

# Introduction for New Users

## Wing IDE Personal

Wingware  
[www.wingware.com](http://www.wingware.com)

Version 5.0.0  
November 14, 2013

Thanks for trying [Wing IDE Personal!](#)

- For a guided tour **try the tutorial**.
- To try Wing on your own, see the **quick start guide**.
- For hints on using Wing with specific GUI toolkits, web development frameworks, rendering and compositing systems, and other Python frameworks and toolkits, see also the **How-Tos**

## Contents

### Wing IDE Tutorial

- 1.1. Tutorial: Getting Started
- 1.2. Tutorial: Getting Around Wing IDE
  - Context Menus
  - Configuring the Keyboard
  - Auto-Completion
  - Other Configuration Options
- 1.3. Tutorial: Check your Python Integration
- 1.4. Tutorial: Set Up a Project
  - Opening Files
  - Transient, Sticky, and Locked Files
- 1.5. Tutorial: Setting Python Path
  - Python Path Hints
- 1.6. Tutorial: Introduction to the Editor
- 1.7. Tutorial: Navigating Code
- 1.8. Tutorial: Debugging
  - 1.8.1. Tutorial: Debug I/O
  - 1.8.2. Tutorial: Debug Process Exception Reporting
    - Advanced Options
  - 1.8.3. Tutorial: Execution Environment
  - 1.8.4. Tutorial: Remote Debugging
- 1.9. Tutorial: Indentation Features

1.10. Tutorial: Other Editor Features

1.11. Tutorial: Searching

Toolbar Search

Keyboard-driven Search

Search Tool

Replacing

Wildcard Searching

Regular Expression Search

Search in Files Tool

File Filters

Searching Disk

Multi-File Replace

1.12. Tutorial: Further Reading

*Copyright (c) 1999-2013 by Wingware. All rights reserved.:*

Wingware

P.O. Box 400527

Cambridge, MA 02140-0006

United States of America



# Wing IDE Tutorial

This tutorial 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: 

When using this tutorial with products other than Wing IDE Professional, please note that the screen shots include tools and features only available in Wing IDE Professional. These can safely be ignored and, when working with the tutorial within Wing IDE's help viewer, those tools will not be discussed in the content that follows.

## 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](http://python.org). This tutorial will work with Python version 2.5 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 and restart Wing IDE.

On Linux/Unix, you can also add a browser command line to your **URL Display Com-**

**mands** 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, which will prompt you for the directory into which to copy the tutorial: **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](mailto:support@wingware.com).

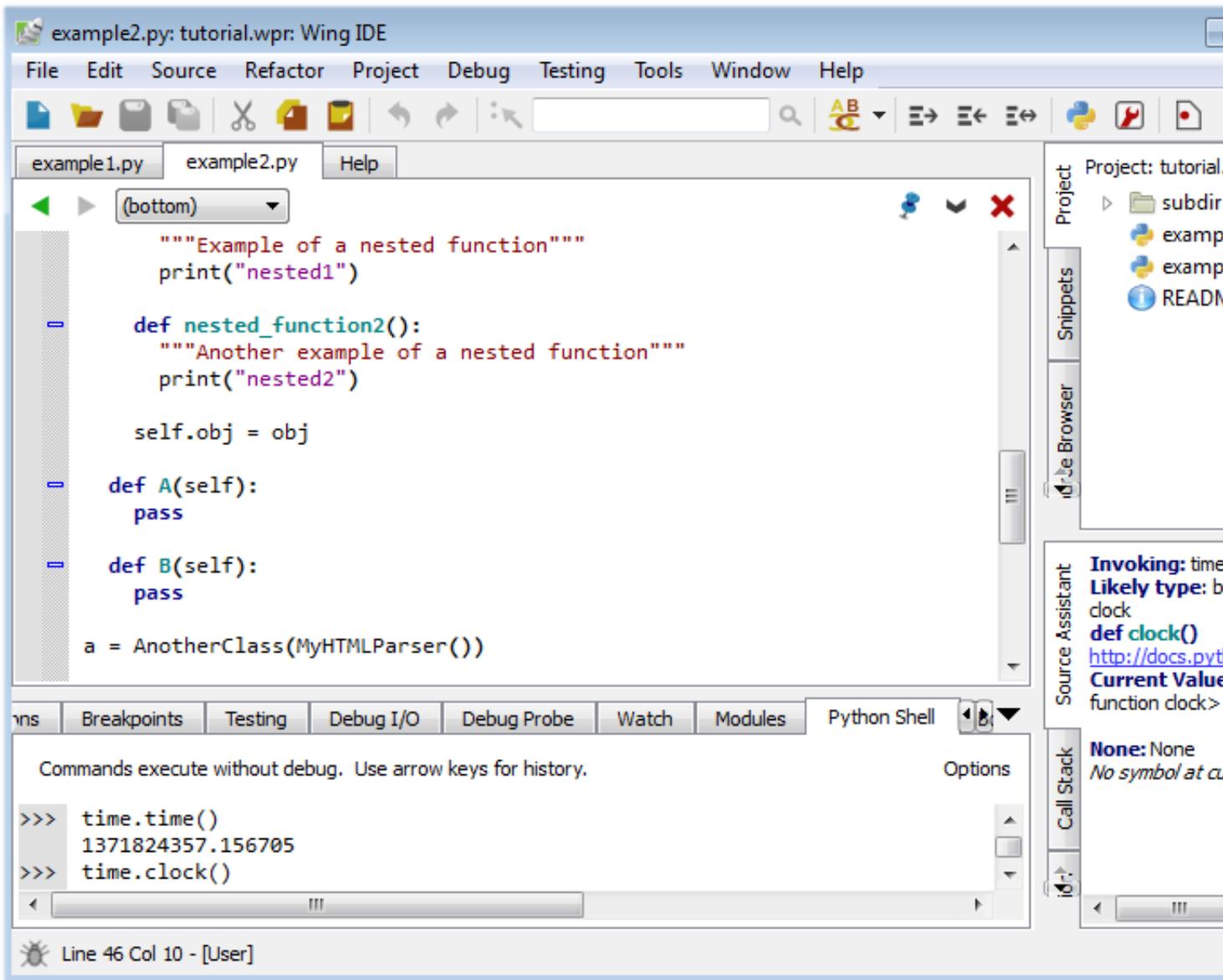
To get to the next page in the tutorial, use the `Next Page` icon shown in the toolbar just above this text:



## 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 toolboxes separated by draggable dividers. Try pressing `F1` and `F2` now to show or hide the two toolboxes. Also try `Shift-F2` to maximize the editor area temporarily, hiding both tool areas and toolbar until `Shift-F2` is pressed again.



Tool and editor tabs can be dragged to rearrange the user interface, optionally creating a new split. Right click on the tabs for a menu of additional options, such as adding or removing splits or to move the toolbox from right to left. The number of splits shown by default in toolboxes will vary according to the size of your monitor.

Notice that you can click on an already-active tool tab to minimize that tool area. Click again on any tab to restore the toolbox to its previous size.

By default, the editor shows all open files in all splits, making it easy to work on different parts of a file simultaneously. This can be changed by unchecking **Show All Files in All Splits** in the right-click context menu on the editor tabs.

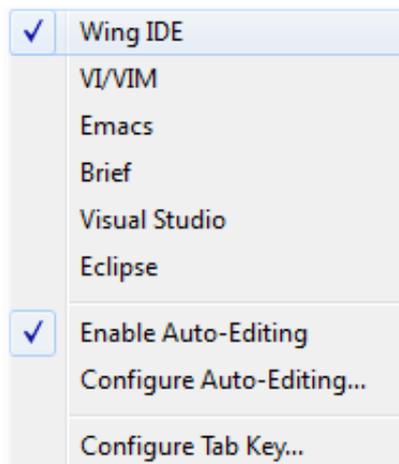
! Splitting your editor area makes it easier to get around this tutorial. To do this now, right click on the editor tab area and select **Split Side by Side**. On small monitors and laptops, it may be preferable to create a new window for the tutorial by right clicking on its tab and selected **Move Wing IDE Help to New Window**.

## Context Menus

In general, right-clicking provides a menu for interacting with or configuring a part of the user interface. The text that follows refers to these menus as “context menus”.

## Configuring the Keyboard

Use the **User Interface > Keyboard > Keyboard Personality** sub-menu in the **Edit** menu or **Keyboard > Personality** preference to tell Wing to emulate another editor, such as Visual Studio, VI/Vim, Emacs, Eclipse, or Brief.



