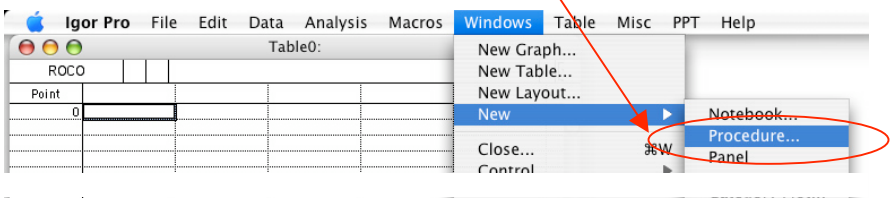
Igor Programming Tutorial

Last Modified: 08/17/2005

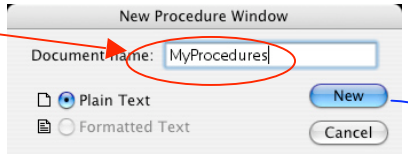
1. Getting Started

This tutorial will provide you with key concepts and tips in programming. It will allow you to create your own macros, which can be added to the DefaultMacro to share with others. Start by launching Igor Pro. You will need to create your own macro file. If you write in the DefaultMacro and an update is later provided, you will loose your macros. So keep them separate. If you find that your macro will be beneficial for others to use, send them for review and they can be added to the DefaultMacro.

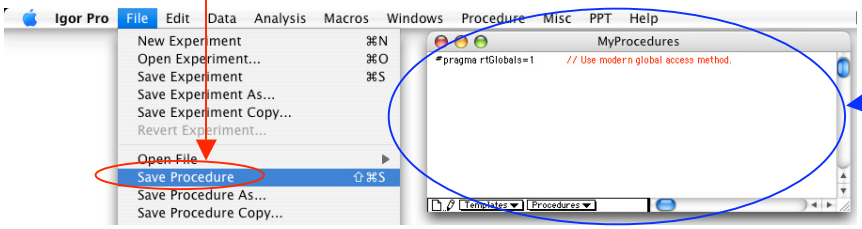
In the Igor menu, click on "Window", then "New", then "Procedure" as follow:



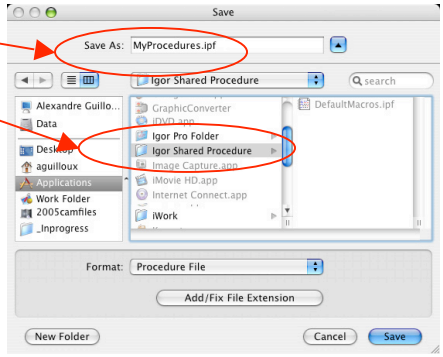
A new window appears, type "MyProcedures" as the document name and click on "New"



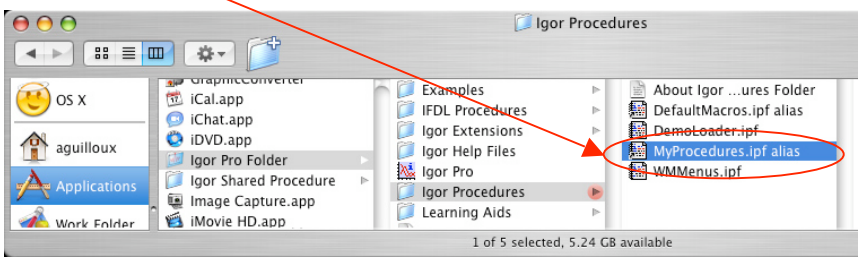
Save it right away, click on "File", "Save Procedure"



Type MyProcedures.ipf in the "Save As".
Select "Igor Shared Procedure" as the folder (you can find it in the "Applications" folder).



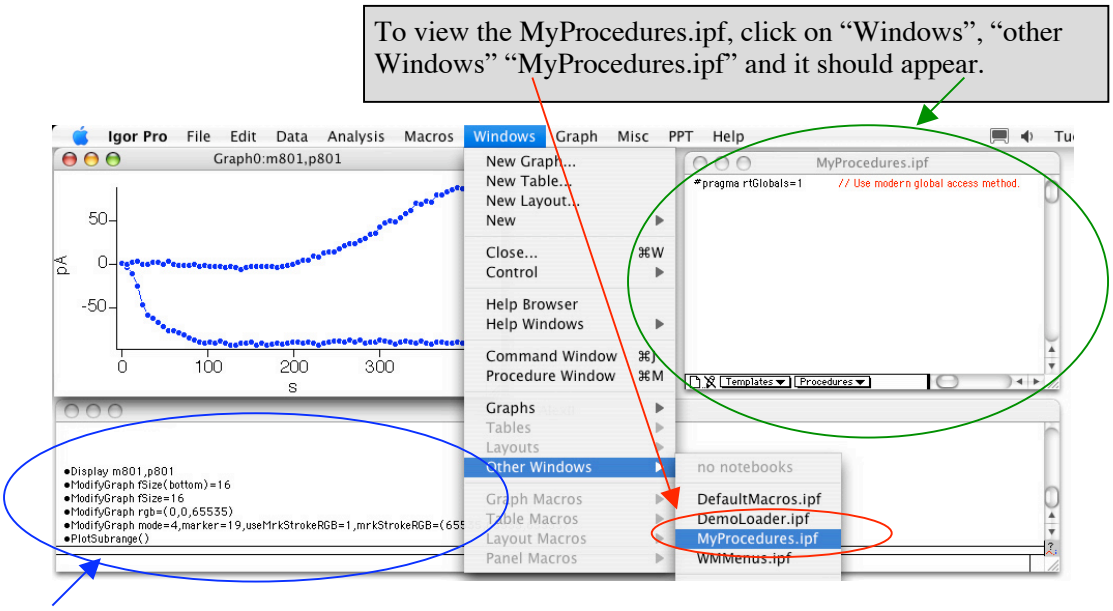
Make sure to create a MyProcedures.ipf Alias in the "Igor Procedures"



Quit Igor, and restart it.

