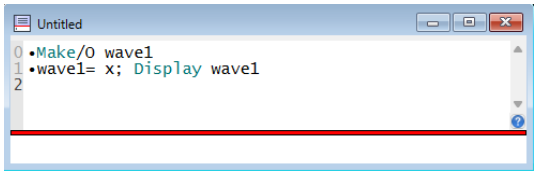
CONTENTS

ビジュアルヘルプ – コマンド

コマンドを使っての作業



コマンドを実行するには、コマンドラインにコマンドを入力し、Enter キーを押します。

コマンドの入力にはノートブックも使用できます。
ヘルプ Notebooks as Worksheets を参照してください。

コマンドを最初から入力することもできますが、多くの場合は Igor のダイアログでコマンドを作成して実行することになります。
コマンド ウィンドウの履歴エリアで、これまで実行した操作の履歴を確認できます。
また、そこに保存されたコマンドを簡単に再入力、編集、再実行できます。
詳細については、ヘルプ Command Window を参照してください。

複数のコマンド

コマンドをセミコロンで区切ると、１行に複数のコマンドを記述することができます。
例えば、次のように記述します：

```
wave1= x; wave1= wave2/(wave1+1); Display wave1
```

最後のコマンドの後にセミコロンは必要ありませんが、あっても問題ありません。

コメント

コメントは // で始まり、コマンドラインの実行部分を終了します。
コメントは行の終わりまで続きます。

コマンドの最大長

コマンドラインの長さは 2500 バイトを超えてはなりません。

コマンドラインには行継続文字はありません。
しかし、ほとんどの場合、中間変数を使用することで１つのコマンドを複数の行に分割することができます。
例えば、次のようにします：

```
Variable a = sin(x-x0)/b + cos(y-y0)/c
```

は次のように書くことができます：

```
Variable t1 = sin(x-x0)/b
Variable t2 = cos(y-y0)/c
Variable a = t1 + t2
```

パラメーター

Igor が数値パラメーターを期待するコマンド内のあらゆる場所で、数値式を使うことができます。
同様に、Igor が文字列パラメーターを期待するあらゆる場所で、文字列式を使うことができます。
コマンドのフラグ（/N=<番号> など）では、式を括弧で囲む必要があります。
詳細は、「パラメーターとしての式」のセクションを参照してください。

リベラルオブジェクト名とプログラミング

一般的に、Igor のオブジェクト名は、制限された文字セットに限定されています。
文字、数字、およびアンダースコア文字のみ使用できます。
このような名前は「標準名」と呼ばれます。
この制限は、コマンドで使う時に、Igor がその名前がどこで終わるかを判別するために必要です。

ウェーブとデータフォルダーのみ、「リベラルな（自由な）」名前も使用できます。
リベラル名には、スペースやドットを含むほぼすべての文字を使用できます（詳細については、ヘルプ Liberal Object Names を参照してください）。
ただし、Igor がリベラル名の終わりを認識できるように、それらを単一引用符で囲む必要があります。

次の例では、ウェーブ名にはスペースが含まれているため、引用符で囲む必要があります。

```
'wave 1' = 'wave 2'                    // 正しい
wave 1 = wave 2                        // 誤り - リベラル名はクオートが必要
```

(この構文は、コマンドラインおよびマクロにのみ適用されます。ウェーブを読み書きするには、ウェーブ参照を使わなければならないユーザー定義関数には適用されません。)

注記： リベラル名を提供するには、Igor プログラマーによる追加の作業とテストが必要となります（ヘルプ Programming with Liberal Names を参照)。
そのため、ユーザー定義のプロシージャでリベラル名を使うと、問題が発生する場合があります。

コマンドとデータフォルダー

データフォルダーは、異なるデータセットが互いに干渉しないように分離するための手段です。
Data Browser（Data メニュー）を使って、データフォルダーを確認および作成することができます。
ルートデータフォルダーは常に存在し、多くのユーザーが必要とする唯一のデータフォルダーです。
上級ユーザーは、データを整理するために追加のデータフォルダーを作成することをお勧めします。

ウェーブおよび変数は、現在のデータフォルダー、特定のデータフォルダー、または現在のデータフォルダーからの相対位置にあるデータフォルダーで参照できます。

```
// wave1 はカレントのデータフォルダーにある
wave1 = <expression>

// wave1 は特定のデータフォルダーにある
root:'Background Curves':wave1 = <expression>

// wave1 はカレントのデータフォルダー内のあるデータフォルダーにある
:'Background Curves':wave1 = <expression>
```

（この構文は、コマンドラインおよびマクロにのみ適用されます。ウェーブを読み書きするには、ウェーブ参照を使わなければならないユーザー定義関数には適用されません。）

最初の例では、オブジェクト名（wave1）のみを使っており、Igor は現在のデータフォルダー内でそのオブジェクトを検索します。

２番目の例では、データフォルダーのフルパス（root:'Background Curves':）とオブジェクト名を使っています。Igor は、指定されたデータフォルダー内でオブジェクトを検索します。

３番目の例では、相対データフォルダーパス（:'Background Curves':）とオブジェクト名を使っています。Igor は、現在のデータフォルダー内で Background Curves という名前のサブデータフォルダーを探し、そのデータフォルダー内でオブジェクトを探します。

重要：代入文（代入文の項で説明）の右側は、宛先オブジェクトを含むデータフォルダーのコンテキストで評価されます。
例えば：

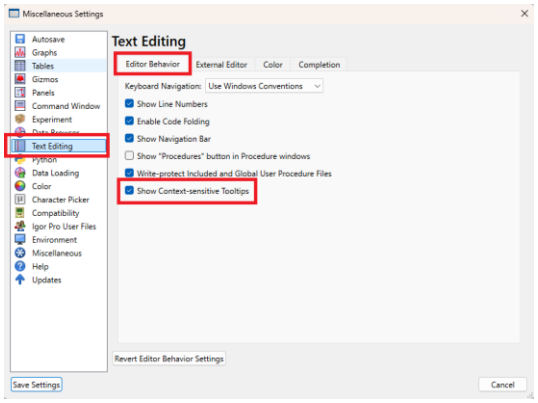
```
root:'Background Curves':wave1 = wave2 + var1
```

これが機能するには、wave2 および var1 が「Background Curves」データフォルダー内に存在している必要があります。

このヘルプファイルの残りの部分では、オブジェクト名のみを使っており、現在のデータフォルダー内のデータを参照しています。
データフォルダーの詳細については、「データフォルダー」を参照してください。

コマンドのツールチップ

プロシージャウィンドウおよびコマンドウィンドウで、操作や関数などのコマンドの上にマウスカーソルを置くと、Igor はそのコマンドに関する情報をツールチップで表示します。
WMButtonAction などの組み込み構造体の名前にカーソルを置くと、Igor はその構造体の定義を表示します。
この機能をオフにするには、Misc→Miscellaneous Settings を選択し、Text Editing セクションを選択、Editor Behavior タブを選択し、Show Context-sensitive Tooltips チェックボックスのチェックを外してください。

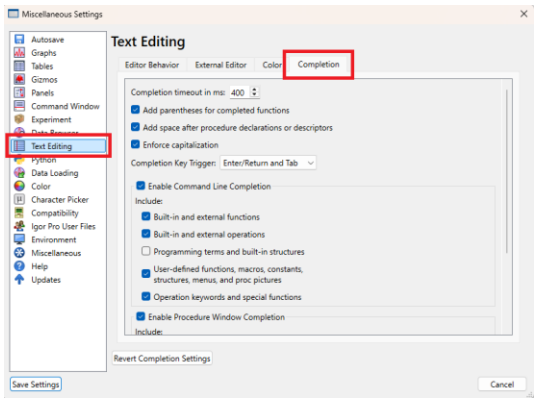


コマンドの補完

Procedure ウィンドウおよびコマンドラインは、コマンドの自動補完に対応しています。

コマンドの最初の数文字を入力すると、ポップアップが表示され、コマンドを素早く選択できます。
矢印キーを使って、補完オプションのリストを上下に移動します。
Enter キーまたは Tab キーを押して、ハイライト表示されたテキストを文書に挿入します。

