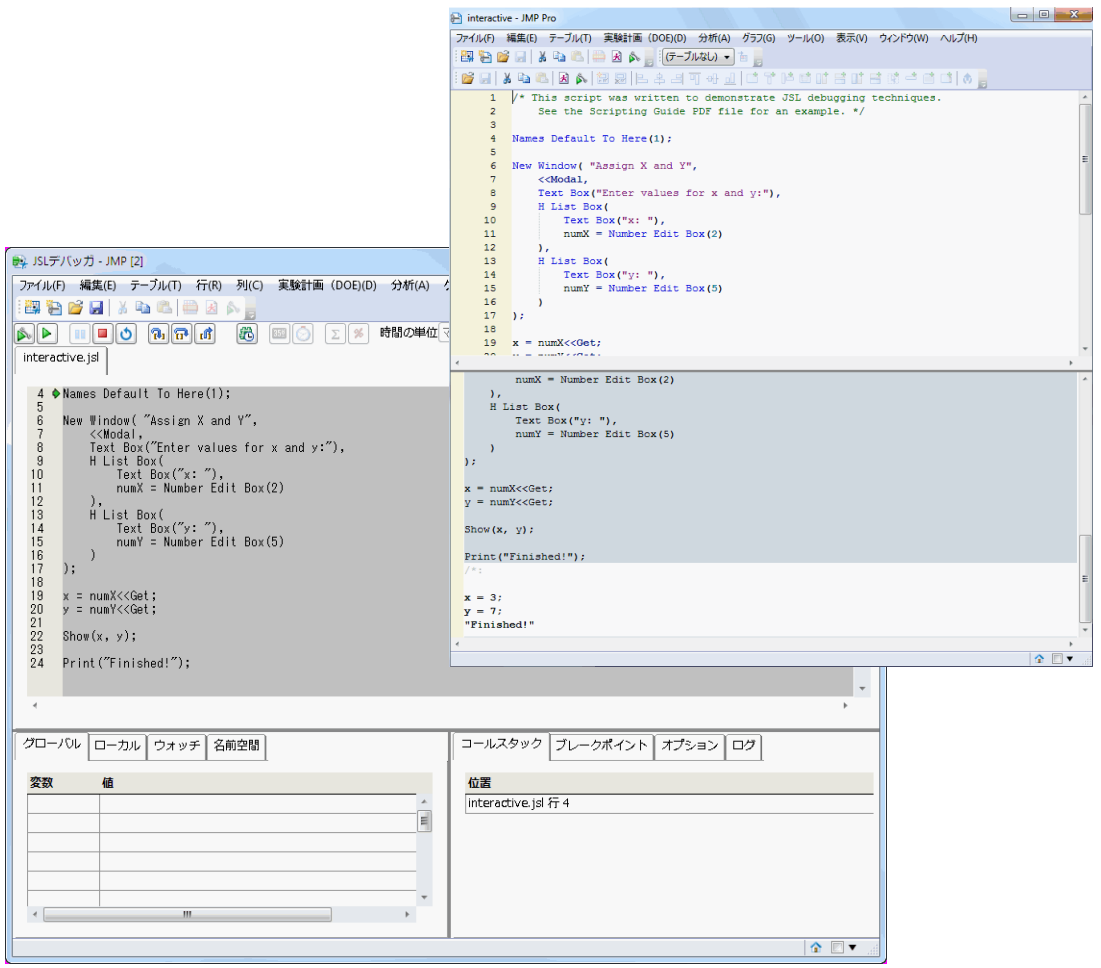
第4章

スクリプト作成のツール

スクリプトエディタ、ログウィンドウ、デバッガ、プロファイルの使用

JMP にはスクリプト作成のためのプログラミングツールが用意されています。スクリプトエディタには、構文による色分け、入力のオートコンプリート、対となる括弧の強調表示、コードの折りたたみをはじめ、スクリプトをすばやく作成するための機能が各種用意されています。エラーメッセージやコマンドの戻り値などが出力されるログウィンドウは、スクリプトエディタの中に表示することもできます。JMP スクリプト言語（JSL）デバッガおよびプロファイルは、スクリプトのトラブルシューティングに役立ちます。

図4.1 ログを含むスクリプトエディタとデバッガ

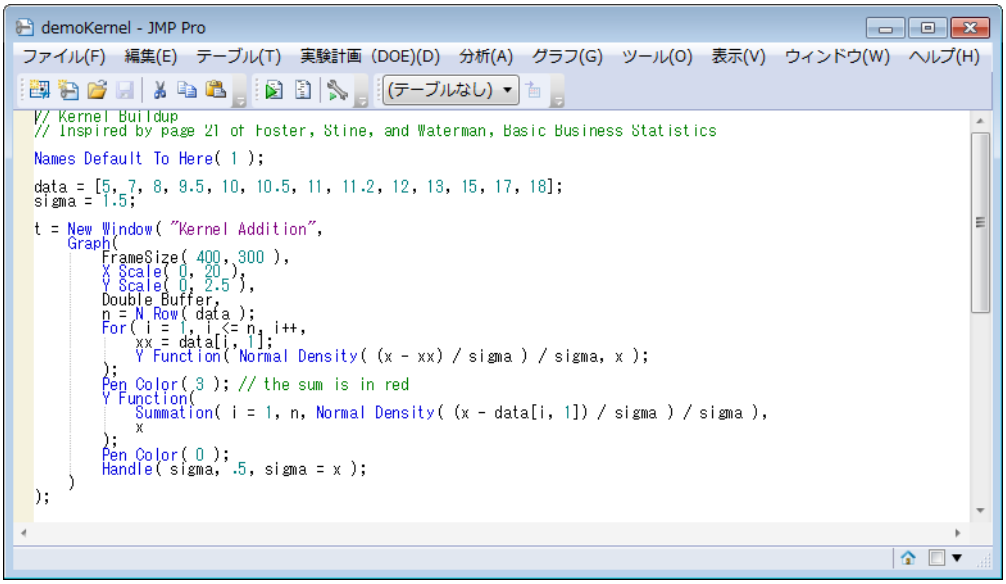


スクリプトエディタの使用

スクリプトエディタを使うと、JSLスクリプトを簡単に書いたり読んだりできます。図4.2に、構文による色分け、インラインのコメント、自動フォーマットなどの基本的な機能を示します。その他の一般的なプログラミングオプションについては、この節で後述します。

スクリプトエディタの機能は、ログウィンドウをはじめ、スクリプトの編集や記述ができる場所ならどこででも使用できます（スクリプトの索引やアプリケーションビルダーなど）。

図4.2 スクリプトエディタ



ヒント： 環境設定の［一般］グループにある項目で、自動保存の間隔を設定できます。これにより、指定した間隔（単位は分）でデータテーブルが自動的に保存されるようになります。この自動保存の設定は、データテーブル、ジャーナル、スクリプト、プロジェクト、およびレポートにも適用されます。

スクリプトの実行

スクリプト全体を実行するには、［**編集**］>［**スクリプトの実行**］を選択します。

特定の行のスクリプトのみを実行するには、それらの行を選択し、［**編集**］>［**スクリプトの実行**］を選択します。

隣接していない複数の行を実行するには、Ctrlキーを押しながら行を選択し、［**編集**］>［**スクリプトの実行**］を選択します。

Windowsでは、1行または複数の行を選択してから、数字キーパッドのEnterを押して実行することもできます。

スクリプトを開いたとき、そのスクリプトが自動的に実行されるようにするには、次のいずれかの方法を使用します。

- 最初の行に「//!」と入力しておきます。
- Open() ステートメントに Run JSL(1) を含めます。

```
Open( "$SAMPLE_SCRIPTS/scoping.jsl", Run JSL( 1 ) );
```

スクリプトの停止

スクリプトを停止するには、Windows では Esc キーを、Mac では command キーとピリオドキーを同時に押します。または、［**編集**］>［**スクリプトの停止**］を選択します。Mac では、スクリプトの実行中のみ、［**編集**］>［**スクリプトの停止**］が使用可能になります。

スクリプトの編集

Windows でスクリプトを編集するには、まず Insert キーを押します。すると、既存の JSL コードの上に直接入力（上書き）ができるようになります。この機能は、Mac では使用できません。

カラーコーディング

Windows の場合、スクリプトエディタのデフォルトのフォントは 10 ポイントの ＭＳ ゴシックです。
Macintosh の場合、スクリプトエディタのデフォルトのフォントは 12 ポイントの Courier です。

スクリプトウィンドウでは次のような色が使用されます。

- テキスト、識別子（JSL 関数）、括弧、ユーザマクロは黒
- スクリプトエディタの背景は白
- 無効になっている部分の背景とガイドはグレー
- コメントは緑
- 文字列は紫
- プラットフォーム名は栗色

メモ：プラットフォーム名がメッセージ内で使われている場合は、JSL メッセージの色が使われます。

- 数値は緑がかった青
- 演算子記号、最初のキーワード、JSL メッセージは濃い青
- 演算子名、2 番目と 3 番目のキーワード、マクロは青
- 不明なオブジェクトは赤

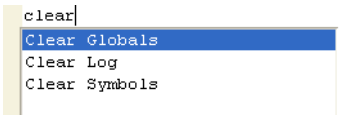
色やフォントは環境設定の［スクリプトエディタ］グループにある「スタイルのカスタマイズ」セクションでカスタマイズできます。これらのオプションは、JSL、ログ、SAS コードなどのタイプのコードに対して使用できます。

オートコンプリート機能

関数の名前を正確に覚えていないときは、自動入力機能を使えば、途中までタイプした内容に一致する関数のリストが表示されます。Windowsの場合はCtrlキーを押しながらスペースキーを押します（Macintoshではoptionキーを押しながらEscキー）。

JSL変数をクリアしたいものの、そのコマンドを覚えていないとしましょう。その場合、clearとタイプし、Ctrlキーを押しながらスペースキーを押すと、クリアコマンドの候補が表示されます。挿入したいコマンドを選択します。

図4.3 オートコンプリートの例

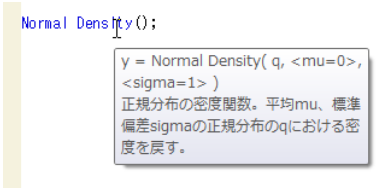


ツールヒント

関数を使用する際に構文を覚えていない場合や、詳しい情報を知りたい場合は、その関数の上にカーソルを置くと短い説明が表示されます。ツールヒントはJSL関数のみで使用でき、プラットフォームのコマンドやメッセージ、ユーザが作成した関数では使用できません。JSL関数名は、スクリプトエディタに青色で表示されます。

ツールヒントには、関数の構文や引数、簡単な説明が表示されます（図4.4）。ヒントは、スクリプトエディタウィンドウのステータスバーにも表示されます。

図4.4 JSL関数のツールヒント



スクリプトを実行した後、変数名の上にカーソルを置くと、変数の現在の値がわかります。変数のツールヒントをオフにするには、[**環境設定**]>[**スクリプトエディタ**]>[**変数値のヒントを表示する**]の選択を解除します。

関数のツールヒントをオフにするには、［**環境設定**］>［**スクリプトエディタ**］>［**演算子または関数のヒントを表示する**］の選択を解除します。

JSL変数のツールヒントの例

1. スクリプトウィンドウに次の内容を入力し、実行します。

   ```
   my_variable = 8;
   ```

2. 実行後、変数名の上にカーソルを置きます。

   ツールヒントに変数の名前と値（8）が表示されます。

3. 次の内容を入力し、実行します。

```
my_variable = "eight";
```

4. 実行後、変数名の上にカーソルを置きます。

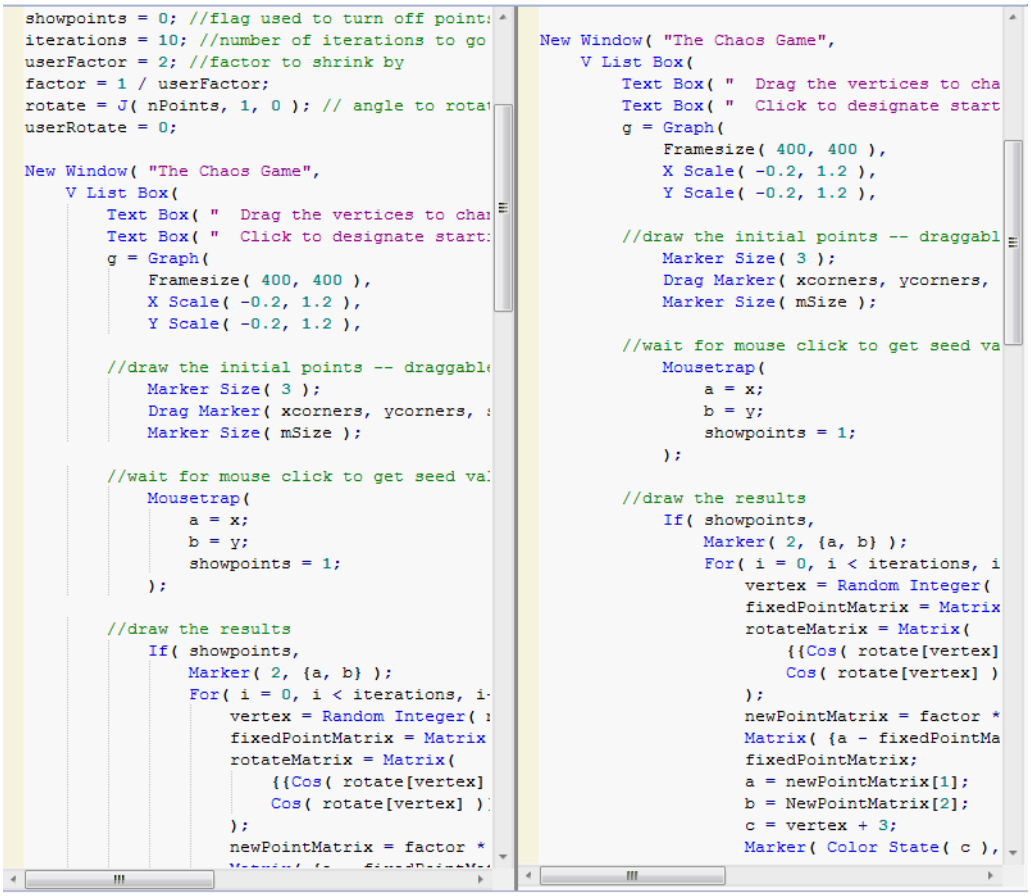
   ツールヒントに変数の名前と値（"eight"）が表示されます。

ウィンドウの分割

スクリプトエディタウィンドウは、縦または横に 2 つに分割できます。ウィンドウを分割すると、コード内の 2 つの箇所を別々にスクロールし、編集することができます。一方のウィンドウでコードに変更を加えた場合、他方のウィンドウでもただちにその変更が反映されます。

- 開いているスクリプトエディタウィンドウを分割するには、ウィンドウ内で右クリックし、[**分割**] > [**横**] または [**縦**] を選択します。
- 分割したウィンドウを元に戻すには、右クリックして [**分割の削除**] を選択します。

図4.5 ウィンドウを横に分割した例

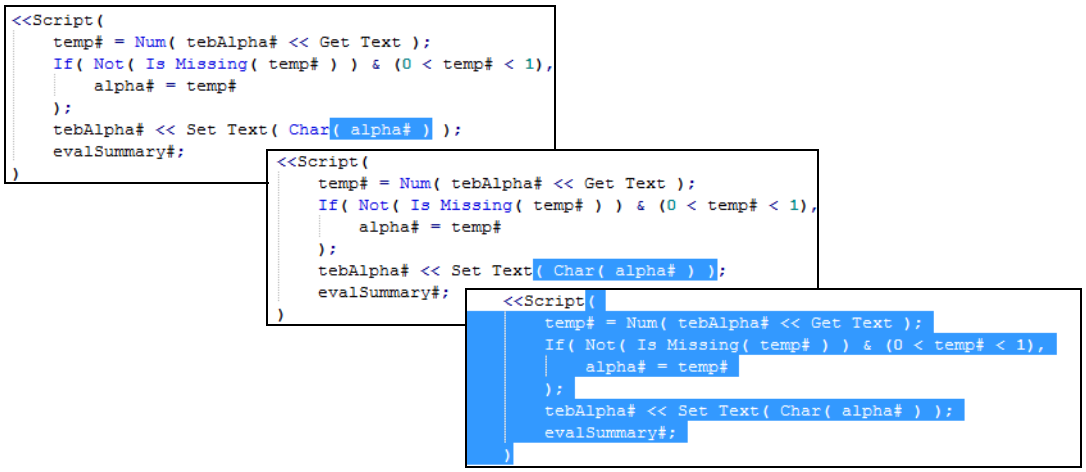


括弧の自動マッチ

スクリプトエディタでは、**括弧**（丸括弧、角括弧、中括弧）を次の方法でマッチさせられます。

- 開き括弧をタイプすると、閉じ括弧が追加されます。
- 開き括弧または閉じ括弧の横にカーソルを置くと、その括弧とそれにマッチする括弧が青で強調表示されます。マッチする括弧がない場合、括弧は赤で強調表示されます。
- 括弧をダブルクリックすると、マッチする括弧と、その間にあるものがすべて選択されます。
- 式の中にカーソルを挿入し、Ctrl+] キー（Windows）または command+B キー（Macintosh）を押すと、式全体が選択されます（式を囲む括弧を含む）。次に上位にある式を強調表示するには、このプロセスを繰り返します。図4.6に例を示します。

図4.6 括弧の自動マッチの各ステップ



開き括弧をタイプすると、閉じ括弧が自動的に追加されます。括弧の中にコードを入力した後、閉じ括弧をタイプすると、カーソルが自動的に挿入されていた閉じ括弧の後ろに移動します。これは、余計な閉じ括弧が追加されるのを防ぐ機能です。

括弧の自動マッチ機能は「環境設定」ウィンドウでオンまたはオフにできます。詳細は、「スクリプトエディタの環境設定」（60ページ）の節を参照してください。

四角形のテキストブロックの選択

四角形のテキストブロックを選択するには、Alt キーを押しながらカーソルをブロックの開始位置に置き、そのまま終了位置までドラッグします。選択したブロックは、コピーしたり切り取ったりできます。

次のコードを、コメントマークを除いて選択したいとします。

```
// Y( :Y ),
// X( :X ),
```

Yから始まる四角形の領域を選択します。それをコピーしてペーストすると、次のコードになります。

```
Y( :Y ),
X( :X ),
```

四角形の領域を選択した場合、テキストの構造を維持するために必要な個所に改行（リターン）が挿入されます。次の例では、両方の行の`Get Menu Item State`を選択します。

```
bb << Get Menu Item State(1),
bb << Get Menu Item State(2),
```

これをコピーしてペーストすると、各行の末尾に改行（リターン）が挿入されます。

```
Get Menu Item State
Get Menu Item State
```

連続していないテキストの選択

連続していないテキストを選択するには、WindowsではCtrlキーを、Macintoshではcommandキーを押しながら、そのテキスト上をドラッグします。選択したいすべてのテキストに対し、同じ操作を繰り返します。最後に、選択範囲をコピーして新しいスクリプトに貼り付けるか、選択したテキストを実行します。テキストがスクリプトに挿入されるか、選択した順序で実行されます。

テキストのドラッグ＆ドロップ

スクリプトエディタ内、ウィンドウ間、またはデータテーブルからスクリプトエディタウィンドウで、テキストをドラッグ＆ドロップできます。Windowsで、テキストのコピーをドロップしたいときは、まずCtrlキーを押してからドラッグ＆ドロップします。Macintoshの場合、テキストはデフォルトでコピーされます。

テキストをドラッグ＆ドロップするには、次の手順を行います。

- データテーブルの行または列を選択し、少し間を置いてからスクリプトエディタウィンドウにドロップします。
- テキストフィールド内のテキストをダブルクリックし、スクリプトエディタウィンドウにドロップします。データテーブルのセルに含まれているテキストなど、選択可能なテキストすべてに適用できます。

Windowsの場合、最小化したウィンドウにテキストをドラッグ＆ドロップすることもできます。

1. テキストを、ウィンドウの右下隅にあるホームウィンドウボタン の上にドラッグします。

   ホームウィンドウが開きます。
2. 「ウィンドウリスト」にある任意のウィンドウ名の上にテキストをドラッグします。

   該当するウィンドウが表示されます。
3. 任意の場所にテキストをドロップします。

検索／置換

スクリプトウィンドウには、正規表現のサポートを含め、多くの検索／置換オプションが用意されています。たとえば、次のような正規表現で検索できます。

```
get.*name
```

これは、「Get Button Name」、「GetFontName」などのメッセージを戻します。

^および$（それぞれ行の冒頭および行の末尾に一致）や/n（改行文字に一致）などの基本的な正規表現もサポートされています。

「検索」ウィンドウの詳細については、『JMPの使用法』の「データの入力と編集」章を参照してください。

自動フォーマット

スクリプトエディタでは、読みやすいようにスクリプトの体裁を整えることができます。JMPで（たとえばプラットフォームスクリプトの保存時に）生成されるスクリプトには、適切な場所に自動的にタブや改行が挿入されます。

また、手で入力したスクリプトの場合も、読みにくいようであれば（すべてのコマンドが空白文字を挟まずにつながっているなど）の体裁を整えることもできます。［**編集**］メニューから［**スクリプトを再フォーマット**］を選択します。

ヒント： このコマンドは、スクリプトに括弧が一致していないなどの不具合がある場合に警告メッセージを表示します。

コード折りたたみマーカーの追加

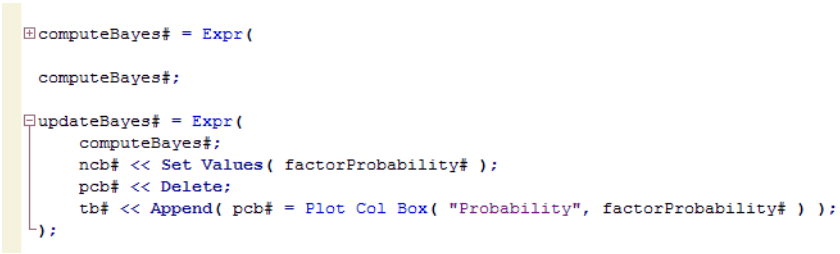
コード折りたたみマーカーを使ってコードのブロックの開始位置と終了位置を指定すると、単独の関数内のコードを折りたたんだり、展開したりできます。

この機能を使用するには、［スクリプトエディタ］の環境設定で［**JSLコードの折りたたみ**］をオンにします。コードのブロックを展開する（または折りたたむ）には、スクリプトを右クリックし、［**詳細**］>［**すべてを展開する**］または［**すべてを折りたたむ**］を選択します。

ヒント： スクリプト全体のすべてのコードを折りたたんだり展開したりするには、Shiftキーと Ctrlキーを押した状態でコードマーカーをクリックします。

この環境設定を選択すると、FunctionとExprの式が折りたためるようになります。他の式に折りたたみマーカーを追加する方法については、「コード折りたたみのキーワードの追加」（59ページ）を参照してください。

図4.7 スクリプトに表示されたコード折りたたみマーカー



デフォルトでは、スクリプトを保存してJMPを再開した場合、コードは折りたたんだ状態で表示されます。コードの折りたたみ状態を保存するには、［スクリプトエディタ］環境設定で［**コードの折りたたみ情報を保存および復元する**］を選択します。

コード折りたたみのキーワードの追加

その他の単独の関数にも、次のような指定をすることによりコードの折りたたみを適用できます。

```
{"If", "For", "For Each Row", "While", "Try", "New Window", "V List Box", "H List
  Box"}
```

JMPは、複数のキーワードを含むリストをサポートしています。システム管理者はjmpKeywords.jslに一連のキーワードを定義し、C:¥ProgramData¥SAS¥JMP¥または以下に示す指定のディレクトリに保存します。ユーザは、自分で作成したjmpKeywords.jslをC:¥Users¥<ユーザ名>¥Documents¥フォルダに保存できます。JMPは、指定のディレクトリにあるキーワードリストをすべて結合します。

Windowsの場合、次のディレクトリがこの順番に調べられます。

- C:¥ProgramData¥SAS¥JMP¥13¥
- C:¥ProgramData¥SAS¥JMP¥
- C:¥Users¥<ユーザ名>¥AppData¥Roaming¥SAS¥JMP¥13¥
- C:¥Users¥<ユーザ名>¥AppData¥Roaming¥SAS¥JMP¥
- C:¥Users¥<ユーザ名>¥Documents¥

Macintoshの場合、次のディレクトリがこの順番に調べられます。

- /Library/Application Support/JMP/13/
- /Library/Application Support/JMP/
- ~/Library/Application Support/JMP/13/
- ~/Library/Application Support/JMP/
- ~/Documents/

JMP Proを使っている場合も、jmpKeywords.jslは（JMPProではなく）JMPディレクトリに保存してください。

メモ：

- jmpKeywords.jsl内のリストは大文字と小文字を区別します。

- コードの折りたたみは、メッセージやプラットフォーム、ユーザ定義の関数、コメントには使用できません。
- jmpKeywords.jsl 内のリストを編集し、保存した後、[**コード折りたたみのキーワードの追加を許可する**] 環境設定をオフにしてから再度オンにして、変更を有効にします。キーワードがロードされたことを示すメッセージがログに出力されます。

詳細オプション

スクリプトエディタでテキストの一部を選択して右クリックし、[詳細] をポイントすると、次のようなオプションが表示されます。

すべてを展開する （JSL コードの折りたたみ機能が有効な場合のみ表示されます。）コードのブロックをすべて展開します。

すべてを折りたたむ （JSL コードの折りたたみ機能が有効な場合のみ表示されます。）コードのブロックをすべて折りたたみます。

ブロックのコメント化 選択したテキストがコメントになります。

ブロックのコメント化解除 選択したコメントがコメントでなくなります。

大文字にする 選択したテキストが大文字に変わります。

小文字にする 選択したテキストが小文字に変わります。

スクリプトエディタの環境設定

JMP の環境設定では、フォント、色、スペースのオプションといったスクリプトエディタの設定をカスタマイズできます。

フォントの設定

1. [**ファイル**] > [**環境設定**] を選択します。
2. [**フォント**] グループを選択します。
3. [**モノ**] をクリックして、スクリプトエディタ用のフォントを設定します。

フォントの環境設定の詳細については、『JMP の使用法』の「JMP の環境設定」章を参照してください。

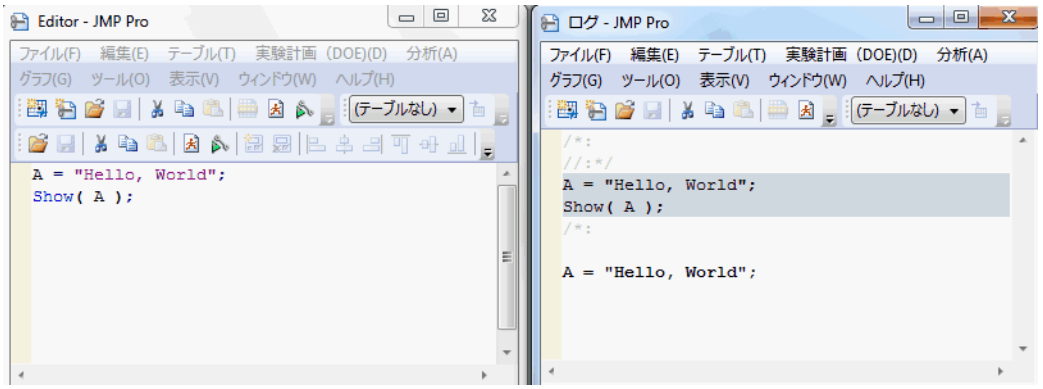
スクリプトエディタの環境設定

スクリプトエディタをより詳細にカスタマイズするには、[**ファイル**] > [**環境設定**] > [**スクリプトエディタ**] を選択します。フォントの環境設定の詳細については、『JMP の使用法』の「JMP の環境設定」章を参照してください。

ログの使用

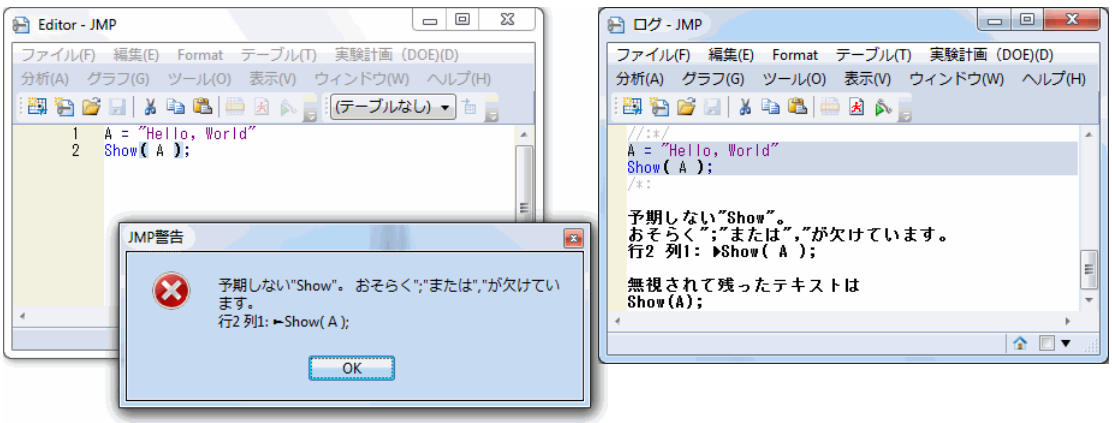
スクリプトを実行すると、ログウィンドウに出力が表示されます。実行されたスクリプトはログ内でグレー表示され、その下に出力が表示されます（図4.8）。

図4.8 スクリプトウィンドウ（左）とログウィンドウ（右）



構文エラーと互換性エラーは、行番号と処理できなかったコードとともにログ内に出力されます(図4.9)。

図4.9 構文エラーメッセージ



［**表示**］>［**ログ**］を選択してログウィンドウを表示します（Macintoshの場合は［**ウィンドウ**］>［**ログ**］）。

ヒント:

- Windows の場合、ログを表示するタイミングを、JMP が起動したとき、テキストが書き込まれたとき、または明示的に開いたときのいずれかに指定できます。JMP ログウィンドウを開く設定を変更するには、［**ファイル**］>［**環境設定**］>［**Windows のみ**］を選択します。

- 互換性に関する警告メッセージをログから除外するには、JMPの［一般］環境設定で［**JMP 12でJSLの互換性に関する変更がある場合、ログに警告を表示**］をオフにします。
- ログのフォントと色は［スクリプトエディタ］環境設定で変更できます。詳細は、「カラーコーディング」（53ページ）の節を参照してください。

スクリプトウィンドウ内にログを表示

スクリプトウィンドウ内にログを表示するには、ウィンドウ内を右クリックし、［**ウィンドウ内にログを表示**］を選択します。このオプションを使えば、スクリプトを編集、実行し、変更の結果をすばやく確認してスクリプトの作成を継続することがより簡単になります。

スクリプトの索引のスクリプトウィンドウには、埋め込みのログが常に表示されますが、アプリケーションビルダーおよびデバッガには表示されません。

ログの保存

以下の手順でログをテキストファイルとして保存すると、どんなテキストエディタでも開くことができます。ログのテキストファイルをダブルクリックすると、JMP ではなく、普段使っているテキストエディタでログが開きます。

1. 「ログ」ウィンドウをアクティブにします（「ログ」ウィンドウをクリックして最前面に持ってきます）。
2. ［**ファイル**］メニューの［**上書き保存**］（Macintoshでは［**保存**］）か、［**名前を付けて保存**］（Macintoshでは［**別名で保存**］）を選択します。
3. Windowsの場合は、拡張子.txtをつけたファイル名を指定します。Macintoshの場合は.txtの拡張子では保存できず、常に.jslの拡張子となります。
4. ［**保存**］をクリックします。

スクリプトのデバッグ／プロファイル

開いたスクリプト内で、［**スクリプトのデバッグ**］ボタン をクリック（または［**編集**］>［**スクリプトのデバッグ**］を選択）すると、スクリプトがJSLデバッガウィンドウに表示されます。次のようなショートカットキーを使用することもできます。

- Ctrl + Shift + R（Windows）
- Shift + Command + R（Macintosh）

JSLデバッガを使うと、スクリプト内でエラーが発生したり、動作しない場所を特定することができます。スクリプトの一部をコメントアウトしたり`Print()`を追加したりしなくても、デバッガを使って問題を検出できます。

JSLデバッガが開いたら、そのまま作業を行うか、[**JSLスクリプトのプロファイル**] ボタン をクリックしてJSLプロファイルモードに切り替えます。JSLプロファイルは、スクリプトを効率的にする手助けになります。スクリプトの実行中にプロファイルが作成され、特定の行の実行にかかった時間や、特定の行が実行された回数などがわかります。

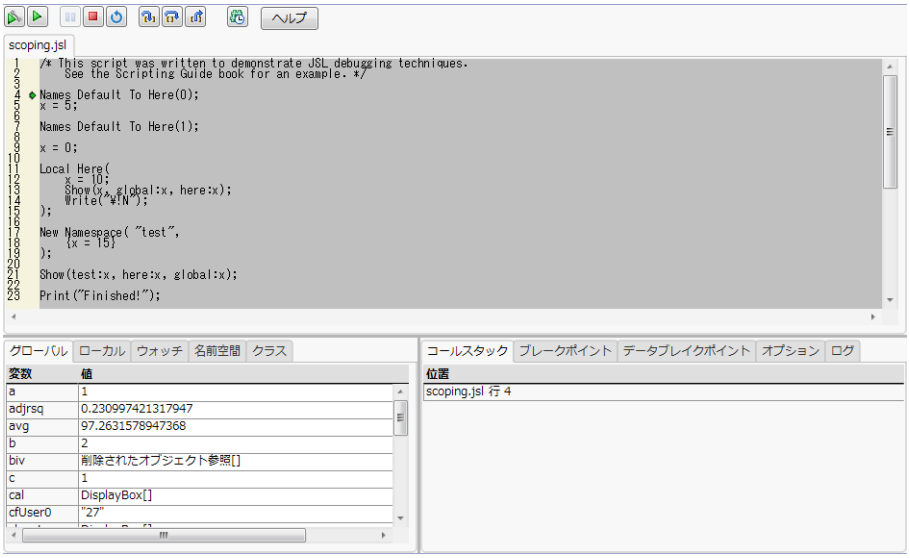
ヒント： スクリプトのデバッグを自動的に行うには、Open() ステートメントに Run JSL(1) を含めます。

```
Open( "$SAMPLE_SCRIPTS/scoping.jsl", Run JSL( 1 ) );
```

デバッガとプロファイルのウィンドウ

デバッガはJMPの新しいインスタンスとして起動します（図4.10）。元のインスタンスは、スクリプトによって何らかの操作を必要とするものが作成されるまで、操作不可能になります。操作の必要なものが作成された場合、ユーザがそれを実行するまで、デバッガウィンドウの方が操作不可能になります。その後、制御が再びデバッガに戻ります。JMPの元のインスタンスでの作業に戻るには、デバッガを閉じなければなりません。

図4.10 デバッガウィンドウ



デバッガとJSLプロファイルを操作するには、上部にあるボタンを使用します。デバッグまたはプロファイル作成の対象であるスクリプトがタブに表示されます。スクリプトの中に他のスクリプトが含まれているときは、それぞれが新しいタブに表示されます。

デバッガの最下部にあるタブには、変数や名前空間、ログ、現在の実行ポイントを表示するためのオプション、ブレークポイントを操作するためのオプション、および設定オプションがあります。

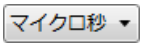
実行ボタンの使用

デバッガまたはJSLプロファイル内でスクリプトの実行を制御するには、上部にあるボタンを使用します。

表4.1 デバッガボタンの説明

Button	ボタン名	アクション
	Run	デバッガ内のスクリプトを、ブレークポイントまたはスクリプトの末尾まで実行します。 **メモ：**スクリプトにエラーがある場合は、ウィンドウの上部に短縮された形式のエラーメッセージが表示されます。[**より詳細**]をクリックすると、完全なエラーメッセージが表示されます。
	ブレークポイントなしで実行 プロファイルの実行	スクリプトを、途中で停止することなく最後まで実行します。
	すべて中断	たとえば、非常に長いループを実行しているときなど、スクリプトがビジーであるとき、[すべて中断]をクリックすると、スクリプト内のすべてのアクションが中断されてデバッガまたはJSLプロファイルに戻ります。 実行中のスクリプトが、ダイアログの入力や開いたウィンドウへの操作など、ユーザの操作を待機している場合、デバッガおよびJSLプロファイルは実行を中断できないことがあります。
	Stop	スクリプトのデバッグを停止し、デバッガまたはJSLプロファイルを終了します。
	Restart	デバッガの現在のセッションを閉じ、新しいセッションを開きます。
	ステップイン	関数内またはインクルードされたファイル内にステップインします。それ以外は、ステップオーバーと同じ働きをします。
	ステップオーバー	呼び出された式、関数、または`Include()`ファイルにステップインすることなく、1行のすべての式、または複数行にまたがる複雑な式を実行します。
	ステップアウト	現在のスクリプトまたは関数をブレークポイントまたは最後まで実行し、呼び出しポイントまで戻ります。主スクリプトにおいてデバッガが最後まで到達した場合は、「プログラムの実行が終了しました」というメッセージが表示されます。デバッガ自体は開いたままなので、プログラムの最終的な状態を点検することができます。

表4.1 デバッガボタンの説明（続き）

Button	ボタン名	アクション
	JSL スクリプトのプロファイル	JSL プロファイルを開きます。(JSL プロファイルを開始するには、［プロファイルの実行］ボタンを押します。) JSL プロファイルを使うと、特定の行の実行にかかった時間や、特定の行が実行された回数などがわかります。次の点を念頭に置いてください。 • デバッガと JSL プロファイルの間を自在に切り替えることができるのは、プログラムの開始前のみです。 • デバッガのボタンの一部は、プロファイルの実行中は無効になります。 • JSL プロファイルモードでは、ブレークポイントがすべて無効になります。
	行の実行回数のプロファイルを表示	各行が何回実行されたかを表示します。
	行の処理時間のプロファイルを表示	行の実行にかかった時間を表示します。
	総数でプロファイルを表示	実行回数の場合は、その行が何回実行されたかを表示します。処理時間の場合は、その行が完了するまでにかかったマイクロ秒数（またはミリ秒数、秒数）を表示します。
	パーセントでプロファイルを表示	実行回数の場合は、個々の行の実行回数を総数で割った値を表示します。処理時間の場合は、個々の行にかかった時間の割合（行の処理時間÷合計時間×100）を表示します。
	時間の単位	時間の単位をマイクロ秒、ミリ秒、秒のいずれかに設定します。JSL プロファイルで［プロファイルの実行］ボタン ▶ をクリックすると、使用できるようになります。
	カラーテーマ	JSL プロファイルの陰影の色を設定します。JSL プロファイルで［プロファイルの実行］ボタン ▶ をクリックすると、使用できるようになります。

変数リスト

デバッガの左下にあるタブでは、グローバル変数やローカル変数、ウォッチ変数、名前空間内の変数が確認できます。

グローバル ［グローバル］タブにはグローバル変数がリストされ、スクリプトの各ステップを実行するにつれて値が更新されます。変数は、初期化の際に追加されます。実行済みのスクリプトによってグローバル変数が定義されている場合は、デバッガの起動時に、それらの変数と現在の値が表示されます。詳細については、「変数の確認」（69ページ）を参照してください。

ローカル　［ローカル］タブにはすべての変数がスコープ別にリストされ、スクリプトの各ステップを実行するにつれて値が更新されます。メニューからスコープを選択できます。詳細については、「変数の確認」（69ページ）を参照してください。

ウォッチ　コードの各ステップを実行していく際、値を確認したい変数または式の値がある場合は、ここに追加します。これは、スクリプトで多くの変数を使用していて、［グローバル］や［ローカル］のリスト内で確認するのが難しい場合に特に便利です。詳細については、「ウォッチの操作」（70ページ）を参照してください。

名前空間　定義された名前空間は、このメニューに追加されます。名前空間を選択すると、名前空間内で使用されている変数とその値が表示されます。詳細については、「変数の確認」（69ページ）を参照してください。

デバッガオプション

デバッガの右下のタブでは、コールスタックの表示、ブレークポイントの操作、オプションの設定、ログの表示ができます。

コールスタック　コールスタックには、スクリプト内、関数内の現在の実行ポイントがリストされます。主スクリプトは常にリストの最初に表示されます。関数を呼び出すと、その関数は呼び出しスクリプトの上に追加されます。同様に、インクルードされたファイルもリストの一番上に追加されていきます。関数またはスクリプトの実行を終了すると、それがリストから削除され、次の関数またはスクリプトに戻ります。各ステップを実行するに従って、現在の行番号が更新されます。

コールスタック内の行をダブルクリックすると、カーソルがその行に移動します。

ブレークポイント　行に対してブレークポイントを追加、編集、削除したり、無効または有効にしたりできます。詳細については、「ブレークポイントの操作」（66ページ）を参照してください。［ブレークポイント］タブで行をダブルクリックすると、カーソルがその行に移動します。

データブレークポイント　変数に対してブレークポイントを追加、編集、削除したり、無効または有効にしたりできます。データブレークポイントは、行内の変更を監視するブレークポイントとは異なり、変数内の変更を監視します。

オプション　このタブでは、デバッガの環境設定をインタラクティブに設定できます。詳細については、「デバッガでの環境設定の変更」（71ページ）を参照してください。

ログ　デバッグ中のスクリプトのログが表示されます。

ブレークポイントの操作

ブレークポイントはスクリプトの実行を中断します。スクリプトのデバッグは一行ずつ進めることができますが、スクリプトが長い、または複雑な場合、退屈で時間のかかる作業になってしまいます。そのような場合、確認したい位置にブレークポイントを設定し、デバッガでスクリプトを実行することができます。スクリプトは、ブレークポイントに達するまで通常どおりに実行されます。ブレークポイントの位置で実行が中断されるので、変数の値を確認したり、そこから一行ずつのデバッグを開始したりできます。

右下隅の［ブレークポイント］タブに表示されるブレークポイントは行内の変更を監視します。［データブレークポイント］タブに表示されるブレークポイントは変数内の変更を監視します。

JMPは、セッションを越えてブレークポイントを保持するため、JMPを閉じて再度開いても、ブレークポイントは同じ位置に表示されます。

ヒント： 行番号をオンにするには、スクリプト内を右クリックし、**［行番号を表示する］**を選択します。すべてのスクリプトでデフォルトで行番号を表示させるには、スクリプトエディタの環境設定で［行番号を表示する］をオンにします。

ブレークポイントの作成

ブレークポイントを作成する際、条件やブレーク時の動作を指定することができます。それには、［ブレークポイント］タブまたは［データブレークポイント］タブの  をクリックし、ブレークポイントの情報を入力します。

また、次のいずれかの手順ですばやくブレークポイントを作成できます。

- デバッガの余白で、該当する行をクリックします（行番号が表示されている場合は、行番号の右側）。
- デバッガの余白で、ブレークポイントを設定したい位置を右クリックし、**［ブレークポイントの設定］**を選択します。

ブレークポイントを挿入した箇所と［ブレークポイント］タブに、赤いブレークポイントアイコンが表示されます。

ブレークポイントの削除

次のいずれかを実行します。

- デバッガの余白で、ブレークポイントアイコンをクリックします。
- デバッガの余白で、ブレークポイントアイコンを右クリックし、**［ブレークポイントのクリア］**を選択します。
- ［ブレークポイント］タブでブレークポイントを選択し、  をクリックします。
- ［ブレークポイント］タブで  をクリックし、（選択したブレークポイントだけではなく）すべてのブレークポイントを削除します。

赤いブレークポイントアイコンが消え、［ブレークポイント］タブにもブレークポイントが表示されなくなります。

ブレークポイントの有効化および無効化

スクリプトのエラーを修正した後、そのブレークポイントを通過して正常に実行できるかどうかを確認するには、ブレークポイントを無効にします。ブレークポイントは、いつでも必要なときに有効にできるため、作成し直す必要がありません。

次のいずれかを実行します。

- デバッガの余白で、ブレークポイントアイコンを右クリックし、［**ブレークポイントを有効にする**］または［**ブレークポイントを無効にする**］を選択します。
- ［ブレークポイント］タブで、ブレークポイントのチェックボックスを選択するか、選択を解除します。
- ［ブレークポイント］タブで  をクリックし、すべてのブレークポイントを無効または有効にします。

無効になったブレークポイントのアイコンは白、有効になったブレークポイントのアイコンは赤で表示されます。

ブレークポイントへの条件式の指定およびクリア

コードを1ステップずつデバッグする代わりに、ブレークポイントに条件を設定することができます。1ステップごとに実行して各式の変数を確認するのではなく、条件が合ったときにだけ、スクリプトを中断するよう指定できます。中断したら、コードをステップごとに実行して問題が生じる位置を特定できます。

たとえば、スクリプト内の計算が間違っていて、問題が起こるのは`i==19`のときだと推測しているとしましょう。この場合、`i==18`に条件付きブレークポイントを設定しておくと、その条件が満たされた時点でデバッガによる実行が中断されます。その後、コードを1ステップずつ確認して問題を特定します。

ブレークポイントの条件の指定

1. ブレークポイントアイコンを右クリックし、［**ブレークポイントの編集**］を選択します。
2. ［条件］タブで［**条件**］を選択し、条件式を入力します。
3. 式が［**真の場合**］または［**変更された場合**］のどちらに中断するかを指定します。
4. ［**OK**］をクリックします。

条件の無効化または有効化

1. ブレークポイントアイコンを右クリックし、［**ブレークポイントの編集**］を選択します。
2. ［条件］タブで、［**条件**］を選択解除または選択します。

条件の削除

［ブレークポイント］タブで、ブレークポイントの条件フィールドをクリックし、Deleteキーを押します。

ブレークオプションの指定

ブレークポイントを右クリックして［**ブレークポイントの編集**］を選択すると、ブレークポイントの動作を簡単に管理できます。または、［ブレークポイント］タブでブレークポイントを選択し、  をクリックします。どちらの方法でも「ブレークポイント情報」ウィンドウが開くので、［ヒットカウント］タブと［アクション］タブで設定をカスタマイズします。

ヒットカウントの変更

ブレークポイントがヒットする回数を指定して、中断のタイミングを制御します。たとえば、条件が2回満たされたときに中断させるには、［ヒットカウント］タブで［**ヒットカウントが指定された値と等しい場合に中断:**］を選択し、「2」をタイプします。

アクションの定義

ブレークポイントがヒットしてスクリプトの実行が中断されたときに、デバッガにスクリプトを実行させることができます。このスクリプトを**アクション**といいます。［アクション］タブで、実行するスクリプトを入力します。

カーソルの位置までスクリプトを実行

右クリックして［**カーソルの位置まで実行**］を選択すると、カーソルの位置より前にあるすべての式が実行されます。現在の行までの値だけを確認したいときにこのオプションを選択します。各式が実行されたときの値を確認するには、ステップオプションを使用します。

ブレークポイントの設定に関するヒント

- あるループの中でエラーの確認が不要な場合は、ループが終わった後の位置にブレークポイントを設定します。デバッガは、ループを1行ずつ実行するのではなく、次のブレークポイントまで実行します。
- アクションをトリガしない行（コメント、改行、閉じ括弧など）にはブレークポイントを挿入しないでください。これらの行では中断できません。
- いったんブレークポイントを挿入すると、デバッガを閉じてスクリプトを編集しても、ブレークポイントが元の行番号とともに残ります。必要なら、ブレークポイントを削除し、挿入し直してください。
- ブレークポイントはデバッガのセッションを越えて保存されます。つまり、ブレークポイントリストには、現在デバッグを行っているスクリプトだけでなく、すべてのスクリプトに設定されたブレークポイントが含まれます。
- ブレークポイントは、スクリプトにではなく、デバッガセッションによって保存されます。つまり、移動または削除されたスクリプトのブレークポイントであってもリストされます。
- ［ブレークポイント］タブで  をクリックすると、スクリプトが開いているかどうか、スクリプトがまだ存在するかどうかに関係なく、スクリプト内のブレークポイントがすべて削除されます。

変数の確認

変数のリストの情報は、スクリプト内で変数が使用されるたびに集められます。変数の値は、スクリプトによって変更されるたびに更新されます。スクリプトの実行中、変数がなぜその値になるのかが不明な場合は、ステップごとの変数の値をウォッチして、経過を確認できます。

また、変数に任意の値を割り当てることもできます。たとえば、For()ループを1ステップずつ実行していて、特定の回の反復処理を開始したときに何が起こっているのかということだけに関心がある場合は、反復を制御している変数にその値を割り当てることができます。ループの冒頭でその変数に値を割り当てる部分をとばし、その後、変数リストの中でその変数に値を割り当てます。そして1ステップずつ実行すると、ループがその位置から始まります。

変数の管理に関するヒント

- グローバルスコープを使用する複数のスクリプトを実行した場合は、グローバル変数をクリアまたは削除しましょう。そうすることで、デバッガの変数のリストが、余計なものを含まないコンパクトなものになります。削除には、Delete Symbols()関数を使います。また、JMPを閉じて再起動してもグローバルの名前空間がクリアされます。
- スクリプトに使用している変数の数が多く、変数リストで見つけてウォッチするのが難しい場合、関心のある変数にウォッチを追加します。

ウォッチの操作

JMP は、セッションを越えてウォッチ変数を保持するため、JMP を閉じて再度開いても、ウォッチ変数は［ウォッチ］タブにリストされたままになります。

ウォッチの作成

- ［ウォッチ］タブで  をクリックし、ウィンドウに値を入力します。
- デバッガで、ウォッチする行を右クリックして［**ウォッチに追加**］を選択し、ウィンドウに変数名を入力します。
- デバッガで、変数名またはその横にカーソルを置き（または変数名を選択し）、右クリックして［**ウォッチに追加**］を選択します。
- ［ウォッチ］タブで、空の**「変数」**フィールドに変数を入力します。

ウォッチの変更

ウォッチする変数を変更するには、［ウォッチ］タブで次のいずれかを行います。

- ウォッチを選択し、  をクリックします。
- 「変数」フィールドをクリックし、新しい変数名を入力します。

ウォッチの削除

ウォッチを削除するには、［ウォッチ］タブで次のいずれかを行います。

- ウォッチを選択し、  をクリックします。
- すべてのウォッチを削除するには、  をクリックします。

デバッガでの環境設定の変更

デバッガでの作業に適用される環境設定は、変更が可能です。［オプション］タブには、次のような設定があります。

行番号を表示 デバッガ内でスクリプトの行番号の表示/非表示を切り替えます。

行あたりに複数のステートメントがあれば中断 1行に複数の式がある場合、各式ごとにスクリプトの実行を中断します。

例外時に中断 Try() 関数内で例外エラーがスローされたときにスクリプトの実行を中断します。なお、Try() の括弧内で Throw() を実行させても、例外エラーをスローできます。デバッガは、式の残りを実行せずに Throw() で中断します。これにより、どこで問題が生じているのかが特定でき、デバッグに戻れます。

実行エラー時に中断 エラー発生時に、デバッガを停止するのではなく、スクリプトの実行を中断します。

条件に割り当て式が入力されたら警告 ブレークポイントに割り当て式を含む条件が入力されると、警告を表示します。たとえば、x = 1にあるブレークポイントにx = 1という条件を追加すると、xの割り当て式を確認するための警告が表示されます。

プログラム終了後デバッガに戻る JSL プログラムの実行が終わったとき、デバッガを開いたままにします。デフォルトではオンになっていて、実行したプログラムの状態を確認することができます。

デバッガセッションの持続性

ブレークポイントやウォッチは、ユーザが削除するまで保存されたままです。タブの幅やデバッガウィンドウのサイズなど、ユーザ固有の設定は、JMP のセッションの間保持されます。

これらの設定は JMP.jdeb という名前のファイルにまとめられ、USER_APPDATA 変数で定義された場所に保存されます。

- Windows 7以降: "C:¥Users¥<ユーザ名>¥AppData¥Roaming¥SAS¥JMP¥13¥"
- Macintosh: "/Users/<ユーザ名>/Library/Application Support/JMP/13/"

ローカル変数やグローバル変数、名前空間の値は、通常どおり、JMP を閉じてもう一度開くとクリアされます。

メモ: Windows では、JMP のバージョンによってパスが異なります。JMP Pro の場合、「JMP」の部分は「JMPPro」になります。JMP Shrinkwrap（シングルユーザーライセンス）の場合は、「JMP」の部分が「JMPSW」になります。

スクリプトのデバッグとプロファイルの例

ここでは、変数をウォッチするためのブレークポイントの設定、式のステップイン、ステップオーバー、ステップアウト、異なるスコープや名前空間での変数のウォッチ、インタラクティブなスクリプトのデバッグ、JSL プロファイルを使ったスクリプトのデバッグの例を紹介します。

例で使っているスクリプトは「Samples¥Scripts」フォルダにあります。

ヒント： まず、デバッガの［オプション］タブで［**行番号を表示**］が選択されていることを確認してください。

ブレークポイントの使用とグローバル変数のウォッチ

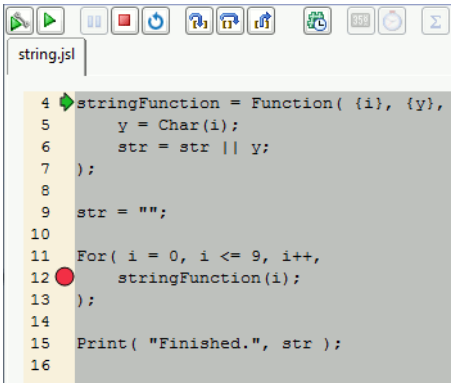
次の例では、ループ内にブレークポイントを設定し、ループの各反復における変数の変化をウォッチする方法を紹介します。

1. 「string.jsl」サンプルスクリプトを開き、［**スクリプトのデバッグ**］ボタンをクリックします。
2. 12行目の余白をクリックしてブレークポイントを追加します（図4.11）。

   これにより、For() ループの中の次の式にブレークポイントが設定されます。

   ```
   stringFunction(i);
   ```

図4.11 ブレークポイントの設定



3. ［**実行**］をクリックします。

   最初の2つの式が評価されます。

   – stringFunctionは関数として定義されます。

   – strは空の文字列として定義されます。

   両方の変数が、そのタイプと値とともに［グローバル］リストに追加されました。さらに、For() ループが、図4.11で示したブレークポイントのある行まで評価されました。

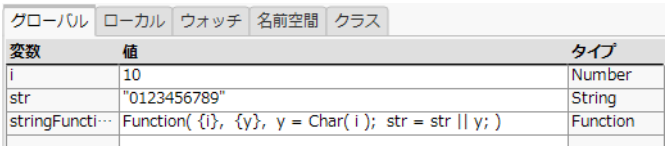
   – iに0が割り当てられました。

   – iが、その値とタイプとともに［グローバル］リストに追加されました。

   – iは9以下であると判断されました。

   – stringFunction() はまだ呼び出されていません。

図4.12 開始時のグローバル変数


グローバル ローカル ウォッチ 名前空間 クラス

変数	値	タイプ
i	10	Number
str	"0123456789"	String
stringFuncti…	Function({i}, {y}, y = Char(i); str = str || y;)	Function

4. もう一度［**実行**］をクリックします。

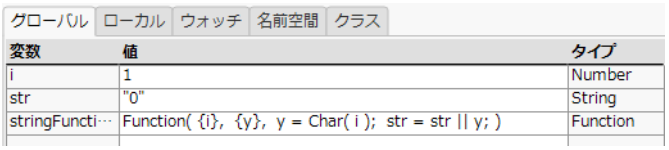
   スクリプトはブレークポイントに達するまで実行されます。結果は図4.13のようになります。

   – `stringFunction()`が呼び出され、評価され、その後、ループに戻りました。

   – `i`がインクリメントされ、9以下であると判断されました。

   – ［グローバル］リストで、`i`は1、`str`は“0”に変更されました。

図4.13 1回目のブレークポイントでのグローバル変数


グローバル ローカル ウォッチ 名前空間 クラス

変数	値	タイプ
i	1	Number
str	"0"	String
stringFuncti…	Function({i}, {y}, y = Char(i); str = str || y;)	Function

5. もう一度［**実行**］をクリックします。

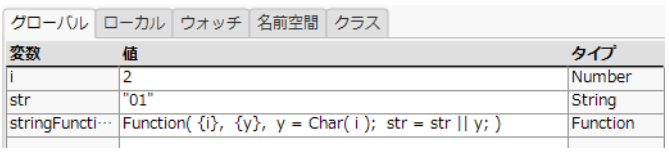
   スクリプトはブレークポイントに達するまで実行されます。結果は図4.14のようになります。

   – `stringFunction()`が呼び出され、評価され、その後、ループに戻りました。

   – `i`がインクリメントされ、9以下であると判断されました。

   ［グローバル］リストで、`i`は2、`str`は“01”に変更されました。

図4.14 2回目のブレークポイントでのグローバル変数


グローバル ローカル ウォッチ 名前空間 クラス

変数	値	タイプ
i	2	Number
str	"01"	String
stringFuncti…	Function({i}, {y}, y = Char(i); str = str || y;)	Function

続けて［**実行**］をクリックし、ループの各反復での`i`と`str`の変化をウォッチすることができます。または、［**ブレークポイントなしで実行**］をクリックしてスクリプトの実行を完了し、デバッガを終了します。

ステップイン、ステップオーバー、ステップアウト

［**ステップイン**］、［**ステップオーバー**］、［**ステップアウト**］を使うと、スクリプトに式や関数、他のJSLファイルが含まれている場合に、より柔軟にデバッグできます。

1. 「scriptDriver.jsl」サンプルスクリプトを開き、［**スクリプトのデバッグ**］ボタンをクリックします。

2. このスクリプトはログに情報を書き込むので、メッセージを確認できるよう、デバッガの最下部にある［ログ］タブを選択します。
3. ［**ステップオーバー**］をクリックします。

   スクリプトの最初の行が評価されます。
4. ［**ステップオーバー**］をもう一度クリックします。

   現在の式が評価され、デバッガは次の行に移動します。この例の式は数行にわたり、変数に式が割り当てられています。
5. ［**ステップオーバー**］をもう一度クリックします。

   この式は数行にわたり、変数に関数が割り当てられています。

   30行目は、先に作成されていた式を呼び出します。
6. ［**ステップオーバー**］をクリックします。

   デバッガは式の中にステップインし、1行ずつ実行します。
7. 式が終了するまで［**ステップオーバー**］をクリックし続けます。

   デバッガは、式の呼び出しに続く行に戻ります。

   31行目は先に定義されていた関数を呼び出します。
8. ［**ステップオーバー**］をクリックし、ステップインせずに関数を実行します。デバッガは関数全体を実行した後、関数の呼び出しに続く行に戻ります。

   33行目では別のスクリプトをインクルードしています。
9. ［**ステップイン**］をクリックします。

   デバッガはインクルードされたスクリプトを別のタブで開いて待機します。
10. ［**ステップオーバー**］をクリックします。

    インクルードされたスクリプトの次の行が実行されます。
11. ［**ステップアウト**］をクリックします。

    デバッガはインクルードされたスクリプトの残りを実行した後、`Include()`関数の次の行に戻ります。

異なるスコープおよび名前空間内の変数のウォッチ

デバッガウィンドウの下にあるタブでは、変数が作成され、変更される様子をウォッチできます。この例では、複数のスコープと1つの名前空間内の変数について見ていきます。

1. 「scoping.jsl」サンプルスクリプトを開き、［**スクリプトのデバッグ**］ボタンをクリックします。
2. ［**ステップオーバー**］をクリックします。

   1行目（行番号4）は`Names Default To Here`をオフにします。このスクリプトを同じJMPセッションの中でもう一度実行した場合、この行はスコープをリセットし、最初に作成された変数をグローバルスコープに置きます。

3. ［**ステップオーバー**］をクリックします。

   *x*という名前のグローバル変数が作成されます。［グローバル］タブのリストに*x*が追加され、値が5、タイプが数値であることが示されます。

4. ［**ローカル**］タブを選択し、スコープのリストから［**Global**］を選択します。

   グローバル変数の*x*がここにも表示されます。

5. ［**ステップオーバー**］を2回クリックします。

   `Names Default To Here`がオンになり、スクリプトの残りがHereスコープ内に置かれます。その後、そのスコープ内に新しい変数*x*が作成されます。

   グローバル変数*x*の値は変わらないことに注意してください。

6. ［ローカル］タブのリストから［**Here**］を選択します。

   ［**Here**］の下に、ローカル変数*x*がその値やタイプとともにリストされます。

7. ［**ステップオーバー**］をクリックします。

   `Local Here`スコープが作成されます。［ローカル］のリストに2つ目のHereスコープが表示されます。

8. ［**ステップオーバー**］をクリックします。

   このHereスコープ内に新しい変数*x*が作成されます。［ローカル］タブにリストされた3つのスコープ（**Here**、**Here**、**Global**）のそれぞれを選択して、3つの異なる*x*変数を確認します。

9. ［**ステップオーバー**］をクリックします。

   デバッガのログで出力を確認します。*here:x*スコープはローカルのhereを示し、スクリプトウィンドウのhereを示すものではありません。

10. ［**ステップオーバー**］をクリックします。

    スクリプトは、ログに空の1行を挿入した後、Local Hereスコープを終了します。2つ目のHereが、その*x*変数とともに［ローカル］リストから消えます。

11. ［**ステップオーバー**］をクリックします。

    “test” という名の名前空間が、*x*という別の変数とともに作成されます。それを確認するには、［**名前空間**］タブを選択します。

12. ［**ステップオーバー**］をクリックし、ログを確認します。

13. ［**ステップオーバー**］をクリックし、デバッガを終了します。

インタラクティブなスクリプトでのデバッガを使用

スクリプトがインタラクティブな要素を作成する場合、ユーザによる操作ができるようにJMPの主インスタンスに制御が戻されます。ユーザが操作を終了したら、制御はまたデバッガに戻ります。

1. 「interactive.jsl」サンプルスクリプトを開き、［**スクリプトのデバッグ**］ボタンをクリックします。
2. ［**ステップオーバー**］を2回クリックします。

   `New Window`式が評価された後、モーダルウィンドウが開いて入力を待ちます。デバッガを移動しないと新しいモーダルウィンドウが見えないことがあります。

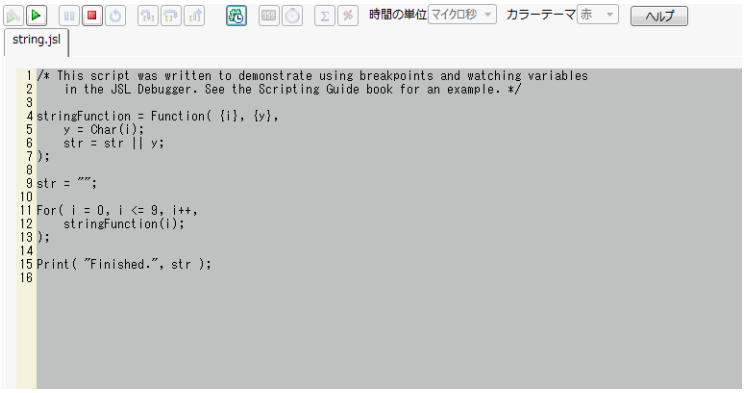
3. 「Assign X and Y」ウィンドウに2つの数値を入力し、[**OK**]をクリックします。
   制御がデバッガに戻ります。
4. [**ステップオーバー**]を3回クリックし、デバッガのログを確認します。
   ログには先ほど入力した2つの数値が表示されます。
5. [**ステップイン**]をクリックし、デバッガを終了します。

JSL プロファイルの使用

JSL プロファイルを使うと、特定の行の実行にかかった時間や、特定の行が実行された回数などがわかります。

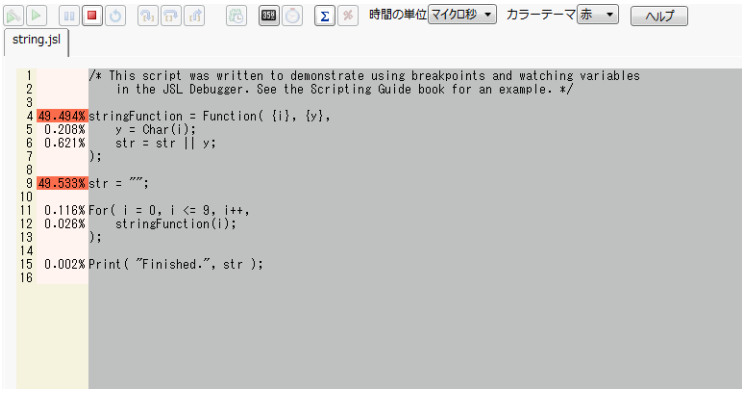
1. 「string.jsl」サンプルスクリプトを開きます。
2. [**スクリプトのデバッグ**]ボタン をクリックします。
3. [**JSL スクリプトのプロファイル**]ボタン をクリックします。

図 4.15 開始時の JSL プロファイルウィンドウ



4. [**実行**]ボタン をクリックしてプロファイルを開始します。
   プロファイルは、あるステートメントが実行された回数、および実行にかかった時間の情報を収集します。実行時間は累積値で、JSL ステートメントが実行されるたびに記録されます。

図4.16 スクリプトのプロファイル



左側の余白に、指定した統計量が表示されます。デフォルトでは、処理時間のパーセントが表示されます。代わりにステートメントの実行回数のパーセントを表示するには、［**総数でプロファイルを表示**］ボタン Σ をクリックします。左側の余白に表示される値は、色分けされているため、パフォーマンスの問題を引き起こしている可能性のある部分が一目で特定できます。

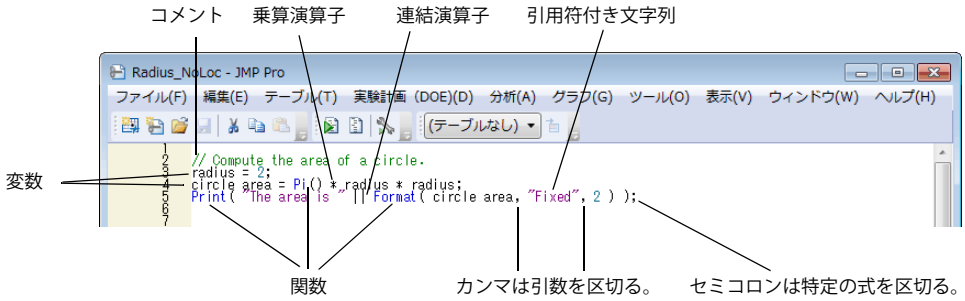
第5章

JSLの構成要素

JSLの基礎を学ぶ

初心者であれ上級ユーザであれ、JSLの構文と基礎を学ぶことは重要です。概念の一部（ループや変数など）は他のスクリプト言語と共通ですが、記述ルールはJSLに独特なものです。

図5.1 JSLスクリプトの例



この章では、構文規則からファイルパス規則、条件および名前空間まで、JSLの基本概念について説明します。

JSLの構文規則

スクリプト言語やプログラミング言語は、独自の構文規則を持っています。CやJavaといった言語でのプログラミングの経験がある人なら、JSLを見慣れた言語と感じるでしょう。ただし、JSLでは、記述ルールやスペースに関する規則が異なります。

以下の節で、基本的なJSL構文規則を説明します。

値の区切り文字

JSLでは、ワードを括弧やカンマ、セミコロン、スペース、各種演算子（＋、－など）で区切ります。この節では、区切り文字を使用するときの規則を説明します。

カンマ

カンマは、リスト内の項目、行列内の行、関数の引数といったアイテムを区切ります。

```
my list = {1, 2, 3};
your list = List( 4, 5, 6) ;
my matrix = [3 2 1, 0 -1 -2];
If( Y < 20, X = Y );
Table Box( String Col Box( "年齢", a ) );
```

メモ: 複数のコマンドをつないで関数内の1つの引数とするには、各コマンドをセミコロンで区切ります。詳細は、「セミコロン」（81ページ）を参照してください。

括弧

括弧は、JSL内でいくつかの目的を持ちます。

- 括弧は式の中の演算をグループ化します。次の括弧は、If()式内の演算をグループ化しています。

```
y = 10;
If(
   Y < 20, X = Y,
   X = 20
);
```

- 関数の引数を括弧で囲みます。次の例では、Open()関数の引数を括弧が囲んでいます。

```
Open( "$SAMPLE_DATA/Big Class.jmp" );
```

- 括弧は、引数が不要な場合でも、関数名の最後を示します。たとえば、Pi 関数は引数を持ちません。しかし、JMP が *Pi* を関数と識別するには括弧が必要です。

```
Pi();
```

メモ: 括弧は必ず対で使用してください。どの (にも) が必要です。対になっていない場合はエラーとなります。

スクリプトエディタでは、括弧（丸括弧、角括弧、大括弧）の対を確認することができます。スクリプト内の任意の場所にカーソルを置き、Ctrl キーを押しながら] を押します（Macintosh の場合は command キーを押しながら b）。スクリプトエディタが括弧を検索し、最初の開き括弧と閉じ括弧の間のテキストを強調表示します。次に上位の括弧を強調表示するには、このプロセスを繰り返します。例については、「スクリプト作成のツール」章の「括弧の自動マッチ」（56ページ）を参照してください。

セミコロン

セミコロンで区切った式は連続して評価され、最後の式の結果が戻されます。次のコードでは、まず変数 i に 0 が割り当てられ、次に変数 j に 2 が割り当てられます。

```
i = 0;
j = 2;
```

セミコロンは、次の If() 式のように、カンマで区切られた引数をつなぐこともできます。

```
If( x < 5, y = 3; z++; );
```

他の言語では、式の終わりを表す文字としてセミコロンを使用します。JSL のセミコロンは、コマンドが続く可能性を示すものとして機能します。セミコロンで式を区切る方法については、「式を結合する他の方法」（98ページ）を参照してください。

スクリプトの最後、または一連の引数の最後にセミコロンをつけても問題ないので、セミコロンを終わりの文字と考えることもできます。一連のスクリプトの最後や、閉じ丸括弧または閉じ中括弧の後にセミコロンを付けることができます。実際、最後にセミコロンを付ける習慣をつけておくと、短いスクリプトを大きなスクリプトに貼り付ける際に間違いを避けることができます。

セミコロンと同じ機能を持つ関数は Glue() です。セミコロンと Glue() の詳細については、「演算子」（85ページ）を参照してください。

二重引用符

二重引用符はテキスト文字列を囲みます。スペースや大文字／小文字を含め、二重引用符内のすべてが入力したとおりに受け取られ、評価はされません。"Pi() ^ 2" を二重引用符で囲んだ場合、それは単なる文字の羅列であり、10 に近い値を指すわけではないのです。

テキスト文字列の使用には十分な注意が必要です。二重引用符内のテキスト文字列は、ユーザの入力したとおり使用されるため、スペースや句読点も出力に影響します。

引用符付き文字列の中で二重引用符を使用する場合は、各二重引用符の前にエスケープシーケンス\!（バックスラッシュと感嘆符）を挿入します。たとえば、次のスクリプトを実行してウィンドウのタイトルを見てみましょう。

```
New Window( "\!" こんにちは \!" は引用符付き文字列です ",
   Text Box( Char( Repeat( "*", 70 ) ) )
);
```

表5.1 引用符付き文字列のエスケープシーケンス

\!b	スペース
\!t	タブ
\!r	キャリッジリターンのみ
\!n	改行のみ
\!N	ホスト環境に適した改行文字を挿入[a]
\!f	改ページ（ページ区切り）
\!0	ヌル文字 **メモ：**ヌル文字は通常、文字列の終了とみなされるので、使用するのは危険です。ローマ字の O ではなく、数値のゼロをタイプするよう注意してください。
\!\	円記号
\!"	二重引用符

a. Macintoshでは、このエスケープシーケンスはCR（キャリッジリターン、16進数‘0D’）です。Windowsでは、このエスケープシーケンスはCR LF（キャリッジリターンとその後の改行、16進数‘0D0A’）です。

あるまとまった箇所で、エスケープ文字を何回も指定しなければならない場合があります。そのような場合は、\[...]\を使用すれば、括弧内でエスケープシーケンスが不要（使用不可能）になります。次に、二重引用符文字列の中で\[...]\を使用している例を示します。

```
jslPhrase = " これを JSL で実行するには、次のように記述します。\[
   a = "hello";
   b = a|| " world.";
   Show(b);
   ]\ そして、スクリプトを実行します。";
```

スペース

JSLでは、名前の中に空白文字を挿入することが許可されています。また、JSLワード内またはJSLワード間のスペース、タブ、改行、および空白行は無視されます。これは、ほとんどのJSLワードがユーザインターフェースから採用されており、これらのコマンドのほとんどにスペースが含まれているからです。たとえば、「モデルのあてはめ」プラットフォームのJSL式は、`Fit Model()`または`FitModel()`のどちらでも動作します。

演算子内のスペース、および1つの数値内の桁間のスペースは許可されません。そのようなケースではエラーが生じます。たとえば、`i++`にある2つの`+`の間にスペースを入れたり（`i+ +`）、数値の途中にスペースを挿入したりすることはできません（`4 3`は`43`と同じではありません）。

メモ: JSLで名前の中に空白文字が使えるのはなぜでしょうか。理由のひとつは、JSLのコマンドやオプションにはJMPメニューやウィンドウにあるコマンドがそのまま使われていることです。また、データテーブルの列名によくスペースが使われることももうひとつの理由です。

数

数は整数、小数、日付、時間、日付時間値として記述できます。また、`E`の後ろに10のべきを付けた指数表記も使用できます。たとえば、次に挙げるものはすべて数です。

```
.   1   12   1.234  3E3  0.314159265E+1 1E-20 01JAN98
```

メモ: ピリオド1つだけが入力されているときは、欠測数値（または、**NAN**: not a number）です。

日付、時間、日付時間値について詳しくは、「データタイプ」章の「日付時間の関数と形式」（123ページ）を参照してください。数値に通貨記号を付けて表示させる方法については、「データタイプ」章の「通貨」（134ページ）を参照してください。

名前

名前は、ずばり、ものを呼ぶためのものです。たとえば、変数に3という数値を割り当てる式`a=3`では、「a」が名前です。

コマンドと関数にも名前があります。式`Log( 4 )`では、「Log」が対数関数の名前です。

名前には、次のようなルールがあります。

- 名前は**必ず**英文字または下線で始まり、その後ろに次の文字を使うことができます。
  - 英文字（a-z A-Z）
  - 数字（0-9）
  - 空白文字（スペース、タブ、改行、改ページ）
  - 数学記号（⊆など）

  - 特殊文字（アポストロフィ（'）、パーセント記号（%）、ピリオド（.）、バックスラッシュ（\）、および下線（_））
- 名前を比較するとき、空白文字（スペース、タブ、改行など）は無視され、大文字と小文字も区別されません。たとえば、Forageとfor ageは、空白文字と大文字／小文字の違いに関わらず、同じ名前として扱われます。

実際には、どんな文字列でも名前として入力できますが、前述のルールに従っていないものは、引用符で囲み、Name()という特殊な命令の中に入れなければなりません。たとえば、taxable income(2011)という名前のグローバル変数を指定するときは、この変数をスクリプトで使うたびに、Name()の中に入れなければなりません。

```
Name( "taxable income( 2011 )" ) = 456000;
tax = .25;
Print( tax * Name( "taxable income( 2011 )") );
  114000
```

必要ないときにName()を使っても、問題となることはありません。たとえば、taxとName("tax")はまったく同じものを表します。

メモ: Name()はスクリプトが実行されるときに呼び出される関数ではありません。これは、引数に指定したものが名前であることを表す構文マーカーです。式の中で列の名前を指定するには、Column()を使用します。

JMPによる名前の解釈については、「名前解決のルール」（91ページ）を参照してください。

コメント

コメントは、ユーザがコードに追加するメモで、JSLプロセッサ（**構文解析プログラム**）では無視されます。コメントは、その部分のスクリプトが何をしているかなどを説明するために追加します。また、スクリプトの一部を一時的に無効にする場合にも便利です。たとえば、エラーを生じさせている可能性があるコードの前後にコメント記号を挿入し、スクリプトを再実行します。

コメントにしたいコードの前後にコメント記号を入力します。次の例では、コードの途中で/* */に囲まれてコメントになっているところがあります。JMPは次の2つのスクリプトをまったく同じものとして扱います。

```
tax /*percentage*/ = .25;
tax = .25;
```

表5.2に、コメント記号の種類を示します。

表5.2 コメント記号

記号	構文	説明
//	// comment	コメントの開始を示す。行頭に置く必要はありません。この記号から行末までがすべてコメントとなります。

表5.2 コメント記号（続き）

記号	構文	説明
/* */	/* comment */	コメントの開始と終了を示す。行の途中にも挿入でき、コメントの前後のスクリプトには影響が及びません。
//!	//!	Add//! をスクリプトの最初の行に追加すると、JMP を開いたときにスクリプトが自動的に実行される。（スクリプトエディタは開かない。）

演算子

演算子は一般的な数値演算を行う1文字または2文字の記号です。演算子にはさまざまな種類があります。

- **二項演算子**（3 + 4の+、a = 7の=のように、両側にオペランドをとるもの）
- **接頭演算子**（論理否定の!aのように、右側にオペランドを1つとるもの）
- **接尾演算子**（aをインクリメントするa++のように左側にオペランドを1つとるもの）

JSLのすべての演算子に、同じ機能を持つ関数があります。

式を簡単に記述できるよう、JSLでは特定の文字演算子（四則演算の記号など）を使用できます。これらの演算子は、関数として記述した場合と同じ意味を持ちます。たとえば、次の2つのステートメントは等価です。

```
Net Income After Taxes = Net Income - Taxes;
Assign( Net Income After Taxes, Subtract( Net Income, Taxes ) );
```

割り当て演算を書くときは、Assign()関数か、二項演算子の=を使います。同様に、引き算を行うには、Subtract()関数またはマイナス記号を使います。どちらも、JMP内部では同じものとして扱われます。

メモ：通常、JSLでは空白が無視されるので、“netincomeaftertaxes” と “net income after taxes” は同じものとみなされます。ただし、2文字の演算子の文字間にスペースを**入れてはなりません**。次の演算子は、必ずスペースなしで入力してください。

||、|/、<=、>=、!=、==、+=、-=、*=、/=、++、--、<<、::、:*、:/、/*、*/

他に、一般的な演算子としてセミコロン（;）があります。セミコロンは次のような目的で使用します。

- プログラミングシーケンスで、複数の式を区切り、同時につなげる。セミコロンは最後の式の結果を戻します。つまり、a;bはGlue(a,b)と等価です。
- 式の最後。セミコロンを式の最後に付けることはできますが、他の言語のようにステートメントの終わりを表すものとして機能するわけではありません。

式には複数の演算子を含めることができます。その場合、演算子は優先度順にグループ化されていきます。たとえば、*は+より優先されます。

```
a + b * c
```

まずb * cの掛け算が行われた後、その結果がaに加算されます。

+は-より優先されます。

```
a + b * c - d
```

まずb * cの掛け算が行われた後、その結果がaに加算されます。その後、a + b * cの結果からdが引かれます。

表5.3は、演算子と、対応する関数をまとめたものです。演算子は優先順位の高い順にリストしてあります。

表5.3 演算子、および対応する関数（優先順位の高い順）

演算子		関数構文	説明
{ }	List	{*a*,*b*} List(*a*,*b*)	リストを作成する。
[ ]	Subscript	*a*[*b*,*c*] Subscript(*a*,*b*,*c*)	データ要素*a*の中にある特定の要素を指定する。*a*は、リスト、行列、データ列、プラットフォームのオブジェクト、ディスプレイボックスのどれでもかまいません。
++	Post Increment	*a*++ Post Increment(*a*)	*a*に1を加えた値を*a*に格納する。 JMPには、プリインクリメント演算子がありません。代わりに、Add To()演算子(+=)を使用します。
--	Post Decrement	*a*-- Post Decrement(*a*)	*a*から1を引いた値を*a*に格納する。 JMPには、プリデクリメント演算子がありません。代わりに、Subtract To()演算子(-=)を使用します。
^	Power	*a*^*b* Power(*a*,*b*) Power(*x*)	*a*を*b*乗する。引数を1つだけ指定したときは、2乗と解釈されます。たとえば、Power(*x*)と指定した場合は、x^2が計算されます。
-	Minus	-*a* Minus(*a*)	*a*の符号を反転する。
!	Not	!*a* Not(*a*)	論理否定。0（偽）を0でない値（または真）にする。1（真）を0の値（偽）にする。
*	Multiply	*a**b* Multiply(*a*,*b*)	*a*に*b*を掛ける。
:*	EMult	*a*:**b* EMult(*a*,*b*)	行列*a*と*b*を要素ごとに掛ける。（行列*a*の各要素に行列*b*の各要素を掛ける。）

表5.3 演算子、および対応する関数（優先順位の高い順）（**続き**）

演算子		関数構文	説明
/	Divide	*a*/*b* Divide(*a*,*b*) Divide(*x*)	Divide(*a*, *b*) は *a* を *b* で割る。 Divide(*x*) と指定した場合には、分母がx、分子が1と解釈され、1/*x* が求まります。
:/	EDiv	*a*:/*b* EDiv(*a*,*b*)	行列 *a* を *b* で要素ごとに割る。（行列 *a* の各要素を行列 *b* の各要素で割る。）
+	Add	*a*+*b* Add(*a*,*b*)	*a* と *b* を足す。
–	Subtract	*a*–*b* Subtract(*a*,*b*)	*a* から *b* を引く。
||	Concat	*a*||*b* Concat(*a*,*b*)	2つ以上の文字列または2つ以上のリストを接合する。行列を横方向に連結する。詳細については、「データ構造」章の「リストの連結」（166ページ）または『スクリプト構文リファレンス』を参照してください。
|/	VConcat	*matrix1*|/*matrix2* VConcat(*matrix1*, *matrix2*)	行列を縦方向に連結する。（行列を横方向に連結する場合は || または Concat() を使用。）
::	Index	*a*::*b* Index(*a*,*b*)	*a* ～ *b* の整数の要素を持つ行列を作成する。 （コロンは、有効範囲を指定する演算子としても使われます。その場合、:a はデータテーブル列 a を、::a はJSLのグローバル変数 *a* を意味します。「スコープ演算子」（93ページ）を参照。）
<<	Send	object << message Send(*object*, *message*)	オブジェクト（*object*）にメッセージ（*message*）を送る。

表5.3 演算子、および対応する関数（優先順位の高い順）（続き）

演算子		関数構文	説明
==	Equal	*a*==*b* Equal(*a*,*b*)...	比較のためのブール値。これらはすべて、真のときは1、偽のときは0を戻します。*a*または*b*に欠測値があると、欠測値が戻され、真も偽も評価されません。欠測値の扱いについては、「欠測値」（111ページ）を参照。
!=	Not Equal	*a*!=*b* Not Equal(*a*,*b*)...	
<	Less	*a*<*b* Less(*a*,*b*)...	
<=	Less or Equal	*a*<=*b* Less or Equal(*a*,*b*)	
>	Greater	*a*>*b* Greater(*a*,*b*)	
>=	Greater or Equal	*a*>=*b* Greater or Equal(*a*,*b*)	
<=, <	Less Equal Less	*a*<=*b*<*c* Less Equal Less(*a*,*b*,*c*)	範囲チェック。真のときは1、偽のときは0を戻します。*a*か*b*のどちらかが欠測値のときは、欠測値になります。
<, <=	Less Less Equal	*a*<*b*<=*c* Less Less Equal(*a*,*b*,*c*)	
&	And	*a*&*b* And(*a*,*b*)	論理積。両方が真のとき、真を戻します。左の値が偽の場合、右の値は評価されません。欠測値の扱いについては、「欠測値」（111ページ）を参照。
|	Or	*a*|*b* Or(*a*,*b*)	論理和。どちらかまたは両方が真のとき、真を戻します。欠測値の扱いについては、「欠測値」（111ページ）を参照。
=	Assign	*a*=*b* Assign(*a*,*b*)	*b*の値を*a*に代入する。*a*の現在の値が置き換わります。
+=	Add To	*a*+=*b* AddTo(*a*,*b*)	*b*にaを足し、結果を*a*に代入する。
-=	Subtract To	*a*-=*b* SubtractTo(*a*,*b*)	*a*から*b*を引き、結果を*a*に代入する。
*=	Multiply To	*a**=*b* MultiplyTo(*a*,*b*)	*a*に*b*を掛け、結果を*a*に代入する。
/=	Divide To	*a*/=*b* DivideTo(*a*,*b*)	*a*を*b*で割って、結果を*a*に代入する。
;	Glue	*a*;*b* Glue(*expr*, *expr*, ...)	*a*、*b*の順に実行する。

グローバル変数とローカル変数

変数は、後にスクリプト内で参照する値を保持した名前です。変数には2種類あります。

- **グローバル変数**は、JMPセッション内のスクリプトすべてで共有されます。
- **ローカル変数**は、それを定義したスクリプトのコンテキストだけに適用されます。特定の関数に適用される変数や、スクリプトの一部だけに適用される変数もあります。

変数の使用範囲を制限するには、**名前空間**内で変数を定義します。名前空間とは、変数、関数、その他の一意の名前の集まりです。JMPには、すべてのスクリプトでデフォルトで使用されるグローバル変数の名前空間が1つあります。名前を限定的な構文なしでそのまま使用すると、**範囲指定のない**変数となり、グローバル名前空間に入ります。

```
x = 1;
```

ローカル名前空間

変数をグローバル名前空間に入れると、競合が生じることがあります。2つのスクリプトで同じ名前の変数を使用した場合、最初のスクリプトの変数の値が、あとで実行されたスクリプトにより変更されてしまいます。

この問題を回避するには、各スクリプトの冒頭に次のような行を挿入します。

```
Names Default To Here( 1 );
```

`Names Default To Here( 1 );`関数は、スクリプトの中にある範囲指定されていない変数すべてがそのスクリプトのローカル変数であることを宣言し、グローバル変数名前空間に影響を与えないようにします。「高度な適用範囲指定と名前空間」（230ページ）に詳しい説明があります。

メモ: カスタムメニューおよびツールバーボタンのスクリプトについて、`Names Default to Here`オプションは、デフォルトで真になっています。カスタムメニュー項目を選択するか、またはカスタムツールバーボタンをクリックした際に実行されるスクリプトは、グローバル変数に影響を与えません。

名前付き名前空間

変数を特定の名前空間内に作成することもできます。次の例は、`aa`という名前空間の中に変数`x`を作成します。

```
aa:x = 1;
```

ローカル変数の前に`Local()`関数を置くという方法もあります。次の例では、`a`も`b`もローカル変数です。

```
Local( {a = 1, b}, ... );
```

スコープ演算子も、グローバル変数とローカル変数を区別します。詳細は、「名前解決のルール」（91ページ）を参照してください。

以下の節で、変数の管理に役立つ関数を紹介します。

Show Symbols、Clear Symbols、Delete Symbols

Show Symbols() 関数は、グローバルとローカルの両方で定義されている変数と名前空間を、現在値とともにリストします。以下は、ログに表示されたShow Symbols() メッセージの結果の例です。

```
Show Symbols();
  // Here
  a = 5;
  b = 6;
  // 2 Here

  // Global
  c = 10;
  // 1 Global
```

ヒント： JSL デバッガでも、変数と名前空間の値を表示することができます。詳細については、「スクリプト作成のツール」章の「スクリプトのデバッグ／プロファイル」（62 ページ）を参照してください。

Clear Symbols() 関数は、グローバルとローカルの両方で定義されている変数の設定値をクリアします。たとえば、Clear Symbols の後に Show Symbols を使うと、変数が空になっていることがわかります。

```
Clear Symbols();
Show Symbols();
  // Here
  a = Empty;
  b = Empty;
  // 2 Here

  // Global
  c = Empty;
  // 1 Global
```

メモ： 以前のバージョンで使用されていたShow Globals() 関数とClear Globals() 関数は、新しいShow Symbols() 関数とClear Symbols() 関数の別名です。

すべてのグローバル変数および名前空間を削除するには、Delete Symbols() 関数を使用します。次のスクリプトの最後にあるShow Symbols() が実行されても、ログには何も表示されません。すべての変数がメモリから完全に削除されるためです。

```
Delete Symbols();
Show Symbols();
```

すべての名前空間内の変数をリストするには、Show Namespaces() を使用します。特定の名前空間のみを削除するには、ns << Deleteを使用します。Clear Symbols() とDelete Symbols() は、名前空間への参照を含む変数をクリアまたは削除しますが、各名前空間内の変数をクリアまたは削除するわけではありません。範囲指定のない変数については、「名前解決のルール」（91 ページ）を参照してください。

メモ: Clear Symbols()とDelete Symbols()は、現在使用されているすべてのスクリプトを中断します。これらの関数は、プログラミング環境やデバッグ環境に非常に便利ですが、配布予定のスクリプトには含めないようにしてください。スクリプトにNames Default To Here(1)を含めた場合、グローバルシンボルのクリアや削除は必要ありません。

シンボルのロックおよびロック解除

変数が変更されるのを回避するためにロックしたい場合、Lock Symbols()関数を使用します。(Lock Globals()は別名です。)

```
Lock Symbols( name1, name2, ... );
```

ロックを解除してグローバル変数の変更を可能にするには、Unlock Symbols()関数を使用します。(Unlock Globals()は別名です。)

```
Unlock Symbols( name1, name2, ... );
```

この2つのコマンドの主な用途は、変数が誤って変更されるのを防ぐことです。たとえば、別のスクリプトで使用している変数がClear Symbols()でクリアされると困る場合に、その変数をロックしておきます。

メモ: Lock Symbols()を、名前空間をロックする目的で使用することはできません。ns << Lockを使用してください。

グローバル変数を隠す

グローバル変数を隠す(つまり保護する)には、名前の前に2つの下線(__)を付けます。グローバル変数を保護するということは、グローバル変数を非表示にすることで、その変数を調べたり表示したりすることができなくなることを意味します。ただし、その動作は内容によって異なります。

次の例では、変数が保護されているため、その変数は出力されません。

```
Show(::__xyz);
```

次の例では、保護されている変数の場合NULL値が戻されます。

```
::a = Name Expr( ::__xyz );
```

次の例では、保護されている変数の場合でも値が戻されます。

```
Show( Eval( ::__xyz ) );
```

名前解決のルール

次の種類のオブジェクトは、名前で識別できます。

- データテーブル内の列とテーブル変数

- セッションの間、値を保持するグローバル変数
- スクリプトで記述することが可能なオブジェクトタイプ
- 計算式内の引数とローカル変数

オブジェクトを参照するには、ほとんどの場合、オブジェクトの名前を直接使うことができます。次の例を見てみましょう。

```
ratio = :Name(" 身長 ( インチ )") / :Name(" 体重 ( ポンド )");
```

スクリプトの複雑さにもよりますが、「ratio」が変数で、「身長 (インチ)」と「体重 (ポンド)」がデータテーブルの列名だというのはすぐわかりますね。では、意味がはっきりしない場合はどうでしょうか。「ratio」をグローバル変数や列名として使用しているスクリプトもあるかもしれません。

引数として使用される変数名

メッセージの引数は原則的に評価されます。旧バージョンのJMPでは、一部のメッセージで変数名が文字列値として解釈されていました。次のスクリプトは、グラフの軸にメッセージを送っています。現在のバージョンのJMPでは軸が線形スケールとなり、JMP 11では対数スケールとなります。

```
Log = "Linear";
axis box << Scale( Log );
```

Scale("Log") の構文は、リテラル文字列を設定するものと規定されています。曖昧なケースを残さないために、環境設定の［一般］グループにある［JSLで引用符なしのストリングを許可する］を、［許可する (警告あり)］または［許可しない］のいずれかに設定することをお勧めします。

名前を解決するためのルール

JMPは**名前の解決**を使ってオブジェクト名を解釈します。適用範囲が明示されていない名前には、次の規則が順に適用されます。

1. 名前がオブジェクトに送るスクリプトの一部である場合、それは通常、オブジェクト内のオプションまたはメソッドの名前です。たとえば、Show Points()はBivariateオブジェクト内のオプションの1つです。

   ```
   dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
   dt << Bivariate( y( :Name(" 体重 ( ポンド )") ), x( :Name(" 身長 ( インチ )") ) );
   obj << Show Points( 1 );
   ```

2. 接頭部に : スコープ演算子が付いている名前の場合、GlobalやHereなどの名前空間の中を検索します。
3. 後ろに1組の丸括弧 () がついている名前の場合、（ユーザ定義の関数ではなく）ビルトイン関数として検索します。
4. 接頭部に : スコープ演算子が付いている名前の場合、データテーブル列またはテーブル変数として検索します。
5. 接頭部に :: スコープ演算子が付いている名前の場合、グローバル変数として検索します。
6. ローカル変数として検索します。
7. プラットフォーム起動名（DistributionやBivariateなど）として検索します。

8. 割り当て式の左辺（L-value）として使用されている名前で、かつ、スクリプトの冒頭にNames Default To Here(0)がある場合は、グローバル変数を作成して使います。

例外

- 一部の名前は、データテーブル、データ列、プラットフォームなどのオブジェクトを参照する変数であり、値の取得や設定には使用されません。これらの名前は解決されず、渡されます（文字どおり解釈される）。
- 関数の定義、列計算式、および「非線形回帰」プラットフォームの計算式において、スコープは列内のすべての行で同じです。
- 名前が、閉じているデータテーブル内の列を直接参照している場合、その名前はテーブルが再度開いたときに該当の列に対して解決されます。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
Show( :Name("体重（ポンド）") << Get As Matrix );
   // "体重（ポンド）"は列名として解決される
Close( dt, NoSave );
Show( :Name("体重（ポンド）") << Get As Matrix );
   // "体重（ポンド）"は解決されない
/* データテーブルを再度開く */
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
Show( :Name("体重（ポンド）") << Get As Matrix );
   // "体重（ポンド）"は列名として解決される
```

ただし、次の例では、変数をデータテーブルの2番目のインスタンスとして解決しません。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
col = Column( dt, 1 ); // colはColumn( "体重（ポンド）" );
Close( dt, NoSave );
/* データテーブルを再度開く */
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
Show( col << Get As Matrix ); // 最初のデータテーブルへの参照はもう存在しない
```

以下の節で、データテーブルの列の名前がどのように解決されるかを説明します。名前の解決について詳しくは、「プログラミング手法」章の「高度な適用範囲指定と名前空間」（230ページ）を参照してください。

スコープ演算子

スコープ演算子は、名前が曖昧な場合（たとえば、変数と列名の両方を指している場合）の解決の助けとなります。

次の例では、接頭部の二重コロン（::）により、zがグローバル変数であることを示しています。接頭部の一重コロン（:）は、xとyが列名であることを示します。

```
::z = :x + :y;
```

ヒント： Names Default to Here(1) 関数も名前の解決に影響を与えます。詳細は、「プログラミング手法」章の「Names Default To Here とグローバル関数」（231 ページ）の節を参照してください。

スコープ演算子と同様に使用できるJSL関数が2つあります。表5.4は、その関数と構文を示したものです。

表5.4 スコープ演算子

演算子と等価の関数	関数構文	説明
: As Column	:name dt:name As Column(*dt*, *name*)	*name*をデータテーブル列として評価させる。オプションのデータテーブル参照引数*dt*は、現在のデータテーブルを設定します。例については、「適用範囲が指定された列名」（94ページ）を参照。
:: As Global	::name As Global(name)	*name*をグローバル変数として評価させる。 **メモ：**二重コロンは、範囲を表す二項演算子としても使われることに注意。

適用範囲が指定された列名

列名の適用範囲を指定すれば、変数名の競合を簡単に回避できます。スコープ演算子を使い、スクリプト内の名前を列への参照として評価させます。

1. 接頭部のコロン（:）は、名前がグローバル変数ではなく、テーブル列またはテーブル変数を参照することを意味します。接頭部のコロンは、現在のデータテーブルの内容を参照します。

   ```
   :年齢;
   ```

2. 二項演算子のコロン（:）は、この考え方を拡張して、データテーブル参照を使って、どのデータテーブルの列なのかを指定します。これは、スクリプト内で複数のデータテーブルが参照されている場合に特に重要です。

   次の例では、dt変数が「Big Class.jmp」へのデータテーブル参照を設定します。二項演算子のコロンは、データテーブル参照と「年齢」列を分離します。

   ```
   dt = Data Table( "Big Class.jmp" );
   dt:年齢 // コロンは二項演算子。
   ```

   As Column()の場合も同じ結果となります:

   ```
   dt = Data Table( "Big Class.jmp" );
   As Column( dt, 年齢 );
   ```

したがって、「Big Class.jmp」が開いている場合に限り、次の式も等価です。

```
:年齢;
As Column( dt, 年齢 );
dt:年齢;
```

Column関数は列を特定する目的で使用することもできます。「Big Class.jmp」では、以下の式はすべてテーブル内の2番目の列である**「年齢」**を参照します。

```
Column( "年齢" );
Column( 2 );
Column( dt, 2 );
Column( dt, "年齢" );
```

列名と変数名の競合を回避する

列名と同名の変数があるスクリプトを実行すると、「無効な行番号です。」というエラーが発生します。この問題を防ぐには、列名と変数名が重複しないように注意するか、次のように名前の適用範囲（スコープ）を指定します。

- グローバル変数と列の名前が同じ場合は、グローバル変数の名前が優先します。そのため、列名の方の適用範囲（スコープ）を指定する必要があります。

```
:: 年齢 = [];
年齢 = : 年齢 << Get As Matrix;
```

- グローバル変数と列名の混同を避けるには、両方の変数の適用範囲を指定してください。

```
:: 年齢 = : 年齢 << Get As Matrix;
```

- 複数のデータテーブルが開いている可能性がある場合は、データテーブル参照を変数に割り当てます。適切なテーブルを列の適用範囲として指定します。

```
dt1 = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt2 = Open( "$SAMPLE_DATA/Students.jmp" );
:: 年齢 = dt1: 年齢 << Get As Matrix;
::Name(" 身長 ( インチ )") = dt2: 身長 << Get As Matrix;
```

JMP は、列にある連続したセルすべてについて列計算式を評価します。そのため、通常は列名の適用範囲を指定する必要はありません。ただし、計算式内に割り当てられた変数が列と同名の場合、その列名の適用範囲を指定する必要があります。詳細は、「プログラミング手法」章の「適用範囲が指定された名前」（233 ページ）を参照してください。

適用範囲が指定されていない列名

適用範囲が指定されていない名前に対して値を設定したり、値を取得する場合があります。次のような場合、JMP は名前を（グローバル変数ではなく）データテーブル内の列として解決します。

- その名前をすでに使用しているグローバル変数、ローカル変数、または引数が存在しない
- かつ、コンテキスト内のデータテーブルに同名の列がある
- かつ、次のどちらかである
  - 現在の行の番号が正の値に設定されている
  - 名前に添え字がある（たとえば、`:Name(" 体重 ( ポンド )")[1]` の添え字 `[1]` は、「**体重 (ポンド)**」列の最初の値を選択する）

データテーブルにその名前のテーブル変数がある場合、テーブル変数が優先します。その他のすべてのケースでは、名前はグローバル変数、ローカル変数、または引数に結び付けられます。グローバル変数とローカル変数についての詳細は、「グローバル変数とローカル変数」（89 ページ）を参照してください。

例外

列計算式と非線形回帰の計算式では、列名がグローバル変数よりも優先します。

現在のデータテーブル行の設定

デフォルトでは、現在の行の番号は0で、これは無効な数値です。そのため、次の式はグローバル変数**ratio**に欠測値を割り当てます。

```
ratio = :Name("身長(インチ)") / :Name("体重(ポンド)");
```

Row()関数を使って行番号を指定します。次の例は、行を3に設定します。その行の身長を体重で割り、その結果をグローバル変数**ratio**に割り当てます。

```
Row() = 3;
ratio = :Name("身長(インチ)") / :Name("体重(ポンド)");
```

別の方法としては、添え字を使って行番号を指定する方法があります。次の式は、行3の身長を行4の体重で割ります。

```
ratio = :Name("身長(インチ)")[3] / :Name("体重(ポンド)")[4];
```

スクリプトがデータテーブルの各行に対して実行される場合、行番号の指定は必要ありません。次の例は、「**比率**」列を作成します。各行の身長を体重で割っています。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << New Column( "比率" );
For Each Row( :比率 = :Name("身長(インチ)") / :Name("体重(ポンド)") );
```

JMPは計算式を評価し、列全体にわたって反復して事前評価された統計量を計算します。このような場合も、行番号の指定は必要ありません。(事前計算された統計量は、「データテーブル」章の「事前計算される統計量」(371ページ)で説明するように、データテーブルから計算される単一の数値です。)

変数と列名のトラブルシューティング

As Name()を使用して列名を参照し、Names Default To Here(1)を設定した場合、JMPは変数参照を戻します。そして、その参照は標準の参照ルールを使って処理されます。

次の例では、Here:スコープに「**身長(インチ)**」変数はありません。そのため、JMPはエラーを戻します。

```
Names Default To Here( 1 );
Open( "$SAMPLE_DATA/Big Class.jmp" );
As Name( "身長(インチ)" )[3];
  As Name( "身長(インチ)" )[/*###*/3];
```

この問題を回避するには、次のいずれかの方法を使用します。

- As Name()ではなくAs Column()を使用します。

  ```
  Names Default To Here( 1 );
  Open( "$SAMPLE_DATA/Big Class.jmp" );
  As Column( "身長(インチ)" )[3];
  ```

- As Name()を使って明示的に「**身長(インチ)**」の適用範囲を指定します。

  ```
  Names Default To Here( 1 );
  ```

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt:( As Name( "身長(インチ)" ) )[3];
```

これらのスクリプトは、「Big Class.jmp」の3行目の「**身長(インチ)**」の値である55を戻します。

変数とキーワードのトラブルシューティング

変数と引用符なしのキーワードが同じ名前を持っている場合も、名前解決のエラーが発生する可能性があります。たとえば、`<<Preselect Role()`の1つの引数は「Y」です。スクリプト内でYを変数としても使用している場合は、この引数を引用符で囲みます。

名前解決に関するよくある質問

適用範囲（スコープ）を指定する必要がありますか。

はい。迷った場合は、適用範囲（スコープ）を指定してください。特に、多くの人がさまざまなデータテーブルとともに使う可能性があるスクリプトでは、必ず適用範囲を指定してください。気づかないで、同じ名前を2つのコンテキストで使用している場合（たとえば、データ列と同じ名前のグローバル変数を定義する、など）があるからです。

そのようなスクリプトを書く場合は、適用範囲と名前空間を明確に指定してください。詳細については、「プログラミング手法」章の「高度な適用範囲指定と名前空間」（230ページ）を参照してください。

接頭スコープ演算子は、いったん解決された後は、実行時オーバーヘッドがかかりません。二項スコープ演算子は、常に、実行時オーバーヘッドがかかります。

名前（グローバル変数）によって参照される列と、直接列名を指定する場合の違いは何ですか。グローバル変数によって列を参照する場合、どのようにその列のセルに値を割り当てるのでしょうか。

列を参照すると、列の特定の属性を変更するメッセージや、その値（セルの色付けや計算式の設定など）にアクセスするメッセージを送ることができます。

グローバル変数に列が割り当てられている場合は、添え字を使って列内のセルに値を割り当てます。列名「**身長(インチ)**」が変数xに割り当てられているとします。

```
x = Column( "身長(インチ)" );
```

「**身長(インチ)**」列の3行目に、次のように値を割り当てます。

```
x[3] = 64 // 身長の 3 行目の値を 64 に設定する
```

メモ：JSL スクリプトの現在行は、行を選択することや行にカーソルを置くことで**決定されるのではありません**。現在行は、デフォルトではゼロ（行なし）になっています。Row() を使って（例：Row() = 3）現在行を設定できますが、このような設定はそのスクリプトが実行されている間だけ有効で、スクリプトの実行が終了すると、Row() はデフォルトの値ゼロに戻ります。そのため、一度にスクリプトすべてを実行したときと、一度に数行ずつスクリプトを実行したときとでは、異なる結果になる場合があることに注意してください。

スクリプトの現在の行を設定するもう1つの方法は、For Each Row() で囲む方法です。これは、現在のデータテーブルの行ごとにスクリプトを1回ずつ実行します。例として、「If」（104ページ）を参照してください。データテーブルの操作方法については、「データテーブル」章（271ページ）を参照してください。

スコープ演算子は、ずっとその名前に有効ですか。

はい。いったんスコープ演算子を名前に使うと、以後その名前が出てきた場合、名前は常にスコープ演算子として解決されます。たとえば、スクリプトに age という名前の列と変数があるとします。スクリプトの冒頭で、スコープ演算子 : : を使ってグローバル変数 age を宣言したら、そのスクリプト内で age は常にグローバル変数として解釈されます。「age」列内の値は変数によって影響を受けません。

```
::age = 70;
Open( "$SAMPLE_DATA/Big Class.jmp" );
age = 5; // age はグローバル変数。
Show( age ); // age はまだグローバル変数。
```

スコープ演算子を使う場合、":" と "[]" でどちらの方が優先されますか。

スコープ演算子（:）は、添え字（[]）より前に評価されます。これは、次の2つの行が等価であることを示します。

```
dataTable:colName[i]
(dataTable:colName)[i]
```

式を結合する他の方法

式は、同じ行であっても異なる行であっても、セミコロンを使って分離できます。JMP は、それらの式を順番に評価し、最後の結果を戻します。ここに、まず a を 2 に設定し、続いて b に 3 を設定する式があります。

```
a = 2;
b = 3;
```

セミコロンは2つの式を結合し、最後の式の値を戻します。つまり、x = (a = 2; b = 3) の場合、x の値は3です。

Glue() 関数は最後の式の結果を戻します。この関数を使うと、セミコロンを使ったときと同じ結果になります。次の式は、どちらも3を戻します。

```
Glue( a = 2, b = 3 );
a = 2; b = 3;
```

First() 関数も、各引数を順々に評価しますが、最初の式の結果を戻します。次の式は2を戻します。

```
First( a = 2, b = 3);
```

例

次のスクリプト内の First() は何を戻すでしょうか。

```
x = 1000;
First( x, x = 2000 );
```

First() 関数は x の値（1000）を戻します。その後、x に 2000 が割り当てられます。

反復

JSL では、指定の条件に従ってアクションを繰り返す（**反復**する）関数として For()、While()、Summation()、Product() が用意されています。

メモ: For Each Row() という関数は、データテーブルの行に対してアクションを反復します。例については、「If」（104ページ）を参照してください。テーブルの行にわたる反復については、「データテーブル」章の「データ値にアクセスするその他の方法」（366ページ）にも説明があります。

For

For() 関数は、カンマで区切った4つの引数をとります。最初の3つの引数はループを何回繰り返すかを指定するためのもので、4つ目の引数が繰り返し実行する内容です。

次に、For() の基本的な構文を示します。

```
For( initialization, while, iteration, body );
```

たとえば、次のスクリプトは0～20までの整数を合計しています。

```
s = 0;
For( i = 0, i <= 20, i++, s += i );
```

このスクリプトは次のように機能します。

`s = 0;`	s 変数を0に設定する。後でこの変数に合計が代入されます。
`For(`	For() ループを開始する。
`i = 1,`	ループの制御カウンタの変数（i）を0に設定する。この式は一度だけ実行されます。
`i <= 20,`	ループが開始されるたびに i を20と比較する。i が20以下である限りループの評価を繰り返し、i が20を超えるとループを抜けます。

`i++,`	ループの最後に`i`を1だけ増やす。このステップはループのbody部分（次の行）の評価後に実行されます。
`s += i`	ループのbody部分を評価する。`i`の値を`s`に加えます。bodyを実行した後、`i`の値を増やします（前の行）。
`);`	ループを終了する。

無限ループ

常に真と評価される`For`ループは無限ループを生成し、終わることがありません。スクリプトを停止するには、Windowsでは Escキーを、Macでは commandキーとピリオドキーを同時に押します。または、［**編集**］>［**スクリプトの停止**］を選択します。Macでは、スクリプトの実行中のみ、［**編集**］>［**スクリプトの停止**］が使用可能になります。

JSLのForループとCおよびC++

JSLの`For()`ループは、C（およびC++）の場合と句読法は異なりますが、同様に機能します。

ヒント： Cをご存知のユーザは、セミコロンとカンマの使い方の違いに注意してください。JSLの場合、`For()`は、カンマで引数を区切り、セミコロンで式を結合する関数です。Cの場合、`for`は、セミコロンで引数を区切り、カンマで式を結合する特別な節です。

While

関連する関数に`While()`があります。この関数は、条件を繰り返しテストし、条件が真であればbodyのスクリプトを評価します。構文は次のとおりです。

```
While( condition, body );
```

たとえば、次に示す2つのプログラムは、x (287)以上かつ最小の2のべき乗を見つける`While()`ループを使っています。どちらのプログラムでも結果は512になります。

```
x = 287;

// ループ1:
y = 1;
While( y < x, y *= 2 );
Show( y );

// ループ2:
k = 0;
While( 2 ^ k < x, k++ );
Show( 2 ^ k );
```

このスクリプトは次のように機能します。

x = 287;	xを287に設定する。
// ループ1	
y = 1;	yを1に設定する。
While(	While()ループを開始する。
y < x,	yがxより小さい限りループの評価を繰り返す。
y *= 2	1に2を掛け、結果をyに割り当てる。その後、yが287より大きくなるまで、ループを繰り返します。
);	ループを終了する。
Show(y);	y (512)の値を表示する。
// ループ2	
k = 0;	kを0に設定する。
While(	While()ループを開始する。
2 ^ k < x,	2をkの指数でべき乗し、結果が287より小さい限り評価を繰り返す。
k++	kを1増やす。その後、ループを2^kが287より大きくなるまで繰り返します。
);	ループを終了する。
Show(2 ^ k);	2^kの値（512）を表示する。

For()ループと同様、常に真と評価されるWhile()ループも、終わりのない無限ループを生成します。スクリプトを停止するには、Windowsでは Escキーを、Macでは commandキーとピリオドキーを同時に押します。または、［**編集**］>［**スクリプトの停止**］を選択します。Macでは、スクリプトの実行中のみ、［**編集**］>［**スクリプトの停止**］が使用可能になります。

Summation

Summation()関数は、iの値すべてについてbody部分の式の結果を加算します。構文は次のとおりです。

```
Summation( initialization, limitvalue, body );
```

例:

```
s = Summation( i = 1, 10, i );
```

は、1+2+3+4+5+6+7+8+9+10の結果である55を戻します。

このスクリプトは次のように機能します。

s =	s変数に関数の値を設定する。
Summation(	Summation()ループを開始する。
i = 1,	iを1に設定する。
10,	iの限界を10に設定する。
i	1から10までのiのすべての値を合計し、結果の55を戻す。
);	ループを終了する。

この動作は計算式エディタのΣと同じです。

```
Summation( i = 1, N Row(), x ^ 2 );
```

計算式エディタでこのJSLと等しいものは、次のようになります。

$$\sum_{i=1}^{\text{NRow}} x^2$$

Product

Product()関数は、body部分の式の結果を加算するのではなく乗算することを除けばSummation()と同じように動作します。構文はSummation()と同じです。例:

```
p = Product( i = 1, 5 , i );
```

は、1*2*3*4*5の結果である120を戻します。

この例では、iの開始値が1、上限が5で、iの5までの整数がすべて掛け合わされます。

計算式エディタでこのJSLと等しいものは、次のようになります。

$$\prod_{i=1}^{5} i$$

BreakおよびContinue

Break()とContinue()を使うと、ループをさらに制御することができます。Break()は、ループを直ちに中断し、ループの後に続く式を実行します。Continue()は、Break()を少し柔和にしたもので、現在のループの反復を直ちに中断し、次の反復を続行します。

Break

Break()は通常、条件文の中で使用します。例:

```
For( i = 1, i <= 5, i++,
   If( i == 3, Break() );
   Print( "i=" || Char( i ) );
);
```

の結果は、次のようになります。

```
"i=1"
"i=2"
```

このスクリプトは次のように機能します。

For(	For() ループを開始する。
i = 1,	iを1に設定する。
i <= 5,	iが5以下である限りループの評価を繰り返す。
i++,	iを1だけ増やす。このステップはIFループの評価後に実行されます。
If(	If() ループを開始する。
i == 3, Break()	iが3に等しい場合、ループを中断する。
);	ループを終了する。
Print(	iが3に等しい場合、Print() ループを開く。
"i="	文字列"i="をログに出力する。
||	後続の値を、"i="と同じ行に出力する。
Char(i));	iの値をログに出力する。 その後、iの値が5以下の間はFor() ループを繰り返し、iが3より小さい場合は中断して出力します。
);	ループを終了する。

Print() の後に If() 式と Break() 式が続く場合、"i=3"が出力された後にループが中断するので、1～3のiの値が出力されることになります。

```
"i=1"
"i=2"
"i=3"
```

Continue

Continue() も、Break() と同様に条件式の中で使用されます。例:

```
For( i = 1, i <= 5, i++,
   If( i < 3, Continue() );
   Print( "i=" || Char( i ) );
);
```

の結果は、次のようになります。

```
"i=3"
"i=4"
"i=5"
```

このスクリプトは次のように機能します。

For(	For() ループを開始する。
i = 1,	i を1に設定する。
i <= 5,	1が5以下であると評価する。
i++,	i を1だけ増やす。このステップはIFループの評価後に実行されます。
If(	If() ループを開始する。
i < 3, Continue()	i を1と評価し、i が3より小さい限り続行する。
);	If() を終了する。
Print(	i が3以上になったら、Print() ループを開く。
"i="	文字列 "i=" をログに出力する。
||	後続の値を、"i=" と同じ行に出力する。
Char(i));	i の値をログに出力する。 その後、i の値が5以下の間は For() ループを繰り返し、i が3以上の場合は続行して出力します。
);	ループを終了する。

条件付き関数

JSLには、If()、Match()、Choose()、Interpolate()、Step() という、条件付き処理を行う5の関数があります。

If

If() 関数は、条件（condition）が真（0でない非欠測値）であると評価された場合に、結果（result）のステートメントを戻します。真でない場合には、先に進み、次に条件が真と評価されたときの結果を戻します。

構文は次のとおりです。

```
If ( condition, result1, result2 );
```

たとえば、次のスクリプトは年齢が12より小さい場合、"若い" を戻します。そうでない場合は、"気持ちが若い" を戻します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
For Each Row( 年齢 = If( :年齢 < 12, "若い", "気持ちが若い" ) );
```

複数の条件と結果をつなげることもできます。構文は次のとおりです。

```
If( condition1, result1,
   condition2, result2,
   ...,
   resultElse );
```

この例では、条件1（condition1）が真でない場合、関数は真の条件を見つけるまで評価を続けます。そして、真である条件の結果を戻します。

すべての条件が偽のときは、最後の結果を戻します。値がない場合は欠測値を戻します。そのため、式の最後にデフォルトの結果を含めておくことが大切です。次に、「Big Class.jmp」の性別の略称をコード変更する例を見てみましょう。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
For Each Row( 性別 =
   If(
      性別 == "F", "女性",
      性別 == "M", "男性",
      "不明" );
);
```

このスクリプトは次のように機能します。

`For Each Row( 性別 =`	テーブルの各行で「**性別**」の列の値を変更するよう指定する。
`If(`	`If()` ループを開始する。
`性別 == "F", "女性",`	「**性別**」の値が「F」であれば、その値を「女性」に変更する。
`性別 == "M", "男性",`	「**性別**」の値が「M」であれば、その値を「男性」に変更する。
`"不明" );`	どちらの条件も真でない場合は値を「不明」に変更する。この部分が指定されず、「**性別**」が欠測値になっていた場合は、欠測値を戻します。
`);`	ループを終了する。

結果の式にアクションや割り当てを挿入することもできます。次のスクリプトは、最初の条件（`y < 20`）が偽なので、xに20を割り当てます。

```
y = 25;
z = If( y < 20, x = y, x = 20 );
```

メモ: 等しいことを調べる場合には、=ではなく、==を使います。If関数に *name=value* といった引数が付いている場合は、値を調べるのではなく割り当てます。

Match

Match()関数を使うと、比較対象となる値をすべて手で入力しなくても、等しいかどうかの比較を繰り返すことができます。構文は次のとおりです。

```
Match( x, value1, result1, value2, result2, ..., resultElse );
```

たとえば次のスクリプトは、「Big Class.jmp」の性別の略称をコード変更します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
For Each Row( 性別 =
   Match(
      性別,
      "F", "女性",
      "M", "男性",
      "不明");
   );
```

このスクリプトは次のように機能します。

For Each Row(性別 =	テーブルの各行で**「性別」**の列の値を変更するよう指定する。
Match(	Match()ループを開始する。
性別,	**「性別」**をMatchの引数に指定する。
"F", "女性",	値が「F」であれば、「女性」に置き換える。
"M", "男性",	値が「M」であれば、「男性」に置き換える。
"不明");	FとMのどちらでもない場合は、「不明」に置き換える。
);	ループを終了する。

このMatch()の例は、「If」（104ページ）の例を単純にしたものです。Match()の利点は、比較の変数を条件式ごとに繰り返す必要がなく、一度定義するだけでよいことです。欠点は、Ifとは異なり、演算子のある式が使えないことです。Match()式で性別 == "F"という引数を使うと、エラーになります。

条件と結果のグループが多いほど、Match()が本領を発揮します。次のようなスクリプトにIf()を使ったとすると、もっと多くの行が必要になります。

```
dt = Open( "$SAMPLE_DATA/Travel Costs.jmp" );
For Each Row(
   予約した曜日 = Match( 予約した曜日,
      "Sunday", "SUN",
      "Monday", "MON",
      "Tuesday", "TUE",
      "Wednesday", "WED",
      "Thursday", "THU",
      "Friday", "FRI",
```

```
        "Saturday", "SAT",
        "Not Specified"
    )
);
```

条件と結果のデータタイプに気をつけてください。前述の例では、条件も結果も文字データです。データタイプが一致しないと、列のデータタイプが自動的に変更されてしまうため、予期しない結果になります。

次のスクリプトでは、列のデータタイプが、最初のセルのデータタイプに基づいて数値から文字に変更されます。最初の値「12」は「Twelve」に置き換わり、残りのセルには「Other」が挿入されます。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
For Each Row(
   年齢 = Match( 年齢 , 12, "Twelve", 13, "Thirteen", 14, "Fourteen", 15, "Fifteen",
   16, "Sixteen", "Other" )
);
```

比較する値が1、2、3・・・という整数値の場合には、Choose() を使用するとさらに入力する引数を少なくできます。詳細については、「Choose」（107ページ）を参照してください。

Choose

Choose() 関数は、引数を整数と比較する場合にMatch() よりも短く記述することができます。構文は次のとおりです。

```
Choose( expr, result1, result2, result3, ..., resultElse );
```

データテーブルのgroup列に、1から7までの数値が入っているとします。次のスクリプトは、最初のセルに数字の1があるときにx="低"を戻します。

```
x = ( Choose( group, "低", "中", "高", "不明" ); );
Show( x );
```

このスクリプトは次のように機能します。

x =	x変数を作成する。
Choose(	Choose() ループを開始する。
group,	groupの値を評価する。
"低",	groupの値が1の場合、"低"を戻す。
"中",	groupの値が2の場合、"中"を戻す。
"高",	groupの値が3の場合、"高"を戻す。
"不明"	それ以外の場合は、"不明"を戻す。
)	ループを終了する。

);	x変数を閉じる。
Show(x);	xの値を戻す。

式が範囲外の整数（たとえば、置換用の値が4つしかないのに7が入力されたとき）を評価した場合は、最後の結果が戻されます。前述の例では、"不明"が戻されます。

If()およびMatch()では、Choose関数と同じ結果を得るためにより多くのコードが必要になります。

```
If(
   group == 1, "低",
   group == 2, "中",
   group == 3, "高",
   "不明"
);
Match( group, 1, "低", 2, "中", 3, "高", "不明" );
```

メモ: 式内のデータタイプが一致しないときは、列のデータタイプが自動的に変更されます。

Interpolate

Interpolate()関数は、(x1, y1）と（x2, y2）の2点間で、与えられたx値に対応するy値を戻します。値は線形補間されます。Interpolate()は、データ点間の欠測値の計算に使用できます。

各データ点は、次のように列挙することにより指定できます。

```
Interpolate( x, x1, y1, x2, y2, ... );
```

また、x値とy値が含まれた行列として指定することもできます。

```
Interpolate( x, xmatrix, ymatrix );
```

20～25歳の人の身長を記録したデータセットがあるとします。ただし、23歳の人のデータがありません。23歳の人の身長を推定するには、補間を使用します。次の例は、評価したい値（23歳)、年齢（20～25歳）の行列、そして身長（インチ; 59～75）の行列を示しています。

```
Interpolate( 23, [20 21 22 24 25], [59 62 56 69 75] );
```

この式は、次の値を戻します。

```
62.5
```

62.5という値は、23がx値の22と24の真ん中であるように、y値の56と69の真ん中です。

- 各リストまたは行列内のデータ点は、正の傾きをなすように並べる必要があります。たとえば、Interpolate(2,1,1,3,3)は2を戻しますが、Interpolate(2,3,3,1,1)は欠測値（.）を戻します。
- Interpolateは連続量のデータに最適で、Step()は離散値に適しています。詳細は、「Step」（109ページ）の節を参照してください。

Step

Step() は、2点間の線形補間ではなく、階段関数(ステップ関数)によりxに対応するyを求めることを除けば、Interpolate() 関数と同じです。離散型のy値にはStep() を使用してください（この場合、y値に対応するx値はy_1またはy_2のみとなります）。ただし、y値に対応するx値がy_1とy_2の間である場合は、Interpolate() を使用します。

Interpolateと同様、データ点はリストで指定します。

```
Step( x, x1, y1, x2, y2, ... );
```

また、x値とy値が含まれた行列として指定することもできます。

```
Step( x, xmatrix, ymatrix );
```

たとえば、$25、$50、$75、$100の購入額に対する割引率をまとめたデータテーブルがあるとします。購入額$35のときの割引額を示すグラフを作成したいのですが、データテーブルには記入されていません。次の例は、まず評価したい値である35を示し、次に購入額$25～$100の行列を示しています。

```
Step( 35, [25 50 75 100], [5 10 15 25] );
```

この式は、次の値を戻します。

```
5
```

割引率がスライド制であるなら（この場合は、5と10の間）、Interpolate() を使用します。

```
Interpolate( 35, [25 50 75 100], [5 10 15 25] );
```

この式は、次の値を戻します。

```
7
```

Interpolate() と同じく、各点は正の傾きをなすように並べる必要があります。

不完全または一致しないデータの比較

欠測値を含んだデータを比較する場合は、常に真であるような条件を指定するか、Is Missing() やZero Or Missing() などの関数を使用しない限り、間違った結果が戻される可能性があります。また、（数値と文字など）タイプが異なるデータや行列内のデータを比較する場合も、混乱を招くことがあります。

表5.5は、そのような比較や行列の例と結果を示しています。比較で使用する演算子については、「演算子」（85ページ）を参照してください。比較演算子と論理演算子については、表の後の節で詳しく説明しています。

メモ: ここでの行列は、行数と列数が同じものでなければなりません。

表5.5 特殊なケースの比較テスト（一部）

テスト	結果	説明
m=.; m==1	.	等しいかどうかのテストに欠測値を指定した場合は、欠測値を戻す。
m=.; m!=1	.	等しくないかどうかのテストに欠測値を指定した場合は、欠測値を戻す。
m=.; m<1; m>1; など	.	欠測値を使った比較はどれも欠測値を戻す（真である可能性がある場合は、次を参照）。
m=.; 1<m<0	0	（論理演算子と同様）2つ以上のオペランドを取る比較では、偽は欠測値に優先するため、真である可能性のない欠測値が含まれた比較は偽を戻す。
{a, b}==List(a, b)	1	リストの項目が等しいかどうかのテストは、結果として1つの値を戻す。
{a, b}<{a, c}	.	リストの項目の比較はできない。
1=="abc"	0	等しいかどうかのテストでデータのタイプが異なっている場合は、偽を戻す。
1<="abc"	.	データのタイプが異なっている比較は、欠測値を戻す。
[1 2 3]==[2 2 5]	[0 1 0]	行列が等しいかどうかのテストは、要素ごとの結果を行列で戻す。行列と行列を比較する場合、要素ごとに比較を行い、結果の1と0の行列を戻します。
[1 2 3]==2	[0 1 0]	行列とすべての要素が2である行列が等しいかどうかのテスト。行列を数値と比較する場合、数値はすべての要素がその値である行列として扱われます。
[1 2 3] < [2 2 5]	[1 0 1]	行列の比較は、要素ごとの比較の結果を行列で戻す。
[1 2 3] < 2	[1 0 0]	行列とすべての要素が2である行列との比較。
Is Missing(*m*)	1	欠測値の場合には1を戻し、そうでない場合は0を戻す。文字の欠測値の場合は、空白の引用符を使って比較することもできます（例: m == ""）。
Zero Or Missing(*m*)	1	値が0または欠測値の場合に1を戻す。引数は数値か行列でなければならず、文字列は不可。
All([2 2]==[1 2])	0	要素ごとの比較を要約する。**すべて**の比較が真の場合にのみ1を戻し、それ以外の場合は0を戻す。
Any([2 2]==[1 2])	1	要素ごとの比較を要約する。**いずれか**の比較が真の場合に1を戻し、そうでない場合は0を戻す。

欠測値

欠測値を含む比較の多くは欠測値を戻し、真または偽は戻されません。そのため、スクリプトの処理が止まらないよう、常に真となる結果を含めることが大切な場合があります。データテーブル列に1、2、3という値が含まれており、列「**A**」に欠測値が1つあるとします。列「**B**」の計算式が比較を設定します。次のスクリプトを見てみましょう。

```
New Table( "比較のテスト",
   Add Rows( 4 ),
   New Column( "A",
      Numeric,
      "Continuous",
      Format( "Best", 10 ),
      Set Values( [1, 2, 3, .] )
   ),
   New Column( "B", Character, "Nominal", Formula( If( :A, "true", 1, "false" ) ) )
);
```

次の値が列「B」に入れられます。

```
"true"
"true"
"true"
"false"
```

このスクリプトは次のように機能します。

`If(`	比較を開始する。
`:A, "true",`	「**A**」の値が欠測値でも0でもない場合、結果は"true"。最初の3行では、この比較の結果は真です。
`1, "false"`	1の値は常に真であるため、欠測値は"false"を戻します。
`)`	比較を閉じる。

欠測値を既知の値と比較する場合、例外が2つあります。

- 1つの値が真で、もう1つが欠測値の場合、`Or()`は真を戻します。(`Or()`テストでは、いずれかの値が真であれば、結果は真となります。)
- 1つの値が偽で、もう1つが欠測値の場合、`And()`は偽を戻します。(`And()`テストでは、両方の値が真でなければ結果は真となりません。)

Is Missing

いずれかの値が欠測値であるとわかっている場合は、`Is Missing()`を使った比較も可能です。前述の例は、欠測値に対して"missing"を戻すように書き直すことができます。その場合、次のようになります。

```
If( :A, "true", Is Missing( :A ), "missing", "false" );
```

この例では、Aが欠測値でもゼロでもないときに"true"を戻し、Aが欠測値のときに"missing"を戻し、それ以外の場合は"false"を戻します。

Zero Or Missing

欠測値が0の可能性がある場合は、代わりにZero Or Missing()関数を使用します。

```
Zero Or Missing( A );
```

この式は、Aが0または欠測値の場合に1を戻します。

ヒント： 既知の値を、明示的な欠測値と比較することはできません。比較できるのは、欠測値を**含む**変数や行列などだけです。

問い合わせ関数

問い合わせ関数を使うと、文字列、リスト、行列などの要素のタイプを調べることができます。要素のタイプがわかれば、それに合わせてスクリプトを記述することができます。

また、JMPは、問い合わせ関数を使ってディレクトリやファイルが書き込み可能かどうかを特定したり、コンピュータのオペレーティングシステムやJMPのバージョンを識別したりできます。

一般的な要素のタイプ

Type()関数は、結果の値のタイプを示す文字列を戻します。例:

```
Show( Type( 1 ), Type( "hi" ), Type( {"a", 2} ), Type( [10 24 325] ) );
```

の結果は、次のようになります。

```
Type(1) = "Integer"
Type("hi") = "String"
Type({"a", 2}) = "List"
Type([10 24 325]) = "Matrix";
```

特定の要素のタイプ

その他の問い合わせ関数（Is Matrix()、Is List()、Is Scriptable()など）は、特定のタイプのオブジェクトをテストします。次の例は、Is Matrix()が真の場合に特定の計算を実行します。

```
a = [2 3];
b = [1, 1];
c = a * b;
If( Is Matrix( c ),
   (c ^ a) / (a * b),
   Print( "c is not a matrix." )
```

```
);
  [5 25]
```

Is Scriptable()は、オブジェクトがスクリプト可能なものの場合に1を戻します。次の例の4つの変数は、データテーブル、列、プラットフォーム、レポートを参照します。どのオブジェクトもスクリプト可能なので、Is Scriptable()はすべての例で1を戻します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
col = Column( "体重(ポンド)" );
plat = Bivariate( Y( :Name("体重(ポンド)") ), X( :Name("身長(インチ)") ) );
rep = Report( plat );
Show( Is Scriptable( dt ) );
  Is Scriptable(dt) = 1;
Show( Is Scriptable( col ) );
  Is Scriptable(col) = 1;
Show( Is Scriptable( plat ) );
  Is Scriptable(plat) = 1;
Show( Is Scriptable( rep ) );
  Is Scriptable(rep) = 1;
```

Is Empty()は、変数に値、関数、式、またはオブジェクトへの参照があるかどうかを調べます。作成されていない変数や、まだ値が割り当てられていない変数を指定すると、エラーが戻されます。プログラマはこういった変数を**「初期化されていない変数」**と呼んでいます。

次の例は、データテーブルが開いているかどうか、つまりdt変数に割り当てられているかどうかを調べる例です。データテーブルが開いていない場合は、Open()関数がユーザにテーブルを開くよう促します。

```
If( Is Empty( dt = Current Data Table() ),
   dt = Open()
);
```

Is Empty()関数はどの変数（グローバル変数、ローカル変数、列など）にも使用できます。

表5.6に、オブジェクトのタイプを識別する関数をまとめます。

表5.6 オブジェクトのタイプを識別する問い合わせ関数

構文	説明
Is Associative Array(*x*)	引数が連想配列のときは1、そうでなければ0を戻す。
Is Directory(x)	引数xがディレクトリのときは1、そうでなければ0を戻す。
Is Empty(*global*) Is Empty(*dt*) Is Empty(*col*)	グローバル変数、データテーブル、またはデータ列に値がない（初期化されていない）ときは1、あれば0を戻す。
Is Expr(*x*)	引数が式のときは1、そうでなければ0を戻す。
Is File(x)	引数xがファイルのときは1、そうでなければ0を戻す。

表5.6 オブジェクトのタイプを識別する問い合わせ関数（続き）

構文	説明
Is List(*x*)	引数がリストのときは1、そうでなければ0を戻す。
Is Matrix(*x*)	引数が行列のときは1、そうでなければ0を戻す。
Is Name(*x*)	引数が名前のときは1、そうでなければ0を戻す。詳細は、「結果ではなく保存された式を取り出す」（216ページ）の節を参照してください。
Is Namespace(*x*)	引数が名前空間のときは1、そうでなければ0を戻す。
Is Number(*x*)	引数が数値か欠測値のときは1、そうでなければ0を戻す。
Is Scriptable(*x*)	引数がスクリプト可能なオブジェクトのときは1、そうでなければ0を戻す。
Is String(*x*)	引数が文字列のときは1、そうでなければ0を戻す。
Type(*x*)	引数（x）のタイプを示す文字列を戻す。

オブジェクトの属性

JMPには、ファイルまたはディレクトリへの書き込みを行う前にそれが書き込み可能であるかどうかを特定するため、次のような関数が用意されています。これらの関数は、スクリプトの対象や属性を検証するIs Directory(path)およびIs File(path)と共に使用してください。その他の情報については、『スクリプト構文リファレンス』の「関数」章を参照してください。

Is Directory Writable(path)関数は、引数pathで指定したディレクトリが書き込み可能であるときに1、そうでなければ0を戻します。

Is File Writable(path)関数は、引数pathで指定したファイルが書き込み可能であるときに1、そうでなければ0を戻します。

次の例は、パスがディレクトリかどうか、そして、そのディレクトリが書き込み可能かどうかを検証します。

```
If( Is Directory( "$SAMPLE_DATA/Loss Function Templates" ),
   If( Is Directory Writable( "$SAMPLE_DATA/Loss Function Templates" ),
      "Directory is writable.",
      "Directory is read only!"
   ),
   "Is a read only directory."
);
```

ホスト情報

Host Is()問い合わせ関数は、現在のオペレーティングシステムを識別します。その後、オペレーティングシステムに合ったアクションの実行が可能になります。

たとえば、オペレーティングシステムがWindowsなら、次のスクリプトはWindowsダイナミックリンクライブラリ（DLL）をロードします。

```
If( Host is( "Windows" ),
   dll_obj = Load DLL( "C:/Windows/System32/user32.dll" )
);
```

Host Is()を使うと、レポート用に各オペレーティングシステムで異なるテキストサイズを指定することもできます。Windows上で作成したスクリプトをMacintoshのユーザと共有すると、結果の表示が意図していたものと多少異なる場合があります。たとえば次の行を使用した場合、テキストは、Macintoshなら大きめ、Windowsなら小さめに表示されます。

```
textsize = If( Host is( "Mac" ), 12, 10 );
```

バージョン情報

JMP Version()問い合わせ関数は、JMPバージョンを文字列で戻します。この関数を使ってJMPのバージョンを確認した後、そのバージョンと互換性のあるスクリプトを実行できます。

```
JMP Version(); // JMP 13 では "13.0.0" などを戻す
JMP Version(); // JMP 9 では " 9.0.0" などを戻す
```

バージョン番号が10.0.0より小さい場合は、2桁のバージョンと比較しやすいように先頭にスペースが挿入されます。先頭のスペースがなければ、9.0.0は10.0.0よりも大きな数（新しいバージョン）と解釈されてしまうためです。

第6章

データタイプ
数、文字列、日付、通貨、その他の使用

この章では、次のような基本的なデータタイプについて説明します。

- 数および文字列
- 特殊な文字列であるパス
- 特殊な数または特殊な文字列である日付および時間
- 通貨
- 16進数値およびBLOB

章の最後の2つの節では、文字列を使った高度な操作と正規表現を使ったパターンマッチの手法を紹介しています。

数および文字列

数は、整数や小数、Eの後ろに10のべき指数をつけた科学的表記、および日付時間値として表すことができます。ピリオド1つだけのものは欠測値です。

たとえば、次に挙げるものはすべて数です。

```
.   1   12   1.234  3E3  0.314159265E+1 1E-20
```

二重引用符に囲まれた1つまたは複数の文字は**文字列**となります。たとえば、次に挙げるものはすべて文字列です。

```
"Green" "Hello,\NWorld!" "54"
```

数が二重引用符に囲まれた場合は文字列で、数値ではないことに注意してください。数値を文字列に、または文字列を数値に変換するには、次の2つの関数を使用します。

- 文字列を数値に変換するにはNum()を使用します。例:

```
Num( "54" );
  54
```

メモ:Num()は数以外の文字を変換できないため、次のような場合には欠測値が戻されます。

```
Num( "Hello" );
  .
```

- 数値を文字列に変換するにはChar()を使用します。例:

```
Char( 54 );
  "54"
Char( 3E3 )
  "3000"
```

Num()またはChar()の出力で、ロケール固有の形式を保持するには、次の例のように<<Use Locale(1)オプションを含めます。

```
Char( 42, 5, 2, << Use Locale( 1 ) );
  // フランスのロケールで実行した場合、"42,00"になる
```

文字列の中の文字を取り出すには、Substr()関数を使用します。次の例では、文字列内で文字「a」が検出され、メッセージがログに出力されます。

```
ch = Substr( "alphabetic", 1, 1 ); // 最初の文字で始まり、終わる
If( ch == "a",
   Print( "最初の文字は a です。" )
);
  "最初の文字はa です。"
```

Unicode文字

JMPでは、世界の言語のほとんどを、Unicode UTF-8規格とUTF-16規格でエンコーディングおよびテキスト表示できます。Unicode規格のコード表と詳細については、ユニコードコンソーシアム（The Unicode Consortium）を参照してください。

JMPでUnicode文字を表示するには、その文字のUnicodeコードの前に「\!」をつけます。例：

- ギリシャ文字のシグマ（σ）のUnicodeコードはU+03C3なので、JMPでは\!U03C3と指定します。
- ギリシャ文字のミュー（μ）のUnicodeコードはU+03BCなので、JMPでは\!U03BCと指定します。

Unicodeを使って肩文字や添え字を表示する場合

- 下付きの1（$_1$）のUnicodeはU+2081なので、JMPでは\!U2081と指定します。
- 上付きの2（^{2}）のUnicodeはU+00B2なので、JMPでは\!U00B2と指定します。

x^2をUnicodeで表現する場合、JMPでは\!U0078\!U00B2と指定します。

パス変数

パス変数は、ディレクトリやファイルへのショートカットです。ディレクトリやファイルへのパス全体を入力する代わりに、スクリプト内でパス変数を使用できます。パス変数は特殊な文字列であり、常に二重引用符で囲んで使用します。

JMPで広く使用される定義済みのパス変数の1つに$SAMPLE_DATAがあります。この変数はJMPまたはJMP Proインストールフォルダ内のサンプルデータのフォルダを指します。次の例は、「Big Class.jmp」サンプルデータテーブルを開きます。

```
Open( "$SAMPLE_DATA/Big Class.jmp" );
```

JMPではいくつかのパス変数が定義済みです。次の表は、現在のJMPバージョンの定義をまとめたものです。旧バージョンのJMPの変数はこれとは異なる場合があります。

表6.1 パス変数の定義

変数	Path
ADDIN_HOME アドインが保存されているフォルダを定義します。	デフォルトでは、アドインビルダーはお使いのオペレーティングシステムに応じて次の場所に保存されます。 • Windows: "¥C:¥Users¥<ユーザ名>¥AppData¥Roaming¥SAS¥JMP¥Addins¥" • Macintosh: "/Users/<ユーザ名>/Library/Application Support/JMP/Addins/" アドインを共有ネットワークにインストールするには、`Register Addin()` 関数を使用します。詳細については、「アプリケーションの作成」章の「JSLを使ったアドインの登録」（687ページ）を参照してください。
ALL_HOME マシン上のすべてのユーザがアクセスできるフォルダを定義します。	• Windows（JMP）: "C:¥ProgramData¥SAS¥JMP¥<バージョン番号>¥" • Windows（JMP Pro）: "C:¥ProgramData¥SAS¥JMPPro¥<バージョン番号>¥" • Windows（JMP Shrinkwrap）: "C:¥ProgramData¥SAS¥JMPSW¥<バージョン番号>¥" • Macintosh: "/Library/Application Support/JMP/<バージョン番号>/" ディレクトリが存在するかどうかを確認するには、`Is Directory("$ALL_HOME");` スクリプトを実行します。フォルダが存在する場合は、1が戻されます。
DESKTOP	• Windows: "C:¥Users¥<ユーザ名>¥Desktop¥" • Macintosh "/Users/<ユーザ名>/Desktop/"
DOCUMENTS	• Windows: "C:¥Users¥<ユーザ名>¥Documents¥" • Macintosh: "/Users/<ユーザ名>/Documents/"
DOWNLOADS	• Windows: "C:¥Users¥<ユーザ名>¥Downloads¥" • Macintosh: "/Users/<ユーザ名/Downloads/"
GENOMICS_HOME	"¥<JMP Genomicsのインストールディレクトリ>¥"
HOME	• Windows（JMP）: "C:¥Users¥<ユーザ名>¥AppData¥Roaming¥SAS¥JMP¥<バージョン番号>¥" • Windows（JMP Pro）: "C:¥Users¥<ユーザ名>¥AppData¥Roaming¥SAS¥JMPPro¥<バージョン番号>¥" • Windows（JMP Shrinkwrap）: "C:¥Users¥<ユーザ名>¥AppData¥Roaming¥SAS¥JMPSW¥<バージョン番号>¥" • Macintosh: "/Users/<ユーザ名>/"

表6.1 パス変数の定義（続き）

変数	Path
SAMPLE_APPS	• Windows: C:¥<JMPインストールディレクトリ>¥Samples¥Apps¥" • Macintosh: "/Library/Application Support/<JMPインストールディレクトリ>/Samples/Apps/"
SAMPLE_DASHBOARDS	• Windows: C:¥<JMPインストールディレクトリ>¥Samples¥Dashboards¥" • Macintosh: "/Library/Application Support/<JMPインストールディレクトリ>/Samples/Dashboards/"
SAMPLE_DATA	• Windows: "C:¥<JMPインストールディレクトリ>¥Samples¥Data¥" • Macintosh: "/Library/Application Support/<JMPインストールディレクトリ>/Samples/Data/"
SAMPLE_IMAGES	• Windows: "C:¥<JMPインストールディレクトリ>¥Samples¥Images¥" • Macintosh: "/Library/Application Support/<JMPインストールディレクトリ>/Samples/Images/"
SAMPLE_IMPORT_DATA	• Windows: "C:¥<JMPインストールディレクトリ>¥Samples¥Import Data¥" • Macintosh: "/Library/Application Support/<JMPインストールディレクトリ>/Samples/Import Data/"
SAMPLE_SCRIPTS	• Windows: "C:¥<JMPインストールディレクトリ>¥Samples¥Scripts¥" • Macintosh: "/Library/Application Support/<JMPインストールディレクトリ>/Samples/Scripts/"
TEMP	• Windows: "C:¥Users¥<ユーザ名>¥AppData¥Roaming¥Temp¥" • Macintosh: "/private/var/folders/.../Temporary Items/"
USER_APPDATA JMP環境設定やメニュー、ホームウィンドウに加えた変更、およびデバッガセッションの設定がここに保存されます。	• Windows（JMP）: "C:¥Users¥<ユーザ名>¥AppData¥Roaming¥SAS¥JMP¥<バージョン番号>¥" • Windows（JMP Pro）: "C:¥Users¥<ユーザ名>¥AppData¥Roaming¥SAS¥JMPPro¥<バージョン番号>¥" • Windows（JMP Shrinkwrap）: "C:¥Users¥<ユーザ名>¥AppData¥Roaming¥SAS¥JMPSW¥<バージョン番号>¥" • Macintosh: "/Users/<ユーザ名>/Library/Application Support/JMP/<バージョン番号>/"

パス変数の定義は、使用しているJMPのバージョンに応じたものとなります。たとえば、JMP 9で作成されたスクリプトであっても、JMP 12で実行した場合は、JMP 12におけるパス変数の定義が使用されます。

パス変数の定義を確認するには、Get Path Variable関数を使用します。

```
Get Path Variable( "HOME" );
  "C:¥Users¥<ユーザ名>¥AppData¥Roaming¥SAS¥JMP¥13¥"
```

Set Path Variable()またはGet Path Variable()にはドル記号は指定しません。ただし、スクリプト内で変数を使用する際には、ドル記号を指定する必要があります。

末尾のスラッシュ

パス変数の最後に必ずスラッシュ（または \）を付けてください。次の例では、dtName変数に"Big Class"というルート名が割り当てられています。Open()式は、$SAMPLE_DATAと末尾のスラッシュを評価し、dtNameの値とファイル拡張子の.jmpを付加します。

```
dtName = "Big Class";
dt = Open( "$SAMPLE_DATA/" || dtName || ".jmp" );
```

このパスは、次のように解釈されます。

```
C:¥Program Files¥SAS¥JMP¥13¥Samples¥Data¥Big Class.jmp
```

$SAMPLE_DATAの後にスラッシュがない場合、パスは次のように解釈されます。

```
C:¥Program Files¥SAS¥JMP¥13¥Samples¥DataBig Class.jmp
```

パス変数の作成とカスタマイズ

Set Path Variable()を使って、独自のパス変数を作成したり、ビルトインのパス変数を上書きしたりできます。次の例で、パス変数はrootです。この変数はc:¥ディレクトリを指します。

```
Set Path Variable( "root", "c:¥ ");
```

新しい変数の値を取得するには、Get Path Variable()を使用します。

```
Get Path Variable( "root" );
  "c:¥"
```

独自のパス変数も他の変数と同様に使用します。次の式は、c:¥ディレクトリのmyimportdata.txtファイルを開きます。

```
Open( "$root¥myimportdata.txt" );
```

パス変数の取得の場合と同様、パス変数を設定する際はドル記号を指定しません。

相対パス

変数で相対パスを使用する予定がある場合は、デフォルトのディレクトリを設定しなければなりません。そうすることで、先頭にドライブ文字がないパスはすべて、デフォルトのディレクトリへの相対パスになります。以下はその例です。

```
Set Default Directory( "c:¥users¥smith¥data" );
```

デフォルトのディレクトリの値を戻すには、Get Default Directory() を使用します。

```
Get Default Directory();  // "c:¥users¥smith¥data" を戻す
```

そこで、次の式は C:¥users¥smith¥data¥cleansers.jmp と解決されます。

```
Open("cleansers.jmp");
```

ファイルパスの区切り

JMP では、通常、区切り記号にスラッシュ（/）を使用する Portable Operating System Interface（POSIX）（UNIX）形式のファイルパスを使用します。これにより、Windows と Macintosh で実行するスクリプト内で現在のオペレーティングシステムを識別する必要はありません。ただし、各ホストは、そのホストでのネイティブ形式のファイルパスも許容しています。

ファイルパス形式を Windows から POSIX（またはその逆）に変換するには Convert File Path() を使用します。パスをファイルや別のアプリケーションに出力する必要がある場合は、POSIX から Windows のパス形式に変換すると便利です。構文は次のとおりです。

```
Convert File Path (path, <absolute|relative>, <POSIX|windows>, <base(path)>);
```

たとえば、次のスクリプトは POSIX パスを Windows パスに変換します。

```
Convert File Path( "c:/users/smith", windows);
  c:¥users¥smith
```

二重引用符内のパスをパス変数（$HOME など）に置き換えることもできます。

日付時間の関数と形式

日付時間値は、日付や時間の要素で構成されます。値は、秒（3388594698）、完全な日付（“2011 年 5 月 18 日” など）、日付と時間（“11/05/18 20:18:18”）、週番号（3）などです。

JMP では、日付時間値を共通の形式に変換し、算術演算を行ったり、さまざまな方法で処理したりできます。

ヒント： すべての日付時間関数と関連する引数の詳細については、『スクリプト構文リファレンス』の「JSL 関数」章を参照してください。

日付時間値

日付時間値は、1904 年 1 月 1 日午前 0 時から数えた秒数として保存され、計算されます。例：

```
Today(); // これは、2011 年 5 月 19 日午前 12:00:00 の場合、3388649872 を戻す
```

Today() と同様、Date DMY() 関数と Date MDY() 関数も、月、日、年の引数を秒数で戻します。たとえば、現在が 2011 年 5 月 19 日午前 12:00:00 だとしたら、次のすべてのステートメントは同じ値を戻します。

```
Date DMY( 19, 5, 2011 );
Date MDY( 5, 19, 2011 );
Today();
  3388608000
```

As Date() 関数は引数として指定された秒数を日付または時間の長さとして戻します。

- 1年以上の期間を表す値は、日付として戻されます。

```
As Date( 3388608000 );
  19May2011
```

- 1年未満を表す値は、時間の長さとして戻されます。

```
As Date( 50000 );
  :0:13:53:20
```

日付時間値は次の2つの形で使用できます。

- リテラル値。例: 19May2011:10:10
- 文字列。例: "2011年5月19日 木曜日"

日付時間のリテラルは、秒数を基本数とし、算術演算に使用できます。

```
As Date( 19May2011 + 1 );
  19May2011:00:00:01
```

日付時間関数を使用したプログラム

表6.2に、秒数を日付時間値に変換する関数と、日付時間値を秒数に変換する関数を示します。

表6.2 日付時間関数

関数	説明
Abbrev Date(*date*)	指定された日付（**date**）からOSで指定されているロケールの日付表示を戻す。形式はコンピュータの地域設定に基づきます。英語（米国）ロケールの場合、日付は "02/29/2004" のような形式になります。英語版のJMPを別のロケールで実行している場合も、ロケールの形式が適用されます。

表6.2 日付時間関数（続き）

関数	説明
As Date(*expression*)	指定された数値または式を日付形式または期間で戻す。たとえば、1年以上の期間を表す値は、日付として戻されます。 `x = As Date( 8Dec2000 + In Days( 2 ) );` は、次のように表示されます。 *10Dec2000* 1年未満を表す値は、時間の長さとして戻されます。 `As Date( 50000 );` は、次のように表示されます。 *:0:13:53:20*
Date DMY(*day*, *month*, *year*)	指定された日付を、1904年1月1日午前0時からの秒数で表示する。たとえば、2004 年 2 月 29 日は DateDMY(29,2,2004) と指定します。これは3160857600を戻します。
Date MDY(*month*, *day*, *year*)	指定された日付を、1904年1月1日午前0時からの秒数で表示する。たとえば、2004 年 2 月 29 日は DateMDY(2,29,2004) と指定します。これは3160857600を戻します。
Day Of Week(*date*)	指定された日付（*date*）が何曜日であるかを表す整数値を戻す。この関数では、週は、日曜日から始まり、土曜日で終わるとみなします。
Day Of Year(*date*)	指定された日付（*date*）がその年の何日目であるかを表す整数値を戻す。
Day(*date*)	指定された日付（*date*）がその月の何日であるかを表す整数値を戻す。
Format(*date*, *"format"*)	2番目の引数で指定された形式（**format**）で*値*を戻す。秒数を日付時間値で表示するときに最もよく使われます。選択できる形式は、**「列情報」**ダイアログボックスに表示されています。表6.3「JMPの2桁年の解釈方法」（130ページ）も参照してください。
Hour(*datetime*)	指定された日付時間値（*date-time*）の時間を表す整数値を戻す。
In Days(*n*)	これらの関数はそれぞれ、*n*分、*n*時間、*n*日、*n*週間、*n*年を秒で表した値を戻す。秒数で表した時間間隔をこれらの関数で割ると、別の時間単位に変換できます。
In Hours(*n*)	
In Minutes(*n*)	
In Weeks(*n*)	
In Years(*n*)	

表6.2 日付時間関数（続き）

関数	説明
Long Date(*date*)	指定された日付（*date*）からOSで指定されているロケールの日付表示を戻す。形式はコンピュータの地域設定に基づきます。英語（米国）ロケールの場合、日付は"Sunday, February 29, 2004"のような形式になります。英語版のJMPを別のロケールで実行している場合も、ロケールの形式が適用されます。
MDYHMS(*date*)	指定された日付（*date*）から"2/29/04 00:02:20 AM"のような形式の日付表示を戻す。
Minute(*date-time*)	指定された日付時間値（*date-time*）の分を表す整数値を戻す。
Month(*date*)	指定された日付（*date*）の月を数値で戻す。
Num(*date-time*)	指定された日付時間値（*date-time*）を表す整数値を戻す。
InFormat(*string*, "*format*") Parse Date(*string*, *"format"*)	指定された形式（*format*）の文字列（*string*）を解析し、日付時間値を戻す。As Date()と同様に、日付をddMonyyyy形式で戻します。
Second(*date-time*)	指定された日付時間値（*date-time*）の秒を表す整数値を戻す。
Short Date(*date*)	指定された日付（*date*）をロケールに関係なくmm/dd/yyyyの形式で戻す（たとえば"02/29/2004"）。
Time Of Day(*date*)	指定された日時（*date-time*）がその日の何秒目かを整数値で戻す。
Today()	1904年1月1日午前0時から現在の日時までの秒数を戻す。引数はとりませんが、括弧は必要です。
Week Of Year(*date*, *<ルール番号>*)	日付がその年の何週目であるかを戻します。一年の第1週を定義するルールは3つあり、オプションとして指定できます。 • ルール1（デフォルト）では、週は日曜日から始まり、年の最初の日曜日が第2週となります。第1週は一部だけの週となるかまたは存在しません。 • ルール2では、最初の日曜日が第1週となり、その前にある日は第0週となります。 • ルール3は、ISO-8601方式の週番号を戻します。週は月曜日から始まり、その年に入ってからの4日間を含む最初の週が第1週となります。年の最初の3日間または最後の3日間が前年または翌年の週番号に属する場合があります。
Year(*date*)	指定された日付（*date*）の年を数値で戻す。

共通の日付時間関数の例

日付時間を戻す関数内では、秒数を戻すすべての関数を使用できます。

たとえば、今日が 2011 年 5 月 19 日の午前 11:37:52 だとすると、Today() は秒数を戻し、それ以下の関数は基本時間からの秒数を別の日付時間形式で戻します。

```
Today()
  3388649872
Short Date( Today() );
  "05/19/2011"
Long Date( Today() );
  "2011年5月19日"
Abbrev Date( Today() );
  "2011/5/19"
MDYHMS( Today() );
  "05/19/2011 11:37:52"
```

括弧内の日付の引数には、秒数（または秒数を戻す関数）または日付時間のリテラル値を指定できます。たとえば、次の2つの式は同じ値を戻します。

```
Long Date( 3388649872 );
Long Date( 19May2011 );
  "2011年5月19日"
```

メモ: Long Date() と Abbrev Date() の値は、お使いのコンピュータの地域設定に応じて異なります。

日付の一部の抽出

関数Month()、Day()、Year()、Day Of Week()、Day Of Year()、Week Of Year()、Time Of Day()、Hour()、Minute()、Second() を使って、日付値の一部を抽出できます。これらの関数は、すべて整数を戻します。現在日付が2011年5月24日の場合、以下の例はすべて年の144日目を表す144ををを戻します。

```
Day of Year( Today() );
Day of Year( 24May2011 );
Day of Year( Date MDY( 5, 24, 2011 ) );
  144
```

例

データテーブルの「日付」という名前の列に、"m/d/y" 形式の日付時間値が含まれています。ここで、時間だけが含まれる列を作成するとします。次のスクリプトは、2列目の計算式が1列目の「日付」の値から時間を抽出します。

```
New Table( "Assembly Tests",
   Add Rows( 1 ),
   New Column( " 日付 ",
      Numeric, Continuous,
      Format( "m/d/y" ),
```

```
        Set Values( [3389083557] )
    ),
    New Column( "時間",
        Numeric, Continuous,
        Formula( Format( Time Of Day( :日付 ), "h:m:s") )
    )
);
```

図6.1はこの結果です。時間は「**日付**」列には表示されないことに注意してください。これは、Format関数によって“m/d/y”形式を適用しているためです。

図6.1 時間の抽出の例

	日付	時間
1	05/24/2011	12:05:57

年の週を決定するルール

Week of Year() は、その年の何週目であるかを、日付時間値で戻します。一年の第 1 週を定義するルールは3つあり、オプションとして指定できます。

- ルール1（デフォルト）では、週は日曜日から始まり、年の最初の日曜日が第2週となります。第1週は一部だけの週となるかまたは存在しません。

  ```
  Week Of Year( Date DMY( 19, 6, 2013 ), 1 );
    25
  ```

- ルール2では、最初の日曜日が第1週となり、その前にある日は第0週となります。

  ```
  Week Of Year( Date DMY( 19, 6, 2013), 2 ) ;
    24
  ```

- ルール3は、ISO-8601方式の週番号を戻します。週は月曜日から始まり、その年に入ってからの4日間を含む最初の週が第1週となります。年の最初の3日間または最後の3日間が前年または翌年の週番号に属する場合があります。

  ```
  Week Of Year( Date DMY( 19, 6, 2013 ), 3 );
    25
  ```

日付の演算

日付時間データは、他の数値データと同じように、算術演算に使用できます。たとえば、日付時間値から数値を引くといった単純な計算ができます。

また、計算式を書いて実行することもできます。

例

データテーブルの「**日付**」列に、顧客がクレジットカードを使ってガソリンを購入した日付が入っています。顧客が何日おきにガソリンを購入しているかを知りたいとします。次のスクリプトは「**経過日数**」列を作成します。その列の計算式は、直前の行の日付値から現在の行の「**日付**」列の値を引きます。

```
New Table( "ガソリンの購入",
   Add Rows( 3 ),
   New Column( "日付",
      Numeric,
      "Continuous",
      Format( "m/d/y" ),
      Set Values( [3392323200 3393532800 3394828800] )
   ),
   New Column( "経過日数",
      Formula(
         If( Row() == 1,
            ., // 1行目については欠測値を戻す
            (:日付[Row()] - :日付[Row() - 1]) / In Days()
         )
      )
   )
);
```

図6.2はこの結果です。

図6.2 日付時間値の計算例

	日付	経過日数
1	07/01/2011	•
2	07/15/2011	14
3	07/30/2011	15

時間の間隔

In Minutes、In Hours、In Days、In Weeks、In Yearsの各関数は、秒以外の単位で時間を表現するときに使用します。これらの関数はすべて、特定の時間間隔を秒数で戻します。たとえば、次の式は、現在から2012年7月4日までの週数を戻します。

```
(Date DMY( 04, 07, 2012 ) - Today()) / In Weeks();
  -208.225444775132
```

間隔関数の引数が空の場合、間隔は1と見なされます。これを変更するには別の数値を入力します。たとえば、In Years(10)は間隔を10年に変換します。たとえば、次の式は、現在から2037年12月31日までに10年間がいくつ含まれるかを戻します。

```
(Date DMY( 31, 12, 2037) - Today() ) / In Years( 10 );
  2.18927583529799
```

2桁および4桁の年

JMPは、オペレーティングシステム固有の日付時間形式をサポートするのではなく、独自のアルゴリズムで日付時間値を解釈し、表示します。ただし、日付時間区切りは、「地域と言語」コントロールパネル（Windows）または「日付と時刻」の環境設定（Macintosh）で選択されているものが使用されます。

2桁年は、現在のシステムクロック年とJMPのルールに従って解釈されます。たとえば、スクリプト内の年が11で、1990年以降にそのスクリプトを実行した場合、年は2011と表示されます。

```
Long Date( 25May11 );
  "2011年5月25日"
```

曖昧さを回避するには、4桁の年を入力します。次の式は（2011年ではなく）1911年を戻します。

```
Long Date( 25May1911 );
  "1911年5月25日"
```

表6.3に、JMPの2桁年の解釈方法を示します。

表6.3 JMPの2桁年の解釈方法

2桁の年の値	評価する時	結果	例	結果
00–10	1989年以前（Windows）	19__	1979年に5と入力	1905
	1990年以前（Macintosh）			
	1990年以降（Windows）	20__	1991年に5と入力	2005
	1991年以降（Macintosh）			
11–89（Windows）	いつでも	現在の世紀	1988年に13と入力	1913
11–90（Macintosh）			2024年に13と入力	2013
90–99（Windows）	2010年以前	19__	1999年に99と入力	1999
91–99（Macintosh）	2011年以降	20__	2015年に99と入力	2099

メモ: JMPでは、「地域」の設定に関係なく、年が必ず4桁で表示されます。何らかの理由で、2桁で年を表示する必要がある場合は、文字列関数を使ってください。詳細については、「データタイプ」章（117ページ）を参照してください。

データテーブル内の日付時間値

日付時間入力と表示形式の変更

データテーブルでは、日付時間値を1つの形式（**入力形式**）でのみ入力できます。入力した値は、基準日からの秒数として内部に保存され、別の日付時間形式で表示されます。Informat() 関数と Format() 関数を使って、これを制御できます。

- Informat() は、文字列の日付時間値と、その文字列に使用されている日付形式を指定すると、ddMonyyyy 形式で日付を戻します。

```
Informat( "19May2011 11:37:52", "ddMonyyyy h:m:s" );
  19May2011:11:37:52
```

- Format() は、基準日からの秒数（または秒数を戻す日付時間関数）を取り、指定の形式で日付を戻します。

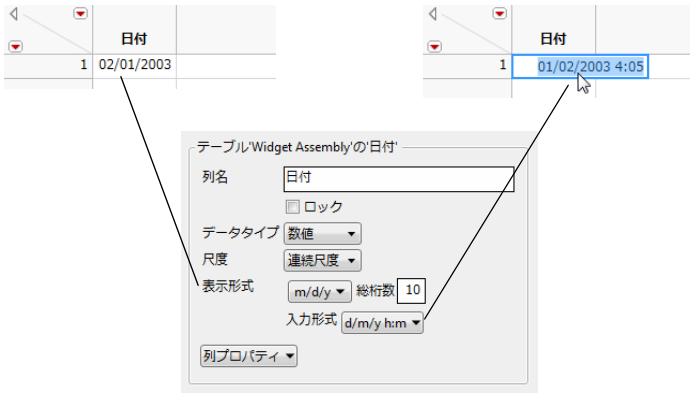
```
Format( 3388649872, "ddMonyyyy h:m:s" );
  "19May2011 11:37:52"
Format( Today(), "ddMonyyyy h:m:s" );
  "19May2011 11:37:52"
```

列に d/m/y h:m 形式の日付を入力し、日付を m/d/y 形式で表示してみましょう。Informat() 関数は入力形式を定義し、Format() 関数は表示形式を定義します。例:

```
New Table( "製品の組立",
   Add Rows( 1 ),
   New Column( "日付",
      Numeric,
      "Continuous",
      Format( "m/d/y" ),
      Informat( "d/m/y h:m" ),
      Set Values( [3126917100] )
   )
);
```

Format() と Informat() の値は、データテーブルの列プロパティに表示されます（図6.3）。セルをクリックして編集する際、日付時間値は入力時の形式で表示されます。値を変更したり、新しい値を入力したりすると、その列の［表示形式］で指定されている形式で値が表示されます。

図6.3 日付時間の表示と入力値の例



メモ:

- 列の値を文字から数値に変換するスクリプトでは、`Format()` と `Informat()` を指定して欠測値を回避します。詳細は、「プログラム例の紹介」章の「文字の日付を数値の日付に変換」（692ページ）の節を参照してください。
- 使用するコンピュータによっては、このマニュアルとは異なり、日付区切り文字がスラッシュ（/）文字ではないことがあります。
- 時間値は、24時間形式、または午前・午後という識別子を付けた形式で入力します。

表6.4に、日付時間関数の引数として、またはデータテーブル形式として使用できる形式を示します。これらの形式 は、データ列への`Format`メッセージの*`format`*引数として使うこともできます。詳細については、「データテーブル」章の「形式の設定または取得」（329ページ）を参照してください。

特定の日付時間関数については、『スクリプト構文リファレンス』の「JSL関数」章を参照してください。

表6.4 日付時間形式

タイプ	形式引数	例
日付だけ	"m/d/y"	"01/02/1999"
	"mmddyyyy"	"01021999"
	"m/y"	"01/1999"
	"d/m/y"	"02/01/1999"
	"ddmmyyyy"	"02011999"
	"ddMonyyyy"	"02Jan1999"
	"Monddyyyy"	"Jan021999"
	"y/m/d"	"1999/01/02"

表6.4 日付時間形式（続き）

タイプ	形式引数	例
	"yyyymmdd"	"19990102"
	"yyyy-mm-dd"	“1999-01-02“
	"yyyyQq"	1999Q1
日付と時間	"m/d/y h:m"	"01/02/1999 13:01" "01/02/1999 午後 1:01"
	"m/d/y h:m:s"	"01/02/1999 13:01:55" "01/02/1999 午後 1:01:55"
	"d/m/y h:m"	"02/01/1999 13:01" "02/01/1999 午後 1:01"
	"d/m/y h:m:s"	"02/01/1999 13:01:55" "02/01/1999 午後 1:01:55"
	"y/m/d h:m"	"1999/01/02 13:01" "1999/01/02 午後 1:01"
	"y/m/d h:m:s"	"1999/01/02 13:01:02" "1999/01/02 午後 1:01:02"
	"ddMonyyyy h:m"	"02Jan1999 13:01" "02Jan1999 午後 1:01"
	"ddMonyyyy h:m:s"	"02Jan1999 13:01:02" "02Jan1999 午後 1:01:02"
	"ddMonyyyy:h:m"	"02Jan1999:13:01" "02Jan1999: 午後 1:01"
	"ddMonyyyy:h:m:s"	"02Jan1999:13:01:02" "02Jan1999: 午後 1:01:02"
	"Monddyyyy h:m"	"Jan021999 13:01" "Jan021999 午後 13:01"
	"Monddyyyy h:m:s"	"Jan021999 13:01:02" "Jan021999 午後 13:01:02"
経過日数と時間	":day:hr:m"	"34700:13:01" ":33:001:01 PM"
	":day:hr:m:s"	"34700:13:01:02" ":33:001:01:02 PM"
	"h:m:s"	"13:01:02" " 午後 01:01:02"

表6.4 日付時間形式（続き）

タイプ	形式引数	例
	"h:m"	"13:01" " 午後 01:02"
	"yyyy-mm-ddThh:mm"	1999-01-02T13:01
	"yyyy-mm-ddThh:mm:ss"	1999-01-02T13:01:02
時間の長さ	":day:hr:m"	“52:03:01” （52日3時間1分）
	":day:hr:m:s"	“52:03:01:30” （52日3時間1分30秒）
	"hr:m"	“17:37” （17時間37分）
	"hr:m:s"	“17:37:04” （17時間37分4秒）
	"min:s"	“37:04” （37分4秒）
メモ：以下の形式は、日付時間をコンピュータの地域設定に従って表示します。これらの形式は、表示にのみ適用され、データテーブルへの日付の入力には使用できません。例は、米国のロケールを使用した場合です。		
短い日付	"Date Abbrev"	（表示のみ）"1999/01/02"
長い日付	"Date Long"	（表示のみ）"1999 年 1 月 2 日 "
ロケールの日付	“Locale Date”	（表示のみ）"1999/01/02"
ロケールの日時	“Locale Date Time h:m”	（表示のみ）"1999/1/2 13:01" または "1999/01/02 01:01 PM"
	“Locale Date Time h:m:s”	（表示のみ）"1999/01/02 13:01:02" または "1999/01/02 01:01:02 PM"

通貨

JMP は、Format() 関数を使って数値を通貨の形式で表示します。関数の構文は次のとおりです。

```
Format(x,"Currency", <"currency code">, <decimal>);
```

ここで

- *x*は列または数値
- *"currency code"*は国際標準化機構（ISO）4217コード
- *decimal*は小数桁数

次はFormat関数の例です。

```
Format( 12345.6, "Currency", "GBP", 3 );
  "£12,345.600"
```

通貨コード（currency code）を指定しない場合、オペレーティングシステムのロケールに基づいた通貨記号が使用されます。たとえば、次のスクリプトを日本語のオペレーティングシステムで実行した場合、円記号が付きます。

```
Format( 12345.6, "Currency", 3);
  "¥12,345.600"
```

JMPでサポートされていない通貨コードを指定すると、数値の前に通貨コードの文字列が付きます。

```
Format( 12345.6, "Currency", "BBD", 3 );
  "BBD 12,345.600"
```

表6.5に、JMPでサポートされている通貨を示します。

表6.5 JMPでサポートされている通貨

コード	通貨	コード	通貨	コード	通貨
AUD	オーストラリアドル	ILS	イスラエル新シェケル	RUB	ロシアルーブル
BRL	ブラジルレアル	INR	インドルピー	SEK	スウェーデンクローネ
CAD	カナダドル	JPY	日本円	SGD	シンガポールドル
CHF	スイスフラン	KRW	韓国ウォン	THB	タイバーツ
CNY	中国元	MXN	メキシコペソ	TRY	新トルコリラ
COP	コロンビアペソ	MYR	マレーシアリンギット	TWD	ニュー台湾ドル
DKK	デンマーククローネ	NOK	ノルウェークローナ	USD	米国ドル
EUR	ユーロ	NZD	ニュージーランドドル	ZAR	南アフリカランド
GBP	英国ポンド	PHP	フィリピンペソ		
HKD	香港ドル	PLN	ポーランドズウォティ		

16進数の関数とBLOB関数

JMPでは、BLOBと呼ばれるバイナリ大容量オブジェクト（Binary Large OBject）も扱うことができます。以下の関数は、16進数の値、数値、文字、BLOBの間での変換を行います。一部の関数については、表6.6の後に詳しい説明があります。

詳細については、『スクリプト構文リファレンス』の「JSL関数」章を参照してください。

表6.6 16進数の関数とBLOB関数

構文	説明
Hex("*text*") Hex("*num*") Hex("*blob*")	テキスト（**text**）、数字（**num**）、またはBLOB（**blob**）を16進数表記に変換した文字列を戻す。 Char To Hexは別名です。
Hex To Blob("*hexstring*")	16進数表記の文字列を、BLOBに変換する。
Hex To Char("*hexstring*", "*encoding*")	16進数表記の文字列を、指定されたエンコーディングに従い、文字列に変換する。 文字列のデフォルトのエンコーディングはutf-8です。また、utf-16le、utf-16be、us-ascii、iso-8859-1、ascii~hex、shift-jis、およびeuc-jpもサポートされています。
Hex to Number	16進数表記の文字列を、数値に変換する。
Char To Blob("*string*") Char To Blob("*string*", "*encoding*")	文字列をバイナリ（blob）に変換する。 blobのデフォルトのエンコーディングはutf-8です。また、utf-16le、utf-16be、us-ascii、iso-8859-1、ascii~hex、shift-jis、およびeuc-jpもサポートされています。
Blob To Char("*blob*") Blob To Char("*blob*", "*encoding*")	バイナリデータ（blob）を文字列に変換する。 文字列のデフォルトのエンコーディングはutf-8です。また、utf-16le、utf-16be、us-ascii、iso-8859-1、ascii~hex、shift-jis、およびeuc-jpもサポートされています。
Blob Peek("*blob*", *offset*, *length*)	元のBLOBから、オフセット*offset*から始まる長さ**length**バイトだけを抜き出し、新しいBLOBとして戻す。オフセットは0を基準にします。

Hex(string)は、引数の各文字に対応する16進数のコードを返します。例:

```
Hex( "Abc" );
```

は次を戻します。

```
"416263"
```

41、62、63はそれぞれ、A、b、cを表す16進数のコード（ASCII）です。

Hex to Char(string)は、16進数を文字に変換します。戻り値の文字が、有効な表示文字でない場合があります。すべての文字は、0-9、A-Z、またはa-zのいずれかの2文字で表されます。スペースまたはカンマは、無視されます。例:

```
Hex To Char( "4142" );
```

は次を戻します。

```
"AB"
```

41と42は、それぞれAとBを表す16進数のコードです。

HexとHex To Charは、逆の処理を行うため、たとえば、

```
Hex To Char( Hex( "Abc" ) );
```

は次を戻します。

```
"Abc"
```

Hex To Blob(string)は、16進表記の文字列（string）をバイナリオブジェクトに変換します。

```
a = Hex To Blob( "6A6B6C" );
Show( a );
  a = Char To Blob("jkl", "ascii~hex")
```

Blob Peek(blob,offset,length)は、引数で指定されたバイトをblobから抽出します。

```
b = Blob Peek( a, 1, 2 );
Show( b );
  b = Char To Blob("kl", "ascii~hex")
b = Blob Peek( a, 0, 2 );
Show( b );
  b = Char To Blob("jk", "ascii~hex")
b = Blob Peek( a, 2 );
Show( b );
  b = Char To Blob("l", "ascii~hex")
```

Hex(blob)は、blobを16進に変換します。

```
c = Hex( a );
Show( c );
  c = "6A6B6C"
d = Hex To Char( c );
Show( d );
  d = "jkl"
```

Concat(blob1,blob2)とblob1 || blob2は、2つのblobを連結します。

```
e = Hex To Blob( "6D6E6F" );
Show( e );
```

```
f = a||e;
Show( f );
  e = Char To Blob("mno", "ascii~hex")
  f = Char To Blob("jklmno", "ascii~hex")
```

Length(blob)は、blobのバイト数を戻します。

```
g = Length( f );
Show( g );
  g = 6
```

メモ:blobがログにリストされる場合、コンストラクタ関数のChar To Blob("...")とともに表示されます。

このとき、ASCIIの範囲外は、ASCII文字ではなく、波線のあとに16進数を示す方法で表記されます。なお、ASCIIの範囲は、スペース（space）から、閉じ中括弧(})までです（16進数のコードで言うと、20から7Dまでです）。例:

```
h = Hex To Blob( "19207D7E" );
Show( h );
i = Hex( h );
Show( i );
  h = Char To Blob("~19 }~7E", "ascii~hex")
  i = "19207D7E"
```

Char To Blob(string)は、文字列からBLOBを作成します。Blob To Char(blob)は、BLOBから文字列を作成します。これらにおいて、表示できないコードやASCII文字ではないコードは、波線と16進数(~XX)によって表わされます。

文字関数の使用

この節では、『スクリプト構文リファレンス』の「JSL関数」章で説明している複雑な関数の一部について、その使用方法を示します。

Concat

Concat関数は、文字列と文字列を連結します。なお、評価された時に名前を戻す式に対しては、その名前を文字列として扱います。次の例は、変数の名前を文字列としてConcat関数で扱うために、Name ExprとChar関数を用いています。

```
n = {abc};
c = n[1] || "def";
Show( c );
  "abcdef"

m = Expr( mno );
```

```
c = m || "xyz";
Show( c );
  名前が解決できません: mno 'mno'へのアクセスまたは評価 , mno/*###*/

m = Expr( mno );
c = Name Expr( m ) || "xyz";
Show( c );
  "mnoxyz"

m = Char( Expr( mno ) );
c = m || "xyz";
Show( c );
  "mnoxyz"
```

Concat Items()は、文字列のリストを、1つの文字列に変換します。各文字列は、区切り文字で区切られます。区切り文字を指定しなかった場合は、スペースで区切られます。構文は、次のとおりです。

```
resultString = Concat Items ({list of strings}, <“delimiter string”>);
```

例:

```
a = {"ABC", "DEF", "HIJ"};
result = Concat Items(a, "/");
```

は次を戻します。

```
"ABC/DEF/HIJ"
```

または、

```
result = Concat Items( a );
```

は次を戻します。

```
"ABC DEF HIJ"
```

Munger

Mungerは、引数として何が指定されているかによって異なる動作をします。

次の表では、Munger(text, *offset*, *find* | *length*, <*replace*>)を使って説明しています。

表6.7 さまざまな種類の引数をとったときのMungerの動作

引数find、length、replace	例
*find*に文字列を指定し、*replace*に文字列を指定しなかった場合、最初に*find*文字列が見つかった位置（offsetの位置から数えた文字数）を戻す。	`Munger( "the quick brown fox", 1, "quick" );` *5*

表6.7 さまざまな種類の引数をとったときのMungerの動作（続き）

引数find、length、replace	例
*length*に正の整数を指定し、*replace*に文字列を指定しなかった場合、offset番目から（*offset*+*length*）番目までの文字列を戻す。	`Munger( "the quick brown fox", 1, 5 );` `  "the q"`
findに文字列を指定し、*replace*に文字列を指定した場合、*text*内のoffsetの後ろで最初に一致した*find*の文字列を*replace*の文字列に置き換える。	`Munger( "the quick brown fox", 1, "quick", "fast" );` `  "the fast brown fox"`
*length*に正の整数を指定し、*replace*に文字列を指定した場合、offset番目から（*offset*+*length*）番目までの文字列を*replace*の文字列に置き換える。	`Munger( "the quick brown fox", 1, 5, "fast");` `  "fastuick brown fox"`
*length*に正の整数を指定し、*offset*+*length*が*text*の文字数以上の場合、*text*の*offset*番目から最後の文字までの文字列を戻す。*replace*に文字列が指定されている場合は、*text*のその部分をreplaceの文字列に置き換える。	`Munger( "the quick brown fox", 5, 25);` `  "quick brown fox"` `Munger( "the quick brown fox", 5, 25, "fast" );` `  "the fast"`
*length*に0を指定し、*replace*に文字列を指定しなかった場合、空の文字列を戻す。	`Munger( "the quick brown fox", 1, 0 );` `  ""`
*length*に0を指定し、*replace*に文字列を指定した場合、Mungerは*offset*番目の前に、その文字列を挿入する。	`Munger( "the quick brown fox", 1, 0, "see " );` `  "see the quick brown fox"`
*length*に負の整数を指定し、*replace*に文字列を指定しなかった場合、*text*の*offset*番目から最後の文字までの文字列を戻す。	`Munger( "the quick brown fox", 5, -5 );` `  "quick brown fox"`
*length*に負の整数を指定し、*replace*に文字列を指定した場合、Mungerは、textのoffset番目から最後の文字までの文字列を、*replace*の文字列に置き換える。	`Munger( "the quick brown fox", 5, -5, "fast" );` `  "the fast"`

Repeat

Repeat()関数は、第1引数のコピーを戻します。第2（ときには第3）引数は繰り返す回数です。これが1のとき、コピー回数は1回です。

第1引数が文字値またはリストのとき、結果はその文字のコピーになります。

```
Repeat( "abc", 2 );
  "abcabc"
Repeat( {"A"}, 2 );
  {"A","A"}
```

```
Repeat( {1, 2, 3}, 2 );
  {1,2,3,1,2,3}
```

第1引数が数値または行列のとき、結果は行列になります。第2引数は行の繰り返し回数で、第3引数で列の繰り返し回数を指定できます。指定されている引数が2つだけの場合、列の繰り返し回数は1になります。

```
Repeat( [1 2, 3 4], 2, 3 );
  [ 1 2 1 2 1 2,
    3 4 3 4 3 4,
    1 2 1 2 1 2,
    3 4 3 4 3 4]
Repeat( 9, 2, 3 );
  [ 9 9 9,
    9 9 9]
```

Repeat関数は、SAS/IML 言語の同じ名前の関数と互換性がありますが、SASのDATA ステップ関数とは互換性がありません。DATA ステップ関数は、この関数より1回多く繰り返します。

Substituteと Substitute Into

Substitute()関数は、置き換えた式を含む文字列のコピーを戻します。Substitute Into()関数は、文字列の入った変数の内容を直接置き換えます。

次のスクリプトを見てみましょう。

```
str1 = str2 = "All things considered";
str3 = Substitute( str1, "All", "Some" );
// str3 に Substitute の結果が格納され、str1 の内容は変更されません。
str4 = Substitute Into( str2, "All", "Some" );
// Substitute Into は何も戻さないので str4 は空白となり、str2 には Substitute Into の
// 結果が格納されます。
Show( str1, str2, str3, str4 );
```

このスクリプトは以下の出力を戻します。

```
str1 = "All things considered";
str2 = "Some things considered";
str3 = "Some things considered";
str4 = .;
```

正規表現

正規表現とは、データのクリーンアップや抽出に頻繁に使用される特定のパターンです。パターンを検索して、別の文字列で置換したり、文字列の一部を抽出したりできます。パターンは、Regex()またはRegex Match()関数内で定義します。

Regex

Regex() は、*source* 文字列内で *pattern* を検索し、文字列を戻します。文字列内にあるパターンを識別したり、文字列を別の文字列に変換することができます。

```
Regex(source, pattern, (<replacementString>, <GLOBALREPLACE>), <format>,
  <IGNORECASE>);
```

IGNORECASE は、大文字と小文字の違いを無視します。GLOBALREPLACE は、文字列全体が処理されるまでマッチを繰り返します。format は、マッチしたグループへの後方参照です。マッチしない場合、Regex() は欠測値を戻します。

文字列のマッチの例

bus|car は正規表現です（文字列であるため、引用符で囲みます）。この式は、「bus（バス）」か「car（自動車）」を検索することを意味しています。

```
sentence = "I took the bus to work.";
vehicle = Regex( sentence, "bus|car" );
  "bus"
```

文字列の置換の例

Regex() のオプションである第3引数では、結果の文字列を指定します。デフォルト値の \0 は、正規表現によってマッチされたものすべてに対する後方参照です。前の例では、「bus」という単語が sentence 内でマッチします。第3引数のデフォルト値である \0 は文章全体を「bus」に置き換えます。

今度は、括弧を使って後方参照を増やしてみましょう。

```
sentence = "I took the bus to work.";
Regex( sentence, "(.*) bus (.*)", "\1 car \2" );
  "I took the car to work."
```

bus の前後にある (.*) は正規表現の一部です。括弧によって、キャプチャするグループが作成されます。「.」には任意の文字がマッチします。「*」には、0個以上の前の表現がマッチします。結果として、1組目の括弧には bus の前にあるものがマッチし、2組目の括弧には bus の後ろにあるものがマッチします。第3引数の \1 car \2 は、テキストを組み替え、bus の代わりに car を挿入します。

詳細については、「後方参照とキャプチャするグループ」（150ページ）を参照してください。

グローバルな置き換えの例

GLOBALREPLACE は、Regex() の動作を変更します。マッチが成功すると、ソース文字列全体が、パターンのマッチした箇所を置換した形で戻されます。マッチがない場合は、ソース文字列が変更なしで戻されます。

```
sentence = "I took the red bus followed by the blue bus to get to work today.";
Regex( sentence, "bus", "car", GLOBALREPLACE);
  "I took the red car followed by the blue car to get to work today."
```

後方参照を使用することもできます。この例では、最初に別の文を使います。

```
sentence = "I took the red bus followed by the blue car to get to work today.";
Regex(
   sentence,
   "(\w*) (bus|car)",
   "bicycle (not \2) that was \1",
   GLOBALREPLACE
);
  "I took the bicycle (not bus) that was red followed by the bicycle (not car)
    that was blue to get to work today."
```

\w* は、0個以上の文字をマッチし、括弧で囲まれているため第1の後方参照となります。同じく括弧で囲まれている bus|car が、第2の後方参照になります。第3引数の bicycle (not \2) that was \1は、ソーステキストの中でマッチした部分をどのように置き換えるかを記述しています。

これで後方参照を使ってデータの位置を入れ替える方法がわかりました。この方法は、たとえば姓と名の位置を入れ替えたいときに便利です。

Regex Match

文字列がマッチしなかった場合、Regex Match() はゼロを要素とする空白のリストを戻します。マッチが成功すると、最初のリストはマッチ全体のテキストになります（後方参照0）。2番目のリストは後方参照1をマッチするテキスト、3番目のリストは後方参照2をマッチするテキスト・・・といった具合になります。

```
Regex Match(source, pattern, <NULL>, <MATCHCASE>);
```

Regex() とは異なり、Regex Match() には大文字と小文字の区別がありません。大文字と小文字を区別してマッチするには、MATCHCASE を含めます。大文字と小文字を区別してマッチし、置換するテキストがない場合は、NULL を指定します。

名前と値の解析の例

次の例では、名前と値のペアを解析します。

```
Regex Match(
   "person=Fred id=77 friend= favorite=tea",
   "(\w+)=(\S*) (\w+)=(\S*) (\w+)=(\S*) (\w+)=(\S*)"
);
  {"person=Fred id=77 friend= favorite=tea", "person", "Fred", "id", "77",
    "friend", "", "favorite", "tea"}
```

\w+ は、1個以上の文字をマッチします。\S* は、スペース以外の0個以上の文字をマッチします。結果のJSLリストでは、フィールド名（person、id、friend、favorite）とそれに対応する値（Fred、77、""、tea）が個別の文字列になります。

Regex Match() の第1引数が変数で、第3引数が置換後の値を指定する場合、マッチしたテキストは変数内で置き換えられます。

RegexとRegex Matchの比較

Regex()とRegex Match()は、どちらも与えられた文字列内でパターンをマッチしますが、戻す結果が異なります。文字列を別の文字列に変換するには、Regex()を使用します。パターンの特定部分にマッチする文字列部分を見つけるには、Regex Match()を使用します。

Regex()と比較して、Regex Match()のほうがより効率よく処理できる例を紹介します。sourceは、6つの文字列から成るリストです。目的は、6つの文字列の中から該当する部分を、データテーブルの「**subject（主語）**」、「**verb（動詞）**」、「**object（目的語）**」の各列に抽出することです（図6.4）。

図6.4 最終的なデータテーブル

	subject	verb	object
1	cat	ate	chicken
2	dog	chased	cat
3	ralph	like	mary
4	girl	pets	dog
5	cat	chased	dog

```
source = {"the cat ate the chicken", "the dog chased the cat", "did ralph like
   mary", "the girl pets the dog",
"these words are strange", "the cat was chased by the dog"};

dt = New Table( "English 101", // データテーブルを作成する
   New Column( "subject", character ),
   New Column( "verb", character ),
   New Column( "object", character )
);

For( i = 1, i <= N Items( source ), i++,
// リスト内の文字列すべてで反復する

   matchList = Regex Match(
      source[i],
// 各マッチの結果をmatchListに割り当てる

      ".*?(cat|dog|ralph|girl).*?(ate|chased|like|pets).*?(chicken|cat|mary|dog)"
   );
   // 各文字列をスキャンする。各グループで0個以上の文字と
   // 1つの項目をマッチする

   If( N Items( matchList ) > 0,
      dt << Add Rows( 1 );
      dt:subject = matchList[2]; // 最初の括弧のマッチ
      dt:verb = matchList[3]; // 2番目の括弧のマッチ
      dt:object = matchList[4]; // 3番目の括弧のマッチ
   );
);
```

```
// matchList の項目数がゼロの場合（文字列 5）、テーブルに
// 行を加えない。マッチした文字列を個別の
// データテーブルセルに入れる。
```

Regex Match() は、1回の試行で {"the cat was chased by the dog", "cat", "chased", "dog"} を戻します。答えがそれぞれ個別の文字列に入っています。似た例で Regex() を使う場合、1度に1つの答えが戻され、後方参照を使って最終的な文字列を構築する必要があります。

```
For( i = 1, i <= N Items( source ), i++,
   s = Regex( source[i],
   ".*?(cat|dog|ralph|girl).*?(ate|chased|like|pets).*?(chicken|cat|mary|dog)",
   "\1" ); // 最初のグループの項目をマッチする
   v = Regex( source[i],
   ".*?(cat|dog|ralph|girl).*?(ate|chased|like|pets).*?(chicken|cat|mary|dog)",
   "\2" ); // 2 番目のグループの項目をマッチする
   o = Regex( source[i],
   ".*?(cat|dog|ralph|girl).*?(ate|chased|like|pets).*?(chicken|cat|mary|dog)",
   "\3" ); // 3 番目のグループの項目をマッチする
   If( !Is Missing( s ) & !Is Missing( v ) & !Is Missing( o ),
      dt << Add Rows( 1 );
      dt:subject = s; // \1 のマッチを戻す
      dt:verb = v; // \2 のマッチを戻す
      dt:object = o; // \3 のマッチを戻す
   );
);
```

後方参照については、「後方参照とキャプチャするグループ」（150ページ）を参照してください。

正規表現に使う特殊文字

正規表現では、通常特殊文字が使用されます。ピリオドは、指定した文字の1つのインスタンスにマッチする特殊文字です。ピリオドとして認識させるためには、バックスラッシュを付けてエスケープする必要があります。次の表現は、ピリオドを感嘆符で置き換えます。

```
Regex( "Bicycling makes traveling to work fun.", "\.", "!", GLOBALREPLACE );
  "Bicycling makes traveling to work fun!"
```

表6.8で、特殊文字とその例を解説します。

表6.8 正規表現に使う特殊文字

\	• リテラル文字の前に置きます。 \ は、HTML のアンカーの終了タグでスラッシュをそのまま文字として解釈します。 • エスケープシーケンスの前に置きます。 \n は改行文字にマッチします。

表6.8 正規表現に使う特殊文字（続き）

^	改行文字を除く、文字列の最初の位置にマッチします。 ^appleは文字列の冒頭にある“apple”にマッチします。
$	改行文字を除く、文字列の最後の位置にマッチします。 apple$は文字列の末尾にある“apple”にマッチします。
.	改行文字を含む、任意の1つの文字にマッチします。 .appleは任意の1つの文字とそれに続く“apple”にマッチします。
|	置換する値を区切る論理和（OR）を示します。 (apple|orange|banana)は、“apple”、“orange”、“banana”にマッチします。
?	0～1個のインスタンスにマッチします。 apple (pie)?は、0～1個の“pie”のインスタンスにマッチします。
*	0個以上のインスタンスにマッチします。
+	1個以上のインスタンスにマッチします。
( )	部分式を囲みます。 (apple|orange|banana)は、“apple”、“orange”、“banana”にマッチします。 ^(\w+)は、行の冒頭にある1つ以上の文字にマッチします。
[ ]	一連の文字にマッチする式を囲みます。 [\s]は空白文字または数字にマッチします。 [a-z0-9]は、“a”から“z”までと数字の“0”から“9”までにマッチします。
{ }	繰り返しを表す式を囲みます。 apple{3}は3回繰り返します。 apple{3,}は、少なくとも3回、できるだけ多く繰り返します。 apple{3, 10}は3～10回繰り返します。 繰り返しの回数をできるだけ少なくするには、疑問符を追加します。たとえば、apple{3,}?は、少なくとも3回、できるだけ少なく繰り返します。

正規表現に使うエスケープ文字

正規表現のバックスラッシュは、リテラル文字の前に置きます。一般的な文字クラスを示す特定の文字をエスケープすることもできます。文字の場合は\w、スペースの場合は\s、などです。次の例は、文字（英数字と下線）およびスペースにマッチします。

```
Regex(
   "Are you there, Alice?, asked Jerry.", // ソース
   "(here|there).+(\w+).+(said|asked)(\s)(\w+)\." ); // 正規表現
  "there, Alice?, asked Jerry."
```

(here|there).+	“there” とカンマ、スペースにマッチします。
(\w+)	“Alice” にマッチします。
.+	“?, “ にマッチします。
(said|asked)(\s)	後にスペースが続く “asked” にマッチします。スペースがない場合、マッチはここで終了します。ソース文字列には、“asked” の直後にスペースがあります。
(\w+)\.	“Jerry” とピリオドにマッチします。

表6.9は、JMPでサポートされているエスケープ文字をまとめた表です。\C、\G、\X、\zはサポートされていません。

表6.9 エスケープ文字

\\	1つのバックスラッシュ
\A	文字列の最初
\b	語の境界。\wと\W、あるいは\Wと\wの間にある長さゼロの文字列。
\B	語の境界以外
\c*X*	ASCII制御文字
\d	1桁の数字 [0-9]
\D	数字ではない1つの文字 [^0-9]
\E	エスケープ文字の処理を停止
\l	1つの小文字 [a-z]
\L	小文字ではない1つの文字 [^a-z]
\Q	\Eが見つかるまでエスケープ文字を無視
\r	キャリッジリターン
\s	1つの空白文字
\S	空白でない1つの文字
\u	1つの大文字 [A-Z]
\U	大文字ではない1つの文字 [^A-Z]
\w	文字 [a-zA-Z0-9_]

表6.9 エスケープ文字（続き）

\W	文字でない1つの字 [^a-zA-Z0-9_]
\x*00*-\x*FF*	16進数の文字
\x{*0000*}-\x{*FFFF*}	Unicodeのコードポイント
\Z	改行前の文字列の最後

貪欲な正規表現と控えめな正規表現

?、*、+の各演算子はデフォルトで貪欲なマッチを行う正規表現です。前にあるできるだけ多くの文字にマッチします。?演算子は、それらを控えめにします。??は0をマッチし、必要に応じて1にマッチします。+?は1にマッチし、さらに他の文字にマッチします。*?は0にマッチし、さらに他の文字にマッチします。

次の例は、文字nで始まり、それをパターン内の最初の\d（数字）と比較します。数字ではないのでマッチしません。パターンが^（行の最初）で始まっていないため、マッチはuに進みます。3が最初の\dにマッチし、2が2番目の\dにマッチするまで、このプロセスが繰り返されます。

```
Regex( "number=32.5", "\d\d" );
  "32"
```

パターンを貪欲にするには、+（1つ以上にマッチ）を使います。

```
Regex( "number=324.5", "\d+" );
  "324"
```

前述の例は、最初はだいたい同じですが、3が見つかり、\dがマッチすると、+が2と4にも貪欲にマッチします。

通常、貪欲なパターンマッチを行うと、文字列が早く消費されるため、マッチが高速になります。場合によっては、控えめなマッチの方が良いこともあります。?を*または+の後ろに追加すると、貪欲から控えめに変わります。

```
Regex( "number=324.5", "\d+?" );
  "3"
```

ここで、+は少なくとも1つの数字のマッチを必要としますが、?によって「できるだけ多く」が「できるだけ少なく」に変わります。3の後で、パターンが満たされたため停止します。

次の結果を比較します。

貪欲:

```
Regex( "number=324.5", "(\d+)(\d+)\.", "first=\1 second=\2" );
  "first=32 second=4"
```

控えめ:

```
Regex( "number=324.5", "(\d+?)(\d+?)\.", "first=\1 second=\2" );
  "first=3 second=24"
```

上記の貪欲な例では、最初の\d+に対して3、2、4が貪欲にマッチされます。その後、2つ目の\d+がマッチできるように、4が戻されます。控えめな例では、別の経路を使い、異なる答えにたどり着きます。最初、2番目の値は2でしたが、パターンがピリオドを4にマッチできなかったため、2つ目の\d+?が 控えめに4にもマッチしました。

控えめなマッチで高速化

貪欲なマッチと控えめなマッチは、同じ結果を出すことが多いですが、そうでない場合もあります。前の節を参照してください。控えめなマッチを使用する理由の一つに、速度が挙げられます。「The quick fox…」で始まる100万字の文字列があり、「fox」の直前の語を知りたいとします。\1に「quick」が含まれると予想して、次のような式を書くかもしれません。

```
The (.+) fox
```

\1に「quick」が含まれる可能性はあります。.+が文字列の最後まで、計100万個の文字を捉え、「fox」が見つかるまで1語ずつ手放します。「fox」が2個以上ある場合は、最初ではなく最後の「fox」が見つかります。速く、確実に最初の「fox」が見つかるようにするには、?演算子を追加します。

```
The (.+?) fox
```

?は、1文字ずつ前進し、「quick」を通過して最初の「fox」を見つけます。この方法なら、遠くへ進み過ぎるよりずっと速く処理できます。

通常、+または*の演算子は、\d*のような限定的な表現に適用され、一連の数字をマッチします。貪欲なマッチの方が控えめなマッチより高速です。

「fox」が複数ある可能性を別にすれば、貪欲なマッチと控えめなマッチでは最終的に同じ答えが出ます。適切な演算子を使えば、マッチの速度が上がります。貪欲が適しているときもあれば、控えめな方がよい場合もあります。それは、マッチする対象によって異なります。

貪欲な.*は、まず最後まで移動した後、最後の「fox」を見つけます。

```
Regex(
   "The quick fox saw another fox eating grapes",
   "The (.*) fox",
   "\1"
);
  "quick fox saw another"
```

控えめな.*?は最初の「fox」で停止します。

```
Regex(
   "The quick fox saw another fox eating grapes",
   "The (.*?) fox",
   "\1"
);
  "quick"
```

貪欲な.*は、処理を何度も繰り返さなければなりません。2つ目の「fox」はありません。

```
Regex(
   "The quick fox saw another animal eating grapes",
   "The (.*) fox",
   "\1"
);
  "quick"
```

次の例では、貪欲な文字マッチの方が適しています。

```
Regex(
   "The quick fox saw another fox eating grapes",
   "The (\w*) fox",
   "\1"
);
  "quick"
```

後方参照とキャプチャするグループ

正規表現は、括弧でグループ分けしたパターン（キャプチャするグループ）で構成される場合があります。`([a-zA-Z])\s([0-9])`では、`([a-zA-Z])`が1つ目のグループで、`([0-9])`が2つ目のグループです。

キャプチャするグループによってマッチしたパターンを置換するには、後方参照を使います。Perlでは、これらのグループを$1、$2、$3といった特殊変数で表します（$1は最初の括弧グループでマッチされたテキストを示します）。JMPでは、バックスラッシュの後にグループ番号をつけます（\1、\2、\3）。

次の例は、置換するテキストを示す第3引数と後方参照を含んでいます。

```
Regex(
   "   Are you there, Alice?, asked Jerry.", // ソース
   " (here|there).+ (\w+).+(said|asked) (\w+)\.", // 正規表現
"  I am \1, \4, replied \2." ); // 体裁を指定する引数
  " I am there, Jerry, replied Alice."
```

`" I am \1,`	「I am」というテキスト、スペース、そして最初にマッチしたパターン、「there」を作成します。
`\4,`	4つ目にマッチしたパターン`(\w+)`の「Jerry」というテキストを作成します。
`replied \2."`	「replied」というテキストとスペースを作成します。2番目にマッチしたパターン`(\w+)`の「Alice.」が続きます。

ルックアラウンドアサーション

ルックアラウンドアサーションは、パターンをチェックしますが、結果の中でそのパターンを戻しません。**先読み**はパターンの前を探します。**後読み**はパターンの後ろを探します。

否定的先読みの例

否定的先読みは、指定のパターンの前にパターンがないことをチェックします。?! は、否定的先読みを示します。次の正規表現は、後に数字またはスペースが**続かない**カンマをマッチし、パターンをカンマとスペースで置き換えます。

```
Regex( "one,two 1,234 cat,dog,duck fish, and chips, to go",
   ",(?!\d|\s)", ", ", GLOBALREPLACE );
  "one, two 1,234 cat, dog, duck fish, and chips, to go"
```

肯定的先読みの例

肯定的先読みは、指定のパターンの前にパターンがあることをチェックします。?=は、肯定的先読みを示します。次の正規表現は、前述の否定的先読みと同じ結果になりますが、後に任意の小文字が続くカンマにマッチします。

```
Regex( "one,two 1,234 cat,dog,duck fish, and chips, to go",
   ",(?=[a-z])", ", ",GLOBALREPLACE );
  "one, two 1,234 cat, dog, duck fish, and chips, to go"
```

肯定的後読みの例

この例では、肯定的後読みの正規表現に「ssn=」または「salary=」のキーワードがマッチしますが、マッチしたテキストにキーワードは含まれません。マッチしたテキストは、ゼロ個またはそれ以上のドル記号、数字、ハイフンから成る文字列です。

```
data = "name=bill salary=$5 ssn=123-45-6789 age=13,name=mary salary=$6
   ssn=987-65-4321 age=14";
redacted = Regex(data, "(?<=(ssn=)|(salary=))[$\d-]*", "###", GLOBALREPLACE);
  "name=bill salary=### ssn=### age=13,name=mary salary=### ssn=### age=14"
```

後方参照を用いた置換で同じ結果を得るには、次のような正規表現を使います。((ssn=)|(salary=))はキャプチャするグループです。"\1"は、そのグループへの後方参照です。

```
data = "name=bill salary=$5 ssn=123-45-6789 age=13,name=mary salary=$6
   ssn=987-65-4321 age=14";
redacted = Regex(data, "((ssn=)|(salary=))[$\d-]*", "\1###", GLOBALREPLACE);
  "name=bill salary=### ssn=### age=13,name=mary salary=### ssn=### age=14"
```

後方参照については、「後方参照とキャプチャするグループ」（150ページ）を参照してください。

パターンマッチ

JSLのパターンマッチを使うと、文字列を柔軟に検索・処理することができます。

パターン変数は、JMPの他の変数と同様に定義・使用します。

```
i = 3; // 数値変数
a = "Ralph"; // 文字変数
t = textbox("Madge"); // ディスプレイボックス変数
p = ( "this" | "that" ) + patSpan(" ")
      + ( "car" | "bus" ); // パターン変数
```

上のステートメントを実行すると、pにパターン値が割り当てられます。パターン値は、他のパターンを作成したり、パターンマッチを実行したりするのに使用できます。patSpan関数は、引数で指定された文字列にマッチするパターンを戻します。たとえば、patSpan("0123456789")は0から9までの数字とマッチします。

```
p2 = "Take " + p + "."; // p を使って他のパターンを作成する
If( Pat Match( "Take this bus.", p2 ), // マッチの実行
   Print( "matches" ),
   Print( "no match" )
);
```

上記のように、パターンがソーステキストとマッチすることだけを知りたい場合もあれば、マッチしたのがバスか車かなど、何がマッチしたのかを知りたい場合もあります。

```
p = ("this" | "that") + Pat Span( " " ) + ("car" | "bus") >?
vehicleType; // パターンがマッチした場合のみ、変数に値を割り当てる
If( Pat Match( "Take this bus.", p ),
   Show( vehicleType ),
   Print( "no match" )
); // ELSE では vehicleType が設定されていないので使用しない
```

上のプログラムでは、ifにおいてパターンマッチの結果をチェックする必要がないので、変数(vehicleType)にデフォルト値をあらかじめ代入しておけばよいでしょう。条件割り当て演算子>?には2つのオペランドがあります。最初のオペランド（左辺）がパターンで、2番目（右辺）が変数名です。>?は、第1引数でパターンを作成し、全体のパターンマッチが成功したら、第1引数とのマッチングの結果を変数（第2引数）に保存します。>>も同様に変数に結果を保存しますが、全体のパターンマッチの成功を待ちません。>>に指定されたパターンがマッチするとそのたびに即座に割り当てが実行されます。

```
findDelimString = Pat Len( 3 ) >> beginDelim + Pat Arb() >?middlePart
+Expr( beginDelim );
testString = "SomeoneSawTheQuickBrownFoxJumpOverTheLazyDog'sBack";
rc = Pat Match( testString, findDelimString, "<<<" || middlePart || ">>>" );
Show( rc, beginDelim, middlePart, testString );
```

上の例では、patMatch関数の3番目の引数として置き換え文字列が使用されています。ここでは、3つの文字列を連結（||演算子）したものに置換されます。3つの文字列のうち、middlePartは、testStringから>?によって抽出されたものです。置き換えはパターンマッチが成功（rc == 1）しなければ実行されないため、このような指定は適切です。

findDelimStringに割り当てられたパターンを見てみましょう。3つのパターンの連結です。最初のPatLen(3)で任意の3文字とマッチさせ、その結果が>>演算子によりbeginDelimに割り当てられます。2番目は>?で任意の文字数の文字列とマッチさせ、>?演算子によりすべてのマッチングが成功した場合、結果をmiddlePartに割り当てます。最後は式で、パターンが作成されたときではなく、パターンが実行されたときにbeginDelimに保持された文字列で構成されます。expr()と同様、引数の評価は実行時に行われます。それにより、パターンは、文中の2箇所に現れる同一の3文字の文字列を検索します。

適切なパターン関数を使うことにより、処理が速くなったり、プログラミングが簡単になったりする場合があります。たとえば、"a"|"b"|"c"はpatAny("abc")と等価です。patNotAny("abc")は"abc"を含まない文字列とマッチします。patBreak("0123456789")は、最初に出会った数字までの文字列（その数字は含まない）とマッチします。このようなパターンは、patNotAnyやPatBreak関数を用いずに記述するのは大変です。

次の例は、小数、指数、記号から構成される数値の文字列とマッチするパターンです。このパターンは、数字のない特殊なケースのマッチも行います。数字に割り当てられたパターンに特に注目してください。

```
digits = Pat Span( "0123456789" ) | "";

number = (Pat Any( "+-" ) | "") >?signPart + (digits) >?wholePart + ("."
+digits | "") >?fractionPart + (Pat Any( "eEdD" ) + (Pat Any( "+-" ) | "")
 + digits | "") >?exponentPart;

If( Pat Match( "-123.456e-78", number ),
   Show( signPart, wholePart, fractionPart, exponentPart )
);
```

固定フィールドの文字列の解析

データは固定フィールドで保存されている場合があります。patTab、patRTab、patLen、patPos、patRPos関数を使用すれば、固定フィールドの文字列を簡単に抽出できます。patTabとpatRTabは、文字列の左右の端から数えて何番目なのかを引数に取ります。指定のタブ位置までの文字列とマッチします。例:

```
p = patPos(10) + patTab(15);
```

PatPos(10)は、10の位置にあるヌル文字列とマッチします。そのためマッチしたときには、位置10までマッチングが勧められます。そして、次にpatTab(15)が現在の位置（10）から位置15までテキストをマッチします。このパターンは、patPos(10)+patLen(5)と等価です。次に、別の例を示します。

```
p = patPos(0) + patRTab(0);
```

この例は開始の0文字から終了の0文字まで、文字列全体をマッチさせます。patRem()関数は、patRTab(0)の省略形で「文字列の残り」を意味し、引数を取りません。パターンマッチングは、次のように文字列の最初にアンカーを設定することもできます。

```
patMatch( "now is the time", patLen(15) + patRPos(0), NULL, ANCHOR );
```

上の例では置換後の文字列ではなくNULLを使用し、オプションでANCHORを使用しています。両方とも大文字です。NULLは置き換えが行われないことを意味します。ANCHORはパターンマッチの開始位置が文字列の最初に固定されることを意味します。デフォルト値はUNANCHOREDです。

次に示すパターンは、再帰的では**ありません**。

```
p = "a" | "b"; // 1文字をマッチさせます。
p = p + p; // 2文字
p = p + p; // 4文字
Pat Match( "babb", patPos(0) + p + patRPos(0) );
```

再帰的なパターンはexpr()を使って現在の定義を参照します。

```
p = "<" + expr(p) + "*" + expr(p) + ">" | "x";
Pat Match( "<<x*<x*x>>*x>", patPos(0) + p + patRPos(0) );
```

expr()はいわゆる「遅延」演算子であり、1行目でpにパターンを割り当てる際には、expr(p)の（p）は評価されません。その次のステートメントではpatMatchによりパターンマッチが実行され、この時、expr()に遭遇するたびに引数の現在の値を参照します。この例では、マッチングの間の値は変化しません。pが自身によって定義されているなら、どのように処理されるのでしょうか。

pは2つの選択肢によって構成されます。右側の選択肢は簡単です。"x"一文字です。左側は少し難解で、<p*p>というパターンです。そして、<p*p>の各pは、"x"の1文字か、もしくは、<p*p>になっています。最後のいつくかの例では、パターンマッチがソーステキスト全体に対して行われるように、patPos(0) + ... + patRPos(0)を使用しています。テキスト全体にマッチングが必要な場合と、サブテキストだけにマッチングが必要な場合があります。ソーステキストを変更しながらこれらの例を試す場合は、テキスト全体へのマッチングを行った方が、何がマッチしたかを簡単に確認できるでしょう。Pat Matchの結果は0または1です。

次の例では「左の」再帰を使用しています。

```
x =  Expr(x) + "a" | "b"; // +は|よりも優先される
```

patMatch関数をFULLSCANモードで使用した場合、マッチングが進められていく際にメモリが使い果たされたり、再帰の階層が膨れ上がってしまう可能性があります。ただし、デフォルトではpatMatch関数はFULLSCANを使用せず、ある仮定に基づいて再帰を停止させ、マッチングを成功させます。FULLSCANモードで実行するには、patMatch関数の第4以降の引数にFULLSCANと指定します。先ほどの例のパターンは、「b」一文字か、「b」の後続がすべて「a」になっている文字列にマッチします。

```
rc = Pat Match( "baaaaa", x );
```

パターンと大文字／小文字

正規表現と違って、パターンマッチでは大文字と小文字の区別がありません。大文字と小文字を区別させたい場合は、Pat Match() または Regex Match() に名前付き引数MATCHCASEを追加します。例:

```
string = "abcABC";

result = Regex Match( string, Pat Regex( "[aBc]+" ) );
Show( string, result );
  / * 戻り値 string = "abcABC」
  result = {"abcABC"}*/

result = Regex Match( string, Pat Regex( "[cba]+" ), NULL, MATCHCASE );
Show( string, result );
  /* 戻り値 string = "abcABC」
  result = {"abc"}*/
```

第7章

データ構造
さまざまなデータの処理

JSLでは、次のようなデータ構造を使って1つの変数にさまざまなデータを入れることができます。

- リストは、入れ子構造のリストや式も含め、多数の値を保持できます。
- 行列は、行と列から成る数値のテーブルです。
- 連想配列は、値にキーをマップします。連想配列以外のほとんどのタイプのデータが使えます。

リスト

リストには、次のような項目を格納できます。

- 数値
- 変数
- 文字列
- 式（割り当てや関数呼び出しなど）
- 行列
- 入れ子構造のリスト

リストは次のいずれかの方法で作成します。

- List関数を使用する
- { } 中括弧を使用する

例

数値や変数を含むリストを作成するには、List() 関数または中括弧を使用します。

```
x = List(1, 2, b);
x = {1, 2, b};
```

リストには、テキスト文字列、入れ子構造のリスト、および関数呼び出しを含めることができます。

```
{"Red", "Green", "Blue", {1, "true"}, sqrt( 2 )};
```

リストに変数を入れ、同時にその変数に値を割り当てることができます。

```
x = {a = 1, b = 2};
```

リストの評価

リストを含んだスクリプトを実行すると、リストが戻り、リストの中の項目は評価されません。

```
b = 7;
x = {1, 2, b, Sqrt( 3 )};
Show( x );
  x = {1, 2, b, Sqrt(3)};
```

リスト内の項目を評価するには、Eval List() 関数を使います。

```
b = 7;
x = {1, 2, b, Sqrt( 3 )};
c = Eval List( x );
  {1, 2, 7, 1.73205080756888}
```

リストを参照する変数のリストを使用する際には、Eval() 関数を使用する必要があります。

次の例を見てください。ここでは、12項目のリストである**fullMonth**と呼ばれる変数が使われています。

```
::fullMonth = {January, February, March, April, May, June, July, August,
  September, October, November, December};
::abbrevMonth = {Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec};
::dow = {Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday};
::levels = {Very Low, Low, Medium Low, Medium, Medium High, High, Very High};
::feel = {Strongly Disagree, Disagree, Neutral, Indifferent, Agree, Strongly
  Agree};
::rating = {Failing, Unacceptable, Very Poor, Poor, Bad, Acceptable, Average,
  Good, Better, Very Good, Excellent, Best};
N Items(::fullMonth); // fullMonth には 12 個の項目があるため 12 を戻す。
N Items(::mlist[1]); /* ::mlist[1] は変数（fullMonth）であり、かつ fullMonth はリストで
  あるため、「N Items() 引数はリストでなければなりません」というメッセージを戻す。*/
```

Eval()関数を追加すると、

```
N Items( Eval( ::mlist[1] ) );
  12 /* ::mlist[1]には::fullMonthが格納されており、それを評価すると::fullMonthのリストが
  戻されるため、12を戻す。*/
```

ループでは、入れ子になったループ内のEval()関数が変数の内容を評価するようにします。例:

```
::mlist = {::fullMonth, ::abbrevMonth, ::dow, ::levels, ::feel, ::rating};
For( g = 1, g <= N Items( ::mlist ), g++,
    For( i = 1, i <= N Items(::mlist[g]), i++,
        Show( ::mlist[g][i] )
    )
);
// 「N Items() 引数はリストでなければなりません」というメッセージを戻す。
```

この問題を修正するには、Eval関数を追加します。

```
::mlist = {::fullMonth, ::abbrevMonth, ::dow, ::levels, ::feel, ::rating};
For( g = 1, g <= N Items( ::mlist ), g++,
    For( i = 1, i <= N Items( Eval( ::mlist[g] ) ), i++,
        Show( ::mlist[g][i] )
    )
);
```

リストを使った割り当て

リストを使って変数に値を割り当てることができます。

例

```
{a, b, c} = {1, 2, 3}; // a に 1、b に 2、c に 3 を割り当てる
{a, b, c}--; // a、b、c を 1 ずつ減らす
{{a}, {b, c}}++; // a、b、c を 1 ずつ増やす
mylist = {1, log( 2 ), e()^pi(), height[40]}; // 式を格納する
```

リスト内の処理の実行

リスト内で処理を実行することができます。

```
a = {{1, 2}, 3, {4, 5}};
b = {{10, 20}, 30, {40, 50}};
c = a + b;
  c = {{11, 22}, 33, {44, 55}}
```

リスト内の項目の数を求める

リスト内の項目の数を求めるには、N Items() 関数を使用します。

```
x = {1, 2, y, Sqrt( 3 ), {a, b, 3}};
N = N Items( x );
Show( n );
  n = 5;
```

添え字

リストの添え字は、リストから指定した項目を抽出します。リスト内の複数の項目を戻すには、リストを添え字として使用します。

メモ: JSL は 1 からカウントを始めるため、リストの最初の要素は [1] となることに注意してください。他の一部の言語のように [0] とはなりません。

例

リスト a には 4 つの項目が含まれます。

```
a = {"bob", 4, [1, 2, 3], {x, y, z}};
Show( a[1] );
  a[1] = "bob";
Show( a[{1, 3}] );
  a[{1, 3}] = {"bob", [1, 2, 3]};
a[2] = 5; // リストの 2 番目の項目に 5 を割り当てる
```

添え字は、リスト内の項目の選択または変更にも使用できます。

```
Show( a );
  a = {"bob", 5, [1, 2, 3], {x, y, z}};
c = {1, 2, 3};
c[{1, 2}] = {4, 4};
// c[{1, 2}] はどちらも 4 となる
Show( c );
  c = {4, 4, 3};
```

割り当てのリストや関数のリストの場合は、添え字に引用符付きの名前を使用できます。

```
x={sqrt( 4 ), log( 3 )};
xx= {a = 1, b = 3, c = 5};
x["sqrt"];
  4
xx["b"];
  3
```

変数名は引用符で囲む必要があります。そうでないと、JMPは指定された変数を評価して、その値を使ってしまいます。次の例では、リスト内のaの値ではなく、リスト内の2番目の項目の値が戻されます。

```
a = 2;
Show( xx[a] );
  xx[a] = b = 3;
```

次の点を念頭に置いてください。

- 次のような状況では、左側に複数の添え字（たとえば、a[i][j] = value、ここでaは添え字可能な項目のリストを含む）を置くことができます。
  - 一番外側のレベル以外はすべてリストでなければなりません。前述の例では、aがリストでなければならず、a[i]は添え字可能なものであれば何でもかまいません。
  - 最後の添え字を除いて、添え字はすべて数値でなければなりません。前述の例では、iが数値でなければならず、jは行列またはリストでもかまいません。
- 添え字は入れ子構造のどのレベルに対しても実行できます。例:

  ```
  a[i][j][k][l][m][n] = 99;
  ```

メモ: 式タイプの列に格納されている行列の値を取得するには、:Discrim Data Matrix[][i] または :Discrim Data Matrix[row()][i]のように二重添え字を使用します。そうすれば、行番号の代わりに値が戻されます。

リスト内で項目を検索する

リスト内の値を探すには、Loc()関数またはContains()関数を使用します。

```
Loc( list, value );
Contains( list, value );
```

Loc()およびContains()は、値の場所を戻します。Loc()は結果を行列で戻し、Contains()は結果を数値で戻します。

次の点を念頭に置いてください。

- Loc関数は、反復する値をそのつど戻しますが、Contains()は、反復する値は最初の1度だけ戻します。
- 値が見つからなかった場合、Loc関数は空の行列を戻し、Contains()はゼロを戻します。

- ある項目がリストに含まれているかどうかを評価するには、Loc() および Contains() を >0 とともに使用します。項目がリスト内にないときの戻り値はゼロです。項目がリスト内に最低1つは存在する場合は、1が戻されます。

メモ: 行列の処理に関する詳細や、行列を戻す Loc() コマンドについては、「行列」（167ページ）を参照してください。

例

```
nameList = {"Katie", "Louise", "Jane", "Jane"};
numList = {2, 4, 6, 8, 8};
```

nameListから"Katie"を検索します。

```
Loc( nameList, "Katie" );
  [1]
Contains( nameList, "Katie" );
  1
```

nameListから"Erin"を検索します。

```
Loc( nameList, "Erin" );
  []
Contains( nameList, "Erin" );
  0
```

numListから数値の8を検索します。

```
Loc( numList, 8 );
  [4, 5]
Contains( numList, 8 );
  4
```

numListに数値の5があるかどうかを調べます。

```
NRow( Loc( numList, 5 ) ) > 0;
  0
Contains( numList, 5 ) > 0;
  0
```

リスト演算子

表7.1は、リスト演算子とその構文を示しています。

表7.1 リスト演算子

演算子および等価の関数		構文	説明
	As List()	As List(*matrix*)	行列をリストに変換して戻す。行列に複数の列がある場合は、各行を1つのリストとしたリストのリストを戻します。
= += -= *= /= ++ --	Assign() Add To() SubtractTo() MultiplyTo() DivideTo() Post Increment() Post Decrement()	{*list*} = {*list*} {*list*} += *value* {*list*} -= {*list*} ...	割り当て演算子の対象がリストで、割り当てられる値がリストの場合は、個々の項目について割り当てが行われる。左側にあるリストの最終的な値は、L-value（左辺に指定できるもの）、つまり値を割り当てることができる名前でなければなりません。 **メモ:** • リストが等しいことを調べるときは、=ではなく、==を使います。 • JMP には、プレインクリメント演算子やプレデクリメント演算子がありません。代わりに、(+=)である Add To() 演算子、または (-=) である Subtract To() 演算子を使用します。
||=	Concat To()	Concat To(*list1*, *list2*, ...)	2番目とそれに続くリストを1番目のリストの最後に挿入します。
||	Concat()	Concat(*list1*, *list2*, ...);	最初のリストのコピーを戻します。その後に追加のリストが挿入されている場合はそのコピーも戻します。
	Eval List()	Eval List(*list*)	リスト（list）内の式を評価した後のリストを戻す。詳細については、「リストの評価」（158ページ）を参照してください。
	Insert Into()	Insert Into(*list*, *x*, <*i*>)	リスト（*list*）の指定箇所（*i*）に新しい項目（*x*）を挿入する。*i*を指定しなかった場合、項目は最後尾に挿入されます。この関数は元のリストを変更します。

表7.1 リスト演算子（続き）

演算子および等価の関数	構文	説明
Insert()	*list* = Insert(*list*, *x*, <*i*>)	リスト（*list*）の指定箇所（*i*）に新しい項目（*x*）を挿入した *list* を戻す。*i* を指定しなかった場合、項目は最後尾に挿入されます。この関数は元のリストを変更しません。
Is List()	Is List(*arg*)	*arg* が List(items) または {items} で作成されたリストである場合に真（1）を戻し、そうでない場合に偽（0）を戻す。空のリストであってもリストなので、IsList({ }) は真を戻します。miss=.（miss という変数の値が欠測値である）の場合、IsList(miss) は欠測値ではなく偽を戻します。
{ } List	List(*a*, *b*, *c*) {*a*, *b*, *c*}	一連の項目を持つリストを作成する。項目は、他のリストを含め、どのような式でもかまいません。項目はカンマで区切る必要があります。テキストは二重引用符（" "）で囲むか、変数に保存して変数として呼び出す必要があります。
N Items	N Items(*list*)	指定されたリスト（*list*）内の項目数を戻す。変数に割り当てることも可能です。
Remove From()	Remove From(*list*, <*i*>, <*n*>)	リスト（*list*）の *i* 番目から *n* 個の項目を削除する。*n* を指定しなかった場合、*i* にある1項目だけが削除されます。*n* と *i* を指定しなかった場合、最後の1項目だけが削除されます。この関数は元のリストを変更します。
Remove()	Remove(*list*, <*i*>, <*n*>)	リスト（*list*）の *i* 番目から *n* 個の項目を削除した結果を戻す。*n* を指定しなかった場合、*i* にある1項目だけが削除されます。*n* と *i* を指定しなかった場合、最後の1項目だけが削除されます。この関数は元のリストを変更しません。
Reverse Into()	Reverse Into(*list*)	リスト（*list*）の項目の順序を逆にする。この関数は元のリストを変更します。
Reverse()	Reverse(*list*)	リスト（*list*）の項目の順序を逆にした結果を戻す。この関数は元のリストを変更しません。

表7.1 リスト演算子（続き）

演算子および等価の関数	構文	説明
Shift Into()	Shift Into(*list*, <*n*>)	リスト（*list*）の最初の*n*個の項目をリスト（*list*）の末尾に移動する。*n*を指定しなかった場合、最初の1項目だけを末尾に移動します。この関数は元のリストを変更します。
Shift()	Shift(*list*, <*n*>)	リスト（*list*）の最初の*n*個の項目を末尾に移動した結果を戻す。*n*を指定しなかった場合、最初の1項目だけを末尾に移動します。この関数は元のリストを変更しません。
Sort Ascending()	Sort Ascending(*list*)	リスト（*list*）の項目を昇順に並べた結果を戻す。この関数は元のリストを変更しません。
Sort Descending()	Sort Descending(*list*)	リスト（*list*）の項目を降順に並べた結果を戻す。この関数は元のリストを変更しません。
Sort List Into()	Sort List Into(*list*)	リスト（*list*）の項目を昇順に並べる。この関数は元のリストを変更します。
Sort List()	Sort List(*list*)	リスト（*list*）の項目を昇順に並べた結果を戻す。この関数は元のリストを変更しません。
[ ] Subscript()	*list*[*i*] *x* = *list*[*i*] *list*[*i*] = *value* *a*[*b*, *c*] Subscript(*a*, *b*, *c*)	リスト（*list*）から*i*番目の項目を抽出する。添え字は、リストまたは行列でもかまいません。
Substitute()	Substitute(*list*, *pattExpr1*, *replExpr1*, ...)	各パターン式のインスタンスに、対応する代替式を代入して、文字列、リスト、または式のコピーを戻す。詳細については、「データタイプ」章の「Substitute と Substitute Into」（141ページ）を参照してください。
Substitute Into()	Substitute Into(*list*, *pattExpr1*, *replExpr1*, ...)	各パターン式のインスタンスに、対応する代替式を代入して、文字列、リスト、または式を変更する。**メモ:** リスト（*list*）または式は変数でなければなりません。詳細については、「データタイプ」章の「Substitute と Substitute Into」（141ページ）を参照してください。

リスト内での反復

リスト内で反復処理を行い、各値を操作したり、特定の値を検索したりできます。次のスクリプトは、リスト内の各項目を調べ、項目が10以下の場合はその値の2乗を戻します。

```
x = {2, 12, 8, 5, 18, 25};
n = N Items( x );
For( i = 1, i <= n, i++,
   If( x[i] <= 10,
      x[i] = x[i] ^ 2
   )
);
Show( x );
  x = {4, 12, 64, 25, 18, 25};
```

Loc()を使うと、新しいリストの中で25と等しい項目を検索できます。

```
Loc( x, 25 );
  [4, 6] // リストの4番目と6番目の項目が25と等しい
```

リストの連結

2つ以上のリストを1つのリストに連結するには、Concat()または||演算子を使用します。元のリストは変更されません。

次の例は、Concat()を使ってリスト*a*とリスト*b*を連結しています。

```
a = {1, 2};
b = {7, 8, 9};
Concat( a, b );
  {1, 2, 7, 8, 9}
```

次の例は、同じ2つのリストを||演算子を使って連結しています。

```
{1, 2} || {7, 8, 9}
  {1, 2, 7, 8, 9}
```

異なる種類（文字列のリストと数値のリストなど）を連結することもできます。

```
d = {"apples", "bananas"};
e = {"oranges", "grapes"};
f = {1, 2, 3};
Concat( d, e, f);
  {"apples", "bananas", "oranges", "grapes", 1, 2, 3}
```

2つ以上のリストを連結し、最初のリストを連結後のリストに置き換えるには、Concat to()または||=演算子を使用します。

次の例では、Concat to()を使用して連結を実行しています。

```
d = {"apples", "bananas"};
e = {"peaches", "pears"};
Concat to(d,e);
Show( d );
  d = {"apples", "bananas", "peaches", "pears"}
```

次の例は、同じ2つのリストを||=演算子を使って連結しています。

```
d = {"apples", "bananas"};
e = {"peaches", "pears"};
d||=e;
Show( d );
  d = {"apples", "bananas", "peaches", "pears"}
```

既存のリストへ入れ子のリストを挿入

既存のリストに入れ子のリストを挿入するには、次のように挿入する位置を指定します。

```
list2 = {{"a", "b"}, {"c", "d"}};
list2[3] = {"apple", "banana"}; // リストを 3 番目の項目として指定する
Show( list2 );
  list2 = {{"a", "b"}, {"c", "d"}, {"apple", "banana"}};
```

リスト内の項目数が不明な場合は、N Items()を使用します。次の例では、リストの末尾に空のリストが挿入されます。

```
list1 = {"a", "b"};
list1[N Items( list1 ) + 1] = {};
Show( list1 );
  list1 = {"a", "b", {}};
```

別のリストを使ったリストのインデックス化

別のリストを使ったリストのインデックス化は、スクリプトを高速にし、また記述を短くする上でとても便利な方法です。詳細については、「別の行列またはリストを使った行列またはリストのインデックス化」（177ページ）を参照してください。

行列

行列とは、数値の行と列を長方形に配列したものを指します。行列に数値を格納すれば、その数値と行列代数を使った計算が実行できます。

この節では、次の点に注意してください。

- 行列は、太字で書かれた大文字の変数で表されます（例: **A**）。

- 行または列が1つしかない行列を**ベクトル**といいます（具体的には、それぞれ**行ベクトル**または**列ベクトル**）。
- わかりやすいように、ここではベクトルの行列を小文字の太字で表します（例: **x**）。
- **スカラー**は行列内にない数値です。

行列の作成

行列を作成するときは、次の点に注意してください。

- 行列の文字は角括弧で囲みます。[...]
- 行列内の数値として、小数部のある値や、負または正の値、また科学表記も使用できます。
- 列の項目は空白のスペースで区切ります。空白のスペースはいくつでも使用できます。
- 行はカンマで区切ります。

高度な行列を作成するには、「特殊な行列」（184ページ）を参照してください。

例

3行、2列の行列**A**を作成します。

```
A = [1 2, 3 4, 5 6];
```

Rは行ベクトルで**C**は列ベクトルです。

```
R = [10 12 14];
C = [11, 13, 15];
```

Bは1x1の行列、つまり、1行1列の行列です。

```
B = [20];
```

Eは空の行列です。

```
E = [];
```

次の例は、空の行列を作成するための2通りの方法を示しています。

```
J( 0,4 );
  [](0,4)

[]( 5,0 )
  [](5,0)
```

オプションで、空の行列内の行数と列数を指定できます。JMPは、必要に応じて行列を作成します。たとえば、次のスクリプトは、空の行列を作成し、そこにデータテーブルの各行の身長と体重を挿入します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
data = []( 0, 2 );
For Each Row( data |/= Matrix( {{dt:Name("身長（インチ）"), dt:Name("体重（ポンド
  )")}} ) );
```

```
Show( data );
  data = [59 95, 61 123, 55 74,...]
```

スクリプトが空の行列を戻す場合もあります。「Big Class.jmp」において、次の式は、年齢が8歳に等しい行を探し、何も検出されなかったので空の行列を戻しています。

```
a = dt << Get Rows Where( 年齢 == 8 );
Show( a );
  a = [](0,1);
```

リストから行列を作成する

リストを行列に変換するには、Matrix()関数を使用します。リストが1つの場合は、1つの列ベクトルに変換されます。リストが複数ある場合は、行に変換されます。

次の例は、1つのリストから列ベクトルを作成します。

```
A = Matrix( {1, 2, 3} );
  [1,2,3]
```

次の例は、リストのリストから行列を作成します。各リストが行列の各行に対応します。

```
A = Matrix( {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}} );
  [1 2 3,
  4 5 6,
  7 8 9]
```

式から行列を作成する

式から行列を作成するには、Matrix()を使用します。要素に指定した式は、評価すると数値になる必要があります。

```
A = Matrix( {4 * 5, 8 ^ 2, Sqrt( 9 )} );
  [20, 64, 3]
```

添え字

添え字演算子（[ ]）を使うと、行列から要素や部分行列を取り出せます。Subscript()関数は通常、対象となる行列の後に、行と列を表す引数を大括弧で囲んで記述します。

単一要素の抽出

A[i,j]は、行列Aからi行j列の要素を取り出し、スカラーとして戻します。同じ機能を持つ関数は、Subscript(A,i,j)です。

```
P = [1 2 3, 4 5 6, 7 8 9];
P[2, 3]; // 6を戻す
Subscript( P, 2, 3 ); // 6を戻す
```

変数 test に、行列 **A** の 3 行目の 1 列目の値（5）を割り当てます。

```
A = [1 2, 3 4, 5 6];
test = A[3, 1];
Show( test );
  test = 5;
```

行列やリストによる添え字

部分行列を抽出するには、行列やリストによる添え字を使用します。選択された行と列の行列が戻されます。次の式では、2行目と3行目の2列目と1列目を抽出します。

```
P = [1 2 3, 4 5 6, 7 8 9];
P[[2 3],[1 3]]; // 添え字を行列で指定
P[{2, 3},{1, 3}]; // 添え字をリストで指定
```

この 2 つはどちらも次のような結果になります。

```
  [4 6,
  7 9]
```

1つしか引数をとらない添え字

添え字の引数を1つだけ指定した場合、その添え字は、すべての行を連結して1つの行にした形での通し番号になります。このため、引数を2つ指定する場合の `A[i,j]` は、引数を1つにした場合の `A[(i-1)*ncol(A)+j]` と同じになります。

例

```
Q = [2 4 6, 8 10 12, 14 16 18];
Q[8]; // Q[3,2] と同じ
  16
```

以下の例は、すべて列ベクトル [10, 14, 18] を戻します。

```
Q = [2 4 6, 8 10 12, 14 16 18];
Q[{5, 7, 9}];
Q[[5 7 9]];
ii = [5 7 9];
Q[ii];
ii = {5, 7, 9};
Q[ii];
Subscript( Q, ii );
```

以下のスクリプトでは、行列 **P** の 1 ～ 9 番目までの値を順に戻します。

```
P = [1 2 3, 4 5 6, 7 8 9];
For( i = 1, i <= 3, i++,
   For( j = 1, j <= 3, j++,
      Show( P[i, j] )
   )
);
```

行および列の削除

行または列の削除は、空の行列をその行または列に割り当てることによって実行できます。

```
A[k, 0] = [];  // k 番目の行を削除する
A[0, k] = [];  // k 番目の列を削除する
```

行全体または列全体を選択する

添え字の引数を0にすると、すべての行または列を指定できます。

```
P = [1 2 3, 4 5 6, 7 8 9];
```

2 列目を選択します。

```
P[0, 2];
  [2, 5, 8]
```

3列目と2列目を選択します。

```
P[0, [3, 2]];
  [3 2, 6 5, 9 8]
```

3行目を選択します。

```
P[3, 0];
  [7 8 9]
```

2行目と3行目を選択します。

```
P[[2, 3], 0];
  [4 5 6, 7 8 9]
```

すべての列と行を選択します（行列全体）。

```
P[0, 0];
  [1 2 3, 4 5 6, 7 8 9]
```

添え字を使った代入

添え字を使って、行列内の値を変更することができます。添え字として、スカラーや、行列またはリスト、またはすべての行や列を意味する数0を指定できます。挿入する行や列の次元は、挿入される要素の次元と一致しているか、または、要素は指定された箇所に繰り返し挿入される必要があります。

例

2行目の3列目の値を99に変更します。

```
P = [1 2 3, 4 5 6, 7 8 9];
P[2, 3] = 99;
Show( P );
  P=[1 2 3, 4 5 99, 7 8 9]
```

4つの位置の値を変更します。

```
P = [1 2 3, 4 5 6, 7 8 9];
P[[1 2], [2 3]] = [66 77, 88 99];
Show( P );
  P=[1 66 77, 4 88 99, 7 8 9]
```

1つの列にある3つの値を変更します。

```
P = [1 2 3, 4 5 6, 7 8 9];
P[0, 2] = [11, 22, 33];
Show( P );
  P=[1 11 3, 4 22 6, 7 33 9]
```

1つの行にある3つの値を変更します。

```
P = [1 2 3, 4 5 6, 7 8 9];
P[3, 0] = [100 102 104];
Show( P );
  P=[1 2 3, 4 5 6, 100 102 104]
```

1行のすべての値を同じ値に変更します。

```
P = [1 2 3, 4 5 6, 7 8 9];
P[2, 0] = 99;
Show( P );
  P=[1 2 3, 99 99 99, 7 8 9]
```

代入演算子

行列や行列の添え字に、代入演算子（+=など）を使用することができます。たとえば、次の例では、行列の i 行 j 列の要素に1を加えます。

```
P = [1 2 3, 4 5 6, 7 8 9];
P[1, 1] += 1;
Show( P );
  P=[2 2 3,
     4 5 6,
     7 8 9];

P[1, 1] += 1;
Show( P );
  P=[3 2 3,
  4 5 6,7 8 9];
```

行と列の範囲指定

範囲を指定した行列を作成するには、Index() 関数 :: を使用します。

```
T1 = 1 :: 3; // ベクトル [1 2 3] を作成する
T2 = 4 :: 6; // ベクトル [4 5 6] を作成する
```

```
T3 = 7 :: 9; // ベクトル [7 8 9] を作成する
T = T1 |/ T2 |/ T3; // ベクトルを 1 つの行列に連結する
T[1 :: 3, 2 :: 3]; // 1 行目から 3 行目の 2 列目と 3 列目を参照する
  [2 3, 5 6, 8 9]
T[Index( 1, 3 ), Index( 2, 3 )]; // Index 関数と等価
  [2 3, 5 6, 8 9]
```

問い合わせ関数

NCol() 関数と NRow() 関数は、それぞれ行列（またはデータテーブル）の列数または行数を戻します。

```
NCol( [1 2 3, 4 5 6] ); // 列が 3 つなので 3 を戻す
NRow( [1 2 3, 4 5 6] ); // 行が 2 つなので 2 を戻す
```

値が行列であるかどうかを調べるには、Is Matrix() 関数を使います。引数が行列を参照している場合は 1 を戻します。

```
A = [20, 64, 3];
B = {20, 64, 3};
Is Matrix( A ); // 真の場合は 1 を戻す
Is Matrix( B ); // 偽の場合は 0 を戻す
```

比較演算子、範囲チェック演算子、論理演算子

JMP の比較演算子、範囲チェック演算子、および論理演算子は、行列に対しても使えます。要素にブール値を持つ行列が結果として戻されます。整合する行列を比較できます。

```
A < B; // より小さい
A <= B; // 以下
A > B; // より大きい
A >= B; // 以上
A == B; // 等しい
A != B; // 等しくない
A < B < C; // 連続的な比較（範囲チェック）
A | B; // 論理和 OR
A & B; // 論理積 AND
```

Any() 演算子や All() 演算子を使うと、行列の比較結果を要約できます。Any() は、0 以外の要素が 1 つでもあれば 1 を戻します。All() は、要素がすべて 0 以外のときに 1 を戻します。

```
[2 2] == [1 2]; // 結果は [0 1] なので
All( [2 2] == [1 2] ); // 結果は 0
Any( [2 2] == [1 2] ); // 結果は 1
```

Min() と Max() は、引数として与えられた行列について、それぞれ、要素の最小値と最大値を戻します。

```
A = [1 2 3, 4 5 6, 7 8 9, 10 11 12];
B = [0 1 2, 2 1 0, 0 1 1, 2 0 0];
Min( A ); // 結果は 1
```

```
Max( A ); // 結果は 12
Min( A, B ); // 結果は 0
```

数値演算子

行列に対して、数値の演算（引き算、足し算、掛け算など）を実行できます。統計的手法の多くは、簡潔な行列表記で表現し、JSL に組み込むことができます。

たとえば次の式は、最小 2 乗回帰を行列の乗算と逆行列で表したものです。

$$b = (X'X)^{-1}X'y$$

この式を、次の JSL 式に組み込みます。

```
b = Inv( X`*X )*X`*y;
```

数値計算の基本

行列に対して、次の基本的な算術を実行できます。

- 足し算
- 引き算
- 掛け算
- 割り算（逆数を掛ける）

メモ: 標準的な乗算は、行列の掛け算であり、要素ごとの掛け算ではないことに注意してください。

行列の掛け算を実行するには、次のいずれかの方法を使用します。

- * 演算子
- `Multiply()` 関数
- `Matrix Mult()` 関数

行列の割り算を実行するには、次のいずれかの方法を使用します。

- / 演算子
- `Divide()` 関数

行列の掛け算および割り算では、次のことに注意してください。

- スカラーの掛け算または割り算は可換（ab = ba）ですが、行列の掛け算または割り算は**可換ではない**ことに注意してください。
- 2つの要素のうちの1つがスカラーの場合、要素ごとの掛け算または割り算が実行されます。
- 要素ごとの掛け算をする場合、`:*` または `EMult()` 関数を使用します。
- 要素ごとの割り算をする場合、`:/` か、または同じ作用を持つ `EDiv()` 関数を使用します。

例

```
A = [1 2 3, 4 5 6, 7 8 9, 10 11 12];
B = [0 1 2, 2 1 0, 0 1 1, 2 0 0];
C = [1 2 3 4, 4 3 2 1, 0 1 0 1];
D = [0 1 2, 2 1 0, 1 2 0];
```

行列の足し算

```
R = A + B;
  [1 3 5,
  6 6 6,
  7 9 10,
  12 11 12]
```

行列の引き算

```
R = A - B;
  [1 1 1,
  2 4 6,
  7 7 8,
  8 11 12]
```

行列の掛け算（**A**の行と**C**の列の内積）

```
R = A * C;
  [9 11 7 9,
  24 29 22 27,
  39 47 37 45,
  54 65 52 63]
```

行列の割り算（A*Inverse(D)と等価）

```
R = A / D;
  [1.5 0.5 0,
  3 2 0,
  4.5 3.5 0,
  6 5 0]
```

行列の要素ごとの掛け算

```
R = A :* B;
  [0 2 6,
  8 5 0,
  0 8 9,
  20 0 0]
```

行列とスカラーの掛け算

```
R = C * 2;
  [2 4 6 8,
  8 6 4 2,
  0 2 0 2]
```

行列のスカラーによる割り算

```
R = C / 2;
  [0.5 1 1.5 2,
  2 1.5 1 0.5,
  0 0.5 0 0.5]
```

行列の要素ごとの割り算（ゼロで割った場合は欠測値を戻す）

```
R = A :/ B;
  [.2 1.5,
  2 5 .,
  .8 9,
  5 ..]
```

行列に対する数値（スカラー）関数の使用

数値関数は、行列の各要素に対して作用します。純粋な数値関数の多くは行列にも適用でき、結果は行列として得られます。行列とスカラーを混在させてもかまいません。

数値関数の例には次のようなものがあります。

- Sqrt()、Root()、Log()、Exp()、^ Power()、Log10()
- Abs()、Mod()、Floor()、Ceiling()、Round()、Modulo()
- Sine、()Cosine()、Tangent()、ArcSine()、およびその他の三角関数
- Normal Distribution()、および他の確率関係の関数

例

```
A = [1 2 3, 4 5 6, 7 8 9, 10 11 12];
B = Sqrt( A ); // 各要素ごとに平方根をとる
  [1 1.414213562373095 1.732050807568877,
    2 2.23606797749979 2.449489742783178,
    2.645751311064591 2.82842712474619 3,
    3.16227766016838 3.3166247903554 3.464101615137754]
```

結合

Concat() 関数は、2つの行列を横に結合し、1つの行列にします。2つの行列の行の数は一致している必要があります。縦の2重バー（||）は、Concat と同じ機能を持つ二項演算子です。

```
Identity( 2 ) || J( 2, 3, 4 );
  [1 0 4 4 4, 0 1 4 4 4]

B = [1, 1];
B || Concat( Identity( 2 ), J( 2, 3, 4 ) );
  [1 1 0 4 4 4, 1 0 1 4 4 4]
```

VConcat() 関数は、2つの行列を縦に結合し、1つの行列にします。2つの行列の列の数は一致している必要があります。縦のバーとスラッシュの組み合わせ（|/）は、VConcatと同じ機能を持つ二項演算子です。

```
Identity( 2 ) |/ J( 3, 2, 1 );
// または VConcat( Identity( 2 ),J( 3, 2, 1 ) );
  [1 0, 0 1, 1 1, 1 1, 1 1]
```

Concat() と VConcat() はともに、空行列やスカラー、リストとの結合ができます。

```
a=[];
a || [1]; // 生成される行列は [1]
a || {2}; // 生成される行列は [2]
a || [3 4 5]; // 生成される行列は [3 4 5]
```

結合のための代入演算子として、||=と|/=があります。それぞれ、Concat To() 関数およびV Concat To() 関数と等価です。

- a || = bはa = a || bと等価です。
- a |/ = bはa = a | /bと等価です。

Transpose（転置）

Transpose() 関数は、行列の行と列を入れ換えます。逆引用符（`）は、Transpose() と同じ作用をする接尾演算子です。行列表記では、Transpose() は、一般的にプライム符号や肩文字のTを使って表現されるもの（A′ または A^T）と同じです。

```
A = [1 2 3, 4 5 6, 7 8 9, 10 11 12];
A`;
  [1 4 7 10,
  2 5 8 11,
  3 6 9 12]
Transpose( [1 2, 3 4] );
  [1 3, 2 4]
```

別の行列またはリストを使った行列またはリストのインデックス化

別の行列（またはリスト）を使った行列（またはリスト）のインデックス化は、スクリプトを高速にし、また記述を短くする上でとても便利な手法です。JMP 13以降では、あるエッジケースの扱いが以前のバージョンと異なります。

ある行列をデータ列のインデックスとして使用する以下の例を見てみましょう。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
tall rows = Loc( :Name("身長（インチ）") << Get Values() >= 67 );
  // [25, 27, 30, 37, 39, 40]
:Name("身長（インチ）")[tall rows]; // [68, 69, 67, 68, 68, 70]
N Rows( :Name("身長（インチ）")[tall rows] ); // 6
```

ここで重要なのは2行目です。:Name("身長(インチ)") << Get Values()は「**身長(インチ)**」列の値を行列として戻します。比較の>= 67は、式が偽（0）または真（1）のどちらであるかに対応した、0と1の行列を戻します。Loc()関数は前の行列内における1の場所を戻します。

この例では、「身長(インチ)」列が67以上である行番号として[25, 27, 30, 37, 39, 40]が得られます。この結果だけでも十分に便利なのですが、この例の重要なポイントは3行目と4行目にあります。ご覧のとおり、得られた行列を列または別の行列のインデックスとして用いることが可能です。

この手法の別の方法として、For()ループを書いてすべての行を明示的に反復することもできます。どちらの手法でもJMPは行をループしますが、JSLを使わずに内部で演算を行う場合には、ループを使う手法の方がかなり高速です。ただし、JSLの方が明示的なループを使わないため、簡潔に記述できます。

データ列参照の代わりに行列変数を使っても同じことが行えます。

```
m = :Name("身長(インチ)") << Get Values();
tall rows = Loc( m >= 67 ); // [25, 27, 30, 37, 39, 40]
m[tall rows]; // [68, 69, 67, 68, 68, 70]
N Rows( m[tall rows] ); // 6
```

同じ例で、今度は身長の基準を変えてみましょう（67の代わりに70）。

```
tall rows = Loc( :Name("身長(インチ)") << Get Values() >= 70 ); // [40]
:Name("身長(インチ)")[tall rows]; // [70]
N Rows( :Name("身長(インチ)")[tall rows] ); // 1

m = :Name("身長(インチ)") << Get Values();
tall rows = Loc( m >= 70 ); // [40]
m[tall rows]; // 70
N Rows( m[tall rows] ); // エラー
```

このように、data[tall rows]は[70]ではなく70を戻していました。多くの場合、1x1行列と数値は同じように扱われますが、必ずしも同じというわけではありません。その一例としてN Rows()関数が挙げられます。インデックス化で1x1行列が1つの数値に簡素化されるとこのようなことが起こり得ます。

N Rows()を使用する前にIs Matrix()を呼び出す方法が対応策の1つです。
別の方法として、空の行列を結果に連結することもできます。そうすれば、ソースが数値と行列のどちらであっても行列が作成されます。

```
m = m |/ []
```

JMP 13以降では1x1行列が維持されるため、インデックス自体が行列であれば、インデックス演算の結果は行列になります。リストのインデックス化にも同じ原理が適用されます。データテーブル列のインデックス化は以前からこの原理に従っていました。

行列とデータテーブル

JMP のデータテーブルと行列との間で情報を移動させることができます。行列代数を使って、JMP データテーブルにある数値に対して独自の計算を実行し、その結果を JMP データテーブルに戻して保存することができます。

データテーブルから行列へデータを移動させる

ここでは、データテーブルのデータを行列に移動する方法について説明します。

すべての数値を移動させる

Get As Matrix() 関数は、データテーブルまたは列のすべての数値を含む行列を生成します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
A = dt << Get As Matrix;
  [12 59 95,
  12 61 123,
  12 55 74, ...]
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
col = Column( "身長 ( インチ )" );
A = col << Get As Matrix;
  [59, 61, 55, 66, 52, ...]
```

すべての数値および文字列を移動させる

Get All Columns As Matrix() 関数は、文字列を含め、データテーブルのすべての列の値を行列で戻します。文字型の列には、名前の昇順に 1 から順に番号が付けられます。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
A = dt << Get All Columns As Matrix;
  [21 12 1 59 95,
  28 12 1 61 123, ...]
```

特定の列のみを移動させる

データテーブルの特定の列だけを取得するには、列リスト引数（名前または文字）を使用します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
x = dt << Get As Matrix( {"身長 ( インチ )", "体重 ( ポンド )"} );
```

または

```
x = dt << Get As Matrix( {Name("身長 ( インチ )"), Name("体重 ( ポンド )")} );
  [59 95, 61 123, 55 74, ...]
```

現在選択されている行

データテーブルで現在選択されている行の行列を求めるには：

```
dt << Get Selected Rows;
```

メモ: どの行も選択されていない場合は、空の行列が戻されます。

行の位置を検索する

式が真となる行番号の行列を求めるには:

```
dt << Get Rows Where( expression );
```

たとえば、次のスクリプトは性別が男性（M）である行番号を戻します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
A = dt << Get Rows Where( 性別 == "M" );
  [6, 7, 8, 12, 13, 14, ...]
```

行列のデータをデータテーブルに移動させる

ここでは、行列のデータをデータテーブルに移動させる方法について説明します。

列ベクトルを移動させる

Set Values() 関数は、列ベクトルの値を既存のデータテーブル列にコピーします。

```
col << Set Values( x );
```

col はデータテーブル列への参照で、**x** は列ベクトルです。

たとえば、次のスクリプトは、**test** という新しい列を作成し、ベクトル **x** の値を **test** 列にコピーします。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << New Column( "test" );
col = Column( "test" );
x = 1::40;
col << Set Values( x );
```

行列のすべての値を移動させる

Set Matrix() は、既存のデータテーブルに、行列の値を格納するのに必要なだけの新しい行と列を作成して、行列の値をコピーします。新しい列には、「**列1**」、「**列2**」という名前が付けられます。

```
dt = New Table( "B" );
dt << Set Matrix([1 2 3 4 5, 6 7 8 9 10]);
```

このスクリプトは、2つの行と5つの列を含んだBという新しいデータテーブルを作成します。

行列引数から新しいデータテーブルを作成するには、As Table(matrix) コマンドを使用します。列には、「**列1**」、「**列2**」という名前が付きます。たとえば、次のスクリプトは、**A** の値を含んだ新しいデータテーブルを作成します。

```
A = [1 2 3, 4 5 6, 7 8 9, 10 11 12];
dt = As Table( A );
```

列の要約

行列における各列の要約統計量を、行ベクトルで戻す関数がいくつか用意されています。

```
mymatrix = [11 22, 33 44, 55 66];
V Max( mymatrix ); // 各列の最大値を戻す
  [55 66]
V Min( mymatrix ); // 各列の最小値を戻す
  [11 22]
V Mean( mymatrix ); // 各列の平均値を戻す
  [33 44]
V Sum( mymatrix ); // 各列の合計を戻す
  [99 132]
V Std( mymatrix ); // 各列の標準偏差を戻す
  [22 22]
```

行列とレポート

レポートから値を取り出して行列を作成できます。まず、取得する項目の位置を指定する必要があります。この情報はレポートのツリー構造の中です。

次のスクリプトを実行して、「二変量の関係」レポートにパラメータ推定値のテーブルを作成します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" ) ;
biv = dt << Bivariate( X( :Name("身長(インチ)") ), Y( :Name("体重(ポンド)") ), Fit
  Line );
```

次に、ツリー構造を開いて、パラメータ推定値を含んでいる項目を識別します。

- グレーの開閉アイコンを右クリックし、[**編集**] > [**ツリー構造の表示**] を選択します。

  パラメータ推定値は NumberColBox(13) に含まれています。次のようにスクリプトを続けます。

  ```
  colBox = Report( biv )[Number Col Box( 13 )];
  beta = colBox << Get As Matrix;
    [-127.145248610915, 3.711354893859555]
  ```

次の点を念頭に置いてください。

- 変数にテーブルボックスへの参照が含まれている場合、Get As Matrix() は、テーブル内の全数値列の値を使って行列 **A** を作成します。

  ```
  A = tableBox << Get As Matrix;
  ```

- 変数にレポートテーブルの数値列への参照が含まれている場合、Get As Matrix() は、列の値を使って列ベクトルの行列 **A** を作成します。

  ```
  A = colBox << Get As Matrix;
  ```

Loc 関数

Loc()、Loc Nonmissing()、Loc Min()、Loc Max()、およびLoc Sorted()関数はすべて、行列内の特定の値の位置を示す行列を戻します。

Loc()

Loc()関数は、行列Aにおいて要素が「真」となっている位置を示す行列を作成します。つまり、行列**A**において、「0以外かつ欠測値以外」である要素の位置を示す行列を作成します。

```
A = [0 1 . 3 4 0];
B = [2 0 0 2 5 6];
```

次の例は、**A**において、0以外かつ欠測値以外である要素のインデックスを戻します。

```
I = Loc( A );
  [2, 4, 5]
```

次の例は、**B**の対応する要素よりも小さい**A**の要素のインデックスを戻します。2つの行列の行と列は同数でなければならないことに注意してください。

```
I = Loc( A < B );
  [1, 5, 6]
```

次のスクリプトは、**A**の4より小さい要素をすべて0に置き換えます。

```
A = [0 1 0 3 4 0];
A[Loc( A < 4 )] = 0;
Show( A );
  A = [0 0 0 0 4 0];
```

次のスクリプトは空の行列を戻します。

```
Loc( [2 3 4 5 6] > 7 );
  [](0, 1)
```

Loc Nonmissing()

Loc Nonmissing()関数は、欠測値を含まない行列の行番号のベクトルを戻します。例:

```
A = [1 2 3, 4 . 6, 7 8 ., 8 7 6];
Loc Nonmissing( A );
  [1, 4]
```

Loc Min()とLoc Max()

Loc Min()関数とLoc Max()関数は、それぞれ行列の最初の最小要素および最大要素の位置を戻します。行列の要素には、先頭行の最初の列から始め、左から右に向かって連続して番号が付けられます。

```
A = [1 2 2, 2 4 4, 1 1 1];
B = [6, 12, 9];
Show( Loc Max( A ) );
```

```
Show( Loc Min( B ) );
  Loc Max(A) = 5;
  Loc Min(B) = 1;
```

Loc Sorted()

Loc Sorted() 関数は、主に、ある数値がどのくらいの位置にあるかを調べるために使用します。この関数は、行列 **B** で与えられた値以下である値の中で最大のものが、**A** のどの位置にあるのかを戻します。結果のベクトルには、**B** の各要素に対応する項目が含まれます。

```
A = [10 20 30 40];
B = [35];
Loc Sorted( A, B );
  [3]
A = [10 20 40];
B = [35 5 45 20];
Loc Sorted( A, B );
  [2, 1, 3, 2]
```

次の点を念頭に置いてください。

- **A** は昇順に並んでいる必要があります。
- 戻り値は必ず 1 以上の整数です。**B** の要素が **A** のどの要素よりも小さい場合、関数は 1 の値を戻します。**B** の要素が **A** のどの要素よりも大きい場合、関数は *n* を戻します。*n* は **A** の要素の数です。

順位づけと並べ替え

Rank() 関数は、ベクトルまたはリスト内で昇順に並べた場合の各数値の位置を戻します。

```
E = [1 -2 3 -4 0 5 1 8 -7];
R = Rank( E );
  [9, 4, 2, 5, 7, 1, 3, 6, 8]
```

E を小さい値から順に並べ替えた場合、最初の数値は -7 です。**E** における -7 の位置は 9 番目です。

行列 **R** を **E** の添え字として利用すれば、元の行列 **E** を昇順に並べ替えることができます。

```
sortedE = E[R];
  [-7,- 4, -2, 0, 1, 1, 3, 5, 8]
```

Ranking Tie() 関数は、ベクトルまたはリスト内での値の順位を戻します。同順位のデータに対しては平均順位が与えられます。同様に、Ranking() も、ベクトルまたはリスト内での値の順位を戻しますが、同順位のデータに対しては任意の順序が与えられます。

```
E = [1 -2 3 -4 0 5 1 8 -7];
Ranking Tie( E );
  [5.5, 3, 7, 2, 4, 8, 5.5, 9, 1]

E = [1 -2 3 -4 0 5 1 8 -7];
Ranking( E );
  [5, 3, 7, 2, 4, 8, 6, 9, 1]
```

Sort Ascending() 関数および Sort Descending() 関数はベクトルを並べ替えます。

```
E = [1 -2 3 -4 0 5 1 8 -7];
Sort Ascending( E );
  [-7 -4 -2 0 1 1 3 5 8]

E = [1 -2 3 -4 0 5 1 8 -7];
Sort Descending( E );
  [8 5 3 1 1 0 -2 -4 -7]
```

引数がベクトルかリストではない場合、エラーメッセージが表示されます。

特殊な行列

単位行列の作成

Identity() 関数を使うと、任意の次元で単位行列を作成できます。単位行列とは、対角線上の要素が1で、それ以外の要素が0の正方行列です。引数には次元数のみを指定します。

```
Identity( 3 );
  [1 0 0,
  0 1 0,
  0 0 1]
```

指定の値を持つ行列の作成

関数 J() は、第1引数を行数、第2引数を列数、第3引数を全要素の値とする行列を作成します。

```
J( 3, 4, 5 );
  [5 5 5 5,
  5 5 5 5,
  5 5 5 5]
J( 3, 4, Random Normal() ); // 結果は異なる
  [0.407709113182904 1.67359154091978 1.00412665221308 0.240885679837327,
  -0.557848036549455 -0.620833861982722 0.877166783247633 1.50413740148892,
  -2.09920574748608 -0.154797501010655 0.0463943433032137 0.064041826393316]
```

対角行列の作成

Diag() 関数は、（同数の行と列を持つ）正方行列やベクトルから対角行列を作成します。対角行列とは、非対角要素がすべて0の正方行列です。

```
D = [1 -1 1];
Diag( D );
  [1 0 0,
  0 -1 0,
  0 0 1]
Diag([1, 2, 3, 4]);
  [1 0 0 0,
  0 2 0 0,
```

```
  0 0 3 0,
  0 0 0 4]

A = [1 2,3 4];
f = [5];
D = Diag( A, f );
  [1 2 0
  3 4 0
  0 0 5]
```

三つ目の例では、一見すると、すべての非対角要素がゼロというわけではありません。行列表記を使用すると、次のようになります。

```
[A 0,
0` f]
```

ここで、Aとfは例の行列で、0はゼロの列ベクトルです。

対角要素からの列ベクトルの作成

VecDiag()関数は、行列の対角要素から成るベクトルを作成します。

```
v = Vec Diag(
[1 0 0 1, 5 3 7 1, 9 9 8 8, 1 5 4 3]);
  [1, 3, 8, 3]
```

2次形式の計算

Vec Quadratic()関数は、回帰分析における予測の標準誤差や、外れ値に関するMahalanobis距離やT2乗統計量などを計算します。Vec Quadratic(Sym, X)は、Vec Diag(X*Sym*X`)を計算するのと同じです。第1引数は対称行列でなければなりません。通常は、共分散行列の逆行列を指定します。第2引数は、列数が第1引数の行列と等しい矩形行列です。

対角要素の合計を戻す

Trace()関数は、正方行列の対角要素の和を戻します。

```
D = [0 1 2, 2 1 0, 1 2 0];
Trace( D ); // 1を戻す
```

整数値の行ベクトルの作成

Index()関数は、最初の引数から最後の引数までの整数値を要素とする行ベクトルを作成します。2重コロン::は等価の二項演算子です。

```
6::10;
  [6 7 8 9 10]
Index( 1, 5 );
  [1 2 3 4 5]
```

オプションの第3引数を指定して、増分をデフォルト値の+1から変更することもできます。

```
Index( 0.1, 0.4, 0.1 );
  [0.1, 0.2, 0.3, 0.4]
```

増分は負の数でもかまいません。

```
Index( 6, 0, -2 );
  [6, 4, 2, 0]
```

デフォルトの増分は1、または、最初の引数が2番目の引数よりも大きい場合は-1です。

行列の再構築

Shape()関数は、既存の行列を指定の次元に作り直します。たとえば次のように指定すると、3x4の行列であるaが12x1の行列に変わります。

```
a = [1 1 1, 2 2 2, 3 3 3, 4 4 4];
Shape( a, 12, 1 );
  [1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4]
```

計画行列の作成

Design()関数は、ベクトルまたはリストの計画行列を作成します。引数の一意の値ごとに1つの列を作成します。計画行列の要素は0または1です。たとえば、下の**x**は、1、2、3の値から成っているため、計画行列には1の列、2の列、そして3の列があることになります。行列の各行には、行の値に対応する列に1が入ります。次の例では、第1行（1）に対しては、1の列（第1列）に1が入り、他の列には0が入ります。第2行（2）に対しては、2の列に1が入り、他の列には0が入ります。以下、同じような操作で行列が作成されます。

```
x = [1, 2, 3, 2, 1];
Design( x );
  [1 0 0,
  0 1 0,
  0 0 1,
  0 1 0,
  1 0 0]
```

この変形としてDesign Nom()関数またはDesign F()関数があり、これは最後の列を削除して、それを他の列の値から引きます。したがって、Design Nom()またはDesign F()でできる行列の要素は、0、1、または–1となります。また、Design Nom()またはDesign Fでできた行列の列の数は、元のベクトルが持つ要素の種類の数より1つ少なくなります。()この演算子は、効果のフルランクの計画行列を作成します。

```
x = [1, 2, 3, 2, 1];
Design Nom( x );
  [1 0, 0 1, -1 -1, 0 1, 1 0]
```

Design Nom()については、「分散分析（ANOVA）の例」（198ページ）で詳しく説明しています。

順序尺度の因子を簡単にコーディングするには、Design Ord()関数を使います。この関数は、ベクトルまたはリストの一意の値の数より1つ少ない列を使ってフルランクのコーディングを作成します。低水準の行をすべてゼロにし、それに続く水準は計画行列の行にひとつずつ1を加えていきます。

```
x = [1, 2, 3, 4, 5, 6];
Design Ord( x );
  [0 0 0 0 0,
  1 0 0 0 0,
  1 1 0 0 0,
  1 1 1 0 0,
  1 1 1 1 0,
  1 1 1 1 1]
```

Design()、Design Nom()、およびDesign Ord()では、全水準の値とその順序を指定する第2引数をとることができます。この機能により、たった1行に対する計画行列も作成することができます。

- Design(values, levels)は計画行列を作成します。
- Design Nom(values, levels)はフルランクの計画行列を作成します。

次の点を念頭に置いてください。

- *values*引数は、単一の要素でも、要素の行列またはリストでもかまいません。
- *levels*引数は、計画行列作成の対象となる全水準を値として持つリストまたは行列です。
- 作成される行列は、*values*引数として指定した値の要素数と同じ行数になります。
- 作成される行列は、*levels*引数として指定した項目数と同じ列数になります。Design Nom()とDesign Ord()の場合は、*levels*引数として指定した項目数より1つ少ない列数になります。
- 値が見つからない場合は、行全体がゼロになります。

例

```
Design( 20, [10 20 30] );
  [0 1 0]
Design( 30, [10 20 30] );
  [0 0 1]
Design Nom( 20, [10 20 30] );
  [0 1]
Design Nom( 30, [10 20 30] );
  [-1 -1]
Design Ord( 20, [10 20 30] );
  [1 0]
Design( [20, 10, 30, 20], [10 20 30] );
  [0 1 0,
  1 0 0,
  0 0 1,
  0 1 0]
Design({"b", "a", "c", "b"}, {"a", "b", "c"});
  [0 1 0,
  1 0 0,
  0 0 1,
  0 1 0]
```

直積を求める

Direct Product() 関数は、2つの行列の直積（Kronecker 積）を求めます。行列と行列の直積は行列となり、その要素は、**A** と **B** の要素の積となります。

```
G = [1 2, 3 5];
H = [2 3, 5 7];
Direct Product( G, H );
  [2 3 4 6,
  5 7 10 14,
  6 9 10 15,
  15 21 25 35]
```

H Direct Product() 関数は、行数の等しい2つの行列の行ごとの直積を求めます。

```
H Direct Product( G, H );
  [2 3 4 6, 15 21 25 35]
```

HDirect Product() は、計画行列における交互作用の列を作る際に便利です。

```
XA = Design Nom( A );
 XB = Design Nom( B );
 XAB = HDirect Product( XA, XB );
 X = J( NRow( A ), 1 ) || XA || XB || XAB;
```

逆行列と連立一次方程式

JMP には、Inverse()、GInverse()、および Sweep() という、逆行列を計算するための関数があります。Solve() 関数は、連立一次方程式の式を解くのに使用します。

Inverse または Inv

Inverse() 関数は、正則な正方行列の逆行列を戻します。Inverse() は、省略して Inv と書くこともできます。行列 **A** において、**A** と Inverse(**A**) の積（よく $A(A^{-1})$ と記載される）は、結果として単位行列を戻します。

```
A = [5 6, 7 8];
AInv = Inv( A );
A*A Inv;
  [1 0,0 1]

A = [1 2, 3 4];
AInv = Inverse( A );
A*AInv;
  [1 1.110223025e-16,0 1]
```

メモ: 2番目の例にあるように、浮動小数点の精度の限界が原因で、値がわずかに異なる場合があります。

GInverse

行列 **A** の（Moore-Penrose型の）一般逆行列とは、次のような行列 **G** です。

AGA=A

GAG=G

(AG)´ = AG

(GA)´ = GA

`GInverse()` 関数は、非正方行列を含めた任意の行列を引数にとり、特異値分解を使ってMoore-Penrose型の一般逆行列を求めます。この関数は、フルランクでない行列の逆行列を求めるときに便利です。次の方程式系を見てください。

$$\begin{aligned} x + 2y + 2z &= 6 \\ 2x + 4y + 4z &= 12 \\ x + y + z &= 1 \end{aligned}$$

この方程式系の解を求めるには、次のスクリプトを使用します。

```
A = [1 2 2, 2 4 4, 1 1 1];
B = [6, 12, 1];
Show( GInverse( A ) * B );
  GInverse(A) * B = [-4, 2.5, 2.5];
```

Solve

`Solve()` 関数は、連立一次方程式の解を求めます。`Solve()` は、$x = A^{-1}b$ となるベクトル **x** を求めます（ここで、A は正方行列、b はベクトル）。この行列 **A** とベクトル **b** の行数は同じである必要があります。`Solve(A,b)` は `Inverse(A)*b` と同じです。

```
A = [1 -4 2, 3 3 2, 0 4 -1];
b = [1, 2, 1];
x = Solve( A, b );
  [-16.9999999999999, 4.99999999999998, 18.9999999999999]
A*x;
  [1, 2, 0.999999999999997]
```

メモ: 例にあるように、浮動小数点の精度の限界が原因で、値がわずかに異なる場合があります。

Sweep

Sweep() 関数は、正方行列の部分（**ピボット**）の逆行列を求めます。これをすべてのピボットに対して行うと、最終的には逆行列が求められます。通常、この行列は対角軸が0にならないよう、正定値行列（または負定値行列）である必要があります。Sweep() は、与えられた行列が正定値行列であるかどうかはチェックしません。正定値行列で**ない**場合も、掃き出し法において、対角軸に0以外の値が現れる限り演算は続けられます。0（または0に近い値）が対角軸に現れた場合、その行と列を0にします。結果として g2 型の一般逆行列が求められます。

Sweep 関数について

行列 **E** が、次に示すように、さらに小さな行列 **A**、**B**、**C**、**D** に分割されているとします。

$$E = \begin{bmatrix} A & B \\ C & D \end{bmatrix}$$

Sweep() の構文は次のようになります。

```
Sweep( E, [...] );
```

[...] は行列 A の部分を示します。

これにより、次のような結果の行列が作成されます。

$$\begin{bmatrix} A^{-1} & A^{-1}B \\ -CA^{-1} & D - CA^{-1}B \end{bmatrix}$$

次の点を念頭に置いてください。

- **A** の位置の部分行列は逆行列になります。
- 部分行列 **B** の位置は $Ax = B$ の解になります。
- 部分行列 **C** の位置は $xA = C$ の解になります。

Sweep 関数の使用法

Sweep() は逐次的で可逆的です。

- `A = Sweep( A, {i,j} )` は `A = Sweep( Sweep( A, i ), j )` と同じで、逐次的です。
- `A = Sweep( Sweep( A, i ), i )` は **A** を元の値に復元します。これは可逆的です。

次のような交差積の部分から成る行列があるとします。

$$C = \begin{bmatrix} X'X & X'y \\ y'X & y'y \end{bmatrix}$$

X′X を引数とした Sweep 演算の結果は、次のようになります。

$$\begin{bmatrix} (X'X)^{-1} & (X'X)^{-1}X'y \\ -yX(X'X)^{-1} & y'y-y'X(X'X)^{-1}X'y \end{bmatrix}$$

統計的な観点からは、次のように見ることができます。

- 右上部はモデル Y = Xb + e の最小2乗推定値
- 右下部は誤差の平方和
- 左上部は推定値の共分散に比例する行列

Sweep関数は、ステップワイズ回帰で必要な部分的な解を求める際に有効です。

2つ目の引数は、掃き出しを行う行（列）を列挙したベクトルです。たとえば、**E** が 4x4 の行列で、4行すべてに掃き出し法を適用して、$\mathbf{E}^{-1}$ を得るには、以下のように指定します。

```
E = [ 5 4 1 1, 4 5 1 1, 1 1 4 2, 1 1 2 4];
Sweep( E, [1, 2, 3, 4] );
  [0.56 -0.44 -0.02 -0.02,
  -0.44 0.56 -0.02 -0.02,
  -0.02 -0.02 0.34 -0.16,
  -0.02 -0.02 -0.16 0.34]
Inverse( E ); // これらの結果は同じ
  [0.56 -0.44 -0.02 -0.02,
  -0.44 0.56 -0.02 -0.02,
  -0.02 -0.02 0.34 -0.16,
  -0.02 -0.02 -0.16 0.34]
```

メモ： Sweep() の詳細と、逆行列を作成する Gauss-Jordan 法との関連については、Goodnight, J.H. (1979) “A tutorial on the SWEEP operator.” **The American Statistician**, August 1979, Vol. 33, No. 3. pp. 149-58 を参照してください。

Sweep() については、「分散分析（ANOVA）の例」（198ページ）でさらに説明しています。

行列式（Determinant）

Det() 関数は、正方行列の行列式を戻します。2 x 2 行列の行列式は、下に示すように、対角線上の要素の積の差です。*n* x *n* 行列の行列式は、(*n* - 1) × (*n* - 1) の行列式の重み和として再帰的に定義されます。逆行列を作成するには、行列の行列式が 0 以外でなければなりません。

$$\begin{vmatrix} 1 & 2 \\ 3 & 5 \end{vmatrix} = (1 \cdot 5) - (3 \cdot 2) = -1$$

```
F = [1 2, 3 5];
Det( F );
  -1
```

分解と正規化

この節では、固有値および固有ベクトルを計算する関数と、行列を分解する関数について説明します。

固有値

Eigen() 関数は、対称行列の固有値分解を実行します。固有値分解は、主成分分析と正準相関分析をはじめとするさまざまな統計的手法で用いられています。主成分分析と正準相関分析は、最大の固有値に対応した変換が分散を最大化する変換となっています。

Eigen() は行列のリストを戻します。戻されたリストの最初の行列は固有値の列ベクトルで、2番目の行列には列として固有ベクトルが含まれています。

```
A = [ 5 4 1 1, 4 5 1 1, 1 1 4 2, 1 1 2 4];
Eigen( A );
  {[10, 5, 2, 1][0.632455532033676 - 0.316227766016838 - 2.77555756156289e-16
    -0.707106781186547, 0.632455532033676 - 0.316227766016837 -
    1.66533453693773e-16 0.707106781186547, 0.316227766016838
    0.632455532033676 0.707106781186548 0, 0.316227766016837 0.632455532033676
    - 0.707106781186547 0]}
```

この関数は行列のリストを結果として戻します。結果を受け取るには、一方には固有値のベクトル、もう一方には固有ベクトルの行列が代入されるように、2つのグローバル変数のリストに代入してください。

```
{evals, evecs} = Eigen( A );
```

固有値分解をすることで、$n \times n$ の行列 **A** に対する方程式 $Ax = \lambda x$ がゼロベクトルでない解 **x** を持つときの、すべての λ（ラムダ）とベクトル **x** が求められます。この λ を固有値、**x** ベクトルを固有ベクトルと呼びます。これは $(A - \lambda I)\mathbf{x} = 0$ を解くことと同じです。下のようなスクリプトで、固有値と固有ベクトルから **A** を再構成できます。

```
newA = evecs * Diag( evals ) * evecs`;
  [5.00000000000001 4 1 1,
  4 5 1 1,
  1 1 4 2,
  1 1 2 4]
```

固有値および固有ベクトルに関しては、次のことに注意してください。

- 固有ベクトルの行列は直交行列で、転置行列が逆行列になります（$E'E = EE' = I$）。
- 固有値が一意であるときのみ、固有ベクトルも一意に求められます。
- 固有値がゼロの行列は、特異行列です。
- 固有値の逆行列と固有ベクトルにより、逆行列も求められます。Moore-Penrose の一般化逆行列は、ゼロ以外の固有値の逆行列から求められます（「GInverse」（189ページ）を参照）。

> **メモ：**非常に小さい固有値を 0 とみなすかどうかを判断する必要があります。

- 固有値分解をすることで、正方行列の掛け算を次のように考えることができます。

– 回転（直交行列を掛ける）

– スケーリング（対角行列を掛ける）

– 逆回転（直交行列の逆行列、つまり転置行列を掛ける）

```
A * x = E` * Diag( M ) * E * x;
```

Eで回転、Diag(M)でスケーリング、E`で逆回転します。

Cholesky分解

Cholesky()関数はCholesky分解を行います。半正定値行列**A**を、非特異下三角行列**L**とその転置行列の積の形、**L*L'** = **A**に分解します。

```
E = [ 5 4 1 1, 4 5 1 1, 1 1 4 2, 1 1 2 4];
L = Cholesky( E );
  [2.23606797749979 0 0 0,
  1.788854381999832 1.341640786499874 0 0,
  0.447213595499958 0.149071198499986 1.9436506316151 0,
  0.447213595499958 0.149071198499986 0.914659120760047 1.71498585142509]
```

結果を確認するには、次のように入力します。

```
L*L`;
  [5 4 1 1,
  4 5 1 1,
  1 1 4 2,
  1 1 2 4]
```

Cholesky分解について

Cholesky()は、行列の式を扱いやすい形式に再構成するのに便利です。たとえば、JMPでは固有値を求めるには、行列が対称行列である必要があります。したがって、対称行列の積**AB**の固有値は直接求められない場合があります（**A**と**B**が対称行列であっても、その積が対称行列であるとは限らないからです）。しかし、この問題は、**L'BL**の固有値の問題に置き換えることができます。ここで、**L**は**A**のCholesky根です。L`BLはABと同じ固有値を持ちます。

Cholesky()の他の使用法としては、Trace()式で行列の対角和を求めるときに、行列の順序を変えるということが考えられます。Trace(A*B*A`)などの式で、**A**の行数が多い場合、大きな計算量を要することになります。しかし、**B**がLL'とCholesky分解できるのであれば、元の式はTrace(A*L*L`*A`)と書き直せます。これは、Trace(L`A`*AL)と等しく、したがって**AL**の要素の平方和を求めるだけでよくなるので、演算量はかなり少なくなります。

Chol Update()関数を使用すると、Cholesky分解を効率的に更新できます。*n* x *n*行列**A**のCholesky根を**L**とした場合、cholUpdate(L, C, V)を呼び出すと、A+V*C*V'のCholesky根が算出されます。**C**は*m* x *m*の対称行列、**V**は*n* x *m*行列です。

例

Cholesky分解を手動で更新する手順は次のとおりです。

```
exS = [16 1 0 11 -1 12, 1 11 -1 1 -1 1, 0 -1 12 -1 1 0, 11 1 -1 11 -1 9, -1 -1 1 -1
  9 -1, 12 1 0 9 -1 12];
exAchol = Cholesky( exS ); // Cholesky 分解を実行する

exV = [1 1, 0 0, 0 1, 0 0, 0 0, 0 1];
// 計画行列に 2 つの列ベクトルを追加する

exC = [1 0, 0 -1];
/* 最初の列ベクトルは、計画行列の行の 1 つに足される。
2 番目の列ベクトルは、計画行列の行の 1 つから引かれる */ */

exAnew = exS + exV * exC * exV`;
exAcholnew = Cholesky( exAnew );
// Cholesky 分解を手動で更新する
```

手動で更新するのではなく、Chol Update() を使って Cholesky 分解をより効率的に更新する手順は、次のとおりです。

```
exAcholnew_test = Chol Update( exAchol, exV, exC );
// Cholesky 分解をより効率的に更新する
Show( exAcholnew_test );
  exAcholnew_test =
  [ 4 0 0 0 0 0,
  0.25 3.30718913883074 0 0 0 0, ...]
// 結果は手動での手順の場合と同じ
Show( exAcholnew );
  exAcholnew=
  [ 4 0 0 0 0 0,
  0.25 3.30718913883074 0 0 0 0, ...]
```

特異値分解

SVD() 関数を使うと、行列の特異値分解が求められます。つまり、行列 **A** について、SVD() は U*diag(M)*V`=A を満たす 3 つの行列、**U**、**M**、**V** のリストを戻します。

次の点を念頭に置いてください。

- 結果の行列 **M** では、余計な 0 の対角要素は省かれています。
- 特異値分解によって、**A** は USV′ の形で表されます。ここで、
  - **U** と **V** は、互いに直交している列ベクトルを含んでいる行列です（「直交するベクトル」とは、「直角」もしくは「独立」なベクトルのことを指します）。
  - **S** は *n* x *n* 対角行列で、**A** の特異値、つまり **A′A** の固有値の負でない平方根を要素として持っています。
- 特異値分解は対応分析などで使われています。

例

```
A = [1 2 1 0, 2 3 0 1, 1 0 1 5, 0 1 5 1];
```

```
{U, M, V} = SVD( A );
newA = U * Diag( M ) * V`;
  [1 2 0.999999999999997 -2.99456640040496e-15,
  2 3 -1.17505831453979e-15 1,
  0.999999999999997 -2.16851276518826e-15 0.999999999999999 5,
  2.22586706011274e-15 1 5 0.999999999999997]
```

正規直交化

Ortho() 関数は、列の直交化を行い、さらにベクトルの大きさで割って正規化します。この関数はGram-Schmidt法を使用します。直交行列の列ベクトルは、大きさが単位長で互いに直交（内積が0）します。

```
B = Ortho( [1 -1, 1 0, 0 1] );
  [0.408248290463863 -0.707106781186548, 0.408248290463863 0.707106781186548,
    -0.816496580927726 3.14018491736755e-16]
```

各列ベクトルが直交していることは、**B** と **B** の転置行列を掛け合わせると単位行列になることから確かめられます。

```
C = B` * B;
  [1 -3.119061760824e-16, -3.119061760824e-16 1]
```

特に指定がない限り、ベクトルは正規化されます。つまり、ベクトルの大きさでベクトルそのものを割るスケーリングで、大きさ1のベクトルを作ります。Scaled(0) を指定して、スケーリングをしない場合は、次のようになります。

```
Ortho( [1 -1, 1 0, 0 1], Scaled( 0 ) );
  [0.408248290463863 -0.353553390593274, 0.408248290463863 0.353553390593274,
    -0.816496580927726 1.57009245868377e-16]
```

全要素の和が0になるベクトルを作るには、オプションの Centered(1) を指定します。このオプションは対比の行列を作る場合に便利です。

```
result = Ortho( [1 -1, 1 0, 0 1], Centered( 1 ) );
  [0.408248290463863 -0.707106781186548, 0.408248290463863 0.707106781186548,
    -0.816496580927726 3.14018491736755e-16]
```

各列の要素の和が0になることは、要素がすべて1のベクトルを前から掛けて列の要素の和を求めることで確認できます。

```
J(1, 3) * result;
  [1.11022302462516e-16 2.02996189274239e-16]
```

直交多項式

Ortho Poly() 関数は、引数で指定された次元までのベクトルの直交多項式 を戻します。直交多項式は、多項式モデルを普通にあてはめると回帰係数の間に強い相関が生じてしまう場合に有用です。

```
Ortho Poly( [1 2 3], 2 );
  [-0.707106781186548 0.408248290463862, 0 -0.816496580927726,
    0.707106781186548 0.408248290463864]
```

次元はベクトルの次元より小さいものである必要があります。「正規直交化」（195ページ）の項にあるように、Scaled(1)を指定することで単位長のベクトルを作れます。

QR分解

QR()分解は、数値的に安定した行列の処理に便利です。QR()は2つの行列のリストを戻します。以下はその例です。

```
QR( [11 22, 33 44] );
```

QとRに結果が入ります。*m* x *n*行列Xに対して、QR()はX = Q*Rと分解します。ここで、Qは、*m* x *m*の直交行列、Rは*m* x *n*の上三角行列です。

逆行列の更新

M`M行列の逆行列に行を追加または削除するには、Inv Update(S, X, w)関数を使用します。逆行列の更新は、1行除去による影響診断や候補計画の評価に役立ちます。

次の点を念頭に置いてください。

- 第1引数**S**は更新する行列です。
- 第2引数**X**は、追加または削除する行を含む行列です。
- 第3引数wには、行を追加する場合は1、削除する場合は-1を指定します。
- 複数の行を追加または削除できます。

Inv Update(S, X, w)関数は、次の式を計算するのと同じです。

```
S - w * S * X` * Inv( I + w * X * S * X` ) * X * S
```

ここで、**I**は単位行列、Inv(A)は**A**の逆行列です。

ユーザ定義の行列演算子の作成

ユーザ自身が定義した演算子をマクロに登録できます。詳細については、「プログラミング手法」章の「マクロ」（223ページ）を参照してください。同様に、カスタム行列演算子も作成できます。たとえばベクトルの大きさを求める行列演算子Mag()を定義するには、次のように記述します。

```
mag = Function( {x},
   Sqrt( x` * x )
);
```

同様に、大きさでベクトルを割るNormalizeを定義するには、次のように記述します。

```
normalize = Function( {x},
   x / Sqrt( x` * x )
);
```

統計処理の例

この節では、行列を使った統計処理の例を紹介します。

回帰の例

回帰分析の計算手法について、JMP に組み込まれている機能を使わず、ユーザ自身が独自の方法を実施したい場合を考えてみましょう。そのような場合、簡潔な行列表現を使って、ほんの数行で書くことができます。

```
Y = [98, 112.5, 84, 102.5, 102.5, 50.5, 90, 77, 112, 150, 128, 133, 85, 112];
X = [65.3, 69, 56.5, 62.8, 63.5, 51.3, 64.3, 56.3, 66.5, 72, 64.8, 67, 57.5, 66.5];
X = J( N Row( X ), 1 ) || X; // 切片の列に 1 を入れる
beta = Inv( X` * X ) * X` * Y; // 最小 2 乗推定
resid = Y - X * beta; // 残差、Y - 推定値
sse = resid` * resid; // 誤差平方和
Show( beta, sse );
```

これは、データテーブルからデータを取得し、関連する統計量を計算してレポートウィンドウに表示するスクリプトへと拡張できます。

```
// データテーブルを開く
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );

// データを行列に移す
x = (Column( "年齢" ) << Get Values ) || (Column( "身長 (インチ)" ) << Get Values);
x = J(N Row( x ), 1, 1) || x;
y = Column( "体重 (ポンド)" ) << Get Values;

// 回帰分析の計算
xpxi = Inv( x` * x );
beta = xpxi * x` * y; // パラメータ推定値
resid = y - x * beta; // 残差
sse = resid` * resid; // 誤差平方和
dfe = N Row( x ) - N Col( x );  // 自由度
mse = sse / dfe; // 誤差の平均平方、誤差分散推定値

// 推定値の追加計算
stdb = Sqrt( Vec Diag( xpxi ) * mse ); // 推定値の標準誤差
alpha = .05;
qt = Students t Quantile( 1 - alpha / 2, dfe );
betau95 = beta + qt * stdb; // 上側 95%信頼限界
betal95 = beta - qt * stdb; // 下側 95%信頼限界
tratio = beta :/ stdb; // Student の t 値
probt = ( 1 - TDistribution( Abs( tratio ), dfe ) ) * 2; // p 値

// 結果の表示
New Window( "ビッグ クラスの回帰分析",
   Table Box(
```

```
String Col Box( "項",{"切片", "年齢", "身長(インチ)"} ),
Number Col Box( "推定値", beta ),
Number Col Box( "標準誤差", stdb ),
Number Col Box( "t値", tratio ),
Number Col Box( "p値", probt ),
Number Col Box( "下側95%", beta195 ),
Number Col Box( "上側95%", betau95 ) ) );
```

分散分析（ANOVA）の例

行列演算を用いて一元配置分散分析も実行できます。この例では、3水準の要因（薬品の投与量で低用量、中用量、高用量）と1つの応答変数から成る問題を扱います。つまりこの例は、次のような一般線形モデルで表されます。

$$Y = a + bX + e$$

ここで

- **Y**は応答変数のベクトル
- *a*は切片項
- **b**は係数から成るベクトル
- **X**は要因の計画行列
- *e*は誤差項

```
factor = [1, 2, 3, 1, 2, 3, 1, 2, 3];
y = [1, 2, 3, 4, 3, 2, 5, 4, 3];
```

まず、要因の計画行列を作成する必要があります。

```
Design Nom( factor );
  [1 0, 0 1, -1 -1, 1 0, 0 1, -1 -1, 1 0, 0 1, -1 -1]
```

次に、切片項として、計画行列に1だけから成る列を加えます。これを行うには、単にJとDesign Nom()を結合するだけです。

```
x = J( 9, 1, 1) || Design Nom( factor );
  [1 1 0,
  1 0 1,
  1 -1 -1,
  1 1 0,
  1 0 1,
  1 -1 -1,
  1 1 0,
  1 0 1,
  1 -1 -1]
```

ここで、一般的な方程式を解くために、次のような行列 **M** を作成する必要があります。

$$\begin{bmatrix} X'X & X'y \\ y'X & y'y \end{bmatrix}$$

行列 **M** は、次のような結合をすれば1ステップで作成できます。

```
M = ( x` * x || x` * y )
        |/
  ( y` * x || y` * y );
  [9 0 0 27,
    0 6 3 2,
    0 3 6 1,
    27 2 1 93]
```

モデル全体の推定結果は、行列 **M** において、**X′X** の部分をすべて掃き出せば得られます。また、最初の列だけ掃き出せば切片のみのモデルの推定結果が得られます。

```
FullFit = Sweep( M, [1, 2, 3] ); // フルモデルのあてはめ
InterceptOnly = Sweep( M, [1] ); // 切片のみのモデル
```

分散分析の一部の結果は、これら2つのモデルを比較して計算されます。ここでは、フルモデル（切片と傾きを含んだモデル）の結果を見てみましょう。フルモデルに関して掃き出された行列は、次のような行列になります。

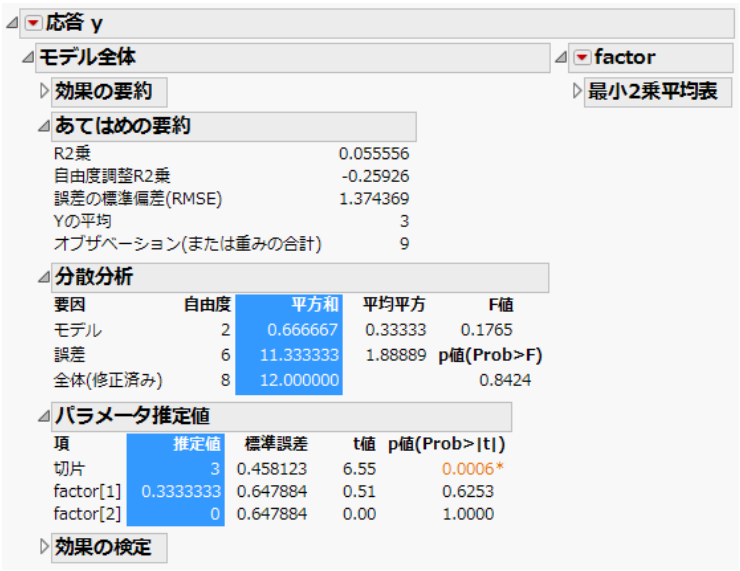
```
[0.111111111111111 0 0 3,
0 0.222222222222222 -0.11111111111111 0.333333333333333,
0 -0.11111111111111 0.222222222222222 0,
-3 -0.33333333333333 0 11.333333333333333]
```

行列右上部のモデル係数を見てください。左下部の係数も、符号が正負逆になっている以外は同じ、3, 0.333, 0となっています。結果は次のように解釈できます。

- 切片項の係数は3です。
- 要因の1番目の水準の係数は0.333です。
- 2番目の水準の係数は0です。
- `Design Nom()` を使っているので、3番目の水準の係数は-0.333です。
- 行列右下部には誤差平方和 11.333 が入ります。

このプログラム例では、特定の数値を指定していますが、それらを適切な引数に置き換えることにより、より汎用性があるスクリプトに変更できます。上の結果は、モデルのあてはめプラットフォームの結果と一致しています。詳細については、図7.1を参照してください。

図7.1 モデルのあてはめのANOVAレポート



以下に、図7.1に示すレポートを作成する方法を示します。

1. 「データテーブル」章（271ページ）で説明している方法で、データテーブルを作成します。

```
dt = New Table( "Data" );
dt << New Column( "y", Set Values( [1, 2, 3, 4, 3, 2, 5, 4, 3] ) );
dt << New Column( "factor", "Nominal", Values( [1, 2, 3, 1, 2, 3, 1, 2, 3] ) );
```

2. 「プラットフォームのスクリプト」章の「分析を起動し、そのスクリプトをウィンドウに保存する」（374ページ）で説明している方法で、モデルのあてはめを実行します。

```
obj = Fit Model(
   Y( :y ),
   Effects( :factor ),
   Personality( "Standard Least Squares" ),
   Run(
      y << {Plot Actual by Predicted( 0 ), Plot Residual by Predicted( 0 ),
      Plot Effect Leverage( 0 )}
   )
);
```

3. 「表示ツリー」章の「ディスプレイボックスオブジェクトの参照」（419 ページ）で説明している方法で、ディスプレイを操作します。

```
ranova = obj << Report;
ranova[Outline Box( 6 )] << Close( 0 );
ranova[Outline Box( 7 )] << Close( 1 );
ranova[Outline Box( 9 )] << Close( 1 );
```

```
ranova[Outline Box( 5 ), Number Col Box( 2 )] << Select;
ranova[Outline Box( 6 ), Number Col Box( 1 )] << Select;
```

この結果が図7.1に示されているものになります。

連想配列

連想配列は、値に固有のキーを関連付けます（固有でない場合もあります）。連想配列は、辞書、マップ、ハッシュマップ、またはハッシュテーブルと呼ばれることもあります。

キーについては、以下の点に留意してください。

- キーは引用符で囲みます。
- キーに関連付けられる値は、数値、日付、行列、リストなどです。
- キーを浮動小数点数にすることはできません。サポートされているのは整数のみです。
- 行列もリストも、キーと値の両方に使用できますが、行列とリストを混在させることはできません。つまり、行列をキーとして使用し、リストを値として使用することはできません。その逆も同様です。
- 通常、連想配列には順序がありませんが、JMPの連想配列では、反復処理や逐次処理のプログラミングのために、キーをアルファベット順（文字コード順）で戻します。

大量のリストの場合、連想配列を使った方がより効率的で高速です。

連想配列の作成

空の連想配列を作成するには、`Associative Array()`関数または`[=>]`を使用します。

```
cary = Associative Array();
cary = [=> ];
  [=> ]
```

キーと値には、任意のJSLオブジェクトが使用できます。項目は添え字を使った指定により、追加・変更できます。

```
cary = Associative Array();
cary["state"] = "NC";
cary["population"] = 116234;
cary["weather"] = "cloudy";
cary["population"] += 10;
cary["weather"] = "sunny";
cary["high schools"] = {"Cary", "Green Hope", "Panther Creek"};
```

デフォルト値

連想配列にないキーの値は、デフォルト値によって決定します。連想配列にないキーを参照しようとすると、エラーが生じます。連想配列にデフォルト値を定義した場合は、存在しないキーを参照すると、次のように動作します。

- そのキーを連想配列に追加する
- デフォルト値をその新しいキーに割り当てる
- エラーではなく、新しいキーの値（デフォルト値）を戻す

キーに値を割り当てないで文字列のリストから連想配列を作成した場合、キーには1という値が割り当てられます。この連想配列のデフォルト値は0に設定されます。

デフォルト値を設定するには、次のようにします。

```
cary = Associative Array();
cary << Set Default Value( "Cary, NC" );
```

連想配列にデフォルト値が設定されているかどうかを確かめるには、<<Get Default Valueメセージを使用します。

```
cary << Get Default Value;
  "Cary, NC"
```

デフォルト値がない場合は、Empty()が戻されます。

Set Default Valueメッセージ以外に、連想配列のリテラルを指定する際にキーなしの=>valueを使用することによってもデフォルト値を設定できます。

```
counts = ["a" => 10,
"b" => 3,
=> 0]; // デフォルト値は0
counts["c"] += 1;
Show( counts );
  counts = ["a" => 10, "b" => 3, "c" => 1, => 0];
```

1行目はデフォルト値を0に設定します。2行目はキー"c"がcounts内にないことを示します。結果として、デフォルト値0を持つキー"c"を作成し、1だけ増やします。

連想配列作成用関数

空の連想配列を作成します。

```
map = [=>];
map = Associative Array();
```

デフォルト値を持つ空の連想配列を作成します。

```
map = [=>0];
map = Associative Array( 0 );
```

値を指定して連想配列を作成します。

```
map = ["yes" => 0, "no" => 1];
```

各キーの値およびデフォルト値を指定して連想配列を作成します。

```
map = ["yes" => 0, "no" => 1, => 2];
```

キーと値のセットを表すリストから、連想配列を作成します。

```
map = Associative Array( {{"yes", 0}, {"no", 1}} );
```

キーと値のセットを表すリストにデフォルト値の指定も加えて、連想配列を作成します。

```
map = Associative Array( {{"yes", 0}, {"no", 1}}, 2 );
```

キーのリストと値のリストから、連想配列を作成します。

```
map = Associative Array( {"yes", "no"}, {0, 1} );
```

キーのリストと値のリストおよびデフォルト値を指定して連想配列を作成します。

```
map = Associative Array( {"yes", "no"}, {0, 1}, 2 );
```

2つの列参照から、連想配列を作成します。1番目の列がキー、2番目の列が値になります。

```
map = Associative Array( :name, :height );
```

2つの列参照およびデフォルト値を指定して連想配列を作成します。

```
map = Associative Array(:name, :height, .);
```

キーの1つのリストまたはキーの1つの列参照から、連想配列を作成します。この場合、デフォルト値は0になります。

```
set = Associative Array( {"yes", "no"} );
set = Associative Array( :name );
```

連想配列の使用

キー数の検出

連想配列に含まれているキー数を調べるには、N Items()関数を使います。

```
cary = Associative Array();
cary["state"] = "NC";
cary["population"] = 116234;
cary["weather"] = "cloudy";
cary["population"] += 10;
cary["weather"] = "sunny";
cary["high schools"] = {"Cary", "Green Hope", "Panther Creek"};
N Items( cary );
  4
```

キーおよび値の追加と削除

連想配列へのキーと値のペアの追加や削除には、次の関数を使用します。

- Insert()
- Insert Into()
- Remove()
- Remove From()

次の点を念頭に置いてください。

- Insert() と Remove() は、キーと値のペアを追加または削除した連想配列のコピーを戻します。
- Insert Into() と Remove From() は、指定の連想配列に対して、直接キーと値のペアの追加または削除を行います。
- Insert() と Insert Into() は、連想配列、キー、および値の3つの引数を取ります。
- Remove() と Remove From() は、連想配列とキーの2つの引数を取ります。
- 値を指定しないでキーを挿入した場合は、キーには値1が割り当てられます。

例

次の例は、Insert() と Insert Into() を説明しています。

```
newcary = Insert( cary, "time zone", "Eastern" );
Show( cary, newcary );
  cary = Associative Array({{"high schools", {"Cary", "Green Hope", "Panther
    Creek"}}, {"population", 116244}, {"state", "NC"}, {"weather", "sunny"}});
  newcary = Associative Array({{"high schools", {"Cary", "Green Hope", "Panther
    Creek"}}, {"population", 116244}, {"state", "NC"}, {"time zone",
    "Eastern"}, {"weather", "sunny"}});

Insert Into(cary, "county", "Wake");
Show( cary );
```

aa << Insertは連想配列に送られるメッセージで、Insert Into() 関数と同じ処理を実行します。たとえば、次の2つのステートメントは同じ結果になります。

```
cary << Insert( "county", "Wake" );
Insert Into( cary, "county", "Wake" );
```

次の例は、Remove() と Remove From() を説明しています。

```
newcary = Remove( cary, "high schools" );
Show( cary, newcary );
  cary = Associative Array({{"county", "Wake"}, {"high schools", {"Cary", "Green
    Hope", "Panther Creek"}}, {"population", 116244}, {"state", "NC"},
    {"weather", "sunny"}});

Remove From( cary, "weather" );
Show( cary );
```

```
cary = Associative Array({{"high schools", {"Cary", "Green Hope", "Panther
  Creek"}}, {"population", 116244}, {"state", "NC"}});
```

aa << Removeは連想配列に送られるメッセージで、Remove From()関数と同じ処理を実行します。たとえば、次の2つのステートメントは同じ結果になります。

```
cary << Remove( "weather" );
Remove From( cary, "weather" );
```

連想配列内のキーまたは値の検出

連想配列の中に特定のキーが含まれているかどうかを調べるには、Contains()を使用します。

```
cary = Associative Array();
cary["state"] = "NC";
cary["population"] = 116234;
cary["weather"] = "cloudy";
cary["population"] += 10;
cary["weather"] = "sunny";
cary["high schools"] = {"Cary", "Green Hope", "Panther Creek"};
Contains(cary, "high schools");
  1
Contains(cary, "lakes");
  0
```

連想配列に含まれるすべてのキーのリストを取得するには、<< Get Keysメッセージを使用します。

```
cary << Get Keys;
  {"high schools", "population", "state", "weather"}
```

連想配列に含まれるすべての値のリストを取得するには、<< Get Valuesメッセージを使用します。

```
cary << Get Values;
  {{"Cary", "Green Hope", "Panther Creek"}, 116244, "NC", "sunny"}
```

特定のキーの値だけを取得するには、キーを引数として指定します。その際、キーはリストで指定する必要があります。

```
cary << Get Values({"state", "population"});
  {"NC", 116244}
```

1つのキーの値を取得するには、<<Get Valueメッセージを使用します。指定できるキーは1つだけで、リストを指定することはできません。

```
cary << Get Value("weather");
  "sunny"
```

連想配列に含まれるすべてのキーと値のペアのリストを取得するには、<< Get Contentsメッセージを使用します。

```
cary << Get Contents;
  {{"high schools", {"Cary", "Green Hope", "Panther Creek"}},
```

```
{"population", 116244},
{"state", "NC"},
{"weather", "sunny"}}
```

メモ: <<Get Contents メッセージを使用して取得したリストには、デフォルト値は含まれません。キーはアルファベット順（文字コード順）で表示されます。

連想配列内の反復

連想配列の中を反復させるには、<<First メッセージと <<Next メッセージを使用します。<<First は、連想配列の中の最初のキーを戻します。<<Next(key) は、引数として指定されたキー（*key*）の後のキーを戻します。

次のコマンドは、連想配列 cary からキーと値のペアをすべて削除し、空の連想配列を残します。

```
currentkey = cary << First;
total = N Items( cary );
For( i = 1, i <= total, i++,
   nextkey = cary << Next( currentkey );
   Remove From( cary, currentkey );
   currentkey = nextkey;
);
Show( cary );
  cary = [=>];
```

連想配列の応用

連想配列を使用すると、いろいろなタスクをすばやく効率的に実行することができます。

データテーブル列から一意の値を取得する

メモ: ここで説明している方法は、小数点以下の値を含む列に対しては役立ちません。代わりに Summarize を使用してください。詳細については、「データテーブル」章の「要約統計量をグローバル変数に格納する」（295ページ）を参照してください。

連想配列内では1つのキーを一度しか使用できないため、列の値を連想配列に入れれば自動的に一意の値となります。たとえば、「Big Class.jmp」サンプルデータテーブルには行が40個あります。**「身長(インチ)」**列に一意の値がいくつあるかを調べるには、次のスクリプトを実行します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
unique heights = Associative Array( dt:Name("身長(インチ)") );
nitems( unique heights );
  17
```

「身長(インチ)」の一意の値は17個しかありません。キーを取得することで、これらの一意の値を使用できます。

```
unique heights << Get Keys;
  {51, 52, 55, 56, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70}
```

メモ: これが可能なのは、JMPでは、文字列だけではなくあらゆるデータタイプが連想配列のキーとして使用できるためです。

連想配列を使えば、列内の一意の値を簡単に効率的に見つけることができます。次のスクリプトは、100,000行もあるデータテーブルを作成するので時間がかかります。しかし39個の一意の値を探す作業はほとんど時間がかかりません。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
nms = dt:name << Get Values;
dtbig = New Table( "Really Big Class",
   New Column( "name",
      Character,
      Set Values( nms[J( 100000, 1, Random Integer( N Items( nms ) ) )] )
   )
);
Wait( 0 );
t1 = Tick Seconds();
Write(
   "\!N# names from Really Big Class = ",
   N Items( Associative Array( dtbig:name ) ),
   ", elapsed time=",
   Tick Seconds() - t1
);
  # names from Really Big Class = 39, elapsed time=0.116666666639503
```

列の値を文字コード順に並べ替える

キーは文字コード順に並べられるため、連想配列に値を入れれば、値が文字コード順に並べ替えられます。たとえば、<<Get Keysメッセージはキー（「**名前**」列の一意の値）を昇順で戻します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
unique names = Associative Array( dt:名前 );
unique names << Get Keys;
  {"ALFRED", "ALICE", "AMY", "BARBARA", "CAROL", "CHRIS", "CLAY", "DANNY",
     "DAVID", "EDWARD", "ELIZABETH", "FREDERICK", "HENRY", "JACLYN", "JAMES",
     "JANE", "JEFFREY", "JOE", "JOHN", "JUDY", "KATIE", "KIRK", "LAWRENCE",
     "LESLIE", "LEWIS", "LILLIE", "LINDA", "LOUISE", "MARION", "MARK", "MARTHA",
     "MARY", "MICHAEL", "PATTY", "PHILLIP", "ROBERT", "SUSAN", "TIM", "WILLIAM"}
```

2つの異なるデータテーブルの列を比較する

連想配列を使用すると、1つの列のどの値が別の列にないか（またはどの値が両方の列にあるか）を簡単に調べることができます。たとえば、国に関するデータテーブルが2つあり、どの国が両方のデータテーブルに記載されているかを調べたいとします。

各データテーブルの国名が含まれている列から連想配列を作成します。

```
dt1 = Open( "$SAMPLE_DATA/BirthDeathYear.jmp" );
dt2 = Open( "$SAMPLE_DATA/World Demographics.jmp" );
aa1 = Associative Array( dt1:国 );
aa2 = Associative Array( dt2:国 );
```

N Items()を使用して、各データテーブルに出現する国の数を調べます。

```
N Items(aa1);
  23
N Items(aa2);
  238
```

<<Intersectメッセージを使用して、共通の値を調べます。

```
aa1 = Associative Array( dt1:国 );
aa1 << Intersect( aa2 );
```

結果を表示します。

```
Show(N Items(aa1), aa1 << Get Keys);
  N Items(aa1) = 21;
  aa1 << get keys = {"Australia", "Austria", "Belgium", "France", "Greece",
     "Ireland", "Israel", "Italy", "Japan", "Mauritius", "Netherlands", "New
     Zealand", "Norway", "Panama", "Poland", "Portugal", "Romania",
     "Switzerland", "Tunisia", "United Kingdom", "United States"};
```

この例では、交差と呼ばれるセット処理を使用しています。連想配列を使用したセット処理で値を比較する方法については、「集合演算における連想配列」（211ページ）でさらに例が紹介されています。

グラフ理論における連想配列

連想配列は、以下の有向グラフの例のようなグラフ理論データ構造にも使用できます。

```
g = Associative Array();
g[1] = Associative Array({1, 2, 4});
g[2] = Associative Array({1, 3});
g[3] = Associative Array({4, 5});
g[4] = Associative Array({4, 5});
g[5] = Associative Array({1, 2});
```

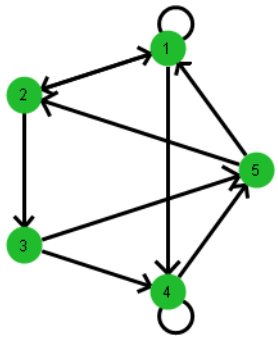
ここでは連想配列が入れ子になっています。この連想配列gには5つの連想配列（1、2、3、4、5）が含まれています。外側の配列gでは、キー（1～5）と値（マップを定義する配列）の両方が重要です。内側の連想配列では、値は関係なく、キーだけが意味を持っています。

連想配列は、図7.2のグラフを次のように表します。

- ノード1はノード1、2、および4につながる
- ノード2はノード1および3につながる

- ノード3はノード4および5につながる
- ノード4はノード4および5につながる
- ノード5はノード1および2につながる

図7.2 有向グラフの例



次に示す深さ優先探索を行う関数は、前述のような連想配列として表現された有向グラフを探索するのに使用できます。

```
dfs = Function( {ref, node, visited},
    {chnode, tmp},
    Write( "\!NNode: ", node, ", ", ref[node] << Get Keys );
    visited[node] = 1;
    tmp = ref[node];
    chnode = tmp << first;
    While( !Is Missing( chnode ),
        If( !visited[chnode],
            visited = Recurse( ref, chnode, visited )
        );
        chnode = tmp << Next( chnode );
    );
    visited;
);
```

次の点を念頭に置いてください。

- 最初の引数はマップ構造を含んだ連想配列です。
- 2番目の引数は開始点として使用するノードです。
- 3番目の引数は、関数が、確認したノードの追跡に使用するベクトルです。

この関数の動作を確認するには、次の式をスクリプトの末尾に追加します。

```
dfs( g, 2, J( N Items( g << Get Keys ), 1, 0 ) );
```

出力は次のとおりです。

```
Node 2: {1, 3}
Node 1: {1, 2, 4}
Node 4: {4, 5}
Node 5: {1, 2}
Node 3: {4, 5}
[1, 1, 1, 1, 1]
```

出力の最初の5行は、ノード2から開始して、リストされた順番にその他のノードすべてに到達できることを示しています。各ノードは、そこからつながっているノード（キー）もリストします。各キーの値は1です。最後の行は、2から各ノードに到達できることを示す行列です。ノード2から到達できないノードがある場合、その値は0で表されます。

ノードのトラバーサルは次のように読みます。

1. ノード2から開始し、ノード1に行く
2. ノード1からノード4に行く
3. ノード4からノード5に行く
4. ノード5からノード2に戻り、その後、ノード3に行く

マップスクリプト

次は、図7.2のマップを生成するスクリプトです。

```
New Window( "Directed Graph",
   Graph Box(
      Frame Size( 300, 300 ),
      X Scale( -1.5, 1.5 ),
      Y Scale( -1.5, 1.5 ),
      Local( {n = N Items( g ), k = 2 * Pi() / n, r, i, pt, from, to,
         edge, v, d},
         Fill Color( "green" );
         Pen Size( 3 );
         r = 1 / (n + 2);
         For( i = 1, i <= n, i++,
            pt = Eval List( {Cos( k * i ), Sin( k * i )} );
            edges = g[i];
            For( edge = edges << First, !Is Empty( edge ),
               edge = edges << Next( edge ),
               to = Eval List( {Cos( k * edge ), Sin( k * edge )} );
               If( i == edge,
                  Circle( Eval List( 1.2 * pt ), 0.9 * r ), // else
                  v = pt - to;
                  d = Sqrt( Sum( v * v ) );
                  {from, to} = Eval List(
                     {pt * (d - r) / d + to * r / d, pt * r / d + to *
```

```
                        (d - r) / d}
                    );
                    Arrow( from, to );
                );
            );
            Circle( pt, r, "fill" );
            Text( Center Justified, pt - {0, 0.05}, Char( i ) );
        );
    )
)
);
```

集合演算における連想配列

連想配列を使用して、集合演算を行うこともできます。次の例は、2つの集合の和集合、差集合および積集合を得る方法を示しています。

まず、3つのセットを作成し、ログで確認します。

```
set_y = Associative Array( {"椅子", "人", "リレー", "蛇", "三脚"} );
set_z = Associative Array( {"人", "蛇"} );
set_w = Associative Array( {"りんご", "みかん"} );
// セットをログに書き込む
Write(
   "\!N例:\!N\!tset_y = ",
   set_y << Get Keys,
   "\!N\!tset_z = ",
   set_z << Get Keys,
   "\!N\!tset_w = ",
   set_w << Get Keys
);
  例:
    set_y = {"リレー", "椅子", "三脚", "蛇", "人"}
    set_z = {"蛇", "人"}
    set_w = {"みかん", "りんご"}
```

和集合

2つの集合の和集合を求めるには、1つの集合を他方に挿入します。

```
set_z << Insert( set_w );
Write( "\!N\!N和集合 (set_w, set_z):,\!N\!tset_z = ", set_z << Get Keys );
  和集合 (set_w, set_z):,
    set_z = {"みかん", "りんご", "蛇", "人"}
```

差集合

2つの集合の差集合を求めるには、一方を他方から削除します。

```
set_y << Remove( set_z );
Write( "\!N\!N差集合 (set_z from set_y):\!N\!tset_y = ", set_y << Get Keys );
  差集合 (set_z from set_y):
    set_y = {"リレー", "椅子", "三脚"}
```

積集合

2つの集合の積集合を求めるには、aa << Intersectメッセージを使用します。

```
set_w << Intersect( set_z );
Write( "\!N\!N積集合 (set_w, set_z):\!N\!tset_w = ", set_w << Get Keys );
  積集合 (set_w, set_z):
    set_w = {"みかん", "りんご"}
```

集合演算の例

名前のリストから、「Big Class.jmp」に含まれていない名前を探します。そのためには、名前を含む2つの集合から差集合を求めます。

1. 名前のリストを作成し、データテーブルを開きます。

```
names list = {"ROBERT", "JEFF", "CHRIS", "HARRY"};
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
```

2. 名前のリストを連想配列に入れます。

```
names = Associative Array( names list );
```

3. リストから列の値を削除することで、差集合を求めます。

```
names << Remove( Associative Array( dt:名前 ) );
```

4. 結果を確認します。

```
Write( "\!N{ROBERT, JEFF, CHRIS, HARRY}のうちでBig Classに含まれていないもの = ",
   names << Get Keys );
  {ROBERT, JEFF, CHRIS, HARRY}のうちでBig Classに含まれていないもの = {"HARRY",
    "JEFF"}
```

第8章

プログラミング手法
複雑なスクリプト技術とその他の関数

この章では、例外のスローとキャッチ、スクリプトの暗号化、複雑な式の使用など、さらに高度な技術について説明します。

リストと式

保存された式

式（expression）とは、評価を行うことができるものの総称です。「JSLの構成要素」章の「名前解決のルール」（91ページ）の最初の節では、JMPがどのように式を評価するか説明しました。ここでは、**いつ**JMPが式を評価するかについて学習します。

JMPは、できる限りすぐに式を評価し、その結果を戻します。式が割り当ての右辺にある場合、式は評価され、その結果が左辺に割り当てられます。多くの場合、このような処理で構わないでしょうが、時には評価を延期できるようにする必要が生じることもあります。

式のクォートと非クォート

いつ式を評価するかを制御する演算子は、Exprと Evalです。これらは、評価を遅延させる演算子と評価を促す演算子とみなすことができます。Exprは、引数を評価するのではなく、式としてそのままコピーします。Evalはその反対の処理をします。つまり、引数の値を求めてから、その結果を使って式全体を評価します。

Exprと Evalは、JMPに対していつ式として認識し、いつ式の評価結果を戻すかを指示する、クォート演算子と非クォート演算子とみなすことができます。

以下のすべての例では、次の2つが割り当てられているものとします。

```
x = 1; y = 20;
```

Exprを使って*x+y*を式として囲み、この式をaに割り当てた場合、*a*が評価されるときは、*x*と*y*の現在の値を使って式を計算し、結果を戻します。（例外として、Show、Write、および Printのユーティリティがあります。これらは、引数として名前だけを指定した場合、その名前に割り当てられている式を評価しません。）

```
x = 1; y = 20;
a = Expr( x + y );
a;
  21
```

評価された式の結果を求めるのではなく、変数に保存された式そのものが得たい場合は、NameExpr関数を使います。詳細については、「結果ではなく保存された式を取り出す」（216ページ）を参照してください。

```
x = 1; y = 20;
Show( Name Expr( a ) );
  NameExpr(a) = x + y
```

式のクォートを入れ子にした場合、*a*が評価されるときに1階層だけが評価され、結果は式x+yになります。

```
x = 1; y = 20;
a = Expr( Expr( x + y ) );
Show( a );
  a = Expr(x + y)
```

式の値を求めるときは、Evalを使ってすべての層の式を評価します。

```
x = 1; y = 20;
Show( Eval( a ) );
  Eval(a) = 21
```

この操作は、次に示すようにどのレベルでも実行できます。

```
x = 1; y = 20;
a = Expr( Expr( Expr( Expr( x + y ) ) ) );
b = a;
  Expr( Expr( x + y ) )
c = Eval( a );
  Expr( x + y )
d = Eval( Eval( a ) );
  x+y
e = Eval( Eval( Eval( a ) ) );
  21
```

式を文字列としてクォート

JSL Quote() 関数は、式の内容を、引用符で囲んだ文字列として戻します。文字列内のコメントと空白はそのまま維持されます。ログウィンドウでも構文の色分けが適用されます。

次のスクリプトはその例です。

```
x = JSL Quote( /* クォートの開始 */
For (i = 1, i <= 5, i++,
   // i の値を出力
Print( i );
);
   // 式の終わり
);
Show( x );
```

出力では、JSL Quote() 関数の内容が引用符で囲まれます。

```
x = " /* クォートの開始 */
For (i = 1, i <= 5, i++,
   // iの値を出力
Print(i);
);
   // 式の終わり
";
```

グローバル変数にスクリプトを格納する

Expr は、主にグローバル変数にスクリプト（マクロなど）を格納するときに使います。

```
dist = Expr( Distribution( Column( :Name("身長(インチ)") ) ) );
```

スクリプトを実行するときは、その名前を入力します。

```
dist;
```

ループの中に入れて、スクリプトを繰り返し実行することもできます。

```
For( i = 0, i < 10, i = i + 1, dist );
```

Eval()を使うと、式の評価を明示的に指定できます。

```
Eval( dist );
```

ただし、列計算式における列に対しては、eval()は思い通りに動作しないことに注意してください。たとえば、

```
Formula( Log( Eval( Column Name( i ) ) ) );
```

はエラーになります。代わりに、次のように指定します。

```
Formula( Eval( Substitute( Expr( Log( xxx ) ), Expr( xxx ), Column Name( i ) ) ) );
```

また、

```
Formula( Eval( Column Name( i ) ) + 10 );
```

でもエラーが生じます。計算式において列が評価されているためです。代わりに、次のように指定します。

```
Formula(Eval(Substitute(Expr(xxx+10), Expr(xxx), column name(i))))
```

結果ではなく保存された式を取り出す

グローバル変数を評価（実際にプラットフォームを起動）するのではなく、グローバル変数に保存されている式（上の例のdistに格納されている式Distribution(Column(:Name("身長(インチ)")))など）が必要な場合は、どうするのでしょうか。そのときは、Name Expr関数を使います。Name Exprは引数を評価しないで式として取り出します。ただし、引数がひとつの変数名だけの場合は、その変数に保存された式を、評価されていない状態で取り出します。

Exprが引数そのものを戻すのに対して、Name Exprは引数に保存されている式を検索します。Name Exprは式を得るために一番上の層だけを「開封」するのであり、すべての階層を開封するのではありません。

たとえば、変数に式を保存していて、その式を編集する必要がある場合などに、この関数を使うことになります。

```
popVar = Expr( Summation( i = 1, N Row(), (y[i] - Col Mean( y )) ^ 2 / N Row() ) );
  Summation( i = 1, N Row(), (y[i] - Col Mean( y )) ^ 2 / N Row() )

unbiasedPopVar = Substitute( Name Expr( popVar ), Expr( Wild()/N Row() ), Expr(
   (y[i] - Col Mean( y )) ^ 2 / ( N Row() - 1 ) ) );
  Summation( i = 1, N Row(), (y[i] - Col Mean( y )) ^ 2 / (N Row() - 1) )
```

次のスクリプトを実行してから、x、Expr(x)、NameExpr(x)、およびEval(x)を比較してみます。

```
a = 1; b = 2; c = 3;
x = Expr( a + b + c );
```

表8.1 Eval、NameExpr、Exprの比較

コマンドと結果	説明
x; *6*	*x* をa+b+cと評価し、さらにその式を評価して結果の6を戻す（*外側の2つの層を開封する*）。
Eval(x); *6*	単に *x* を呼び出しているのと同じ。 *x* をa+b+cと評価し、さらにその式を評価して結果の6を戻す（*外側の2つの層を開封する*）。
NameExpr(x); *a+b+c*	*x* に格納されていた式a+b+cを戻す（*1番外の層を開封する*）。
Expr(x); *x*	式xを戻す（*1つの層に封入する*）。

JSLには式にアクセスして式全体を調べる関数も用意されており、これらの関数はすべて、引数として名前または文字式のどちらかをとります。以下の例では、*expressionArg*は、変数の名前、または、変数と演算子などから構成される複合式のいずれかです。

NArg(*expressionArg*)は、*expressionArg*の中の引数の数を調べます。

*expressionArg*は、式を含む変数の名前、評価結果が式になるような式、またはExpr()で囲まれた式のいずれかです。

NArg (name)は、nameから式（評価されていない状態）を取り出し、引数の数を戻します。

NArg (*expression*)は、expressionを評価し、引数の数を戻します。

NArg (Expr(expression))は、式expressionの引数の数を戻します。

たとえば、aExpr = {a+b,c,d,e+f+g}; の場合

- NArg(aExpr)は4
- NArg(Arg(aExpr,4))は3
- NArg(Expr({1,2,3,4}))は4となります。

Head(*expressionArg*)は、演算子を使わないで表した式の先頭の関数（引数なし）を戻します。式が特殊文字の二項演算子、接頭演算子、接尾演算子であるときは、それらと等価な関数が戻されます。

*expressionArg*は、式を含む変数の名前、評価結果が式になるような式、またはExpr()で囲まれた式のいずれかです。

たとえば、aExpr = expr(a+b); の場合

- r = Head(aExpr)はAdd()
- r = Head (Expr(sqrt(r)))はSqrt()
- r = Head({1,2,3})は{}となります。

Arg(*expressionArg*,*indexArg*)は、式の中から指定の引数を抽出し、式として戻します。

例:

Arg(*expressionArg*,i)は*expressionArg*のi番目の引数を抽出します。

*expressionArg*は、式を含む変数の名前、評価結果が式になるような式、またはExpr()で囲まれた式のいずれかです。

- Arg(*name*,i)は、*name*に含まれた式（評価されていない状態）を取得し、i番目の引数を検索します。
- Arg(*expression*,i)は、*expression*を評価し、i番目の引数を検索します。
- Arg(Expr(*expression*),i)は、expressionのi番目の引数を検索します。

別の例としてaExpr = Expr(12+13*sqrt(14-15));の場合を示します。

- Arg(aExpr,1)は12
- Arg(aExpr,2)は13*sqrt(14-15)
- Arg(Expr(12+13*sqrt(14-15)),2)は13*sqrt(14-15)

式の中の引数の引数を抽出するには、Argコマンドを入れ子にします。

- Arg(Arg(aExpr,2),1)は、aExprの第2引数の中の最初の引数、つまり13を戻します。
- Arg(Arg(aExpr,2),2)はSqrt(14 - 15)
- Arg(Arg(Arg(aExpr,2),2),1)は14 - 15
- Arg(Arg(Arg(aExpr,2),2),3)はEmpty()

以下に、最後の式がどのように開封されていくかを説明します。

1. 内側のArgステートメントが評価されます。

```
Arg(aExpr,2)
  13 * Sqrt( 14 - 15 )
```

2. 次に、内側から2番目のArgが評価されます。

```
Arg(Arg(aExpr,2),2)
// これはArg(Expr(13 * Sqrt( 14 - 15 ) ), 2)と等価
  Sqrt( 14 - 15 )
```

3. 最後に、外側のArgが評価されます。

```
Arg(Arg(Arg(aExpr,2),2),3)
// これはArg (Expr(Sqrt( 14 - 15 ) ), 3)と等価
  Empty()
```

Sqrt式の要素は1つだけなので、第3引数への要求に対してEmpty()が戻されます。Sqrt式内の2つの引数にアクセスするには、次のようにします。

```
Arg(Arg(Arg(Arg(aExpr,2),2),1),2);
  15
```

HeadName(*expressionArg*)は、式の先頭の関数の名前を文字列として戻します。式が特殊文字の二項演算子、接頭演算子、接尾演算子であるときは、それらと等価な関数の名前が戻されます。

*expressionArg*は、式を含む変数の名前、評価結果が式になるような式、またはExpr()で囲まれた式のいずれかです。

たとえば、aExpr = expr(a+b);の場合

- r = HeadName (aExpr)は"Add"
- r = HeadName (Expr(sqrt(r)))は"Sqrt"
- r = HeadName ({1,2,3})は"List"となります。

JMPのこれまでのバージョンでは、Arg、Narg、Head、およびHeadNameがそれぞれArgExpr、NArgExpr、HeadExpr、およびHeadNameExprとして実装されていました。基本的に働きは同じでしたが、引数の評価を行わない点が異なります。旧バージョンの形式は、今後のマニュアルには記載されなくなる予定です。

文字列の部分置換

Eval Insertでは、特定の文字に囲まれた式を評価することによって、文字列の部分置換を行うことができます。Perlでは、これは**補間**（interpolation）と呼ばれます。

Eval Insertでは、式の前後の文字を指定すると、その間の式が評価、展開されます。

結果を返す関数と、値を置き換える関数の2通りがあります。

```
resultString = EvalInsert( 式が埋め込まれた文字列 , 開始区切り文字 , 終了区切り文字 )
EvalInsertInto( 式が埋め込まれた文字列を持つ変数 , 開始区切り文字 , 終了区切り文字 )
```

区切り文字の指定はオプションです。デフォルトの開始文字は"^"、デフォルトの終了文字は、開始文字と同じ文字です。

```
xstring = "def";
r = Eval Insert( "abc^xstring^ghi" );  // 戻り値が "abcdefghi"; になる

r = "abc^xstring^ghi"; // 値を置き換え
Eval Insert Into( r ); // r が "abcdefghi"; になる

// 区切り文字の指定
r = Eval Insert( "abc%xstring%ghi","%" );  // 戻り値が "abcdefghi"; になる

// 開始値と終了値が異なる例
r = Eval Insert( "abc[xstring]ghi","[","]" );  // 戻り値が "abcdefghi"; になる
```

数値にロケール固有の表示形式が含まれている場合は、<<Use Locale(1)オプションを含めます。次の例は、小数点の代わりにカンマを使用するロケール環境で実行したものです。

```
Eval Insert( "^1.2^", <<Use Locale( 1 ) );
  1,2
```

リストの中の式を評価する

Eval Listは、リスト内の式を評価し、その結果をリストにして戻します。

```
x = { 1 + 2, 3 + 4 };
y = Eval List( x );   // yは{3,7}になる
```

Eval Listは、Column Dialogまたは引数Modalを使用したNew Windowから戻された、ユーザの選択によるリストをロードするのに便利です。

式の中の式を評価する

Eval Expr()は、内側の式だけを評価し、その結果を含む式を戻します。これと比較して、Evalは内側の式を評価し、結果を用いて式をさらに評価します。

データテーブルに**X3**という名前の列があるとしましょう。ここで、Eval Expr()を使って内側の式を最初に評価する例を紹介します。

```
x = Expr( Distribution( Column( Expr("X"||Char( i ) ) ) ) );
i = 3;
y = Eval Expr( x );   // Distribution( Column( "X3" ) )を戻す
```

結果をさらに評価するには、その後のステップで結果を呼び出すか、Eval Expr()をEval()で囲む必要があります。以下の例では、「一変量の分布」レポートを作成します。

```
// 2つのステップを取る方法
x = Expr( Distribution( Column( Expr( "X" || Char( i ) ) ) ) );
i = 3;
y = Eval Expr( x );
y;

// 1つのステップで済む方法
x = Expr( Distribution( Column( Expr( "X" || Char( i ) ) ) ) );
i = 3;
Eval( Eval Expr( x ) );
```

最初にEval Exprを実行せずに、直接*x*にEvalを使おうとした場合にどのようなことが起こるかについては、表8.3（222ページ）を参照してください。

文字列を式に、式を文字列に解析する

構文解析とは、プログラム言語の式として文字列を取り込むことです。JSLの式を文字列として保持しており、その文字列から式を構築したいとします。Parse関数は式を戻します。Parse関数は式を戻すだけなので、式を評価するにはEval関数を使います。

```
x = Parse( "a=1" ) ;  // 現在xには式a=1が入っている
Eval( Parse( "a=1" ) ); // 現在aには値1が入っている
```