2. Writing Macros

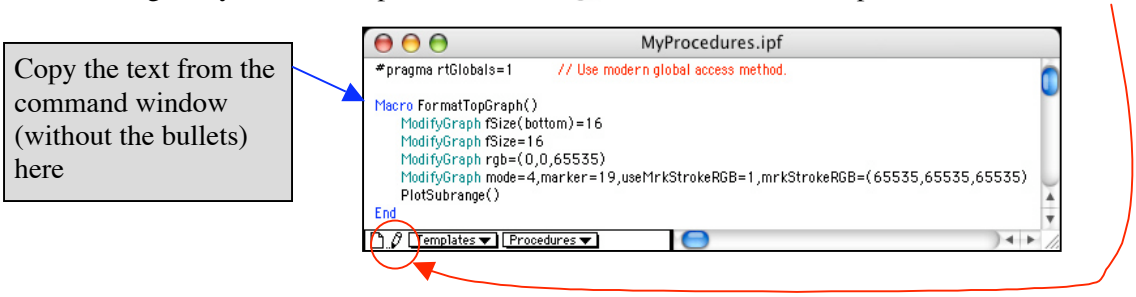
Many tasks in Igor can be automated. Sometimes you see yourself applying the same format to several graphs. Formatting a graph may take five or more commands. Instead of performing all those same commands for each graph, you can make a macro out of them.



In the Command Line Window, you can see that a new graph was created (Display m801,P801). Then four formatting commands were applied to it. Also a macro was applied PlotSubrange( ). Let’s write a macro that will easily allow you to reapply all seven formats and PlotSubrange( ) to a new graph.

2.1. Writing a macro

The Pencil icon tells you whether the MyProcedures.ipf file can be modified or not. This prevents accidental changes. By default, the pencil is locked. So make sure that the pencil is unlocked:



The text “#pragma rtGlobals=1” should be left untouched. Do not delete or modify it.

All macros start with “Macro” and the name of the macro (in this case FormatTopGraph). The parentheses after the name are necessary. The macro must have an End.

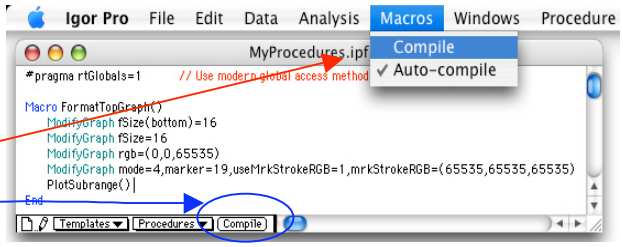
```
#pragma rtGlobals=1          // Use modern global access method.

Macro FormatTopGraph()
      ModifyGraph fSize(bottom)=16
      ModifyGraph fSize=16
      ModifyGraph rgb=(0,0,65535)
      ModifyGraph mode=4,marker=19,useMrkStrokeRGB=1, …
      PlotSubrange()
End
```

Special colors are used to make the text easier to read and identify commands. Red words are comments, which are ignored by the program, but help you understand your codes in the future.

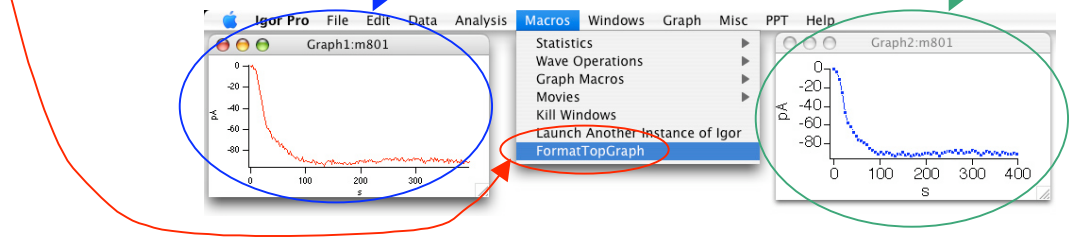
2.2. Compiling

Before your macro can be used, it must be compiled. The compilation allows the program to check for errors and to optimize the macro for faster execution. There are three ways you can compile:

- From the Macro menu
- From the Procedure window
- Click any other window. The auto-compile feature of Igor will try to compile it automatically.

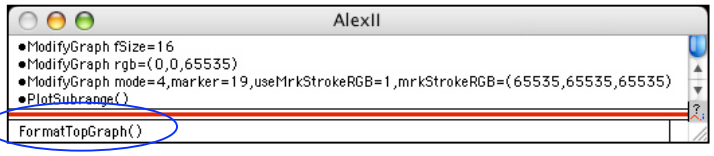


Go ahead and compile your macro. You macro should now be available under the macro menu as "FormatTopGraph". First, create a new graph, then run the macro. You will see the final result.