マウスカーソルを補完オプションの上に移動させたり、矢印キーを使ってハイライトされたオプションを変更したりすると、選択されたオプションのテンプレートを表示するツールチップが表示されます。



コマンド補完の設定は、Miscellaneous Settings ダイアログの Text Editing カテゴリの Completion タブで調整できます。
プロシージャウィンドウとコマンドラインで、補完を個別に有効または無効にすることができます。
最後のキー入力から補完オプションのポップアップ表示までの遅延時間を調整できます。
また、カッコが必要なアイテムを挿入する時、開くカッコを自動的に追加するかどうかを設定できます。

ウェーブや変数などのオブジェクトの補完はサポートされていません。

コマンドの補完は文脈依存ではありません。
これは、入力中のテキストの文脈では意味を成さない補完オプションが提示されることがあることを意味します。
例えば、「Display/HID」と入力すると、次のような補完候補が表示されます：HideIgorMenus、HideInfo、HideProcedures、HideTools。
これらのオプションは Display コマンドのフラグとして有効ではありませんが、コマンド補完アルゴリズムはそれらをフィルタリングできません。

コマンドのタイプ

コマンドラインから実行できるコマンドには、根本的に異なる 3 つの種類があります：

- 代入文
- 操作コマンド
- ユーザー定義のプロシージャコマンド

それぞれの例は次のようになります：

```
wave1 = sin(2*pi*freq*x)                    // 代入文
Display wave1,wave2 vs xwave                // 操作コマンド
MyFunction(1.2,"hello")                     // ユーザー定義のプロシージャコマンド
```

Igor は、入力したコマンドを実行する時に、入力したコマンドが 3 つの基本コマンドのどれに該当するかを判断しなければなりません。
コマンドがウェーブまたは変数名で始まる場合、Igor はそれを代入文とみなします。
コマンドが組み込みまたは外部コマンドの名前で始まる場合、そのコマンドはコマンドとして扱われます。
コマンドがユーザー定義のマクロ、ユーザー定義の関数、または外部関数の名前で始まる場合、そのコマンドは該当するタイプとして処理されます。

組み込み関数は、代入文の右側、または演算や関数のパラメーターとしてのみ使用できることに注意してください。
従って、次のコマンド：

```
sin(x)
```

は許可されておらず、Igor は「Expected wave name, variable name, or operation.」と表示します。
一方、以下のコマンドは許可されています：

```
Print sin(1.567)                    // sin は Print コマンドのパラメーター
wave1 = 5*sin(x)                    // sin は代入の右辺
```

スペルミスにより、Igor があなたの意図を判断できない場合、エラーダイアログが表示され、コマンドラインでエラーが強調表示されます。

代入文

代入文コマンドは、ウェーブまたは変数名で始まります。
このコマンドは、指定したオブジェクトのすべてまたは一部に値を代入します。
代入文は、 3 つの部分で構成されます。つまり、代入先、代入演算子、および式です。
例えば：

```
wave1       =           1+2*3^2
代入先      代入演算子  式
```

これは、wave1 のすべてのポイントに 19 を代入します。

上記の例でスペースは必須ではありません。次のように書くこともできます。

```
wave1=1+2*3^2
```

ウェーブ代入文の詳細については、ヘルプ Waveform Arithmetic and Assignment を参照してください。

以下の例では、str1 は String コマンドによって作成された文字列変数、var1 は Variable コマンドによって作成された数値変数、wave1 は Make コマンドによって作成されたウェーブです。

```
str1 = "Today is " + date()                 // 文字列の代入
str1 += ", and the time is " + time()       // 文字列の連結
var1 = strlen(str1)                         // 変数の代入
var1 = pnt2x(wave1,numpnts(wave1)/2)        // 変数の代入
wave1 = 1.2*exp(-0.2*(x-var1)^2)            // ウェーブの代入
wave1[3] = 5                                // ウェーブの代入
wave1[0,;3] = wave2[P/3] *exp(-0.2*x)       // ウェーブの代入
```

これらはすべて、カレントのデータフォルダー内のオブジェクトに対して動作します。
別のデータフォルダー内のオブジェクトに対して操作を行うには、データフォルダーのパスを使う必要があります。

```
root:'run 1':wave1[3] = 5                   // ウェーブの代入
```

（この構文は、コマンドラインおよびマクロにのみ適用されます。ウェーブを読み書きするには、ウェーブ参照を使わなければならないユーザー定義関数には適用されません。）

詳細は、ヘルプ Data Folders を参照してください。

リベラルなウェーブ名（オブジェクト名を参照）を使う場合は、引用符を使う必要があります：

```
'wave 1' = 'wave 2'                         // 正しい
wave 1 = wave 2                             // 誤り
```

（この構文は、コマンドラインおよびマクロにのみ適用されます。ウェーブを読み書きするには、ウェーブ参照を使わなければならないユーザー定義関数には適用されません。）

代入演算子

代入演算子は、式が代入先と結合される方法を決定します。
Igor は、次の代入演算子をサポートしています。

演算子	代入処理
=	代入先の内容は、式の評価結果に設定されます。
+=	式は代入先に加算されます。

-=	式は代入先から減算されます。
*=	代入先が式によって乗算されます。
/=	代入先が式によって除算されます。
:=	式の一部が変更されるたびに、代入先はその式の値に動的に更新されます。:= は、代入先が式に依存する「依存関係」を確立すると言われます。

例えば、

```
wave1 = 10
```

は wave1 の各 Y 値を 10 に設定します。
一方、

```
wave1 += 10
```

は wave1 の各 Y 値に 10 を加算します。
これは次と同じです。

```
wave1 = wave1 + 10
```

代入演算子 =、:=、および += は文字列の代入文で使用できますが、-=、*=、および /= は使用できません。
例えば、

```
String str1; str1 = "Today is "; str1 += date(); Print str1
```

「Today is Fri, Mar 31, 2000」のような文字列を出力します。

:= 演算子に関する詳細情報は、ヘルプ Dependencies を参照してください。

演算子

Igor が代入文の式部分でサポートする演算子の優先順位順の完全なリストは、次のとおりです。

演算子	**効果**
++, --	プリフィックスおよびポストフィックスのインクリメントおよびデクリメント。Igor Pro 7 以降が必要です。ユーザー定義関数内のローカル変数にのみ使用できます。
^, <<, >>	べき乗、ビット単位の左シフト、ビット単位の右シフト。シフトには Igor Pro 7 以降が必要です。
-, !, ~	否定、論理的補数、ビット単位の補数。
*, /	乗算、除算。
+, -	加算または文字列の連結、減算。
==, !=, >, <, >=, <=, &, |, %^	比較演算子、ビット単位の AND、ビット単位の OR、ビット単位の XOR。
&&, ||, ? :	論理積（AND）、論理和（OR）、条件演算子。
$	以下の文字列式を名前として置換。

比較演算子は、NaN パラメーターでは機能しません。
これは、定義上、NaN は他の値（別の NaN であっても）と比較すると false となるためです。
値が NaN であるかどうかをテストするには、numtype を使ってください。