この逆をするには、式を文字列に変換するChar関数を使います。通常、Charは引数を評価してから文字列に変換するため、Char関数の引数をExpr関数（または変数のNameExpr関数）にします。

```
y = Char( Expr( a = 1 ) ); // y は「a=1」という文字値になる
z = Char( 42 ); // "42" になる
```

引数が数値の場合、Char関数では、フィールド幅と小数桁数の引数を指定できます。デフォルトでは、フィールド幅は18で、小数桁数は99です（[最適] の形式）。

```
Char( 42, 5, 2 ); // 文字値は "42,00" になる
```

数値にロケール固有の形式を維持するには、次の例のように、<<Use Locale(1) オプションを含めます。

```
Char( 42, 5, 2, <<Use Locale(1) ); // フランスのロケールで実行した場合、"42,00" になる
```

Charの逆は、それほど簡単ではありません。文字列を式に変換するときはParseを使いますが、文字列を数値に変換するときはNumを使います。

```
Parse( y );
Num( z );
```

表8.2 式の保存と計算をする関数

関数	構文	説明
Char	Char(Expr(*expression*)) Char(*name*)	*expression*（式）を文字列に変換する。式はExprで囲む必要があります。囲まない場合、その評価結果が文字列に変換されます。
	string = char(*number*, *width*, *decimal*)	*number*（数値）を文字値に変換する。*width* と *decimal* は、形式を指定するためのオプションの引数です。デフォルトでは、width（フィールド幅）は18で、decimal（小数桁数）は99です。
Eval	Eval(*x*)	*x* を評価し、次いで *x* の結果を評価する（非クォート）。
Eval Expr	Eval Expr(*x*)	*x* の中の式すべてを評価して、式を戻す。
Eval List	Eval List(*list*)	リスト（*list*）内の式を評価した後のリストを戻す。
Expr	Expr(*x*)	引数を評価しないまま戻す（式のクォート）。
NameExpr	NameExpr(*x*)	*x* に格納されている式を評価せずに戻す。NameExprは、*x* が1つの変数名の場合、名前xを評価せずに戻すのではなく、その変数 *x* に保存されている式を評価せずに戻します。その点を除けば、NameExprはExprと同じです。
Num	Num("*string*")	文字列を数値に変換する。
Parse	Parse("*string*")	文字列をJSLの式に変換する。

まとめ

表8.3では、*x*を使った評価制御演算子のさまざまな使い方を比較しています。データテーブルにX3という名前の列があり、*x*と*i*が割り当てられているとしましょう。

```
x = Expr( Distribution(Column( Expr("X"||Char( i ) ) ) ) );
i = 3;
```

表8.3 計算を制御する演算子

コマンドと結果	説明
`x; // または` `Eval(x);` *見つかりません。'distribution' へのアクセスまたは評価 , 不正な引数 ({"X" || Char(i)}), distribution(Column(Expr("X" || Char(i))))*	Eval(x)とxの呼び出しは同じ。 式distribution(column(expr("X" || Char(i))))を評価します。結果はエラーとなります。列名は、Expr() 関数によって囲まれているため、"X"||Char(i)のように認識されます。
`Expr(x);` *x*	式xを戻す（*追加の*層によりxを包みこむ）。
`Name Expr(x);` *Distribution(Column(Expr("X" || Char(i))))*	*x*に保存されている式Distribution(Column(Expr("X" || Char(i))))をそのまま戻す。
`y=Eval Expr(x);` *Distribution(Column("X3"))*	内側の式を評価するが、外側の式は評価しないため、*y*はDistribution(Column("X3"))。
`y; // または` `Eval(Eval Expr(x));` *Distribution[]*	Eval(eval expr(x))とyの呼び出しは同じ。 Distribution(Column("X3"))を評価し、プラットフォームを起動します。
`z = Char(nameexpr(x));` *"Distribution(Column(Expr (¥!"X¥!" || Char(i))))"*	式全体を文字列としてクォートする。必要に応じて、¥!"というエスケープ文字を追加します。 Char(x)と指定すると、最初に*x*を評価しようとしてエラーが発生するため、欠測値を引用符で囲んだ"."が戻されることに注意してください。
`Parse(z);` *Distribution(Column(Expr("X" || Char(i))))*	文字列を解析（パース）して式を戻す。

表8.3 計算を制御する演算子 （続き）

コマンドと結果	説明
a = Parse(Char(NameExpr(x))); Eval(EvalExpr(a)); *Distribution[]*	とても極端な例。 この例は、少なくともこれを2つのステップに分解する必要があることに注意してください。これを1つの大きなステップに組み合わせると、Eval Expr関数が、Parse層以下の内容を評価せずにそのまま残してしまうため、別の結果となります。 Eval(EvalExpr(Parse(Char(NameExpr(x))))); *Distribution(Column(Expr("X" || Char(i))))*

マクロ

保存された式（expression)は、マクロとして使用することも可能です。汎用的な処理を式として変数に保存しておくと、その変数を呼び出せばいつでも中の汎用的な処理を実行できます。この例では、Ifの引数として4つのマクロ（グローバル変数）を指定しています。

```
lastStdzdThickness=expr(
   (thickness[nrow()]-col mean(thickness)) / col std dev(thickness));
continue=expr(...<データを読み込むスクリプト>...);
log=expr(print(char( long date(today()))||" の分です。"));
limitvalue=1;

if(lastStdzdThickness<limitvalue,log;continue,break);
```

Exprを使って式を保存すると、その式はスクリプトそのもので、その時点では評価されません。式を保存した変数を実際に呼び出すまで、評価は行われません。式に含まれているすべての変数、データ、または式は、式の実行時に初めて評価されます。式を保存し、後で評価するときの詳細なルールについては、「保存された式」（214ページ）を参照してください。

リストの操作

次に挙げる演算子でリストを操作します。また、これらの演算子は、次の節、「式の操作」（226ページ）で説明するように、式の操作にも使うことができます。コマンドとその説明は、表8.4（228ページ）にまとめられています。

大部分の関数には、新しい値を作成する形と、その場で直接引数に作用する形の2種類があります。たとえば、次の場合が該当します。

```
A = Remove( A, 3 ); // リスト A の 3 番目の項目を削除し、A に結果を保存する
Remove From( A, 3 ); // その場でリスト A の 3 番目の項目を削除する
Show( A ); // A = {2, 3, 2, 1, 2, 1};

onetwo = Insert( {1}, 2 );  // {1,2}
Insert Into( B, {1, 2}, 4 ); // 現在の 4 番目の項目の前に 1,2 を配置する
Show( B ); // B = {2, 3, 4, 1, 2, 1, 2, 1, 2, 1};
```

メモ：Insert Into 関数で挿入する場所を省略した場合は、リストの最後尾に項目が挿入されます。

```
a = Shift( {1,2,3,4}, 1 ); // a にリスト {2,3,4,1} を保存する
Shift Into( a, -1 );
Show( a ); // a = {1, 2, 3, 4};

b = Reverse( a ); // b は {4,3,2,1} になる
Reverse Into( a ); // a は {4,3,2,1} になる
Show( b ); // b = {2, 3, 4, 1};

s = Sort List( {1,4,2,5, -7.2, Pi(), -11, cat, apple, cake} );
Show( s ); // s = {-11, -7.2, 1, 2, 4, 5, apple, cake, cat, Pi()};

c = {5, pie, 2, Pi(), -2};
Sort List Into( c );
Show( c ); // c = {-2, 2, 5, Pi(), pie};
```

インプレース演算子

インプレース演算子は、リストまたは式を直接操作する演算子です。この演算子は、たとえば、Remove From、Insert Into などのように、演算子名に From または Into が付いています。結果を戻さないため、結果を見るにはリストを表示させる必要があります。インプレース演算子の第1引数は、L-value でなければなりません。**L-value** とは、値を代入できるグローバル変数などのエンティティです。

```
myList = {a, b, c, d};
Insert Into( myList, 2, 3 );
Show( myList );
  myList:{a, b, 2, c, d}
```

以下の例では、入れ子になったリストを用いて、Insert Into と Remove From の使い方を示しています。

```
a = {{1, 2, 3}, {"A", "B", "C"}};
Show( a );
  a = {{1, 2, 3}, {"A", "B", "C"}}

Insert Into( a[1], 99, 1 );
Show( a );
  a = {{99, 1, 2, 3}, {"A", "B", "C"}}

Remove From( a[1], 1 );
```

```
Show( a );
  a = {{1, 2, 3}, {"A", "B", "C"}}
```

インプレースでない演算子

インプレースでない演算子の場合には、リストを直接記述するか、評価結果がリストとなる変数の名前を指定する必要があります。このような演算子には、名前にFromまたはIntoは付きません。このタイプの演算子は、第1引数の元のリストや式には直接、変更を加えないで、変更した後のリストや式を戻します。

```
myNewList = Insert( {a, b, c, d}, 2, 3 );
  {a, b, 2, c, d}

oldList = {a, b, c, d};
newList = Insert( oldList, 2, 3 );
  {a, b, 2, c, d}
```

Substitute（置換）

Substitute()とSubstitute Into()については、詳しく説明する必要があるでしょう。どちらの関数も、リスト（または式）でパターンに一致するものをすべて検索し、それを別の式に置換します。パターン(pattern)は名前でなければなりません。引数は適用される前に評価されるため、ほとんどの場合、Expr()関数を使って引数をクォートする必要があります。

```
Substitute( {a,b,c}, Expr( a ), 23); // {23,b,c}を戻す
Substitute( Expr( Sine( x ) ), Expr( x ), Expr( y ) ); // Sine(y)を戻す
```

引数の評価を延期するときは、Exprの代わりにName Exprを使います。

```
a = {quick, brown, fox, jumped, over, lazy, dogs};
b = Substitute( a, Expr( dogs ), Expr( cat ) );
canine = Expr( dogs );
equine = Expr( horse );
c = Substitute( a, Name Expr( canine ), Name Expr( equine ) );
Show( a, b, c );
  a = {quick,brown,fox,jumped,over,lazy,dogs}
  b = {quick,brown,fox,jumped,over,lazy,cat}
  c = {quick,brown,fox,jumped,over,lazy,horse}
```

Substitute Intoは同じ操作をインプレースで実行します。

```
Substitute Into( a, Expr( dogs ), Expr( horse ) );
```

引数に複数のパターンを列挙すると、一度のステップで2つ以上の置換を実行できます。

```
d = Substitute( a,
   Name Expr( quick ), Name Expr( fast ),
   Name Expr( brown ), Name Expr( black ),
   Name Expr( fox ), Name Expr( wolf )
);
  {fast,black,wolf,jumped,over,lazy,dogs}
```

同じパターンが式中に複数ある場合には、すべてのパターンに対して代入が行われます。例:

```
Substitute( Expr( a + a ), Expr( a ), Expr( aaa ) );
```

の結果は、次のようになります。

```
aaa + aaa
```

式の操作

リストを操作する演算子は、ほとんどの式を処理することもできます。必ずExpr()を使って式をクォートしてください。

例:

```
Remove( Expr( A + B + C + D ), 2 ); // 式A + C + Dになる
b = Substitute( Expr( Log( 2 ) ^ 2 / 2 ), 2, 3 ); // 式Log( 3 ) ^ 3 / 3になる
```

リストの場合と同様、インプレース演算子の第1引数はL-valueでなければなりません。L-valueとは、値を代入できるグローバル変数などのエンティティです。インプレース演算子は、リストまたは式を直接操作する演算子です。この演算子は、たとえば、Remove From、Insert Intoなどのように、演算子名にFromまたはIntoが付いています。結果を戻さないため、結果を見るには式を表示する必要があります。

```
polynomial = Expr( a * x ^ 2 + b * x + c );
Insert Into( polynomial, Expr( d * x ^ 3 ), 1 );
Show( polynomial );
  polynomial = d * x ^ 3 + a * x ^ 2 + b * x + c
```

インプレース演算子でない場合には、式を直接記述するか、式を含む変数の名前をNameExprに指定する必要があります。このような演算子には、名前にFromまたはIntoは付きません。このタイプの演算子は、第1引数の元のリストや式には直接、変更を加えないで、変更した後のリストや式を戻します。

```
cubic = Insert( Expr( a * x ^ 2 + b * x + c ), Expr( d * x ^ 3 ), 1 );
  d * x ^ 3 + a * x ^ 2 + b * x + c

quadratic = Expr( a * x ^ 2 + b * x + c );
cubic = Insert( Name Expr( quadratic ), Expr( d * x ^ 3 ), 1 );
  d * x ^ 3 + a * x ^ 2 + b * x + c
```

Substitute（置換）

置換はとても強力な機能です。すでに説明した「Substitute（置換）」（225ページ）を参照してください。ここでは、式で置換を行うときの注意点をいくつか説明します。

Substitute（pattern,name,replacement）は、式の名前を置き換えます。

NameExpr()は指定された変数に含まれている式を、評価せずに戻します。

```
a = Expr(
   Distribution( Column( x ), Normal Quantile Plot )
);
```

```
Show( Name Expr( a ) );
  Name Expr(a) = Distribution(Column(x), Normal Quantile Plot);
```

Substitute()では引数がすべて評価されるため、引数を正しくクォートする必要があります。

```
b = Substitute( Name Expr( a ), Expr( x ), Expr( :Name("体重(ポンド)") ) );
Show( Name Expr( b ) );
  Name Expr(b) = Distribution(Column(:Name("体重(ポンド)")), Normal Quantile
    Plot);
```

SubstituteInto()の場合、第1引数はL-value（左辺に指定できるもの）です。NameExprを用いる必要はありません。

```
Substitute Into( a, Expr( x ), Expr( :Name("体重(ポンド)") ) );
Show( Name Expr( a ) );
  Name Expr(a) = Distribution(Column(:Name("体重(ポンド)")), Normal Quantile
    Plot);
```

Substitute()は式の一部を変更するときに便利な関数です。次の例は、Is NumberやIs Matrix関数などの複数の関数を用いる例です。

```
data = {1, {1, 2, 3}, [1 2 3], "abc", x, x( y )};
ops = {is number, is list, is matrix, is string, is name, is expr};
m = J( N Items( data ), N Items( ops ), 0 );
test = Expr(
   m[r, c] = _op( data[r] )
);
For( r = 1, r <= N Items( data ), r++,
   For( c = 1, c <= N Items( ops ), c++,
      Eval( Substitute( Name Expr( test ), Expr( _op ), ops[c] ) )
   )
);
Show( m );
  m =
  [1 0 0 0 0 0,
  0 1 0 0 0 1,
  0 0 1 0 0 0,
  0 0 0 1 0 0,
  0 0 0 0 1 1,
  0 0 0 0 0 1];
```

SubstituteInto()を使うと、JMPで2次方程式を求めることができます。次の例では、$4x^2 - 9 = 0$を求めています。

```
/* 次の方程式の解を求める */
/*         a*x^2 + b*x + c = 0        */
// 2次の式はx=(-b + - sqrt(b^2 - 4ac))/2a
// リストを使って、+- 演算の + と - 両方の結果を保存する
x = {Expr(
   (-b + Sqrt( b ^ 2 - 4 * a * c )) / (2 * a)
```

```
), Expr(
   (-b - Sqrt( b ^ 2 - 4 * a * c )) / (2 * a)
)};
Substitute Into( x, Expr( a ), 4, Expr( b ), 0, Expr( c ), -9 );
// 係数に挿入する
Show( x ); // 置換結果を表示する
Show( Eval Expr( x ) ); // 解を表示する
  x = {Expr(((-0) + Sqrt(0 ^ 2 - 4 * 4 * -9)) / (2 * 4)), Expr(((-0) - Sqrt(0 ^
     2 - 4 * 4 * -9)) / (2 * 4))};
  Eval Expr(x) = {1.5, -1.5};
```

リストと式を操作する演算子については、この前の節、「リストの操作」（223ページ）で説明していますが、その概要を表8.4に示します。

表8.4 リストと式を操作する関数

関数	構文	説明
Remove	*x* = Remove(*list*|*expr*) *x* = Remove(*list*|*expr*, *position*) *x* = Remove(*list*|*expr*, {*positions*}) *x* = Remove(*list*|*expr*, *position*, *n*)	指定の場所（*position*）の項目を削除したリスト（*list*）または式（*expr*）を戻す。場所（*position*）を省略したときは、最後の項目が削除されます。*position*をリストで指定することもできます。追加の引数*n*を指定すると、1つの項目ではなく*n*個の項目が削除されます。
Remove From	Remove From(*list*|*expr*, *position*) Remove From(*list*|*expr*) Remove From(*list*|*expr*, *position*, *n*)	リストから項目を削除する。よって、戻り値を何かに割り当てることはできません。第1引数はL-value（左辺に指定できるもの）でなければなりません。
Insert	*x* = Insert(*list*|*expr*, *item*, *position*) *x* = Insert(*list*|*expr*, *item*)	リスト（*list*）または式（*expr*）の指定箇所(*position*)に項目(*item*)を挿入する。位置を指定しない場合、文字列は末尾に挿入されます。

表8.4 リストと式を操作する関数（続き）

関数	構文	説明
Insert Into	Insert Into(*list*|*expr*, *item*, *position*) Insert Into(*list*|*expr*, *item*)	Insert関数と同じだが、結果を元の変数に格納する。リスト（*list*）や式（*expr*）はL-value（左辺に指定できるもの）でなければなりません。
Shift	*x* = Shift(*list*|*expr*) *x* = Shift(*list*|*expr*, *n*)	1つまたは*n*個の項目を、リスト（*list*）または式（*expr*）の前から後ろへシフトする。*n*の値が負のときは、後ろから前へシフトします。
Shift Into	Shift Into(*list*|*expr*) Shift Into(*list*|*expr*, *n*)	項目をシフトして、その結果を元の変数に割り当てる。
Reverse	*x*=Reverse(*list*|*expr*)	リスト（*list*）や式（*expr*）の項目の順序を逆にする。
Reverse Into	Reverse Into(*list*|*expr*)	リスト（*list*）や式（*expr*）の項目の順序を逆にし、結果を元の変数に割り当てる。
Sort List	*x*=Sort List(*list*|*expr*)	リスト（*list*）や式（*expr*）の項目を並べ替える。まず数値を昇順に並べ、その後に名前・文字列・演算子の内部コード順に並べます。たとえば、+（プラス記号）は -（マイナス記号）よりも、文字コードでは前に位置しているので、1+2は1-2より小さいと判断されます。また、{1,2}は{1,3}より小さく、{1,3}は{1,3,0}より小さいと判断されます。{1000}は{"a"}より小さくなりますが、{a}と{"a"}は同じです。
Sort List Into	Sort List Into(*list*|*expr*)	リスト（*list*）や式（*expr*）の項目を並べ替え、その結果を元の変数に割り当てる。
Sort Ascending	Sort Ascending(*list*|*matrix*)	リスト（*list*）または行列（*matrix*）の要素を昇順に並べたリストを戻す。
Sort Descending	Sort Descending(*list*|*matrix*)	リスト（*list*）または行列（*matrix*）の要素を降順に並べたリストを戻す。
Loc Sorted	Loc Sorted(*A*, *B*)	行列**A**の値と行列**B**の値が一致する位置を、添え字の行列で作成する。**A**は昇順に並べた行列でなければなりません。

表8.4 リストと式を操作する関数（続き）

関数	構文	説明
Substitute	R = Substitute(*list*|*expr*, Expr(*pattern*), Expr(*replacement*), ...)	リスト（*list*）または式（*expr*）の中で、指定のパターン（pattern）に一致する部分を検索し、*replacement* で置き換える。パターン(pattern)は名前でなければなりません。第2引数と第3引数は適用される前に評価されるため、ほとんどの場合、Expr関数を使って引数をクォートする必要があります。引数の評価を延期するときは、Exprの代わりにName Exprを使います。複数の置換を実行する場合、1つのステートメントでpatternとreplacementのペアを複数指定できます。
Substitute Into	Substitute Into(*list*|*expr*, Expr(*pattern*), Expr(*replacement*), ...)	Substitute関数と同じだが、結果を元の変数に格納する。リスト（list）や式（expr）はL-value（左辺に指定できるもの）でなければなりません。

高度な適用範囲指定と名前空間

プロダクション環境で使用されるスクリプトには、スクリプト間の競合を回避する目的で、より高度な適用範囲の指定技術を使用する必要があります。JMP にはより高度な技術が 3 つ用意されています。

- Names Default To Here()関数。簡単なスクリプトを作成するだけなら、このコマンドで十分です。詳細については、「Names Default To Here」（230ページ）を参照してください。
- JMP で定義済みの適用範囲。詳細については、「適用範囲が指定された名前」（233ページ）を参照してください。
- 独自のスクリプトに作成できる名前空間。詳細については、「名前空間」（237ページ）を参照してください。

Names Default To Here

プロダクションで使用するスクリプトを作成する場合、そのスクリプトを現在のユーザ環境から分離する必要があります。そうでなければ、スクリプト内で使用している変数が、ユーザや別のスクリプトによって使用される他の変数の影響を受ける可能性があります。分離するためには、名前をローカル環境で使用します。それには、次のステートメントを使って実行モードを設定します。

```
Names Default To Here( 1 );
```

Names Default To Hereモードがオンになっているスクリプトの中の非修飾の名前は、そのスクリプトだけに関連付けられます。ただし、名前はスクリプトが存続する限り、または、そのスクリプトによって作成されたオブジェクトやそのスクリプトを保持しているオブジェクトがアクティブである限り、存続します。特別な理由がない限り、プロダクション環境で使用するすべてのスクリプトは、`Names Default To Here(1)`で開始することをお勧めします。スクリプトがこのモードで非修飾の名前を使用する場合、名前はローカルの名前空間内で解決されます。

グローバル変数を参照するには、名前の適用範囲を明確にグローバル変数として指定します（たとえば`::global_name`）。データテーブル内の列を参照するには、名前の適用範囲を明確にデータテーブル列として指定します（たとえば`:column_name`）。

メモ: `Names Default To Here( 1 )`は、特定のスクリプトのモードを定義します。グローバルな定義ではありません。あるスクリプトでこのモードを有効にし、別のスクリプトでは無効にすることができます。デフォルトではオフに設定されています。

JMP 8以前のバージョンでは、スクリプト同士を分離する唯一の方法が、他のスクリプトで使用されていないような長い名前を使用することでした。`Names Default To Here(1)`を使用すると、この方法が必要でなくなります。

`Local()`は、スクリプト内の特定のコンテキストにだけローカルな適用範囲を作成し、相互に作用する関数のある長いスクリプトを含めることはできません。一方、`Names Default To Here(1)`は、スクリプト全体に対してローカルな適用範囲を作成できます。

簡単なスクリプトを作成するだけなら、`Names Default To Here(1)`で十分です。

非修飾の名前付き変数参照の操作

`Names Default To Here()`関数は、**非修飾**の名前付き変数参照の解決方法を決定します。`here:var_name`を使って明示的に変数のスコープを指定すれば、`Names Default To Here()`のオン／オフに関わらず常に適用範囲で動作します。`here`およびその他の適用範囲については、「適用範囲が指定された名前」（233ページ）を参照してください。

Names Default To Hereモードを有効にすると、`Here`というスコープが実行スクリプトに関連付けられます。`Here`スコープには、作成された非修飾の名前付き変数のうち、割り当ての対象（L-value）であるものすべてが含まれます。JMP 8以前のバージョンでは、これらの変数は通常、グローバルスコープに置かれていました。`Here`スコープを使うと、複数の実行スクリプト内の変数がお互いに分離され、名前の競合が回避されるので、変数名の管理やスクリプト作成が簡単になります。`Global`スコープを使えば名前を共有できます。

Names Default To Here とグローバル関数

このスクリプト例を一度に1行ずつ実行し、`Names Default To Here()`関数が変数名の解決にどのような変化をもたらすかを見てみましょう。

スクリプト例

```
a = 1;
Names Default To Here( 1 );
```

```
a = 5;
Show( global:a, a, here:a );
  global:a = 1;
  a = 5;
  here:a = 5;
```

1. 1行目を実行し、名前が*a*、値が1のグローバル変数を作成します。
2. 2行目を実行し、**Names Default To Here**モードをオンにします。
3. 3行目を実行し、名前が*a*、値が5のローカル空間を作成します。この行は、グローバル変数*a*に割り当てられた値を**変更しません**。
4. 4 行目を実行し、範囲指定された変数と範囲指定されていない変数がそれぞれどのように解決されるのかを見ます。

非修飾の*a*はhere:aと認識されます。Names Default To Here()がオンでない場合、*a*は*a*という名前のグローバル変数と認識されます。

ただし、Show()関数内でglobal:aの代わりに::aを使用した場合、出力は若干異なります。

```
Show(::a, a, here:a);
  a = 1;
  a = 5;
  here:a = 5;
```

Names Default To Here()関数の使用例

ここでは、次のような定義の2つのスクリプトがあり、どちらもNames Default To Here()が**オフ**（デフォルト設定）になっています。

メモ: この例では、2つのスクリプトのスクリプトウィンドウが別々でなければなりません。

```
a = 1; // スクリプト 1
Show( a );

a = 3; // スクリプト 2
Show( a );
```

1. スクリプト1を実行します。結果は次のとおりです。

   ```
   a = 1
   ```

2. スクリプト2を実行します。結果は次のとおりです。

   ```
   a = 3
   ```

3. スクリプト1のshow(a);行のみを実行します。結果は次のとおりです。

   ```
   a = 3
   ```

変数*a*はグローバルで、スクリプト2によって最後に変更されたため、ログにはa = 3と表示されます。これは、JMP 9以降ではデフォルトの動作ですが、JMP 8以前のバージョンでは唯一可能な動作でした。

4. では、**両**スクリプトとも、Names Default To Here()をオンにしてみましょう。

   ```
   Names Default To Here(1);
   ```

メモ: Names Default To Here() は、特定のスクリプトに対してローカルです。グローバル設定では**ありません**。

5. スクリプト1を実行します。結果は次のとおりです。

   a = 1

6. スクリプト2を実行します。結果は次のとおりです。

   a = 3

7. スクリプト1の Show(a); 行のみを実行します。結果は次のとおりです。

   a = 1

変数 *a* のコピーが各スクリプトに保持されるため、ログには a = 1 と表示されます。

メモ: この関数を使用する際の問題は、通常、修飾名と非修飾名のグローバル変数への参照が混合することによって起こります。名前の適用範囲を常に明示的に指定することによって、意図しない変数への参照を防ぐことができます。

適用範囲が指定された名前

scope:name という形式のスコープを使用することにより、名前を解決する場所を指定できます。ここで、scope は名前の解決方法を示します。たとえば、here:name は、名前がローカルで解決されるべきであることを意味します。**Names Default To Here** モードを使用すると、here:name は name と等価です。スコープは名前の参照方法を指示します。

構文では、コロンを添えたスコープ演算子を使用します。

```
scope:name
```

スコープには、次のいくつかの種類があります。

- 解決ルールを示すもの。たとえば、here:x は、x がスクリプトにローカルな名前として解決されるべきであることを意味します。Global:x は、x がグローバル名として解決されるべきであることを意味します。
- 名前空間の参照変数。たとえば、ref:a は、a が、ref が参照する名前空間で解決されるべきであることを意味します。
- 名前を列名として参照するデータテーブル参照。たとえば、dt:height は、height が、dt が参照するデータテーブルの列として解決されるべきであることを意味します。
- 独自に作成した名前空間の名前。たとえば、myNamespace:b。ここで、myNamespace は作成した名前空間。"myNamespace":b も等価です。詳細については、「名前空間」（237ページ）を参照してください。

列計算式の適用範囲指定の例

以下の例は、計算式を含んだ列の適用範囲の指定方法を示しています。スクリプトの中で、*x* はグローバル変数、ローカル変数、列名として使用されています。

最初のスクリプトでは、列名xは適用範囲が指定されていません。2番目の列内の計算式は、列**x**の値を100倍するものです。この場合、列の値は100、200、300という結果になります。

```
::x = 5;
New Table( "テスト",
   New Column( "x", Values( [1, 2, 3] ) ),
   New Column( "y", Formula( 100 * x ) ),
);
```

次のスクリプトでは、列**y**の計算式はローカル変数**x**に500を割り当てた後、**x**に50を足します。列の各セルの値は550になります。

```
::x = 5;
New Table( "テスト",
   New Column( "x", Values( [1, 2, 3] ) ),
   New Column( "y", Formula( Local( {x = 500}, x + 50 ) ) ),
);
```

定義済みスコープ

JMPには、削除や置換ができない定義済みスコープが用意されています。各スコープには、関連するオブジェクトに応じて、特定のルールがあります。

表8.5 定義済みスコープ

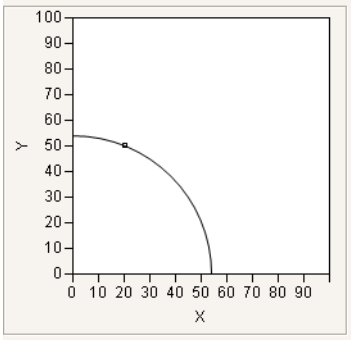
スコープ	説明
Global	グローバル名はJMP環境全体で共有される。
Here	実行スクリプトを範囲としたスコープ。
Builtin	JMPビルトイン関数。たとえば、Builtin:Sqrt()。名前は、JMP環境全体で共有されます。 JSL関数をカスタム関数で上書きした場合でも、このスコープを使ってビルトインのJSL関数にアクセスできます。
Local	ローカルスコープに似たスコープ。Function()、Local()、Parameter()の関数内で使用できます。
Local Here	Names Default to Here(1)内に名前空間ブロックを提供する。ローカルブロックには寿命があるのでLocal({Default Local},)は常に機能するとは限りませんが、Local Here()は呼び出し全体にわたって維持されます。
Window	含まれているユーザ定義ウィンドウのスコープ。(まれ)
Platform	現在のプラットフォームを範囲としたスコープ。(まれ)
Box	ContextBox()を範囲としたスコープ。ContextBoxはユーザ定義ウィンドウ内で使用されます。(まれ)

Windowスコープの使用例

この例は、Windowスコープを使って、実行中に情報を渡します。変数xおよびyの適用範囲をウィンドウに指定することで、xとyはデータテーブルなどの他のコンテキストに適用されなくなります。変数xおよびyは、指定のWindow環境内だけで作成および使用されます。Windowスコープは、Local()を使用するのと似ていますが、それより便利です。なぜなら、Local()は使用できる範囲が限られているからです。

```
New Window( "例",
   window:gx = 20;
   window:gy = 50;
   Graph Box(
      Frame Size( 200, 200 ),
      Handle(
         window:gx,
         window:gy,
         Function( {x, y},
            window:gx = x;
            window:gy = y;
         )
      );
      Circle( {0, 0}, Sqrt( window:gx * window:gx + window:gy * window:gy ) );
   );
);
```

図8.1 現在のウィンドウ名前空間の例



Hereスコープの使用例

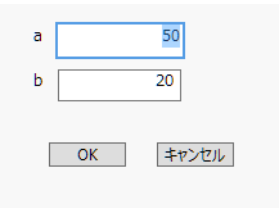
この例は、Hereスコープを使い、同じスクリプトによって作成されたウィンドウ間で情報を渡します。Here:を使って変数の適用範囲を指定する際、Names Default To Here()がオンである必要はありません。Here:スコープは常に使用できます。

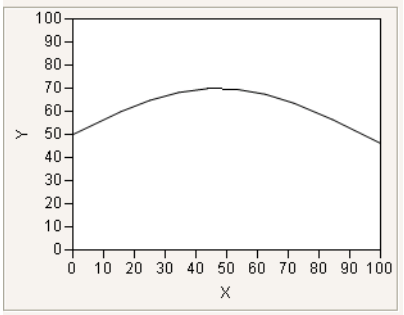
次のスクリプトは、2つのウィンドウを作成し、2つの異なるスコープを使用します。

「起動」ウィンドウがユーザに2つの値を入力するよう求めます。その2つの値が「結果」ウィンドウに渡され、それらの値を使って関数がグラフ化されます。「起動」ウィンドウは、aBoxとbBoxの適用範囲をそのウィンドウに限定します。基本的に、それらの変数（Number Edit Boxesへの参照）は「起動」ウィンドウ内にのみ存在し、「結果」ウィンドウでは使用できません。その後、2つのボックスの値がHereに範囲指定された変数内にコピーされ、このスクリプトによって作成された両方のウィンドウで使用可能となります。

```
launchWin = New Window( "起動",
   <<Modal,
   V List Box(
      Lineup Box( N Col( 2 ), Spacing( 10 ),
         Text Box( "a" ),
         window:aBox = Number Edit Box( 50 ),
         Text Box( "b" ),
         window:bBox = Number Edit Box( 20 ),

      ),
      Lineup Box( N Col( 2 ), Spacing( 20 ),
         Button Box( "OK",
            // ウィンドウを閉じる前に値をコピーする
            here:a = window:aBox << Get;
            here:b = window:bBox << Get;
         ),
         Button Box( "キャンセル", Throw( 1 ) )
      ),

   )
);

New Window( "結果",
   Graph Box( Y Function( here:a + here:b * Sin( x / 30 ), x ) )
);
```

図8.2 起動と結果





名前空間

名前空間とは、一意の名前およびそれに対応する値の集まりです。名前空間への参照を変数に格納できます。JMP には名前空間マップが1つしかないので、名前空間の名前はグローバルです。名前空間参照は、オブジェクトを参照する他の変数と同様の変数です。それぞれの適用範囲または名前空間の中で一意でなければなりません。名前空間のメンバーは、:スコープ演算子を使って参照されます。たとえば my_namespace:x は、my_namespace という名前空間の中の x という名前のオブジェクトを参照します。独自の名前空間の作成方法および管理方法については、「ユーザ定義による名前空間の関数」（237 ページ）を参照してください。名前空間は、異なるスクリプト間での名前の競合を回避するのに特に便利です。

ユーザ定義による名前空間の関数

独自の名前空間を作成し、関連する変数および関数の定義セットを入れることができます。名前空間の管理に使用できる関数には、次のものがあります。

New Namespace

```
nsref = New Namespace( <"nsname">, <{ name = expr, ... }> );
```

これは、nsname という新しい名前空間を作成し、その名前空間への参照を戻します。引数はすべてオプションです。

nsname は、名前空間名の内部グローバルリストに格納される名前空間の名前です。nsname は、スコープ変数の接頭辞としても使用できます。この関数は名前空間への参照を戻します。また、スコープ変数参照の接頭辞としても使用できます。nsname を指定しなかった場合、名前空間が匿名となり、JMP によって作成された一意の名前が付けられます。Show Namespace() は、すべての名前空間とそれらの名前（指定名／匿名に関わらず）を表示します。

重要: nsname で指定した名前の名前空間がすでに存在する場合は、それが上書きされます。つまり、スクリプトの開発時に、名前空間をクリアまたは削除しなくても、スクリプトを変更して再実行できます。間違って上書きしてしまうのを避けるには、匿名の名前空間を使用するか、または、次のようにして、特定の名前空間がすでに存在しているかどうかを確認します。

```
If( !Namespace Exists( "nsname" ), New Namespace( "nsname" ) );
```

名前空間を作成する際の、名前付き式のリストはオプションです。name は、名前空間内でのみ存在する JMP 変数です。

メモ: 名前付き式は、カンマで区切ったリストで指定する必要があります。セミコロンで区切ったリストは無視されます。

複数のユーザ定義の名前空間が使用される場合の競合を避けるため、名前空間には一意の名前を指定する必要があります。名前空間を匿名にすると、競合が回避されます。

名前空間

```
nsref = Namespace( "nsname" | nsref);
```

名前空間参照を戻します。引数は次のいずれかを取ります。

- 名前空間を含んだ引用符付き文字列
- 名前空間への参照

メモ: Namespace() はすでに存在する名前空間への参照を戻します。新しい名前空間は作成しません。

Is Namespace

```
b = Is Namespace( nsref );
```

*nsref*が名前空間の場合は1（真）、そうでない場合は0（偽）を戻します。

As Scoped

```
b = As Scoped( "nsname", var_name);
nsname:var_name;
```

As Scoped() はスコープ参照の関数形です。この関数は、指定の適用範囲にある指定の変数への参照を戻します。

Namespace Exists

```
b = Namespace Exists( "nsname" );
```

*nsname*がグローバル名前空間のリスト内に存在する場合は1（真）、そうでない場合は0（偽）を戻します。

Show Namespaces

```
Show Namespaces();
```

グローバル名前空間のリストに含まれるすべての名前空間の中身を表示します。名前空間は、New Namespace 関数または Namespace 関数のどちらかを使って参照されるまで表示されません。

名前空間へのメッセージ

名前空間は、管理のための関数だけでなく、その内容にアクセスして操作するための一連のメッセージにも対応します。

これらのメッセージは、他のすべてのメッセージと同様に、スクリプト可能なオブジェクトに送る必要があります。名前空間名は、定義済みのスクリプト可能なオブジェクトではないため、Send 操作で使用できません。ただし、名前空間の参照として使用することができます。たとえば、nsmane::var は nsref::var と等価です。

表8.6に、ユーザ定義の名前空間参照に対応するメッセージの定義を示します。

表8.6 名前空間へのメッセージ

名前空間へのメッセージ	説明
`ns << Contains( "var_name" );`	*var_name*が名前空間内に存在すれば1、そうでなければ0を戻す。
`ns << Delete;`	内部のグローバルリストからこの名前空間を削除する。 名前空間内の変数を削除するには、<<Removeを使用します。この表の<<Removeの項目を参照してください。
`ns << First;`	名前空間内で使用されている最初の変数名を引用符付き文字列として戻す。
`ns << Get Contents;`	名前と値のペアのリストを、それぞれ含んだリストとして戻す。名前は変数名を含んだ引用符付き文字列で、各値は変数に含まれている式（未評価）です。
`ns << Get Keys;`	名前空間内の変数名のリストを戻す。
`ns << Get Name;`	名前空間の名前を戻す。
`ns << Get Value( "var_name" );`	名前空間内の*var_name*に含まれている式を、評価せずに戻す。
`ns << Get Values;`	名前空間内の各変数に含まれている式を、評価せずにリストとして戻す。
`ns << Get Values( { "var_name1", "var_name2", ... } );`	リスト引数で指定された、名前空間内の各変数に含まれている式を、評価せずにリストとして戻す。指定の変数名が見つからない場合、エラーを戻します。
`ns << Insert( "var_name", expr );`	指定の式（*expr*）を含む指定の名前の変数（*var_name*）を、名前空間に挿入する。
`ns << Lock;` `ns << Unlock;`	名前空間内のすべての変数をロックし、変数が追加または削除されるのを防ぐ。<<Unlockは、名前空間の変数すべてのロックを解除します。
`ns << Lock( <"var_name", ...> );` `ns << Unlock( <"var_name", ...> );`	名前空間内で指定の変数をロックする。変数を指定しなかった場合、すべての変数がロックまたはロック解除されます。
`n = ns << N Items;`	名前空間に含まれている変数の数を戻す。
`next k = ns << Next( "var_name" );`	指定に変数に続く変数の名前を戻す。
`ns << Remove( "var_name", ...);`	指定の変数または変数のリストを削除する。

名前空間参照の使用

次に示すものはすべて、nsrefという参照を持つnsnameという名前の名前空間にある、bという名前の変数への参照です。

```
nsref:b
nsname:b
"nsname":b
nsref["b"]
nsref<<Get Value("b") // 右辺値として使用される
```

名前空間とインクルードされたスクリプト

インクルードされたスクリプトは、親スクリプトの名前空間内で実行されます。インクルードされたスクリプトに独自の名前空間が定義されている場合、次のいずれかを行う必要があります。

- 名前の競合を回避するため、名前空間の名前を管理する
- New Namespace関数によって作成される匿名の名前を使用する

どちらの場合も、名前空間への変数参照を管理する必要があります。

Include関数には別のオプション（New Context）があります。これは名前空間を作成し、インクルードされたスクリプトをその中で実行します。この名前空間は匿名で、親スクリプトの名前空間から独立しています。例:

```
Include("file.jsl", <<New Context);
```

この匿名の名前空間は、Hereを使って参照できます。

親と含まれているスクリプトの両方がグローバル名前空間を使用する場合は、名前の競合を避けるために、New ContextオプションにNames Default To Hereを追加します。例:

```
Include("file.jsl", <<New Context, <<Names Default To Here);
```

Include関数の詳細については、「Include」（251ページ）を参照してください。

ユーザ定義の名前空間の例

式を使った基本的な名前空間の作成および使用

ここでは、匿名の名前空間を作成し、その中で関数および変数を使用する方法を例示します。

```
new_emp = New Namespace(
   {name_string = "Hello, *NAME*!",

   print_greeting = Function( {a},
      Print( Substitute( new_emp:name_string, "*NAME*", Char( a ) ) )
   )}
);
```

名前空間内に定義する変数には、必ず完全修飾名を使用します。

```
new_emp:print_greeting( 6 );
  "Hello, 6!"
```

複素数の演算

この例では、まず複素数を表す2元リストをサポートするような関数を含む名前空間を作成し、次にその名前空間をロックします。

```
If( !Namespace Exists( "complex" ),
   New Namespace(
      "complex"
   );
   complex:makec = Function( {a, b},
      Eval List( {a, b} )
   );
   complex:addc = Function( {a, b}, a + b );
   complex:subtractc = Function( {a, b}, a - b );
   complex:multiplyc = Function( {a, b},
      Eval List( {a[1] :* b[1] - a[2] :* b[2], a[1] :* b[2] + a[2] :* b[1]} )
   );
   complex:dividec = Function( {a, b},
      d = b[1] ^ 2 + b[2] ^ 2;
      Eval List(
         {a[1] :* b[1] - a[2] :* b[2] / d, a[2] :* b[1] - a[1] :* b[2] / d}
      );
   );
   complex:charc = Function( {a},
      Char( a[1] ) || "+" || Char( a[2] ) || "i"
   );
);
Namespace( "complex" ) << Lock;
```

次に、このユーザ定義の名前空間に含まれる関数の使用例を示します。

```
c1 = complex:makec( 3, 4 );
c2 = complex:makec( 5, 6 );
cadd = complex:addc( c1, c2 ); // {8, 10} を戻す
csum = complex:subtractc( c1, c2 ); // {-2, -2} を戻す
cmul = complex:multiplyc( c1, c2 ); // {-9, 38} を戻す
cdiv = complex:dividec( c1, c2 ); // {14.6065573770492, 19.7049180327869} を戻す
Show( complex:charc( c1 ) ); // complex:char(c1) = "3+4i"; を戻す
cm1 = complex:makec( [1, 2, 3], [4, 5, 6] ); // {[1, 2, 3], [4, 5, 6]} を戻す
```

名前空間とスコープの参照

名前付き変数参照の解決方法を左右する要因は数多くあります。表 8.7 に、名前付き変数参照が特定の状況でどのように解決されるかを示します。

メモ:

- a、:a、::aは、**Names Default To Here**モードをオフにした場合、JMP 9以降でも同義です。
- 現在の実行ポイントがユーザ定義の関数内、またはLocal 関数やParameter関数のモデル定義部分にある場合、`Default Local`を使用すると、Local 名前空間が使用されます。

表8.7 名前空間参照

形式	参照タイプ	参照ルール	作成ルール
a	非修飾	**Names Default To Here**モードがオンの場合、次の場所で変数を探します。 • `Local`名前空間 • `Here`名前空間 • 現在のデータテーブル **Names Default To Here**モードがオフの場合、次の場所で変数を探します。 • `Local`名前空間 • `Here`名前空間 • `Global`名前空間 • 現在のデータテーブル	• **Names Default To Here**モードがオンの場合、`Local`名前空間内または`Here`名前空間内に変数を作成する。 • **Names Default To Here**モードがオフの場合、`Local`名前空間内または`Global`名前空間内に変数を作成する。
:a	現在のデータテーブル	現在のデータテーブル内で変数を探す。	（該当なし）
::a Global:a	グローバル	`Global`名前空間内で変数を探す。	`Global`名前空間内に変数を作成する。
ns:a dt:a Here:a "name":a expr:a	修飾	指定された名前空間内で変数を探す。変数が見つからない場合、エラーが戻されます。	指定された名前空間内に変数を作成する。ぞれ以前の値は上書きされます。

表8.7 名前空間参照

形式	参照タイプ	参照ルール	作成ルール
`ns["a"]` `ns[expr]`	添え字	指定された名前空間内で変数を探す。変数が見つからない場合、エラーが戻されます。	指定された名前空間内に変数を作成する。それ以前の値は上書きされます。
`Platform:a`	修飾	プラットフォームに含まれている変数を探す。	プラットフォーム内に変数を作成する。
`Local:a`	修飾	ローカル関数内において、該当する変数を探す。なお、他の関数を呼び出している場合、呼び出しに使われている引数も、範囲に含みます。詳細については、「Local:aの例」（244ページ）を参照してください。	変数が指定されたローカル関数内に、変数を作成する。なお、他の関数を呼び出している場合、呼び出しに使われている引数も、範囲に含みます。
`Window:a`	修飾	`New Window`のwindow名前空間内で、変数を探す。	`New Window`のwindow名前空間内に変数を作成する。
`Box:a`	修飾	`New Window`内に指定された`Context Box`の中で、変数を探す。	`New Window`内に指定された`Context Box`の中に変数を作成する。

Local:a の例

サンプルスクリプト	ログ出力
`Delete Symbols();` `Local( {d1l1 = 12},` `  local:f1f1 = Function( {fa1, fa2},` `  {fl1 = 99},` `  local:fa12 = fa1 + fa2;` `  Local( {d2l1 = 56},` `  local:l2l2 = 78;` `  Show( fa12 );` `  Show( fl1 );` `  Try( Show( d1l1 ), Write(` `"\!n\!n***Error=" || Char( exception_msg` `) || "\!n" ) );` `  Show Symbols();` `  );` `  local:fa12;` `  );` `  f1f1( 2, 3 );` `);`	*fa12 = 5;* *fl1 = 99;* ****Error={" 名前が解決できません :* *d1l1"(1, 2, "d1l1", d1l1 /*###*/)}* *// Local* *d2l1 = 56;* *l2l2 = 78;* *// 2 Local* *// Local* *fa1 = 2;* *fa12 = 5;* *fa2 = 3;* *fl1 = 99;* *// 4 Local* *// Local* *d1l1 = 12;* *// 1 Local* *// Global* *exception_msg = {" 名前が解決できません :* *d1l1"(1, 2, "d1l1", d1l1/*###*/)};* *// 1 Global* *5*

名前付き変数参照の解決

JMP スクリプト内で変数が参照されると、JMP は指定のルールセットを使用して変数の格納場所を解決します。変数が修飾名で参照された場合、その修飾の設定に基づいて解決されます。変数が非修飾名で参照された場合は、解決方法が少し複雑です。JMP は、実行スクリプトを使用して、実行ポイントを表す適用範囲の階層内を探します。ここでは、名前付き変数参照の解決に使用されるルールについて説明します。

JMP 9以降でも、変数名のデフォルトの解決方法がJMP 8以前と同様なので、既存のJSL スクリプトは同様に実行されます。JMP 9以降では、修飾のある名前参照とそうでない名前参照の違いを理解することが重要です。

修飾のある名前参照

修飾のある名前参照は、: 演算子および :: 演算子を使って、参照する変数の場所やその作成場所に関する特定の情報を提供します。次に示すのは、修飾のある名前参照の例です。

```
:var
::globalvar
datatable:var
nsref:var
"nsname":var
```

修飾のない名前参照

修飾のない名前参照では、変数の場所や作成場所が明示的に特定されません。参照にはスコープ演算子（: または ::）の指定がありません。修飾のない名前参照を解決する際のJMP の動作を変更するには、Names Default To Here(1) 関数を使用します。変数名の解決に関する詳細は、「JSL の構成要素」章の「名前解決のルール」（91 ページ）を参照してください。

変数参照の解決のルール

JMP では、変数参照の解決に、次のようなルールを（記載の順序で）使用します。

1. 変数の後ろに1組の丸括弧 ( ) が付いている場合、関数とみなして探します。
2. 変数の接頭部に : スコープ演算子が付いている場合、または明示的なデータテーブルへの参照がある場合、データテーブル列またはテーブル変数とみなして探します。
3. 変数の接頭部に :: スコープ演算子が付いている場合、グローバル変数とみなして探します。
4. 変数が明示的なスコープ参照（group:vowel など）の場合は、ユーザ定義の group 名前空間内を探します。
5. 変数が Local 関数または Parameter 関数の中にある場合は、ローカル変数とみなして探します。変数に添え字がある場合（変数が入れ子になっている場合）には、展開できなくなるまで、最後まで展開します。
6. 変数がユーザ定義の関数の中にある場合は、関数の引数またはローカル変数とみなして探します。
7. 現在のスコープおよびその親スコープ内を探します。Here スコープに到達するまで繰り返します。
8. Here スコープ内の変数とみなして探します。
9. グローバル変数とみなして探します。

10. スクリプトの冒頭に Names Default to Here(1) がある場合は、探すのを中止します。そのスコープはローカルです。
11. データテーブル列またはテーブル変数とみなして探します。
12. 演算子またはプラットフォーム起動名（一変量の分布、二変量の関係、チャートなど）とみなして探します。
13. 名前が見つからなかった場合
    - 名前が参照されている場合、ログにエラーを出力します。
    - 名前が割り当ての対象（L-value）として使われている場合は、次を確認します。

      変数の接頭部に :: スコープ演算子が付いている場合、グローバル変数を作成して使用します。

      変数が明示的なスコープ参照の場合、指定された名前空間またはスコープの中に変数を作成して使用します。

      スクリプトの冒頭に Names Default to Here(0) がある場合は、グローバル変数を作成します。

      スクリプトの冒頭に Names Default to Here(1) がある場合は、Here 名前空間変数を作成します。

高度なスクリプトを作成する際のベストプラクティス

グローバル名前空間の汚染を最小化し、スクリプトの相互作用を回避する

スクリプトを常に次の行で開始します。

```
Names Default To Here(1);
```

スクリプト間で変数を共有する

名前付き名前空間を使用します。名前空間の名前はグローバルスコープに置かれます。

匿名の名前空間を使用する

匿名の名前空間への名前空間参照を使用すると、他の名前空間との競合を回避できます。

高度なプログラミングの概念

ここでは、複雑なスクリプトの作成に役立つ、より高度なプログラミング技術について説明します。

例外のスローとキャッチ

Throw() 関数を実行することにより、そのスクリプト自体を停止できます。スクリプトがエラー状態に入ってしまったときにその部分から抜け出すためには、Try() 関数でそのスクリプトを囲んでおきます。

Try 関数は、2 つの式を引数としてとります。Try 関数はまず 1 つ目の式を評価します。1 つ目の式が Throw() の評価によって例外を戻したときは、次を実行します。

1. 評価をそこですぐに停止する。
2. 何も戻さない。
3. 2 番目の式を評価する。

Throw には引数は必要ありませんが、オプションで文字列を引数として指定することができます。引数を指定した場合、Throw が実行されると、*exception_msg* という名前のグローバル変数にその文字列が格納されます。これについては、次の最初の例で示します。

例

たとえば、Try() と Throw() を使うと、For ループの入れ子の中から抜け出ることができます。

```
a = [1 2 3, 4 5 ., 7 8 9];
b = a;
nr = N Row( a );
nc = N Col( a );
// a[2, 3 ] = 2; // "Missing b" という結果を表示させる場合は、この行のコメントを外す

Try(
   sum = 0;
   For( i = 1, i <= nr, i++,
      For( j = 1, j <= nc, j++,
         za = a[i, j];
         If( Is Missing( za ),
            Throw( "Missing a" )
         );
         zb = b[j, i];
         If( Is Missing( zb ),
            Throw( "Missing b" )
         );
         sum += za * zb;
      )
   );
,
   Show( i, j, exception_msg );
   Throw();
);
  i = 2;
  j = 3;
  exception_msg = "Missing a";
```

Tryと Throwを使って、JMPそのものが出す例外を受け取ってスクリプトを停止させることもできます。

```
Try(
   dt = Open( "Mydata.jmp" ); // 開くことができないファイル
   Summarize( a = by( 年齢 ), c = count, meanHt = Mean( :Name("身長(インチ)") ) );
   Show( a, c, meanHt );
,
   Print( "このスクリプトは、データセットがないと動作しません" );
   Throw();
);
```

Throwを使うために、必ずしもTryを使う必要はありません。次の例では、Tryによって Throwを受け取るのではなく、自らスクリプトを停止させます。

```
dt = New Table(); // 空のデータテーブルを作成
If( N Row( dt ) == 0,
   Throw( "!空のデータテーブル" )
);
```

Function（関数）

JSLでは、ローカル変数を引数とした関数を作成することもできます。関数は、Function関数によって作成されます。関数の作成は、マクロの概念を拡張したものと言えます。例として、平方根を求める関数を作成します。その関数では引数が負の場合はエラーではなくゼロを戻すようにします。まず、中括弧{}を使ってローカル引数のリストを指定してから、式を直接入力します。Functionで指定した式はその場では評価されず、式として保存されるため、Exprで式を囲む必要はありません。

```
myRoot = Function( {x},
   If( x > 0, Sqrt( x ), 0 )
);
a = myRoot( 4 );  // aは2となる
b = myRoot( -1 ); // bは0となる
```

関数は値と同じように、変数に保存されます。このため、Rootという関数とRootという変数の両方を持つことはできません。また、関数それ自体の内部にいるときを除き、いつでも関数を再定義できることも意味しています。

関数は呼び出されると、引数を評価し、第1引数のリストで指定されているローカル変数にその引数を与えます。次に関数の本体、つまり第2引数を評価します。

引数の値は、関数の一時使用のためのものです。関数が終了すると、値は捨てられます。戻される唯一の値は、戻り値です。複数の値を戻したい場合は、1つの値ではなく、リストにして戻すようにしてください。

関数を定義した場合、関数内部の式そのものにアクセスすることはできません。これはName Exprコマンドを使用しても同様です。関数内部の式にアクセスするには、その関数全体をexpr()節の中に入れる必要があります。

例:

```
makeFunction = Expr(
   myRoot = Function( {x},
      If( x > 0, Sqrt( x ), 0 )
   )
);
d = Substitute( Name Expr( MakeFunction ), Expr( x ), Expr( y ) );
Show( d );
makeFunction;
```

オプションの引数

オプションの引数を作成することもできます。例:

```
f1 = Function( {x, y, z}); // すべての引数が必須
f2 = Function( {x, y=2, z=4}); // x が必須、y と z はオプション
```

次の点を念頭に置いてください。

- 引数をオプションにするには、関数を定義するときにその引数にデフォルト値を指定します（x ではなく x=1）。
- オプションの引数は必須の引数の後に指定する必要があります。たとえば、次の例は使用できません。

  ```
  Function( {x = 1, y}, ... )
  ```

  この関数を定義するときにはエラーは発生しませんが、指定したデフォルト値は使用されないため、呼び出す際 x と y に値を渡さなければエラーとなります。
- この関数を呼び出すとき、たとえオプションの引数であっても、途中のものを飛ばしてその後ろのものを指定することはできません。次の例で z の値を指定するには、必ず y の値も指定する必要があります。

  ```
  ex = Function( {x, y = 2, z = 3},
     Return( x + y + z )
  );
  ex( 1, 4 ); // x に 1、y に 4 が渡され、z は 3 になるため、8 を戻す
  ```

ローカルシンボル

変数が関数のローカルであることを宣言し、グローバルシンボルスペースに影響されないようにすることができます。これは特に再帰関数で便利です。再帰関数では、関数呼び出しの各レベルでローカル変数の値を分けておく必要があるためです。

前述したように、関数を定義するには次のように指定します。

```
functionName=Function({arg1, ...}, body);
```

次のように指定すると、適用範囲が明示されていない名前すべてをローカル変数として扱うようになります。

```
functionName=Function({arg1, ...}, {Default Local}, body);
```

Default Local は、以下のような名前をローカルにします。

- グローバルとして適用範囲が指定（たとえば ::name）されていない名前
- データテーブルとして適用範囲が指定（たとえば :name）されていない名前
- それらの後に括弧なしで続く名前（たとえば name(...) という形式でないもの）

次の例では、3つの数字を合計する関数を定義しています。

```
add3 = Function( {a, b, c},
   {temp},
   temp = a + b;
   temp + c;
);
X = add3( 1, 5, 9 );
  15
```

次の関数も同じ処理をしますが、ローカル変数を自動的に設定します。

```
add3 = Function( {a, b, c},
   {Default Local},
   temp = a + b;
   temp + c;
);
X = add3( 1, 5, 9 );
  15
```

どちらの場合も、変数 temp はグローバル変数ではなく、たとえ関数の外で同じ名前の変数がグローバル変数として設定されていても、そのグローバル変数は関数の実行に影響されません。

メモ: 保存された式の中にローカルな変数を指定した場合、その変数名はコンテキストにより、グローバルな領域ではグローバル変数の名前として使用されます。一方、グローバル変数として明示的に指定した変数は、グローバル変数だけを指します。

ユーザ定義の関数内で Default Local() を使用した場合、コンテキストによって動作が異なるため、混乱を招く場合があります。つまり、同名の変数がスコープの中と外に存在する場合、同じ関数でも動作が異なる可能性があります。ユーザは、ローカルにしたい変数をすべて列挙する必要があります。そうすれば、スコープの外にある変数の値についての混乱や誤りを最小限に抑えることができます。

Recurse（再帰）

Recurse() 関数は、定義した関数を再帰的に呼び出します。たとえば、Recurse 関数を使うと、階乗を計算する関数が作れます。階乗とは、ある数から、その数が1になるまで1ずつ引いていったときのすべての数の積を指します。

```
myfactorial = Function( {a},
   If( a == 1,
      1,
```

```
        a * Recurse( a - 1 )
    )
);
myfactorial( 5 );
  120
```

Recurse() を使わなくても、再帰的計算を定義することはできます。たとえば、Recurse() を myfactorial に置き換えても、このスクリプトは機能します。ただし、Recurse() には次のような利点があります。

- 関数と同じ名前を持ったローカル変数がある場合の競合を回避できます。
- 関数自体に名前が付けられていない場合でも（たとえば、前述の *myfactorial* など、グローバル変数に割り当てられている場合）、再帰的計算ができます。

Include

Include() 関数は、スクリプトファイルを開き、その中のスクリプトを解析し、JSL を指定のファイル内で実行します。

```
Include( "pathname" );
```

例：

```
Include( "$SAMPLE_SCRIPTS/myStartupScript.jsl" );
```

ファイルを評価するのではなく、ファイルから式だけを取得するオプションがあります。

```
Include( "pathname", <<Parse Only );
```

名前空間を作成し、インクルードされたスクリプトがその中で実行されるようにするための名前付きオプションもあります。この名前空間は匿名の名前空間で、親スクリプトの名前空間から独立しています。

```
Include( "file.jsl", <<New Context );
```

スクリプトでの名前空間の使用方法については、「高度な適用範囲指定と名前空間」（230 ページ）を参照してください。

Include で読み込むファイルに関しては、次のことに注意してください。

- JSL ファイル以外の JMP ファイルは使用できません。
- その他、JMP で認識されるファイル（イメージファイル、SAS データセット、Microsoft Excel ファイルなど）も使用できません。
- 認識できない種類のファイルは、JSL ファイルとして扱われます。
- 拡張子が .txt のファイルは、JSL ファイルとして扱われます。データを含むテキストファイルは読み込まれますが、有効な JSL ではないためエラーメッセージが表示されます。

テキストファイルのロードと保存

Load Text File()およびSave Text File()コマンドを使うと、JSLでテキストファイルを操作することができます。以下のコードで、pathは文字列を表します。

```
text = Load Text File( "path" );
Save Text File( "path", text );
```

Webサイトから、テキストファイルを読み込むこともできます。

```
Load Text File( "URL", <blob> );
```

上の指定において"URL"は、テキストファイルのURLを含む引用符付き文字列です。テキストファイルは文字列として戻されます。オプションの名前付き引数*blob*を追加した場合は、BLOB（Binary Large OBject）が戻されます。

ファイルやディレクトリの操作

Pick Directory()やPick File()を使用すると、スクリプトの実行時にファイルやディレクトリを選択できます。ディレクトリに含まれているファイルのリストを取得するには、Files In Directory()を使用します。

ディレクトリまたはファイルの選択

Pick Directory()関数を使って、ユーザにディレクトリの選択を促すことができます。このコマンドは、ユーザがフォルダを選択できる、プラットフォーム固有のウィンドウを表示します。Windowsの場合、オプションのプロンプト文字列を指定すると、「ディレクトリの選択」ダイアログの最上部にその文字列が表示されます。また、画面に表示する最初のディレクトリのパスを指定したり、ディレクトリ選択ウィンドウにファイルを表示するかどうかを指定したりできます。

```
path = Pick Directory ( "ディレクトリを選択してください。" );
```

ユーザがファイルを選択できるようにするには、Pick File()関数を使用します。

```
path = Pick File(
   <"prompt message">, <"initial directory"> <{filter list}>,
   <first filter>, <save flag>, <"default file">,
   <multiple>);
```

"prompt message"はウィンドウのタイトルとして使用されます。"initial directory"は、最初に表示するフォルダを指定します。ディレクトリを空の文字列として定義した場合は、デフォルトのディレクトリが使用されます。

Open()ウィンドウに使用する{filter list}を定義し、特定のタイプのファイルだけを表示することもできます。このリストには、次の構文を使用します。

```
{"Label1|suffix1;suffix2;suffix3", "Label2|suffix4;suffix5"}
```

各引用符付き文字列が、Open() ウィンドウの **File name** リストのエントリになります。Label は、各メニューオプションに表示されるテキストを定義します。次の接尾辞のリストは、対応するラベルが選択されている場合に表示されるファイルの種類を定義します。"*" を使用すると、すべての種類のファイルがウィンドウに表示されます。

```
path = Pick File(
   "JMP ファイルを選択してください ", // 操作を促すメッセージ
   "$SAMPLE_DATA", // 最初に表示するディレクトリ
   {"JMP ファイル |jmp;jsl;jrn", " すべてのファイル |*"}, // ファイルフィルタリスト
   1, // 最初に選択された項目
   0, // ファイルの保存操作をユーザに促さない
   "Analgesics.jmp" // デフォルトで選択されるファイル
);
```

ヒント: すべての引数はオプションですが、位置は固定です。つまり、省略が可能なのは、スクリプトの末尾のオプションのみとなります。それ以外のオプションを省略したい場合は、空の文字列を指定します。

下のスクリプトは、デフォルトのディレクトリもデフォルトのファイルも指定していません。

```
path = Pick File(
   "JMP ファイルを選択してください ",
   "", // デフォルトのディレクトリはない
   {"JMP ファイル |jmp;jsl;jrn", " すべてのファイル |*"},
   1,
   0,
   "" // デフォルトのファイルはない
);
```

<first filter>引数には、デフォルトの選択肢を指定します。n はリスト項目のインデックスです。上のスクリプトでは、<first filter> がリストの最初の項目、つまり "JMP Files|jmp;jsl;jrn" になっています。

<Save Flag> が偽の場合は、Multiple 引数を追加して、ユーザが 1 つのウィンドウで複数のファイルを選択できるように設定できます。

```
path = Pick File(
   "JMP ファイルを選択してください ",
   "",
   {"JMP ファイル |jmp;jsl;jrn", " すべてのファイル |*"},
   1,
   0,
   "",
   multiple // save flag が 0、複数ファイルの選択が可能
);
```

メモ: コンピュータの物理メモリ内のバッファサイズは、ユーザが開くことのできるファイルの数に影響します。バッファが小さいほど、開くことのできるファイルは少なくなります。

ファイル名リストの取得

特定のディレクトリのファイル名リストを取得するには、Files In Directory コマンドを使います。

```
names = Files In Directory( path, <recursive> );
```

次の例のように、ファイル名とサブディレクトリ名の両方が戻されます。

```
names = Files In Directory( "$SAMPLE_DATA" );
  { "2D Gaussian Process Example.jmp", "Abrasion.jmp", ... "Design Experiment",
    "Detergent.jmp", ... }
```

この場合、Design Experimentなどのサブディレクトリ内のファイルは含まれず、$SAMPLE_DATAの直下にあるファイルしかリストされていません。

サブディレクトリ内のファイルも含む、すべてのファイル名のリストを戻すには、Files In Directoryにオプションの引数recursiveを追加します。

```
names = Files In Directory( "$SAMPLE_DATA", recursive );
  { "2D Gaussian Process Example.jmp", "Abrasion.jmp", ... "Design
    Experiment/2x3x4 Factorial.jmp", "Design Experiment/Borehole Factors.jmp",
    ... }
```

ファイルのフルパス名を取得するには、ディレクトリ内を再帰的に参照し、ファイルパスとファイル名を連結します。次の例は、$SAMPLE_DATAディレクトリとサブディレクトリの中の各ファイルを繰り返し処理します。各ファイル名にファイルパスが連結されます。

```
names = Files In Directory( "$SAMPLE_DATA", recursive );
For( i = 1, i <= N Items( names ), i++,
   names[i] = Convert File Path( "$SAMPLE_DATA" ) || names[i]
);
names;
  {"/C:/Program Files/SAS/JMP/13/Samples/Data/2D Gaussian Process Example.jmp",
  "/C:/Program Files/SAS/JMP/13/Samples/Data/Abrasion.jmp", ... }
```

Files in Directoryコマンドでは、ネイティブ形式のパス、POSIXパス、およびパス変数を使用できます。パスの操作の詳細については、「データタイプ」章の「パス変数」（119ページ）を参照してください。

BYグループを使ったスクリプト

Byグループ引数に対応する関数は、ColMean()、ColStdDev()、ColNumber()、ColNMissing()、ColMinimum()、ColMaximum()です。

BY引数はいくつでも指定可能です。また、式を指定することもできます。BY引数は、必ず列計算式またはForEachRow()のコンテキスト内で使用します。第1引数には、一般的な数値式を指定することもできます。

以下はその例です。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
```

```
dt << New Column( "男女別の平均身長", Numeric, Formula( Col Mean( :Name("身長(インチ)"), :性別 ) ) );

dt << New Column( "男女別、年齢別の最小身長",
   Numeric,
   Formula( Col Minimum( :Name("身長(インチ)"), :性別, :年齢 ) )
);

dt << Distribution( Continuous Distribution( Column( :Name("身長(インチ)") ) ),
   By( :性別 ) );

dt << Tabulate(
   Show Control Panel( 0 ),
   Add Table(
      Column Table(
         Analysis Columns( :Name("身長(インチ)")),
         Statistics( Mean, N, Std Dev, Min, Max, N Missing )
      ),
      Row Table( Grouping Columns( :年齢, :性別 ) )
   )
);
```

スクリプトの暗号化と暗号解読

スクリプトに基本レベルの保護を付加するために、暗号化して、パスワードを知っている人だけが見たり実行したりできるようにできます。これは、スクリプトの共有を管理したい場合に便利です。

スクリプトを暗号化するには

1. 暗号化するスクリプトを開きます。
2. ［**編集**］>［**スクリプトの暗号化**］を選択します。
3. ユーザがスクリプトを見るのに必要となる、暗号解読のためのパスワードを入力します。
4. （オプション）実行パスワードを入力します。ユーザは、このパスワードを入力しなければ暗号化されたスクリプトを実行できません。

   メモ：パスワードには 1 バイトの文字を使用しなければなりません。IME（Input Method Editor）を使ったテキストの入力は無効です。

5. ［**OK**］をクリックします。
6. 暗号解読パスワードのみを指定した場合、［**はい**］をクリックして、実行パスワードを指定しないことを確認します。

   暗号化されたスクリプトが、新しいウィンドウに表示されます。

例:

```
//-e6.0.2
S@FTQ;VGMUTF?J<;LS;B<=IRLXCU=BV;@NS<TW;LR<ZFOP=JJS>NNDA@T<V><DZA>SU@MG;LR<ZFOP=JJ
   S>NNDA@T<V><DZA>SU@MG;LR<ZFOP=JJS>NNDA@T<V><DZA>SU@MG;LR<ZFOP=JJS>NNDA@T<HNIZ;W
   DN?RMJ;FR>KYAXTEPPF?;XFJJOP=RQGBIAGXOYNNZ>PLIF>SW>L>ACL<KGP;=QQTCEG??U<PUXLV?TR
   BO?J>QGWTJCFJA@BNHWLVORNNGQYPIKL<IM<>JX>@G?LJ>=;RBODH@PTKK@SIUE;IJOR<TUTRMTGSYR
   SVGOR<XK<F=IWQYE=LVZFP;AUHA?YJLL;EIT?ZJZC;*
```

7. 暗号化されたスクリプトを保存します。

JSLスクリプトを暗号解読するには、次のようにします。

1. 暗号化されたスクリプトをJMPで開きます。
2. ［**編集**］>［**スクリプトの暗号解読**］を選択します。
3. 暗号解読パスワードを入力し、［**OK**］をクリックします。

   暗号解読されたスクリプトが新しいウィンドウに表示されます。

暗号化されたJSLスクリプトを実行するには

メモ: 暗号化されたスクリプトを実行する前に、スクリプトを実行する対象となるデータテーブルを把握しておく必要があります。データテーブルの名前がわからない場合は、実行する前にスクリプトを暗号解読して、データテーブルの名前を確認しておく必要があります。

1. 暗号化されたスクリプトをJMPで開きます。
2. ［**編集**］>［**スクリプトの実行**］を選択します。
3. 実行パスワードを入力し、［**OK**］をクリックします。

   スクリプトは次のように実行されます。

   – スクリプトがデータテーブルを参照している場合、データテーブルを開くように求められます。データテーブルを開くと、スクリプトが実行されます。

   – スクリプトが空のデータテーブルを必要とする場合は、まずデータテーブルを作成します。その後、暗号化されたスクリプトを実行します。

実行パスワードを入力した場合、スクリプトは実行されますが、表示はされません。スクリプトを表示するには、暗号解読パスワードを入力する必要があります。

暗号化とグローバル変数

暗号化だけでは、グローバル変数とそれらの値を隠すことはできません。Show Globals()コマンドを使えば、それらは通常どおり表示されます。暗号化スクリプト内のグローバル変数を隠す必要がある場合は、それらに特別な名前を付けます。

名前が2つの下線（__）で始まるグローバル変数は、すべて非表示となります。Show Globals() では名前も値も表示されません。

例:

```
myvar = 2;
__myvar = 5;
Show Symbols();
  // Globals
  myvar = 2;
  // 2 Global (1 表示しない)
```

この方法は、スクリプトが暗号化されているかどうかに関わりなく有効です。

データテーブルのスクリプトの暗号化

データテーブルに保存されたスクリプトは、JSL Encrypted() 関数または Include(Char to Blob()) 関数を使って暗号化することもできます。

- JSL Encrypted() の方が、1つの関数を含むだけなので簡単です。暗号化スクリプトの中にコメントを含めることもできます。
- Include(Char to Blob()) では、コメントを含めることはできますが、スクリプトの中ではありません。

メモ: 暗号化スクリプト内の列計算式は暗号化されません。これらの列計算式を暗号化するには、JSL Encrypted() 関数内に列計算式を含めます。

データテーブルのスクリプトを暗号化するには、次の手順に従います。

1. スクリプトをスクリプト編集ウィンドウに配置します。

   すでにデータテーブルに保存されているスクリプトを直接暗号化することはできません。
2. 「スクリプト」ウィンドウで、［**編集**］>［**スクリプトの暗号化**］を選択します。
3. 暗号解読パスワードを入力します。
4. （オプション）実行パスワードを入力します。ユーザは、このパスワードを入力しなければ暗号化されたスクリプトを実行できません。
5. 暗号解読パスワードのみを指定した場合、［**はい**］をクリックして、実行パスワードを指定しないことを確認します。

   暗号化されたスクリプトが、新しいスクリプトウィンドウに表示されます。
6. 暗号化されたスクリプト全体をコピーします。
7. 新しいデータテーブルスクリプトを作成するか、既存のスクリプトを開きます。
8. ウィンドウ内のスクリプトの部分に、次のいずれかの関数を入力します。

   ```
   JSL Encrypted( "" );
   ```

   ```
   Include( Char to Blob( "" ) );
   ```

9. 関数の引用符の中に暗号化スクリプトを貼り付けます。
10. [OK] をクリックします。

図8.3 暗号化されたデータテーブルスクリプトの例


```
JSL Encrypted(
    "//-e6.0.2
WE@GSACGT<?CKEG=NG;B<=IRLXCU=BV;@NS<TW;LR<ZFOP=JJS>NNDA@T<V><DZA
>SU@MG;LR<ZFOP=JJS>NNDA@T<V><DZA>SU@MG;LR<ZFOP=JJS>NNDA@T<V><DZA
>SU@MG;LR<ZFOP=JJS>NNDA@T<FFHG=R;GYNBNME>ZBMIB?O;G<>?;UTG@=HVYCC
@MFYCMVQEOTNWASNOEV=Y<TCELCQHW>AFY>;PGSSF<IBIU?H<YLQABWEGSQBMLHW
BJE?HUC=BTP?;?;LVTT>ZQI=IXTJ<P=IHYJIPAC?PGZH;OCCMXPOJHLIRHERSW;V
;@GZDHDNF><YCN@WLBTOFZRAILT?L;NT<OOCLLLWYAIVCSPBMVL???KK=TBZJ@KU
BD>=@;EOUFMUUFLSP<HMZWL?VLHXKK?S@WAFXC>EZODQOHREMNYKRBKAWS=PSXSQ
CRYFOPVJRBNHIVLE?BNPVNM=LO>BAT<?CQF?JNNQAUEHMM;BTU><=WFVOO<;WVKB
;RSBEXTS;QGBEKNDCANPFP;DYDHTSMQNAGVESB;TCWJKGWNBG>JBESVNKOVOBPBE
;?ZPYS;X?=VE=>YFHD;;;*"
)
```


暗号化スクリプト内の列計算式の暗号解読

列計算式が暗号化スクリプトによって作成された場合は、その計算式も暗号化されています。計算式を表示したい場合は、Eval (Parse) 関数を使用してその計算式だけを暗号解読できます。たとえば、次のスクリプトが暗号化されている場合、「体重 (ポンド)」の計算式は計算式エディタ内で暗号化されます。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << New Column( "Encrypted",
      Numeric,
      Continuous,
      Format( "Best", 12 ),
      Formula( :Name("体重 (ポンド)") * 2 )
   );
```

ただし、「Encrypted」の計算式を暗号解除しておきたい場合は、次のスクリプトを入力します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << New Column( "Not Encrypted",
      Numeric,
      Continuous,
      Format( "Best", 12 ),
      Formula( Eval( Parse( ":Name(\"体重 (ポンド)\") * 2" ) ) )
   );
```

その他の数値演算子

JSL では、計算式エディタではあまり意味をなさない、関数の行列計算や数値微分も実行できます。微分については、以下で詳しく説明します。

行列に対しても、足し算や引き算などの基本的な演算を、行列をオペランドとして実行することができます。また、行列には、要素ごとの掛け算、要素ごとの割り算、行列同士の連結、要素の指定といった行列固有の演算子もいくつかあります。詳細は、「データ構造」章の「行列」（167ページ）の節を参照してください。

微分

JSLには、微分を求めるための関数は3つあります。Derivative、NumDeriv、およびNumDeriv2です。これらの関数は、計算式エディタでは用意されていません。

Derivative関数は、第2引数に指定した変数で、第1引数に指定された式を微分した結果を戻します。この第2引数には1つの変数を指定することも、リストにして複数の変数を指定する（つまり、中括弧{}で複数の変数名を囲む）こともできます。

メモ: Derivative関数と同様の機能は、計算式エディタではコマンドとして用意されています。このコマンドは、計算式エディタの上部中央のドロップダウンリストにあります（キーパッドの上）。このコマンドを使用するには、式の中の1つの変数を強調表示（どの変数についての導関数を求めるかを指定）してから、メニューから［**微分した式**］コマンドを選びます。計算式全体が、強調表示された変数で微分した導関数に置き換わります。

スクリプトでこの関数を最も簡単に使用する方法は、変数を1つ指定する方法です。この場合は、導関数のみが戻されます。

$f(x) = x^3$ の場合、第1次導関数は $f'(x)$ または $\frac{d}{dx}x^3 = 3x^2$ です。

```
result = Derivative( x ^ 3, x );
Show( result );
  result = 3 * x ^ 2
```

ある1つの式から複数の変数に対する導関数を効率的に求めたい場合は、リストに複数の変数を指定してください。**一時変数に分割された**元の式と、その導関数の式を含むリストが結果として戻されます。必要に応じて、導関数で用いる部分式が、**一時変数**に割り当てられます。

次の例は、3つの変数が含まれている式の例です。3つの変数をリストに入れると、それぞれの変数に対する第1次導関数を戻します。結果は、元の式と、要求した順に並べた導関数を含むリストになります。ただし、この例において、JMPが式x^2を格納するための一時的な変数T#1を作成し、この一時的な変数を使って計算を省略していることに注意してください。

```
result2 = Derivative( 3 * y * x ^ 2 + z ^ 3, {x, y, z} );
Show( result2 );
  result2 = {3 * y * (T#1 = x ^ 2) + z ^ 3, 6 * x * y, 3 * T#1, 3 * z ^ 2}
```

第2次導関数をとるには、第3引数としてもう1つ変数を指定します。第2引数と第3引数は、どちらもリストでなければなりません。JMPは、元の式、第1次導関数、そして第2次導関数の順番で結果を含んだリストを戻します。

```
second = Derivative( 3 * y * x ^ 2, {x}, {x} );
```

```
Show( second );
  second = {3 * y * x ^ 2, 6 * x * y, 6 * y}

second = Derivative( 3 * y * x ^ 2, {y}, {y} );
Show( second );
  second = {3 * y * (T#1 = x ^ 2), 3 * T#1, 0}

second = Derivative( 3 * y * x ^ 2, {y}, {x} );
Show( second );
  second = {3 * y * (T#2 = x ^ 2), 3 * T#2, 6 * x}
```

Num Deriv()は、数値微分を行うための関数です。第1引数と、それに小さい値（Δ）を足した値に対する関数値を計算し、その差をΔで割ることにより、1次微分の値を求めます。Num Deriv2()は、同様の方法で2次微分の値を数値的に求めます。これらの関数は、非線形モデルで内部的に使用されますが、JSLで直接利用することは少ないでしょう。これらの関数は変数では微分せず、関数に対する引数についてのみ微分することに注意してください。xについて微分するためには、xを、式の内部に埋め込まれている記号ではなく、関数の中の引数の1つにする必要があります。

$x=3$で、$y=3*x^{\wedge 2}$を微分するとします。このとき次のスクリプトを実行すると、**間違い**になります。

```
x = 3;
n = Num Deriv( 3 * x ^ 2 );
```

xを関数の中の引数にするのが、**正しい**方法です。

```
x = 3;
f = Function( {x}, 3 * x ^ 2 );
n = Num Deriv( f( x ), 1 );
  18.000029999854
```

数式による表記と、JSLでの表記方法を例を挙げて検討してみましょう。

$f(x)=x^2$の場合、$\frac{\mathrm{d}}{\mathrm{d}x}x^2=\frac{(x+\Delta)^2-x^2}{\Delta}$という計算になります。$x_0=3$では、$\frac{\mathrm{d}}{\mathrm{d}x}x^2=6.00001$です。

```
x = 3;
y = Num Deriv( x ^ 2 ); // または、y = Num Deriv( 3 ^ 2 );
  6.0000099999513
```

もう少し例を挙げます。

```
x = Num Deriv( Sqrt( 7 ) ); // 0.188982168980445 を戻す
y = Num Deriv( Normal Distribution( 1 ) ); // 0.241969514669371 を戻す
z = Num Deriv2( Normal Distribution( 1 ) ); // -0.241969777547979 を戻す
```

表8.8 微分を行う関数

関数	構文	説明
Derivative	Derivative(*expr*, {*name*, ...})	*name*に対する*expr*の導関数を戻す。第2引数は、中括弧{}でリストにして指定するか、1つだけの場合は単純に変数として指定できます。2つの導関数をとるときは、2つの名前リストを指定します。
NumDeriv	NumDeriv(*expr*)	式（*expr*）の最初の引数で、1次微分した結果（1階導関数の値）を数値的に求める。
NumDeriv2	NumDeriv2(*expr*)	式（*expr*）の最初の引数で、2次微分した結果（2階導関数の値）を数値的に求める。

代数的な処理

JSLには、逆関数を代数的に求めるInvert Expr()関数が用意されています。

```
Invert Expr(expression, name, y)
```

この式で、

- *expression*には逆関数を求めたい式、もしくは、その式を含むグローバル変数を指定します。
- *name*について解かれた逆関数が求められます。
- *y*は元の式の従属変数の変数名です。

例:

```
Invert Expr( Sqrt( log( x ) ), x, y );
```

この結果は、変数xについて解かれた次のような逆関数になります（なお、Invert Expr()関数は、変数xが、元の式に1箇所だけで使われている必要があります)。

```
Exp( y ^ 2 )
```

これは、次のように手計算で代数的に求めるのとまったく同じように行われます。

$y = \mathrm{Sqrt}(\log(x))$;

$y^2 = \mathrm{Log}(x)$;

$\mathrm{Exp}(y^2) = x$;

Invert Expr関数は、逆関数が存在する大部分の基本演算をサポートしています。また、必要に応じて、正の平方根の中だけで、三角関数の場合は可逆な定義域内だけで逆関数を求めます。

F、Beta、Chi Square、t、Gamma、およびWeibullの*Distribution*と*Quantile*関数がサポートされています。第1引数に関しての逆関数である分布関数と分位点関数が戻されます。変換できない式の場合は、Invert Expr()関数はEmpty()を戻します。

JSLには、整理されていない複雑な式を、さまざまな代数的規則を用いて簡略化する、Simplify Exprコマンドがあります。このコマンドの使用方法は、以下のとおりです。

```
result = Simplify Expr(expr(expression));
```

または

```
result = Simplify Expr(nameExpr(global));
```

例:

```
Simplify Expr( Expr( 2 * 3 * a + b * (a + 3 - c) - a * b ) );
```

の結果は、次のようになります。

```
6*a + 3*b + -1*b*c
```

Simplify Expr()はまた、入れ子構造のIf式を展開します。例:

```
r = Simplify Expr( Expr( If( cond1, result1, If( cond2, result2, If( cond3,
  result3, resultElse ) ) ) ) );
```

の結果は、次のようになります。

```
If(cond1, result1, cond2, result2, cond3, result3, resultElse);
```

最大化と最小化

Maximize()関数およびMinimize()関数は、式を最適化（最大化または最小化）する因子の値を求めます。式は、因子の値に対して連続関数でなければいけません。

この関数を呼び出す方法は、次のとおりです。

```
result = Maximize(objectiveExpression,{list of factor names}, <<option(value))
result = Minimize(objectiveExpression,{list of factor names}, <<option(value))
```

*objectiveExpression*は、最適化する対象の式です。式そのもの、または式が保存されているグローバル変数名のどちらかを指定します。

{*list of factor names*}には、*objectiveExpression*に含まれる因子名のリストを指定します。

因子名の後に、name(lowerBound,upperBound)のように、許容値の範囲を指定することもできます。

許容値の上限または下限だけを指定する場合は、以下の例のように片側を欠測値にします。

```
{beta} // 制限なし
{beta (0,1)} // 0以上1以下
{beta (.,1)} //1以下
{beta (0,.)} or {beta (0)} // 0以上
```

因子の値は、数値または行列のいずれかです。

また、以下のオプションも使用できます。デフォルト値は、以下のとおりです。

```
<< Tolerance(.00000001) // 収束基準
<< Max Iter( 250) // 最大反復数
<< Limits() //
```

関数を呼び出す前に、因子には初期値が割り当てられているものとします。

これらの関数は、複数の局所解を持つ関数において、大域的な最適解を見つけるとは限りません。指定された初期値から、ある局所的な最適解（問題によっては、大域的な最適解かもしれませんが）を求めることしかできません。

現在のところ、戻り値は目的関数の値です。

最小2乗法の例

Minimizeを使用して、指数モデルにおいて最小2乗法の推定値を求める例を示します。サンプルデータのフォルダ内にある「Nonlinear Examples」の「US Population.jmp」データテーブルのデータを使用します。

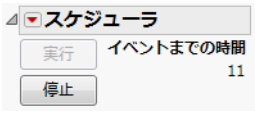
```
xx = [1790, 1800, 1810, 1820, 1830, 1840, 1850, 1860, 1870, 1880, 1890, 1900, 1910,
  1920, 1930, 1940, 1950, 1960, 1970, 1980, 1990];
yy = [3.929, 5.308, 7.239, 9.638, 12.866, 17.069, 23.191, 31.443, 39.818, 50.155,
  62.947, 75.994, 91.972, 105.71, 122.775, 131.669, 151.325, 179.323, 203.211,
  226.5, 248.7];
b0 = 3.9;
b1 = .022;
sseExpr = Expr(
   Sum( (yy - (b0 * Exp( b1 * (xx - 1790) ))) ^ 2 )
);
sse = Minimize( sseExpr, {b0, b1}, <<Tolerance( .00001 ) );
Show( b0, b1, sse );
  b0 = 13.9991388055261;
  b1 = 0.0147104409355048;
  sse = 1862.14141218875;
```

スクリプト実行のスケジューリング

Schedule()関数を使うと、指定した秒数後にスクリプトを実行できます。

```
Schedule( 15, Print( "こんにちは" ) );
```

図8.4 JMPスケジューラ



「スケジューラ」ウィンドウには、次のイベントまでの時間が表示されます。また、スケジュールを再開するためのボタン（［実行］）や停止するためのボタン（［停止］）があります。このウィンドウのポップアップメニューには［スケジュールの表示］がコマンドあり、現在のスケジュール内容をログに出力します。たとえば、上の「こんにちは」の例の実行中に何度かスケジュールを確認すると、次のような表示が見られます。

```
スケジュール時間 11.55000000000018 :Print("こんにちは")
スケジュール時間 4.716666666666697 :Print("こんにちは")
スケジュール時間 3.083333333333485 :Print("こんにちは")
```

このスクリプトは、格納された式を参照する名前でもかまいません。たとえば、次のようにそのスクリプト自体を呼び出すスクリプトを実行してみましょう。

```
quickieScript = Expr(
   Show( "やあ、どうも" );
   Schedule( 15, quickieScript );
);
quickieScript;
```

このスクリプトは、15秒後にログウィンドウに文字列「やあ、どうも」を表示し、そのさらに15秒後に同じスクリプトが実行されるように再スケジュールします。これは［停止］ボタンがクリックされるまで続きます。

また、プロダクションの設定として、次のようなスケジュールを設定することもあるでしょう。

```
FifteenMinuteCheck = Expr(
   Show( "データのチェック" );
   Open( "my file", options... );
   distribution( Column( column1 ), capability( spec limits ) );
   Schedule( 15 * 60, FifteenMinuteCheck );
);
FifteenMinuteCheck;
```

Schedule()はイベントキューを開始します。イベントがキューに置かれると、スクリプト内の次のステートメントの実行に移ります。たとえば、次の例のような結果になることもあります。

```
Schedule( 3, Print( "いち" ) );
Print( "に" );
  "に"
  "いち"
```

イベントが終了するまで、それ以降のスクリプトが実行されないように待機させる方法の1つに、Wait( )を使って適当な停止時間を設定する方法があります。別の方法としては、以降の処理をスケジュールのキューに入れます。Schedule()では、たくさんのイベントを連続的にキューに入れられるようになっています。複数のイベントをスケジュールに設定した場合、各イベントは、個別に呼び出されます。各イベントの実行時間はすべて、「今」から（または、［**実行**］がクリックされてから）の時間で指定してください。たとえば、以下の複数のイベントは、12秒間ではなく5秒間ですべて終了します。

```
Schedule( 3, Print( "こんにちは" ) );
Schedule( 4, Print( "、世界のみなさん" ) );
Schedule( 5, Print( "-- さようなら" ) );
```

スケジュールキュー上のイベントをすべてキャンセルするには Clear Schedule を使います。

```
scheduler[1] << Clear Schedule( );
```

メモ: Schedule を使ってマルチスレッドで動作させることはできません。

表 8.9 スケジュールのコマンド

メッセージ	構文	説明
Schedule	*sc*=Schedule(*n*, *script*)	*n* 秒後に指定のスクリプト（*script*）を実行するというイベントをキューに入れる。
Clear Schedule	*sc*<<Clear Schedule()	スケジュールのキューにあるイベントを、すべて停止する。

メッセージを出力する関数

Show()、Print()、および Write() は、ログウインドウにメッセージを表示します。Speak()、Caption()、Beep()、および StatusMsg() は、ユーザに情報を与える方法を提供します。Mail() では、プロセス管理者に警告の電子メールを送ることができます。

JMP のスクリプト言語を使うと、ユーザにデータ列の選択やその他の情報入力を促すためのダイアログボックスを作成できます。詳細については、「表示ツリー」章の「モーダルウィンドウ」（496 ページ）を参照してください。

ヒント： Show()、Print()、Write() の出力でロケール特有の数値の表示形式を使用するには、<<Use Locale(1) オプションを含めます。

ログへの書き込みを行う

Show

Show() は、指定の項目をログに表示します。変数を表示させる場合、結果のメッセージは、変数名、等号、および変数の現在の値となります。

```
X = 1;
A = "Hello, World";
Show( X, A, "foo" );
  x = 1
  A = "Hello, World";
  "foo"
```

Print

Print() は、指定のメッセージをログに送ります。Print() は、変数名と等号なしで変数の値だけを表示する点を除けば、Show() と同じです。

```
X = 1;
A = "Hello, World";
Print( X, A, "foo" );
  1
  "Hello, World"
  "foo"
```

Write

Write() は、指定のメッセージをログに送ります。Write() と Print() の違いは、Write() では文字列の両側の引用符が省略されることと、改行させるにはエスケープシーケンスの \!N を挿入する必要があるという点です。

```
Write( " メッセージがあります。" );
  これがメッセージです。

Write( myText || " ミルクを買うのを忘れないで。" ); // 連結するには || を使用
Write( "\!N それからパンも。" ); // 改行するには \!N を使用
// 改行するには \!N を使用
  メッセージがあります。ミルクを買うのを忘れないで。
  それからパンも。
```

シーケンス \!N は、ホスト環境に応じた復帰文字および改行文字を挿入します。3つの改行エスケープシーケンスについては、「JSL の構成要素」章の「二重引用符」（81 ページ）を参照してください。

ユーザに情報を送る

Beep

Beep() は、警告音を鳴らします。

Speak

Speak() は、テキストを読み上げます。Macintosh では、Speak() にブール値（1 または 0）をとるオプションの Wait() があり、読み上げの終了まで次のステップへ進むのを待つかどうかを指定できます。デフォルトでは待たない設定になっているため、待つようにするには毎回 Wait(1) を送る必要があります。たとえば、ここによく意味のわからない文を読み上げるスクリプトがあるとします。Wait(1) を使えば言葉は聞き取れるので、すぐに実行を中断したくなるはずです。しかし、Wait(1) ではなく Wait(0) を使った場合、繰り返しの方が読み上げが完了するより早く処理されるため、結果として変な音が出力され、一体何が起こっているのかもわかりません。Windows では、Wait(n) コマンドが同じ機能を果たします。

```
For( i = 99, i > 0, i--,
   Speak(
```

```
   Wait( 1 ),
   Char( i ) || " bottles of beer on the wall, " || Char( i ) || " bottles of
beer; " ||
   "If one of those bottles should happen to fall, " || Char( i - 1 ) || "
bottles of beer on the wall."
) );
```

もっと実用的な例としては、JMP に 60 秒ごとに時間を読み上げさせるというものがあります。

```
script = Expr(
   tod = Mod( Today(), In Days( 1 ) );
   hr = Floor( tod / In Hours( 1 ) );
   min = Floor( Mod( tod, In Hours( 1 ) ) / 60 );
   timeText = "time, " || Char( hr ) || " " || Char( min );
   text = Long Date( Today() ) || ", " || timeText;
   Speak( text );
   Show( text );
   Schedule( 60, script );    // 次のスクリプトまでの秒数
);
script;
```

同じような方法を使って、プロセスが制御不可能になったことをJMPから警告させることもできます。

Caption

Caption()は、小さなウィンドウを開き、ユーザへのメッセージを表示します。この機能により、結果のウィンドウに余分なオブジェクトを加えることなしに、デモに注釈をつけられます。第1引数の{h,y}はオプションで、スクリーン上の位置を左上からのピクセル数で指定するものです。第2引数は、ウィンドウに表示するテキストです。位置の引数が省略された場合、ウィンドウは左上隅に表示されます。

読み上げを一定時間遅らせるには、名前付き引数Delayedと時間（秒）を指定します。この指定により作成される注釈と、**それ以降のすべての**注釈は、指定された秒数分遅れて表示されます。これは、別のCaptionコマンドで設定が更新されるまで続きます。遅れをすべてなくすには、Delayed(0)を使います。

フォントの種類、フォントのサイズ、テキストの色、または背景色は、次の引数で指定します。

```
Font( font );
Font Size( size );
Text Color("color");
Back Color("color");
```

Spokenオプションを指定すると、注釈が読み上げられます（OSに音声出力システムが組み込まれている場合）。Spokenはブール値（1または0）の引数をとり、Spoken引数を持つCaptionコマンドによって改めて変更されるまでは現在の設定（オンまたはオフ）が有効になります。

下のスクリプトでは、読み上げをオンにし、最後の注釈までオンのままにしています。最初のキャプションで、フォントの種類、色、背景色が指定されます。スクリプトを実行すると、フォントと色の設定が最初のキャプションだけに適用されることがわかります。

```
Caption(
   {10, 30},
   "A Tour of the JMP Analyses",
   Font( "Arial Black" ),
   Font Size( 16 ),
   Text Color( "blue" ),
   Back Color( "yellow" ),
   Spoken( 1 ),
   Delayed( 5 )
);
Caption( "Open a data table." );
bigClass = Open( "$SAMPLE_DATA/Big Class.jmp" );
Caption( "A data table consists of rows and columns of data." );
Caption( "The rows are numbered and the columns are named." );
Caption( {250, 50}, "The data itself is in the grid on the right" );
Caption(
   {5, 30},
   Spoken( 0 ),
   "A panel along the left side shows columns and other attributes."
);
```

新しいCaptionを指定するたびに、以前の注釈が非表示になります。つまり、一度に表示できる注釈ウィンドウは1つだけです。新しい注釈を表示せずにウィンドウを閉じるには、名前付き引数Removeを使います。

```
Caption( remove );
```

StatusMsg

このコマンドは、ステータスバーにメッセージを送信します。

```
StatusMsg( "string" );
```

Mail

Mail()は、ユーザに電子メールを送信します。たとえば、プロセス制御管理者は、管理図にテストの警告スクリプトを含めておいて、警告の電子メールが管理者に届くように指定できます。

```
Mail(
   "JaneDoe@company.com",
   " 管理外 ",
   Format( Today(), "d/m/y h:m:s" ) || " プロセス 12A が管理外 "
);
```

Mail()は、電子メールの添付ファイルも送ることができます。オプションとして4つ目の引数に添付ファイルを指定します。添付ファイルは、ディスク上に存在することが確認されると、バイナリー形式で送信されます。たとえば、「Big Class.jmp」データテーブルを添付するには、次のスクリプトを実行します。

```
Mail(
    "JohnDoe@company.com",
    " 興味深いデータセット ",
    " このクラスのデータをご覧ください。",
    "$SAMPLE_DATA\Big Class.jmp" );
```

メモ: Macintoshの場合、`Mail()`はMountain Lionおよびそれ以降のオペレーティングシステムで動作します。Mountain Lionでは、オペレーティングシステムの制約により、電子メールにアドレスと件名を入力する必要があります。[**送信**]ボタンをクリックして電子メールを送信します。

第9章

データテーブル
データテーブルオブジェクトの操作

データテーブルやデータテーブル内のオブジェクトを操作するには、まずデータテーブルを開き、そのオブジェクトに**参照**を割り当てる必要があります。このマニュアルでは、データテーブルオブジェクトの参照を**dt**と表記します。

JSLでオブジェクトを操作するには、そのオブジェクトを表す参照にメッセージを送り、タスクのどれかを実行するよう要求します。**メッセージ**とは、特定の状況のもとで、特定のタイプのオブジェクトだけが理解できるコマンド群のことです。たとえば、データテーブルオブジェクトへのメッセージには、Save、New Column、Sortなどがあります。これらのメッセージのほとんどは、たとえばプラットフォームオブジェクトなど、他のオブザベーションに対しては何の意味も持ちません。

この章では、以下の項目について解説します。

- データテーブルに参照を割り当てる方法
- 参照にメッセージを送り、指定のアクションを実行させる方法
- データテーブルオブジェクトが理解できるメッセージの種類

この章の大部分は、データテーブルや列、行、値などのデータテーブルオブジェクトに送ることができる各種メッセージに焦点を当てています。

ヒント： データテーブルの操作に使用できるJSLコマンドの大半は説明してますが、すべてを網羅しているわけではありません。包括的なリストについては、JMPの［スクリプトの索引］を参照してください。［**ヘルプ**］>［**スクリプトの索引**］を選択します。メニューから［**オブジェクト**］、次に［**データテーブル**］を選択します。

はじめに

ヒント： ログウィンドウを開いたままにして、実行したスクリプトの出力を確認しましょう。[**表示**] > [**ログ**] を選択し、ログウィンドウを表示します。詳細については、「スクリプト作成のツール」章の「ログの使用」（61 ページ）を参照してください。

データテーブル内の値を処理する場合、次のような手順が一般的です。

1. 使用したい値が入っているデータテーブルを、現在のデータテーブルとして設定する。すでにデータテーブルの参照がある場合は、その参照を使用することができます。
2. 使用したい値が入っている行または列を指定し、使用したい値が含まれている列名を指定します。

次の例では、「Big Class.jmp」データテーブルを開き（それにより、このデータテーブルが「現在のデータテーブル」になります）、「体重」列の行2を指定します。ログを見ると、123という値が戻されていることがわかります。これは、行2の「Louise」の体重です。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt:weight[2];
  123
```

データテーブルがすでに開いている場合は、次の例のいずれかを使用します。

```
dt = Data Table( "My Table" ); // すでに開いている「My Table」という名前のテーブル
dt = Current Data Table(); // アクティブなウィンドウにあるテーブル
dt = Data Table( 3 ); // 3番目に開かれたテーブル
```

参照のあるデータテーブルが開いたら、<<演算子か Send 関数を使ってデータテーブルにメッセージを送ります。次の例では、両方の使い方を示します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Save();
Send( dt, Save() );
```

メッセージについて、次の点に留意してください。

- メッセージは、どのデータテーブルオブジェクトにも送ることができます（データテーブル、列、行など）。
- データテーブルオブジェクトに送られるメッセージは、常に次のようなパターンを持ちます。

  ```
  参照 << メッセージ( 引数 );
  ```

- 通常は、参照を作成し、その参照にメッセージを送るのが最も簡単です。しかし、メッセージを1つしか送らない場合は、参照を使わずに直接送ることもできます。次の例は、現在のデータテーブルを保存します。

  ```
  Current Data Table() << Save();
  ```

- 一連のメッセージを 1 つのステートメントに連ねることもできます。コマンドは左から右へと評価され、各コマンドが、影響するオブジェクトの参照を戻します。次の例は、「My Table」という名前の新しいデータテーブルを作成し、そこに2つの列を追加した後、データテーブルを保存します。

```
dt = New Table( "My Table" );
dt << New Column( "Column 1" ) << New Column( "Column 2" ) << Save( "" );
```

この例では、各メッセージがデータテーブル（dt）の参照を戻します。

- メッセージの引数が足りないと、JMPは必要な情報の入力を求めるウィンドウを表示します。JMPは、スクリプトが不完全な場合によくウィンドウを表示します。JMPのこの動作は、ユーザによる入力が必要なスクリプトを作成するときに利用できます。
- 一部のメッセージは対になっており、一方は各属性を設定（set）または割り当て、もう一方は各属性の現在の設定を取得（get）または問い合わせます。

オブジェクトに送るコマンド（メッセージ）と単独で使うコマンドがある理由

New TableとOpenは、まだ存在しないオブジェクトを作成するためのコマンドです。オブジェクトが作成できたら、そのオブジェクトに変更を求めるメッセージを送ります。オブジェクトは自身を削除できないので、オブジェクトを閉じるには、オブジェクトの**コンテナ**を閉じる必要があります。

次の例は、テーブルを作成し、**dt**変数にデータテーブル参照を割り当てます。そのデータテーブルの参照にNew Columnメッセージを送ります。これらの列のいずれかを削除する場合は、列そのものではなく、データテーブルの参照にDelete Columnsメッセージを送ります。

```
dt = New Table( "Airline Data" );
dt << New Column( "Date" );
dt << New Column( "Airline" );
dt << Delete Columns( "Date" );
```

データテーブルオブジェクトに送ることができるメッセージ

データテーブルオブジェクトに送ることができるメッセージは、次のような手順で、［スクリプトの索引］で確認できます。

1. ［**ヘルプ**］>［**スクリプトの索引**］を選択します。
2. リストの中から［**すべてのカテゴリ**］を選択します。
3. リストの中から［**Data Table**］を選択します。

Show Properties()コマンドを使用することもできます。このコマンドは、メッセージのリストを印刷したい場合やコピーしたい場合に便利です。Show Properties()コマンドは、データテーブルに送ることのできるメッセージを「ログ」ウィンドウに一覧表示します。Show Properties()は、データテーブルや列など、スクリプト可能なオブジェクトを引数としてとるコマンドです。データテーブルのプロパティを表示するには：

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
Show Properties( dt );
```

結果のメッセージは、階層型のリストに表示されます。

[サブテーブル]は、JMPのメニューに表示されるコマンドセットを指します。たとえば、「Tables」というサブテーブルは、JMPの［テーブル］メニューを表します。このサブテーブルに、インデントされた形で表示されるメッセージは、［テーブル］メニューのコマンドに該当します。メッセージの多くに、短い説明がついています。

[アクション]のラベルがあるメッセージはすべて、何らかのアクションが実行されます。スクリプトでしか使用できないものには、[スクリプトの場合のみ]のラベルがつきます。ブール値の引数をとるメッセージには、[ブール値]のラベルがつきます。

［分析］メニューと［グラフ］メニューのプラットフォームがプロパティのリストに表示されているのは、プラットフォーム名をメッセージとしてデータテーブルに送れるからです。プラットフォーム名をメッセージとして送ると、そのデータテーブルに対して通常の起動ウィンドウが開きます。プラットフォームに関連するスクリプトの書き方については、「プラットフォームのスクリプト」章（373ページ）を参照してください。

```
dt << Distribution( Y( :Name("身長(インチ)") ) );
```

メモ: Show Propertiesは、データテーブルだけでなく、プラットフォームとディスプレイボックスにも使用できます。

これらのプロパティに関する詳細は、JMPの［スクリプトの索引］を参照してください。［スクリプトの索引］では、サンプルスクリプトの実行や変更もできます。［**ヘルプ**］>［**スクリプトの索引**］を選択し、［オブジェクト］リストの中でプロパティを探します。

データテーブルのスクリプトの基本

データテーブルオブジェクトを使用するには、まずデータテーブルを開くか、または新しく作成し、そのデータテーブルに参照を割り当てる必要があります。この節では、データテーブルに対して実行できる、命名、保存、サイズ変更などの基本的なアクションについて説明します。

データテーブルを開く

データテーブルを開くには、Open()関数を使用します。

- データテーブルの参照を戻さずに、ただデータテーブルを開くには:

```
Open( "$SAMPLE_DATA/Big Class.jmp" ); // データテーブルを開くのみ
```

- データテーブルを開いてその参照を維持するには:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" ); // 開いて参照を保存する
```

データテーブルのパスとしては、引用符で囲んだ文字リテラル（絶対パスまたは相対パス）か、パス名を戻す引用符のない式が使えます。相対パスは、.jslファイルの場所を基準として相対的に解釈されます（保存されたスクリプトの場合）。保存されていないスクリプトの場合、パスは、プライマリパーティション（Windowsの場合）または<*ユーザ名*>/Documentsフォルダ（Macintosh）との相対で解釈されます。

```
Open( "../My Data/Repairs.jmp" ); // Windows および Macintosh における相対パス
Open( "::My Data:Repairs.jmp" ); // Macintosh における相対パス
Open( "C:/My Data/Repairs.jmp" ); // 絶対パス
```

JMP には、ディレクトリやファイルにより簡単にアクセスする方法（**パス変数**）が用意されています。この方法では、ディレクトリやファイルの完全なパスを入力する代わりに、Open() 式にパス変数を含めます。たとえば、JMP のサンプルスクリプトで広く使われるパス変数の $SAMPLE_DATA は、Samples/Data フォルダにあるファイルを開きます。パス変数の詳細については、「データタイプ」章の「パス変数」（119 ページ）を参照してください。

データテーブルを開くたびに完全パスを入力する手間を省くには、ファイルパスの文字列を定義して、それをファイル名に連結します。

```
myPath = "C:/My Data/Store25/Maintenance/Expenses/";
Open( myPath || "Repairs.jmp" );
```

この方法でデータテーブルを開くと、開いたデータテーブルがメモリに格納されます。つまり、スクリプトで開けようとしているデータテーブルがすでに開いている場合は、コンピュータに保存されているデータテーブルが開く代わりに、開いた状態でメモリに格納されたデータテーブルが表示されます。

開いたデータテーブルのテスト

開いたデータテーブルに依存するスクリプトでは、テーブルが開いているかどうかを Is Empty() または Is Scriptable() を使ってテストします。次の例では、スクリプトが「Big Class.jmp」で二変量の分析を行った後、データテーブルを閉じます。スクリプトで後述されている一元配置分析に進む前に、If Scriptable() によって、開いたデータテーブルがテストされます。1（真）が戻された場合、テーブルは開いており、スクリプトは続行します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
obj = dt << Bivariate( Y( :Name("身長 (インチ)") ), X( :年齢 ), Fit Line );
Close( dt );
If( Not( Is Scriptable( dt ) ),
   dt = Open( "$SAMPLE_DATA/Big Class.jmp" ),
);
obj = dt << Oneway( Y( :Name("身長 (インチ)") ), X( :年齢 ), Means( 1 ), Mean
  Diamonds( 1 ) );
```

データテーブルに対して別のスクリプトを実行する前に、データテーブルが開いているかどうかをチェックし、開いている場合はいったんデータテーブルを閉じて任意の変更を破棄するとよい場合もあります。「Big Class.jmp」が開いている場合には閉じる、というスクリプトは、次のように記述できます。

```
Try( Close( Data Table( "Big Class" ) ) );
```

データテーブルを開くようユーザに促す

開いているデータテーブルがないときに、データテーブルを開くようユーザに促すには、If() 式を使用します。ユーザがテーブルを開かないときはスクリプトが終了するようにします。次のスクリプトは、その一例です。

```
dt = Current Data Table();
If( Is Empty( dt ),
   Try( dt = Open(), Throw( " データテーブルは開かれませんでした。" ) )
);
```

ユーザは、データテーブルを開くよう促されます。ユーザがデータテーブルを開かずに［**キャンセル**］をクリックした場合、ログにエラーが表示されます。

特定の列だけを表示する

データテーブルを開いて特定の列セットだけを表示するには、Open() 式の中でそれらの列を指定します。これは、大規模なデータテーブルのうち、一部の列だけが必要な場合に有効です。

次の例は「Big Class.jmp」を開き、「**年齢**」、「**身長 (インチ)**」、「**体重 (ポンド)**」の3列のみを含めます。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp", Select Columns( " 年齢 ", " 身長 ( インチ )", "
   体重 ( ポンド )" ) );
```

開いているデータテーブルのうち、指定の列だけを除外することもできます。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp", Ignore Columns( " 名前 ", " 性別 " ) );
```

新しいデータテーブルの作成

新しいデータテーブルを作成したり、新しいデータテーブルを作成してその参照をグローバル変数に格納したりできます。どちらの場合も、テーブル名を引数として指定します。

```
New Table( "My Table" );
```

または

```
dt = New Table( "My Table" );
```

以下の節で、New Table() 関数のオプションの引数について説明します。

Private、Invisible、Visible

visibility は、新しいデータテーブルが表示される場所を指定するオプションの引数です。visibility("invisible") はデータテーブルを非表示にします。そのデータテーブルは、JMP ホームウィンドウおよび［ウィンドウ］メニューにのみ表示されます。visibility("private") はデータテーブルを開くという動作を回避します。visibility("visible") はデータテーブルを表示します。"visible" はデフォルト値です。

```
New Table( "My Table", visibility( "private" ) );
```

テーブルを Private にすると、スクリプトが多くのテーブルを開いたり閉じたりする際に、処理のスピードを上げられる場合があります。Private に指定したデータテーブルは、物理的なウィンドウを持たないため、内部でウィンドウを作成する動作が不用となります。

アクション

新しいデータテーブルの定義に使えるオプションの引数です。たとえば、次のスクリプトは **Little Class** という名前の新しいデータテーブルを作成します。行を3つと名前のついた列を2つ追加し、テーブルの各セルに値を入力します。

```
dt = New Table( "Little Class",
   Add Rows( 3 ),
   New Column( "名前",
      Character,
      "Nominal",
      Set Values( {"KATIE", "LOUISE", "JANE"} )
   ),
   New Column( "身長 ( インチ )",
      "Continuous",
      Set Values( [59, 61, 55] )
   )
);
```

データの読み込み

テキストファイルやMicrosoft Excelファイルは、JMPに読み込む際、データテーブル形式に変換されます。Windowsでは、3文字のファイル拡張子に従ってファイルの種類とファイル内容の解釈方法が特定されます。次に、JMPによって識別されるファイル拡張子の例を挙げます。

```
Open( "$SAMPLE_IMPORT_DATA/Bigclass.xlsx" ); // Microsoft Excel ファイル
Open( "$SAMPLE_IMPORT_DATA/Bigclass.txt" ); // テキストファイル
Open( "$SAMPLE_IMPORT_DATA/Carpoll.xpt" ); // SAS 移送ファイル
```

Macintosh上では、Macintoshのタイプおよびクリエータのコード（もしあれば）に基づいて解釈し、2次的に3文字のファイル名拡張子を使います。ファイルをMacintoshに読み込む場合は、あらかじめファイル拡張子を追加することを忘れないでください。

- タイプおよびクリエータコードは、Finderが作成元のアプリケーションに対応する正しいアイコンでファイルを表示できるようにするための情報です。
- 一般アイコン（他のシステムから取得したファイルに表示されているもの）で表示されるファイルにはファイル名拡張子が付いています。

  ```
  Open( "$SAMPLE_IMPORT_DATA/Bigclass.txt", text );
  ```

その他に使えるフォーマットには、.csv、.jsl、.dat、.tsv、および.jrnがあります。

読み込みオプションの詳細については、『スクリプト構文リファレンス』の「JSL関数」章を参照してください。

テキストファイルからのデータの読み込み

［テキストデータファイル］の環境設定にある［読み込みの設定］では、テキストファイルの読み込み方法を指定します。たとえば、1行目を列名、2行目以降をデータとするのがデフォルトの設定です。別の設定を使用するには、Open()のオプションとして読み込みの設定を記述します。

デフォルトの読み込みの設定と独自の設定は、データテーブルの「ソース」スクリプトに保存されるため、データを同じ設定で再度読み込むことができます。ただし、このスクリプトでは、デフォルトの読み込みの設定はオプションです。

Open()には、次のようなオプションがあります。

```
CharSet("option")
   // "Best Guess"、"utf-8"、"utf-16"、"us-ascii"、"windows-1252"、"x-max-roman"、
   "x-mac-japanese"、"shift-jis"、"euc-jp"、"utf-16be"、"gb2312"
Number of Columns(Number)
Columns(colName=colType(colWidth),... )
   // colTypeはCharacter|Numeric
   // colWidthは列幅を指定する整数
Treat Empty Columns as Numeric(Boolean)
Scan Whole File(Boolean)
End Of Field(Tab|Space|Comma|Semicolon|Other|None)
EOF Other("char")
End Of Line(CRLF|CR|LF|Semicolon|Other)
EOL Other("Char")
Strip Quotes|Strip Enclosing Quotes(Boolean)
Labels | Table Contains Column Headers(Boolean)
Year Rule | Two digit year rule ("Decade Start")
Column Names Start | Column Names are on line(Number)
Data Starts | Data Starts on Line(Number)
Lines to Read(Number)
Use Apostrophe as Quotation Mark
CompressNumericColumns(Boolean)
CompressCharacterColumns(Boolean)
CompressAllowListCheck(Boolean)
```

次のスクリプトは、テキストがカンマで区切られているテキストファイルを開きます。このファイルには、列名は含まれていません。このスクリプトで、列名と列の幅を定義します。

```
Open(
   "$SAMPLE_IMPORT_DATA/EOF_comma.txt",
   End of Field( comma ),
   Labels( 0 ),
   Columns(
      name = Character( 12 ),
      age = Numeric( 5 ),
      sex = Character( 5 ),
      height = Numeric( 3 ),
```

```
        weight = Numeric( 3 )
    )
);
```

次の例では、フィールドの区切りにスペースが使われているテキストファイルを開きます。

```
Open(
    "$SAMPLE_IMPORT_DATA/EOF_space.txt",
    Labels( 0 ),
    End of Field( Space )
);
```

読み込みオプションを対話式に設定するには、Text Wizard引数を指定します。テキスト読み込みウィンドウにテキストファイルのプレビューが開きます。

```
Open( "$SAMPLE_IMPORT_DATA/EOF_space.txt", "Text Wizard" );
```

以下の節で、各引数について詳しく説明します。読み込みオプションの詳細については、『スクリプト構文リファレンス』の「JSL関数」章を参照してください。

列数／Number of Columns

ソースファイルの合計列数を指定します。データが明確に区切られていない場合に重要です。

列／Columns

これまでの例にあったように、Columns引数を使って列名、列の種類、列の幅を指定します。

ファイルの2列目以降に対して設定を行う場合は、それより前にある列もすべて設定する必要があります。たとえば、**「名前」**、**「性別」**、**「年齢」**、**「ID」**という4つの列がこの順番にあるテキストファイルを開きたいとします。**「年齢」**は数値列で、幅を5に設定します。その場合は、**「名前」**と**「性別」**の列についても種類と幅を設定し、同じ順番でリストする必要があります。

```
Columns(
    名前 = Character( 15 ),
    性別 = Character( 5 ),
    年齢 = Numeric( 5 )
);
```

後続の列（この例では**「ID」**）に対しては、設定を行う必要はありません。

データが読み込まれたら、列の尺度を使用します。詳細については、「データと尺度の設定または取得」（329ページ）を参照してください。

メモ：次の引数のほとんどは、JMPの環境設定で定義されています。環境設定を上書きしたいときは、次のうちで該当する引数を、読み込みスクリプトの中に含めます。

空の列を数値タイプとする／Treat Empty Columns as Numeric

欠測データの列を、文字データではなく数値として読み込みます。欠測値を示すフラグとして、ピリオド、Unicodeのドット、NaN、または空白文字列を使用できます。これはブール値です。デフォルト値は偽です。

ファイル全体をスキャン／Scan Whole File

列のデータタイプを特定する目的でJMPがどれくらいファイルをスキャンするかを指定します。これはブール値です。デフォルトの値は真で、データの種類が特定できるまでファイル全体をスキャンします。大きいファイルを読み込むときは、値を偽に設定すると、ファイルが5秒間だけスキャンされます。

引用符を取り除く／Strip Quotes | Strip Enclosing Quotes

文字列の値を囲む二重引用符（"）を含めるか、削除するかを指定します。これはブール値です。デフォルト値は、1（真）。

たとえば、フィールドがスペースで区切られているとします。

- 「**John Doe**」は2つの文字列と解釈されます（「**John**」と「**Doe**」）。
- 「"**John Doe**"」は1つの文字列と解釈されます。JMPを始めとする多くのプログラムは、引用符は読みますが、2つ目の引用符が現れるまでにある他の区切り文字は無視します。
- `Strip Quotes(1)`を含めると、「"**John Doe**"」は「**John Doe**」（引用符のない1つの文字列）として解釈されます。

多くのワープロには、二重引用符文字（"）を自動的に左右の（“ ”）に変換する機能があります。テキストファイルの読み込み時にJMPによって二重引用符が取り除かれる場合でも、この形式の二重引用符はそのまま文字として解釈されます。

行の終わり／End of Line（CR | LF | Semicolon | Other）

行を区切る文字を指定します。次のような選択肢があります。

- CR（復帰文字）Macintosh OSで作成されたテキストファイルで標準的に使用されている区切り文字。
- LF（改行）UNIXとMacintosh OS Xのテキストファイルで標準的に使用されている区切り文字。
- CRLF（復帰文字と改行の両方）Windowsのテキストファイルで標準的に使用されている区切り文字。

この3つの文字が、デフォルトで行の区切り文字として設定されています。

行の区切り文字を別の文字にするには、その他（Other）オプションを使用し、`EOLOther`引数を使って具体的に文字を指定します。JMPは、この文字とデフォルトの文字のいずれかを行の区切り文字と解釈します。

フィールドの終わり／End of Field（Tab | Space | Spaces | Comma | Semicolon | Other | None）

フィールドの区切りに使われる文字を指定します。次の点を念頭に置いてください。

- デフォルトのフィールド区切り文字はタブ（**Tab**）です。
- フィールドの区切り文字を別の文字にするには、その他（`Other`）オプションを使用し、`EOFOther`引数を使って具体的に文字を指定します。

- Spaceオプションは1つのスペースを区切り文字として使用し、
- Spacesオプションは複数のスペースを区切り文字として使用します。

EOF Other、EOL Other

フィールドまたは行の区切り文字を指定します。たとえば、EOLOther("*")は、テキストファイル内の行がアスタリスク（*）で区切られていることを示します。

ラベル | テーブルに列名を含む／Labels | Table Contains Column Headers (Boolean)

テキストファイルの最初の行が、列名として使用されているかどうかを指定します。これはブール値です。デフォルト値は、1（真）。

年表記ルール／2桁の年表記ルール／Year Rule|Two digit year rule

2桁の年の値の読み込み方を指定します。最も早い日付が1979年の場合は、「1970」と指定します。最も早い日付が2001年の場合は、「20xx」と指定します。

列名の開始 | 列名のある行／Column Names Start | Column Names Are on Line

列名の開始行を指定します。次の例では、テキストファイルの列名が第3行から始まることを指定します。

```
Open(
    "$SAMPLE_IMPORT_DATA/Animals_line3.txt",
    Columns(
        Column( "species", Character, "Nominal" ),
        Column( "subject", Numeric, "Continuous", Format( "Best", 10 ) ),
        Column( "miles", Numeric, "Continuous", Format( "Best", 10 ) ),
        Column( "season", Character, "Nominal" )
    ),
   Column Names Start( 3 )
);
```

データの開始 | データの開始行／Data Starts | Data Starts on Line

データの開始行を指定します。

次の例は、テキストファイルのデータが第5行から始まることを指定します。

```
Open(
   "$SAMPLE_IMPORT_DATA/Bigclass_L.txt",
   Columns(
      Column( "name", Character, "Nominal" ),
      Column( "age", Numeric, "Continuous", Format( "Best", 10 ) ),
      Column( "sex", Character, "Nominal" ),
      Column( "height", Numeric, "Continuous", Format( "Best", 10 ) ),
      Column( "weight", Numeric, "Continuous", Format( "Best", 10 ) )
   ),
   Data Starts( 5 )
);
```

読み込む行数／Lines to Read

データテーブルに含める行の数を指定します。JMPは、列名が読み込まれた後でカウントを開始します。

次の例は、最初の10行だけをデータテーブルに含めます。

```
Open(
    "$SAMPLE_IMPORT_DATA/Bigclass_L.txt",
    Columns(
        Column( "name", Character, "Nominal"),
        Column( "age", Numeric, "Continuous", Format( "Best", 10 ) ),
        Column( "sex", Character, "Nominal" ),
        Column( "height", Numeric, "Continuous", Format( "Best", 10 ) ),
        Column( "weight", Numeric, "Continuous", Format( "Best", 10 ) )
    ),
   Lines To Read( 10 )
);
```

引用符としてアポストロフィを使用／Use Apostrophe as Quotation Mark

アポストロフィで囲まれているデータの場合に、アポストロフィを引用符として扱い、省略します。たとえば、「'2010'」は「2010」として読み込まれます。これはブール値です。デフォルト値は偽です。

テキストデータが標準的な形式ではなく、フィールドが引用符ではなくアポストロフィで囲まれている場合以外、このオプションは推奨されません。

テキストの読み込み環境設定の指定

テキストを読み込むための環境設定を指定したい場合は、最初にすべての環境設定のリストを見ることができると便利です。そのためには Show Preferences (All) 関数を使用します。

その後、必要なセクションだけをコピーして Preferences 関数の中に貼り付けます。たとえば、読み込み設定を指定する場合の式は次のようになります。

```
Preferences(
   Import Settings(
      End Of Field( Tab, Spaces, Comma )
   )
);
```

Microsoft Excel ファイルからのデータの読み込み

JMPでExcelブックを開くと、自動的にデータテーブルに変換されます。JMPは、.xls、.xlsm、.xlsx 形式のファイルに対応しています。Microsoft Excelのサポートの詳細については、『JMPの使用法』の「データの読み込み」章を参照してください。

Excelに関する環境設定

JMPの環境設定では、［一般］にワークシートの読み込みに関する設定があります。

- ［**Excel ファイルを開く方法**］は、標準の Open ステートメントを使って Microsoft Excel ファイルを開く際の、デフォルトの方法です。
  - ［**Excel ウィザードを使用**］に設定すると、ファイルの読み込み時に Excel 読み込みウィザードが開きます。これがデフォルトの設定です。
  - ［**すべてのシートを開く**］に設定すると、Microsoft Excel ファイルに含まれるすべてのシートが開きます。
  - ［**個々の Excel シートを選択**］に設定すると、ユーザがファイルを開く際、1つまたは複数のワークシートを選択するよう促されます。
- ［**Excel のラベルを列名として使用**］は、ワークシートの最初の行にあるテキストを、データテーブルの列名に変換するかどうかを指定します。

  デフォルトでは、自動識別が行われます。最初の行のすべてのセルに名前が定義されている場合は、それらのセル内のテキストが列名に変換されます。名前が定義されていないセルがある場合は、「**列 1**」、「**列 2**」・・・という列名が使用されます。

環境設定を無視するには、この節で後述している、対応する引数を含めます。

メモ : データテーブルを Excel ワークシートに書き出す方法については、「Excel ブックの作成」（294 ページ）を参照してください。

Open() 関数

Open() 関数を引数なしで使用して Excel ファイルを開くと、状況に応じて異なった動作となります。

- Open() 関数がスクリプトの一部として直接実行される場合、Excel ファイルはユーザが指定した Excel 読み込みの環境設定を使用して、データテーブル内に表示されます。次の例は、ウィザードを使用せずに、両ワークシートをデータテーブル内に開きます。

```
Path = Convert File Path( "$SAMPLE_IMPORT_DATA/Team Results.xlsx", absolute );
dt = Open(Path);
```

- ただし、Open() 関数がボタンをクリックすることによって実行されるスクリプトの一部である場合、プレビューウィンドウが開き、ユーザはそれを操作する必要があります。次の例を実行し、ボタンをクリックして、Excel 読み込みウィザードを確認してみましょう。

```
Path = Convert File Path( "$SAMPLE_IMPORT_DATA/Team Results.xlsx", absolute );
New Window( "button", Button Box( "Open", dt = Open( Path ) ) );
```

- このスクリプトにより、Excel ファイルを読み込むためのプレビューウィンドウが開かないようにするには、Open() 関数に引数を追加します。例を実行し、ボタンをクリックします。2つのワークシートが、Excel 読み込みウィザードを介さずにデータテーブル内に開きます。

```
Path = Convert File Path( "$SAMPLE_IMPORT_DATA/Team Results.xlsx", absolute );
New Window( "button", Button Box( "Open", dt = Open( Path, Use for all sheets(1) )
  ) );
```

このほか、次のように、環境設定のExcel Open Methodを使って、すべてのシートを開くよう指定することもできます。

```
Preference(Excel Open Method( "Open All Sheets" ));
Path = Convert File Path( "$SAMPLE_IMPORT_DATA/Team Results.xlsx", absolute );
New Window( "button", Button Box( "Open", dt = Open( Path ) ) );
```

Excel読み込みウィザードでワークシートを開く

Excel読み込みウィザードでは、データを読み込む前にデータのプレビューを表示し、設定に変更を加えることができます。次のように、引数として"Excel Wizard"を指定します。

```
Open( "$SAMPLE_IMPORT_DATA/Team Results.xlsx", "Excel Wizard" );
```

また、列見出しの行数やデータの最終行などの設定をカスタマイズすることもできます。

```
dt = Open(
   "C:\Excel Wizard Demo\PotatoProduction.xls",
   Worksheets( "Worldprod" ),
   Use for all sheets( 1 ),
   Concatenate Worksheets( 0 ),
   Create Concatenation Column( 0 ),
   Worksheet Settings(
      1,
      Has Column Headers( 1 ),
      Number of Rows in Headers( 1 ),
      Headers Start on Row( 2 ),
      Data Starts on Row( 5 ),
      Data Starts on Column( 1 ),
      Data Ends on Row( 40 ),
      Data Ends on Column( 0 ),
      Replicated Spanned Rows( 1 ),
      Replicated Spanned Headers( 0 ),
      Suppress Hidden Rows( 1 ),
      Suppress Hidden Columns( 1 ),
      Suppress Empty Columns( 1 ),
      Treat as Hierarchy( 0 ),
      Multiple Series Stack( 0 ),
      Import Cell Colors( 0 ),
      Limit Column Detect( 0 ),
      Column Separator String( "-" )
   )
);
```

スクリプトで作成した独自のボタンをクリックしてワークブックを開くと、［Excel ファイルを開く方法］の環境設定が適用されます。ワークシートを直接開くようにするには、次のようにスクリプト内にExcel Open Methodを指定します。

```
Preference( Excel Open Method( "Open All Sheets" ) );
```

特定のワークシートの読み込み

ブックにある特定のワークシートからデータを読み込みたいとしましょう。その場合は、Worksheets という引数を使ってワークシートを指定します。次の例は、「小」という名前のワークシートを読み込みます。

```
Open( "C:\My Data\cars.xlsx", Worksheets( "小" ) );
```

または、ワークシートの番号を指定します。この例では、3番目のワークシートを読み込みます。

```
Open( "C:\My Data\cars.xlsx", Worksheets( "3" ) );
```

複数またはすべてのワークシートを読み込む場合は、リストの形でワークシート名を指定します。

```
Open( "C:\My Data\cars.xlsx", Worksheets( {"小", "中", "大"} ) );
```

SAS データセットの読み込み

SAS ファイルを、SAS サーバーに接続せずにデータテーブルとして開きます。

```
sasxpt = Open( "$SAMPLE_IMPORT_DATA/carpoll.xpt" );
```

ラベルを列名に変換するには、Use Labels for Var Names という引数を使用します。

```
sasdbf = Open( "$SAMPLE_IMPORT_DATA/Bigclass.sas7bdat", Use Labels for Var Names(
  1 ));
```

.xpt および .stx 形式のファイルも使用できます。

Windows では、SAS サーバーに接続して SAS データを開くこともできます。詳細は、「JMP の拡張」章の「SAS Metadata Server への接続」（631 ページ）の節を参照してください。

Web ページおよびリモートファイルの読み込み

Web サイトや別のコンピュータからデータを読み込むことができます。JMP データテーブル、Web ページで定義されているテーブル、JMP でサポートされている他の種類のファイルを開くことができます。

Web サイト上のデータを開く、または読み込む

Web サイトにあるファイルを開くには、Open() コマンドの中で URL を指定します。その際、URL を引用符で囲んでください。JMP のデータテーブルほか、JMP で使用できる種類のファイルはこの方法で開くことができます。

```
Open( "http://company1.com/Repairs.jmp" );
Open( "http://company1.com/My Data.txt", text); // Macintosh 上のテキストファイルを指定
```

Web ページの読み込み

Web ページには、表形式のデータが含まれていることがあります。次の例では、表を JMP データテーブルとして読み込みます。

```
Open( "http://company1.com", HTML Table( n ) );
```

*n*は、読み込みたい表を表します。たとえば、ページ上にある4つ目の表を読み込むには、HTML Table(4)と指定します。値を省略した場合、ページ上の最初の表だけが読み込まれます。

JMPは、HTMLタグの<th>で定義されているテーブル見出しを保持しようとします。これらのテーブル見出しはデータテーブル内の列名に変換されます。<th>タグが間違っている場合やない場合には、ColumnNames(*n*)を使用して*n*番目の行を指定してください。デフォルトでは、DataStarts(*n*)が次の行になりますが、DataStartsで行を指定することもできます。

Webページからのイメージの読み込み

データタイプが式である列を使用して、Web上のイメージをデータテーブルに読み込むことができます。以下の例では、新しいデータテーブルが作成され、そこに2つのイメージが追加されます。

```
dt = New Table( "test", New Column( "Image", Expression ) );
dt << Add Rows( 1 );
dt:Image = New Image( Open(
"http://www.jmp.com/support/help/13/images/MosaicPlot.png" ) );
dt << Add Rows( 1 );
dt:Image = New Image( Open(
"http://www.jmp.com/support/help/13/images/tTest_NoLoc.png" ) );
```

共有コンピュータからのファイルの読み込み

JMPでは、共有している別のコンピュータやネットワークドライブに保存されたファイルを読み込むことができます。ファイルパスとしては、絶対パスと相対パスの両方が使えます。次の例では、**Data**という名前の共有コンピュータにあるファイルを開きます。スクリプトを共有する予定がある場合は、マップしたドライブではなく、コンピュータへの相対パスを使用するようにしてください。

```
Open( "\\Data\Repairs.jmp" );
Open( "\\Data\My Data.txt" );
```

HDF5ファイルの読み込み

Hierarchical Data Format、バージョン5（HDF5）は、データを保存するためのポータブルファイル形式です。HDF5ファイルはグループとデータセットで構成されています。ファイルを読み込むとき、JMPはグループを開きます。このグループには、データセットの名前が含まれています。

JMPで扱えるのは、数値（integer、float、double）または文字列のタイプの列のみ、3次元以下のsimpleタイプのcompoundファイルのみです。

1,000,000個までの列を読み込むことができます。行数には制限はありません。

構文は次のとおりです。

```
Open( "filename.h5", {"list_of", "dataset_names"});
```

無効なデータセット名が渡されると、エラーがログに書き込まれます。

ESRI シェープファイルの読み込み

ESRI シェープファイルは、地図の作成に使用される地理空間ベクトルデータの一形式です。JMP は、シェープファイルをデータテーブルとして読み込みます。.shp シェープファイルは、各シェープの座標で構成されます。.dbf シェープファイルには、地域を表す値が含まれます。JMP で地図を作成するには、データの構造を調整し、ファイルに専用の接尾辞をつけて保存します。

次の例は、.shp ファイルを読み込み、-XY という接尾辞をつけて保存します。

```
dt = Open( "$SAMPLE_IMPORT_DATA/Parishes.shp",
:X << Format( "Longitude DDD", 14, 4 );
:Y << Format( "Latitude DDD", 14, 4 ) );
dt << Save( "c:/Parishes-XY.jmp" );
```

.dbf ファイルを -Name という接尾辞をつけて保存します。

```
dt = Open( "$SAMPLE_IMPORT_DATA/Parishes.dbf" );
dt << Save( "c:/Parishes-Name.jmp" );
```

データの構造を調整するには、-Name.jmp ファイル内の名前に「地図の役割」列プロパティを追加するなど、いくつかの手順が必要となります。詳細については、『グラフ機能』の「地図の作成」章を参照してください。

パスワードで保護された Microsoft Excel 2007 ファイルの読み込み

JMP は、パスワードで保護された .xlsx ファイルの読み込みをサポートしていません。パスワードで保護された Excel 2007 の .xls ファイルを読み込むには、Password 引数を含めます。

```
Open( "Housing.xls", Password( "helloworld" ) );
```

メモ: パスワードで保護された Excel 2010 ／ 2013 の .xlsx ファイルは、.xls ファイルとして保存した場合でも読み込めません。

データベースの読み込み

Open Database() は、ODBC（Open Database Connectivity）を使ってデータベースを開き、データを抽出し、JMP データテーブルに読み込みます。詳細については、「JMP の拡張」章の「データベースアクセス」（624 ページ）を参照してください。

JMP は、データベースファイル（.dbf）ファイルもデータテーブル形式に変換します。

```
sasdbf = Open( "$SAMPLE_IMPORT_DATA/Bigclass.dbf",
   Use Labels for Var Names( 1 )
);
```

現在のデータテーブルの設定

ヒント： 目的のデータテーブルが現在のデータテーブルになっているかどうかの判断には注意が必要です。スクリプトの冒頭で現在のデータテーブルに設定した場合でも、途中のアクションによって状況が変わっている場合があります。

次のように処理されたデータテーブルは、現在のデータテーブルになります。

- 次の方法で開いたデータテーブル

```
dt1 = Open( "$SAMPLE_DATA/Big Class.jmp" );
```

- 次の方法で新しく作成したデータテーブル

```
dt2 = New Table( "Cities" );
```

開いている別のデータテーブルを現在のデータテーブルとするには、Current Data Table() コマンドを使い、データテーブル名を指定します。

```
dt1 = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt2 = New Table( "Cities" );
Current Data Table( dt1 ); // Big Class.jmp データテーブルを現在のデータテーブルにする
```

Current Data Table() も、スクリプト可能なオブジェクトの参照を引数としてとることができます。次の式は、2番目のBivariateオブジェクトによって使用されているデータテーブルを現在のデータテーブルにします。

```
Current Data Table( Bivariate[2] );
```

分析プラットフォームのオブジェクトの参照を使用する方法については、「プラットフォームのスクリプト」章の「プラットフォームへのメッセージの送信」（377ページ）を参照してください。

データテーブルに名前をつける

データテーブルに名前をつけるには、Set Name メッセージを送ります。引数には、引用符で囲んだファイル名か、ファイル名を評価する式を指定します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Set Name( "New Big Class.jmp" );

s = "New Big Class";
dt << Set Name( s );
```

名前を取得するには、データテーブルに Get Name メッセージを送ります。

```
dt << Get Name();
  "New Big Class"
```

データテーブルの保存

データテーブルを保存するには、データテーブルに Save メッセージを送ります。たとえば、次の場合が該当します。

```
dt << Save(); // 現在の名前を使用して保存する
dt << Save( "Newest Big Class.jmp" ) // 新しいファイルとして保存する
dt << Save( "c:/My Data/New Big Class.jmp" );  // 新しいファイルとして保存する
dt << Save( "My Table", JMP( 5 ) ); // JMP 5 のテーブルとして保存する
dt << Save("") // ディレクトリを選択して任意の形式で保存するようユーザを促す
dt << Save( "Big Class.xls" ); // Microsoft Excel ファイルとして保存する
Generate Excel Workbook( "c:\MyData\data.xlsx", {"Abrasion", "Big Class"},
{"Abrasive", "Class"} ); // 複数のデータテーブルを 1 つのブックとして保存する
```

メモ: パスを使わずにファイル名を指定し、デフォルトのディレクトリを設定していない場合、ファイルは、プライマリパーティション（Windows の場合）または *<ユーザ名>*/Documents フォルダ（Macintosh の場合）に保存されます。デフォルトのディレクトリを設定する方法については、「データタイプ」章の「相対パス」（122 ページ）を参照してください。

Windows では、拡張子 .txt を付けて保存すると、環境設定の［テキストデータファイル］で設定されている方法で書き出されます。Macintosh では、次のように Save 関数の 2 番目の引数として Text を追加します。

```
dt << Save( "New Big Class.txt", Text );
```

データテーブルの名前を設定し、後で Save メッセージを送る予定がある場合は、Save メッセージだけを使って名前を指定することができます。

```
dt << Set Name( "New Big Class.jmp" );
dt << Save();
```

上の組み合わせと、次のメッセージは等価です。

```
dt << Save( "New Big Class.jmp" );
```

Save とパス名を含める方法も、Save As とパス名を使う方法と等価です。

データテーブルを保存されている状態に戻す

最後に保存した状態のデータテーブルに戻すには、Revert メッセージをデータテーブルに送ります。

```
dt << Revert();
```

データテーブルを非表示にする

データテーブルを表示させる必要がない場合は、2 通りの方法でそれらのテーブルを非表示にできます。invisible として開く方法と private として開く方法です。

非表示のデータテーブル

invisible で非表示にしたデータテーブルは、画面上では見えませんが、実行する分析にはリンクされています。次のように、データテーブルを非表示にして開きます。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp", "invisible" );
```

次の例では、10行から成る Abc という非表示のデータテーブルを作成します。1つの列に **X** という名前をつけます。

```
dt = New Table( "Abc", "invisible", New Column( "X" ), Add Rows( 10 ) );
```

開いたデータテーブルが非表示かどうかを調べるには、データテーブルオブジェクトにブール値の Has Data View メッセージを送ります。次の式は、データテーブルが非表示の場合に 0（偽）を戻します。

```
dt << Has Data View();
```

非表示のデータテーブルでの作業を終えたら、必ず Close() 関数でデータテーブルを閉じてください。そうしないと、JMP を終了するまでデータテーブルがメモリに残ります。

非表示のデータテーブルを表示する

非表示のデータテーブルは、Show Window() 関数で表示できます。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp", "invisible" );
dt << Show Window( 1 );
```

ユーザは JMP インターフェースからテーブルを開くこともできます。Windows の場合、invisible のデータテーブルはホームウィンドウのウィンドウリストと［**ウィンドウ**］>［**再表示**］メニューに表示されます。Mac の場合も、データテーブルは JMP ホームウィンドウと［**ウィンドウ**］>［**表示しない**］メニューに表示されます。

プライベートデータテーブル

データテーブルを完全に非表示にするには、private 引数を含めます。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp", "private" );
```

テーブルを Private にすると、スクリプトが多くのテーブルを開いたり閉じたりする際に、メモリの問題を回避できます。Private に指定したデータテーブルは、物理的なウィンドウを持たないため、必要なメモリが非表示のテーブルより少なくなります。

Invisible の場合と同様、Private のデータテーブルに対して実行した分析はデータテーブルとリンクしています。

非表示になっているデータテーブルが失われるのを防ぐため、前の例のように、必ずデータテーブルに参照を割り当ててください。そうしないと、Private のデータテーブルはメモリから直ちに削除されます。このため、その後の処理でそのテーブルを使おうとしても、エラーになってしまいます。

メモ: データテーブルをPrivateに指定することの利点は、テーブル内のデータのサイズに左右されます。たとえば、データテーブルに5列と20行しか含まれていない場合には、データテーブルをPrivateに指定することにより、相応のメモリを節約できます。しかし、テーブルに100列と1,000,000行が含まれているような場合には、そのデータに多くのメモリが必要となるため、データテーブルをPrivateに指定する意味がなくなってしまします。

データテーブルの印刷

データテーブルを印刷するには、`Print Window`メッセージを送ります。JMPは、印刷の際、コンピュータのデフォルトの設定を使用します。

```
dt << Print Window();
```

このメッセージは、他のディスプレイボックスにも適用できます。

データテーブルのサイズ変更

dt `<< Maximize Display`は、すべての列のサイズを再調整して、データテーブルを最適なサイズのウィンドウに拡大します。

```
dt << Maximize Display;
```

データテーブルを閉じる

データテーブルを閉じるには、引数にデータテーブルの参照を指定して、`Close`コマンドを使います。データテーブルに加えられた変更（データテーブルのリンクを含む）がまだ保存されていない場合、その変更は自動的に保存されます。そのデータテーブルを元に作成されたレポートやグラフも同時に閉じられます。

変更を保存してデータテーブルを閉じるには:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
Close( dt );
```

変更を保存せずにデータテーブルを閉じるには:

```
Close( dt, No Save );
```

コピーを新しい名前で保存し、元のバージョンを閉じるには:

```
Close( dt, Save("c:/My File.jmp") );
```

また、すべてのデータテーブルを同時に閉じることもできます。その場合も、変更を保存するかしないかを指定します。

```
Close All( Data Tables );
Close All( Data Tables, No Save );
```

プライベートまたは非表示のデータテーブルをすべて閉じるには、該当する引数を指定します。

```
Close All( Data Tables, private );
Close All( Data Tables, invisible );
```

ヒント： Close All() は、ジャーナルやレイアウトにも使用できます。その場合は、引数で "Journals" または "Layouts" を指定します。

データテーブルの設定と取得

ディスプレイボックス、たとえばNew Data Box()でデータテーブルを指定するには、Set Data Table()やGet Data Table()を使用して、データテーブルを設定または戻します。

次の例では、「Big Class.jmp」と「Cities.jmp」サンプルデータテーブルを使用しています。まず、これらのサンプルデータテーブルが非表示で開きます。次に、新しいウィンドウに、「Big Class.jmp」を含む新しいデータボックスが作成されます。1秒待機した後、データボックスの内容が「cities.jmp」の表示に変わります。

```
dtC = Open( "$SAMPLE_DATA/Cities.jmp", "invisible" );
dtA = Open( "$SAMPLE_DATA/Big Class.jmp", "invisible" );
nw = New Window( "My Table", H List Box( dtBox = dtA << New Data Box() ) );
Wait( 1.0 );
dtBox << Set Data Table( Data Table( "Cities.jmp" ) );
```

開いているすべてのデータテーブルに対するアクションの実行

現在開いているデータデーブルすべてに対してアクションを行う場合は、N Table()を使用して各データテーブルへの参照リストを取得できます。

```
dt1 = Open( "$SAMPLE_DATA/Cities.jmp" );
dt2 = Open( "$SAMPLE_DATA/Big Class.jmp" );
openDTs = List();
For( i = 1, i <= N Table(), i++,
   Insert Into( openDTs, Data Table( i ) );
);
```

これで、openDTsは、開いているすべてのデータテーブルへの参照リストとなります。openDTs[n]を使用すると、任意のデータテーブルにメッセージを送ることができます。開いているデータテーブルすべてにメッセージを送るには、forループを使用できます。次のループでは、各データテーブルにMy Columnという名前の新しい列を追加します。

```
For( i = 1, i <= N Items( openDTs ), i++,
   openDTs[i] << New Column( "My Column" );
);
```

データテーブルへの参照ではなく、テーブル名のリストだけが必要な場合は、Get Name メッセージを使用します。

```
For( i = 1, i <= N Table(), i++,
   Insert Into( openDTs, Data Table( i ) << Get Name());
   );
```

取得したリストを使い、開いているデータテーブルすべての名前をリストボックス（ウィンドウ）に含めれば、ユーザがデータテーブルの操作を行うときに、その中から対象となるデータテーブルを選択することができます。また、リストにあるデータテーブル名をファイルに書き出すこともできます。

ジャーナルとレイアウトの作成

ジャーナルは、JMP グラフやレポート、グラフィック、テキスト、Web ページやファイルなどへのリンクで構成されます。さまざまな要素をジャーナルにまとめることで、プレゼンテーションなどの機会に繰り返し利用することが可能になるほか、他のドキュメントへの読み込みも簡単になります。

レイアウトは、複数のレポートを組み合わせたファイルで、保存する前にレポートの配置を調整することができます。

ジャーナルとレイアウトの基本的な構文は、次のようになります。

```
dt << Journal;
dt << Layout;
```

また、New Window() コマンド内に加えることにより、新しいジャーナルウィンドウを作成します。次の例は、空白で無題のジャーナルとレイアウトを作成します。

```
New Window( <<  Journal);
New Window( <<  Layout);
```

次の例は、「売り上げ」という名前で空白のジャーナルとレイアウトを作成します。

```
New Window( "売り上げ", << Journal );
New Window( "売り上げ", << Layout );
```

ここで、2つのボタンを含む、「ボタンのテスト」というジャーナルを作成する例を紹介します。

```
New Window( "ボタンのテスト", << Journal,
   Button Box( "テスト 1", New Window( "こんにちは 1", << Modal ) ),
   Button Box( "テスト 2", New Window( "こんにちは 2", << Modal ) )
);
```

ジャーナルの取得

ジャーナルとジャーナルウィンドウのコマンドには、オプションの引数があります。この引数は、現在の表示を取得する（フリーズさせる）ものです。つまり、表示ツリーを 1 つのイメージに変換します。

引数には次の4つのオプションがあります。

- 編集可能なディスプレイボックスの構造をコピーするのではなく、その時点での表示の「スナップショット」をビットマップ形式で保存し、ジャーナルに送ります。表示をビットマップファイルとして保存することで、グラフ内でスクリプトの変数が参照されていても、問題が生じにくくなります。

  ```
  << Journal("Freeze All");
  ```

- Freeze Allに似ていますが、ピクチャーと呼ばれるレポートの領域のみをフリーズさせます。

  ```
  << Journal("Freeze Pictures");
  ```

- Freeze Picturesに似ていますが、（ピクチャー内の）フレームボックスのみをフリーズさせます。

  ```
  << Journal("Freeze Frames");
  ```

- Freeze Framesに似ていますが、何かを描画するスクリプトがあるフレームボックスのみをフリーズさせます。

  ```
  << Journal("Freeze Frames with Scripts");
  ```

Current Journal

Current Journal()は、現在のジャーナル表示ウィンドウの最上位のディスプレイボックスへの参照を戻します。ジャーナルが開かれていない場合は、1つ作成されます。引数はありません。

Appendコマンドを使ってジャーナルに追加できます。次の例は、現在のジャーナルの下にテキストボックスを追加します。

```
Current Journal() <<  Append( Text Box( "Hello World" ) );
```

また、文字列を角括弧（[]）で囲んで追加すると、既存のジャーナル内の該当文字列が検索されます。現在のジャーナルに「パラメータ推定値」という文字列が含まれているとします。次の例は、そのジャーナルの下にテキストボックスを追加します。

```
Current Journal()[" パラメータ推定値 "] << Append( Text Box( " アスタリスクは有意確率
  0.05 で有意な項目を示す。" ) );
```

Excelブックの作成

開いているデータテーブルは、Create Excel Workbook()関数を使用して新規または既存のExcelブックに保存できます。以下の例では、Big Class Families.jmpおよびSan Francisco Crime.jmpサンプルデータテーブルのデータを含む、MyWorkbook1.xlsxを作成します。

```
dt1 = Open( "$SAMPLE_DATA/Big Class Families.jmp" );
dt2 = Open( "$SAMPLE_DATA/San Francisco Crime.jmp" );
tableList = {"Big Class Families", "San Francisco Crime"};
// データテーブルを指定する
sheetList = {"Families", "Crime"};
// ワークシートを指定する（オプション）
Create Excel Workbook( "c:/MyWorkbook1.xlsx", tableList, sheetList );
```

ワークシート名を変更せずに開いているデータテーブルをすべて追加するには、最初の引数だけを指定します。

```
Create Excel Workbook( "c:/MyWorkbook2.xlsx" );
```

メモ:

- 保存されるファイルは.xlsxファイルです。.xls形式はサポートされていません。
- JMPデータテーブルは、Excelの制限である約100万行と16,535列までの大きさでなければなりません。これより大きい場合、データの一部がExcel内に保存されなくなります。

データテーブルの高度なスクリプト

この節では、データテーブルに対して実行できる、要約統計量の収集、サブセットの作成、並べ替え、連結などの高度なアクションについて説明します。

要約統計量をグローバル変数に格納する

Summarizeコマンドは、データテーブルの要約統計量を求め、グローバル変数に格納します。Summarizeコマンドは、要約統計量を求めて新しいデータテーブルに表示するSummaryコマンドとは異なります。

1つ目の引数は、オプションのデータテーブル参照です。複数のデータテーブルが開いている可能性がある場合に含めます。

```
dt1 = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt2 = Open( "$SAMPLE_DATA/Animals.jmp" );
Summarize( dt1,
   exg = By( :sex ),
   exm = Mean( :Name("身長(インチ)") )
);
Show( exg );
Show( exm );
```

名前付き引数は、Count、Sum、Mean、Min、Max、StdDev、First、Corr、Quantileで、これらの統計量は数値列に対してのみ計算できます。このそれぞれがデータ列を引数にとります。

次の点を念頭に置いてください。

- *name*=By(*groupvar*)ステートメントが含まれている場合は、サブグループの統計量が*name*という名前のリストに割り当てられます。
- Countの場合、列引数は必須ではありませんが、列を指定して非欠測値の数をカウントすると役に立つことがよくあります。
- Quantileは、どの分位点かを指定する第2引数もとります（たとえば、10パーセント点なら0.1）。

メモ: 除外された行は、Summarizeの計算から除外されます。すべてのデータが除外された場合、Summarizeは欠測値のリストを戻します。すべてのデータが削除された（行がない）場合、Summarizeは空のリストを戻します。

次の例は、「Big Class.jmp」サンプルデータテーブルを使用します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
Summarize(
   a = By( :年齢 ),
   c = Count,
   sumHt = Sum( :Name("身長(インチ)") ),
   meanHt = Mean( :Name("身長(インチ)") ),
   minHt = Min( :Name("身長(インチ)") ),
   maxHt = Max( :Name("身長(インチ)") ),
   sdHt = Std Dev( :Name("身長(インチ)") ),
   q10Ht = Quantile( :Name("身長(インチ)"), .10 )
);
Show( a, c, sumHt, meanHt, minHt, maxHt, sdHt, q10Ht );
```

スクリプトにByグループが含まれているので、結果は1つのリストと6つの行列になります。

```
a = {"12", "13", "14", "15", "16", "17"}
c = [8, 7, 12, 7, 3, 3]
c = [465, 422, 770, 452, 193, 200]
meanHt = [58.125, 60.28571428571428, 64.16666666666667, 64.57142857142857,
   64.33333333333333, 66.66666666666667]
c = [51, 56, 61, 62, 60, 62]
c = [66, 65, 69, 67, 68, 70]
sdHt = [5.083235752381126, 3.039423504234876, 2.367712103711172,
   1.988059594776032, 4.041451884327343, 4.163331998932229]
q10Ht = [51, 56, 61.3, 62, 60, 62]
```

TableBoxを使って結果をフォーマットすることもできます。

```
New Window( "要約の結果",
   Table Box(
      String Col Box( "年齢", a ),
      Number Col Box( "度数", c ),
      Number Col Box( "合計", sumHt ),
      Number Col Box( "平均", meanHt ),
      Number Col Box( "最小値", minHt ),
      Number Col Box( "最大値", maxHt ),
      Number Col Box( "標準偏差", sdHt ),
      Number Col Box( "10パーセント点", q10Ht )
   )
);
```

図9.1 要約の結果

年齢	度数	合計	平均	最小値	最大値	標準偏差	10 パーセント点
12	8	465	58.125	51	66	5.08324	51
13	7	422	60.2857	56	65	3.03942	56
14	12	770	64.1667	61	69	2.36771	61.3
15	7	452	64.5714	62	67	1.98806	62
16	3	193	64.3333	60	68	4.04145	60
17	3	200	66.6667	62	70	4.16333	62

次のように、ウィンドウに全体の統計量を含めることもできます。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );

Summarize(
   a = By( :年齢 ),
   c = Count,
   sumHt = Sum( :Name("身長(インチ)") ),
   meanHt = Mean( :Name("身長(インチ)") ),
   minHt = Min( :Name("身長(インチ)") ),
   maxHt = Max( :Name("身長(インチ)") ),
   sdHt = Std Dev( :Name("身長(インチ)") ),
   q10Ht = Quantile( :Name("身長(インチ)"), .10 )
);

Summarize(
   tc = Count,
   tsumHt = Sum( :Name("身長(インチ)") ),
   tmeanHt = Mean( :Name("身長(インチ)") ),
   tminHt = Min( :Name("身長(インチ)") ),
   tmaxHt = Max( :Name("身長(インチ)") ),
   tsdHt = Std Dev( :Name("身長(インチ)") ),
   tq10Ht = Quantile( :Name("身長(インチ)"), .10 )
);

Insert Into( a, "全体" );
c = c |/ tc;
sumHt = sumHt |/ tsumHt;
meanHt = meanHt |/ tmeanHt;
minHt = minHt |/ tminHt;
maxHt = maxHt |/ tmaxHt;
sdHt = sdHt |/ tsdHt;
q10Ht = q10Ht |/ tq10Ht;

New Window( "要約の結果",
   Table Box(
      String Col Box( "年齢", a ),
      Number Col Box( "度数", c ),
```

```
        Number Col Box( "合計", sumHt ),
        Number Col Box( "平均", meanHt ),
        Number Col Box( "最小値", minHt ),
        Number Col Box( "最大値", maxHt ),
        Number Col Box( "標準偏差", sdHt ),
        Number Col Box( "10パーセント点", q10Ht )
    )
);
```

図9.2 要約と全体の統計量

年齢	度数	合計	平均	最小値	最大値	標準偏差	10 パーセント点
12	8	465	58.125	51	66	5.08324	51
13	7	422	60.2857	56	65	3.03942	56
14	12	770	64.1667	61	69	2.36771	61.3
15	7	452	64.5714	62	67	1.98806	62
16	3	193	64.3333	60	68	4.04145	60
17	3	200	66.6667	62	70	4.16333	62
全体	40	2502	62.55	51	70	4.24234	56.2

Byグループを指定*しない*場合、各統計量の結果は、次のように単一の値になります。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
Summarize(
// a = By( :年齢 ),
   c = Count,
   sumHt = Sum( :Name("身長(インチ)") ),
   meanHt = Mean( :Name("身長(インチ)") ),
   minHt = Min( :Name("身長(インチ)") ),
   maxHt = Max( :Name("身長(インチ)") ),
   sdHt = Std Dev( :Name("身長(インチ)") ),
   q10Ht = Quantile( :Name("身長(インチ)"), .10 )
);
Show( c, sumHt, meanHt, minHt, maxHt, sdHt, q10Ht );
  c = 40;
  sumHt = 2502;
  meanHt = 62.55;
  minHt = 51;
  maxHt = 70;
  sdHt = 4.24233849397192;
  q10Ht = 56.2;
```

Summarizeでは、複数のByグループを使用できます。たとえば、「Big Class.jmp」では、次のように指定できます。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
Summarize( g = By( :年齢, :性別 ), c = Count() );
Show( g, c );
  g = {"12", "12", "13", "13", "14", "14", "15", "15", "16", "16", "17", "17"},
     {"F", "M", "F", "M", "F", "M", "F", "M", "F", "M", "F", "M"}}
```

```
c = [5,3,3,4,5,7,2,5,2,1,1,2]
```

By グループを指定すると、結果は常に行列になります。指定しない場合、結果はスカラーになります。

要約統計量の表を作成する

Summary コマンドは、ユーザが選択したグループ化列に基づいて、要約統計量から成る新しいテーブルを作成します。Summary は、データテーブルの要約統計量を求めてグローバル変数に格納する Summarize とは異なるので、注意が必要です。詳細は、「要約統計量をグローバル変数に格納する」（295 ページ）の節を参照してください。

```
summDt = dt << Summary(
   Group( groupingColumns ),
   Subgroup( subGroupColumn ),
   Statistic( columns ),// 統計量は、Mean( 平均 )、Min( 最小値 )、Max( 最大値 )、Std Dev( 標準偏差 ) など
   Output Table Name( newName ) );
```

次の例は、年齢別の平均身長と平均体重の列、年齢別の最大身長と最小体重の列を含めた新しいデータテーブルを作成します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
summDt = dt << Summary(
   Group( :年齢 ),
   Mean( :Name("身長 ( インチ )"), :Name("体重 ( ポンド )") ), Max( :Name("身長 ( インチ )") ), Min( :Name("体重 ( ポンド )") ),
   Output Table Name( "身長・体重の表" ) );
```

ヒント： Output Table Name は、引用符付き文字列、または文字列を含んだ変数をとることができます。

デフォルトでは、要約テーブルは元のデータテーブルにリンクしています。元のデータテーブルとリンクしていない要約を作成したいときは、Summary メッセージに次のオプションを追加します。

```
summDt = dt << Summary( Group( :年齢 ), Mean( :Name("身長 ( インチ )") ),
   Link to Original Data Table( 0 )
);
```

グループ変数の水準ごとの統計量に加え、全体に対する統計量を出力データテーブルに含めることもできます。また、最初のグループ変数の各水準の要約行をデータテーブルの末尾に追加することも可能です。全体の統計量を追加するには、次のオプションを Summary メッセージに追加します。

```
summDt = dt << Summary( Group( :年齢 ), Mean( :Name("身長 ( インチ )") ),
   Include marginal statistics
);
```

統計量の列名の形式は、statistics column name format() を使用して指定できます。列名は以下のいずれかの形式となります。

- 統計量 (列名)

- 列名
- 列名の統計量
- 列名 統計量

たとえば、次のようなオプションをSummaryメッセージに追加します。

```
summDt = dt << Summary( Group( :年齢 ), Mean( :Name("身長(インチ)") ),
   statistics column name format("列名の統計量")
);
```

データテーブルのサブセットを作成する

Subset()は、指定された行から新しいデータテーブルを作成します。行を指定しない場合、Subsetは選択されている行を使います。行が選択も指定もされていない場合は、すべての行を使います。列が選択も指定もされていない場合は、すべての列を使います。Subsetに引数を指定しなかった場合、「サブセット」ウィンドウが表示されます。

```
dt << Subset(
   Columns( columns ),
   Rows( row matrix ),
   Linked,
   Output Table Name( "name" ),
   Copy Formula( 1 or 0 ),
   Sampling rate( n ),
   Suppress Formula Evaluation( 1 or 0 ) );
```

メモ: その他の引数については、［ヘルプ］メニューの［スクリプトの索引］を参照してください。

たとえば、Big Class.jmpのうち、年齢が「12」であるすべての行を選択するには:

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
For Each Row( Selected( Row State() ) = (:年齢 == 12) );
subdt = dt << Subset( Output Table Name( "サブセット" ) );
```

3つの列とすべての行を選択するには、次のスクリプトを実行します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
subDt1 = dt << Subset(
   Columns( :名前, :年齢, :Name("身長(インチ)") ),
   Output Table Name( "Big Class 2" )
);
```

2つの列の指定した行を取り出し、元のデータテーブルとリンクしたテーブルを作成するには、次のスクリプトを実行します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
subDt2 = dt << Subset(
   Columns( :名前, :Name("体重(ポンド)") ),
```

```
    Rows( [2, 4, 6, 8] ),
    Linked
);
```

年齢が12歳のすべての行を選択するには、次のスクリプトを実行します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Select Where( :年齢 == 12 );
dt << Subset( ( Selected Rows ), Output Table Name( "サブセット" ) );
```

Data Filterを使ってデータのサブセットを作成する

Data Filterは、データをサブセットしながら探索するためのさまざまな方法を提供します。Data Filterのコマンドとオプションを使えば、データの複雑なサブセットを選択し、それをプロット内で非表示にしたり、分析から除外したりできます。

基本的な構文は次のとおりです。

```
dt << Data Filter( <local>, <invisible>, <Add Filter (...)>, <Mode>, <Show Window(
  0 | 1 ), <No Outline Box( 0 | 1 )> );
```

Data Filterのオプションには、次のようなものがあります。

Add Filter、Animation、Auto Clear、Clear、Close、Columns、Conditional、Data Table Window、Delete、Delete All、Display、Get Filtered Rows、Location、Match、Mode、Report、Save and Restore Current Row States、Set Select、Set Show、Set Include、Show Column Selector、Show Subset、Use Floating Window

その他の引数については、［ヘルプ］メニューの［スクリプトの索引］を参照してください。

オプションなしでData Filterメッセージをデータテーブルに送ると、Data Filterの開始ウィンドウが表示され、「フィルタ列の追加」パネルにデータテーブル内の変数がリストされます。

Modeは、Select(bool)、Show(bool)、Include(bool)というオプションの引数をとります。これらの引数は、それぞれ対応するオプションをオンまたはオフにします。Selectのデフォルト値はtrue（1）です。ShowとIncludeのデフォルト値はfalse（0）です。

Add Filter()は、列とデータテーブルをサブセットするWhere節を指定します。基本的な構文は次のとおりです。

```
Add Filter( Columns( col, ... ), Where( ... ), ... )
```

データフィルタに列を追加するには、列名をカンマで区切って指定します。これは、リストデータ構造ではありません。

フィルタ列を指定する場合は、1つまたは複数のWhere節を次のように定義します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
df = dt << Data Filter(
   Mode( Show( 1 ) ),
   Add Filter(
```

```
        Columns( :年齢, :性別, :Name("身長(インチ)") ),
        Where( :年齢 == {13, 14, 15} ),
        Where( :性別 == "M" ),
        Where( :Name("身長(インチ)") >= 50 & :Name("身長(インチ)") <= 65 )
    )
);
```

このスクリプトは、身長が50インチ以上65インチ以下の13歳、14歳、15歳の男子を選択します。前述のスクリプトにInvert Selectionメッセージを追加すると、このフィルタされたデータで除外されている年齢を選択できます。