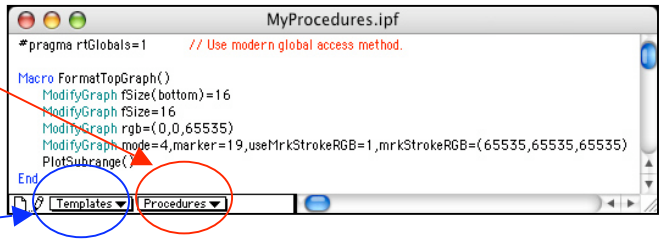
While running, the macro should also call the PlotSubrange() macro.

Note: You can also execute the macro from the command line. Just type "FormatTopGraph()".



2.3. Procedure window

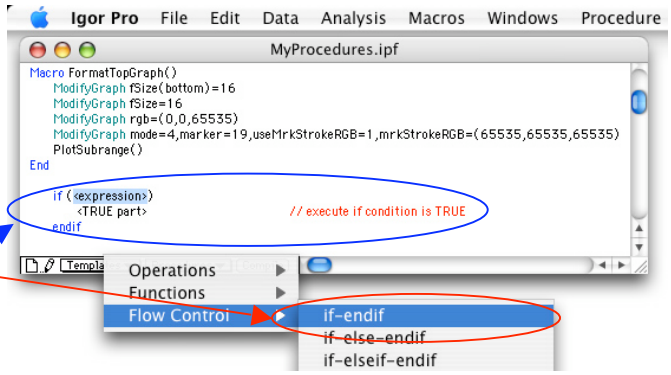
The procedure window also have keeps track of all your macros in the Procedures button. This allows you to quickly jump from one macro to another. This is helpful when you have lots of macros (e.g. the DefaultMacro).



There is also a Templates Button. This button brings up all of the commands that you can type into the procedure window and helps you on how to write it.

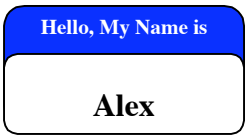
For example, if you don't remember how to write an "if-endif" statement (we'll come to this soon), you can bring up the template for the "if-endif" and click on it.

The template is then written in the procedure window.



3. Memory Variables

3.1. What's a Memory Variable?



A variable is a container that holds a piece of information. You can think of the "Hello, My Name Is" tag. You can write your name in the tag. If you use a pencil, you can erase it and write someone else's name inside it. You can have several of those tags, so you need to label them with unique names, such as sHelloMyNameIs01. Variable names can have letters and numbers, but NO symbols such as @#$%^&*.

There are two types of memory variables: Numeric and String variables. As a good practices (see appendix A), you should start every String variable name with a "s" (e.g. "sMyString01") and every Numeric variable with a "v" (e.g. "vMyNumeric01"). That way you will remember easily the type of variable it is and therefore, which commands to use with them. Note that Igor is not case sensitive. So vMyString01 is the same variable as vmystring01. Caps are easier to read.

3.2. Numeric Variables (or Variable)

A variable must only contain numerical values, such as 0, -59, 4.35, 3.14159265, 3.12345-38, inf (infinity), -inf (minus infinity), NaN (Not a Number). Please refer to the Appendix for Data Types.

```
Macro MyTest1()     // Numeric Variables
      Variable vMyValue01 = 10
      Variable vMyValue02 = 20
      Print vMyValue01
      Print vMyValue02
      Print vMyValue01 + vMyValue02
End
```

Run the macro ⇨

```
• 10
• 20
• 30

MyTest1()
```

Some operations that you can do with numeric variables:

Effect	How to Use It	Result	Notes
Operations: + - * /	Print vMyValue01 + vMyValue02	30	Add, subtract, etc.
Trigonometry: sin, cos, …	Print cos(vMyValue01)	-0.839072	Sin, cos, tan, asin, acos, etc.
Exponential: ln, log, exp, …	Print log(vMyValue01)	1	
Round: abs, ceil, max,…	Print abs(vMyValue01)	10	Absolute, ceiling, max, etc.

You can find more help in the Igor Help, "Command Help", "Functions"

3.3. String Variables (or String)

A string variable can contain any character and symbols, such as A-Z, a-z, 0-9, @#$%^&*. It is used to contain a list of wave names, a single wave name, etc. To view the content of a string in the command window, you can use the Print command. For example:

```
Macro MyTest2()     // String Variables
      String sMyString01 = "There are"
      String sMyString02 = "30 waves."
      Print sMyString01
      Print sMyString02
      Print sMyString01 + " " +sMyString02
End
```

Run the macro ⇨

```
• There are
• 30 waves.
• There are 30 waves.

MyTest2()
```

Some operations that you can do with strings:

Effect	How to Use It	Result	Notes
Concatenation	Print sMyString01 + " " + sMyString02	There are 30 waves.	
Mixing numbers with Strings	Print sMyString01 + vMyValue01 + "waves." Print sMyString01 + num2str(vMyValue01) + " waves."	ERROR!!! There are10 waves.	Can't mix them Convert using num2str
Convert to numeric value	Print str2num(sMyString02)	30	Ignores the text after
Uppercase	Print UpperStr(sMyString01)	THERE ARE	
Lowercase	Print LowerStr(sMyString01)	there are	
Number of characters	Print StrLen(sMyString01)	9	
Compare 2 strings' content (ignores case).	Print CmpStr(sMyString01, sMyString01) Print CmpStr(sMyString01, sMyString02)	0 1	No different content. Yes, different content.
Remove a string from end (ignores case).	Print RemoveEnding(sMyString01) Print RemoveEnding(sMyString01, "e are")	There ar Ther	Last letter is removed. Matched is removed.
Replace part of a string with another string	Print ReplaceString("are", sMyString01, "is")	There is	
Check if the string match a pattern. Use wild card *.	Print StringMatch(sMyString01, "the*") Print StringMatch(sMyString01, "*are")	1 1	Yes, string starts with "the" Yes, string ends with "are"
Search the string to see if it contains a word. Case sensitive. No wild card.	Print StrSearch(sMyString01, "are", 0) Print StrSearch(sMyString01, "ARE", 0) Print StrSearch(sMyString01, "ARE", 0, 2)	6 -1 6	Word is on the 6th char. Not found! Ask Igor to ignore case

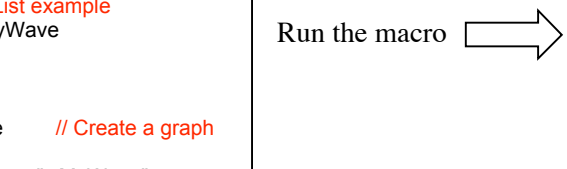
You can find more help in the Igor Help, "Command Help", "Functions", "String"

3.4. Waves

A wave is a collection of numeric variables. Waves are mostly used to plot graphs; however, they can also be used in programming to temporarily store data. Waves created in a macro will continue to exist even after the macro is over. Therefore, if the wave is to be used temporarily for data, it should be deleted at the end of the macro. You can't delete a wave that is displayed in a graph. To delete a wave, use:
KillWaves /Z <waveName> (/Z means that if the wave doesn't exist, don't prompt an error message).

```
Macro MyTest4()     // List example
    Make /O/N=3 wMyWave
    wMyWave[0] = 10
    wMyWave[1] = 1
    wMyWave[2] = 7
    Display wMyWave        // Create a graph

    String sWaveName = "wMyWave"
    Print sWaveName
    Display $sWaveName   // Create a graph
End
```

Run the macro



Two graphs will appear. This example shows you how to create a customized wave using the Make:

- The "/O" option tells Igor to overwrite any wave with the same name if it already exists.
- The "/N=3" option means that the wave will have 3 data points.

Since a wave is a collection of numeric variables, each variable is accessed using the wave name and an **INDEX**. An index is a sequential numbering of elements in a wave or list). The index starts from 0… so if you define 3 points, you will have indexes starting from 0 to 2. Defining "wMyWave[3]=5" will not cause an error, instead it will just be ignored.

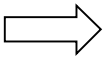
String sWaveName holds the wave's name only, not the wave and its numeric values. To refer to the wave and not the string, you must use a "$" symbol. **"Display sWaveName" will cause an error.**

You're not likely going to manipulate waves in programming, except to display them on a graph.

3.5. Lists

A list is a String that contains 0 or more items identified with a separator. An item is a word or phrase, such as a wave name. A separator is usually a semicolon, but can be a tab, colon, etc. Lists are very useful because it helps keep track of large number of items (strings or variables) into just one string rather than dealing with lots of memory variables.

```
Macro MyTest4()     // List example
      String sColor = "blue;red;yellow"
      Print ItemsInList(sColor)
      Print StringFromList(0,sColor)
      Print StringFromList(2,sColor)
End
```

Run the macro 

```
• 3
• blue
• yellow
```

```
MyTest4()
```

In this case, sColor contains 3 colors. However, just like waves, **Igor counts items from 0 (index)**. So for Igor, blue is item 0, red is item 1 and yellow is item 2. You can add new items to the list, delete an item, modify the text, etc. Here are some of the operations that you can do with lists.

Effect	How to Use It	Result	Notes
Find number of items	Print ItemsInList(sColor)	3	Useful for loops
Select 1 string from list	Print StringFromList(2, sColor)	yellow	
Find the character's position of an item	Print FindListItem("red", sColor) Print FindListItem("RED", sColor)	5 -1	red starts at the 5th character in the string
Find any item matching a word. Note: * is wild.	Print ListMatch(sColor, "re*") Print ListMatch(sColor, "*llow")	red; yellow;	red starts with "re" yellow ends with "llow"
Remove item by index	Print RemoveListItem(2, sColor)	blue;red;	
Replace ab item with another	Print ReplaceString("red", sColor, "gold")	blue;gold;yellow	
Sort list	Print SortList(sColor,"",0) Print SortList(sColor,"",1)	blue;red;yellow; yellow;red;blue;	Ascending Descending
Find the index of this item	Print WhichListItem("red", sColor)	1	red is at index 1
Remove an item from list.	Print RemoveFromList("red", sColor)	blue;yellow	
Add an item to the list	Print AddListItem("white", sColor)	white;blue;red;yellow	

Lists are useful to get the list of waves in a graph: String sWaveList = WaveList("*", ";","Win:") // * is wild
To get a list of waves ending with mean: String sWaveList = WaveList("*_mean", ";","Win:") // * is wild

3.6. Constants

```
#pragma rtGlobals=1   // Use modern global access method.
Constant cvPI = 3.14159265              // More precision
StrConstant csCancel = "CANCEL"     // Cancel value
```

You may also decide to use a PEN instead of a pencil to write your name: The content will not be changeable. For memory variables, they are called constants. Constants are usually defined on top the procedure file (right below the #Pragma statement) and will be available to all other procedure files, functions, macros, and command prompt. So you cannot define the same constant in two different procedure files (e.g. in MyProcedures and DefaultMacros). Constants are available to use as soon as the procedure is compiled. You should name your constants starting with "c", the variable type ("v" or "s"), then the name of the constant.