比較演算子、ビット単位の AND、ビット単位の OR、ビット単位の XOR は、右から左に結合します。
従って、a==b>=c は (a==(b>=c)) を意味します。
例えば、2==1>= 0 は 1 ではなく 0 と評価されます。

その他のすべての二項演算子は左から右に結合します。

一般的な演算子の優先順位は誰もが直感的に理解していますが、混乱を避けるために括弧を付けることが役立ちます。
あなたは優先順位と結合性を理解しているかもしれませんが、あなたのコードを読む人はそうではないかもしれません。

一項否定は、そのオペランドの符号を変更します。
論理補数は、非ゼロのオペランドをゼロに、ゼロのオペランドを 1 に変更します。
ビット単位の補数は、オペランドを符号なし整数に切り捨ててから、その二進数補数に変更します。

べき乗は、左側のオペランドを右側のオペランドで指定された累乗にします。
つまり、32 は 3^2 と表記されます。
式 a^b で、結果が実数変数またはウェーブに代入される場合、b が整数でない限り、a は負の値であってはなりません。
結果が複素式で使われる場合、a が負、b が分数、または a と b が複素数のいずれかの組み合わせが許可されます。

指数が整数の場合、Igor は乗算のみを使って式を評価します。
効率的な評価を行うために a^2 を a*a と書く必要はありません。
Igor は自動的に同等の処理を行います。
一方、指数が整数でない場合、対数を用いて評価が行われるため、実数式における負の a に対する制限が生じます。

論理和（||）と論理積（&&）は、 2 つの式の真偽を決定します。
AND 演算子は、両方の式が真の場合にのみ真を返します。OR は、いずれかが真の場合に真を返します。
C 言語と同様に、true は 0 以外の任意の値であり、false は 0 です。
NaN に対しては、これらの演算は未定義です。
これらの演算子は、複素数式では使用できません。

論理演算子は左から右に評価され、必要のないオペランドは評価されません。
例えば、

```
if(MyFunc1() && MyFunc2())
```

MyFunc1() が false（0）を返す場合、MyFunc2() は評価されません。
なぜなら、全体の式が既に false であるためです。
これは、右側の式にウェーブの生成やグローバル値の設定などの副作用がある場合、予期しない結果をもたらす可能性があります。

ビット単位の AND（&）、OR（|）、および XOR（%^）は、オペランドを符号なし整数に切り捨てて変換し、その後、その二進数の AND、OR、または排他的 OR を返します。

ビット単位のシフト、<< および >> は、指定したビット数だけ左側のオペランドをシフトして結果を返します。
これらは Igor Pro 7 以降が必要です。
左側のオペランドは、シフトする前に整数に切り捨てられます。

条件演算子（? : ）は、if-else-endif 式を簡潔に表す形式です。
次の文では：

```
<expression> ? <TRUE> : <FALSE>
```

最初のオペランド < expression > はテスト条件です。
これが 0 以外の場合、Igor は <TRUE> オペランドを評価します。
それ以外の場合、<FALSE> が評価されます。

テスト条件に従って評価されるオペランドは 1 つだけです。
これは、次のように書いた場合と同じです：

```
if( <expression> )
	<TRUE>
else
	<FALSE>
endif
```

条件演算子内の「:」文字は、必ず両側のオペランドとスペースで区切る必要があります。
いずれかのスペースを省略すると、式はデータフォルダーのパスの一部として解釈されるため、エラー（「No such data folder」）が発生します。
安全のため、演算子記号とオペランドの間には必ずスペースを挿入してください。**<ここまで>**

オペランドは数値である必要があります。
文字列の場合は、SelectString 関数を使ってください。
条件演算子と複素式を組み合わせる場合、演算子が式を評価する際には実数部分のみが使われます。

条件演算子は混乱を招きやすいので、使用には注意が必要です。
例えば、Igor が何を返すかは、単純に確認しただけでは不明です。

```
1 ? 2 : 3 ? 4 : 5
```

この場合は 4 を返します。一方、

```
1 ? 2 : (3 ? 4 : 5)
```

は 2 を返します。
常に括弧を使って、曖昧さを排除してください。

比較演算子は、比較の結果が真の場合に 1 を返し、偽の場合に 0 を返します。
例えば、== 演算子は、そのオペランドが等しい場合は 1 を返し、等しくない場合は 0 を返します。
!= 演算子は逆の値を返します。
比較演算子は 1 または 0 の値を返すため、興味深い用途に利用できます。
代入文：

```
wave1 = sin(x)*(x<=50) + cos(x)*(x>50)
```

は、wave1 を、x=50 以下の部分では正弦波、x=50 以上の部分では余弦波になるように設定します。

二重の等号（==）は等しさを表すのに対し、単一の等号（=）は代入を表すことに注意してください。

丸め誤差のため、== 演算子を使って 2 つの数値の等しさを比較すると、誤った結果が生じる可能性があります。
数値が狭い範囲内に含まれるかどうかを確認するには、<= と >= を使う方が安全です。
例えば、ある変数が 3 分の 1 と同じかどうかを比較したいとします。
次の式：

```
(v1 == 1/3)
```

は、丸め誤差のため、失敗する可能性があります。
より安全な方法として、次のようなものを使うことをおすすめします：

```
((v1 > .33332) && (v1 < .33334))
```

数値が整数である場合は、2^{53}（およそ 10^{16}）未満の整数は倍精度浮動小数点数で正確に表現されるため、== を使っても問題はありません。

ここまでの演算子に関する議論では、数値のオペランドを前提としていました。
+ 演算子は、数値と文字列の両方のオペランドととして動作する唯一の演算子です。
例えば、str1 が文字列変数である場合、代入文：

```
str1 = "Today is " + "a nice day"
```

は、str1 に「Today is a nice day」という値を代入します。
もう 1 つの文字列演算子 $ については、「$ を使った文字列置換」のセクションで説明しています。

カッコで特に指定されていない限り、一項否定または補数演算が最初に実行され、次に指数演算、その後乗算または除算、さらに加算または減算、最後に比較演算子が実行されます。
ウェーブの代入：

```
wave1 = ((1 + 2) * 3) ^ 2
```

は、wave1 のすべてのポイントに値 81 を割り当てますが、

```
wave1 = 1 + 2 * 3 ^ 2
```

は、値 19 を代入します。

-a^b はこの規則の例外であり、-(a^b) と評価されます。

文字列の置換、部分文字列、およびウェーブインデックスの優先順位は、やや複雑です。
不明な場合は、括弧を使って、希望する優先順位を強制してください。

廃止された演算子

Igor Pro 4.0 以降、旧式のビット単位の補数 (%~)、ビット単位の AND (%&)、およびビット単位の OR (%|) 演算子は、演算子から % 文字が省略された新しいバージョンに置き換えられました。
これらの古いビット単位の演算子は、新しいバージョンと互換性がありますが、新しいコードでは使わないことが推奨されます。

オペランド

3.141 や 27 のような文字通りの数値の他に、演算子は変数や関数の値にも作用します。
代入文：

```
var1 = log(3.7) + var2
```

では、演算子 + は、log 関数が返す関数値と変数 var2 に作用します。
関数と関数値については後で説明します。

数値型

Igor では、各数値の宛先オブジェクト (ウェーブまたは変数) には、それぞれ固有の数値型があります。
数値型は、数値の精度（倍精度浮動小数点など）と数値のタイプ（実数または複素数）で構成されます。
ウェーブは単精度または倍精度の浮動小数点、あるいはさまざまなサイズの整数ですが、変数は常に倍精度の浮動小数点です。

