# Introduction for New Users
# Wing IDE Professional

Wingware
www.wingware.com

Version 4.1.13
May 3, 2013

Thanks for trying Wing IDE Professional! To get started, please choose from the following:

- For a guided tour **try the tutorial**.
- To try Wing on your own, see the **quick start guide**.
- For hints on using Wing with GUI or Web development frameworks, *also* see the relevant **How-Tos**

## Contents

# Wing IDE Tutorial

This document introduces Wing IDE by taking you through its feature set with a small coding example. For a faster but less informative introduction, see the **Wing IDE Quick Start Guide**.

If you are new to programming, you may want to check out the book Python Programming Fundamentals and accompanying screen casts, which use Wing IDE 101 to teach programming with Python.

To get started, press the `Next` (down arrow) icon in the toolbar immediately above this page.

## 1.1. Tutorial: Getting Started

In addition to **installing Wing IDE**, you also need to take the following steps before starting the tutorial:

**(1) Install Python**

To get Python, download it now from python.org or wingware.com. This tutorial will work with Python version 2.0 or later.

If the above links don't work or bring up the wrong browser, you may need to define the `BROWSER` environment variable to the name of the browser executable you wish to use (for example: `mozilla`) and restart Wing IDE.

On Linux/Unix, you can also add a browser command line to your **URL Display Commands** preference. This is recommended only if your preferred browser doesn't work when specified with the `BROWSER` environment variable. Setting `BROWSER` will generally do a better job reusing browser instances and creating and raising browser windows as needed.

**(2) Copy the Tutorial Directory**

Next, copy the entire `tutorial` directory out of your Wing IDE installation to a location where you will have write access to the files in it. You can do this manually or use the following link to execute a script that will prompt you for a target directory to copy the tutorial info: **Copy Tutorial Now**

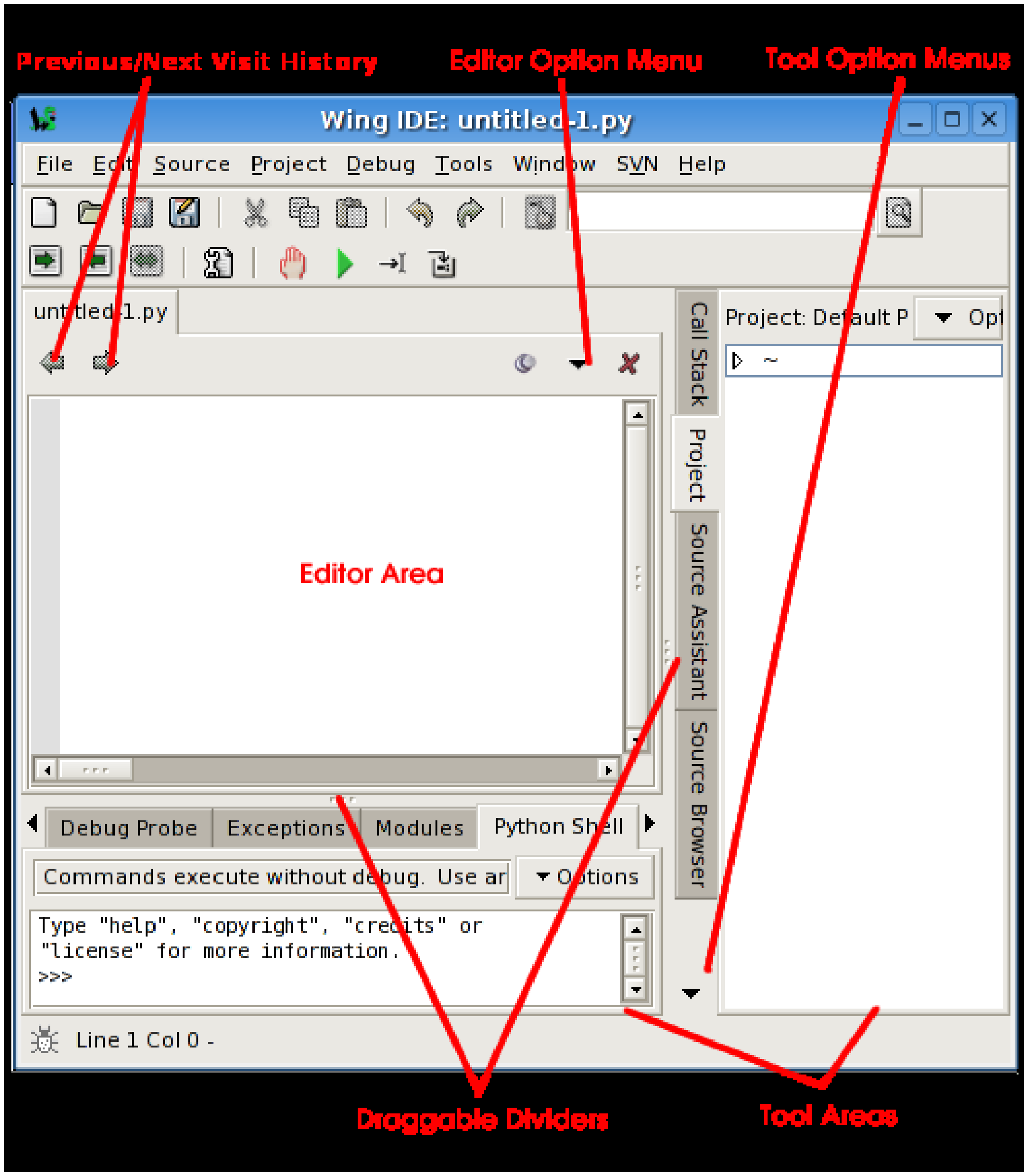
> We welcome feedback and bug reports, both of which can be submitted directly from Wing IDE using the `Submit Feedback` and `Submit Bug Report` items in the Help menu, or by emailing us at support at wingware.com.

To get to the next page in the tutorial, use the `Next Page` icon shown in the toolbar just above this text.

## 1.2. Tutorial: Getting Around Wing IDE

Let's start with some basics that will help you get around Wing IDE while working with this tutorial.

Wing IDE's user interface is divided into an editor area and two tool boxes separated by draggable dividers. Use the option menus in each area to create splits or move tools around. The Previous/Next Visit History buttons and the `Next Document`, `Previous Document` and `Most Recent Document` items in the Window menu can be used to switch quickly between documents in the editor area, such as this tutorial and the source files you'll be working with later.



Commands execute without debug. Use ar

```
Type "help", "copyright", "credits" or
"license" for more information.
>>>
```

## Configuration Options

There are many configuration options available for customizing the user interface. Some of these are described below. Once you make changes to any of these, your settings will be remembered in your project file and preferences.

**Editor Personality** -- If you are used to another editor such as Visual Studio, VI or Vim, Emacs, or Brief, you may want to put Wing into a more familiar keyboard mode using the **Personality** preference. Be sure to click `OK` or `Apply` so the changes take effect.

**Tab Key Action** -- In Python code, the tab key in Wing defaults to indenting a selected region or the current line to the "correct" computed indent level, to the extent that Wing can determine this from context. In non-Python files, the tab key increases indent one level. To change this, use the **Tab Key Action** preference.

**Minimizing Tool Boxes** -- By clicking on an already-active tool tab in one of the tool boxes, the entire area will be minimized down so that only the tabs for the area are visible. Clicking again on any tab will restore the tool box to its previous size. Or, use F1 and F2 to toggle the state of the two tool boxes. This is a convenient way to increase space available to the editor or other tool box.

Shift-F2 can also be used to maximize the editor area temporarily, hiding the tools and toolbar until Shift-F2 is pressed again.

**Splitting Panels** -- The editor area and tool boxes can be split into multiple sub-panels by using the editor and tool box option menus. These can be changed with the editor options menu, which is accessed either by clicking on the dropdown icon in the top right of the editor area, or by right-clicking on the notebook tabs. Note that when splitting the editor area, each new split will show the same files as all others; this allows for editing multiple parts of the same file.

Splitting your editor area or creating a separate `Help` tool window may make it much

easier to get around this tutorial.

The number of splits shown by default in tool boxes will vary depending on the size of your monitor.

**Moving and Adding Tools** -- Tools can be moved among the tool boxes or out to separate windows by using the tool box option menu. Additional instances of any tool can be created from the tool box option menu or in a separate window from the `Window` menu.

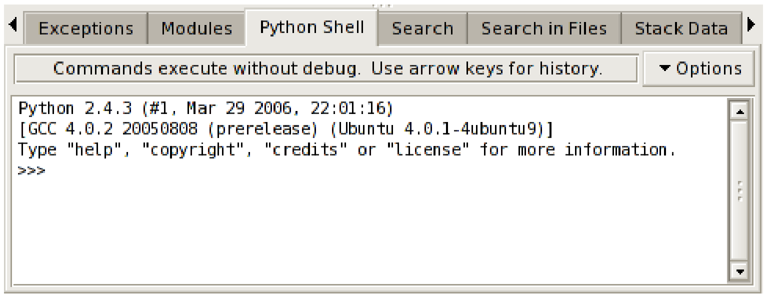
**Adding Document Windows** -- Additional document windows can also be created from the `Window` menu. Each separate document window contains its own set of open files.

**Other Options** -- **Source Code Font/Size** and **Display Font/Size** can be altered. The toolbar's appearance can be changed using the **Toolbar Size** and **Toolbar Style** preferences. The tool boxes can be moved from right to left or bottom to top by right-clicking on the tool tabs. The editor option menu, accessed by clicking on the drop down indicator at top right of the editor area, allows selecting between using notebook tabs or a popup menu to navigate between open editors.

For more information on adjusting the user interface to your needs, see the **Customization** chapter of the manual.