3.7. Globals

```
Macro FormatTopGraph()
   Variable/G gvUserNumber = 10000   // for user's selection
   String/G    gsWaveName   = "Alex"   // wave name

   … (write your code here)

   KillVariables/Z gvUserNumber
   KillStrings/Z    gsWaveName
End
```

Global variables are defined within a macro or function by using "/G", and will be available to all other procedure files, functions, macros, and command prompt. Unlike constants, you can redefine the same global variable in many functions, but they will overwrite each other's value. Therefore, it's NOT recommended to use global variables. If you must use a global variable (e.g. for panels, such as the one used in the MultipleWaveSelection function), you should erase it from memory when you don't need it anymore. To delete a global variable, use the KillStrings and KillVariables command.

4. Macros vs. Functions

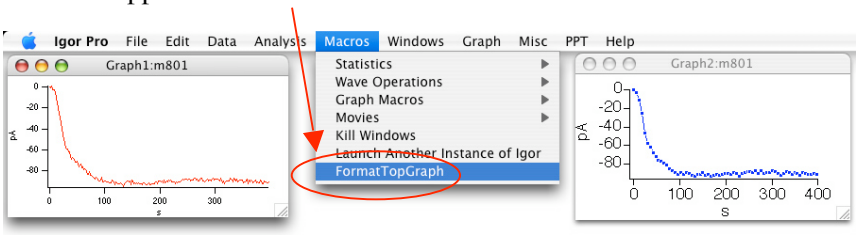
Macros and functions are series of commands that are executed when you run them. The main difference is that a function MAY return a value. A value is a piece of information that can be used by another macro or function. Both can be executed on the command prompt by typing their name.

4.1. Macros

As you have seen previously: All macros start with "Macro" and the name of the macro (in this case FormatTopGraph). The parentheses after the name are necessary. The macro must have an End.

```
Macro FormatTopGraph()
	ModifyGraph fSize(bottom)=16
	ModifyGraph fSize=16
	ModifyGraph rgb=(0,0,65535)
	ModifyGraph mode=4,marker=19,useMrkStrokeRGB=1, …
	PlotSubrange()
End
```

Macros will appear in the Macro menu



4.2. Functions

Functions start with "Function" and the name of the function (in this case fnFormatTopGraph2). fn is used to denote a function (see the appendix on naming convention).

```
Macro FormatTopGraph2()
	fnFormatTopGraph2()
End

Menu "Macros"
	"Format the top graph", /Q fnFormatTopGraph2()
	help = {"Format the top graph with my customized settings"}
End

Function fnFormatTopGraph2()
	ModifyGraph fSize(bottom)=16
	ModifyGraph fSize=16
	ModifyGraph rgb=(0,0,65535)
	ModifyGraph mode=4,marker=19,useMrkStrokeRGB=1, …
	PlotSubrange()
End
```

Unlike macros, functions do not appear on the Macro menu. So you need to have a macro that calls it or a Menu. If you create a menu for your MyProcedures file, all macros for your MyProcedures will no longer appear on the Macro menu. You can use Macros for simple programs. For all others, you should use functions if possible. Put all your codes in the function and keep the macro at the bare minimum.

The difference between functions and macros is that functions MAY return a string or numeric value. Take for example the function fnSquare(vMyXValue), where vMyXValue is a **parameter**. A parameter is a value (string or numeric) passed to a function to compute.

```
Function fnSquare(vMyXValue)
	// This function returns the square value
	Variable vMyXValue
	Return (vMyXValue * vMyXValue)
End
```

Run the macro ⇨

```
• 25

Print fnSquare(5)
```

Using a function to call another function:

```
Function /S fnFormula(vValue)
	// Computes x*x + 1
	Variable vValue
	Return "Result: " + num2str( fnSquare(vValue) + 1)
End
```

Run the macro ⇨

```
• New Result: 50

Print "New " + fnFormula(7)
```

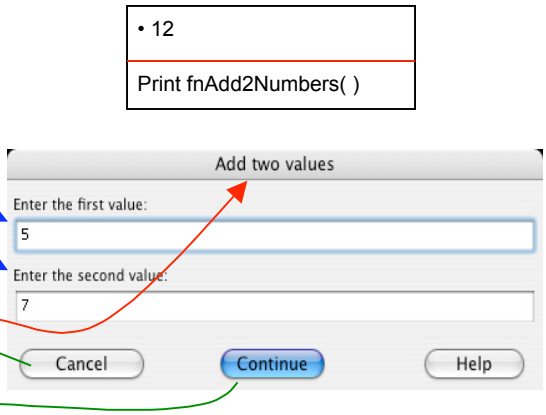
Note: The /S after a function means that the function will return a String.
Note: num2str is necessary to convert a numeric value to a string (for this example).