代入先の数値の精度は計算に影響しません。
FFT など、その場で行われるいくつかの演算を除き、すべての計算は倍精度で行われます。

ウェーブは整数型を持つことができますが、ウェーブ式は常に倍精度浮動小数点型で評価されます。
浮動小数点値は、ウェーブに値を格納する前の最後のステップとして、丸めによって整数に変換されます。
格納する値が、指定された整数型が表現可能な値の範囲を超える場合、結果は未定義となります。

Igor Pro 7 以降、Igor は整数ローカル変数および 64 ビット整数ウェーブをサポートしています。
これらのいずれかが代入文の代入先である場合、Igor は整数演算を使って計算を実行します。
詳細については、ヘルプ Expression Evaluation を参照してください。

代入先のデータ型は、代入式の数値型（実数または複素数）を決定します。
これは、Igor は数値型の「予期しない」または「実行時の」変更を処理できないため、重要です。
例えば、負の数の平方根を求めると、その後のすべての演算は複素数を使って行う必要があります。

以下に例を挙げます：

```
Variable a, b, c, var1
Variable/C cvar1
Make wave1

var1= a*b
cvar1= c*cmplx(a+1,b-1)
wave1= var1 + real(cvar1)
```

最初の式は実数型で評価されます。
２番目の式には２つの型の混合が含まれています。
c と cmplx 関数の結果の乗算は複素数として評価されますが、cmplx 関数の引数は実数として評価されます。
３番目の例は、実数関数の引数が複素数として評価される点を除き、実数として評価されます。

定数型

Igor プロシージャファイルで、名前付きの数値定数および文字列定数を定義し、ユーザー定義関数の本体で使うことができます。

定数は、次の構文を使ってプロシージャファイルで定義されます。

```
Constant <name1> = <literal number> [, <name2> = <literal number>]

StrConstant <name1> = <literal string> [, <name2> = <literal string>]
```

静的プリフィックスを使うと、スコープを指定されたソースファイルに限定できます。
デバッグ時には、関数と同様に Override キーワードを使用できます。

これらの宣言は、以下の方法で使用できます：

```
Constant kFoo=1,kBar=2
StrConstant ksFoo="hello",ksBar="there"

static Constant kFoo=1,kBar=2
static StrConstant ksFoo="hello",ksBar="there"

Override Constant kFoo=1,kBar=2
Override StrConstant ksFoo="hello",ksBar="there"
```

プログラマーは、「k」と「ks」の接頭辞を使うことで、コードの読みやすさが向上するかもしれません。

数値定数と文字列定数の名前は、他のすべての名前と衝突する可能性があります。
指定された型の定数は、異なるファイル内の静的定数および Override を使う場合を除き、重複して定義することはできません。
唯一の真の競合は、変数名と、pi などのパラメーターを受け付けない特定の組み込み関数との競合です。
変数名は定数を上書きしますが、定数は pi のような関数を上書きします。

依存代入文

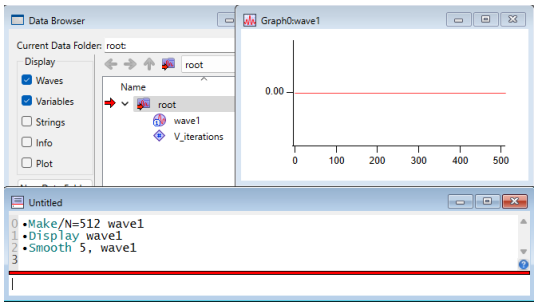
グローバル変数やウェーブを設定して、他のグローバルオブジェクトが変更されたときにその内容を自動的に再計算するようにすることができます。
詳細は、ヘルプ Dependencies を参照してください。

操作コマンド

コマンドとは、アクションを実行する組み込みまたは外部のルーチンですが、関数とは異なり、直接、値を返しません。

以下に例を示します。

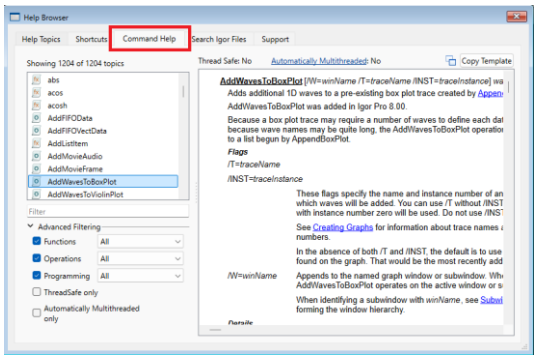
```
Make/N=512 wave1
Display wave1
Smooth 5, wave1
```



コマンドは、Igor の作業の大部分を実行し、ダイアログを使って Igor で作業すると、自動的に生成され実行されます。

これらのダイアログを使って、目的の操作を試すことができます。

ダイアログ内をクリックすると、Igor がコマンドを作成します。これにより、コマンドの構文を確認したり、ユーザー定義プロシージャで使うコマンドを生成したりするための便利な方法が提供されます。
組み込みコマンドの完全なリストについては、Igor Pro リファレンスを参照してください。
構文を学習するもう 1 つの方法は、Igor ヘルプブラウザの Command Help タブを使うことです。



操作コマンドの構文はさまざまですが、一般的には、コマンド名、その後にフラグのリスト（例：/N=512）、そしてパラメーターリストの順で構成されます。
コマンドの名前は、コマンドの主な動作を指定し、コマンドの残りの部分の構文を決定します。
フラグのリストは、コマンドのデフォルト動作のバリエーションを指定します。
コマンドのデフォルトの動作が適切であれば、フラグは不要です。
パラメーターリストは、コマンドの対象となるオブジェクトを指定します。
一部のコマンドはパラメーターを必要としません。
例えば、次のコマンドでは：

```
Make/D/N=512 wave1, wave2, wave3
```

コマンドの名前は「Make」です。
フラグのリストは「/D/N=512」です。
パラメーターのリストは「wave1、wave2、wave3」です。

Igor が数値パラメーターを期待するコマンドのパラメーターリストでは、数値式を使用できますが、コマンドのフラグでは、式を括弧で囲む必要があります。
例えば、

```
Variable val=1.0
Make/N=(val) wave0, wave1
Make/N=(numpnts(wave0)) wave2
```

最も一般的なパラメーターのタイプは、リテラル数値または数値式、リテラル文字列または文字列式、名前、およびウェーブです。
上記の例では、wave1 は Make コマンドに渡される名前パラメーターです。
Display および Smooth コマンドに渡されるときは、ウェーブパラメーターです。
名前パラメーターは、すでに存在する、あるいは存在しないウェーブを参照することができますが、ウェーブパラメーターは、既存のウェーブを参照しなければなりません。

すべてのコマンドに共通する一般的な情報については、「パラメーターのリスト」のセクションを参照してください。

ユーザー定義のプロシージャコマンド

ユーザー定義のプロシージャコマンドは、プロシージャ名で始まり、括弧で囲まれたパラメーターのリストが続きます。
例えば、

```
MyFunction1(5.6, wave0, "igneous")
```

マクロと関数パラメーター

１つ以上の入力パラメーターが欠落しているマクロは呼び出すことができますが、関数は呼び出すことはできません。
この場合、Igor は、欠落しているパラメーターを入力するためのダイアログを表示します。

ユーザー定義関数に同様の機能を追加するには、Prompt と DoPrompt キーワードを使用できます。

マクロはプログラミング機能に非常に制限があったため、まれな例外を除き、関数を使ってプログラミングを行うべきです。

関数コマンド

関数は、数値または文字列の値を直接返すルーチンです。
Igor では、３種類の関数を使用できます。

- ビルトイン
- 外部（XFUNC）
- ユーザー定義

組み込みの数値関数は、外部関数やユーザー定義関数に比べて１つの利点があります。
いくつかは実数型および複素数型で提供されており、Igor は式内の現在の数値型に応じて適切なバージョンを自動的に選択します。
外部関数とユーザー定義関数は、異なる型を必要とする場合、異なる名前でなければなりません。
一般に、実際のユーザー関数と外部関数を提供する必要があります。