## 1.3. Tutorial: Check your Python Integration

Before starting with some code, let's make sure that Wing has succeeded in finding your Python installation (the latest version is preferred if you have multiple versions installed). To check this, bring up the **Python Shell** tool. After a moment, it should show you the Python command prompt like this:



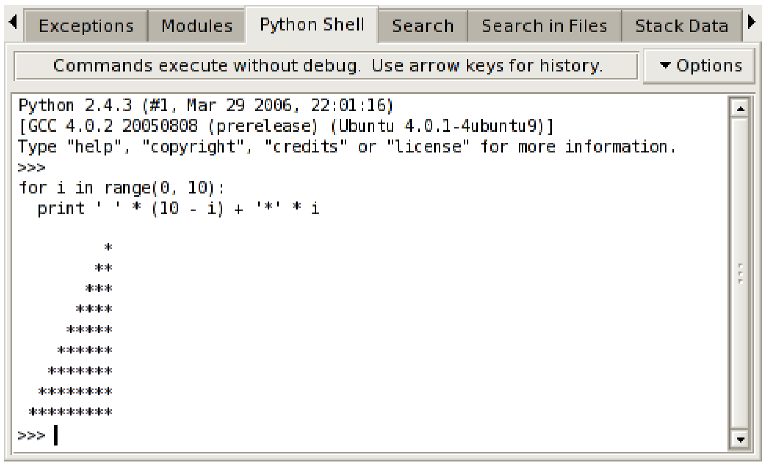
```
Python 2.4.3 (#1, Mar 29 2006, 22:01:16)
[GCC 4.0.2 20050808 (prerelease) (Ubuntu 4.0.1-4ubuntu9)]
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

If this is not working, or the wrong version of Python is being used, you can point Wing in the right direction with the **Python Executable** setting in `Project Properties`, available from the toolbar and `Project` menu. You will need to `Restart Shell` from `Options` in the Python Shell tool after altering this property.

Once the shell works, copy/paste or drag and drop these lines of Python code into it:

```
for i in range(0, 10):
  print(' ' * (10 - i) + '*' * i)
```

This should print a triangle as follows:



```
Python 2.4.3 (#1, Mar 29 2006, 22:01:16)
[GCC 4.0.2 20050808 (prerelease) (Ubuntu 4.0.1-4ubuntu9)]
Type "help", "copyright", "credits" or "license" for more information.
>>>
for i in range(0, 10):
  print ' ' * (10 - i) + '*' * i

         *
        **
       ***
      ****
     *****
    ******
   *******
  ********
 *********
>>> 
```

Notice that the shell removes common leading white space when blocks of code are copied into it. This is useful when trying out code from source files.

Now type something in the shell, such as:

```
import sys
sys.getrefcount(i)
```

Note that Wing offers auto-completion as you type and shows call signature and documentation information in the **Source Assistant**.

You can create as many instances of the Python Shell tool as you wish; each one runs in its own private process space that is kept totally separate from Wing IDE and your debug process.

## 1.4. Tutorial: Set Up a Project

Now we're ready to get started with some coding. The first step in working with Wing IDE is to set up a project file so that Wing can find and analyze your source code and store your work across sessions.

Wing starts up initially with the Default Project. You can use that project to start your work on the tutorial. If you would prefer to create a new project instead, use `New Project` in the `Project` menu. **Note:** After doing this, you will need to open up the tutorial again from the Help menu, since Wing closes open documents when projects are closed.

To make it easier to work on source code and read this tutorial at the same time, you may want to right click on the editor tab area and select `Split Side by Side`.

Next, add your source files to the project. You can do this with the `Add` items in the `Project` menu, or by right clicking on the **Project** tool. For the purposes of this tutorial, use `Add Directory` to add all files in your copy of the `tutorials` directory. If you haven't already copied the `tutorials` directory from your Wing IDE installation, please do so now as described in **Tutorial: Getting Started**.

Once your files have been added, save the project to disk with `Save Project` or `Save Project As` in the `Project` menu. Use `tutorial.wpr` as the project file name and place it in the `tutorial` directory that you created earlier.

## Browsing Files

Files in your project can be opened by double clicking or right-clicking on the file list in the `Project` tool. When the `Follow Selection` item in the `Options` menu at top right of the Project view is checked, Wing will also display the source code for files that are single clicked. However, these files are opened in a transient mode so that they are automatically closed again when another file is brought up. This mechanism helps to prevent huge numbers of files being opened when stepping in the debugger or browsing files.

This mode in which a file is opened is indicated with the stick pin icon in the top right of the editor area:

-- Indicates the file is opened permanently until it is closed explicitly by the user.

-- Indicates that the file is opened transiently and will auto-close except if it is edited.

Clicking on the pin icon toggles between the available modes. Right-clicking on the icon displays a menu of recently visited files. Note that this contains both transient and sticky files, while the `Recent` list in the `File` menu contains only sticky files.

The number of transient editors to keep open, in addition to those that are visible is set with the **Transient Threshold** preference.

Note that you can alter the project display to sort files into a deep hierarchy, a flattened hierarchy, or by mime type. These are available from the `Options` menu in the project view.

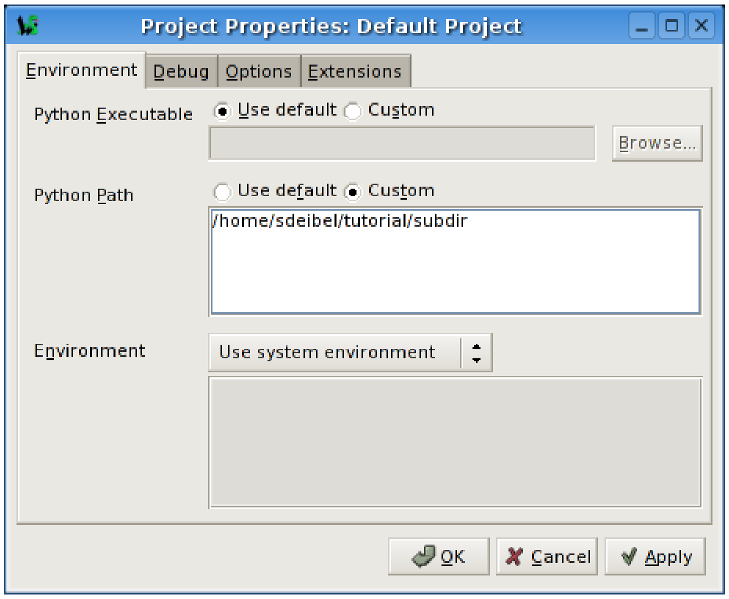
### Shared Project Files

If you plan to use Wing IDE in a development team that shares project files in a revision control system such as `CVS`, `Subversion`, or `Perforce SCM`, be sure to change your project to `Shared` using the **Project Type** property. This separates the project into two files: `*.wpr` with shared project data and `*.wpu` with user-specific data. Check only the `*.wpr` file into revision control to avoid revision conflicts resulting from concurrent edits.

## 1.5. Tutorial: Setting Python Path

Whenever your Python source depends on `PYTHONPATH` (either set externally or by altering `sys.path` internally), you will also need to tell Wing about your path.

This value can be entered from the `Project Properties` dialog, which is accessible from the `Project` menu and the toolbar:



Python Executable: [✓ Use default] Custom

Python Path: Use default [✓ Custom]

/home/sdeibel/tutorial/subdir

Environment: Use system environment

For this tutorial, you will at least need a `PYTHONPATH` that includes the `subdir` sub-directory of your `tutorials` directory as shown in the figure above. This contains a module used as part of the first coding example.

Note that in the screen shot above the `PYTHONPATH` has been set with the full path to the directory `subdir`. This is strongly recommended because it avoids potential problems finding source code when the starting directory is ambiguous, both for source code analysis purposes and in Wing's debugger. A partial path can be specified, but Wing will issue a warning explaining why this is a bad idea.

The configuration is used here for illustration purposes. You could easily run the example code without a `PYTHONPATH` by moving the `path_example.py` file to the same location as the example scripts, or by placing it into your Python installation's site-packages directory. Either of these allows Python to find the modules without altered `PYTHONPATH`.

## 1.6. Tutorial: Introduction to the Editor

By now Wing will have found and analysed the tutorial examples, and all the modules that are imported and used by them. This analysis process runs in the background and allows Wing to present you with better support during inspection and editing of code. With larger code bases, you may notice the CPU load from this process, but with this tutorial the analysis will happen instantaneously after the project has been configured.

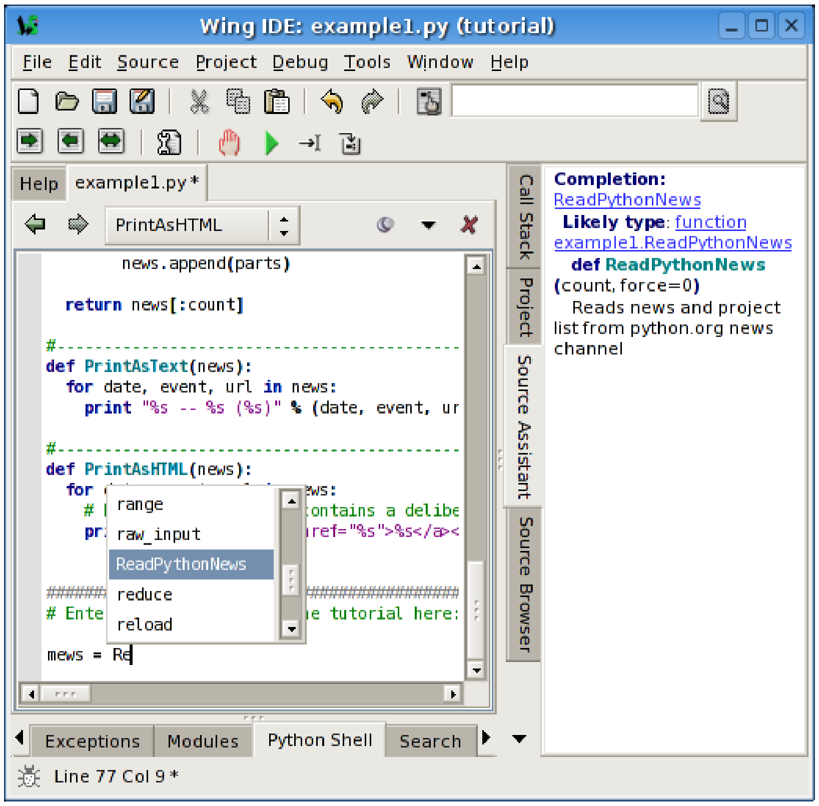
The editor's auto-completer and Source Assistant are two of the most important analysis-driven tools in Wing IDE.

To try these out, double click or right-click on the file `example1.py` in the project panel. Also bring the **Source Assistant** tool to front. This is where Wing IDE shows documentation, call signature, and other information as you move around your source code or work within other tools, so it's a good idea to keep it visible while editing.

Scroll down to the bottom of `example1.py` and enter the following code by typing (not pasting) it into the file:

```
news = Re
```

Notice that Wing shows you a popup menu of completion options as you type. You can press tab to enter the currently selected value, or scroll around in the list with the arrow keys. When you typed "news" this completer wasn't helpful because you had not yet defined `news` as a symbol in your source. However, once you move on to type " = Re", Wing will display another completion list with `ReadPythonNews` highlighted. Notice that the Source Assistant updates to show call information for that function, or for whatever value is selected in the auto-completer:



```
        news.append(parts)

    return news[:count]

#-----------------------------------------
def PrintAsText(news):
  for date, event, url in news:
    print "%s -- %s (%s)" % (date, event, ur

#-----------------------------------------
def PrintAsHTML(news):
  for                                 ws:
    # |                               ontains a delibe
    pr                                ref="%s">%s</a><

#######                               ###################
# Ente                                e tutorial here:

news = Re
```

```
range
raw_input
ReadPythonNews
reduce
reload
```

**Completion:**
ReadPythonNews
**Likely type**: function
example1.ReadPythonNews
**def ReadPythonNews**
(count, force=0)
Reads news and project list from python.org news channel

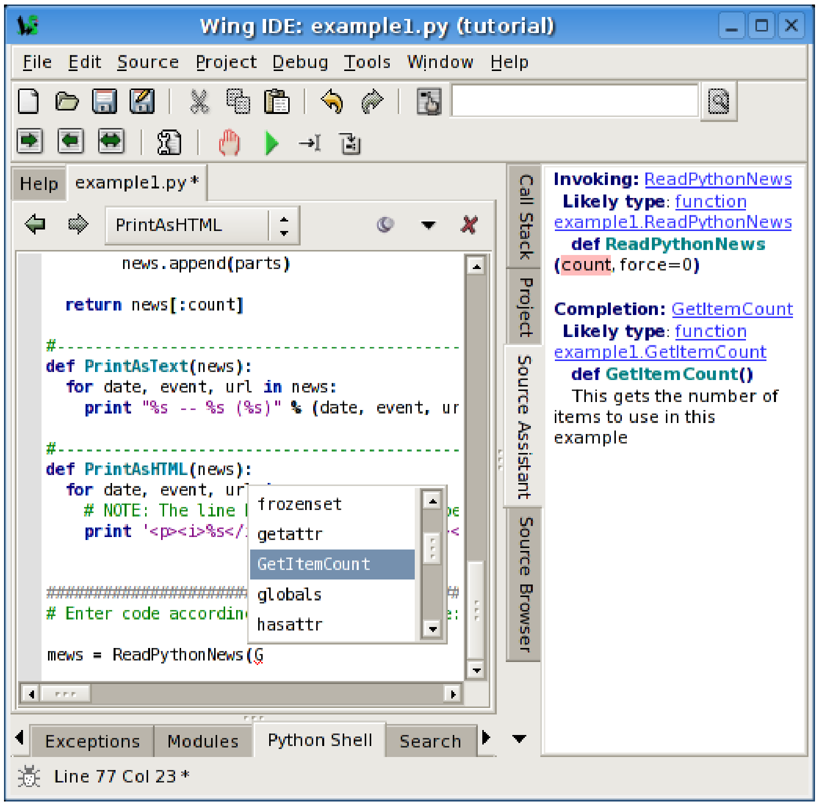
Next, press the `Tab` key to enter the completion of `ReadPythonNews` and enter `(`. You should now have this code in your editor:

```
news = ReadPythonNews(
```

If you are used to using the `Enter` key for auto-completion, add it to the **Completion Keys** preference.

Duplicate substitution definition name: “note”.

Type `Get` to start entering the first argument to `ReadPythonNews`. You will see the Source Assistant alters its display to highlight the first argument in the call information for `ReadPythonNews` and adds information on the argument’s completion value:



```
        news.append(parts)

  return news[:count]

#------------------------------------------
def PrintAsText(news):
  for date, event, url in news:
    print "%s -- %s (%s)" % (date, event, ur

#------------------------------------------
def PrintAsHTML(news):
  for date, event, ur
    # NOTE: The line l                       be
    print '<p><i>%s</                        <


##########################                 ##
# Enter code accordin                        e:

mews = ReadPythonNews(G
```

```
frozenset
getattr
GetItemCount
globals
hasattr
```

**Invoking:** ReadPythonNews
**Likely type**: function
example1.ReadPythonNews
**def ReadPythonNews**
(count, force=0)

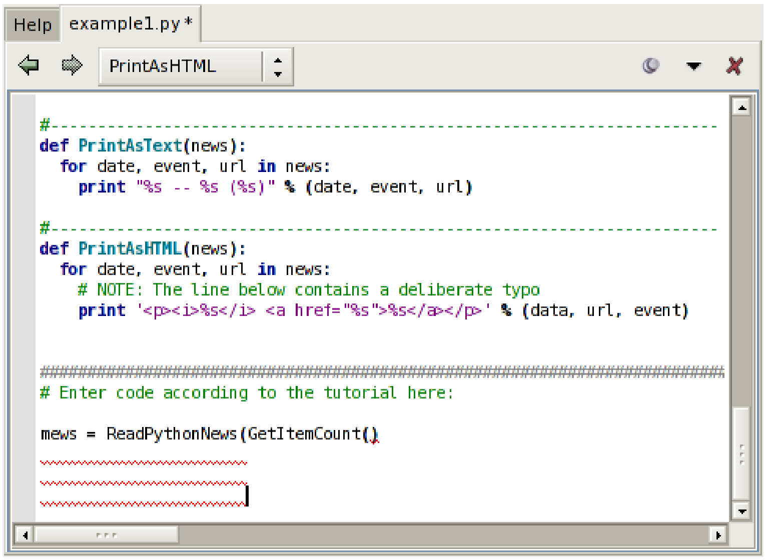
**Completion:** GetItemCount
**Likely type**: function
example1.GetItemCount
**def GetItemCount()**
This gets the number of items to use in this example

The docstring for `ReadPythonNews` is temporarily hidden to conserve screen space (but this can be toggled with the `Show docstring during completion` option in the context menu obtained by right-clicking on the surface of the Source Assistant).

Now continue entering the rest of the source line so you have the following almost-complete line of source code (the trailing `)` is missing):

```
    news = ReadPythonNews(GetItemCount()
```

Press enter a few times. Note that Wing IDE auto-indents the subsequent lines and adds red error indicators under them shortly after you stop typing. This indicates that there is a syntax error in your code:



```
#----------------------------------------------------------------------
def PrintAsText(news):
  for date, event, url in news:
    print "%s -- %s (%s)" % (date, event, url)

#----------------------------------------------------------------------
def PrintAsHTML(news):
  for date, event, url in news:
    # NOTE: The line below contains a deliberate typo
    print '<p><i>%s</i> <a href="%s">%s</a></p>' % (data, url, event)


##########################################################################
# Enter code according to the tutorial here:

mews = ReadPythonNews(GetItemCount()
```

Once you correct the line and complete it by typing the final ), the error indicators will be removed. You should now have this complete line of code in your file:

```
news = ReadPythonNews(GetItemCount())
```

The Source Assistant also updates as you move your insertion caret around the editor. For example, try moving onto `GetItemCount`. Also notice that the blue links in the Source Assistant can be used to jump to the points of definition of each symbol listed there. For variables, the link after `Symbol:` goes to the point of definition of that variable, while any links after `Likely Type:` go to the point of definition of that data type (these are the same if the symbol is a function, method, or class; we'll try the Source Assistant with more interesting code later).

To play around with these tools a bit more, enter the following two additional lines of code:

```
PrintAsText(news)
PrintAsHTML(news)
```

At this point you have a complete program that can be run in the debugger. There are many other editor features worth learning, but we'll get back to those later in this tutorial.

## A Note on Proxies

If you are behind a firewall and use a web proxy, you may need to alter the tutorial before it will work. In particular, set up a proxy mapping like this:

```
proxies = {'http': 'http://192.168.3.7:3128'}
```

And then add a second parameter to the urllib call that obtains the news in `ReadPythonNews` so it looks like this:

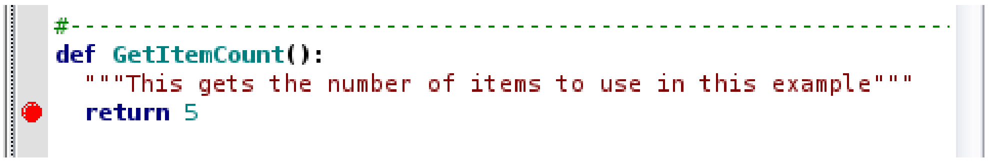
svc = urllib.urlopen("http://www.python.org/channews.dat", proxies=proxies)

You will of course need to set the http proxy url according to your local network's configuration. See also how to determine proxy settings.

## 1.7. Tutorial: Debugging

In case you haven't already figured it out, the `example1.py` program you have created connects to `python.org` via HTTP, reads and parses the Python-related news feed that is hosted there, and then prints the most recent five items as text and HTML. Don't worry if you don't have an internet connection on your machine; the script has canned data it will use when it cannot connect to `python.org`.

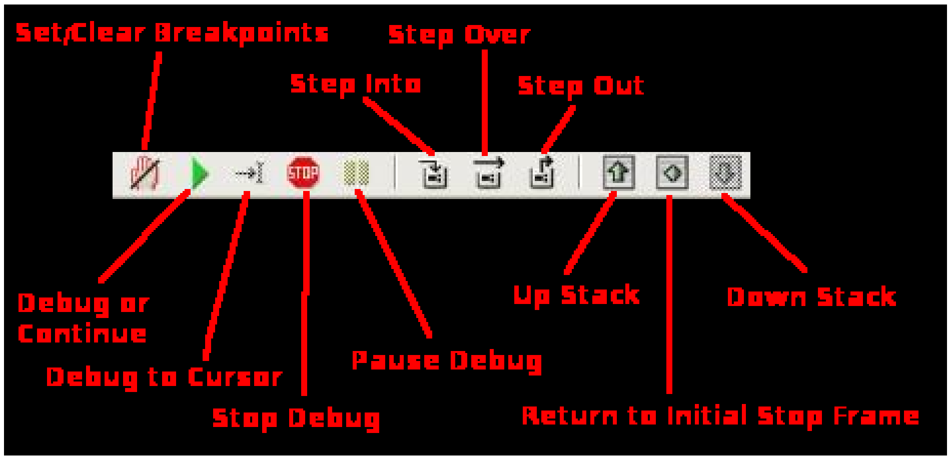
To start debugging, set a breakpoint on the line that reads `return 5` in the `GetItemCount` function. This can be done by clicking on the line and selecting the `Break` toolbar item, or by clicking on the dark margin to the left of the line. The breakpoint should appear as a filled red circle:



```
#------------------------------------------------------------
def GetItemCount():
  """This gets the number of items to use in this example"""
  return 5
```

Next start the debugger with the green arrow icon in the toolbar or the `Start/Continue` item in the `Debug` menu. Wing will show the Debug Properties dialog with the properties that will be used during the debug run. Just ignore this for now, uncheck the `Show this dialog before each debug run` checkbox at the bottom, and press `OK`.

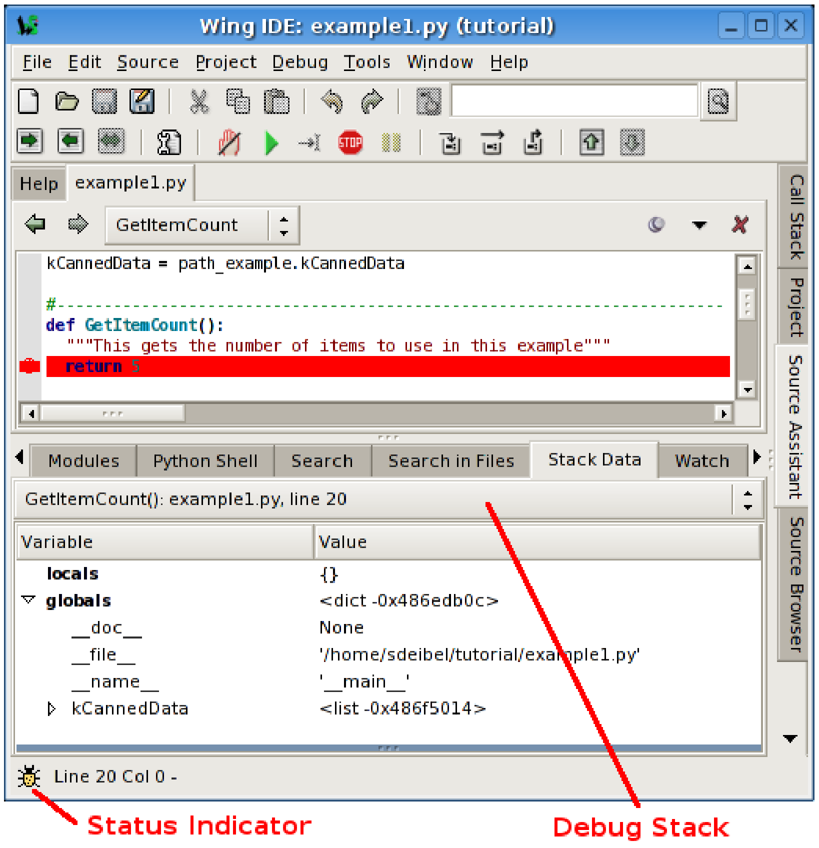
Wing will run to the breakpoint and stop, placing a red indicator on the line. Notice that the toolbar changes to include additional debug tools, as shown below:



Your display may vary depending on how you have configured the **Toolbar Size** and **Toolbar Style** preferences. Note that Wing displays tooltips explaining what the tools do when you mouse over them.

Now you can inspect the program state at that point with the **Stack Data** tool and by going up and down the stack from the toolbar or `Debug` menu. The stack can also be viewed as a list using the **Call Stack** tool.

Notice that the Debug status indicator in the lower left of Wing's main window changes color depending on the state of the debug process. Mouse over the indicator to see detailed status in a tooltip:



```
kCannedData = path_example.kCannedData

#------------------------------------------------------------------------
def GetItemCount():
  """This gets the number of items to use in this example"""
  return 5
```

GetItemCount(): example1.py, line 20

```
Variable            Value
  locals            {}
▽ globals           <dict -0x486edb0c>
    __doc__         None
    __file__        '/home/sdeibel/tutorial/example1.py'
    __name__        '__main__'
  ▷ kCannedData     <list -0x486f5014>
```

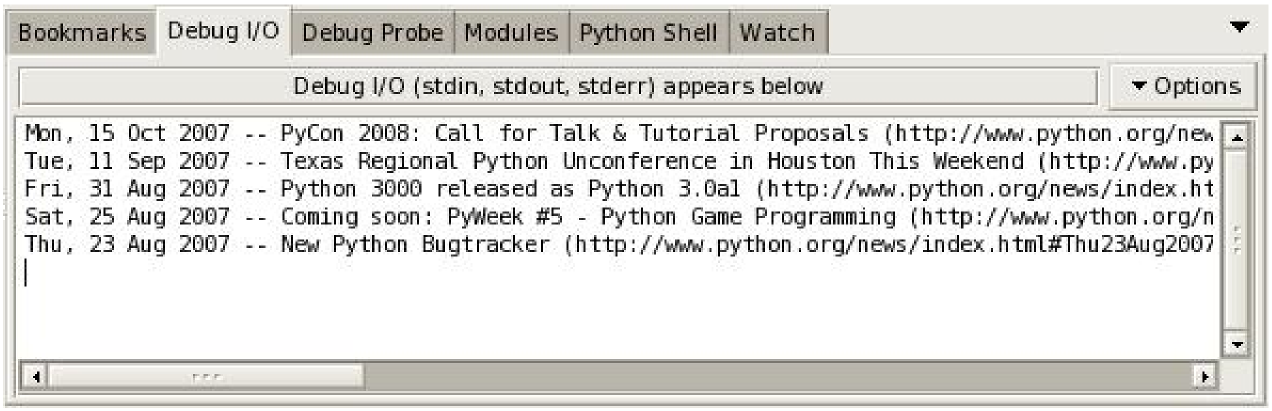
Line 20 Col 0 -

Next, try stepping out to the enclosing call to `ReadPythonNews`. In this particular context, you can achieve this in a single click with the `Step Out` toolbar icon or `Debug` menu item (two clicks on `Step Over` also work). This is a good function to step through in order to familiarize yourself with the basic debugger features covered above.

## 1.7.1. Tutorial: Debug I/O

Before continuing any further in the debugger, bring up the **Debug I/O** tool so you can watch the subsequent output from the program. This is also where keyboard input takes place in debug code that requests for it.

Once you step over the line `PrintAsText(news)` you should see output appear as follows:



```
Mon, 15 Oct 2007 -- PyCon 2008: Call for Talk & Tutorial Proposals (http://www.python.org/new
Tue, 11 Sep 2007 -- Texas Regional Python Unconference in Houston This Weekend (http://www.py
Fri, 31 Aug 2007 -- Python 3000 released as Python 3.0a1 (http://www.python.org/news/index.ht
Sat, 25 Aug 2007 -- Coming soon: PyWeek #5 - Python Game Programming (http://www.python.org/n
Thu, 23 Aug 2007 -- New Python Bugtracker (http://www.python.org/news/index.html#Thu23Aug2007
```

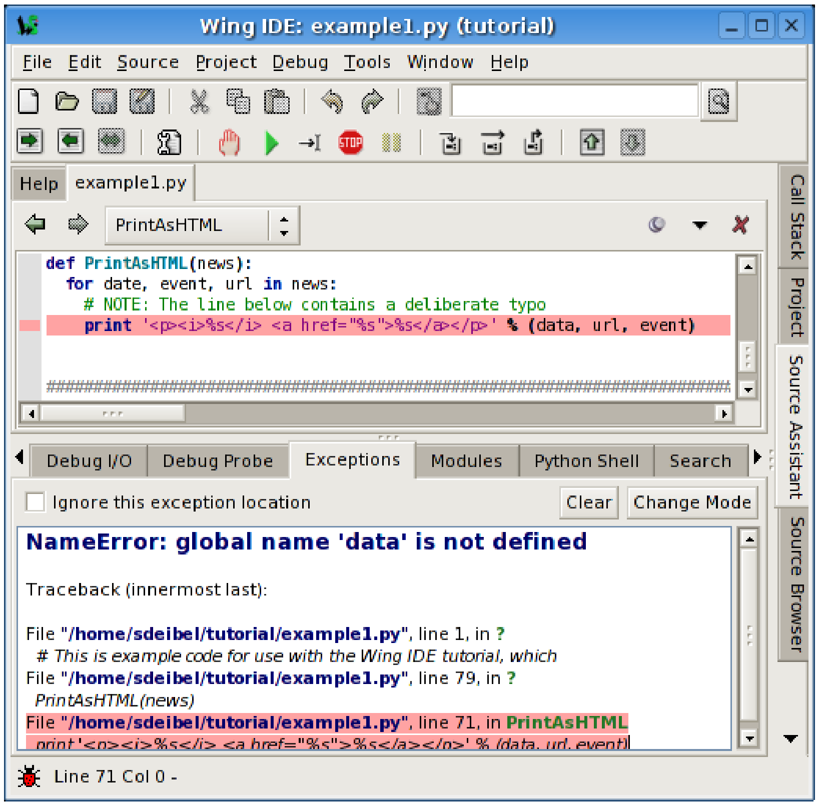
Note that you can also configure Wing to use an external console from the `Options` menu in the Debug I/O tool. This is useful for code that depends on details of the Debug I/O environment (such as cursor control with special output characters).

## 1.7.2. Tutorial: Debug Process Exception Reporting

Wing's debugger reports any exceptions that would be printed when running the code outside of the debugger.

Try this out by continuing execution of the debug process with the `Debug` toolbar item or `Start / Continue` item in the `Debug` menu.

Wing will stop on an incorrect line of code in `PrintAsHTML` and will report the error in the **Exceptions** tool:



```
def PrintAsHTML(news):
  for date, event, url in news:
    # NOTE: The line below contains a deliberate typo
    print '<p><i>%s</i> <a href="%s">%s</a></p>' % (data, url, event)
```

```
NameError: global name 'data' is not defined

Traceback (innermost last):

File "/home/sdeibel/tutorial/example1.py", line 1, in ?
  # This is example code for use with the Wing IDE tutorial, which
File "/home/sdeibel/tutorial/example1.py", line 79, in ?
  PrintAsHTML(news)
File "/home/sdeibel/tutorial/example1.py", line 71, in PrintAsHTML
  print '<p><i>%s</i> <a href="%s">%s</a></p>' % (data, url, event)
```