4.3. Data Input

Sometimes you may want to ask for more than one value as parameter, or simply to have a nice interface to get values from the user. Making interface for data input is different for Macros and Functions: we will cover only the interface for functions.

In functions, you can use the Prompt and DoPrompt commands. You will also have default values for your variables:

```
Function fnAdd2Numbers( )
	// Get 2 values from the user and add them
	Variable vValue01 = 5	// Set 5 as default value
	Variable vValue02 = 7	// Set 7 as default value

	Prompt vValue01, "Enter the first value:"

	Prompt vValue02, "Enter the second value:"

	DoPrompt "Add two values", vValue01, vValue02

	If (V_Flag == 1)
		Return 0		// User pressed cancel... exit
	EndIf

	Return (vValue01 + vValue02)
End
```



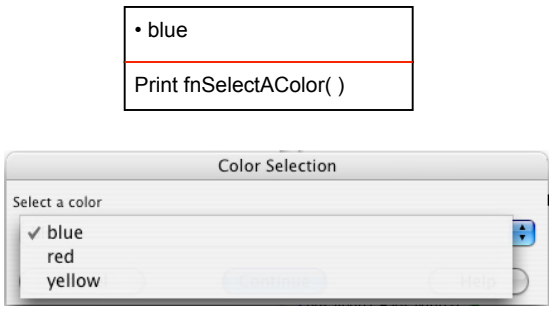
To use the prompt: Type the variable name and the customized text with it.
To use the DoPrompt: Type the window's title and which values to include.

If the user pressed the "Cancel" button, the function will return 0 (**must** return a value)
If the user pressed the "Continue" button, the function will return (vValue01 + vValue02) = 12

You can ask up to 10 variables in a DoPrompt. After that you must create a second DoPrompt.

You may also use PopUp to provide the user with a limited amount of choices. You use PopUp in conjunction with Prompt. For example:

```
Function /S fnSelectAColor( )
	// Ask the user to select a color
	String sColor = "blue;red;yellow"	// colors selection
	Variable vColor = 1		// Default is blue, NOT red

	Prompt vColor, "Select a color", PopUp, sColor

	DoPrompt "Color Selection", vColor

	If (V_Flag == 1)
		Return ""		// User pressed cancel... exit
	EndIf

	Return "Color:" + StringFromList (vColor - 1, sColor)
End
```



This function returns a String. So if the user cancels, it must return an empty string ""

Strangely enough, with PopUps, the list starts with an index of 1, not 0. That is the only exception with Igor. All other commands start with an index of 0. So, you must deduct 1 from the selection! Be careful.

5. Flow Control

5.1. Boolean and Boolean Operators

Booleans can be True or False. False is always represented with a 0 value; true with a 1 value. True/False are mostly used in flow control statements, such as If-EndIf, etc. Booleans are usually the comparison between two variables. They use different operators:

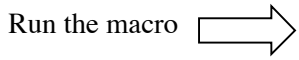
Effect	How to Use It	Result	Notes
Equal comparison	Print 1 == 2	0	No spaces between symbols
Not equal comparison	Print 1 ! = 2	1	No spaces between symbols
Negation	Print ! (1 == 2)	1	not (false) = true
Less than	Print 1 < 2	1	
Less or equal to	Print 1 <= 2	1	
Greater than	Print 1 > 2	0	
Greater or equal to	Print 1 >= 2	0	

There are other operators, but you probably won't need to use them.

5.2. If-EndIf

Checks if a condition is true. If it's true, execute the commands. If it's not true, just skip the command. The value of sResult will remain unchanged:

```
Function /S fnIsGreaterThan40(vMyXValue)
    // Checks if value is greater than 40
    Variable vMyXValue
    String sResult = "No"      // Default value needed

    If (vMyXValue > 40)
        sResult = "Yes"        // Do this if true
    EndIf
    Return sResult
End
```

Run the macro 

• Yes

Print fnIsGreaterThan40 (55)

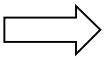
• No

Print fnIsGreaterThan40 (35)

5.3. If-Else-EndIf

Checks if a condition is true. If it's true, execute the commands. If it's not true, do the next command after the else:

```
Function /S fnIsGreaterThan50(vMyXValue)
    // Checks if value is greater than 50
    Variable vMyXValue
    String sResult             // No default value needed

    If (vMyXValue > 50)
        sResult = "Yes"        // Do this if true
    Else
        sResult = "No"         // Do this if false
    EndIf
    Return sResult
End
```

Run the macro 

• Yes

Print fnIsGreaterThan50 (55)

• No

Print fnIsGreaterThan50 (35)

5.4. If-ElseIf-EndIf

Checks if a condition is true. If it is, execute the first statement. If not, check if the second condition is true (using the ElseIf). If it is, execute the second statement. If it is not, execute the Else statement.

```
Function /S fnIsGreaterThan60(vMyXValue)
	// Checks if value is greater than 60
	Variable vMyXValue
	String sResult			// No default value needed

	If (vMyXValue > 60)
		sResult = "Yes"		// Do this if true
	ElseIf (vMyXValue < 60)	// Check second condition
		sResult = "No"		// Do this if second is true
	Else
		sResult = "It's equal" // Do this if second is false
	EndIf
	Return sResult
End
```

Run the macro ⇨

```
• Yes
Print fnIsGreaterThan60 (65)
```

```
• No
Print fnIsGreaterThan60 (35)
```

```
• It's equal
Print fnIsGreaterThan60 (60)
```

So, you can have as many ElseIf as you want… but it can get quite complicated.