例えば、次のウェーブの代入において、

```
wave1 = enoise(1)
```

wave1 が実数である場合、関数 enoise は実数の値を返します。
wave1 が複素数である場合、enoise は複素数の値を返します。

関数は、その関数が返すデータ型が、その関数を使うコンテキストで意味を成す限り、別の関数、演算、マクロ、または算術式や文字列式でパラメーターとして使うことができます。

ユーザー定義関数と外部関数は、単独でコマンドとして使用することもできます。
これは、数値の計算以外の目的（グラフの表示や新しいウェーブの作成など）を持つユーザー関数を作成する場合に使います。
組み込み関数はこのようには使用できません。
例えば：

```
MyDisplayFunction(wave0)
```

外部関数およびユーザー定義関数は、組み込み関数と同じように使用できます。
さらに、数値関数はカーブフィッティングでも使用できます。
ヘルプ User-Defined Functions および Fitting to a User-Defined Function を参照してください。

ほとんどの関数は、関数名、左括弧、パラメーターリスト、右括弧で構成されています。
このセクションの冒頭で示したウェーブの代入では、関数名は enoise です。
パラメーターは 1 です。
パラメーターは括弧で囲まれます。
この例では、関数の結果がウェーブに割り当てられています。
変数に代入したり、出力したりすることもできます。

```
K0 = enoise(1)
Print enoise(1)
```

ユーザー定義関数と外部関数は、組み込み関数を除き、コマンドラインや他の関数、マクロ内で実行する時、結果を代入したり出力したりする必要なく実行できます。
この機能は、関数の目的が明示的な結果ではなく、その副作用にある場合に役立ちます。

パラメーターリストが空の場合でも、ほぼすべての関数には括弧が必要です。
例えば、関数 date() にはパラメーターはありませんが、とにかく括弧が必要です。
いくつかの例外があります。
例えば、関数 Pi は π を返し、括弧やパラメーターは使いません。

Igor の組み込み関数は、ヘルプ Igor Pro Reference で詳しく説明しています。

パラメーターリスト

パラメーターリストは、コマンド、関数、およびマクロに使われ、 1 つ以上の数値、文字列、キーワード、または Igor オブジェクトの名前で構成されます。
パラメーターリスト内のパラメーターは、コンマで区切らなければなりません。

パラメーターとしての式

数値パラメーターを含むコマンド、関数、またはマクロのパラメーターリストでは、リテラル数値の代わりに数値式を常に使うことができます。

数値式は、リテラル数値、数値変数、数値関数、および数値演算子の規則に則った組み合わせです。
例えば、次のコマンドを考えてみます：

```
SetScale x, 0, 6.283185, "v", wave1
```

これは、wave1 の X スケーリングを設定します。
次のように記述することもできます：

```
SetScale x, 0, 2*PI, "v", wave1
```

リテラル数値には、「0x」で始まる 16 進数リテラルが含まれます。
例えば：

```
Printf "The largest unsigned 16-bit number is %d\r", 0xFFFF
```

/N=(<式>) には括弧が必要

多くのコマンドでは、形式「/A=n」のフラグを受け付けます。
ここで、A は任意の文字、n は任意の数字です。
n に数値式を使うこともできますが、その場合は式を括弧で囲む必要があります。

例えば、次の 2 つ：

```
Make/N=512 wave1
Make/N=(2^9) wave1
```

は正しいですが、

```
Make/N=2^9 wave1
```

は正しくありません。

変数名は数値式の 1 つの形式です。
従って、v1 が変数の名前であると仮定すると：

```
Make/N=(v1)
```

は正しいですが、

```
Make/N=v1
```

は正しくありません。
この括弧付けは、数値式をコマンドのフラグで使う場合にのみ必要です。

文字列式

Igor が文字列パラメーターを期待する場所では、文字列式を使うことができます。
文字列式は、リテラル文字列、文字列変数、文字列関数、UTF-16 リテラル、および文字列を連結する文字列演算子「+」の規則に則った組み合わせです。

UTF-16 リテラルは Unicode コード値を表し、「U+」 に続いて 4 つの 16 進数で構成されます。
例えば：

```
Print "This is a bullet character: " + U+2022
```

ビットパラメーターの設定

多くのコマンドでは、特定のパラメーターを設定するためにビット値を指定する必要があります。
このような場合、コマンド内の特定のビット値を使って、特定のビット番号を設定します。
ビット値は 2^n で、n はビット番号です。
従って、ビット 0 を設定するにはビット値 1 を使い、ビット 1 を設定するにはビット値 2 を使い、以下同様です。

TraceNameList 関数の例では、最後のパラメーターはビット設定です。
通常のトレースを選択するには、ビット 0 を設定する必要があります：

```
TraceNameList("",";",1)
```

コンタートレースを選択するには、ビット 1 を設定します：

```
TraceNameList("",";",2)
```

最も重要な点は、ビットの値を合計することで複数のビットを同時に設定できることです。
従って、TraceNameList では、次のようにして通常のトレース（ビット 0）とコンタートレース（ビット 1）の両方を同時に選択できます：

```
TraceNameList("",";",3)
```

ヘルプ Using Bitwise Operators を参照してください。

RGBA 値

このセクションでは、Gizmo コマンド以外のコマンドで色を指定するために使う RGBA 値について説明します。
Gizmo コマンドについては、ヘルプ Gizmo Color Specification を参照してください。

コマンドでは、色は RGBA 値の形式（ *r,g,b*[,*a*] ）で指定します。

r、*g*、および *b* は、色における赤、緑、青の量を 0 から 65535 までの整数で指定します。

オプションのパラメーター *a* は、色の不透明度を表す「アルファ」を 0 (完全に透明) から 65535 (完全に不透明) までの整数で指定します。
a のデフォルトは 65535 (完全に不透明) です。

(0,0,0) は不透明な黒を表し、(65535,65535,65535) は不透明な白を表します。

例えば：

```
ModifyGraph rgb(wave0)=(0,0,0)                  // 不透明な黒
ModifyGraph rgb(wave0)=(65535,65535,65535)      // 不透明な白
ModifyGraph rgb(wave0)=(65535,0,0,30000)        // 半透明の赤
```

文字列の処理

Igor は、文字列を扱う豊富な機能を備えています。
また、独自の文字列関数を定義することもできます。
以下で説明するテクニックの多くは、プログラマーの方のみに関心のあるものだと思います。

多くのコマンドでは、文字列パラメーターが必要です。
例えば、グラフの軸にラベルを付けるには、Label コマンドを使います。

```
Label left, "Volts"
```

その他のコマンドでは、パラメーターとして名前が必要です。
例えば、ウェーブを作成するには、Make コマンドを使います。

```
Make wave1
```

文字列置換テクニック（「$ を使った文字列の置換」のセクションを参照）を使うと、名前を含む文字列を作成し、$ 演算子を使って名前パラメーターを生成することができます。

```
String stringContainingName = "wave1"
Make $stringContainingName
```

文字列式

文字列パラメーターを必要とする場所では、文字列式を使用できます。
文字列式は、次のとおりです：

- リテラルな文字列（`"Today is"`）
- 文字列関数の出力（`date()`）
- テキストウェーブの要素（`textWave0[3]`）
- UTF-16 リテラル（`U+2022`）
- 文字列式のいくつかの組み合わせ（`"Today is" + date()`）