```
df << ( Filter Column( :年齢 ) << Invert Selection );
```

この例では、フィルタされたデータの13歳、14歳、15歳以外の年齢が選択されます。

Add Filter()を使って、多重応答のプロパティまたは尺度が設定されている列から一致する文字列を選択することもできます。

```
dt = Open( "$SAMPLE_DATA/Consumer Preferences.jmp" );
df = dt << Data Filter(
    Location( {437, 194} ),
    Add Filter(
        Columns( :歯磨き カンマ区切り ),
        Match None( Where( :歯磨き カンマ区切り == {"Before Sleep", "Wake"} ) ),
        Display( :歯磨き カンマ区切り, Size( 121, 70 ), Check Box Display )
    )
);
```

このスクリプトは、**「歯磨き カンマ区切り」**列内で、指定された値（「Before Sleep」、「Wake」）のいずれにも一致しない行を選択します。使用可能なその他のスクリプトオプションには、Match Any、Match All、Match Exactly、Match Onlyがあります。多重応答プロパティおよび多重応答尺度の詳細については、『JMPの使用法』の「列情報ウィンドウ」章を参照してください。

既存のData Filterオブジェクトにメッセージを送ることもできます。

```
Clear(), Display( ... ), Animate(), Mode(), ...
```

Clearは、引数をとらず、データフィルタをクリアします。

スクリプトの実行時にデータフィルタが表示されないようにするには、次のようにShow Windowを0に設定します。

```
obj  = ( dt << Data Filter( Show Window( 0 ) ) );
```

除外する値を指定するには、!=演算子を使用します。次の例では、フィルリング後の値に16歳と17歳の年齢が含まれなくなります。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
df = dt << Data Filter(
    Add Filter( Columns( :年齢, :性別 ) ),
```

```
      Match( Columns( :年齢, :性別 ),
         Where( :性別 = "M" ), Where( :年齢 != {16, 17} )
      )
   );
```

データフィルタのコンテキストを定義する

データフィルタを使うと、データの複雑なサブセットを選択し、それをプロット内で非表示にしたり、分析から除外したりできます。選択の範囲は、データテーブルのすべての分析に影響します。

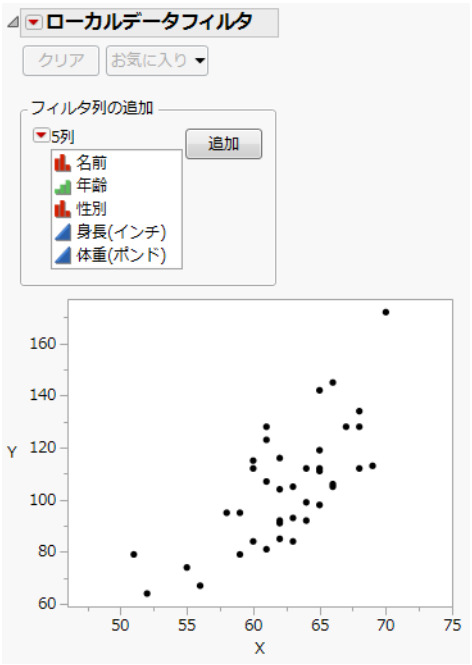
別のオプションとしては、特定のプラットフォームまたはディスプレイボックスからデータをフィルタする方法があります。Data Filter Context Box()関数の中にローカルデータフィルタを作成します。これにより、コンテキストはデータテーブルとしてではなく、現在のプラットフォームまたはディスプレイボックスとして定義されます。

次の例は、グラフボックス用のローカルデータフィルタを作成します。出力については、図9.4を参照してください。

```
New Window( "Marker Seg Example",
   Data Filter Context Box(
      V List Box(
         dt << Data Filter( Local ),
         g = Graph Box(
            Frame Size( 300, 240 ),
            X Scale( Min( xx ) - 5, Max( xx ) + 5 ),
            Y Scale( Min( yy ) - 5, Max( yy ) + 5 ),
            Marker Seg( xx, yy, Row States( dt, rows ) )
         )
      )
   )
);
```

ヒント： 上記のスクリプトはさらに大きなスクリプトの一部であり、そのスクリプトでMarker Segに必要な配列が作成されます。上の結果を得るには、「Local Data Filter for Custom Graph.jsl」サンプルスクリプトを開きます。

図9.3 ローカルデータフィルタとグラフ

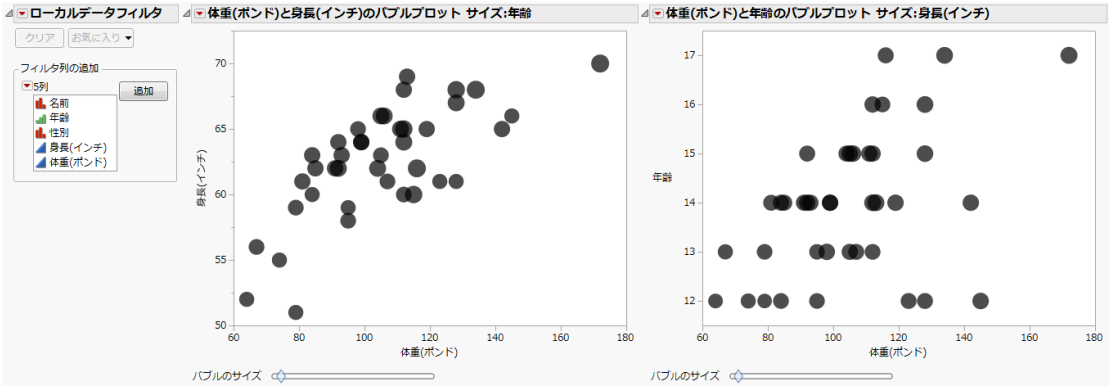


1つのコンテキストボックスに、1つのローカルフィルタと複数のグラフを含めることもできます。フィルタリングはすべてのグラフに適用されます。次のスクリプトは、1つのコンテキストボックスにバブルプロットを2つ、データフィルタを1つ作成します。出力については、図9.4を参照してください。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
New Window( "Shared Local Filter",
   Data Filter Context Box(
      H List Box(
         dt << Data Filter( Local ),
         Platform(
            dt,
            Bubble Plot( X( :Name("体重(ポンド)") ), Y( :Name("身長(インチ)") ),
   Sizes( :年齢 ) )
         ),
         Platform(
            dt,
            Bubble Plot( X( :Name("体重(ポンド)") ), Y( :年齢 ), Sizes( :Name("身長
   (インチ)") ) )
         )
      )
   )
);
```

ヒント： このスクリプトを実行するには、「Local Data Filter Shared.jsl」サンプルスクリプトを開きます。

図9.4 2つのバブルプロットを使ったローカルフィルタ



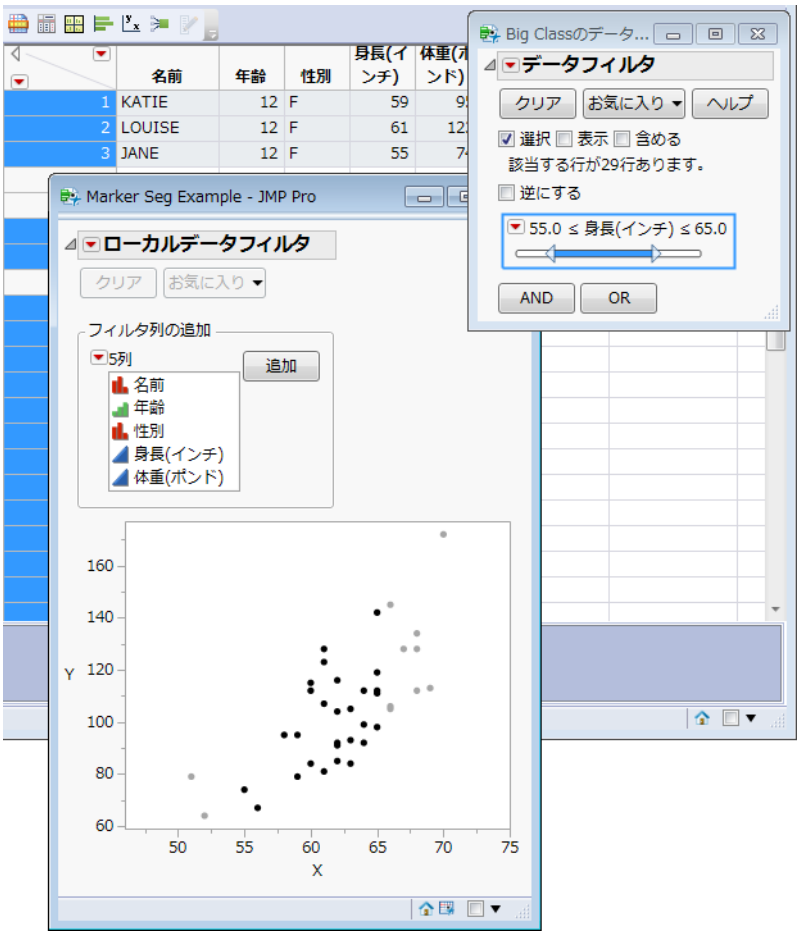
階層的なデータフィルタを作成することもできます。1つのスクリプトでデータフィルタとローカルデータフィルタが生成される場合があります。データテーブルの外側のデータフィルタが、ローカルデータフィルタに使用できるデータを決定します。これをわかりやすく示すため、例を紹介します。次のスクリプトは、図9.5に示すように両方のタイプのデータフィルタを作成します。

ヒント： 次のスクリプトはさらに大きなスクリプトの一部であり、そのスクリプトで Marker Seg に必要な配列が作成されます。これらの配列を作成するコードについては、「Local Data Filter for Custom Graph.jsl」サンプルスクリプトを参照してください。

```
dt << Data Filter(
   Add Filter( Columns( :Name("身長(インチ)") ), Where( :Name("身長(インチ)") >=
   55 & :Name("身長(インチ)") <= 65 ) )
);
New Window( "Marker Seg Example",
   Data Filter Context Box(
      V List Box(
         dt << Data Filter( Local ),
         g = Graph Box(
            Frame Size( 300, 240 ),
            X Scale( Min( xx ) - 5, Max( xx ) + 5 ),
            Y Scale( Min( yy ) - 5, Max( yy ) + 5 ),
            Marker Seg( xx, yy, Row States( dt, rows ) )
         )
      )
   )
);
```

55インチ以上65インチ以下の身長は最初にフィルタされます。その後、ユーザはローカルデータフィルタを使ってグラフの列をフィルタできます。

図9.5 データフィルタの階層



データテーブルを並べ替える

Sort()は、1つ以上の列の値に基づいてテーブルの行を並べ替え、その結果で現在のテーブルを置き換えるか、または新しいテーブルを作成します。それぞれの**By**列に対し、並べ替えの順序（昇順または降順）を指定します。

```
dt << Sort(
   "Private", "Invisible", Replace table, By( columns ), Order( Descending |
  Ascending ) );
```

次の例は、Big Class.jmp を元に新しくデータテーブルを作成し、年齢の降順、名前の昇順で並べます。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
sortedDt = dt << Sort(
   By( :年齢 , :名前 ),
   Order( Descending, Ascending ),
   output table name( " 年齢 名前で並べ替え " ) );
```

連想配列を使って列内の値を並べ替えることもできます。連想配列を使うと、列内の固有の値を簡単に見つけられます。詳細は、「データ構造」章の「列の値を文字コード順に並べ替える」(207ページ)を参照してください。

データテーブル内の値を積み重ねる

Stack() は、複数の列の値を1つの列に積み重ねます。

```
dt << Stack(
   Columns ( columns ),  // ひとつに積み重ねる列
   Source Label Column ( "name" ), // ソース列の識別
   Stacked Data Column ( "name" ), // 新しく積み重ねた列の名前
   Keep ( columns ), // データテーブルに保持する列
   Drop ( columns ), // データテーブルから除去する列
   Output Table ( "name" ), // 新しいデータテーブルの名前
   Columns( columns ) ); // 積み重ねたテーブルに含める列を指定
```

たとえば、次のスクリプトは「Big Class.jmp」の「**体重(ポンド)**」列と「**身長(インチ)**」列を積み重ねます。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
StackedDt = dt << Stack(
    Columns( :Name(" 体重 ( ポンド )"), :Name(" 身長 ( インチ )") ),
    Source Label Column( "ID" ),
    Stacked Data Column( "Y" ),
    Name( " 積み重ねない列 " )( Keep( :年齢 , :性別 ) ),
    Output Table( " 積み重ねた列 " )
);
```

Columns(columns)引数には、列のリスト、またはリストを導き出す式を指定できます。

積み重ねた後のデータテーブルで値を分割する

Splitは、積み重ねられた列を複数の列に分割します。

```
dt << Split(
   Split( columns ),
   // 分割する列（必須）

   Split by( column ),
   // 分割の基準となる列（必須）
```

```
Group(column),
// グループごとにデータを分割する

<Private>|<Invisible>,
// 結果のテーブルをプライベートまたは非表示にする

Remaining Columns( Keep All | Drop All | Select( columns ) ),
/* 残りの列を結果のテーブル内に含めるか指定する（デフォルトは Keep All）*/

<Copy formula( 0|1 )>,
/* ソーステーブルの列の計算式を結果のテーブルに含める（デフォルトは 1、真）*/

<Suppress formula evaluation( 0|1 )>,
// コピーされた列の計算式を自動評価しない（デフォルトは 1、真）

Sort by Column Property( "Value Ordering" {"string", "string"} | "Row Order
Levels" ),
/* 結果のテーブルの行を、
（Split by(columns) で指定した列のプロパティを基準にして並べ替える */

Output Table( "name" ));
// 出力テーブルの名前を指定する
```

次の例は、前の Stack() の例の逆で、元のテーブルと同じ内容を戻します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Stack(
    Columns( :Name("体重（ポンド）"), :Name("身長（インチ）") ),
    Source Label Column( "ID" ),
    Stacked Data Column( "Y" ),
    Name( "積み重ねない列" )( Keep( :年齢, :性別 ) ),
    Output Table( "列を積み重ねた列" )
);
dt2 = Data Table( "列を積み重ねた列" );
dt2 << Split(
   Split( Y ),
   Split by( ID ),
   Output Table( "列を分割したテーブル" ) );
```

データテーブルを転置する

Transposeは、行と列を入れ替えることにより、新しいデータテーブルを作成します。行を指定しない場合、Transpose() は選択されている行を使います。行が選択も指定もされていない場合は、すべての行を使います。

```
dt << Transpose(
   "private", "invisible",
```

```
        columns( columns ),
        Rows( row matrix ),
        By ( column ),
        Label column name( "name" ),
        Output Table( "name" )
);

    dt << Transpose(
        Columns( columns ),
        Rows( row matrix ),
        Output Table Name( "name" ) );
```

次の例は、「Big Class.jmp」データテーブルの「**身長(インチ)**」列と「**体重(ポンド)**」列を転置します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
tranDt = dt << Transpose( Columns( :Name("身長(インチ)"), :Name("体重(ポンド)") ),
    Output Table Name( "転置した列" ) );
```

メモ: シンプルに dt << Transpose のみのコマンドを記述した場合は、転置の起動ウィンドウが表示されます。ウィンドウを表示したくない場合は、dt << Transpose(no option) と指定して転置を実行します。

データテーブルを縦方向に連結する

Concatenate（縦方向の連結）は、複数のデータテーブルの行を上下に連結します。

```
dt << Concatenate( DataTableReferences,...,Keep Formulas,
    Output Table Name( "name" ) );
```

たとえば、データテーブルを男子と女子のサブセットに分けた場合、Concatenate を使うとそれを元に戻すことができます。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Select Where( :性別 == "M" );
m = dt << Subset( Output Table Name( "M" ) );
dt << Invert Row Selection;
f = dt << Subset( Output Table Name( "F" ) );
both = m << Concatenate( f, Output Table Name( "両方" ) );
```

また、連結したデータは、新しいテーブルを作成せずに、現在のデータテーブルに追加することもできます。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Select Where( :性別 == "M" );
m = dt << Subset( Output Table Name( "M" ) );
dt << Invert Row Selection;
f = dt << Subset( Output Table Name( "F" ) );
both = dt << Concatenate( m, f, "最初のテーブルに追加" );
```

データテーブルを横方向に連結する

Join（横方向の連結）は、データテーブルを左右に連結します。

```
dt << Join( // 主テーブルへメッセージを送る
   With(dataTable), // 結合するデータテーブル
   Select With(columns), // 出力テーブルに追加する、
                         // 結合するテーブルの列を選択する
   Select With(columns), // 出力テーブルに追加する副テーブル
                               // の列を選択する
   // 結合の方法。ほかにも次のものがある。
   Cartesian join、By Row Number、By matching columns(col1=col2, ...)
   Merge Same Name Columns, // 同名の列をマージ
   Copy Formula(0), // デフォルトはオン。0 でオフになる
   Suppress Formula Evaluation(0), // デフォルトはオン。0 でオフになる
   Match Flag, // 対応する列の値で結合する場合に、結合後のデータテーブルに「対応フラグ」列を含
   めないようにする
   Update, // 主テーブルのデータを結合するテーブルのデータで更新する
   // 各テーブルのオプション：
   Drop Multiples(Boolean, Boolean), // 重複する行を削除
   Include Non Matches(Boolean, Boolean), // 一致しない行も含める
   Preserve Main Table Order(), // 主テーブルの順序を保持する
   Output Table Name("name")); // 出力テーブル
```

結合を試すために、まず「Big Class.jmp」を2つに分割しましょう。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
part1 = dt << Subset(
   Columns( :名前, :年齢, :Name("身長(インチ)") ),
   Output Table Name( "Big Class 1" )
);
part2 = dt << Subset(
   Columns( :名前, :性別, :Name("体重(ポンド)") ),
   Output Table Name( "Big Class 2" )
);
```

操作を実際の状況に近くするために、part 2の行を並べ替えます。

```
part2 << Sort( By( :名前 ), Output Table Name( "Sorted_Big Class" ) );
```

これで、別々の2つの部分に分割され、それぞれの行の順序が異なるデータテーブルができました。2つのデータテーブルは、共通して持つ列の一致によって結合できます。

```
part1 << Join(
   With( sortedPart2 ),
   By Matching Columns( :名前 == :名前 ),
   Preserve Main Table Order();
   Output Table Name( "結合" );
);
```

結果テーブルには、各部分から1つずつ抽出された2つの「名前」変数のコピーがあり、これらを調べるとJoin（結合）の動作を理解できます。Robertの行が4つあることに注意してください。これは、各データテーブルにRobertの行が2つあり（元のテーブルにRobertの行が2つあった）、Join（結合）によってすべての可能な組み合わせが作成されたからです。

ヒント： 結合されたテーブルで元のデータテーブルと同じ順序を維持する（対応のある列で並べ替えるのではなく）には、Preserve Main Table Order()を含めます。このオプションにより、結合プロセスがスピードアップします。

Joinの引数の詳細については、『スクリプト構文リファレンス』の「JSLメッセージ」章を参照してください。

データテーブルの仮想結合

仮想結合はメインデータテーブルと1つまたは複数の補助データテーブルとをリンクします。この機能を使用すると、実際にテーブルを結合しなくても、メインデータテーブルが補助データテーブルのデータにアクセスできるようになります。詳細については、『JMPの使用法』の「データの再構成」章を参照してください。

次の例は、データテーブルを縦方向に結合し、そのデータで一変量の分布を実行する方法を示しています。

```
dt1 = Open( "$SAMPLE_DATA/Pizza Profiles.jmp" );
dt1:ID << Set Property( "リンクID", 1 );
// リンクIDを追加し、それをオンにする

dt2 = Open( "$SAMPLE_DATA/Pizza Responses.jmp" );
dt2:選択肢1 << Set Property( "リンク参照", Reference Table( "$SAMPLE_DATA/Pizza
   Profiles.jmp" ) );
dt2:選択肢2 << Set Property( "リンク参照", Reference Table( "$SAMPLE_DATA/Pizza
   Profiles.jmp" ) );
dt2:選択 << Set Property( "リンク参照", Reference Table( "$SAMPLE_DATA/Pizza
   Profiles.jmp" ) );
// リンク参照を「選択肢1」、「選択肢2」、および「選択」列に追加する

obj = dt2 << Distribution( // 生地[選択肢1]の分布を作成する
   Weight( :被験者 ),
   Nominal Distribution(
      Column(
         Referenced Column(
            "生地[選択肢1]",
            Reference( Column( :選択肢1 ), Reference( Column( :生地 ) ) )
         )
      )
   ),
   Nominal Distribution( Column( :選択肢1 ) )
);
```

データテーブル内のデータを置換する

メモ: Merge Update() は Update() の別名です。

Update() は、あるデータテーブルのデータを別のデータテーブルのデータで置き換えます。

```
dt << Update( // 元のテーブルへのメッセージ
   With( dataTable ), // もう一方のデータテーブル
   By Row Number, // デフォルトの結合タイプ。または
                  // By Matching Columns(col1==col2)
   Ignore Missing, // 欠測値で置き換えることはしない（オプション）
   );
```

実際の例として、「Big Class.jmp」のサブセットを作成してみましょう。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
NewHt = dt << Subset( Columns( :名前, :Name("身長(インチ)")), Output Table Name(
   "hts" ) );
```

次に、各生徒の身長に0～6インチを追加します。

```
diff = Random Uniform( 0, 6 );
For Each Row( NewHt, :Name("身長(インチ)") = :Name("身長(インチ)") + diff );
```

最後に、「Big Class.jmp」の生徒の身長をサブセットテーブルの新しい身長で更新します。

```
dt << Update(
   With( NewHt ),
   By Matching Columns( :名前 == :名前 ),
   );
```

更新されたテーブルに追加する列を制御する

更新後のテーブルには、元のテーブルよりも多くの列が含まれる場合があります。オプションの Add Column from Update Table() を使えば、更新されたテーブルに含める列を選択できます。

列を追加しない場合は、次のように指定します。

```
Data Table( "table" ) << Update(
   With( Data Table( "update data" ) ),
   Match Columns( :ID = :ID ),
   Add Columns from Update Table( None )
);
```

列を追加する場合は、次のように指定します。

```
Data Table( "table" ) << Update(
   With( Data Table( "update data" ) ),
   Match Columns( :ID = :ID ),
   Add Columns from Update table( :col1, :col2, :col3 )
);
```

Tabulateを使って表を作成する

Tabulateは記述統計量のテーブルを作成します。表には、グループ列、分析列、および各種統計量が表示されます。次の例は、男子生徒と女子生徒の身長と体重の標準偏差と平均を含むデータテーブルを作成します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Tabulate( // データテーブルへのメッセージ
   Add Table( // 新しいテーブルの追加
      Column Table( Grouping Columns( :性別 ) ), // 「性別」の列でグループ化
      Row Table( // 行テーブルを追加
         Analysis Columns( :Name("身長(インチ)"), :Name("体重(ポンド)") ),
         // 分析列として「身長( インチ)」と「体重( ポンド)」を使用
                  Statistics( Std Dev, Mean )
               // 標準偏差と平均を表示
));
```

Tabulateで使用するテーブルの列の変換列を作成したり、その列の表示形式を設定することもできます。これには、Analysis Column()内でTransform Column()を使用します。たとえば、次のスクリプトでは、「**身長(インチ)**」を対数変換し、その平均と全体に対する%の列の表示形式を設定しています。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Tabulate(
   Set Format(
      Mean(
         :Name("身長(インチ)")( 10, 1 ),
         Analysis Column(
            Transform Column(
               "対数[身長(インチ)]",
               Formula( Log( :Name("身長(インチ)") ) )
            ),
            Format( 10, "最適" )
         )
      ),
      Name( "全体に対する%" )(:Name("身長(インチ)")( 12, 2 ),
      Analysis Column(
         Transform Column(
            "対数[身長(インチ)]",
            Formula( Log( :Name("身長(インチ)") ) )
         ),
         Format( 12, 2 )
      ))
   ),
   Add Table(
      Column Table(
         Analysis Columns(
            :Name("身長(インチ)"),
            Transform Column(
```

```
                    " 対数 [ 身長 ( インチ )]",
                    Formula( Log( :Name(" 身長 ( インチ )") ) )
                )
            ),
            Statistics( Mean, Name( " 全体に対する % " ) )
        ),
        Row Table( Grouping Columns( : 性別 ) )
    )
);
```

メモ: 列の幅を変更して、「表の作成」内で追加の処理を実行することもできます。詳細については、［スクリプトの索引］を参照してください。

欠測値のパターンを見つける

データテーブルに欠測値があるとき、欠測値にパターンが見られるかどうかを調べたい場合があります。

```
dt << Missing Data Pattern(    //  データテーブルへのメッセージ
   Columns( :miss ),  // この列の欠測値を検索
   Output Table( " 欠測値のパターン " ) // 出力テーブルの名前を指定
);
```

データテーブルを比較する

JMP では、開いた2つのデータテーブルを比較し、データ、スクリプト、テーブル変数、列名、列プロパティ、および列の属性の相違点をレポートに表示できます。JSL を使ってデータテーブルを比較するには:

```
obj = dt << Compare Data Tables(
Compare With( Data Table ("Data Table Name"));
```

たとえば、「Students1.jmp」と「Students2.jmp」を比較するには、次のようなスクリプトを書きます。

```
dt = Open( "$SAMPLE_DATA/Students1.jmp" );
dt2 = Open( "$SAMPLE_DATA/Students2.jmp" );
obj = dt << Compare Data Tables( Compare With( Data Table( "Students2" ) ) );
```

相違点を要約した行列を表示するには、次の関数を使用します。

```
mtx = ( obj << Get Difference Summary Matrix );
```

次のような行列がログウィンドウに表示されます。

```
[-1 1 2 2,
-1 2 4 3,
-1 1 7 4,
1 3 8 4,
1 1 10 9,
0 1 11 11,
-1 1 14 14,
```

```
-1 1 16 15,
1 1 18 16,
-1 1 19 18,
0 1 22 20,
0 1 26 24,
1 1 29 27,
1 1 34 33]
```

この例で表示される行列で、第1列の値は列に対して行われたアクションを示します。-1は削除、0は置換、1は追加を表します。第2列は、そのアクションの作用を受けた行の数を示します。第3列と第4列は、2つのデータテーブルの中で作用を受けた行の数を、データテーブルの順に（Students1.jmp、Students2.jmp）示します。

Summaryによる要約テーブルの作成

要約テーブルは、平均、中央値、標準偏差、最小値、最大値など、データテーブルの要約統計量を含むテーブルです。デフォルトでは、要約テーブルはでその元となったテーブルにリンクされています。要約テーブル内で行を選ぶと、元のテーブル内でも対応する行が強調表示されます。

```
dt << Summary( <private>, <invisible>, <Group( column )>, <Weight( column )>,
  <Freq( column )>, <N>, <Mean( column )>, <Std Dev( column )>, <Min( column )>,
  <Max( column )>...
```

「要約」起動ウィンドウにリストされる統計量であれば、どれでも含めることができます。

次の例は、「Fitness.jmp」サンプルデータテーブルから要約テーブルを作成します。デフォルトでは、要約テーブルに各水準の行数を示す「N」列が含まれます。次のスクリプトを実行すると、「**性別**」ごとの「**走行時間**」の平均の表が作成されます。

```
dt = Open( "$SAMPLE_DATA/Fitness.jmp" );
dt << Summary(
        Group( :性別 ),
        Mean( :走行時間 ),
);
```

スクリプトでFreqやWeightの列を指定すると、データテーブルで事前に設定されている［度数］や［重み］の役割の列より優先されます。事前に設定された［度数］や［重み］の役割の列を使用しないようにするには、"none"というキーワードを指定します。

```
dt = Open( "$SAMPLE_DATA/Fitness.jmp" );
dt << Summary(
        Group( :性別 ),
        Mean( :走行時間 ),
        Weight( "none" )
);
```

要約テーブルが元のテーブルにリンクしないようにするには、`Link to Original Data Table( 0 )`を指定します。

```
dt = Open( "$SAMPLE_DATA/Fitness.jmp" );
dt << Summary(
      Group( :性別 ),
      Mean( :走行時間 ),
      Link to Original Data Table( 0 )
);
```

データテーブルに登録（Subscribe）する

データテーブルに変化があったときにメッセージを受け取るようにするようには、Subscribeメッセージを使います。たとえば、列が追加または削除されたときにログにメッセージを送りたいとしましょう。

基本的な構文は次のとおりです。

```
dt << Subscribe( "name"(<"client">), On Delete Columns | On Add Columns | On Add
   Rows | On Delete Rows | On Rename Column | On Close | On Save | On Rename (
   function ) );
```

第1引数は、登録（または"クライアント"）の名前です。これにより、後で設定を解除することが可能になります。

アプリケーションは、データテーブルのクライアント（「一変量の分布」など、ほとんどのビルトインプラットフォーム）としても登録できます。閉じようとしているデータテーブルにクライアントがある場合、データテーブルを必要とする可能性のあるアプリケーションが開いていることが警告されます。

それぞれの登録は、解除するまで有効です。登録の解除は次のように行います。

```
dt << Unsubscribe("keyname", On Delete Columns | On Add Columns | On Add Rows | On
   Delete Rows | On Close | On Col Rename | All);
```

次の例は、行が追加または削除された場合に、ログにメッセージを送ります。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
delRowsFn = Function( {a, b, rows},
   dtname = ( a << Get Name());
   Print( dtname );
   Print( b );
   Print Matrix( rows );
);
addRowsFn = Function( {a, b, insert},
   dtname = ( a << Get Name() );
   Print( dtname );
   Print( b );
   Print( insert );
);
dt << Subscribe( "Test Delete", onDeleteRows( delRowsFn, 3 ) );
dt << Subscribe( "Test Add", onAddRows( addRowsFn, 3 ) );
```

Subscribe to Data Table List()は、新しいデータテーブルを追加したとき、または閉じたときに、通知が出されるようにするデータテーブルのリストへの登録を行います。次の例は、登録および登録解除の方法を示しています。

```
f2 = Function( {x}, Show( x ) );
Show( Subscribe to Data Table List( "My Data", OnClose( f2 ) ) );
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
Wait( 2 );
Close( dt );
Show( Unsubscribe to Data Table List( "My Data" ) );
Subscribe to Data Table List( "My Data", OnClose( f2 ) ) = " 一意の名前 ";
x = Data Table( "Big Class" );
Unsubscribe to Data Table List( "My Data" ) = Empty();
```

空のアプリケーション名

空のアプリケーション名とともにSubscribeが呼び出された場合、JMPは一意の名前を生成して呼び出し元に戻します。次の例では、appname2がクライアントとしてデータテーブルに登録されます。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
appname1 = dt << Subscribe( "", On Close( Print( " データテーブルを閉じる " ) ) );
appname2 = dt << Subscribe(
   ""( "client" ),
   On Close(
      Function( {dtab},
         dtname = ( dtab << Get Name() );
         Print( dtname );
      )
   )
);
dt << Unsubscribe( appname1, On Close );
dt << Run Script( " 一変量の分布 " )
92h
```

行列とデータテーブル間でデータを移動する

行列とデータテーブルの間でデータを移動する方法については、「データ構造」章の「行列とデータテーブル」（179ページ）を参照してください。

列

この節では、データテーブルの列に対して実行できる、作成、グループ化、属性やプロパティの設定および取得などのアクションについて説明します。

メモ: JMPでは、数学記号とギリシャ文字が表示できます（[環境設定]の「フォント」で設定）。つまり、T2乗限界などの列を保存する場合、列名は「T2乗限界」（特殊記号なし）と「T^2限界」（特殊記号あり）のどちらでもかまいません。ただし、その列を参照するときに、列名を正確に記述しないとエラーになるので注意が必要です。

データ列のオブジェクトにメッセージを送る

データテーブルの参照にデータテーブルメッセージを送るときと同様に、データ列オブジェクトへの参照に列メッセージを送ることができます。Column関数は、データ列の参照を戻します。この関数の引数は、引用符で囲んだ名前（または、引用符で囲んだ名前を導き出すもの）または番号のどちらかです。

```
Column( "年齢" );          // 「年齢」列への参照
col = Column( 2 );       // 2番目の列への参照を割り当てる
:Name( "利益($M)" ) // 引用符で囲まれた名前
```

このマニュアルでは、データ列の参照を*col*で表します。データ列オブジェクトに送ることのできるメッセージは、[**ヘルプ**] > [**スクリプトの索引**] を選択した後、[**オブジェクト**] > [**Data Table**] > [**Column Scripting**] で確認できます。または、Show Properties()コマンドを次のように使用します。

```
Show Properties( col );
```

メモ: 列の個々の値を参照する場合、列の参照に添え字を付けなければなりません。添え字を使わないと、列オブジェクト全体を参照することになります。

データ列の参照をグローバル変数に格納した後で、列に変更を加えたいときは、データ列の参照にメッセージを送ります。

列にメッセージを送る方法は、データテーブルにメッセージを送る方法と同じです。オブジェクト、<<演算子、その後に括弧で囲んだ引数を持つメッセージを記述するか、またはSend()関数を使ってオブジェクトとその後にメッセージを記述します。引数を必要としないメッセージもあるので、末尾の括弧はオプションです。

```
col << message( arg, arg2, ... );
Send( col, message(arg, arg2, ... ) );
```

メッセージは、データテーブルや他のタイプのオブジェクトの場合と同様、次のようにして1つにまとめたり、リストにしたりできます。

```
col << message << message2 << ...
col << {message, message2, ...};
```

ヒント: 列を削除するにはメッセージをデータ*テーブル*の参照に送る必要があります。オブジェクトを削除できるのはそのオブジェクトのコンテナだけで、オブジェクトは自身を削除できないからです。

列の参照によるセル値へのアクセス

列のセルに含まれている値にアクセスするには、列の参照で添え字を使います。添え字を使わないと、列オブジェクト全体を参照することになります。

```
x = col[irow]    // 特定の行
x = col[]        // 現在の行
col[irow] = 2;   // 割り当て式の左辺として
dt << Select Where( col[]<14 );  // WHERE節で使用
```

値ラベルを持つセル値へのアクセス

列に［値ラベル］プロパティが設定されていて、実際のデータ値ではなく、その値ラベルにアクセスしたいときがあります。そのような場合は、formattedオプションを使用してください。次の例を見てみましょう。

```
dt = Open( "$SAMPLE_DATA\Big Class.jmp" );
dt << Run Script( "年齢に値ラベルをセット" );
x = Column( dt, "年齢", formatted )[1];
      Show( x );
y = Column( dt, "年齢" )[1];
      Show( y );
```

このログには、*x*として表示形式適用後の値（twelve）、および*y*として実際のデータ値（12）が表示されます。

列の作成

データテーブルに新しい列を追加するには、New Columnメッセージをデータテーブルの参照に送ります。第1引数の列名は必須です。列名は、引用符で囲むか、名前を導き出す式の形で指定します。

```
dt = Open( "MyData.jmp" );
dt << New Column( "ウエハー" );
```

または

```
a = "ウエハー";
dt << New Column( a );
```

テーブルにすでに同名の列がある場合は、新しい列名に連番式の数字が追加されます（**ウエハー**、**ウエハー2**、**ウエハー3**など）。

特に指定しない限り、列の値は連続量の数値で、列幅は12文字です。

- データタイプ（数値、文字、行の属性、または式）
- 尺度（連続、名義、順序、多重応答、非構造化テキスト、またはベクトル）
- 列幅（数値列の場合のみ）
- 数値の表示形式

次の例は、データタイプが数値で、尺度が連続尺度、列幅が5の新しい列を作成します。数値の表示形式は「最適」に設定します。

```
dt << New Column( "ウエハー", Numeric, "Continuous", Format( "最適", 5 ) );
```

次の列は、文字列の列を作成し、自動的に名義尺度を割り当てます。

```
dt << New Column( "姓", Character );
```

列の属性に合わせて、式やその他のスクリプトメッセージを入れることもできます。

```
dt = Open( "$SAMPLE_DATA\Big Class.jmp" );
dt << New Column( "比率", Numeric, "Continuous", Formula( :Name("身長(インチ
  )")/:Name("体重(ポンド)") ) );
dt << New Column( "マーカー",
   Row State,
   Set Formula( Marker State( age - 12 ) )
);
```

特定の列を後で操作する（グループ化する、データタイプを変更するなど）予定がある場合は、次のようにして列の参照を作成します。

```
myCol = dt << New Column( "生年月日" );
```

列にデータを挿入する

列にデータを挿入するには、Values、またはそれと等価のSet Valuesを使用します。各セルの値をリストに含めます。

次の例は、新しいデータテーブルに「姓」という新しい列を追加し、そこに「Smith」、「Jones」、「Anderson」という3つの値を挿入します。

```
dt = New Table( "My Data");
dt << New Column( "姓", Character, Values( {"Smith", "Jones", "Anderson"} ) );
```

列には、数値を入れることもできます。次の例は、新しいデータテーブルに**「行番号」**という新しい列を追加します。N Row関数がテーブル内の行の数を戻し、新しい列のすべての行に1から始まる数値が挿入されます。

```
dt = New Table( "My Data");
dt << New Column( "行番号",
   Numeric,
   Values( 1 :: N Row() ),
   Format( "最適", 5 )
);
```

乱数の列を追加するには、乱数関数と式を挿入します。

```
dt = Open( "$SAMPLE_DATA\Big Class.jmp" );
dt << New Column( "乱数", Numeric, Formula( Random Uniform() ) );
```

また、各セルに一定の数値を挿入することもできます。次の例は、各セルに「5」を挿入します。

```
dt << New Column( "数" );
:数 << Set Each Value( 5 );
```

New Column()はビルトイン関数として使用することもできます。データテーブルの参照に対して<<(Send)コマンドを使用しない方法です。ビルトイン関数として使うと、列は現在のデータテーブルに追加されます。

```
dt << New Column( "住所" ); // コマンドをデータテーブルの参照に送る
dt << New Column( "住所" ); // 列は現在のデータテーブルに追加される
```

一度に複数の列を追加

Add Multiple Columnsメッセージは、一度に複数の列を作成します。引数は、列名の接頭辞、列の数、列の挿入位置（Before First、After Last、After(col)）、データタイプ（Numeric、Row State、Character(width)）です。また、数値列を対象とするオプションの引数に、フィールド幅があります。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Add Multiple Columns( "最初", 2, Before First, Row State );
// 「最初 <n>」という行の属性列を 2 つ、最初の列の前に挿入する

dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Add Multiple Columns( "中間", 3, After( :Name("身長 (インチ)") ), Numeric );
// 「中間 <n>」という数値列を 3 つ、「身長 ( インチ )」列の後ろに挿入する

dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Add Multiple Columns( "最後", 4, After Last, Character( 4 ) );
// 「最後 <n>」という名前の文字列の列を 4 つ、最後の列の後ろに追加する
```

列名は接頭辞です。同名の列が複数追加される場合、列名に連番が付きます（**最初 1**、**最初 2**、など）。

列のグループ化

列をグループ化するには、対象となる列のリストを引数として指定したGroup Columnsメッセージをデータテーブルに送ります。たとえば、次のコードは、「Big Class.jmp」データテーブルを開いて、「**年齢**」と「**性別**」の列をグループ化します。

```
dt = Open( "$SAMPLE_DATA\Big Class.jmp" );
dt << Group Columns( {: 年齢 , : 性別 } );
  "年齢など"
```

また、グループに含める最初の列の列名に、グループに含める列数を添えて、送ることもできます。その場合、名前で指定した列から数えて *n* 個の列がグループに含まれます。次の行は、前述の行と等価で、「**年齢**」と「**性別**」をグループ化します。

```
dt << Group Columns( : 年齢 , 2 );
```

グループ名は、引数で指定した第1列に基づきます。前述の例では、グループに自動的に「**年齢など**」という名前が付きます。名前をカスタマイズするには、グループ名を第1引数に含めます。

```
dt << Group Columns( "グループ", : 年齢 , 2 );
```

列のグループ化を解除するには、対象となる列のリストを引数として指定したUngroup Columnsメッセージを使います。たとえば、次の行は、前述の例でグループ化した2つの列のグループ化を解除します。

```
dt << Ungroup Columns( {: 年齢 , : 性別 } );
```

グループ化とその解除に関しては、次のことに注意してください。

- どちらのメッセージも引数には1つのリストを指定します。リストは括弧で囲む必要があります。

- 1つのメッセージで複数のグループを作成することはできません（Group Columnsで2つの列リストを指定するなど）。2つの別々のGroup Columnsメッセージを送る必要があります。
- Ungroup Columnsメッセージでは、列のグループ名ではなく、グループ解除する列のリストを指定します。グループから一部の列を削除できます。たとえば、次の行は、4つの列から成るグループを作成します。

```
dt << Group Columns( {: 年齢 , : 性別 , :Name(" 身長 ( インチ )"), :Name(" 体重 ( ポンド )")}
   );
```

次の行は、このうち2つの列だけをグループから削除します。

```
dt << Ungroup Columns( {: 年齢 , : 性別 } );
```

グループ列は「**身長(インチ)**」と「**体重(ポンド)**」ですが、グループ名は「**年齢**」を含んだままであることに注目してください。いったんグループが作成されたら、元々グループ化されていた最初の列を削除しても名前は変わりません。

列グループまたは名前の取得

列グループまたは名前を取得するには、Get Column GroupまたはGet Column Groups Namesを使用します。この例では、まず前の節で示した手順でグループの列を2つ作成します。Get Column Group()は指定のグループに含まれる列の名前を戻します。Get Column Groups Names()は列グループの名前を戻します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Group Columns( {: 年齢 , : 性別 } );
  "年齢など"
dt << Group Columns( {:Name(" 体重 ( ポンド )"), :Name(" 身長 ( インチ )")} );
  "身長(インチ)など"
dt << Get Column Group( " 年齢など " );
  {:age, :sex}
dt << Get Columns Groups Names();
  {"年齢など", "身長(インチ)など"}
```

列グループの選択

列グループを選択する、または選択を解除するには、次のメッセージを使用します。

```
dt << Select Column Group( " 名前 " ); // グループ内の列を選択
dt << Deselect Column Group( " 名前 " ); // グループ内の列の選択を解除
```

次の例は、**X**と**Y**の列、「**オゾン**」から「**鉛**」までの列をそれぞれグループ化し、データテーブル内でこれらの列を選択します。

```
dt = Open( "$SAMPLE_DATA/Cities.jmp" );
dt << Group Columns( "xy", {:X, :Y} );
dt << Group Columns(
   " 汚染物質 ",
   : オゾン :: : 鉛
);
dt << Select Column Group( "xy", " 汚染物質 " );
```

列グループの名前の変更

列グループの名前を変更するには、次のメッセージを使用します。

```
dt << Rename Column Group( "名前", "変更後の名前" );
```

次の例は、列グループの名前を「**xy**」から「**座標**」に変更します。

```
dt = Open( "$SAMPLE_DATA/Cities.jmp" );
dt << Group Columns( "xy", {:X, :Y} );
dt << Group Columns(
   "汚染物質",
   :オゾン :: :鉛
);
Wait( 3 );
dt << Rename Column Group( "xy", "座標" );
```

列グループの移動

列グループを移動させるには、次のメッセージを使用します。

```
dt << Move Column Group (To First | To Last | After (col) "名前")
```

次の列は、「**汚染物質**」グループの列を最後尾に移動させます。

```
dt = Open( "$SAMPLE_DATA/Cities.jmp" );
dt << Group Columns( "xy", {:X, :Y} );
dt << Group Columns(
   "汚染物質",
   :オゾン :: :鉛
);
dt << Move Column Group( To Last, "汚染物質" );
```

列の選択

列を選択するには、Set Selectedメッセージを使用します。

```
col << Set Selected( 1 );
```

たとえば、「Big Class.jmp」データテーブルのすべての連続尺度の列を選択するには、次のスクリプトを使用します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
cc = dt << Get Column Names( "Continuous" );
ncols = N Items( cc );
For( i = 1, i <= ncols, i++,
   cc[i] << Set Selected( 1 )
);
```

選択されている列の取得

現在選択されている列のリストを取得するには、Get Selected Columnsメッセージを使います。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
:age << Set Selected( 1 );
:sex << Set Selected( 1 );
dt << Get Selected Columns();
  {:age, :sex}
```

選択されている列のリストを、"string"引数を使って文字列として戻します。

```
dt << Get Selected Columns( "string" );
  {"年齢", "性別", "身長(インチ)"}
```

どの列が選択されているかがわかったら、それらの列に対してアクションを行うスクリプトを書くことができます。または、列を繰り返し選択し、一度に1つずつ処理するスクリプトを書きます。

列を選択してから取得する場合は、列にSet Selectedメッセージを送ります。詳細は、「列属性」（327ページ）を参照してください。

列への移動

特定の列を選択し、そこに移動するには、Go Toメッセージを使用します。

```
dt << Go To( 列名 | 列番号 );
```

列数が多いデータテーブルに対し、このメッセージを使うと、データテーブルが左へとスクロールされ、第1列が選択された形で画面に表示されます。

```
dt = Open( "$SAMPLE_DATA/Tiretread.jmp" );
dt << Go To( 1 );
```

列を並べ替えて移動

以下のメッセージを使うと、データテーブル内の列を並べ替えることができます。

```
dt << Reorder By Name;           // 名前順
dt << Reverse Order;             // 現在と逆の順序
dt << Reorder By Data Type;      // 行の属性、文字列、数値の順
dt << Reorder By Modeling Type;  // 連続尺度、順序尺度、名義尺度の順
dt << Original Order;            // 保存されている順序
```

次のコマンドは、現在選択されている列を指定した移動先に移動します。

```
dt << Move Selected Columns( To First );
dt << Move Selected Columns( To Last );
dt << Move Selected Columns( After( "名前" ) );
```

次の構文を使うと、事前にデータテーブル内の列を選択しなくても、列を移動できます。

```
dt << Move Selected Columns( {"名前"}, To First );
```

```
dt << Move Selected Columns( {"名前"}, To Last );
dt << Move Selected Columns( {"名前"}, After( "名前" ) );
```

列スイッチャーの追加

JSLを使ってレポートウィンドウに列スイッチャーを追加します。列スイッチャーがあると、分析を再度呼び出す手間をかけずに、異なる列をすばやく分析できます。

```
obj << Column Switcher(<default_col>(<col1>, <col2>, ...);
// レポートに列スイッチャーを追加する

obj << Remove Column Switcher();
// レポートから列スイッチャーを削除する
```

たとえば、「Car Poll.jmp」データテーブルの「分割表」レポートに列スイッチャーを追加するには、次のようにします。

```
dt = Open( "$SAMPLE_DATA/Car Poll.jmp" );
obj = Contingency(
   Y( :サイズ ),
   X( :Name("既婚 / 未婚") )
);
ColumnSwitcherObject = obj <<
Column Switcher(
   :Name("既婚 / 未婚"),
   {:性別 , :生産国 , :Name("既婚 / 未婚")}
);
ColumnSwitcherObject << Set Size( 200 );
// スイッチャーの幅を示すピクセル数
ColumnSwitcherObject << Set NLines( 6 );
// スイッチャーに表示する列の数
```

列スイッチャーの詳細については、『JMPの使用法』の「JMPのレポート」章を参照してください。

選択された列の圧縮

データテーブルが大きい場合に、サイズを最小に抑えるには、Compress Selected Columnsメッセージを使用します。各列ができる限りコンパクトな形に圧縮されます。

```
dt << Compress Selected Columns( { 列名 , 列名 } );
```

たとえば、「Big Class.jmp」内の「**年齢**」、「**性別**」、「**身長(インチ)**」、「**体重(ポンド)**」の各列を圧縮するには、次のようにします。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Compress Selected Columns(
{:年齢 , :性別 , :Name("身長(インチ)"), :Name("体重(ポンド)")}
);
```

この機能の詳細については、『JMPの使用法』における「データの入力と編集」章を参照してください。

列の削除

列を削除するには、Delete Columnsメッセージを送って、削除する列（複数可）を指定します。複数の列を削除するには、列を複数の引数として個別に指定するか、または1つのリストで指定します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Delete Columns( "体重（ポンド）" );
dt << Delete Columns( "体重（ポンド）", "年齢", "性別" );
dt << Delete Columns( {"体重（ポンド）", "年齢", "性別"} );
```

引数がない場合、Delete Columnsは、選択されている列を削除します。詳細については、「列属性」（327ページ）を参照してください。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
Column("年齢") << Set Selected;
dt << Delete Columns();
```

列名の取得

Column Name(*n*)は、*n*番目の列の参照を戻します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
Column Name( 2 );
  年齢
```

戻される値は、引用符で囲まれた文字列ではなく、「年齢」列の参照値です。これは、スクリプト内で通常、実際の列の名前を使う場所で、どこでもこの演算子を使用できます。たとえば、この演算子に添え字を使うことができます。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
Column Name( 2 )[1];
  12
```

名前をテキストの文字列として取得したい場合は、Charでクォートします。

```
Char( Column Name( 2 ) );
  "年齢"
```

データテーブルのすべての列の名前をリストで取得するには、Get Column Namesを実行します。

```
dt << Get Column Names( 引数 );
```

オプションの*引数*は、次のようにGet Column Names関数の出力を制御します。

- Numeric、Character、またはRow Stateを指定すると、これらのタイプに相当する列の名前だけが戻ります。
- "Continuous"、"Ordinal"、または"Nominal"を指定すると、指定した尺度の列の名前だけが戻ります。

- Stringを指定すると、列名ではなく文字列のリストが戻ります。

たとえば、「Big Class.jmp」データテーブルで連続尺度の数値列を選択するには、次のスクリプトを使用します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
names = dt << Get Column Names(Numeric, "Continuous")
  {Name( "身長(インチ)" ), Name( "体重(ポンド)" )}
```

列属性

データテーブル列へのメッセージを使うと、名前、データ、状態、メタデータなど、さまざまな列の属性や特性をすべて制御できます。これらのメッセージは対になっており、一方は各属性を設定（set）または割り当て、もう一方は各属性の現在の設定を取得（get）または問い合わせます。

たとえば、列の非表示、除外、ラベル、スクロールロック／ロック解除などをスクリプトで実行できます。値はブール値なので、1を入力するとその属性がオンになり、ゼロを送るとオフになります。

次の例では、**「名前」**列が「表示する」、「除外しない」、「ラベルあり」、「スクロールロック」の状態になります。

```
Column( "名前" ) << Hide( 0 );
Column( "名前" ) << Exclude( 0 );
Column( "名前" ) << Label( 1 );
Column( "名前" ) << Set Scroll Locked( 1 );
```

メモ: 各種の引数を設定するメッセージ（Set Name、Set Values、Set Formulaなど）はすべてSetで始まりますが、Setという語はSet Name 以外のメッセージでは省略可能です（Nameはすでに名前に特殊な文字を使うときのコマンドとして使われています）。好きな方の形式、または覚えやすい方の形式を使ってください。引数の現在の設定値を取得するメッセージ（Get Formulaなど）は、SetではなくGetで始まり、Getは*省略できません*。

列の選択を解除するには、データテーブルオブジェクトにClear Column Selectionメッセージを送ります。

```
dt << Clear Column Selection;
```

列名の設定または取得

Set Nameを使って、列に名前をつけたり列の名前を変更したりできます。Get Nameは列の名前を戻します。次の例は、第2列（**「年齢」**）を**「比率」**に変更します。その後、現在の列名をログに戻します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
col = Column( 2 );
col << Set Name( "比率" );
col << Get Name;
  "比率"
```

列の値の設定または取得

同様に、Set Valuesは列に値を設定します。変数が文字タイプの場合、引数はリストである必要があります。数値タイプの場合は行列（ベクトル）です。値の数がデータテーブルの現在の行数より多い場合は、必要な行が追加されます。Get Valuesは、値をリストまたは行列の形で戻します。Get As MatrixはGet Valuesに似ていますが、数値列の値を戻します。

```
col << Set Values( myMatrix ); // 数値変数の場合
col << Set Values( myList ); // 文字変数の場合
col << Get Values;           // 行列を戻す。文字の場合はリストを戻します
col << Get Matrix(<列名のリスト>|<列番号のリスト>|<列範囲>; // 指定の列を行列として戻す
```

次の例は、値のリストと行列を戻します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
Column( "名前" ) << Values( {Fred, Wilma, Fred, Ethel, Fred, Lamont} );
myList = :name << Get Values;
  /* {"Fred", "Wilma", "Fred", "Ethel", "Fred", "Lamont", "JAMES", "ROBERT",
     "BARBARA", ...]を戻す*/

Column( "年齢" ) << Values( [28, 27, 51, 48, 60, 30] );
myVector = :年齢 << Get Values;
  // [28, 27, 51, ... ]を戻す
myMatrix = :Name("体重（ポンド）") << Get as Matrix;
  // [95, 123, 74, ... ]を戻す
```

値ラベルの設定または取得

メモ: 値ラベルの詳細については、『JMPの使用法』の「列情報ウィンドウ」章を参照してください。

値ラベルを使うと、簡略化されたデータの意味を説明するラベルを表示できます。たとえば、0と1の値を持つデータ列があり、0は男性、1は女性を表すとします。0の値ラベルを「男性」、1の値ラベルを「女性」とすると、わかりやすくなります。

値ラベルは、次の3つの方法のどれかで指定できます。「Big Class.jmp」サンプルデータテーブルでは、Mが男性、Fが女性を表しています。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
:性別 << Value Labels( {"F", "M"}, {"女性", "男性"} ); // 2つのリストを使う
:性別 << Value Labels( {"F", "女性", "M", "男性"} ); // ペアのリストを使う
:性別 << Value Labels( {"F" = "女性", "M" = "男性"} ); // 割り当てのリストを使う
```

値のラベルを有効にするには、列にUse Value Labelsメッセージを送ります。

```
:性別 << Use Value Labels( 1 );
```

列の実際の値を表示するには、次のように指定します。

```
:性別 << Use Value Labels( 0 );
```

同じメッセージを使って、データテーブルのすべての列に対して値ラベルのオン／オフを切り替えることができます。

```
dt << Use Value Labels( 1 );
```

データと尺度の設定または取得

JSLを使って列のデータタイプを設定または取得することができます。設定できるタイプは、Character（文字）、Numeric（数値）、Row State（行の属性）です。次の例は、「Big Class.jmp」サンプルデータテーブルに文字タイプのデータを入れる新しい列を追加します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << New Column( "新規" );
Column( "新規" ) << Data Type( Character );
Column( "新規" ) << Get Data Type;
  "文字"
```

列の尺度を設定または取得するには：

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
col = New Column( "新規" );
col << Modeling Type( "Continuous" );
col << Get Modeling Type;
  "順序尺度"
```

列のデータタイプを変更するときは、次のようにして列の形式を指定できます。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
col = New Column( "日付" );
Column( "日付" ) << Data Type( Numeric, Format( "ddMonYYYY" ) );
```

形式の設定または取得

Formatメッセージは、数値と日付／時間の形式を制御します。最初の引数は、列情報ウィンドウにある表示形式の選択肢を引用符で囲んだ文字列です。後続の引数は、表示形式によって異なります。フィールド幅も個別に設定できます。

```
col << Format( "Best", 5 ); // 幅が 5
col << Format( "Fixed Dec", 9, 3 ); // 幅が 9、小数点以下の桁数が 3
col << Format( "PValue", 6 );
col << Format( "d/m/y", 10 );
col << Set Field Width( 30 );
```

日付の形式では、Formatメッセージにより日付をデータテーブルにどのように表示するかを指定します。データの入力形式や、入力または編集に備えて現在のセルを選択したときの表示形式を設定するには、Input Formatメッセージを使用します。

```
col << Format( "d/m/y", 10 ); // 日付を日 - 月 - 年の順序で表示する
col << Input Format( "m/d/y" ); // 日付を月 - 日 - 年の順序で入力する
```

日付時間の詳細については、「データタイプ」章の「日付時間の関数と形式」（123ページ）を参照してください。

メモ：Format *メッセージ*と、指定された形式に従って数値を文字列に変換する Format *関数*を混同しないでください（「データタイプ」章の「日付時間の関数と形式」（123ページ）で説明しているように、通常、Format関数は日付／時間の表記に使われます）。オブジェクトにメッセージを送った場合は、同じ名前の関数を使った場合と結果がまったく異なります。

列の現在の形式を取得するには、次のようにGet Formatメッセージを実行します。

```
col << Get Format;
```

計算式の設定、取得、評価

次の例は、計算式を設定、取得、および評価する方法を示しています。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
col = New Column( "比率" ); // 列を作成し、その参照を格納する
col << Set Formula( :Name("身長（インチ）")/:Name("体重（ポンド）") );
// 計算式を設定する
col << Eval Formula; // 計算式を評価する
col << Get Formula; // 式 :height / :weight を戻す
```

列の値をスクリプトの中で使用する場合は、計算式を評価するコマンドを必ず加えてください。計算式の評価のタイミングは、JMPのバージョンによって異なります。次の点を念頭に置いてください。

- 計算式を追加すると、バックグラウンドで評価するようにスケジュールされます。スクリプト実行中の列の値に依存するような場合は、これが問題となる可能性があります。
- 1つの列を評価するときには、列にEval Formulaコマンドを送ることができます。これは、列を作成するコマンド内の計算式の節の直後で実行できます。

  ```
  dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
  dt << New Column( "比率",
     Numeric,
     Formula( :height / :weight )
     Eval Formula
  );
  ```

  ここでのFormula()はSet Formula()の別名です。

  ただし、計算式を追加する作業がすべて終わってから、次のようにRun Formulasコマンドを使って、正しい順序ですべての計算式を実行させるようにしてください。

  ```
  dt << Run Formulas;
  ```

- Eval Formulaは計算式の評価をしますが、バックグラウンドでの再計算を行うため、Run Formulasコマンドの方がEval Formulaより適しています。システムのバックグラウンドで行われる計算タスクは、計算式が正しい順序で評価されるよう、十分な注意を払う必要があります。Run Formulasは、すべての計算処理が終了するまで、他のタスクを呼び出しません。

- 同一の値のセットを生成させるために乱数関数とRandom Reset(*seed*)の機能を使用している場合には、Run Formulasコマンドを使うと2回目の評価を行わずに済みます。

範囲チェックおよびリストチェックの設定と取得

JSLを使って、リストチェックおよび範囲チェックのプロパティを操作できます。次の例は、「Big Class.jmp」サンプルデータテーブルを使用します。

「性別」という列のリストチェックプロパティを設定し、クリアします。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
Column( "性別" ) << List Check( {"M", "F"} ); // プロパティを設定
Column( "性別" ) << List Check(); // プロパティをクリア
```

範囲チェックでは、表9.1の構文を使って範囲を指定する必要があります。

表9.1 範囲チェックの構文

指定する範囲	使用する関数
$a \leq x \leq b$	LELE(a, b)
$a \leq x < b$	LELT(a,b)
$a < x \leq b$	LTLE(a,b)
$a < x < b$	LTLT(a,b)

次の例は、「年齢」列の値が0より大きく、120未満でなければならないことを指定します。

```
Column( "年齢" ) << Range Check( LTLT( 0, 120 ) );
```

どの演算子の前にもNotを置くことができます。また、上側または下側だけの範囲も指定できます。次の例は、「年齢」列の値が12以上でなければならないことを指定します。

```
Column( "年齢" ) << Range Check( not( LT( 12 ) ) );
```

範囲チェックの属性をクリアするには、次のように空のRange Check()を実行します。

```
Column( "年齢" ) << Range Check();
```

列に割り当てられているリストチェックまたは範囲チェックを取得するには、列にGet List CheckまたはGet Range Checkメッセージを送ります。

```
Column( "性別" ) << Get List Check;
Column( "年齢" ) << Get Range Check;
```

たとえば、「Big Class.jmp」の「**年齢**」列にGet Range Checkを送ると、次のような出力が表示されます。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
Column( "年齢" ) << Range Check( LTLT( 0, 120 ) );
Column( "年齢" ) << Get Range Check;
  Range Check( LTLT( 0, 120 ) )
```

Set Property、Get Property、およびDelete Propertyを使って、リストチェックと範囲チェックを設定、取得、削除することもできます。詳細については、「列プロパティ」（332ページ）を参照してください。

メモ: 列の範囲チェックに関する処理をJSLで行う場合は、すべての警告が、インタラクティブなウィンドウではなくログウィンドウに表示されます。

列スクリプトの取得

Get Scriptは、列を作成するためのスクリプトを戻します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
New Column( "比率", Set Formula( :Name("身長(インチ)")/ :Name("体重(ポンド)")) );
Column( "比率" ) << Get Script;
  New Column("比率",
     Numeric,
     "Continuous",
     Format( "Best", 10 ),
     Formula(Name("身長(インチ)") / :Name("体重(ポンド)"))
  )
```

役割の事前選択

列の役割をあらかじめ選択するには、Preselect Roleメッセージを使用します。選択できる役割は、No Role、X、Y、Weight、およびFreqです。Get Roleメッセージは、現在の設定を戻します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
col = New Column( "新規" );
col << Preselect Role( X );
col << Get Role;
```

列のロック

列をロックまたはロック解除するには、ブール引数を指定してLockまたはSet Lockを使います。Get Lockは現在の設定を戻します。

```
col << Lock( 1 ); // ロック
col << Set Lock( 0 ); // ロック解除
col << Get Lock; // 現在の状態を表示
```

列プロパティ

ヒント: JSLで使用できるプロパティは、「列情報」ウィンドウの［**列プロパティ**］メニューにあるものと同じです。各プロパティの引数は、「列情報」ウィンドウの設定に対応します。構文を習得する簡単な方法は、まず列情報ウィンドウで目的のプロパティを設定し、次にGet Propertyを使ってJSLを表示することです。

データ列には、多数のメタデータの属性オプションがあります。この属性は、Get Property、Set Property、およびDelete Property使って、設定、取得、または削除することができます。

```
col << Set Property( " プロパティ名 ", { 引数リスト } );
col << Get Property( " プロパティ名 " );
col << Delete Property( " プロパティ名 " );
```

設定する際のプロパティ名は、常にSet Propertyの第1引数になり、設定するプロパティによって後続の引数が決まります。

- Get PropertyとDelete Propertyは常にプロパティ名を指定する引数を1つだけとります。
- Get Propertyは、プロパティの設定を戻します。Delete Propertyは、列からプロパティを完全に削除します。
- 列に設定されている列プロパティ名のリストを取得するには、列を指定し、Get Property Listを使います。

複数のプロパティを設定する場合は、Set Propertyメッセージを個別に複数回送る必要があります。単一のJSLステートメントに複数のメッセージをまとめることもできます。

```
col << Set Property( "Axis",{ Min(50), Max(180)} )<< Set Property( " ノート ", " 比率
   を取得する " );
```

プロパティの値を取得するには、引数に目的のプロパティの名前を指定してGet Propertyメッセージを送ります。

```
Column(" 比率 ") << Get Property( "Axis" ); // 軸の設定を戻す
```

列をラベル列として設定するには:

```
dt << Set Label Columns( col1, col2, col3 );
```

すべてのラベル列をクリアするには:

```
dt << Set Label Columns();
```

同じ構文をSet Scroll Lock Columnsでも使用できます。

変数を使用した列プロパティの設定

JSLで列プロパティを記述する際、列はすべての列プロパティのリポジトリであるという点を理解しておくことが重要です。列には、列プロパティの名前と、評価や引数の検証が行われていない状態の値が格納されます。JSLコードで変数を使用して列プロパティを追加するには、スクリプト内のすべての変数が実際の値に展開されなければなりません。

そのための1つの手段として、式を使う方法があります。次の例では、Eval Expr()関数がExpr()関数で使われている各変数を評価し、それぞれの値に置き換えます。外側のEval()関数は、置換が行われた後にステートメント全体を評価します。

```
dt = Open( "$SAMPLE_DATA\Big Class.jmp" );
// サンプルデータテーブルを開く

lLimit = 55;
uLimit = 70;
tLimit = 62.5;
// 仕様限界の値を割り当てる

Eval(
   Eval Expr(
      :Name("身長(インチ)") << Set Property(
      // 「仕様限界」列プロパティに限界値を設定する
         "Spec Limits",
         {LSL( Expr( lLimit ) ), USL( Expr( uLimit ) ), Target( Expr( tLimit ) ),
         Show Limits( 0 )}
      )
   )
);
```

列プロパティの設定と取得

表9.2に、よく使用される列プロパティを設定する例と取得する例を示します。各プロパティの詳細については、『JMPの使用法』の「列情報ウィンドウ」章を参照してください。

表9.2 データテーブルの列のプロパティ

プロパティ	説明	引数の使用例
Notes（ノート）	列についての注記を保存する。引用符で囲んだ文字列。	`col<<Set Property( "Notes", "Fisher のアヤメのデータから抽出 " );` `col<<Get Property( "Notes" );`
List Check（リストチェック） Range Check（範囲チェック）	列に入力できる値を指定する。	`col<<Set Property( List Check,{"F","M"} );` `col<<Set Property( "Range Check",LTLT(0, 120) );` `col<<Get Property( "List Check" );` `col<<Delete Property( "Range Check" );`
Missing Value Codes（欠測値のコード）	欠測値として扱う列の値を指定する。	`col<<Set Property( "Missing Value Codes", {0, 1} );`

表9.2 データテーブルの列のプロパティ（続き）

プロパティ	説明	引数の使用例
Value Labels（値ラベル）	値の代わりに表示するラベルを指定する。	col<<Value Labels({0 = "Male", 1 = "Female"}); ラベルのオンとオフは、ブール値を使って切り替えます。 col<<Use Value Labels(1);.
Value Ordering（値の順序）	レポートに表示するデータの順序を指定する。	col<<Set Property("Value Ordering", {"Spring", "Summer", "Fall", "Winter"});
Value Colors（値の色）およびColor Gradient（カラーグラデーション）	カテゴリカルデータまたは連続尺度のデータに色を設定する。	各値にJMPの色番号を割り当てることで色を指定する。 col<<Set Property("Value Colors", {"Female" = 3, "Male" = 5}); カラーテーマを指定するには: col<<Set Property("Value Colors", Color Theme("White to Blue")); カラーテーマを指定しなかった場合、カラーテーマの環境設定が使用されます。 カラーグラデーション、グラデーションの範囲および中間点を指定する。 col<<Set Property("Color Gradient", {"White to Blue", Range({18, 60, 25})});
Axis（軸）	ほとんどのプラットフォームで、グラフ軸の構築に使われる。ブール値を使うのが普通。	col<<Set Property("Axis",{Min(50), Max(180), Inc(0), Minor Ticks(10), Show Major Ticks(1), Show Minor Ticks(1), Show Major Grid(0), Show Labels(1), Scale(Linear)});
Units（単位）	カスタムで使われる。計測単位を指定する	col<<Set Property("units", "grams"); col<<Get Property("units");
Coding（コード変換）	実験計画（DOE）とあてはめに使われる。下限値と上限値のリスト。	col<<Set Property("Coding", {59,172}); col<<Get Property("Coding");

表9.2 データテーブルの列のプロパティ（続き）

プロパティ	説明	引数の使用例
Mixture（配合）	DOE、あてはめ、およびプロファイルに使われる。リストの形で「配合」列プロパティを指定する。	col<<Set Property("Mixture", {0.2, 0.8, 1, L PseudoComponent Coding}); col<<Get Property("Mixture");
Row Order Levels（データの出現順）	水準を、値の順にではなくデータ内での出現順に並べるよう指定する。	col<<Set Property("Row Order Levels", 1);
Spec Limits（仕様限界）	工程能力分析と変動性図に使われる。	col<<Set Property("Spec Limits", {LSL(-1), USL(1), Target(0)}); col<<Get Property("Spec Limits");
Control Limits（管理限界）	管理図に使われる。	col<<Set Property("Control Limits", {XBar(Avg(44), LCL(29), UCL(69))}); col<<Get Property("Control Limits");
Response Limits（応答変数の限界）	DOEで指定し、満足度関数で使われる。	col<<Set Property("Response Limits", {Goal("Match Target"), Lower(1,1), Middle(2,2), Upper(3,3)}); col<<Get Property("Response Limits"); Goal（目標）に指定できるのは、Maximize、Match Target、Minimize、Noneです。他の引数は、数値引数と満足度の引数をとります。
Design Role（因子の役割）	DOEに使われる。役割を1つ指定する	col<<Set Property("Design Role", "Covariate"); col<<Get Property("Design Role"); 指定できる役割は、Continuous、Discrete Numeric、Categorical、Blocking、Covariate、Mixture、Constant、Uncontrolled、Random Block、Signal、Noiseです。
Factor Changes（因子の変更）	因子変更の困難さ（Easy、Hard、Very Hard）を設定する。	col<<Set Property("Factor Changes", Hard) ; col<<Get Property("Factor Changes");

表9.2 データテーブルの列のプロパティ（**続き**）

プロパティ	説明	引数の使用例
Sigma	管理図に使われる。既知の標準偏差値を指定する。	col<<Set Property("Sigma",1.332); col<<Get Property("Sigma"); 管理図の種類によってsigmaの計算方法が異なります。
Distribution（分布）	列にあてはめる分布の種類を設定する。	col<<Set Property("Distribution", Distribution(GLog)); col<<Get Property("Distribution");
Time Frequency（時間の単位）	時間の単位の種類を設定する。	col<<Set Property("Time Frequency", Time Frequency("Annual")); col<<Get Property("Time Frequency");
Map Role（地図の役割）	地図シェープデータと名前データを接続するための列の使用方法を設定する。役割と、必要に応じてその他の情報を指定する。	col<<Set Property("Map Role", Map Role(Shape Name Use("filepath to data table", "column name"))); col<<Get Property("Map Role");
Supercategories（上位カテゴリ）	特定のカテゴリを1つのカテゴリにまとめる。「カテゴリカル」プラットフォームでのみ使用できる。	col<<Set Property("Supercategories", {Group("Genders", {"F", "M"})}); col<<Get Property("Supercategories");
Multiple Response（多重応答）	セル内の応答を区切る文字を指定する。	col<<Set Property("Multiple Response", Multiple Response(Separator(","))); col<<Get Property("Multiple Response"));
Profit Matrix（利益行列）	意思決定モデルを定義する重みを割り当てる。 重みを行列の形で、また、カテゴリを別のリストとして含むリストを指定する。行列の各行には、行列の列で指示されたカテゴリを予測したときの利益または重みが含まれる。 **注意：** 引数の行列は、「列プロパティ」ウィンドウに表示される利益行列の転置。	col<<Set Property("Profit Matrix", {[5 -1, -1 4, -2 -2], {"M", "F", "Others"}}); col<<Get Property("Profit Matrix");

表9.2 データテーブルの列のプロパティ（続き）

プロパティ	説明	引数の使用例
Expression Role（式の役割）	式を含む列に適用される。式をピクチャー、行列、式のいずれとして解釈するかを指定する。	`col<<Set Property( "Expression Role", Expression Role( "Picture", MaxSize( 640, 480 ), StretchToMaxSize( 1 ), PreserveAspectRatio( 1 ), Frame( 0 ) ) )`
[カスタムプロパティ]	カスタムで使われる。列情報ウィンドウ内の［**列プロパティ**］>［**その他**］に対応する。 第1引数はカスタムプロパティの名前で、第2引数は式。	`col<<Set Property( "記録日",12Dec1999 );` `long date( col<<Get Property( "記録日") );`

行

この節では、データテーブルの行の追加や操作を行うメッセージについて説明します。行メッセージは、データテーブルの参照に送られ、ほとんどの場合、現在選択されている行に対して動作します。行メッセージの中には、スクリプトでは実用的でないものも多くあります（たとえば、Move Rows）。

行の追加

行を追加するには、Add Rows メッセージを送って行数を指定します。また、新しい行をどの行の後に挿入するかも指定できます。引数に指定できるのは、数値または数値を導き出す式のどちらかです。

```
dt << Add Rows( 3 ); // データテーブルの最後に 3 行追加する
dt << Add Rows( 3, 10 ); /* 10 行目の後、11 行目の前に 3 行追加する */
```

Add Rows を利用すると、割り当てのリストを1つの引数として指定できます。割り当て式はカンマまたはセミコロンで区切ります。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Add Rows(
   {:名前 = "Peter", :年齢 = 14, :性別 = "M", :Name("身長（インチ）") = 61, :Name("体重（ポンド）") = 124}
);
add point = Expr(
   dt << Add Rows( {:xx = x ; :yy = y} )
);
```

複数の引数を指定して、複数のリストやリストのリストを送ることもできます。次のスクリプトは、さまざまな方法でAdd Rows コマンドを使ってデータテーブルを作成します。

```
dt = New Table( "都市" );
dt << New Column( "xx", Numeric );
```

```
dt << New Column( "cc", Character, width( 12 ) );

dt << Add Rows( {xx = 12, cc = "シカゴ"} ); // 1つのリスト
dt << Add Rows( {xx = 13, cc = "ニューヨーク"}, {xx = 14, cc = "ニューヨーク"} );
// 複数のリスト
dt << Add Rows(
   {{xx = 15, cc = "サンフランシスコ"}, {xx = Sqrt( 256 ), cc = "オークランド"}}
); // リストのリスト

a = {xx = 20, cc = "マイアミ"};
dt << Add Rows( a ); // 1つのリストとして評価する

b={{xx = 17, cc = "サンアントニオ"},{xx = 18, cc = "ヒューストン"}, {xx = 19, cc =
   "ダラス"}};
dt << Add Rows( b ); // リストのリストとして評価する
```

メッセージで指定できる行の詳細については、「行の属性と演算子」（348ページ）を参照してください。

行の削除

行を削除するには、Delete Rowsメッセージを送って、削除する行（複数可）を指定します。複数の行を削除するには、行番号（*rownum*）引数でリストまたは行列を指定するか、Delete RowsコマンドをForなどの他のコマンドと組み合わせて使います。行番号（*rownum*）引数として指定できるのは、番号、番号のリスト、番号の範囲、行列、またはこれらのどれかを導き出す式です。引数が指定されていない場合、Delete Rowsは選択されている行の数を戻し、これらの行を削除します。引数が指定されていないか、または行が選択されていない場合、Delete Rowsは選択されている行の数、つまり0を戻すだけです。

```
dt << Delete Rows( 10 ); // 行10を削除する
dt << Delete Rows( {11, 12, 13} ); // 行11～13を削除する
myList = {11, 12, 13};
dt << Delete Rows( mylist ); // 行11～13を削除する
dt << Delete Rows( 1 :: 20 ); // 最初の20行を削除する
dt << Delete Rows( [1 2 3] );  // 最初の3行を削除する
```

たとえば次のスクリプトは、「Big Class.jmp」を開き、10行目を削除します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
Selected( Row State( 10 ) ) = 1; // 行10を選択する
dt << Delete Rows;  // 行10を削除する
```

同じ行を複数回リストしてもかまいません。リストする順序も自由です。

以下に、任意のサイズのデータテーブルから最後の*x*行を削除する一般的な方法を示します。次の例は、データテーブルの最後の5行を削除します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
x = 5;
n = N Row( dt );
```

```
For( i = n, i > n - x, i--,
   dt << Delete Rows( i )
);
```

NRowは、テーブル内の行をカウントします。詳細は、「各行に対してスクリプトを反復する」（346ページ）を参照してください。

行の選択

Select All Rowsは、データテーブル内のすべての行を選択（強調表示）します。

```
dt << Select All Rows;
```

すべての行が選択されている場合、Invert Row Selectionを使用してそれらの選択を解除できます。このコマンドは、各行の選択の状態を逆にするので、選択されている行は選択が解除され、選択されていない行は選択されます。

```
dt << Invert Row Selection;
```

メモ: 現在の選択内容によって結果が異なるInvert Row Selectionを除き、選択メッセージを新しく使用した場合、新たに選択が行われます。すでに選択している行があり、選択された行に新しいメッセージを送ると、それらの行は、まず選択が解除されます。

特定の行を選ぶには、Go To Rowを使います。

```
dt << Go To Row( 9 );
```

データテーブルの特定の行を行番号で選択するには、Select Rowsコマンドを使います。コマンドの引数は行番号のリストです。たとえば、データテーブルの行1、3、5、および7を選択するには、次のように指定します。

```
dt << Select Rows( {1, 3, 5, 7} );
```

行の範囲を選択するには、次のメッセージのいずれかを指定します。

```
dt << Select Rows( Index( 7, 10 ) );
dt << Select Where( Any( Row() == Index( 7, 10 ) ) );
```

どちらの例も、現在のデータテーブルの中で7行目から10行目までを選択します。

データ値に基づいて行を選ぶには、Select Whereを使い、引数として条件式を指定します。

ヒント: Select Whereメッセージの中で使用できる関数と演算子については、「JSLの構成要素」章の「演算子」（85ページ）を参照してください。

たとえば、「Big Class.jmp」データテーブルのうち、生徒の年齢が14歳以上である行を選択するには：

```
dt = Open( "$SAMPLE_DATA\Big Class.jmp" );
dt << Select Where( :年齢 > 13 );
```

生徒の年齢が14歳未満である行を選択するには:

```
dt = Open( "$SAMPLE_DATA\Big Class.jmp" );
col = Column( dt, 2 );
dt << Select Where( col[] < 14 );
```

次の例は、生徒の年齢が15歳未満で性別がF（女性）である行を選択します。

```
dt = Open( "$SAMPLE_DATA\Big Class.jmp" );
dt << Select Where( :年齢 < 15 & :性別 == "F" );
```

前に選択した行を解除しないまま、もう1つの行を選択するには、<< Select Whereを<< Select Whereおよび Current Selection("extend")引数と組み合わせます。これは、ORステートメントの代わりになります。

```
dt = Open( "$SAMPLE_DATA\Big Class.jmp" );
dt << Select Where( :年齢 == 14 );
dt << Select Where( :性別 == "F", Current Selection( "extend" ) );
```

現在除外されている行、表示されていない行、またはラベル付きの行を選択するには:

```
dt << Select Excluded;
dt << Select Hidden;
dt << Select Labeled;
```

*除外されていない*行、表示されている行、またはラベルがない行を選ぶには、選択メッセージと選択逆転メッセージの両方を同じステートメントに入れるか、またはこの2つのメッセージを続けて送ります。

```
dt << Select Hidden << Invert Row Selection;
dt << Select Hidden;
dt << Invert Row Selection;
```

特定のセルを参照するには、そのセルの行番号に添え字を割り当てます。次の例では、「**体重(ポンド)**」列に添え字[1]が使われ、計算式によって「**体重(ポンド)**」列の最初の値に対する各「**身長(インチ)**」の比率が算出されます。

```
dt = Open( "$SAMPLE_DATA\Big Class.jmp" );
New Column( "比率", Formula( Name("身長(インチ)") / Name("体重(ポンド)")[1] ) );
```

無作為に選択した行を取得するには、次の構文を使用します。

```
dt << Select Randomly( number )
dt << Select Randomly( probability )
```

条件付き確率を使って、必要な度数を選択しています。

JSLには、行メニューコマンドSelect Matching Cellsもあります。

```
dt << Select Matching Cells;
// 現在のデータテーブルの中で一致するセルを選択する
dt << Select All Matching Cells;
// 開いているすべてのデータテーブルの中で一致するセルを選択する
```

複雑な選択を行う場合や、選択を行の属性データとして永続的に格納する方法については、「行の属性と演算子」（348ページ）を参照してください。

Where使用時の列参照の解決

列参照を使用してWhereステートメント内の列名を参照する際、正しいデータテーブルに解決されるように列参照を評価する必要があります。たとえば、次のスクリプトでは、X(Xcol)およびY(Ycol)列参照に対するパラメータがdt内のデータテーブルにリンクされています。しかし、プラットフォームの実行はWhereサブセットデータテーブルに関連付けられています。そのため、このスクリプトではエラーが発生します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
Ycol = Column( dt, "体重（ポンド）" ); // 体重（ポンド）への列参照
Xcol = Column( dt, "身長（インチ）" ); // 身長（インチ）への列参照
dt << Bivariate( Y( Ycol ), X( Xcol ), Fit Line(), Where( :性別 == "F" ) );
```

列名を正しいデータテーブルに評価するには、次のようにEval()式を使用するか、列名を直接参照します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Bivariate(
   Y( Eval( Ycol ) ), // またはY( :Name("体重（ポンド）") )
   X( Eval( Xcol ) ), // またはX( :Name("身長（インチ）") )
   Fit Line(),
   Where( :性別 == "F" )
);
```

行の検索

Get Rowsは、指定の条件に一致する行を行列の形で戻します。次の例は、年齢が16歳以上の行を選択します。

```
dt = Open( "$SAMPLE_DATA\Big Class.jmp" );
dt << Get Rows Where( :年齢 >= 16 );
  [35, 36, 37, 38, 39, 40]
```

Get Selected Rowsは、現在選択されている行を行列の形で戻します。次の列は、1、3、5、7行目を選択し、行番号を行列の形で戻します。

```
dt << Select Rows( {1, 3, 5, 7} );
dt << Get Selected Rows;
  [1, 3, 5, 7]
```

選択されている行の検索

Next SelectedとPrevious Selectedは、画面の外にある次の選択行を表示するために、データテーブルウィンドウを上下にスクロールします。テーブルは循環するので、Next Selectedでは、選択行の最後に来ると、最初の選択行に移動します。Previous Selectedではそれとは*逆に*先頭の選択行から最後の選択行に移動します。

```
dt << Next Selected;
dt << Previous Selected;
```

選択されている行のクリア

選択を取り消して、選択されている行がない状態にするには、次のようにClear Selectを使います。

```
dt << Clear Select;
```

行の移動

次のコマンドは、現在選択されている行を指定した移動先に移動します。

```
dt << Move Rows( At Start );
dt << Move Rows( At End );
dt << Move Rows( After( rowNumber ) );
```

行に色とマーカーを割り当てる

ColorsメッセージとMarkersメッセージを使うと、行に使われる色とマーカーを割り当てたり変更したりできます。通常、これらの設定は、データテーブルから生成されるグラフに影響します。どちらのメッセージも、どの色またはマーカーを使うかを指定する数値の引数をとります。色とマーカーの番号については、「色とマーカー」（358ページ）を参照してください。

```
dt << Colors( 3 );  // 選択した行に赤色のマーカーを設定する
dt << Markers( 2 ); // 選択した行にXマーカーを設定する
```

他の行メッセージと同様、行を選択するメッセージもその他のメッセージと一緒に使うことができます。

```
dt = Open( "$SAMPLE_DATA\Big Class.jmp" );
dt << Select Where( :年齢 == 13 ); // 最年少の被験者を選択する
    << Colors( 8 ) << Markers( 8 ); // そして、それらに紫色の円のマーカーを設定する
```

Color by ColumnとMarker by Columnは、指定された列の値に基づいて、それぞれ色とマーカーを設定します。

```
dt = Open( "$SAMPLE_DATA\Big Class.jmp" );
dt << Color by Column( :年齢 );
dt << Marker by Column( :年齢 );
```

その他の名前付き引数は次のとおりです。

- Continuous Scale（Color by Columnのみ） 指定の列の値に従って、グラデーションになった色を割り当てます。
- Reverse Scale 使用中の色を反転させます。
- Make Window with Legend 凡例のウィンドウを別に作成します。
- Excluded Rows 除外されている列に行の属性を適用します。
- Marker Theme マーカーの種類を指定します。
- Color Theme カラーテーマを指定します。

セルの色

データグリッドの個々のセルを色分けできます。たとえば、次の行は行の属性の色を使ってセルを色分けします。

```
dt << Color Rows by Row State;
```

カテゴリの列、連続尺度の列のいずれもカラーテーマを指定することができます。

```
dt = Open( "$SAMPLE_DATA\Big Class.jmp" );
:Name("身長(インチ)") << Set Property(
   "Color Gradient",
   {"White to Blue", Range( 40, 80 )}
);
:Name("身長(インチ)") << Color Cell By Value( 1 ); // セルの色分けを有効にする

:年齢 << Set Property(
   "Value Colors", // 色の値を割り当てる
   {12 = Red, 13 = Yellow, 14 = Green, 15 = Blue, 16 = Magenta, 17 =
   Gray}
);

:年齢 << Color Cell By Value( 0 ); // セルの色分けを無効にする
```

特定のセルに色をつけることもできます。次の例は、「名前」列の1、5、8行目を赤に設定します。

```
:名前 << Color Cells( red, {1, 5, 8} );
```

特定のセルの色を削除するには、色を黒に設定します。次の例は、「名前」列の1行目の色を削除します。

```
:名前 << Color Cells( black, {1} );
```

セルに、値に応じて色をつけることができます。次の例は、「**身長(インチ)**」列のセルに色をつけます。値が60より大きいセルは青、値が60以下のセルは紫に設定します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
For( i = 1, i <= N Row( dt ), i++,
   If( Column( dt, "身長(インチ)" )[i] > 60,
      Column( dt, "身長(インチ)" ) << Color Cells( 5, {i} ),
      Column( dt, "身長(インチ)" ) << Color Cells( 8, {i} )
   )
);
```

メモ: Color Cellsの最初の引数は、色の値を表します。2番目の引数には、行番号を含めます。

行の非表示、除外、ラベル

メモ: 行の属性演算子を使って行を非表示にしたり、除外したり、ラベルをつけたりする方法について詳しくは、「行の属性と演算子」（348ページ）を参照してください。

行を非表示にしたり、除外したり、ラベルをつけたりするには、Hide、Exclude、Label メッセージを使用します。これらのメッセージは切り替え式なので、1度送るとオンになり、もう1度送るとオフになります。ブール値の引数を使って明示的にオンまたはオフにすることもできます。

たとえば、「Big Class」内の「年齢」が13を超えるすべての行に非表示の属性を設定するには、次のようにします。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Select Where( :年齢 > 13 );
dt << Hide( 1 );
```

同じオブジェクトへのメッセージは、単一のステートメントにまとめることができるため、上記のスクリプトは次のように簡潔にすることができます。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Select Where( :年齢 > 13 ) << Hide( 1 );
```

テーブルの行に対する反復

JSLには、組み込みの反復用プログラミング演算子の他に、データテーブルの行、グループ、または行の条件付き選択を使って反復するための演算子もあります。

通常、式はデータテーブルの現在の行に対してだけ実行されます。例外は、計算式列内の式、Summarize演算子と統計量を事前計算する演算子、および分析プラットフォームがデータテーブルの列を使用する場合です。

現在の行の設定

メモ: スクリプトの現在行は、データテーブルで選択（強調表示）されている行や、データテーブルウィンドウでの現在のカーソル位置とは**関係ありません**。スクリプト操作の対象となる行の番号です。デフォルトではゼロ（行なし）になっています。

スクリプトのために現在の行を設定するには、For Each RowまたはRow() = Xを使用します。

```
Row() = 3; ...
For Each Row( ... );
```

For Each Rowは、現在のデータテーブルの各行に対し、スクリプトを1回実行します。Row()=1はスクリプトが実行されている間だけ有効で、実行が終了すると、Row()はデフォルトの値であるゼロに戻ります。そのため、一度にスクリプトすべてを実行したときと、一度に数行ずつスクリプトを実行したときとでは、異なる結果になる場合があることに注意してください。

この章では、現在の行を明示的に指定していない例の場合、現在の行を決定するコンテキストの中だけで実行されるものとします。詳細は、「現在の行とは」（346ページ）を参照してください。

現在の行とは

デフォルトでは、現在の行の番号は0です。テーブル内の先頭の行が行1なので、行0は原則として行がないことを表します。つまり、**デフォルトでは、操作はどの行にも実行されません**。現在の行を設定するか、または行のセットを指定しない限り、データがないために欠測値となります。たとえば、列名は、現在の行の該当する列の値を戻します。あいまいさを避けるために（強制的に名前を列名として解釈させるために）、接頭演算子「:」を使います。

```
:性別; // "" を戻す
:年齢; // . を戻す
```

接頭演算子を使うことで、実際には1行だけに基づいているのに、データテーブル全体に当てはまるように見える結果を取得しないようにするためです。また、大部分の環境で、値をあいまいな名前に割り当てた場合に、不用意にデータ値が上書きされないようにします（このような事態を完全に防ぐには、接頭または間に入れる演算子「:」を使ってデータ列を明確に参照し、接頭演算子「::」を使ってグローバルスクリプト変数を明確に参照します。詳細は、「プログラミング手法」章の「高度な適用範囲指定と名前空間」（230ページ）を参照してください。

Row() 演算子を使って、現在の行の番号を取得または設定できます。Row() は、JSLの**L-value**式の一例です。この演算子は、代入演算子（=、+=など）の前に置いて値を設定しない限り、デフォルトの値を戻します。

```
Row(); // 現在の行の番号を戻す（デフォルトでは 0）
x = Row(); // 現在の行の番号を x に格納する
Row() = 7; // 7 番目の行を現在の行にする
Row() = 7; :年齢; // 7 番目の行を現在の行にし、12 を戻す
```

現在の行の設定は、ユーザが選択して実行するスクリプト部分が終わるまで有効です。スクリプトの実行が終わると、現在の行の設定はデフォルト（行0または行なし）にリセットされます。そのため、同じスクリプトでも、一度にすべて実行した場合と数行ずつ実行した場合では、異なる結果が出ることがあります。

行と列の数

演算子N Rows とN Cols は、データテーブル内の行と列の数をカウントします。

```
N Rows( dt ); // 行数
N Cols( dt ); // 列数
```

N Rows とN Cols は、行列内の行の数もカウントします。NRow とNCol は同義語です。詳細については、「データ構造」章の「問い合わせ関数」（173ページ）を参照してください。

各行に対してスクリプトを反復する

現在のデータテーブルの各行にスクリプトを繰り返すには、スクリプトに For Each Row を入れます。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
For Each Row( If( :年齢 > 15, Show( :年齢 ) ) );
```

開いているデータテーブルを指定するには、第1引数としてデータテーブル参照を含めます。

```
dt1 = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt2 = Open( "$SAMPLE_DATA/San Francisco Crime.jmp" );
For Each Row( dt1, If( :年齢 > 15, Show( :年齢 ) ) );
```

For Each Rowは、データテーブルに新しい計算式列を作成するのではなく、行の属性を設定するために使います。次の2つのスクリプトは似ていますが、最初のスクリプトでは行の属性列を作成し、For Each Rowのスクリプトでは列を作成せずに行の属性を設定します。

```
New Column( "行の属性", Row State, Formula( Color State( :年齢 - 9 ) ) );
For Each Row( Color of( Row State() ) = :年齢 - 9 );
```

指定した条件に一致する各行にスクリプトを繰り返すには、次のようにFor Each RowとIfを組み合わせます。

```
For Each Row( Marker of( Row State() ) = If( :性別 == "F", 2, 6 ) );
```

For Each Rowループの実行を制御するには、BreakとContinueを使用します。詳細は、「JSLの構成要素」章の「BreakおよびContinue」（102ページ）を参照してください。

行の値を戻す

DifとLagは、統計量計算、特に時系列データや累積データを処理するときに有効です。

- Lagは、現在の行から*n*行前の列の値を戻します。
- Difは、現在の行の列の値と*n*行前のその列の値との差を戻します。

以下の式は同じ結果になります。

```
dt  <<  New Column( "身長の差" );
For Each Row( :身長の差 = :Name("身長(インチ)") - Lag( :Name("身長(インチ)"), 1 )
  );
For Each Row( :身長の差 = Dif( :Name("身長(インチ)"), 1 ) );
```

シーケンスデータの追加

Sequence()は、計算式エディタのSequence関数に相当し、テータテーブル列のセルを埋めるのに使用します。4つの引数のうち、後の2つはオプションです。

```
Sequence( from, to, stepsize, repeat );
```

必須である開始値（from）と終了値（to）は、セルに挿入する値の範囲を指定します。from=4、to=8とした場合、セルは4、5、6、7、8、4、...という値で埋められます。

ステップサイズ（stepsize）はオプションです。指定しない場合のデフォルト値は1です。ステップサイズは範囲内の値の増分です。上記のfromとtoの値を使い、stepsize = 2とした場合、セルは4、6、8、4、6、...という値で埋められます。

繰り返し（repeat）はオプションです。指定しない場合のデフォルト値は1です。繰り返しは、次の値にインクリメントするまでに同じ値を繰り返す回数を指定します。上記のfrom、toおよびstepsizeの値を使い、repeat=3とした場合、セルは4、4、4、6、6、6、8、8、8、4、...という値で埋められます。繰り返し値（repeat）を指定した場合は、ステップサイズ（stepsize）も指定しなければなりません。

列のすべての行が埋まるまで数列（シーケンス）が繰り返されます。

```
dt = New Table( "Sequence Example" ); // 新しいデータテーブルを作成する
dt << New Column( "5まで数える" ); // 2列を追加する
dt << New Column( "4間隔で17まで数える" );
dt << Add Rows( 50 ); // 50行を追加する

For Each Row (
   Column( 1 )[ ] = Sequence( 1, 5 );
   Column( 2 )[ ] = Sequence( 1, 17, 4, 2 );
);
/* 最初の列を1、2、3、4、5、...の数列で埋める
2列目を1、1、5、5、9、9、13、13、17, 17、...の数列で埋める */
```

Sequence()は計算式関数なので、列の計算式にSequence()を設定して列を埋めることもできます。次の例では「Sequence関数の計算式」という新しい列を作成し、それに計算式を追加しています。この計算式によって作成される数列は、25から29までの値をそれぞれ回繰り返しながら、1ずつ増加します（25、25、26、26、27、27、28、28、29、29、25、...）。

```
dt = New Table( "Sequence関数の計算式の例" );
dt << New Column( "Sequence関数の計算式", Formula( Sequence( 25, 29, 1, 2 ) ) );
```

Sequence()の結果例をさらにいくつか紹介しましょう。

- Sequence(1,5)の場合、1,2,3,4,5,1,2,3,4,5,1,...となります。
- Sequence(1,5,1,2)の場合、1,1,2,2,3,3,4,4,5,5,1,1,...となります。
- Sequence(10,50,10)の場合、10,20,30,40,50,10,...となります。
- 10*Sequence(1,5,1)の場合、10,20,30,40,50,10,...となります。
- Sequence(1,6,2)の場合、1,3,5,1,3,5,...となります。リミットはありません。

メモ：通し番号の値で行列を作成したい場合は、Sequence関数ではなく、Index関数を使用します。

行の属性と演算子

データテーブルには、さまざまな属性を格納する「行の属性」という特殊なデータ要素があります。行の属性は、次のような属性を格納します。

- 行が選択されているか、除外されているか、非表示になっているか、ラベルがついているか
- マーカーの種類、色、色合い、グラフ上での色相

JSLでは、行の属性演算子を使って行の属性を操作することができます。

行の属性について

行の属性に応じて、JMPによる処理の内容が変わってきます。表9.3で各属性について説明します。複数の行の属性を組み合わせて使用することができます。

表9.3 行の属性

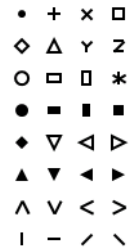
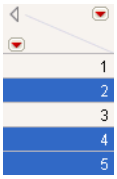
行の属性	結果への影響
除外する 	行が除外されている場合、JMPはそれらの行を統計分析（テキストレポートとチャート）の計算から除く。結果は、データが入力されていないときと同じになります。ただし、その行は**プロット**内に点として表示されます。（プロットから点を除くには「表示しない」を設定します。すべての結果から行を除外するには、「除外する」と「表示しない」の両方を指定します。）
表示しない 	行が「表示しない」になっている場合、JMPはそれらの行をプロットに表示しない。ただし、行はまだテキストレポートおよび**チャート**内に含まれています。（レポートおよびチャートから行を除くには「除外する」を指定します。すべての結果から行を除くには、「除外する」と「表示しない」の両方を指定します。）
ラベルあり 	行にラベルがついている場合、JMPは、散布図内の点に、行番号のラベルまたは指定されたラベル列の値を配置する。
色 	行に色が指定されている場合、JMPはそれらの色を使って、散布図内の点をわかりやすく表示する。
マーカー 	行にマーカーが指定されている場合、JMPはそれらのマーカーを使って、散布図と回転プロット内の点をわかりやすく表示する。

表9.3 行の属性（続き）

行の属性	結果への影響
選択 	行が選択されている場合、JMPはプロットおよびチャート内の対応する点や棒を強調表示する。

行の属性の演算子について

行の属性の演算子に引数として数値（または数値を導き出す式）を指定すると、演算子はその数値を可能な値へのインデックスと解釈します。

表9.4は、行の属性演算子を比較した表です。この表を見ると、行の属性から数値に、または数値から行の属性に変換する演算子がわかります。表には、各演算子で使用できる数値も記載されています。

表9.4 行の属性の演算子

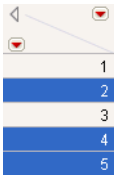
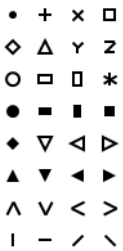
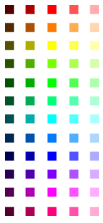
数	数値から行の属性への変換 →	行の属性	行の属性から数値への変換 →	数
1または0	`Excluded State(n)`	除外する 	`Excluded(rowstate)`	1または0
1または0	`Hidden State(n)`	表示しない 	`Hidden(rowstate)`	1または0
1または0	`Labeled State(n)`	ラベルあり 	`Labeled(rowstate)`	1または0
1または0	`Selected State(n)`	選択 	`Selected(rowstate)`	1または0

表9.4 行の属性の演算子（続き）

数	数値から行の属性への変換 →	行の属性	行の属性から数値への変換 →	数
0 ～ 31	`Marker State(n)`	マーカー 	`Marker Of(rowstate)`	0 ～ 31
0 ～ 84 （0-15は基本色、16-31は暗い、32-47は明るい、48-63は非常に暗い、64-79は非常に明るい、80-84は灰色）	`Color State(n)`	色 	`Color Of(rowstate)`	0 ～ 84 （0-15は基本色、16-31は暗い、32-47は明るい、48-63は非常に暗い、64-79は非常に明るい、80-84は灰色）
0 ～ 11 （虹の色の順）	`Hue State(n)`	色相 		
-2 ～ 2 （暗い～明るい）	`Shade State(n)`	色合い 		

行の属性の割り当て

JSLを使って行の属性を割り当てるには、Select Whereを使って対象となる行を指定してから、それらの行に割り当てる属性を指定します。次の例は、性別が「F」の行にマーカーの種類「5」を割り当てます。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Select Where( :性別 == "F" ) << Markers( 5 );
```

計算式を使って行の属性を割り当てることもできます。それをデータテーブルで行う方法を紹介します。

1. 行の属性列を作成します。
2. 列に、性別が「F」である行にマーカーを割り当てる計算式を追加します。
3. 「列」パネルで、行の属性列の隣にある星を右クリックし、[**行の属性へコピー**]を選択します。

この手順をJSLで表現すると、次のようになります。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << New Column( "行の属性",
   Row State,
   Set Formula( If( :性別 == "F", Marker State( 4 ) ) )
);
Column( "行の属性" ) << Copy To Row States();
```

行の属性情報を格納

行の属性の列は、行の属性を情報として**格納**するだけで、属性を実際に有効にするわけではありません。行の属性の列を有効にするには、続いてFor Each Rowを使用します。行の属性列の詳細については、『JMPの使用法』の「列情報ウィンドウ」章を参照してください。

次の例は、行の属性の列を作成し、それらの属性を有効にします。

1. 次のスクリプトを実行、新しいデータテーブルの作成を開始します。

   ```
   dt = New Table( "行の属性テスト" );
   ```

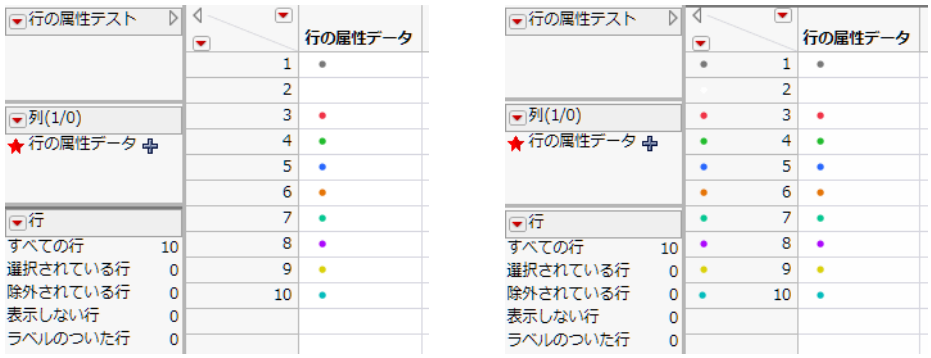
2. 次のスクリプトを実行して列に行の属性を追加し、10行を追加します。

   ```
   dt << New Column( "行の属性データ", Row State, Set Formula( Color State( Row() ) )
     );
   dt << Add Rows( 10 );
   ```

3. 次のスクリプトを実行し、行の属性を有効にします。

   ```
   For Each Row( Row State() = :行の属性データ );
   ```

図9.6 行の属性のないテーブル（左）と行の属性のあるテーブル（右）



このスクリプトは、現在有効な行の属性を、行の属性の列に設定された組み合わせで置き換えます。他の属性を変更したり取り消したりしないで、行の属性のうち選択したものだけを変える方法については、「1つの特性だけを設定し、他の特性をキャンセルする」（356ページ）を参照してください。

行の属性の設定または取得

Row State演算子を使うと、JSLから直接、行の属性を設定または取得できます。行*n*の属性は、Row State(n)で設定または取得できます。引数を指定しなければ、現在の行を設定または取得します。すべての行を処理するには、For Each Rowループを使うか、計算式列で処理します。

行の属性を設定するには、行の属性に式を割り当て、式の左辺（L-value）に置きます。

```
Row State( 1 ) = Color State( 3 ); // 行1を赤にする
Row() = 8; Row State() = Color State( 3 ); // 8番目の行を赤にする
For Each Row( Row State() = Color State( 3 ) ); // すべての行を赤にする
```

データテーブルを指定するには、第1引数としてデータテーブル参照を含めます。

```
dt1 = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt2 = Open( "$SAMPLE_DATA/San Francisco Crime.jmp" );
Row State( dt1, 1 ) = Color State( 3 ); // 行1を赤にする;
```

行の属性を取得するには、行の属性式を割り当て式の右辺に置きます。

```
x = Row State( 5 );  // 行5の行の属性をxに保存する
x = Row State();   // 現在の行の行の属性
```

行の属性の設定に関するメモ

Row State() のすべての属性を設定するのか、Color Of(Row State()) のように特定の1つの属性だけを設定するのかという点に注意してください。動作を確かめるには、まず、すべての行に色とマーカーの属性を設定します。

```
dt1 = Open( "$SAMPLE_DATA/Big Class.jmp" );
For Each Row(
   Row State() = Combine States( Color State( Row() ), Marker State( Row() ) ) );
```

ここで、行の属性のうち1つの属性だけを変更してみましょう。

```
Color Of( Row State( 1 ) ) = 3; // マーカーを変更しないで行１を赤にする
```

行の属性のすべての要素をクリアして1つの属性を設定するには、次のようにします。

```
Row State( 1 ) = Color State( 5 ); // 行１を青にし、そのマーカーを削除する
```

現在の行の属性をすべて行属性列にコピーするには、次のようにします。

```
New Column( "rscol", Set Formula( Row State() );
For Each Row( rscol = Row State() );
```

計算式に合わせて"rscol"列のタイプが行の属性に変更されました。

現在の行の属性をすべてではなく一部（複数）だけ行属性列にコピーするには、次のようなスクリプトを使います（必要のない属性は、コメントにするか削除してください）。

```
New Column( "rscol2",
   Set Formula(
      Combine States(
         Color State( Color Of() ),
         Excluded State( Excluded() ),
         Hidden State( Hidden() ),
         Labeled State( Labeled() ),
         Marker State( Marker Of() ),
         Selected State( Selected() )
      )
   )
);
```

行の属性のうち１つの特性を設定するには：

```
Color Of( Row State( i ) ) = 3; // 行ｉの色を赤に変更
Selected( Row State( i ) ) = 1; // 行ｉの行の属性を選択し、１に設定
```

上の例のように、演算子の中には、数値を属性に変換するものと、属性を数値に変換するものがあります。このどちらであるかを判別するためのヒントを示します。

数値を属性に変換する演算子には、「State」という語が含まれています。 数値引数をとり、属性を戻すかまたは属性割り当てを受け取る演算子はすべて、次のように、名前に「State」という語が含まれています。Row State、As Row State、Color State、Combine States、Excluded State、Hidden State、Hue State、Labeled State、Marker State、Selected State、Shade State。

属性を数値に変換する演算子は、１語か、または「Of」という語が含まれています。 行の属性引数をとる（引数が指定されていない場合は、Row State()が引数とみなされる）演算子と、数値を戻すか数値に設定されている演算子は、次のように、1語であるか、または2番目の語が「Of」です。Color Of、Excluded、Hidden、Labeled、Marker Of、Selected。

表9.4（350ページ）は、これらの演算子の違いを説明した便利な表です。

次のコマンドは、インタラクティブなコマンドと同じものです。

```
Copy From Row States
Add From Row States
Copy To Row States
Add To Row States
```

行の属性を取得する例

たとえば、行の属性を含むデータテーブルを作成してみましょう。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << New Column( "行の属性データ", Row State, Set Formula( Color State( Row() ) )
  );
dt << Add Rows( 10 );
For Each Row( Row State() = :行の属性データ );
```

行の属性を取得するには、次のようなスクリプトを実行します。

```
Row State( 1 ); // 行1の行の属性を戻す
  Color State(1)
Row() = 8; Row State();  // 現在の行（8番目の行）の行の属性を戻す
  Color State(8)
For Each Row( Print( Row State()) ); // 各行の行の属性を戻す
  Color State(1)
  Color State(2)
  Color State(3)
  ...
```

行の属性を取得し、グローバル変数に格納するには、割り当て式の右辺に Row State() を置きます。次の行を上記のスクリプトに追加します。

```
::x = Row State( 1 ); // 行1の行の属性をグローバル変数xに保存する
Row() = 8; ::x = Row State();  // 8番目の行の行の属性をxに保存する
Show( x ); // x = Color State( 8 )を戻す
dt << New Column( "rsco|", Row State) ;
For Each Row( :rscol = Row State() );
// 行の属性を、「rscol」（行の属性の列）に保存する
```

行の属性のうち1つの特性だけを取得するには、行の属性式を割り当て式の右辺に置き、左辺に指定できる演算子のどれかを使います。

```
x = Selected( Row State() ); // 現在の行が選択されているかどうかのインデックス
```

複数の属性を一度に取得／設定する

Combine States の中で属性の設定を組み合わせることで、多数の属性を一度に取得または設定できます。また、属性が組み合わされた後も、各属性ごとに取得または設定することが可能です。次の例は、男子に緑のYマーカーを設定し、プロットではそれらを非表示にします。また、女子にはXマーカーを設定し、それらをプロットに表示します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
For Each Row(
```

```
If( 性別 == "M",
   /* then */ Row State() = Combine States(
      Color State( 4 ), Marker State( 6 ), Hidden State( 1 ) ),
   /* else */ Row State() = Combine States(
      Color State( 3 ), Marker State( 2 ), Hidden State( 0 ) ) ) );
```

1つの行、たとえば6行目の行の属性を取得します。

```
Row State( 6 );
  Combine States(Hidden State(1), Color State(4), Marker State(6))
```

JMPは、Combine Stateで属性の組み合わせを戻すことに注意してください。これは、行の属性データは、色などの1つの属性だけではなく、除外、非表示、ラベル付け、選択、マーカー、色、色相、濃淡など、設定されたすべての属性をまとめたものだからです。このような特性のリストは、**行の属性の組み合わせ**と呼ばれます。

複数の行の属性が有効になっている場合もあるので、行の属性列は複数の特性を値として持つことができるようになっています。

1つの特性だけを設定し、他の特性をキャンセルする

行の属性のすべてを取得または設定するための全体的なRow State演算子の他に、一度に1つの特性を優先的に取得または設定する演算子があります。つまり、1つの行に1つの特性だけを与えるため、他の特性をすべて取り消します。1つの特性を設定し、その他をキャンセルする演算子は、Color State、Combine States、Excluded State、Hidden State、Hue State、Labeled State、Marker State、Selected State、Shade Stateです。

たとえば、行4を表示しないという属性だけを設定するには、次のようにします。

```
Row State( 4 ) = Hidden State( 1 );
```

一度に1つの特性を設定または取得する

行の属性とは、1つの特性ではなく多数の特性が組み合わされたものです。一度に1つだけ処理するには、Row Stateと左辺に指定できる演算子のどれかを、取得か設定かによって等号のどちらかの側で使います。左辺に指定できる演算子として、Color Of、Excluded、Hidden、Labeled、Marker Of、Selectedがあります。

この例は、他の特性には影響を与えずに、行4を非表示にします。

```
Hidden( Row State( 4 ) ) = 1
```

この例は、他の特性には影響を与えずに、行3の色を格納します。

```
::color = Color Of( Row State( 3 ) );
```

行の属性の変化を特定する

（データテーブルオブジェクトに送られた）MakeRowStateHandlerメッセージは、行の属性が変更されるとコールバックを取得します。例:

```
f = Function( {X}, Show( x ) );
obj = dt << Make Row State Handler( f );
```

これで、棒グラフなどで行をグループ選択すると、行の属性が変更されている行の番号がログに送られます。例:

```
x:[3, 4, 28, 40, 41]
```

複数行が強調表示されている場合は、選択がクリアされた行に対して1回、新しく選択された行に対して1回の計2回、ハンドラが呼び出されることがあります。

除外、非表示、ラベル、選択

この節では、オンまたはオフのブール値をとる状態について説明します。このような状態としては、行の除外、非表示、ラベル、選択があります。

- Excluded は、除外されているかどうか示す除外インデックスを取得または設定します。インデックスの値が1なら真、0なら偽です。
- Hiddenは、非表示は1、表示は0の非表示インデックスを取得または設定します。
- Labeledは、ラベルがついているなら1、ついていないなら0のラベルインデックスを取得または設定します。
- Selectedは、選択されている場合は1、選択されていない場合は0の選択インデックスを取得または設定します。

次の例は、これらの状態を示します。

```
Excluded( Row State() ); // 現在の行が除外されている場合は 1、除外されていない場合は 0 を戻す
Hidden();                // 現在の行が表示されない場合は 1、表示される場合は 0 を戻す
Labeled( Row State() ); // 現在の行にラベルがついている場合は 1、
  // ラベルがついていない場合は 0 を戻す
Selected();              // 現在の行が選択されている場合は 1、選択されていない場合は 0 を戻す

Excluded( Row State() ) = 1; // 現在の行を除外する
Hidden() = 0;                // 現在の行を表示する
Labeled( Row State() ) = 1;  // 現在の行にラベルをつける
Selected() = 0;              // 現在の行の選択を取り消す
```

これらの関数では、引数が指定されていない場合は、Row State()が引数であるとみなされます。

Excluded State、Hidden State、Labeled State、Selected Stateは、逆に、引数に基づいて、行の属性の状態を真または偽として取得または設定します。ゼロ以外の値は行の属性を真に、ゼロの値は偽に設定します。欠測値の場合は、行の属性に変更はありません。

```
Row State() = Excluded State( 1 ); // 現在の行の属性を「除外されている」に変更する
Row State() = Hidden State( 0 );   // 現在の行の属性を「再表示」に変更する
Row State() = Labeled State( 1 );  // 現在の行の属性を「ラベルがついている」に変更する
Row State() = Selected State( 0 );
// 現在の行の属性を「選択されていない」に変更する
```

上記の最初の2つの式は、行の属性のうち除外または表示を置き換えるだけで、他の既存の行属性の特性は失われません。通常、設定したいすべての特性のstateコマンドをCombine Statesの中に記述します。次の例は、3番目の行を非表示にし、緑の正方形のマーカーに変更します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
Row State( 3 ) = Combine States(
   Color State( 4 ),
   Marker State( 3 ),
   Hidden State( 1 ) );
```

また、行の属性に（累積特性として）追加できる値を持つ行の属性データ列内で -Stateコマンドを使うのも一般的な方法です。次の例は、各奇数行を除外します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << New Column( "myExcl",
   Row State,
   Set Formula( Excluded State( Modulo( Row(), 2 ) ) )
);
For Each Row( Row State() = Combine States( :myExcl, Row State() ) );
```

行の属性をクリア

Clear Row States()は、データテーブルから行の属性を削除します。次のスクリプトは、行の属性を各行に割り当てた後、その行の属性を削除します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
For Each Row( Row State() = Color State( Row() ) );
Wait( 3 ); // 結果がわかりやすいように一時停止する
dt << Clear Row States();
```

選択した行から行の属性をクリアするには、まず行を選択してから、選択した行の属性をクリアします。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
For Each Row( Row State() = Color State( Row() ) );
dt << Select Rows( [ 5 ] );
Wait( 3 ); // 結果がわかりやすいように一時停止する
dt << Clear Selected Row States();
```

色とマーカー

この節では、多数の選択肢がある状態について説明します。このような状態には、色、マーカー、色相、色合いがあります。

Color Of()は、色番号を戻すか、設定します。色番号とは、*row state*に対応するJMPカラーマップの色番号で、色が割り当てられていない場合は0となります。

```
Color Of( Row State() );    // 現在の行の色番号を戻す
Color Of() = 4;             // 現在の行を色 4 に設定する
```

同様に、Marker Ofは、マーカー番号を戻すか、設定します。マーカー番号は、アクティブなマーカーに対応するJMPマーカーマップのマーカー番号で、マーカーが割り当てられていない場合は0となります。

```
Marker Of();                          // 現在の行のマーカー番号を戻す
Marker Of( Row State() ) = 4; // 現在の行をマーカー4に設定する
```

Color Of()とMarker Of()はどちらも、すべての行の属性の式もしくは列、またはRow State()を引数として受け取り、引数が指定されていない場合は、引数Row State()を引数とみなします。

Color State()とMarker State()は、逆方向に動作することを除くと、Color Of()およびMarker Of()と同じです。-Of関数は、実際の属性をインデックスに変え、-State関数は、インデックスを属性に変えます。

```
Row State() = Color State( 4 );  // 現在の行を緑に変更する
Row State() = Marker State( 4 ); // 現在の行をひし形のマーカーに変更する
```

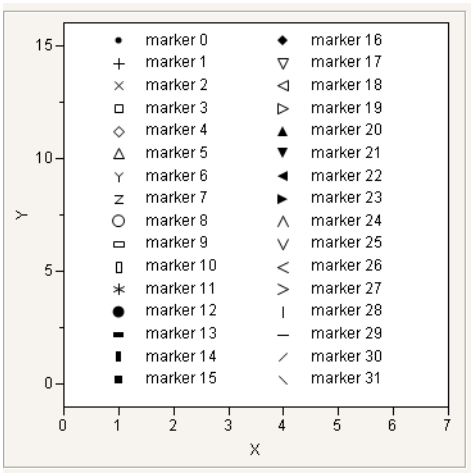
最後の2つのコマンドは、行の属性のうち色またはマーカーを置き換えるだけで、他の既存の特性は失われません。通常、設定したいすべての特性の-Stateコマンドを Combine States()の中に記述します。次の例は、3番目の行に緑の正方形のマーカーを配置し、その行を非表示にします。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
Row State( 3 ) = Combine States(
   Color State( 4 ),
   Marker State( 3 ),
   Hidden State( 1 ) );
```

次のスクリプトは、JMPの標準的なマーカー（0～31）を表示します。0～31の範囲外にあるインデックスの動作は定義されていません。

```
New Window( "マーカー",
   Graph Box(
      FrameSize( 300, 300 ),
      Y Scale( -1, 16 ),
      X Scale( 0, 7 ),
      For(
         i = 0;
         jj = 15;,
         i < 16;
         jj >= 0;,
         i++;
         jj--;, // 16行、2列
         Marker Size( 3 );
         Marker( i, {1, jj + .2} ); // マーカー0～15
         Marker( i + 16, {4, jj + .2} ); // マーカー16～31
         Text( {1.5, jj}, "marker ", i ); // マーカーラベル0～15
         Text( {4.5, jj}, "marker ", i + 16 ); // マーカーラベル16～31
      )
   )
);
```

図9.7 JMPマーカー



ヒント： このスクリプトについて詳しくは、「表示ツリー」章（413ページ）を参照してください。

色については、以下の点に留意してください。

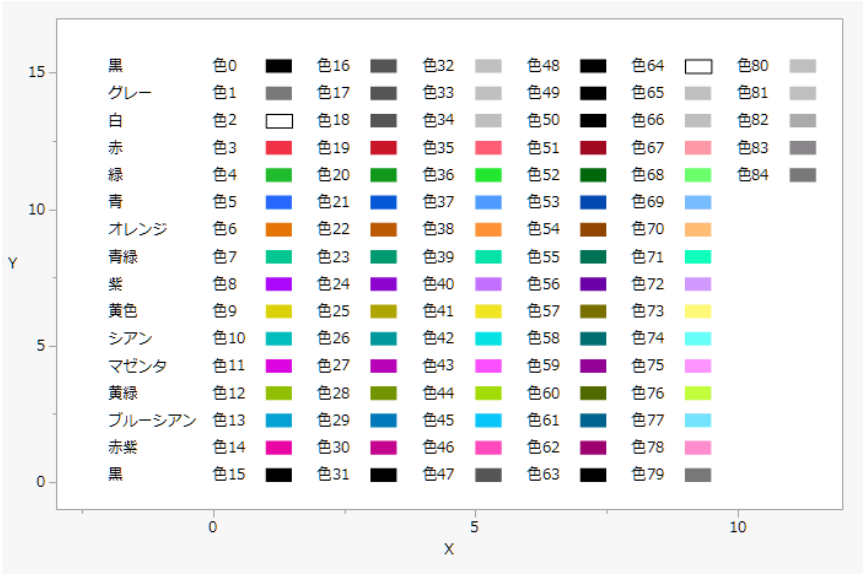
- JMPカラーには0～84の番号が付いています。
- 最初の16個は基本色です。下のスクリプトを参照してください。
- 17以降は、それぞれの基本色の色合いを暗め、または明るめにしたものです。
- 0～84の範囲外にあるインデックスの動作は定義されていません。
- JMPカラーの使用方法については、「スクリプトによるグラフ作成」章の「色を指定する」（549ページ）を参照してください。

次のスクリプトは、JMPの標準の色を表示します。

```
Text Color( 0 );
New Window( "色",
   Graph Box(
      FrameSize( 640, 400 ),
      Y Scale( -1, 17 ),
      X Scale( -3, 12 ),
      k = 0;
      For( jj = 1, jj <= 12, jj += 2,
         l = 15;
         For( i = 0, i <= 15 & k < 85, i++,
            thiscolor = Color To RGB( k );
            Fill Color( k );
            thisfill = 1;
            If( thiscolor == {1, 1, 1},
               Pen Color( 0 );
```

```
                thisfill = 0;
            ,
                Pen Color( k )
            );
            Rect( jj, l + .5, jj + .5, l, thisfill );
            Text( {jj - 1, l}, "色", k );
            k++;
            l--;
        );
    );
    jj = -2;
    color = {"黒", "グレー", "白", "赤", "緑", "青", "オレンジ", "青緑",
    "紫", "黄色", "シアン", "マゼンタ", "黄緑", "ブルーシアン", "赤紫", "黒"};
    For(
        i = 0;
        l = 15;, i <= 15 & l >= 0,
        i++;
        l--;,
        Text( {jj, l}, color[i + 1] )
    );
) );
```

図9.8 JMPカラー



RGB値を使うには、赤、緑、青の順でそれぞれの含有率をリストして、各色を指定する必要があります。たとえば、次のように指定すると、緑がかった青になります。

```
Pen Color( {.38,.84,.67} ); // 小鴨色
```

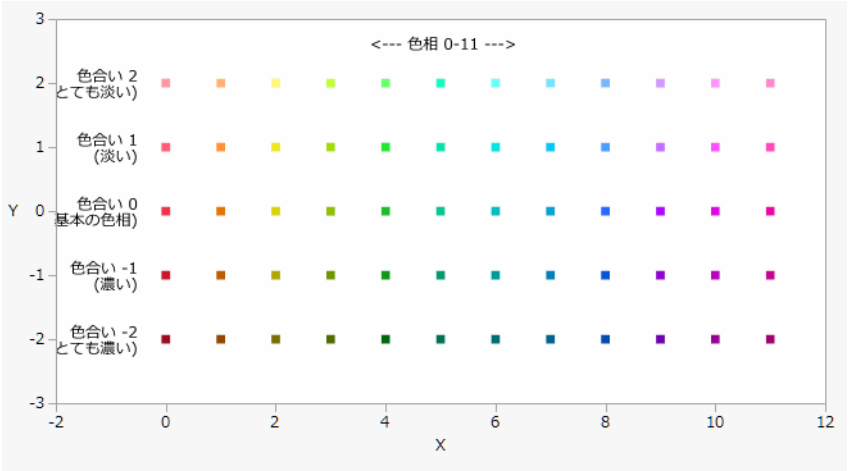
色相と色合いの例

色を選ぶ際、Color Stateを使う代わりにHue StateとShade Stateを一緒に使うこともできます。Hue Stateを使った場合は、黒、白、またはグレーの色合いは選べません。このような場合は、Shade Stateを単独で使うか、Color Stateを使う必要があります。

次のスクリプトは、色相および色合いの値と色との関係を表示します。

```
New Window( "色相と色合い",
   Graph Box(
      FrameSize( 600, 300 ),
      Y Scale( -3, 3 ),
      X Scale( -2, 12 ),
      k = 0;
      For( h = 0, h < 12, h++,
         For( s = -2, s < 3, s++,
            myMk = Combine States( Hue State( h ), Shade State( s ), Marker State(
   15 ) );
            Marker Size( 3 );
            Marker( myMk, {h, s} );
         )
      );
      Text( Center Justified, {5, 2.5}, " <--- 色相 0-11 ---> " );
      Text( 右揃え ,
         {-.5, -2}, "色合い -2", {-.5, -2.25}, "(とても濃い)",
         {-.5, -1}, "色合い -1", {-.5, -1.25}, "(濃い)",
         {-.5, 0}, "色合い 0", {-.5, -.25}, "(基本の色相)",
         {-.5, 1}, "色合い 1", {-.5, .75}, "(淡い)",
         {-.5, 2}, "色合い 2", {-.5, 1.75}, "(とても淡い)"
      );
   )
);
```

図9.9 色相と色合い



HueとShadeには、-Of演算子はありません。Color Ofは、Hue StateとShade State、またはそのどちらかで設定された色の行属性について、Color Stateと同じ番号を戻します。たとえば、行4と行5に同じ濃赤色のマーカーを設定します。

```
Row State( 4 ) = Combine States( Hue State( 0 ), Shade State( -1 ), Marker State(
   12 ) );
Row State( 5 ) = Combine States(
   Color State( Color Of( Row State( 4 ) ) ),
   Marker State( Marker Of( Row State( 4 ) ) )
);
```

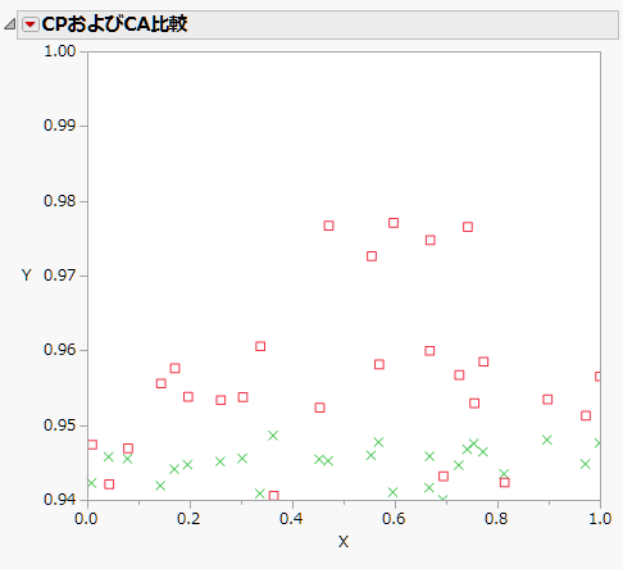
行の属性と行列の例

次の例では、行の属性の値は事前に準備され、座標の情報とともにMarkerルーチンに渡されます。

```
dt = New Table( "CP および CA データの例 ",
   Add Rows( 26 ),
   New Column( "cover_cp",
      Numeric,
      "Continuous",
      Formula( Random Uniform() / 100 + 0.94 )
   ),
   New Column( "cover_ca",
      Numeric,
      "Continuous",
      Formula( Random Uniform() * 0.04 + 0.94 )
   ),
   New Column( "p", Numeric, "Continuous", Formula( Random Uniform() ) )
);
dt << Run Formulas;
```

```
greenMark = Combine States( Marker State( 2 ), Color State( 4 ) );
redDiamond = Combine States( Marker State( 3 ), Color State( 3 ) );
New Window( "CP および CA 比較 ",
   Graph Box(
      Title( "CP および CA 比較 " ),
      FrameSize( 400, 350 ),
      X Scale( 0, 1 ),
      Y Scale( 0.94, 1 ),
      For Each Row(
         Marker( greenMark, {p, cover_cp} );
         Marker( redDiamond, {p, cover_ca} );
      )
   )
);
```

図9.10 行の属性の値と行列の例



データ値へのアクセス

データテーブル内の値を処理する場合、次のような手順が一般的です。

1. 使用したい値が入っているデータテーブルを、現在のデータテーブルとして設定する。すでにデータテーブルの参照がある場合は、その参照を使用することができます。詳細については、「現在のデータテーブルの設定」（288ページ）を参照してください。
2. 使用したい値が入っている行または列を指定し、使用したい値が含まれている列名を指定します。詳細については、「列名による値の設定または取得」（365ページ）を参照してください。

次の例では、「Big Class.jmp」データテーブルを開き（それにより、このデータテーブルが「現在のデータテーブル」になります）、「体重」列の行2を指定します。ログを見ると、123という値が戻されていることがわかります。これは、行2の「Louise」の体重です。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt:weight[2];
  123
```

列名による値の設定または取得

列を参照する最も簡単な方法は、名前で参照することです。同名のグローバル変数と列がある場合は、混同を避けるため、接頭演算子「:」を使って列名の適用範囲を指定してください。

現在の行にあるセルの値を設定するには、列名と新しい値を指定します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
Row() = 5; // 行 5 を選択する
dt:年齢 = 19; // 行 5 の「年齢」を 19 に設定する
dt:名前 = "Sam"; // 行 5 の「名前」を Sam に設定する
```

行10にあるセルの値を設定するには、添え字で行番号を指定します。

```
dt:age[10] = 20; // 行 10 の「年齢」の値を 20 に設定する
```

何も含まれていない添え字は現在行を表すため、**:年齢[]**は**:年齢**と同じです。

行16にあるセルの値を取得するには、列名を指定します。

```
Row() = 16; // 現在の行を 16 にする
myGlobal = :年齢; // 行 16 の年齢を変数に保存する
:年齢; // 行 16 の年齢、14 を戻す
Show( :年齢 ); // 年齢 = 14; を戻す
```

特定の行にあるセルの値を取得するには、列名と行番号を指定します。次の2つの例は、どちらも「Big Class.jmp」の行12の**「年齢」**である「13」を戻します。

```
:年齢[12];
myGlobal = :年齢[12];
```

データ列の参照の値を取得するには、Column()とAs Column()を使用します。詳細は、「列の参照によるセル値へのアクセス」（318ページ）を参照してください。

メモ:

- 行番号を指定しなかった場合は、現在の行が戻されます。
- 現在の行を参照するときは、**:年齢[ ]**のように、空白の添え字を使用します。
- 必ず、存在しているテーブルの行を指定してください。デフォルトの行番号はゼロなので、行ゼロを参照する**:name**というステートメントを指定すると、「無効な行番号です。」というエラーが表示されます。

データ値にアクセスするその他の方法

データテーブル、行、列は、他の方法で指定することもできます。これら3つの要素を1つの式で指定するには、次のように二項演算子と添え字を使います。

```
dt: 年齢 [2] = 12;// テーブル、列、および行
```

複数の行を対象にする場合は、行番号のリストまたは行列で添え字を使用できます。

```
年齢 [i] = 3;
年齢 [{3, 12, 32}] = 14;
list = 年齢 [{3, 12, 32}]; // 値をリストに入れる
vector = 年齢 [1 :: 20]; // 値を行列に入れる
dt[1,1] = dt[2,1] // Big Class.jmp で、(行 1、列 1) の
   // “KATIE” が (行 2、列 1) と同じ “LOUSIE” になる
dt[0,{height}] = dt[0,{weight}]; // 全員の身長がそれぞれの体重の値になる
dt[1,{height,weight}] = dt[2,4::5]; // 行 1 の身長と体重が
   //Louise のものと同じになる
dt[[5 3 1], 0] = .; // 行 5、3、1 をすべて欠測値に設定する。右側に添え字のある
   // 行列またはデータテーブルの場合は、添え字を昇順にしない方がよい
```

値の変更に関するメモ

データテーブル内の値を変更するたびに、最新の情報が表示されるように、メッセージが送られます。しかし、スクリプトに多数の変更箇所がある場合、この方法では更新を完了するのに時間がかかってしまいます。

変更速度を上げるには、変更する前に Begin Data Update を使って更新メッセージを止め、すべて変更し終わってから End Data Update を屋って更新メッセージを解放し、表示を更新します。

```
dt << Begin Data Update;
...<変更作業>...
dt << End Data Update;
```

必ず End Data Update メッセージを送ってください。そうしないと、何らかの方法で強制的に更新しない限り、表示は更新されません。

メモ: Begin Data Update は、データ値を変更する以外のテーブル操作によるデータの再描画には影響しません。たとえば、列を削除または追加する場合は、データテーブルが更新された後、データ値の更新が開始されます。

データテーブルへのメタデータの追加

データテーブルは、サブジェクトの特定のセットに関するさまざまな変数の計測値である観測データを格納します。ただし、JMP データテーブルには、**メタデータ**、またはデータについてのデータも格納できます。メタデータには、次のようなものがあります。

- テーブル変数（メモなどのテキスト文字列）
- プロパティ（スクリプトなどの式）
- スクリプト
- 計算式

テーブル変数

テーブル変数は、「ノート」のような単一のテキスト文字列を保存するために使います。変数の働きを理解するために、まず、Get Table Variableメッセージを送って変数の既存の値を取得してみましょう。

```
dt = Open( "$SAMPLE_DATA/Solubility.jmp" );
dt << Get Table Variable( "ノート" );
```

異なる溶剤で化合物の溶解度を測定したデータ。このテーブルには、6つの水性/無極性システムの72の有機溶質のLogP(partition coefficient)が表示されている。

次に、文字列の既存の値を、Set Table VariableまたはNew Table Variableで変更し、再びGet Table Variableを使って文字列が更新されたことを確認します。

```
dt << Set Table Variable( "ノート", "化合物の溶解度" );
```

または

```
dt << New Table Variable( "ノート", "化合物の溶解度" );
dt << Get Table Variable( "ノート" );
```

"化合物の溶解度"

次の例は、データテーブルに2つの新しいテーブル変数を追加します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
myvar = "これは、「JMP Big Class」サンプルデータのマイバージョンです。";
dt << Set Table Variable( "key1", myvar );
dt << Set Table Variable( "key2", myvar );
```

値を設定したとき、指定した変数が存在しない場合のみ、新しい変数が作成されます。同名のテーブル変数を2つ追加した場合、変数は1つしか作成されません。

テーブルスクリプト

JSLを使って、新しいスクリプトの追加、スクリプトの実行、スクリプトの取得ができます。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << New Script( "二変量の例", Bivariate( Y( :Name("体重(ポンド)")), X( :Name("身長(インチ)") ) ) );
dt << Get Table Variable( "二変量の例" );
  // Bivariate( Y( Name("体重(ポンド)") ), X( Name("身長(インチ)") ) )を戻す
dt << Run Script( "二変量の例" )
```

次の例は、前述のスクリプトで作成されたテーブルスクリプトの内容を置き換えます。

```
dt << Set Property( "二変量の例", Bivariate( Y( :Name("体重(ポンド)") ), X( :Name("身長(インチ)") ), Fit Line ) ); // 「二変量の例」スクリプトに Fit Line を含める
```

同僚に電子メールで送ったり、スクリプトの一部として使ったりするために、データテーブルをテキスト形式で表したものが必要な場合があります。Get Scriptを使うと、データテーブルの情報を再構成するスクリプトを取得できます。次の例は、「Big Class.jmp」を開き、データ、テーブル変数、および列プロパティをログに出力します。その出力の一部をここに示します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Get Script;
  New Table( "Big Class",
    Add Rows( 40 ),
    New Script(
      ["en" => "Distribution",...],
      Distribution(
        Continuous Distribution( Column( :Name("体重(ポンド)") ) )
        Nominal Distribution( Column( :年齢 ) )
      ) ),
```

データテーブルを開いたときにスクリプトを自動的に実行する

On OpenまたはOnOpenという名前のテーブルスクリプトは、データテーブルを開くときに自動的に実行できます。デフォルトでは、スクリプトを実行するか確認するメッセージが表示されます。その際の選択は記憶され、現在のJMPセッションで再度そのデータテーブルを開くときは同じ選択が適用されます。

On Openスクリプトを作成するには、次のアクションのいずれかを実行します。

- [スクリプトの保存] > [データテーブルへ] オプションを使用してスクリプトを作成した後、プロパティ名をダブルクリックし、名前をOn Openに変更します。
- New Scriptメッセージを使用してスクリプトを保存します。

次の例では、「Big Class.jmp」にOnOpenスクリプトを作成します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << New Script(
  "OnOpen", // スクリプトを作成する
```

```
    sortedDt = dt << Sort( By( :名前 ), Output Table Name( "並べ替えた Big Class" ) )
    // データを並べ替え、新しいデータテーブルを出力する
);
```

JMPの［OnOpenスクリプトの評価］という環境設定では、いつスクリプトを実行するかを定義します。デフォルトでは、スクリプトを実行するか確認するメッセージが表示されます。この環境設定で、On Openスクリプトを常に実行するか、まったく実行しないように指定することもできます。

```
Preference( Evaluate OnOpen Scripts( "always" ) );
Preference( Evaluate OnOpen Scripts( "never" ) );
Preference( Evaluate OnOpen Scripts( "prompt" ) ); // デフォルト設定
```

他のプログラムを実行するOn Openスクリプトは、環境設定に関わらず実行されません。予防措置として、他者から受け取ったデータテーブルを開く際はスクリプトが自動実行されないよう設定することをお勧めします。

メモ: スクリプトで新しいデータテーブルを作成し、On Open()関数を含める場合は、データテーブルの作成後ではなく作成前にOn Open()が呼び出されます。

計算式

Suppress Formula Evalメッセージは、自動評価を実行するかどうかを指定するブール値の引数をとります。データテーブルに多数の変更を加える必要があり、ステップ間で計算式が更新されるのを待ちたくない場合は、式の自動評価機能をオフにできます。

```
dt << Suppress Formula Eval( 1 );
dt << Suppress Formula Eval( 0) ;
```

すべてのデータテーブルで自動評価をさせないようにするには、Suppress Formula Evalコマンドを使い、全体にわたって自動評価をオフにします。このコマンドの機能は、前のメッセージと同じですが、データテーブルオブジェクトには送りません。

```
Suppress Formula Eval( 1 ); // セッション内のすべての計算式を静的にする
Suppress Formula Eval( 0 ); // セッション内のすべての計算式を動的にする
```

計算式は、列に組み込まれているときは評価されません。強制的に評価しようとしても、今度はバックグラウンドで評価されます。スクリプト実行中の列の値に依存するような場合は、これが問題となる可能性があります。評価を制御する手段が必要な場合は、EvalFormulaコマンドまたはRun Formulasコマンドを使用します。

1つの列の評価を実行するには、その列にEval Formulaコマンドを送ります。計算式の節の後で、列を作成するコマンド内でも実行できます。その例を示します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << New Column( "比率",
   Numeric,
   Formula( :Name("身長(インチ)") / :Name("体重(ポンド)") ),
   Eval Formula
);
```

dt << Run Formulasは、他の計算式を評価した後で評価するために保留されている式も含め、データテーブルのすべての式の評価を実行します。この関数は、すべての列の計算式を実行する場合に便利です。

ヒント： この方法は、EvalFormulaより適しています。EvalFormulaも計算式を評価しますが、バックグラウンドで再評価されないようにはしません。バックグラウンドで行われるタスクは、計算式が正しい順序で評価されるよう、十分な注意を払う必要があります。

Run Formulasコマンドをデータ列に送った場合は、コマンドを送った時点で評価が実行されますが、そのためにスケジュール設定されてペンディング中だった計算式が本来の時間に評価されなくなるわけではありません。そのため、コマンドをデータテーブルとデータ列にも送ると、計算式が2回評価される場合があります。2回評価されることは、乱数関数が含まれている計算式では望ましい場合もありますが、設定されている乱数シード値に依存する場合は望ましくないこともあります。同一の値のセットを生成させるために乱数関数とRandom Reset(seed)の機能を使用している場合には、Run Formulasコマンドを使うと2回目の評価を行わずに済みます。

メモ: すべてのプラットフォームでRun Formulasコマンドがデータテーブルに送られ、すべての計算式の評価が完了してから、分析が開始されます。

計算後の値を設定する

col << Set Each Value(expression)は、データテーブルの行ごとに式を計算し、結果を列に割り当てます。この式は、計算式としては格納されません。

メタデータの削除

テーブルに含まれるテーブル変数、テーブルプロパティ（スクリプトなど）、計算式は、次のコマンドを使って削除することができます。

```
dt << Delete Table Variable( name );
dt << Delete Table Property( name );
col << Delete Formula;
col << Delete Property( name );
col << Delete Column Property( name );
```

計算

この節では、列または行ごとに統計量を事前計算する関数について説明し、計算式エディタの裏でJSLの式がどのように動作しているかを示します。

事前計算される統計量

JMPには特殊な「事前計算」関数、Col Maximum、Col Mean、Col Minimum、Col N Missing、Col Number、Col Quantile、CV（Coefficient of Variation：変動係数）、Col Standardize、Col Std Dev、Col Sum、Maximum、Mean、Minimum、NMissing、Number、Std Dev、Sumがあります。

メモ：統計量は Summarize（「要約統計量をグローバル変数に格納する」（295 ページ））でも計算されます。Summarize の名前付き引数でこれらの事前計算統計関数と同じ名前のものがありますが、それらの引数が事前計算統計関数を呼び出すということではありません。たまたま名前が一致しているだけです。

統計量はすべて**事前計算**されます。つまり、JMPは指定された行または列で統計量を一度計算しており、その結果を定数として使っているのです。一度計算された統計量が何度も繰り返して使われるので、同等の計算式で算出された結果を使うよりも計算効率が良くなります。

JMPは、スクリプト内に事前計算関数を見つけると、その関数をすぐに計算し、以後はその結果を定数として使います。そのため、事前計算関数を使うと、列ごとの計算結果を行ごとの計算に使用できるようになります。たとえば、列の計算式内でCol Meanを使う場合は、まず指定された列の平均を計算し、次に、各行に関する残りの計算式の計算でその結果を定数として使います。たとえば、事前計算した平均と標準偏差を使って列を標準化する計算式などが考えられます。

```
( :Name("身長(インチ)")-Col Mean( :Name("身長(インチ)") ) ) / Col Std Dev( :Name("
身長(インチ)") )
```

「Big Class.jmp」データセットの場合、Col Mean(:Name("身長(インチ)"))は62.55、Col Std Dev(:Name("身長(インチ)"))は4.24です。したがって、上の計算式では、各行の「身長(インチ)」の値から62.55を引いた後で4.24で割ります。

メモ：事前計算関数は、行の除外の属性を無視するので、除外された行が計算に含まれます。除外された行を無視して要約統計量を得るには、「一変量の分布」プラットフォームを使ってください。

列ごとの関数

名前が「Col」で始まる関数はすべて、**列ごと**、つまり指定された列の値を上から下に評価し、単一の数値を戻します。たとえば、Col Mean(:Name("身長(インチ)"))は、列「身長(インチ)」のすべての行の値の平均を算出し、それをスカラー値の結果として戻します。次に例を挙げます。

```
Average Student Height = Col Mean( :Name("身長(インチ)") );
Height Sigma = Col Std Dev( :Name("身長(インチ)") );
```

Col関数を使うと、欠測値のコードなどの列プロパティにより、意図と異なる計算につながるデータ値が割り当てられる場合があります。たとえば、「欠測値のコード」列プロパティが**x1**列に割り当てられていて、「999」が欠測値として扱われるとします。別の列に、平均を求める計算式があるとします。欠測値ではなく「999」の値を使って平均を計算したいという場合は、計算式でCol Stored Value()を指定します。

```
Mean( Col Stored Value( :x1 ), :x2, :x3 )
```

行ごとの関数

次に示す「Col」が**付いていない**関数は、指定された変数の値について**行ごと**に動作し、結果を戻します。たとえば、Mean(:Name("身長(インチ)"), :Name("体重(ポンド)"))は、データテーブルの現在の行の**「身長(インチ)」**と**「体重(ポンド)」**の平均を算出します。行ごとの統計量は、適切なデータテーブルの行コンテキスト内で使われた場合にだけ有効です。いくつかの例を挙げます。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );

Row() = 7;
::scalar = Mean( :Name("身長(インチ)"), :Name("体重(ポンド)") );
// 行7のデータで求められたスカラー値をグローバル変数に割り当て

dt << New Column( "計算列",
   Formula( Mean( :Name("身長(インチ)"), :Name("体重(ポンド)") ) / 年齢 )
);
// データテーブル内に計算式の列を作成

vector = J( 1, 40 ); // 結果を入れる1x40 の行列を作成
For Each Row( vector[Row()] = Mean( :Name("身長(インチ)"), :Name("体重(ポンド)") )
  ); // ベクトルを設定
```

行ごとの関数は、次のように、ベクトル（列ベクトル）またはリストの引数をとることもできます。

```
myMu = Mean( [1 2 3 4] );
mySigma = Std Dev( {1, 2, 3} );
```

計算式エディタの計算式

自動的に評価され、列のセルの値を生成する計算式を列に保存できます。計算式を開くと、計算式を構造的に編集するための計算式エディタインターフェースが表示されます。計算式はJSLにより実装されているので、計算式エディタ内の式をダブルクリックすれば、JSLテキスト形式の式を取得できます。テキストは編集でき、フォーカスされていないときは、構造形式に戻ります。

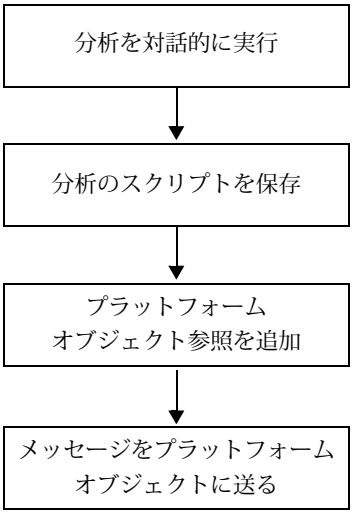
メモ: 計算式エディタウィンドウで作成した計算式の列と、New Column(..., Formula(...))やCol << Formula(...)などのコマンドを使ってJSLから直接作成した計算式の間に違いはありません。

第10章

プラットフォームのスクリプト
分析の作成、反復、変更

同じ分析を頻繁に実行する場合は、そのスクリプトを作成してプロセスを自動化できます。そうすれば、誰でもそのスクリプトを実行することにより、毎回同じ結果を導き出すことができます。そのためにはまず、通常通りにJMPを使って分析を対話的に実行し、その後、分析を再現するためのスクリプトを保存します。そのスクリプトは、修正を加えることにより、さらにカスタマイズできます。

図10.1 プラットフォームのスクリプトを作成するための典型的なワークフロー



この章では、レポートではなく、プラットフォームのスクリプトについて説明しています。**プラットフォーム**オブジェクト参照と**レポート**オブジェクト参照では、受け取るJSLメッセージのタイプが異なります。プラットフォームは、検定を実行したり、プロットを作成することなどができます。レポートは画像をコピーしたり、ディスプレイボックスを選択したり、アウトラインノードを閉じたりできます。レポートのスクリプトについては、「表示ツリー」章の「JMPのレポートの操作」（414ページ）を参照してください。

ヒント： さらにスクリプトに関する情報が必要な場合には、スクリプトの索引（[**ヘルプ**] > [**スクリプトの索引**]）および『スクリプト構文リファレンス』を参照してください。

プラットフォームのスクリプト例

JSLに慣れていない場合は、まずJMPで分析を対話的に実行し、そこからスクリプトを保存するとよいでしょう。JMPがスクリプトを生成してくれます。必要に応じてプラットフォームオブジェクトを作成したり、そのオブジェクトにメッセージを送ったりするなど、そのスクリプトに対して追加や変更を行ってください。

概要

この例は、次のステップで構成されています。

1. 「分析を起動し、そのスクリプトをウィンドウに保存する」
2. 「プラットフォーム参照オブジェクトを追加してメッセージを送る」
3. 「データテーブル参照を追加し、スクリプトを保存する」

分析を起動し、そのスクリプトをウィンドウに保存する

1. ［**ヘルプ**］>［**サンプルデータライブラリ**］を選択し、「Big Class.jmp」を開きます。

   この架空のデータセットには、40人の学生に関する名前、年齢、性別、身長(インチ)、および体重(ポンド)が含まれています。
2. ［**分析**］>［**二変量の関係**］を選択します。
3. 「**身長(インチ)**」を選択し、［**Y, 目的変数**］をクリックします。
4. 「**年齢**」を選択し、［**X, 説明変数**］をクリックします。
5. ［**OK**］をクリックします。

   学生の年齢と身長の関係を示す一元配置プロットが表示されます。
6. この一元配置分析の赤い三角ボタンをクリックし、次のオプションを選択します。
   - ［**平均/ANOVA**］
   - ［**平均の比較**］>［**各ペア,Studentのt検定**］
7. 一元配置分析の赤い三角ボタンをクリックし、［**スクリプトの保存**］>［**スクリプトウィンドウへ**］を選択します。

その結果、次のスクリプトが表示されます。

```
Oneway(
   Y( :Name("身長(インチ)")),
   X( :age ),
   Each Pair( 1 ),
   Means( 1 ),
   Mean Diamonds( 1 ),
   Comparison Circles( 1 )
);
```

次の点に注意してください。

- このスクリプトは、一元配置オブジェクトを戻すOneway()関数の呼び出しで始まっています。
- Oneway()関数の括弧の中には、オブジェクトの作成方法をOneway関数に指示するための引数が含まれています。
  - 最初の2つの引数YとXは起動時に指定しなければならなかったものです。
  - その後の4つの引数Each Pair、Means、Mean Diamonds、およびComparison Circlesはオプションです。
  - ［各ペア,Studentのt検定］オプションを選択したときに、Each PairとComparison Circlesの機能が有効になりました。
  - ［平均/ANOVA］オプションを選択したときに、MeansとMean Diamondsの機能が有効になりました。
- スクリプト内では、オプションの有効と無効がブール値の引数0（無効）と1（有効）で表されます。

プラットフォーム参照オブジェクトを追加してメッセージを送る

1. （オプション）スクリプト内を右クリックし、［**ウィンドウ内にログを表示**］を選択します。

   これで、ログへの出力がスクリプトの下に表示され、見やすくなります。

2. 一元配置プラットフォームオブジェクトへの参照を格納するJSL変数を指定します。

   この例では、oneObjという名前を使用します。

   ```
   oneObj = Oneway(
      Y( :Name("身長(インチ)")),
      X( :age ),
      Each Pair( 1 ),
      Means( 1 ),
      Mean Diamonds( 1 ),
      Comparison Circles( 1 )
   );
   ```

   これでプラットフォームオブジェクトを参照し、そのオブジェクトにメッセージを送れるようになりました。

3. ［**スクリプトの実行**］  をクリックします。

   これで一元配置オブジェクトが作成され、oneObj変数が設定されます。それでは、「分散が等しいことを調べる検定」レポートを有効にするようプラットフォームに指示するメッセージを送ってみましょう。

4. スクリプトの末尾に、oneObj << Unequal Variances(1)という行を追加します。スクリプトは次のようになります。

   ```
   oneObj = Oneway(
      Y( :Name("身長(インチ)")),
      X( :age ),
      Each Pair( 1 ),
      Means( 1 ),
   ```

```
   Mean Diamonds( 1 ),
   Comparison Circles( 1 )
);
oneObj << Unequal Variances( 1 );
```

5. JSLのUnequal Variances行を強調表示し、[**スクリプトの実行**] をクリックします。

ヒント:Ctrlキーを押しながらRキーを押しても、JSLの強調表示した行を実行できます。

レポートウィンドウの下部に、「分散が等しいことを調べる検定」レポートが表示されます。ここで、そのレポート（およびグラフ）だけを表示したまま、レポートの残りの部分を閉じたいとします。

6. 以下の行をスクリプトに追加します。

```
rep = Report( oneObj );
rep[" 一元配置の分散分析 "] << Close( 1 );
rep[" 平均の比較 "] << Close( 1 );
```

Report()関数は「一元配置」プラットフォームのレポートオブジェクトを戻し、repというJSL変数にそのレポートへの参照を保存します。このレポートオブジェクトには、メッセージを送ったり、特定のレポートアウトラインを閉じるといったアクションを実行することができます。

7. スクリプト内のこれらの3行を強調表示し、[**スクリプトの実行**] をクリックします。

レポートウィンドウで、「一元配置の分散分析」レポートと「平均の比較」レポートが閉じます。つまり、表示されているのは最初のグラフと「分散が等しいことを調べる検定」レポート（「Welchの検定」レポートを含む）だけです。

この時点で「Big Class.jmp」サンプルデータテーブルを閉じ、その後スクリプトを実行しようとすると、データテーブルを開くよう促すウィンドウが表示されます。もっとも良い方法は、Open()関数により使用するデータテーブルを開いてから、「一元配置」プラットフォームを呼び出すことです。

データテーブル参照を追加し、スクリプトを保存する

1. 次の行をスクリプトの最初の行として追加します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
```

この指定により、スクリプトの実行時に該当のデータテーブルが開いていない場合は開くようになります。

完成したスクリプトは次のようになります。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
oneObj = Oneway(
   Y(:Name(" 身長 ( インチ )")),
   X(: 年齢 ));
   Each Pair( 1 ),
   Means( 1 ),
   Mean Diamonds( 1 ),
   Comparison Circles( 1 )
);
oneObj << Unequal Variances( 1 );
```

```
rep = Report( oneObj );
rep[" 一元配置の分散分析 "] << Close( 1 );
rep[" 平均の比較 "] << Close( 1 );
```

スクリプトが完成したら、今後このスクリプトにアクセスできるように保存します。

2. スクリプトウィンドウから［**ファイル**］>［**名前を付けて保存**］を選択します。スクリプトの名前を入力し、任意のディレクトリに保存します。

プラットフォームへのメッセージの送信

プラットフォームにメッセージを送るには、そのメッセージの送り先となるプラットフォームオブジェクトを作成する必要があります。それには、Send 演算子（<<）を使用します。

次の行では、oneObj という JSL 変数を使って「一元配置」プラットフォームへの参照を指定しています。

```
oneObj = Oneway( Y(:Name(" 身長 ( インチ )")), X(: 年齢 ) );
```

この後、Send 演算子を使ってプラットフォームオブジェクトにメッセージを送ります。次の例は、「分散が等しいことを調べる検定」レポートを有効にするよう指示するメッセージを送ります。

```
oneObj << Unequal Variances( 1 );
```

特定のオブジェクトに送ることのできるメッセージを表示するには、次のいずれかの操作を行います。

- オートコンプリートショートカットを使用する。新しい行で、オブジェクト名、Send 演算子（<<）の順に入力し、Ctrl キーとスペースキーを押すか（Windows の場合）、Ctrl キーと Option キーとスペースキーを押します（Macintosh の場合）。次の例は、Windows で oneObj プラットフォームオブジェクトに送ることのできるメッセージを表示します。

  ```
  oneObj <<  (Ctrl+ スペース)
  ```

  メッセージのリストが表示されるので、挿入するメッセージをクリックします。
- Show Properties () 関数を使用する。括弧内にオブジェクトを指定します。次の例は、oneObj プラットフォームオブジェクトに送ることのできるメッセージを表示します。

  ```
  Show Properties (oneObj)
  ```

  ログにメッセージのリストが表示されます。

メッセージと引数の規則

- ほとんどのメッセージの場合は、ブール値を取るオプションの引数を省略すると、オプションはオンになります。たとえば、以下のメッセージはすべて、分散が等しいことを調べる検定を作成します。

  ヒント : 推奨されるのは最初の 2 つです。

  ```
  oneObj << Unequal Variances;
  oneObj << Unequal Variances( 1 );
  oneObj << Unequal Variances( "true" );
  oneObj << Unequal Variances( "yes" );
  ```

```
oneObj << Unequal Variances( "present" );
oneObj << Unequal Variances( "on" );
```

メモ：行の属性メッセージの場合は、引数を省略すると、オプションが**トグル**します。つまり、オプションがオフの場合は、メッセージによってオプションがオンになり、オプションがオンの場合は、メッセージによってオプションがオフになります。また、"toggle"、"switch"、"flip" の各キーワードを r << Excluded("toggle") のように使用することもできます。

- JMPオプションの名前にカンマやスラッシュで区切られた複数の語が含まれている場合は（[**平均/ANOVA**] や [**T2 乗, T^2**]）、これらの名前のうち任意の**一語**を使用できます。あるいは、Name() 関数に入れて使用する場合は、カンマやスラッシュを含めた名前をそのまま使用できます。たとえば、次のメッセージはいずれも平均/ANOVAの検定のレポートを出力します。

```
oneObj << 平均( 1 )
oneObj << ANOVA( 1 )
oneObj << Name( "平均/ANOVA" )( 1 )
```

- JMPのメニューにオプションが表示される場合、それに対応するスクリプトのメッセージはそのオプションだけ（親メニューは含めない）となります。たとえば、一元配置分析の赤い三角ボタンをクリックして表示される［ノンパラメトリック］メニューには、［Wilcoxon検定］などの複数のオプションがあります。この場合、スクリプトでは単に［Wilcoxon検定］（Wilcoxon test）をオプションまたはメッセージとして使用します。

```
oneObj = Oneway( Y( :Name("身長(インチ)") ), X( 年齢 ), Wilcoxon Test( 1 ));
oneObj << Wilcoxon Test( 1 );
```

- JMPのメニューに含まれているのがオプションではなく数値の場合は、親メニューを指定した後、値を引数として指定します。たとえば、一元配置分析の赤い三角ボタンをクリックして表示される［α水準の設定］（Set Alpha Level）メニューには、［0.10］、［0.05］、［0.01］、および［その他］の値があります。スクリプトで0.01を指定するには、次のような行を追加します。

```
oneObj << SetAlphaLevel(0.01);
```

複数のメッセージの送信

複数のメッセージを送るには、<<演算子を追加するか、Sendの引数を追加します。

```
dist << Quantiles( 1 ) << Moments( 1 ) << More Moments( 1 ) << Horizontal Layout(
  1 );
Send( dist, Quantiles( 1 ), Moments( 1 ), More Moments( 1 ), Horizontal Layout(1));
```

<<は**省略（eliding）演算子**なので、複数のオペランドを連結し、オペランドがグループ化されている場合とそうでない場合とでは異なる動作をします。<<記号を使って複数のメッセージを1つにまとめ、左から右へ順に実行できます。

メッセージのリストにより、メッセージを1つにまとめて送ることもできます。

```
dist << {Quantiles( 1 ), Moments( 1 ), More Moments( 1 ), Horizontal Layout( 1 )};
```

これらのアプローチは、どのメッセージも結果として値を戻さないと仮定した場合にはうまく機能します。しかし、あるメッセージを別のメッセージの**結果**に送る場合は、括弧を追加してグループ化する必要があります。次の例では、最後の指定が正しく記述されたものでであり3行目の指定は間違いです。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
// 次の行は、Fit Mean メッセージが元のデータテーブルオブジェクトに送られるため、間違い
bv = dt << Run Script( "Bivariate" ) << Fit Mean( 1 );
// 次の行は、Fit Mean メッセージが二変量のオブジェクト／レポートに送られるため、正しい
bv2 = ( dt << Run Script( "Bivariate" ) ) << Fit Mean( 1 );
```

オブジェクトに適したメッセージの検索

メッセージによってはすべてのプラットフォームオブジェクトに適用できるものと、特定のプラットフォームオブジェクトにしか適用できないものがあります。プラットフォームオブジェクトを作成した後、どのようなメッセージをそのオブジェクトに送ることができるのかを確認するには、以下のいずれかの方法を使用します。

- JMPの［スクリプトの索引］を使用する。
  1. ［**ヘルプ**］>［**スクリプトの索引**］を選択します。
  2. 関心のあるオブジェクトの名前を入力します。
  3. 表示されるリストで該当するオブジェクトをクリックします。そのオブジェクトに適用できるメッセージが項目リストに表示されます。
- プラットフォームオブジェクトの作成後、`Show Properties( obj )`関数を使用する。`obj`をプラットフォームオブジェクトの名前に置き換えます。そのオブジェクトが受け取ることのできるすべてのメッセージのリストと簡単な説明がログに表示されます。Show Propertiesの出力の詳細については、「Show Propertiesリストの読み方」（380ページ）を参照してください。

ヒント：`Show Properties()`関数はデータテーブルやディスプレイボックスに対しても使えます。「データテーブル」章の「データテーブルオブジェクトに送ることができるメッセージ」（273ページ）および「表示ツリー」章の「<< 演算子」（423ページ）を参照してください。

- JMPの起動ウィンドウやレポートウィンドウを調べる。これらのウィンドウで見つかるほとんどのオプションは、JSLでも同じ名前や引数を使って利用できます。

Show Properties リストの読み方

Show Properties は、データテーブル、分析プラットフォーム、ディスプレイなどのスクリプト可能なオブジェクトに送ることのできるメッセージの一覧をログにテキスト形式で出力します。各メッセージタイプはそれぞれカテゴリ別に分類されており、括弧内に表示されます。メッセージタイプには以下のものがあります。

メッセージタイプ	説明	例
[アクション]	JMP インターフェース内のオプションに対応している。	Redo Analysis [アクション] (Rerun this same analysis in a new window.The analysis will be different if the data has changed.)
[アクションの選択] [選択肢]	引数に指定する特定の選択肢をリストする。	Fit Polynomial [アクションの選択] {2, quadratic, 3, cubic, 4, quartic, 5, 6} Draw [選択肢] {Filled, Outlined, Filled and Outlined}
[ブール]	オプションのオンとオフを切り替える。通常、引数は1または0です。引数を指定しないでメッセージを送ると、そのたびに逆の状態に切り替わります。[デフォルトはオン]は、そのオプションがデフォルトでオンになっていることを示します。	Show Points [ブール] [デフォルトはオン]
[サブテーブル]	サブメニュー内のオプションを指す。親項目ではなくオプション自体を指定します。	Script [サブテーブル] Redo Analysis [アクション] (Rerun this same analysis in a new window.The analysis will be different if the data has changed.) Save Script to Datatable [アクション] (Return to the launcher for this analysis.)
[新規エンティティ]	ユーザーインターフェースに新しいウィンドウを開く。	Prediction Interval [新規エンティティ] (Prediction Interval to contain a single future observation or the mean of m future observations.)

たとえば、次のJSLを実行してください。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
biv = Bivariate(Y( :Name("身長(インチ)") ), X( :年齢 ) );
Show Properties( biv );
```

このbivオブジェクトがサポートするメッセージのタイプは、サブテーブル、ブール、アクション、およびアクションのものです。選択肢と新規エンティティのタイプのメッセージはサポートされていません。

分析に使用する列の指定

スクリプトを対話的に作成した場合は、すでに起動ウィンドウで列を指定しているため、それらを指定し直す必要はありません。しかし、スクリプトを最初から自分で作成する場合は、通常、分析する列を指定します。たとえば、次の行は「一変量の分布」プラットフォームを起動し、**「身長 (インチ)」**列と**「体重 (ポンド)」**列をＹ変数として指定しています。

```
Distribution( Y( :Name("身長 ( インチ )"),:Name("体重 ( ポンド )")));
```

列参照の作成

ヒント： 列に送ることのできるすべてのメッセージを表示するには、［**ヘルプ**］＞［**スクリプトの索引**］＞［**Data Table**］＞［**Column Scripting**］を選択してください。

列への参照を作成すると、その列にアクセスしたり、メッセージを送ったりできるようになります。それぞれの列参照をJSL変数内に格納すると、ＹやＸの役割、およびスクリプト内のその他の場所で、列名の代わりにそれらの変数を使用できます。

たとえば「Big Class.jmp」では、次のようにして**「体重 (ポンド)」**の列参照として**Ycol**を、**「身長 (インチ)」**の列参照として**Xcol**を作成できます。最後の行はメッセージを**Xcol**に送ってデータタイプを取得しています。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
Ycol = Column( "体重 ( ポンド )" );
Xcol = Column( "身長 ( インチ )" );
Xcol << Get Data Type;
```

ログには、**Xcol**（**「身長 (インチ)」**列）のデータタイプである“数値”が出力されます。

複数の列名を一度に指定する

多数の列名がある場合は、各列を個々にリストするのではなく、リストを列の引数として使用できます。次の例は、リストから列名を取得し、その各列について一変量の分布を実行します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
Distribution( Y( 1::N Col( dt ) ) );
```