5.5. Switch-Case-EndSwitch

If you have many conditions, having many ElseIf is really not the solution. I use the Switch command more frequently in this situation. It's easier to understand and cleaner. You must have a Break command to tell Igor that you are done comparing. Otherwise, it will continue and you may end up with strange results.

```
Function /S fnIsGreaterThan70(vMyXValue)
	// Checks if value is greater than 70
	Variable vMyXValue
	String sResult			// No default value needed

	vMyXValue = vMyXValue - 70		// try to normalize
	vMyXValue = vMyXValue / abs(vMyXValue)
	Switch (vMyXValue)
		Case 1:				// condition 1
			sResult = "Yes"		// Do this if true
			Break
		Case -1:			// condition 2
			sResult = "No"		// Do this if true
			Break
		Default:			// if nothing is true, do this
			sResult = "It's equal"	// Default value
	EndSwitch
	Return sResult
End
```

Run the macro ⇨

```
• Yes
Print fnIsGreaterThan70 (80)
```

```
• No
Print fnIsGreaterThan70 (60)
```

```
• It's equal
Print fnIsGreaterThan70 (70)
```

The switch case usefulness is most obvious when the selection is finite but not overwhelming:

In this example, the graph's waves will change color based on the user's selection.

```
•
fnChangeWaveColor()
```

```
Function fnChangeWaveColor( )
	// Ask the user to select a color
	String sColor = "blue;red;yellow"	// colors selection
	Variable vColor = 1		// Default is blue, NOT red
	Prompt vColor, "Select a color", PopUp, sColor
	DoPrompt "Change Wave Color", vColor

	Switch (vColor)
		Case 1:				// condition 1
			ModifyGraph rgb=(0,0,65535)	// blue
			Break
		Case 2:				// condition 2
			ModifyGraph rgb=(52428,1,1)	// red
			Break
		Default:			// if nothing is true, do this
			ModifyGraph rgb=(52428,52425,1) // yellow
	EndSwitch
	Return 0
End
```

5.6. StrSwitch-Case-EndSwitch

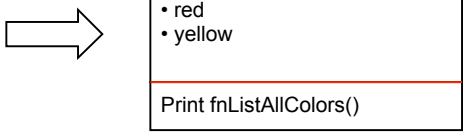
Sometimes you may want to compare strings, instead of numeric values. Then you will need to use the StrSwitch command. For example, you can have:

```
Function fnChangeWaveColorByName( )
	String sColor = "blue"			// colors selection
	Prompt sColor, "Type a color: "
	DoPrompt "Change Wave Color by Name", sColor

	StrSwitch (sColor)
		Case "blue":				// condition 1
			ModifyGraph rgb=(0,0,65535)		// blue
			Break
		Case "red":				// condition 2
			ModifyGraph rgb=(52428,1,1)		// red
			Break
		Default:				// if nothing is true, do this
			ModifyGraph rgb=(52428,52425,1) // yellow
	EndSwitch
	Return 0
End
```

5.7. For-EndFor

The For loop is very useful when you want to enumerate all the items in a list. It is most recommended when you know how many indexes are to be performed. For example:

```
Function fnListAllColors( )
	String sColor = "blue;red;yellow"	// colors selection
	Variable vNbOfColors = ItemsInList (sColor)
	Variable vIndex

	For (vIndex = 0;vIndex <  vNbOfColors; vIndex = vIndex + 1)
		Print ItemInList(vIndex, sColor)
	EndFor
	Return vIndex				// display the last value
End
```

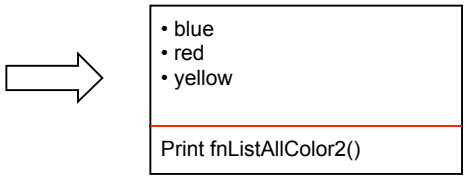


You can also use the For loop for any other purposes, such as a Factorial function. Note that Igor starts lists from 0.

5.8. Do-While

The Do while loop is similar to the For loop. It's mostly used if you do not know how many times you need to loop (or other factors). The following example is based on the previous one, so you can make comparisons.

```
Function fnListAllColors2( )
	String sColor = "blue;red;yellow"	// colors selection
	Variable vNbOfColors = ItemsInList (sColor)
	Variable vIndex = 0		// you must provide a default value

	Do
		Print ItemInList(vIndex, sColor)
		vIndex = vIndex + 1
	While (vIndex < vNbOfColors )
	Return vIndex				// display the last value
End
```



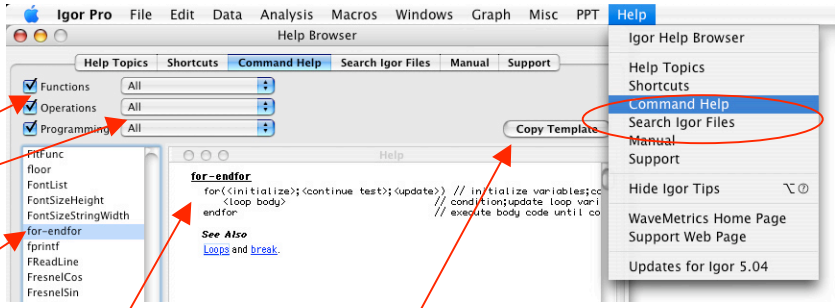
Another difference between the Do-While and the For-EndFor is that the statements inside a Do-While will be executed once, regardless of whether the condition is true or false. The reason is that the condition is placed after the statements.