さらに、別の文字列式からインデックスを指定して文字列式を取得することができます。
例えば：

```
Print ("Today is" + date())[0,4]
```

は、”Today” を出力します。

文字列変数は、文字列式の結果を格納できます。
例えば：

```
String str1 = "Today is" + date()
```

文字列変数は、文字列式の一部としても使用できます。
例えば、

```
Print "Hello. " + str1
```

テキストウェーブ内の文字列

テキストウェーブには、テキスト文字列の配列が含まれます。
ウェーブの各要素は、利用可能なすべての文字列操作の手法を使って処理できます。
さらに、テキストウェーブは、棒グラフのカテゴリ軸の作成によく使われます。
詳細については、ヘルプ Text Waves を参照してください。

文字列のプロパティ

Igor の文字列は、最大で約 20 億バイトまで格納できます。

Igor 文字列は通常、テキストデータの保存に使われますが、バイナリデータの保存にも使用できます。

文字列をテキストデータとして扱う場合、例えば、コマンドウィンドウの履歴エリアに出力したり、注釈やコントロールにその内容を表示したりすると、Igor の内部ルーチンは、Null バイトを「文字列の終了」と解釈します。
従って、バイナリ文字列をテキストとして扱おうとすると、予期しない結果が生じる可能性があります。

文字列をテキストとして扱う場合、Igor はそのテキストが UTF-8 テキストエンコーディングでエンコードされていると想定します。
詳細については、ヘルプ String Variable Text Encodings を参照してください。

Unicode リテラル文字

Unicode リテラルは、UTF-16 コード値を使って文字を表します。
これは「U+」に続いて 4 桁の 16 進数で構成されます。
例えば、

```
Print "This is a bullet character: " + U+2022
```

Unicode 文字コードの一覧は、<http://www.unicode.org/charts> で管理されています。

文字列内のエスケープシーケンス

Igor は、コマンドラインでリテラル (引用符で囲まれた) 文字列を読み込む際に、バックスラッシュ文字を特別な方法で処理します。
バックスラッシュは「エスケープシーケンス」を導入するために使われます。
これは、バックスラッシュと次の文字または次の数個の文字が、引用符で囲まれた文字列に通常は含めることのできない別の文字として扱われることを意味します。
エスケープシーケンスは次のとおりです：

\t	タブ文字を示す
\r	リターン文字（CR）を示す
\n	改行文字（LF）を示す
\’	シングルクオートを示す
\”	ダブルクオートを示す
\\	バックスラッシュを示す
\ddd	任意のバイトコードを表す ddd は３桁の１０進数
\xdd	任意のバイトコードを表す dd は２桁の１６進数 Igor Pro 7.0 以降が必要
\udddd	UTF-16 コードポイントを表す dddd は４桁の１６進数 Igor Pro 7.0 以降が必要
\Udddddddd	UTF-32 コードポイントを表す dddddddd は８桁の１６進数 Igor Pro 7.0 以降が必要

例えば、「fileName」という文字列変数がある場合、次のようにして履歴エリアに出力することができます。

```
fileName = "Test"
Printf "The file name is \"%s\"\r", fileName
```

これは、次を出力します。

```
The file name is "Test"
```

Printf コマンドラインでは、\" は、フォーマット文字列に二重引用符を埋め込むことを Igor に指示します。
バックスラッシュを省略した場合、Igor は " をフォーマット文字列の終わりと認識します。
\r は、フォーマット文字列にキャリッジリターン（CR）を挿入することを Igor に指示します。

文字列内の Unicode エスケープシーケンス

Igor は、テキストを内部で UTF-8 形式で表現します。
UTF-8 は、Unicode 用のバイト指向のエンコード形式です。
このセクションでは、リテラル文字列内で Unicode 文字を表すために \u、\U、および \x のエスケープシーケンスを使う方法について説明します。

UTF-8 およびこれらのエスケープシーケンスを使うには、Igor Pro 7.0 以降が必要です。
Igor Pro 6.x 以前でこれらを使うと、データが正しく表示されない、またはエラーが発生します。

次の最初の例では、16 進コード 0041 を使っています。
これは、大文字の A を表す Unicode コードポイントの 16 進表記です。

```
String str

str = "\x41"                    // UTF-8 エスケープシーケンスでコードポイントを指定
Print str

str = "\u0041"                  // UTF-16 エスケープシーケンスでコードポイントを指定
Print str

str = "\U00000041"              // UTF-32 エスケープシーケンスでコードポイントを指定
Print str
```

\xdd を使うと、Igor はエスケープシーケンスを dd で指定されたバイトに置き換えます。
従って、有効な UTF-8 文字を表すバイト値を指定する必要があります。
41 は、UTF-8 エンコーディングにおける大文字の A の 16 進数です。

\udddd または \Udddddddd を使うと、Igor はエスケープシーケンスを、UTF-16 または UTF-32 形式で表される対応する Unicode コードポイントの UTF-8 バイトに置き換えます。
指定したコードポイントが有効な UTF-16 または UTF-32 コードポイントではない場合、Igor はエスケープシーケンスを Unicode 置換文字に置き換えます。

次は、日本語の漢字で「豊田」の文字を指定する例です：

```
String str

str = "\xE8\xB1\x8A\xE7\x94\xB0"          // UTF-8
Print str

str = "\u8C4A\u7530"                      // UTF-16
Print str

str = "\U00008C4A\U00007530"              // UTF-32
Print str
```

正しい文字を表示するには、日本語文字を含むフォントを使う必要があります。

\U は、主に、Basic Multilingual Plane に含まれていない、珍しい Unicode 文字を入力する場合に使います。
このような文字は、16 ビットの UTF-16 では指定できません。
以下に例を示します。

```
String str

str = "\U00010300"                    // OLD ITALIC LETTER A に対する UTF-32
Print str

str = "\xF0\x90\x8C\x80"              // OLD ITALIC LETTER A に対する UTF-8
Print str
```

OLD ITALIC LETTER A は、Unicode Supplementary Multilingual Plane（Plane 1）に位置しています。

参照： http://en.wikipedia.org/wiki/Unicode
http://en.wikipedia.org/wiki/UTF-16
http://en.wikipedia.org/wiki/UTF-8

リテラル文字列に埋め込まれた Null

バイト指向の文字列における Null は、値が 0 のバイトです。

文字列に Null バイトを埋め込むことが可能です:

```
String test = "A\x00B"                // Igor Pro 7 以降では OK
Print strlen(text)                    // 3 を出力
Print char2num(test[0]), char2num(test[1]), char2num(test[2])   // 65 0 66 を出力
```

ここでは、Igor はエスケープシーケンス \x00 を Null バイトに変換しています。

通常、文字列には Null を埋め込む必要はありません。
これは、文字列は通常、読み取り可能なテキストを格納するために使われ、Null は読み取り可能な文字を表さないためです。
ただし、文字列にテキストデータではなくバイナリデータを格納している場合、その必要が生じる可能性があります。
例えば、少量のバイナリデータを機器に送信する必要がある場合、\x エスケープシーケンスを使って、データをリテラル文字列で表現することができます。

リテラル文字列に Null を埋め込むことはできますが、Igor の他の部分では Null を処理できません。
例えば、

```
String test = "A\x00B"
Print test                        // "A<null>B" ではなく "A" を出力
```

C 言語では、Null は「文字列の終わり」を意味します。
Igor では C 言語文字列および C 言語ライブラリルーチンが使っているため、Igor の多くの部分では、埋め込まれた Null は文字列の終わりとして扱われます。
そのため、上記の Print ステートメントでは「A」だけが印刷されます。