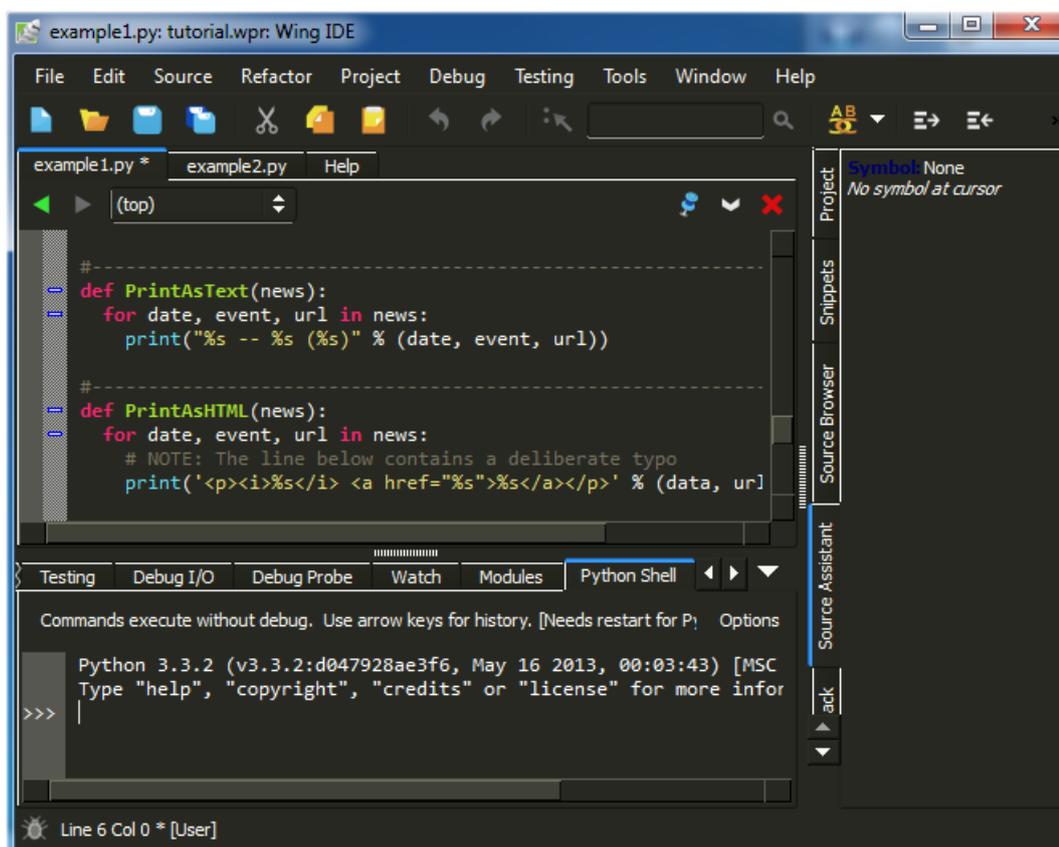
The **Configure Tab Key** item in the **Keyboard Personality** sub-menu or the **User Interface > Keyboard > Tab Key Action** preference can be used to select among available behaviors for the tab key. The default is to match the selected Keyboard Personality. When the Keyboard Personality is set to Wing IDE, the tab key acts differently according to context. For example, if lines are selected, repeated presses of the tab key moves the lines among syntactically valid indent positions. And, when the caret is at the end of a line, pressing the tab key adds one indent level.

## Auto-Completion

There are many options for Wing's auto-completer. These are set in the **Editor > Auto-completion** preferences group. For example, if you are used to using the **Enter** key for completion, you may wish to add that now to the **Editor > Auto-completion > Completion Keys** preference.

## Other Configuration Options

Wing's cross-platform GUI adjusts to the OS on which you are running it. This can be overridden with the **User Interface > Display Style** preference. For example, to set a dark background display style select **Match Palette** and set the **User Interface > Color Palette** preference to either **Black Background** or **Monokai**:



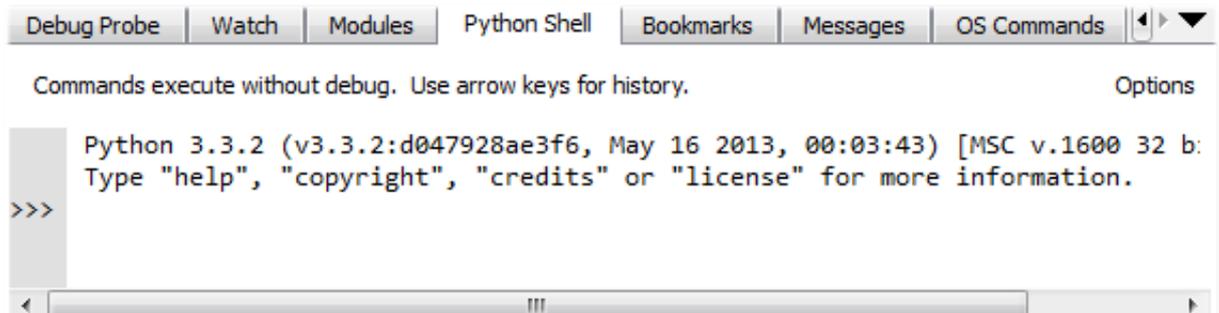
The **User Interface > Fonts > Display Font/Size** and **User Interface > Fonts > Editor Font/Size** preferences select fonts for the user interface and editor.

The size and type of tools used in the toolbar at the top of Wing IDE's main window can be changed by right clicking on one of the enabled tools.

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. Bring up the **Python Shell** tool now from the **Tools** menu. If all goes well, it should start up Python and show you the Python command prompt like this:



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('*' * i)
```

This should print a triangle as follows:

```

>>> for i in range(0, 10):
...     print('*' * 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**. Use the **Tab** key to enter a selected completion. Other keys can be set up as completing keys using the **Editor > Auto-completion > Completion Keys** preference.

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.

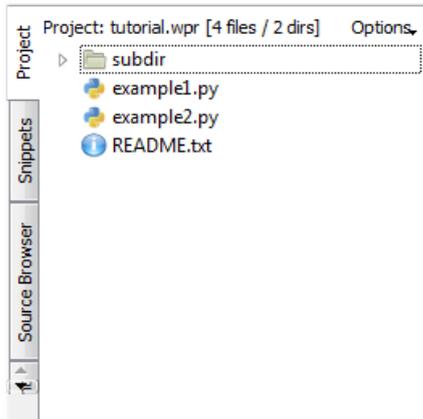
## 1.4. Tutorial: Set Up a Project

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.

If you haven't already copied the `tutorials` directory from your Wing IDE installation, please do so now as described in **Tutorial: Getting Started**.

Wing starts up initially with the Default Project. Start by creating a new project for your work on this tutorial, using **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.

Next, use the **Add Existing Directory** item in the **Project** menu to add your copy of the `tutorials` directory. Leave the default options checked so that all files in that directory are added to the project.

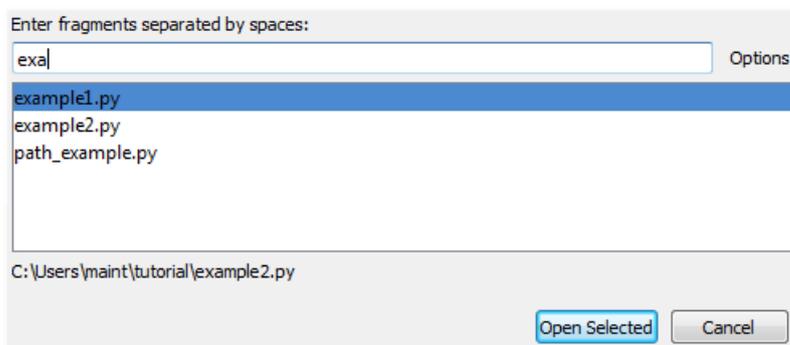


To make it easier to work on source code and read this tutorial at the same time, you may want to right-click on the editor tab area and select **Split Side by Side**.

## Opening Files

Files in your project can be opened by double-clicking in the **Project** tool, by typing fragments into the **Open From Project** dialog, and in other ways that will be described later.

Try the **Open From Project** dialog now by using the key binding listed for it in the **File** menu. Type `ex` as the file name fragment and use the arrow keys and then **Enter** to open the file `example1.py`. Now try it again with the fragment `sub ex`. This matches only files with both `sub` and `ex` in their full path names. In larger projects, **Open From Project** is usually the easiest way to open a file.