ヒント： スクリプトの作成時に列名がわからない場合は、この例のように列インデックスを使用します。

ユーザによる列の指定

分析列を自分で指定する代わりに、ユーザが列を指定するように設定することもできます。その場合でも、列の指定後の動作は自分で決めることができます。このためには空の指定を使用します。次の例は、「一変量の分布」起動ウィンドウを実行します。

```
Distribution();
```

このシナリオでは、「一変量の分布」起動ウィンドウが表示され、ユーザが分析する列を指定できます。ユーザが列を選択し、[OK] をクリックした後は、スクリプトの実行が継続され、スクリプトで指定されている出力が表示されます。

By 変数の指定

JMP の多くのプラットフォームでは、列を By 変数として指定できます。スクリプトでこれを行うには、プラットフォームコマンド内に By() 関数を含め、各列を引数としてリストします。

次の例では、40 人の学生の「名前」、「年齢」、「身長 (インチ)」、および「体重 (ポンド)」を含む「Big Class.jmp」データテーブルを使用しています。「性別」を By 変数として使用し、さまざまなあてはめを追加して、「身長 (インチ)」に対する「体重 (ポンド)」の「二変量の関係」レポートを作成してみましょう。

By 変数を使用した「二変量の関係」レポートの作成

1. [ファイル] > [新規作成] > [スクリプト] を選択します。
2. 次の行を追加します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
biv = dt << Bivariate( Y( :Name("体重 (ポンド)") ), X( :Name("身長 (インチ)") ), By(
   :性別 ));
```

最初の行は、「Big Class.jmp」サンプルデータテーブルを開きます。2 行目は、**「性別」**を By 変数として**「体重 (ポンド)」**と**「身長 (インチ)」**の「二変量の関係」レポートを実行する **biv** というプラットフォームオブジェクトを作成します。

3. [スクリプトの実行]  をクリックします。

レポートウィンドウに 2 つのグラフが表示されます。1 つは性別が F、もう 1 つは性別が M のものです。

メッセージをプラットフォーム全体または単一の By 水準に送る

1. ログが表示されていない場合は、ログを表示します。スクリプト内を右クリックし、[ウィンドウ内にログを表示] を選択します。
2. 次の行をスクリプトに追加します。

```
Show( biv );
```

3. Show(biv) の行を強調表示し、[スクリプトの実行]  をクリックします。

ログには、プラットフォームへの単一の参照二変量 [] が戻されるのではなく、biv = {二変量 [], 二変量 []} のような参照のリストが戻されます。これらの2つの参照は、By 変数である「性別」の2つの水準（F と M）に対応しています。

メッセージは、各BY水準に個別に送ることも、すべてのBy水準に送ることもできます。

4. すべての By 水準にメッセージを送って、線形回帰のあてはめを追加します。次の行をスクリプトに追加します。

```
biv << Fit Line;
```

5. 作成したばかりの行を強調表示し、[**スクリプトの実行**]  をクリックします。

レポートウィンドウの両方のグラフに、単回帰直線が追加され、それに対応する「直線のあてはめ」レポートが表示されます。

6. 水準Fにのみメッセージを送って、3次多項式のあてはめを追加します。次の行をスクリプトに追加します。

```
biv[1] << Fit Polynomial( 3 );
```

ヒント：各 By 水準の番号は、レポートウィンドウ内でのその水準の順序に対応します。By 水準の列に「値の順序」列プロパティが含まれている場合は、その順序に従います。

7. 作成したばかりの行を強調表示し、[**スクリプトの実行**]  をクリックします。

レポートウィンドウのFのグラフに多項式のあてはめの曲線が追加され、対応する「多項式のあてはめ」レポートが表示されます。

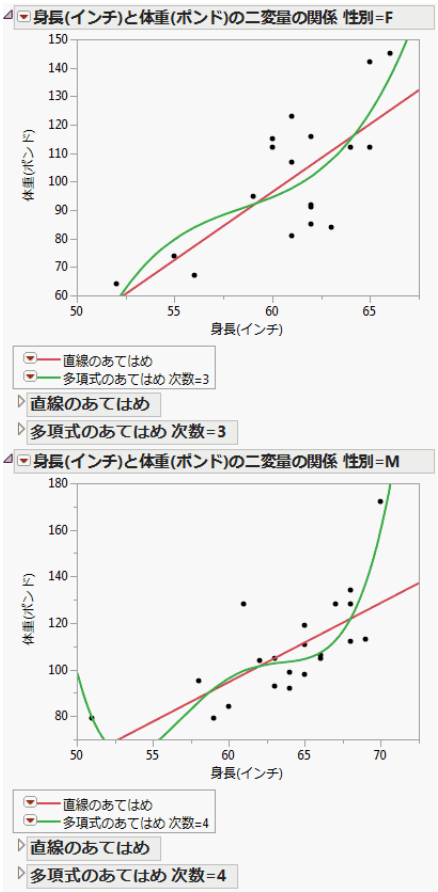
8. 水準Mにのみメッセージを送って、4次多項式のあてはめを追加します。次の行をスクリプトに追加します。

```
biv[2] << Fit Polynomial( 4 );
```

9. 作成したばかりの行を強調表示し、[**スクリプトの実行**]  をクリックします。

レポートウィンドウのMのグラフに4次のあてはめの曲線が追加され、対応する「多項式のあてはめ」レポートが表示されます。

図10.2 Byグループレポート



By() 関数で複数の列を指定した場合は、BY 変数の組み合わせのサブグループごとにグラフが表示されます。上記の例で、By(：性別 , ：年齢) とすると、12 歳の女性、13 歳の女性という具合に 17 歳までのグラフが表示され、同様に 12 歳から 17 歳までの男性のグラフも表示されます。また、12 歳から 17 歳までの男性のグラフも表示されます。

レポートからの結果の抽出

次の例は、By グループを使ってプラットフォームを起動し、各グループから結果（ここでは誤差平方和）を取得しています。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
onew = dt << Oneway( x( ：年齢 ), y( :Name("身長(インチ)") ), by( ：性別 ), anova );
// onewはプラットフォームのリストを保持するJSL変数
r = onew << Report;
// rはレポートのリスト
nBy = N Items( r );
```

```
// nBy は生成されるレポートの数
vc = J( nBy, 1, 0 );
// vc は、レポート数と同じ行数と 1 つの列を持ち、すべての値がゼロに設定された行列
For( i = 1, i <= nBy, i++,
// 各レポートについて以下を行う
   vc[i] = r[i]
// vc[i] は i 番目のレポートの誤差平方和
[Outline Box( " 分散分析 " ),
// このアウトラインと
   Column Box( " 平方和 " )][2]
);
// この列を検索し、2 つ目の値を得る
Show( vc );
// デバッグ、ログでこの値を確認
Summarize( byValues = By( :性別 ) );
// byValues が「性別」列の値のリストになる
New Table( " 誤差平方和 " )
// 2 行（M,F）と 2 列の新しいテーブルを作成する
<< New Column( " 性別 ",
   character,
   width( 8 ),
   values( byValues )
// 「性別」という新しい列を作成する
) << New Column( " 誤差平方和 ", Numeric, "Continuous", Values( vc ) );
// 「誤差平方和」という新しい列を作成する
```

値または列によるフィルタリング

Where節を使用すれば、特定の値や列でフィルタリングすることができます。列参照を含むスクリプトにWhere節を追加する場合は、先にその参照を解決する必要があります。たとえば、次の行はName Expr()関数でYcolおよびXcolという参照を使用しており、Where節を追加して分析対象を女性のみに制限しています。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
Ycol = Column( " 体重 ( ポンド )" );
Xcol = Column( " 身長 ( インチ )" );
biv2 = dt << Bivariate(
Y( Name Expr( Ycol ) ),
X( Name Expr( Xcol ) ),
Fit Line( ),
Where( :性別 == "F" )
);
```

プラットフォームのスクリプトがデータのサブセットを選択するときに、列参照が元のデータテーブルに対して正しく解決されます。

ユーザによる入力を可能にする

Action() 関数をプラットフォームに送ると、指定した式が評価されます。たとえば、起動ウィンドウが表示されるようにしたい場合は、Action メッセージを使用します。ユーザが起動ウィンドウで列を選択し、分析を実行した後に、指定したスクリプトが実行されます。

次の例は、「一変量の分布」、「二変量」、「一元配置」、「分割表」の4つのプラットフォームを起動します。各プラットフォームの起動ウィンドウが表示され、ユーザは分析列を選択し、[OK] をクリックします。各分析のレポートウィンドウが表示され、その後、次のプラットフォームの起動ウィンドウが開きます。この動作が、すべてのプラットフォームが起動されるまで繰り返されます。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Distribution( Action( doit ) );
doit = Expr(
   New Window( "二変量", Bivariate( Action( doit2 ) ) )
);
doit2 = Expr(
   New Window( "一元配置", Oneway( Action( doit3 ) ) )
);
doit3 = Expr(
   New Window( "分割表", Contingency( Action( doit4 ) ) )
);
doit4 = Expr();
```

埋め込まれた赤い三角ボタンのオプションの実行

レポートの先頭以外にも、赤い三角ボタンのあるタイトルバーを持つレポートウィンドウがあります。これら赤い三角ボタンは、プロファイルなど、そのレポートだけに適用するオプションを持つレポートアイテムに表示されます。これらの赤い三角ボタンのメニューにあるオプションを実行するには、Get Scriptable Object メッセージを使用します。次の例は、「寿命の一変量」レポートを作成し、その後、「ハザードプロファイル」の赤い三角ボタンのメニューにある [信頼区間] オプションを無効にしています。

```
dt = Open( "$SAMPLE_DATA/Reliability/Fan.jmp" );
obj = dt << Life Distribution(
       Y( :時間 ),
       Censor( :打ち切りの有無 ),
       << Fit Lognormal
);
// 対数正規分布を使って「寿命の一変量」レポートを実行する
haz_prof = ( report( obj )[ "ハザードプロファイル" ] << Get Scriptable Object);
// 「ハザードプロファイル」の赤い三角ボタンのメニューへの参照を取得する
haz_prof << Confidence Intervals( 0 );
// ハザードプロファイルの信頼区間を無効にする
```

プラットフォームを非表示にする

プラットフォームの分析結果を非表示にしたい場合があります。プラットフォームのウィンドウを表示せず、バックグラウンドでプラットフォームを実行したいという意味です。

スクリプトでシミュレーションやブートストラップ分析を実行する場合には、スクリプトによってプラットフォームが数百回または数千回呼び出されることがあります。しかし、各プラットフォームレポートから得ようとしている結果は1つか2つです。次の例は、シミュレーションを実行した後、結果のテーブルを作成し、それらを「一変量の分布」レポートに表示します。

```
dt = As Table( (0 :: 10)`, << Column Names( {"X"} ) );
// X変数を作成する
Random Reset( 12345 );
// 結果を再現できるように、乱数シードの初期値を設定する
dt << New Column( "Y",
   "Continuous",
   "Numeric",
   Set Formula( :X * 2 + 1 + Random Normal() )
);
// Y値の計算式には乱数の要素を含む
res = []; // 結果の行列を定義する（最初は空）
For( i = 1, i <= 100, i++,
// シミュレーションの実行をループ
   // 実行ごとに回帰直線をあてはめ、パラメータ推定値を保存する
   bv = dt << Bivariate( Y( :Y ), X( :X ), Fit Line( 1 ), << Invisible );
   res |/= (Report( bv )["パラメータ推定値"][
   Number Col Box( "推定値" )] << Get as Matrix)`;
   // 結果を行列の下に追加していく
   bv << Close Window;
   //（非表示ではあるが）ウィンドウを閉じる
   dt:Y << Eval Formula;
   // 新しいY値を生成する
);
dtres = As Table( res, << Column Names( {"切片", "傾き"} ) );
// 結果をテーブルに出力する
dtres << Distribution( Y( :切片, :傾き ) );
//  「一変量の分布」レポートを作成する
```

スクリプトで中間的なステップとしての分析を実行するときに、その分析そのものをユーザに見せる必要がない場合は、invisibleオプションを使って非表示にしてください。このとき、レポート内の特定の結果だけをジャーナルまたはログに出力して、ユーザに見せることもできます。

- 次の例は、「一変量の分布」プラットフォームを非表示で実行し、累積確率プロットだけを新しいジャーナルに出力します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dist = dt << Distribution( Y( :Name("身長（インチ）") ), CDF Plot( 1 ), invisible );
```

```
Report( dist )[" 累積確率プロット "] << Journal;
dist << Close Window;
```

- 次の例は、非表示の「二変量」レポートから *F* 値を抽出し、それをログに出力します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
biv = dt << Bivariate( x( :Name(" 身長 ( インチ )") ), y( :Name(" 体重 ( ポンド )") ),
  invisible );
biv << Fit Line;
r = biv << Report;
fratio = r[ColumnBox( " F 値 " )][1];
r << Close Window;
Show( fratio );
  fratio = 1.15609545178219;
```

ヒント： 非表示のウィンドウによって使用されるリソースは手動で解放する必要があります。スクリプトの中で非表示のウィンドウの役目が終わったときには、そのウィンドウを閉じてください。

［テーブル］メニューにある［サブセット］、［並べ替え］、［列の積み重ね］などのオプションに invisible オプションを使用することもできます。次の例では、サブセットの処理を非表示にしています。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Select Where( :年齢 == 14 );
subDt = dt << Subset( invisible );
subDt << Bivariate( x( :Name(" 身長 ( インチ )") ), y( :Name(" 体重 ( ポンド )") ), Fit
  Line );
```

サブセットを使用せず、WHERE 節を使用して同様の結果を得ることも可能です。

```
subDt << Bivariate( x( :Name(" 身長 ( インチ )") ),y( :Name(" 体重 ( ポンド )") ), Where(
  :年齢 == 14 ), Fit Line );
```

レポートタイトルの指定

起動要求に title コマンドを追加することで、タイトル（プラットフォームのレポートのタイトルバーに表示されるもの）が指定できます。次の例は、二変量の関係で表示される標準のレポートタイトルを、ユーザ定義のもの（My Title）に置き換えます。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Bivariate( x( :Name(" 身長 ( インチ )") ), y( :Name(" 体重 ( ポンド )") ),
  Title( " タイトル " ) );
```

プラットフォームウィンドウの一般メッセージ

ほとんどのプラットフォームに送ることのできる一般メッセージのリストについては、『スクリプト構文リファレンス』の「JSL メッセージ」章を参照してください。

プラットフォーム別 スクリプト作成の注意点

この節では、次のプラットフォームのスクリプトを作成する際の考慮事項について説明します。

カテゴリカル

「カテゴリカル」の自由回答オプションを使用すると、「テキストエクスプローラ」レポートが表示されます。「カテゴリカル」の自由回答オプションを呼び出すスクリプトは、「テキストエクスプローラ」プラットフォームを使用するように修正する必要があります。

上位カテゴリのスクリプト

データセットに順位（たとえば、5 点段階評価など）が関係している場合に、上位 2 段階など、特定のサブセットに含まれる応答の割合を調べたいことがあります。データ内の順位のグループは、列プロパティではなく、上位カテゴリスクリプトオプションを使って定義できます。

次の例は、「**Floss Delimited 2**」という新しい列を作成し、「Before Sleep」および「Wake」という 2 つの値を含む上位カテゴリ（「Sleep Related」）を追加します。

```
dt = Open( "$SAMPLE_DATA/Consumer Preferences.jmp" );
dt << New Column( "デンタルフロス カンマ区切り 2",
   Character,
   Set Property(
```

```
        "Supercategories",
        {Group( "Sleep Related", {"Before Sleep,", "Wake,"} )}
    ), ...);
```

データテーブルの左側にある「列」パネルを下方向にスクロールして、新しい**「デンタルフロス カンマ区切り 2」**列を表示します。列プロパティが設定されていることを示す✱をクリックし、［上位カテゴリ］を選択します。2つの値を含む「Sleep Related」上位カテゴリが表示されます。

「カテゴリカル」の起動コマンドでも上位カテゴリを指定できます。プロパティは、列名の後の括弧の中にリストします。

```
dt = Open( "$SAMPLE_DATA/Consumer Preferences.jmp" );
dt << Categorical(
   Supercategories(
      :デンタルフロス カンマ区切り (
         {Group( "Sleep Related", {"Before Sleep,", "Wake,"} )}
      )
   ),
   Multiple Response( :デンタルフロス カンマ区切り )
);
```

「カテゴリカル」レポートを見ると、「Sleep Related」カテゴリに203個のケースがあることがわかります。この上位カテゴリを作成せずに「カテゴリカル」プラットフォームを実行した場合は、「Before Sleep」と「Wake」が別々のカテゴリとなり、それぞれに143個と60個のケースが表示されます。

上位カテゴリの詳細については、『消費者調査』のカテゴリカルに関する章を参照してください。

管理図

JSLを使用して、管理図を作成したり、テストをカスタマイズしたり、警告スクリプトを実行したりできます。

管理図ビルダーにおけるテストのカスタマイズ

Customize Tests関数を使用すると、カスタムテストを作成し、複数のテストを一度に選択または選択解除できます。テスト名、n、ラベルを指定できます。このオプションは、計量値または計数値の管理図でのみ使用できます。

次の例は、「Test 1」というカスタムテストを作成し、そのテストをオンにします。

```
dt = Open( "$SAMPLE_DATA/Quality Control/Diameter.jmp" );
dt << Control Chart Builder(
   Variables( Subgroup( :日付 ), Y( :直径 ) ),
   Customize Tests( Test 1( 2, "1" ) ), // 説明、テスト番号、ラベル
   Chart( Position( 1 ), Warnings( Test 1( 1 ) ) ), // 「テスト 1」をオンにする
   Chart( Position( 2 ) )
);
```

警告スクリプトの実行

警告スクリプトは、1つ以上のテストで異常が検出された場合に警告をします。警告スクリプトおよびそれに続く JSL スクリプトでは、次の変数を使用できます。

qc_col（列の名前）

qc_test（異常が検出されたテスト）

qc_sample（標本番号）

qc_firstRow（標本の第1行）

qc_lastRow（標本の最終行）

例1: ログへの書き込みを自動化する

警告が自動的に実行されるようにする1つの方法は、作成したスクリプトをデータテーブル内に「**QC Alarm Script**」というテーブルプロパティとして保存する方法です。

メモ: 現在のところ、このスクリプトは管理図ビルダーではサポートされていません。

次の例では、テストで異常が検出されるたびに、メッセージがログに自動的に書き込まれます。

```
dt = Open( "$SAMPLE_DATA/Quality Control/Coating.jmp" );
obj = dt << Control Chart(
   Sample Size( :サンプル ),
   KSigma( 3 ),
   Chart Col( :重量 , XBar, R ),
   Alarm Script(
      Write(
         " テストで異常 ",
         qc_test,
         " 列 ",
         qc_col,
         " サンプル ",
         qc_sample
      )
   )
);
obj << Test 1( 1 );
```

例2: 管理図のテスト結果を音声で通知させる

Speak 関数を使って、管理図のテスト結果を音声で通知させることもできます。以下に例を示します。

```
dt = Open( "$SAMPLE_DATA/Quality Control/Coating.jmp" );
dt << Control Chart(
   Alarm Script(
      Speak(
         Match( QC_Test,
```

```
                1, "1 点が A ゾーンを超えています ",
                QC_Test, 2,
                " 連続した 9 点が C ゾーン以上にあります ", QC_Test,
                5,
                   " 連続した 3 点のうち 2 点が A ゾーン以上に
                   あります "
             )
          )
       ),
       Sample Size( :サンプル ),
       Ksigma( 3 ),
       Chart Col( :重量 , Xbar( Test 1( 1 ), Test 2( 1 ), Test 5( 1 ) ), R )
    );
```

これらのスクリプトで、JSL 警告コマンドの Speak、Write、Mail を使用することができます。

フェーズごとの管理限界の設定

「フェーズ（phase）」とは、データテーブル内の連続するオブザベーションをグループにまとめたものです。たとえば、新しい工程で生産を開始する前と後は異なるフェーズだと定義できます。指定したフェーズ変数の水準ごとに新しいシグマ、限界値のセット、ゾーン、テストの結果が計算されます。

「Diameter.jmp」の各フェーズに対し、限界値を設定する例を次に示します。

```
dt = Open( "$SAMPLE_DATA/Quality Control/Diameter.jmp" );
dt << Control Chart(
   Phase( :フェーズ ),
   Sample Size( :日付 ),
   KSigma( 3 ),
   Chart Col(
      :直径 ,
      XBar(
         Phase Level(
            "1",
            Sigma( .29 ),
            Avg( 4.3 ),
            LCL( 3.99 ),
            UCL( 4.72 )
         ),
         Phase Level(
            "2",
            Sigma( .21 ),
            Avg( 4.29 ),
            LCL( 4 ),
            UCL( 4.5 )
         )
      ),
      R( Phase Level( "1" ), Phase Level( "2" ) )
   )
);
```

一変量の分布

JSL では、「管理限界」列プロパティまたは Spec Limits() 関数を使用して管理限界を設定できます。

```
dt = Open( "$SAMPLE_DATA/Cities.jmp" );
dt << Distribution(
   Column( :オゾン ),
   Fit Distribution(
      Weibull(
         Spec Limits( LSL( 0.075 ), Target( 0.15 ), USL( 0.25 ) )
      )
   )
);
```

K シグマに対する仕様限界を設定するには、Set Spec Limits for KSigma メッセージを使用します。デフォルトでは、仕様限界は両側に設定されます。片側にするには、sided=1 を指定します。

```
dt = Open( "$SAMPLE_DATA/Cities.jmp" );
obj = Distribution( Column( :CO ) );
obj << Fit Distribution(
   LogNormal( Set Spec Limits for KSigma( 3 ) )
);
```

実験計画 (DOE)

ヒント： DOE のスクリプトオプションの完全なリストについては、［**ヘルプ**］>［**スクリプトの索引**］を選択し、メニューから［**オブジェクト**］を選択して、DOE を検索してください。

DOE の作業を再現するために、ほとんどの実験計画（DOE）プラットフォームでは以下のアプローチを利用できます。

- 「実験計画 (DOE)」ウィンドウの赤い三角ボタンのメニューにある［スクリプトをスクリプトウィンドウに保存］オプションを使用すると、入力した作業内容を再現するスクリプトが得られます。このオプションは、「非線形計画」および「タグチ配列」プラットフォームでは使用できません。
- ほとんどの実験計画（DOE）プラットフォームによって作成された出力の計画テーブルでは、「DOE ダイアログ」というスクリプトを使って、その計画テーブルを作成した「実験計画 (DOE)」ウィンドウでの作業を再現できます。このスクリプトには、計画の正確に再現するための乱数シード値（および該当する場合は開始点の数または開始列）も含まれています。

多くの実験計画（DOE）プラットフォームで作成された計画テーブルには、適切な分析を実行するための1つまたは複数のスクリプトも含まれています。これらのスクリプトには、計画に適した設定があらかじめ入力されています。

乱数シード値

実験計画（DOE）プラットフォームを使用して計画を作成する際、JMP は乱数シード値を使用してランダムな要素を持つ特定の動作を制御します。乱数シート値では以下のことを制御できます。

- ［応答のシミュレート］オプションによる応答の乱数シミュレーション
- データテーブル作成時における実験順序のランダム化
- 最適計画を探索するときの初期計画の選択

スクリプトを使用して計画やシミュレートした応答を再現するには、それらを生成する乱数シード値を指定する必要があります。ランダム開始点を使用する計画の場合は、計画を作成する前にスクリプトで乱数シード値を設定します。シミュレートした応答や実験順序を制御するには、計画テーブルを作成する前にスクリプトで乱数シード値を設定します。

次の例は、カスタム計画を作成し、乱数シード値を設定して、計画を作成します。

```
DOE(
   Custom Design,
   Add Factor( Continuous, -1, 1, "X1", 0 ),
   Add Factor( Continuous, -1, 1, "X2", 0 ),
   Set Random Seed( 34067086 ),
   Make Design
);
```

カスタム計画または主効果スクリーニング計画の再現

カスタム計画と主効果のスクリーニング計画（「スクリーニング計画」プラットフォームで構築したもの）は、計画の検索に費やす最大秒数（計画の検索時間）を内部的に割り当てることによって生成されます。「計画の検索時間」のデフォルト設定は、計画の複雑さによって異なります。同じ乱数シード値を使って2つのカスタム計画または主効果スクリーニング計画を構築する場合でも、コンピュータの処理能力によっては異なる計画が作成されることがあります。

［スクリプトをスクリプトウィンドウに保存］オプションを使っていずれかの計画のスクリプトを保存する際、または「DOE ダイアログ」スクリプトをデータテーブルに保存する際には、スクリプトによって乱数シード値と開始点の数が指定されます。開始点の数は、計画の検索時間中に使用されるランダム開始点の数です。［乱数シード値の設定］オプションと［開始点の数］オプションを組み合わせることにより、計画を再現できるようになります。

メモ: 独自のスクリプトを作成する場合は、スクリプト内でオプションを指定する順序が結果に影響することを念頭に置いてください。たとえば、計画を作成する前に計画オプションを指定する必要があります。特に乱数シード値を使って計画を構築する場合は、計画を生成する前に乱数シード値を設定してください。

グローバル変数

以下に示すグローバル変数は、検索アルゴリズムの開始や調整に使用できます。

Starting Design 開始点の計画を指定できます。次の例は、ランダム開始点の計画を指定した行列に置き換えます。

```
DOE Starting Design = matrix;
```

開始点の計画を指定した場合は、その計画だけから検索が開始されます。

K Exchange Value 座標交換アルゴリズムは、すべての反復における交換に対して、すべての行を考慮します。1回の反復で交換を考慮する行数を制限すれば、変更される可能性の高い少数の行だけが考慮されるようになります。次の行は、各反復で変更される可能性の高い3行だけを考慮するようアルゴリズムに指示を出します。

```
DOE K Exchange Value = 3;
```

Bayes Diagonal $X'X$行列の対角要素に追加するベクトルを定義します。この新しい行列は、D最適計画を見つけるのに使用されます。次の例は、vectorの要素を$X'X$行列の対角要素に追加します。

```
Bayes Diagonal = vector:
```

モデルのあてはめ

以下に、「モデルのあてはめ」プラットフォームのスクリプトを作成する際に起動ウィンドウの動作を制御するためのヒントをいくつか挙げます。

- 起動ウィンドウを開いたまま、同時にモデルをあてはめるには、以下のいずれかを行ってください（どちらも同じ結果が得られます）。
  - スクリプトに Run Model メッセージを含める。
  - Run メッセージと Keep Dialog Open(1) メッセージの両方を含める。
- 起動ウィンドウを表示せずにモデルを実行するには、スクリプトに Run メッセージを含めます。

データテーブルの「モデル」スクリプト

データテーブルに**「モデル」**スクリプトが含まれており、「モデルのあてはめ」プラットフォームを起動して対話的に操作する場合は、テーブルスクリプトによって割り当てられている役割に基づいて、「モデルのあてはめ」起動ウィンドウに予め情報が入力されます。

あてはめのグループ

1つのプロファイルで複数のモデルを表示したい場合は、Fit Group関数を使用します。最小2乗、非線形、ニューラル、Gauss過程、および混合モデルといったモデルのあてはめを1つのウィンドウにまとめ、結果を1つにまとめたプロファイルを表示できます。次の例は、標準最小2乗モデルとGauss過程モデルの両方を作成します。

```
dt = Open( "$SAMPLE_DATA/Tiretread.jmp" );
```

```
obj = dt << Fit Group(
   Fit Model(
      Y( :摩擦 ),
      Effects(
         :シリカ & RS,
         :シラン & RS,
         :硫黄 & RS,
         :シリカ * :シリカ,
         :シリカ * :シラン,
         :シラン * :シラン,
         :シリカ * :硫黄,
         :シラン * :硫黄,
         :硫黄 * :硫黄
      ),
      Personality( "Standard Least Squares" ),
      Emphasis( Minimal Report ),
      Run()
   ),
   Gaussian Process(
      Y( :硬度 ),
      X( :シリカ, :シラン, :硫黄 ),
      Set Correlation Function( "Cubic" )
   )
);
```

レポートで「あてはめのグループ」の赤い三角ボタンをクリックし、[**プロファイル**]を選択します。そして、下方向にスクロールして「予測プロファイル」を表示します。同じプロファイル内に、最小2乗モデルの「摩擦」行、そしてGauss過程モデルの「硬度」行が表示されているのがわかります。

効果

モデルの効果には複数の列のリストを指定でき、そのための特別な構文があります。

```
Effect( 効果のリスト、効果のマクロのリスト、またはその両方のリスト );
```

1つの列名、複数の列の交差にはアスタリスク（*）、枝分かれする列には添え字の括弧 [ ] を使って効果を指定します。追加の効果オプションは、アンパーサンド（&）文字の後に指定できます。例をいくつか示します。

```
A,                  // 1つの列名単独で1つの主効果
A*B,                // 交差効果、交互作用、または多項式
A[B],               // 枝分かれ
A*B[C D],           // 交差および枝分かれ
effect&Random,      // 変量効果
effect&LogVariance, // 対数分散効果
effect&RS,          // 応答曲面の項
effect&Mixture,     // 配合の効果
effect&Excluded,    // モデル引数を持たない効果
effect&Knotted,     // 節点スプライン効果
```

効果のマクロは以下のとおりです。

```
Factorial( columns ),   // 完全実施要因
Factorial2( columns ),  // 最大 2 次までの交互作用のみ
Polynomial( columns ),  // 2 次の多項式のみ
```

MANOVA の応答と効果

個別の応答関数（Response Function）の分析を指定するには、添え字付きの Response を使います。

```
manovaObj << ( Response[ 1 ] << { 応答オプション } );
manovaObj << ( Response[ " 対比 " ] << { 応答オプション } );
```

各応答関数でそれぞれ Custom Test メッセージを使用できます。

```
Custom Test( matrix, <Power Analysis( ... )>, <Label( "..." )> )
```

行列（*matrix*）の各行は、モデル内のすべての引数の係数を示します。

個別の効果の検定を指定するには、名前または番号の添え字をつけた Effect を使います。

```
manovaObj << ( Response[1] << ( Effect[" モデル全体 "] << { 効果オプション } ) );
manovaObj << ( Response[1] << ( Effect[i] << { 効果オプション } ) );
```

効果には次のように番号が付けられます。

- 切片には 0
- 標準の効果には 1、2、3、...
- 「モデル全体」の検定には *n*+1（*n* は切片を**含まない**効果の数）

各応答関数の各効果で、次のメッセージを使用できます。行列（*matrix*）の各行には、効果のすべての水準に対する係数があります。

```
Test Details( 1 ),
Centroid Plot( 1 ),
Save Canonical Scores,
Contrast( matrix, <Power Analysis(...)> );
```

たとえば、次の JSL スクリプトは効果の検定の詳細をレポートウィンドウに追加します。

```
dt = Open( "$SAMPLE_DATA/Dogs.jmp" );
manObj = dt << Fit Model(
   Y(Name("log( ヒスタミン 0)"),Name("log( ヒスタミン 1)"),Name("log( ヒスタミン
   3)"),Name("log( ヒスタミン 5)")),
   Effects( ヒスタミンの消耗 y or n, 薬剤 , 薬剤 * ヒスタミンの消耗 y or n),
   Personality( "MANOVA" ),
   Run Model
);
manObj << Response Function( "Contrast" );
manObj << (Response[" 対比 "] << (Effect[" モデル全体 "] <<
Test Details( 1 )));
```

```
// Test Details メッセージを、レポートウィンドウの「対比」の下の「モデル全体」に送る
manObj << (Response[1] << (Effect[3] << Test Details( 1 )));
// Test Details を応答 1（対比）と効果 3（薬剤 * ヒスタミンの消耗 y or n）に送る
```

Send 関数

ヒント： << 演算子は Send() 関数と同じ結果をもたらします。

複数の応答がある場合には、特定の応答列のあてはめにメッセージを送ることができます。次の JSL を使用します（*responseName* は特定の応答）。

```
fitObj << ( responseName << { オプション, ...});
```

2つ目の Send() 関数は指定された応答を見つけ、その応答にメッセージのリストを送ります。

メモ： 1つの Send() 関数でメッセージを直接 *fitObj* に送ると、そのメッセージはすべての応答に送られます。

次のスクリプトでは、最後の行でメッセージが「摩擦」応答の AICc レポートに送られます。

```
dt = Open( "$SAMPLE_DATA/Tiretread.jmp" );
fitObj = dt << Fit Model(
   Y( :摩擦, :引張応力, :伸び, :硬度 ),
   Effects(
      :シリカ & RS,
      :シラン & RS,
      :硫黄 & RS,
      :シリカ * :シリカ,
      :シラン * :シリカ,
      :シラン * :シラン,
      :硫黄 * :シリカ,
      :硫黄 * :シラン,
      :硫黄 * :硫黄
   ),
   Personality( "Standard Least Squares" ),
   Run
);
fitObj << (:摩擦 << {AICc( 1 )}); // Box-Cox 変換プロットの下に表示される
```

個々の効果にメッセージを送るには、さらに入れ子にします。

```
fitObj << ( responseName << ((effectName) << effectOption ));
```

標準最小2乗

固定効果と複数の Y 変数のみの標準最小 2 乗モデルの場合は、Y 変数をまとめて、または個別にモデルをあてはめることができます。一部の行に欠測値がある場合は、次の規則が適用されます。

- スクリプトでは、デフォルトで、複数のYがまとめてあてはめられます。モデルは、すべてのY変数が欠測値でない行だけを使って各Yをあてはめます。たとえば、1つまたは複数の欠測値がある行はモデルから**除外**されます。また、スクリプトに Run("Fit Together") を明示的に含めても、同じ結果が得られます。
- Run("Fit Separately") オプションを使用すると、このモデルは各Yを、そのYが欠測値でないすべての行を使ってあてはめます。たとえば、ある行に1つまたは複数のYの欠測値があっても、その行はモデルに**含まれる**可能性があります。

次のスクリプトは応答 Y をまとめてあてはめます。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Fit Model(
   Y( :Name("身長(インチ)"), :Name("体重(ポンド)") ),
   Effects( :性別 ),
   Personality( "Standard Least Squares" ),
   Emphasis( "Minimal Report" ),
   Run // または Run( "Fit Together" )
);
```

次のスクリプトは応答 Y を個別にあてはめます。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Fit Model(
   Y( :Name("身長(インチ)"), :Name("体重(ポンド)") ),
   Effects( :性別 ),
   Personality( "Standard Least Squares" ),
   Emphasis( "Minimal Report" ),
   Run( "Fit Separately" )
);
```

JMP でモデルを実行するとき、ユーザは応答をまとめてあてはめるのか、それとも個別にあてはめるのかを選択するよう求められます。「モデルのあてはめ」起動ウィンドウには［個別にあてはめ］オプションがあり、デフォルトでオフになっています。欠測値の処理方法の詳細については、『基本的な回帰モデル』の標準最小2乗に関する章を参照してください。

メモ: 変量効果のあるモデルの場合、Yはデフォルトで個別にあてはめられます。

計算式デポ

JSLを通じて「計算式デポ」プラットフォームのレポートにコマンドを送るには、まず空の計算式デポを作成する必要があります。「計算式デポ」レポートへの参照が作成されたら、そこにコマンドを送ることができます。次のスクリプトは「計算式デポ」レポートを開き、デポにモデルを追加します。

```
dt = Open( "$SAMPLE_DATA/Liver Cancer.jmp" );
fd = Formula Depot(); // 新しい計算式デポを開く
obj1 = dt << Run Script( "Lasso Poisson分布 検証列" );
model = obj1 << (Fit[1] << Publish Prediction Formula); /* 予測式を発行する。以下のスクリプトは、「計算式デポ」レポートを開き、デポにモデルを追加して、計算式のスコアリングコードを生成した後、コードをファイルに保存し、すべてのウィンドウを閉じる。*/
case_name = "fitLS_prediction";
dest_dir = "c:\";  //必要に応じてこのパスを変更

dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
Print( "Running " || case_name );
fm_ls = Fit Model(
   Y( :weight ),
   Effects( :年齢, :性別, :Name("身長(インチ)") ),
   Personality( Standard Least Squares ),
   Emphasis( "Minimal Report" ),
   Run
);
fd = fm_ls << Publish Prediction Formula;

Try(
   Print( "Generate Python Code" );
   fd << Generate Python Code();
   Wait( 1 );
   cw = Window( "Python コード出力" );
   ed = cw[Script Box( 1 )];
   code = ed << Get Text;
   Save Text File( dest_dir || case_name || ".py", code );
   cw << Close Window( No Save );
   Print( "Done." );
,
   Print( "Code generation failed for " || case_name || ", reason: " );
   Show( exception_msg );
);
fdw = Window( "Liver Cancer - 計算式デポ" );
fdw << Close Window( No Save );
Close( dt, No Save );
```

「ニューラル」と「ニューラルネット」

「ニューラル」（Neural）プラットフォームは「ニューラルネット」（Neural Net）プラットフォームに代わるものです。「ニューラルネット」を使ったスクリプトは、将来廃止され、動かなくなる可能性があります。そのため、新しいスクリプトを作成する際には「ニューラル」プラットフォームを使用し、「ニューラルネット」を使用した古いスクリプトは「ニューラル」を使用するように更新することをお勧めします。

「PLS 回帰」と「PLS」

「PLS」プラットフォームの後継として「PLS回帰」（Partial Least Squares）プラットフォームが導入されました。PLSを使用するスクリプトは、今後動作しなくなる可能性があります。そのため、新しいスクリプトを作成する際には「PLS回帰」プラットフォームを使用し、「PLS」を使用した古いスクリプトは「PLS回帰」を使用するように更新することをお勧めします。

工程能力

「工程能力」（Process Capability）プラットフォームが新しくなりました。以前の「工程能力」（Capability）を使用するスクリプトは将来使用できなくなる可能性があります。そのため、新しいスクリプトを作成する際には新しい「工程能力」プラットフォームを使用し、以前の「工程能力」を使用した古いスクリプトは新しいものを使用するように更新することをお勧めします。

JSLスクリプトでの仕様限界の指定

仕様限界は、JSL スクリプト、列プロパティ、仕様限界データテーブルから読み込むことができます。以下のスクリプトはスクリプトで直接仕様限界を指定する例です。

```
dt = Open( "$SAMPLE_DATA/Cities.jmp" );
dt << Process Capability(
   Process Variables( :オゾン , :一酸化炭素 , :二酸化硫黄 , :一酸化窒素 ),
   Spec Limits(
      オゾン ( LSL( 0.075 ), Target( 0.15 ), USL( 0.25 ) ),
      一酸化炭素 ( LSL( 5 ), Target( 7 ), USL( 12 ) ),
      二酸化硫黄 ( LSL( 0.01 ), Target( 0.04 ), USL( 0.09 ) ),
      一酸化窒素 ( LSL( 0.01 ), Target( 0.025 ), USL( 0.04 ) )
   )
);
```

仕様限界データテーブルとJSLの使用

JSL で仕様限界のデータテーブルを読み込む場合、2 つの種類（横長、縦長）を区別するための指定は必要ありません。この例では、CitySpecLimits.jmp データテーブルを使用しています。仕様限界データテーブルを Import Spec Limits() 式の中に入れます。

```
dt = Open( "$SAMPLE_DATA/Cities.jmp" );
dt << Process Capability(
   Process Variables( :オゾン , :一酸化炭素 , :二酸化硫黄 , :一酸化窒素 ),
```

```
      Spec Limits(
         Import Spec Limits(
            "$SAMPLE_DATA/CitySpecLimits.jmp"
   )));
```

三次元散布図

お使いのコンピュータがアクセラレーションをサポートしており、三次元散布図をよりすばやく表示したい場合は、［ハードウェアアクセラレーションを使用］オプションを使用するか、または JSL で Scene3DHardwareAcceleration グローバル変数を指定できます。値 1 でオンになり、値 0 でオフになります。

```
dt = Open( "$SAMPLE_DATA/Solubility.jmp" );
Scene3DHardwareAcceleration = 1;
dt << Scatterplot 3D(
   Y( :Name( "1- オクタノール " ), : エーテル , : クロロフォルム , : ベンゼン , : 四塩化炭素 , :
   ヘキサン )
);
```

外れ値を調べる

「外れ値を調べる」プラットフォームには、一変量や多変量のデータで外れ値を識別、探索、および管理するためのオプションがあります。

「Probe.jmp」サンプルデータテーブルには、9999 という外れ値を含む列がいくつかあります。産業分野の多くで、9 の羅列からなる数字が欠測値コードとして使用されます。次の例は、9 の羅列からなる最大値を含む列、**VDP_PCOLL**、**VDP_PINNBASE**、および **VDP_PINPBASE** を選択し、これらの列に「欠測値のコード」列プロパティを追加します。その後、列を再スキャンし、レポートからこれらの列が削除されます。

```
dt = Open( "$SAMPLE_DATA/Probe.jmp" );
obj = Explore Outliers(
   Y(
      :VDP_M1,
      :VDP_M2,
      :VDP_NBASE,
      :VDP_NEMIT,
      :VDP_NENBNI,
      :VDP_NSINK,
      :VDP_PBASE,
      :VDP_PBL,
      :VDP_PCOLL,
      :VDP_PEMIT,
      :VDP_PINNBASE,
      :VDP_PINPBASE,
      :VDP_PSINK,
      :VDP_SICR
   ),
```

```
   Quantile Range Outliers( 1 ),
   Show only columns with outliers( 1 ),
);
obj << Add Highest Nines to Missing Value Codes( :VDP_PCOLL, :VDP_PINNBASE,
   :VDP_PINPBASE );
obj >> Rescan;
```

各プラットフォームのスクリプト専用メッセージおよび引数

プラットフォームに送ることのできる一部のメッセージや引数はスクリプト専用です。JMP の起動ウィンドウやレポートには、これらに対応するオプションがありません。ただし、スクリプト専用のオプションの中には、環境設定に表示されるものもあります。

この節では、特定のプラットフォームに対するスクリプト専用のメッセージおよび引数について説明します。

ヒント： これらのメッセージや引数の詳細、あるいは具体的な例については、［スクリプトの索引］（**［ヘルプ］ > ［スクリプトの索引］**）を参照してください。

二変量

Bivariate オブジェクトに対する `Fit Where` メッセージは、指定したグループのデータだけに対してあてはめを行います。

```
biv_object << Fit Where( WHERE_clause );
```

選択モデル

「選択モデル」プラットフォームでパラメータを推定する際には、次の JSL 関数を使って許容できる収束基準を指定できます。

```
Convergence Criterion( fraction );
```

クラスター分析

Hierarchical Cluster オブジェクトに対する次のメッセージは、JSL でのみ使用できます。

- `hier_cluster_object << Get Column Names`
  `// 変数間クラスターにおけるクラスター順に列名を戻す`
- `hier_cluster_object << Get Clusters`
  `// クラスター番号のベクトルを戻す`
- `hier_cluster_object << GetDisplayOrder`
  `// クラスター内の各行の表示順序のベクトルを戻し、表示されない行は欠測値として扱う`
- `hier_cluster_object << GetColumnDisplayOrder`
  `// 変数間クラスターにおける各列の表示位置のベクトルを戻す`

```
- hier_cluster_object << Get Distance Matrix
  // 階層型クラスターに使用された距離行列を戻す
```

KMeans クラスター分析

KMeans cluster オブジェクトに対する次のメッセージは、JSL でのみ使用できます。

```
kmeans_cluster_object << Get Statistics
// 各クラスター内の平均と標準偏差を戻す
```

管理図

次の Control Chart() 関数の引数は、JSL でのみ使用できます。

```
Use Excluded Points on MR( Boolean )
// 移動範囲の計算において、除外された点を含める
```

累積損傷

Cumulative Damage オブジェクトに対する次のメッセージは、JSL のみで使用できます。

```
cumulative_damage_object << Get Results
// モデルをあてはめた結果のリストを戻す
```

カスタムプロファイル

Custom Profiler オブジェクトに対する次のメッセージは、JSL でのみ使用できます。

```
profiler_object << Get Objective
// 目的関数の現在の値を戻す
profiler_object << Objective Formula
// 目的関数の計算式を戻す
profiler_object << Get Objective Formula
// 目的関数の計算式を式として戻す
```

一変量の分布

Distribution オブジェクトに対する次のメッセージは、JSL でのみ使用できます。

```
dist_object << ( Fit Handle[n] << ...);
// 特定の分布のあてはめにコマンドを送る
```

次の Fit Distribution() 関数の引数は、「パラメータ推定値」表を右クリックして表示されるメニューから［データテーブルに出力］オプションを選択するのと同じ働きをします。

```
dt = Open( "$SAMPLE_DATA/Cities.jmp" );
obj = Distribution( Column( :一酸化炭素 ) );
obj << Fit Distribution( LogNormal( Make Parms Table ) );
```

寿命の二変量

Fit Life by X オブジェクトに対する次のメッセージは、JSL でのみ使用できます。

- `fit_life_object << Set Scriptables( ... );`
  `// 出力の複数のセクションで、プロファイルに対するスクリプト可能なオプションを設定する`
- `fit_life_object << Get Results`
  `// 各分布の推定値、標準誤差、共分散行列、収束結果を戻す`

モデルのあてはめ

以下の節では、MANOVA、一般化線形モデル、名義および順序ロジスティック、標準最小2乗のスクリプト専用メッセージについて説明します。

MANOVA

Fit MANOVA オブジェクトに対する次のメッセージは、JSL でのみ使用できます。

```
fit_model_object << ( Response[1] << (Effect[1] << ... ) );
```

3 つ目のメッセージは次のいずれかになる場合があります。

```
Test Details // 個々の効果に対する検定の詳細の表示／非表示を切り替える
Centroid Plot // 個々の効果に対する正準プロットの表示／非表示を切り替える
Save Canonical Scores // 個々の効果に対する正準スコアを保存する
Contrast // モデルの効果のさまざまな水準を対比する、カスタマイズした F 検定を実行する
```

次の例は、最初の応答と効果のペア（このデータでは「Log(ヒスタミン 0)」と「薬剤」）の正準スコアを保存します。

```
dt = Open( "$SAMPLE_DATA/Dogs.jmp" );
obj = dt << Fit Model(
   Y( :Name("Log( ヒスタミン 0)"), :Name("Log( ヒスタミン 1)"), :Name("Log( ヒスタミン
   3)"), :Name("Log( ヒスタミン 5)") ),
   Effects( : 薬剤 , : ヒスタミンの消耗 y or n, : 薬剤 * : ヒスタミンの消耗 y or n ),
   Personality( "Manova" ),
   Run( Response Function( Sum ) )
);
obj << (Response[1] << (Effect[1] << Save Canonical Scores) );
```

一般化線形モデル

Fit Generalized Linear Model オブジェクトに対する次のメッセージは、JSL でのみ使用できます。

```
Parametric Formula( )
// パラメータを含んだモデルの計算式を、現在のデータテーブルの新しい列に保存する
```

名義および順序ロジスティック

Fit Nominal Logistic and Fit Ordinal Logistic オブジェクトに対する次のメッセージは、JSL でのみ使用できます。

- `fit_model_object << Get SAS Data Step`
  `// データにスコアをつけるための SAS DATA ステップを作成する`
- `fit_model_object << Get MM SAS Data Step`
  `// SAS Model Manager に登録できる SAS コードを作成する`

標準最小2乗

次の JSL メッセージは、あてはめられたモデルから、分散成分、p 値、パラメータ推定値などの要求された項目を戻します。

```
fit_model_object << Get Variance Components( );
fit_model_object << Get Effect Names( );
fit_model_object << Get Effect PValues( );
fit_model_object << Get Estimates( );
fit_model_object << Get Parameter Names( );
fit_model_object << Get Random Effect Names( );
fit_model_object << Get Std Errors( );
fit_model_object << Get X Matrix( );
fit_model_object << Get XPX Inverse( );
fit_model_object << Get Y Matrix( );
```

Standard Least Squares オブジェクトに対する次のメッセージは、JSL でのみ使用できます。

```
obj << Get SQL prediction expression;
```

次の例は、予測式を SQL 式として保存し、それらをログに出力します。

```
dt = Open( "$SAMPLE_DATA/Tiretread.jmp" );
obj = dt << Fit Model(
   Y( :摩擦 , :硬度 ),
   Effects( :シリカ , :シラン , :硫黄 ),
   Personality( "Standard Least Squares" ),
   Run
);
code = obj << Get SQL Prediction Expression;
```

生存時間 (パラメトリック) のあてはめ

次の JSL メッセージは、あてはめられたモデルから、パラメータ名、パラメータ推定値、標準誤差などの要求された項目を戻します。

```
survival_object << Get Parameter Names
survival_object << Get Estimates
survival_object << Get Std Errors
```

```
survival_object << Get Effect Names
survival_object << Get Effect PValues
```

寿命の一変量

次の JSL メッセージは、指定されたプロットがレポートに表示されないようにしたり、あてはめられたモデルから結果、推定値、計算式などの要求された項目を戻したりします。

```
life_dist_object << Suppress Plot(plot_name)
life_dist_object << Get Results
life_dist_object << Get Estimates
life_dist_object << Get Formula
```

多変量の相関

Multivariate オブジェクトに対する次のメッセージは、JSL でのみ使用できます。

```
multivariate_object << Get Correlation Matrix;
multivariate_object << Get Inv Correlation Matrix;
multivariate_object << Create SAS Job( );
// SAS を使って同様の推定方法を実行する SAS PROC MIXED コードを生成する
```

ニューラル

メモ:「ニューラル」（Neural）プラットフォームは「ニューラルネット」（Neural Net）プラットフォームに代わるものです。「ニューラルネット」を使ったスクリプトは、将来廃止され、動かなくなる可能性があります。そのため、新しいスクリプトを作成する際には「ニューラル」プラットフォームを使用し、「ニューラルネット」を使用した古いスクリプトは「ニューラル」を使用するように更新することをお勧めします。

Neural オブジェクトに対する次のメッセージは、JSL でのみ使用できます。

```
neural_object << Get NBoost
// ブースティングに使用されるモデルの数を戻す
```

Neural Fit オブジェクトに対する次の JSL メッセージは、あてはめられたモデルからエントロピー R2 乗、一般化 R2 乗、誤分類率などの要求された項目を戻します。

```
neural_object << ( Fit[n] << Get RSquare Training )
neural_object << ( Fit[n] << Get RSquare Validation )
neural_object << ( Fit[n] << Get RSquare Test )
neural_object << ( Fit[n] << Get Gen RSquare Training )
neural_object << ( Fit[n] << Get Gen RSquare Validation )
neural_object << ( Fit[n] << Get Gen RSquare Test )
neural_object << ( Fit[n] << Get Misclassification Rate Training )
neural_object << ( Fit[n] << Get Misclassification Rate Validation )
neural_object << ( Fit[n] << Get Misclassification Rate Test )
```

```
neural_object << ( Fit[n] << Get Average Log Error Training )
neural_object << ( Fit[n] << Get Average Log Error Validation )
neural_object << ( Fit[n] << Get Average Log Error Test )
neural_object << ( Fit[n] << Get RMS Error Training )
neural_object << ( Fit[n] << Get RMS Error Validation )
neural_object << ( Fit[n] << Get RMS Error Test )
neural_object << ( Fit[n] << Get Average Absolute Error Training )
neural_object << ( Fit[n] << Get Average Absolute Error Validation )
neural_object << ( Fit[n] << Get Average Absolute Error Test )
neural_object << ( Fit[n] << Get ROC Area Training )
neural_object << ( Fit[n] << Get ROC Area Validation)
neural_object << ( Fit[n] << Get ROC Area Test)
neural_object << ( Fit[n] << Get Confusion Matrix Training)
neural_object << ( Fit[n] << Get Confusion Matrix Validation)
neural_object << ( Fit[n] << Get Confusion Matrix Test)
neural_object << ( Fit[n] << Get Confusion Rates Training)
neural_object << ( Fit[n] << Get Confusion Rates Validation)
neural_object << ( Fit[n] << Get Confusion Rates Test)
neural_object << ( Fit[n] << Get Seconds)
```

パレート図

Pareto Plot オブジェクトに対する次のメッセージは、JSL でのみ使用できます。

```
No Plot( Boolean );
// パレート図のアウトラインを閉じる
```

パーティション

Partition オブジェクトに対する次の JSL メッセージは、あてはめられたモデルからエントロピー R2 乗、一般化 R2 乗、誤分類率などの要求された項目を戻します。

```
partition_object << Get RSquare Training
partition_object << Get RSquare Validation
partition_object << Get RSquare Test
partition_object << Get Gen RSquare Training
partition_object << Get Gen RSquare Validation
partition_object << Get Gen RSquare Test
partition_object << Get Misclassification Rate Training
partition_object << Get Misclassification Rate Validation
partition_object << Get Misclassification Rate Test
partition_object << Get ROC Area Training
partition_object << Get ROC Area Validation
partition_object << Get ROC Area Test
partition_object << Get Average Log Error Training
partition_object << Get Average Log Error Validation
```

```
partition_object << Get Average Log Error Test
partition_object << Get RMS Error Training
partition_object << Get RMS Error Validation
partition_object << Get RMS Error Test
partition_object << Get Average Absolute Error Training
partition_object << Get Average Absolute Error Validation
partition_object << Get Average Absolute Error Test
partition_object << Get Seconds
partition_object << Get SAS Data Step
partition_object << Get Tolerant SAS Data Step
partition_object << Get MM SAS Data Step
partition_object << Get MM Tolerant SAS Data Step
```

曲面プロット

Surface Plot オブジェクトに対する次のメッセージは JMP でも制御できますが、赤い三角ボタンのメニューには対応するオプションがない場合があります。

- `surface_plot_object << Mode( "Isosurface" | "Sheet" )`
  `// Isosurface は、3 つの独立変数（X 軸、Y 軸、X 軸）を使用し、その 3 変数上に従属変数の曲面をプロットする。Sheet には、2 つの独立した X、Y 変数に従属する 1 つの応答 Z がある。`
- `surface_plot_object << Set Z Axis( col, Current Value( n ) )`
  `// 指定の列を Z 軸に設定する`
- `surface_plot_object << Set Variable Axis( col, Current Value( n ) )`
  `// 指定の各列を独立変数の軸に設定する`
- `surface_plot_object << Response( ... )`
  `// 点をプロットしたい応答列を 4 つまで指定する`
- `surface_plot_object << Formula( ... )`
  `// 「従属変数」セクションで表示されている順番に従い、列に保存されている計算式を割り当てる`

テキストエクスプローラ

Text Explorer オブジェクトに対する次の JSL メッセージは、JSL でのみ使用できます。

- `text_explorer_object << Minimum Frequency for Phrase( n )`
  `// 句リストに含める句の最小度数を指定する`
- `text_explorer_object << Save Indicators for Most Frequent Words( ... )`
  `// Save Document Term Matrix() の別名`
- `text_explorer_object << Save Word Table( ... )`
  `// Save Term Table() の別名`
- `text_explorer_object << Score Words by Column( ... )`
  `// Save Terms by Column() の別名`
- `text_explorer_object << Set Regex( ... )`
  `// Regex によるトークン化で使用されるデフォルトの正規表現を置き換える`
- `text_explorer_object << Save Regex Column( string )`

```
// トークン化された結果を含む新しい列を作成する
- text_explorer_object << Add Delimiters( string )
// 区切り文字のリストに、ユーザが指定した文字列を追加する
- text_explorer_object << Set Delimiters( string )
// 区切り文字のリストを、ユーザが指定した文字列で置換する
- text_explorer_object << Add Stop Words( list )
- text_explorer_object << Add Phrases( list )
- text_explorer_object << Add Stem Exceptions( list )
- text_explorer_object << Add Stem Overrides( list )
// 常に語幹抽出する単語のリストを追加する
- text_explorer_object << Add Stop Word Exceptions( list )
// ストップワードのリストから削除する単語のリストを指定する
- text_explorer_object << Add Phrase Exceptions( list )
// 削除する句のリストに追加する
- text_explorer_object << Add Recodes( list )
// 再コード化する単語のペアのリストを追加する
- text_explorer_object << Add Recode Exceptions( list )
// 削除すべき、再コード化されたテキスト文字列のリストを追加する
- text_explorer_object << Terms Alphabetical( Boolean )
- text_explorer_object << Phrases Alphabetical( Boolean )
- text_explorer_object << Cloud Width( pixels )
// ワードクラウドの幅をピクセル単位で設定する
```

Time Series（時系列分析）

次の Time Series() 関数の引数は、JSL でのみ使用できます。

```
- Autocorrelation Lags( n )
// 自己相関を計算するための最大期間数を設定する。これは起動時のオプション
- Forecast Periods( n )
// 予測レポートで何期先を予測するかを設定する。これは起動時のオプション
```

Time Series オブジェクトに対する次の JSL メッセージは、JSL でのみ使用できます。

```
- time_series_object << Maximum Iterations( n )
// ARIMA モデルのあてはめで使用する、最適化の反復最大回数をリセットする
- time_series_object << Get Models
// モデルの記述を名前とした、モデル結果のリストを戻す
- time_series_object << Get Model Specs
// モデルの指定を名前とした、モデルの結果のリストを戻す
```

変動性図

Variability Chart オブジェクトに対する次のメッセージは、JSL でのみ使用できます。

```
variability_object << Show Box Plot Whisker Bars( Boolean );
```

第 11 章

表示ツリー
ウィンドウの作成と操作

スクリプトの基礎を学習したら、次はJMPレポートを操作したり、新しいJMPウィンドウを作成し、ウィンドウとの情報のやりとり行う方法を身につけましょう。この章では、次のようなトピックを扱います。

- レポートウィンドウ内の項目を操作する
- ビルトインウィンドウにアクセスする
- 独自の結果を含む新しいウィンドウのディスプレイツリーを作成する
- モーダルウィンドウを理解する
- 以前の `Dialog()` スクリプトを `New Window()` スクリプトに変換する

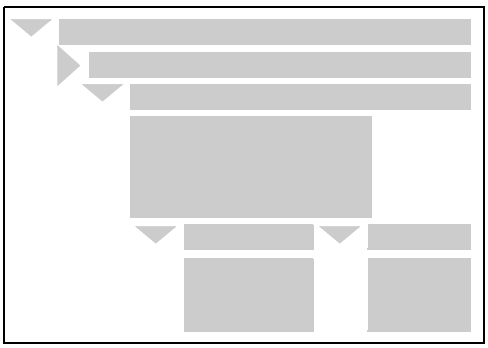
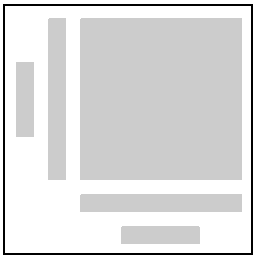
メモ: この章では、よく使用されるディスプレイボックスを取り上げています。その他のディスプレイボックスについては、［スクリプトの索引］を参照してください。

JMPのレポートの操作

JMP のレポートは、さまざまな種類の長方形のボックス（**ディスプレイボックス**）を入れ子にしたり、並べたりすることにより、階層的な形で作成されます。この節では、JMP のレポートについて概要を紹介した後で、レポートを操作する方法、レポートからデータを抽出する方法、レポートをカスタマイズする方法を解説します。

表11.1は、JMP レポートで頻繁に使用される3つのディスプレイボックスを示します。

表11.1 JMPのレポートの代表的なディスプレイボックス

アウトラインボックス（Outline Box）では、アウトラインによる階層が作成されます。	
ピクチャーボックス（Picture Box）では、軸、フレーム、およびラベルを組み合わせてグラフが構成されます。	
テーブルボックス（Table Box）は特殊な横方向ボックス（H List Box）で、文字列および数値列から構成されます。	

ディスプレイボックスの階層的な（親子）関係により、相対的なサイズが決まります。最も外側にあるディスプレイボックスが親で、内側にあるディスプレイボックスが子です。レポートを表示する際、親ディスプレイボックスは、まず最後の子ボックス（最下位）のサイズを検出し、下位から順に次の子ボックスのサイズを検出します。ツリーの上位へと進みながらすべてのボックスのサイズを検出します。その後、ツリーの最上位から順にボックスの位置を決め、すべてのボックスを適切に配置します。

ディスプレイボックス間の階層的な（親子）関係は、ディスプレイボックスのツリー（ディスプレイツリー）を形成します。他の（子）ボックスを内包する（親）ボックスの場合、親ボックスのサイズは、何らかの固定した値ではなく子ボックスのサイズによって決まり、それに応じて全体の表示が調整されます。同様に、ボックスの位置は、その前にあるボックスの位置によって決まります。ツリー用語で言うと、サイズはボトムアップ式、配置はトップダウン式に決まります。

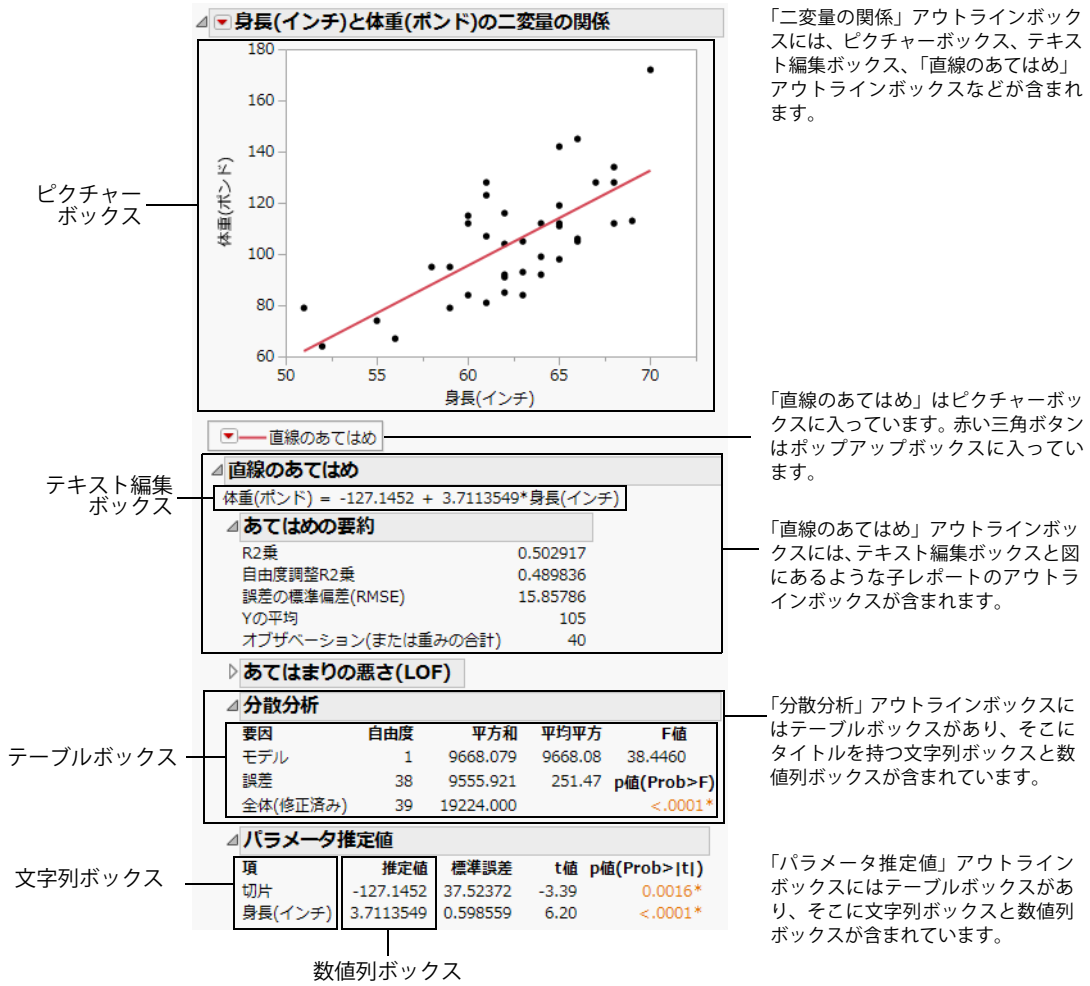
互換性に関する警告

JMPのプラットフォームは、全般に安定していて、リリースごとに大きな変化はありません。しかし、時にはメジャーバージョンが上がる際にレポート構造の変更を余儀なくされることがあります。そのような場合、構造が新しくなったことが原因で以前のスクリプトが正常に動作しない可能性もあります。そのようなスクリプトは、変更しなければなりません。スクリプトを記述する際、構造への依存をできるだけ小さくするのが賢明です。

よく使用されるディスプレイボックスの例

図11.1は、よく使用されるディスプレイボックスで構成された「二変量」のレポートです。

図11.1 レポートのディスプレイボックス



このレポートを作成するには、次のスクリプトを実行します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
biv = dt << Bivariate( Y( :Name(" 体重 ( ポンド )") ), X(:Name(" 身長 ( インチ )") ), Fit
   Line() );
```

ディスプレイツリーの表示

「ツリー構造の表示」ウィンドウでは、レポートのディスプレイツリーの構造をアウトライン形式で確認することができます。構造を表示し、ディスプレイボックスがディスプレイボックスツリー内にどのように配列されるかを見てみましょう。

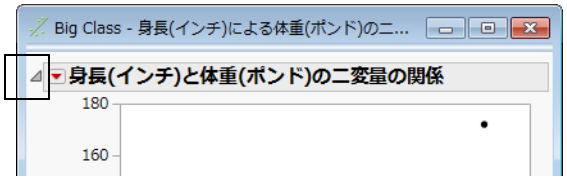
構造を表示するには、次の手順に従います。

1. 次のスクリプトを実行します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
biv = dt << Bivariate(
   Y( :weight ),
   X( :height ),
   Fit Mean( {Line Color( {57, 177, 67} )} )
);
```

2. 一番上の開閉アイコンを右クリックし、［**編集**］>［**ツリー構造の表示**］を選択します。

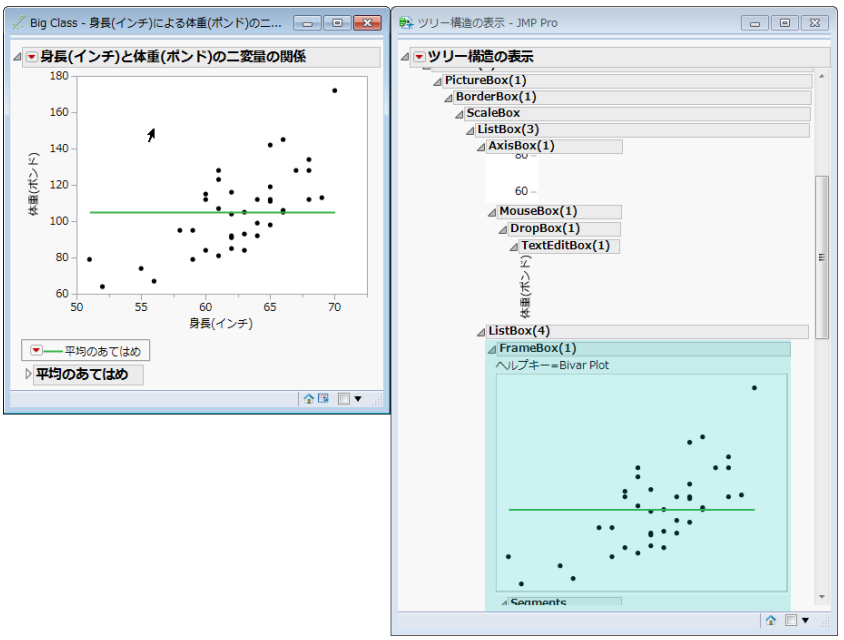
図11.2 一番上の開閉アイコン



「ツリー構造の表示」ウィンドウが開き、アウトラインボックスの入れ子の形でレポートのディスプレイボックスの構造が表示されます。それぞれのアウトラインボックスは、元のレポートのディスプレイボックスに相当します。

3. 「二変量」のグラフをクリックします。図 11.3 に示すように、グラフを含むフレームボックスが強調表示されます。

図11.3 ツリー構造の表示

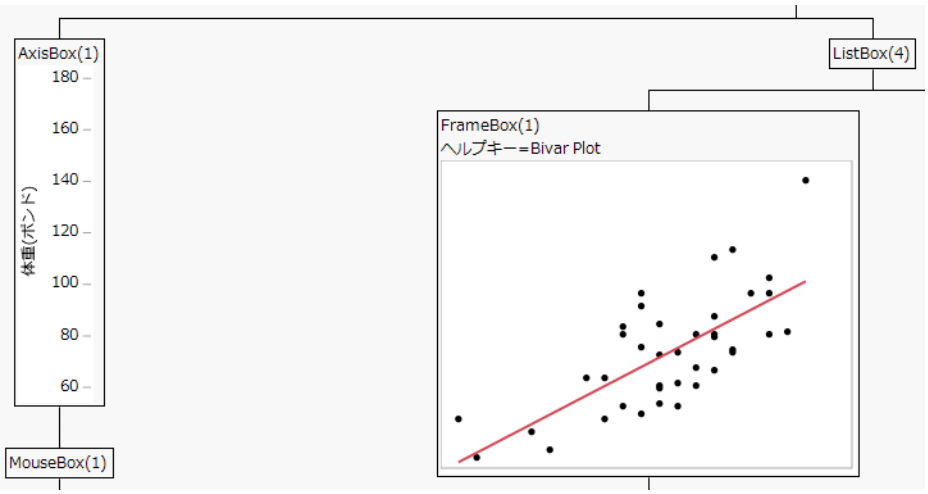


スクリプトによりツリー構造を表示させることもできます。それには、任意のレポートに Show Tree Structure() メッセージを送ります。特定の部分のツリー構造を表示する場合は、このメッセージをレポートの該当部分（ディスプレイボックスオブジェクト）に送ります。

アウトライン形式ではない、フラットな形式のディスプレイボックスツリーもあります。この従来型のディスプレイツリーを表示するには、Shift キーを押しながら開閉アイコンを右クリックし、［**編集**］>［**ツリー構造の表示**］を選択してください。新しいツリー構造の表示と従来型の表示には、次のような違いがあります。

- 従来型のツリー構造は、元のレポートにリンクしておらず、ツリーの特定の部分を強調表示することができません。
- 従来型のツリー構造では兄弟（同レベルの子）が左右に並べられます。新しいツリー構造では、兄弟が上下に並べられます。

図11.4 従来型のツリー構造（一部）

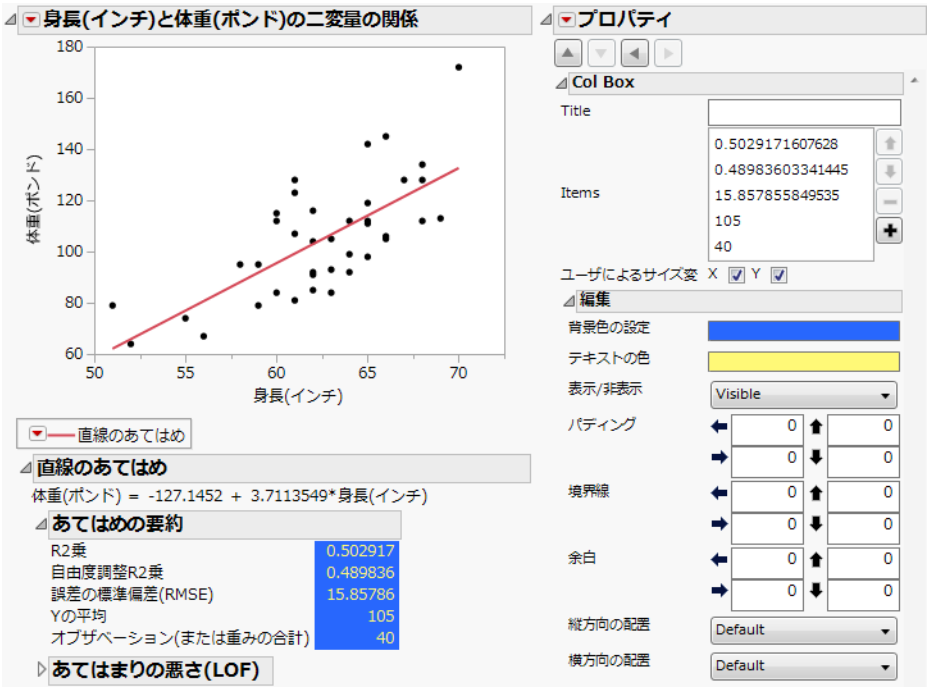


ディスプレイボックスのプロパティの表示

「ツリー構造の表示」ウィンドウでは、色や位置といったディスプレイボックスのプロパティを編集できます。この機能により、レポートをスクリプトに保存する前に、表示の設定をすることができます。「ツリー構造の表示」ウィンドウで、赤い三角ボタンのメニューから［**プロパティの表示**］を選択し、編集したいノードをクリックします。

図11.5では、グラフの下の「あてはめの要約」の統計量が選択され、「プロパティ」で背景色とテキストの色が変更されました。これらの変更は、レポートをスクリプトまたはジャーナルとして保存した場合も維持されます。

図11.5 プロットの［プロパティの表示］の例



ディスプレイボックスオブジェクトの参照

結果は、プラットフォームを起動すると、結果は2種類のオブジェクトになります。1つは、ディスプレイボックスで構成されたレポートであり、もう1つは、プラットフォームと呼ばれ、コマンド（またはメッセージ）を受け入れる非表示のオブジェクトです。

プラットフォームに送るメッセージの1つに、Reportメッセージがあります。Reportメッセージは、プラットフォームから、レポート内のディスプレイボックスの参照を作成します。

レポートの冒頭にある赤い三角ボタンのメニューは、プラットフォームオブジェクトにつながっており、このメニューを使うと対話式にメッセージをプラットフォームに送れます。

次の例は、「二変量」プラットフォームで平均をあてはめるには、プラットフォームに参照を割り当てています。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
biv = dt << Bivariate( // bivは分析レイヤーを参照
   X( :Name("身長(インチ)") ),
   Y( :Name("体重(ポンド)") ),
   Fit Mean,
   Fit Polynomial( 4 )
);
```

結果を表示しているディスプレイボックスへの参照を取得するには、Reportメッセージを使用します。たとえば、先ほどのスクリプトに次のような式を追加します。

```
rbiv = biv << Report; // rbivはレポート層を参照
```

この後、以下の節の説明に従い、レポートレイヤーのディスプレイボックスを操作することができます。

ディスプレイボックスでできることを調べる

次の例では、「二変量」レポートを作成し、その参照を rbiv レポートオブジェクトに格納します。その後、rbiv レポートオブジェクトのフレームボックスへの参照を、fbx 変数に格納します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
biv = dt << Bivariate( // bivは分析レイヤーを参照
   Y( :weight ),
   X( :height ),
   Fit Mean( {Line Color( {57, 177, 67} )} )
);
rbiv = biv << Report; // レポートレイヤーを rbiv に割り当てる
fbx = rbiv[Frame Box( 1 )];
```

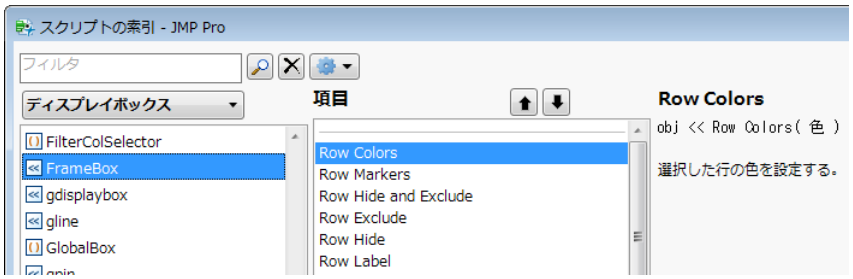
fbxが定義されたら、それにメッセージを送ることができます。

```
fbx << Set Background Color( blue ); // フレームボックスを青色にする
```

Set Background Colorメッセージは、グラフ内を右クリックして［**背景色の設定**］>［**青**］を選択する作業に相当します。

フレームボックスに送ることのできるメッセージの種類を調べるには、［**ヘルプ**］>［**スクリプトの索引**］を選択し、ディスプレイボックスのリストから［FrameBox］を検索します（図11.6）。

図11.6「スクリプトの索引」のディスプレイボックスリスト



「スクリプトの索引」の代わりに Show Properties() 関数を使用する方法もあります。その場合、使用できるメッセージはログに出力されます。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
biv = dt << Bivariate( Y(:Name("体重(ポンド)") ), X(:Name("身長(インチ)") ) );
Show Properties( biv );
  Show Points [ブール] [デフォルトはオン]
```

```
Histogram Borders [ブール]
Fit Mean [アクション](Fits a flat line at the mean.)
...
```

どのオブジェクトの場合も、メッセージの多くは、ポップアップメニューの項目と一致します。主要なディスプレイボックスへのメッセージについては、次の節の「独自のウィンドウの作成」（432ページ）で、さらに詳しく説明します。

添え字の使用

添え字を使うと、レポート内の特定のディスプレイボックスを参照できます。ディスプレイボックスは、番号を指定して参照できます。次の式は、ディスプレイツリー内で 2 番目に位置するアウトラインボックスを指定しています。

```
var = Outline Box[2];
```

レポートを操作するもう一つの方法は、文字列の検索することです。次の式は、特定の文字列を含むレポートを指定しています。

```
var = report[text];
```

この式は、レポートの中で *text* というタイトルのディスプレイボックスを検索します。タイトル（*text*）には、タイトルの一部ではなく完全な文字列を指定する必要があります。"*text*" の部分には、文字列を含む任意の式を指定することもできます。この方法は、添え字を使ってディスプレイボックスを参照する方法に比べると確実性が低くなります。なぜなら、レポート内の文字列は頻繁に変更されるためです。

通常、メッセージを簡単に送れるように、レポートの一部を変数に割り当てます。たとえば、「二変量」レポートの「分散分析」アウトラインボックスを見つけるには、次の式を使用します。

```
var = rbiv[" 分散分析 "];
```

以下の節で、添え字演算子を使ってディスプレイボックスを参照するその他の方法について説明します。

ワイルドカード文字列

文字列の一部とワイルドカード文字（? など）を使って、アウトラインタイトルの残りの部分と一致させることができます。指定する文字列の中には、どんな文字の代わりにもなるワイルドカード文字「?」を使用できます。次のスクリプトは、「平均」で始まる文字列を探し、「平均のあてはめ」を見つけ、平均のあてはめのアウトラインを開きます。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
biv = dt << Bivariate(
   Y( :weight ),
   X( :height ),
   Fit Mean( {Line Color( {57, 177, 67} )} ),
   Fit Polynomial( 4 )
);
rbiv = biv << Report; // 「二変量」レポートへの参照を戻す
out1 = rbiv["? 平均 "]; // 「二変量」レポート内の「平均のあてはめ」を見つける
out1 << Close( 0 ); // 「平均のあてはめ」のアウトラインを開く
```

ボックスタイプとワイルドカード文字列

ディスプレイボックスを見つけるもう一つの方法は、添え字演算子にディスプレイボックスとワイルドカード文字列を含めることです。

```
var = rbiv[Display Box( "?text" )];
```

この式は、rbivレポート内で指定の文字列を含むアウトラインボックスを見つけます。out1変数は、アウトラインボックスに参照を割り当てます。

```
out1 = rbiv[Outline Box( " 平均?" )];
```

ボックスタイプとインデックス

ディスプレイボックスを見つける別の方法としては、*boxType*でインデックス番号*n*を指定する方法があります。

```
rpt[Display Box ( n )];
```

上の式は、タイプが*boxType*である指定のディスプレイボックスのうち、*n*番目のものを見つけます。たとえば、この式は、rbivレポート内の最初のアウトラインボックスを見つけ、それにout1変数を割り当てます。

```
out1 = rbiv[Picture Box( 1 )];
```

ヒント： ディスプレイボックスのインデックス番号を特定するには、「入れ子と優先順位」（423ページ）で説明している方法でツリー構造を表示します。

入れ子になったディスプレイボックス

レポート内で階層の下のほうに表示されるディスプレイボックスを参照するには、この形式を使用できます。

```
var = rpt[arg1][arg2][arg3][...];
```

上の式は、最後の引数を、最後から2番目の添え字のアウトラインノードに**含まれている**ディスプレイボックスにマッチし、これは、アウトラインツリーの階層を徐々に下がりながら入れ子になっているディスプレイボックスを探す方法です。

アウトラインボックス内で複数の層になっている項目を識別するには、前述の方法において、複数の添え字を指定します。JMPは、最初の項目を見つけた後、**その項目内で**次の項目を順次探していきます。

```
rpt[arg1][arg2][arg3] // 最初の項目を見つけ、その後、その位置から始めていく
rpt[arg1, arg2, arg3] // 構文は異なるが、結果は同じ
```

テキストを含む添え字とインデックス番号を含む添え字を使用することができます。下の式は、「多項式のあてはめ次数=4」のアウトラインノード内にある「パラメータ推定値」を見つけ、最初のテーブルの3番目のディスプレイボックスを選択します。

```
out = rbiv[" 多項式のあてはめ次数 =4"][" パラメータ推定値 "][1][3] << Select;
```

「レポートの作成例」（429ページ）は、レポートからパラメータ推定値を抽出する方法を示します。

ディスプレイボックスにメッセージを送る

表示の参照は、Sendまたは<<演算子を使ってメッセージを表示要素に送るときに使用できます。たとえば、out2がアウトラインノードの参照である場合、開いているときは閉じるよう、out2自体に要求することができます。

```
out2 << Close(); // アウトラインノードを閉じる
```

Close()は、赤い三角ボタンのメニューで項目を選択する場合と同様、アウトラインノードの開閉を交互に切り替えます。「閉じるまたは閉じたまま」にするには1、「開くまたは開いたまま」にするには0を指定します。

```
rbiv[" 平均のあてはめ "] << Close;  // 開閉を交互に切り替える
rbiv[" 平均のあてはめ "] << Close( 1 ); // 閉じるまたは閉じたままにする
rbiv[" 平均のあてはめ "] << Close( 0 ); // 開くまたは開いたままにする
```

ヒント：［一般］環境設定の［無効なディスプレイボックスメッセージを報告］オプションは、無効なディスプレイボックスメッセージを特定するのに役立ちます。

- このオプションは、デフォルトではオフになっています。ディスプレイボックスが無効なメッセージを受け取ると、JMPは、エラーを無視し、スクリプトの実行を継続します。
- この環境設定がオンの場合、ログにエラーが表示されます。

このオプションは、スクリプトの開発中は役立ちますが、実際に使用するにあたっては、無効なメッセージが使われたかどうかは不要な情報であることが多いです。

<<演算子

send（<<）演算子を使ってディスプレイボックスにメッセージを送ることができます。この演算子を使うと、指定されているものが、それに含まれる子オブジェクトなのか、それともオプションなのかを明確に区別することができます。また、グラフ内において、どれがオプションで、どれが実行されるスクリプトであるのかも明確にできます。

次の例では、グラフボックスにメッセージを送ります。

```
win = New Window( " メッセージ ",
   gb = Graph Box(
      Frame Size( 400, 400 ),
      X Scale( 0, 25 ),
      Y Scale( 0, 25 )
   )
);
gb << Background Color( "red" );
```

入れ子と優先順位

メッセージを組み合わせる（または**入れ子**にする）と、左から右へとメッセージが評価されます。

- 次の式は、message1をboxへ、次にmessage2をboxへ、最後にmessage3をboxへ送ります。次のメッセージが送られる時点で、前のメッセージによりboxが変更されている可能性があります。

```
box << message1 << message2 << message3;
```

- 次の式は、message1をboxへ送り、結果を取得します。その結果にmessage2が送られ、message2の結果にmessage3が送られます。

```
( (box << message1) << message2) << message3
```

- message3の結果が変数xに割り当てられます。

```
x = box << message1 << message2 << message3;
```

次に、入れ子にしたメッセージを使った例を紹介します。

```
win = New Window( "メッセージ", gb = Graph Box() );
sz = gb << Background Color( "red" )
// message 1は背景色を設定する

<<Save Picture( "$DOCUMENTS/red.png", "png" )
// message 2はグラフボックスをPNGファイルとして保存する

<<Background Color( "white" )
// message 3はグラフボックスの背景色を白に設定する

<<Get Size();
// message 4はグラフボックスのサイズを戻し、ログに出力する
```

複数のメッセージを入れ子にする場合は、スクリプトが意図したとおりに実行されるように、括弧を使ってメッセージをグループ化します。

Send to Report関数とDispatch関数を使ってレポートをカスタマイズする

Send To Report()とDispatch()の各関数は、ディスプレイボックスの編集内容を分析スクリプトの中に埋め込んで記録します。たとえば、2つの関数に含まれる引数で、アウトラインノードの開閉、グラフフレームのサイズ変更、グラフフレーム内の色のカスタマイズなどを実行できます。JSLステートメントで引数を指定することもできます。

Send To Report()は、ディスプレイツリーを変更するオプションのリストを含みます。

Dispatch()は、ディスプレイツリーを変更する4つの引数を含みます。

- 第1引数は、表示ツリーの目的の部分を見つけるために通過する必要があるアウトラインノードのリストです。
- 第2引数と第3引数はペアで機能します。第2引数は表示要素の名前、第3引数は表示要素のタイプです。これら2つの引数で、カスタマイズの対象とする表示ツリーの特定の部分を指定します。
- 第4引数はオプションのリストです。

たとえば、Big Class.jmpを開き、そこに保存されている「二変量の関係(二変量)」スクリプトを実行します。これにより、あてはめた直線のレポートが生成されます。「あてはまりの悪さ(LOF)」アウトラインノードを開き、「分散分析」アウトラインノードを閉じます。[**スクリプトの保存**]>[**スクリプトウィンドウへ**]を選択します。スクリプトエディタウィンドウに次のスクリプトが表示されます。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
biv = dt << Bivariate(
   Y( :weight ),
   X( :height ),
   Fit Line(),
   SendToReport(
      Dispatch(
         {"Linear Fit"},
         "Lack Of Fit",
         OutlineBox,
         {Close( 0 )} ),
      Dispatch(
         {"Linear Fit"},
         "Analysis of Variance",
         OutlineBox,
         {Close( 1 )}
      )
   )
);
```

Send To Report() 関数は、デフォルトのレポートをカスタマイズする 2 つの Dispatch() 関数を含みます。

- 最初の引数は、「Linear Fit」（直線のあてはめ）という名前のアウトラインノードを見つけます。
- 第2、第3の引数は、「Linear Fit」（直線のあてはめ）アウトラインの下にある「Lack of Fit」（あてはまりの悪さ (LOF)）という名前のアウトラインボックスを見つけます。
- 第 4 引数は、このアウトラインボックスに送る引数です。この例では、メッセージは Close(0)、つまりノードを開くコマンドです。

メモ: 同じ名前のアウトラインノードが複数ある場合は、添え字が割り当てられます。たとえば、「二変量の関係」の分析に2つの2次のあてはめがある場合（同じタイトルになる）、2番目のあてはめにコマンドを適用すると、重複するタイトルに添え字 [2] が追加された Dispatch コマンドが生成されます。

Send to Report() と Dispatch() を効果的に使用するには、まず、対話式にレポートをカスタマイズします。レポートをスクリプトとして保存し、JMP によって生成されたそのスクリプトを確認します。JMP 自身が、実は最高の JSL プログラマーなのです。

ジャーナルを作成する

ジャーナルを作成すると、ある時点における JMP ウィンドウのコンテンツをキャプチャすることができます。JMP ウィンドウにグラフが表示され、チェックボックスの一部がオンになっているとしましょう。そこからジャーナルを作成すると、ジャーナルには、そのグラフと、その時点でオンになっていたチェックボックスが表示されます。

次の例で、ジャーナルを使ってどのようなことができるかを示します。

「二変量」レポートから始めてみましょう。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
biv = dt << Bivariate( Y( :Name("体重(ポンド)")), X( :Name("身長(インチ)")));
rbiv = biv << Report;
```

結果をジャーナルにまとめます。

```
rbiv << Journal Window;
```

次のように指定し、ジャーナルをファイルに保存します。

```
rbiv << Save Journal(
   "$DOCUMENTS/test.jrn"
);
```

上の例では、Macintoshのファイルパス規則を使用しています。Windowsでは、スラッシュ（全プラットフォームに有効）またはバックスラッシュ（円マーク、Windowsのみに有効）を使用します。

スクリプトにBy変数を使用した場合、各Byグループの分析結果への参照が含まれたリストが戻り値になります。すべてのByグループのレポートをジャーナルにまとめるには、parentメッセージを使用してレポートの先頭に移動する必要があります。

たとえば、次のスクリプトは、By グループを使用した二変量のレポートを作成し、レポート全体のジャーナルを作成します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
biv = dt << Bivariate(
   Y( :Name("体重(ポンド)") ),
   X( :Name("身長(インチ)") ),
   By( :性別 )
);
( ( Report( biv[1] ) << Parent ) << Parent ) <<
Save Journal( "test.jrn" );
```

よく使用されるメッセージ

以下の節では、よく使用されるメッセージを用いて次のような処理を実行する方法を説明します。

- 「数値列を更新する」（426ページ）
- 「文字列の列を更新する」（427ページ）
- 「テキストを折り返す」（427ページ）
- 「テキストボックスを箇条書きにする」（427ページ）
- 「選択されたディスプレイボックスを反転表示する」（428ページ）
- 「テーブルの見出しに網掛けや枠線を追加する」（429ページ）

数値列を更新する

ディスプレイボックステーブルの数値列を更新するには、Set Valuesメッセージを使います。

```
win = New Window( "数値列",
   num = Number Col Box( "値", [9, 10, 11] )
);
num << Set Values( [1, 2, 3] );
```

行列の引数が、テーブルの新しい数値を指定しています。

数値列に桁区切りを追加する

数値列に桁区切りを追加するには、Set Formatメッセージを使用します。

```
New Window( "例",
   ncb = Number Col Box( "乱数",
      {Random Uniform(), Random Uniform() * 10,
      Random Uniform() * 100, Random Uniform() * 1000,
      Random Uniform() * 10000}
   )
);
ncb << Set Format(10.3, "Fixed Dec", "Use thousands separator");
```

文字列の列を更新する

ディスプレイボックステーブルの文字列の列を更新するには、Set Valuesメッセージを使います。

```
win = New Window( "文字列",
   str = String Col Box( "値", {"a", "b", "c"} )
);
str << Set Values( {"A", "B", "C"} );
```

リストの引数が、テーブルの新しい文字列を指定しています。

テキストを折り返す

通常、Text Box()内ではテキストは自動的に改行されます。ただし、Set Wrap (*n*)メッセージを使うと、デフォルトの改行ポイントを上書きすることができます。次のスクリプトは、200ピクセルの地点でテキストを折り返します。

```
win = New Window( "テキストの折り返し幅",
   tb = Text Box(
      "データ点にカーソルを置くと情報が表示されます。"
   )
);
tb << Set Wrap( 200 );
```

テキストボックスを箇条書きにする

箇条書きにするには、Bullet Point(1)メッセージを使います。

```
win = New Window( "箇条書きのリスト",
   text1 = Text Box( "データ点にカーソルを置くと情報が表示されます。" ),
```

```
   text2 = Text Box( "統計量の上でカーソルを丸く動かすと情報が表示されます。" )
);
text1 << Bullet Point( 1 );
text2 << Bullet Point( 1 );
```

Bullet Point(1)メッセージをテキストボックスに送ると、テキストの前に中黒が挿入され、テキストボックス内の後続の行はインデントされます。

Bullet Point(1)メッセージは、次の例のようにインラインで使用することもできます。

```
text1 = Text Box( "データ点にカーソルを置くと情報が表示されます。" ), << Bullet Point( 1
  ) ),
```

選択されたディスプレイボックスを反転表示する

Select、Reshow、およびDeselectを使うと、ディスプレイボックス上で選択を示す強調表示を点滅させたり解除させたりすることができます。前述のように、複数の<<節を一緒に並べて、1つのオブジェクトに複数のメッセージを1行で送ることができます。結果は左から右に読んでいきます。たとえば、次の例の場合、まず、Selectが処理され、次にReshow、その次にDeselect、その次にもう一つのReshowが処理されます。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
biv = dt << Bivariate(
   Y( :weight ),
   X( :height ),
   Fit Mean( )
);
rbiv = biv << Report;
// 二変量の分析レイヤーを rbiv に割り当てる

out2 = rbiv[Outline Box( "平均 ?" )];
out2 << Close( 0 ); // アウトラインボックスを開く
// rbiv レポート内で指定の文字列を含む
// アウトラインボックスを見つけ、それを out2 変数に割り当てる

scbx = rbiv[String Col Box( 1 )];
// rbiv の最初の文字列ボックスを見つけ、
// scbx 変数に割り当てる

Wait( .25 ); // 0.25 秒待機する
For( i = 0, i < 20, i++,
// i を 0 に設定し、20 回ループする

   scbx << Select << Reshow << Deselect << Reshow
   // 文字列ボックスを選択して再表示し、
   // 選択を解除して再表示する
);
```

テーブルの見出しに網掛けや枠線を追加する

見出しは、デフォルトで網掛けと枠線がついています。これらをオフにするには、Set Shade Headings (0) メッセージとSet Heading Column Borders (0)メッセージを使用します。次の例は、網掛けと枠線をそれぞれオンからオフにしたときの違いを示しています。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
Summarize(
   meanHt = Mean( Name("身長(インチ)")),
   minHt = Min( Name("身長(インチ)") ),
   maxHt = Max( Name("身長(インチ)") ),
);
win = New Window( "結果の要約",
   tb = Table Box(
      Number Col Box( "平均", meanHt ),
      Number Col Box( "最小", minHt ),
      Number Col Box( "最大", maxHt )
   )
);
Wait( 2 );
tb << Set Shade Headings( 0 );
Wait( 1 );
tb << Set Heading Column Borders( 0 );
Wait( 2 );
tb << Set Shade Headings( 1 );
Wait( 1 );
```

レポートの作成例

ここでは、レポートの作成を開始し、完了させるスクリプトを紹介します。まず、データテーブルを開きます。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
```

ここで、二変量プラットフォームを起動し、その参照を「biv」に割り当てます。

```
biv = dt << Bivariate( Y( :Name("体重(ポンド)")), X( :Name("身長(インチ)")) );
// プラットフォームの参照
```

「二変量」プラットフォームで何ができるのかを調べるには、［ヘルプ］>［スクリプトの索引］を選択し、オブジェクトのリストから「Bivariate」を検索します。いくつかのオプションを紹介します。

```
biv << Fit Spline( 1000000 ) << Fit Mean;
biv << Fit Polynomial( 4, RGBColor( 1, 0.5, 0) ); // 次数4、オレンジ色の曲線
biv << ( Curve[1] << Line Color( red ) ); // 最初の曲線は赤色
```

レポートレイヤーをrbivに割り当てます。

```
rbiv = biv << Report;
```

次に、フレームボックスを見やすいサイズにし、グラフを表示するために先頭までスクロールします。

```
rbiv[Frame Box( 1 )] << Size Window( 500, 700 );
biv << Scroll Window( {0, 0} );
```

ここから、レポートの処理を始めます。まず、参照を作成し、次に、その参照でできることを調べる必要があります。

```
rbiv = biv << Report; // レポートへの参照
Show Properties( rbiv ); // レポートのプロパティを表示する
```

ログに、レポートに使用できるメッセージがリストされます。スクリプトの索引の代わりに、Show Properties()を使用することができます。

```
Close [ブール]
GetOpen [アクション] [スクリプトの場合のみ]
SetOpen [ブール] [スクリプトの場合のみ]
...
```

アウトラインの「平均のあてはめ」ノードを開きます。

```
rbiv[" 平均のあてはめ "] << Close( 0 );
```

Close()は、ブール関数（真または偽）で、0を渡すとアウトラインは閉じません。

実際に、いくつかの結果を選択してみます（個別の結果を見るには、各行を単独で実行）。

```
rbiv[" あてはめの要約 "] << Select;
rbiv[" パラメータ推定値 "] << Select;
rbiv[" 分散分析 "] << Select;
rbiv[" 多項式のあてはめ 次数 =2", " パラメータ ?", Column Box( " 推定値 " )] << Select;
// アウトラインツリーの下位層にある要素を選択
rbiv << Deselect;
```

2番目の分散分析項目を選択するには、次のようにします。

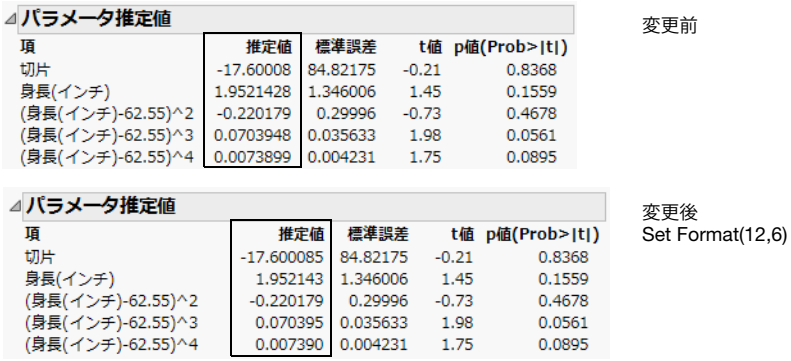
```
rbiv[" 多項式のあてはめ 次数 =2", " 分散分析 "] << Select;
```

ここで、4次多項式のパラメータ推定値レポートにおいて、推定値列の表示形式を変更してみましょう。

```
pe = rbiv[" 多項式のあてはめ 次数 =4", " パラメータ ?"];
ests = pe[Number Col Box( " 推定値 " )];
ests << Set Format( 12, 6 );
```

Set Formatの第1引数では、全体の列幅を表示する文字数で指定します。第2引数では小数点以下の桁数を指定します。

図11.7 レポートに変更を適用



パラメータ推定値

項	推定値	標準誤差	t値	p値(Prob>|t|)
切片	-17.60008	84.82175	-0.21	0.8368
身長(インチ)	1.9521428	1.346006	1.45	0.1559
(身長(インチ)-62.55)^2	-0.220179	0.29996	-0.73	0.4678
(身長(インチ)-62.55)^3	0.0703948	0.035633	1.98	0.0561
(身長(インチ)-62.55)^4	0.0073899	0.004231	1.75	0.0895

変更前

パラメータ推定値

項	推定値	標準誤差	t値	p値(Prob>|t|)
切片	-17.600085	84.82175	-0.21	0.8368
身長(インチ)	1.952143	1.346006	1.45	0.1559
(身長(インチ)-62.55)^2	-0.220179	0.29996	-0.73	0.4678
(身長(インチ)-62.55)^3	0.070395	0.035633	1.98	0.0561
(身長(インチ)-62.55)^4	0.007390	0.004231	1.75	0.0895

変更後
Set Format(12,6)

テーブルから単一の数値を取得することもできます。たとえば3次の項の推定値を取得するには、次のように記述します。

```
terms = pe[String Col Box( "項" )]; // 項の列を見つける
estimate = .; // 3次の項がない場合にそなえて初期値を設定
Try(
    For( i = 1, i < 10, i++,
        If( Contains( terms << Get( i ), "^3" ),
         // get(i) は、Try によって処理されるテーブルの末尾で停止する
            estimate = ests << Get( i );
            Break();
        )
    )
);
Show( estimate );
  0.070394822744608
```

For() による反復処理は、目的の項までの行数を数えるために使われています。For() の2番目の引数が条件であることを思い出してください。条件のテストが真である限り、ループは続きます。この場合は、「**項**の列が "^3" でなく、10番目の行に到達していない」という条件がテストされています。探している文字列が**見つかった**時点で（Break() により）ループは終了し、*i* の値は一致した行の番号になっています。これを利用して、For() による反復処理の後、推定値列に対する Get メッセージの引数として *i* を用いています。

また、ボックスから値を行列で取得し、その値を使って、次の計算を行ったり、データテーブルを作成したりすることもできます。次の例は、データテーブルの作成方法を示しています。

```
myVector = rbiv[Table Box( 4 )][Number Col Box( "平方和" )] << Get as Matrix;
// 平方和列の値を行列として取得する

dt << New Column( "平方和", Values( myVector ) );
// Big Class.jmp に「平方和」という新しい列を作成する
// 列に myVector の値を挿入する
```

```
rbiv[Table Box( 4 )] << Make Data Table( "分散分析表" );
// 4番目テーブルボックスの値を新しいデータテーブルに挿入する
```

ここで、軸のスケールを調整します。

```
rbiv[Axis Box( 1 )] << Min( 70 ) << Max( 170 ); // Y軸を設定する
rbiv[Axis Box( 2 )] << Min( 50 ) << Max( 70 );  // X軸を設定する
```

レポートの冒頭にあるグラフをコピーします。グラフが含まれているピクチャーボックスを選択する必要があります。グラフだけを選択すると、軸が欠落してしまいます。

```
rbiv[Picture Box( 1 )] << Copy Picture;
```

独自のウィンドウの作成

表示ツリーを構築する関数を使って、オブジェクトを作り、独自の表示を作成することができます。たとえば、ユーザに入力を促す、独自のレポートを作成する、独自の起動ウィンドウを作成する、ウィンドウにチェックボックスやラジオボタンを追加するなどの作業が可能です。

まずNew Window()を使い、引用符で囲ったタイトルと、そのウィンドウ内に配置するディスプレイボックスを指定します。変数に新しいウィンドウにを割り当てることで参照を作成し、メッセージを送れるようにします。表示オブジェクトの参照は、別のディスプレイボックスにコピーされるか、ウィンドウが閉じて表示されなくなるまで、自由に使用できます。表示オブジェクトを別の表示ツリーに入れる指定をすると、JMPは、そのコピーを作成し、所有者を新しいボックスにします。

ディスプレイボックスを構成する関数にはすべて、最後に「Box」が付いています。グラフィックフレームの内部には、ディスプレイセグメントと呼ばれる同様のオブジェクトが存在します（マーカー、ラインなど）。ディスプレイセグメントオブジェクトは、すべて末尾に「Seg」が付いています。

グラフボックスの作成例

次の例は、グラフボックスを含んだ新しいウィンドウを作成します。

```
win = New Window( "粗い衛星写真の地図",
   gb = Graph Box(
      X Scale( -180, -60 ),
      Y Scale( 20, 80 ), // グラフボックスパラメータの終わり
      <<Background Map( Images( "粗い衛星写真" ) ) // Grph Box のスクリプト
   )
);
```

パラメータは、X軸とY軸のスケールとフレームサイズを設定します。そして、Background Mapメッセージが地図を表示します。Graph Box()、X Scale、Y Scaleのパラメータが先にあり、それらの後ろにある最初のカンマの後にスクリプトを記述します。New Window()関数内のパラメータの途中にスクリプトを配置した場合、スクリプトは実行されても、期待する結果が出ません。また、新しいウィンドウの冒頭にパラメータがあった方が、ユーザによってスクリプトが読みやすいという利点もあります。

ウィンドウから値を取得する

ユーザがウィンドウでオプションを選択した後、Return Result メッセージ、Get メッセージ、Get Selected メッセージを使ってこれらの選択内容をウィンドウから取得することができます。

値の取得：方法1

ユーザが［OK］をクリックしたときに New Window() スクリプトによって自動的に結果が戻されるようにするには、Modal メッセージの後に Return Result メッセージを含めます。

```
win = New Window( " 値の指定 ",
   <<Modal,
   <<Return Result,
   Text Box(" ここに値を入力してください。"),
   variablebox = Number Edit Box( 42 ),
   Button Box( "OK"),
   Button Box( " キャンセル ")
);
Write( win["variablebox"] ); // variablebox 変数を添え字として指定
  33 // ユーザがNumber Edit Boxに「33」と入力した場合
```

値の取得：方法2

チェックボックスが選択されている場合は1、選択されていない場合は0を戻す Get メッセージを含めます。選択内容を表示するには、スクリプトの終わりに Show() 式を追加します。

```
win = New Window( "V List Box",
   <<Modal,
   V List Box(
      kb1 = Check Box( "a" ),
      kb2 = Check Box( "b" ),
      kb3 = Check Box( "c" )
   ),
   Button Box( "OK",
      val1 = kb1 << Get; // 最初のチェックボックスの値を取得する
      val2 = kb2 << Get;
      val3 = kb3 << Get;
   )
);
Show( val1, val2, val3 ); // ウィンドウが閉じた後に変数を戻す
  val1 = 1; // 最初と2番目のチェックボックスは選択されていた
  val2 = 1;
  val3 = 0; // 3番目のチェックボックスは選択されていなかった
```

値の取得：方法3

選択されている列の名前を戻す Get Selected メッセージを含め、その列を「二変量」プロットに挿入します。

メモ: 値を取得する方法2と方法3は、例は異なりますが、方法としては同一です。どちらの例でも、ダイアログ内にあるコントロールからのJSLコールバックを使用しています。JSLコールバックは、ダイアログ内のディスプレイボックスの値をグローバル変数に割り当てることで、ダイアログを閉じた後もその値が使用できるようにします。方法2では、ボタンボックスにJSLコールバックがあります。方法3では、リストボックスにJSLコールバックがあります。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
xvar = .;
yvar = .;
win = New Window( " 値を戻す例 ",
   <<Modal,
   <<On Validate(
   // ユーザに対し、[OK] をクリックする前に 2 つの変数を選択するよう求める
      Show( xvar, yvar );
         If( Is Missing( xvar ) | Is Missing( yvar ),
            0, // xvar または yvar がない場合は、[OK] がクリックされても何もしない
         1
      );
   ),
   Text Box( " 数値タイプの列をそれぞれ 1 つ選択してください。" ),
   H List Box(
      Text Box( "X, 説明変数 " ),
      x = Col List Box(
         dt, // データテーブル参照
         all, // データテーブルの列をすべて表示する
         xvar = (x << Get Selected)[1];
         // ウィンドウが閉じる前に選択されている列の名前を取得する
         Show( xvar );
      ),
      Text Box( "Y, 応答変数 " ),
      y = Col List Box(
         dt,
         all,
         yvar = (y << Get Selected)[1];
         Show( yvar );
      )
   )
);
xcol = Column( dt, xvar ); //  列を取得する
ycol = Column( dt, yvar );

dt << Bivariate( Y( ycol ), X( xcol ) ); // 「二変量」プロットを作成する
```

新しいウィンドウの作成

JSL では、モーダルと非モーダル両方のウィンドウを作成できます。

- モーダルウィンドウの場合、そのウィンドウを閉じないと他のウィンドウを使用することができません。モーダルウィンドウの外側をクリックするとエラー音が鳴り、ユーザがモーダルウィンドウを閉じるまで、ほぼすべてのスクリプトの実行が停止されます。
- JMP のレポートやプラットフォーム起動ウィンドウは、非モーダルウィンドウです。非モーダルウィンドウの場合、同時に複数のウィンドウを開いておき、自由に切り替えることができます。モーダルウィンドウの例としては、グラフ内の「透明度」ダイアログが挙げられます。「透明度」ウィンドウを開いたら、それを閉じないと次に進むことができません。

この節では、ウィンドウ内で作成できる主要なディスプレイボックスを紹介します。JMP には、JSL で作成できないディスプレイボックスが多数あります（軸ボックスなど）。作成できないディスプレイボックスは、JMP のプラットフォームによって使用されます。JSL で作成できないディスプレイボックスにも、JSL で使用できるメッセージがあります。

ヒント： ［スクリプトの索引］（［**ヘルプ**］ > ［**スクリプトの索引**］）では、作成できないディスプレイボックスに « というマークが付いています。作成できるディスプレイボックスには {} のマークが付いています。

メモ：

- モーダルウィンドウを作成する `Dialog()` 関数は廃止されるため、今後のバージョンでは動作しなくなる可能性があります。`Modal` メッセージとともに `New Window()` を使用するか、または列を選択するウィンドウの場合は `Column Dialog()` を使用してください。詳細については、「列ダイアログの作成」（498 ページ）と「廃止される Dialog を New Window に変換する」（505 ページ）を参照してください。
- JMP のディスプレイボックスのうち、`Border Box()`、`Center Box()`、`If Box()`、`Mouse Box()`、`Scroll Box()`、`Sheet Panel Box()` には、直接の子ボックスを 1 つだけ持たせることができます。複数の子を持つ Border Box を作成するには、唯一の子として `V List Box()` または `H List Box()` を使用し、`V List Box()` または `H List Box()` に複数の子を含めます。

コンテナディスプレイボックス

コンテナディスプレイボックスは、子のディスプレイボックスを囲むものです。

Border Box

`Border Box()` は、*displaybox* 引数の周りにスペースを追加します。

```
Border Box (Left( pix ), Right( pix ), Top( pix ), Bottom( pix ), Sides( int ),
  display box args);
```

`Left`（左）、`Right`（右）、`Top`（上）、および `Bottom`（下）で、*displaybox* 引数のどこにどれだけのスペースを追加するかを指定します。`Sides` は、表 11.2 にあるように、ボックスの周りに枠を描きます。`Sides` では、その他の効果も適用できます（表 11.2 を参照）。効果と枠の両方を追加するには、2 つの数値を足します。

たとえば、次の例は、上下に枠のあるテキストボックス（枠の位置の値は5）を作成し、その枠の色を赤に設定します（効果の値は6）。

```
win = New Window( "Border Box の例 ",
   bb = Border Box( Sides( 5 ), // 上下の枠
      Text Box( " 世界のみなさん、こんにちは " )
   )
);
bb << Set Color( " 赤 " ); // 枠の色を赤に設定する
```

Border Box は bb 変数に割り当てられているため、最後の行で色を設定することができます。

メモ: Border Box では、ディスプレイボックスの引数が1つしか指定できません。複数の子を持つBorder Box を作成するには、唯一の子として V List Box() または H List Box() を使用し、V List Box() または H List Box() に複数の子を含めます。

Sides() 関数に次のような値を指定することで、枠を表示する箇所を指定できます。複数の箇所に枠を表示するには、値を足して指定します。たとえば、「5」を追加すると上下の枠が作成されます。

1: 上

2: 左

4: 下

8: 右

Border Box() では、表11.2にあるような追加の効果も用意されています。

表11.2 枠ボックスの Sides() の追加効果

上記の値に加算	追加の効果
16	枠線を、オペレーティングシステムの強調色で表示します。コンピュータによって異なる色が使用されるため、推奨しません。
32	枠ボックスの子を描く前に、枠ボックスを白で塗りつぶします。
64	枠ボックスの子を描く前に、枠ボックスを背景色で塗りつぶします。このオプションに効果があるのは、枠ボックスの親がまず枠ボックスの後ろに何かを描く場合に限られます。

Col List Box

Col List Box() は、現在のデータテーブルの列をリストするディスプレイボックスを戻します。

```
Col List Box( <Data Table( <name> ), <all>|<character>|<numeric>, <Width( n )>,
   <Max Selected( n )>, <NLines( n )>, <Max Items( n )), <Min Items( n )>, <On
   Change( expression )>, <script> );
```

All は、現在のデータテーブルの列をすべて含めるよう指定します。文字タイプまたは数値タイプの列だけを含めることもできます。Width（幅）の単位はピクセル数です。Max Selected は、リストボックスで選択できる項目の最大数を表します。NLines は、ボックスに表示する行数を表します。Max Items は、Col List Box に表示される項目の最大数を表します。Min Items は、Col List Box に表示される項目の最小数を表します。On Change は、選択が変更されたときに実行されるスクリプトを表します。

Col List Box に Get Items メッセージを送ると、選択されている列のリストを取得できます。Get Items を使用したスクリプト例を紹介します。その他のメッセージや例については、［ヘルプ］メニューの［スクリプトの索引］を参照してください。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
win = New Window( "Get Items のデモ ",
   H List Box(
      chooseme =
      Col List Box( "All", width( 100 ), NLines( 6 ) ),
      // Col List Box は幅 100 ピクセル、6 行のサイズ、すべての列を表示する
      Line Up Box( N Col( 1 ), Spacing( 2 ),
      // Line Up Box はボタンボックスを 1 列に配置し、
      // ボックスの周りを 2 ピクセルのスペースで囲む
         Button Box( " 列の追加 >>",
            listcols << Append( chooseme << Get Selected );
            // 選択された列を、これまで選択されていた列に
            // 追加する
            Chosen Columns = listcols << Get Items;
            // Col List Box から選択されている列のリストを取得する
         ),
         Button Box( "<< 列の削除 ",
            listcols << Remove Selected; // 選択された列を削除する
            Chosen Columns = listcols << Get Items; // 選択された列を取得する
         ),
      ),
         listcols = Col List Box( width( 100 ), NLines( 6 ) ),
         // 列を含まない別の Col List Box を作成する
   ),
   Text Box( " " ),

   stuff = Global Box( Chosen Columns )
   // H List Box の下部に
   // Get Items が戻した列を表示する

);
```

Col List Box() の All 引数は、ディスプレイボックスにデータテーブルの列をすべて含めます。この場合、Remove Selected や Remove All を使って選択されている列を削除することはできません。削除できるようにするには、空白の Col List Box を作成し、すべての列を追加します。そこから、選択されている列を削除します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
colNames = dt << Get Column Names; // 列名のリストを取得する

New Window( "例",
myBox = Col List Box( width( 200 ), nlines( 10 ) ) );
myBox << Append( colNames );
// データテーブルの列名をすべて含んだCol List Boxを作成する

myBox << Set Selected( 3 ); // myBoxの列名から1つまたはすべてを選択する
myBox << Remove Selected(); // myBoxで選択した列を削除する
```

Col Span Box

Col Span Box() は、Table Box 内で複数の列にまたがる列見出しを作成します。最上部の列見出しは、2つの子列へッダにまたがります。

```
Col Span Box( "title", display box args);
```

次のスクリプトでは、「信頼限界」の見出しが「上側限界」と「下側限界」の2つの列にまたがるように作成されます。

```
win = New Window( "Col Span Box",
   <<Modal,
   Table Box(
      Col Span Box(
         "信頼限界",
         neb = Number Col Edit Box( "上側限界", [0, 0] ),
         Number Col Edit Box( "下側限界", [0, 0] )
      )
   )
);
```

H List Box と V List Box

H List Box() は、ボックスを横方向に並べます。

```
H List Box( <Align( center|bottom )>, display box args, ...);
```

Alignは、ディスプレイボックスの中身を中央に揃えるか、下に揃えるかを指定します。デフォルトでは中身が上に揃えられます。

次のスクリプトは、チェックボックスを横方向に並べます。

```
win = New Window( "H List Box",
   <<Modal,
   H List Box(
      kb1 = Check Box( "a" ),
      kb2 = Check Box( "b" ),
      kb3 = Check Box( "c" )
   ),
);
```

V List Box() は、ボックスを縦方向に並べます。

```
V List Box( <Align( center|right )>, display box args, ...);
```

Align は、ディスプレイボックスの中身を右に揃えるか、中央に揃えるかを指定します。デフォルトでは中身が中央に揃えられます。

```
win = New Window( "V List Box",
   Text Box( "一番好きな果物はどれですか。" ),
   V List Box(
      Check Box( "りんご" ),
      Check Box( "バナナ" ),
      Check Box( "オレンジ" ),
   ),
);
```

デフォルトの最大幅は180ピクセルです。ボックスのサイズを中身に合わせるには、幅を0に設定します。

```
listA = {"a", "b", "c",
"これは、メアリー・アン・ジョンストンのようにとても長い名前です。"};
New Window( "例",
   V List Box( listBoxA = List Box( listA, width( 0 ) ) ) );
```

If Box

If Box() は、条件によって中身が表示されるディスプレイボックスを作成します。

```
If Box( 0|1, display box args );
```

ブール値の引数1はIf Boxの内部にあるディスプレイボックスを表示し、0は表示しません。

次のスクリプトは、3つのScroll Boxを含むSplitter Boxを作成します。2番目のScroll Boxは、If Box() に囲まれています。If Box() が1に設定されているため、2番目のScroll Boxもウィンドウに表示されます。

```
win = New Window( "If Box",
   H Splitter Box(
      Size( 500, 250 ),
      Scroll Box(),
      If Box( 1, Scroll Box() ), // If Box の設定により、Scroll Box は表示される
      Scroll Box()
   )
);
```

If Boxを非表示に設定すると、内部にあるものすべてがレポートウィンドウで非表示になります。ツリー構造の表示では、非表示のディスプレイボックスに番号が付きません。この例でIf Boxを非表示にすると、ツリー構造の表示で2番目のScroll Boxに番号が付かなくなります。Scroll Box(3) がScroll Box(2) になります。このスクリプトのIf Boxの引数を1から0に変更し、win Show Tree Structure() メッセージを送って構造を表示してください。

メモ: If Box では、ディスプレイボックス引数が1つしか指定できません。

H Splitter Box と V Splitter Box

H Splitter Box() と V Splitter Box() は、引数によって指定されたディスプレイボックスを横方向または縦方向にレイアウトするディスプレイボックス（パネル）を作成します。ユーザは分割線をドラッグしてパネルのサイズを変更できます。

```
H Splitter Box( <Size( h,v )>, display box args, <arguments> );
V Splitter Box( <Size( h,v )>, display box args, <arguments> );
```

Size(h, v) は、Splitter Box のサイズをピクセルで指定します。内側のディスプレイボックスのサイズは、外側の Splitter Box の幅と高さに合わせて変更されます。引数のリストについては、［ヘルプ］メニューの［スクリプトの索引］を参照してください。

次の例は、「二変量」と「一元配置」のグラフを H Splitter Box() の内側にある V List Box() の中に配置します。Splitter Box は、この例にあるように中身が自動伸縮するように設定されている場合に特に効果的です。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
win = New Window( "H Splitter Box",
   H Splitter Box( // V List Box を H Splitter Box の中に入れる
      Size( 600, 900 ),
      V List Box( // 二変量と一元配置のグラフを V List Box に入れる
         biv = dt << Bivariate(
            Y( :weight ),
            X( :height ),
            Fit Line(), Line Style( Dotted )
            )
         )
      ),
      one = Oneway(
         Y( :Name("身長(インチ)")),
         X( :性別 ),
         Name( "平均/ANOVA" )( 1 ),
         Mean Diamonds( 1 )
      )
   )
);
(biv << Report)[FrameBox( 1 )] << Set Auto Stretching( 1, 1 );
(one << Report)[FrameBox( 1 )] << Set Auto Stretching( 1, 1 );
// レポート層への参照を作成する
// ユーザが分割線をドラッグしたときに FrameBox の X 軸と Y 軸が
// 自動伸縮するように設定する
```

Line Up Box

Line Up Box() は、*Display Box* 引数を *n* 列に表示します。オプションで、列と列のスペースをピクセル数で指定することができます。

```
Line Up Box( NCol( n ), <Spacing( pixels )>, display box args, ... );
```

Spacingは、ボックスの周りのピクセル数を指定します。

次の例では、「一変量の分布」レポートが6列に並びます。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
win = New Window( "Line Up Box",
   Line Up Box( NCol( 6 ), // 「一変量の分布」を 6 列に並べる
      dist = Distribution(
         Continuous Distribution( Column(:Name("身長 ( インチ )") ) ),
         By( :年齢 )
      )
   )
);
```

Outline Box

Outline Box() は、他のディスプレイボックスを格納したディスプレイボックスを作成し、アウトライン形式で表示します。

```
Outline Box("title", display box args, ... );
```

次の例では、各アウトラインの下に「一変量の分布」が表示されます。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
win = New Window( "Outline Box",
   obj1 = Outline Box( "年齢と体重",
      Distribution( Column( :年齢, :Name("体重 ( ポンド )") ) ),

   ),
   obj2 = Outline Box( "体重と身長",
      Distribution( Column( :Name("体重 ( ポンド )"), :Name( "身長 ( インチ )" ) ) )
   )
);
```

アウトラインを閉じるには、アウトラインオブジェクトにClose(1)メッセージを送ります。

```
obj2 << Close( 1 );
```

アウトラインボックスの最初の引数はタイトルです。タイトルは、スクリプト内で指定するか、スライダーボックス（Slider Box）の値など別の要素に合わせて決めるようにします。次の例では、ユーザがスライダーを調整すると、アウトラインボックスのタイトルが更新されます。

```
sliderValue = .6;
win = New Window( "Outline Box",
```

```
      Panel Box( "Slider Box",
         tb = Outline Box( "値： " || Char( sliderValue ) ),
         sb = Slider Box(
            0,
            1,
            sliderValue,
            tb << Set Title( "値： " || Char( sliderValue ) )
         )
      )
   );
```

アウトラインボックスに独自の赤い三角ボタンメニューを作成したいときは、アウトラインボックス参照に Set Menu Script メッセージを送ります。次の例では、Set Submenu メッセージも使って入れ子のメニュー項目を作成します。

```
win = New Window( "赤い三角ボタンメニューの項目",
ob = Outline Box( "Outline Box" ) );
ob << Set Menu Script(
   {"A", "", "A1", Print( "A1" ), "A2", Print( "A2" ), "B", "",
   "B1", Print( "B1" ), "B2", Print( "B2" ), "B3",
   Print( "B3" ), "C", Print( "C" )}
);
ob << Set Submenu( 1, 2 ); // A1 と A2 をメニュー A のサブメニューにする
ob << Set Submenu( 4, 3 ); // B1、B2、B3 をメニュー B のサブメニューにする
```

Panel Box

Panel Box() は、*displaybox* 引数をラベルつきの枠で囲みます。

```
Panel Box("title", display box args );
```

次の例では、「一変量の分布」起動ウィンドウの周りに Panel Box が現れます。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
win = New Window( "Panel Box",
   H List Box( Panel Box( "変更版：一変量の分布", Distribution() ) )
);
```

表示要素のボックス

Graph Box

Graph Box() は、フレームボックスと軸を含むディスプレイボックスを作成します。

```
Graph Box( "title", <X Scale( low, high ), <Y Scale( low, high )>, <Frame Size( h,
   v )>,< <XName( "name" )>, <YName( "name" )>, <SuppressAxes>, script );
```

X Scale() と Y Scale() は、X 軸と Y 軸の下限と上限を指定します。Frame Size() は、グラフを囲むフレームのサイズを指定します。XName と YName は、X 軸と Y 軸の名前を指定します。

次に、シンプルなグラフの例を示します。

```
win = New Window( "Graph Box",
   Graph Box(
      Frame Size( 300, 300 ),
      Marker( Marker State( 3 ), [11 44 77], [75 25 50] );
      Pen Color( "Blue" );
      Line( [10 30 70], [88 22 44] );
   )
);
```

次の例は、目盛りが表示されないようにするために、軸に2つの引数を追加したものです。

```
win = New Window( "Graph Box",
   Graph Box(
      Frame Size( 300, 300 ),
      xaxis( // X軸の目盛りを削除する
         Show Major Ticks( false ),
         Show Minor Ticks( false ),
         Show Labels( false )
      ),
      yaxis( // Y軸の目盛りを削除する
         Show Major Ticks( false ),
         Show Minor Ticks( false ),
         Show Labels( false )
      ),
      Marker( Marker State( 3 ), [11 44 77], [75 25 50] );
      Pen Color( "Blue" );
      Line( [10 30 70], [88 22 44] );
   )
);
```

JMPには、対話式のグラフ（オブジェクトをドラッグする、マウスクリックをキャプチャするなど）を作成するための機能が揃っています。「スクリプトによるグラフ作成」章（517ページ）に、グラフボックスオプションに関する詳しい説明があります。

H Sheet Box と V Sheet Box

Sheet Box()を使うと、複数のグラフを縦横に並べることができます。V Sheet Box()とH Sheet Box()は、そこに含まれているディスプレイボックスを列と行に並べます。

```
H Sheet Box( <<Hold( report ), display box args );
V Sheet Box( <<Hold( report ), display box args );
```

まず、どのディスプレイボックスを、どのように配置するかを検討します。H Sheet BoxかV Sheet Boxを作成し、そこに各グラフのHoldメッセージを送ることで、シートボックスにどの要素を保持するかを指定します。最後に、内側に配置するH Sheet BoxまたはV Sheet Boxを作成し、それぞれにどのグラフを保持するかを指示します。

次の例では、4つのグラフ（二変量の散布図、一変量のヒストグラム、ツリーマップ、バブルプロット）を含むシートを作成します。

まず、データテーブルを開き、新しいウィンドウを作成します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
nw = New Window( "シートボックスの例",
```

V Sheet Boxを使い、ウィンドウを2列に並べることにしましょう。

```
V Sheet Box(
```

V Sheet Box()に、グラフごとに1つ、計4つのHoldメッセージを送ります。このとき、順序が重要です。

```
<<Hold(
   dt << Bivariate( // グラフ1
      Y( :weight ),
      X( :height ),
      Fit Line()
   )
),
<<Hold(
   Distribution( // グラフ2
      Continuous Distribution(
         Column( :Name("身長(インチ)") ),
         Horizontal Layout( 1 ),
         Outlier Box Plot( 0 )
      )
   )
),
<<Hold( Treemap( Categories( :年齢 ) ) ), // グラフ3
<<Hold(
   Bubble Plot( // グラフ4
      X( :height ),
      Y( :weight ),
      Sizes( :年齢 ),
      Coloring( :性別 ),
      Circle Size( 6.226 ),
      All Labels( 0 )
   )
),
```

最後に、V Sheet Box内に2つのH Sheet Boxを追加し、どのグラフを保持するかを指示します。各H Sheet Boxに、横に並べて表示する2つのグラフを保持します。H Sheet BoxはV Sheet Boxによって保持されているので、H Sheet Box全体は縦に表示されます。

```
H Sheet Box(
   Sheet Part(
      "",
```

Sheet Partは、Excerpt Boxに指定されたグラフを表示します。第1引数はグラフの番号で、グラフを定義した際の順番を示します。ですから、この最初のH Sheet Boxには、左側に二変量の散布図が、右側に一変量のヒストグラムが表示されます。{Picture Box(1)}の部分は、レポートのどのピクチャーボックスを表示するかを指定しています。通常は、1、つまりレポート内の最初のピクチャーボックスを使用します。

```
            Excerpt Box( 1, {Picture Box( 1 )} ) // excerpt box 1
         ),
         Sheet Part(
            "身長(インチ)の分布",
            Excerpt Box( 2, {Picture Box( 1 )} ) // excerpt box 2
         )
      ),
      H Sheet Box(
         Sheet Part(
            "",
            Excerpt Box( 3, {Picture Box( 1 )} ) // excerpt box 3
         ),
         Sheet Part(
            "指定したタイトル",
            Excerpt Box( 4, {Picture Box( 1 )} ) // excerpt box 4
         )
      )
   )
);
```

Sheet Part()のタイトルは必須です。空白の文字列をタイトルとして含めると、シートパートのタイトルに、デフォルトのレポートタイトル（たとえば「身長(インチ)と体重(ポンド)の二変量の関係」）が使用されます。

Journal Box

Journal Box()は、ジャーナルに保存されている指示に従ってディスプレイボックスを作成します。

```
box = Journal Box("journal text");
```

ここで、*"journal text"*（ジャーナルテキスト）は、ジャーナルファイルから抽出されたテキストです。

ジャーナルテキストには、可能なボックスの組み合わせについて多くのルールがあるため、レポートのある部分を強調表示してから［編集］>［ジャーナル］を選択してジャーナルテキストを取得し、ジャーナルとして保存することをお勧めします。そのジャーナルをテキストエディタで開き、スクリプトの中にJournal Box()引数として貼り付けます。

この際、"\[...]\"という引用符を使うことを強くお勧めします。このように指定すると、ジャーナルテキスト内で二重引用符を使用することができます。次の抜粋は、ジャーナルテキストをエスケープする方法を示します。

```
win = New Window( "モザイク図",
   Journal Box(
```

```
        "\[ // 引用符がそのまま使える範囲の始まり
    ...display box arguments...
    ]\"
    ) // 引用符がそのまま使える範囲の終わり
);
```

ジャーナルテキストを取得するもう一つの方法は、ディスプレイボックスに Get Journal メッセージを送る方法です。たとえば、次のスクリプトを実行し、ログからジャーナルテキストをコピーして Journal Box() 関数に入れることができます。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
biv = dt << Bivariate( y( :Name("体重(ポンド)") ), x( :Name("身長(インチ)") ) );
rbiv = biv << Report;
Print( rbiv << Get Journal ); // ジャーナルをログに出力する
```

ピクチャーオブジェクト

JSL には、JMP の出力や計算式のピクチャーを入れるピクチャーオブジェクトがあります。ディスプレイボックスの中にあるものをイメージで取り出したり、テキスト形式の計算式を、計算式エディタで見るようなイメージで作成したりできます。

イメージデータを作成するには、*ディスプレイボックス*に Get Picture メッセージを送ります。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
biv = dt << Bivariate( y( :Name("体重(ポンド)") ), x( :Name("身長(インチ)") ) );
rbiv = biv << Report;
rbiv << Get Picture;
```

Expr As Picture() 関数は、引数を評価し、計算式エディタと同じ表示形式メカニズムを使って、計算式のイメージを作成します。式そのものを引数に指定する場合は、そのままイメージとして表示されるように、忘れずに Expr() で式を囲んでください。そうすれば、式が評価されません。

```
win = New Window( "計算式",
    Expr As Picture( Expr( a + b * c + r / d + Exp( x ) ) )
);
```

イメージには、次の2つの利用方法があります。

- ディスプレイボックス作成用関数を使い、新しいディスプレイツリーにそのピクチャーを追加する。
- ピクチャーを、Save Picture() を使ってファイルに書き込む。

  picture << Save Picture("*path*", *type*);
  type は、EMF（Windows）、PICT（Macintosh）、JPEG または JPG、GIF、PNG のいずれかです。

Windows の場合は、［Windows のみ］環境設定で解像度（DPI）を指定できます。または、次の式で設定することもできます。

```
Pref( Save Image DPI( <数値> ) );
```

Macintosh では、オペレーティングシステムによって DPI が決められます。

入力選択ディスプレイボックス

入力選択ディスプレイボックスでは、ユーザは情報を入力し、ディスプレイボックスを操作することができます。このディスプレイボックスのスクリプトには、ユーザがディスプレイボックスを操作したときに実行されるスクリプトが含まれます。スクリプトの記述方法については、「Set Function と Set Script」（477ページ）を参照してください。

Button Box

Button Box() は、テキストを含んだボタンを描画します。

```
Button Box("text", <script>);
```

次の例は、［OK］ボタンと［キャンセル］ボタンを含む新しいウィンドウを作成します。

```
win = New Window( "値の指定",
   <<Modal,
   Text Box("この値を指定"),
   variablebox = Number Edit Box( 42 ),
   Button Box( "OK" ),
   Button Box( "キャンセル" ) );
```

モーダルウィンドウには、少なくとも1つのボタンが必要です。スクリプトにButton Box()が含まれない場合、［OK］ボタンが1つ自動的に含められます。

ボタンのツールヒントを追加するには、ボタンにSet Tipメッセージを送ります。以下のスクリプトは、［送信］ボタンを作成します。このボタンにカーソルを置くと、「私の情報を送信する」というテキストのツールヒントが表示されます。

```
Button Box( "送信", << Set Tip( "私の情報を送信する" ) );
```

また、次のアウトラインボックスを開くOpen Next Outlineメッセージをスクリプトコマンドとしても送ることができます。ボタンボックスに複数のメッセージを送る場合、このメッセージを最初のコマンドとして記述する必要があります。ユーザがこの例の［次へ］ボタンをクリックすると、2番目のアウトラインボックスが開きます。

```
win = New Window( "次のアウトラインボックスを開く例",
   Outline Box( "最初のアウトラインボックス",
      ex = Button Box( "次へ", ex << Open Next Outline )
   ),
   Outline Box( "2つ目のアウトラインボックス", Text Box( "これが最後です。" ), << Close
   )
);
```

Set Menu Itemsメッセージを使うと、メニューを含むボタンを作成できます。

```
win = New Window( "メニューの選択肢の設定と取得",
   bb = Button Box( "デザートの選択",
   // ボタンがクリックされたときのメニューの選択肢を取得する
      choice = bb << Get Menu Choice;
```

```
        Show( choice );
    )
);
bb << Set Menu Items( {"果物", "いりません", "-", "チョコレート", "アイスクリーム"}
  );
```

この例のスクリプトの"-"は、メニューの区切り線を作成します。区切り線はリスト内の項目として数えられます。メニューから［チョコレート］を選んだ場合、3ではなく、4が戻されます。

メモ: Button Boxでは改行文字が無視されます。

Check Box

Check Box()は、1つまたは複数のチェックボックスを表示するディスプレイボックスを作成します。チェックボックスの項目は、同時にいくつでも選択できます。

```
Check Box ({"item 1", "item 2", ...}, <script>);
```

次の例では、チェックボックスに Get Selected メッセージを送ってユーザの選択内容を取得します。選択されている項目がそれぞれログに出力されます。

```
win = New Window( "Check Box",
    cb = Check Box(
        {"りんご", "バナナ", "みかん"},
        scb = cb << Get Selected();
        Show( scb );
    )
);
  scb = {"バナナ", "みかん"};
```

Combo Box

Combo Box()は、ドロップダウンリストを作成します。

```
Combo Box( {"item 1", "item 2", ...}, <script> );
```

「item1」と「item2」は、ドロップダウンリストに表示される文字列です。

Editable引数は、コンボボックスへのテキスト入力を可能にします。次の例は、新しいウィンドウに編集可能なコンボボックスを作成し、コンボボックスのリストに「いち」、「に」、「さん」という項目を含めます。このスクリプトは、選択された項目の名前とインデックス番号をログに出力します。

```
win = New Window( "Combo Box",
    cb = Combo Box(
        {"いち", "に", "さん"},
        Editable,
        selection = cb << GetSelected();
        Print( "選択されたもの： " || selection );
        Print( "番号： " || Char( cb << Get() ) );
```

```
    )
);
```

編集可能なコンボボックスのウィンドウを閉じると、予期しない結果になる場合があります。Windowsの場合、Print()スクリプトは、ユーザがリストから項目を選択したとき、または項目名を入力したときに実行されます。Macintoshの場合、Print()スクリプトは、ユーザがリストから項目を選択したとき、または項目名を入力してReturnキーを押したときに実行されます。

プラットフォーム間での互換性を維持するため、!Is Emptyを含めてスクリプトの実行前に既存のコンボボックスの存在を確認してください。

```
win = New Window( "Combo Box",
   cb = Combo Box(
      {"いち", "に", "さん"},
      Editable,
      If( !Is Empty( cb ),
         selection = cb << GetSelected();
         Print( "選択されたもの： " || selection );
         Print( "番号： " || Char( cb << Get() ) );
      )
   )
);
```

Global Box

Global Boxは、JSLグローバル変数の名前と現在の値を表示します。

```
Global Box( name );
```

ユーザは、ウィンドウで値を直接編集し、EnterキーまたはReturnキーを押して変更を確定することで、新しい値をグローバル変数に割り当てることができます。Global Boxは、変数が変化すると自動的にその変数の表示値を更新します。

```
ex = Sqrt( 4 );
win = New Window( "Global Box", Global Box( ex ) );
```

上のスクリプトは、「Global Box」という名前の新しいウィンドウを作成し、次の結果を表示します。

ex=2

次の例は、myGlobal変数の値をグラフ上に表示します。

```
myGlobal = 6;
   win = New Window( "Global Box",
      V List Box(
          gr = Graph Box(
              Frame Size( 300, 300 ),
              X Scale( 0, 10 ),
              Y Scale( 0, 10 ),
              Y Function( x, x ),
```

```
            Text( {5, 5}, "myGlobal は ", myGlobal )
        ),
        Global Box( myGlobal ) // myGlobal の値
    )
);
```

メモ: Global Box を使用すると、変数の値が変化するたびにウィンドウの更新が発生します。このため、Global Box オブジェクトの数によってはウィンドウの更新に時間がかかってしまう可能性があります。最終版のスクリプトでは、グローバルボックスの数が多くならないようにすることをお勧めします。グローバルボックスではなく、必要なときに手動で更新できるテキストボックスを使用する方法もあります。

List Box

List Box() は、選択可能な項目を含んだリストを表示するディスプレイボックスを作成します。

```
List Box({"item 1", "item 2", ...}, <Width( n )>, <MaxSelected( n )>,
  <NLines( n )>, <script>);
```

項目名では、デフォルトで大文字・小文字が区別されます。Width（幅）の単位はピクセル数です。MaxSelected は、List Box で選択できる項目の最大数を表します。NLines は、ボックスに表示する行数を表します。デフォルトは 3 です。

Append または Insert を使用して、項目をリストボックスに追加できます。

```
win = New Window( " テスト ", lb = List Box( {"a", "e"} ) );
lb << Append( {"f", "g"} ); // 結果は a、e、f、g
lb << Insert( {"b", "c", "a", "d"}, 1 ); // 結果は a、b、c、d、e、f、g
```

Append は、常に、リストボックスの最後にリストを追加します。Insert は、指定した位置の後ろにリストを追加します。

一方のボックスで選択された項目を他方のボックスで選択すると、最初のボックスの選択が解除されるといった、互いに排他的な 2 つのリストボックスを作成できます。項目の選択を解除するには、Clear Selection メッセージを送ります。

```
win = New Window( " 互いにクリアし合うボックス ",
   a = List Box(
      {" ブロッコリー ", " ピーマン ", " ほうれん草 "},
      <<Set Script( window:lb << Clear Selection )
   ),
   lb = List Box(
      {" アボカド ", " かぼちゃ ", " トマト "},
      <<Set Script( window:la << Clear Selection )
   )
);
```

リストボックスには、イメージを含めることもできます。次の例では、ユーザのコンピュータにあるイメージが 1 番目のリスト項目内に表示されます。2 番目のリスト項目には、JMP の名義尺度アイコンが表示されます。

```
win = New Window( "画像を含む List Box",
    List Box(
        {{"first", "$SAMPLE_IMAGES/pi.gif"}, {"second", "nominal"}},
        width( 200 ),
    )
);
```

Mouse Box

Mouse Box() は、ドラッグ＆ドロップ、マーキング、クリック、トラックなどのマウス動作に対する JSL コールバックを作成するボックスを作成します。

```
Mouse Box( display box args, messages );
```

次の例は、「Samples/Scripts」フォルダの「DragDrop.jsl」サンプルスクリプトから取ったものです。一方のリストボックスには、「a」、「b」、「c」が含まれます。他方のリストボックスには、「d」、「e」、「f」が含まれます。一方のリストボックスの文字を他方のリストボックスへドラッグすることができます。

メモ: Mouse Box では、ディスプレイボックス引数を1つしか指定できません。

```
win = New Window( "Mouse Box",
   Mouse Box( // 最初の Mouse Box
      Text Box( "ここからドラッグ" ),
      <<setDragText( "hello" ),
      <<setTooltip( "ドラッグ元" ),
      <<setDragEnable( 1 ),
      <<setDragBegin( // ドラッグ操作を許可するか決める
         Function( {this, clickpt},
         )
      ),
      <<setDragEnd( // ドラッグ操作の完了またはキャンセル後、クリーンアップする
         Function( {this, clickpt, how},
         // how は移動するか、コピーするか、無視するかを示す
            If(
               how != "ignore" & !Is Empty( this << getDestBox ) & this
            /* getDestBox により、ドラッグ＆ドロップの行き先が
             兄弟であり、他のプログラムでないことを確認する */
               <<getDestBox == this << sib
            ,
               (this << child) << setText(
                  "Done!" // "move" はドラッグ元をクリアする
               )
            )
         )
      )
   ),
   Mouse Box( // 兄弟の Mouse Box
```

```
      Text Box( "ここへドラッグ" ),
      <<setTooltip( "ドロップ先" ),
      <<setDropEnable( 1 ),
      /* ドロップの前に、ドロップ操作を許可する。
       getSourceBox により、ドラッグ&ドロップの元が
       兄弟であり、他のプログラムでないことを確認する */
      <<setDropTrack(
         Function( {this, clickpt},
            If( !Is Empty( this << getSourceBox ) & this == (this << getSourceBox)
<< sib,
               1, /*else*/ 0
            )
         )
      ),
      <<setDropCommit( // ドロップを受け入れる
         Function( {this, clickpt, text},
            (this << child) << setText( text )
         )
      )
) );
```

Number Col Edit Box

Number Col Edit Box() は、列内に編集可能な数値を表示するディスプレイボックスを作成します。数値は、リストまたは行列として表示されます。

```
Number Col Edit Box("title", numbers);
```

次のスクリプトは、ユーザが値を入力できるようにNumber Col Edt Boxを作成します。Return Result メッセージは値を抽出します。

```
x = y = z = 0;
win = New Window( "Number Col Edit Box",
   <<Modal,
   <<Return Result,
   Table Box(
      neb = Number Col Edit Box( "values", {x, y, z} )
   )
);
  {neb = {2, 4, 6}, Button( 1 )}
```

Number Edit Box

Number Edit Box() は、引数で指定した初期値を持つ、編集可能な数値のボックスを作成ます。ボックスの幅を文字で設定することもできます。

```
Number Edit Box( initValue, <width>);
```

次のスクリプトは、編集可能なボックスを作成し、その中に「42」という初期値を表示します。ユーザが入力する値は、ログに出力されます。

```
win = New Window( "Number Edit Box で値を指定 ",
   <<Modal,
   <<Return Result,
   Text Box( " 値を指定してください。" ),
   variablebox = Number Edit Box( 42 ),
   Button Box( "OK"),
   Button Box( " キャンセル ")
);
Write( win["variablebox"] );
// variablebox 変数の添え字を指定する
  33 // ユーザがNumber Edit Boxに「33」と入力した場合
```

次のスクリプトは、初期値を日付にする方法を示し、ユーザが別の日付を選択できるようにしています。

```
New Window( " 日時の選択 ",
     <<Modal,
     neb = Number Edit Box(
          Today(),
          23,
          <<Set Format( Format( "m/d/y h:m:s", 23, 0 ) )
     )
);
```

Popup Box

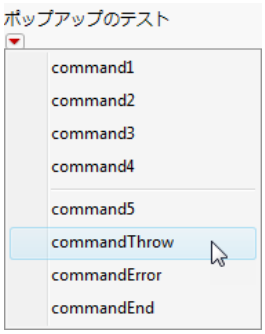
Popup Box() は、コマンドをメニュー項目として含んだ赤い三角ボタンのメニューを作成します。

```
Popup Box( {"command1", script1, "command2", script2, ...} );
```

次の例では、コマンドが含まれたリストを変数に格納し、Popup Box() でこのリストを表示します。

```
List = {
"command1", Print( "command1" ),
"command2", Print( "command2" ),
"command3", Print( "command3" ),
"command4", Print( "command4" ),
"", Empty(), // 区切り線を挿入
"command5", Print( "command5" ),
"commandThrow", Throw( "commandThrow1" ),
"commandError", Sqrt( 1, 2, 3 ),
"commandEnd", Print( "commandEnd" )};

win = New Window( " 例 ",
  Text Box( " ポップアップのテスト " ),
  Popup Box( List );
);
```

図11.8 赤い三角ボタンのメニュー



Enable(Boolean)を使用することで、メニューの有効／無効を切り替えることができます。引数が1の場合、メニューが有効になり、引数が0の場合は無効になります。前述の例を使用して、ポップアップボックスを変数に割り当て、その変数にenableメッセージを送ります。

```
win = New Window( "例",
  Text Box( "ポップアップのテスト" ),
  mymenu = Popup Box( commandList );
);

mymenu << Enable( 0 ); // メニューを無効にする
```

アウトラインボックスにポップアップメニューを追加することもできます。例については、「Outline Box」（441ページ）を参照してください。

Radio Box

Radio Box()は、ラジオボタンのリストを作成します。

```
Radio Box({"item 1", "item 2", ...}, <script>);
```

Radio Boxの項目は、一度に1つしか選択できません。

次のスクリプトは、「A、「B」、「C」というラジオボタンを作成し、2番目のボタンをオンにします。

```
win = New Window( "Radio Box",
   V List Box(
      rb = Radio Box( {"A", "B", "C"}, <<Set( 2 ) ),
   )
);
```

「New Windowの例」（505ページ）には、Radio Boxの作成例もあります。

Slider Box

Slider Box()は、最小値（min）と最大値（max）で指定された範囲内で、変数の値を指定するスライダコントロールを作成します。

```
Slider Box( min, max, global variable, script, <Set Width( n )>, <Rescale Slider(
  min, max )> );
```

Set Widthは、Slider Boxの幅を指定します。Rescale Sliderは、最小値（min）と最大値（max）を指定します。

スライダを移動すると、スライダの現在の位置による値がグローバル変数に割り当てられます。このため、Slider Box()を用いることにより、ユーザが指定した値をグラフに反映させることができます。

```
ex = .5;
win = New Window( "Slider Box",
   tb = Text Box( "値： " || Char( ex ) ),
   sb = Slider Box(
      0,
      1,
      ex,
      tb << Set Text( "値： " || Char( ex ) )
   )
);
sb << Set Width( 100 ) << Rescale Slider( 0, .8 );
```

次のスクリプトは、複数のSlider Boxを作成しますが、各Slider Boxの現在の値を保持するための一意のグローバル変数は使用しません。

```
sb1 = Slider Box( 1, 10 );
sb2 = Slider Box( -10, 10 );
// 最小値と最大値だけを指定してスライダボックスを作成する
sb1 << Set Function( // スクリプトまたは関数を設定する
   Function( {this},
      {},
      Show( this << Get, sb2 << Get )
   )
);
sb2 << Set Script( Show( sb2 << Get, sb1 << Get ) );
New Window( "値", sb1, sb2 ); // スライダボックスをウィンドウに入れる
```

スライダを動かすと、値がログに出力されます。

「スライダボックスと範囲スライダボックスの例」（465ページ）には、スライダボックスの別の例があります。

Spacer Box

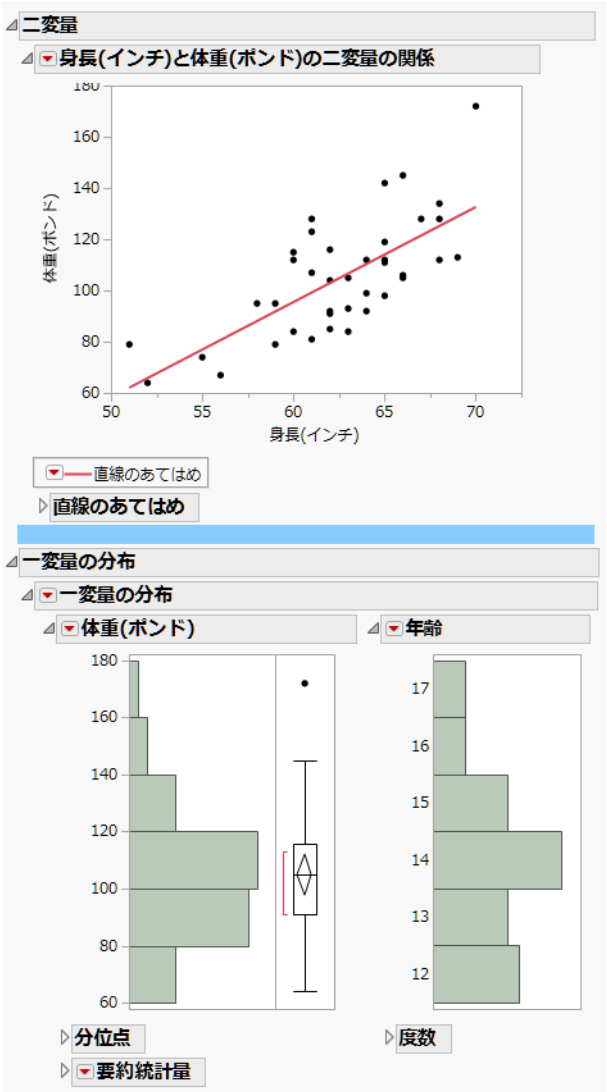
Spacer Box()は、ディスプレイボックスの間にスペースを作成します。

```
Spacer Box(<size(horizontal_pixels,vertical_pixels)>, <color(color)>)
```

次のスクリプトは、2つのアウトラインボックスの間に、幅450ピクセル、高さ15ピクセルのスペーサーボックスを配置します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
win = New Window( "Spacer Box",
   V List Box(
      Outline Box( "二変量",
         biv = dt << Run Script( "二変量の関係(二変量)" ),
         Spacer Box( Size( 450, 15 ), Color( {.53, .80, .98} ), )
         // Color の文字列は各 RGB 値のパーセンテージを示す
      ),
      Outline Box( "一変量の分布",
         V List Box( dist = dt << Run Script( "一変量の分布" ), )
      )
   )
);
```

図11.9 スペーサーボックスの例



String Col Edit Box

String Col Edit Box() は、リストされた文字列（{strings}）を含んだ列を、指定された名前（title）とともに表の中に作成します。この関数によって作成された列の文字列は、編集できます。

```
String Col Edit Box( "title", {strings} );
```

次のスクリプトは、編集可能なテキストボックスの列を作成し、冒頭に「名前」というタイトルを付けます。

```
a = b = c = "";
win = New Window( "String Col Edit Box",
   <<Modal,
   <<Return Result,
   Table Box(
      seb =
      String Col Edit Box(
         "名前",
         {a, b, c}
         )
      )
);
  {seb = {"りんご", "みかん", "バナナ"}, Button( 1 )}
```

Tab Box と Tab Page Box

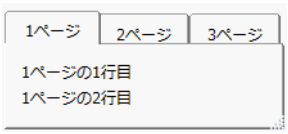
Tab Page Box() は、ページのタイトル（title）と内容（contents）を1つのディスプレイボックスにまとめます。Tab Page Box() を Tab Box() の中に入れて使用した場合、複数のタブページを含んだタブ付きウィンドウが表示されます。

```
Tab Box( Tab Page Box( "page title 1", [options], contents of page 1), Tab Page
  Box( "page title 2", [options], contents of page 2), ...);
```

次のスクリプトは、3つのタブページを持つボックスを作成します。

```
win = New Window( "Tab Box",
   Tab Box(
      Tab Page Box(
         "1ページ",
         Text Box( "1ページの1行目" ),
         Text Box( "1ページの2行目" )
      ),
      Tab Page Box(
         "2ページ",
         Text Box( "2ページの1行目" ),
         Text Box( "2ページの2行目" )
      ),
      Tab Page Box(
         "3ページ",
         Text Box( "3ページの1行目" ),
         Text Box( "3ページの2行目" )
      )
   )
);
```

図11.10 タブボックス



どのタブを選択状態にするかは、タブボックスオブジェクトに`Set Selected( n )`メッセージを送ることで指定できます。*n*はタブ番号を表します。

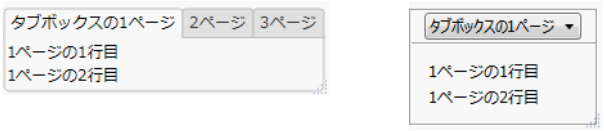
`Set Style`メッセージにより、タブボックスの外観を選択できます。デフォルト値は`tab`（タブ）です。他に次のようなオプションがあります。

- `combo`はコンボボックスを作成します。
- `outline`はアウトラインノードを作成します。
- `vertical spread`はタブのタイトルを縦に表示します。
- `horizontal spread`はタブのタイトルを横に表示します。
- `minimize size`はタブのスタイルがタイトルの幅に応じて決まります。次の例では、1番目のタブのタイトルを長めに設定しています。

```
win = New Window( "Tab Box",
   tb = Tab Box(
      Tab Page Box(
         "タブボックスの１ページ",
         Text Box( "1ページの１行目" ),
         Text Box( "1ページの２行目" )
      ),
      Tab Page Box(
         "2ページ",
         Text Box( "2ページの１行目" ),
         Text Box( "2ページの２行目" )
      ),
      Tab Page Box(
         "3ページ",
         Text Box( "3ページの１行目" ),
         Text Box( "3ページの２行目" )
      )
   )
);
tb << Set Style( "minimize size" );
```

タブボックスのサイズを最小化すると、タブボックスがコンボボックスに変換されます。図11.11は、デフォルトのタブボックスと最小化したタブボックスです。

図11.11 デフォルトのタブボックス（左）と最小化したタブボックス（右）



Tab Page Box()の例については、「2つのレポートからダッシュボードを作成する例」（484ページ）を参照してください。

ヒント:

- Set Styleメッセージは、そのままの語と引用符で囲んだ語の両方に対応します。ただし、変数にスタイルを割り当て、それを引数として渡すことはできません。
- Tab Box()とSplitter Box()を組み合わせると、ウィンドウ内のゾーンにドラッグ＆ドロップしてレイアウトを変えられるようなドッキングタブを作成することができます。詳細は、「2つのレポートからダッシュボードを作成する例」（484ページ）の節を参照してください。
- タブを削除するには、タブボックスにDelete(index)を送るか、タブページボックスにDelete Box()を送ります。1つのタブだけではなく、すべてのタブを含むディスプレイボックス全体が削除されます。
- タブを非表示にするには、Visibility("Collapsed")メッセージを使用します。
- Set Overflow Enabled(1)メッセージを使うと、リストの幅が十分でなくてすべてのタイトルが表示できない場合にタブリストの右側に「^」の記号が表示されます。
- Tab Page Box()で作成されたタブページボックス、またはTab Box()の外にドラッグされたタブページボックスは、単独のコンテナとなります。Sheet Part()のタイトルと同様に、ページ冒頭にある陰影の付いたボックスにタイトルが表示されます。対話式のタブにはタイトルが表示されません。
- Tab Box()とTab Page Box()の違いについて、詳しくは「Tab BoxとTab Page Boxのスクリプトの記述」（515ページ）を参照してください。
- バブルプロット（Bubble Plot）とグラフビルダー（Graph Builder）のスクリプトでFit to Windowメッセージを「On」に設定すると、Tab Box()内でグラフのサイズを変更したときに、ウィンドウのサイズが変更されます。

Text Box

Text Box()は、編集不可能なテキストボックスを作成します。他のコントロールのラベルとして使用されるのが一般的です。

```
Text Box("text")
```

テキストには、HTMLタグで書式を設定することができます。たとえば、次のスクリプトはテキストを太字にします。

```
win = New Window( "書式付きテキスト",
   Text Box( "これは <b> 太字 </b> のテキストです。",
   <<Markup) );
```

入れ子になったタグは、次のように正しく閉じることが重要です。

```
" これは <b><i><u> 太字斜体 </u></i></b> のテキストです "
```

メモ: 回転後のテキストボックスは、Markupテキストに対応しますが、複数の行や複数の書式、行端揃えがある大きなテキストボックスはサポートしません。

Text Edit Box

`Text Edit Box()` は、編集可能なテキストボックスを作成します。

```
Text Edit Box ( "text" );
```

スクリプトを追加するには、Text Edit Boxにスクリプトメッセージを送ります。これは、ボックスの作成時に実行するのが最も簡単です。スクリプトメッセージを最後の引数として追加するだけです。

次の例では、Text Edit Boxを変更するたびに、「変更されました」というテキストがログに出力されます。

```
win = New Window( "Text Edit Box",
   Text Edit Box( " ここを変更してください ", <<Script( Print( " 変更されました " ) ) )
);
```

Text Edit Boxの内容にアクセスするには、ボックスに参照を割り当てます。次の例では、Text Edit Boxに変更を加えるたびにその値がログに出力されます。

```
win = New Window( "Text Edit Box",
   teb = Text Edit Box( " ここを変更してください ",
      <<Script( Print( teb << Get Text ) )
   )
);
```

テキスト編集ボックスが空のとき、そこにどのような値を入力すればよいかを示すヒントを、プレースホルダテキストとして挿入することができます。プレースホルダテキストは、ヒントとして表示されるだけで、テキストフィールドの値には影響しません。

次の例は、図11.12のText Edit Boxを作成します。`Hint()` により、`"mm/dd/yyyy"` が淡いグレーで表示されます。

```
win = New Window( "Text Edit Box",
   Text Edit Box( " 現在の日付 " ),
   Text Edit Box( "", Hint( "mm/dd/yyyy" ) )
);
```

図11.12 プレースホルダテキストが表示されたText Edit Box

パスワード入力用のテキスト編集ボックスを作成する場合は、入力された文字をアスタリスクに置き換える形式を適用できます。たとえば、次の例は、「a」の文字を値とするテキスト編集ボックスを作成します。ユーザがこのテキスト編集ボックスに新しい文字列をタイプすると、すべての文字がアスタリスクとして表示され、「変更されました!」というメッセージがログに出力されます。

```
win = New Window( "Text Edit Box",
   teb = Text Edit Box( "a",
      Password Style( 1 ),
      Set Script( Print( "変更されました!" ) )
   )
);
```

テキスト編集ボックスに、パスワード形式を使用するか使用しないかを指定するメッセージを送ることもできます。

```
q << Password Style( 1 ); // テキスト編集ボックスをパスワード形式に設定する
q << Password Style( 0 ); // テキスト編集ボックスを標準形式に設定する
```

フィルタリングを行うディスプレイボックス

Data Filter Source Box()とData Filter Context Box()を併用すれば、複数のグラフでデータをフィルタリングできます。Data Filter Source Box()は、どのグラフを選択フィルタの「元」にするかを定義します。元のグラフからデータを選択すると、共通のData Filter Context Box()にあるグラフが更新されます。

Data Filter Context Box

Data Filter Context Box()は、複数のグラフでデータをフィルタリングできるディスプレイボックスを作成します。この機能により、フィルタを適用するグラフの数（たとえば、すべてのグラフ）が決まります。

```
Data Filter Context Box( display box args );
```

次の例では、グラフを含むH List Box()がData Filter Context Box()の中に入っています。スクリプトを実行した後、フィルタ内で値を選択し、グラフが更新されるのを確認してみましょう。

```
dt = Open( "$SAMPLE_DATA/Hollywood Movies.jmp" );
win = New Window( "共有ローカルフィルタ",
   Data Filter Context Box(
   // 一変量の分布とグラフビルダーを含むH List Boxを囲む
      H List Box(
         dt << Data Filter( Local ), // ローカルデータフィルタを追加する
         dt << Distribution(
            Weight( :収益性 ),
            Nominal Distribution( Column( :制作会社 ) ),
            Nominal Distribution( Column( :ジャンル ) ),
            Histograms Only
         ),
         dt << Graph Builder(
            Variables(
```

```
                X( :ジャンル ),
                Y( :米国内収入 ),
                Y( :米国外収入 , Position( 1 ) )
            ),
            Show Control Panel( 0 ),
            Elements(
                Bar(
                    X,
                    Y( 1 ),
                    Y( 2 ),
                    Legend( 2 ),
                    Bar Style( "Side by side" ),
                    Summary Statistic( "Mean" )
                ),
                Frequencies( 0 ),
            )
        )
    )
);
```

Data Filter Source Box

Data Filter Source Box()とData Filter Context Box()を併用すれば、グラフでデータをフィルタリングできます。Data Filter Source Box()は、どのグラフを選択フィルタの「元」にするかを定義します。元のグラフからデータを選択すると、共通のData Filter Context Box()にあるグラフが更新されます。

```
Data Filter Source Box( display box args );
```

次のスクリプトでは、「一変量の分布」グラフで選択したデータだけが「グラフビルダー」のグラフに表示されるようになります。

```
dt = Open( "$SAMPLE_DATA/Hollywood Movies.jmp" );
win = New Window( "共有ローカルフィルタ",
    Data Filter Context Box(
    // Data Filter Source Box とグラフビルダープラットフォームを囲む
        H List Box(
            Data Filter Source Box(
                dt << Distribution( // 一変量の分布をフィルタの元とする
                    Weight( :収益性 ),
                    Nominal Distribution( Column( :制作会社 ) ),
                    Nominal Distribution( Column( :ジャンル ) ),
                    Histograms Only
                )
            ),
            dt << Graph Builder(
                Variables(
```

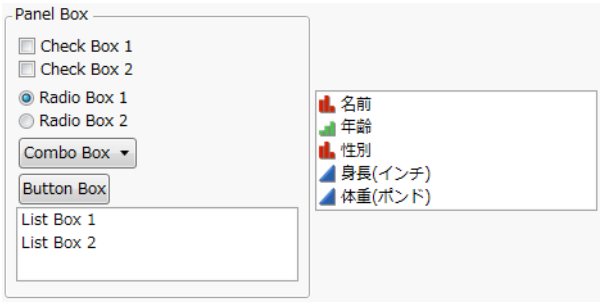
```
                X( :ジャンル ),
                Y( :米国内収入 ),
                Y( :米国外収入 , Position( 1 ) )
             ),
             Show Control Panel( 0 ),
             Elements(
                Bar(
                   X,
                   Y( 1 ),
                   Y( 2 ),
                   Legend( 2 ),
                   Bar Style( "Side by side" ),
                   Summary Statistic( "Mean" )
                ),
                Frequencies( 0 ),
             )
          )
       )
    )
);
```

ディスプレイボックスを組み合わせた例

次の例は、これまでの節で解説されている多数のコントロールのサンプルを生成します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
win = New Window( "ウィンドウのコントロール",
   Line Up Box( N Col( 2 ), Spacing( 3 ),
      Panel Box( "Panel Box",
         Check Box( {"Check Box 1", "Check Box 2"} ),
         Radio Box( {"Radio Box 1", "Radio Box 2"} ),
         Combo Box( {"Combo Box"} ),
         Button Box( "Button Box" ),
         List Box( {"List Box 1", "List Box 2"} )
      ),
      Col List Box( "all" )
   )
);
```