```
Line 71 Col 0 -
```

Notice that this tool highlights the current stack frame and that you can click on frames to navigate the exception backtrace. Whenever you are stopped on an exception, the Debugger Status indicator in the lower left of Wing's main window turns red.

### Advanced Options

Wing's debugger provides several exception handling modes, which differ in how they determine which exceptions should be reported. It is also possible to add specific exception types to always report or never report. This is described in more detail in **Managing Exceptions**. Most users will not need to alter these options, but being aware of them is potentially useful in advanced debugging scenarios.

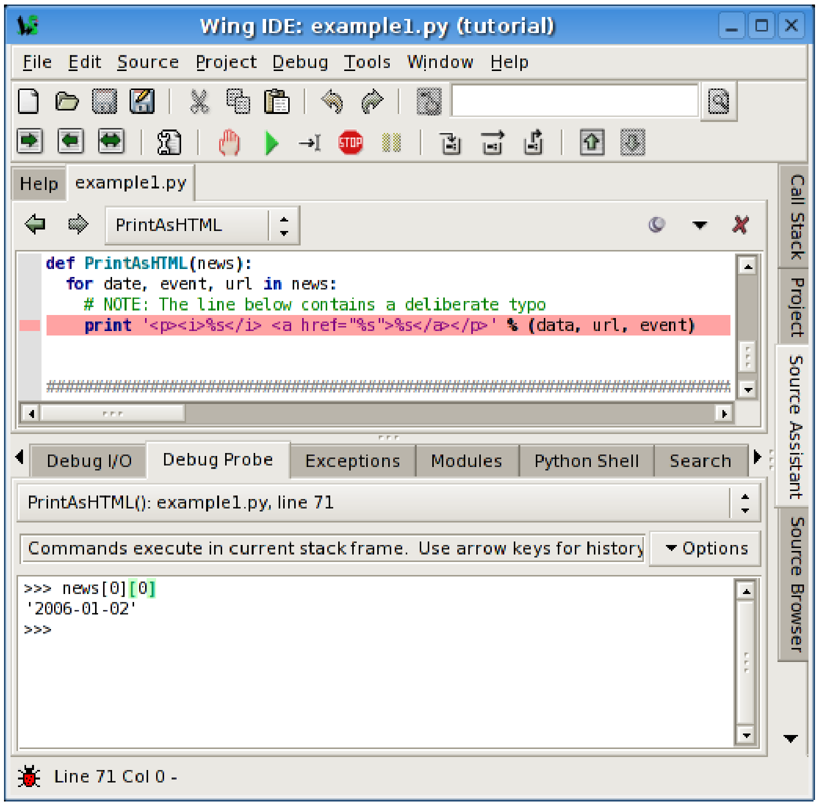
## 1.7.3. Tutorial: Command Line Power Debugging

Wing IDE Professional's **Debug Probe** provides a powerful way to find and fix complex bugs. This works much like the Python Shell but lets you interact directly with your paused debug program, in the context of the current stack frame:

Try it out from the point of exception reached earlier by typing this:

```
news[0][0]
```

This will print the date of the first news item:



```
def PrintAsHTML(news):
  for date, event, url in news:
    # NOTE: The line below contains a deliberate typo
    print '<p><i>%s</i> <a href="%s">%s</a></p>' % (data, url, event)
```

PrintAsHTML(): example1.py, line 71

Commands execute in current stack frame. Use arrow keys for history

```
>>> news[0][0]
'2006-01-02'
>>>
```

Line 71 Col 0 -

Next, try this:

```
news[0][0] = '2004-06-15'
```

This is one way to change program state while debugging, which can sometimes be useful when testing out code that will go into a bug fix. Try this now:

```
PrintAsText(news)
```

This executes the function call and prints its output to the Debug Probe. Note that the Debug I/O tool is not used for input or output whenever it results from commands typed in the Debug Probe. All Debug I/O is temporarily redirected here.

Note that Wing offers auto-completion as you type and shows call signature and documentation information in the **Source Assistant**.

Here is another possibility. Copy/paste or drag and drop this block of code to the Debug Probe:

```
def PrintAsHTML(news):
  for date, event, url in news:
    print('<p><i>%s</i> <a href="%s">%s</a></p>' % (date, url, event))
```

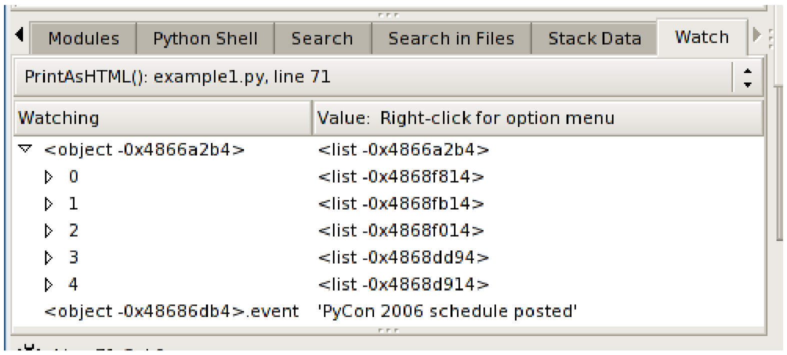
This actually replaces the buggy definition of `PrintAsHTML` that's in the `example1.py` source file, so that you can now execute it without errors as follows:

```
PrintAsHTML(news)
```

This can be useful in designing bug fixes when the fix depends on lots of program state, or happens in a context that is hard or time-consuming to reproduce in the debugger: Quick interactive trial and error replaces multiple edit/debug cycles.

### 1.7.4. Tutorial: Watching Debug Data

Another useful feature when working through complex bugs is Wing's ability to watch debug data values in a variety of ways. This is done with the **Watch** tool:



PrintAsHTML(): example1.py, line 71

| Watching | Value: Right-click for option menu |
|---|---|
| <object -0x4866a2b4> | <list -0x4866a2b4> |
| 0 | <list -0x4868f814> |
| 1 | <list -0x4868fb14> |
| 2 | <list -0x4868f014> |
| 3 | <list -0x4868dd94> |
| 4 | <list -0x4868d914> |
| <object -0x48686db4>.event | 'PyCon 2006 schedule posted' |

## Watching Values

While still at the same exception in `PrintAsHTML`, right-click on the `locals` value `news` in the **Stack Data** tool. This will present you with the following options for watching the value over time:

**Watch by Symbolic Path** -- This causes Wing to look for the symbolic name `news` in the current stack frame whenever you are debugging. When you select this item, the Watch panel will be displayed with one item in it that reads:

```
    news                <list 0x40401eec>
```

(the object id will of course vary)

This is useful for quick access to values without digging through a long locals or globals list in the Stack Data view.

Since the watch makes sense across debug sessions, it will be remembered in the Watch tool until you clear it.

**Watch by Direct Reference** -- This causes Wing to keep a reference to this particular object instance (a list). It will be shown in the Watch tool as long as it exists. If the reference count for the object instance goes to zero, Wing will report `<value not found>`.

This is useful for watching a particular object while stepping through portions of code that may not hold a reference to it, or from which it is difficult to reach the referenced instance data.

Since object references aren't meaningful across debug sessions, these entries will be removed from the Watch tool as soon as the debug process terminates.

**Watch by Parent Slot** -- This combines the above two modes by keeping a reference to the parent of the selected value and looking up the sub-part of the value by symbolic name.

If you try this on `event` in `locals`, you are watching the value `event` within the particular locals dictionary, rather than `event` in the current stack frame.

This technique is more useful when working in object-oriented code where it can be used to watch particular attributes within a specific object instance.

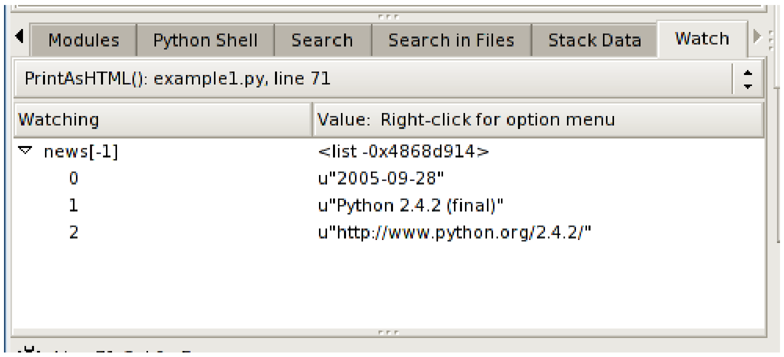
Since the parent is tracked by object reference, these entries are also removed from the Watch tool as soon as the debug process ends its life.

**Watch by Module Slot** -- This option can be used to watch values within modules, by looking up the module by name in `sys.modules` and tracking the value symbolically. It is only available when right-clicking on values in the Modules tool, which will be discussed later.

Since these are meaningful across debug sessions, values watched by module will be retained in the Watch tool until they are removed by the user.

### Watching Expressions

It is also possible to watch the value of any Python expression in the Watch panel. Just click on an empty part of the `Watching` column and type in the expression you wish to watch:


Modules
Python Shell
Search
Search in Files
Stack Data
Watch
PrintAsHTML(): example1.py, line 71
Watching
Value: Right-click for option menu

```
news[-1]      <list -0x4868d914>
    0         u"2005-09-28"
    1         u"Python 2.4.2 (final)"
    2         u"http://www.python.org/2.4.2/"
```

Try this now, while still stopped at the exception in `PrintAsHTML`, by entering this:

```
news[-1]
```

This will show the last item of `news` as long as there is one, or `<undefined>` or `<error evaluating>` if the value cannot be determined.

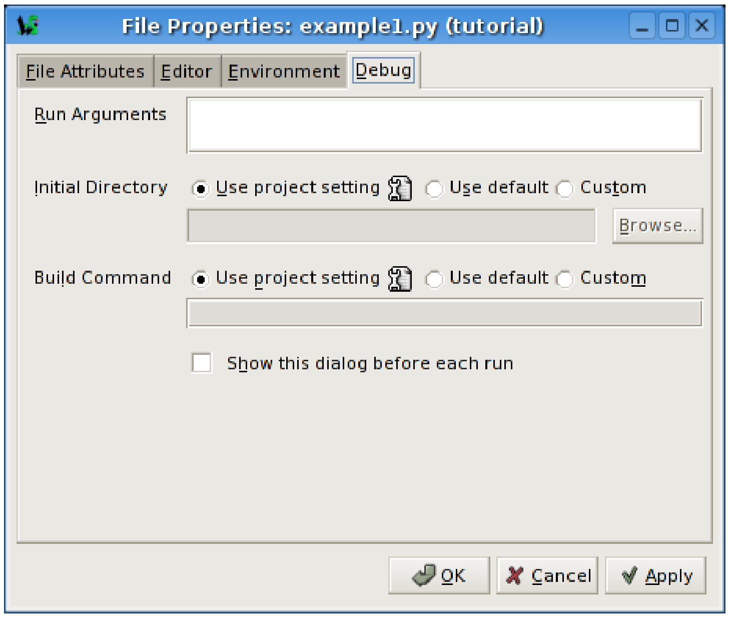
Expressions are remembered in the Watch tool across debug sessions, until they are removed by the user.

## 1.7.5. Tutorial: Other Debugger Features

Before moving on to the rest of the IDE's features, here are a few highlights of the debugger's other capabilities that are worth knowing about from the start:

- **Main Debug File** -- You can specify one file in your project as the main entry point for debugging. When this is set, debugging will always start there unless you use the `Debug Current File` item in the `Debug` menu. To set a main debug file use `Set Current as Main Debug File` in the `Debug` menu, right click on the `Project` tool and select `Set as Main Debug File`, or use the `Main Debug File` property in the `Debug` tab of Project Properties. Whether or not you set a main debug file depends on the nature of your project.

- **File Properties** -- Each file in your project can override or modify your project-wide debug properties. This is useful in projects with multiple debug entry points. File properties can also be used to specify command line arguments for debugging. They are accessed from the `Current File Properties` item in the `Source` menu or by using `Properties` in the editor or project (right-click) context menus:


File Properties: example1.py (tutorial)

File Attributes | Editor | Environment | Debug

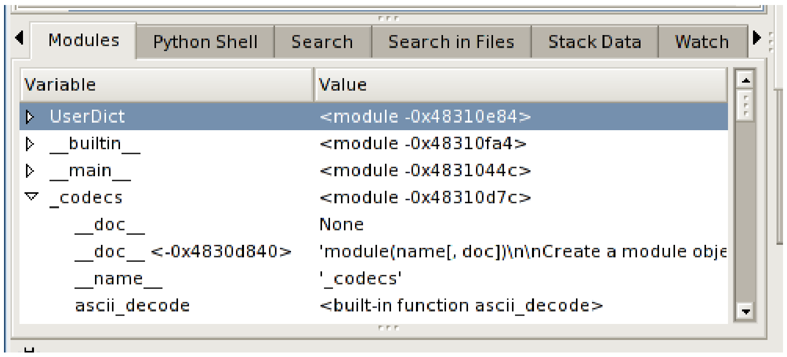
Run Arguments:

Initial Directory: [● Use project setting] ○ Use default ○ Custom

Build Command: [● Use project setting] ○ Use default ○ Custom

☐ Show this dialog before each run

- **Modules Data View** -- By default, Wing filters out modules and some other data types from the values shown in the Stack Data tool. In some cases, it is useful to view values stored in modules. This can be done with the **Modules** tool, which is simply a list of all modules found in `sys.modules`:



| Variable | Value |
|---|---|
| UserDict | <module -0x48310e84> |
| __builtin__ | <module -0x48310fa4> |
| __main__ | <module -0x4831044c> |
| _codecs | <module -0x48310d7c> |
| __doc__ | None |
| __doc__ <-0x4830d840> | 'module(name[, doc])\n\nCreate a module obje |
| __name__ | '_codecs' |
| ascii_decode | <built-in function ascii_decode> |

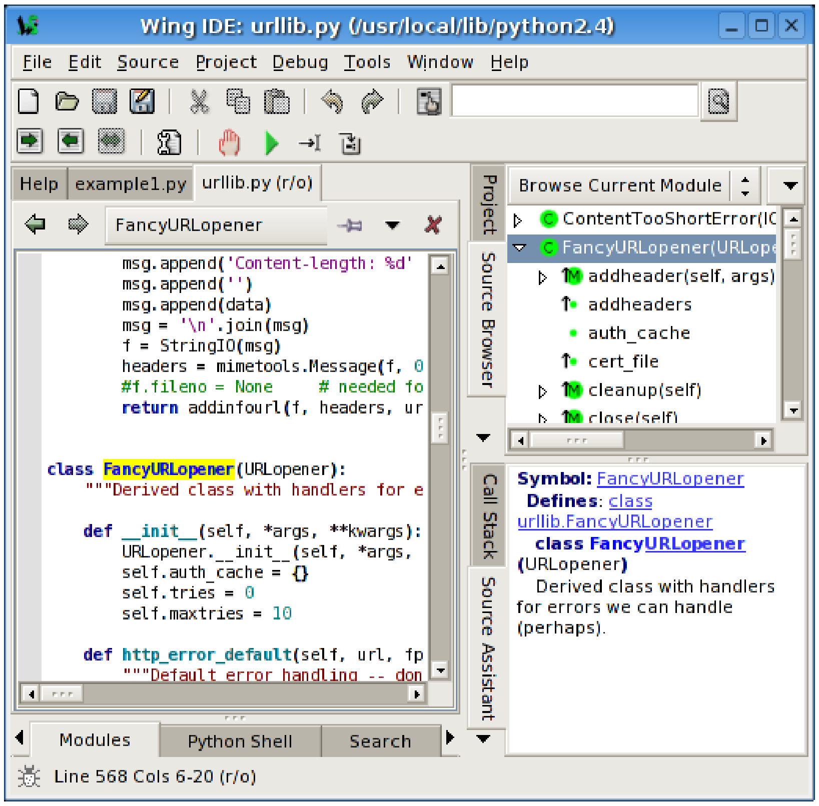
- **Conditional Breakpoints** -- Use the `Debug` menu's `Breakpoint Options` group or right click on the breakpoints margin to set a conditional breakpoint. These can be very useful if you need to stop in code before an error occurs, so that you can step through the code that leads up to the error. Conditionals can be any Python expression, but beware of expressions that alter your program state as a side-effect. Note that Wing will always stop on a conditional breakpoint when an exception is raised by the conditional expression itself.

- **Breakpoint Manager** -- The `Breakpoints` tool accessed from the `Tools` menu shows a list of all defined breakpoints and allows enabling/disabling, editing the breakpoint conditional, setting an ignore count, and inspecting the number of times a breakpoint has been reached when a debug process is active.

- **Remote Debugging** -- Wing can debug processes that are running under a web server or web development framework, or that get launched from the command line and not from Wing. This is beyond the scope of this tutorial, and is described in **Debugging Externally Launched Code** and in the the relevant **How-To guides**.

## 1.8. Tutorial: Source Browser

Wing IDE Professional includes a **Source Browser** that can be used to inspect and navigate the module and class structure of your source code.



```
        msg.append('Content-length: %d'
        msg.append('')
        msg.append(data)
        msg = '\n'.join(msg)
        f = StringIO(msg)
        headers = mimetools.Message(f, 0
        #f.fileno = None     # needed fo
        return addinfourl(f, headers, ur


class FancyURLopener(URLopener):
    """Derived class with handlers for e

    def __init__(self, *args, **kwargs):
        URLopener.__init__(self, *args,
        self.auth_cache = {}
        self.tries = 0
        self.maxtries = 10