結論としては、Igor 文字列には Null を含むバイナリデータを格納できますが、Igor の多くの部分は、読み取り可能なテキストを期待するため、Null を文字列の終了として扱います。
詳細については、ヘルプ Working With Binary String Data を参照してください。

文字列のインデックス付け

インデックス付けは、文字列の一部を抽出します。
これは、文字列式に続いて括弧内に 1 つまたは 2 つの数字を指定することで行われます。
数値はバイト位置を表します。
0 は最初のバイトのバイト位置です。
n-1 は n バイトの文字列値の最後のバイトのバイト位置です。
例えば、s1 という文字列変数を作成し、それに次のように値を代入するとします。

```
String s1="hello there"
```

h	e	l	l	o		t	h	e	r	e
0	1	2	3	4	5	6	7	8	9	10

この場合。

```
Print s1[0,4]                       // hello       を出力
Print s1[0,0]                       // h           を出力
Print s1[0]                         // h           を出力
Print s1[1]+s1[2]+s1[3]             // ell         を出力
Print (s1+" jack")[6,15]            // there jack  を出力
```

s1[p] のような 1 つのインデックスでインデックス付けされた文字列は、p が範囲内（つまり、0 ≤ p ≤ n-1）の場合、1 バイトの文字列です。
p が範囲外の場合、s1[p] はバイトを含まない文字列です。
例えば：

```
Print s1[0]                         // h           を出力
Print s1[-1]                        // 何も出力しない
Print s1[10]                        // e           を出力
Print s1[11]                        // 何も出力しない
```

s1[p1,p2] のような 2 つのインデックスでインデックス付けされた文字列には、s1[p1] から s1[p2] までのすべてのバイトが含まれます。
例えば、

```
Print s1[0,10]                      // hello there       を出力
Print s1[-1,11]                     // hello there       を出力
Print s1[-2,-1]                     // 何も出力しない
Print s1[11,12]                     // 何も出力しない
Print s1[10,0]                      // 何も出力しない
```

INF は「文字列の終了」を意味するために使用できます：

```
Print s1[0,INF]                     // hello there       を出力
```

これらの例のインデックスは、文字の位置ではなく、バイトの位置です。
この違いについては、ヘルプ Characters Versus Bytes を参照してください。
ユーザー定義関数を使って文字を 1 文字ずつ処理する方法の例については、ヘルプ Character-by-Character Operations を参照してください。

文字列のインデックス指定の構文はウェーブのインデックス指定の構文と同じであるため、テキストウェーブを使う場合は注意が必要です。
例えば：

```
Make/T textWave0 = {"Red", "Green", "Blue"}
```

```
Print textWave0[1]                  // Green       を出力
Print textWave0[1][1]               // Green       を出力
Print textWave0[1][1][1]            // Green       を出力
Print textWave0[1][1][1][1]         // Green       を出力
Print textWave0[1][1][1][1][1]      // r     を出力
```

最初の４つの例は、列 0 の行 1 を出力します。
ウェーブは最大４次元まで存在するため、最初の４つの [1] は次元インデックスとして機能します。
列、レイヤー、およびチャンクのインデックスが範囲外のため、0 にクリップされます。
最後の例では、次元が足りなくなり、文字列のインデックス付けになります。

ウェーブと文字列のインデックスの曖昧さを回避するには、次のように括弧を使います：

```
Print (textWave0[1])[1]             // r     を出力
```

プロシージャでは、可読性とデバッグの容易さのために、次のようにローカル変数を使うのが最適です。

```
String tmp = textWave0[1]
Print tmp[1]                        // r     を出力
```

文字列の代入

文字列インデックスを使って、文字列変数のサブレンジ（一部）に値を割り当てることもできます。
s1 という文字列変数を作成し、次のように値を割り当てるとします。

```
String s1="hello there"
```

この場合、

```
s1[0,4]="hi"; Print s1              // hi there          を出力
s1[0,4]="greetings"; Print s1       // greetings there   を出力
s1[0,0]="j"; Print s1               // jello there       を出力
s1[0]="well "; Print s1             // well hello there  を出力
s1[100000]=" jack"; Print s1        // hello there jack  を出力
s1[-100]="well "; Print s1          // well hello there  を出力
```

s1[p1,p2]=<文字列式> という構文を使う場合、文字列の代入の右側は、p1 および p2 が 0 から n まで切り捨てられた後、左側で指定される文字列変数のサブレンジ（一部）に置き換えられます。
ここで、n は宛先のバイト数です。

s1[p]=<文字列式> 構文を使う場合、文字列代入の右辺は、p が 0 から n に切り詰められた後に、p で指定されるバイトの前に挿入されます。

文字列変数に対して前述のサブレンジの割り当ては、テキストウェーブが宛先の場合にはサポートされていません。
テキストウェーブ要素の範囲に値を割り当てるには、一時的な文字列変数を作成する必要があります。
例えば、

```
Make/O/T tw = {"Red", "Green", "Blue"}
String stmp= tw[1]
stmp[1,2]="XX"
tw[1]= stmp;

Print tw[0],tw[1],tw[2]             // Red GXXen Blue    を出力
```

これらの例のインデックスは、文字の位置ではなく、バイトの位置です。
この区別については、ヘルプ Characters Versus Bytes を参照してください。

$ を使った文字列の置き換え

Igor が、ウェーブ名などのオペランドのリテラル名を期待する場所では、その代わりに、$ 文字を先頭にした文字列式を指定することができます。
$ 演算子は文字列式を評価し、その結果を名前として返します。

Make コマンドでは、作成するウェーブの名前が指定される必要があります。
例えば、wave0 という名前のウェーブを作成するとします。

```
Make wave0                          // OK: wave0 はリテラル名

Make $"wave0"                       // OK: $"wave0" は wave0 と評価される

String str = "wave0"
Make str                            // 誤り: これは str という名前のウェーブを作る
Make $str                           // OK: $str は wave0 と評価される
```

$ は、作成するウェーブの名前をパラメーターとして受け取る関数を書く場合によく使われます。
簡単な例を以下に示します。

```
Function MakeWave(wName)
        String wName                // ウェーブの名前

        Make $wName
End
```

この関数は次のように呼び出します：

```
MakeWave("wave0")
```

ウェーブ名が必要なので $ を使いますが、ウェーブ名を含む文字列があります。
$ を省略して次のように記述した場合、

```
Make wName
```

wave0 という名前のウェーブではなく、wName という名前のウェーブを作成します。

文字列置換は、文字列式を 1 つの名前に変換することができます。
複数の名前は処理できません。
例えば、以下は機能しません。

```
String list = "wave0;wave1;wave2"
Display $list
```

その方法については、ヘルプ Processing Lists of Waves を参照してください。

$ を使ってユーザー定義関数内で $ を使う方法の詳細については、ヘルプ Converting a String into a Reference Using $ を参照してください。

コマンド内の $ の優先順位の問題

文字列の置換が予想どおりに機能しないケースが 1 つあります。
次の例を考えてみます。

```
String str1 = "wave1"
wave2 = $str1 + 3
```

（このセクションでの $ の使用は、コマンドラインおよびマクロにのみ適用されます。ウェーブを読み書きするには、ウェーブレファレンスを使わなければならないユーザー定義関数には適用されません。）

これにより、wave2 を wave1 と 3 の和に設定すると予想されるかもしれません。
しかし、実際には「expected string expression（文字列式を期待）」エラーが発生します。
その理由は、$ によって示される置換を行う前に、str1 と 3 を連結しようとしているからです。
+ 演算子は２つの文字列式を連結するためにも使われ、$ 演算子よりも優先順位が高くなります。
str1 は文字列ですが、3 は文字列ではないため、連結を行うことができません。