図11.13 インタラクティブな表示要素の例



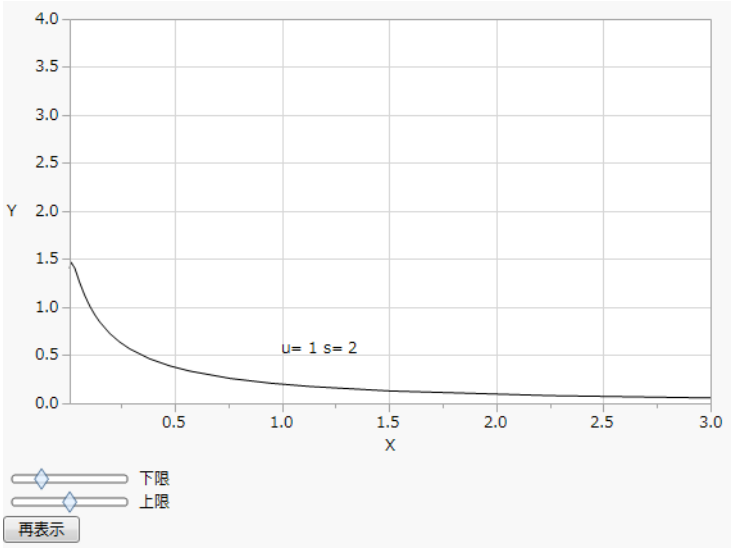
スライダボックスと範囲スライダボックスの例

次の例では、グラフボックス、2つのH List Box、1つのボタンをV List Box内に貼り付けることによって、2つのスライダと1つのボタンをったグラフを作成しています。

```
// スライダ機能を使った対数正規分布
lU = 1;
lS = 2;
win = New Window( "対数正規密度",
   V List Box(
      gr = Graph Box( // グラフボックスの引数とスクリプトを定義する
         Frame Size( 500, 300 ),
         X Scale( 0.01, 3 ),
         Y Scale( 0, 4 ),
         XAxis( Show Major Grid ),
         YAxis( Show Major Grid ),
         Y Function(
            Exp( -(Log( x ) - Log( lU )) ^ 2 / (2 * lS ^ 2) ) / (lS * x *
            Sqrt( 2 * Pi() )),
            x
         );
         Text( {1, .5}, "u= ", lU, "s= ", lS );
         // X軸の1の位置とY軸の0.5の位置に
         // 指定のテキストを表示
      ),
      H List Box( Slider Box( 0, 4, lU, gr << Reshow ), Text Box( "下限" ) ),
      // 最小と最大の範囲を定義
      // lUの値を挿入し、スライダが動いたら表示を更新
      H List Box(
         Slider Box( 0, 4, lS, gr << Reshow ),
         Text Box( "上限" )
      ),
      Button Box( "再表示", gr << Reshow )
   )
);
```

```
Show( gr );
gr << Reshow; // グラフボックスの表示を更新
```

図11.14 レポートウィンドウ内のスライダとボタンの例

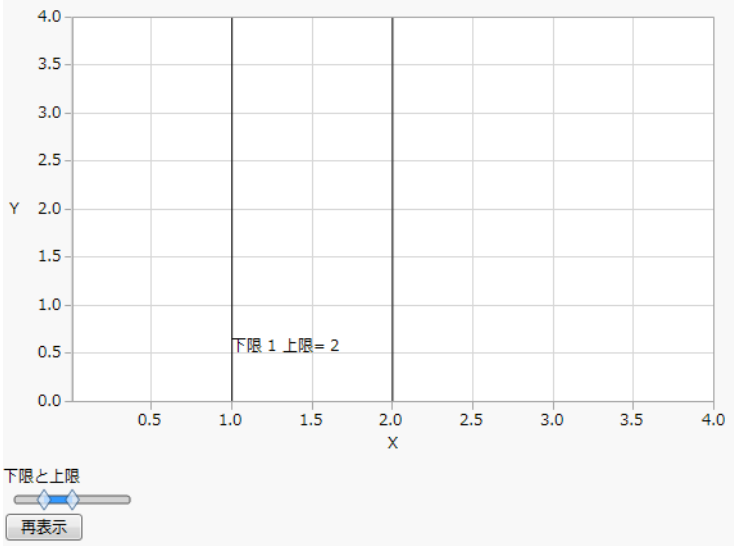


次のスクリプトは、2つのSlider Box()と同じ機能を果たすRange Slider Box()の例です。

```
lLow = 1;
lHigh = 2;
win = New Window( "範囲スライダ",
   V List Box(
      gr = Graph Box(
         Frame Size( 500, 300 ),
         X Scale( 0.01, 4 ),
         Y Scale( 0, 4 ),
         XAxis( "Show Major Grid" ),
         YAxis( "Show Major Grid" ),
         X Function( lLow, x );
         X Function( lHigh, x );
         Text( {1, .5}, "下限= ", lLow, " 上限= ", lHigh );
      ),
      V List Box(
         Text Box( "下限と上限" ),
         sb = Range Slider Box( 0, 4, lLow, lHigh, gr << Reshow )
      )
   ),
   Button Box( "再表示",
      lLow = 1;
      lHigh = 2;
```

```
        gr << Reshow;
        sb << Reshow;
    )
);
Show( gr );
gr << Reshow;
```

図11.15 Range Slider Boxの使用例

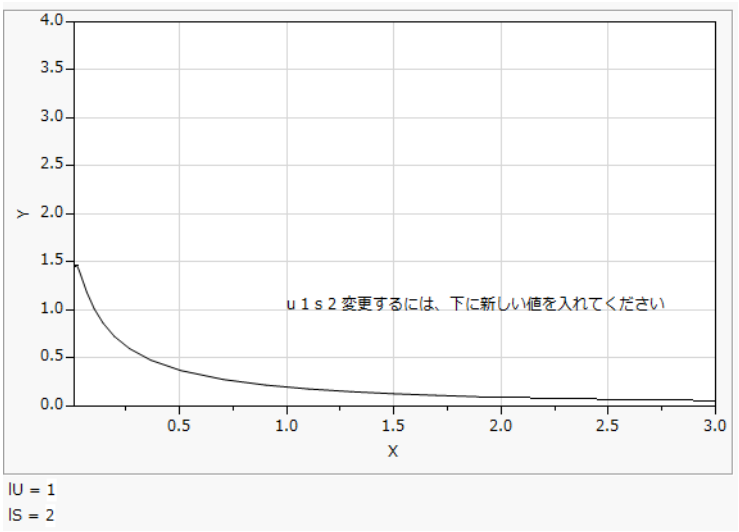


下のスクリプトは、Slider Box()のコントロールではなく、Global Box()を編集可能なテキストボックスとして使用しています。

```
// Global Box を使った対数正規分布
lU = 1;
lS = 2;
win = New Window( "対数正規密度",
    V List Box(
        gr = Graph Box( // Graph Box の引数とスクリプトを定義する
            Frame Size( 500, 300 ),
            X Scale( 0.01, 3 ),
            Y Scale( 0, 4 ),
            XAxis( "Show Major Grid" ),
            YAxis( "Show Major Grid" ),
            Y Function(
                Exp( -(Log( x ) - Log( lU )) ^ 2 / (2 * lS ^ 2) ) / (lS * x *
                Sqrt( 2 * Pi() )),
                x
            );
```

```
            Text( {1, 1}, "u= ", lU, " s= ", lS, " 変更するには、下に新しい値を入れてく
        ださい " );
            // X軸とY軸の1の位置に指定のテキストを表示
        ),
        H List Box( Global Box( lU ) ),
        // 最初のH List BoxにIU変数の値を表示
        H List Box( Global Box( lS ) )
        // 2番目のH List BoxにIS変数の値を表示
    )
);
```

図11.16 スライダの代わりにグローバルボックスを使用した例



「スクリプトによるグラフ作成」章の「Drag関数」（565ページ）の例で、Button Boxの別の使い方を示しています。

カレンダーセレクタと日付セレクタの例

カレンダーセレクタまたは日付セレクタは、Calendar Box()とNumber Edit Box()で作成します。

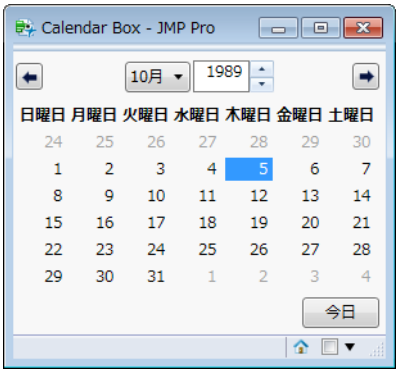
Calendar Box

Calendar Box()を使うと、カレンダーで年を入力し、日付を選択することができます。次の例は、1989年10月5日が選択されている状態のカレンダーを作成します。1989年10月5日の前後60日間から任意の日付を選択できます。

```
win = New Window( "Calendar Box", cal = Calendar Box() );
date = Date MDY( 10, 5, 1989 );
cal << Date( date );
cal << Show Time( 0 ); // 時間は省略
```

```
cal << Min Date( Date Increment(date, "Day", -60, "start" ) );
// 選択範囲の最初の日付は 1989/10/05 から 60 日前
// "start" で時間の値を省略

cal << Max Date( Date Increment(date, "Day",  60, "start" ) );
// 選択範囲の最後の日付は 1989/10/05 から 60 日後

cal << Set Function( Function( {this}, Print( Abbrev Date(this << Get Date()) ) )
  );
// 短い表示形式で日付をログに出力
```

図11.17 カレンダーの作成



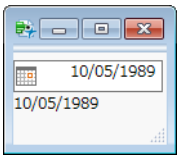
Number Edit Box

Number Edit Box() を使うと、ボックスに日付を入力するか、カレンダーボタンをクリックしてカレンダーから日付を選択することができます。

次の例は、Number Edit Box() を使って対話型のカレンダーを作成します。ユーザーは、ボックスに日付を入力するか、カレンダーをクリックして日付を選択します。Number Edit Box の下のテキストボックスに日付が表示されます。最初は “1989/10/05” が選択されています。

```
f = Function( {this}, // コールバック
   textbox << Set Text( Format( this << Get, "m/d/y" ) ) // 日付を取得し Text Box に設定
);
win = New Window( " 日付の選択 ",
   numbox = Number Edit Box( 0, 20, << Set Function( f ) ),
   textbox = Text Box( "" ) // 指定した日付を入れる Text Box
);
numbox << Set Format(
   Format( "m/d/y", 12 )
   // 日付の形式と表示される文字数
   );
numbox << Set( Date MDY( 10, 5, 1989 ) ); // デフォルトの日付と形式
```

図11.18 日付セレクタの作成



変数fにコールバックが割り当てられています。コールバックの中のthisは、後で関数を呼ぶことになるNumber Edit Boxを参照します。thisを使用することで、コールバック内でnumbox変数を使用する必要がなくなります。Getメッセージはthisに送られ、指定された日付を取得します。

要約結果のレポート作成の例

次のスクリプトは、Summarize() 関数を使って、「Big Class.jmp」の「**身長(インチ)**」列に関する要約統計量を収集し、ディスプレイボックスを構成する関数を使って見やすく整理した結果をウィンドウに表示します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
Summarize(
   a = by( 年齢 ),
   c = count,
   sumHt = Sum( Name("身長(インチ)")),
   meanHt = Mean( Name("身長(インチ)")),
   minHt = Min( Name("身長(インチ)") ),
   maxHt = Max( Name("身長(インチ)") ),
);
win = New Window( " 要約の結果 ",
   Table Box(
      String Col Box( " 年齢 ", a ),
      Number Col Box( "N", c ),
      Number Col Box( " 合計 ", sumHt ),
      Number Col Box( " 平均 ", meanHt ),
      Number Col Box( " 最小値 ", minHt ),
      Number Col Box( " 最大値 ", maxHt )
   )
);
```

これで、図11.19のような結果の要約が作成されます。

図11.19 カスタマイズした要約レポートの生成

年齢	N	合計	平均	最小値	最大値
12	8	465	58.125	51	66
13	7	422	60.2857	56	65
14	12	770	64.1667	61	69
15	7	452	64.5714	62	67
16	3	193	64.3333	60	68
17	3	200	66.6667	62	70

ディスプレイボックスに、通常のコマンドを使用できます。次のスクリプトは、レポートをMicrosoft PowerPointプレゼンテーションとして保存します。

```
Show Properties( win );
win << Save Presentation( "$DOCUMENTS/Summary.pptx" );
```

ウィンドウを閉じる操作

ユーザは、［OK］ボタンや［キャンセル］ボタンをクリックして、またはウィンドウの上隅にある赤色の［閉じる］ボタンをクリックしてウィンドウを閉じることができます。また、メッセージを送ってウィンドウを閉じることもできます On Validate と On Close には、ウィンドウの操作時に行うことを指定します。

- On Validate は、**［OK］ボタンがクリックされる**と、式を評価します。他のボタンをクリックしたり、他の操作を行ったりしても式は評価されません。式の評価が真の場合、ウィンドウは閉じます。式の評価が偽の場合、ウィンドウは閉じません。［OK］ボタン用のスクリプトは、On Validate の式とは無関係に、［OK］ボタンがクリックされると必ず実行されます。
- On Close は、**ウィンドウが閉じる直前に**式を評価します。On Close で戻される値は、On Validate の戻り値と同様に、ウィンドウを開いたままにする場合があります。しかし、On Close を使う際の最良の方法は、常に1を戻すようにすることです。そうすれば、閉じないウィンドウを作成してしまうことがありません。

  ウィンドウが閉じるとき、On Close が評価されます。On Validate の式が偽を戻した場合はウィンドウが閉じないので、On Close は評価されません。

メモ:

- On Close は、モーダルウィンドウと非モーダルウィンドウのどちらでも機能します。On Validate は、モーダルウィンドウでのみ機能します。
- On Close は［OK］と［キャンセル］を区別しないので、検証に使うことはできません。On Close は、ウィンドウが閉じるときに必ず実行されるので、リソース（非表示のデータテーブルなど）の整理に利用できます。
- On Validate をリソースの整理に利用しないでください。On Validate は、［キャンセル］ボタンまたは赤色の［閉じる］ボタンでは実行されないため、代わりに On Close を使用するか、On Validate に加えて On Close を使用してください。

On Close() の例

ノンモーダルウィンドウであるレポートウィンドウがあり、整理する必要のあるリソースとして非表示のデータテーブルがあるとしましょう。On Close スクリプトが、そのデータテーブルを閉じます。ユーザはデータテーブルが開いていることに気づかない場合が多く、たとえ気づいたとしても簡単に閉じる方法はないため、この方法はおすすめです。

```
dt = New Table( "無題", << Invisible,
    New Column( "x", Set Values( [1, 2, 3, 4] ) ),
    New Column( "y", Set Values( [4, 2, 3, 1] ) )
);

dt << Bivariate(
    y( dt:y ), x( dt:x ), <<On Close( Close( dt, "nosave" ) )
);
```

スクリプトが実行される前、開いているデータテーブルはありません。「無題」という非表示のテーブルが作成され、ホームウィンドウのデータテーブルリストにあるスクリプトの上に表示されます。次にグラフが作成されます。レポートウィンドウを閉じると、On Close によって非表示のテーブルが閉じられます。

On Validate() の例

モーダルウィンドウに質問が表示され、ユーザが回答しないとウィンドウが閉じない仕組みになっているとします。次の例は、ユーザがウィンドウを閉じるときに入力値が検証されるようにする方法を示します。

```
win = New Window( "検証例",
   <<Modal,
   <<Return Result,
   <<On Validate(
      If( (!Is Missing( variablebox << Get ) & 40 <= variablebox << Get <= 50),
         Return( 1 ), // そうでない場合は、ユーザが正しくない回答を入力した
         tb << Set Text(
            Match( Random Integer( 1, 3 ),
               1, "本当にそう思いますか?",
               2, "もう一度考えてみましょう。",
               3, "質問を読んでください。"
            )
         );
         Return( 0 );
      )
   ),
   Text Box( "私は 40 ～ 50 の数字のうち 1 つを思い浮かべました。当ててください。" ),
   variablebox = Number Edit Box( . ),
   H List Box( Button Box( "OK" ), Button Box( "キャンセル" ) ),
   tb = Text Box( "" )
);
```

```
New Window( "モーダルウィンドウ",
   <<Modal,
   Text Box(
      If(
         win["Button"] == -1, "調子のいいときにもう一度挑戦してください。",
         // キャンセルボタンまたは赤色の閉じるボタンをクリックした場合
         win["variablebox"] == 42, "よくできました！", // 42 を入力した場合
         "もう一度挑戦してください。" // 42 以外の値を入力して OK をクリックした場合
      )
   )
);
```

既存の表示の更新

作成されるレポートに表示されるディスプレイボックスの数がわからない場合があります。たとえば、1つまたは複数の変数を分析してレポートする一般的なスクリプトを作成する場合などです。スクリプトを実行するたびに変数の数が変わる可能性があるので、必要なディスプレイボックスの数はわかりません。

以下の節では、Append()、Prepend()、Delete()、およびSib Append()を使用して、レポートのディスプレイボックスを追加したり削除したりする方法を説明します。

Append

Appendメッセージを使うと、既存の表示の末尾にディスプレイボックスを追加できます。スクリプトで、空のボックスを1つ作成し、Appendを使って分析の各変数用にそのボックスを追加します。Appendメッセージを使用して、既存の表示に独自の情報や組織名を追加することもできます。

次のコード例では、変数effectsList内に有効な名前のリストがあり、その名前が行列varprop内の列に対応していることを前提としています。つまり、effectsList[1]はvarprop[0,1]のラベルに、effectsList[2]はvarprop[0,2]のラベルにというように順次対応します。

```
varprop = [0 1 2, 3 4 5, 6 7 8];
effectsList = {"いち", "に", "さん"};
```

まず、H List Boxを含む空のOutline Box（アウトラインボックス）が作成されます。内部の空のコンテナにはhbという名前が付けられます。

```
win = New Window( "H List Box の例",
   Outline Box( "分散", hb = H List Box() )
);
```

次に、effectsListの項目数分だけforループを繰り返し、effectsListの各要素に対応したNumber Col Box（数値列ボックス）を追加します。

```
For( i = 1, i <= N Items( effectsList ), i++,
   Eval(
      Substitute(
            Expr(
```

```
                hb << Append(
                   Number Col Box( effectslist[i], varprop[0, i] )
                )
             ),
          Expr( i ), i
       )
    )
);
```

Sib Append() は、Append() に似ていますが、ディスプレイボックスを最後の子ディスプレイボックスの兄弟として作成します。図11.21に、Append() と Sib Append() を比較する例があります。

Prepend

Prepend メッセージは Append と同様に動作しますが、項目をディスプレイボックスの最後にではなく先頭に追加します。ディスプレイボックスが追加不可なタイプの場合、その子ディスプレイボックスのうち追加が可能なものに、Prepend メッセージが送られます。このため、表示ツリーの先頭にコマンドを送っても問題ありません。

たとえば、次の例は最上部にボタンボックスのある「二変量」レポートを作成します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
biv = dt << Bivariate( Y( :Name("身長(インチ)") ), X( :Name("体重(ポンド)") ), Fit
  Line );
(biv << Report)(Outline Box( 1 )) << Prepend(
// biv レポートレイヤーの最初のアウトラインボックスに、
// Prepend メッセージを送る
      Button Box( "2次曲線", biv << Fit Polynomial( 2 ) )
// 2次曲線をあてはめる Button Box を追加
);
```

「2次曲線」ボタンをクリックすると、グラフに2次曲線が追加されます。

ツリーの最上部にこのボタンを追加しても、同じ結果となります。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
biv = dt << Bivariate( Y( :Name("身長(インチ)") ), X( :Name("体重(ポンド)") ), Fit
  Line );
biv << Report << Prepend( // レポートに prepend メッセージを送る
   Button Box( "2次曲線", biv << Fit Polynomial( 2 ) )
   // Button Box をレポートの先頭に追加
);
```

Delete

Delete メッセージは、指定されたディスプレイボックスとその子をすべてレポートから削除します。動的な表示を作成する場合は、このメッセージと Append および Prepend メッセージを一緒に使うと便利です。次の例では、テキストボックスを別のテキストボックスに置き換えます。このケースでは Set Text を使う方法もありますが、他の大半のディスプレイボックスは内容を変更できません。

```
win = New Window( "X",
   list = V List Box(
      t1 = Text Box( "t1" ),
      t2 = Text Box( "t2" )
   )
);
t1 << Delete;
list << Append( t1 = Text Box( "t1new" ) );
```

メモ: JMPのプラットフォームからは、ディスプレイボックスを削除しないようにしてください。Hide(1) メッセージまたはVisibility("Hidden")メッセージを、非表示にしたいディスプレイボックスに送ります。

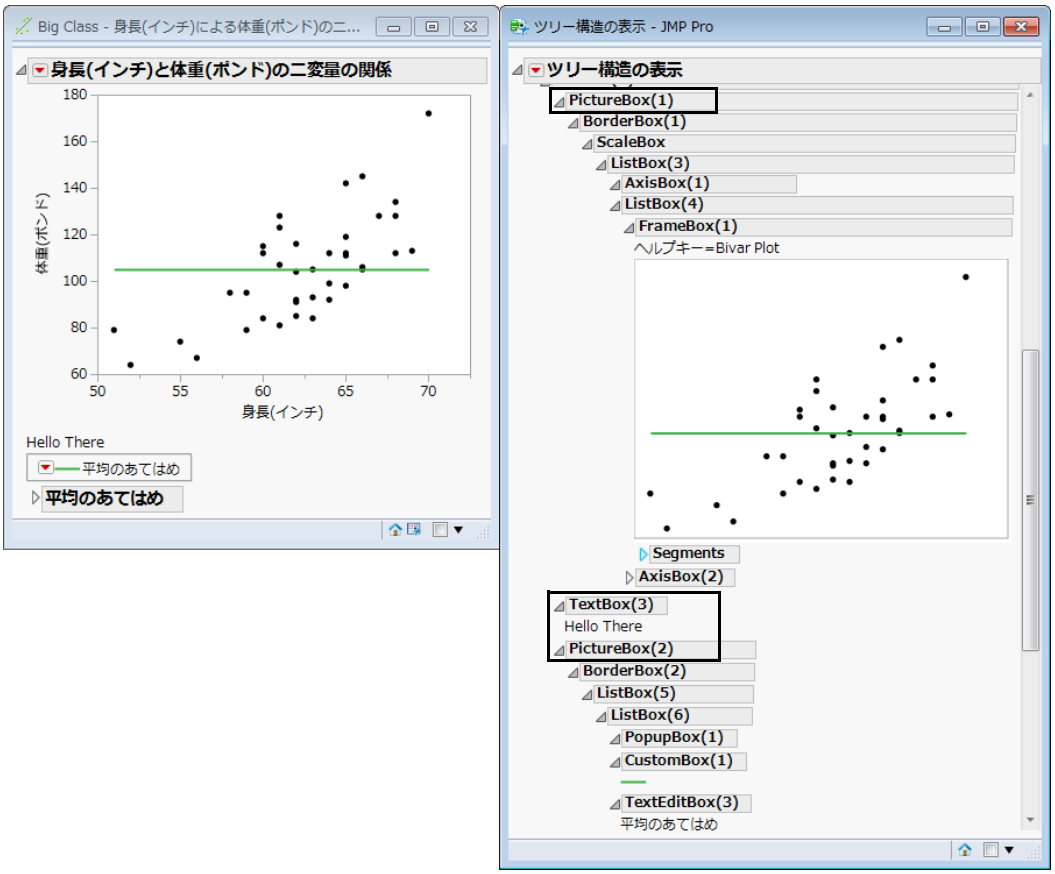
Sib Append

Sib Appendメッセージを使うと、既存のディスプレイボックスのすぐ後ろにディスプレイボックスを追加できます。図11.20は、2つのピクチャーボックスツリーを示します。PictureBox(1)には、二変量の関係の散布図が含まれています。Picture Box (2)には、緑の線と「平均のあてはめ」というテキストボックスが含まれており、これらは「平均のあてはめ」メニューに対応しています。

これらの2つのボックスの間にテキストボックスを挿入したいとします。Picture Box (1)と同じレベルにボックスを追加する必要があるので、Sib Appendメッセージを送ります。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
biv = dt << Bivariate(
   Y( :weight ),
   X( :height ),
   Fit Mean( {Line Color( {57, 177, 67} )} )
);
Report( biv )[Picture Box( 1 )] << Sib Append( Text Box( "Hello There" ) );
/* Report関数はbivオブジェクトからレポートオブジェクトを戻す。
最初のPicture Boxのすぐ後ろにText Boxを追加する */
```

図11.20 同じレベルにテキストボックスを追加



メモ: 図11.20では、追加された兄弟テキストボックスが「ツリー構造の表示」ウィンドウの最初のピクチャーボックスの下に表示されています。従来の「ツリー構造の表示」ウィンドウのディスプレイツリーでは、兄弟は並べて表示されます。詳細は、「ディスプレイツリーの表示」（416ページ）の節を参照してください。

図11.21は、`Append()`と`Sib Append()`の違いを示しています。`Append()`は、ディスプレイボックスを親ディスプレイボックスの**子**として作成します。`Sib Append()`は、ディスプレイボックスを最後のディスプレイボックスの**兄弟**として作成します。

図11.21 Append（左）とSib Append（右）

メモ: 親ディスプレイボックスがV List Box()ではない場合、Sib Append()は、そのディスプレイボックスをV List Box()に入れます。

Set FunctionとSet Script

Set Scriptメッセージを使って、ディスプレイボックスのコントロール（Button BoxやCombo Box）がクリックされたときにスクリプトを実行させることができます。

```
win = New Window( "Set Scriptの例",
   ex = Button Box( "クリックしてください" )
);
ex << Set Script( Print( "クリックされました" ) );
```

上のスクリプトは、ボタンボックスがクリックされたときに、次のテキストをログに出力します。

"クリックされました"

または、Set Functionメッセージを使うと、ディスップレイボックスのコントロールに、ある特定の関数を実行させることができます。この関数の最初の引数はそのディスプレイボックスを表します。Set Functionは、オブジェクト指向のスクリプトと大きなプログラムの作成を可能にします。

```
win = New Window( "Set Functionの例",
   Button Box( "クリックしてください",
      << Set Function(
         Function(
            {this},
         // Set Functionで指定した関数が'this'ディスプレイボックスを取得する
            this << Set Button Name( "ありがとう" )
         )
      )
   )
);
```

上のスクリプトは、「クリックしてください」という名前のButttton Boxを作成します。このボタンがクリックされたとき、Set Functionによって呼び出された関数がボタンの名前を「ありがとう」に変更します。

同じスクリプトで複数のボタンを操作したい場合、Set Functionを使うとSet Scriptより簡単です。なぜなら、Set Functionは送り先を記述しないで済むからです。この例では、どちらのチェックボックスも、1行目で始まるスクリプトを使用します。

```
f = Function( {this, idx}, // idx を変更
   Write(
/* <<Get Items はすべての項目を戻す。リスト内で idx が付いたものは
   変更された項目の名前
   <<Get はその項目の新しい値を戻す */
      "\!n changing item=" || (this << Get Items())[idx] || " to " || Char( this <<
   Get( idx ) ) ||
      "\!n new selection=" || Char( this << Get Selected ) || "\!n"
   )// <<Get Selected は現在チェック（選択）されている項目のリストを戻す
);
New Window( "例",
   H List Box( // f' は名前付き関数、ここでは 2 回使用される
      V List Box( Text Box( "Column 1" ), Check Box( {"a", "b"}, <<Set Function( f
   ) ) ),
      Spacer Box( size( 50, 50 ) ),
      V List Box( Text Box( "Column 2" ), Check Box( {"c", "d"}, <<Set Function( f
   ) ) )
   )
);
```

次に紹介するSet Functionの例は、同じ関数を使ってボタンのアクションを定義しています。

```
New Window( "貯金箱",
   V List Box(
      H Center Box( TB = Text Box() ), // TB がコインの金額の合計を表示
      LB = Lineup Box( N Col( 3 ) ), // LB が、ボタンを整列させる
      H Center Box( // クリアボタンが金額をリセット
         Button Box( "クリア",
            total = 0;
            TB << Set Text( Char( total ) );
         )
      )
   )
);
coins = {1, 5, 10, 25, 50, 100}; // coins はボタンのラベルを持つリスト
total = 0;
For( iButton = 1, iButton <= N Items( coins ), iButton++,
   LB << Append( Button Box( Char( coins[iButton] ), Set Function( buttonFunction )
   ) )
); /* ループがボタンを作成し、それぞれに同じ関数が使用される */
```

```
buttonFunction = Function( {this},
   total = total + Num( this << Get Button Name );
   TB << Set Text( Char( total ) );
);
// クリア以外のボタンに呼び出される関数は
// ボタンの名前からアクションを特定する。ボタン名を使う代わりに
// 兄弟のディスプレイボックスを使用するため、this<<sib または
// (this<<parent)<<child でこのボタンの最初の兄弟を見つけることもできる
```

メモ:

- Set Script と Set Function の各メッセージは、Button Box、Calendar Box、Check Box、Combo Box、List Box、Popup Box、Radio Box、Range Slider Box、Slider Box、Spin Box に対応します。
- Set Script と Set Function は同時に使用できません。特定のディスプレイボックスを参照する場合は、Set Function を使用してください。

リストを戻す表示要素の選択された値を取得・設定する

Set Selected (Item Number, <State>, <Run Script(0|1)>) メッセージを使うと、項目を最初から選択された状態にすることができます。このメッセージは、保存されているディスプレイボックス参照に対して単独で使用するか、List Box の << メッセージとして使用できます。

選択された値を取得するには、Get Selected メッセージ使用します。これは、選択された項目の値を戻します。Get Selected Indices メッセージは、選択された項目のインデックス番号を戻します。

```
antennaList = {" パラボラ "," ヘリカル "," 極性 "," ラジアントアレイ "};

win = New Window( " リストのテスト ", // 方法 1: ディスプレイボックスの参照
   listObj = List Box(
      antennaList,
      Print(
         "iList",
         listObj << Get Selected,
         listObj << Get Selected Indices
      )
   )
);
listObj << Set Selected( 2, 1 );

win = New Window( " リストのテスト ", // 方法 2: インライン
   listObj = List Box(
      antennaList,
      <<Set Selected( 2, 1 ),
      Print(
         "iList",
         listObj << Get Selected,
```

```
            listObj << Get Selected Indices
         )
      )
   );
```

この2つのスクリプトは、どちらも次のテキストをログに出力します。

```
"iList"
{"Helical"}
{2}
```

前述の例では、Set Selectedメッセージが完了すると、Print式が実行されます。スクリプトが実行されないようにするには、最後の引数としてRun Script(0)を含めます。Run Script(0|1)は、ディスプレイボックスの変化に反応するスクリプトをSetまたはSet Selectedメッセージの後で実行するかどうかを制御します。

```
antennaList = {" パラボラ "," ヘリカル "," 極性 "," ラジアントアレイ "};
win = New Window( " リストのテスト ",
   listObj = List Box(
      antennaList,
      Print(
         "iList",
         listObj << Get Selected,
         listObj << Get Selected Indices
      )
   )
);
listObj << Set Selected( 2, 1, Run Script( 0 ) );
```

Run Script(1)を指定すると、Setメッセージの完了後にスクリプトが実行されます。値に変化がない場合でも実行されます。(ユーザが同じ値を選択した場合は、スクリプトは実行されません。) Run Script(0)を指定すると、スクリプトは実行されません。

インタラクティブなディスプレイボックスのほとんどでは、Run Script()を使用しない場合、スクリプトは実行されません。ただし、List Box()では、以前の動作と同様、デフォルトでスクリプトが実行されます。

作成した表示にメッセージを送る

作成したウィンドウに名前を割り当てると、その名前は、そのディスプレイボックスを所有するウィンドウへの参照になります。その後で、添え字を使って、そのウィンドウ内のディスプレイボックスにメッセージを送ることができます。

たとえば、次のスクリプトは、インタラクティブな正弦波を作成します。このスクリプトは、ウィンドウ内のフレームボックスにメッセージを送ることで、正弦波を自動的に変化させています。(スクリプトの途中で*tf*に割り当てています。)

```
amplitude = 1;
freq = 1;
phase = 0;
win = New Window( "揺れ動く波",
   Graph Box(
      Frame Size( 500, 300 ),
      X Scale( -5, 5 ),
      Y Scale( -5, 5 ),
      Y Function( amplitude * Sine( x / freq + phase ), x );
      Handle(
         phase,
         amplitude,
         /* 四角いハンドルのフェーズおよび振幅位置の現在値。
            スクリプトは振幅（amplitude）の後のカンマの後始まる */
            phase = x;
            amplitude = y;
            /* X と Y はハンドルの位置に設定されたが、
            最初の 2 つの引数で指定された値が更新されない限り
            ハンドルは動かない */
      );
      Handle( freq, .5, freq = x );
      // Handle は同様に動作するが、y が 0.5 という点が異なる
      Text( // グラフにテキストを表示する
         {3, 4},
         "amplitude: ",
         Round( amplitude, 4 ), // 現在値を表示する
         {3, 3.5},
         "frequency: ",
         Round( freq, 4 ), // 現在値を表示する
         {3, 3},
         "phase: ",
         Round( phase, 4 ) // 現在値を表示する
      );
   )
);
tf = win[Frame Box( 1 )]; // Frame Box（グラフ）を取得
For( amplitude = -4, amplitude < 4, amplitude += .1,// 振幅のアニメーション
   tf << Reshow // グラフを強制的に更新
);
amplitude = 1; // ループに使用して複雑な動きをさせる
freq = 1;
phase = 0;
For( i = 0, i < 1000, i++,
   amplitude += (Random Uniform() - .5);
   amplitude = If(
      amplitude > 4, 4,
```

```
        amplitude < -4, -4,
        amplitude
    );
    freq += (Random Uniform() - .5) / 20;
    phase += (Random Uniform() - .5) / 10;
    tf << Reshow;
    Wait( .05 );
);
```

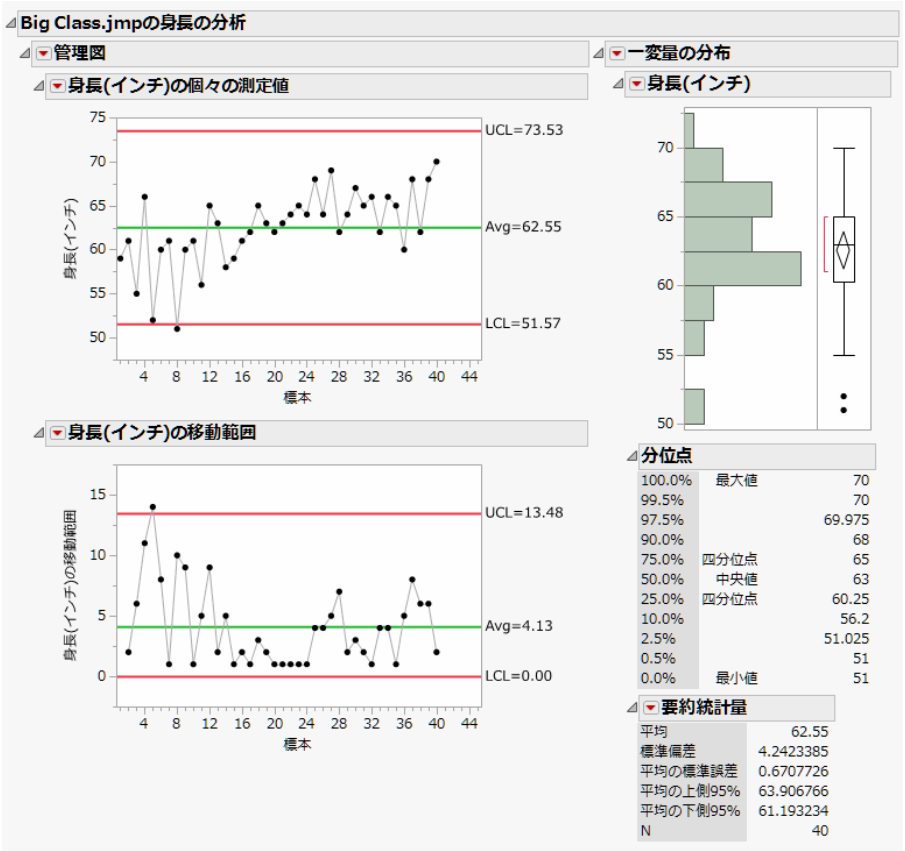
プラットフォームを含むディプレイボックスを作成する

JMP の分析プラットフォームの結果を組み合わせたレポートを作りたい場合もあるでしょう。ディスプレイボックスの中にプラットフォームスクリプトを作成し、ディスプレイボックスを組み合わせてウィンドウの中に入れます。後でメッセージを簡単に送れるように、全体を1つの参照に割り当てます。

次の例では、複数のグラフとレポートを1つのウィンドウ内に作成します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
win = New Window( "プラットフォーム例",
   Outline Box( "Big Class.jmp の身長の分析",
      H List Box(
         cc = Control Chart(
            Chart Col( :Name("身長 ( インチ )"), "個々の
            測定値", "移動範囲" ),
            K Sigma( 3 )
         ),
         dist = Distribution(Column(:Name(" 身長 ( インチ )")));
      )
   )
);
```

図11.22 例：1つのレポートウィンドウ内に表示された複数のグラフ



先ほどのスクリプトを実行した後、参照 *csp* に対してメッセージを送ると、ウィンドウを処理できます。この例における csp はディスプレイボックスの参照で、プラットフォームに対する Report の機能に似ています。csp に対して複数の添え字を使うと、アウトラインツリー内の特定の項目を見つけることができます。

```
csp[" 管理 ?", "? 移動範囲 "] << Close;
csp["? 分布 ", " 分位点 "] << Close;
```

前述の例では、ウィンドウ全体を参照（csp）に割り当てるだけでなく、プラットフォーム起動スクリプトをディスプレイボックス内の名前（cc および dist）に割り当てています。これらの参照を使えば、プラットフォームへメッセージを簡単に送れます。ディスプレイボックスを操作する前述の例とは別の方法として、プラットフォームからレポートを取得する方法もあります。次のスクリプトでは、プラットフォームからレポートを取得して、ノードを再度開きます。

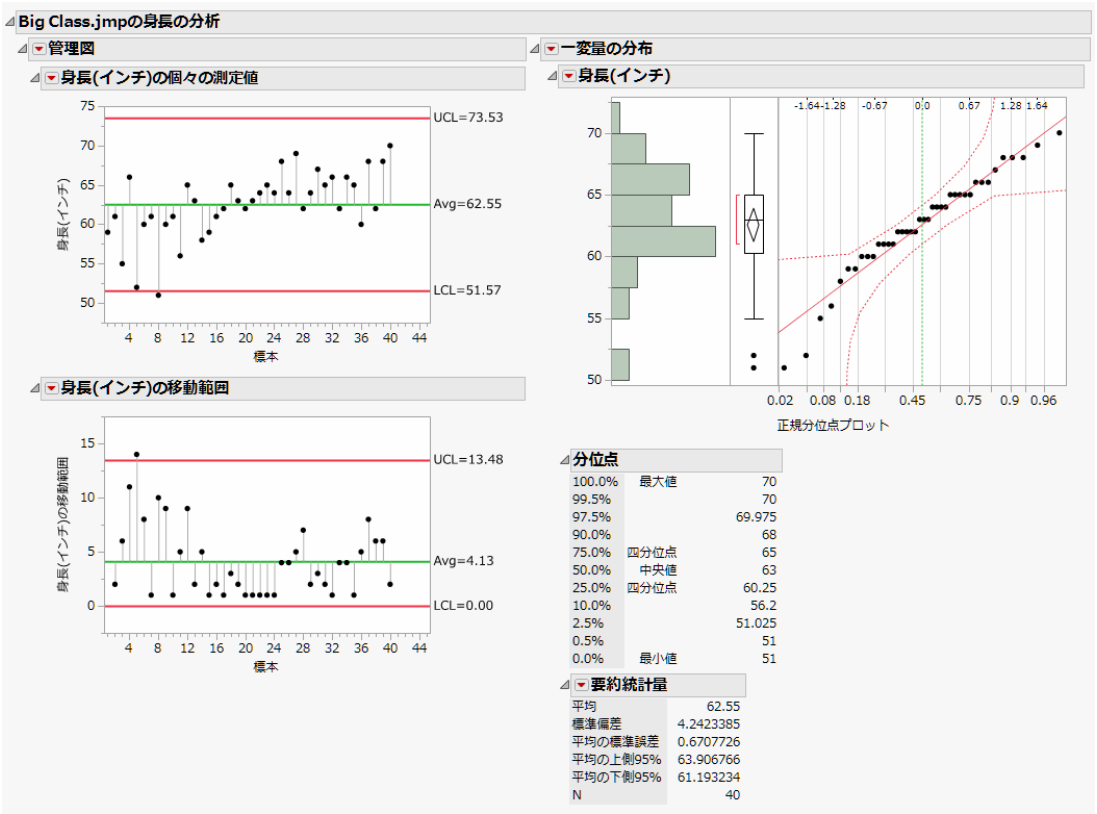
```
rcc = cc << Report;
rdist = dist << Report;
rcc["? 移動範囲 "] << Close;
rdist[" 分位点 "] << Close;
```

メッセージを直接プラットフォームの参照に送ることができます。JMPの［スクリプトの索引］で「Distribution」や「Control Chart」を検索し、どのようなオプションがあるかを確認してください。たとえば、Control Chartでは、Needleというメッセージで垂線グラフを作成できます。Distributionには、Normal Quantile Plotというメッセージがあります。

JSLで実行するには、これらのオプションをメッセージとしてプラットフォーム参照に送ります。

```
cc << Needle;
dist << Normal Quantile Plot;
```

図11.23 カスタムレポートの変更



2つのレポートからダッシュボードを作成する例

ダッシュボードは、定期的にレポートを実行し、表示する視覚的ツールです。ダッシュボードには、レポート、データフィルタ、選択フィルタ、データテーブル、グラフを表示することができます。ダッシュボードの内容は、ダッシュボードを開いたときに更新されます。

レポートウィンドウでは、レポートがタブページボックスに表示されます。レポートの位置を変更するには、タブのタイトルをクリックし、タブページを強調表示されたドロップゾーンにドラッグします。強調表示されたドロップゾーンは、レポートを横、縦、タブの形式で並べる方法を示唆します。有効なドロップゾーンが選択されていない場合は、ボックスが元の位置に戻ります。

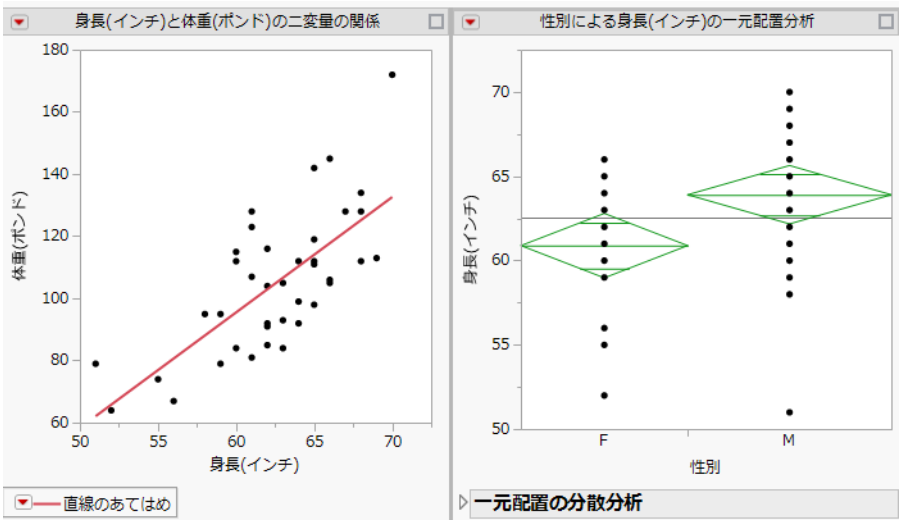
JMPでは、ダッシュボードビルダーでダッシュボードを作成します。詳細については、『JMPの使用法』の「JMPの拡張」章を参照してください。この節では、ダッシュボードを作成するためのスクリプトの記述方法を紹介します。

1段のレポートで構成されたダッシュボード

この例では、スクリプトで2つのレポートを並べたダッシュボードを作成します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
New Window( "マイダッシュボード",
   H Splitter Box( // レポートを Splitter Box に入れる
   Size( 700, 400 ),
      Tab Page Box( // レポートをタブページボックスに入れる
         "身長(インチ)と体重(ポンド)の二変量の関係",
         dt << Bivariate( // 二変量レポート
            Y( :weight ),
            X( :height ),
            Fit Line,
            Report View( "Summary" ) // グラフだけ表示する
         ),
         <<Moveable( 1 ) // レポートを移動可能にする
      ),
      Tab Page Box(
         "性別による身長(インチ)の一元配置分析",
         dt << Oneway( // 一元配置レポート
            Y( :Name("身長(インチ)")),
            X( :性別 ),
            Means( 1 ),
            Mean Diamonds( 1 ),
            Report View( "Summary" )
         ),
         <<Moveable( 1 )
      ),
      /* タブボックスを別の位置にドラッグすることができる
         ディスプレイボックスを移動可能とするために必要 */
      <<Dockable( 1 )
   )
);
```

図11.24 2つのレポートを含むダッシュボード



メモ: ドッキング可能で移動可能なタブにプラットフォームが1つだけ含まれている場合、最上位のOutline Boxはタブのタイトルに包含されます。そのため、赤い三角ボタンのオプションはタブのタイトルバーに表示されます。

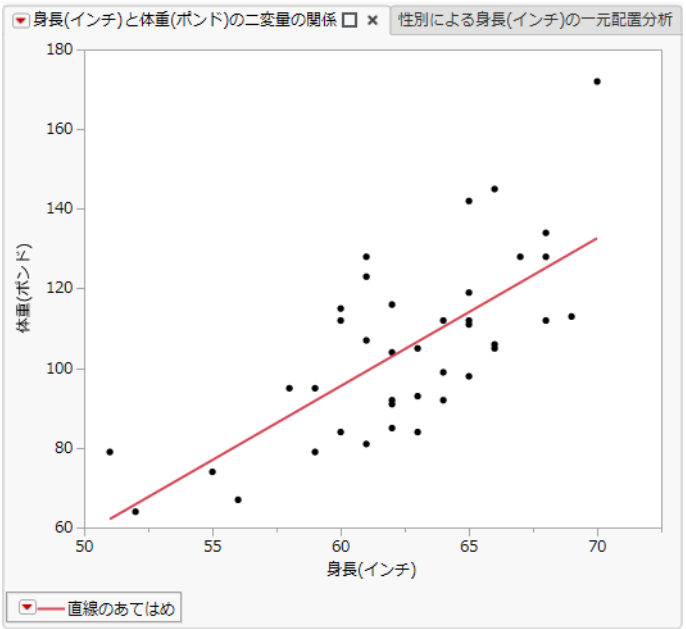
タブ形式のダッシュボード

次のスクリプトは、レポートを一つずつタブに配置したダッシュボードを作成します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
New Window( "マイダッシュボード",
   Tab Box( // レポートを Tab Box に入れる
      Tab Page Box( // レポートを Tab Page Box に入れる
         "身長(インチ)と体重(ポンド)の二変量の関係",
         dt << Bivariate( // 二変量レポート
            Y( :weight ),
            X( :height ),
            Fit Line,
            Report View( "Summary" ) // グラフだけ表示する
         ),
         <<Moveable( 1 ), // レポートを移動可能にする
         <<Closeable( 1 ) // レポートを閉じられるようにする
      ),
      Tab Page Box(
         "性別による身長(インチ)の一元配置分析",
         dt << Oneway( // 一元配置レポート
            Y( :Name("身長(インチ)")),
            X( :性別 ),
```

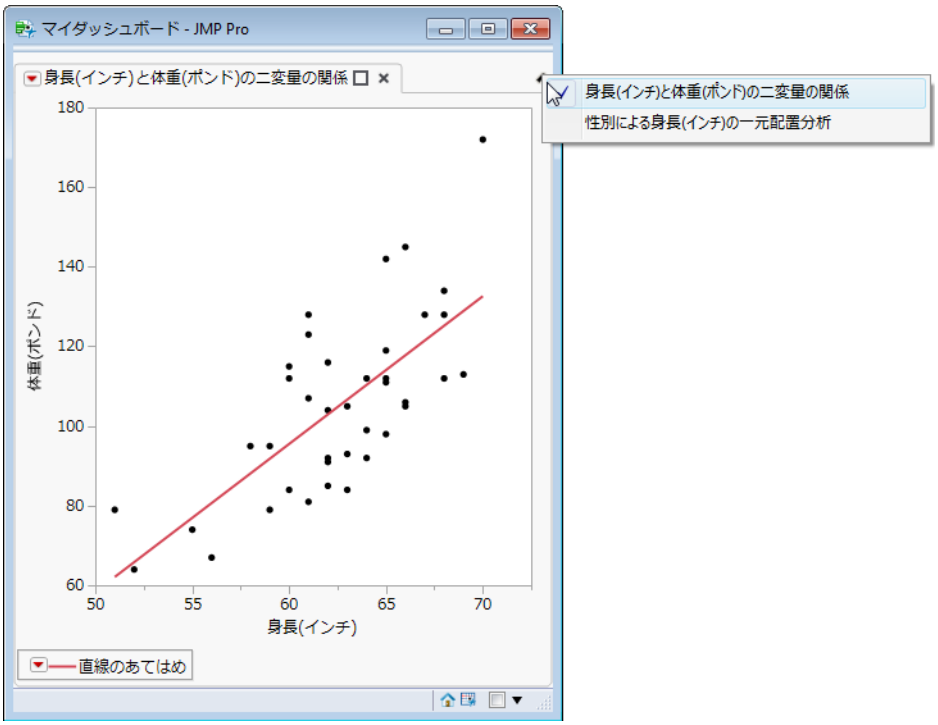
```
            Means( 1 ),
            Mean Diamonds( 1 ),
            Report View( "要約" )
        ),
        <<Moveable( 1 ),
        <<Closeable( 1 )
    ),
    <<Dockable( 1 ),
    /* タブボックスを別の位置にドラッグすることができる
       ディスプレイボックスを移動可能とするために必要 */
    <<Set Overflow Enabled( 0 ) // 両方のタブページが表示される
  )
);
```

図11.25 タブ形式のダッシュボード



複数のタブで構成されたダッシュボードでは、タブ形式になったすべてのレポートが表示されるよう、ウィンドウの幅が調整されます。最初のレポートだけを表示し、その他のレポートはユーザーにリストから選んでもらうこともできます。その場合は、スクリプトの終わりの方にある`Set Overflow Enabled( 0 )`を`Set Overflow Enabled( 1 )`に変更します。図11.26はこの結果です。

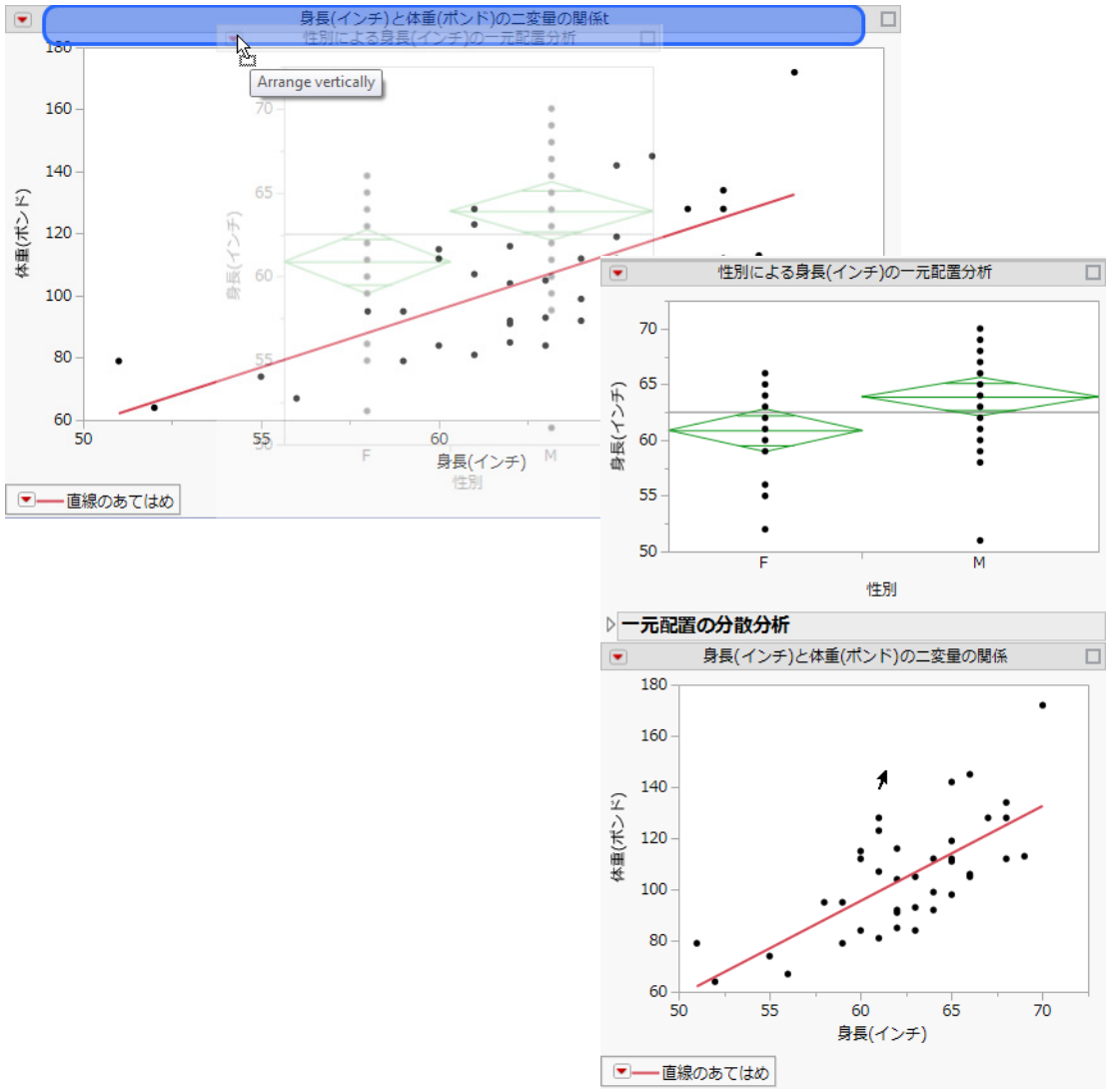
図11.26 オーバーフロータブ形式のダッシュボード



タブをタブボックスにドッキングさせる方法

ドッキングレイアウトとは、Tab Page Box()オブジェクトとドッキングコンテナをまとめ、レポートの並べ替えを可能にしたレイアウトです。レポートウィンドウでレイアウトを変えるには、タブを強調表示されたドロップゾーンにドラッグします。図11.24では、2つのレポートがドッキングしています。図11.27は、「一変量の分布」レポートを強調表示されたドロップゾーンにドラッグし、レポートを縦に並べたところです。

図11.27 ダッシュボードでのタブのドッキング



ドッキングしたタブを作成するには、Tab Box() と Splitter Box() を組み合わせます。Tab Box と Splitter Box は、他のドッキングコンテナか、内容を定義するタブページを直接含んでいなければなりません。以下は前述の例の一部で、ボックスを並べているところです。

```
H Splitter Box( // レポートを Splitter Box に入れる
   Size( 700, 400 ),
   Tab Page Box( // レポートを Tab Page Box に入れる
```

以下の式は、Tab Box や Splitter Box をドッキングコンテナとし、その中にタブページをドッキングできるようにしています。

```
Tab Box << Set Dockable();
Splitter Box << Set Dockable();
```

レポートウィンドウでタブページの位置を変更するには、タブのタイトルをクリックし、タブページを強調表示されたドロップゾーンにドラッグします。強調表示されたドロップゾーンは、レポートを横、縦、タブの形式で並べる方法を示唆します。有効なドロップゾーンが選択されていない場合は、ボックスが元の位置に戻ります。

タブを閉じる

次の式は、タブページのレポートタイトルの右側に閉じるボタン を作成します。

```
Tab Page Box << Set Closeable();
```

閉じるボタン をクリックすると、タブとその内容が削除されます。不要になった親コンテナが削除されることもあります（たとえば、Tab Box のすべてのタブがなくなった場合や、Splitter Box に子が 1 つしかなくなった場合）。

Tab Page Box <<Set Close() メッセージを使用すると、閉じるボタン のクリック時に JSL スクリプトが実行されます。戻り値が 1 の場合、タブは閉じています。戻り値が 0 の場合は、その時点でタブを閉じることはできません。ユーザーの入力または選択の結果や、プログラムの状態により、タブを閉じることができない場合があります。

幅の広いタブの制御

ドッキングレイアウトにタブページボックスが含まれる場合、一つの Tab Box() コンテナにたくさんのページを入れることがあります。その場合、タイトルの幅が内容の幅より広くなったり、全体のバランスが取れないほどの幅になったりする可能性があります。Tab Box <<Set Overflow Enabled(0|1) メッセージは、タブのタイトルのせいでタブボックスが大きくなることを抑制します。タイトルが収まりきらないときは、タブボックスの右上角にポップアップボタンが表示され、「オーバーフローリスト」からタブが選択できるようになります。例については、図 11.26（488 ページ）を参照してください。

「クラスター分析」プラットフォームの起動ウィンドウを作成する例

次の例は、「クラスター分析」プラットフォームの簡易版ディスプレイボックスを描き、指定の引数を使ってプラットフォームを起動します。

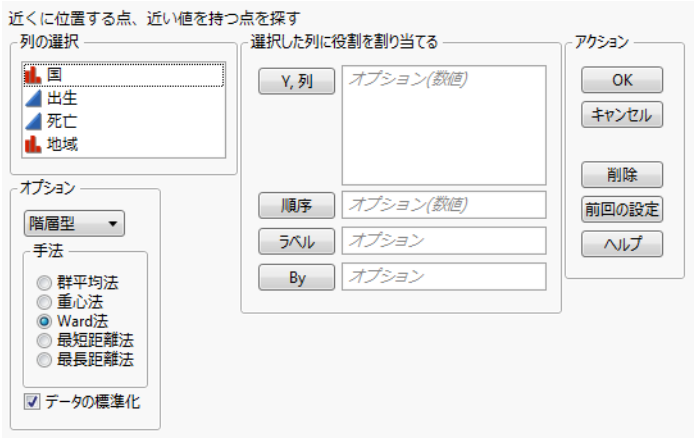
メモ: 機能の一部（前回の設定とヘルプ）はスクリプトから実行できないので、該当するボタンをクリックすると警告のウィンドウが表示されます。さらに、階層的クラスター法から K-Means クラスター法に切り換えても、ユーザインターフェースで表示されるときと違い、ウィンドウ自体は変化しません。

```
dt = Open( "$SAMPLE_DATA/Birth Death.jmp" );
nc = N Col( dt ); // ウィンドウ内の列数
lbWidth = 130; // ウィンドウの幅
methodList = {" 群平均法 ", " 重心法 ", "Ward 法 ", " 最短距離法 ", " 最長距離法 "};
```

```
// 手法のリストを定義
notImplemented = Expr(
   win = New Window( " この機能はまだ実行できません ", <<Modal, Button Box( "OK" ) )
);
clusterDlg = New Window( " クラスター分析」, // ウィンドウを作成
   <<Modal,
   Border Box( Left( 3 ), Top( 2 ),
      V List Box(
         TextBox(" 近くに位置する点、近い値を持つ点を探す " ),
         H List Box(
            V List Box(
               Panel Box( " 列の選択 ",
                  colListData = Col List Box(
                     All,
                     width( lbWidth ),
                     NLines( Min( nc, 10 ) )
                  )
               ),
               Panel Box( " オプション ",
                  V List Box(
                     comboObj = Combo Box(
                        {" 階層型 ", "K-Means 法 "},
                        <<Set( 1 )
                     ),
                     Panel Box( " 手法 ",
                        methodObj = Radio Box( methodList, <<Set( 3 ) )
                     ),
                     checkObj = Check Box( {" データの標準化 "}, <<Set( 1, 1 ) )
                  )
               )
            ),
            Panel Box( " 選択した列に役割を割り当てる ",
               Line Up Box( N Col( 2 ), Spacing( 3 ),
                  Button Box( "Y, 列 ",
                     colListY << Append( colListData << GetSelected )
                  ),
                  colListY = Col List Box(
                     width( lbWidth ),
                     NLines( 5 ),
                     " 数値 "
                  ),
                  Button Box( " 順序 ",
                     colListO << Append( colListData << GetSelected )
                  ),
                  colListO = Col List Box(
                     width( lbWidth ),
```

```
                        NLines( 1 ),
                        "数値"
                    ),
                    Button Box( "ラベル",
                        colListL << Append( colListData << GetSelected )
                    ),
                    colListL = Col List Box( width( lbWidth ), NLines( 1 ) ),
                    Button Box( "By",
                        colListB << Append( colListData << GetSelected )
                    ),
                    colListB = Col List Box( width( lbWidth ), NLines( 1 ) )
                )
            ),
            Panel Box( "アクション",
                Line Up Box( N Col( 1 ),
                    Button Box( "OK",
                        If( (comboObj << Get) == 1,
                            Hierarchical Cluster(
                                Y( Eval( colListY << GetItems ) ),
                                Order( Eval( colListO << GetItems ) ),
                                Label( Eval( colListL << GetItems ) ),
                                By( Eval( colListB << GetItems ) ),
                                Method( methodList[methodObj << Get]
                                ),
                                Standardize( checkObj << Get( 1 ) )
                            ),
                            KMeansCluster( Y( colListY << GetItems ) )
                        );
                        clusterDlg << Close Window;
                    ),
                    Button Box( "キャンセル", clusterDlg << Close Window ),
                    Text Box( " " ),
                    Button Box( "削除",
                        colListY << RemoveSelected;
                        colListO << RemoveSelected;
                        colListL << RemoveSelected;
                        colListB << RemoveSelected;
                    ),
                    Button Box( "前回の設定", notImplemented ),
                    Button Box( "ヘルプ", notImplemented )
                )
            )
        )
    )
);
```

図11.28「クラスター分析」起動ウィンドウ



カスタムプラットフォームを作成する例

「プログラミング手法」章の「式の操作」（226ページ）の例では、JSLのSubstitute Into()関数を使って、2次式の係数にある値を与えた後、その指定された2次式の解を求める方法を示しています。その例では、2次式の係数をSubstitute Into()の引数として指定しています。

「列ダイアログの作成」（498ページ）の節では、モーダルウィンドウを介してユーザに係数を入力させる例を示しています。

この節では、これらの例をさらに改善し、完全にカスタマイズされたインターフェースを作成します。最初にウィンドウを表示して係数を入力させます。そして、根を計算した後、独自に作成したグラフとともに結果を表示します。

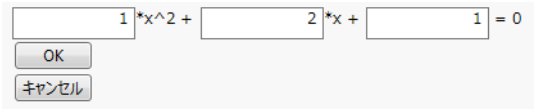
```
myCoeffs = New Window( "2 次式の根を見つける ",
// ウィンドウを表示して、ユーザに係数を入力させる
   <<Modal,
   H List Box(
      a = Number Edit Box( 1 ),
      Text Box( "*x^2 + " ),
      b = Number Edit Box( 2 ),
      Text Box( "*x + " ),
      c = Number Edit Box( 1 ),
      Text Box( " = 0" )
   ),
   Button Box( "OK",
      a = a << Get;
      b = b << Get;
      c = c << Get;
      Show( a, b, c );
   ),
```

```
      Button Box( "キャンセル")
   );

   x = {Expr(
      (-b + Sqrt( b ^ 2 - 4 * a * c )) / (2 * a)
   ), Expr(
      (-b - Sqrt( b ^ 2 - 4 * a * c )) / (2 * a)
   )};
   /* 結果を算出する： 2 次式は
       x=(-b + - sqrt(b^2 - 4ac))/2a。係数を 2 次式に
      入れる */
   xx = Eval Expr( x );
   // 解のリストを保存する

   results = Expr(
   /* 実根が見つかったかどうかテストし、適切な表示をする
      yes の場合（たとえば、ウィンドウのデフォルトで）は、根とグラフを表示する */
      xmin = xx[1] - 5;
      xmax = xx[2] + 5;
      ymin = -20;
      ymax = 20;
      win = New Window( "2 次関数の根",
         V List Box(
            Text Box( "方程式の実根" ),
            Text Box( "    " || Expr( po ) || " = 0" ),
            H List Box( Text Box( "are x=" ), Text Box( xxx ) ),
            Text Box( " " ), // 空白行を取得するため
            Graph Box(
               Frame Size( 200, 200 ),
               X Scale( xmin, xmax ),
               Y Scale( ymin, ymax ),
               Line Style( 2 ),
               H Line( 0 ),
               Line Style( 0 ),
               Y Function( polynomial, x ),
               Line Style( 3 ),
               Pen Color( 3 ),
               V Line( xx[1] ),
               V Line( xx[2] ),
               Marker Size( 2 ),
               Marker( 0, {xx[1], 0}, {xx[2], 0} )
            )
         )
      );
   );
   error = Expr(
```

```
/* no の場合（たとえば、a=3、b=4、c=5 で）は、エラー
ウィンドウと理解を手助けするグラフを表示する */
   win = New Window( "エラー",
      V List Box(
         Text Box( " " ),
         Text Box( "  多項式 " || po || " には実根がない。" ),
         Text Box( " " ),
         Text Box( "原因を知るために、関数のグラフを表示" ),
         Graph Box(
            Frame Size( 200, 200 ),
            X Scale( -20, 20 ),
            Y Scale( -20, 20 ),
            Line Style( 2 ),
            H Line( 0 ),
            Line Style( 0 ),
            Y Function( polynomial, x )
         )
      )
   )
);
polynomial = Expr( a * x ^ 2 + b * x + c );
// どちらの場合も、スクリプトは準備のための文字列を必要とする
// 指定された係数で多項式を書き直す
po = Char( Eval Expr( polynomial ) );
// 多項式のこのインスタンスを文字列で保存：

xxx = Char( Eval Expr( x ) );
// 解のリストを文字列で保存

If( Is Missing( xx[1] ) | Is Missing( xx[2] ), // テスト準備完了
   error,
   results
);
```

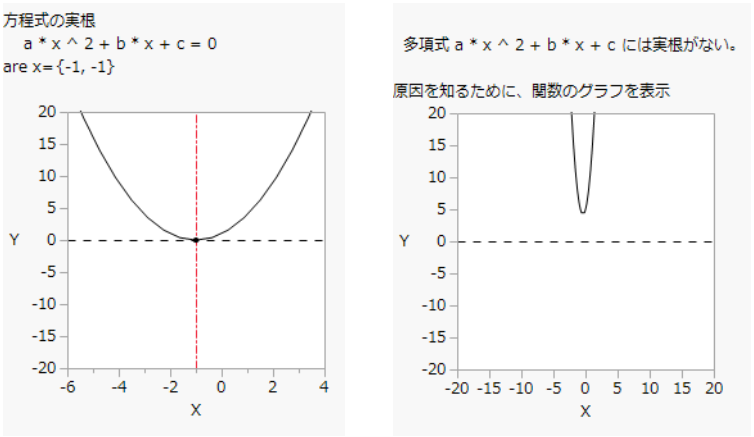
前述のスクリプトを実行すると、最初に、次のようなウィンドウが表示されます。

図11.29 独自に作成したウィンドウ



［**OK**］をクリックすると、根またはエラーメッセージのどちらかを示す、結果のウィンドウが表示されます（図11.30）。もう一度スクリプトを実行し、フィールドに5、4、5とそれぞれ入力して［**OK**］をクリックします。JMPは実根がないというメッセージを表示します（図11.30の右側）。

図11.30 カスタムプラットフォームのレポート



モーダルウィンドウ

Modal（モーダル）というのは、ユーザがまずそのウィンドウに反応しなければならない、ということを意味します。ウィンドウの外側をクリックするとエラー音が鳴り、ユーザがウィンドウに対して応答するまでスクリプトの実行が停止されます。

JMPには、2種類のモーダルウィンドウ関数が用意されています。

- `Modal`メッセージを指定した`New Window()`。新しいウィンドウを作成し、その中にディスプレイボックスを配置します。
- `Column Dialog()`は、起動ウィンドウの多くで使われているような、列を割り当てるためのウィンドウを作成します。`Column Dialog()`の中に作成できる要素は、`New Window()`ほど柔軟性が高くありません。しかし、`Column Dialog()`を使うと起動ウィンドウが簡単に作成できます。

モーダルウィンドウを作成する

モーダルウィンドウを表示するスクリプトが実行されると、JMP はウィンドウを作成し、ユーザがそれに応答して［**OK**］をクリックするのを待ちます。ウィンドウの外をクリックしてもエラー音が鳴るだけで、ユーザが［**OK**］または［**キャンセル**］をクリックするまではスクリプトの実行が停止されます。

`Column Dialog()`関数は、ユーザに現在（最前面）のデータテーブルから列を選択させることに特化した機能です。`Column Dialog()`で作成されるウィンドウも、モーダルウィンドウです。

以下は、スクリプトでモーダルウィンドウを使う際のアドバイスです。

1. 可能ならば、すべてのモーダルウィンドウをスクリプトの冒頭に置きます。そうすることで、ユーザとJMPの間のすべてのインタラクティブなやりとりが一度に行われ、その後はユーザがついていなくてもJMPが後の処理を進めます。

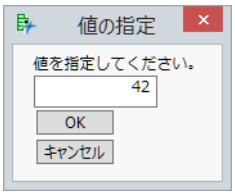
2. 作成したモーダルウィンドウが、ユーザに必要な情報を提示するようにしてください。数値を入力するフィールドを作成するだけではなく、入力された数値がどのように使われるかも、ユーザに知らせるようにします。入力内容に制限がある場合はそれを伝えます。

シンプルなモーダルウィンドウの例として、変数の値を1つ指定させるものを考えてみます。

```
win = New Window( " 値の指定 ",
   <<Modal,
   Text Box( " 値を指定してください。 " ),
   variablebox = Number Edit Box( 42 ),
   Button Box( "OK"),
   Button Box( " キャンセル ")
);
```

Number Edit Box() の引数42がその変数のデフォルト値となることに注意してください。

図11.31 モーダルディスプレイボックスの例



[OK] をクリックすると、ウィンドウが閉じ、{Button(1)} が戻されます。ボックスに入力された値を取得するには、Return Result メッセージを指定し、後でウィンドウに添え字を付けて記述します。将来廃止される Dialog() が、変数の割り当てのリストを戻していたのと同様です。

```
win = New Window( " 値の指定 ",
   <<Modal,
   <<Return Result,
   Text Box( " 値を指定してください。 " ),
   variablebox = Number Edit Box( 42 ),
   Button Box( "OK" ),
   Button Box( " キャンセル ")
);
Write( win["variablebox"] );
// 変数名 variablebox を添え字に指定する
```

この例で、ユーザがNumber Edit Boxに「33」と入力ししたとしましょう。この値は、win に格納されます。

[キャンセル] をクリックすると、ウィンドウは閉じ、{Button(-1)} が戻されてスクリプトの実行が継続されます。

ユーザが［キャンセル］をクリックしたかどうかを検出するには、次の式を追加します。

```
If( win["Button"] == 1,
   Print( win["variablebox"] );
   ,
   Print("キャンセルされました。")
);
```

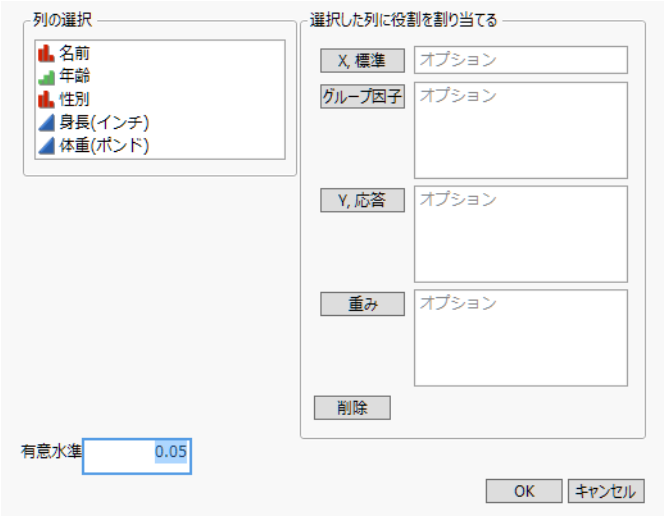
メモ: モーダルウィンドウには、ウィンドウを閉じるためのボタンが少なくとも1つ必要です。モーダルウィンドウのボタンのラベルには、[OK]、[Yes]（はい）、[No]（いいえ）、または［Cancel］（キャンセル）が使用できます。モーダルウィンドウにボタンがまったく含まれていない場合、JMPは［OK］ボタンを追加します。

列ダイアログの作成

Column Dialog()は、将来廃止されるDialog()関数の変形で、現在のデータテーブルから列を選ぶよう促します。[OK]、[キャンセル]、そして［削除］の各ボタンと、選択の対象となるデータテーブル列のリストは、どれも自動的に付加されます。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dlg = Column Dialog(
   Col ID = Col List( "X, 標準", Max Col( 1 ) ),
   Group = Col List( "グループ因子" ),
   Split = Col List( "Y, 応答" ),
   w = Col List( "重み" ),
   H List( "有意水準", alpha = Edit Number( .05 ) )
);
```

図11.32 Column Dialog



次の例では、ユーザの選択に応じて、次の例のようなリストが戻されます。

```
{Col ID = {}, Group = {}, Split = {}, w = {}, alpha = 0.05, Button( -1 )}
```

これらのリストを得るには、Col List節はColumn Dialog()の直接の引数である必要があります（他の引数の入れ子になっていてはいけません）。オプションでMaxCol(*n*)引数を指定すると、選択できる列の数を*n*に制限できます。結果のリストには、括弧で囲まれた「名前」のリストが含まれています。リストは**常に**戻されますが、空リストの場合もあります。ColList節は、最大12まで指定することができます。

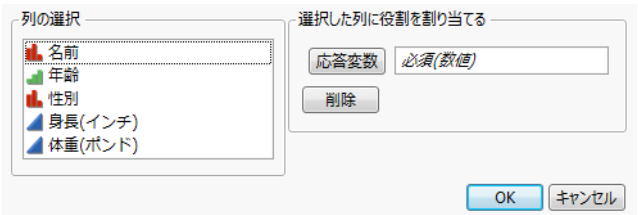
廃止されるDialog()コマンドで受け付けられていた他の項目はColumn Dialog()でも受け付けられ、同じ機能を果たします。MinColパラメータとMaxCol引数を使うと、列ダイアログボックスで選択できる列の最大数と最小数を指定できます。

選択できる列のデータタイプ（Ordinal、Nominal、またはContinuous）を指定することもできます。「列の選択」リストボックスの幅は、Select List Width(*pixels*)引数を使って設定できます。また、選択された列のリストボックスの幅を設定するには、Col List()関数内でWidth(*pixels*)を使います。

次の例は、1つの数値列だけを選択できる列ダイアログボックスを生成します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
rt_dlg = Column Dialog(
   cv = Col List( "応答変数", MaxCol( 1 ), MinCol( 1 ), DataType( "Numeric" ) )
);
```

図11.33 選択列の制限



DataTypeとして指定できるのは、Numeric、Character、およびRowStateです。

また、特定の列をあらかじめ割り当てておくには、Columns を使います。たとえば、次のスクリプトは、役割Xに「**身長(インチ)**」列を、役割Yに「**体重(ポンド)**」列と「**年齢**」列を割り当てています。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dlg = Column Dialog(
   xCols = Col List( "X, 因子", Columns( :Name("身長(インチ)") )),
   yCols = Col List( "Y, 応答", Columns( :Name("体重(ポンド)"), :年齢 ) )
);
```

Column Dialog と New Window の違い

New Window() と Column Dialog() は、どちらも列の選択ができる起動ウィンドウを作成できます。シンプルな起動ウィンドウを作成する場合は、Column Dialog() を使用してください。［削除］ボタン、［キャンセル］ボタン、列のリスト、［OK］ボタン（特にボタンが定義されていない場合）といった要素が自動的に追加されます。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dlg = Column Dialog(
   x = ColList( "X" ),
   y = ColList( "Y" )
);
```

New Window() は、オプションの引数が多数あり、より柔軟です。次の例では、ユーザは、2つの列を選択しないと［OK］ボタンをクリックできません。［OK］をクリックすると、二変量のグラフが作成されます。このような選択内容の検証は、Column Dialog() では実行できません。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
xvar = .;
yvar = .;
win = New Window( " 戻り値の確認の例 ",
   <<Modal,
   <<On Validate(
   // ユーザに対し、［OK］をクリックする前に 2 つの変数を選択するよう求める
      Show( xvar, yvar );
         If( Is Missing( xvar ) | Is Missing( yvar ),
         // xvar または yvar がない場合は、［OK］がクリックされても何もしない
         0,
         1
      );
   ),
   Text Box( "2 つの数値タイプの列を選択してください。" ),
   H List Box(
      Text Box( " X, 説明変数 " ),
      x = Col List Box(
         dt, // データテーブル参照
         all, // データテーブルの列をすべて表示する
         xvar = (x << Get Selected)[1];
         // ウィンドウが閉じる前に選択されている列の名前を取得する
         Show( xvar );
      ),
      Text Box( "Y, 応答 " ),
      y = Col List Box(
         dt,
         all,
         yvar = (y << Get Selected)[1];
         Show( yvar );
```

```
        )
    )
);
If (win["Button"] == 1, // ユーザが OK をクリックしたら ...
xcol = Column( dt, xvar ); //  列を取得する
ycol = Column( dt, yvar );
dt << Bivariate( Y( ycol ), X( xcol ) ); // 「二変量」プロットを作成する
);
```

Column Dialog の作成用関数

列の選択ができる起動ウィンドウを作成する Column Dialog では、Edit Number() や Check Box() を使用することができます。次の例は、Edit Number() ボックスを使って標準的な列ダイアログを作成します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
Column Dialog(
   ex x = ColList( "X", Max Col( 1 ) ),
   HList( " 有意水準 ",
   ex = Edit Number( .05 ) )
);
```

表 11.3 は、列ダイアログの作成用関数をまとめたものです。次の点を念頭に置いてください。

- 列ダイアログには、データテーブル内の列のリストが自動的に含まれます。
- ボタンを定義しない場合、自動的に［OK］ボタンが含まれます。
- 列ダイアログには、［OK］、［キャンセル］および［削除］ボタンも自動的に追加されます。

表 11.3 Column Dialog の作成用関数

作成用関数	構文	説明
Button	Button("OK")、Button("Cancel")	［OK］または［キャンセル］ボタンを描画します。［OK］がクリックされると、Button(1) を戻します。［キャンセル］がクリックされると、Button(-1) を戻します。

表11.3 Column Dialogの作成用関数（続き）

作成用関数	構文	説明
Check Box	*var*=Check Box("*Text after box*", <*1*|*0*>) *var*(CheckBox("*Text after box*", <*1*|*0*>))	[OK] がクリックされると、選択されているチェックボックスの変数には1が割り当てられます。選択されていないチェックボックスの変数には、0が割り当てられます。 オプションで、ウィンドウが開いたときに選択された状態（オン）にするのチェックボックスに1を、選択されていない（オフ）状態にするチェックボックスに0を追加できます。デフォルトの値は0（オフ）です。 Column Dialog()の場合は、var(Check Box(...))を使用します。
Col List	*var*=Col List("*role*", <MaxCol(*n*)>, <Datatype(*type*)>)	役割（*role*）ボタンを使って選択の対象を作成します。ユーザが選択したものは、*var*={*choice 1, choice 2, ..., choice n*}という形式のリスト項目で戻されます。 列の最小数または最大数を指定する場合は、MaxCol(*n*)またはMinCol(*n*)を使用します。 列のデータタイプを指定する場合は、Datatype(*type*)を使用します。*type*に指定できるのは、Numeric、Character、およびRowstateです。
Combo Box	*var*=Combo Box("*choice1*", "*choice2*", ...) *var*(ComboBox("*choice1*", "*choice2*", ...))	リストされた選択肢を使ってメニューを作成します。最初の選択肢がデフォルトです。選択肢は引用符で囲んだ文字列で指定する必要があります。または、リストで指定します。 Column Dialog()の場合は、var(Combo Box(...)を使用します。
Edit Number	*var*=Edit Number(*number*) *var*(Edit Number(*number*))	*number*をデフォルト値とした、数の編集フィールドを作成します。[OK] がクリックされると、フィールドに入力された数を変数に割り当てます。 Column Dialog()の場合は、var(Edit Number (...)を使用します。

表11.3 Column Dialogの作成用関数（続き）

作成用関数	構文	説明
Edit Text	*var*=Edit Text("*string*", <width(*x*)>) *var*(Edit Text("*string*", <width(*x*)>))	*string*をデフォルト値とした、文字列の編集フィールドを作成します。また、ボックスの最小幅をピクセルで指定できます。デフォルトの幅は72ピクセルです。［OK］がクリックされると、フィールドに入力されたテキストを変数に割り当てます。 Column Dialog()の場合は、 var(Edit Text (...)を使用します。
HList	HList(*item*, *item*, ...)	項目（*item*）を上端に沿って一定間隔で横一列に並べます。2つのVListをHListの引数にすると、上端に沿って一定間隔に並んだ2つの列ができます。
Line Up Box	Line Up Box(*n*, *item_11*, *item_12*, ..., *item_1n*, ..., *item_nn*)	項目（*item*）を*n*列に並べます。*item_ij*は*i*行目の*j*番目の項目です。
List Box	var=List Box({"item", "item", ...}, width(50), max selected(2), NLines(6))	項目リストを含むディスプレイボックスを作成します。
Radio Buttons	*var*=Radio Buttons("*choice1*", "*choice2*", ...) *var*(RadioButtons("*choice1*", "*choice2*", ...))	指定した選択肢で、縦に並んだ左揃えのラジオボタンのリストを作成します。最初の選択肢がデフォルトです。［OK］がクリックされると、選択されているボタンが変数に割り当てられます。選択肢は引用符で囲んだ文字列で指定する必要があります。 Column Dialog()の場合は、 var(Radio Buttons(...)を使用します。
文字列	"*string*"	ウィンドウ内にテキストを表示します。たとえば、Edit Numberフィールドの前に文字列のラベルをつけることができます。文字列は引用符で囲む必要があります。
VList	VList(*item*, *item*, ...)	項目（*item*）を左端に沿って一定間隔で縦一列に並べます。2つのHListをVListの引数にすると、左端に沿って一定間隔に並んだ2つの行ができます。

スクリプトエディタのコマンド

スクリプトエディタウィンドウも表示ツリーの1つなので、JSLスクリプトを記述して、スクリプトエディタウィンドウの内容を変更したり保存したりできます。

New Script()という関数はありません。代わりに、新しいスクリプトウィンドウを開く場合は、New Window()関数を使い、それがスクリプトウィンドウであることを知らせるメッセージを送ります。

```
win = New Window( "title", <<Script, "Initial Contents" );
```

最後の引数はオプションです。文字列を指定した場合は、新しいスクリプトウィンドウにその文字列が含まれます。

前述のNew Window()の例では、winはウィンドウ全体であるディスプレイボックスへの参照です。スクリプトウィンドウに書き込みを行うには、書き込み先であるディスプレイボックス部分への参照を取得する必要があります。これはScript Box()と呼ばれます。

```
ed = win[Script Box( 1 )];
```

このように取得した参照（この例では、ed）を使えば、テキストの追加、削除、取得ができます。

```
ed << Get Text();
  "Initial Contents"
```

スクリプトウィンドウ内のすべてのテキストを設定するには、Set Textメッセージを使用します。次のメッセージは、スクリプトウィンドウ内のすべてのテキストをクリアした後、aaa=3;を追加し、改行を入力します。

```
ed << Set Text( "aaa=3;\!N" );
```

スクリプトウィンドウの最後にテキストを追加するには、Appendメッセージを使用します。

```
ed << Append Text( "bbb=1/10;" );
ed << Append Text(" \!Nccc=4/100;" );
```

指定の行番号のテキストを取得するには、Get Line Textメッセージを使用します。指定の行番号のテキストを新しいテキストに置き換えるには、Set Line Textメッセージを使用します。

```
ed << Get Line Text( 2 );
ed << Set Line Text( 2, "bbb = 0.1;" );
```

スクリプト内の総行数を取得するには、Get Line Countメッセージを使用します。Get Linesメッセージは、各行のテキストを要素としたリストを戻します。

```
ed << Get Line Count();
ed << Get Lines();
```

スクリプトの体裁を読みやすく整えるには、Refromatメッセージを使用します。

```
ed << Reformat();
```

Save Text Fileメッセージを使うと、スクリプトが拡張子.jslをつけたテキストファイルに保存されます。

```
Save Text File(
   "$DOCUMENTS/Example.txt",
   "The quick brown fox"
);
```

スクリプトウィンドウ内のスクリプト全体を実行するには、Run メッセージを用います。

```
ed << Run();
```

スクリプトウィンドウを閉じるには、他のJMPウィンドウの場合と同じく、ウィンドウにClose Windowメッセージを送ります。

```
win << Close Window( nosave );
```

廃止されるDialog を New Windowに変換する

Dialog() は今後廃止されますが、代わりにNew Window() とModal メッセージを使用できます。Dialog() を使用したスクリプトは、今後、実行できなくなります。

以下の節で、Dialog() を使ったスクリプトをNew Window() で書き換える方法を解説します。

New Windowと Dialogの比較

次の2つの例は、同じウィンドウを作成しています。1つ目はModal 引数を指定したNew Window()、2つ目はDialog() 関数を使用しています。New Window() には、より多くの表示オプションがよいされており、また、ウィンドウの内容や機能をより細かくコントロールできます。

New Windowの例

次の例は、縦と横のリストボックスが複数あるウィンドウを作成します。

```
win = New Window( "New Windowの例",
   <<Modal,
   <<ReturnResult, // Dialogと同じように結果を取得する
   V List Box(
      V List Box(
         Text Box( "分析のパラメータ" ),
         Line Up Box(
            NCol( 2 ),
            Text Box( "下側仕様限界" ),
            lsl_box = Number Edit Box( 230 ),
            Text Box( "上側仕様限界" ),
            usl_box = Number Edit Box( 340 ),
            Text Box( "閾値" ),
            threshold_box = Number Edit Box( 275 )
         ),
```

```
            H List Box(
               Panel Box( "ラジオの種類",
                     rb_box1 = Radio Box( {"RCA", "Matsushita",
                     "Zenith", "Sony"} )),
               Panel Box( "アンテナの種類" ,
                     rb_box2 = Radio Box( {"パラボラ", "ヘリカル", "極性",
                     "Radiant Array"} ) )
            ),
            cb_box1 = Check Box( "放射同期" ),
            Text Box( "グラフのタイトル" ),
            title_box = Text Edit Box( "分析結果" ),
            H List Box( Text Box( "品質" ),
            cb_box2 = Combo Box( {"最優良", "優良", "良", "可"} ) )
         ),
         H List Box(
            Align( Right ),
            Spacer Box(),
            Button Box( "OK",
               lsl = lsl_box << Get;
               usl = usl_box << Get;
               threshold = threshold_box << Get;
               radio_type = rb_box1 << Get;
               antenna = rb_box2 << Get;
               synch = cb_box1 << Get;
               title = title_box << Get Text;
               quality = cb_box2 << Get;
            ),
            Button Box( "キャンセル")
         )
      )
   );
   If(win["Button"] == 1,
   // ユーザが［OK］をクリックしたときは、選択内容をログに表示する
      Show( "OK", lsl, usl, threshold, radio_type, antenna, synch, title, quality);
      ,
      Show( "Canceled" ); // ユーザがキャンセルをクリックしたときは、「Canceled」と出力する
   );
```

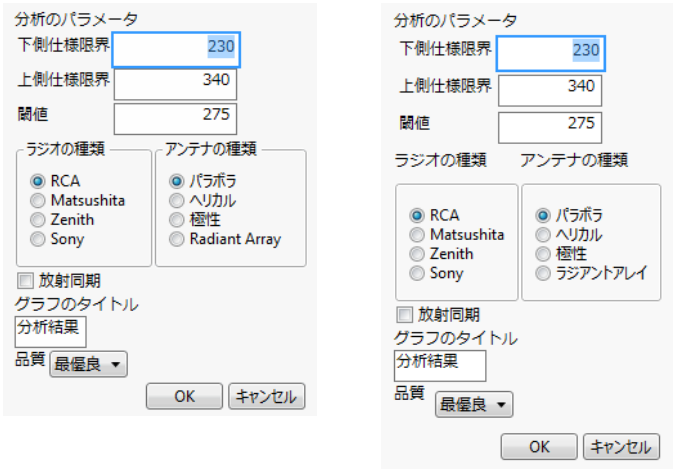
廃止される Dialog の例

先ほどの New Window() の例とこの Dialog() の例を比較してみましょう。

```
dlg = Dialog(
   Title( "Dialog の例" ),
   H List(
      V List(
         "分析のパラメータ",
```

```
            Lineup( 2,
               "下側仕様限界", lsl = Edit Number( 230 ),
               "上側仕様限界", usl = Edit Number( 340 ),
               "閾値", threshold = Edit Number( 275 )
            ),
            H List(
               V List(
                  "ラジオの種類",
                  type = Radio Buttons( "RCA", "Matsushita", "Zenith", "Sony" )
               ),
               V List(
                  "アンテナの種類",
                  antenna = Radio Buttons( "パラボラ", "ヘリカル", "極性",
                     "ラジアントアレイ" )
               )
            ),
            synch = Check Box( "放射同期", 0 ),
            "グラフのタイトル",
            title = Edit Text( "分析結果" ),
            H List(
               "品質",
               quality = Combo Box( "最優良", "優良", "良", "可"
               )
            )
         ),
         V List( Button( "OK" ), Button( "キャンセル" ) )
      )
   );
   If( dlg["Button"] == 1,
      Show(
         "OK",
         dlg["lsl"],
         dlg["usl"],
         dlg["threshold"],
         dlg["type"],
         dlg["antenna"],
         dlg["synch"],
         dlg["title"],
         dlg["quality"]
      ),
      Show( "Canceled" )
   );
```

図 11.34 New Window の結果（左）と Dialog の結果（右）



メモ: Dialog() と同様の結果を得るには、Return Result を使用してください。例については、「モーダルウィンドウを作成する」（496 ページ）を参照してください。

New Window と Dialog の違い

この節では、New Window() のディスプレイボックスと今後廃止される Dialog() スクリプトの違いを紹介します。表 11.4 は違いをまとめたものです。表の後の節で、詳細を説明しています。

表 11.4 New Window と Dialog のディスプレイボックス

ボックスの種類	New Window	Dialog
「文字列とリスト」	`win = New Window( "Combo Box",` `  <<Modal,` `  <<ReturnResult,` `  cb = Combo Box( {"真", "偽"}` `),` `);`	`Dialog( Title( "Combo Box" ),` `  cb = Combo Box( "真", "偽" ),` `);`
「List Box と Check Box」	`win = New Window( "H List Box",` `  <<Modal,` `  <<ReturnResult,` `  H List Box(` `  kb1 = Check Box( "a" ),` `  kb2 = Check Box( "b" ),` `  kb3 = Check Box( "c" )` `  ),` `);`	`dlg = Dialog( Title( "H List" ),` `  H List(` `  kb1 = Check Box( "a", 0 ),` `  kb2 = Check Box( "b", 0 ),` `  kb3 = Check Box( "c", 0 )` `  ),` `);`

表11.4 New WindowとDialogのディスプレイボックス（続き）

ボックスの種類	New Window	Dialog
「項目の整列」	win = New Window("Line Up Box", <<Modal, <<ReturnResult, V List Box(Line Up Box(NCol(2), Text Box(" この値を設定 "), var1=Number Edit Box(42), Text Box(" 別の値を設定 "), var2=Number Edit Box(86),),), H List Box(Button Box("OK"), Button Box(" キャンセル "))); // {Button(1)} を戻す	dlg = Dialog(V List(Line Up(2), " この値を設定 ", variable=Edit Number(42), " 別の値を設定 ", var2=Edit Number(86)), H List(Button("OK"), Button(" キャンセル ")))); // {variable = 42, var2 = 86, Button(1)} を戻す
「Radio Box」	win = New Window("Radio Box", <<Modal, <<ReturnResult, Panel Box(" 選択 ", rbox = Radio Box({"a", "b", "c"})));	dlg = Dialog(Title("Radio Buttons"), rb = Radio Buttons({"a", "b", "c"}),);
「Text Edit Box」	win = New Window("Text Edit Box", <<Modal, <<ReturnResult, V List Box(Text Box(" 文字列 "), str1 = Text Edit Box("The"), str2 = Text Edit Box("quick"),)); 例の全容は「Text Edit Box」(513ページ)を参照してください。	dlg = Dialog(Title("Edit Text"), V List(" 文字列 ", str1 = Edit Text("The"), str2 = Edit Text("quick"),)); 例の全容は「Text Edit Box」(513ページ)を参照してください。

表 11.4 New Window と Dialog のディスプレイボックス（続き）

ボックスの種類	New Window	Dialog
「Number Edit Box」	win = New Window("Number Edit Box", <<Modal, <<ReturnResult, V List Box(Text Box("乱数"), num1 = Number Edit Box(Random Uniform()), num2 = Number Edit Box(Random Uniform() * 10), ...); 例の全容は「Number Edit Box」（514 ページ）を参照してください。	dlg = Dialog(Title("Edit Number"), V List("乱数", num1 = Edit Number(Random Uniform()), num2 = Edit Number(Random Uniform() * 10), ...);

文字列とリスト

New Window() 関数と Dialog() 関数の主な違いは、リストと文字列を引数として指定する方法にあります。New Window() の場合、項目をリストに入れる必要があります。Dialog() では、項目は一般にカンマで区切られます。たとえば、Combo Box の次のような例を比べてみましょう。

```
win = New Window( "Combo Box",// モーダルの New Window
   <<Modal,
   <<Return Result,
   cb = Combo Box( {"真", "偽"} ), // 項目を中括弧で囲んだリストで指定する
);

Dialog( Title( "Combo Box" ),// モーダルの Dialog
   cb = Combo Box( "真", "偽" ), // 項目をカンマで区切って指定する
  );
```

メモ: スクリプトは明示的に［OK］ボタンを定義していません。モーダルウィンドウの場合、［OK］ボタンは自動的に追加されますが、その他のボタンは明示的に定義してください。

さらに、空のテキストを含めるとき、Dialog() では " " という形で指定できますが、New Window() の場合は Text Box ("") と指定する必要があります。New Window() の場合は Text Box (" ") と指定する必要があります。

List Box と Check Box

New Window() では、H List Box() を使ってボックスを横に並べ、V List Box() を使ってボックスを縦に並べます。Dialog() では、代わりに H List と V List を使用します。

メモ: H List Box() または V List Box() を省略した場合、ボックスはデフォルトで縦に並べられます。

次の例は、横に並んだ3つのチェックボックスと、1つの［OK］ボタンのあるウィンドウを作成します。

```
win = New Window( "H List Box",
   <<Modal,
   <<Return Result,
   H List Box(
      kb1 = Check Box( "a" ),
      kb2 = Check Box( "b" ),
      kb3 = Check Box( "c" )
   ),
);
```

同じウィンドウを、Dialog() とH List() を使って作成します。

```
dlg = Dialog( Title( "H List" ),
   H List(
         kb1 = Check Box( "a", 0 ),
         kb2 = Check Box( "b", 0 ),
         kb3 = Check Box( "c", 0 )
    ),
);
```

項目の整列

New Window() で、指定の列数に項目を整列させるには、Line Up Box() を使用します。Dialog() では、Line Upを使用していました。

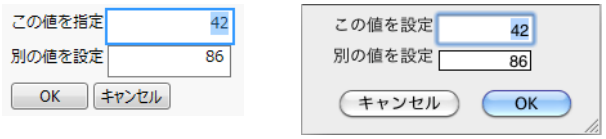
次の例は、Text Boxを1列に配置し、Number Edit Boxをもう1列に配置します。

```
win = New Window( "Line Up Box",
   <<Modal,
   <<Return Result,
   V List Box(
      Lineup Box( N Col( 2 ),
         Text Box( "この値を指定" ),
         var1 = Number Edit Box( 42 ),
         Text Box( "別の値を設定" ),
         var2 = Number Edit Box( 86 ),

      ),
   ),
   H List Box( Button Box( "OK" ), Button Box( "キャンセル" ) )
);
```

［**OK**］をクリックすると、次の結果がログウィンドウに表示されます。

リスト(3個の要素)が割り当てられました。

図 11.35 Windows（左）と Macintosh（右）の Dialog によるデフォルトの配置



次は、同じウィンドウを Dialog() と Line Up を使って作成します。

```
dlg = Dialog(
   V List(
      Line Up( 2,
         " この値を設定 ", variable = Edit Number( 42 ),
         " 別の値を設定 ", var2 = Edit Number( 86 )
      ),
      H List( Button( "OK" ), Button( " キャンセル " ) )
   )
);
```

［OK］をクリックすると、次の結果がログウィンドウに表示されます。

```
{variable = 42, var2 = 86, Button(1)}
```

メモ：［OK］と［キャンセル］のボタンの位置は、OS の種類に応じ、ダイアログボックスのスタイルに合わせて調整されます。場合によっては、Button("OK") と Button("キャンセル") の配置のために、H List Box()、V List Box()、Line Up Box() の設定が無効になることがあります。このためダイアログボックスが想定していたものと少し異なる場合もあります。

Radio Box

New Window() と Dialog() のもう一つの違いは、Radio Box の使用にあります。

New Window() でパネルを示す枠の中にラジオボタンを配置したい場合、Panel Box() コンテナを定義しなければなりません。

```
win = New Window( "Radio Box",
   <<Modal,
   <<Return Result,
   Panel Box( " 選択 ",
      rbox = Radio Box( {"a", "b", "c"} )
   )
);
```

Dialog() では、Radio Buttons() は自動的にパネルボックス内に表示されます。

```
dlg = Dialog( Title( "Radio Button" ),
   rb = Radio Buttons( {"a", "b", "c"} ),
);
```

Text Edit Box

New Window() で、指定の文字列を含む編集可能なボックスを作成するには、Text Edit Box() を使用します。Dialog() では、Edit Text() を使用していました。

```
win = New Window( "Text Edit Box",
   <<Modal,
   <<Return Result,
   V List Box(
      Text Box( "文字列" ),
      str1 = Text Edit Box( "The" ),
      str2 = Text Edit Box( "quick" ),
      str3 = Text Edit Box( "brown" ),
      str4 = Text Edit Box( "fox" ),
      str5 = Text Edit Box( "jumps" ),
      str6 = Text Edit Box( "over" ),
      str7 = Text Edit Box( "the" ),
      str8 = Text Edit Box( "lazy" ),
      str9 = Text Edit Box( "dog" ) ) );
```

次は、同じウィンドウをDialog() とEdit Text() を使って作成しているものです。

```
dlg = Dialog(
   Title( "Edit Text" ),
   V List(
      "文字列",
      str1 = Edit Text( "The" ),
      str2 = Edit Text( "quick" ),
      str3 = Edit Text( "brown" ),
      str4 = Edit Text( "fox" ),
      str5 = Edit Text( "jumps" ),
      str6 = Edit Text( "over" ),
      str7 = Edit Text( "the" ),
      str8 = Edit Text( "lazy" ),
      str9 = Edit Text( "dog" )
   )
);
```

指定の文字列を含む編集可能なボックスを作成するもう1つの方法は、String Col Edit Box() を使用する方法です。たとえば、次のスクリプトは「String Col Box」という名前のウィンドウを作成します。このウィンドウには「文字列」という名前の編集可能なボックスの列があり、各ボックスに指定の文字列が挿入されます。

```
win = New Window( "String Col Edit Box",
   <<Modal,
   <<Return Result,
   steb = String Col Edit Box(
      "文字列",
```

```
        {"The", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog"}
    )
);
```

メモ: 前述の例で String Col Edit Box() がリストに割り当てる要素は2つです。New Window() と廃止された Dialog() の例では、Text Edit Box() が10の要素をリストに割り当てています。

Number Edit Box

New Window() で、指定の数値を含む編集可能なボックスを作成するには、Number Edit Box() を使用します。Dialog() では Edit Number() を使用していました。

```
win = New Window( "Number Edit Box",
    <<Modal,
    <<ReturnResult,
    V List Box(
        Text Box( "乱数" ),
        num1 = Number Edit Box( Random Uniform() ),
        num2 = Number Edit Box( Random Uniform() * 10 ),
        num3 = Number Edit Box( Random Uniform() * 100 ),
        num4 = Number Edit Box( Random Uniform() * 1000 )
    ),
);
```

次は、同じウィンドウを Dialog() と Edit Number() を使って作成しているものです。

```
dlg = Dialog(
    Title( "Edit Number" ),
    V List(
        "乱数",
        num1 = Edit Number( Random Uniform() ),
        num2 = Edit Number( Random Uniform() * 10 ),
        num3 = Edit Number( Random Uniform() * 100 ),
        num4 = Edit Number( Random Uniform() * 1000 )
    ),
);
```

同じウィンドウを作成するもう1つの方法は、Number Col Edit Box() を使って指定の数値を含む編集可能なボックスを作成する方法です。たとえば、次のスクリプトは「Number Col Edit Box」という名前のウィンドウを作成します。このウィンドウには「乱数」という名前の編集可能なボックスの列があり、各ボックスに指定した種類の乱数から得られた値が表示されます。

```
win = New Window( "Number Col Edit Box",
    <<Modal,
    <<ReturnResult,
    nceb = Number Col Edit Box(
        "乱数",
        {num1 = Random Uniform(), num2 = Random Uniform() * 10, num3 =
```

```
        Random Uniform() * 100, num4 = Random Uniform() * 1000}
    ),
);
```

メモ: 前述の例で Number Col Edit Box() がリストに割り当てる要素は2つです。New Window() と廃止された Dialog() の例では、Text Edit Box() が5つの要素をリストに割り当てています。

New Window で使用できるオプションのスクリプト

New Window() を使用するメリットの1つは、ディスプレイボックスにオプションのスクリプトを追加できることです。Dialog() では、次のようなコンボボックスにオプションのスクリプトを含めることはできませんでした。ディスプレイボックスのコントロールのいずれかに関連したアクションが必要なとき、コントロールの最後の引数としてスクリプトを追加する必要がありました。以下は、New Window の例です。

```
win = New Window( "Combo Box",
<<Modal,
<<ReturnResult,
   comboObj = Combo Box(
      {"真", "偽"},
      << Set( 1 ),
      Print( comboObj << Get )
   )
);
```

ユーザが異なる値を選択したとき、選択された項目の番号（この例では、コンボボックス内の項目数が2なので1または2）がログに出力されます。

技術的な詳細

Tab Box と Tab Page Box のスクリプトの記述

Tab Page Box() は、ページのタイトル（title）と内容（contents）を1つのディスプレイボックスにまとめます。Tab Page Box() を Tab Box() の中に入れて使用した場合、複数のタブページを含んだタブ付きウィンドウが表示されます。「Tab Box と Tab Page Box」（458ページ）に詳細があります。

以前のバージョンの JMP では、V List Box() などのディスプレイボックスがタブの内容を含んでいました。次のスクリプトを見てみましょう。

```
win = New Window( "Tab Box",
   tb = Tab Box(
      "1ページ", // タブの名前
      V List Box( // タブの内容
         Text Box( "1ページの1行目" ),
         Text Box( "1ページの2行目" )
      ),
```

```
        "2 ページ ", // タブの名前
        V List Box( // タブの内容
           Text Box( "2 ページの 1 行目 " ),
           Text Box( "2 ページの 2 行目 " )
        ),
     )
  );
```

JMP 13 からは、Tab Box() にそれぞれのタブの Tab Page Box() を含めるようにしてください。前述の例は、次のように書き換えられます。

```
win = New Window( "Tab Box",
   tb = Tab Box(
      Tab Page Box( // タブの内容
         " タブボックスの 1 ページ ", // タブの名前
         Text Box( "1 ページの 1 行目 " ),
         Text Box( "1 ページの 2 行目 " )
      ),
      Tab Page Box( // タブの内容
         "2 ページ ", // タブの名前
         Text Box( "2 ページの 1 行目 " ),
         Text Box( "2 ページの 2 行目 " )
      )
   )
);
```

Tab Box の外で作成された Tab Page Box や、外にドラッグされた Tab Page Box は、単独のコンテナとなります。Sheet Part() のタイトルと同様に、影の付いたタイトルが一番上に表示されます。タブページの形式にはなりません。「1 段のレポートで構成されたダッシュボード」（485 ページ）に例があります。

Tab Box と Tab Page Box メッセージ

これまで Tab Box() に使われていたメッセージの一部は将来廃止されますが、それに該当するメッセージは Tab Page Box() で使用できます。

表 11.5 廃止される Tab Box() メッセージ

廃止される Tab Box() メッセージ	現在の Tab Page Box メッセージ
Get Title／Set Title	Get Title／Title
Get Tip／Set Tip	Get Tip／Tip
Get Icon／Set Icon	Get Icon／Icon
Get Closeable／Set Closeable	Get Closeable／Closeable

この 2 つのディスプレイボックスの詳細については、「Tab Box と Tab Page Box」（458 ページ）を参照してください。

第12章

スクリプトによるグラフ作成
2次元グラフの編集と作成

グラフ用のJSLスクリプトには、図形や線を作成する式、塗りつぶしのパターンを適用する式、背景の色を割り当てる式などが含まれます。これらの式は、グラフを右クリックし、［カスタマイズ］を選択して追加することもできます。

この章では、グラフをインタラクティブに編集する方法、グラフや背景地図を作成する方法を説明します。3次元グラフのスクリプトについては、「3Dシーン」（571ページ）章を参照してください。

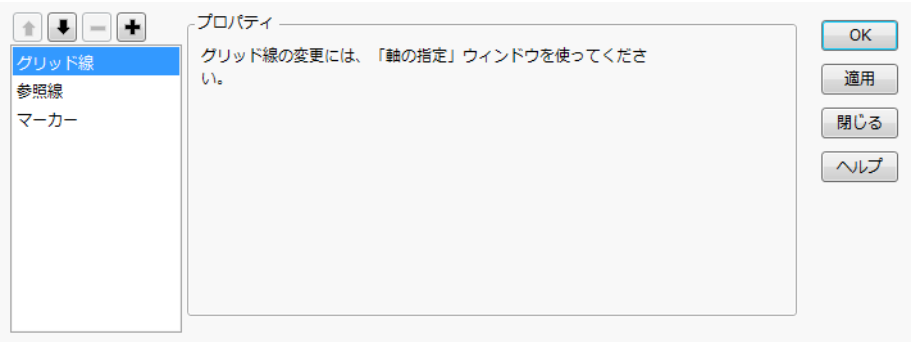
グラフへのスクリプトの追加

グラフフレームを右クリックすると、JSL コマンドを入力したり、貼り付けたりできます。それらのスクリプトには、通常、グラフフレームのコンテキスト内で実行される描画コマンドが含まれています。グラフフレームのコンテキストには、軸のデータ範囲や、データとスクリプトが描画される順序などがあります。

次の例は、グラフにスクリプトを追加する方法を示します。

1. ［**ヘルプ**］>［**サンプルデータライブラリ**］を選択し、「Big Class.jmp」を開きます。
2. ［**分析**］>［**二変量の関係**］を選択します。
3. 「**体重 (ポンド)**」を［Y, 目的変数］、「**身長 (インチ)**」を［X, 説明変数］に指定し、［**OK**］をクリックします。
4. グラフ内を右クリックし、［**カスタマイズ**］を選択します。

図12.1 グラフオプションのカスタマイズ

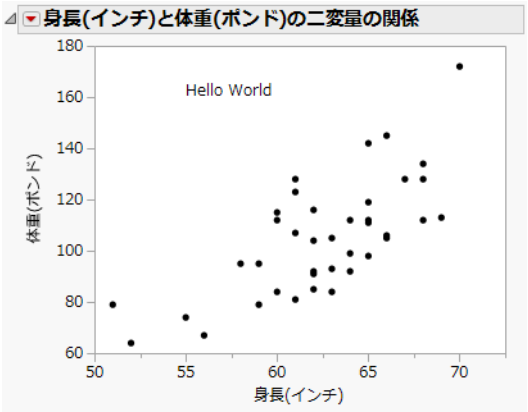


5. ［**追加**］ボタン（＋）をクリックして新しいグラフスクリプトを追加します。
6. 次のテキストを入力し、［**OK**］をクリックします。

```
Text( {55, 160}, "Hello World" );
```

これで、グラフの x 座標55、y 座標160の位置にテキストが表示されました。

図12.2 スクリプトをグラフに追加



デフォルトでは、追加したグラフィックスクリプトによる描画は、散布図のデータ点に上書きされて描かれます。

たとえば、「グラフをカスタマイズ」ウィンドウで次のようなスクリプトを追加します。

```
Fill Color( "Green" ); Rect( 57, 175, 65, 110, 1 );
```

すると、緑で塗りつぶされた四角形が表示されます。スクリプトはリストの順番どおりに描かれるので、リストの最初の項目が最初に描かれます。四角形を一番後ろに配置したい場合は、四角形のスクリプトを選択して［上へ］ボタン（⬆）をクリックし、テキストスクリプトの上へ移動させます。四角形を一番手前に配置したい場合は、四角形のスクリプトを選択して［下へ］ボタン（⬇）をクリックします。グラフ上のスクリプトは、すべて並べ替えて任意の順番に描くことができます。新しいスクリプトは、リスト内の選択されている項目のすぐ下に追加されます。

ヒント： スクリプトで列名を参照するときは、参照範囲（スコープ）を明らかにするため、「`Column(列名)`」、またはコロンを列名の前に置いて「`:列名`」と指定してください。

ヒント： この例の手順によって生成されたJSLプログラムは、赤の三角ボタンのメニューから［**スクリプトの保存**］>［**スクリプトウィンドウへ**］を選択すれば、確認できます。

グラフィック要素の順序を指定する

インタラクティブな方法のほかにも、JSLを使ってグラフィック要素を追加することができます。

```
Frame Box <<Add Graphics Script(<("Back"|"Front") | <n>>, <"description">, script)
```

JSLで追加したグラフィック要素は、グラフの最前面にある要素の上に描かれます。

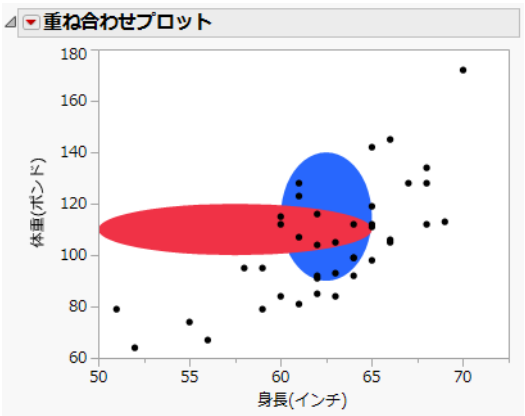
オプションの順序（order）引数を使えば、グラフィック要素の描画順を指定できます。orderにはキーワードのBackまたはFront、または複数のグラフィック要素の描画順を整数で指定します。たとえば、散布図に楕円を追加すると、楕円はマーカーの上に描かれます。キーワードBack（または2）を使用した場合、楕円は一番後ろに描かれます。Front（または1）は、そのオブジェクトが最初に描画されることを意味します。

オプションのDescription引数は、「グラフをカスタマイズ」ウィンドウでグラフィックスクリプトの隣に表示される項目名です。

複数のグラフィック要素の描画順を指定するには、order引数に整数を指定して、それらの描画順を相対的に指定します。次のスクリプトは、まず青い楕円を追加し、次にその手前に赤い楕円を追加します。どちらの楕円も、点の後ろに表示されます。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
op = dt << Bivariate(
   X( :height ),
   Y( :weight ),
);

Report( op )[Frame Box( 1 )] << Add Graphics Script(
   1, // 青い楕円を描画
   Description( "青い楕円" ),
   Fill Color( "Blue" );
   Oval( 60, 140, 65, 90, 1 );
);

Report( op )[Frame Box( 1 )] << Add Graphics Script(
   2, // 青い楕円に重ねて赤い楕円を描画
   Description( "赤い楕円" ),
   Fill Color( "Red" );
   Oval( 50, 120, 65, 100, 1 );
);
```

図12.3 描画順の指定



フレームの内容または設定のコピーと貼り付け

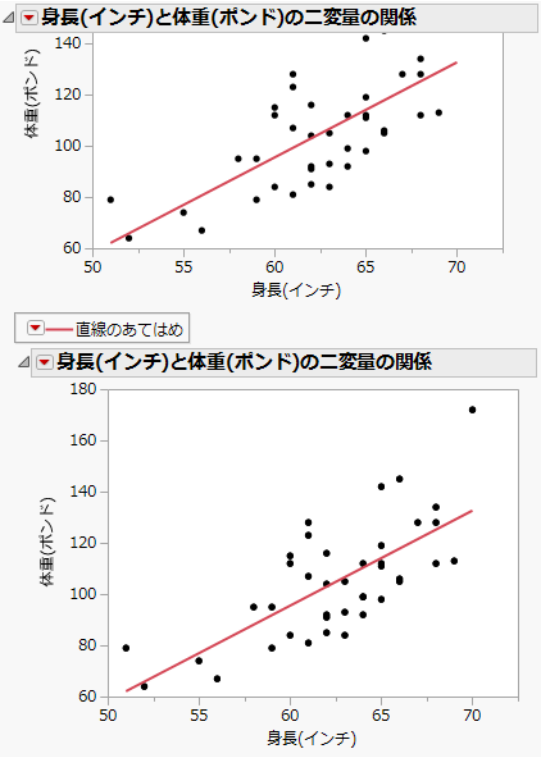
次のJSLコマンドを使って、フレームの内容や設定をコピーし、貼り付けることができます。

```
obj << Copy Frame Contents // グラフやあてはめ線などの内容
obj << Paste Frame Contents
obj << Copy Frame Settings // 背景色などの設定
obj << Paste Frame Settings
```

次の例は、2つの「二変量」グラフを作成し、最初のグラフにあてはめ線を追加し、その線をコピーして2番目のグラフに貼り付けます。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
biv = dt << Bivariate( Y( :Name("体重(ポンド)") ), X( :Name("身長(インチ)") ) );
rbiv1 = biv << Report; // レポートオブジェクトを作成
rbiv2 = rbiv1 << Clone Box; // 二変量グラフのコピーを作成

rbiv1 << Append( rbiv2 );
// 二変量グラフの下にコピーしたボックスを配置して
// 2つ目の二変量グラフを作成

framebox1 = rbiv1[Frame Box( 1 )]; // レポートのフレームボックスに参照を割り当てる
framebox2 = rbiv2[Frame Box( 1 )];
biv << Fit Line; // 最初の二変量グラフに直線をあてはめる

framebox1 << Copy Frame Contents;
// framebox1の内容をコピーする

framebox2 << Paste Frame Contents;
// フレームの内容をframebox2に貼り付ける
```

図12.4 フレームの内容のコピーと貼り付け

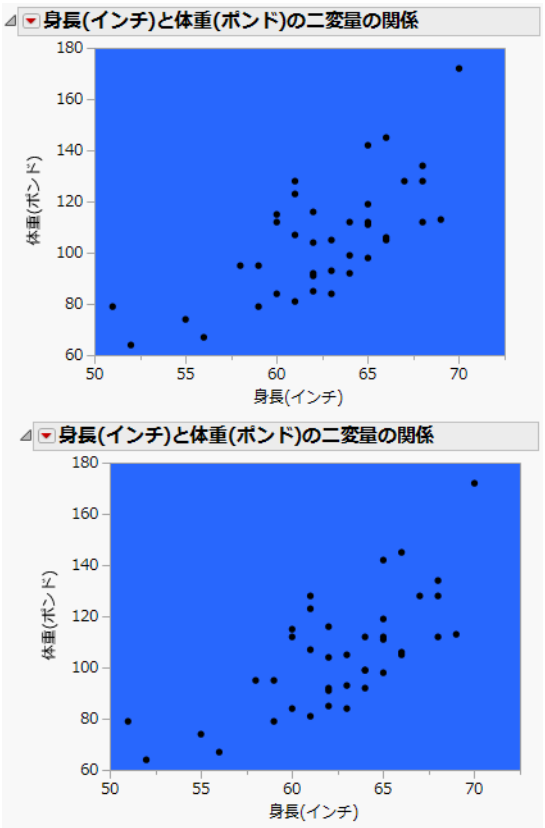


次の例は、2 つの二変量グラフを作成し、最初のグラフの背景を青に設定し、その背景の設定をコピーして 2 つ目のグラフにコピーします。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );

biv1 = dt << Bivariate( Y( :Name("体重(ポンド)") ), X( :Name("身長(インチ)") ));
// 最初の二変量グラフを作成
rbiv1 = biv1 << Report; // レポートオブジェクトを作成
rbiv2 = rbiv1 << Clone Box; // 二変量グラフのコピーを作成

rbiv1 << Append( rbiv2 );
// 二変量グラフの下にコピーしたボックスを配置して
// 2 つ目の二変量グラフを作成

framebox1 = rbiv1[Frame Box( 1 )]; // レポートのフレームボックスに参照を割り当てる
framebox2 = rbiv2[Frame Box( 1 )];

framebox1 << Background Color( "Blue" );
// 背景色を framebox1 に割り当てる
```

```
framebox1 << Copy Frame Settings;
// framebox1 のフレーム設定をコピー

framebox2 << Paste Frame Settings;
// フレーム設定を framebox2 に貼り付ける
```

図12.5 背景のコピーと貼り付け



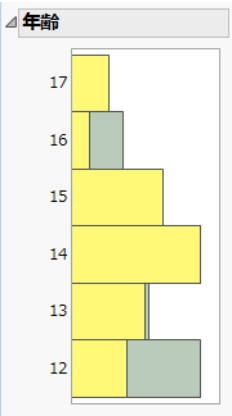
ヒストグラムのコピーと貼り付け

次の例では、2つのヒストグラムを作成し、2つ目のヒストグラムを最初のヒストグラムにコピーします。プラットフォームでは、貼り付けた要素が保存されないため、グラフを再現するには同様のスクリプトを実行する必要があります。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dist = dt << Distribution(
   SendToByGroup( {:性別 == "F"} ),
   Nominal Distribution( Column( :年齢 ) ),
   Histograms Only,
   By( :性別 ),
```

```
    SendToByGroup(
        {:性別 == "M"},
        SendToReport(
            Dispatch(
                {"一変量の分布 性別=M", "年齢"},
                "Distrib Nom Hist",
                FrameBox,
                {DispatchSeg(
                    Hist Seg( 1 ),
                    Fill Color( "Light Yellow" ),
                    // 男性のヒストグラムの色を設定

                )}
            )
        )
    )
);

For( i = 2, i <= N Items( dist ), i++,
    Report( dist[i] )[FrameBox( 1 )] << Copy Frame Contents;
    // 2つ目のヒストグラムを最初のヒストグラムにコピー
    Report( dist[1] )[FrameBox( 1 )] << Paste Frame Contents;
);

New Window( "一変量の分布", Outline Box( "年齢", Report( dist[1] )[Picture Box( 1
    )] ) );
```

図12.6 貼り付けたヒストグラム



フレームの内容と設定をインタラクティブにコピーし、貼り付ける

フレームの設定をインタラクティブにコピーし、貼り付けるには、グラフを右クリックして［**編集**］>［**フレーム設定のコピー**］を選択します。設定を貼り付けたいフレームを右クリックし、［**編集**］>［**フレーム設定の貼り付け**］を選択します。

フレームの内容をインタラクティブにコピーし、貼り付けるには、グラフを右クリックして［**編集**］>［**フレーム内容のコピー**］を選択します。内容を貼り付けたいフレームを右クリックし、［**編集**］>［**フレーム内容の貼り付け**］を選択します。

独自のグラフを始めから作成する

独自のグラフを作成したい場合には、New Window() コマンド内の Graph Box() コマンド内にグラフィックスクリプトを設定します。

```
New Window("title", <arguments>, Graph Box( named arguments,..., script));
```

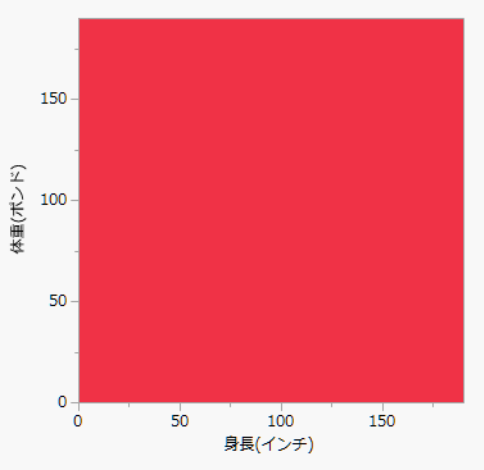
以下の引数は、Graph Box() の名前付き引数です。

```
Frame Size( horizontal, vertical ), // ピクセル単位のフレームサイズ
X Scale( xmin, xmax ), Y Scale( ymin, ymax ), // x、y 軸の範囲
X Name( "x" ), Y Name( "y" ), // x、y 軸の名前
Suppress Axes // 軸を除く
```

たとえば、Graph Box() は、次の例にあるような名前付き引数を受け入れ、背景の赤いグラフを作成します。

```
win = New Window( "名前付き引数",
   Graph Box(
      Frame Size( 300, 300 ), // Graph Box を作成
      X Scale( 0, 190 ), // x 軸の目盛り間隔を設定
      Y Scale( 0, 190), // y 軸の目盛り間隔を設定
      X Name( "身長 ( インチ )" ),
      Y Name( "体重 ( ポンド )" ),
      <<Background Color( "Red" ) // 背景色を設定
   )
);
```

図12.7 グラフの作成



名前付き引数の代わりに、send <<演算子を使ってGraph Boxにコマンドを送ることもできます。次の例も背景の赤いグラフを作成しています。

```
win = New Window( "メッセージ",
   Graph Box(
      <<Frame Size( 300, 300 ),
      <<XAxis( 0, 190 ),
      <<YAxis( 0, 190 ),
      <<Background Color( "Red" )
   )
);
```

グラフのカスタマイズ

スクリプトの中でグラフに変更を加えることもできます。たとえば、グラフを含むウィンドウを作成し、レポートへの参照を取得し、フレームボックスのサイズを設定します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
biv = dt << Bivariate( Y( :Name("体重(ポンド)") ), X( :Name("身長(インチ)") ), Fit
   Line );
rbiv = biv << Report;
rbiv[Frame Box( 1 )] << Frame Size( 400, 400 );
```

任意のディスプレイボックスオブジェクトで使用できるメッセージのリストを確認するには、[**ヘルプ**] > [**スクリプトの索引**] を選択し、リストから［**ディスプレイボックス**］を選択します。「スクリプトの索引」の代わりに Show Properties() コマンドを使用する方法もあります。たとえば、以下は、軸に送ることができるメッセージリストの一部です。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
biv = dt << Bivariate( Y( :Name("体重(ポンド)") ), X( :Name("身長(インチ)") ), Fit
  Line );
rbiv = biv << Report;
Show Properties( rbiv[Axis Box( 1 )] );
  Axis Settings [アクション](Bring up the Axis window to change various settings.)
  Revert Axis [アクション](Restore the settings that this axis had originally.)
  Add Axis Label [アクション]
  Remove Axis Label [アクション]
  ...
```

以下の式は、両方の軸ラベル（上の例の「**体重(ポンド)**」と「**身長(インチ)**」）のフォントを12ポイントで斜体（Italic）の「ＭＳ Ｐ 明朝」に変更します。

```
rbiv[Text Edit Box( 1 )] << Set Font( "ＭＳ Ｐ 明朝", 12, Italic );
rbiv[Text Edit Box( 2 )] << Set Font( "ＭＳ Ｐ 明朝", 12, Italic );
```

または

```
rbiv[Text Edit Box( 1 )] << Set Font( "ＭＳ Ｐ 明朝" );
rbiv[Text Edit Box( 1 )] << Set Font Style( "Italic" );
rbiv[Text Edit Box( 1 )] << Set Font Size( 12 );
rbiv[Text Edit Box( 2 )] << Set Font( "ＭＳ Ｐ 明朝" );
rbiv[Text Edit Box( 2 )] << Set Font Style( "Italic" );
rbiv[Text Edit Box( 2 )] << Set Font Size( 12 );
```

指定したフォントがコンピュータにインストールされていない場合は、環境設定で指定されているデフォルトのフォントが使用されます。

バブルプロットで形状をカスタマイズする

バブルの形状をカスタマイズするには、Set Shape()メッセージを使用します。円、三角形、四角形、ひし形、矢印を指定できます。その他の形状にしたいときは、Customオプションを使用します。

次の例では、Customオプションを使って星形のバブルを作成します。

```
Open("$SAMPLE_DATA/SATByYear.jmp");
Bubble Plot(
      X( :数学 ),
      Y( :言語 ),
      Sizes( :Name( "ACT 受験率 (%)(2004)" ) ),
      Time( :年 ),
      ID( :地域, :州 ),
      Label( "All" ),
      Set Custom Path(
            "M0.00000000043773431269,-1.07460777055145
  L0.299096856653163,-0.408779440996587 L1.02476323781679,-0.330073659953875
  L0.483948891847551,0.160136612800349 L0.633338521445479,0.874607770551445
  L0.00000000043773431269,0.511746026223682 L-0.633338539628289,0.874607770551445
  L-0.483948883092865,0.160136612800349 L-1.02476323781679,-0.330073713828868
```

```
L-0.299096847898477,-0.408779434262213 L0.0000000043773431269,-1.07460777055145
z"
    ),
    Set Shape( "Custom" ),
    Title Position( 505.3, 566.5 ),
    SendToReport(
        Dispatch(
            {},
            "1",
            ScaleBox,
            {Min( 490 ), Max( 580 ), Inc( 10 ), Minor Ticks( 0 )}
        ),
        Dispatch(
            {},
            "2",
            ScaleBox,
            {Min( 490 ), Max( 580 ), Inc( 10 ), Minor Ticks( 0 )}
        ),
        Dispatch( {}, "Bubble Plot", FrameBox, {Frame Size( 619, 336 )} )
    )
);
```

グラフ要素

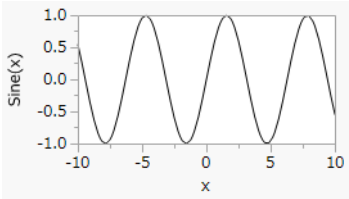
以下は、Graph Box() ステートメント内で使用できるコマンドです。この章では、グラフィック専用の JSL を扱いますが、For や While などの一般的なスクリプトコマンドを使用することもできます。一般的なスクリプトコマンドである For や While などは、プラットフォームや New Window() 関数の内部での使用に適しません。

プロット関数

YFunction 関数は、滑らかな連続関数を描くために使います。最初の引数は、プロットする式です。第 2 引数は、式の中の X 変数の名前です。

```
win = New Window( "正弦関数",
   Graph Box(
      Frame Size( 200, 100 ),
      X Scale( -10, 10 ),
      Y Scale( -1, 1 ),
      X Name( "x" ),
      Y Name( "Sine(x)" ),
      Y Function( Sine( x ), x ) ) );
```

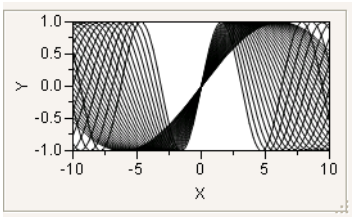
図12.8 正弦波



Forループを使って、複数の正弦波を重ね合わせることもできます。

```
win = New Window( "重ね合わせた正弦波",
   Graph Box(
      Frame Size( 200, 100 ),
      X Scale( -10, 10 ),
      Y Scale( -1, 1 ),
      For( i = 1, i <= 4, i += .1,
         Y Function( Sine( x / i ), x )
      )
   )
);
```

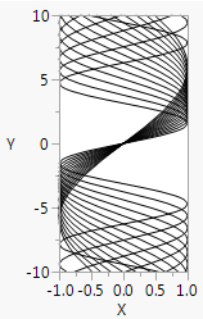
図12.9 重ね合わせた正弦波



同様に、XFunctionを使うと、y変数の値の変化に従ってxの値が変化するグラフを描くことができます。

```
win = New Window( "重ね合わせた正弦波",
   Graph Box(
      Frame Size( 100, 200 ),
      X Scale( -1, 1 ),
      Y Scale( -10, 10 ),
      For( i = 1, i <= 4, i += .2,
         X Function( Sine( y / i ), y )
      )
   )
);
```

図12.10 X軸に沿って重ね合わせた正弦波

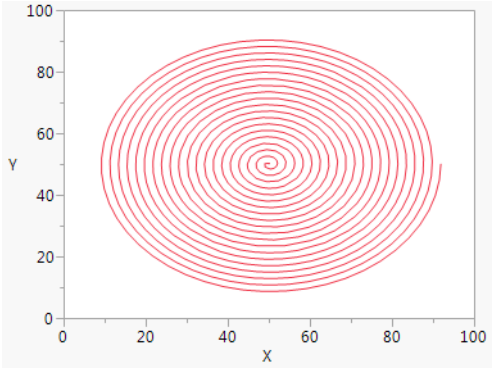


XY Function() は、第3の変数に依存する2つの計算式（パラメトリック方程式）を使って滑らかな曲線を描きます。第3変数の値は、最小値から最大値へと変化して、X-Yのペアを生成します。

```
win = New Window( "渦巻き",
   Graph Box(
      Pen Color( "red" ); // 線は赤色
      xCenter = 50; // X軸における曲線の中心の位置
      yCenter = 50; // Y軸における曲線の中心の位置
      minAngle = 0;
      maxAngle = Pi() * 2 * 20;
      XY Function(
         xCenter + ((ta / 3) * Cos( ta )),
         yCenter + ((ta / 3) * Sin( ta )),
         ta,
         Min( minAngle ),
         Max( maxAngle ),
         Inc( Pi() / 100 )
      );
   )
);
```

この例では、Sin() と Cos() が taを引数（角度）、**および**倍数（広がり）として使用しています。（Sin() と Cos() は度ではなくラジアンを使用します。）

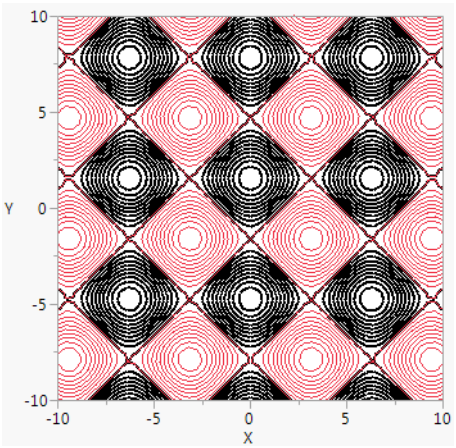
図12.11 渦巻きのパラメトリックプロット



ContourFunction()は、2次元空間で3次元の関数を表現するための方法です。最後の引数には、等高線の値を指定します。単一の値や、::を使って作成した等間隔の値の行列、または任意の値の行列を指定できます。

```
win = New Window( "卵ケースの鳥瞰図",
   Graph Box(
      Frame Size( 300, 300 ),
      X Scale( -10, 10 ),
      Y Scale( -10, 10 ),
      Pen Color( "black" );
      Pen Size( 2 );
      Contour Function( Sine( y ) + Cosine( x ), x, y, (0 :: 20) / 5 );
      Pen Color( "red" );
      Pen Size( 1 );
      Contour Function( Sine( y ) + Cosine( x ), x, y, (-20 :: 0) / 5 );
   )
);
```

図12.12 卵ケース



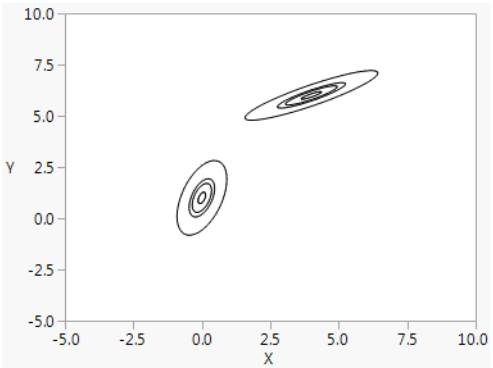
Normal Contour()は、2変量正規分布の等高線を描きます。k個の母集団を指定することができます。最初の引数は、等高線の確率を指定するスカラーまたは行列です。後続の引数は、平均、標準偏差、相関係数を指定する行列です。平均と標準偏差の行列の大きさは$k \times 2$です。相関係数行列の大きさは$k \times 1$で、最初の行は最初の等高線に対応し、2番目の行は2番目の等高線に対応し、以下同様に対応します。最初の列はx、2番目の列はyに対応します。次の例を見てみましょう。

```
Normal Contour(
   [ prob1,
     prob2,
     prob3, ...],
   [ xmean1 ymean1,
     xmean2 ymean2,
     xmean3 ymean3, ...],
   [ xsd1 ysd1,
     xsd2 ysd2,
     xsd3 ysd3, ...],
   [ xycorr1,
     xycorr2,
    xycorr3, ...]);
```

以下のスクリプトは、2母集団の2変量正規分布の累積確率0.1、0.5、0.7、および0.99における等高線を描きます。最初の母集団は、xの平均0、yの平均1、xの標準偏差0.3、yの標準偏差0.6、相関係数0.5です。2つ目の母集団は、xの平均4、yの平均6、xの標準偏差0.8、yの標準偏差0.4、相関係数0.9です。

```
win = New Window( "正規確率の等高線",
   Graph Box(
      X Scale( -5, 10 ),
      Y Scale( -5, 10 ),
      Normal Contour( [.1, .5, .7, .99], [0 1, 4 6], [.3 .6, .8 .4], [.5, .9] )
   )
);
```

図12.13 Normal Contour関数



Normal Contour()は、二変量プラットフォームで作成できる確率楕円を作成する一般的な方法です。確率楕円を見るには、サンプルデータの「Football.jmp」を開き、「二変量」スクリプトを実行してください。

Gradient Function

Gradient Function()は、式で指定された色でグリッド上の長方形を塗りつぶします。

```
Gradient Function( expression, xname, yname, [zlow, zhigh], ZColor( [colorLow,
   colorHigh] ), <XGrid( min, max, incr )>, <YGrid( min, max, incr )>,
   <Transparency( t )>;
```

Gradient Function()を実行するには、次の構文を使用します。

GradientFunction(	
expression	等高線の式。式は、2変数の関数でなければいけません。
xname,yname,	式に含まれている2つの変数の名前。
[zlow, zhigh],	式の上下限値。この2つの値を補間して色が決められます。
ZColor([colorLow, colorHigh])	上限値と下限値それぞれに対応する色
<XGrid(min, max, incr),> <YGrid(min, max, incr)>);	グリッドの指定（オプション）

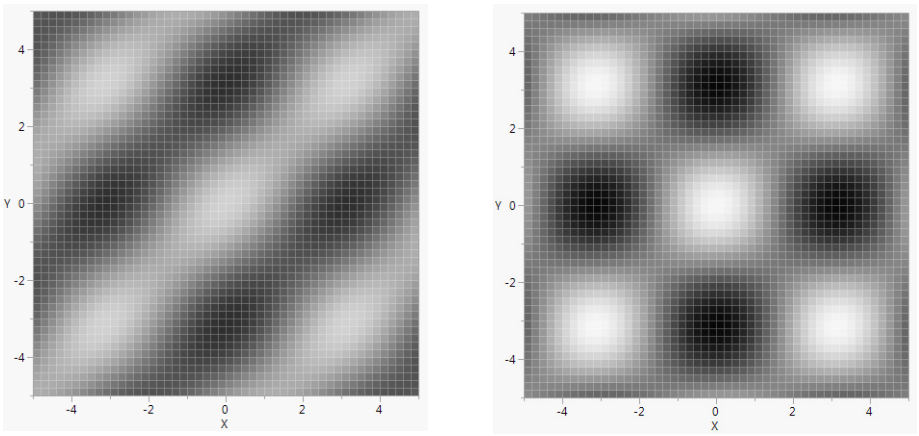
ZColor()値は名前ではなく数値コードでなければなりません。「色を指定する」（549ページ）に記載されているカラー表の番号を使うことができます（0=black（黒）、1=grey（グレー）、2=white（白）、3=red（赤）、4=green（緑）、5=blue（青）など）。

次の例では、Gradient Function()を使って2つのグラフのアニメーションを作成します。

```
phase = 0.7;
win = New Window( "Gradient Function",
   a = Graph(
```

```
        Frame Size( 400, 400 ),
        X Scale( -5, 5 ),
        Y Scale( -5, 5 ),
        Gradient Function(
           phase * Sine( x ) * Sine( y ) + (1 - phase) * Cosine( x ) * Cosine( y ),
           x,
           y,
           [-1 1],
           zcolor( [0, 2] )
        )
     )
  );
  b = a[FrameBox( 1 )];
  For( i = 1, i <= 5, i++,
     For( phase = 0, phase < 1, phase += 0.05,
        b << Reshow;
        Wait( 0.01 );
     );
     For( phase = 1, phase > 0, phase -= 0.05,
        b << Reshow;
        Wait( 0.01 );
     );
  );
```

図12.14 Gradient Function



グラフフレームのプロパティを取得する

既存のグラフフレームのプロパティを取得するには、以下の関数を使うと便利です。

H Size　グラフフレームの横のサイズをピクセル単位で戻す。

V Size　グラフフレームの縦のサイズをピクセル単位で戻す。

X Origin　グラフフレームの左端のx値を戻す。右端のx値は、XOrigin()+XRange()により得ることができます。

X Range　グラフフレームの左端から右端までの距離を戻す。

Y Origin　グラフフレームの下端のy値を戻す。

Y Range　ディスプレイボックスの下端から上端までの距離を戻す。

次の式では、最初の行が右端を計算し、2番目の行が上端を計算します。

```
Oval(
   ...,
   rightEdge = X Origin() + X Range();
   topEdge = Y Origin() + Y Range();
);
```

凡例を追加する

グラフに凡例を追加するには、フレームボックスにRow Legendメッセージを送ります。Row Legendメッセージでは、凡例の基準にする列と、凡例に色およびマーカーを適用するかどうかを指定します。

たとえば「Big Class.jmp」なら、次のJSLにより、「**年齢**」列に基づく凡例を設定することができます。色とマーカーが「**年齢**」列の値に応じて設定されます。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
biv = dt << Bivariate( y( :Name("体重(ポンド)") ), x( :Name("身長(インチ)") ) );
rbiv = biv << Report;
framebox = rbiv[Frame Box( 1 )];
framebox << Row Legend( "年齢", Color( 1 ), Marker( 1 ) );
// 「年齢」というタイトルの凡例を作成
// 凡例に色とマーカーを表示
```

Color()引数とMarker()引数はオプションです。デフォルトでは、色はオン、マーカーはオフに設定されています。

名義尺度または順序尺度の変数に対して、連続的な色使いを適用したい場合には、Continuous Scale(1)を使用します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
biv = dt << Bivariate( Y( :Name("体重(ポンド)") ), X( :Name("身長(インチ)") ) );
rbiv = biv << Report;
framebox = rbiv[Frame Box( 1 )];
framebox << Row Legend( "年齢", Color( 1 ), Continuous Scale( 1 ) );
```

凡例をインタラクティブに追加するには、グラフを右クリックし、[**行の凡例**]を選択し、列の設定を変更します。

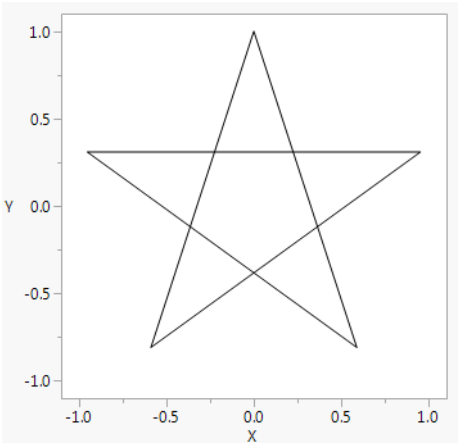
直線、矢印、点、形状、テキストを追加する

線を描く

Line()は点の間に線を引きます。

```
win = New Window( "星",
   Graph Box(
      Frame Size( 300, 300 ),
      X Scale( -1.1, 1.1 ),
      Y Scale( -1.1, 1.1 ),
      Line(
         {Cos( 1 * Pi() / 10 ), Sin( 1 * Pi() / 10 )},
         {Cos( 9 * Pi() / 10 ), Sin( 9 * Pi() / 10 )},
         {Cos( 17 * Pi() / 10 ), Sin( 17 * Pi() / 10 )},
         {Cos( 5 * Pi() / 10 ), Sin( 5 * Pi() / 10 )},
         {Cos( 13 * Pi() / 10 ), Sin( 13 * Pi() / 10 )},
         {Cos( 1 * Pi() / 10 ), Sin( 1 * Pi() / 10 )}
      )
   )
);
```

図12.15 線を使って星を描く



点は、上に示したような2項目のリストか、または*x*座標と*y*座標の行列で指定できます。行列の値は、1行目から順番に使われていくので、要素の数が同じであれば、行ベクトルと列ベクトルのどちらでも使用できます。以下の式は、まったく同じ効果を持ちます。

```
Line( {1,2}, {3,0}, {2,4} ); // 複数の {x,y} リスト
Line( [1 3 2],[2 0 4] ); // 行のベクトル
Line( [1,3,2], [2,0,4] ); // 列のベクトル
```

```
Line( [1 3 2], [2,0,4] ); // 行ベクトルと列ベクトルの組み合わせ
```

したがって、星の例は、以下のような方法で描くこともできます。行列の値を式で入力しているので、簡略形の[ ]ではなく、正式なMatrix({...})表記を使う必要があります。次の例は、Matrix()関数を使用しています。

```
win = New Window( "星",
   Graph Box(
      Frame Size( 300, 300 ),
      X Scale( -1.1, 1.1 ),
      Y Scale( -1.1, 1.1 ),
      Line(
         Matrix(
            { // x座標
            Cos( 1 * Pi() / 10 ), Cos( 9 * Pi() / 10 ), Cos( 17 * Pi() / 10 ),
            Cos( 5 * Pi() / 10 ), Cos( 13 * Pi() / 10 ), Cos( 1 * Pi() / 10 )}
         ),
         Matrix(
            { // y座標
            Sin( 1 * Pi() / 10 ), Sin( 9 * Pi() / 10 ), Sin( 17 * Pi() / 10 ),
            Sin( 5 * Pi() / 10 ), Sin( 13 * Pi() / 10 ), Sin( 1 * Pi() / 10 )}
         )
      ) ) );
```

HLine()は、グラフ上の指定されたy値の位置に横線を引きます。同様に、VLine()は、グラフ上の指定されたx値の位置に縦線を引きます。どちらの関数でも、引数に行列を指定することによって、複数の線を描くことができます。詳細は、「Mousetrap()」（562ページ）の節を参照してください。

矢印を描く

Arrow()は、最初の点から次の点までの矢印を描きます。デフォルトの矢じりの長さは、(矢印の長さの平方根＋1)/2です。矢じりの長さを設定するには、オプションの第1引数を使って矢じりの長さをピクセルで設定します。次の例は、単純な矢印を描きます。

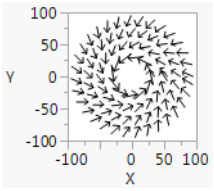
```
win = New Window( "矢じり",
   Graph Box(
      Pen Size( 4 );
      Arrow( 20, [10 30 90], [88 22 44] );
      // 行列1は10，88から
      // 30，22（Y座標）までの矢印を定義する
      // 行列2はその終わりから
      // 90，44（X座標）までの矢印を定義する
   )
);
```

次の例は、矢印を円状に描きます。

```
win = New Window( "ハリケーン",
```

```
   Graph Box(
      Frame Size( 100, 100 ),
      X Scale( -100, 100 ),
      Y Scale( -100, 100 ),
      For( r = 35, r < 100, r += 20,
         ainc = 2 * Pi() * 3 / r;
         For( a = 0, a < 2 * Pi(), a += ainc,
            x = r * Cosine( a );
            y = r * Sine( a );
            aa = a + ainc * 45 / r;
            rr = r - r / 6;
            x2 = rr * Cosine( aa );
            y2 = rr * Sine( aa );
            Arrow( {x, y}, {x2, y2} );
            // リスト 1 は最初の矢印の座標を定義
            // リスト 2 は 2 番目の矢印の座標を定義
         );
      )
   )
);
```

図12.16 矢印を描く

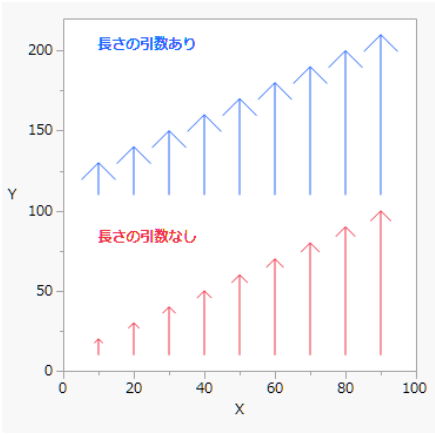


次の例は、指定された長さ（19ピクセル）とデフォルトの長さで矢じりを描いて比較します。

```
win = New Window( "矢じり",
   Graph Box(
      Frame Size( 300, 300 ),
      X Scale( 0, 100 ),
      Y Scale( 0, 220 ),
      x = 10;
      y1 = 10;
      y2 = y1 + 10;
      For( i = 1, i < 10, i++,
         Pen Color( "Red" );
         Arrow( {x, y1}, {x, y2} );

         y2 += 10;
         y1 += 100;
         y2 += 100;
```

```
            Pen Color( "Blue" );
            Arrow( 20, {x, y1}, {x, y2} );

            x += 10;
            y1 -= 100;
            y2 -= 100;

            Text Color( "Red" );
            Text( {10, 80}, "長さの引数なし" );

            Text Color( "Blue" );
            Text( {10, 200}, "長さの引数あり" );
        );
    )
);
```

図12.17 矢じりのサイズ



Line()と同様に、上に示したような2項目のリスト、もしくは、*x*座標と*y*座標の行列によって座標を指定できます。

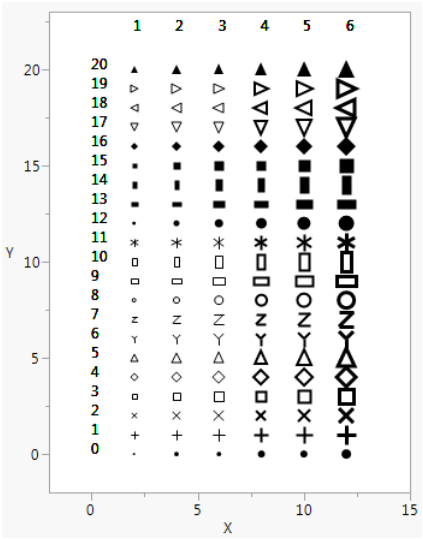
マーカーを描く

Marker()は、第1引数で指定されたタイプ（1～15）のマーカーを、第2引数で指定された座標点に描きます。Marker Size()は、マーカーのサイズを0～6（ドット～XXXL）に設定します。マーカーを環境設定で指定されているサイズに設定するには、-1の値を使用します。

```
ymax = 20;
win = New Window( "マーカー",
    Graph Box(
        Frame Size( 300, 400 ),
        X Scale( -2, ymax - 5 ),
```

```
        Y Scale( -2, ymax + 3 ),
        For( j = 1, j < 7, j++,
           Marker Size( j );
           For( i = 0, i < (ymax + 1), i++,
              Marker( i, {j * 2, i} );
              Text( {0, i}, i );
              Text( {j * 2, ymax + 2}, j );
           );
        )
     )
  );
```

図12.18 マーカーを描く



マーカータイプの引数の前後、またはその代わりに、行の属性を引数として指定することもできます。Combine States() を使うと、Marker() に複数の行の属性を設定できます。以下に示す例を、上記のスクリプトの中で試してみてください。

```
Marker( i, Color State( i ), {j * 2, i} );
Marker( Color State( i ), i, {j * 2, i} );
Marker(
   Combine States( Color State( i ), Marker State( i ), Hidden State( i ) ),
   {j * 2, i}
);
```

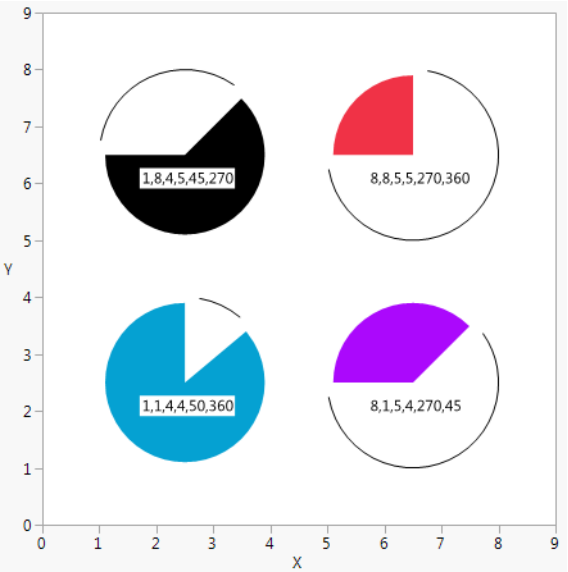
Line() と Arrow() では、点を *x* 座標と *y* 座標の行列で指定することもできます。

扇形と円弧を描く

Pie() と Arc() は、それぞれ扇形と円弧を描きます。最初の4つの引数は、*x1*、*y1*、*x2*、*y2* で、内接する長方形の座標です。最後の2つの引数は、度単位の開始角度と終了角度で、時計回りに円弧または扇形が描画されます。このとき、0度は時計の12時の位置とします。

```
win = New Window( "扇形と円弧",
   Graph Box(
      Frame Size( 400, 400 ),
      X Scale( 0, 9 ),
      Y Scale( 0, 9 ),
      Fill Color( "Black" ), // 左上
      Pie( 1.1, 7.9, 3.9, 5.1, 45, 270 ),
      Text( Erased, {1.75, 6}, "1,8,4,5,45,270" ),
      Arc( 8, 1, 4, 5, 280, 35 ),
      Fill Color( "Red" ), // 右上
      Pie( 7.9, 7.9, 5.1, 5.1, 270, 360 ),
      Text( Erased, {5.75, 6}, "8,8,5,5,270,360" ),
      Arc( 8, 8, 5, 5, 370, 260 ),
      Fill Color( "BlueCyan" ), // 左下
      Pie( 1.1, 1.1, 3.9, 3.9, 50, 360 ),
      Text( Erased, {1.75, 2}, "1,1,4,4,50,360" ),
      Arc( 1, 1, 4, 4, 370, 40 ),
      Fill Color( "Purple" ), // 右下
      Pie( 7.9, 1.1, 5.1, 3.9, 270, 45 ),
      Text( Erased, {5.75, 2}, "8,1,5,4,270,45" ),
      Arc( 8, 1, 5, 4, 55, 260 )
   )
);
```

図12.19 扇形と円弧を描く



一般的な図形：円、長方形、楕円を描く

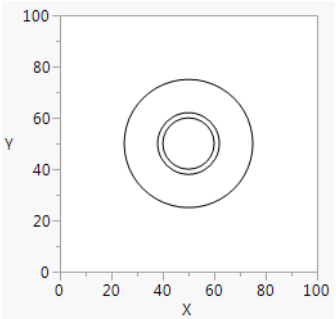
円

`Circle()` は、中心点と半径を指定して円を描きます。後続の引数は追加の半径を指定します。

```
win = New Window( "円",
   Graph Box( Frame Size( 200, 200 ),
      Circle( {50, 50}, 10, 12, 25 )
   )
);
```

`Circle()` には、オプションの最後の引数 `"fill"` もあります。これを指定すると、関数内に定義された円はすべて、現在 fill color で指定されている色で塗りつぶされます。`"fill"` が省略された場合、円は塗りつぶされません。

図12.20 円を描く



グラフの縦横比を変更しても、Circle関数によって描かれた円は常に円のままです。グラフのサイズを変更した際に、それに合わせて円も異なる縦横比に変更されるようにしたい場合には、楕円を描くOval関数を使用してください。

グラフのサイズを変更しても、円のサイズは変更されないようにしたい場合には、半径をピクセルで指定してください。

```
win = New Window( "円",
   Graph Box(
      Frame Size( 200, 200 ),
      Circle( {50, 50}, Pixel Radius( 10 ), Pixel Radius( 12 ), Pixel Radius( 25 )
   )
   )
);
```

長方形

Rect()は、指定された対角座標に基づいて長方形を描きます。座標は順序に従った4つの引数（*左*、*上*、*右*、*下*）か、またはペアのリスト（{*左*,*上*}、{*右*, *下*}）で指定できます。

```
win = New Window( "長方形",
   Graph Box(
      Frame Size( 200, 200 ),
      Pen Color( 1 );
      Rect( 0, 40, 60, 0 );
      Pen Color( 3 );
      Rect( 10, 60, 70, 10 );
      Pen Color( 4 );
      Rect( 50, 90, 90, 50 );
      Pen Color( 5 );
      Rect( 0, 80, 70, 70 );
   )
);
```

Rect()には、オプションの第5引数*fill*（塗りつぶし）があります。長方形を塗りつぶさない場合は0を、塗りつぶす場合は1を指定します。*fill*のデフォルト値は0です。長方形は、現在fill colorで指定されている色で塗りつぶされます。

塗りつぶし（fill）の引数が負の値の場合は、塗りつぶしなしの1ピクセルの枠が生成されます。

```
win = New Window( "長方形の枠",
   Graph Box( Frame Size( 200, 200 ), Rect( 0, 40, 60, 0, -1 ) )
);
```

楕円

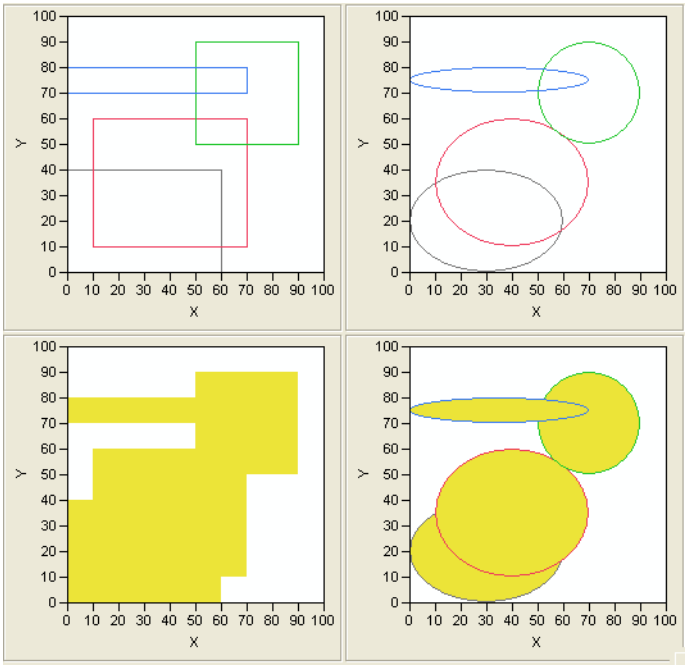
Oval()は、引数*x1*、*y1*、*x2*、*y2*で指定された長方形に内接する楕円を描きます。

```
win = New Window( "楕円",
   Graph Box(
      Frame Size( 200, 200 ),
      Pen Color( 1 );
      Oval( 0, 40, 60, 0 );
      Pen Color( 3 );
      Oval( 10, 60, 70, 10 );
      Pen Color( 4 );
      Oval( 50, 90, 90, 50 );
      Pen Color( 5 );
      Oval( 0, 80, 70, 70 );
   )
);
```

Oval()にも、オプションの第5引数*fill*（塗りつぶし）があります。楕円を塗りつぶさない場合は0を、塗りつぶす場合は1を指定します。*fill*のデフォルト値は0です。楕円は、現在fill colorで指定されている色で塗りつぶされます。

図12.21は、長方形と楕円のグラフです。塗りつぶした場合と、塗りつぶしていない場合の両方を示しています。塗りつぶしを行ったグラフを見ると、長方形には輪郭がなく、楕円には輪郭があります。塗りつぶした長方形に輪郭を描きたい場合は、まず塗りつぶした長方形を描いてから、その後、同じ座標を使用して塗りつぶしていない長方形を描いてください。

図12.21 長方形と楕円（塗りつぶしあり／なし）



その他の図形：多角形と等高線を描く

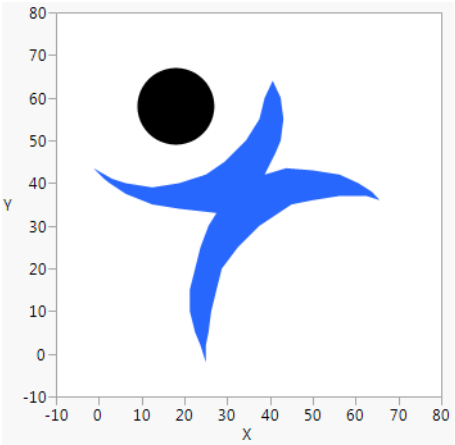
多角形

Polygon() は、Line() と同様に点をつなげます。Polygon() は、さらに最後の点と最初の点を結んで多角形を閉じ、多角形の内部を塗りつぶします。各点の座標は、2項目のリストで個々に指定する（「マーカーを描く」（539ページ））ことも、*x*座標と*y*座標の行列で指定することもできます。行列の値は、1行目から順番に使われていくので、要素の数が同じであれば、行ベクトルと列ベクトルのどちらでも使用できます。次のスクリプトでは、点の行列を設定した後で、その行列を Polygon() に指定しています。

```
gCoordX = [25, 23.75, 22.5, 21.25, 21.25, 22.5, 23.75, 25.625, 27.5, 18.75, 12.5,
   6.25, 2.5,
1.25, -1.25, 3.125, 6.25, 12.5, 18.75, 25, 29.375, 34.375, 37.5, 38.75, 40.625,
   42.5, 43.125,
42.5, 41.25, 38.75, 43.75, 50, 56.25, 60.625, 63.75, 65.625, 62.5, 56.25, 50, 45,
   37.5, 32.5,
28.75, 27.5, 26.25, 25.625, 25];
gCoordY = [-2, 2, 5, 10, 15, 20, 25, 30, 33, 34, 35, 37.5, 40, 41, 43.5, 41, 40,
   39, 40, 42, 45,
50, 55, 60, 64, 60, 55, 50, 47, 42, 43.5, 43, 42, 40, 38, 36, 37, 37, 36, 35, 30,
   25, 20, 15, 10, 5, 2];
win = New Window( "JMP マン ",
```

```
    Graph Box(
        Frame Size( 300, 300 ),
        X Scale( -10, 80 ),
        Y Scale( -10, 80 ),
        Pen Color( "Black" );
        Fill Color( "Blue" );
        Polygon( gCoordX, gCoordY );
        Fill Color( "Black" );
        Circle( {18, 58}, 9, "FILL" );
    )
);
```

図12.22 多角形を描く



関連したコマンドのIn Polygon()は、与えられた点が、指定された多角形の中に入るかどうかを知らせます。次のコードでは、いくつかの点が図12.22のJMPマン内にあるかどうかを調べています。

```
In Polygon( 0,60, GcoordX,GCoordY );  // 0を戻す
In Polygon( 30,38, GcoordX,GCoordY ); // 1を戻す
```

In Polygon()をJMPマンのスクリプトに追加することもできます。次のスクリプトを実行し、グラフ内のさまざまな位置をクリックして、ログウィンドウを確認してください。体の部分をクリックすると、ログに「In」と「out」が出力されます。

```
gCoordX = [25, 23.75, 22.5, 21.25, 21.25, 22.5, 23.75, 25.625, 27.5, 18.75, 12.5,
  6.25, 2.5, 1.25, -1.25, 3.125, 6.25, 12.5, 18.75, 25, 29.375, 34.375, 37.5,
  38.75, 40.625, 42.5, 43.125, 42.5, 41.25, 38.75, 43.75, 50, 56.25, 60.625,
  63.75, 65.625, 62.5, 56.25, 50, 45, 37.5, 32.5, 28.75, 27.5, 26.25, 25.625, 25];
gCoordY = [-2, 2, 5, 10, 15, 20, 25, 30, 33, 34, 35, 37.5, 40, 41, 43.5, 41, 40,
  39, 40, 42, 45, 50, 55, 60, 64, 60, 55, 50, 47, 42, 43.5, 43, 42, 40, 38, 36, 37,
  37, 36, 35, 30, 25, 20, 15, 10, 5, 2];
win = New Window( "JMPマン",
```