## Transient, Sticky, and Locked Files

Wing opens files in one of several modes in order to keep more relevant files open, while auto-closing others. To see this in action, right-click on `os` in `import os` at the top of `example1.py` and select **Goto Definition**. The file `os.py` will be opened non-sticky, so that it is automatically closed when hidden.

The mode in which a file is opened is indicated with an icon in the top right of the editor area:



- The file is sticky and will be kept open until it is closed by the user.

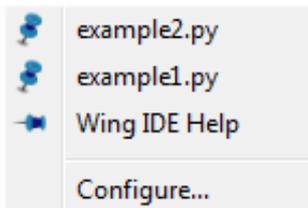


- The file is non-sticky and will be kept open only while visible. When a non-sticky file is edited, it immediately converts to sticky.



- The file is locked in the editor, so that the editor will not be reused to display other newly opened files. This mode is only available when multiple editor splits are present.

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



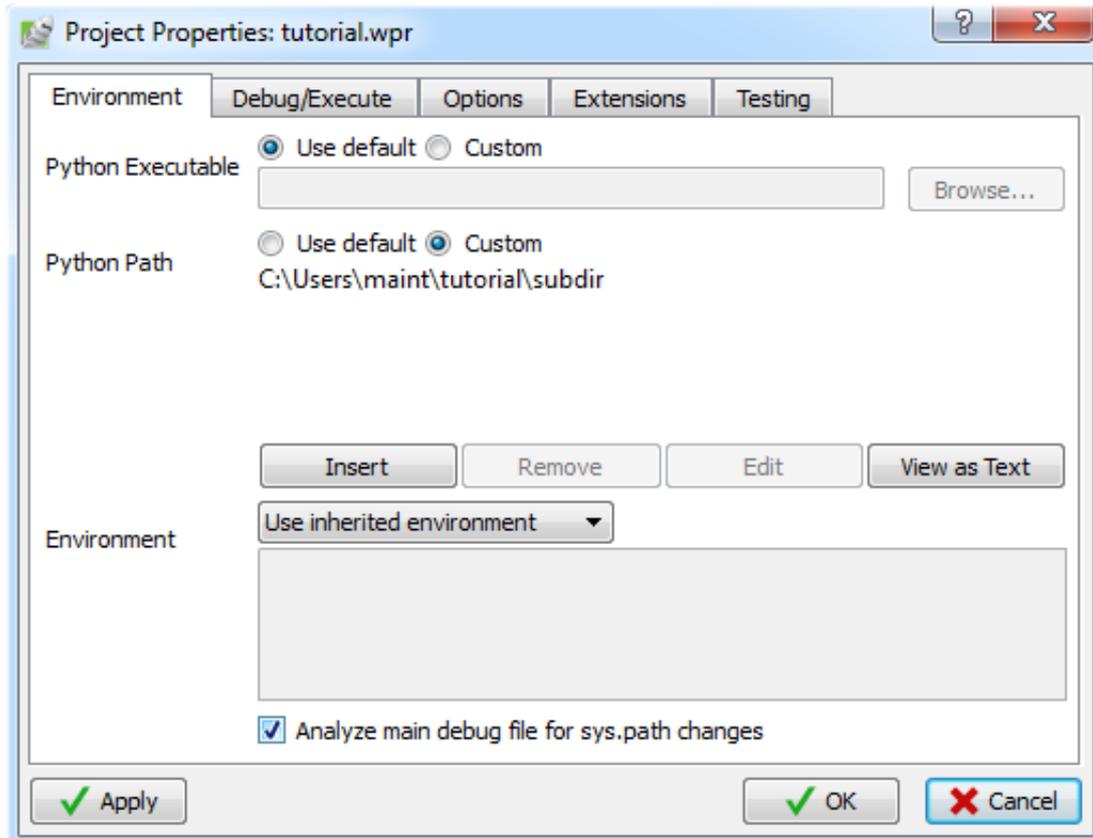
The number of non-sticky editors to keep open, in addition to those that are visible, is set with the **Editor > Advanced > Maximum Non-Sticky Editors** preference.

This mechanism is also used in multi-file searches and other features that navigate through many files. In general you can ignore the modes and Wing will keep open the files you are actually working on, while auto-closing those that you have only visited briefly.

## 1.5. Tutorial: Setting Python Path

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

This value can be entered in **Python Path** in the **Project Properties** dialog, which is accessible from the **Project** menu and the toolbar:



For this tutorial, you need to add the `subdir` sub-directory of your `tutorials` directory to **Python Path**, as shown above. This directory contains a module used as part of the first coding example.

Note that the full path to the directory `subdir` is used. This is strongly recommended because it avoids potential problems finding source code when the starting directory is ambiguous or changes over time. If relative paths are needed to make a project work on different machines, use an environment variable like `${WING:PROJECT_DIR}/subdir`. This is described in more detail in **Environment Variable Expansion**.

The configuration used here is for illustrative purposes only. You could run the example

code without altering `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, which is in the default `PYTHONPATH`.

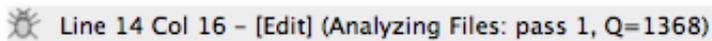
## Python Path Hints

If your main entry point is Python code that alters `sys.path`, and the file is set as the **Main Entry Point** in **Project Properties** then Wing can often determine the correct `PYTHONPATH` to use without any changes to **Python Path** in **Project Properties**.

When in doubt, compare value of `sys.path` at runtime in your code with the value reported by **Show Python Environment** in the **Source** menu.

## 1.6. Tutorial: Introduction to the Editor

Now that you have set up your project, Wing will have found and analyzed the tutorial examples, and all the modules that are imported and used by them. This analysis process runs in the background and is used for auto-completion, call tips, and other features. With larger code bases, you may notice the CPU load from this process, and Wing will indicate that processing is active by displaying **Analyzing Files** in the status area at the bottom left of the main IDE window:



☼ Line 14 Col 16 - [Edit] (Analyzing Files: pass 1, Q=1368)

However, with this tutorial analysis will have happened instantaneously after the project was configured.

### Editing with Wing IDE

Let's start by trying out a subset of Wing's editor features, focusing on the auto-completer, Source Assistant, and some of Wing's auto-editing operations.

Open the file `example1.py` from the **Project** tool. Then bring up the **Source Assistant** tool the **Tools** menu or by clicking on its tab. This is where Wing IDE shows documentation, call signature, and other information as you move around in your source code or work with other tools.