    def http_error_default(self, url, fp
        """Default error handling -- don
```

**Symbol:** FancyURLopener
**Defines**: class urllib.FancyURLopener
**class FancyURLopener (**URLopener**)**
Derived class with handlers for errors we can handle (perhaps).

Line 568 Cols 6-20 (r/o)

By default, the browser will display classes, methods, attributes, functions, and variables defined in the currently displayed source editor (if any). The popup menu at the top left of the source browser can be used to alter the display to include all classes or all modules in the project. The Options menu in the top right allows filtering by origin, accessibility, and type of source symbols. The Options menu also allows sorting the view alphabetically, by type, or in the order that symbols occur in the source file.

As with the Project display, double clicking or right-clicking on items in the Source Browser opens them into an editor. The `Follow Selection` option appears here as well (in the Options menu) and when enabled opens transient editors in order to show the points of definition for symbols selected on the Source Browser by single-clicking or via keyboard navigation.

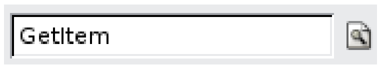
The **Source Assistant** is integrated with the source browser, and will update its content as you move around the source browser tree.

## 1.9. Tutorial: Searching

Wing IDE provides several different interfaces for searching your code. Which you use depends on your task.

### Toolbar Search

A quick way to search through the current editor is to enter your search string in the area provided in the toolbar:



If you enter only lower case the search will be case-insensitive. Entering one or more upper-case letter causes the search to become case-sensitive.

Try this now in `example1.py`: Type `GetItem` in the toolbar search area and Wing will immediately, starting with the first letter typed, search for matching text in the editor. Notice that if you press the `Enter` key, Wing will move on to the next match, wrapping around to the top of the file if necessary.

Toolbar-based searches always go forward (downward) in the file from the current cursor position.

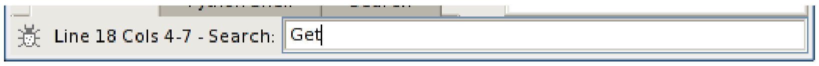
### Keyboard-driven Search

If you prefer to search without your fingers leaving the keyboard, use the key bindings given next to the `Mini-search` items in the `Edit` menu (the bindinges vary by keyboard personality).

From here, you can initiate searching forward and backward in the current editor, optionally using the current selection in the editor as the search string. You can also initiate replace operations.

Try this in the `example1.py` file: If using the default editor mode, press the `Ctrl-U`. If you are using emacs mode, press `Ctrl-S`. For others, refer to the `Mini-search` group in the `Edit` menu.

This will display an entry area at the bottom of the IDE window and will place focus there:


Line 18 Cols 4-7 - Search: Get

Continue by typing `G`, then `e`, then `t`. Notice how Wing searches incrementally with each keypress. This lets you type only as much as you need to find the source code you are looking for.

While the mini-search area is still active, try pressing the same key combination you used to display it again (`Ctrl-U` or `Ctrl-S` in emacs mode) and Wing will search for the next matching occurrence. Note that if no match is found `Failed Search` will be displayed. However, pressing the mini search key combination again will wrap around and start searching again at the top of the file.

As in the toolbar search, typing lower case letters results in case-insensitive search, and using one or more upper case letters results in case-sensitive search.

Search direction can be changed during searching by pressing the key bindings assigned to forward and backward mini-search. You can exit from the search by pressing the `Esc` key or `Ctrl-G` in emacs mode.

The regular expression based search options found in the `Mini-search` menu group work similarly but expect regular expressions for the search criteria (see below).

Keyboard-driven mini-replace also works similarly, except that you will be presented with two entry areas, one for your search string and one for the replace string. Use `Query/Replace` to be prompted for `Y` and `N` for each replace location, and `Replace String` to replace all matches globally in the file.

## Search Tool

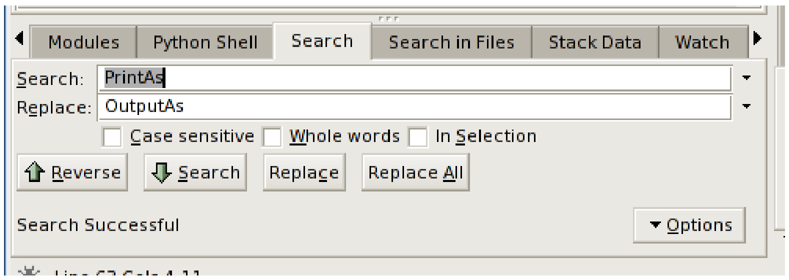
The `Search` tool provides a familiar GUI-based search and replace tool for operating on the current editor. Key bindings for operations on this tool are given in the `Search and Replace` group in the `Edit` menu.

Searches may span the whole file or be constrained to the current selection, can be case sensitive or insensitive, and may optionally be constrained to matching only whole words.

By default, searching is incremental while you type your search string. To disable this, uncheck `Incremental` in the `Options` menu.

## Replacing

When the Show Replace item in Options is activated, Wing will show an area for entering a replace string and adds Replace and Replace All buttons to the Search tool:



Search: PrintAs
Replace: OutputAs
Search Successful

Try replacing example1.py with search string PrintAs and replace string OutputAs.

Select the first result match and then Replace repeatedly. One search match will be replaced at a time. Search will occur again after each replace automatically unless you turn off the Find After Replace option. Changes can be undone in the editor, one at a time. Do this now to avoid saving this replace operation.

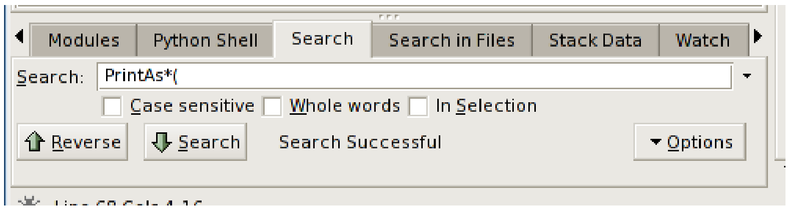
Next, try Replace All instead. Wing will simply replace all occurrences in the file at the same time. When this is done, a single undo in the editor will cancel the entire replace operation.

## Wildcard Searching

By default, Wing searches for straight text matches on the strings you type. Wildcard and regular expression searching are also available in the Options menu.

The easier one of these to learn is wildcard searching, which allows you to specify a search string that contains *, ?, or ranges of characters specified within [ and ]. This is the same syntax supported by the Python glob module and is described in more detail in the **Wildcard Search Syntax** manual page.

Try a wildcard search now by selecting Wild Card from the Options menu and making sure example1.py is your current editor. Set the search string to PrintAs*(. This should display find matches, all occurrences of the string PrintAs, followed by zero or more characters, followed by (:



PrintAs*(

Search Successful

Also try searching on `PrintAs*[A-Z](` with the `Case Sensitive` search option turned on. This matches all strings starting with `PrintAs` followed by zero or more characters, followed by any capital letter, followed by `(`.

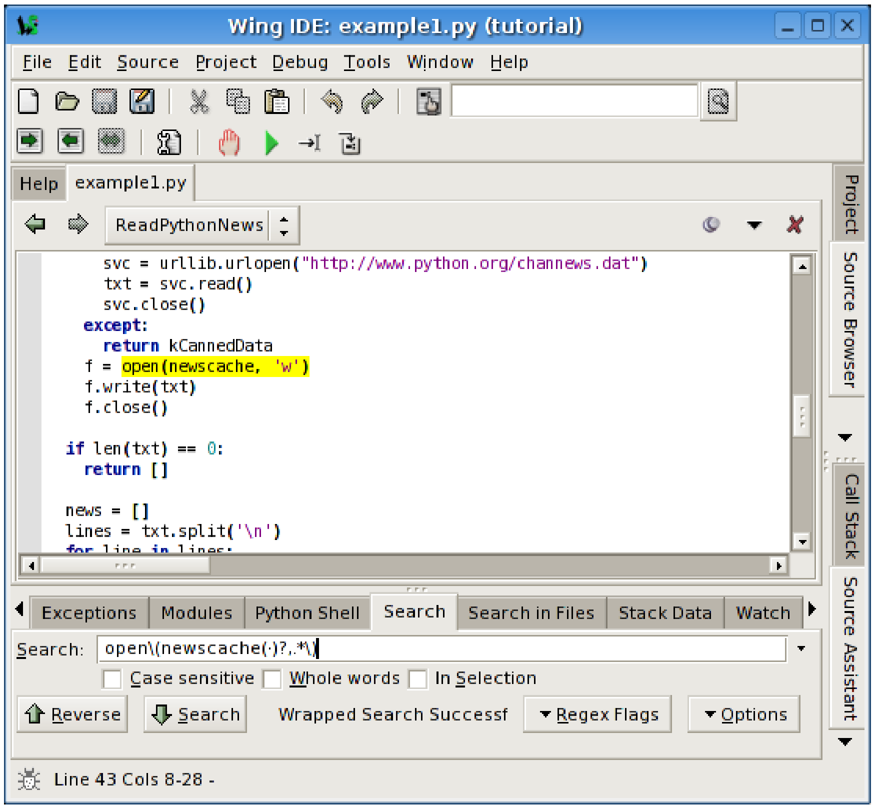
Finally, try `PrintAsT???`, which will match any string starting with `PrintAsT` followed by any three characters (`?` matches any single character).

Wild card searching can be very useful for finding related source symbols all at once.

## Regular Expression Search

Regular expressions can also be used for searching. These are most useful for complicated search tasks, such as finding all calls to a particular function that occur as part of an assignment statement.

For example, `open\(newscache( )?,.*\)` matches only calls to the function `open` where the first argument is named `newscache` and there are at least two parameters. If you try this with `example1.py`, you should get exactly one search match:



```
    svc = urllib.urlopen("http://www.python.org/channews.dat")
    txt = svc.read()
    svc.close()
  except:
    return kCannedData
  f = open(newscache, 'w')
  f.write(txt)
  f.close()

if len(txt) == 0:
  return []

news = []
lines = txt.split('\n')
for line in lines:
```

Search: open\(newscache(·)?,.*\)

Case sensitive: [ ] Whole words: [ ] In Selection: [ ]

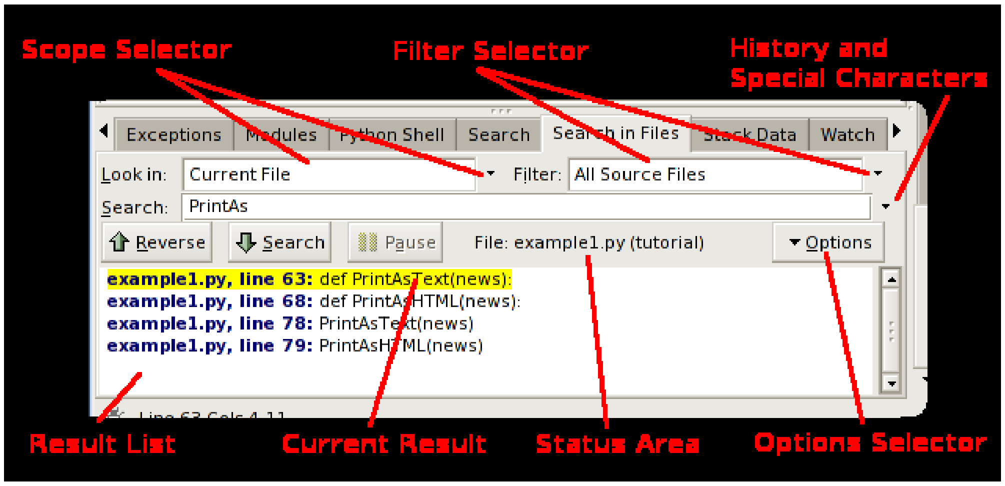
Wrapped Search Successf

Line 43 Cols 8-28 -

The details of regular expression syntax and usage can be very complicated, so this tutorial does not cover them. For that, see the Regular Expression Syntax documentation in the Python manual.

## Search in Files Tool

The **Search in Files** tool is the most powerful search option available in Wing IDE. It supports multi-file batch search of the disk, project, open editors, or other sets of files. It can also search using wildcards and can do regular expression based search/replace.



Look in: Current File
Filter: All Source Files
Search: PrintAs
File: example1.py (tutorial)