```
	Graph Box(
		Frame Size( 300, 300 ),
		X Scale( -10, 80 ),
		Y Scale( -10, 80 ),
		Pen Color( "Black" );
		Fill Color( "Black" );
		Polygon( gCoordX, gCoordY );
		Mousetrap(
			{},
			Print( If( In Polygon( x, y, gCoordX, gCoordY ), "in", "out" ) )
		);
	)
);
```

等高線

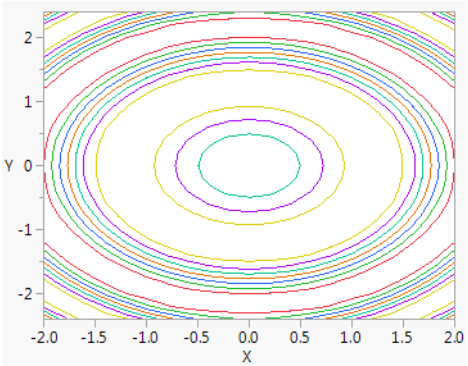
Contour()は、座標のグリッドを使って等高線を描きます。

```
Contour( xVector, yVector, zGridMatrix, zContour, <zColors> );
```

*n*個、**m**個の値から成る*xVector*および*yVector*で定義された曲面に、**n** x **m**行列の*zGridMatrix*で表される曲線があるとします。Contour()は、*zContour*の値で定義された等高線を、*zColors*で定義された色で描きます。以下はその例です。

```
x = (-10 :: 10) / 5;
y = (-12 :: 12) / 5;
grid = J( 21, 25, 0 );
z = [-.75, -.5, -.25, 0, .25, .5, .75];
zcolor = [3, 4, 5, 6, 7, 8, 9];
For( i = 1, i <= 21, i++,
	For( j = 1, j <= 25, j++,
		grid[i, j] = Sin( (x[i]) ^ 2 + (y[j]) ^ 2 )
	)
);
Show( grid );
win = New Window( "ハット",
	Graph Box(
		X Scale( -2, 2 ),
		Y Scale( -2.4, 2.4 ),
		Contour( x, y, grid, z, zcolor )
	)
);
```

図12.23 等高線を描く



テキストを追加する

Text()を使うと、任意の位置にテキストを表示できます。

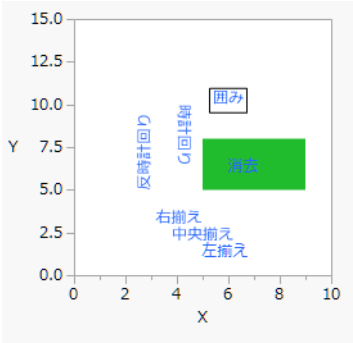
```
Text( <properties>, ( {x, y}|{left, bottom, right, top} ), "text" );
```

位置とテキストは、任意の順序でいくつでも指定できます。座標とテキストの引数に加えて、最初の引数に、Center Justified（中央揃え）、Right Justified（右揃え）、Erased（消去）、Boxed（囲み）、Counterclockwise（反時計回り）、Clockwise（時計回り）をオプションで指定できます。Erasedは、グラフ内のテキストを見にくくするものを削除するためのコマンドです。テキストの背後に背景色付きの長方形を作成します。

以下の例で、Erasedを指定したテキストが、緑の長方形の上の白いボックス内に、どのように表示されるかを見てください。

```
mytext = New Window( "テキストの追加",
   Graph Box(
      Frame Size( 200, 200 ),
      Y Scale( 0, 15 ),
      X Scale( 0, 10 ),
      Text Size( 9 );
      Text Color( "blue" );
      Text( {5, 1}, "左揃え" );
      Text( Center Justified, {5, 2}, "中央揃え" );
      Text( Right Justified, {5, 3}, "右揃え" );
      Fill Color( 4 );
      Rect( 5, 8, 9, 5, 1 );
      Text( Erased, {6, 6}, "消去" );
      Text( Boxed, {6, 10}, "囲み" );
      Text( Clockwise, {4, 10}, "時計回り" );
      Text( Counterclockwise, {3, 5}, "反時計回り" );
   )
);
```

図12.24 グラフボックス内にテキストを描く



Text() 関数は、4つの座標値を引数として、その枠の中に文字列を描くこともできます。構文は次のとおりです。

```
Text( {left, top, right, bottom}, string);
```

色を指定する

以下のコマンドは、色を制御します。

- Fill Color() は塗りつぶし領域の色
- Level Color() はカテゴリカルデータの色
- Pen Color() は直線と点の色
- Back Color() はテキストの背景色（図12.24の、消去したテキストを囲むボックスと同様）
- Background Color() はグラフの背景色
- Font Color() は追加テキストの色

塗りつぶしを行った場合、Fill ColorがPen Colorを上書きします。一部の描画パッケージのように両方を同時に使うことはできません。同時に使うには、一方は塗りつぶし、もう一方は塗りつぶさない2つの図形を描きます。色は1つの数値引数、引用符で囲んだ色の名前、RGB値などで指定できます。JMPが提供している標準の色は、番号0～15（0と15は黒）または名前（表12.1）で指定することができます。

ヒント： 濃淡をつけたいときは、引用符で囲んだ色名の前に Dark、Medium Dark、Medium Light、Light といった言葉を追加します。

表12.1 JMPに用意されている標準の色

番号	Name
0	Black（黒）
1	Gray（グレー）
4	White（白）

表12.1 JMPに用意されている標準の色（続き）

番号	Name
3	Red（赤）
4	Green（緑）
5	Blue（青）
6	Orange（オレンジ）
7	BlueGreen（青緑）
8	Purple（紫）
9	Yellow（黄色）
10	Cyan（シアン）
11	Magenta（マゼンタ）
12	YellowGreen（黄緑）
13	BlueCyan（ブルーシアン）
14	Fuchsia（赤紫）

次のスクリプトは、JMPカラーがすべて揃ったグラフを作成します。

```
colors = {"Black", "Gray", "White", "Red", "Green", "Blue", "Orange", "BlueGreen",
   "Purple", "Yellow", "Cyan", "Magenta", "YellowGreen", "BlueCyan", "Fuchsia"};
ymax = 15;
win = New Window( "JMP カラー ",
   Graph Box(
      Frame Size( 300, 300 ),
      X Scale( 0, ymax ),
      Y Scale( 0, ymax + 2 ),
      For( i = 0, i < ymax, i++,
         If( colors[i + 1] == "White",
            Fill Color( 65 );
            Rect( 0, i + 1 + .5, 15, i + 1 - .5, 1 );
         );
         Pen Color( colors[i + 1] );
         Text Color( colors[i + 1] );
         H Line( i + 1 );
         Text( {2, i + 1}, i );
         Text( {5, i + 1}, colors[i + 1] );
      )
   )
);
```

16以上の番号は、同じ色の並びを濃淡の違いで繰り返します。これをデモンストレーションするスクリプトは、「データテーブル」章の「色とマーカー」（358ページ）にあります。0～84の範囲外にある値は無効です。

表12.2 番号と色のマッピング

番号	結果
16～31	濃い
32～47	淡い
48～63	非常に濃い
64～79	非常に淡い
80～84	グレーの濃淡、淡いから濃いへ

RGB値を使う場合は、赤、緑、青の順でそれぞれの含有率をリストして、各色を指定します。

```
Pen Color( {.38,.84,.67} ); // 緑がかった青
```

RGB Color()とColor to RGB()は、JMPの色番号とRGB（赤、緑、青）値との間の変換を行います。次の例は、JMPカラー3（赤）のRGB値を取得します。

```
Color to RGB( 3 );
  {0.941176470588235, 0.196078431372549, 0.274509803921569}
```

同様に、HLS Color()とColor to HLS()は、JMPの色番号とHLS値（色調、明度、彩度）との間で変換を行います。

```
win = New Window( "カラーホイール",
   Graph(
      Frame Size( 200, 200 ),
      For( hue = 0, hue < 360, hue += 30,
         y = 50 - 40 * Cos( hue * 2 * Pi() / 360 );
         x = 50 + 40 * Sin( hue * 2 * Pi() / 360 );
         Fill Color( HLS Color( hue / 360, 0.5, 1 ) );
         Oval( x - 10, y - 10, x + 10, y + 10, 1 );
      )
   )
);
```

Heat Color()は、セルプロットやツリーマップなどで使用されている任意のカラーテーマにおける値に対し、該当するJMPの色番号を戻します。構文は次のとおりです。

```
Heat Color( n, <<"theme" )
```

themeメッセージはオプションで、デフォルト値は「青->グレー->赤」（Blue to Gray to Red）です。カスタムカラーテーマを含め、任意のカラーテーマを指定できます。次の例にあるように、匿名のカラーテーマを作成し、使用することもできます。

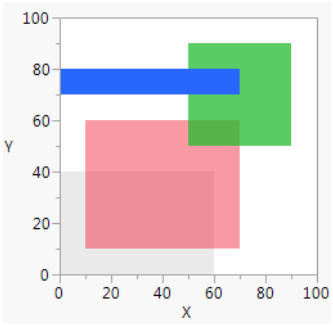
```
Heat Color( z, <<{"", {{1, 1, 0}, {0, 0, 1}} } );
Heat Color( z, <<{"", {blue, green, yellow} } );
```

透明度を指定する

フレームボックスなどのグラフィックでは、Transparency関数を使って透明度のレベルを指定できます。引数には、0～1の範囲の数値を指定します。値が0の場合は透明になり、描画されません。値1の場合は、完全に不透明になります（通常の描画モード）。中間の値を指定すると、すでに描画されているグラフィック上に、半透明の色の層が作成されます。次のスクリプトでは、長方形の透明度を指定しています。

```
win = New Window( "透明度",
   Graph Box(
      Frame Size( 200, 200 ),
      Pen Color( "gray" );
      Fill Color( "gray" );
      Transparency( 0.25 );
      Rect( 0, 40, 60, 0, 1 );

      Pen Color( "red" );
      Fill Color( "red" );
      Transparency( 0.5 );
      Rect( 10, 60, 70, 10, 1 );

      Pen Color( "green" );
      Fill Color( "green" );
      Transparency( 0.75 );
      Rect( 50, 90, 90, 50, 1 );

      Pen Color( "blue" );
      Fill Color( "blue" );
      Transparency( 1 );
      Rect( 0, 80, 70, 70, 1 );
   )
);
```

図12.25 透明度と長方形



塗りつぶしのパターンを追加する

Fill Pattern() は、塗りつぶし領域のパターンを設定します。

行列の例

マスク（0～1の値の行列）またはイメージを指定します。行列内の値は、それぞれピクセルを作成します。

```
win = New Window( "例",
   Graph Box(
      Fill Pattern(
         [1 0.5 0 0, 0.5 0 0 1, 0 0 1 0.5, 0 1 0.5 0]
      );
      Polygon( [10 30 90], [88 22 44] );
   )
);
```

名前付きパターンの例

表12.3にあるパターンの名前を引用符で囲んで指定します。

```
win = New Window( "名前付きパターンの例",
   Graph Box(
      Fill Pattern( "vertical light" );
      Polygon( [10 30 90], [88 22 44] );
   )
);
```

グラフィックの例

イメージを引用符で囲んで指定します。表示しようとしているコンピュータにイメージがインストールされていない場合は、形状の中に疑問符が表示されます。

```
win = New Window( "グラフィックの例",
Graph Box(
   Fill Pattern( Open( "$SAMPLE_IMAGES/pi.gif", "gif" ) );
   Polygon( [10 30 90], [88 22 44] );
);
```

表12.3 塗りつぶしのパターン

パターン名	パターン
left slant light （左斜め・細）	
right slant light （右斜め・細）	

表12.3 塗りつぶしのパターン（**続き**）

パターン名	パターン
vertical light （縦・細）	
horizontal light （横・細）	
grid light （格子・細）	
hatch light （斜め格子・細）	
h wave light （横波・細）	
v wave light （縦波・細）	
hollow circle （中抜き円）	
left slant medium （左斜め・中）	
right slant medium （右斜め・中）	
vertical medium （縦・中）	
horizontal medium （横・中）	
grid medium （格子・中）	
hatch medium （斜め格子・中）	
h wave medium （横波・中）	
v wave medium （縦波・中）	
filled circle （塗りつぶし円）	

表12.3 塗りつぶしのパターン（続き）

パターン名	パターン
left slant heavy （左斜め・太）	
right slant heavy （右斜め・太）	
vertical heavy （縦・太）	
horizontal heavy （横・太）	
grid heavy （格子・太）	
hatch heavy （斜め格子・太）	
h wave heavy （横波・太）	
v wave heavy （縦波・太）	
diamond （ひし形）	
left slant heavy b （左斜め・太B）	
right slant heavy b （右斜め・太B）	
random （ランダム）	
square （正方形）	
square offset （正方形交互）	
wide （横長方形）	
tall （縦長方形）	

表12.3 塗りつぶしのパターン（続き）

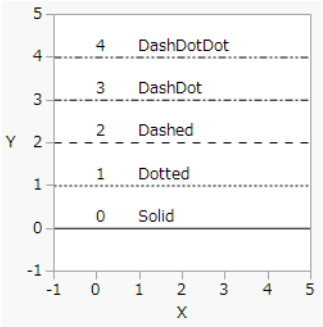
パターン名	パターン
checkerboard （市松模様）	
grid dots （グリッドドット）	
triangle up （上向き三角形）	
triangle down （下向き三角形）	
triangle left （左向き三角形）	
triangle right （右向き三角形）	
weave light （平織り・細）	
weave heavy （平織り・太）	
honeycomb （ハニカム）	

線の種類を指定する

Line Style()（線種）は、番号（0～4）または名前（Solid、Dotted、Dashed、DashDot、DashDotDot）で制御できます。図12.26は、各種類に対応する数値を示しています。

```
linestyles = {"Solid", "Dotted", "Dashed", "DashDot", "DashDotDot"};
win = New Window( "線種",
   Graph Box(
      Frame Size( 200, 200 ),
      X Scale( -1, 5 ),
      Y Scale( -1, 5 ),
      For( i = 0, i < 5, i++,
         Line Style( i );
         H Line( i );
         Text( {0, i + .1}, i );
         Text( {1, i + .1}, linestyles[i + 1] );
      )
   )
);
```

図12.26 線の種類



線の太さを制御するには、Pen Sizeを設定し、線の幅をピクセルで指定します。デフォルトは1で、1ピクセルの線になります。印刷の場合は、Pen Sizeにデフォルトの線の幅を掛けた幅になります。デフォルトの線の幅はプリンタによって異なります。

```
win = New Window( "Pen Size",
   Graph Box(
      Pen Size( 2 ); // 2倍の線幅
      Line( [10 30 90], [88 22 44] );
   )
);
```

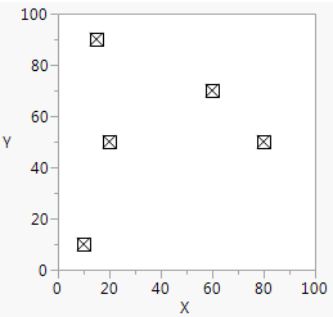
ピクセルを使って描画する

ピクセル座標を使って描画することもできます。まず、グラフ座標における基準点をPixel Origin()で設定し、次に、基準点からの相対的なピクセル座標を、Pixel Move To()やPixel Line To()で設定します。Pixel()関数は、主に、グラフのサイズまたはスケールに左右されないカスタムのマーカーを描くのに使います。マーカーをスクリプトで予め定義しておいて、そのマーカーを任意のグラフから呼び出すこともできます。以下の例では、Function()を使って、引数*x*、*y*をもつPixelコマンドを予め定義しています。

```
ballotBox = Function( {x, y},
   Pixel Origin( x, y );
   Pixel Move To( -5, -5 );
   Pixel Line To( -5, 5 );
   Pixel Line To( 5, -5 );
   Pixel Line To( -5, -5 );
   Pixel Line To( 5, 5 );
   Pixel Line To( -5, 5 );
   Pixel Move To( 5, 5 );
   Pixel Line To( 5, -5 );
);
win = New Window( "カスタムマーカー",
   Graph Box(
      Frame Size( 200, 200 ),
```

```
        ballotBox( 10, 10 );
        ballotBox( 15, 90 );
        ballotBox( 20, 50 );
        ballotBox( 80, 50 );
        ballotBox( 60, 70 );
    )
);
```

図12.27 カスタムマーカーを描く



インタラクティブなグラフ

Handle() と Mousetrap() は、クリックとドラッグに応答するインタラクティブなグラフを作成するための関数です。Handle() は、マウスによるドラッグで移動させることができるハンドルのマーカーを追加します。ハンドルが移動されるたびに、その位置の座標を取得し、設定されたスクリプトを実行します。Mousetrap() は同様の処理を行いますが、ドラッグ可能なハンドルを使用しないで、クリックした位置の座標を取得します。重要な違いは、Handle() は、マウスボタンがハンドル上で押された場合だけ反応しますが、Mousetrap() は、マウスボタンがどの位置で押されても反応する点です。

それ以外に、Button Box()、Slider Box()、Global Box() などを使って、グラフの外にボタンやスライダなどのコントロールを配置する方法もあります。

Handle()

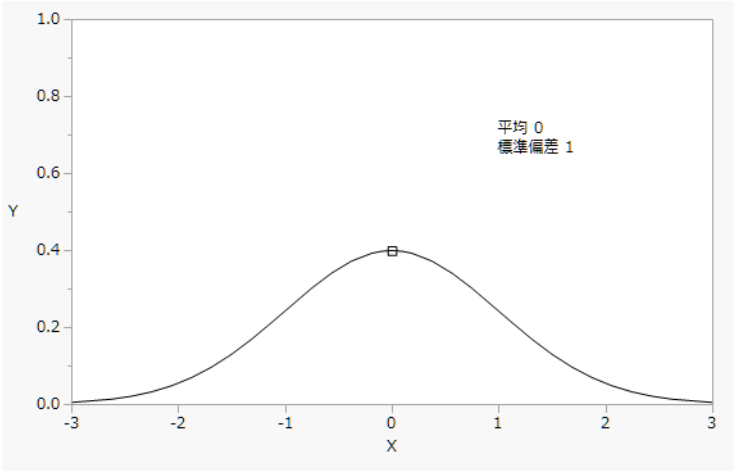
Handle() は、最初の2つの引数の初期値で与えられる座標にハンドルを配置し、引数の初期値を使ってグラフを描きます。その後、そのハンドルは別の位置まで移動できます。初めに指定されたスクリプト（Handle 関数の第3引数）は、マウスボタンが押されてハンドルがドラッグされている間、実行されます。2つ目に指定されたスクリプト（Handle 関数の第4引数。オプション指定であり、この例では使用していません）は、マウスボタンが放されるたびに実行されます。これらのイベントについては、「Mousetrap()」（562ページ）の例を参照してください。

```
// 正規密度
mu = 0;
```

```
sigma = 1;
rsqrt2pi = 1 / Sqrt( 2 * Pi() );
win = New Window( "正規密度",
   Graph Box(
      Frame Size( 500, 300 ),
      X Scale( -3, 3 ),
      Y Scale( 0, 1 ),
      Y Function( Normal Density( (x - mu) / sigma ) / sigma, x );
      Handle(
         mu,
         rsqrt2pi / sigma,
         mu = x;
         sigma = rsqrt2pi / y;
      );
      Text( {1, .7}, "平均 ", mu, {1, .65}, "標準偏差 ", sigma );
   )
);
```

JMPの「Samples」フォルダの「Scripts」フォルダにある「demoPlotProb.jsl」を実行すると、ベータ分布、ガンマ分布、Weibull分布、および対数正規分布の密度関数のグラフが表示されます。正規分布の出力を図12.28に示します。動いているところを確認するため、ぜひご自分でスクリプトを実行してみてください。

図12.28 Handle()の正規密度の例



エラーを回避するために、必ず、ハンドルの座標の初期値をこの例の1行目のように設定してください。

正規密度の例にあるように、ハンドルの座標に関数を使う場合は、Handle()への引数を調整する必要があります。調整しないと、ハンドルマーカーがグラフの外へ出てしまう場合があります。

```
Y Function( a * x ^ b );
Handle( a, b, a = 2 * x, b = y );
```

マーカーを最初の位置から座標（3,4）の位置までドラッグしたとします。すると、引数 a に 6、b に 4 が設定され、グラフは $y = 6x^4$ で再描画され、ハンドルは（6,4）の位置、マウスから離れたところに描かれることになってしまいます。これを修正するには、ハンドルへの最初の引数を、たとえば次のように調整します。

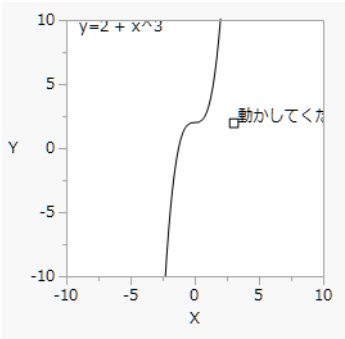
```
Handle( a / 2, b, a = 2 * x; b=y );
```

より一般的な状況を考えるために、Handle() の引数が a=f(x)、b=g(y) と定義されているとします。f(x)=x および g(y)=y の場合は、最初の 2 つの引数に a、b を指定するだけです。それ以外の場合は、$a = f(x)$ を x について、$b = g(y)$ を y について解いて、適切な引数を求める必要があります。

他の関数を使って、Handle() を制限することもできます。以下の例で作成するインタラクティブなグラフは、Round() を使って、指数と切片が整数だけになるようにしています。

```
a = 3;
b = 2;
win = New Window( "切片とべき乗",
   Graph Box(
      Frame Size( 200, 200 ),
      X Scale( -10, 10 ),
      Y Scale( -10, 10 ),
      Y Function( Round( b ) + x ^ (Round( a )), x );
      Handle(
         a,
         b,
         a = x;
         b = y;
      );
      Text( {a, b}, " 動かしてください " );
      Text( {-9, 9}, "y=", Round( b ), " + x^", Round( a ) );
   )
);
```

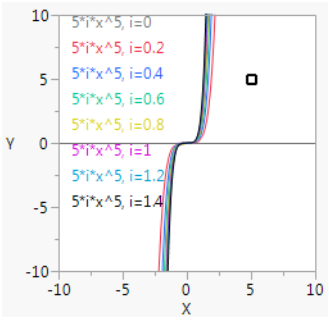
図12.29 Handle() の切片とべき乗の例



Handle() と For() をネストして、複雑なグラフを作成することもできます。

```
a = 5;
b = 5;
win = New Window( "べき乗",
   Graph Box(
      Frame Size( 200, 200 ),
      X Scale( -10, 10 ),
      Y Scale( -10, 10 ),
      For( i = 0, i < 1.5, i += .2,
         Pen Color( 1 + 10 * i );
         Text Color( 1 + 10 * i );
         Y Function( i * x ^ Round( a ), x );
         Handle(
            a,
            b,
            a = x;
            b = y;
         );
         h = 9 - 10 * i;
         Text( {-9, h}, b, "*i*x^", Round( a ), ", i=", i );
      ) ) );
```

図12.30 入れ子になったFor()とHandle()

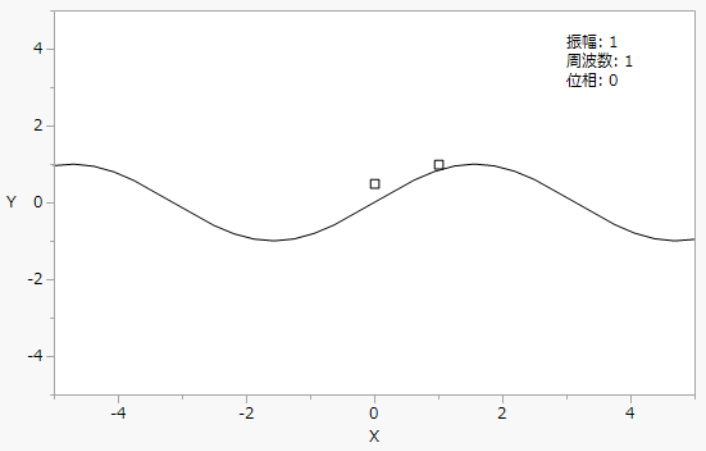


また、1つのグラフ内で複数のハンドルを使用できます。

```
amplitude = 1;
freq = 1;
phase = 0;
win = New Window( "正弦波",
   Graph Box(
      Frame Size( 500, 300 ),
      X Scale( -5, 5 ),
      Y Scale( -5, 5 ),
      Y Function( amplitude * Sine( x / freq + phase ), x );
      Handle( // 最初のハンドル
         freq,
```

```
            amplitude,
            freq = x;
            amplitude = y;
        );
        Handle( phase, .5, phase = x ); // 2つ目のハンドル
        Text(
            {3, 4},
            "amplitude: ",
            Round( amplitude, 4 ),
            {3, 3.5},
            "frequency: ",
            Round( freq, 4 ),
            {3, 3},
            "phase: ",
            Round( phase, 4 )
        );
    )
);
```

図12.31 2つのハンドル



Mousetrap()

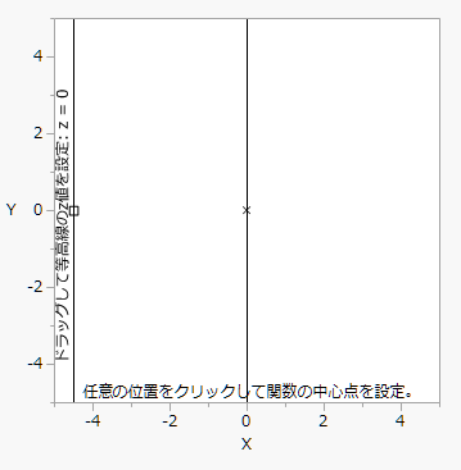
Mousetrap()は、マウスをクリックしたときの座標から、グラフ用の引数を取得します。最初のスクリプトは、マウスボタンを押すたびに実行され、2番目のスクリプトはマウスボタンを放すたびに実行され、ハンドルの新しい座標に従って、グラフを動的に更新します。Handleと同様に、Mousetrapの座標に初期値を設定することが重要になります。グラフにMousetrap()とHandle()の両方を含める場合は、Handle()をMousetrap()の前に配置します。そうすれば、Handle()の方がMousetrap()より先にクリックを捉えることができます。

次の例では、Mousetrap()とHandle()の両方を使って、Mousetrap()の座標によって変化する3次元の関数を定めます。Handle()の値に基づいて、等高線を1本描きます。

```
x0 = 0;
y0 = 0;
z0 = 0;
win = New Window( "3 次元関数の等高線 ",
   Graph Box(
      Frame Size( 300, 300 ),
      X Scale( -5, 5 ),
      Y Scale( -5, 5 ),
      Contour Function(
         Exp( -(x - x0) ^ 2 ) * Exp( -(y - y0) ^ 2 ) * (x - x0),
         x,
         y,
         z0 / 10
      );
      Handle( -4.5, z0, z0 = Round( y * 10 ) / 10 );
      // z の丸め値をハンドルから取得
      V Line( -4.5 );
      Text Size( 9 );
      Text(
         Counterclockwise,
         {-4.6, -4},
         " ドラッグして等高線の z 値を設定 : z = " || Char( z0 / 10 )
      );
      Marker Size( 2 );
      Marker( 2, {x0, y0} );
      Mousetrap( // 基準点をクリックポイントに設定
         x0 = x;
         y0 = y;
      );
      Text(
         {-4.25, -4.9},
         " 任意の位置をクリックして関数の中心点を設定。"
      );
   )
);
```

メモ:Mousetrap()の式でx0とxに、y0をyに設定する代わりに、Function({xx, yy}, Show(xx, yy); x0 = xx; y0 = yy);を使うことができます。xxとyyは、この関数の正式なパラメータで、ユーザがグラフをクリックしたときにMousetrapによって渡される引数に含まれています。Function({xx, yy}...)ではxxとyy名前を明示的に指定しており、xとyに限らず任意の名前を付けられることがわかります。

図12.32 Mousetrap() と Handle()



グラフ上で点を視覚的に補間したりするために、Mousetrap() を使ってデータテーブルに点を収集することもできます。次に、(二変量の関係の散布図のような) プロット上でマウスをクリックし、その点をデータテーブルに追加するスクリプトの例を示します。スクリプトを実行してグラフ内をクリックすると、クリックした点がデータテーブルに記録されます。

```
dt = New Table( "データ 1" );
New Column( "xx", Numeric );
New Column( "yy", Numeric );
x = 0;
y = 0;
Add Point = Expr(
   dt << Add Rows( 1 );
   Row() = N Row();
   :xx = x;
   :yy = y;
);
win = New Window( "点の追加",
   Graph Box(
      Frame Size( 500, 300 ),
      X Scale( -5, 5 ),
      Y Scale( -5, 5 ),
      For Each Row( Marker( {xx, yy} ) );
      Mousetrap( {}, Add Point );
   )
);
```

この例では、MouseTrapの最初のスクリプトの引数が空であることに注意してください。マウスボタンを押しても、何も起こりません。第2引数のスクリプトadd pointは、マウスボタンを放したときに実行され、データ点が追加されます。クリックしてドラッグし、マウスボタンを放した場合、データセットに追加される点は、マウスボタンを押した位置の点ではなく、マウスを移動した後の位置の点です。

Drag関数

Handle() およびMousetrap() と同様の機能を実行する5つのDrag() 関数があります。ただし、これらの関数は同時に複数の点を処理します。最初の2つの引数でリストされた行列内の*n*個の座標に関して、次のように動作します。

- Drag Marker()は、n個のマーカーを描く。
- Drag Line()は、n個の点を（$n-1$）個の線分でつなぐ。
- Drag Rect()は、最初の2つの座標だけを使って他の座標を無視して塗りつぶした長方形を描く。
- Drag Polygon()は、n個の頂点を持つ塗りつぶした多角形を描く。
- Drag Text()は、座標にテキスト項目を描く。テキスト項目のリストがある場合は、リストのi番目の項目をi番目のx、y座標に描きます。リストの項目が座標のペアより少ない場合は、最後の項目が残りの点に繰り返し使われます。

これらのコマンドの構文は以下のとおりです。

```
Drag Marker( xMatrix, yMatrix, dragScript, mouseupScript );
Drag Line( xMatrix, yMatrix, dragScript, mouseupScript );
Drag Rect( xMatrix, yMatrix, dragScript, mouseupScript );
Drag Polygon( xMatrix, yMatrix, dragScript, mouseupScript );
Drag Text( xMatrix, yMatrix, "text", dragScript, mouseupScript );
```

これらすべてには、座標のための左辺値（L-value）の引数が必要です。つまり、行列を値としてもつ変数を引数にとる必要があります。これらの値は、頂点をクリックして新しい位置にドラッグすると変更されます。スクリプトの引数はオプションで、Handle()の場合と同様に動作しますが、Handle()のようにxやyに座標が含まれることはありません。

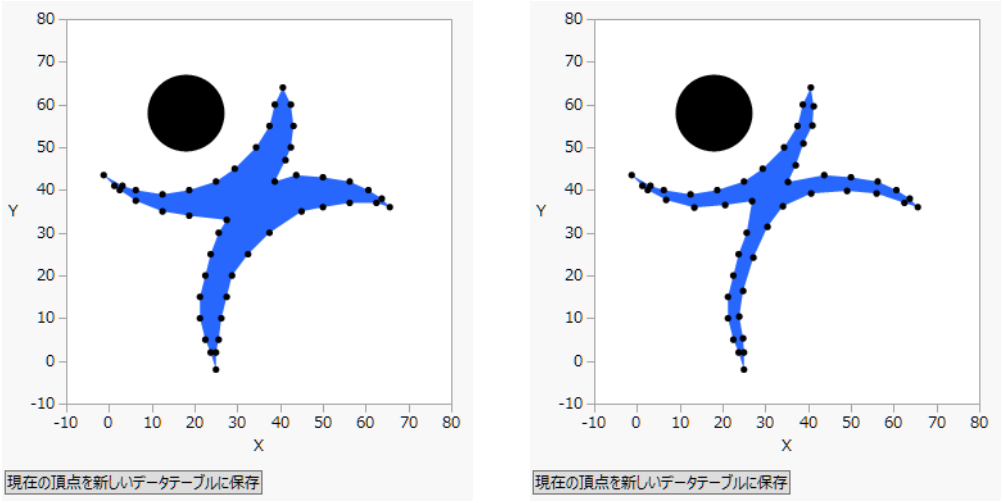
Drag()演算子は、ユーザが調整でき、その後で調整された値を取得し、そのデータを表示するために使います。前記の、JMPマンを描くスクリプトを思い出してください。Drag Polygon()を使うと、編集可能なJMPマンを描くことができます。その時、頂点に対してDrag Marker()ステートメントを一緒に使えば、ドラッグできる点が見やすくなります。また、Mousetrap()の例と同様に、新しい座標をデータテーブルに保存できます。「:」演算子と「::」演算子によって、同じ名前をもつ行列とデータテーブルの列を明確に区別している点に注意してください。

下記のスクリプト例においては、*storepoints*をDrag Polygon()またはDrag Marker()の第4引数にする方が簡単ですが、そうすると、ドラッグするたびにデータテーブルが作成されてしまいます。作業が完了した時にだけ、データテーブルを作成したほうが望ましいでしょう。どちらの場合でも、*gCoordX*と*gCoordY*内の値は、ドラッグによって更新されます。

```
::i = 1;
storepoints = Expr(
   mydt = New Table( "My coordinates" || Char( i ) );
   i++;
   New Column( "GCoordX", Numeric );
   New Column( "GCoordY", Numeric );
   mydt << Add Rows( N Row( GcoordX ) );
   :GCoordX << Values( ::GcoordX );
   :GCoordY << Values( ::GcoordY );
);

::GcoordX = [25, 23.75, 22.5, 21.25, 21.25, 22.5, 23.75, 25.625, 27.5, 18.75, 12.5,
  6.25, 2.5, 1.25, -1.25, 3.125, 6.25, 12.5, 18.75, 25, 29.375, 34.375, 37.5,
  38.75, 40.625, 42.5, 43.125, 42.5, 41.25, 38.75, 43.75, 50, 56.25, 60.625,
  63.75, 65.625, 62.5, 56.25, 50, 45, 37.5, 32.5, 28.75, 27.5, 26.25, 25.625, 25];
::GcoordY = [-2, 2, 5, 10, 15, 20, 25, 30, 33, 34, 35, 37.5, 40, 41, 43.5, 41, 40,
  39, 40, 42, 45, 50, 55, 60, 64, 60, 55, 50, 47, 42, 43.5, 43, 42, 40, 38, 36, 37,
  37, 36, 35, 30, 25, 20, 15, 10, 5, 2];
win = New Window( "JMP マンを描き直そう！",
   V List Box(
      Graph Box(
         Frame Size( 300, 300 ),
         X Scale( -10, 80 ),
         Y Scale( -10, 80 ),
         Fill Color( "blue" );
         Drag Polygon( GcoordX, GCoordY );
         Pen Color( "gray" );
         Drag Marker( GcoordX, GCoordY );
         Fill Color( {0, 0, 0} );
         Circle( {18, 58}, 9, "FILL" );
      ),
      Button Box( "現在の頂点を新しいデータテーブルに保存", storepoints )
   )
);
```

JMPマンはもう少しやせた方がいいかもしれません。下の図は、JMPマンの以前の姿といくつかの頂点を慎重にドラッグした後の姿です。この時点でボタンをクリックすると、*storepoints*スクリプトが実行され、JMPマンの新しいスマートな姿の座標が、データテーブルに保存されます。

図12.33 JMPマンを描き直す



この例では、「表示ツリー」章の「新しいウィンドウの作成」（435 ページ）で説明している次の 2 つの関数を使っています。

- `Button Box()` は、グラフの外にボタンコントロールを作成します。
- `V List Box()` は、グラフボックスとボタンボックスを同じグラフウィンドウ内で縦に並べます。

インタラクティブなグラフのトラブルシューティング

インタラクティブなグラフが思いどおりに動作しない場合は、`Handle()` または `Mousetrap()` 座標（や他のグローバル）の初期値が指定されていることを確認します。また、それらの値がグラフに適した値であることも重要です。

背景地図を作成する

背景地図をJSLスクリプトで指定することができます。まずグラフを作成するスクリプトを書き、次にスクリプトで背景地図をオンにします。

背景地図には2つの種類があり、`Images()` と `Boundaries()` で指定できます。それぞれが、使用する地図の名前をパラメータとして取ります。ウィンドウにリストされているいずれかの地図の名前を使用します。

- `Images()` の場合は、Simple Earth（粗い衛星写真）、Detailed Earth（細かい衛星写真）、NASA、Street Map Service、Web Map Service を指定できます。Web Map Service を使用する場合は、さらに WMS の URL と、WMS サーバーによってサポートされている層の2つのパラメータがあります。
- `Boundaries()` の場合は、ユーザが境界を定義できるため、選択肢はさまざまです。標準的なものとして、World（世界）が挙げられます。

次の例は、Simple Earth をイメージ、World を境界として使用します。

```
dt = Open( "$SAMPLE_DATA/Air Traffic.jmp" );
dt << Graph Builder(
   Size( 1101, 603 ),
   Show Control Panel( 0 ),
   Variables( X( :Longitude ), Y( :Latitude ) ),
   Elements( Points( X, Y, Legend( 8 ) ) ),
   SendToReport(
      Dispatch(
         {},
         "Graph Builder",
         Frame Box,
         {Background Map( Images( "Simple Earth" ), Boundaries( "World" ) ),
         Grid Line Order( 3 ), Reference Line Order( 4 )}
      )
   )
);
```

図 12.34 は、地図の一部です。

図12.34 JSLスクリプトの例

スクリプトに変更を加え、WMS サーバーを使用する場合、コマンドは次のようになります。

```
Background Map ( Images ( "Web Map Service",
   "http://sedac.ciesin.columbia.edu/geoserver/wms",
   "gpw-v3:gpw-v3-population-density_2000" ), Boundaries ( "US States" ) )
```

WMS サーバー上でどのレイヤが使用可能かを確認するには、WMS Explorer アドインをインストールします。アドインは、JMP File Exchange（https://community.jmp.com/docs/DOC-6095）からダウンロードしてください。WMS サーバーの中には、信頼性の低いものもあります。サーバーがダウンしているか、インターネット接続ができない場合、WMS 地図は表示されません。

ヒント:

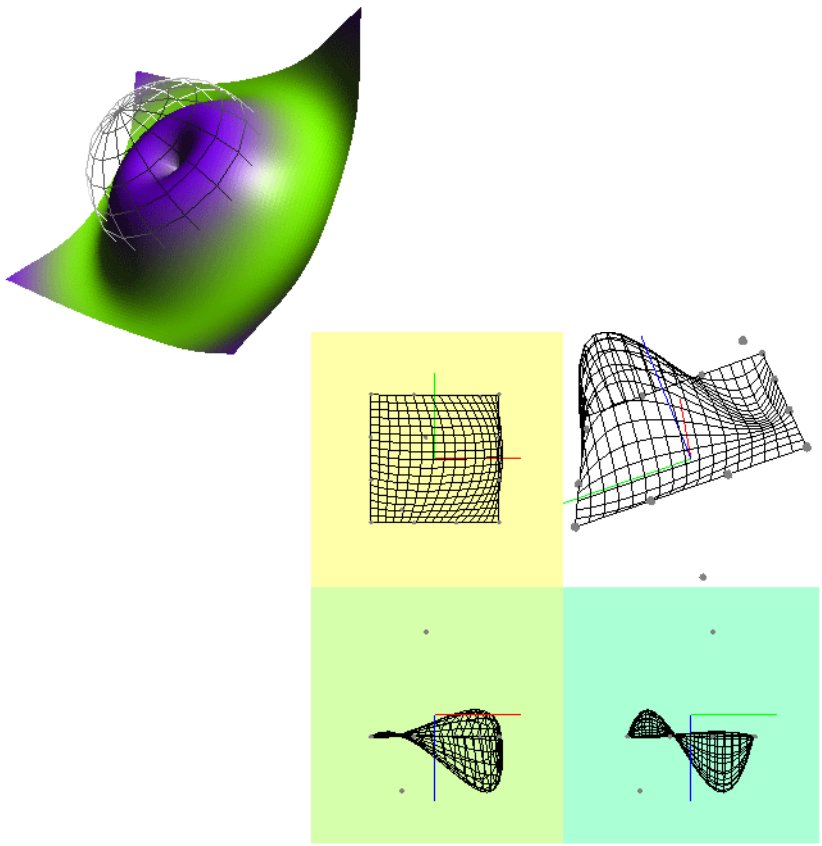
- JSL スクリプトの構文を確認するには、まず、ユーザインターフェースを通じて背景地図を追加します。その後、赤い三角ボタンのメニューから［**スクリプトの保存**］>［**スクリプトウィンドウへ**］を選択し、生成されたスクリプトを確認します。
- JSL スクリプトでイメージと境界の名前を指定するには、「背景地図の設定」ウィンドウに表示された名前を使用します。（地図を右クリックし、［**グラフ**］>［**背景地図**］を選択します。）
- サンプルデータの中には、背景地図のスクリプトが含まれているものがあります。［**ヘルプ**］>［**サンプルデータライブラリ**］を選択し、「Napoleons March.jmp」、「Pollutants Map.jmp」、または「San Francisco Crime.jmp」を開きます。背景地図を使用しているテーブルスクリプトを右クリックし、［**編集**］を選択して `Background Map()` 関数を確認します。

第13章

3Dシーン
3Dシーンのスクリプト

JSLには、OpenGLから派生した3Dシーン（3次元シーン）を作成するためのコマンドがあります。JSLで提供されている3Dシーンは、OpenGL®を完全に実装したものではありませんが、複雑で対話的な3次元グラフを作成できます。なお、JMPの「曲面プロット」プラットフォームのプロットは、JSLのシーンコマンド（scene）を使って作成されています。

図13.1 3D形状の例



JSL 3Dシーンについて

JMPの3Dシーン言語は、OpenGL APIのさまざまな機能を拡張、置換、省略して構築されていますが、Silicon Graphics, Inc.による証明またはライセンス許可を受けて実装されているわけではありません。

この章では、3DシーンJMPのJSLコマンドについて説明します。ただし、OpenGLプログラミングのチュートリアルではありませんので、OpenGLプログラミングに精通していない場合は、補足的な参考書をお読みになることをお勧めします。OpenGLプログラミングに精通している場合でも、この章には特殊な項目が含まれているので、是非お読みください。

JMPの「Sample Scripts」フォルダの「Scene3D」サブフォルダにサンプルファイルが含まれているので、すぐに使用して、使い方をいろいろ工夫できます。スクリプト例の中には、この章で示す例と似ているものもあります。ほとんど完全なアプリケーションと言えるものもあります。

Webサイトのhttp://opengl.orgには、様々な情報が記載されています。

JMPの3Dシーン言語では、OpenGL APIの使用時にユーザが行わなければならないタスクの一部をユーザに代わって実行します。JMPを使うと、テキストの処理が容易になり、組み込みの天体球コントローラを使用できます。モデル表示および投影の行列演算をスタックすることができます。JMPはJMP自身の表示リスト（ディスプレイリスト）によりシーンを保持し、後でシーンを再生できるようにしています。また、ユーザが作成したJSLコードにコールバックして、シーン内でマウスがどのオブジェクトをポイントしているかを知らせるメカニズムを提供します。ユーザは、余分なプログラミングをする手間を減らせます。現時点では、JMPはテクスチャリングなどの幾つかの機能をサポートしていません。

JSL 3Dシーンボックス

次のコマンドは、3Dシーンのセットアップと設定に必要です。

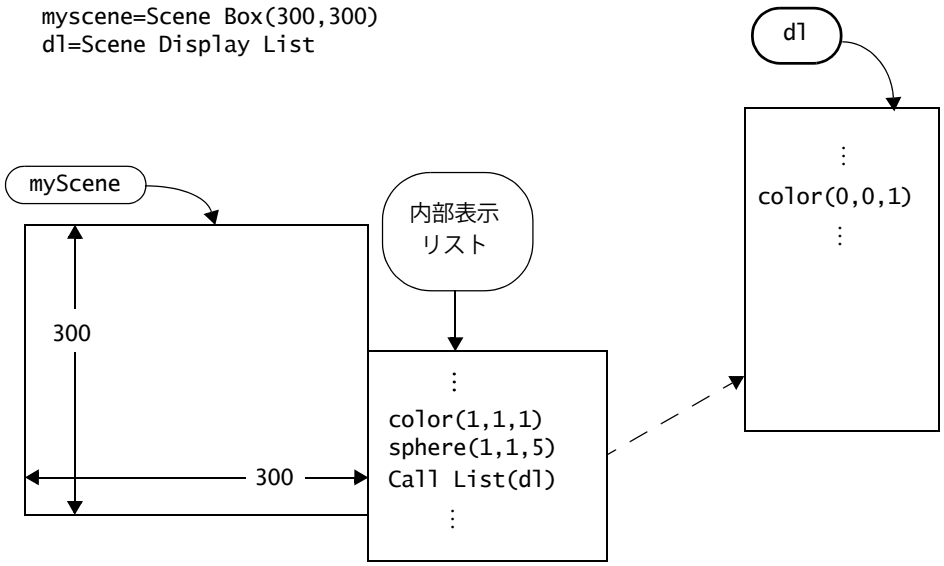
JMPのすべての表示（詳細については、「表示ツリー」章（413ページ）を参照）と同様に、3Dシーンもディスプレイボックス（この場合は、シーンボックス）に配置する必要があります。その後で、そのシーンボックスをウィンドウに配置します。したがって、簡単な3Dシーンのスクリプトは、次のようになります。

```
myScene = Scene Box(300, 300); // 300 x 300 ピクセルのシーンボックスを作成する
...（シーンをセットアップするコマンド）...
New Window("3D シーン ", myScene); // シーンをウィンドウに描画する
...（シーンを操作するコマンド）
```

シーンの要素を作成するメッセージをシーンに送ることができます。代表的なメッセージとして、視点の変更、物理要素の作成、光源およびテクスチャの操作などがあります。これらのメッセージは表示リスト（ディスプレイリスト）内に保持されます。表示リストは、次のどちらかの方法で操作されます。

- メッセージとしてシーンに送られ、ただちにシーンの内部的な表示リストに追加される。
- メッセージとして、グローバル変数に格納された表示リストに送られ、後でシーンの表示リストにより呼び出される。

図13.2 シーンの作成

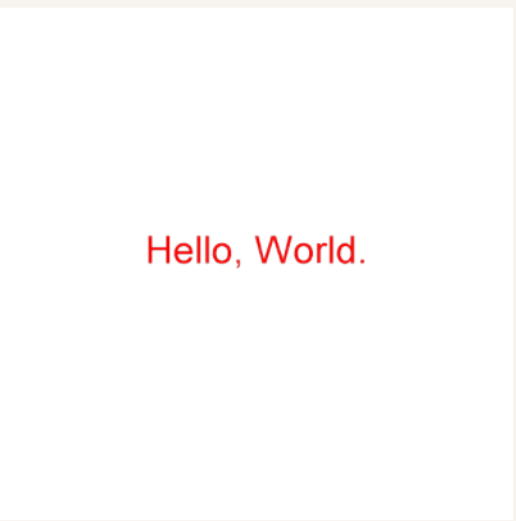


次のスクリプトは、シーンの表示リストに直接コマンドを送る例です。各コマンドについては、この章の後半で詳しく説明します。

```
scene = Scene Box( 400, 400 ); // Scene Box を作成する
New Window( "例 1", scene ); // シーンをウィンドウに配置する
scene << Perspective( 45, 3, 7 ); // カメラを定義する
scene << Translate( 0.0, 0.0, -4.5 ); // (0,0,-4.5) に移動して描画する
scene << Color( 1, 0, 0 ); // テキストの RGB 色を設定する
scene << Text( center, baseline, 0.2, "Hello, World." ); // テキストを追加する
scene << Update; // シーンを更新する
```

最初の2行で、シーンを作成し、ウィンドウに配置します。Perspectiveコマンドは、表示角度と被写界深度を定義します。このコマンドをメッセージとしてシーンに送ると、ただちにシーンの表示リストに追加されます。「Hello World」テキストは原点(0, 0, 0)に描画されるので、Translateコマンドを表示リストに追加し、原点が視野に入るようにカメラを少し後方に移動します。Colorコマンドで色を赤に設定し、テキストを描画し、Updateコマンドでシーンをレンダリングします。これで、コマンドを含む表示リストが描画されます。

図13.3 Hello World



同様に、表示を作成するコマンドを、グローバル変数に格納されている表示リストに累積させた後で、一度にシーンに送ることができます。グローバル変数を表示リストとして定義するには、Scene Display List関数を使って表示リストを割り当てます。たとえば、グローバル変数のgreetingを表示リストとして使うには、次のコマンドを実行します。

```
greeting=Scene Display List();
```

この後で、表示コマンドをメッセージとしてgreetingに送ることができます。次の例では、表示リストを使って「Hello World」を描画します。

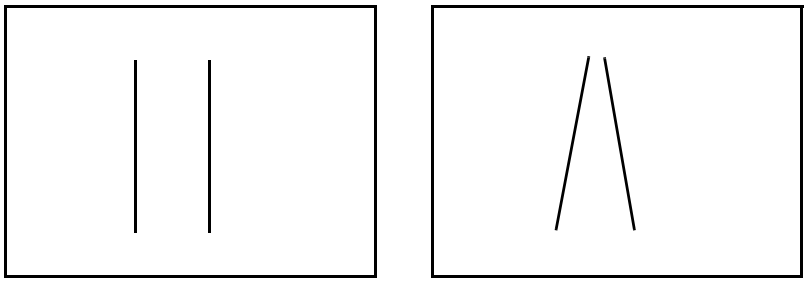
```
greeting = Scene Display List();
greeting << Color( 1, 0, 0 ); // テキストのRGB色を設定する
greeting << Text( center, baseline, 0.2, "Hello, World." ); // テキストを追加する

// ウィンドウを描画して、保存済みの表示リストを送る
scene = Scene Box( 400, 400 ); // Scene Boxを作成する
New Window( "例1", scene ); // シーンをウィンドウに配置する
scene << Perspective( 45, 3, 7 ); // カメラを定義する
scene << Translate( 0.0, 0.0, -4.5 ); // (0,0,-4.5)に移動して描画する
scene << Call List( greeting ); // 表示リストをシーンに送る
scene << Update; // シーンを更新する
```

どのコマンドが別個に表示リストに移されたか、どのコマンドがシーンに直接適用されたかに注意してください。カメラを操作するコマンド（PerspectiveおよびTranslate）は、シーンに適用されています。オブジェクトを定義するコマンド（ColorおよびText）は、表示リストに移されました。こうしておけば、その表示リストを何回でも呼び出して、同じオブジェクトを異なる位置に複製できます。

表示領域の設定

3Dシーンは、2通りの方法でレンダリングできます。**Orthographic**（平行投影,正射影）では、どの視点から見ても配置された要素には遠近感がありません。**Perspective**（透視投影）では、視点の位置と連動して遠近感が出るように表示が調整されます。たとえば、線路のような2本の平行線は、平行投影では平行線のままですが、透視投影では、遠くの1点で交わるように表示されます。

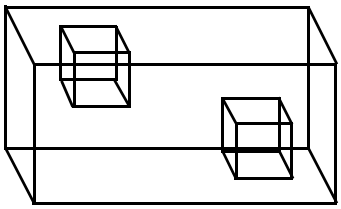
図13.4 平行投影の平行線（左）と透視投影（右）



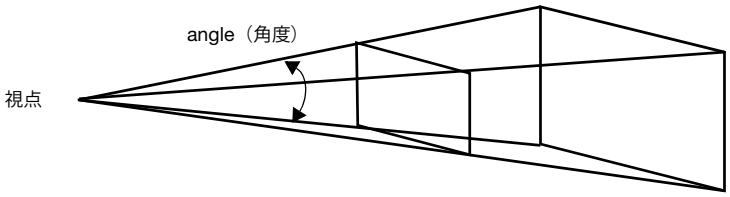
別の例を考えてみましょう。望遠鏡のような筒を真上から見ているとします。平行投影では、筒は厚みのない円として描かれます。透視投影では、円に厚みがあります。つまり、筒の遠端の穴は近端の穴より小さく描かれ、筒の内部が見えます。

したがって、平行投影で表示される領域は直方体であり、透視投影で表示される領域は四角錐台（四角錘の頭部を平面で切り取ったもの）です。

図13.5 投影の比較



平行投影シーン



透視投影シーン

一般に、透視投影は、目やカメラで見る方法を模倣しているので、より現実に近い表示が行われます。平行投影は、建築用のCADプログラムなどのように寸法を保持する必要がある場合に役に立ちます。

透視投影シーンのセットアップ

JSLで透視投影のシーンを構築するには、Perspectiveコマンドを表示に送ります。

```
Perspective (angle, near, far)
```

上の図に示しているように、*angle* は視角、*near* は近い平面までの距離、*far* は遠い平面までの距離です。表示領域を定義するときは、次の2つのことに留意する必要があります。

- 近い（*near*）平面より前方、または遠い（*far*）平面より後方にある要素のように、表示領域の外にある要素は描画されません。それらの要素は表示されません。
- *near* に対する *far* の比は小さくする必要があります。小さい方が、レンダリングエンジンが各要素をシミュレートする時、他の要素の「手前」に描くべき要素を効率よく判定するからです。*near* 引数の値はゼロより大きくなければなりません。

「Hello World」を描くスクリプトには、次の行が含まれています。

```
scene << Perspective( 45, 3, 7 ); // カメラを定義する
```

これは、視角が45度、近い平面までの距離は3単位、遠い平面までの距離は7単位であることを定義しています。

視角は、カメラの広角レンズまたは望遠レンズと同じ働きをします。視角が小さければ図が拡大され、大きければ縮小されます。つまり、視角が小さければ、小さい大きさのシーンに3次元空間が写されるので、シーンのなかの要素が大きく見えます。視角が大きければ、大きな大きさのシーンに3次元空間が写されるので、シーンのなかの要素が小さく見えます。したがって、シーンにおける要素の大きさは、Perspective関数の *angle* 引数を使って操作できます。次の図は、角度を45度と90度に設定した透視投影による「hello world」スクリプトの例です。

図13.6 投影法の変更

Perspectiveコマンドの代わりに、Frustumコマンドで視野の四角錐台を定義することもできます。

```
Frustum(left, right, bottom, top, near, far);
```

四角錐台における近い方の平面の左下隅と右上隅の(*x, y, z*)座標が、(*left, bottom, near*)と(*right, top, near*)で定義されます。*near*には近いクリップ平面までの距離を、*far*には遠いクリップ平面までの距離を指定します。

平行投影シーンのセットアップ

平行投影のシーンは、透視投影のシーンと同様な方法で指定します。次のコマンドを実行します。

```
Ortho(left, right, bottom, top, near, far)
```

このコマンドは、近い平面の4隅の座標、近い平面までの距離、遠い平面までの距離を指定します。

簡単な2次元環境を処理している場合は、このコマンドで2次元の平行射影シーンをセットアップできます。

```
Ortho2D (left, right, bottom, top)
```

このコマンドは、2次元ビューの4隅を指定します。

ビューの変更

3Dシーンを作成する利点の1つは、対象をさまざまな角度と位置から簡単に見ることができるということです。TranslateとRotateコマンドを使って、シーンを見る位置を設定できます。

また、ArcBallコマンドを使うと、視角をインタラクティブに変更できます。

Translateコマンド

前述のサンプルスクリプトで、Translateコマンドを実際に確認しました。このコマンドは、シーンの表示基点を設定します。引数は、現在位置からの、x、y、およびz軸方向への移動量を指定します。

```
Translate (x, y, z)
```

例:

```
Translate( 0.0, 0.0, -2 );
```

これは、原点をz軸の負の方向に2単位移動します。

始めはカメラは原点にありましたが、ここでカメラをz軸の負の方向に下げたので、カメラが原点を見ることができるようになりました。

Rotateコマンド

Rotateコマンドは、シーンの視る角度を修正する場合に使います。次の形式で指定します。

```
Rotate (degrees, xAxis, yAxis, zAxis)
```

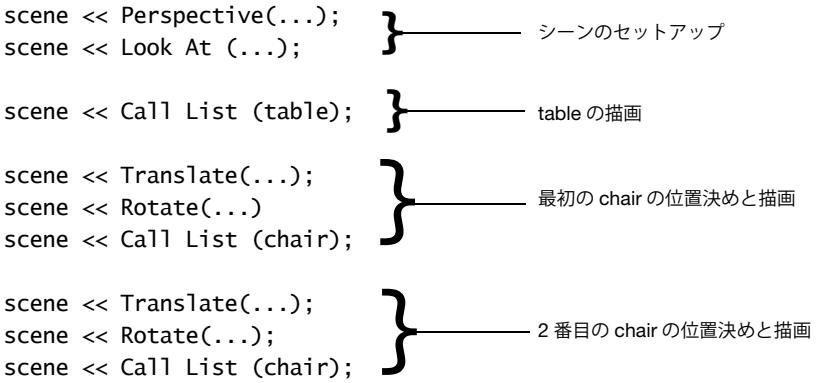
これは、モデルを、ベクトル（*xAxis, yAxis, zAxis*）で指定した軸を中心にしてdegreesで指定した角度だけ回転します。たとえば、モデルをx軸を中心にして90度回転するには、`Rotate( 90, 1, 0, 0 )`を使います。

3本の軸の値を行列で指定することもできます（たとえば、`Rotate( 90, [1, 0, 0] )`）。

メモ: JMPの三角関数における角度はラジアン単位ですが、Rotateコマンドでは度単位です。

TranslateとRotateは、オブジェクトを相対的に位置付ける場合にも使われます。一番初めのTranslateやRotateは、後から指定するすべてのオブジェクトのカメラに対する位置を決めるものと考えることができます。後続のTranslateコマンドとRotateコマンドを使って、球、円柱、円盤などのオブジェクトの位置を決め、Call Listコマンドや ArcBallコマンドでリストを表示します。たとえば、「table」というリストと「chair」という表示リストがあるとします。シーンは、次のように指定できます。

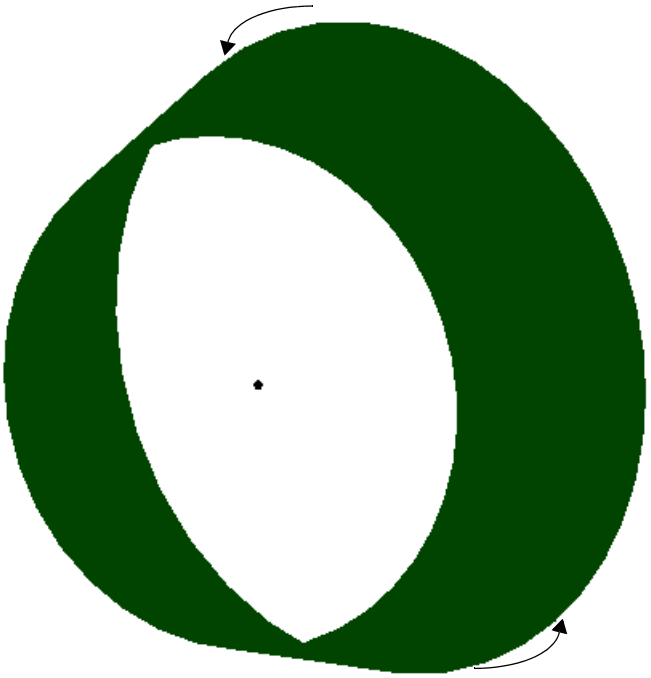
図13.7 TranslateとRotateの使用



次の例では、Rotateコマンドを Forループ内で使って、シーンのカメラアングルを次々に変更します。このスクリプトは、中心点の周りを回転する円柱を描きます。中心点は小さい球で示されます。

```
scene = Scene Box( 600, 600 ); // OpenGL シーンを保持する Scene Box を作成する

New Window( "例 1", scene ); // シーンをウィンドウに配置する

For( i = 1, i < 360, i++,
   scene << Clear;

   scene << Perspective( 45, 1, 10 );
// レンズは 45 度、近い距離（near）はカメラから 1 単位、遠い距離（far）は 10 単位
   scene << Translate( 0.0, 0.0, -2 );

   scene << Rotate( i, 1, 0, 0 );
   scene << Rotate( i * 3, 0, 1, 0 );
   scene << Rotate( i * 3 / 2, 0, 0, 1 );
   scene << Color( 0, 1, 0 ); // 円柱の色は緑
   scene << Cylinder( 0.5, 0.5, 0.5, 40, 10 );
   scene << Color( 0, 0, 0 ); // 球の色は黒
   scene << Sphere( 0.01, 10, 5 );
   scene << Update;
   Wait( 0.01 ); );
```

図13.8 シリンダの回転



シーンへのメッセージの最後でUpdateコマンドを使うことに注意してください。このコマンドは、表示される画面と表示リストの現在の状態を一致させるようにJMPに指示します。前の角度と現在の角度がリスト内に混在しないように、先頭でリストをクリアしてから、変更の後でシーンを更新するよう指定します。

Look Atコマンド

Look Atコマンドは、カメラの視点を設定する別の方法です。

```
Look At( eyeX, eyeY, eyeZ, centerX, centerY, centerZ, upX, upY, upZ )
```

Look Atコマンドは、カメラを視点（*eye*）座標に配置し、中心（*center*）座標に向けます。*up*ベクトルは、視線上でのカメラの回転方法を指定します。通常、モデルは原点に作成されるので、JMPシーンの始めの方でLook Atコマンドまたは Translateコマンドを使ってカメラを原点から離す必要があります。

まず、前のフレームのコマンドのシーンボックスをクリアします。

```
scene << Clear;
```

次に、以下の投影法のいずれかを選択します。

```
scene << Perspective( 45, 2, 10 );
scene << Frustum( -.5, .5, -.5, .5, 1, 10 );
scene << Ortho( -2, 2, -2, 2, 1, 10 );
scene << Ortho2d( -2, 2, -2, 2 );
```

メモ: 投影法をortho2dとした場合は、TranslateまたはLook Atによるカメラ位置の設定は行わないでください。

最後に、TranslateまたはLook Atのいずれかを使用してカメラ位置の設定を行います。

```
scene << Translate(0.0, 0.0, -4.5);
/* カメラをZ軸の負の方向に移動し、
   原点(0,0,0)が視野に入るように移動する */
scene << Look At( /* 視点 */ 3,3,3, /* 中心点 */ 0,0,0, /* アップ */ 1,0,0 );8
/* この指定方法がはるかに簡単 */
```

このような方法でシーンとカメラ位置を設定した後に、モデルを追加していってください。

天体球（アークボール）

マウスの動きに従ってシーンを回転させたい場合があります。JMPの「曲面プロット」プラットフォームは、マウスの動きに従って回転する3Dシーンの例です。

ArcBall（天体球; アークボール）は、3Dシーンの周りに球を設定します。ユーザは、その球の表面をクリックしてドラッグすることにより、シーンを回転させることができます。

Call Listコマンドの代わりにArcBallを使って、シーンを天体球（アークボール）に配置します。天体球に張り付けられたシーンは、自動的にマウスのクリック＆ドラッグに対応するようになります。新しいプログラムを作成する必要はありません。ただし、天体球での回転は保存されません（技術的に説明すると、ArcBallは暗黙のPush MatrixとPop Matrixブロックの中にあるので、戻ったときにはその動きは消滅しています。プッシュとポップの詳細については、「行列スタックの使用」（594ページ）を参照してください）。

例として、「基本要素の例」（584ページ）のスクリプトを調べてみましょう。次のような行があります。

```
scene << CallList(shape); // 表示リストをシーンに送る
```

この行を、次のように変更します。

```
scene << ArcBall(shape,2); // 表示リストを天体球に送る
```

このスクリプトは、表示を直径2の天体球に関連付けます。スクリプトを実行してウィンドウが表示されたら、右クリックし、表示されるメニューから［**天体球の表示**］>［**常に**］を選択します。

メモ: 天体球（ArcBall）は、Academic Press発行の『Graphics Gems IV』に掲載された記事（1994年）でShoemakeが使った用語です。

これにより、天体球が常に表示されるように設定されます。シーンを回転するには、天体球をクリック＆ドラッグします。シーンをプラットフォーム、ジャーナル、JSLのどれで表示する場合でも、［背景色］、［ハードウェアアクセラレーションを使用］、および［天体球の表示］の項目があるポップアップメニューが常に使用できます。

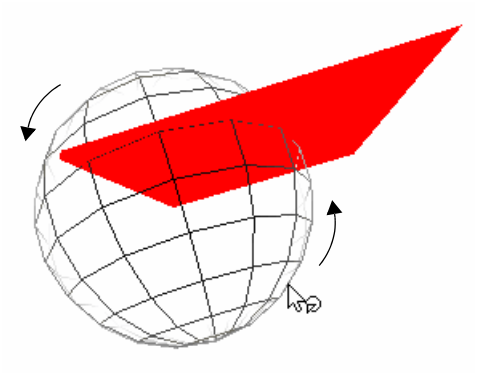
メモ: 天体球を表示しなくても、マウスコマンドは実行されます。ここでは、説明するために表示しているだけです。

Show ArcBallコマンドを使って、JSLで天体球の表示状態を設定することもできます。

```
scene << Show Arcball(state)
```

*state*には、During Drag、Always、またはNeverを指定できます。

図13.9 天体球の表示



グラフィックの基本要素

JSLのすべてのシーンは、少数のグラフィック基本要素から作成されます。これらの基本要素は、複雑なシーンを組み上げるブロックとして機能します。

グラフィックの基本要素はすべて、頂点を指定して実行します。頂点は、単独の点として描かれる場合も、結合されて多角形を構成する場合もあります。基本要素を描くには、基本要素のタイプ、座標、および含まれる頂点のプロパティを指定する必要があります。JSLでは、これらはBeginステートメントとEndステートメントの間に指定します。

```
scene<<Begin(primitive type);
...(頂点とそのプロパティを指定するコマンド)...
scene<<End();
```

頂点の座標を指定するには、vertexコマンドを使います。

```
scene<<Begin(primitive type);
scene<<Vertex(x, y, z);
...
scene<<End();
```

*primitive type*には次のオプションがあります。次の例では、v0、v1などはBeginコマンドとEndコマンドの間に指定されているものとします。

```
scene<<Begin(primitive type);
scene<<Vertex(x0,y0,z0)// 頂点 v0 を指定する
scene<<Vertex(x1,y1,z1)// 頂点 v1 を指定する
...
scene<<Vertex(xn,yn,zn)// 頂点 vn を指定する
scene<<End();
```

表13.1 基本要素の種類

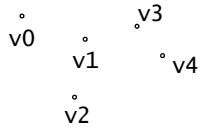
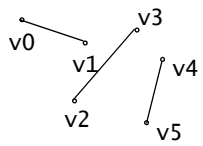
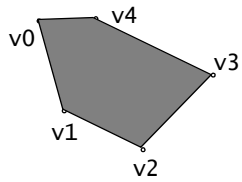
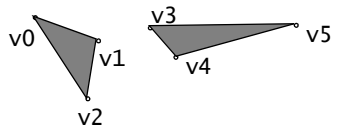
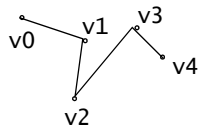
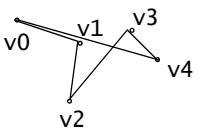
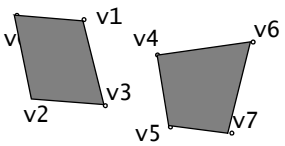
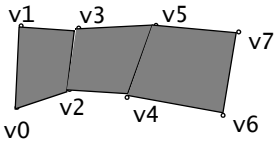
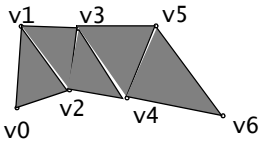
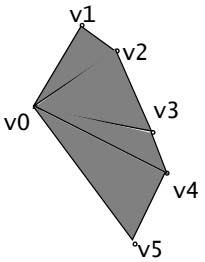
primitive type=POINTS	各頂点に点を描きます。	
primitive type=LINES	複数の線分（連結されていない）を描きます。線分は、v0 と v1、v2 と v3、以下同様に結んで描かれます。n が奇数の場合、最後の頂点は無視されます。	
primitive type=POLYGON	点 v0、...、vn を頂点とする多角形を描きます。3 個以上の頂点がないと、多角形は描かれません。また、指定する多角形は、線が交差しない、凸状の多角形である必要があります。頂点がこれらの条件を満たしていない場合、結果は予期しないものとなります。	
primitive type=TRIANGLES	頂点を v0、v1、v2、次に v3、v4、v5、以下同様に結んで複数の三角形（それぞれ分離している）を描きます。頂点の数が 3 の倍数ではない場合、最後の 1 個または 2 個の頂点は無視されます。	
primitive type=LINE_STRIP	v0 から v1、次に v1 から v2、以下同様に結んで連結する線分を描きます。したがって、n 個の頂点を指定すると n-1 本の線分が描かれます。頂点が 2 個以上ないと何も描かれません。線分を指定する頂点に関する制限はないので、線が交差する場合もあります。	

表13.1 基本要素の種類（続き）

primitive type=LINE_LOOP	LINE_STRIP とほとんど同じですが、最後の線分は最後の頂点から最初の頂点を結んで描かれ、完全に閉じられます。	
primitive type=QUADS	頂点を v0、v1、v2、v3、次に v4、v5、v6、v7、以下同様に結んで複数の四辺形（4 辺の多角形）を描きます。頂点の数が 4 の倍数でない場合、最後の 1 ～ 3 個の頂点は無視されます。	
primitive type=QUAD_STRIP	頂点を v0、v1、v3、v2、次に v2、v3、v5、v4、その次に v4、v5、v7、v6、以下同様に結んで複数の四辺形（4 辺の多角形）を描きます。頂点の数が少なくとも 4 個ないと何も描かれません。頂点の数が奇数の場合、最後の頂点は無視されます。	
primitive type=TRIANGLE_STRIP	頂点を v0、v1、v2、次に v2、v1、v3（順序に注意）、その次に v2、v3、v4、以下同様に結んで複数の三角形（3 辺の多角形）を描きます。三角形が同じ方向に向かってすべて描かれ、正しい面が形成されるためには、頂点の順序を適切に指定する必要があります。頂点が少なくとも 3 個ないと何も描かれません。	
primitive type=TRIANGLE_FAN	TRIANGLE_STRIP とほとんど同じですが、頂点を v0、v1、v2、次に v0、v2、v3、その次に v0、v3、v4、以下同様に結びます。	

基本要素の例

次の例で、グラフィックの基本要素の使い方を説明します。

```
shape = Scene Display List(); // 表示リストを作成してコマンドを送る
shape << Color( 1, 0, 0 ); // テキストの RGB 色を設定する
```

```
shape << Begin( POLYGON );
shape << Vertex( 0, 0, 0 );
shape << Vertex( 0, 3, 0 );
shape << Vertex( 3, 3, 0 );
shape << Vertex( 5, 2, 0 );
shape << Vertex( 4, 0, 0 );
shape << Vertex( 2, -1, 0 );
shape << End();

// ウィンドウを描画して、保存済みの表示リストを送る
scene = Scene Box( 400, 400 ); // Scene Box を作成する
New Window( " 基本要素 ", scene ); // シーンをウィンドウに配置する
scene << Perspective( 90, 3, 7 ); // カメラを定義する
scene << Translate( 0.0, 0.0, -5 ); // (0,0,-5) に移動して描画する
scene << Call List( shape ); // 表示リストをシーンに送る
scene << Update; // シーンを更新する
```

このスクリプトの最初のセクションで「shape」という名前の表示リストを作成します。そして、この表示リスト内で、6個の頂点を使って多角形を定義します。

スクリプトの2番目の部分では、シーンボックスと新しいウィンドウを作成しています。そして、Call List メッセージを使って表示リストをシーンに送っています。

z座標がすべてゼロであること、つまり多角形が1つの平面上にあることを確認してください。多角形が同一平面上にない場合、予期しない結果になります。

スクリプトには、次のような行があります。

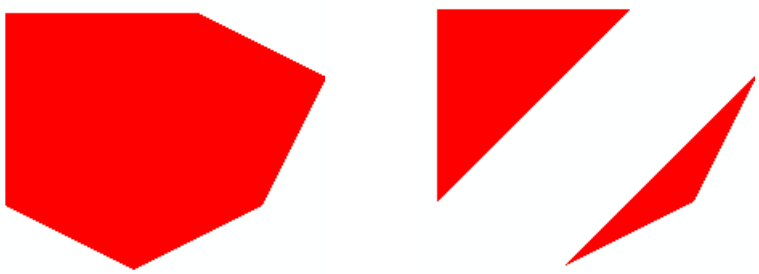
```
shape <<Begin(POLYGON);
```

この行の基本要素を他のタイプに変更してみましょう。たとえば、次のように変更してみましょう。

```
shape <<Begin(TRIANGLES);
```

そうすると、違った図形が描かれます。

図13.10 多角形（左）と三角形（右）



基本要素の外観の制御

JSLには、オブジェクトを描く基本要素の外観を変更するコマンドがあります。線の幅や点描パターン（破線、点線など）を指定することもできます。

サイズと幅

レンダリングされるオブジェクトの点のサイズを設定するには、Point Sizeコマンドを使います。

```
Point Size (n)
```

*n*はピクセル数です。このピクセル数は、アンチエイリアシングやハードウェア構成などの他の設定によっては、レンダリングされる実際のピクセル数と異なる場合があります。

線の幅は、Line Widthコマンドを使って設定します。

```
Line Width(n)
```

*n*はピクセル数です。引数*n*はゼロより大きい必要があり、デフォルトは1です。

点描パターン

点描される線を作成するには、Line Stippleコマンドを使います。

```
Line Stipple(factor, pattern)
```

*Factor*は伸張率です。*Pattern*は、ピクセルをオンまたはオフにする16ビットの整数です。効果をオンにするには、Enable(LINE_STIPPLE)を使います。

点描パターンを作成するには、目的の点描パターンを表す16ビットのバイナリ数を書き込みます。パターンは右から左に読まれるので、表記がレンダリング方向とは逆に見えてしまうことがあります。バイナリ数を整数に変換し、それを*pattern*引数として使います。

たとえば、パターン0000000011111111の点線を描きたいとします。このバイナリ数は10進表記では255なので、Line Stipple(1, 255)と指定します。

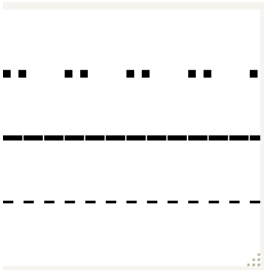
*factor*引数は、各バイナリ桁を2倍に引き伸ばして2桁にします。この例では、Line Stipple(2, 255)は0000000000000000111111111111111になります。

次の例のスクリプトは、それぞれ幅（Line Widthコマンド）と点描パターンが異なる3本の線を描きます。

```
scene = Scene Box( 200, 200 ); // OpenGL シーンを保持する Scene Box を作成する

New Window( "Stipples", scene ); // シーンをウィンドウに配置する
scene << Ortho( -2, 2, -2, 2, -1, 1 );
scene << Color( 0, 0, 0 ); // テキストの RGB 色を設定する

scene << Enable( LINE_STIPPLE );
scene << Line Width( 2 );
scene << Line Stipple( 1, 255 );
```

```
scene << Begin( LINES );
scene << Vertex( -2, -1, 0 );
scene << Vertex( 2, -1, 0 );
scene << End();

scene << Line Width( 4 );
scene << Line Stipple( 1, 32767 );

scene << Begin( LINES );
scene << Vertex( -2, 0, 0 );
scene << Vertex( 2, 0, 0 );
scene << End();

scene << Line Width( 6 );
scene << Line Stipple( 3, 51 );

scene << Begin( LINES );
scene << Vertex( -2, 1, 0 );
scene << Vertex( 2, 1, 0 );
scene << End();
scene << Update;
```

図13.11 点描



メモ: 点描パターンは、回転するモデル上に「這う」ように描かれます。これは、点描パターンがモデルの単位ではなく画面のピクセルで指定され、モデル内の線はモデルの単位に変化がなくても画面上で長さが変化するからです。

塗りつぶしパターン

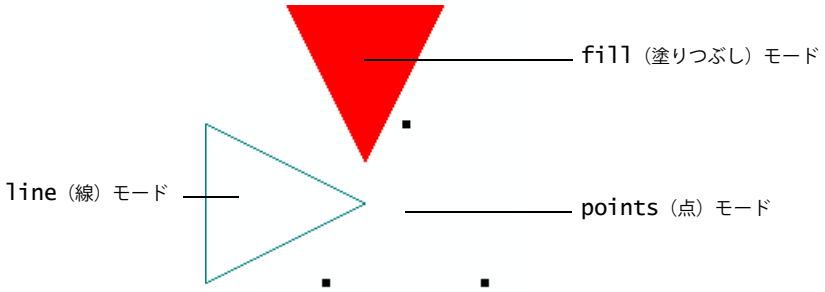
多角形は、前面と背面の両方でレンダリングされ、どちらの側の描画モードもカスタマイズできます。多角形の背面と前面で、異なった描画方法を指定することができます。

多角形の描画モードを設定するには、Polygon Modeコマンドを使います。

```
Polygon Mode (face, mode)
```

*face*には、FRONT、BACK、またはFRONT_AND_BACKを、*mode*には、POINT（点）、LINE（線）、またはFILL（塗りつぶし）を指定できます。

図13.12 点、線、塗りつぶし



次表のスクリプトは、三角形を定義する表示リストを作成します。表示リストにおいて、Translate、Rotate、およびColorと組み合わせて3回使用し、3角形を3つの位置に描きます。また、Polygon Modeコマンドによって、各三角形の描画モードを変更しています。塗りつぶし（FILL）モードはデフォルトなので、明示的に呼び出していないことに注意してください。

次表には、スクリプトの解説も記載しています。TranslateとRotateコマンドによる操作を、どのように表示リストに累積させているかを説明しています。

表13.2 Translateコマンドと Rotateコマンド

上記のスクリプトのコード	説明
`shape = Scene Display List();`	表示リストを作成する。
`shape << Begin(TRIANGLES);` `shape << Vertex(0, 0, 0);` `shape << Vertex(-1, 2, 0);` `shape << Vertex(1, 2, 0);` `shape << End();`	三角形の頂点を保持する「shape」という名前の表示リストを作成する。この例は2次元上に描画するので、頂点のz座標はすべてゼロに設定します。
`scene = Scene Box( 200, 200 );` `New Window( "塗りつぶしモード", scene );` `scene << Ortho2d(-2,2,-2,2);`	シーンをディスプレイボックスに配置し、新しいウィンドウを作成する。

表13.2 Translateコマンドと Rotateコマンド（続き）

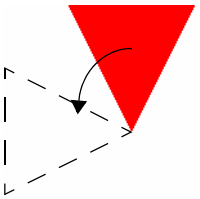
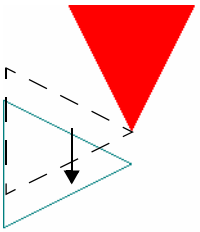
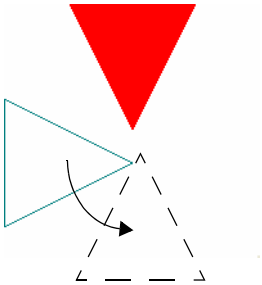
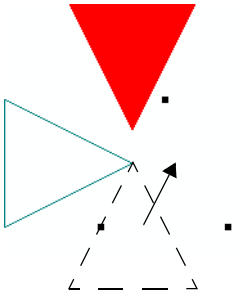
上記のスクリプトのコード	説明
scene << Color(1,0,0); scene << Call List(shape); scene << Update; // シーンを更新して三角形を確認する	最初の三角形を赤で描く。 
scene << Rotate (90, 0, 0, 1); scene << Translate (-0.5, 0, 0); scene << Color(0, 0.5, 0.5); scene << Polygon Mode(FRONT_AND_BACK, LINE); scene << Call List(shape); scene << Update; // シーンを更新して三角形を確認する	2つ目の三角形を緑がかった青で描く。まず三角形を回転します。  それから、移動します。 

表13.2 Translateコマンドと Rotateコマンド（続き）

上記のスクリプトのコード	説明
`scene << Rotate(90, 0, 0, 1);` `scene << Translate(-0.5, -1, 0);` `scene << Color(0, 0, 0);` `scene << Point Size(5); // 見やすいように大きい点を指定する` `scene << Polygon Mode(FRONT_AND_BACK, POINT);` `scene << Call List(shape);` `scene << Update;` `// シーンを更新して三角形を確認する`	3つ目の三角形を黒の点で描く。まず回転します。  それから、移動して、最後の図を描きます。 

塗りつぶした多角形に対して、境界線を描くには、一度、塗りつぶした多角形を描いた後、それに線を重ねて描く必要があります。ただし、レンダリング方法によっては、思ったように線が描画されないことがあります。`Polygon Offset`コマンドを使うと、この「縫い合わせ」問題を解決できます。

```
Polygon Offset (factor, units)
```

オフセットを有効にするには、目的のモードに合わせて、`Enable(POLYGON_OFFSET_FILL)`、`Enable(POLYGON_OFFSET_LINE)`、または`Enable(POLYGON_OFFSET_POINT)`を使います。実際のオフセット値は、*m**(*factor*)+*r**(*units*)で計算されます。*m*は多角形の深度の最大勾配、*r*はウィンドウ座標の深度値で解像可能な差を生成するのに必要な最小値です。必要に応じて、`Polygon Offset(1,1)`から始めてください。

`Polygon Offset`は、「曲面プロット」プラットフォームでも内部的に使われています。曲面とメッシュまたは曲面と等高線を重ねて表示するときにPolygon Offsetを使っています。なぜなら、線を視点の方に近づけるか、曲面を視点から遠ざけるかしないと、曲面によって線の一部が隠されてしまうからです。

BeginおよびEndのその他の用法

通常、begin と end ステートメントの間には頂点を指定しますが、この他にも有効なコマンドがあります。次のコマンドについては、この章の別の節で説明しています。

- Vertexは、頂点をリストに追加します。
- Colorは、現在の色を変更します。
- Normalは、法線ベクトルの座標を設定します。
- Edge Flagは、エッジの描画を制御します。
- Materialは、材質プロパティを設定します。
- Eval CoordとEval Pointは、座標を生成します。
- Call Listは、表示リストを実行します。

球、円柱、円盤の描画

球、円柱、円盤をすばやくレンダリングできるように、いくつかの定義済みコマンドが用意されています。これらのコマンドは、使いやすいだけでなく、特殊な照明プロパティ（法線）が予め組み込まれています。

作図

円柱、円盤、扇形、および球を作図するには、次のコマンドを使います。

円柱

```
Cylinder( baseRadius, topRadius, height, slices, stacks )
```

*baseRadius*は円柱の底面の半径、*topRadius*は円柱の上面の半径です。*height*は円柱の高さです。

*Slices*は円の精度を指定するもので、円状に近くするためには10以上の値を指定します。QuadricNormals(Smooth)を使うと、外観が良くなります。

*Stacks*には、光の反射に使用できる頂点の数を設定します。*Stacks*の値を大きくすると、それだけ「ホットスポット」が正確になります。

円盤

次のコマンドは、中央に*innerRadius*で指定された穴がある非常に薄い円盤を描きます。

```
Disk( innerRadius, outerRadius, slices, loops )
```

Cylinderコマンドと同様に、*slices*は曲線の精度を制御し、*loops*は光の反射の精度を上げるために頂点数を増やします。

```
Partial Disk( innerRadius, outerRadius, slices, loops, startAngle, sweepAngle )
```

Partial Diskコマンドは、Diskコマンドと同様に動作しますが、円盤の一片が取り除かれます。*startAngle*と*sweepAngle*を使って、表示する円盤部分を指定します。

球

次のコマンドは、指定された半径（*radius*）の球を描きます。

```
Sphere( radius, slices, stacks )
```

*slices*は経度、*stacks*は緯度と考えることができます。それぞれ10前後を指定すると、きれいな球が描けます。

ライト

カスタマイズされた曲面の場合とは異なり、球、円盤、および円柱については、法線ベクトルの特別な計算は必要ありません。ただし、次のコマンドを使って、自動的に生成されるライトをカスタマイズできます。

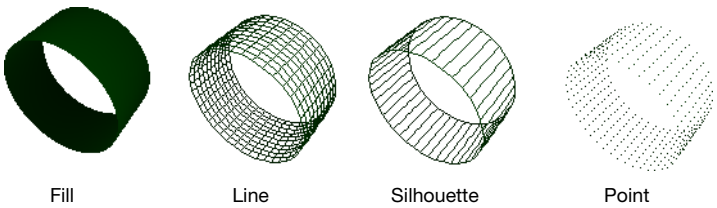
Quadric Normals(*mode*)は、自動的に生成される法線の種類を指定します。*mode*引数には、None、Flat、またはSmoothを指定できます。Flatを指定すると、表面は小平面に分割されます。Smoothを指定すると、各頂点の法線は近接するポリゴンの平均になります。

Quadric Orientation(*mode*)は、法線ベクトルの向きを指定します。*mode*引数には、InsideまたはOutsideを指定できます。

Quadric Draw Style(*mode*)は、描画モードを指定します。*mode*引数には、Fill、Line、Silhouette、またはPointを指定できます。

JMPは、Quadratic Normals、Quadratic Orientation、およびQuadratic Draw Styleに設定された値を、それらの設定後に生成された円柱、円盤、球に適用します。

図13.13 描画の種類



メモ: 他のOpenGLマニュアルでは、いくつかの2次曲面オブジェクトが言及されています。JMPでは1つだけで、常にそれを使用します。

テキストの描画

前述の「Hello World」の例で説明したように、テキストをシーンに追加するには Text コマンドを使います。

```
Text( horz, vert, size, string, <billboard>)
```

- *horz*には、Left（左揃え）、Center（中央揃え）、またはRight（右揃え）を指定できます。
- *vert*には、Top（上揃え）、Middle（中間揃え）、Baseline（基準線揃え）、Bottom（下揃え）を指定できます。
- *size*は、モデル座標での大文字Mの高さを指定します。
- *string*は、描画するテキストです。
- *billboard* はオプションの引数で、テキストを回転させます。このオプションを指定すると、テキストは常にユーザに向かって表示されます。

フォントは、常にJMPのテキストフォントが使われます。［環境設定］メニューからテキストのフォントを変更できます。ただし、JMPはシーン用のフォントをキャッシュするので、JMPを再起動しないと、変更は有効になりません。

メモ: 標準のOpenGL定義にはテキストが含まれません。

TextとRotateおよびTranslateの連動使用

次の例では、Textコマンドを Translate および Rotate コマンドと連動させて使っています。

```
scene = Scene Box( 600, 600 ); // OpenGL シーンを保持する Scene Box を作成する
New Window( "例 2", scene ); // シーンをウィンドウに配置する
scene << Perspective( 45, 3, 7 );
// レンズは 45 度、近い距離（near）はカメラから 3 単位、遠い距離（far）は 7 単位
scene << Translate( 0.0, 0.0, -4.5 );
// カメラで原点 (0,0,0) が見えるように移動する
scene << Rotate( 30, 0, 1, 0 );
//  最初のテキストを Y 軸（画面の縦方向）を中心に回転させる
scene << Color( 1, 0, 0 ); // 赤色
scene << Text( "center", "baseline", .2, " 最初の赤色の文字列 " );
scene << Translate( 0.0, 0.0, -2.0 );
// 次のテキストをカメラからもっと遠ざける
scene << Rotate( 30, 0, 1, 0 );
// 2 番目のテキストを Y 軸（画面の縦方向）を中心に回転させる
scene << Color( 0, 1, 0 ); // 緑色
scene << Text( "center", "baseline", .2, "2 番目の緑色の文字列 " );
scene << Update;
// 現在の表示リストを使って、ウィンドウ内のディスプレイボックスを更新する
```

図13.14 テキスト文字列の回転と移動

緑色の文字列の一部は、遠い方のクリップ平面を超えて後方に位置しています。Perspectiveの7を10に変更すると、文字列全体が表示されます。

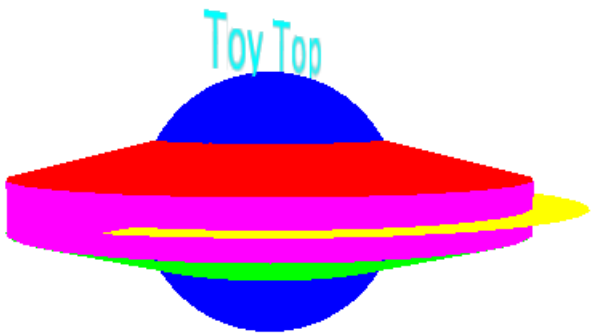
行列スタックの使用

JMPの3Dシーンでは、行列スタックを使って、現在の変換を継続してトラッキングします。スタックは、単位行列に初期化されます。そして、移動、回転、または拡大や縮小のコマンドが指定されるたびに、スタックの最上部の行列が変更されます。

メモ: 多くのOpenGL実装とは異なり、JMPでは転置行列は使いません。

次のプログラム例では、Push MatrixとPop Matrixを使って「toy top」の各部分の位置を決めています。各部分を指定した後には、原点に戻っています。この方法を使うと、Translateコマンドを2回使って元に戻すよりも速く処理できます。

図13.15 行列スタックを使った描画




```
toyTop = Scene Display List();
toyTop << Push Matrix;
toyTop << Translate( 0, 0, .1 );
toyTop << Color( 1, 0, 0 ); // red
toyTop << Cylinder( 1, .2, .2, 25, 5 );
```

```
// baseRadius、topRadius、height、slices、stacksを指定する
toyTop << Pop Matrix;
toyTop << Push Matrix;
toyTop << Translate( 0, 0, -.1 );
toyTop << Rotate( 180, 1, 0, 0 );
toyTop << Color( 0, 1, 0 ); // 緑
toyTop << Cylinder( 1, .2, .2, 25, 5 );
toyTop << Pop Matrix;

toyTop << Color( 0, 0, 1 ); // 青
toyTop << Sphere( .5, 30, 30 ); // radius、slices、stacksを指定する

toyTop << Color( 1, 1, 0 ); // 黄色
toyTop << Partial Disk( 1, 1.2, 25, 2, 0, 270 );
// innerRadius、outerRadius、slices、rings、startAngle、sweepAngleを指定する

toyTop << Push Matrix;
toyTop << Translate( 0, 0, -.1 );
toyTop << Color( 1, 0, 1 ); // マゼンタ
toyTop << Cylinder( 1, 1, .2, 25, 3 );
// baseRadius、topRadius、height、slices、stacksを指定する
toyTop << Pop Matrix;

toyTop << Push Matrix;
toyTop << Rotate( 90, 1, 0, 0 );
toyTop << Translate( 0, .5, 0 );
toyTop << Color( 0, 1, 1 ); // シアン
toyTop << Text( "center", "baseline", .2, "Toy Top" );
toyTop << Pop Matrix;

scene = Scene Box( 600, 600 ); // OpenGLシーンを保持するScene Boxを作成する

New Window( "例3", scene ); // シーンをウィンドウに配置する
scene << Perspective( 45, 3, 7 );
scene << Translate( 0.0, 0.0, -4.5 );
scene << Rotate( -85, 1, 0, 0 );
scene << Rotate( 65, 0, 0, 1 );
scene << Call List( toyTop );

scene << Update; // ディスプレイボックスを更新する
```

スタック上の現在の行列を変更したい場合は、Load Matrixコマンドを使います。

```
Load Matrix(m)
```

*m*は、現在の行列スタックにロードされる4行4列のJMP行列です。

同様なコマンドにMult Matrixがあります。

```
Mult Matrix(m)
```

Mult Matrixコマンドを実行すると、現在の行列スタックの最上部の行列に*m*が乗算されます。

次に、簡単なコマンドを実行する行列の例を示します。

移動

```
[1  0  0  x
0  1  0  y
0  0  1  z
0  0  0  1]
```

次の回転用行列に対する説明において、cはcos(a)、sはsin(a)です（ここで、aは回転の角度です）。

*x*軸を中心にして回転

```
[1  0  0  0
0  c -s  0
0 -s  c  0
0  0  0  1]
```

*y*軸を中心にして回転

```
[c  0  s  0
0  1  0  0
-s  0  c  0
0  0  0  1]
```

*z*軸を中心にして回転

```
[c -s  0  0
s  c  0  0
0  0  1  0
0  0  0  1]
```

例として、表示リストを移動して回転する2通りの方法を示します。初めの例は行列を用いており、2番目の例はTranslateとRotateを用いています。初めの例は、2番目の例と反対の方向に移動します。

```
// 行列を使った方法
gl << Push Matrix;
xt = Identity( 4 ); // これを 0.75 だけ左方向に移動する
xt[1, 4] = -.75;
xr = Identity( 4 ); // これを回転する。cos は度ではなくラジアンで指定する
xr[2, 2] = Cos( 3.14159 * a / 180 );
xr[2, 3] = -Sin( 3.14159 * a / 180 );
xr[3, 2] = Sin( 3.14159 * a / 180 );
xr[3, 3] = Cos( 3.14159 * a / 180 );
yr = Identity( 4 );
```

```
yr[1, 1] = Cos( 3.14159 * a / 180 );
yr[1, 3] = Sin( 3.14159 * a / 180 );
yr[3, 1] = -Sin( 3.14159 * a / 180 );
yr[3, 3] = Cos( 3.14159 * a / 180 );
zr = Identity( 4 );
zr[1, 1] = Cos( 3.14159 * a / 180 );
zr[1, 2] = -Sin( 3.14159 * a / 180 );
zr[2, 1] = Sin( 3.14159 * a / 180 );
zr[2, 2] = Cos( 3.14159 * a / 180 );
gl << Mult Matrix( xt * xr * yr * zr );
// 行列では乗算の順序が重要
gl << Arcball( dl, 1 );
gl << Pop Matrix;

// 関数を使った方法
gl << Push Matrix;
gl << Translate( .75, 0, 0 ); // これを 0.75 だけ右方向に移動する
gl << Rotate( a, 1, 0, 0 ); // これを度単位で回転させる
gl << Rotate( a, 0, 1, 0 ); // ここでは演算子の順序も重要
gl << Rotate( a, 0, 0, 1 );
gl << Arcball( dl, 1 );
gl << Pop Matrix;
```

行列は、表示リストの描画中にだけ保持されているので、以前の変換行列を遡って読み取ることはできません。以前の変換行列を知る必要がある場合には、JSL変数としてその行列を保持しておき、`Load Matrix`を使ってその行列をスタック上に置きます。

ライトと法線

次の方法を使って、図形に、ライト、材質、法線ベクトルを追加できます。これらの方法を使うと、モデルに明暗を付けて形状を明確にすることができます。

光源の作成

光源には、色、位置、および方向を指定します。JSLでは、`Light`コマンドで最大8つ（0～7）の光源を定義できます。*n*は光源の番号です。

```
Light( n, argument, value, ... value )
```

メモ: 各光源をオンにするには、`Enable (Lighting)`と`Enable (lightn)`コマンドを使います。*n*は光源の番号です。次に、`Light(n, POSITION, x, y, z)`コマンドで、光源をシーン内に移動します。

表13.3に、指定できる引数（*argument*）の値を示します。各引数のデフォルト値も示しています。

表13.3 Lightの引数とデフォルト値

引数	デフォルト値	意味
AMBIENT	(0, 0, 0, 1)	環境光（周囲光）のRGBA 輝度
DIFFUSE	(1, 1, 1, 1)	拡散光のRGBA 輝度
SPECULAR	(1, 1, 1, 1)	鏡面光のRGBA 輝度
POSITION	(0, 0, 1, 0)	(x, y, z, w)の位置
SPOT_DIRECTION	(0, 0, -1)	スポットライトの(x, y, z)方向
SPOT_EXPONENT	0	スポットライト指数
SPOT_CUTOFF	180	スポットライトの遮断角度
CONSTANT_ATTENUATION	1	一定減衰率
LINEAR_ATTENUATION	0	線形減衰率
QUADRATIC_ATTENUATION	0	2次減衰率

メモ: この表のDIFFUSEとSPECULARのデフォルト値はLight 0だけに適用されます。その他の光源については、この2つの引数のデフォルト値は(0, 0, 0, 1)です。

最初の3つの引数（AMBIENT、DIFFUSE、SPECULAR）は、光源に色を付ける場合に使います。DIFFUSE（拡散）は、光源の物理的な色に最も近く関連付けられる引数です。AMBIENT（環境）は、光がもつ背景光としての特徴に対応したパラメータです。SPECULAR（鏡面）は、表面に反射する光に関連したパラメータです。

POSITION引数を使って、光源の位置を指定します。4番目の座標（w）に0以外の値を指定すると、光源はオブジェクト座標と同じ座標に配置されます。

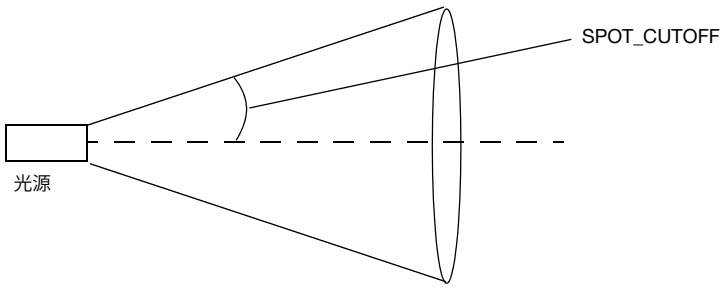
現実世界の光は、光源から離れるに従って暗くなります。ただし、指向性の光に関しては、光源は無限に遠くにあるので、光の輝度の減衰を距離の関数で表すのは意味がありません。JSL では、光源の輝度に、次の減衰率を掛けて光を減衰させます。

$$\text{減衰率} = \frac{1}{c + ld + qd^2}$$

cはCONSTANT_ATTENUATION、lはLINEAR_ATTENUATION、qはQUADRATIC_ATTENUATIONです。

スポットライトを作成するには、光源の形を円錐形にします。次に示すように、SPOT_CUTOFF引数を使って円錐形の側面を定義します。

図13.16 スポットライト



遮断角のほかに、円錐形内の照射の輝度と方向も制御できます。SPOT_DIRECTIONは、スポットライトを向ける方向を指定し、SPOT_EXPONENTは、集光方法に影響します。

ライトのモデル

ライトのモデルは、Light Modelコマンドで指定します。

```
Light Model( argument, value,...,value )
```

ライトのモデルでは、ライトの3つの属性を指定します。

- 広域環境光の輝度
- 視点が局所または無限の遠方のどちらにあるか
- ライトの計算をオブジェクトの前面と背面で別々に実行するか

表13.3（598ページ）に、Light Modelコマンドの有効な引数を示します。

表13.4 Light Modelの引数とデフォルト値

引数	デフォルト値	意味
LIGHT_MODEL_AMBIENT	(0.2, 0.2, 0.2, 1)	シーン全体の環境光（周囲光）のRGBA 輝度
LIGHT_MODEL_LOCAL_VIEWER	0（偽）	正反射角の計算方法
LIGHT_MODEL_TWO_SIDE	0（偽）	ゼロ以外の値は2側面のライトを意味する

法線ベクトル

法線ベクトルは、物体の表面に対して垂直な方向を向いています。平面の場合、法線はすべて同じです。より複雑な曲面の場合は、法線も複雑になります。JSLを使うと、各頂点の法線ベクトルを指定できます。これらの法線は空間での曲面の方向を指定するもので、ライトの計算に不可欠です。法線の精度が高ければ、それだけライトも正確になります。

法線ベクトルは、頂点に対して直角な、長さが1のベクトルです。通常、頂点は複数の多角形（ポリゴン）で共有されており、かつ、滑らかなシェーディング効果が望まれるので、頂点での垂線は多角形の標準の（加重）平均で計算されます。2面のシェーディングが可能でない限り、多角形の外側の面だけにライトが当てられるので、多角形の「外方向」の法線を計算することが重要です。多角形をスケーリングすると、法線の長さが1にならず、ライトが間違ったものになります。

法線ベクトルは、曲面が作成されるときに設定され、Normalコマンドで指定できます。シーンが描かれるたびに法線が1に再正規化されるようにするには、Enable(NORMALIZE)コマンドを使います。

シェーディングモデル

多角形（ポリゴン）のシェーディングモデルは、Shade Modelコマンドを使って設定します。

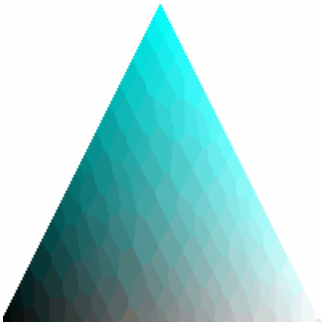
```
Shade Model (mode)
```

*mode*には、SMOOTH（デフォルト）またはFLATを指定できます。SMOOTHシェーディングは、頂点間で基本要素の色が滑らかになるように中間色を補間します。FLATモードは、1つの頂点の色を基本要素全体に適用します。

次のスクリプトは、三角形の各頂点で色を変更します。SMOOTHシェーディングモデルは、内部の色を自動的に補間します。

```
scene = Scene Box( 200, 200 ); // OpenGL シーンを保持する Scene Box を作成する

New Window( "シェーディングモデル", scene ); // シーンをウィンドウに配置する
scene << Clear;
scene << Ortho2D( -1, 1, -1, 1 );

scene << Shade Model( SMOOTH );
scene << Polygon Mode( FRONT_AND_BACK, FILL );
scene << Begin( TRIANGLES );
scene << Color( 0, 0, 0 ); // 黒
scene << Vertex( -1, -1, 0 );
scene << Color( 0, 1, 1 ); // シアン
scene << Vertex( 0, 1, 0 );
scene << Color( 1, 1, 1 ); // 白
scene << Vertex( 1, -1, 0 );
scene << End();

scene << Update;
```

図13.17 シェーディング



材質プロパティ

曲面の材質プロパティを設定するには、Materialコマンドを使います。

```
Material( face, argument, value,...value )
```

*face*には、Front、Back、またはFront_and_backを指定できます（材質プロパティは、多角形の前面と背面に対して別々に設定できます）。

表13.5に、Materialの引数とデフォルト値を示します。

表13.5 Materialの引数とデフォルト値

引数	デフォルト値	意味
AMBIENT	(0.2, 0.2, 0.2, 1.0)	材質の環境反射色
DIFFUSE	(0.8, 0.8, 0.8, 1.0)	材質の拡散反射色
AMBIENT_AND_DIFFUSE		AMBIENTとDIFFUSEの両方
SPECULAR	(0.0, 0.0, 0.0, 1.0)	材質の鏡面反射色
SHININESS	0	0～128の鏡面反射に対する指数
EMISSION	(0, 0, 0, 1)	材質の放射色

アルファブレンド

BlendFuncコマンドでは、アルファブレンドを作成できます。そのためには、次の構文を使って、BlendFuncメッセージをシーンに送ります。

```
scene << BlendFunc(SRC_ALPHA, ONE_MINUS_SRC_ALPHA)
```

SRC_ALPHAとONE_MINUS_SRC_ALPHAはOpenGLの定数で、アルファ値を使って既存の表示バッファに対するブレンドを行います。アプリケーションによっては、zバッファテストや、要素を後ろから前に向かって順番にレンダリングする手法を無効にする必要があります。デフォルトでは、zバッファテストが行われ、透明な多角形の後ろにオブジェクトを描画することはできません。

BlendFuncで使用できるすべての定数（大半はJSLプログラマには不要）の詳しい説明については、opengl.orgにあるOpenGLマニュアルを参照してください。

霧

霧を使うと、図形を遠方にフェードして、モデルをより現実に近付けることができます。すべてのタイプの幾何図形に霧をかけることができます。霧をオンにするには、FOG引数を使います。

```
Enable (FOG)
```

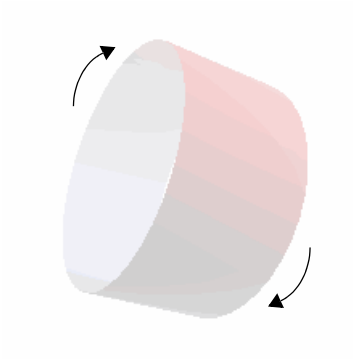
例

次の例では、ライト、霧、正規化など、この節で説明した複数の概念を使っています。このスクリプトは、2つの光源の影響を受ける、回転する円柱を描きます。

```
scene = Scene Box( 300, 300 ); // Scene Box を作成する
New Window( "円柱", scene ); // シーンをウィンドウに配置する

For( i = 1, i < 360, i++,

   scene << Clear;
   // レンズは 45 度、近い距離（near）はカメラから 3 単位、遠い距離（far）は 7 単位
   scene << Perspective( 50, 1, 10 );

   scene << Translate( 0.0, 0.0, -2 );
   // カメラで原点 (0,0,0) が見えるように移動する

   scene << Enable( Lighting );
   scene << Enable( Light0 );
   scene << Enable( Light1 );
   scene << Light( Light0, POSITION, 1, 1, 1, 1 ); // ビュアーに近い
   scene << Light( Light0, DIFFUSE, 1, 0, 0, 1 ); // 赤色の光源

   scene << Light( Light1, POSITION, -1, -1, -1, 1 ); // オブジェクトの背後
   scene << Light( Light1, DIFFUSE, .5, .5, 1, 1 ); // 青灰色の光源

   scene << Enable( Fog );
   scene << Enable( NORMALIZE );

   scene << Rotate( i, 1, 0, 0 );
   scene << Rotate( i * 3, 0, 1, 0 );
```

```
        scene << Rotate( i * 3 / 2, 0, 0, 1 );
        scene << Cylinder( 0.5, 0.5, 0.5, 40, 10 );
        scene << Update;
        Wait( 0.01 );
    );
```

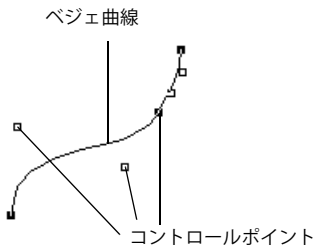
図13.18 霧



ベジェ曲線

このマニュアルでは、ベジェ曲線についての詳しい説明は省略します。JSL には、曲線や曲面とそれに対するメッシュを定義・描画するためのコマンドが用意されています。

図13.19 ベジェ曲線



1次元評価機能

1次元のベジェ曲線を定義するには、Map1コマンドを使います。

```
Map1(target, u1, u2, stride, order, matrix)
```

*target*引数には、制御点で何を表すかを定義します。*target*引数の値は、表13.6に示しています。Enableコマンドを使って、引数を有効にする必要があります。

表13.6 Map1の*target*引数とデフォルト値

*target*引数	意味
MAP1_VERTEX_3	(x, y, z) 頂点座標
MAP1_VERTEX_4	(x, y, z, w) 頂点座標
MAP1_INDEX	色インデックス
MAP1_COLOR_4	R、G、B、A
MAP1_NORMAL	法線座標
MAP1_TEXTURE_COORD_1	s テクスチャ座標
MAP1_TEXTURE_COORD_2	(s,t) テクスチャ座標
MAP1_TEXTURE_COORD_3	(s,t,r) テクスチャ座標
MAP1_TEXTURE_COORD_4	(s,t,r,q) テクスチャ座標

2番目の2つの引数（*u1*と*u2*）には、曲線の範囲を指定します。*stride*は、記憶されている制御点の、各ブロックごとの数値の個数（つまり、ある制御点から、次の制御点までのオフセット）です。*order*は、曲線の次数に1を足した値です。*matrix*には、制御点を指定します。

たとえば、Map1(MAP1_VERTEX_3, 0, 1, 3, 4, *<4x3 matrix>*)のように指定します。これは、2つの端点に対し、制御点を2つ設定してベジェ曲線を定義する典型的な方法です。

MapGrid1とEvalMesh1コマンドを使って、等間隔のメッシュを定義し、適用します。

```
MapGrid1(un, u1, u2)
```

このコマンドは、*u1*から*u2*の範囲にわたって*un*分割されるメッシュをセットアップします。0から1までを範囲とすれば、コーディングが簡素化されます。

```
EvalMesh1(mode, i1, i2)
```

このコマンドは、*i1*から*i2*まで実際にメッシュを生成します。*mode*には、POINTまたはLINEを指定できます。EvalMesh1コマンドは、固有のBegin節およびEnd節を作成します。

次のスクリプト例では、ベジェ曲線を描きます。乱数によって決められた制御点をもとに、滑らかな曲線が描かれます。最初と最後の点だけが曲線上にあります。NPOINTS=4を指定していますので、3次のベジェ曲線が描かれています。

```
boxwide = 500;
boxhigh = 400;

gridsize = 100; // 値を大きくすると、より見やすい間隔になる

NPOINTS = 4;
   /* 2 以上 8 以下の値を指定してください。この範囲外の数値は、実装方法によっては正しく動作しないかもしれません。この値は、曲線の次数に 1 を足したものです。*/
```

```
point = J( NPOINTS, 3, 0 );
   // x、y、zの3次元配列を作成する
For( x = 1, x <= NPOINTS, x++,
   point[x, 1] = (x - 1) / (NPOINTS - 1) - .5;
      // xの範囲は-.5～+.5
   point[x, 2] = Random Uniform() - .5;
      // yは同じ範囲の乱数
   point[x, 3] = 0;
      // 1つの平面上に曲線を描くので、zは常にゼロ
);

spline = Scene Box( boxwide, boxhigh );

spline << Ortho( -.6, .6, -.6, .6, -2, 2 );
   // xとyの範囲は-0.5～0.5なので、それよりわずかに大きくする

spline << Enable( MAP1_VERTEX_3 );
spline << MapGrid1( gridsize, 0, 1 );
spline << Color( .2, .2, 1 ); // 青色の曲線

spline << Map1( MAP1_VERTEX_3, 0, 1, 3, NPOINTS, point );
spline << Line Width( 2 ); // 細すぎない曲線にする
spline << EvalMesh1( LINE, 0, gridsize ); // LINEをPOINTに変更して試してください

spline << Color( .2, 1, .2 );
spline << Point Size( 4 ); // 大きい緑色の点

// 点を表示してラベルを付ける
For( i = 1, i <= NPOINTS, i++,
   spline << Begin( "POINTS" );
   spline << Vertex( point[i, 1], point[i, 2], point[i, 3] );
   spline << End;
   spline << Push Matrix;
   spline << Translate( point[i, 1], point[i, 2], point[i, 3] );
   spline << Text( center, bottom, .05, Char( i ) );
   spline << Pop Matrix;
);

New Window( "スプライン", spline );
```

http://www.tinaja.com/glib/bezconn.pdf では、勾配と変化率が連結点で一致している、区分的な3次式を求める方法が説明されています。上記の例は区分的な曲線は1本だけで、複数の区分的な曲線を連結しているわけではありません。

2次元評価機能

2次元評価機能は、1次元評価機能と対をなすもので、使い方も似ています。

```
Map2(target, u1, u2, ustride, uorder, v1, v2, vstride, vorder, matrix)
Eval Coord2(u, v)
```

*target*引数の値は、*Map1*を*Map2*に置き換えれば、表13.6（604ページ）に示した値と同じです。*u1*、*u2*、*v1*、および*v2*の値は、2次元メッシュの範囲を指定します。

たとえば、Map2(MAP2_VERTEX_3, 0, 1, 3, 4, 0, 1, 12, 4, *<16x3 matrix>*)のように指定します。これは16点によってベジェ曲面を定義する典型的な方法です。

MapGrid2とEvalMesh2コマンドを使って、等間隔のメッシュを定義し、適用します。

```
MapGrid2(un, u1, u2, vn, v1, v2)
```

このコマンドは、*u1*から*u2*、*v1*から*v2*までの範囲にわたって、*un*分割および*vn*分割されるメッシュを作成します。0から1までを範囲とすれば、コーディングが簡素化されます。

```
EvalMesh2(mode, i1, i2, j1, j2)
```

このコマンドは、*i1*から*i2*、*j1*から*j2*までのメッシュを実際に生成します。*mode*には、POINT、LINE、またはFILLを指定できます。EvalMesh2コマンドは、固有のBegin節およびEnd節を作成します。

マウスの使用

マウスの動作に対する処理は、2つのフィードバック関数によってサポートされています。Patch Editor.jslサンプルスクリプトでは、これらの関数を使って、点をドラッグ&ドロップする例を示しています。そのスクリプトの一部、マウス動作に対するコールバック関数について、以下に説明します。スクリプトを実行するには、JMPの「Sample Scripts」フォルダの「Scene3D」フォルダにある「PatchEditor.jsl」を開きます。

```
topClick2d = Function( {x, y, m, k},
   dragfunc( x, boxhigh - y, m, 1, 2 );
   1;
);
frontClick2d = Function( {x, y, m, k},
   dragfunc( x, boxhigh - y, m, 1, 3 );
   1;
);
rightClick2d = Function( {x, y, m, k},
   dragfunc( x, boxhigh - y, m, 2, 3 );
   1;
);

Click3d = Function( {x, y, m, k, hitlist},
   If( m == 1,
```

```
        If( N Items( hitlist ) > 0,
            CurrentPoint = hitlist[1][3], /* リスト内の最初の行列が最も近接している。行列の3
    番目の要素は ID */
            CurrentPoint = 0
        );
        makePatch();
    );
    0; /* 0を戻すと、最初のマウスの押下時だけ処理。ドラッグ中も処理する場合は1を戻すようにする
    (その場合、マウスを放すと処理が終了。天体球は表示されない)。*/
);

/* 3つのClick2d関数から共通して呼び出される関数 */
dragfunc = function( { x, y, m, ix, iy}, /* ixとiyは、座標を含む行列において、X、Y、お
    よびZの列を示すためのインデックス */
    If( CurrentPoint > 0,
        points[CurrentPoint, ix] = (x / boxwide) * (orthoright - ortholeft) +
    ortholeft;
        points[CurrentPoint, iy] = (y / boxhigh) * (orthotop - orthobottom) +
    orthobottom;
        makepatch();
    )
);
```

2-D関数を呼び出すときの引数は、X、Y、M、Kです。

- XとYはマウスの座標です。
- Mは、マウスとボタンの状態を示します。M=0は、マウスが放されていることを表します。M=1は、ボタンが押されたことを表します。M=2は、ボタンが押されたままマウスが移動していることを表します。M=3は、ボタンが放されたことを表します。
- Kは、Shift、Alt、Ctrlの各キーに関連付けられます。K=1はShiftキー、K=2はCtrl（command）キー、K=3はAlt（Option）キーを表します。

3-D関数も同様の方法で呼び出されます。引数は、X、Y、M、K、hitlistで、hitlistは行列のリストです。

```
[znear, zfar, id1, id2, id3, ...]
```

znearは、オブジェクトの近い頂点までの、カメラからのZ方向の距離、zfarは、オブジェクトの遠い頂点までの、カメラからのZ方向の距離です。行列は、znearとzfarの中間ポイントを基準にして近い方から遠い方へと並べ替えられます。リスト内のidは、表示リストに格納したプッシュ名（pushname）、ロード名（loadname）、およびポップ名（popname）の値です。

2-D関数や3-D関数は、戻り値を使って、マウス処理を継続するかどうかを知らせます。継続する場合には、関数で「1」を戻してください。戻り値が1以外の場合、マウスのトラッキングが停止されます。2-D関数と3-D関数は並行実行されないので、この機能は必要です。この例とは異なり、場合によっては、2-Dではなく3-Dでトラッキングされるように、2-D関数で0を戻し、3-D関数で1を戻すようにしてもかまいません。

Pickコマンド

SceneBoxのコールバック関数では、2次元のマウス座標を取得した後、マウスの座標にある「名前が付けられた」オブジェクトをピックアップすることができます。以下の例では、hitlistには、(x,y)点を中心にした5x5ピクセルの選択領域内に含まれる項目(最大1000個)の末端部の名前が戻されます。戻される形式は最後の引数で決まります。1を指定すると1つの配列が、0を指定すると深度で並べ替えられた配列のリストが戻されます。

```
Track2d=function({x, y, m, k},
hitlist = theSceneBox<<pick( x, y, 5, 5, 1000, 1 );
  if ( nrow(hitlist) > 0, // 選択領域が空ではない
     ... hitlist[1..n] // 表示リスト内で定義されたオブジェクトの名前が入れられる
  ) );
```

いっぽう、Track3dコールバック関数では、選択領域が常に1 x 1ピクセルであるのに加え、マウスを移動しない限り選択が行われません。通常はこの方法を使用しますが、1x1ピクセルの選択領域は点と同じくらい小さいため、選択が困難なことがあります。また、Pickコマンドでは、マウスを移動しなくても選択が行われます。

Track3d関数では、常に深度で並べ替えられた配列のリストが戻されます。各配列は、階層を構成する複数のオブジェクト名です(プッシュ名およびポップ名は、オブジェクトの階層を構成します)。大量のオブジェクトが選択されている場合は、並べ替えに長い時間がかかることがあります。なお、Pick関数では、最後の引数(上の例では1)で、並べ替えた配列のリストまたは1つの配列のどちらを戻すかを制御できます。1つの配列を戻す場合は、末端部の名前だけが含まれ、それより上位の名前は含まれません。

引数

引数を使って、特定のモードと設定を指定できます。引数を有効にするには、Enable(*parameter*)コマンドを使います。引数を無効にするには、Disable(*parameter*)コマンドを使います。表13.7に、使用できる引数を示します。

表13.7 引数

ALPHA_TEST	LIGHT5	MAP2_TEXTURE_COORD_3
AUTO_NORMAL	LIGHT6	MAP2_TEXTURE_COORD_4
BLEND	LIGHT7	MAP2_VERTEX_3
CLIP_PLANE0	LIGHTING	MAP2_VERTEX_4
CLIP_PLANE1	LINE_SMOOTH	NORMALIZE
CLIP_PLANE2	LINE_STIPPLE	POINT_SMOOTH
CLIP_PLANE3	MAP1_COLOR_4	POLYGON_OFFSET_FILL
CLIP_PLANE4	MAP1_INDEX	POLYGON_OFFSET_LINE
CLIP_PLANE5	MAP1_NORMAL	POLYGON_OFFSET_POINT
COLOR_LOGIC_OP	MAP1_TEXTURE_COORD_1	POLYGON_SMOOTH
COLOR_MATERIAL	MAP1_TEXTURE_COORD_2	POLYGON_STIPPLE
CULL_FACE	MAP1_TEXTURE_COORD_3	SCISSOR_TEST
DEPTH_TEST	MAP1_TEXTURE_COORD_4	STENCIL_TEST
DITHER	MAP1_VERTEX_3	
FOG	MAP1_VERTEX_4	
LIGHT0	MAP2_COLOR_4	
LIGHT1	MAP2_INDEX	
LIGHT2	MAP2_NORMAL	
LIGHT3	MAP2_TEXTURE_COORD_1	
LIGHT4	MAP2_TEXTURE_COORD_2	

第14章

JMPの拡張

外部データソース、分析ツール、オートメーション

この章では、定期的に行う作業をプログラミングするときに役立つ、以下のスクリプト機能を紹介します。

- 測定機器などからのリアルタイムデータを取り込むデータフィード
- JMPスクリプト言語（JSL）を使ったSAS、MATLAB、Rの使用
- Microsoft Excelとの連携
- データベースへの接続
- OLEオートメーション（外部アプリケーションからのJMPの制御）
- XMLの解析

定期的な作業のプログラミングに関係したJSL関数には、`Caption`、`Speak`、`Print`、`Write`、`Mail`といったものもあります。これらのコマンドについては、「プログラミング手法」章の「メッセージを出力する関数」（265ページ）に説明があります。

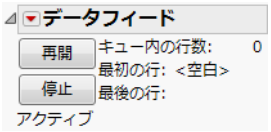
リアルタイムのデータ取得

データフィードは、シリアルポートに接続された測定機器などから、データをリアルタイムで読み込む機能です データフィードオブジェクトによって、入力用のキューを設定し、データの読み込みをバックグラウンドで処理できます。キューに届いたデータをデータテーブルに送るなどの処理も、スクリプトで実行できます。

たとえば、com1ポートからレコードを取得し、それをログにリストするには、次のスクリプトを実行します。

```
feed = Open Datafeed(
   Connect( Port( "com1:" ), Baud( 9600 ), DataBits( 7 ) ),
   Set Script( Print( feed << GetLine ) )
);
```

図14.1 データフィードウィンドウではステータスの確認や制御が可能



データフィードオブジェクトの作成

データフィードオブジェクトを作成するには、接続の詳細を指定したOpen DataFeedコマンドを使います。

```
feed = Open DataFeed( options );
```

引数は必須ではありません。データフィードオブジェクトを引数を指定せずに作成して、後からメッセージを送ることもできます。ポートへ接続したり、入力されたデータを処理するためのスクリプトを設定したりすることは、メッセージでも行えます。しかし、通常は、Open DataFeed関数の引数において、データフィードの基本設定は行っておきます。そして、引き続いて、データフィードを更に制御するのに必要なメッセージを送ります。後述のオプションはどれも、Open DataFeed関数の引数としても、またデータフィードオブジェクトに送るメッセージとしても使用できます。

上記の例のように、データフィードオブジェクトへの参照を、グローバル変数に格納しておくと良いでしょう（上記の例では、*feed*という変数に格納している）。変数に格納しておけば、後から、簡単にオブジェクトにメッセージが送れます。なお、既存のオブジェクトへの参照を、添え字を使って格納することも可能です。たとえば、次のスクリプトは、2番目に作成したデータフィードオブジェクトへの参照を変数に格納しています。

```
feed2 = Data Feed[2];
```

Datafeedのオプション

（Windowsのみ）生のデータソースに接続するには、Connect( )を使ってポートの詳細を指定します。設定項目は、それぞれ1つだけ引数をとります。ここの構文の説明では、引数間の記号「|」は「または」を意味します。接続する場合はPortを必ず指定してください。ポートを指定しなくてもオブジェクトは機能しますが、データフィードには接続されません。最後の3つの項目、DTR_DSR、RTS_CTS、XON_XOFFはブール値の引数をとり、データフィードがデータ取得可能な状態になったことを通知する際に、どの制御文字を送受信するかを指定します。通常、3つのうちの1つをオンにします。

```
feed = Open Datafeed(
   Connect(
      Port( "com1:" | "com2:" | "lpt1:" | ... ),
      Baud( 9600 | 4800 | ... ),
      Data Bits( 8 | 7 ),
      Parity( None | Odd | Even ),
      Stop Bits( 1 | 0 | 2 ),
      DTR_DSR( 0 | 1 ), // データターミナル準備
      RTS_CTS( 0 | 1 ), // 送信をリクエスト | 送信をクリア
      XON_XOFF( 1 | 0 ) // 送信器オン | 送信器オフ
   )
);
```

このコマンドにより、スクリプト可能なデータフィードオブジェクトが作成され、そのオブジェクトへの参照をグローバル変数 *feed* に格納します。引数Connectは、通信ポートを監視してデータのライン（行）をとりまとめるスレッドを立ち上げます。このスレッドは、ラインが確保されるまで文字を受け取ります。そして、それをキュー（待ち行列）に加え、スクリプトをコールするイベントのスケジュールを立てます。

メモ: データフィードの場合、データのライン（行）は単一の値です。データフィードのラインと、データテーブルの行を混同しないでください。後者は、1つの行に対して複数の値を持つことができます。

Set Scriptはデータフィードオブジェクトにスクリプトを割り当てます。このスクリプトは、データが届くたびにOn DataFeedハンドラによって実行されます。Set Script()の引数には、実行するスクリプトをそのまま記述するか、またはスクリプトを含むグローバル変数を指定します。

```
feed = Open Datafeed( Set Script( myScript ) );
feed = Open Datafeed( Set Script( Print( feed << Get Line ) ) );
```

データフィードのスクリプトでは、通常Get Lineを使って1ラインのデータを取得し、そのラインに対して処理を行います。このスクリプトはラインのデータを解析し、結果をデータテーブルに追加します。

リアルタイムデータの読み込み

外部データソースから、物理または論理通信リンクを介して別の機器に情報を送信することを、**ライブデータフィード**といいます。Windowsコンピュータのシリアルポートを介してJMPをライブデータフィード元に接続し、データストリームをリアルタイムで読み取ることができます。次の点を確認してください。

- データフィードは、標準の9ピンシリアルポートを介して行う必要があります。シリアルポートをシミュレートするドライバがなければ、USBポートを介してデータを読み取ることはできません。
- 接続機器の正確なボーレート、パリティ、ストップビット、データビットが必要です。

接続機器に関する数値を、Open Datafeed()コマンドの引数に指定します（以下のスクリプト例の4800、even、2、および7を置き換えてください）。次に、データフィード元の機器をコンピュータに接続し、スクリプトを実行します。

```
streamScript = Expr(
   line = feed << Get Line;
   Show( line );
   len = Length( line );
   Show( len );
   If( Length( line ) >= 1,
      Show( "Hi" );
      Show( line );
      field = Substr( line, 5, 8 );
      Show( field );
      x = Num( field );
      Show( x );
      If( !Is Missing( x ),
         dt << Add Row( {:Column1 = x} );
         Show( x );
      );
   );
);
feed = Open Datafeed(
   Baud Rate( 4800 ),
   parity( even ),
   Stop bits( 2 ),
   Data bits( 7 )
);
feed << Set Script( streamScript );
feed << Connect;
```

JMPと外部のデータフィード機器の通信設定を揃えるには、［**ファイル**］>［**環境設定**］>［**通信**］を選びます。お使いの外部機器の適切な設定については、機器のマニュアルを参照してください。

メッセージ付きデータフィードの制御

データフィードオブジェクトは、ConnectやSet Scriptなどのいくつかのメッセージに応答します。これらのメッセージは、Open Datafeedの引数としてすでに説明しましたが、既存のデータフィードオブジェクトに送るメッセージとしても使用できます。

```
feed << Connect( port( "com1:" ), baud( 4800 ), databits( 7 ), parity( odd ),
  stopbits( 2 ) );
feed << Set Script( myScript );
```

以下のメッセージはOn Data Feedの引数としても使用できます。ただし、既存のデータフィードオブジェクトへ送るメッセージとして使用する方が一般的です。

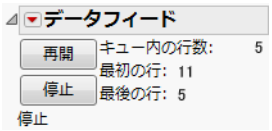
スクリプトからデータフィードへデータを送ることもできます。この方法により、データフィードを簡単にテストできます。引数にはテキストまたはテキストを格納するグローバル変数をとります。

```
feed << Queue Line( "14" );
feed << Queue Line( myValue );
```

以下は5行のデータを待機させるテストです。

```
feed << Queue Line( "11" );
feed << Queue Line( "22" );
feed << Queue Line( "33" );
feed << Queue Line( "44" );
feed << Queue Line( "55" );
```

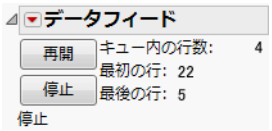
図14.2 データフィード: キューに5個の行



キューにある最初の行を取得するには、Get Line（単数形であることに注意）メッセージを使います。行を取得すると、その行はキューから削除されます。上のテスト用スクリプトでは5つのデータがキューに入れられています。Get Lineは最初のラインを取得した上で、それをキューから削除します。

```
feed << Get Line
  "11"
```

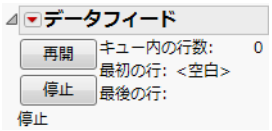
図14.3 データフィード: キューに4個の行



データをすべてリストに移し、キューを空にするにはGet Lines（複数形であることに注意）を使います。例では、テスト用スクリプトから次の4行を{ }形式のリストで戻します。

```
myList = feed << GetLines;
  { "22", "33", "44", "55" }
```

図14.4 データフィード: キューに0個の行



キューにあるデータの処理を停止し、後で再開するには、データフィードウィンドウの［**停止**］ボタンおよび［**再開**］ボタンをクリックするか、または同じ機能を持つ次のメッセージを送ります。

```
feed << Stop;
feed << Restart;
```

データフィードとそのウィンドウを閉じるには、次のメッセージを送ります。

```
feed << Close;
```

ライブデータソースからの接続を切断するには:

```
feed << Disconnect
```

データフィードの例

データの読み取り

以下は、データフィードに対するスクリプトの例です。このスクリプトでは、文字列の長さを読み取り、もし14文字以上であれば、11文字目以降を数値として取り込み、データテーブル中の「太さ」列に、1行を追加します。

```
feed = Open Datafeed( );
myScript = Expr(
   line = feed << Get Line;
   If( Length( line ) >= 14,
      x = Num( Substr( line, 11, 3 ) );
      If( !Is Missing( x ),
         dt << Add Row( {thickness = x} )
      );
   );
);
```

上記のスクリプトを、既存のデータフィードオブジェクトに割り当てるには、次のように`Set Script`を使います。

```
feed << Set Script( myScript );
```

ライブ管理図のセットアップ

以下は、新しいデータテーブルを作成し、データフィードに基づいて管理図の作成を開始するスクリプトの例です。

```
dt = New Table( "ギャップの幅" ); // 1つの列を持つデータテーブルを作成する
dc = dt << New Column( "ギャップ", Numeric, Best );

// 管理図のプロパティを設定する
dt << Set Property(
   "管理限界",
   {XBar( Avg( 20 ), LCL( 19.8 ), UCL( 20.2 ) )}
);
dt << Set Property( "Sigma", 0.1 );

// データフィードの作成
feed = Open Datafeed();
feedScript = Expr(
   line = feed << Get Line;
   z = Num( line );
   Show( line, z ); // ログまたはデバッグ用
   If( !Is Missing( z ),
      dt << Add Row( {:gap = z} )
   );
);
feed << Set Script( feedScript );

// 管理図を描く
Control Chart(
   Sample Size( 5 ),
   K Sigma( 3 ),
   Chart Col(
      ギャップ,
      XBar(
         Connect Points( 1 ),
         Show Points( 1 ),
         Show Center Line( 1 ),
         Show Control Limits( 1 )
      ),
      R(
         Connect Points( 1 ),
         Show Points( 1 ),
         Show Center Line( 1 ),
         Show Control Limits( 1 )
      )
   )
);

/* 機器からのデータフィードを始めるか、またはテスト用データを送って
 動作確認をする（いずれか1行をコメントアウトする）:
feed << Connect( Port( "com1:" ), Baud( 9600 ) );*/
```

```
For( i = 1, i < 20, i++,
   feed << Queue Line( Char( 20 + Random Uniform() * .1 ) )
);
```

データテーブルにスクリプトを格納する

前述のようなデータフィードのスクリプトを、データテーブルのOn Openプロパティに記述しておくことにより、定型的な処理をさらに自動化できます。On Openプロパティに設定されたスクリプトは、データテーブルが開くたびに自動的に実行されます（環境設定で実行しないように設定することもできます）。データフィード処理のスクリプトをOn Openプロパティに設定したデータテーブルを保存しておけば、そのデータテーブルを開くたびに、データフィードのスクリプトが実行され、データデータが取得されます。

表14.1 データフィードへのメッセージ

メッセージ	構文	説明
Open Data Feed Data Feed	feed = Open Datafeed(commands)	データフィードオブジェクトを作成する。以下のメッセージはいずれも、Open DataFeed内のコマンド、または、既存のデータフィードオブジェクトに送るメッセージとして使用できます。Data Feedは同義語。
Set Script	feed << Set Script(script)	データラインが届くたびに実行されるスクリプト（*script*）を設定する
Get Line	feed << Get Line	データフィードのキューの中から1ラインのデータを戻し、削除する。
Get Lines	feed << Get Lines	データフィードのキューのデータすべてをリストにして戻し、削除する。
Queue Line	feed << Queue Line(*string*)	データフィードのキューの最後へ1ラインのデータ送る。
Stop	feed << Stop	データラインへの処理を停止する。
Restart	feed << Restart	データラインへの処理を再開する。
Close	feed << Close	データフィードオブジェクトとそのウィンドウを閉じる。

表14.1 データフィードへのメッセージ（続き）

メッセージ	構文	説明
Connect	feed << Connect(Port("com1:" | "lpt1:" | ...), Baud(9600 | 4800 | ...), Data Bits(8 | 7), Parity(None | Odd | Even), Stop Bits(1 | 0 | 2), DTR_DSR(0 | 1), RTS_CTS(0 | 1), XON_XOFF(1 | 0))	（Windowsのみ）デバイスに接続するために、ポートの設定を行う。引数の間にある記号「|」は「または」を意味し、設定ごとにどれか1つの引数を選びます。
Disconnect	feed << Disconnect	（Windowsのみ）データフィードオブジェクトをアクティブにしたまま、データフィードのキューとデバイスとの接続を切断する。

ダイナミックリンクライブラリ（DLL）

メモ: 64ビット版のJMPは32ビットDLLがロードできず、32ビット版のJMPは64ビットDLLがロードできません。過去に32ビットDLLを使用していて、現在64ビット版のJMPを使用している場合は、DLLを再コンパイルして64ビット版でロードできるようにしなければなりません。

JMPスクリプト言語（JSL）によって、DLLをロードし、そのDLLに含まれている関数を呼び出すことができます。DLLの呼び出しを行うJSL関数が1つと、メッセージが6つあります。

```
dll_obj = Load DLL("path" <, AutoDeclare(Boolean | Quiet | Verbose) |Quiet |
  Verbose > )
```

Load DLL()は、パス（*path*）で指定されたDLLをロードします。ログウィンドウにメッセージが表示されないようにするには、AutoDeclare(Quiet)引数を使用します。

DLLで定義される関数の引数や戻り値のタイプを宣言するには、Declare Functionメッセージを使用します。
関数を宣言すると、呼び出せるようになります。

```
dll_obj <<Declare Function("name", Convention(named_argument), Alias("string"),
  Arg(type, "string"), Returns(type), other_named_arguments)
```

Aliasは、JSLで使用できる代替の名前を定義します。たとえば、DLL内の"Message Box"という関数に対して、Alias("MsgBox")を宣言した場合、その関数を次のようにも呼び出せるようになります。

```
result = dll_obj <<MsgBox(...)
```

Conventionの引数には、次のいずれかを指定します。

- STDCALLまたはPASCAL
- CDECL

ArgとReturnsのtype引数には次のいずれかを指定します。

表14.2 ArgおよびReturnsの種類

Int8	UInt8	Int16	UInt16
Int32	UInt32	Int64	UInt64
Float	実数（double）	AnsiString	UnicodeString
Struct	IntPtr	UIntPtr	ObjPtr

Declare Functionメッセージの引数については、『スクリプト構文リファレンス』の「JSLメッセージ」章を参照してください。

最後に、UnLoadDLLメッセージでDLLをアンロードします。

```
dll_obj << UnLoadDLL
```

メモ: 関数を宣言するときは、DLLの作成者によって提供された関数のマニュアルを参照してください。引数の種類や呼び出し方法がDLL内の実際の関数に合わない場合、JMPが終了してしまう可能性があります。

例

Windowsの32ビットDLLと64ビットDLLのうち、ユーザのコンピュータに応じて適切な方をロードするようなスクリプトを書きたいとします。この例は、Windowsオペレーティングシステムを確認し、次にコンピュータのプロセッサのビットレベルを確認します。

```
If( Host is( Windows ),
   If(
      Host is( Bits64 ),
      //  64ビットDLLをロードする
         dll_obj = Load DLL( "c:\Windows\System32\user32.dll" ),
      // 32ビットDLLをロードする
         dll_obj = Load DLL( "c:\Windows\SysWOW64\user32.dll" ),
      // どちらのDLLも見つからない場合は実行を停止する
         Throw
      );
   dll_obj <<DeclareFunction(
```

```
        "MessageBoxW",
        Convention( STDCALL ),
        Alias( "MsgBox" ),
        Arg( IntPtr, "hWnd" ),
        Arg( UnicodeString, "message" ),
        Arg( UnicodeString, "caption" ),
        Arg( UInt32, "uType" ),
        Returns( Int32 )
    );
    result = dll_obj << MsgBox(
        0,
        "JMP からのメッセージです。",
        "DLL の呼び出し ",
        321
    );
    Show( result );
);
dll_obj << UnLoadDLL
```

その他のDLLメッセージ

Show Functionsメッセージは、Declare Functionで宣言された関数をすべてログに送ります。

```
dll_obj << Show Functions;
```

独自のDLLを作成するときには、DLL内にJSLの関数宣言スクリプトを用意しておくことができます。Get Declaration JSLメッセージは、DLL内の関数宣言スクリプトをログに送ります。

```
dll_obj << Get Declaration JSL;
```

JSLでのソケットの使用

データをフィードするもう1つの方法は、ソケット通信です。JSLでは、2種類のソケットが作成できます。JSLでは、2種類のソケットが作成できます。

ストリーム ストリームソケットは、JMPともう1つのコンピュータとの間で信頼性の高い接続を確立します。JMPを実行しているコンピュータ、業務用機器、データ収集用コンピュータ、プリンタといったソケット通信が可能なデバイスと通信できます。ソケット通信を行えるHTTP Webサーバーなどとも通信できます。

データグラム データグラムソケットも、JMPともう1つのコンピュータとの間に接続を確立しますが、信頼性はやや劣ります。データグラムはコネクションレスで、情報が何度も届いたり、まったく届かなかったり、順不同で届いたりします。データグラム接続には、信頼性が保証されているストリーム接続のようなオーバーヘッドが備わっていません。データグラムは、コネクションレスのため、（同じソケットであっても）接続先のアドレスを毎回指定する必要があります。

ソケットを作成すると、別のソケットに接続するか、別のソケットからの接続を待機することができます。以下に、別のコンピュータのWebサーバーに接続してデータを取得するという簡単なプログラム例を示します。

```
tCall = Socket( );
tCall << Connect( "www.jmp.com", "80" );
tCall << Send( Char To Blob( "GET / HTTP/1.0~0d~0a~0d~0a", "ASCII~HEX" ) );
tMessage = tCall << Recv( 1000 );
text = Blob To Char( tMessage[3] );
Show( text );
tCall << Close( );
```

1行目は、ソケットを作成し、それに参照名（tCall）を与えます。デフォルトでは、ストリームソケットが作成されます。ソケットの種類を指定するには、オプションの引数socket(STREAM)またはsocket(DGRAM)を使います。

2行目は、tCallソケットをJMP Webサイトのポート80（通常はHTTPポート）に接続します。

3行目は、JMP WebサーバーにGET要求を送信します。このメッセージは、JMP Webサーバーに対し、JMPホームページを返信するように要求します。GETの後の / は、開くページのパスでなければなりません。/ はルートページを開きます。

4行目は、JMP Webサーバーから最大1000バイトを受信し、情報のリストをtMessageに格納します。ソケットの呼び出しは、それぞれリストを戻します。リストの最初の要素は、呼び出しの名前です。2番目の要素はテキストメッセージで、okだけのことも、長い診断メッセージのこともあります。場合によっては、各呼び出しに応じた追加の要素が存在します。この例では、リストの3番目の要素が受信したデータです。

5行目は、受信したバイナリ情報を文字列に変換します。tMessage[3]は、Recvによって戻されるリストの3番目の要素で、JMP Webサーバーからのデータです。

6行目はログにデータを表示します。

最後の行はソケットを閉じます。接続先のWebサーバーはすでに閉じているため、このソケットには再接続か適切な廃棄（close）が必要です。

JMPのSamples/Scriptsフォルダにソケットを使用したスクリプトの例がいくつかあります。

ソケット関連のコマンド

場合によっては、ソケットを作成して使用する前に接続先に関する情報を取得しておく必要があります。GetAddrInfo( )およびGetNameInfo( )は、アドレス引数とオプションのポート引数を取得し、情報のリストを戻します。例:

```
Print( Get Addr Info( "www.sas.com" ) );
Print( Get Addr Info( "www.sas.com", "80" ) );
Print( Get Name Info( "149.173.5.120" ) );
  {"Get Addr Info", "ok", {"PF_INET", "SOCK_0", "IPPROTO_0", "149.173.5.120",
    "0"}}
  {"Get Addr Info", "ok", {"PF_INET", "SOCK_0", "IPPROTO_0", "149.173.5.120",
    "80"}}
```

```
{"Get Name Info", "ok", {"PF_INET", "SOCK_0", "IPPROTO_0", "www.sas.com",
  "0"}}
```

複数の回答がある場合があります。その場合は、サブリストが繰り返されます。これらの関数は、処理にかなり時間がかかる場合があるので、すべてのWebサイト名を記載したデータテーブルを作成することはお勧めできません。IPV6との互換性を維持するために、数値アドレスではなく、"www.sas.com"のような名前を使用してください。

ソケットへのメッセージ

Socket( )を使って作成したソケットには、さまざまなメッセージを送ることができます。

connect リスニングソケットに接続する。接続に成功した場合は{"connect", "ok"}、失敗した場合はエラーが戻されます（たとえば、{"connect", "CONNREFUSED: The attempt to connect was forcefully rejected."}）。

close 処理の終了時に接続を閉じます。リストを戻します（たとえば{"Close", "ok"}）。

send ソケットの相手側にSTREAMメッセージを送信します。

sendto ソケットの相手側にDGRAMメッセージを送信します。

recv STREAMメッセージを受信します。データとその他の情報がリスト形式で戻されます。Recvには、受信できるバイト数を指定する数値引数が必要です。

recvfrom DGRAMメッセージを受信します。

ioctl ソケットのブロック動作を制御します。デフォルトでは、ソケットはブロックに設定されています。データが使用できるようになるまで、ソケットはJSLプログラムに制御を戻しません。これによってスクリプトの作成は容易になりますが、接続先がデータの供給に失敗した場合に特に頑健ではありません。非ブロックに設定されているソケットは、即座に"ok"コードとデータを戻すか、または"WOULDBLOCK: ..."コードを戻しますが、ソケットがブロックのときは、データが使用可能になるまで待たなければなりません（次のJSLステートメントへの進行をブロックする）。

重要: この問題を回避するため、JSLコールバックを使用するバックグラウンド処理があります。recv、recvfrom、またはacceptは、ソケットを非ブロックに設定し、Waitステートメントの間やJMPがアイドルのときにポーリングして、バックグラウンドで動作させることができます。

ioctlはリストを戻します。たとえば、{"ioctl", "ok"}、またはソケットがバインド（下のbindを参照）または接続されていない場合は{"ioctl", "NOTCONN: The socket is not connected."}を戻します。

bind クライアントソケットが待機（リッスン）するアドレスを、サーバーソケットに通知します。bindは、ローカルコンピュータのポートをソケットに関連付けます。ソケットがlistenするためには、この処理が必要です（下を参照）。bindで指定したポートは、接続するソケットに対していつも使用されるとは限りません。オペレーティングシステムが、使用されていないポートを選びます。サーバーにはbindが必要です。サーバーに接続したいユーザは、どのポートが使用されるのかを知っている必要があるためです。通常使用されるポートは、HTTPポートである80です。bindはリストを戻します。

たとえば、{"bind", "ok"}、または使用しているコンピュータ上にない名前にバインドしようとしている場合は{"bind", "ADDRNOTAVAIL: The specified address is not available from the local machine."}を戻します。別のソケットは、使用しているコンピュータの名前と番号がわかっていれば、このソケットに接続することができます。

listen 接続をリッスンするように、サーバーのソケットに伝えます。リスニングソケットは、他のソケットからの接続をリッスンします。一度、リッスンの状態にすれば、継続して、リッスンの状態になっています。accept（下を参照）は、他のソケットからの接続を受け入れるのに使用します。listenはリストを戻します。たとえば、{"listen", "ok"}、またはバインドコールに失敗した場合は、{"listen", "INVAL: The socket is (または状況によってはis not) already bound to an address.または、Listen was not invoked prior to accept.または、Invalid host address.または、The socket has not been bound with Bind."}を戻します。

accept サーバーソケットに、接続を受け入れて新しい接続ソケットを戻すよう伝えます。acceptは、何が起こったかを記したリストを戻します。処理が成功した場合は接続先のソケットに接続される新しいソケットもリストに記されます。たとえば、{"Accept", "ok", "localhost", socket( )}などです。ここで、localhostは接続したコンピュータの名前で、4番目の引数はメッセージをsendまたはrecvするのに使用するソケットです。

getpeername 接続先について問い合わせます。GetPeerNameは、接続先のソケットに関する情報のリストを戻します。たとえば、{"getpeername", "ok", "127.0.0.1", "4087"}といったリストを戻します。サーバーソケットの場合、接続したクライアントのアドレスとポートが判明します。クライアントソケットの場合、接続要求で使用したサーバーの名前とポートを再確認できます。

getsockname 接続元について問い合わせます。GetSockNameは、こちら側のソケットに関する情報のリストを戻します。たとえば、{"getsockname", "ok", "localhost", "httpd"}といったリストを戻します。クライアントソケットの場合、オペレーティングシステムが割り当てたポートが判明します。サーバーソケットの場合、bindによってすでにその情報はわかっています。

データベースアクセス

クエリービルダーのクエリーの実行

クエリービルダーは、SQLステートメントを書かずにSQLデータベースのデータを選択し、読み込むのに適した手段です。データテーブルに読み込む前に、データをプレビューすることができます。クエリーを共有し、他のユーザによるカスタマイズや実行を可能にすることができます。

作成後のクエリーは、JSLスクリプトで実行できます。スクリプトを開かずにクエリーを実行するには、Include()を使用します。

```
query = Include( "$DOCUMENTS/Airline.jmpquery" );
query << Run Foreground( );
```

クエリーをフォアグラウンドで実行する代わりに、Run Background()を使ってバックグラウンドで実行するか、Run()を使ってクエリーの実行に関するクエリービルダーの環境設定を適用することもできます。デフォルトでは、クエリーはバックグラウンドで実行されます。

クエリーメッセージの詳細については、『スクリプト構文リファレンス』のSQLの節を参照してください。

Open Database関数

JMPではODBCがサポートされています。Open Database関数を使ってJSLでSQLデータベースへアクセスできます。

```
dt = Open Database(
   "Connect Dialog" | "DSN=...", // データソース
   "sqlStatement" | "dataTableName" | "SQLFILE=...", // SQL ステートメント
   Invisible,  // テーブルの読み込み時にそのテーブルを非表示にするオプションのキーワード
   "outputTableName" // 新しいテーブルの名前
);
```

メモ: データベーステーブル名に $# -+/%()&|;?の文字を使用する場合は、角括弧（[]）で囲む必要があります。

第1引数には、読み込みたいデータソースを指定します。次のいずれかを指定してください。

- "Connect Dialog"でSelect Data Sourceウィンドウ（Windows）またはChoose DSNウィンドウ（Macintosh）を表示します。
- "DSN="の後にデータソースの名前、および、接続に必要な情報を指定すると、そのデータソースに接続します。Windowsでは、ODBC データソース アドミニストレーターの［ユーザー DSN］タブまたは［システム DSN］タブの名前列にデータソース名が表示されます。Macintoshでは、ODBC ManagerまたはiODBC Driver ManagerにDSNが表示されます。その他の文字列は、ODBCデータソースによって異なります。

例:

```
"DSN=dBASE Files;DBQ=C:/Program Files/SAS/JMP/13/Samples/Import Data;"
```

第2引数には、次の3つのうちいずれかを2重引用符で囲んで指定します。

1. 実行するSQLステートメント。たとえば次のように、2重引用符で囲んで、SELECTステートメントを指定してください。

   ```
   "SELECT AGE, SEX, WEIGHT FROM BIGCLASS"
   ```

   SQLは、データソースがサポートしているSQLに適合していなければなりません。つまり、「Big Class」という名前のテーブルは、「Big」と「Class」の間にスペースがあるため、それを考慮して引用符で囲む必要があります（スペースが許可されている場合）。引用符はデータソースによって異なりますが、通常は"、'、‘、[]を使います。

2. データテーブルの名前。データテーブル名だけを指定した場合、"SELECT * FROM" というSQLステートメントを実行するのと同じ処理が行われます。たとえば、第2引数を次のように指定すれば、Open Databaseは"SELECT * FROM BIGCLASS"を実行することになります。

   "BIGCLASS"

3. "SQLFILE="に続いて、実行するSQLステートメントの書かれたテキストファイルへのパス。たとえば次の引数の場合は、ディレクトリC:¥にあるファイル「mySQLFile.txt」を開き、中に書かれたSQLステートメントを実行します。

   "SQLFILE=C:¥mySQLFile.txt"

オプションの引数Invisibleは、非表示のデータテーブルを作成します。なお、非表示のデータテーブルは、明示的に閉じるまでメモリ内にとどまるため、不要になったものは閉じるよう注意してください。非表示のテーブルを閉じるには、Close(dt)を実行します。ここで、dtは、データテーブル参照の変数です。

オプションの引数*outputTableName*は、作成される出力テーブルがあれば、その名前を指定します。Open Databaseが必ずデータテーブルを戻すわけではないことに注意してください。戻り値はヌルになることもあります。データテーブルが戻されるかどうかは、実行されるSQLステートメントの種類によります。たとえば、SELECTステートメントはデータテーブルを戻しますが、DROP TABLEステートメントはデータテーブルを戻しません。

JMPのデータテーブルを、JSLを使ってデータベースに保存するには、データテーブルの参照にSave Database( )メッセージを送ります。

```
dt << Save Database( "connectInfo", "TableName" );
```

第1引数は、Open Databaseの場合と同様に機能します。一部のデータベースでは、既存のテーブルに置き換えてテーブルを保存することができないので注意してください。そのような場合にデータベース上のテーブルを置換するには、Open Databaseコマンドで、「DROP TABLE」というSQLステートメントを実行して、そのテーブルをあらかじめ削除してください。

```
Open Database ( "connectinfo", "DROP TABLE TableName" );
```

メモ: JMP 13は、名前にスペースが入っているテーブルや、大文字と小文字が混在しているテーブルでも、データソースがサポートしている限り、保存することができます。スペースは、Apache HiveとApache Hadoop以外のほとんどのデータソースでサポートされています。大文字と小文字が混在している名前は、そのまま使用されますが、SQLではほとんどの場合大文字と小文字の区別がありません。

次のスクリプトは、SQLクエリーでデータベースを開き、それを新しい名前でデータベースに保存し、その後、新しいテーブルを削除します。

```
dt = Open Database(
   "Connect Dialog",
   "SELECT age, sex, weight FROM \!"Bigclass$\!"",
   "My Big Class"
);
dt << Save Database( "Connect Dialog", "MY_BIG_CLASS" );
Open Database( "Connect Dialog", "DROP TABLE BIGCLASS.MY_BIG_CLASS" );
```

メモ: ODBCデータベースからデータを読み込む際、ユーザIDとパスワード情報を含んだテーブル変数が追加される可能性があります。これを防ぐためには、環境設定の`pref(ODBC Hide Connection String(1))`を設定します。または、[**ファイル**]メニュー（Windows）または[**JMP**]メニュー（Macintosh）を選択し、[**環境設定**] > [**テーブル**]で[**ODBC接続文字列を非表示にする**]を選択します。詳細については、『JMPの使用法』の「JMPの環境設定」章を参照してください。

データベース接続の確立とSQLの実行

以下の関数を使用すると、データベースに対して、より複雑な処理を行えます。

```
db = Create Database Connection( "パスワードを使用した接続文字列" );
Execute SQL( db, "SQL statement", <invisible>, <"New Table Title"> );
Close Database Connection( db );
```

これら3つの関数を使用すれば、接続を開き、`Execute SQL`を数回呼び出した後、接続を閉じることができます。`Create Database Connection`は、`Execute SQL`と`Close Database Connection`で使用できるハンドルを戻します。

送信されたSQL文によって、データベースのテーブルから、JMPのデータテーブルが作成される場合と、作成されない場合があります。「SELECT」は通常、JMPのデータテーブルを作成します。しかし、「INSERT INTO」は、データベース内のテーブルを変更するものなので、JMPのデータテーブルは作成されません。

例

次のプログラムは、データベースへの接続を開きます。

```
dbc = Create Database Connection(
   "DSN=dBASE Files;DBQ=$SAMPLE_IMPORT_DATA/;"
);
```

次のプログラムは、上記の接続を使って、SQLステートメントを実行します。

```
dt = Execute SQL( dbc,
   "SELECT HEIGHT, WEIGHT FROM Bigclass", "NewTable"
);
```

次のプログラムは、処理が完了したら、ODBC接続を閉じます。

```
Close Database Connection( dbc );
```

メモ: `Execute SQL()`で接続文字列を作成すると、生成されるデータテーブルの中にクリアテキストのパスワードが含まれます。クエリービルダーでSQLクエリーを作成し、そのクエリーをJSLで実行する方法を推奨します。そうすると、パスワードがデータテーブルに含まれません。

SQL クエリーの記述

Query() 関数を使うと、SQL ステートメントで JMP データテーブルを操作できます。SQL ステートメントは、Query() で唯一必須の引数です。通常、SQL ステートメントの多くはデータテーブルを参照します。SQL ステートメントが参照するデータテーブルは、**テーブル参照**の引数を使って渡されます。

テーブル参照の引数として、データテーブル参照を使用できます（次の例の dt）。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
result = Query( dt, "SELECT * FROM 'Big Class' WHERE 年齢 > 14;" );
```

上の例の SQL ステートメントは、JMP データテーブルの名前を含んでいます。データテーブルのパスが長く、SQL ステートメントで別名を使用したい場合があります。別名を使用するには、テーブル参照を Table() 引数として渡します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
result = Query( Table( dt, "t1" ), "SELECT * FROM t1 WHERE 年齢 > 14;" );
```

Table() でデータテーブルの別名を使うことで、複数のクエリーで名前の異なるテーブルが使用されている場合でも SQL ステートメントを書き換える必要がなくなります。

プライベートデータテーブルと非表示のデータテーブル

Query() で作成されるデータテーブルをプライベートまたは非表示にしたい場合は、Private または Invisible を Query() の引数として渡します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
result = Query( dt, invisible, "SELECT * FROM 'Big Class' WHERE 年齢 > 14;" );
```

単一の値を戻す

Query() 関数は、通常、クエリーの結果として JMP データテーブルを戻します。しかし、単一の値を戻す SQL クエリーを記述した場合は、その値が入った JMP データテーブルが戻されるより、値だけが戻された方が便利です。値だけが戻されるようにするには、Scalar 引数を渡します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
result = Query(
   Table( dt, "t1" ),
   Scalar,
   "SELECT AVG([身長（インチ）]) FROM t1 WHERE 年齢 > 14;"
);
```

上のクエリーは、15 歳以上の学生の平均身長である 65（インチ）を戻します。

SASの使用

SAS DATAステップの作成

JMPのデータテーブルにMake SAS Data Stepを送ると、SASプログラムが作成されます。このSASプログラムは、「DATAステップ」と呼ばれるものであり、SASシステム上で実行すると、JMPデータテーブルと同じデータの、SASデータセットが作成されます。例:

```
dt << Make SAS Data Step
```

実行すると、SASプログラムエディタで使用できるSASプログラムがログに出力されます。

Make SAS Data Step Windowを送ると、SASプログラムが、別のウィンドウに出力されます。また、プログラムのファイル名には、「.SAS」という拡張子が付けられますので、SASシステムで簡単に利用できます。

計算式を持つ列のSAS DATAステップコードの作成

JMPのデータテーブルに、Get SAS Data Step for Formula Columnsというメッセージを送ると、列の計算式に対応したSASプログラムが作成されます。次の例は、まず、計算式を含む「rate」列を作成し、そのSASプログラムを生成します。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << New Column( "rate", Formula( :Name("身長(インチ)") / :Name("体重(ポンド
  )")));
dt << Get SAS Data Step for Formula Columns;
```

このスクリプトは、次のようなコードをログに出力します。なお、列名における日本語などの文字は、標準的なSASの変数名には使えないため、アンダーバーに置き換えられます。

```
/*%PRODUCER: JMP - DataTable Formulas */
/*%TARGET: rate */
/*%INPUT: __ */
/*%INPUT: __2 */
/*%OUTPUT: rate */
/* Code to score rate */
rate =__/__2
drop ;
```

引数に列名を指定すると、その列の計算式だけが対象となりますが、次のように列名を省略すると、すべての計算式のコードが生成されます。

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );
dt << Get SAS Data Step for Formula Columns;
```

SAS Model Managerのスコアリングコードに列計算式を含めることもできます。データテーブルにGet MM SAS Data Step for Formula Columnsを送ります。

```
dt = Open( "$SAMPLE_DATA/Tiretread.jmp" );
```

```
dt << Get MM SAS Data Step for Formula Columns;
```

このスクリプトの結果もログに表示されます。

Get SAS Data Step for Formula Columnsの場合と同様、列名の指定はオプションです。

SAS変数名

SAS Open For Var Names( )は、SASデータセットから変数名を取得し、それらを文字列のリストで戻します。

SASの変数名には、JMPのものよりも細かい制約があります。SAS Name関数は、JMP変数名をSAS変数名に変換します。その際、特殊文字・空白・日本語を下線に変更するなどのさまざまな変換を行って、有効なSAS変数名を生成します。

```
result = SAS Name(name);
result = SAS Name({list of names});
```

文字列を含んだリストを引数に指定した場合は、各名前をスペースで区切った文字列が戻されます。例:

```
SAS Name({"x 1", "x 2"})
```

は、次の文字列を戻します。

```
" x_1 x_2"
```

SASマクロ変数の値の取得

JMPには、SASマクロ変数を問い合わせるいくつかの方法が用意されています。

たとえば、次のプログラムは、SYSTIMEマクロ変数の値を取得します。

```
systime = sas << Get Macro Var("SYSTIME");
show(systime);
```

次のプログラムは、現在、定義されているすべてのSASマクロ変数を表示します。

```
macro_names = sas << Get Macro Var Names();
show(macro_names);
```

次のプログラムは、すべてのマクロ変数の名前と値を、反復処理によって表示します。

```
macro_names = sas << Get Macro Var Names();
For( i = 1, i <= N Items( macro_names ), i++,
   macro_value = sas << Get Macro Var( macro_names[i] );
   output = macro_names[i] || " " || Char( macro_value );
   Show( output );
);
```

次のプログラムは、「test」という名前でSASマクロ変数を定義するSASコードを実行した後、その値を取得します。

```
sas << Submit( "%let test = 1;" );
test = sas << Get Macro Var( "test" );
Show( test );
```

どのマクロ変数も、可能であれば数値として評価され、可能でない場合は文字となります。

SAS Metadata Serverへの接続

JMPからSASサーバーに接続し、SASデータセットを処理できます。なお、これらの一連の操作は、JSLでもプログラミングできますが、メニューでも用意されています。詳細については、『JMPの使用法』の「データの読み込み」章を参照してください。

接続

まず、Meta Connect()コマンドでSAS Metadata Serverに接続します。

```
connected = Meta Connect( "MyMetadataServer", port );
```

メタデータサーバーのSASのバージョンを指定するには、SASVersion名前付き引数を使用します。

```
connected = Meta Connect( "MyMetadataServer", port, SASVersion("9.4") );
```

コンピュータの名前（myserver.mycompany.comなど）とポートだけを指定した場合、認証ドメイン、ユーザ名、およびパスワードの指定を促すプロンプトが表示されます。これは、JSLで指定することもできます。

```
connected = Meta Connect( "MyMetadataServer", port, "authdomain", "user name",
  "password" );
```

SAS Metadata Serverの使用が終わったら、Meta Disconnect( )で接続を解除します。引数は必要ではなく、このコマンドで現在のメタサーバー接続が閉じられます。

メタデータサーバー上で使用可能なリポジトリを確認し、使用したいリポジトリを設定できます。

```
Meta Get Repositories();
  {"Foundation"}
Meta Set Repository( "Foundation" );
```

使用できるリポジトリが1つしかない場合は、自動的にそれが選択されるため、特に設定する必要はありません。

リポジトリを設定すると、使用可能なサーバーを確認することができます。

```
mylist = Meta Get Servers();
  {"SASMain", "Schroedl", "SASMain_ja", "SASMain_zh", "SASMain_ko",
    "SASMain_fr", "SASMain_de", "SASMain_Unicode"}
```

次に、SAS接続を設定します。このコマンドを使用すると、メタデータサーバーを使用せずに、ローカルまたはリモートサーバーに直接接続できます。

```
conn = SAS Connect( "SASMain" );
```

ここで、connオブジェクトにDisconnectおよびConnectメッセージを送信して、SAS接続を閉じる／開くことができます。

```
conn << Disconnect();
conn << Connect();
```

上記のプログラムは、メタデータのオブジェクトにメッセージを送って、SASに接続しています。JSLでは、関数によって、SAS接続をグローバルに確立することもできます。その場合、メタデータのオブジェクトにDisconnectやConnectメッセージを送っても、グローバルなSAS接続には影響しません。ただし、グローバルなSAS接続がない場合は、メタデータのオブジェクトから開かれたSAS接続が、グローバルなSAS接続に設定されます。

SASライブラリに自動的に接続する

SASサーバーに接続する際に、メタデータで定義されたSASライブラリに自動的に接続するには、Connect Librariesを使用します。

```
conn = SAS Connect( "SASMain", Connect Libraries( 1 ) );
```

メタデータで定義されたすべてのライブラリに接続されるため、接続に時間がかかる場合があります。

特定のライブラリに後で接続するには、SAS Connect Libref関数を使用するか、またはSASサーバーオブジェクトに対してConnect Librefメッセージを送ります。

SASライブラリの表示

SASサーバーに接続し、サーバー上のライブラリを見るにはGet Lib Ref()コマンドを使用します。

```
conn << Get Lib Refs();
  {"BOOKS", "EGSAMP", "GENOMICS", "GISMAPS", "JMPSAMP", "JMPTEST",
  "MAILLIB", "MAPS", "OR_GEN", "ORION_RE", "ORSTAR", "SASHELP",
  "SASUSER", "TEMPDATA", "TSERIES", "V6LIB", "WORK", "WRSTEMP"}
```

必要なデータセットを含んだライブラリが割り当てられていない場合は、そのライブラリを割り当てます。

```
conn << Assign Lib Refs( "MyLib", "$DOCUMENTS\data" );
```

SASデータセットを開く

まず、SASライブラリ参照名を割り当てます。