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

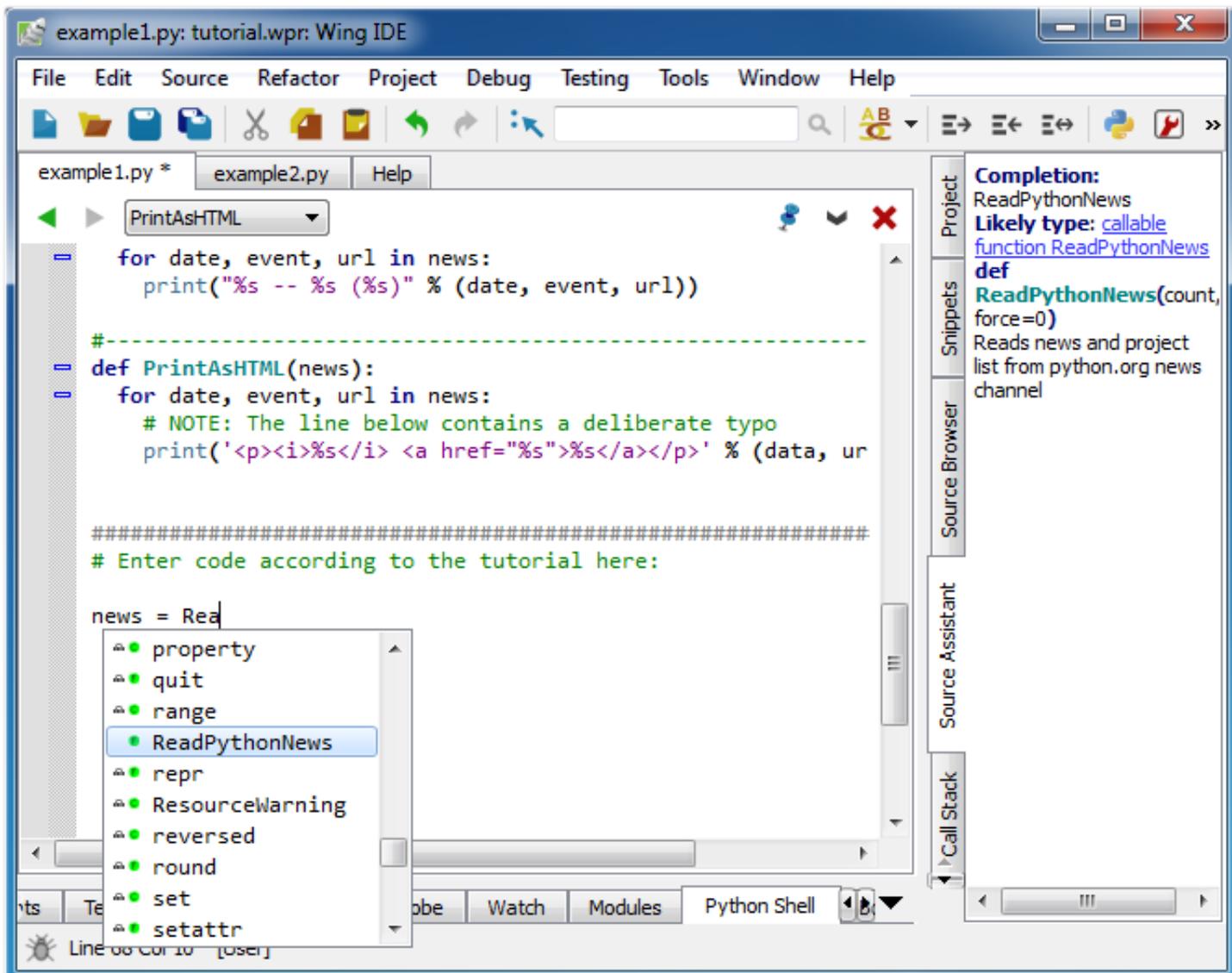
```
news = Rea
```

Wing displays a context-sensitive auto-completer as you type. You can scroll around in the list with the arrow keys, type **Esc** or **Ctrl-G** to abort completion, or **Tab** to enter the currently selected completion.

! If you are used to using the **Enter** key for auto-completion, add it to the **Editor > Auto-Completion > Completion Keys** preference now.

Duplicate substitution definition name: “note”.

When you first 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 displays 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:



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

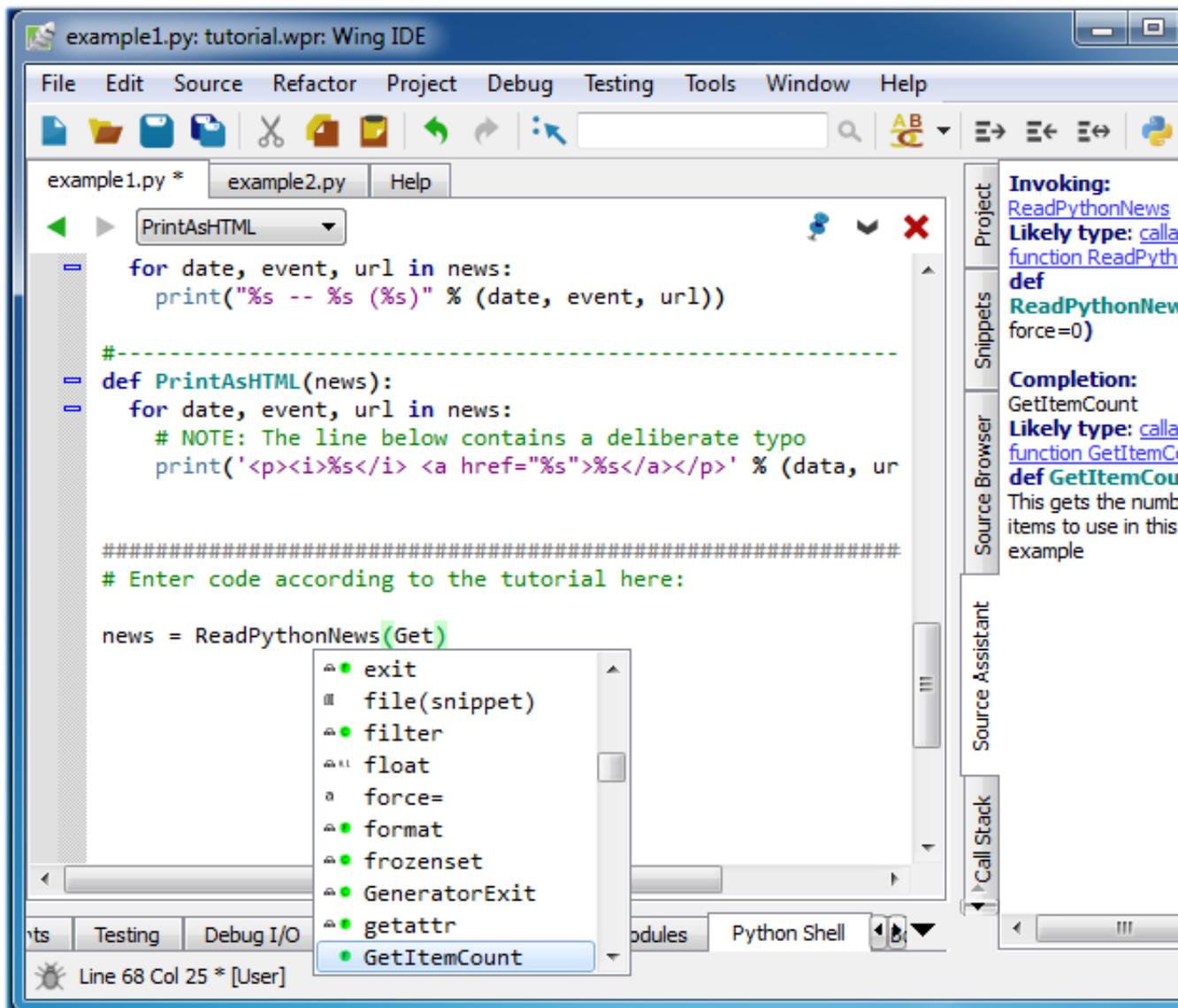
```
news = ReadPythonNews(
```

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

Duplicate substitution definition name: "note".

Now type `Get` to start entering arguments for your invocation of `ReadPythonNews`. You

will see the Source Assistant alter its display to highlight the first argument in the call signature for `ReadPythonNews` and add information on the argument's completion value:



The docstring for `ReadPythonNews` is temporarily hidden to conserve screen space. This behavior can be toggled with the `Show docstring during completion` option in the Source Assistant's context menu.

Now continue entering the rest of the source line so you have the following complete line of source code:

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

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

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

At this point you have a complete program that can be run in the debugger. Don't try it yet, however. It contains some deliberate bugs and first we should take a look at some of Wing's code navigation features.

## 1.7. Tutorial: Navigating Code

As already noted, the **Source Assistant** updates as you move your insertion caret around the editor, or when browsing through the auto-completer. For example, try moving between the invocation of `PrintAsText` and the variable `news` in the code you just typed. The blue links in the **Source Assistant** can be used to jump to the points of definition of each symbol listed there.

If you click on one of the links in the **Source Assistant**, use the green back arrow at the top left of the editor to return from the value or type definition:



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, but they differ for variables. For example, for `news` the point of definition is the line where it is first assigned a value and the type is a Python list.

### Python Documentation

For built-ins and code in the Python standard library, Wing tries to add links into the Python documentation. For example, type `open` in the editor and try out the <http://docs.python.org> link. The documentation will be opened in your default web browser.

**Symbol:** [open](#)  
**Likely type:** [builtin function open](#)  
**def open**(file, mode='r', buffering=\_1, encoding=None, errors=None, newline=None, closefd=True, opener=None)  
<http://docs.python.org/py3k/library/functions.html#open>  
 open(file, mode='r', buffering=-1, encoding=None, errors=None, newline=None, closefd=True, opener=None) -> file object  
 Open file and return a stream. Raise IOError upon failure.

Now use **Undo** or press the **delete** key to remove **open** from your code.

## Goto-Definition

A quicker way to visit the point of definition of a symbol is to click on it and press **F4** or right click and use one of the **Goto Definition** context menu items. Again, you can use the history back/forward arrows at the top left of the editor to return from the point of definition.

Try this for **ParseRDFNews** in `example1.py`. Wing will open up the file `path_example.py` and show the point of definition of **ParseRDFNews**. Notice that the file 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-unstuck”.

Duplicate substitution definition name: “stickpin-stuck”.