```
example1.py, line 63: def PrintAsText(news):
example1.py, line 68: def PrintAsHTML(news):
example1.py, line 78: PrintAsText(news)
example1.py, line 79: PrintAsHTML(news)
```

Line 63 Cols 4-11

Before worrying about the details, try a simple batch search on the `example1.py` file. Select `Current File` from the `Look in` selector on the search manager (these are the defaults). Then enter `PrintAs` into the search area.

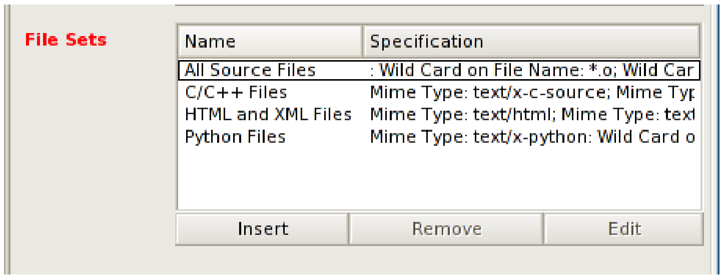
Wing will start searching immediately, restarting the search whenever you alter the search string or make other changes that affect the result set. When you are done, you should see results like those shown in the screen shot above. Click on the first result line to select it. This will also display `example1.py` with the corresponding search match highlighted.

You can use the forward/backward arrows in the Search in Files manager to traverse your results.

## File Sets

Next, change the `Look in` selector to `All Files in Project` and change your search string to `HTML`. This works the same way as searching a single file, but lists the results for all files in your project. You can also search all currently open files in this way.

In many cases, searching is better constrained to a subset of files in your projects. For example, only Python files. This can be done with by selecting `Python Files` in the `Filter` selector. You can also define other file sets using the `Create/Edit File Sets...` item in the Filter Selector. This will display the File Sets preference:



| Name | Specification |
|---|---|
| All Source Files | : Wild Card on File Name: *.o; Wild Car |
| C/C++ Files | Mime Type: text/x-c-source; Mime Typ |
| HTML and XML Files | Mime Type: text/html; Mime Type: text |
| Python Files | Mime Type: text/x-python: Wild Card o |

Each file set has a name and a list of include and exclude specifications. Each of these specifications can be applied to the file name, directory name, or the file's MIME type. A simple example would be to specify `*.pas` wildcard for matching Pascal files by name, or using the `text/html` mime type for all HTML files.

## Searching Disk

Wing can also search directly on disk. Try this by typing a directory path in the `Look in` area. Assuming you haven't changed the search string, this should search for `HTML` in all text files in that directory.

Disk search can also be recursive, in which case Wing searches all sub-directories as well. This is done by selecting a directory in the `Look in` scope selector and checking `Recursive Directory Search` in the `Options` menu.

You can alter the format of the result list with the `Show Line Numbers` item and `Result File Name` group in the Options Selector. The Option Selector contains various other search options as well.

Note that searching `Project Files` is usually faster than searching a directory structure because the set of files is precomputed.

## Multi-File Replace

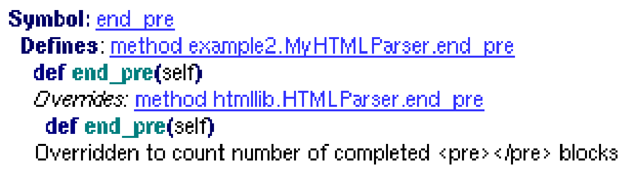
When working with multiple files in the result set, Wing will by default open each changed file into an editor, whether or not it is already open. This allows you to undo changes by not saving files or by issuing `Undo` within each editor.

An alternate replace mode is also available from the Options menu. If you check the `Replace Operates on Disk` item, Wing will change files directly on disk instead of opening editors into the IDE. This can be much faster but is not recommended unless you have a revision control system that can get you out of hot water when mistakes are made.

Note that even when operating directly on disk, Wing will replace changes in already-open editors only within the IDE. This avoids creating two versions of a file if there are already edits in the IDE's copy. We recommend closing all editors when working in `Replace Operates on Disk` mode, or select `Save All` from the file menu immediately after each replace operation. This avoids losing parts of a replace, which might lead to inconsistent application of the replace operation to the files in your source base.

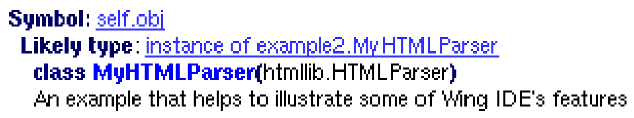
## 1.10. Tutorial: Source Assistant with Classes

The earlier examples of the Source Assistant in action within `example1.py` didn't show some of its features because there are no classes in that file. Let's revisit it now with `example2.py` from your `tutorial` directory. Move the insertion cursor to the definition of the `end_pre` method in `MyHTMLParser` and place it on the word `end_pre`. You should see the following in the Source Assistant:



**Symbol:** end_pre
**Defines:** method example2.MyHTMLParser.end_pre
**def end_pre**(self)
*Overrides:* method htmllib.HTMLParser.end_pre
**def end_pre**(self)
Overridden to count number of completed <pre></pre> blocks

The Source Assistant also displays information about inherited classes when clicking on class names. For example, clicking on `self.obj` in the constructor (`__init__()`) of `AnotherClass` will display this:



**Symbol:** self.obj
**Likely type:** instance of example2.MyHTMLParser
**class MyHTMLParser**(htmllib.HTMLParser)
An example that helps to illustrate some of Wing IDE's features

**Helping Wing's Analyzer**: Notice the statement that reads `isinstance(obj, MyHTMLParser)` at the top of `AnotherClass.__init__()` in `example2.py`: This tells Wing's source analysis engine the type of `obj`. Python's design makes exhaustive analysis of object-oriented code difficult, but since type information is propagated by inference to other values (such as `self.obj` in this case), fairly few `isinstance` hints can go a long way to improving Wing's ability to show useful information in the auto-completer, Source Assistant, and other tools.

Since Wing's analysis engine ignores conditionals in code, the following can be used in a case where the `isinstance` would add a circular import bug to program execution:

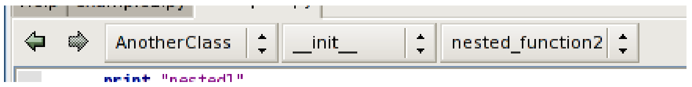
```
if 0:
  import mymodule
  isinstance(myvalue, mymodule.MyClass)
```

## 1.11. Tutorial: Other IDE Features

There are a number of other features available in the IDE that are worth noting:

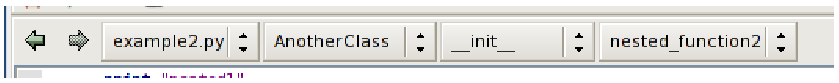
**Source Index** -- The top of the editor area displays a series of popup menus that act as an index into Python source files. Select from them to navigate around your source file.

Try this out by opening `example2.py` from your `tutorial` directory. If you place the cursor on the line that reads `print("nested2")`, you should see the following in the source index area:



Each subsequent menu lists the symbols available within the preceding nested context.

If you have turned off `Show Notebook Tabs` in the Editor Options Menu, the file selector menu will be prepended as follows:



**Goto-Definition** -- There are a number of ways to navigate to the point of definition of symbols in your source code. One is to right-click on the symbol and select `Goto Definition`. Another is to move the insertion cursor to the symbol and select `Goto Selected Symbol Defn` from the `Source Menu` (or press F4). The Source Assistant in Wing IDE Pro also contains links to points of definition.

Try this from `example2.py` with some of the symbols imported from htmllib, such as `HTMLParser` in the class definition for `MyHTMLParser`. Remember that the file `htmllib.py` is opened in non-sticky mode and will auto-close unless you toggle the stick pin icon to

or edit the file.

Duplicate substitution definition name: “stickpin-stuck”.

**Goto-Line** -- Navigate quickly to a numbered source line with the `Goto Line` item in the `Edit` menu. In emacs mode, the line number is typed into the data entry area that appears at the bottom of the window. Press `Enter` to complete the action.

**Keyboard-driven File Open** -- Try the `Open from Keyboard` item in the File menu: This displays an interactive file selector at the bottom of the IDE window that can be much quicker for opening files than using the standard file selection dialog and allows file selection without moving your hands from the keyboard. Use `Esc` to cancel or `Enter` to select a file and the arrow keys to browse around the auto-completion list that it presents as you type.

**Auto-Indentation** -- Wing auto-indents lines as you type according to its static analysis of your code. This can be disabled with the **Auto-Indent** preference.

Another way in which Wing uses code analysis is in auto-indentation as you type, and for altering indentation or wrapping of code. For example, when you select a block of code and press the tab key, the entire block is re-indented according to the correct position of its first line relative to the preceding non-blank line of code. The Justify Text option in the Source menu also uses the source analyser to constrain re-wrapping to a single logical line of Python code.

**Block Indentation** -- The `Tab` key is defined to indent the current line or blocks of lines, rather then entering a tab character (which can be done with `Ctrl-Tab`). The **Tab Key Action** preference can be used to customize how the tab key behaves.

One or more selected lines can be increased or reduced in indentation from the Indentation toolbar group, which contains the following icons for this purpose:

Single lines or whole blocks can also be indented automatically to their appropriate position, as determined by analysis of the preceding line. If a range of lines is selected, the whole block is indented or outdented without changing the relative indents within the block. This is done from the following toolbar icon:

Note that the indentation features are also available in the `Source` menu, where their key bindings are listed.

**Block Commenting** -- Units of code can be commented out or un-commented quickly from the `Source` menu.

**Brace Matching** -- Wing highlights brace matching as you type unless disabled from the **Auto Brace Match** preference. The `Match Braces` item in the `Source` menu causes Wing to select all the code that is contained in the nearest matching braces found from the current insertion point on the editor. Repeated application of the command will traverse outward and forward in the file.

**Text Reformatting** -- Code can be re-wrapped with the `Justify Text` item in the `Source` menu. This will limit wrapping to a single logical line of code, so it can be used for wrapping an argument list or long list or tuple without altering surrounding code.

**Converting Indentation Styles** -- Wing's **Indentation** tool can be used to analyze and convert the style of indentation found in source files. See **Indentation Manager** for details.

**Revision Control** -- Wing provides integrations with the `Subversion`, `Mercurial`, `Bazaar`, `Git`, `CVS`, and `Perforce` revision control systems. These auto-enable based on the contents of your project. See the **Version Control** documentation for details.

**Unit Testing** -- Wing's **Testing** tool makes it easy to run and debug units tests.

**OS Commands** -- The **OS Commands** tool can be used to set up, execute, and interact with external commands, for building, deployment, and other tasks. The `Build Command` field in Project Properties can be used to configure and select one command to execute automatically before any debug session begins.

**Code Snippets** -- The `Snippets` tool in the `Tools` menu can be used to define and use code snippets for commonly repeated motifs, such as class or def skeletons or documentation templates. For details see the **snippets documentation**.

**Bookmarks** -- The `Bookmarks` tool in the `Tools` menu and bookmarking commands in the `Source` menu can be used to define and jump to marked locations in the editor. In

Python files, these bookmarks are defined relative to the named scope in the file so they move around as the file is edited. See **bookmark documentation** for details.

**Folding** -- Unless turned off with the **Enable Folding** preference, Wing allows folding of editor code to hide areas that are not currently of interest. The folding is visual only so selecting across a folding and copying will copy the text including its hidden portions. Folding can be useful to get a quick summary of the contents of a source file. Refer to the **Folding** manual page for details.

**Macros** -- Keyboard/command macros are available. See the **Keyboard Macros** section of the manual for details.

## 1.12. Tutorial: Further Reading

Congratulations! You've finished the tutorial. As you work with Wing IDE on your own software development project, the following resources may be useful:

- Wing IDE Support Website
- **Wing IDE Reference Manual**
- **OS X Quickstart**
- **How-To Guides**