A downfall is the potential infinite loop. If you get one of those, try Command-(period) to break.

6. HELP!!!

6.1. How to Use the Command Help

In the Help menu, select "Command Help".

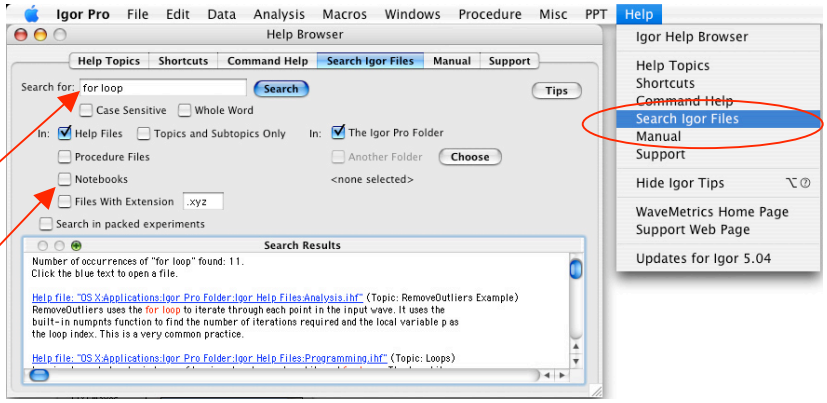


You can limit the list of commands by clicking on the checkboxes and selecting from the list of topics.

Select a command from the list and you will see details and examples (most of the time) for that command. Click on the "Copy Template" if you want to be able to copy the format into your procedure file.

6.2. How to Use the Search Igor Files

In the Help menu, select "Search Igor Files".



Type the words or text that you want to search. Note that Igor search for the exact phrase, such "For loop". You can't type: "For and While examples"

You can ask Igor to include procedures (for their examples), notebooks, etc.

6.3. Understanding the syntax

Syntax is how a command must be structured for it to work. For example:

KillVariables [/A/Z] [variableName [, variableName]...]

The KillVariables operation discards the named global numeric variables.

Flags:

- /A kill all global variables in the current data folder. If you use /A, omit variableName.
- /Z doesn't generate an error if a global variable to be killed does not exist. To kill all global variables in the current data folder, use KillVariables/A/Z.

Items in brackets [and] are optional.
Items in double brackets are optional continuous list.

E.g. KillVariables
E.g. KillVariables /A
E.g. KillVariables /Z vMyValue01, vMyValue02

Appendix A: Good Practices and Things to Know

Naming Conventions

- Temporary waves when programming should start with a w, such as wMyWave.
- String variable's names should start with a "s", such as sMyString01.
- Numeric variable's names should start with a "v", such as vMyValue01.
- Global variables should start with a "g", such as gsMyString, gvMyVariable.
- Constant variables should start with a "c", such as csMyString, cvMyVariable.
- Reserved words are words that you should not use as a name for macros, functions, variables, etc. E.g. Macro ModifyGraph() will not work because it already exist in Igor.
- Memory variable names, wave names, macro names and function names can only contain a-z, A-Z, 0-9. Do not use spaces and symbols (except underscore _). Use Caps or underscore to make a name easy to read, such as MyTopGraph or My_top_graph. Be descriptive: sSelectedWave as opposed to sSW.

Variables

- Do NOT use global variables, unless absolutely necessary. Delete it when you are done using it.

Macros and Functions

- Function's names should start with "fn", such as fnSinus. So you'll know that you are expecting a value back
- Macro's name can have a convention as well. However, because they are listed on the Macro menu, it's not as fun to read mcFormatToGraph as opposed to just FormatTopGraph. If they don't appear on the menu, you can use "mc" to mark it as a macro. This helps identify them as user defined macros, as opposed to reserved commands.

Programming Tips

- Use indentation. It makes the code easier to read.
- Re-use codes… don't reinvent the wheel!
- Use comments, lots of comments… If possible, every line should be commented. To make a comment, use two slash symbols (result is in red), such as // this is a comment

Debugging your Code

- Did you use comments? If not, go back to each line and write comments. By re-reading explaining your code, you may notice what you did wrong.
- Use the "Print" command to keep track of the variable values.

Data Types

Name	Range of Values	Bits	Description
bit	0 or 1	1	Smallest memory unit
boolean	False(0) or True(1)	1	In Igor, this is only visible in condition testing.
byte	0 to 255	8	Not used in Igor, but it's good to know
integer	-999999 to 999999	16	Integers are whole numbers (no decimal points). If more than 6 digits are used, Igor will use powers of 10, such as 2e+06.
single precision	-3.4e38 to 3.4e38 and e-38 to e+38	32	Up to 6 digits can be visible at all time, such as 3.12345e+38.
double precision	-1.79e308 to 1.79e308 and e-308 to e+308	64	Up to 6 digits can be visible at all time, such as 1.12345e+308. Any amount past 1.79e308 will be considered infinity (inf).
complex	unknown	-	Igor can also use complex numbers. E.g. Print cmplx(1,2)
char	a-z, A-Z, 0-9, symbols	8	1 character.
string	Unlimited	-	Any amount of characters.