## Source Index

Wing maintains a set of source index menus at the top of the editor area. The menus are updated as you move around code, and additional levels of menus are added as needed, based on context. Try these now to navigate to **CHandler** in `path_example.py`, and then use the second menu to navigate to **endElement**.

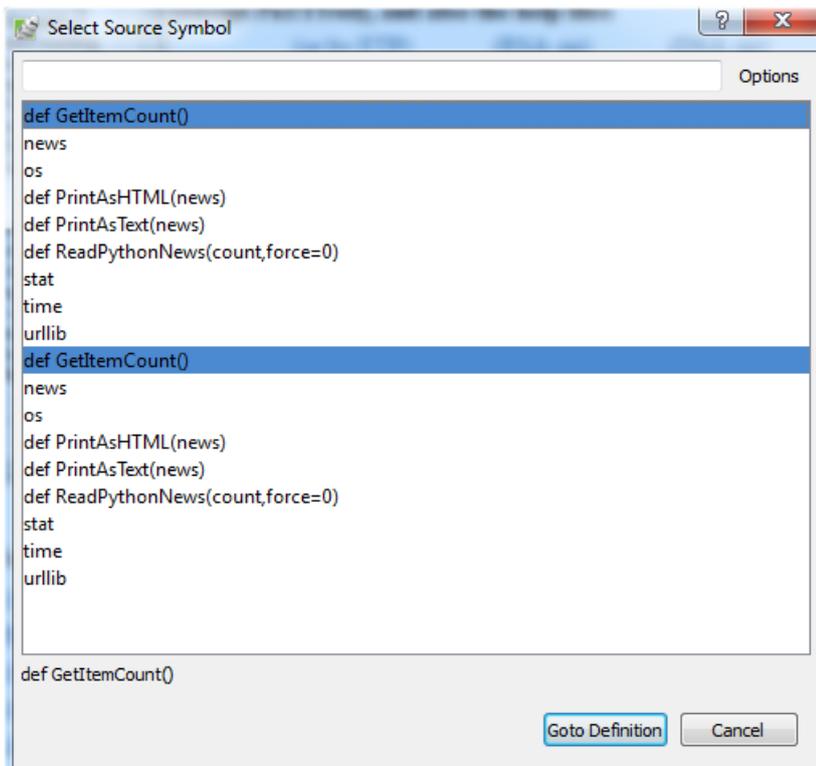


Now use the history back arrow at top left of the editor area to return to the invocation of **ParseRDFNews** in `example1.py`. You will need to press the arrow several times to move back through your visit history.

## Find Symbol

If you are looking for a symbol defined in the current scope, use **Find Symbol** in the

Source menu. This displays a dialog where you can type a fragment matching the symbol name. Use the arrow keys to traverse the matches and press **Enter** to visit the symbol's point of definition.



There are many other editor features worth learning, but we'll get back to those later in this tutorial, after we try out the debugger.

## 1.8. Tutorial: Debugging

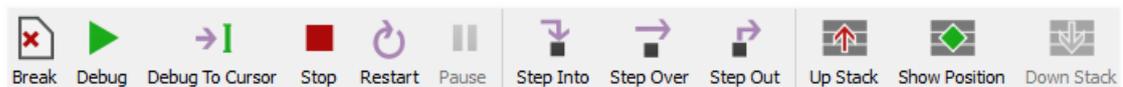
The `example1.py` program you have just created connects to `python.org` via HTTP, reads and parses the Python-related news feed in RDF format, and then prints the most recent five items as text and HTML. Don't worry if you are working offline. The script has canned data it will use when it cannot connect 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 left-most 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 run** checkbox at the bottom, and press **OK**.

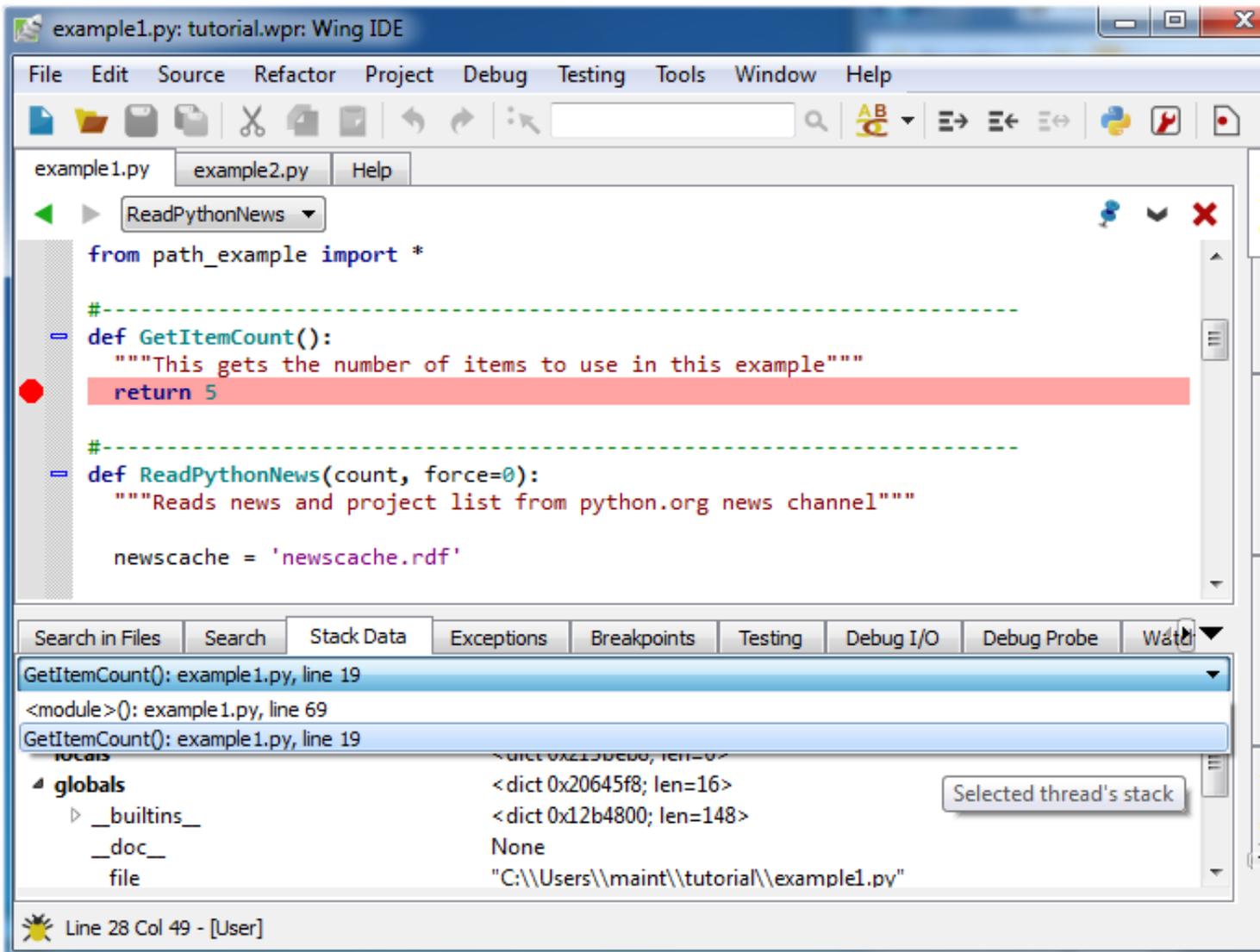
Wing will run to the breakpoint and stop, placing a red indicator on the line. Notice that the toolbar changes to include additional debug tools, as shown below:



Your display may vary depending on the size of your screen or if you have altered the toolbar's configuration. 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:

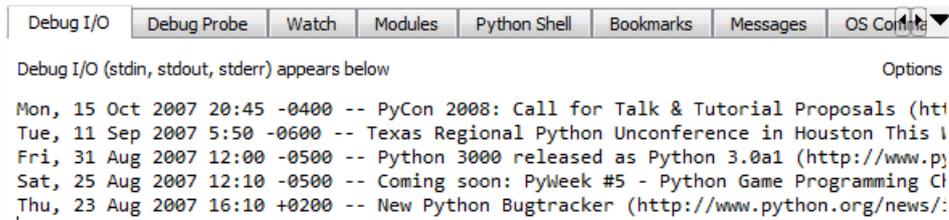


Next, try stepping out to the enclosing call to `ReadPythonNews`. In this particular context, you can achieve this in a single click with the **Step Out** toolbar icon or **Debug** menu item. Two clicks on **Step Over** also work. `ReadPythonNews` is a good function to step through in order to try out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

### 1.8.1. 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:



For code that tries to read from `stdin` or uses `input` (or in Python 2.x `raw_input`), the `Debug I/O` tool is where you would type your input to your program. Try this now by stepping over the `PromptToContinue` call. You will see the prompt “Press Enter to Continue” appear in the `Debug I/O` tool and the debugger will not complete the `Step Over` operation until you press Enter while focus is in the `Debug I/O` tool.