```
conn << Assign Lib Refs( "MyLib", "$DOCUMENTS" );
```

第1引数は、任意のSASライブラリ参照名です。第2引数は、データセットがあるサーバ上のパスです。

次に、選択したライブラリの配下にあるデータセットの名前を、リストで取得します。

```
datasets << Get Data Sets( "MyLib" );
  {"ANDORRA", "ANDORRA2", "ANYVARNAME", "BOOKS", "BOOKSCOPYNOT", "BOOKS_VIEW",
    "CATEGORIES", "DATETIMETESTS", "MOREUGLY", "NOTTOOUGLY", "PAYPERVIEW",
    "PUBLISHERS", "PURCHASES", "PURCHASES_FULL",
```

これで、データセットを開く準備ができました。

```
conn << Import Data( "MyLib", "PURCHASES" );
```

または

```
conn << Import Data( librefs[1], datasets[12] );
```

または

```
conn << Import Data( "MyLib.PURCHASES" );
```

これで、ライブラリ内にあるデータセットについての情報を取得できます。次のプログラムは変数名をリストで取得できます。

```
conn << Get Var Names( "MyLib.PURCHASES" );
  {"purchaseyear", "purchasemonth", "purchaseday", "bookid", "catid",
  "pubid", "price", "cost"}
```

この情報を元に、読み込む変数を指定し、データセットの一部だけを読み込むこともできます。

```
conn << Import Data( librefs[1], datasets[12], columns( bookvars[1], bookvars[2],
   bookvars[4] ) );
```

SASデータセットの保存

JMPデータテーブルを、SASデータセットに保存するには、SAS Export Data()コマンドを使用します。

```
conn << Export Data( dt, librefs[1], datasets[4], ReplaceExisting );
```

ストアドプロセスの実行

ストアドプロセスへの参照を取得するには、次の関数を使用します。

```
stp = Meta Get Stored Process( "Samples/Stored Processes/Sample: Hello World" );
```

JSLでストアドプロセスのリストを取得する方法はありません。実行したいストアドプロセスへのパスを知っている必要があります。

ストアドプロセスを実行するには、ストアドプロセスにメッセージを送ります。

```
stp << Run();
```

JMPからのSASコードのサブミット

SASプログラムを直接サブミットして、その実行結果を取得することもできます。例:

```
conn << Submit( "proc print data=sashelp.class; run;" );
```

オプションの2つの引数により、SASのアウトプットやログをJMP上に表示するかどうかを指定します。

```
conn << Submit( "SAS Code" <,No Output Window(True|False)> <,Get Log(True|False )>
   );
```

また、SASのログは、次のコマンドによりいつでも得ることができます。

```
conn << Get Log();
```

Get Log()は、SASのログを文字列として戻します。この文字列は、JSLにおける通常の文字列と同じように、JSL変数に代入して使用することができます。

環境設定

現在のSASバージョンの環境設定を取得するには、次のコマンドを使用します。

```
Get SAS Version Preference();
```

現在のSASバージョンの環境設定を設定するには、次のコマンドを使用します。

```
Preference( SAS Integration Settings( SASVersion( "9.4" ) ) );
```

サンプルスクリプト

JMPのSamples/Scripts/SAS Integrationフォルダに、サンプルスクリプトが含まれています。ストアドプロセススクリプトを正常に実行するには、ストアドプロセスがお使いのSAS Metadata Server上になければなりません。ストアドプロセスは、同じフォルダのsampleStoredProcesses.spkファイルに入っています。

sampleStoredProcesses.spkをお使いのSAS Metadata Serverに読み込むには

注意：これらのストアドプロセスは、実務用のシステムではなく、テスト用のSAS Metadata Serverに読み込んでください。

1. SAS Management Console（SAS管理コンソール）を実行します。
2. 管理者権限のあるアカウントを使用してSAS Metadata Serverに接続します。
3. SAS管理コンソールの左ペインにある［BI Manager］ノードを展開します。
4. ツリー内で、読み込んだサンプルストアドプロセスの保存先とするフォルダに移動します。
5. SAS 管理コンソールの左ペインまたは右ペインでそのフォルダを右クリックし、［**インポート**］を選択します。

   読み込みウィザードが表示されます。
6. sampleStoredProcesses.spkへのフルパスを入力するか、または［**参照**］ボタンを使って指定します。
7. ウィザードの読み込みオプションで、［**すべてのオブジェクト**］を選択します。
8. ［**Next**］をクリックします。

   読み込みプロセス中、パネルに、**Application Server**と**ソースコードリポジトリ**の値を指定する必要があることが表示されます。
9. ［**Next**］をクリックします。

   パネルで、読み込んだストアドプロセスを実行するのに使うアプリケーションサーバーを、SAS Metadata Serverで定義されたものの中から選択します。

10.「ターゲット」の下のドロップダウンリストからアプリケーションサーバーを選択します。

11.［**Next**］をクリックします。

パネルで、読み込んだストアドプロセスのSASコードを格納するためのソースコードリポジトリ（ディレクトリ）を、SAS Metadata Serverで定義されたものの中から選択します。

12.「ターゲットパス」の下のドロップダウンリストからソースコードリポジトリを選択します。

13.［**Next**］をクリックします。

パネルに、［**実行**］をクリックした場合に実行される処理の概要が表示されます。

14. 表示された情報を確認し、問題がなければ［**実行**］をクリックします。

15. 読み込みプロセスの途中で、メタデータサーバーへの接続に必要なログイン情報の入力を求められる場合があります。管理者権限のあるログイン情報を入力し、［**OK**］をクリックします。

読み込みが完了したら、ストアドプロセスを読み込んだフォルダの下に「BIP Tree」というフォルダが表示されます。「BIP Tree」の下には「JMP Samples」というフォルダが表示されます。「JMP Samples」フォルダには、「Shoe Chart」と「Diameter」の2つのサンプルストアドプロセスが含まれています。

サンプルスクリプトstoredProcessHTML.jslおよびstoredProcessJSL.jslでは、サンプルストアドプロセスを読み込んだフォルダと一致するように、サンプルストアドプロセスへのパスを調整する必要があります。そうしないと、これらのスクリプトは正常に機能しません。

MATLABの使用

MathWorks Inc.のMATLABは、インタラクティブな作業環境で計算モデルを分析、視覚化できる製品です。MATLABは、Windows（32ビット、64ビット）、Macintosh OS X、Linux（64ビット）で使用可能です。Windows版とMacintosh版のJMPは、いずれもMATLABへの接続をサポートしています。

JMPスクリプト言語（JSL）を使って、次のようにMATLABと連携することができます。

- JSLスクリプト内からMATLABにステートメントをサブミットする
- JMPとMATLABの間でデータをやり取りする
- MATLABで作成されたグラフを表示する

JMPでMATLAB関数を使用する方法については、『スクリプト構文リファレンス』の「JSL関数」章を参照してください。

MATLABのテキスト出力やエラーメッセージは、ログウィンドウに表示されます。

MATLABのインストール

MATLABはJMPと同じコンピュータにインストールされている必要があります。

JMPをWindowsの32ビット版と64ビット版のどちらで使用しているかに応じて、該当するバージョンのMATLABをインストールしてください。サポートされているMATLABのバージョンについては、JMP Webサイト（http://www.jmp.com/system/）を参照してください。

JMPによるMATLABの検出方法

JSLスクリプトからMATLABへの接続命令が出されるまで、JMPはMATLABを起動しません。MATLABを呼び出すJSLスクリプトを実行するとき、JMPは、オペレーティングシステムのPATH環境変数に基づいてソフトウェアの場所を特定します（たとえば、C:¥Program Files¥MATLAB¥R2012a¥）。

インストールのテスト

お使いのコンピュータで、JSLベースのスクリプトを使ったMATLABの操作が可能かどうかは、次のJSLスクリプトで検証できます。

1. 次のJSLスクリプトを実行します。

```
MATLAB Init();
MATLAB Submit( "m = magic(3)" );
magicMat = MATLAB Get( m );
Show( magicMat );
MATLAB Term();
```

MATLAB関数M = magic(3)は、1～3^2の整数を使った3x3行列を戻します。この行列は、行の和と列の和が等しく、「**魔方陣**」と呼ばれます。

2. ［**表示**］>［**ログ**］を選択します。

ログウィンドウには次のような応答が表示されます。

```
m =

     8     1     6
     3     5     7
     4     9     2


magicMat =
[  8 1 6,
   3 5 7,
   4 9 2];
0
```

もしログウィンドウに次のようなメッセージが表示された場合は、その下の操作を行ってください。

```
システム上にMATLABがインストールされていません。
```

1. MATLABROOTという名前の新しい環境変数を追加し、値をC:¥Program Files¥MATLAB¥R2012a¥ またはC:\Program Files (x86)¥MATLAB¥R2012a¥に設定します。

メモ：入力するパスは、MATLABのインストール先のパスによって異なります。

2. MATLABのパスがPATH変数に含まれていることを確認します。
3. スクリプトを再実行し、JMPからMATLABにアクセスできるかどうかを検証します。

Rの操作

JSLを使ってできるRの操作は次のとおりです。

- JSLスクリプト内からRコードをRにサブミットします。
- JMPとRの間でデータをやり取りします。
- Rで作成したグラフを表示します。

Rのテキスト出力やエラーメッセージは、ログウィンドウに表示されます。

Rのインストール

RはJMPと同じコンピュータにインストールされている必要があります。RはComprehensive R Archive NetworkのWebサイトからダウンロードできます。

http://cran.r-project.org

JMPをWindowsの32ビット版と64ビット版のどちらで使用しているかに応じて、該当するバージョンのRをインストールしてください。サポートされているRのバージョンについては、次のJMP Webサイトのシステム要件を参照してください。http://www.jmp.com/support/system_requirements_jmp.shtml

デフォルトのRインストールディレクトリを変更する

R_HOMEがWindowsのシステムレジストリに定義されていない場合、通常、JMPはR_HOMEを次のようにみなします。

64ビット版Rのインストール場所
Computer¥HKEY_LOCAL_MACHINE¥SOFTWARE¥R-code¥R¥InstallPath

32ビット版Rのインストール場所
Computer¥HKEY_LOCAL_MACHINE¥SOFTWARE¥Wow6432Node¥R-code¥R¥InstallPath

デフォルトのR インストールの場所を変更するには、次のいずれかの方法でR_HOME 環境変数を定義します。

1. コントロールパネルを使って、システム環境変数内に変数を作成します。それには、まず、［**スタート**］>［**コントロールパネル**］>［**システム**］>［**システムの詳細設定**］を選択します。
2. ［**環境変数**］をクリックします。
3. システム環境変数のペインで［**新規**］をクリックします。
4. ［**変数名**］にR_HOMEとタイプします。
5. R .exeファイルのパスを入力します（C:¥Program Files¥R¥R-2.15.3など）。

6. ［OK］をクリックした後、再度［OK］をクリックし、システムプロパティのウィンドウを閉じます。

または

次のように、JSLのSet Environment Variable()関数を使って環境変数を作成します。

```
Set Environment Variable( "R_HOME", "C:\Program Files\R\R-2.15.3" );
```

JMPによるRの検出方法

JSLスクリプトからRへの接続命令が出されるまで、JMPはRを起動しません。JMPは、Rをロードする必要が生じたときに次のような順序でWindowsコンピュータ上のRを探します。

1. 環境変数R_HOMEを検索します。

   見つかった場合、指定のディレクトリからRをロードします。

2. 環境変数R_HOMEが存在しない場合は、Windowsレジストリ内で次のキーの下にあるInstallPath値を調べます。

   HKEY_LOCAL_MACHINE¥SOFTWARE¥R-core¥R
   64ビットのコンピュータで32ビット版のJMPを実行している場合、InstallPath値は次のキーにあります。

   HKEY_LOCAL_MACHINE¥SOFTWARE¥WOW6432NODE¥R-core¥R
   InstallPath値が存在する場合、指定のディレクトリからRをロードします。

3. InstallPath値が存在しない場合は、Rが見つからないことを示すエラーメッセージが表示されます。

セットアップのテスト

お使いのコンピュータで、JSLベースのスクリプトを使ったRの操作が可能かどうかは、次のJSLスクリプトでテストできます。

```
R Init( );
R Submit( "
   x <- 1:5
   x
" );
R Term( );
```

ログには次のような出力が表示されます。

```
[1] 1 2 3 4 5
```

JMPからRへのインターフェース

JMPには、Rへのインターフェースが関数として用意されています。基本的には、まず、Rへの接続を開始し、次に、何らかの処理をR上で実行し、最後にRの接続を解除します。これらの関数は、Rの処理が正常に実行された場合は0、そうでない場合は、エラーコードを戻すのが普通です。Rの処理が正常に実行されなかった場合は、ログにメッセージが表示されます。ただし、R Get( )関数だけは、エラーコード以外の値を戻します。

RのJSLスクリプト可能なオブジェクトインターフェース

R接続の機能は、JSL関数としてだけではなく、オブジェクトへのメッセージとしても用意されています。R Connect()というJSL関数によって、R接続オブジェクトへの参照を取得できます。R接続オブジェクトに対して、以下で述べるメッセージを送ることができます。

JMPデータタイプとRデータタイプの相互変換

表14.3に、R Send( )関数でJMPからRに変数を送った場合に、JMPのデータタイプ（データ型）が、Rにおいて、どのような型に変換されるかを示します。リストの場合は、リスト内の要素ごとにデータタイプをチェックして、変換します。なお、入れ子になっているリストも、サポートされています。

表14.3 R Send( )でのJMPデータタイプとRデータタイプの対応

JMPデータタイプ	Rデータタイプ
数値	実数（double）
文字列	文字列
行列	実数の行列
リスト	リスト
データテーブル	データフレーム
行の属性	整数
日付時間	日付と時間
時間差	時間差

例

```
R Init();
X = 1;
R Send( X );
S = "Report Title";
R Send( S );
M = [1 2 3, 4 5 6, 7 8 9];
R Send( M );
```

```
R Submit( "
X
S
M
" );
R Term();
```

表14.4に、R Get( )関数でRからJMPに変数を取得した場合に、Rのデータタイプ（データ型）が、JMPにおいて、どのような型に変換されるかを示します。リストの場合は、リスト内の要素ごとにデータタイプをチェックして、変換します。なお、入れ子になっているリストも、サポートされています。

表14.4 R Get( ) でのJMPデータタイプとRデータタイプの対応

Rデータタイプ	JMPデータタイプ
実数（double）	数値
論理（ブール値）	数値 (0 | 1)
文字列	文字列
整数	数値
日付と時間	日付時間
時間差	時間差
因子（factor）	文字列のリスト、または、数値の行列
データフレーム	データテーブル
リスト	リストや行列を含んだリスト
行列	行列
数値ベクトル	行列
文字列ベクトル	文字列のリスト
グラフ	ピクチャー
Time Series（時系列）	行列

JMPスコープ演算子とR

R Send() 関数によって、JMPの変数をRに送った場合、Rオブジェクトの名前には、JMPの変数と同じ名前が付けられます。たとえば、dtというJMP変数をRに送ると、dtという名前のRオブジェクトが作成されます。コロンおよび2重コロンのスコープ演算子（:および::）は、Rオブジェクト名に使用できないため、次のように変換されます。

- 1重コロンのスコープ演算子はピリオド（.）に置き換えられます。

  たとえば、nsref:dtをRに送ると、nsref.dtという名前のRオブジェクトが作成されます。

- 2重コロンのスコープ演算子（グローバル変数を指定）は無視されます。

  たとえば、::dtをRに送ると、dtという名前のRオブジェクトが作成されます。

R Send() でのR Name() の使用

R Send() のR Name() オプションには、有効なRオブジェクト名を引用符付き文字列で指定します。Rに送られたJMPオブジェクトは、指定の名前のRオブジェクトになります。例:

```
R Send( jmp_var_name, R Name( "r_var_name") );
R Submit( "print(r_var_name)" )
```

例

次の例は、Here名前空間内に変数x、グローバル名前空間内に変数y、そしてどの名前空間にも明示的に参照されない変数zを作成します。変数zは、Names Default To Here(1)がオンでない限り、デフォルトでグローバルに設定されます。これらの変数は、その後、Rに渡されます。

```
Here:x = 1;
::y = 2;
z = 3;

R Init(); // R接続を開始する

R Send( Here:x );
/* Here変数をRに送る
Here:xはRオブジェクトのHere.xとなる */
R Submit( "print(Here.x)" );
/* JMPログの出力では、元のJMP変数の参照であるHere:xが使用される */

R Send( ::y ); // ::yはRオブジェクトのyとなる
R Submit( "print(y)" );

R Send( Here:x, R Name( "localx" ) );
// Rオブジェクトに別の名前をつけるには、R Name() オプションを使用する
R Submit( "print(localx)" );
/* R Send() コマンドをR Nameオプションとともに使用すれば、JMP変数"Here:x"に相当するRオブジェクト"localx"が作成できる。この場合も、ログには元のJMP変数名が表示される */

R Send( z ); // zはRオブジェクトzとなる
R Submit( "print(z)" );
```

トラブルシューティング

グラフの記録

Windows版のRにおいて、作成したグラフをグラフウィンドウに記録していくには、R Submit( )関数にて、次のRコードを実行してください。

```
windows.options( record = TRUE );
```

文字ベクトル

JMPにおける文字列のリストは、Rの文字ベクトルと同じではありません。文字列のリストをJMPからRに送った場合、それは、文字ベクトルではなく、文字列のリストに変換されます。これを文字ベクトルにするには、R関数のUnlistを使用してください。

```
R Init();
X = {"Character", "JMP", "List"};
R Send( X );
R Submit( "class(X)" );
/* Rの出力は：
[1] "list"
*/

R Submit( "Y<-unlist(X)
     class(Y)" );
/* これでオブジェクトYは文字ベクトルとなった。Rの出力は：
[1] "character"
*/

R Term();
```

要素の名前

Rのリストは、属性をもつことができ、リストの各要素に対して、特定の名前を与えることができます。属性をもつリストは、リスト内での通し番号がわからなくても、名前によって、要素にアクセスできます。

次の例では、Rにおいて、List() 関数でリストを作成しています。リストでは、x と y という属性が要素に与えられています。このようなリストを、JMP で取得し、再度 R に戻した場合、属性が失われます。そのため、戻されたリストに対しては、「pts$x」といった指定によっては、要素にアクセスできなくなります。代わりに、「pts[[1]]」というように通し番号でアクセスしてください。

```
R Init();
R Submit("
   pts <- list(x=cars[,1], y=cars[,2])
   summary(pts)
");

JMP_pts = R Get( pts );

R Send( JMP_pts );
R Submit("
   Summary( JMP_pts )
");
R Term();
```

例

Rにデータテーブルを送る

次のプログラム例は、R接続を開始し、データテーブルをRに送り、それをログに印刷し、最後にR接続を閉じます。

```
R Init();
dt = Open( "$SAMPLE_DATA/Big Class.jmp", invisible );
R Send( dt ); // dtという開いたデータテーブルをRに送る
R Submit( "print( dt )" );
R Term();
```

Rでのオブジェクトの作成

次のプログラム例は、R接続を開始し、Rオブジェクトを作成し、そのオブジェクトをJMPで取得し、最後にR接続を閉じます。

```
R Init();
R Submit(
   "
   L3 <- LETTERS[1:3]
   d <- data.frame(cbind(x=1, y=1:15), Group=sample(L3, 15, repl=TRUE))
"
);
R Get( d ) << NewDataView;
R Term();
```

Rの関数とグラフの使用

次のプログラム例は、R接続を開始し、Rにおいて正規密度関数のグラフを描きます。次に、そのグラフのイメージをRから取得して、JMPで表示します。最後に、R接続を閉じます。

```
R Init();
R Submit( "\[plot(function(x) dnorm(x), -5, 5, main = "Normal(0,1) Density") ]\" );
picture = R Get Graphics( "png" );
New Window( "Picture", picture );
Wait( 10 );
R Term();
```

R内で簡単な行列を追加する

次のプログラム例は、R接続を開始し、まず、1つの行列をJMPからRに送ります。次に、Rで、もう1つの行列を作成します。そして、この2つの行列を足し合わせた行列をJMPに送り、最後にR接続を閉じます。

```
R Init();
X = J( 2, 2, 1 );
R Send( X );
```

```
R Submit(
   "
   X                                    #X をログに印刷する
   Y <- matrix(1:4, nrow=2, byrow=TRUE)  #2x2 の行列オブジェクト Y を作成する
   Y                                    #Y をログに印刷する
   Z <- X + Y                           # 行列オブジェクト Z は X と Y を 1 つにしたもの
"
);
Z = R Get( Z );
R Term();
Show( Z );
```

ブートストラップの例

Rを用いたブートストラップの例として、サンプルスクリプトフォルダにあるJMPtoR_bootstrap.jslをご覧ください。

このスクリプトは、Rのライブラリを利用して、JMPにおいてブートストラップを実行します。

このスクリプトは、まずユーザに対し、分析対象の変数を指定するためのウィンドウを、JMPにおいて表示します。このウィンドウで、信頼区間を求めたい統計量も選択します。その後、JSLのRインターフェースを使って、データがRに送られます。

そして、Rのbootパッケージにおける`boot()`関数および`boot.ci()`関数によって、各ブートストラップ標本の統計量とブートストラップ信頼区間が計算されます。

結果はJMPに戻され、JMPの「一変量の分布」プラットフォームを使って表示されます。

Microsoft Excelの使用

JMP が提供している Microsoft Excel アドインのうち、「JMP でのプロファイル」の機能はスクリプトでも実行できます。しかし、データを転送する「JMP への転送」はスクリプトでは実行できません。「JMP でのプロファイル」の基本的な構文は次のとおりです。

```
excel_obj = Excel Profiler(
   Workbook( "excel_workbook_path" ),
   Model( "name_of_model" )
);
```

`Model`引数はオプションです。ワークブックのみを指定し、モデルの指定を省略した場合は、モデルを選択するよう促されます。モデルはワークブックのどこにあっても検出されるので、ワークシートを選択する必要はありません。

上記のような方法で作成したオブジェクトに、次のメッセージを送ると、予測プロファイルが描かれます。
例:

```
excel_obj <<Prediction Profiler( 1 );
```

XMLの解析

JSLには、XML解析用のコマンドがいくつか用意されています。

```
Parse XML( string, On Element( "tagname", Start Tag( expr ), End Tag( expr ) ) );
```

といった指定で、XMLタグにOn Element()式を使ったXML式を解析します。

```
value = XML Attr( "attribute name" );
```

は、Parse XML()式を評価する際にXML引数の文字列の値を抽出します。

```
value = XML Text();
```

は、Parse XML()式を評価する際にXMLタグのbodyの文字列テキストを抽出します。

XMLの解析の例

Microsoft ExcelファイルにBig Class.jmpのデータが1行含まれているとします。このファイルを、XMLドキュメントとして保存したものの内容は以下のようになります。このファイルは、BigclassExcel.xmlという名前で、JMPのSamples/Import Dataフォルダにも保存されています。

```
<?xml version="1.0" encoding="UTF-8"?>
<Workbook xmlns="urn:schemas-microsoft-com:office:spreadsheet"
 xmlns:o="urn:schemas-microsoft-com:office:office"
 xmlns:x="urn:schemas-microsoft-com:office:excel"
 xmlns:ss="urn:schemas-microsoft-com:office:spreadsheet"
 xmlns:html="http://www.w3.org/TR/REC-html40">
 <Worksheet ss:Name="Bigclass">
  <Table ss:ExpandedColumnCount="5" ss:ExpandedRowCount="41" x:FullColumns="1"
   x:FullRows="1">
   <Row>
    <Cell><Data ss:Type="String">name</Data></Cell>
    <Cell><Data ss:Type="String">age</Data></Cell>
    <Cell><Data ss:Type="String">sex</Data></Cell>
    <Cell><Data ss:Type="String">height</Data></Cell>
    <Cell><Data ss:Type="String">weight</Data></Cell>
   </Row>
   <Row>
    <Cell><Data ss:Type="String">KATIE</Data></Cell>
    <Cell><Data ss:Type="Number">12</Data></Cell>
    <Cell><Data ss:Type="String">F</Data></Cell>
    <Cell><Data ss:Type="Number">59</Data></Cell>
    <Cell><Data ss:Type="Number">95</Data></Cell>
   </Row>
  </Table>
 </Worksheet>
</Workbook>
```

次のスクリプトは、BigclassExcel.xmlを読み込み、その情報からJMPデータテーブルを作成します。このスクリプトは、ParseXML.jslという名前で、JMPのSamples/Scriptsフォルダに保存されています。

```
file contents = Load Text File( "$SAMPLE_IMPORT_DATA/BigclassExcel.xml" );
Parse XML( file contents,
   OnElement(
      "urn:schemas-microsoft-com:office:spreadsheet^Worksheet",
      StartTag(
         sheetname = XML Attr(
            "urn:schemas-microsoft-com:office:spreadsheet^Name",
            "Untitled"
         );
         dt = New Table( sheetname );
         row = 1;
         col = 1;
      )
   ),
   OnElement(
      "urn:schemas-microsoft-com:office:spreadsheet^Row",
      StartTag(
         If( row > 1, // 最初の行は列名として扱う
            dt << Add Rows( 1 )
         )
      ),
      EndTag(
         row++;
         col = 1;
      )
   ),
   OnElement( "urn:schemas-microsoft-com:office:spreadsheet^Cell", EndTag( col++ )
   ),
   OnElement(
      "urn:schemas-microsoft-com:office:spreadsheet^Data",
      EndTag(
         data = XML Text( collapse );
         If( row == 1,
            New Column( data, Character( 10 ) ),
            // 最初の行は列名として扱う
            Column( col )[row - 1] = data
         ); // その他の行はデータとして読み込む
      )
   )

);
```

OLEオートメーション

JMPの大部分はOLEオートメーションを使って操作することができます。JMPのオートメーションについては、「JMP¥13¥Documentation\ja」フォルダにある「Automation Reference.pdf」を参照してください。このドキュメントでは、Visual BasicおよびMFCを使ったVisual C++によってJMPを自動化する方法を説明しています。また、Visual BasicやVisual C++のようなオートメーションクライアントで利用できるJMPのメソッドやプロパティについて詳しく説明しています。

「Samples¥Automation」フォルダには、Visual Basic .Net、Visual C# .Netのサンプルファイルがあります。

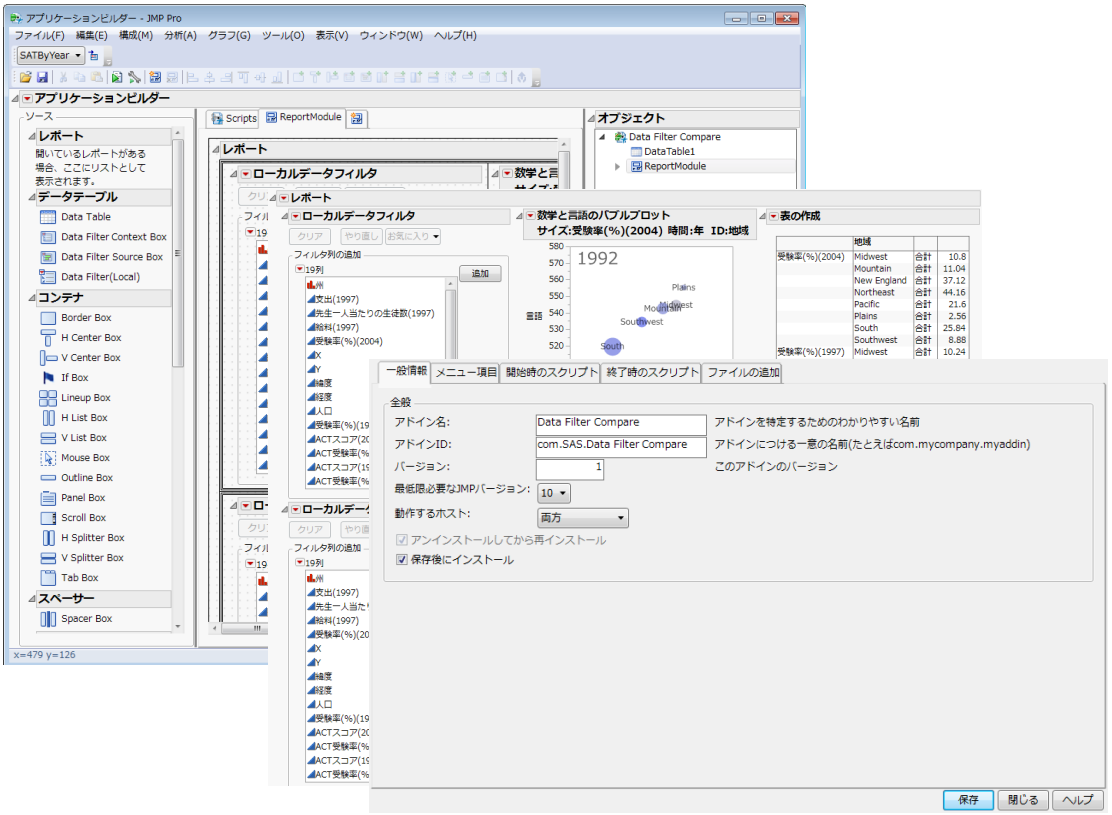
第 15 章

アプリケーションの作成
アプリケーションビルダー

JMP では、プラットフォームによる統計分析だけでなく、自分の目的に沿ったアプリケーションも作成できます。アプリケーションを作成して、ルーチン作業（たとえば、決められたデータに対し、「一変量の分布」と「モデルのあてはめ」を毎日実行するなど）を自動化できます。アプリケーションビルダーを使えば、複数の分析結果を、同一のウィンドウに配置できます。アプリケーションビルダーでは、ドラッグ＆ドロップ操作が行え、大量のスクリプトを書くことなしに、アプリケーションを作成できます。

アドインビルダーでは、JMP にインストールできるアドインを簡単に作成できます。アプリケーションビルダーでアプリケーションを作成し、アドインビルダーでアドインとして保存すれば、アプリケーションを簡単に作成および配布できます。

図15.1 アプリケーションビルダーでカスタマイズしたレポートを作成する



アプリケーションビルダーでアプリケーションを作成する

アプリケーションビルダーは、ボタンやグラフなどのオブジェクトを、ドラッグ&ドロップ操作によって、ウィンドウに配置できます。つまり、オブジェクトを配置するためのスクリプトを記述する必要はなく、各オブジェクトの機能を制御するスクリプトを記述するだけです。

また、JMPスクリプト言語（JSL）でのプログラミング経験が豊富なユーザは、アプリケーションビルダーによって、開発の生産性を高めることができます。アプリケーションビルダーでオブジェクトを配置し、自動作成されたスクリプトを編集します。独自の機能をスクリプトで記述することにより、目的に合ったアプリケーションを開発できます。

アプリケーションビルダーと、アドインビルダーを組み合わせて用いると良いでしょう。まず、アプリケーションビルダーで、特定の処理を行うスクリプトを作成します。続いて、アドインビルダーで、そのスクリプトをアドインにして配布すれば、ユーザは、ファイルを開いて実行するのではなく、メニューからスクリプトを実行できるようになります。詳細については、「JMPアドインビルダーを使ったアドインのコンパイル」（682ページ）を参照してください。

メモ: JMPのバージョンが新しくなるたびに、アプリケーションビルダーにもそれまでのバージョンになかった機能が新しく加わります。作成するアプリケーションに高い互換性を持たせるためには、アプリケーションを実行することになるJMPのうち最も古いバージョンを使い、アプリケーションを作成するようにしてください。

アプリケーション作成の例

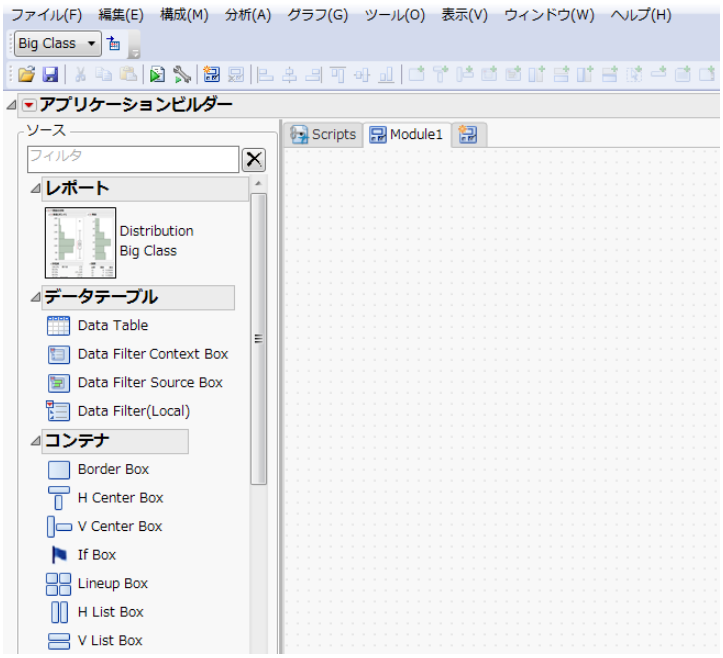
単純なアプリケーションであれば、アプリケーションビルダーで簡単に作成できます。単純なアプリケーションの1つには、レポートやグラフなどの複数のオブジェクトを、同一のウィンドウに配置したものがあります。

ここでは、「一変量の分布」レポートを表示するアプリケーションを作成します。この例では、起動ウィンドウで変数を選択しなくても、事前に指定された変数によって処理が実行されます。

1. ［**ヘルプ**］>［**サンプルデータライブラリ**］を選択し、「Big Class.jmp」を開きます。
2. **「一変量の分布」**テーブルスクリプトを実行してレポートを生成します。
3. ［**ファイル**］>［**新規作成**］>［**アプリケーション**］（Macintoshの場合は［**ファイル**］>［**新規**］>［**アプリケーションの新規作成**］）を選択します。
4. 「空白のアプリケーション」のテンプレートをクリックします。

   新しい空白のアプリケーションが表示されます。左側のペインに一変量の分布のレポートがあることがわかります。

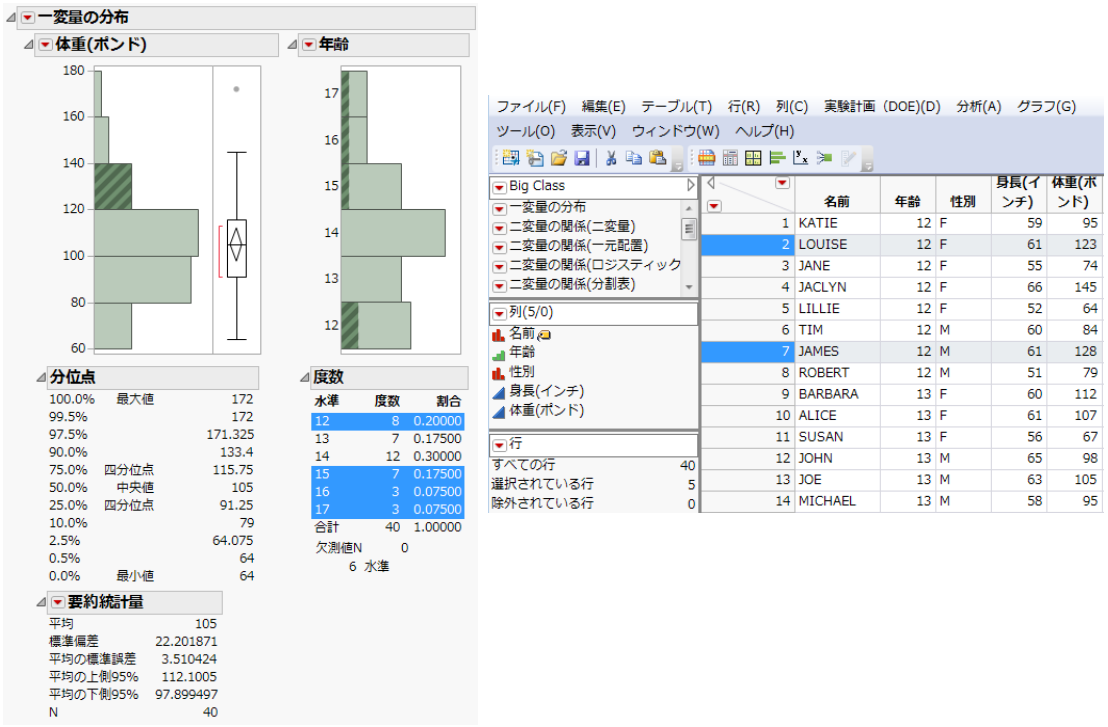
図15.2「ソース」に表示された「一変量の分布」レポート



5. 「ソース」ペインにある「一変量の分布」レポートをアプリケーションワークスペース（［Module1］タブの空白エリア）にドラッグします。
6. 「アプリケーションビルダー」の赤い三角ボタンメニューから、**［アプリケーションの実行］**を選択します。

   「一変量の分布」レポートが、新しいウィンドウに表示されます（図15.3）。ヒストグラムは、対話的で、データテーブルと関連付けられています。分析対象のデータが更新された場合は、アプリケーションを再実行すると、「一変量の分布」レポートも更新されます。

図15.3 単純なアプリケーションの例



アプリケーションビルダーの用語

アプリケーションは、1つ、もしくは、複数のモジュールで構成されます。**モジュール**には、コンパイルして1つのアプリケーションとなるオブジェクトとスクリプトが含まれます。アプリケーションを実行すると、モジュールごとに1つのウィンドウが表示されます。あるウィンドウから、別のウィンドウを呼び出すこともできます。たとえば、あるウィンドウに、「**グラフの作成**」といったボタンを配置し、そのボタンをクリックすると新しいウィンドウにグラフが表示されるようにできます。

アプリケーションビルダーを使い始める前に、いくつかの用語に慣れておく必要があります。自動作成されたスクリプトを編集する必要はほとんどありませんが、これらの用語を知っておくと、モジュールの動作を理解するのに役立ちます。

アプリケーション　1つまたは複数のモジュールで構成された最上位のファイル。

モジュール　オブジェクト、メッセージ、インスタンスなどのJSLステートメントを集めた、アプリケーションの構成要素。

単純なアプリケーション　レポートだけで構成されていて、プログラミングを全く必要としないアプリケーションのこと。単純なアプリケーション、つまり「ダッシュボード」は、ダッシュボードビルダーで作成することもできます。ダッシュボードビルダーも、アプリケーションビルダーと同様にドラッグ&ドロップで操作できますが、構成済みのディスプレイボックスが用意されている点が異なります。

カスタムアプリケーション　自分の目的に沿った処理を、JSL スクリプトによって記述したアプリケーションのこと。

オブジェクトとメッセージ　**オブジェクト**とは、JMP の動的なエンティティで、たとえばデータテーブル、データ列、プラットフォーム結果ウィンドウ、グラフなどのこと。ほとんどのオブジェクトは、何らかのアクションの実行を指示するメッセージを受け取ることができます。

メッセージとは、オブジェクトに送られる式のこと。オブジェクトは、送られたメッセージを評価し、何らかのアクションを実行します。

モジュールインスタンス　モジュールが出現したもの。複雑なアプリケーションの場合、起動時に動作するスクリプトににによって、複数のモジュールインスタンスを生成するといったこともできます。

名前空間　変数名が衝突せず、一意に決められるように定義された変数のまとまり。

アプリケーションビルダーで作成されたアプリケーションでは、「Application」と「ModuleInstance」という名前空間が自動的に作成されます。Application 名前空間内のシンボルは、アプリケーション内のスクリプトだけを範囲とし、アプリケーション外のスクリプトでは使用できません。名前空間の詳細については、「プログラミング手法」章の「高度な適用範囲指定と名前空間」（230ページ）を参照してください。

変数　変数は、固有の名前をもち、何らかの値を含んでいます。アプリケーションビルダーで作成されたアプリケーションを実行すると、「thisApplication」と「thisModuleInstance」という変数が自動的に作成されます。

- thisApplication 変数は、アプリケーションのオブジェクトへの参照を含みます。アプリケーションは、1つ、もしくは複数のモジュールで構成されています。
- thisModuleInstance 変数（ModuleInstance 名前空間で使用される）は、モジュールインスタンスのオブジェクトへの参照を含みます。モジュールは、ボックスやボタンなどで構成されています。

コンテナ　タブやアウトラインなどの、他のオブジェクトを含むことができるディスプレイボックスのこと。

アプリケーションの設計

アプリケーションを作成するにあたっては、まず目的を明確にし、どのようなグラフ要素を配置するのか、また、どの処理を JSL スクリプトでプログラミングする必要があるかを考えてください。

目的　アプリケーションで何を行うのかを明確にしてください。また、起動ウィンドウ、レポート、グラフなどから、カスタマイズの必要がある JMP 機能は何かを考えてください。

グラフ要素　まず、モジュールがいくつ必要かを決めてください。そして、モジュールごとに配置するオブジェクトを決定します。アプリケーションビルダーの「ソース」ペインには、モジュールにドラッグ＆ドロップできるボックスやアイコンが用意されています。また、用意されているサンプルを試してみて、自分の目的に近いモジュールがあるかどうかを確認してください。

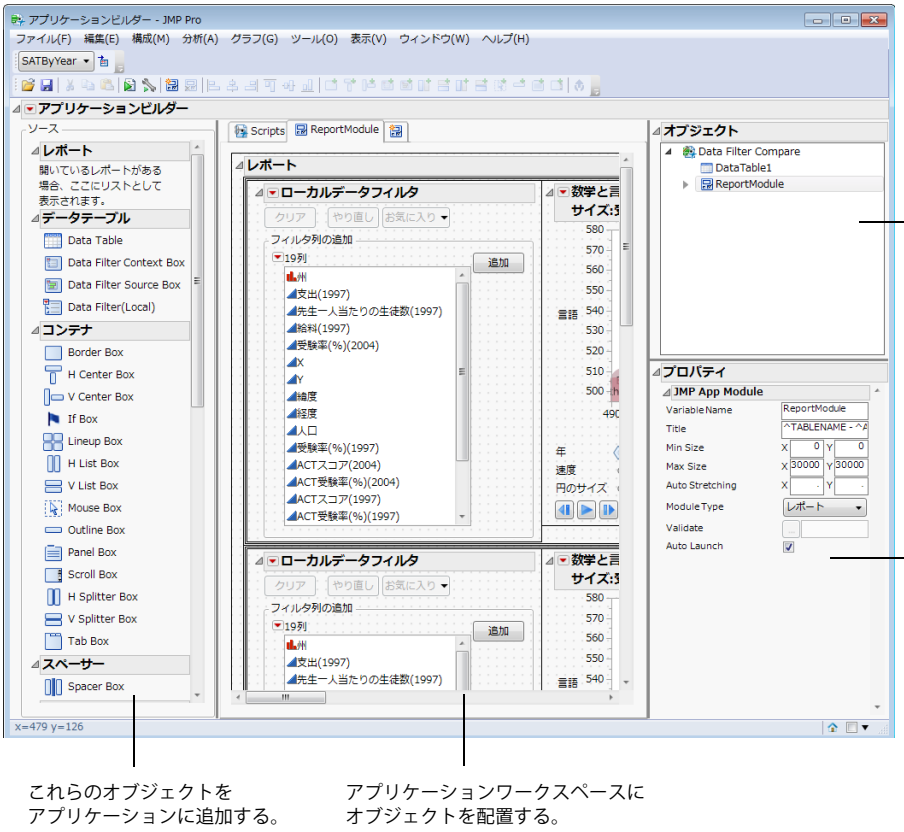
スクリプト　複雑なアプリケーションを作成するには、JSLスクリプトを記述します。アプリケーションのスクリプトの記述方法の詳細については、「スクリプトの記述」（665ページ）を参照してください。

非対話型で、オブジェクトを表示するだけの単純なアプリケーションであれば、スクリプトを記述する必要はありません。たとえば、JMPが標準的に生成するレポートを表示するだけならば、スクリプトを記述する必要はありません。

「アプリケーションビルダー」ウィンドウ

図15.4は、アプリケーションビルダーで作成中のアプリケーションです。

図15.4「アプリケーションビルダー」ウィンドウ



「アプリケーションビルダー」ウィンドウには、次の機能があります。

- ツールバーを使うと、オブジェクトの整列や一般的なディスプレイボックスの挿入など、数多くの機能をすばやく実行できます（ツールバーを表示するには［**表示**］>［**ツールバー**］>［**アプリケーションビルダー**］を選択します）。

- 「ソース」ペインには、アプリケーションに含めることができるオブジェクトが表示されています（ここには、開いたレポートやグラフなども表示されます）。オブジェクトとそのプロパティに関する情報を見るには、そのオブジェクトを右クリックして、[**スクリプトのヘルプ**]を選択してください。
- 中央は、ワークスペースです。点線のグリッド線が表示されており、オブジェクトを整列するのに便利です。ワークスペース内の［モジュール］タブにて、オブジェクトをクリックし、［スクリプト］タブにスクリプトを記述することにより、特定の機能をプログラミングできます。
- 「オブジェクト」ペインには、アプリケーションの階層が表示されます。モジュールと、各モジュールに含まれているオブジェクトが階層的に表示されます。オブジェクトをクリックすると、そのプロパティが表示され、また、ワークスペース上でオブジェクトが選択された状態になります。
- 「プロパティ」ペインでは、オブジェクトの位置や幅、名前などのプロパティを設定します。プロパティは、オブジェクトの種類によって異なります。

ヒント： 作成するすべてのアプリケーションにおいて、グリッドを隠したり、グリッドにスナップするオプションを無効にするには、［**ファイル**］>［**環境設定**］>［**プラットフォーム**］>［**JMP App**］を選択し、該当するオプションの選択を解除します。これらの機能は、アプリケーションビルダーの赤い三角ボタンのメニューで解除することもできます。

アプリケーションビルダーの赤い三角ボタンのオプション

アプリケーションビルダーの赤い三角ボタンのメニューには、アプリケーションの実行やデバッグ、サンプルアプリケーションを開く、グリッドを表示するなどのオプションがあります。

アプリケーションの実行　アプリケーションを開始します。各モジュールのウィンドウが開き、実際にユーザが操作するのと同様に、アプリケーションを対話的に操作できます。

アプリケーションのデバッグ　エラーを解決するために、JSLデバッガでスクリプトを開きます。詳細については、「スクリプト作成のツール」章の「スクリプトのデバッグ／プロファイル」（62ページ）を参照してください。

サンプルを開く　JMPの「Samples¥Apps」フォルダにあるサンプルアプリケーションの1つを開きます。用意されているサンプルは、一般的なアプリケーションを設定するための見本であり、必要に応じて変更することができます。表15.1は、サンプルについて説明したものです。

グリッドにスナップ　ワークスペースにオブジェクトをドラッグすると、オブジェクトが一番近いグリッド点線に沿って配置されます。デフォルトでオンになっています。

グリッドの表示　ワークスペースにグリッド点線を表示します。デフォルトでオンになっています。

ソースの表示　「ソース」パネルの表示／非表示を切り替えます。

オブジェクトとプロパティの表示　「オブジェクト」パネルと「プロパティ」パネルの表示／非表示を切り替えます。

自動スクロール　オブジェクトをワークスペースの端近くまでドラッグすると、縦または横に自動的にスクロールします。デフォルトでオンになっています。

スクリプトの保存 アプリケーションをデータテーブル、ジャーナル、スクリプトウィンドウ、またはアドインに保存します。詳細は、「アプリケーションの保存オプション」（669ページ）の節を参照してください。

表15.1 提供されているサンプルアプリケーション

Six Quality Graphs.jmpappsource	3つの管理図、「一変量の分布」レポート、「工程能力分析」レポートを作成する。
Data Filter Compare.jmpappsource	2つのバブルプロットを作成する。各プロットには、それぞれローカルデータフィルタと「表の作成」レポートがあります。このアプリケーションでは、`Data Filter Context Box`関数を用いています。
Data Table Application.jmpappsource	層化抽出を行う。層別変数の列と、標本抽出率を選択できます。抽出結果は、データテーブルの行が選択された状態になります。
Graph Launcher.jmpappsource	等式の入力、軸の設定、グラフの作成を可能にする。ウィンドウが開いたままなので、等式に変更を加えて新しいグラフを作成することができます。
Instant App.jmpappsource	主成分分析と多変量管理図のレポートを組み合わせて表示する。
Instant App Customized.jmpappsource	Instant App.jmpappsourceに変更を加えたもの。主成分分析レポートを選択するオプション、マーカーのサイズを変更するオプション、平均を表示するオプションが含まれています。
Launcher With Report.jmpappsource	起動ウィンドウでユーザにデータテーブル列を選択させ、グラフを作成する。
Parameterized Instant App.jmpappsource	ユーザにデータテーブル列を選択させ、2つの多変量のレポートを作成する。Yの役割に引数が割り当てられます。つまり、レポートは、アプリケーションで指定されているテーブルだけでなく、開いているどのデータテーブルからでも作成できます。
Parameterized Measurement Systems Analysis (MSA) Combo Chart.jmpappsource	測定システム分析（MSA）の一連のレポートを作成する。
Presentation.jmpappsource	ナビゲーションボタンと組み込みのスクリプトを持つ、スライドショーのようなプレゼンテーションを作成する。
R Application.jmpappsource	分析対象の列を JMP で選択させた後、R の機能を用いて「Chernoff の顔グラフ」を描く。このアプリケーションを実行するには、Rにおいて、TeachingDemosパッケージが必要です。
SAS Application.jmpappsource	SASスクリプトを実行して、結果をレポートに追加する。SASサーバーに接続していない場合は、SASサーバーにログオンするためのダイアログが表示されます。

アプリケーションの作成

アプリケーションの仕様を決めたら、空白のアプリケーションを作成し、オブジェクトとスクリプトを追加していきます。

ここでは、アプリケーションを作成するための基本的な手順を説明します。

- 「新しいアプリケーションの作成」（657ページ）
- 「オブジェクトの整列と削除」（659ページ）
- 「オブジェクトプロパティの変更」（662ページ）
- 「スクリプトの記述」（665ページ）

メモ: JMPのバージョンが新しくなるたびに、アプリケーションビルダーにもそれまでのバージョンになかった機能が新しく加わります。作成するアプリケーションに高い互換性を持たせるためには、アプリケーションを実行することになるJMPのうち最も古いバージョンを使い、アプリケーションを作成するようにしてください。

新しいアプリケーションの作成

空白のアプリケーションを新しく作成するには、次の手順に従います。

1. ［**ファイル**］>［**新規作成**］>［**アプリケーション**］（Macintoshの場合は［**ファイル**］>［**新規**］>［**アプリケーションの新規作成**］）を選択します。
2. 「空白のアプリケーション」を選択します。

   「アプリケーションビルダー」ウィンドウが開きます。
3. ［**ファイル**］>［**名前を付けて保存**］を選択し、ファイルをJMP ソースファイルとして保存します。拡張子は.jmpappsourceです。

空白のアプリケーションを作成した後、開いているレポートやディスプレイボックスその他のオブジェクトをワークスペースに追加します。詳細については、「アプリケーション作成の例」（650ページ）を参照してください。

モジュールの管理

新しいアプリケーションを作成すると、［Module1］というモジュールのタブがデフォルトで表示されます。このモジュールにオブジェクトを追加していきます。アプリケーションに別のウィンドウを追加するには、新しいモジュールを追加します。

モジュールへのオブジェクトの追加

モジュールにオブジェクトを追加するには、次の手順に従います。

1. 「ソース」ペインで、オブジェクトの種類を選択します。
2. オブジェクトを［**Module1**］タブにドラッグします（または、オブジェクトをダブルクリックします）。

3. オブジェクトを選択し、「プロパティ」ペインでプロパティを更新します。詳細は、「オブジェクトプロパティの変更」（662ページ）の節を参照してください。
4. オブジェクトにスクリプトを追加します。詳細は、「スクリプトの記述」（665ページ）の節を参照してください。
5. （オプション）アプリケーションの実行時にモジュールが起動しないようにするには、「オブジェクト」ペインでモジュールを選択し、［**Auto Launch**］の選択を解除します（モジュールのうち1つだけをテストする場合などに便利）。
6. アプリケーションビルダーの赤い三角ボタンメニューから［**アプリケーションの実行**］を選択し、アプリケーションをテストします。

   モジュールの使い方によっては、1つのモジュールだけをテストすることができず、他のモジュールも作成しなければならない場合があります。

モジュールの追加

1. ［**構成**］>［**モジュールの追加**］を選択します。
2. 「プロパティ」ペインで、「**Module Type**」を指定します。
   - ダイアログ
   - メニュー付きダイアログ
   - モーダルダイアログ
   - 起動ウィンドウ
   - レポート

ヒント： アプリケーションの実行時に表示されるウィンドウのタイトルは、データテーブルの名前にハイフンとアプリケーション名を加えたものとなります。アプリケーション名を変更するには、JMP App オブジェクトの「**Name**」プロパティのテキストを変更します。

モジュール名の変更

「**Variable Name**」プロパティを変更します。

モジュールの削除

［**構成**］>［**モジュールの削除**］を選択します。

データテーブルの削除

データテーブルオブジェクトを選択し、右クリックして［**削除**］を選択します。

メモ： オブジェクトが関連付けられているデータテーブルは、削除できません。その場合、まずオブジェクトをすべて削除してから、データテーブルを削除します。

モーダルダイアログのモジュール

モーダルダイアログのモジュールには特殊な動作が見られるため、使用時に注意が必要です。

- `ret = Module1 << Create Instance()` は、ダイアログが完了するまで値を戻しません。その時点では、ダイアログはなくなっています。そのため、戻り値は、モジュールインスタンスへのハンドルにはなりません。`New Window()` と同じく、戻り値は `{Button(1 | -1), User Data}` のような形式になります。(1 | -1) は、OK（1）と Cancel（-1）ボタンのどちらが押されたかを示します。
- `User Data()` は、モジュールインスタンスの新しいプロパティです。ダイアログの実行中、モジュールのスクリプトは、ユーザデータを設定することができます。

  ```
  thisModuleInstance << Set User Data(...);
  ```

  これは、呼び出し側によって解釈される何かを保存するために行われます。ユーザデータは、設定時に評価されます。対応する `Get User Data()` もあります。
- `New Window()` と同様、［OK］ボタンや［キャンセル］ボタンが含まれていない場合は［OK］ボタンが追加されます。
- モジュールインスタンスには、オプションの「Validate」スクリプトプロパティがあります。このスクリプトは、モーダルダイアログにのみ使用され、`New Window()` の `On Validate()` のように動作します。［OK］ボタンが押されたときに呼び出され、入力を受け付けるときは1、閉じる操作を許可しないときは0を戻します。

  ヒント：「Validate」プロパティを使用すると、`thisModuleInstance << Set User Data()` を呼び出すことができます。

- モーダルダイアログは、他の種類のモジュールとは異なり、モジュールスクリプトが完了するまで表示されません。他の種類のモジュールの場合、ウィンドウは、`thisModuleInstance << Create Objects` の呼び出し中に作成されます。モーダルダイアログの場合は、この時点で表示すると、制御が停止し、表示されているボックスの内容を初期化する手段がなくなってしまうため、表示されません。ウィンドウがまだ作成されていないため、ウィンドウタイトルの設定、クローズ時に実行するスクリプトの設定といったアクションを行うことはできません。

オブジェクトの整列と削除

オブジェクトをモジュールに追加するには、オブジェクトを「ソース」ペインからワークスペースにドラッグするか、またはダブルクリックします。なお、現在、選択されているオブジェクトの周りには、青色の枠が表示されます。このとき、いくつかの方法でオブジェクトを整列させることができます。また、オブジェクトのコンテナを変更したり、オブジェクトを新しいコンテナに挿入することもできます。

ヒント：［オブジェクト］ペインでも、オブジェクトを選択できます。特に、外側のコンテナに完全に覆われているオブジェクトや、枠の中にテキストボックスがあるオブジェクトを選択するには、こちらを用いたほうが簡単に選択できます。

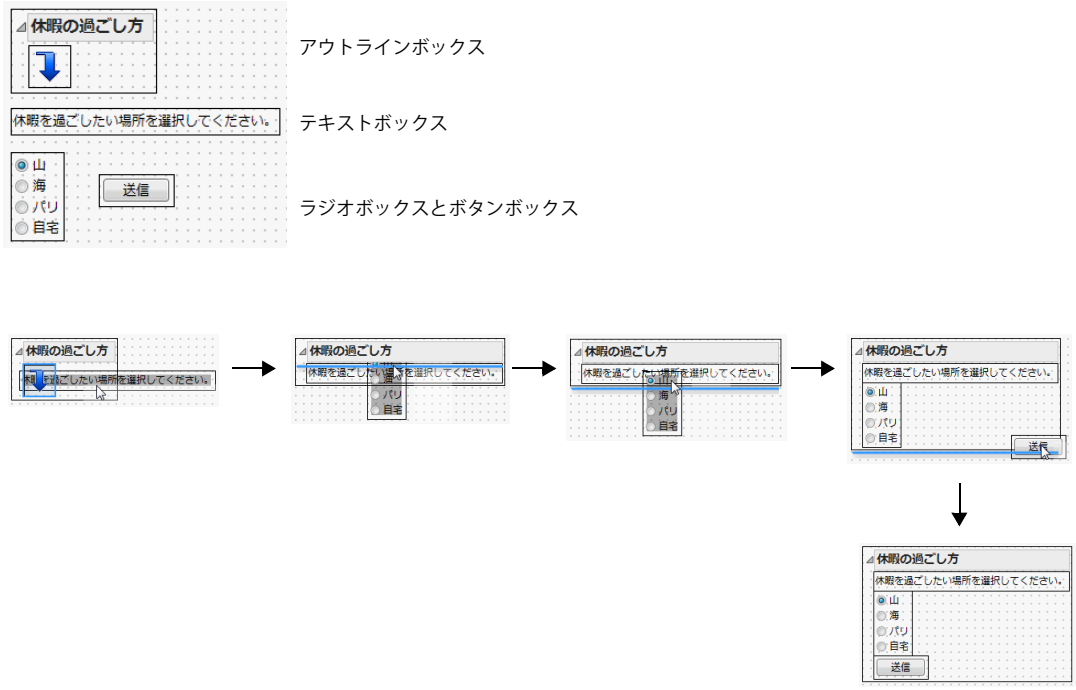
オブジェクトのドラッグ

オブジェクトを配置する方法の1つは、ワークスペースでドラッグする方法です。

- ワークスペースにオブジェクトをドラッグすると、オブジェクトの左上隅が最も近いグリッド点に配置されるように調整されます。より細かく配置したい場合は、アプリケーションビルダーの赤い三角ボタンのメニューで［**グリッドにスナップ**］の選択を解除します。また、JMPの環境設定を使うと、すべてのアプリケーションで［**グリッドにスナップ**］を無効にできます それには、［**ファイル**］（**Macintoshの場合は**［**JMP**］)>［**環境設定**］>［**プラットフォーム**］>［**JMP App**］を選択します。
- オブジェクトを、1つずつワークスペースに配置する代わりに、コンテナに配置します。たとえば、2つのボタンボックスを縦方向ボックス（V List Box）に入れて縦に揃えます。
- オブジェクトをドラッグする際、Shiftキーを押すと、移動を1つの軸方向に制限できます（たとえば、オブジェクトを縦方向に動かさず、横方向のみに動かすことができます）。
- コンテナの中にオブジェクトをドラッグする場合は、オブジェクトをドロップできる場所に矢印が表示されます。
- オブジェクトを別のオブジェクトの上にドラッグすると、ドロップできる場所に青い線が表示されます。

図15.5に、コンテナ内にオブジェクトをドラッグする方法を示します。

図15.5 コンテナ内にオブジェクトをドラッグする例



コンテナオブジェクトのX座標とY座標の変更

- コンテナオブジェクトを配置する座標を細かく決めたい場合は、オブジェクトを選択した後、「**X Position**」と「**Y Position**」のプロパティを変更します。新しいX座標を入力した後、Tabキーを押し、オブジェクトの移動を確認してから、新しいY座標を入力します。
- オブジェクトを左上隅に配置するには、右クリックして［**隅に移動**］を選択します。このオプションは、XとYの座標を0に設定します。Macintoshの場合は、Ctrlキーとcommandキーを押しながら［**隅に移動**］を選択します。

複数のオブジェクトの整列

オブジェクトの縦方向や横方向の座標を揃えるには、複数のオブジェクトを選択し、右クリックメニューの［**配置**］>［**ボックスの配置**］からオプションを選択します。

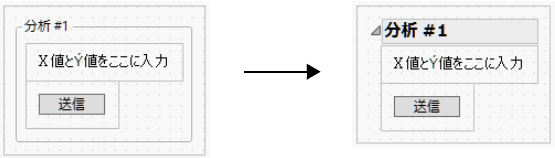
ヒント:

- コンテナ内部に配置されている複数のオブジェクトではなく、コンテナ全体を選択した場合、［**配置**］オプションは使用できません。
- コンテナ内部で右クリックすると、内部にあるオブジェクトが、意図せずに選択されてしまう場合があります。コンテナ外部の、ワークスペースのどこかで右クリックするようにしてください。

コンテナの種類の変更

コンテナを挿入した後、種類を変更したくなった場合、オブジェクトを作成し直す必要はありません。たとえば、テキストとボタンを含むパネルを作成したとしましょう。そのパネルをアウトラインに変更する場合は、パネルを選択した後、ワークスペースで右クリックし、［**コンテナの変更**］>［**Outline Box**］を選択します（図15.6）。

図15.6 パネルをアウトラインに変更



この例では、アウトラインのタイトルは、「Outline1」のように初期化されますので、パネルのタイトルに合わせて変更してください。

ヒント： オブジェクトの［**Horizontal**］プロパティを選択したり、選択を解除したりすると、ボックスの配置方向を簡単に変更できます。

新しいコンテナへのオブジェクトの挿入

アプリケーションビルダー上部のツールバーには、境界ボックスやマウスボックスなどのコンテナのボタンがあります（図15.7）。

選択したオブジェクトをコンテナに挿入するには、ツールバーで該当するボタンをクリックするか、[**構成**] > [**コンテナの追加**] を選択します。

図15.7 コンテナツールバー

オブジェクトの複製

オブジェクトをコピーして貼り付けた場合、複製されたオブジェクトには、新しい名前が付けられます。また、オブジェクトに付随しているスクリプトも名前が変更されます。Ctrl キー（Macintosh の場合は command キー）を押しながらオブジェクトをドラッグしても、オブジェクトの複製を作成できます。

オブジェクトの削除

- 1つ、または複数のオブジェクトを選択して Delete キーを押すと、そのオブジェクトが削除されます。
- オブジェクトをモジュールの外にドラッグしても、そのオブジェクトが削除されます。
- Macintosh の場合、オブジェクトを選択し、Ctrl キーと command キーを押しながら [**削除**] を選択しても、削除されます。
- Windows の場合、[**編集**] > [**クリア**] を選択すると、モジュール内のすべてのオブジェクトが削除されます。
- オブジェクトを選択し、BackSpace キーを押しても、そのオブジェクトが削除できます。

オブジェクトのスクリプトが不要になった場合は、スクリプトも削除できます。

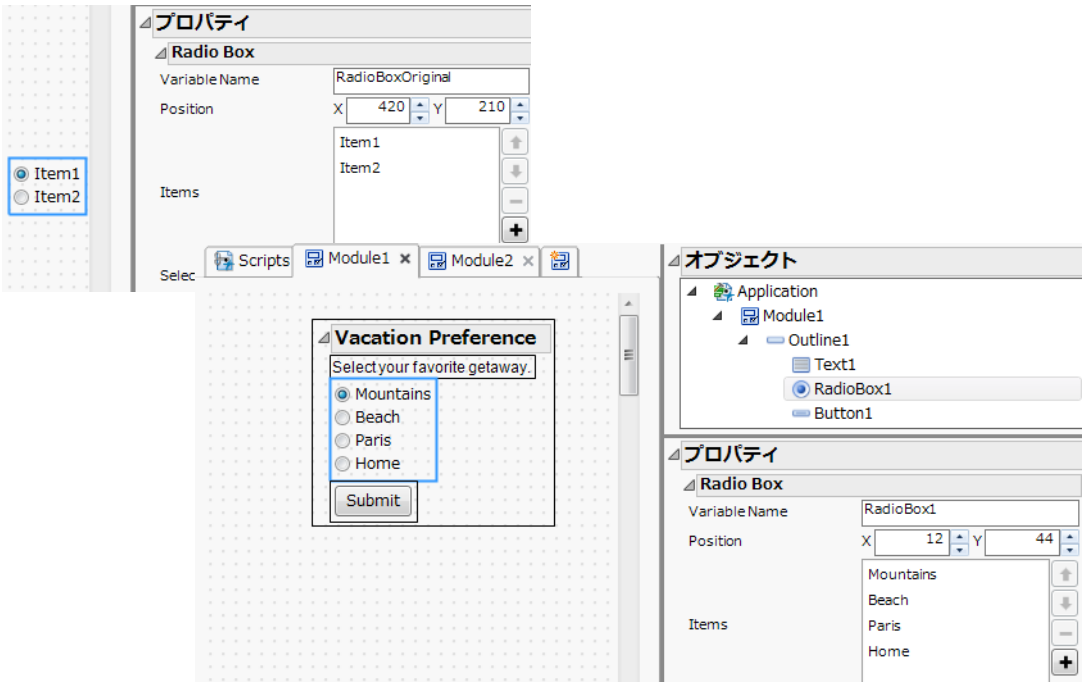
オブジェクトプロパティの変更

ワークスペースに置かれたオブジェクトのプロパティは、「プロパティ」ペインで変更できます。リスト項目やボタン名などの設定は、これらのプロパティを変更するだけで行えますので、JSL を記述する必要はありません。

変数名には大文字／小文字の区別がなく、スペースも認識されません。スクリプトに「Button1」という名前のオブジェクトが含まれているときに、別のオブジェクトの名前を「Button 1」に変更しようとすると、警告が表示されます。

たとえば、図15.8のような Radio Box のリスト項目を設定するには、「Item1」と「Item2」をダブルクリックして、新しいリスト項目を入力していきます。

図15.8 ラジオボックスのオブジェクトプロパティ



オブジェクトの詳細を参照するには、オブジェクトを右クリックし、[**スクリプトのヘルプ**]を選択します。選択したオブジェクトが強調表示され、JMPのスクリプトの索引が開きます。「スクリプトの索引」にスクリプトが用意されている場合は、それを実行するとオブジェクトの例を確認できます。

「プロパティ」ペイン内の項目については、項目の上にカーソルを置くと説明が表示されます。

アプリケーションのプロパティ

アプリケーションに対するプロパティには、実行パスワードやデータテーブル名などがあります。これらのプロパティを設定するには、「オブジェクト」のリストボックスで、「アプリケーション」を選択します。

Name　ここで設定した名前は、アプリケーションのタイトルバーにおいて、データテーブル名の後に表示されます。

Auto Launch（起動ウィンドウ）　ユーザに対し、アプリケーションで定義されている引数を選択するための起動ウィンドウが表示されます。この起動ウィンドウは、ユーザが、モジュールとして定義できるものではありません。この起動ウィンドウは、各モジュールの［Auto Launch］プロパティがオンになっている場合に自動的に呼び出されます。

Encrypt（暗号化）　テキストエディタを使ったアプリケーションの編集が不可能になります。ユーザが実行するアプリケーションだけが暗号化されます（.jmpappファイルとJMPアドイン内のアプリケーション）。スクリプト（データテーブル、ジャーナル、およびアドインに保存されたスクリプト）が、暗号化されま

す。暗号化の詳細については、「プログラミング手法」章の「スクリプトの暗号化と暗号解読」（255ページ）を参照してください。

Run Password（実行パスワード） アプリケーションの実行に必要なパスワードを入力します。パスワードをテストするには、アプリケーションビルダーの外でアプリケーションを実行します。

テーブルモジュールのプロパティ

Data Tableオブジェクトを挿入すると、データテーブルオブジェクトが作成されます。データテーブルパスなどのプロパティを定義するには、まず、「オブジェクト」ペインでそのデータテーブルモジュールを選択します。そして、「プロパティ」ペインで次のオプションを変更します。

Variable Name データテーブルオブジェクトの名前を指定します。この名前は「プロパティ」ペインと「オブジェクト」ペイン、そして、アプリケーションのJSLスクリプト内に表示されます。

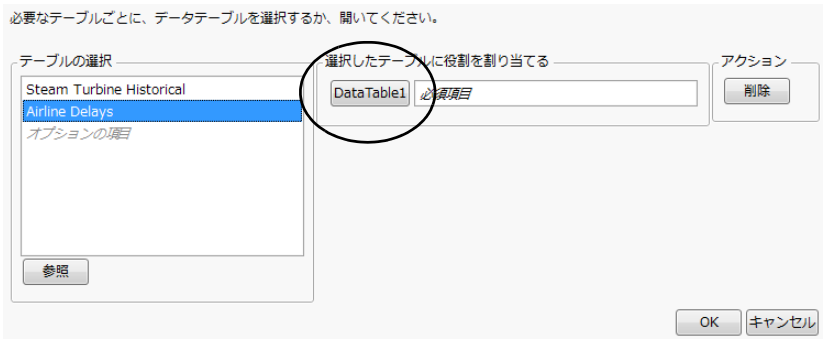
Path アプリケーションで使用されるデータテーブルの絶対パスまたは相対パスを指定します。データテーブル名の前に、$HOMEや$USER_APPDATAといったパス変数を使用することもできます。

このアプリケーションを編集する際、指定したデータテーブルがアプリケーションビルダーによって開かれます。「Path」プロパティが空である場合や、指定のデータテーブルが見つからない場合は、テーブルを開くよう指示するプロンプトが表示されます。

データテーブルを閉じたとき、アプリケーション内にそのデータテーブルに依存するオブジェクトがある場合は、それらのオブジェクトがアプリケーションから削除され、警告メッセージが表示されます。オブジェクトを復元するには、アプリケーションを開き直します。

Label ユーザにデータテーブルを開くことを促す際に使用する文字列を指定します。図15.9では、デフォルトの値が表示されてます。

図15.9 データテーブルを開くためのプロンプト



Location ユーザがアプリケーションを実行したときに、アプリケーション内で使用されるデータテーブルが選択される方法を指定します。

– Current Data Table: 現在のデータテーブルを使用します。開いているデータテーブルがない場合、ユーザにデータテーブルを開くよう促します。

- Full Path: 「Path」プロパティで指定されたデータテーブルを使用します。
- Name: 指定の名前を持つ、最初に開かれたデータテーブルを使用します。そうでない場合、「Path」プロパティで指定されたデータテーブルを使用します。
- Prompt: ユーザに、開かれているデータテーブルを選択するよう、または、データテーブルを指定するよう促します。
- Script: アプリケーションスクリプトまたはモジュールスクリプト内で定義されているデータテーブルを使用します。
- Embedded Script: レポートの作成に使用されたデータテーブルの「ソース」スクリプトを読み込みます。「ソース」スクリプトは、アプリケーションを開いたときに実行されます。

Invisible データテーブルを非表示にします。ただし、JMPホームウィンドウにはリストしたままです。このオプションは、「Location」で［Full Path］および［Name］を選択した場合に使用できます。

スクリプトの記述

オブジェクトに特定の機能を持たせるためには、スクリプトを記述します。たとえば、ユーザがボタンをクリックすると何らかの処理を行う機能や、ディレクトリを選択してデータテーブルを選択する機能などは、スクリプトで記述する必要があります。スクリプトでプログラミングすれば、ラジオボタンの選択に応じて、異なったグラフを表示するといった機能も実現できます。

アプリケーションとモジュールの名前空間

アプリケーションビルダーは、変数がスクリプト間で競合（衝突）しないようにするため、アプリケーションと各モジュールに対して、名前空間を自動的に設定します。

- アプリケーションの名前空間には、アプケーション全体で使用されるスクリプトを記述してください。この名前空間で定義された関数は、どのモジュール内でも使用できます。
- 各モジュールの名前空間には、そのモジュールのインスタンスが作成されたときに実行するスクリプトを記述してください。たとえば、あるアプリケーションにおいて、ボタンをクリックしたときに新たなウィンドウを開くとします。このウィンドウは、そのウィンドウのモジュールのインスタンスです。なお、同じモジュールから2つのインスタンスを作成した場合、各インスタンスに、それぞれ自身の変数や関数のコピーが含まれます。

これらの名前空間内のスクリプトを見るには、［**Scripts**］タブをクリックし、「名前空間」リスト（または「オブジェクト」ペイン）で名前空間を選択します。詳細については、図15.10を参照してください。

図15.10 アプリケーションとモジュールの名前空間

名前付きスクリプトと匿名スクリプトの2種類のスクリプトがあります。

名前付きスクリプトの記述

名前付きスクリプトは、関数の形式で定義されています。複数の異なるコントロールにおいて、共通のコードを使用できるという利点があります。名前付きスクリプトの関数において、`this`引数は、関数を呼び出しているコントロールを示します。次の例では、ボタンをクリックしたときに、`Get Button Name`が`this`引数に送られ、ログにボタン名が印刷されます。

```
Button1Press = Function({this}, Print(this <<Get Button Name))
```

別のボタンで`Button1Press`スクリプトを実行しても同じ結果となります。

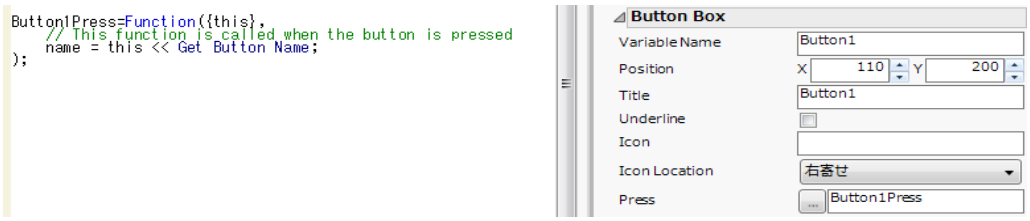
他の利点としては、名前付きスクリプトを追加した場合は、[Scripts]タブにプログラムのテンプレートが挿入されるので、プログラミングの手間が省ける点が挙げられます。さらに、[Scripts]タブでは「メソッド」リストからスクリプトを選択でき、該当の関数までジャンプできます。これは、プログラムが長い場合に便利です。

名前付きスクリプトは、次のように追加します。

1. オブジェクトを右クリックして［**スクリプト**］を選択し、追加したいスクリプトを選択します。この例のボタンでは、［**Press**］を選択します（Macintoshの場合は、Optionを押しながら［**スクリプト**］>［**Press**］を選択）。

   ［**Scripts**］タブに、スクリプトのテンプレートが表示されます（図15.11）。また、名前付きスクリプトの名前である**Button1Press**が、スクリプト中では関数名として使われ、また、オブジェクトのプロパティでも表示されます。

図15.11 新しいスクリプトとスクリプトプロパティ



2. 必要な機能を実現するため、スクリプトとプロパティを編集します。たとえば、図15.11のようにボタンボックスを変更すると、次のようになります。
   - 「Title」プロパティのテキストが、**「送信」**に変更されています。ボタンに、「送信」と表示されます。
   - `Close Window`メッセージを、オブジェクトに送るように、スクリプトが変更されています。これにより、ユーザが［**送信**］ボタンをクリックすると、ウィンドウが閉じます。

ヒント： オブジェクトを削除しても、該当するスクリプトは削除されません。スクリプトを追加した後、オブジェクトを削除し、そのスクリプトも不要であるなら、［**Scripts**］タブから、削除してください。スクリプトを自動的に削除しないのは、将来必要になるかもしれないスクリプトが削除されるのを防ぐためです。

また、オブジェクトのプロパティで、スクリプトの名前を変更した場合は、［**Scripts**］タブでも、その名前に応じて変更してください。該当するスクリプトがアプリケーションの別の部分でも使用されているときは、そこでも名前を変更してください。

匿名スクリプトの作成

匿名スクリプトは、それを定義したオブジェクトに対してしか実行できません。たとえば、ボタンに対して、そこでしか使用しないログ出力の機能（`Print`）が必要な場合、匿名スクリプトで記述すれば、スクリプト全体で定義する名前付きスクリプトの数を減らすことができます。匿名スクリプトは、オブジェクトのプロパティとして設定します。そのため、［Scripts］タブにおいて、名前付きスクリプトが雑多になりすぎるのも回避できます。

次に、2種類の匿名スクリプトの例を示します。

```
Print(Button1 <<GetButtonName) // 単純な匿名スクリプト
Function({this}, Print(this <<GetButtonName)) // パラメータ化された匿名スクリプト
```

1番目のスクリプトでは、「Button1」変数にメッセージを送っています。一方、2番目のスクリプトでは、コントロールを表す「`this`」に対して、メッセージを送っています。

チェックボックスなどのオブジェクトは、`this`を用いて、必要な情報を取得する必要があります。たとえば、チェックボックスを用いた場合、どの項目がチェックされたかを取得する必要があるでしょう。

匿名スクリプトは、次の手順により設定できます。

1. オブジェクトを選択し、オブジェクトのプロパティで…をクリックします。

   匿名スクリプトエディタが表示されます。

2. スクリプトを入力し、[**OK**]をクリックします。

   匿名スクリプトのテキストが、オブジェクトのプロパティに表示されます（なお、名前付きスクリプトを設定した場合、ここには、その関数の名前が表示されます）。

ヒント： プログラムの保守を簡単にするには、同じコードを、匿名関数として何度もコピーして用いるのは避けてください。同じコードを複数の場所で用いる場合は、名前付きスクリプトを使用してください。

特定のスクリプトの表示

特定のオブジェクトのスクリプトを表示する方法はいくつかあります。

- [**Module**]タブでオブジェクトをダブルクリックします。
- オブジェクトに複数のスクリプトがある場合は、オブジェクトを右クリックして[**スクリプト**]を選択し、表示したいスクリプトを選択します。また、[**Scripts**]タブを選択し、「**名前空間**」リストからモジュール名を選択して、「**メソッド**」リストからスクリプト名を選択する方法もあります。同様に、アプリケーションスクリプトを表示するには、「**名前空間**」リストから「**Application**」を選択します。

どちらの場合も、[**Scripts**]タブが表示されたとき、そのオブジェクトに対するスクリプトの一行目に、カーソルは移動します。

ヒント： 雑然としたスクリプトを読みやすくするには、右クリックして[**スクリプトを再フォーマット**]を選択します。

スクリプトとともにオブジェクトをコピーして貼り付け

スクリプトを持つオブジェクトをコピーして貼り付けた場合、新しいオブジェクトとスクリプトには、元のものとは別の名前が与えられます。

アプリケーションの編集または実行

アプリケーションを開いて編集するには、[**ファイル**] > [**開く**]を選択し、.jmpappsourceファイルを選択して[**開く**]を選択します。

メモ：「Table」プロパティが空である場合や、指定のデータテーブルが見つからない場合でも、アプリケーションは実行されます。ただし、データテーブルを要求するオブジェクトは作成できないため、警告が表示されます。

開いているアプリケーションを実行するには、アプリケーションビルダーの赤い三角ボタンのメニューから、[**アプリケーションの実行**]を選択します。

閉じているアプリケーションを実行するには、[**ファイル**] > [**開く**]を選択し、.jmpappファイルを選択して[**開く**]を選択します。

Windowsの場合、次のいずれかを行うことによっても、JMPホームウィンドウからアプリケーションを開いたり、実行したりできます。

- Windowsエクスプローラに表示されたファイルを、JMPホームウィンドウまたは空のアプリケーションウィンドウにドラッグします。
- 「最近使ったファイル」内のファイルをダブルクリックします。
- .jmpappsourceファイルまたは.jmppappファイルを右クリックし、[**アプリケーションの編集**]または[**アプリケーションの実行**]を選択します。

アプリケーションの保存オプション

JMPには、アプリケーションファイルを保存するためのオプションがあります。

- Windowsで[**ファイル**]>[**名前を付けて保存**]を選択した場合、アプリケーションをアプリケーションソースファイル(.jmpappsource)、アプリケーション(.jmppapp)またはスクリプトとして保存できます。
- Macintoshで[**ファイル**]>[**書き出し**]を選択した場合は、アプリケーションをアプリケーションファイルまたはスクリプトとして保存できます。Macintoshでアプリケーションをアプリケーションソースファイルとして保存するには、[**ファイル**]>[**別名で保存**]を選択します。

.jmpappsourceファイル形式の場合、アプリケーション開発者は、開発途中のアプリケーションを保存し、後で作業を続けることができます。アプリケーションビルダーでファイルを再び開くと、保存時と同じ状態でウィンドウが表示されます。アプリケーションの作成に使用したデータテーブルもだ使用可能な状態でなければなりません。

アプリケーションを他のユーザに配布する場合は、.jmppappファイル形式を使用します。ファイルを開くとすぐにアプリケーションが実行されます。アプリケーションは、その設定に従い、現在のデータテーブル、ディスク上のファイル、またはユーザからの入力から、使用するデータテーブルを判断します。

「スクリプトの保存」の赤い三角ボタンのメニューには、スクリプトを保存するためのその他のオプションがあります。暗号化されたスクリプトを保存した場合、そのスクリプトは`JSL Encrypted()`関数で閉じられ、空白とコメントが保持されます。

データテーブルへ　レポートの作成に使用されたデータテーブルに、スクリプトを保存します。これにより、データテーブルから再びスクリプトを実行し、結果を再現することが可能になります。データテーブルからこのスクリプトを編集しようとした場合、スクリプトエディタではなくアプリケーションビルダーが開きます。

ジャーナルへ　スクリプトの実行用ボタンをジャーナルに保存します。スクリプトは現在のジャーナルに追加されます。スクリプトには、データテーブルへのパスが含まれます。データテーブルが見つからない場合、スクリプトは実行されません。

スクリプトウィンドウへ　スクリプトエディタウィンドウを開き、そこにスクリプトを追加します。スクリプトウィンドウにすでにスクリプトを追加している場合は、新しいスクリプトが同じスクリプトウィンドウの下部に追加されます。これらのスクリプトは編集できますが、ここで大幅に変更したアプリケーションは、アプリケーションビルダーで編集できなくなる場合があるので注意してください。このオプションは、アプリケーションをより大きなJSLプロセスに組み込みたいときに便利です。

ヒント：.jmpapp ファイルとして保存し、Open() を使ってアプリケーションを実行することもできますが、スクリプトを保存したほうが、ファイルの数を少なく抑えられます。

アドインへ　アドインを作成します。アドインをインストールすると、開発したアプリケーションがJMPメニューから起動できるようになります。詳細については、「JMPアドインビルダーを使ったアドインのコンパイル」（682ページ）を参照してください。

その他のアプリケーション作成例

以下の例で、JMPにおけるアプリケーションのさまざまな用途を紹介します。

分析対象の列

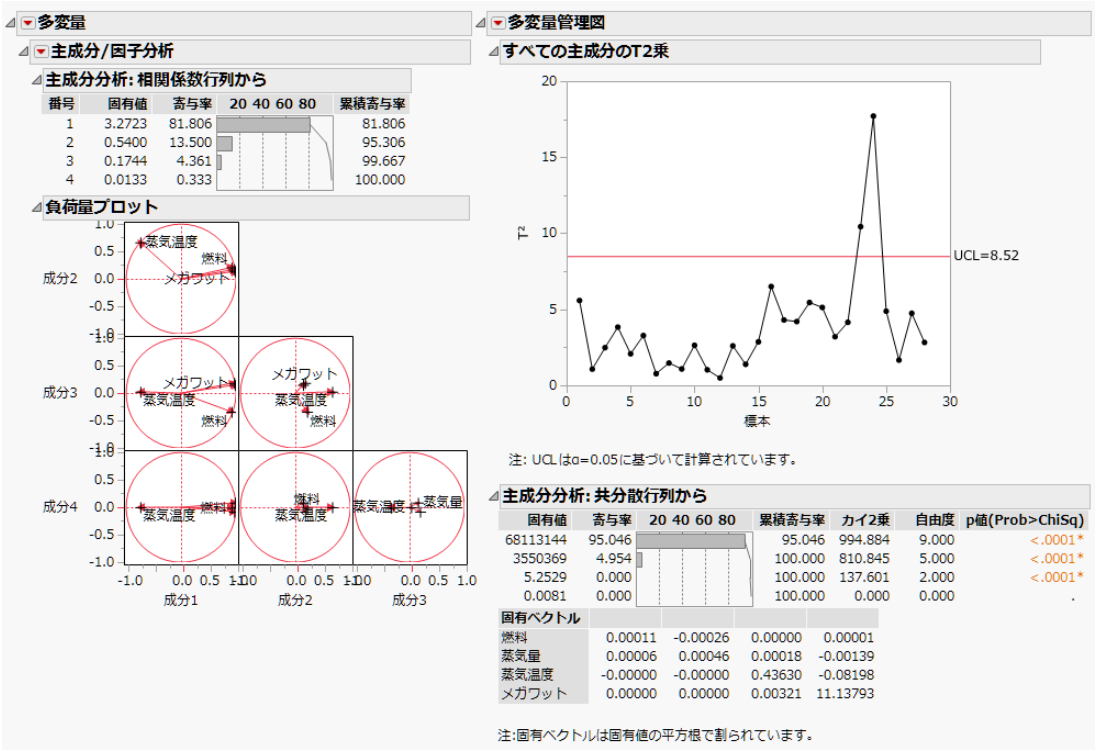
次の例は、分析対象の列を指定できるアプリケーションです。このアプリケーションでは、ユーザは、起動ウィンドウで列を選択できます。選択された列に対して、特定のレポートが作成されます。

1. ［**ヘルプ**］>［**サンプルデータライブラリ**］を選択し、「Quality Control¥Steam Turbine Historical.jmp」を開きます。
2. 「主成分分析」と「主成分負荷量プロット」のテーブルスクリプトを実行してレポートを作成します。
3. ［**ファイル**］>［**新規作成**］>［**アプリケーション**］を選択します。
4. **「空白のアプリケーション」**を選択します。

   「アプリケーションビルダー」ウィンドウが開きます。
5. ウィンドウのサイズを大きくします。
6. 「ソース」ペインの各多変量レポートをアプリケーションワークスペースにドラッグし、横に並べます。
7. ［**編集**］>［**すべて選択**］を選択します。
8. ［**構成**］>［**コンテナの追加**］>［**H List Box**］を選択し、レポートを横方向に並べます。
9. 2つのレポートそれぞれにおいて、レポートを選択した後、「オブジェクト」の**「Y Variable」**プロパティに「yvar」と入力します。
10. 「アプリケーションビルダー」の赤い三角ボタンメニューから、［**アプリケーションの実行**］を選択します。
11. **「燃料」**、**「蒸気量」**、**「蒸気温度」**、**「メガワット」**の各変数を選択し、［**Y**］ボタンをクリックします。
12. ［**OK**］をクリックします。

    選択された列に対する2つのレポートが、1つのウィンドウ内に表示されます（図15.12）。

図15.12 多変量アプリケーション



ヒント： データテーブルのプロパティには、デフォルトでは、データテーブルへの絶対パスが設定されます。サンプルデータをデータとして使った場合、パスに自動的に $SAMPLE_DATA パス変数が含まれます。絶対パスまたは相対パスを入力することもできます。パスは、ユーザにとってアクセス可能でなければなりません。

複数のレポートでフィルタリングしたデータ

複数のレポートを含むアプリケーションでは、1つのレポートでデータを選択し、同じウィンドウ内にある別のレポートに選択されたデータだけを表示させることができます。

フィルタを設定するには、次の手順に従ってください。

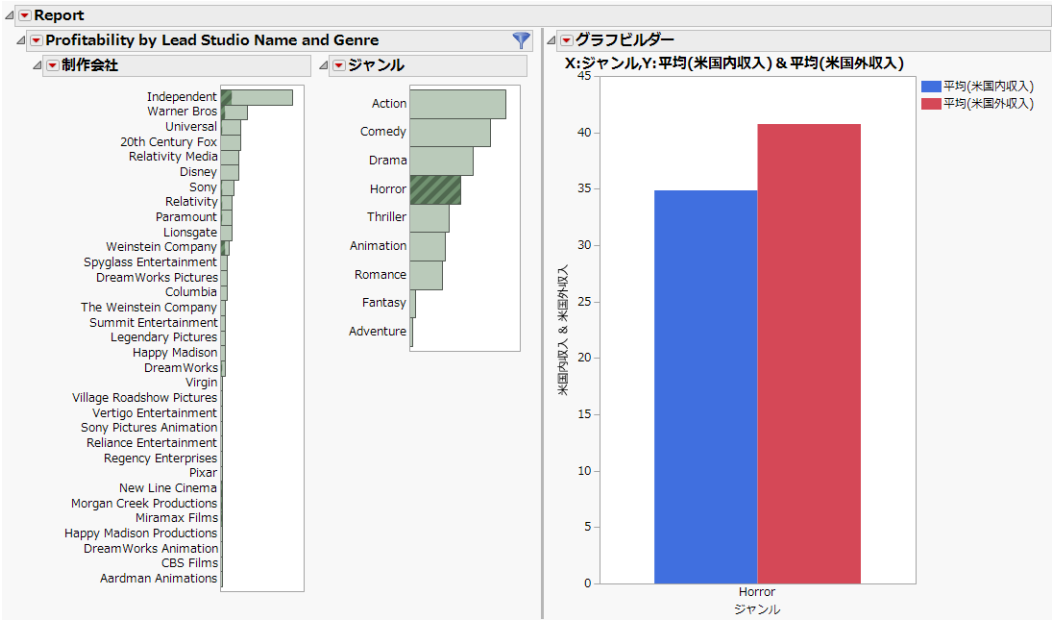
1. アプリケーションを作成し、2つのレポートをワークスペースに追加します。
2. 「アプリケーションビルダー」ウィンドウで、メインのレポートを右クリックし、[**選択フィルタとして使用**]を選択します。

   これで、メインのレポートのディスプレイボックスがData Filter Source Boxに、親レポートのディスプレイボックスがData Filter Content Boxに配置されます。
3. アプリケーションビルダーの赤い三角ボタンのメニューから[**アプリケーションの実行**]を選択します。

   これらのレポートが1つのウィンドウに表示されます（図15.13）。

4. アプリケーションをテストする目的で、メインのレポートでヒストグラムの棒を選択してみます。
   2つ目のレポートに、選択した棒のデータだけが表示されます。

図15.13 フィルタしたコンテンツの例

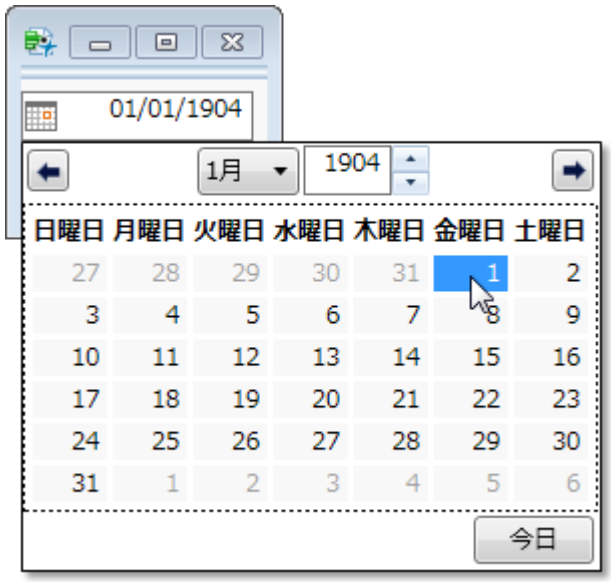


日付の選択

アプリケーションに日付セレクタウィンドウを挿入するには、次の手順に従います。

1. 「ソース」ペインから、「Number Edit Box」をワークスペースにドラッグします。
2. 「Number Edit Box」を選択し、「プロパティ」ペイン内の「Format」の隣のボタンをクリックします。
3. リストから、［**日付**］>［**m/d/y**］を選択します。
4. 「総桁数」ボックスに「12」とタイプします。
5. 形式を指定する枠の外側をクリックします。
6. アプリケーションビルダーの赤い三角ボタンのメニューから［**アプリケーションの実行**］を選択します。
   これで、新しいウィンドウ内に数値編集ボックスが作成され、日付が入力されました。
7. 日付セレクタウィンドウを開くには、マウスをボックスの上に置き、青い三角形が表示されるのを待ちます。
8. 青い三角形をクリックすると、日付セレクタウィンドウが表示されます。

図15.14 日付セレクタの例



日付セレクタを使うと、ボックスに表示する年月日と時間を選択することができます。

起動ウィンドウとレポートの作成

「Launcher with Report」サンプルアプリケーションは、プラットフォームの起動ウィンドウとレポートを組み合わせたものです。このサンプルアプリケーションを表示するには、**［ファイル］>［新規作成］>［アプリケーション］**を選択します。

次の例では、オブジェクトを配置し、スクリプトを追加してアプリケーションを構成する方法を示します。「Launcher with Report」サンプルアプリケーションを再度作成する形となり、独自に類似のアプリケーション作成するのに役立ちます。

メモ: 「Launcher with Report」サンプルアプリケーションの中では、ディスプレイボックスの変数名が、そのディスプレイボックスを説明するようなものに変更されています。たとえば、サンプルアプリケーションの中では「Text1」が「Description」に変わっています。この例では、必要な手順を減らすため、スクリプトが変数名を参照している場合を除き、変数の名前は変更しないでください。

---

1. **［ファイル］>［新規作成］>［アプリケーション］**を選択し、**「Launcher with Report」**サンプルをクリックします。

   このサンプルにあるJSLをコピーし、自分のアプリケーションに貼り付けます。アプリケーションを開くと、「Iris.jmp」サンプルデータテーブルが開きます。

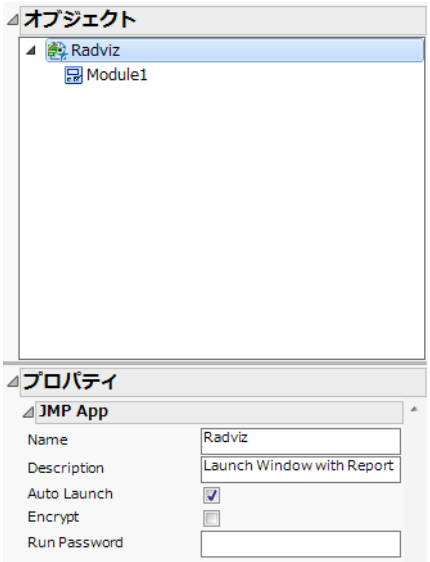
2. **［ファイル］>［新規作成］>［アプリケーション］**を選択し、**「空白のアプリケーション」**をクリックします。

Launch Moduleの作成

Launch Moduleには、起動ウィンドウを構成するオブジェクトが含まれます。

1. 空白のアプリケーションの「オブジェクト」ペインで、アプリケーションを選択します。
2. 「プロパティ」ペインで次の作業を行います。
   - アプリケーションの名前を「Radviz」に変更し、Enterキーを押します。
     アプリケーションを実行したとき、この名前がタイトルバーに表示されます。
   - Descriptionを「Launch Window with Report」に変更します。

図15.15 アプリケーションのカスタマイズ

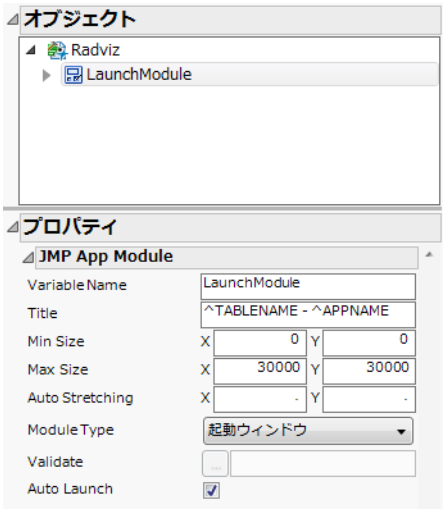


3. [**Scripts**]タブをクリックし、「名前空間」のリストから「Radviz」を選択します。
4. 緑色のコメントの後の新しい行に次の式を入力します。

   ```
   dt = Current Data Table();
   ```

   このアプリケーションに追加するスクリプトの中には、`dt`を参照してデータテーブルを特定するものがあります。
5. 「オブジェクト」ペインで**「Module1」**を選択します。
6. 「プロパティ」ペインで次の作業を行います。
   - 「Variable Name」（変数名）を「LaunchModule」に変更し、Enterキーを押します。
   - 「Module Type」を［起動ウィンドウ］に変更します。
   - ［Auto Launch］が選択されていて、アプリケーションを実行すると自動的に起動ウィンドウが開くようになっていることを確認します。

図15.16 LaunchModuleのカスタマイズ



7. ［**LaunchModule**］タブをクリックし、V List Box（「コンテナ」の下）をワークスペース上にドラッグします。
8. Text Box（「表示」の下）をV List Boxの矢印の上にドラッグします。
9. Text1をダブルクリックして「Radviz」に変更し、Enterキーを押します。
10. H List Box（「コンテナ」の下）をText Boxの枠の下へドラッグします。青色の線が表示されます（図15.17）。

図15.17 H List Boxの追加

11. いろいろな変更を加えたので、ここで［**ファイル**］>［**保存**］を選択して保存しましょう。このときファイルの名前を変更できます。ファイルの種類として.jmpappsourceが選択されていることを確認します。

列を選択するボックスの作成

1. Panel Box（「コンテナ」の下）をH List Boxの矢印の上にドラッグします。
2. 「プロパティ」パネルで、「Title」を「Select Columns」に変更し、Enterキーを押します。
3. Col List Box(All)（「入力」の下）をPanel Boxの矢印の上へドラッグします。
4. 「プロパティ」パネルで、Variable Name（変数名）を「ColumnList」に変更し、Enterキーを押します。後で追加するスクリプトが、この変数名を参照するようになります。

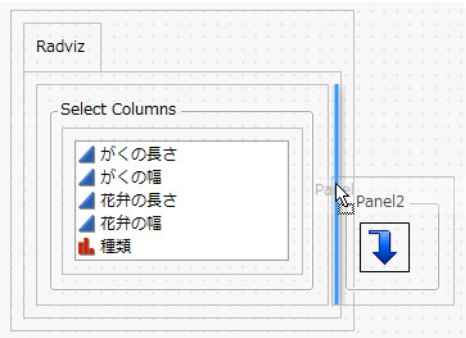
5. 「オブジェクト」ペインで**「DataTable1」**を選択します。

   「Path」ボックスに現在のデータテーブルである「Iris.jmp」が表示されます。

選択した列に役割を割り当てるボックスの作成

1. Panel Box（「コンテナ」の下）を既存のPanel Boxの右側にドラッグします。青色の線が表示されます（図15.18）。

図15.18 Panel Boxの追加



メモ：Panel Box は、図 15.18 にある H List Box にドラッグします。

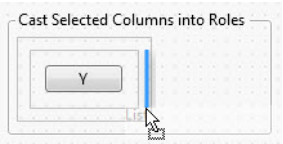
2. 「プロパティ」パネルで「Title」を「Cast Selected Columns into Roles」に変更し、Enterキーを押します。
3. H List Box（「コンテナ」の下）をPanel Boxの矢印の上へドラッグします。

選択した列に役割を割り当てるボックスの中身の作成

1. Button Box（「ボタン」の下）をH List Boxの矢印の上にドラッグします。
2. 「プロパティ」パネルで次の作業を行います。
   - 「Title」を「Y」に変更し、Enterキーを押します。
   - 「Press」ボックスに「SetY」と入力し、Enterキーを押します。

   このボタンのスクリプトは、後でコピーし、貼り付けます。
3. Col List Box（「入力」の下）を「Y」ボタンボックスの右側の枠の中へドラッグします。（図15.19）。

図15.19 Col List Boxの追加



4. 「プロパティ」パネルで次の作業を行います。

– 「Variable Name」（変数名）を「YList」に変更し、Enterキーを押します。

後で追加するスクリプトが、この変数名を参照するようになります。

– 「Data type」を数値に変更します。

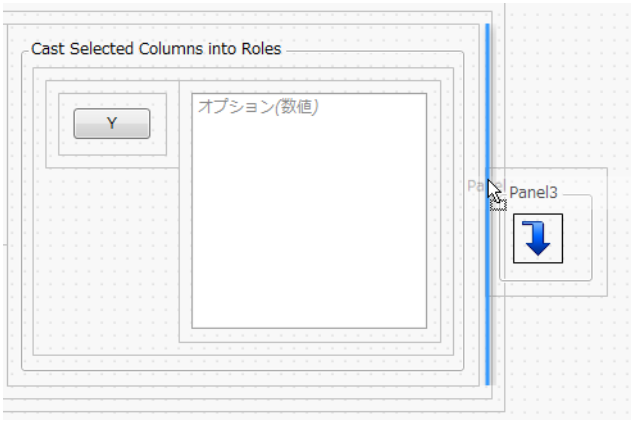
Col List Boxの中に「オプション(数値)」というテキストが表示されます。

ヒント：「Min Items」を変更すると、Col List Boxの表示で必要最小限の項目数がわかるようになります。「3」と入力すると、ボックス内にユーザへのヒントとして「必須(数値)」が3つ表示されます。

アクションパネルの作成

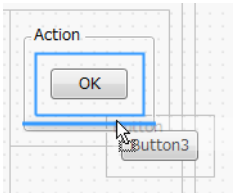
1. Panel Box（「コンテナ」の下）を、先ほど追加したPanel Boxの右側にドラッグします（図15.20）。

図15.20 Panel Boxの追加



2. 「プロパティ」パネルで、「Title」を「Action」に変更し、Enterキーを押します。
3. Button Box（「ボタン」の下）をPanel Boxの矢印の上にドラッグします。
4. 「プロパティ」パネルで次の作業を行います。
   – 「Title」を「OK」に変更し、Enterキーを押します。
   – 「Press」ボックスに「OnOK」と入力し、Enterキーを押します。
5. 別のButton Box（「ボタン」の下）を「OK」ボタンの下にドラッグします（図15.21）。

図15.21 キャンセルボタンの追加



6. 「プロパティ」パネルで次の作業を行います。
   - 「Title」を「Cancel」に変更し、Enterキーを押します。
   - 「Press」ボックスに「OnCancel」と入力します。
7. Spacer Box（「スペーサー」の下）を「Cancel」ボタンの下にドラッグします。
8. Button Box（「ボタン」の下）をSpacer Boxの下にドラッグします。
9. 「プロパティ」パネルで次の作業を行います。
   - 「Title」を「Remove」に変更し、Enterキーを押します。
   - 「Press」ボックスに「OnRemove」と入力し、Enterキーを押します。
10. ワークスペース上でオブジェクトの外枠を選択します。
11. 右クリックし、**［隅に移動］**を選択します。

    選択したオブジェクトの周りにあった余白がなくなります。
12. **［ファイル］＞［上書き保存］**を選択し、.jmpappsourceファイルに変更を保存します。

Launch Moduleスクリプトの作成

1. 「Launcher with Report」サンプルアプリケーションのウィンドウに戻りましょう。
2. 「Scripts」タブをクリックします。
3. 「名前空間」のリストから「LaunchModule」を選択し、スクリプトをコピーします。
4. 自分のアプリケーションのウィンドウに戻り、「Scripts」タブをクリックします。
5. 「LaunchModule」名前空間のスクリプトを削除し、サンプルアプリケーションからコピーしたスクリプトを貼り付けます。
6. **［ファイル］＞［上書き保存］**を選択し、.jmpappsourceファイルに変更を保存します。

Report Moduleの作成

Report Moduleには、グラフを作成するオブジェクトが含まれます。

1. 自分のアプリケーションのウィンドウで、「LaunchModule」タブの右側にあるボタンをクリックし、新しいモジュールを追加します。
2. 「プロパティ」パネルで、Variable Name（変数名）を「ReportModule」に変更し、Enterキーを押します。

3. 「Module Type」が「レポート」になっていることを確認します。
4. アプリケーションの実行時にレポートが開かないようにするため、[Auto Launch] の選択を解除します。
5. Outline Box（「コンテナ」の下）をワークスペース上へドラッグします。
6. 「プロパティ」パネルで、「Title」を「Radviz」に変更し、Enterキーを押します。
7. Graph Box（「表示」の下）をOutline Boxの矢印の上へドラッグします。
8. 「プロパティ」パネルで、Variable Name（変数名）を「Graph」に変更し、Enterキーを押します。
9. ワークスペース上でオブジェクトの外枠を選択します。
10. 右クリックし、[隅に移動] を選択します。

    選択したオブジェクトの周りにあった余白がなくなります。
11. **[ファイル] > [上書き保存]** を選択し、.jmpappsourceファイルに変更を保存します。

Report Moduleスクリプトの作成

1. 「Launcher with Report」サンプルアプリケーションのウィンドウに戻りましょう。
2. 「Scripts」タブが開いていることを確認してください。
3. 「名前空間」のリストから「ReportModule」を選択し、スクリプトをコピーします。
4. 自分のアプリケーションのウィンドウに戻り、「Scripts」タブをクリックします。
5. 「ReportModule」名前空間のスクリプトを削除し、サンプルアプリケーションからコピーしたスクリプトを貼り付けます。
6. **[ファイル] > [上書き保存]** を選択し、.jmpappsourceファイルに変更を保存します。

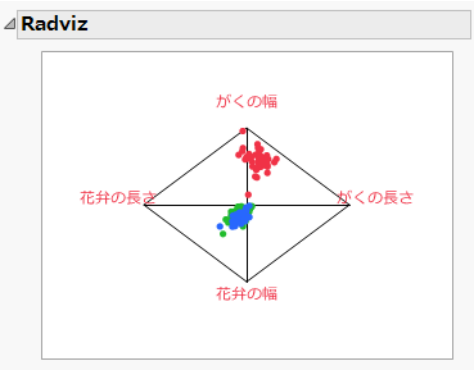
これで、「Launcher with Report」サンプルアプリケーションと同様のアプリケーションができているはずです。

完成したアプリケーションのテストと保存

1. 自分のアプリケーションのウィンドウで、「アプリケーションビルダー」の赤い三角ボタンをクリックし、**[アプリケーションの実行]** を選択します。
2. 「がくの長さ」、「がくの幅」、「花弁の長さ」、「花弁の幅」の各列を選択してから **[Y]** をクリックし、**[OK]** をクリックします。

    データのグラフが表示されます。

図15.22「Iris.jmp」のグラフ



グラフが作成されない場合や、エラーメッセージが表示される場合は、必要に応じて、入力した変数名が正しいかどうかを確認してください。また、各モジュールの「Scripts」タブにスクリプトを追加してあることも重要です。

3. Windowsの場合は、［**ファイル**］>［**名前を付けて保存**］を選択し、「ファイルの種類」で「**JMPアプリケーションファイル**」を指定し、［**保存**］をクリックします。

   Macintoshの場合は、［**ファイル**］>［**書き出し**］を選択し、「JMPアプリケーション」を指定し、［**次へ**］をクリックしてアプリケーションを保存します。

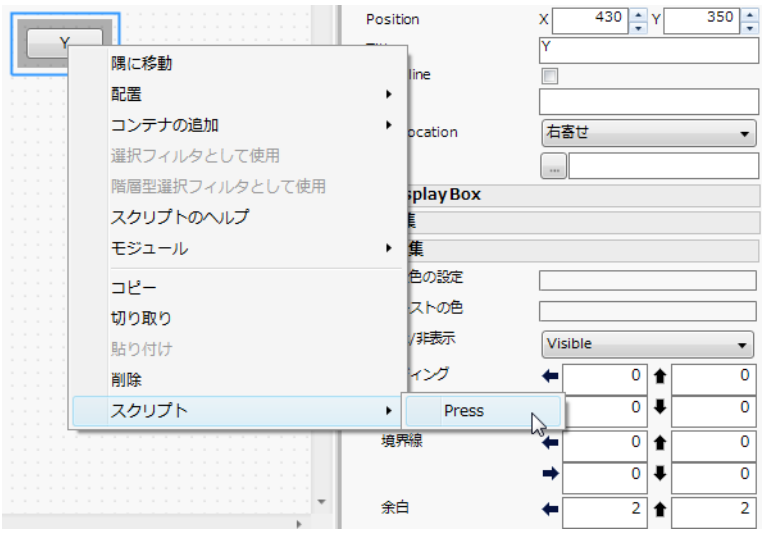
   新しく作成したアプリケーションは、「Launcher with Report」サンプルアプリケーションと同じになります。

4. 「.jmpapp」ファイルを他のユーザに配布します。このファイルは変更ができません。
5. アプリケーションを「.jmpappsource」ファイルとして保存すれば、編集を加え、JMPアプリケーションとして保存し直すことができます。

アプリケーションスクリプトについて

アプリケーションを作成するときは、スクリプトを記述してディスプレイボックスに機能を割り当てます。ディスプレイボックスを右クリックし、［スクリプト］メニューからスクリプトを選択します。

図15.23 スクリプトをインタラクティブに追加する



［スクリプト］メニューからスクリプトを選択すると、選択されているモジュールの［Scripts］タブにプレースホルダースクリプトが追加されます。たとえば、図15.23に表示されている［Press］スクリプトは、［Scripts］タブに次のようなスクリプトを追加します。

```
Button5Press=Function({this},
   // This function is called when the button is pressed
   name = this << Get Button Name;
);
```

プレースホルダースクリプトに、独自のJSLを書き込みます。先ほど作成したアプリケーションでは、2つの変数に「SetY」、「ColumnList」という名前をつけました。プレースホルダースクリプトを次のように書き換えています。

```
SetY = Function({},
   // This function will be called when the Y button is
   // pressed

   items = ColumnList << Get Selected;
   YList << Append(items);
);
```

Get Selectedは、ColumnListに渡されます。なぜなら、オブジェクトの「プロパティ」ペインでCol List BoxにColumnListという変数名を付けたためです。Get Selectedは、選択されている列のリストを戻します。Append(items)は、選択されている列をYList（「Cast Selected Columns into Roles」パネルのCol List Boxに割り当てられた変数）に追加します。

ここで作成したアプリケーションのように、スクリプトを一から記述し、オブジェクトのプロパティでスクリプト名を指定することができます。選択したオブジェクト用の構文が知りたい場合は、オブジェクトを右クリックし、［スクリプトのヘルプ］を選択します。

JMPアドインビルダーを使ったアドインのコンパイル

JMP アドインは、JMP の［アドイン］メニューからいつでも実行できる JSL スクリプトです。JMP アドインをサブメニューにまとめたり、必要に応じて複数レベルのメニューを作成したりできます。

JSLスクリプトを習熟しているユーザは、JMPの機能を拡張する色々なプログラムを作成できます（たとえば、独自の分析ツールや、データベースと通信するユーザインターフェースなど）。JMPアドインのアーキテクチャは、複雑なスクリプトを簡単に配布し、使えるようにします。

会社の仲間に一連のスクリプトを送り、どのように実行するか説明することもできます。しかし、それらを1つのアドインにすれば、JMPを使用している人は簡単にインストールでき、JMPが標準提供している機能と同じように利用することが可能になります。

逆に、会社の仲間からアドインを提供されることもあるかもしれません。また、JMP Webサイト（http://www.jmp.com/addins）にもアドインが用意されています。

アドインビルダーでスクリプトからアドインを作る

JMPアドインを作成するには、まずスクリプトのファイルを準備します。その後、スクリプトをアドインにコンパイルします。

- Windowsでは［**ファイル**］>［**新規作成**］>［**アドイン**］を選択します。
- Macintoshでは［**ファイル**］>［**新規**］>［**アドインの新規作成**］を選択します。
- ダッシュボードビルダーでは、赤い三角ボタンのメニューから［**スクリプトの保存**］>［**アドインへ**］を選択できるようになっています。

スクリプトをアドインにコンパイルするには、次のような手順が必要です。

- 「一般情報の追加」（682ページ）
- 「メニュー項目の作成」（684ページ）
- 「開始時や終了時のスクリプトの指定（オプション）」（685ページ）
- 「その他のファイルの追加」（685ページ）
- 「アドインの保存」（685ページ）
- 「アドインのテスト」（686ページ）

一般情報の追加

まず、［**一般情報**］タブで、アドインの識別と設定に必要な一般情報を入力します。

図15.24 アドインビルダーの［一般情報］タブ

1. アドインの名前を入力します。

   これがアドインの登録名となり、［**表示**］>［**アドイン**］のウィンドウに表示されます。

2. 一意のID文字列を入力します。

   一意のID文字列では、大文字と小文字が区別されます。一意であることを確実にするため、DNS名の逆（たとえば`com.mycompany.myaddin`）を使用することを推奨します。ID文字列は、次の要件を満たす必要があります。

   – 64文字以内である。

   – 文字で始まる。

   – 文字、数字、ピリオド、および下線のみを使用する。

   – スペースを含まない。

   JSLでは、アドインの参照にこの文字列を使用します。

3. アドインのバージョンを入力します。

   後でアドインに変更を加える予定がある場合は、バージョン番号を更新していけば、ユーザに正しいバージョンが配布されているかどうかを確認することができます。

4. アドインが使用できる最低のJMPバージョンを選択します。

   メモ：アドインはJMP 9で導入されたため、それ以前のバージョンではサポートされません。また、アプリケーションをアドインとして保存する場合は、JMPの最低バージョンとして10または11を選択してください。

   ---

5. アドインをWindowsとMacintoshのどちらでサポートするか、または両方でサポートするかを選択します。

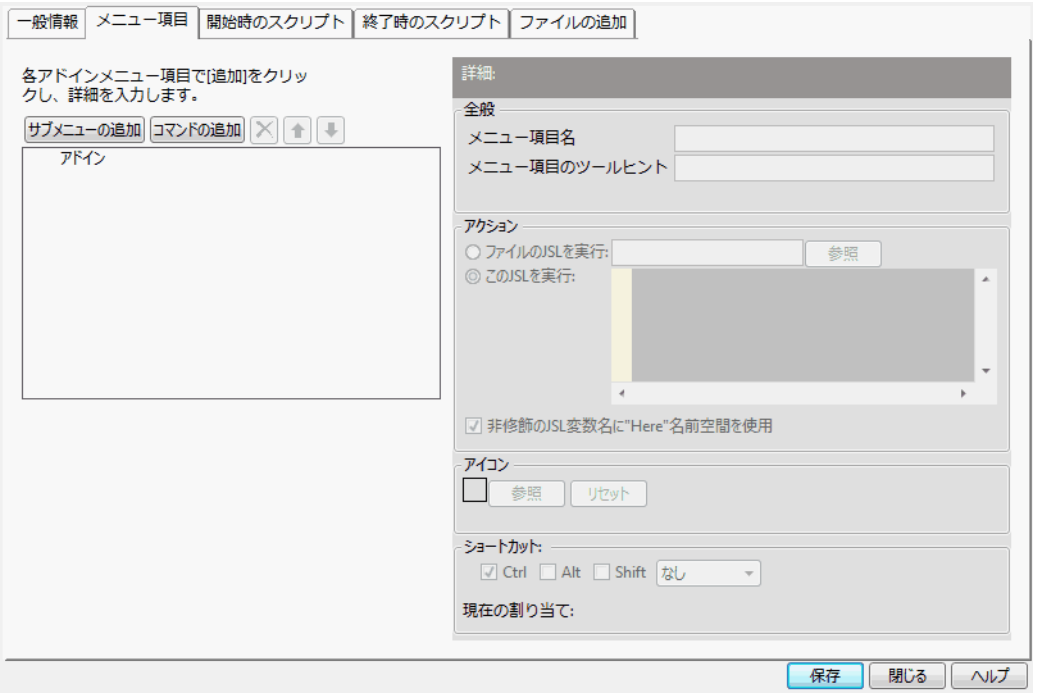
6. （オプション）アドインを保存後にインストールする場合は、［**保存後にインストール**］のチェックボックスにチェックを入れます。

   このオプションにチェックが入っていない場合、アドインを保存してもインストールはされません。なお、アドインをインストールすると、［**アドイン**］メニューの項目に表示されます。

メニュー項目の作成

1. ［**メニュー項目**］タブをクリックします。

図15.25 アドインビルダーの［メニュー項目］タブ



2. （オプション）［**サブメニューの追加**］をクリックします。

   複数のアドインがある場合、それを1つのサブメニューにまとめることができます。
3. サブメニューを追加するときは、［**メニュー項目名**］の隣にサブメニューの名前を入力します。

   この名前が［**アドイン**］メニューに表示されます。
4. ［**コマンドの追加**］をクリックします。
5. ［**メニュー項目名**］の隣にアドインコマンドの名前を入力します。
6. （オプション）ツールヒントを表示させたい場合には、「**メニュー項目のツールヒント**」の横に、ツールヒントのテキストを入力します。このテキストは、ユーザがメニュー項目の上にカーソルを置いたときに表示されます。
7. スクリプトを追加します。［**このJSLを実行**］を選択し、スクリプトをコピーして貼り付けるか、［**ファイルのJSLを実行**］を選択し、［**参照**］をクリックしてスクリプトを含むファイルを選択します。
8. （オプション）すべての非修飾のJSL変数がHere名前空間に含まれ、該当のスクリプトに対してのみローカルとなるようにするには、［**非修飾のJSL変数名に“Here”名前空間を使用**］を選択します。

メモ:

- カスタムメニューまたはツールバーを作成するスクリプトの場合、変数はデフォルトでHere名前空間に含まれます。
- Here名前空間の詳細については、「プログラミング手法」章の「高度な適用範囲指定と名前空間」（230ページ）を参照してください。

9. （オプション）必要に応じて、アイコンを追加します。ここで設定されたアイコンは、[**アドイン**]メニューのメニュー項目の横に表示されます。
10. （オプション、Windowsのみ）必要に応じて、アドインに対するキーボードショートカットを設定してください。
11. 複数のメニュー項目を追加する場合は、この手順を繰り返します。

    サブメニューとアドインコマンドを追加し、複数のレベルを構成することができます。
12. [**保存**]をクリックし、アドインを任意のディレクトリに保存します。
13. [**閉じる**]をクリックします。

メモ: JMPのメニューのカスタマイズについて詳しくは、『JMPの使用法』の「JMPのカスタマイズ」章を参照してください。

開始時や終了時のスクリプトの指定（オプション）

[**開始時のスクリプト**]タブでは、JMP（とアドイン）の起動時に実行されるスクリプトを設定できます。既存のスクリプトを選択するか（[**ファイルのJSLを実行**]）、スクリプトをコピーして貼り付けます（[**このJSLを実行**]）。たとえば、起動の直後にアドインがインストールされていることを知らせるメッセージを表示できます。

[**終了時のスクリプト**]タブでは、JMPの終了時、または、アドインを無効にしたときに実行されるスクリプトを設定できます。既存のスクリプトを選択するか（[**ファイルのJSLを実行**]）、スクリプトをコピーして貼り付けます（[**このJSLを実行**]）。たとえば、アドインを終了するとき、または無効にするときに、開いているJMPデータテーブルを保存するかどうかを確認するプロンプトを表示できます。

その他のファイルの追加

スクリプトが別のスクリプトを呼び出す場合や、グラフィックまたはデータテーブルを含む場合、それらのファイルをここに追加します。

アドインの保存

いずれかのタブの[**保存**]ボタンをクリックすることで、アドインを保存します。この操作により、アドインのファイルが作成されます。

- [一般情報]タブで[**保存後にインストール**]オプションが選択されている場合は、すぐに[**アドイン**]メニューにアドインメニュー項目が表示されます。

- ［**保存後にインストール**］オプションが選択されていない場合は、保存したアドインファイルを開いたときに、インストールするかどうかを確認するプロンプトが表示されます。

アドインのテスト

アドインをインストールした後、次のようにしてアドインをテストします。

1. ［**表示**］>［**アドイン**］を選びます。
2. アドインを選択して、［**登録解除**］をクリックします。
3. ［**ファイル**］>［**開く**］を選択するか、.jmpaddinファイルをダブルクリックして、アドインを再インストールします。
4. メニューまたはツールバーボタンからスクリプトが正常に実行されること、そして、スクリプト自体が正常に動作することを確認します。

アドインの編集

保存したアドインを編集するには

1. ［**ファイル**］>［**開く**］を選びます。
2. アドインファイルのある場所に移動し、選択します。
3. 次のいずれかのオプションを選択します。
   - Windowsの場合、［**開く**］ボタンの右の矢印をクリックし、［**アドインビルダーを使って開く**］を選択します。
   - Macintoshの場合、［**開いた後に編集**］オプションを選択します。
4. ［**開く**］をクリックします。

   アドインビルダーでファイルが開きます。引数を更新し、変更を保存します。

アドインの共有

.jmpaddinファイルとして保存したアドインは、他のユーザに配布できます。.jmpaddinファイルを電子メールで送付するか、またはネットワークフォルダや、(インターネット上のJMP User Communityにある)JMP File Exchangeなどにアップロードしてください。

JMPユーザが.jmpaddinファイルを開くと、ファイルが適切なフォルダ内に展開され、アドインの登録とインストールが行われます。そして、JMPの［アドイン］メニューから、そのスクリプトを起動できるようになります。

複数のアドインのインストール

複数のアドインをインストールしたい場合は、アドインを次の場所にコピーします。

- Windowsのアドインファイルの場所

  - %ALLUSERSPROFILE%¥SAS¥JMP¥AddIns（このコンピュータを使うすべてのユーザがアドインにアクセスできる）
  - %LOCALAPPDATA%¥SAS¥JMP¥AddIns（このコンピュータの現在のユーザのみがアドインにアクセスできる）
- Macintoshのアドインファイルの場所
  - /Library/Application Support/JMP/AddIns（このコンピュータを使うすべてのユーザがアドインにアクセスできる）
  - ~/Library/Application Support/JMP/AddIns（このコンピュータの現在のユーザのみがアドインにアクセスできる）

JMPは起動の際に、「addinRegistry.xml」ファイルを読み込みます。このファイルには、これまでに登録されたJMPアドインの情報が含まれています。続いて、JMPはアドインフォルダ内のその他のアドインを検出し、自動的にインストールします。

次の点を念頭に置いてください。

- アドインのホームフォルダは、「addin.def」ファイルを含んでいるフォルダである必要はありません。つまり、「addin.def」ファイルを含むフォルダ以外の場所に、スクリプトなどのファイルを保存してもかまいません。実際のアドインファイルがある場所は、homeオプションで指定できます。
- 新たに自動検出されたアドインのIDが、以前に明示的に登録されたアドインのIDと同じである場合、自動的に検出されたアドインの方が使用されます。

JSLを使ったアドインの登録

アドインが.jmpaddinファイル形式になっていない場合も、JSL関数の`Register Addin()`を使って、addin.defファイルに基づいて、アドインを手動で登録できます。この関数は、アドインのインストールと登録を行います。

- JSL関数について詳しくは、『スクリプト構文リファレンス』の「JSL関数」章を参照してください。
- 「addin.def」ファイルの作成方法については、「手動でのアドインの作成」（688 ページ）を参照してください。

次の点を念頭に置いてください。

- JMPが、指定のホームフォルダ内で「addin.def」ファイルを検出した場合、`Register Addin()`関数で指定されていないオプションの引数については、そのファイルの値が使用されます。
- `Register Addin()`関数で指定されていない値のみ、「addin.def」ファイルの値が使用されます。関数による引数の指定は開発中には便利な機能かもしれませんが、通常、アドインを登録するには「addin.def」ファイルだけで十分です。

手動でのアドインの作成

「addin.def」ファイルは、JMP アドインの登録情報を示す名前と値を含んだテキストファイルです。「addin.def」ファイルに含まれる名前と値は、次のとおりです。

name （オプション）表示名。アドインがGUIに表示されるときの名前です。制限のあるID名の代わりになる、ユーザにわかりやすい名前をつけてください。この表示名は、翻訳名が指定されていない場合、また、翻訳が指定された言語以外でJMPが実行されている場合に使用されます。アドインビルダーの**「アドイン名」**に該当します。

name_xx （オプション）表示名をさまざまな言語に翻訳できます。*xx*は、各言語の2桁のISO 639-1コードです。翻訳名を指定しても、翻訳名を指定しなかった地域設定の下でJMPが実行される場合に備えて、言語に左右されない名前も指定しておきましょう。アドインビルダーでは使用できません。

id 必須。アドインの一意のID。最大64文字まで使用できます。文字列の最初は文字でなければならず、文字、数字、ピリオド、下線を使用できます。一意である確率を上げるために、DNSの逆の名前を使用することをお勧めします。アドインビルダーの**「アドインID」**に該当します。

home （オプション）アドインファイルのパス。指定されていない場合、「addin.def」ファイルがあるフォルダが、アドインのホームフォルダとなります。ホームフォルダが別の場所（たとえば、ネットワーク上の共有フォルダなど）にある場合は、このhomeオプションで指定する必要があります。アドインビルダーでは使用できません。

home_win （オプション）Windows で JMP を実行する場合に使用されるアドインファイルのパス。Windows上のホームに値が指定されている場合、home_winの値で上書きされます。アドインビルダーでは使用できません。

home_mac （オプション）Macintosh版JMPでのアドインファイルのパス。Macintosh上でhomeに値が指定されている場合、home_macの値で上書きされます。アドインビルダーでは使用できません。

autoLoad （オプション）ブール値。デフォルト値は、1（真）。JMPの起動時にアドインを自動的にロードするかどうかをあらかじめ設定しておくことができます。アドインビルダーの**「保存後にインストール」**に該当します。

host （オプション）有効な値はWinおよびMac。アドインビルダーの**「動作するホスト」**に該当します。

minJMPVersion （オプション）指定できる値は、アドインが使用できる最低のJMPメジャーバージョンを表す整数。アドインビルダーの**「最低限必要なJMPバージョン」**に該当します。

maxJMPVersion （オプション）指定できる値は、アドインが使用できる最高のJMPメジャーバージョンを表す整数。この設定は、アドインとJMPの特定のバージョンとの間に互換性がないことがわかっている場合**のみ**使用してください。以降のJMPバージョンについては、アドインの新バージョンを提供する必要があります。アドインビルダーでは使用できません。

「addin.def」ファイルの例

```
id="com.mycompany.myaddin"
name="My Add-In's Friendly Name"
name_fr="My Add-In's French Name"
name_de="My Add-In's German Name"
```

```
home="¥¥server¥share¥myjmpaddin"
Autoload=1
MinJMPVersion=9
```

JMPアドインの例

以下のフォルダに、Simple Calculator.jmpaddinという名前のサンプルアドインが用意されています。

- Windowsの場合: C:¥Program Files¥SAS¥JMP¥13¥Samples¥Scripts
- Macintoshの場合: /Library/Application Support/JMP/13/Sample/Scripts

メモ: WindowsでJMP Proを使用している場合は、「JMP」の部分が「JMPPro」になります。JMP Shrinkwrap（シングルユーザーライセンス）の場合は、「JMP」の部分が「JMPSW」になります。

アドインの内容を見るには、拡張子を.zipに変更し、それを新しいフォルダ内に展開します。アドインの動作を確認するには、拡張子を.jmpaddinに変更し、インストールします。

アドインには次のファイルが含まれています。

addin.def

アドインをJMPに登録するための設定です。次の2行が含まれています。

```
id="com.jmp.sample.calculator"
name="Simple Calculator"
```

addin.jmpcust

対話式にカスタムメニューを構築する際に作成されるメニューカスタマイズファイルです。この例では、アドインメニュー項目をデフォルトの［**アドイン**］メニューに配置します。

calculator.jsl

単純な計算機のJSLスクリプト

calc_icon.gif

イメージファイル。計算機のアイコンとして使用されています。

アドインは、https://community.jmp.com/t5/File-Exchange/ct-p/FileExchangeからダウンロードできます。

JMPアドインの管理

アドインのインストール

JMPアドインのファイルは、.jmpaddinという拡張子を持ちます。JMPアドインをインストールするには次の2つの方法があります。

1. ［**ファイル**］>［**開く**］を選びます。
2. .jmpaddinファイルのある場所に移動し、選択します。
3. ［**開く**］をクリックします。

.jmpaddinファイルをダブルクリックするか、ファイルをJMPホームウィンドウにドラッグしてインストールすることもできます。

アドインの表示

［**表示**］>［**アドイン**］を選択すると、インストール済みのアドインを確認できます。

アドインの更新

すでにインストールしてあるアドインの更新版を入手した場合は、元のアドインと同じ手順でインストールできます。更新版をインストールすれば、古いバージョンが自動的に上書きされます。

アドインの無効化、有効化、削除

アドインを、削除せずに一時的に無効にするには

1. ［**表示**］>［**アドイン**］を選びます。
2. 無効にしたいアドインを選択します。
3. ［**有効**］チェックボックスをオフにします。

無効にしたアドインを有効にするには

1. ［**表示**］>［**アドイン**］を選びます。
2. 有効にしたいアドインを選択します。
3. ［**有効**］チェックボックスをオンにします。

アドインの登録を解除するには

1. ［**表示**］>［**アドイン**］を選びます。
2. 削除したいアドインを選択します。
3. ［**登録解除**］をクリックします。

第16章

プログラム例の紹介
サンプルによるプログラミングの学習

プログラミングを勉強するには、実際のプログラムコードを見るのが一番良い方法でしょう。この章では、日付時間値の変換や、レポートからの数値の取得など、JSLによるプログラミングで良く行われる処理を、実例にもとづき紹介します。JMPのインストール時に、このような処理を扱うサンプルスクリプトが「Samples¥Scripts」フォルダに保存されるため、いつでも自分でスクリプトを実行することができます。

起動時のスクリプトの実行

JMPの起動時に必ず実行したいスクリプトにはjmpStart.jslという名前を付け、使用しているオペレーティングシステムに応じて、次のフォルダに格納します。起動時に、JMPはここにリストされた順番でフォルダ内のjmpStart.jslスクリプトを検索します。最初に検出されたjmpStart.jslスクリプトが実行され、検索は直ちに停止します。

メモ: この節で言及しているパス名の一部には、「JMP」というフォルダ名を使用しています。WindowsでJMP Proを使用している場合は、「JMP」の部分が「JMPPro」になります。JMP Shrinkwrap（シングルユーザーライセンス）の場合は、「JMP」の部分が「JMPSW」になります。

Windows の場合 :

1. C:¥Users¥<ユーザ名>¥AppData¥Roaming¥SAS¥JMP¥13
2. C:¥Users¥<ユーザ名>¥AppData¥Roaming¥SAS¥JMP

Mac の場合 :

1. /Users/<ユーザ名>/Library/Application Support/JMP/13
2. /Users/<ユーザ名>/Library/Application Support/JMP

jmpStart.jslスクリプトは、コンピュータ上の特定のユーザに対してのみ実行されます。jmpStartAdmin.jslという名前をつけたスクリプトを、使用しているオペレーティングシステムに応じて次のいずれかの場所に格納できます。このスクリプトは、コンピュータ上のすべてのユーザに対して実行されます。JMPはまず管理者のスタートアップスクリプトを検索し、見つかればそれを実行します。次にユーザのスタートアップスクリプトを検索し、見つかればそれを実行します。

Windows の場合 :

1. C:¥ProgramData¥SAS¥JMP¥13
2. C:¥ProgramData¥SAS¥JMP

Mac の場合 :

1. /Library/Application Support/JMP/13
2. /Library/Application Support/JMP

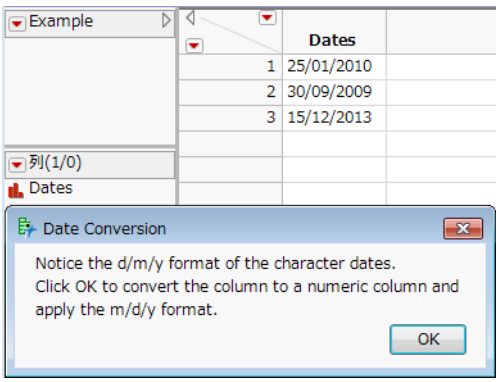
文字の日付を数値の日付に変換

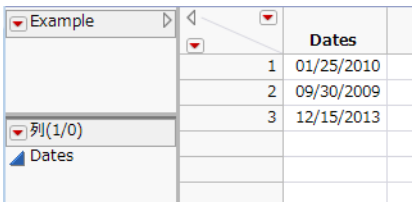
データテーブルの列において、自分は「数値」としてデータを扱いたいのに、列プロパティで見ると、データタイプが「文字」に設定されてしまっている場合があります。データを日付時間値として処理するには、列を数値列に変換し、値の表示形式を指定します。

「Convert Dates.jsl」は、まず、データテーブルを作成します。次に、データ入力形式を指定し、データタイプを「文字」から「数値」に、尺度を連続尺度に変更します。最後に、その列に、m/d/y形式を適用します（図16.1）。

```
// Create a data table with character dates.
dt = New Table( "Example",
   Add Rows( 3 ),
   New Column( "Dates",
      Character,
      Nominal,
      Set Values( {"25/01/2010", "30/09/2009", "15/12/2013"} )
   )
);

// Display a modal dialog for the user to confirm the format conversion.
nw = New Window( "Date Conversion",
   <<Modal,
   tb = Text Box(
      "Notice the d/m/y format of the character dates.
Click OK to convert the column to a numeric column and apply the m/d/y format."
   )
);

/* Apply the Numeric data type.
Specify the Informat (input format) value "d/m/y".
Specify the Format (display format) value "m/d/y".
Apply the Continuous modeling type */
col = Column( dt, "Dates" );
col << Data Type( "Numeric", Informat( "d/m/y" ), Format( "m/d/y" ) );
col << Modeling Type( "Continuous" );

// Display the data table in front of the script.
dt << Data Table Window();
```

図16.1 文字の日付の変換（変換前と変換後）





列のデータタイプを文字から数値に変更した場合は、データの入力形式の定義が重要になります。この例では、Informat("d/m/y") が入力形式を定義します。Format("m/d/y") は、表示形式を定義します。Informat() を省略した場合、Format() 値は入力形式と表示形式の両方として適用されます。この例の場合は、結果としてデータが欠測値となります。

Convert Dates.jsl に変更を加え、確認してみましょう。

1. サンプルスクリプトのフォルダにある「Convert Dates.jsl」を開きます。
2. スクリプトウィンドウを右クリックし、[**行番号を表示する**] を選択します。
3. 9行目で「25/01/2010」を「01/25/2010」に変更します。
4. 27行目で「Informat("d/m/y"),」を（カンマも含め）削除します。
5. スクリプトを実行します。

   Format("m/d/y") が列に適用されます。列には「01/25/2010」だけが表示されます。「30/09/2009」と「15/12/2013」は有効な m/d/y 値ではないため、欠測値となります。

日付によるデータ抽出

次のプログラムは、日付データを扱う例です。JMP では、日付や時刻に関して、多数の機能が用意されています。次の例は、元のデータから、特定の期間のデータだけを抽出する例です。

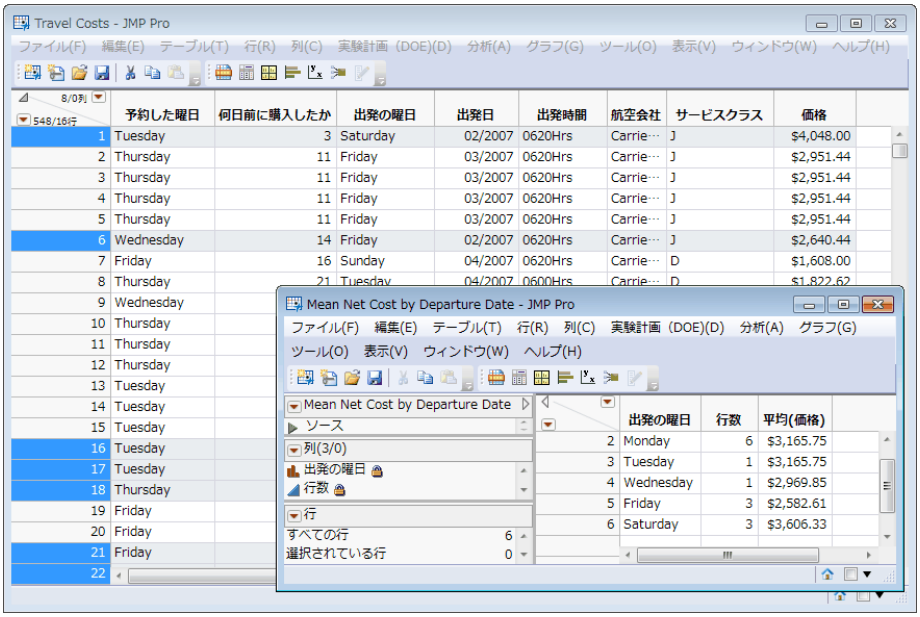
「Select Where Using Dates.jsl」は、出発日（Departure Date）の列をもとに、MDY 関数を利用して、特定の期間だけを含んだデータを作成します。そして、そのデータから、正味費用（Net Costs）の曜日（Departure Day of Week）ごとの平均を求めています（図16.2）。

なお、この例では、データテーブルの列名に英語名が使われています。「Travel Costs.jmp」サンプルデータの列には、英語名と日本語名が与えられています。日本語 JMP でこのデータテーブルを開いた場合、日本語名が列名として表示されます。しかし、プログラムでは英語名も用いることができます。

```
/*  How can you work with dates in JSL? JMP provides a number of formats
that you can use to make comparisons and then subset data based on the date.
*/

hdt = Open( "$SAMPLE_DATA/Travel Costs.jmp" );

/* Apply the Date MDY format to Departure Date values and then select only
February dates.*/
hdt << Select Where(
      (Date MDY( 02, 01, 2007 ) <= :Departure Date < Date MDY( 03, 1, 2007 ))
      );

/* Subset the selected rows into two tables: one table contains February
departure dates, the other contains all data for those departure dates.*/
nt1 = hdt << Subset( Columns( :Departure Date ),
```

```
   Output Table Name( "February Departure Date" ) );
nt2 = hdt << Subset( Output Table Name( "February Data" ) );

/* Create a summary table, grouping mean cost by day of week that departure
took place.*/
sumDt = nt2 << Summary(
   Group( :Departure Day of Week ),
   Mean( :Net Cost ),
   Output Table Name( "Mean Net Cost by Departure Date" )
);
```

図16.2 元のテーブルと最終的な要約テーブル



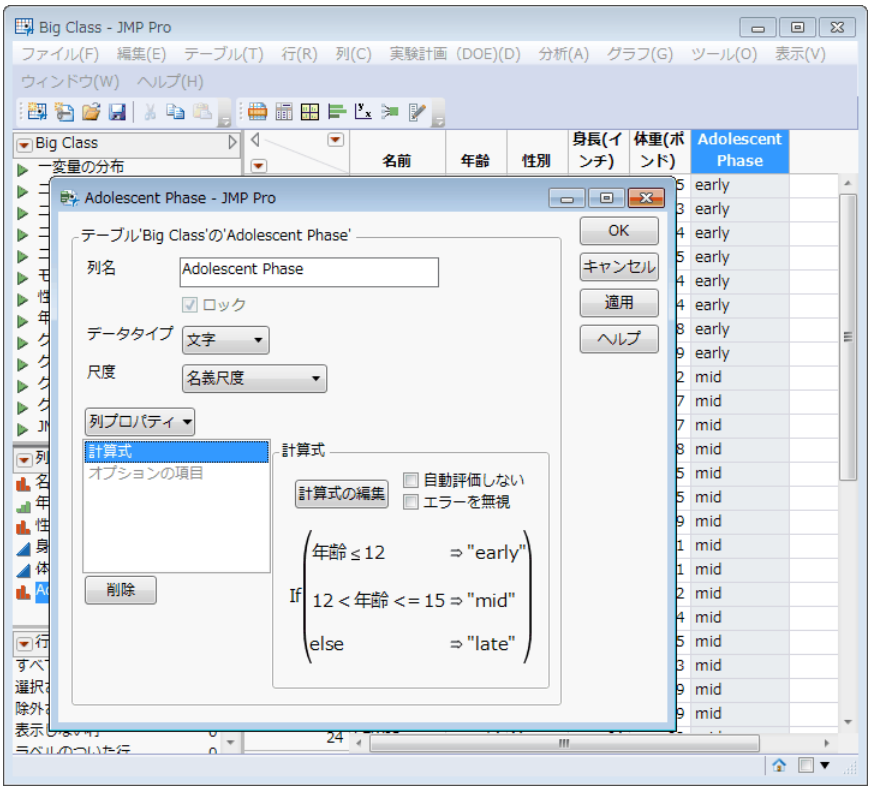
計算式を含んだ列の作成

この例では、If文の計算式を含んだ列を、データテーブルに追加します。「Create a Formula Column.jsl」は、「Big Class.jmp」サンプルデータにおいて、「年齢」(Age)列に対するIf文の結果を含む列を、新規に作成します（図16.3）。

```
/* Scenario:

How do you create a formula column that combines conditional expressions
with value comparisons? This script shows how to create a new formula
column that evaluates ages in Big Class.jmp and returns the conditional
result in the new column.*/
```

```
dt = Open( "$SAMPLE_DATA/Big Class.jmp" );

/* Create a new character column for the formula.
Insert "early" in the new column if the age is less than or
equal to 12.Insert "mid" if the age is less than or equal to
15 but greater than 12.For ages greater than 15, insert "late".
*/

dt << New Column( "Adolescent Phase",
   Character,
   Formula(
      If( :age <= 12, "early",
         12 < :age <= 15, "mid",
         "late"
         )
      )
);
```

図16.3 計算式内の条件式



分析結果の一部を抜き出す

この例では、レポートの一部だけを抜き出し、別のウィンドウに表示します。

JMPの［分析］と［グラフ］の各メニューで実行されるプラットフォームには、分析レイヤーとレポートレイヤーという2つのオブジェクトがあります。分析レイヤーのオブジェクトに特定のメッセージを送ることにより、ユーザの求めている結果が出力されます。

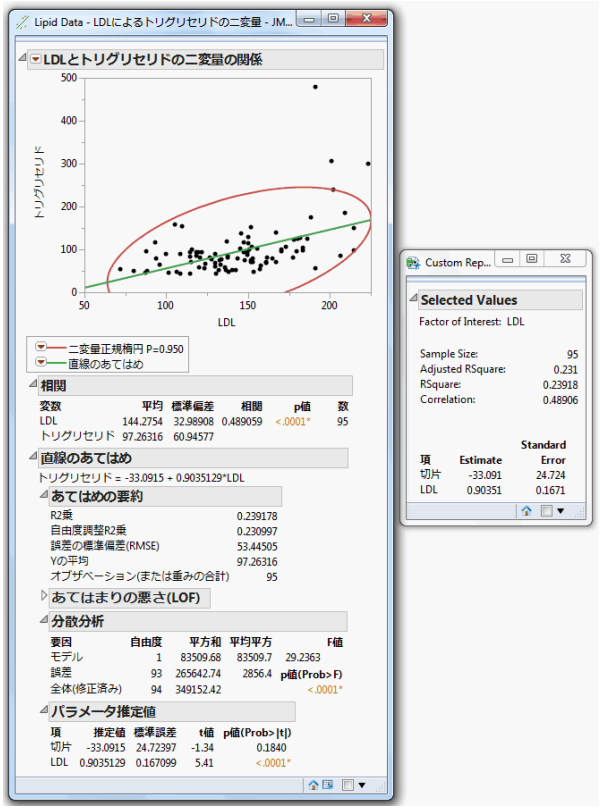
「Extract Values from Reports.jsl」は、「二変量の関係」プラットフォームから、標本サイズ、R2乗、相関などの一部の分析結果だけを取得し、それを別の新たなウィンドウに表示します。図16.4は、元の「二変量の関係」レポートと、カスタマイズしたレポートです。なお、スクリプトが終了した時点で、「二変量の関係」レポートのウィンドウは閉じられます。

```
/* Scenario: How do you capture specific results of an analysis
in a report using JSL?

The JMP platforms in the Analyze and Graph menus contain two objects
known as the analysis and report layers.

Messages are sent to the analysis layer that generate the desired results.

This script performs a Bivariate analysis and shows results
such as the sample size, RSquare, and Correlation in a new report window.
*/

sd = Open( "$SAMPLE_DATA/Lipid Data.JMP" );

biv = Bivariate(        // biv is the analysis layer.
   Y( :Triglycerides ),
   X( :LDL ),
   Density Ellipse( 0.95, {Line Color( {213, 72, 87} )} ),
   Fit Line( {Line Color( {57, 177, 67} )} ),
);

// Make sure the second Outline Box (called "Correlation")
// in the Bivariate report is open.You can then see which content
// is extracted into the Custom report.
report(biv) [Outline Box( 2 )] << Close( 0 );

reportbiv = biv << Report;  // reportbiv is the report layer.

// The density ellipse is generated first.
// Extract the correlation coefficient.
corrvalue = reportbiv[Outline Box( 2 )][Number Col Box( 3 )] << Get( 1 );

// ...followed by Fit Line
```

```
// Extract the numeric values from the Summary of Fit report
// and place them in a matrix.
sumfit = reportbiv[Outline Box( 4 )][Number Col Box( 1 )] << Get as Matrix;

// Extract the values of RSquare and AdjRSquare as one by one matrices.
rsquare = sumfit[1];
adjrsq = sumfit[2];
avg = sumfit[4];
samplesize = sumfit[5];

// Extract the first column of the Parameter.
// Estimates report as two objects.
term = reportbiv[Outline Box( 7 )][String Col Box( 1 )] << Get();

// Clone the report layer as a String Col Box.
cloneterm = reportbiv[Outline Box( 7 )][String Col Box( 1 )] << Clone Box;

// Extract the Parameter Estimates values as a matrix.
est = reportbiv[Outline Box( 7 )][Number Col Box( 1 )] << Get as Matrix;

// Extract the Standard Error values as a matrix.
stde = reportbiv[Outline Box( 7 )][Number Col Box( 2 )] << 7Get as Matrix;

dvalues = [];
dvalues = samplesize |/ adjrsq |/ rsquare |/ corrvalue;
sfactor = term[2];

dlg = New Window( "Custom Report",
   Outline Box( "Selected Values",
      /* The Lineup box defines a two-column layout, each of which contains
      a Text Box.*/
      Lineup Box( N Col( 2 ),
      Text Box( "Factor of Interest: " ),
      Text Box( sfactor ), ),
      tb = Table Box(
         /* Display an empty string in the first column
            and the text in the second column.*/
         String Col Box( "  ",
            {"Sample Size: ", "Adjusted RSquare: ", "RSquare: ",
            "Correlation:"}
         ),
         // Insert a 30 pixel x 30 pixel spacer between the columns.
         Spacer Box( Size( 30, 30 ) ),
         ,
         /* Display an empty string in the first column
         and the dvalues in the second column.*/
```

```
            Number Col Box( "  ", dvalues )
         ),
         // Insert a 1 x 30 spacer.
         Spacer Box( Size( 0, 30 ) ),
         ,
         Table Box(
            /* Display the cloned String Col Box followed by a spacer.
               Then insert the Parameter Estimates and Standard Error values.*/
            CloneTerm,
            Spacer Box( Size( 10, 0 ) ),
            ,
            Number Col Box( "Estimate", est ),
            Spacer Box( Size( 10, 0 ) ),
            ,
            Number Col Box( "Standard Error", stde )
         )
      )
);
tb << EXTSet Shade Headings( 0 ); // Turn off shaded table headings.
tb << Set Heading Column Borders( 0 ); // Turn off table column borders.
Close( sd ); // Close the data table.
```

図16.4 二変量分析の結果のカスタマイズレポート



対話型プログラムの作成

この例では、ユーザに数値を入力させ、それを元に計算を実行し、結果を新しいウィンドウに表示します。

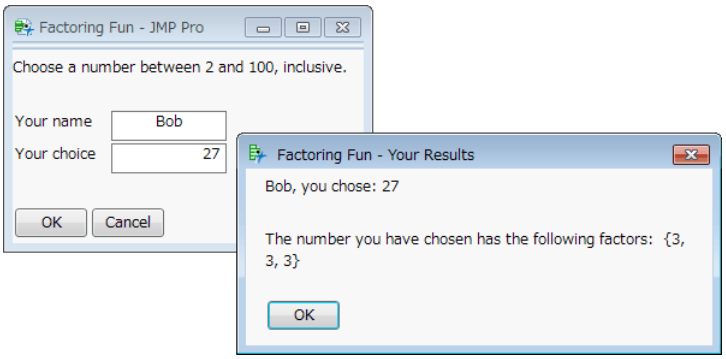
「Prime Numbers.jsl」は、ユーザに整数の入力を促し、素因数分解の結果、もしくは、その整数は素数であるというメッセージを、新しいウィンドウに表示します（図16.5）。このスクリプトでは、異なる種類のディスプレイボックスを整列したり、テキストを結合したり、条件関数を用いたりしています。

```
/* How do you gather numeric input from the users, perform a
calculation on that input, and show the results in a new window?

This script demonstrates how to create an interactive program that
asks the user to enter a number and then factors the number or confirms
it as a prime number.*/

// Ask the user to enter a name and number.
nw = New Window( "Factoring Fun",
```

```
V List Box(
   Text Box( "Choose a number between 2 and 100, inclusive." ),
   Spacer Box( Size( 25, 25 ) )
    ),
    V List Box(
        Lineup Box(
            2,
            Text Box( "Your name   " ),
            uname = Text Edit Box( "< name > ", << Justify Text( Center ) ),
            Text Box( "Your choice " ),
            uprime = Number Edit Box( 2 )
        ),
        Spacer Box( Size( 25, 25 ) ),
        H List Box(
            Button Box( "OK",
          // Unload responses.
                username = uname << Get Text;
                fromUser0 = uprime << Get;

         // Test input for out of range condition.
                If( fromUser0 <= 1 | fromUser0 > 100,
           // Send message to user that input value is out of range.
                    nw2 = New Window( " Factoring Fun: Message for "
                        || username,
                        <<Modal,
                        Text Box(
                            "The number you chose, " || Char( fromUser0 ) ||
                            " is not between 2 and 100, inclusive.Please try
                              again."
                        ),
                        Button Box( "OK" )

                    ),   // Else the number is within range.
                    // Test for a prime number.If not prime, factor it.
                    // Create a vector which holds the prime numbers
                     // within specified range.
                    primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37,
                     41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97];
                    // Count the number of primes in the vector.
                    p# = N Row( primes );

                    isprime = 0; // Set flag.
                     // Make a copy of the value for processing.
                     fromuser1 = fromuser0;
                    factors = {}; // Initialize list.

                     // Process the value by checking for prime then
```

```
                        // factoring if needed.
                    While( isPrime == 0,

                        // Compare value to vector of prime numbers.
                        If( Any( fromuser0 == primes ),
                        // If found, place value in factor list.
                            Insert Into( factors, fromUser0 );
                            isPrime = 1 // Set condition to exit While loop.
                            ;
                        ); // End For loop.

                        If( isprime == 0,
                            For( q = 1, q <= p#, q++,
                                If( Mod( fromuser0, primes[q] ) == 0,
                                    fromUser0 = fromUser0 / primes[q];
                                    Insert Into( factors, primes[q] );
                                    q = p# + 1 // End if-then loop.
                                    ;
                                ); // End If loop.
                            ); // End For loop.
                        );  // End If/Then loop.

                    ); // End while loop.

                    cfUser0 = Char( fromUser1 );
                    nf = N Items( factors );
                    If( nf >= 2,
                        fmsg = "The number you have chosen has the following
                          factors: ",
                        fmsg = "The number you have chosen is a prime number: "
                    );
                // Show message to user about results.
                    nw3 = New Window( " Factoring Fun - Your Results",
                        <<Modal,
                        Text Box( username || ", you chose: " || cfUser0 ),
                        Spacer Box( Size( 25, 25 ) ),
                        Text Box( fmsg || " " || Char( factors ) ),
                        Spacer Box( Size( 25, 25 ) ),
                        Button Box( "OK" )
                    );
                );
            ), // End the main OK button script.
            // Close the window and the program.
            Button Box( "Cancel", nw << Close Window )
        )
    )
);
```

図16.5 素因数分解の結果



付録 A

互換性に関するメモ
JMP 13.1 における JMP 13 からの変更点

この付録では、JSL スクリプトの互換性に影響する変更点を取り上げます。

互換性の問題

パーティションプラットフォームの Dispatch メッセージ

Dispatch メッセージの中で使用されていた `Frame Box` の名前が「Partition」から「Partition Report」に変更されました。以前のバージョンで「パーティション」プラットフォームから保存したスクリプトは、更新する必要があります。

行の削除

`Delete Rows` は、選択されている行を削除するだけでなく、削除した行の数を戻すようになりました。これまでは何も戻しませんでした。

混合モデル

「Exchangeable（交換可能）」な共分散構造が「Compound Symmetry（複合対称）」という名前に変わりました。また、モデルに分散成分しか含まれない場合の「変量効果の共分散パラメータ推定値」レポートでは、「共分散パラメータ（Covariance Parameter）」列が「分散成分（Variance Components）」という名前に変わりました。以前のバージョンで「混合モデル」プラットフォームから保存したスクリプトは、更新する必要があります。

付録 B

用語集
用語、概念、表記

| 構文の要約において、「|」は「または」を意味し、複数のオプションを分けるために用いられます。通常、「|」で分かれたオプションは互いに排他的です。つまり、選べるものは1つだけで、いくつも選ぶことはできません。

col 構文の要約で、データテーブルの列への参照を意味する表記です。例: `Column("年齢")`。

db 構文の要約で、ディスプレイボックスへの参照を意味する表記です。例: `report(Bivariate[1])`

dt 構文の要約で、データテーブルの列への参照を意味する表記です。例: `Current Data Table()`、`Data Table("Big Class.jmp")`。

obj 構文の要約で、分析プラットフォームへの参照を意味する表記です。例: `Bivariate[1]`

ODBC データベース Open DataBase Connectivity(ODBC) はマイクロソフトによる規格です。JSL では、`Open Database` コマンドを使って、ODBC に対応したあらゆるデータソースにアクセスできます。

POSIX POSIX は Portable Operating System Interface の頭字語で、IEEE の登録商標です。POSIX パス名は JMP が動作するどのオペレーティングシステムでも使用できるため、オペレーティングシステムごとにパス構文を使い分ける必要がありません。

予め評価された統計量 一度計算され、以後定数として使われる統計量です。

演算子 通常、演算子は1文字か2文字の記号で表されます。例: 加算を示す「+」や、以下を示す「<=」

オブジェクト オブジェクトは、JMP の動的なエンティティで、たとえばデータテーブル、データ列、プラットフォーム結果ウィンドウ、グラフなどがこれにあたります。ほとんどのオブジェクトは、自身に対して何らかのアクションを実行するよう指示するメッセージを受け取れます。

関数 関数名に続く括弧の中に引数または引数のリストをとります。たとえば、二項演算子 + は `Add()` と等価です。また、ステートメント `3 + 4` と `Add(3, 4)` は等価です。JSL の演算子のすべてに、等価な関数がありますが、関数の中には等価な演算子がないものもあります。たとえば、`Sqrt(a)` は関数でしか表現できません。関数をある名前で保存する `Function` も参照してください。

行の属性 データ行の属性のあらゆる組み合わせを格納するデータ要素の型のことです。属性には除外する、表示しない、ラベルあり、選択されている、色、マーカー、色の濃淡、色相があります。

行列 行列は、数値の行と列から成る長方形配列で、JMP のデータの型の1つです。JSL では、行列は大括弧（`[ ]`）表記か、`Matrix` 演算子で作成します。

グローバル変数 グローバル変数とは、セッションの中で存在し続ける名前です。グローバル変数には、たとえば、数値、文字列、リスト、オブジェクトの参照など、さまざまなタイプの値を入れることができます。

グローバル変数と呼ばれるのは、特定のコンテキストだけでなく、ほとんどどこででも参照できるためです。

現在行 スクリプト操作の対象となる行の番号です。デフォルトではゼロ（行なし）になっています。Row()やFor Each Rowなどで現在行を設定できます。

現在のデータテーブル 現在のデータテーブルとは、Current Data Table()で指定されているデータテーブルのことです。

コマンド 処理を行うJSLステートメントの一般表現です。このマニュアルでは、**演算子**、**関数**または**メッセージ**のように、具体的な用語の方をできるだけ使っています。

左辺値 左辺値（L-value）とは、値を代入することができる式のことです。このマニュアルにおいて、「左辺値」の式とは、現在の値を戻すこともできるが、代入演算によって値を設定することもできる式を指します。たとえば、Row()関数は、現在行の通し番号を求め、これをx=Row()のように他の変数に割り当てることができます。例：x=Row()。しかし、Row()関数は左辺値であるので、代入演算の左辺に置いて、Row()=10のように値を設定することもできます。例：Row()=10。

参照 スクリプトで表現可能な**オブジェクト**に**メッセージ**を送るために、そのオブジェクトを示す方法です。例:column("年齢")、Current Data Table()、Bivariate[1]。一般に参照は便宜上、**グローバル変数**で保存されます。

省略（eliding）演算子 省略演算子は、両側のオペランドをまとめて評価する演算子で、厳密に左から右へ評価する場合とは結果が異なります。たとえば、12<a<13は*a*が12～13の範囲にあるかどうかを評価する式で、JMPでは式全体を読み込んでから評価します。<が省略演算子でない場合、式は左から右へ評価されます。たとえば、(12<a)<13は、括弧の中の比較がまず評価され、真の場合は1、偽の場合は0が戻されます。そして、その戻り値が13より小さいかを評価します。そのため、(12<a)<13の結果は、常に1（真）となります。object<<message のように使われる演算子 << も、省略演算子です（これは、Send(object, message)と等価）。

スカラー 行列ではない、ただの数値のことです。

スコープ演算子 スコープ演算子は名前を特定のデータの型と解釈させます。たとえば、:nameの演算子「:」は*name*が列であるとし、::nameの演算子「::」は*name*がグローバル変数であるとします。

接頭演算子 接頭演算子は、否定の!aのように、右側（演算子の後）に引数をとります。

接尾演算子 接尾演算子は、1ずつ加算するa++や1ずつ減算するa--のように、左側（演算子の前）にオペランドをとります。

データフィード データフィードとは、リアルタイムデータを連続して読み込む方法の1つです。リアルタイムデータとは、たとえばシリアルポートに接続された測定機器から読み込んだデータです。

データベース この用語は非常に広い意味で使われますが、JMPでは、JSLのOpen Databaseコマンドを使いODBCを通じてアクセスできる、あらゆる外部データソース（SQLなど）を意味します。

トグル 行の属性コマンドでブール値の引数を省略すると、その設定がトグルします。オプションがオフなら、メッセージがオンになります。オプションがオンなら、メッセージがオフになります。このようなコ

マンドを繰り返し送ることで、オンとオフの間を往復します。ブール引数を指定すれば、コマンドが明示的にオンまたはオフに設定するので、同じコマンドを繰り返し送っても逆の設定にはなりません。その他のメッセージでブール値の引数を省略した場合は、オプションがオンになります。

名前　名前は、JSLオブジェクトへの参照です。たとえば、グローバル変数に3という数値を割り当てる`a=3`というステートメントでは、「a」がグローバル変数の名前です。

名前空間　名前空間とは、一意の名前およびそれに対応する値の集まりです。名前空間は、異なるスクリプト間での名前の競合を回避するのに役立ちます。

名前付き引数　名前付き引数は、特定の名前によって明示的に定義される引数のことです。たとえば、`Graph Box`関数などにおける`title("My Histogram")`は、名前付き引数です。一方、`New Window`関数においては、`title`は位置指定の引数であり、第1引数の位置に必ず指定する必要があります。

二項演算子　二項演算子は、両側に1つずつオペランドをとるものです。たとえば数値演算3＋4の`+`、代入演算`a=7`の`=`などです。

引数　引数とは、JSLの演算子、関数、メッセージなどの括弧の中で指定するものです。たとえば、`Open("Big Class.jmp")`の引数は、Big Class.jmpです。

多くの場合、引数は、指定された位置によって、その意味が決められます。たとえば、`size(200, 100)`において、**200**と**100**は位置指定の引数であり、第1引数は幅、第2引数は高さと常に解釈されます。**名前付き引数**の項も参照してください。

ブール値　ブール値とは、はい／いいえのような2値のことです。たとえば、オン／オフ、表示／非表示、真／偽、1／0、または、はい／いいえです。**ブール演算子**は、真か偽か（または欠測しているか）を評価する演算子です。

ベクトル　1列だけ、または1行だけから成る行列のことです。

マウスアップ　マウスのボタンを放すと実行できるイベントです。「Handle()」（558ページ）および「Mousetrap()」（562ページ）を参照してください。

マウスダウン　マウスのボタンを押すと実行できるイベントです。「Handle()」（558ページ）および「Mousetrap()」（562ページ）を参照してください。

メタデータ　JMPのデータテーブルでは、メタデータとはデータを記述したデータのことです。たとえば、データソース、列のノート、テーブルスクリプトなどです。

メッセージ　メッセージはJSLのステートメントで、実行できる**オブジェクト**へ送られます。

リスト　リストとは、項目をいくつも含むデータの型です。リストは、中括弧（{}）を用いた表記か、`List`演算子で作成します。リストを使えば、たくさんの項目をスクリプトで一度に扱えます。

索引
スクリプトガイド

記号

数字

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W-Z

ア

イ

ウ

エ

オ

カ

キ

ク

ケ

コ

サ

シ

ス

セ

ソ

タ

チ

ツ

テ

ト

ナ

ニ

ヌ

ネ

ノ

ハ

ヒ

フ

ヘ

ホ

マ

ロ

ワ