このウェーブの代入を次のいずれかに変更することで、この問題を解決できます。

```
wave2 = 3 + $str1
wave2 = ($str1) + 3
```

どちらも、wave2 を wave1 と 3 の和に等しく設定するという、望ましい効果を実現しています。
同様に、

```
wave2 = $str1 + $str2          // Igor は "$(str1 + $str2)" とみなす
```

は、同じ「expected string expression」エラーが発生します。
理由は、str1 と $str2 を連結しようとしているからです。
$str2 は名前であり、文字列ではありません。解決策は次のとおりです。

```
wave2 = ($str1) + ($str2)     // これは wave2 に２つの名前のウェーブの合計を設定
```

$ 演算子と [を使う場合、別の状況が発生します。
[記号は、ウェーブ内のポイントのインデックス指定、または文字列内のバイトのインデックス指定のいずれにも使用できます。
コマンド：

```
String wvName = "wave0"
$wvName[1,2] = wave1[p]       // “wave0” という名前のウェーブに２つの値を設定
```

は、Igor によって、wave0 の１番目と２番目のポイントが wave1 の設定値であると解釈されます。

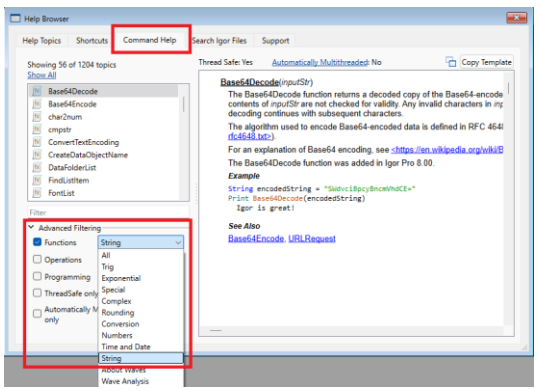
「$wvName[1,2] = wave1」を、wvName 文字列の１バイト目と２バイト目（「av」）の名前を持つウェーブのすべての値を wave1 から設定することを意味する場合、括弧を使う必要があります。

```
$(wvName[1,2]) = wave1        // "av" という名前のウェーブにすべての値を設定
```

文字列ユーティリティ関数

文字列の処理に役立つ複数のユーティリティ関数を提供しています。
組み込みの文字列関数のリストを確認するには：

1. Help Browser の Command Help タブを開く
2. Advanced Filtering コントロールを開く
3. Functions 以外のすべてのチェックボックスを外す
4. Functions ポップアップメニューから String を選択する



ユーザー定義関数を使って文字を１文字ずつ処理する方法の例については、ヘルプ Character-by-Character Operations を参照してください。

特別なケース

このセクションでは、Igor のコマンド言語に関して発生する特定の特殊な状況に対処するために考えられたいくつかのテクニックについて説明します。

インスタンス表記

グラフに同じウェーブが複数存在する場合、またはレイアウトに同じオブジェクトが複数存在する場合、問題が発生します。
例えば、yWave を xWave0、xWave1、および xWave2 に対してグラフ化したいとします。
これを行うには、以下を実行する必要があります。

```
Display yWave vs xWave0
AppendToGraph yWave vs xWave1
AppendToGraph yWave vs xWave2
```

結果は、yWave が 3 回出現するグラフになります。
次に、yWave を

```
RemoveFromGraph yWave
```

または

```
ModifyGraph lsize(yWave)=2
```

で削除または変更しようとすると、Igor は yWave の最初のインスタンスを常に削除または変更します。

インスタンス表記は、特定のウェーブの特定のインスタンスを指定する方法です。
この例では、コマンド：

```
RemoveFromGraph yWave#2
```

は、yWave のインスタンス番号 2 を削除します。
そして、

```
ModifyGraph lsize(yWave#2)=2
```

は、yWave のインスタンス番号 2 を修正します。
インスタンス番号は 0 から始まるため、「yWave」は「yWave#0」と等価です。
インスタンス番号 2 は、この例で xWave2 に対してプロットされた yWave のインスタンスです。

曖昧さを避ける必要がある場合、Igor の操作ダイアログ（Modify Trace Appearance など）は自動的にインスタンス表記を使います。
トレース名（ModifyGraph など）またはレイアウトオブジェクト名（ModifyObject など）を受け入れる操作は、インスタンス表記を受け入れます。

グラフには、異なるデータフォルダーにある同じ名前のウェーブも複数表示することができます。
この場合も、インスタンス表記が適用されます。

インスタンス表記と $

$ 演算子はインスタンス表記と組み合わせて使用できます。
# 記号は文字列オペランドの内側にある場合も、外側にある場合もあります。

例えば、$”wave0#1” または $”wave0”#1 などです。
ただし、# 記号は文字列内に含まれる場合があるため、Igor によって文字列が解析される必要があります。

従って、他の $ の使用法とは異なり、リベラル名を使う場合は、ウェーブ名部分をシングルクオートで囲む必要があります。
例えば、リベラル名が ‘ww#1’ のウェーブを２回プロットするとします。
最初のインスタンスは $”’ww#1’”、２番目のインスタンスは $”’ww#1'#1” となり、$”ww#1” はウェーブ ww の２番目のインスタンスを参照します。

オブジェクトのインデクシング

グラフ、テーブル、およびページレイアウトを変更するために使われる ModifyGraph、ModifyTable、および ModifyLayout コマンドは、それぞれ、変更するオブジェクトを識別する別の方法をサポートしています。

この方法、オブジェクトのインデクシングは、Igor でスタイルマクロを生成するために使われます (ヘルプ Graph Style Macros を参照)。
他の状況でも役立つ場合があります。

通常、変更したいオブジェクトの名前を知っている必要があります。
例えば、３つのトレースを含むグラフがあり、プロシージャからトレースのマーカーを設定したい場合を考えます。
次のように記述します。

```
ModifyGraph marker(wave0)=1, marker(wave1)=2, marker(wave2)=3
```

このコマンドは指定されたトレースの名前を使うため、特定のグラフに固有のコマンドです。
任意のグラフ内の３つのトレースのマーカーを設定するコマンドを書きたい場合はどうすればよいでしょうか？
ここでオブジェクトインデックスが役立ちます。

オブジェクトインデックスを使うと、次のように記述できます：

```
ModifyGraph marker[0]=1, marker[1]=2, marker[2]=3
```

このコマンドは、グラフ内の最初の３つのトレースのマーカーを設定します。トレースの名前は関係ありません。

インデックスは 0 から始まります。
グラフの場合、オブジェクトインデックスはグラフに最初に配置されたトレースから始まるトレースを指します。
テーブルの場合、インデックスは左から右の列を指します。
ページレイアウトの場合、インデックスはレイアウトに最初に配置されたオブジェクトから始まるオブジェクトを指します。

/Z フラグ

グラフに３つのウェーブがあることがわかっている場合は、上記の ModifyGraph マーカーコマンドは問題なく機能します。
ただし、ウェーブの数が 3 未満のグラフで使うと、エラーが発生します。
ModifyGraph コマンドは、これを処理するために使用できるフラグをサポートしています。

```
ModifyGraph/Z marker[0]=1, marker[1]=2, marker[2]=3
```

/Z フラグは、コマンドが存在しないオブジェクトを変更しようとした場合でも、Igor に無視するよう指示します。
/Z フラグは、SetAxis および Label コマンド、ならびに ModifyGraph、ModifyTable、および ModifyLayout コマンドで機能します。
オブジェクトのインデクシングと同様に、/Z フラグは主にスタイルマクロの作成に使われますが、Igor はこれを自

動的に行います。
ただし、他の用途にも役立つ場合があります。