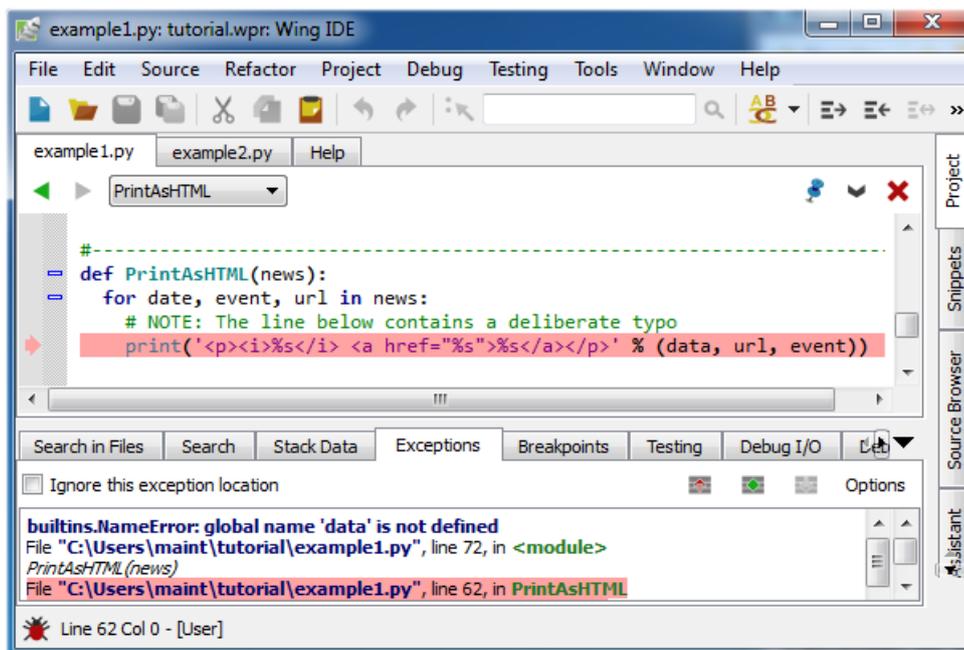
Note that you can also configure Wing to use an external console from the `Options` menu in the `Debug I/O` tool. This is useful for code that depends on details of the `Debug I/O` environment (such as cursor control with special output characters).

### 1.8.2. 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:



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 useful.

### 1.8.3. Tutorial: Execution Environment

In this tutorial we've been running code in the default environment and with the default Python interpreter. In a real project you may want to specify one or more of the following:

- Python interpreter and version
- PYTHONPATH
- Environment variables

- Initial run directory
- Options sent to Python
- Command line arguments

Wing lets you set these for your project as a whole and for specific files.

## Project Properties

The **Environment** and **Debug/Execute** tabs in the **Project Properties** dialog, accessed from the **Project** menu, can be used to select the Python interpreter that is being used, the effective `PYTHONPATH`, the values of environment variables, the initial directory for the debug process, and any options passed to Python itself.

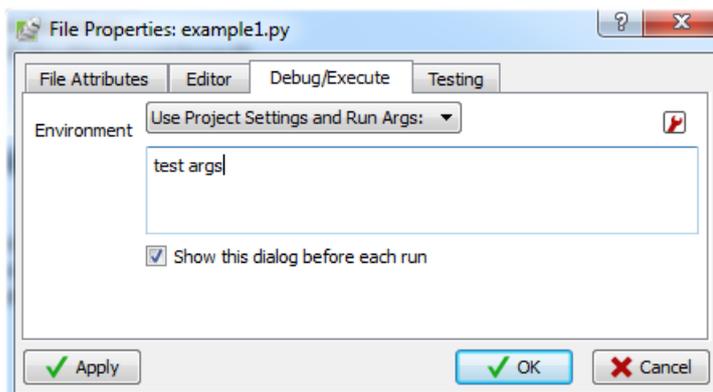
In most cases, **Project Properties** is where you will make changes to the runtime environment for all the project code that you execute and debug.

Try this out now by adding an environment variable `TESTPROJECT=1` to the **Environment** in **Project Properties**. Then restart debugging and look at `os.environ` to confirm that the new environment variable is defined.

## File Properties (and Launch Configurations)

**File Properties** are used to configure the command line arguments sent to a file when it is executed or debugger, and (optionally) to override the project-defined environment.

The **File Properties** dialog is accessed from the **Current File Properties** item in the **Source** menu or by right-clicking on a file in the editor or **Project** tool and selecting **Properties**.



The most common use of **File Properties** is simply to set the command line arguments to use with a file. Try this now by bringing up **File Properties** for `example1.py` and set the run arguments in the **Debug/Execute** tab to `test args`.

Now if you restart debugging and type the following in the `Debug Probe` you will see that the environment and arguments have been set:

```
os.environ.get('TESTPROJECT')
sys.argv[1:]
```

The output should be:

```
1
['test', 'args']
```

To also override the project-defined environment for a particular file, define a `Launch Configuration` and select it in `File Properties`. Launch configurations set up an environment like that which can be specified in `Project Properties`, paired with a particular set of command line arguments.

Try this now by bringing up `File Properties` for `example1.py` again and selecting `Use Selected Launch Configuration for Environment` under the `Debug/Execute` tab. Press the `New` button that appear, use “My Launch Config” as the name for the new launch configuration, and press `OK`. Wing will show the properties dialog for the new launch configuration.

Next enter run arguments `other args` and change the `Environment` to `Add to Project Values` and enter `TESTFILE=2` and `TESTPROJECT=`. This adds environment variable `TESTFILE` and removes the `TESTPROJECT` from the inherited project-defined environment.

Now restart debugging again and enter this in the `Debug Probe`:

```
os.environ.get('TESTPROJECT')
os.environ.get('TESTFILE')
sys.argv[1:]
```

The output should be:

```
None
2
['other', 'args']
```

## 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 `Debug Current File` 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 the **Project Properties** dialog.

Try this now by setting `example1.py` as the main debug file. Now it is no longer necessary to bring `example1.py` to front in order to start debugging it.

Whether or not you set a main debug file depends on the nature of your project.

### **Named Entry Points**

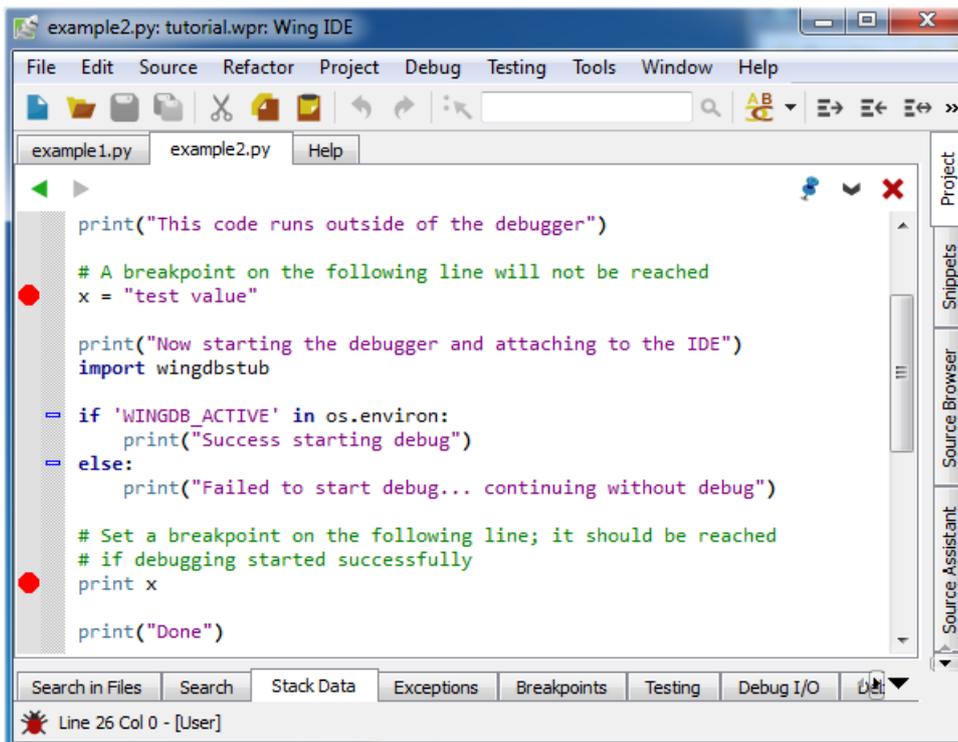
In some projects it is more convenient to define multiple entry points for executing and debugging code. To accomplish this, **Named Entry Points** can be set up from the **Debug** menu. Each named entry point binds an environment, either specified in the project or in a launch configuration, to a particular file. Once defined, they can be assigned a key binding or accessed from the **Debug Named Entry Point** and **Execute Named Entry Point** items in the **Debug** menu.

Named Entry Points are a good way to launch a single file with different arguments or environment.

## **1.8.4. Tutorial: Remote Debugging**

So far we've been debugging code launched from inside of Wing. Wing can also debug processes that are running in a web framework, as scripts in a larger application, or that get launched from the command line.

Let's try this now with `example2.py` in your tutorial directory. First, copy `wingdb-stub.py` out of your Wing IDE installation directory, which is listed 4th in Wing's **About** dialog box. Place this in the same directory as `example2.py`. Next, click on the bug icon in the lower left of Wing IDE's main window and select **Accept Debug Connections**. Then set a breakpoint on lines 10 and 22 of `example2.py`:



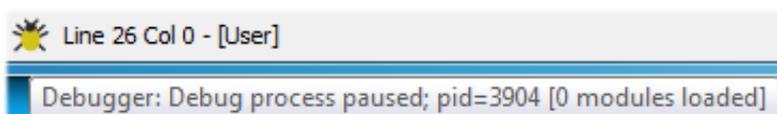
If you are working on OS X, using the Windows zip install, or using the Linux tar install of Wing, you will need to edit `wingdbstub.py` to set `WINGHOME` to the full path to the Wing IDE installation directory. This is done automatically by the regular Windows installer and Debian and RPM installs on Linux. If you are using one of those you can skip this step.

Now we're ready to debug `example2.py` when it is launched from outside of the IDE. To launch it, use the DOS Command prompt on Windows, a bash or similar command prompt on Linux, or Terminal or an xterm on OS X by typing:

```
python example2.py
```

You may need to specify the full path to python if it is not on your path.

This should start up the code, print some messages, connect to the IDE, and stop on the breakpoint on line 22. Read through the code and the messages printed to understand what is happening. You can verify that the debugger attached by looking at the color of the bug icon in the lower left of the IDE window, and by hovering the mouse over it:



Once you are stopped at a breakpoint or exception in externally launched code, the debugger works just as it would had you launched the debug process from the IDE. The only difference is that the environment is set up by the process itself and the settings specified in **Project Properties** and **File Properties** are not used.

When you continue the debugger from the toolbar or **Debug** menu, the program should print the value of `x` and exit.

This is a very simple example to illustrate how externally launched code can be debugged. The import of `wingdbstub` can also be placed in functions or methods, and there is a **debugging API** that provides control over starting and stopping debugging.

See **Debugging Externally Launched Code** for details and the **How-To guides** for information on setting this up with specific web frameworks, compositing and rendering tools, and other applications.

### Remote Debugging

Using the same mechanism, but with some additional configuration, it is also possible to debug Python code running on another machine. This is documented in **Remote Debugging** in the Wing IDE manual.

## 1.9. Tutorial: Indentation Features

Since indentation is syntactically significant in Python, Wing provides a number of features to make working with indentation easier.

### Auto-Indentation

By now you will have noticed that Wing auto-indent lines as you type, according to context. This can be disabled with the **Auto-Indent** preference.

Wing also adjusts the indentation of blocks of code that are pasted into the editor. If the indentation change is not what you wanted, a single **Undo** removes the indentation adjustment, if there was one.

### Block Indentation

In Wing's default keyboard personality, the **Tab** key is defined to indent the current line or blocks of lines, rather than entering a tab character (which can be done with **Ctrl-Tab**). As noted earlier, 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, or matched indentation according to context, from the Indentation toolbar group:



Repeated presses of the **Match Indent** tool will move the selected lines among the possible correct indent levels for that context.

These indentation features are also available in the **Source** menu, where their key bindings are listed.

## 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.

## Folding

Unless the feature is disabled with the **Enable Folding** preference, Wing can fold editor code by indentation levels to hide areas that are not currently of interest or as a way to see a quick summary of the contents of a source file.

The folding operations are enumerated in the **Folding** sub-menu of the **Source** menu and in the fold margin context menu.

Folding acts in such a way that selecting across a fold and copying will copy the text, including its hidden portions. Take a look at the **Folding** sub-menu in the **Source** menu and refer to **Folding** for details.

## 1.10. Tutorial: Other Editor Features

There are a number of other editor features that are worth knowing about:

### Goto-Line

Navigate quickly to a numbered source line with the **Goto Line** item in the **Edit** menu, or with the key binding displayed there. In some keyboard personalities, the line number is typed into the data entry area that appears at the bottom of the window. Press **Enter** to complete the action.

Line numbers can be shown in the editor with the **Show Line Numbers** item in the **Edit** menu.

### Selecting Code

Wing supports character, line, and block mode selection from the **Selection Mode** item in the **Edit** menu.

In Python code, the **Select** sub-menu in the **Edit** menu can be used to easily select and traverse logical blocks of code. The **Select More** and **Select Less** operations are particularly useful when preparing to type over or copy/paste ranges of text. Try these out now on `urllib` in `ReadPythonNews` in `example1.py`. Each repeated press of **Ctrl-Down** will select more code in logical units. Press **Ctrl-Up** to select less code.

The other operations in the **Select** sub-menu can be used for selecting and moving forward or backward over whole statements, blocks, or scopes. If you plan to use these and your selected **Keyboard Personality** does not support them, then you will want to define key bindings for them using the **Custom Key Bindings** preference. The command names are `select-x`, `next-x`, and `previous-x` where `x` is either `statement`, `block`, or `scope`.

### Line Editing

Lines can quickly be inserted, deleted, duplicated, swapped, or moved up or down with the operations in the **Line Editing** sub-menu of the **Source** menu. If your keyboard personality does not support them, then you can define key bindings for those you are interested in using. The command names are: `new-line-before`, `new-line-after`, `duplicate-line-above`, `duplicate-line`, `move-line-up`, `move-line-down`, `delete-line`, and `swap-lines`.

### Block Commenting

Lines of code can be commented out or un-commented quickly from the **Source** menu. In Python code, the **Block Commenting Style** preference controls the type of commenting that is used. The default is to use indented single `#` characters since this works better with some of Wing's other features.

### 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 invocations of the command will traverse outward or forward in the file.

### Text Reformatting

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

## 1.11. 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. Press the **Enter** key to 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.

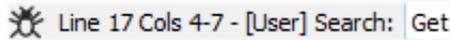
### Keyboard-driven Search

If you prefer a more powerful search interface using the keyboard only, try the key bindings for the items in the **Mini-search** sub-menu of the **Edit** menu (the bindings 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 or using regular expression matching. You can also initiate replace operations.

Try this in the `example1.py` file: If using the default editor mode, press the **Ctrl-U**. 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:



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 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, if there are any matches.

As in 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`.

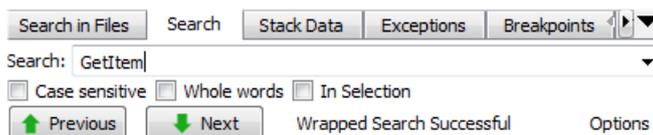
The regular expression based search options found in the **Mini-search** menu group work similarly but expect regular expressions for the search criteria (see below).

Keyboard-driven mini-replace 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.

Wing adjusts some details of how mini-search behaves according to keyboard personality. For example, in emacs mode `Ctrl-G` will cancel the search and in vi mode the search is always case sensitive, as in VI/VIM.

## 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.

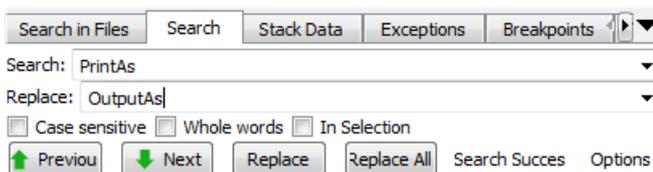


Searches may span the whole file or be constrained to the current selection, can be case sensitive or insensitive, and may optionally be constrained to matching only whole words.

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 add **Replace** and **Replace All** buttons to the Search tool:



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.

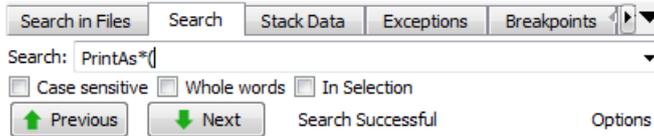
## Wildcard Searching

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 `*` to match anything, `?` to match a single character, or ranges of characters specified within `[` and `]` to match any of the specified characters. 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 match all occurrences of the string `PrintAs`, followed by zero or more characters, followed by `(`:



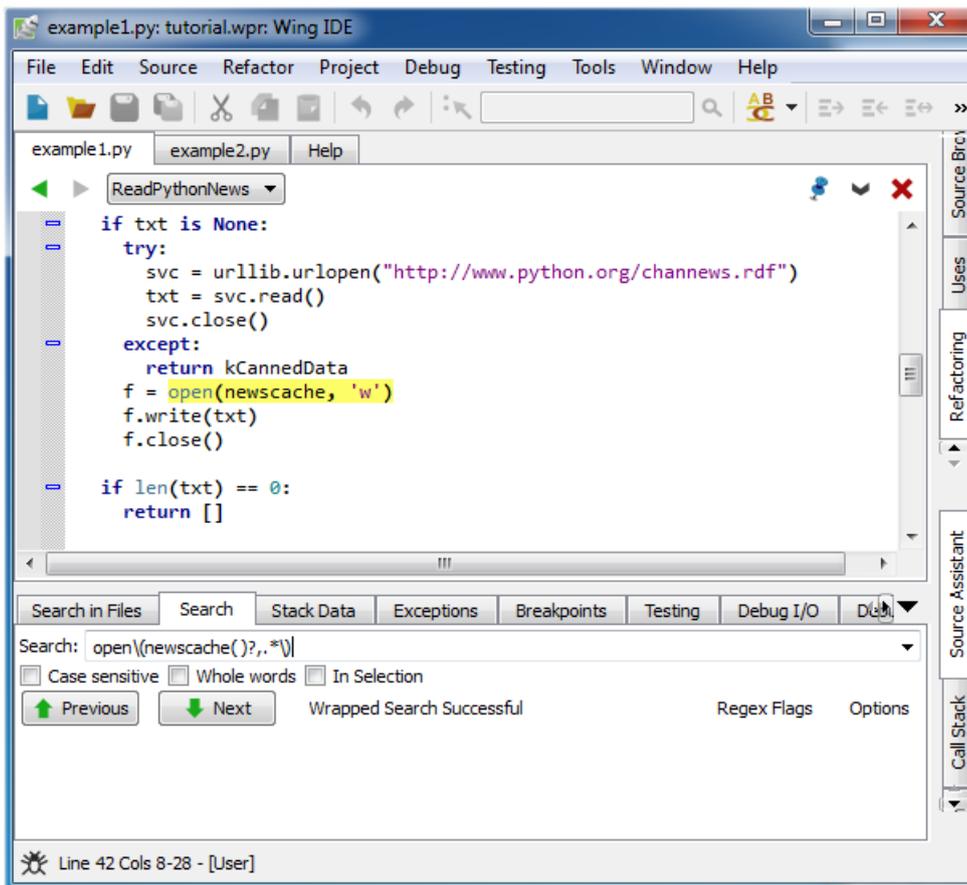
Also try searching on `PrintAs*[A-Z](` with the **Case Sensitive** search option turned on. This matches all strings starting with `PrintAs` followed by zero or more characters, followed by any capital letter from `A` to `Z`, followed by `(`.

Finally, try `PrintAsT???`, which will match any string starting with `PrintAsT` followed by any three characters.

## 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.

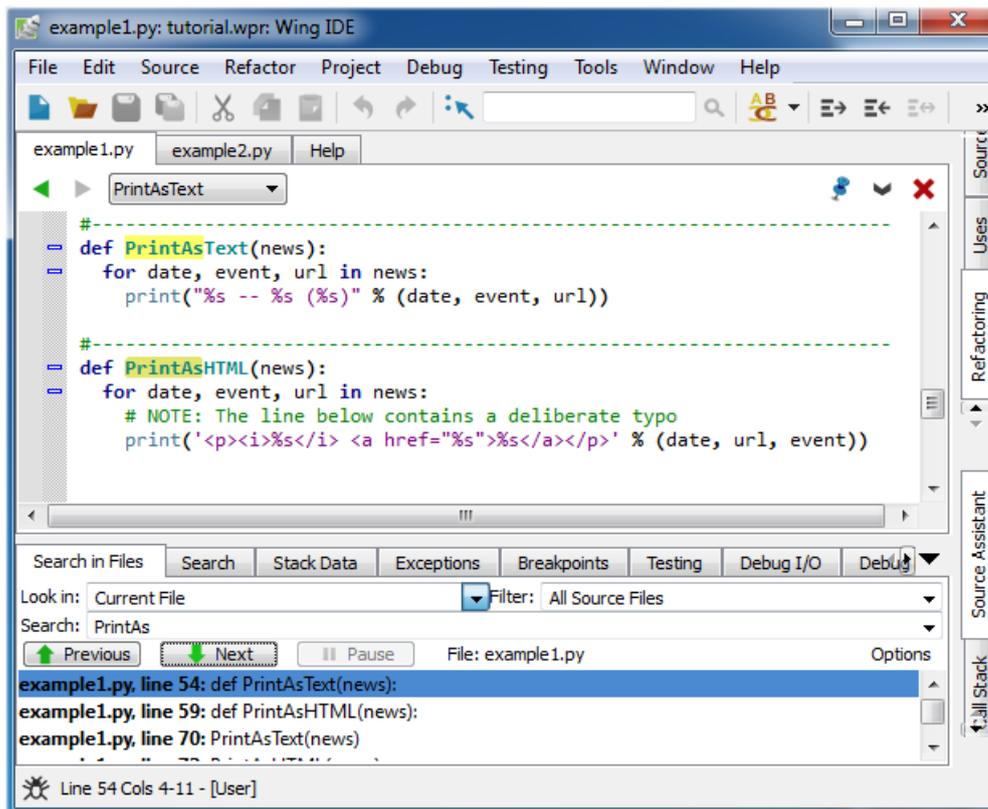
For example, `open\(newscache( )?, .*\)` matches only calls to the function `open` where the first argument is named `newscache` and there are at least two parameters. If you try this with `example1.py`, you should get exactly one search match:



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.



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 in Files` tool. 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` tool to traverse your results.

## File Filters

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 more useful if constrained to a subset of files in your projects

such as only Python files. This can be done with by selecting **Python Files** in the **Filter** selector. You can also define your own file filters using the **Create/Edit Filters...** item in the **Filter** selector. This will display the **Files > File Types > File Filters** preference:

File Filters	Name	Specification
	All Source Files	: Wild Card on File Name: *.pyo; Wild Card on File Name: *\$py.class; Wild Card .
	C/C++ Files	Mime Type: text/x-c-source; Mime Type: text/x-cpp-source; Wild Card on Direct
	HTML and XML Files	Mime Type: text/html; Mime Type: text/xml; Mime Type: text/x-zope-pt; Wild C;
	Hidden & Temporary Files	Wild Card on File Name: *.pyo; Wild Card on File Name: *\$py.class; Wild Card ...
	Python Files	Mime Type: text/x-cython; Mime Type: text/x-python; Wild Card on Directory N
	<input type="button" value="Insert"/>	<input type="button" value="Remove"/> <input type="button" value="Edit"/>

Each file filter 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 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** menu, which contains several 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 and thus the search only needs to look in the files and not spend time discovering them.

## Multi-File Replace

When working with multiple files in the result set, Wing opens 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.

If you check **Replace Operates on Disk** in the **Options** menu within the **Search in Files** tool, 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 if 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 selecting **Save All** from the file menu immediately after each replace operation. This avoids losing parts of a replace, resulting in inconsistent application of a replace operation to the files in your source base.

## 1.12. Tutorial: Further Reading

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

- [Wing IDE Support Website](#) which includes our mailing lists and other information for Wing IDE users.
- **Wing IDE Reference Manual** which documents the features in detail.
- **How-To Guides** for information on using Wing with frameworks like Django, Plone, Google App Engine, matplotlib, Autodesk Maya, NUKE, and others.