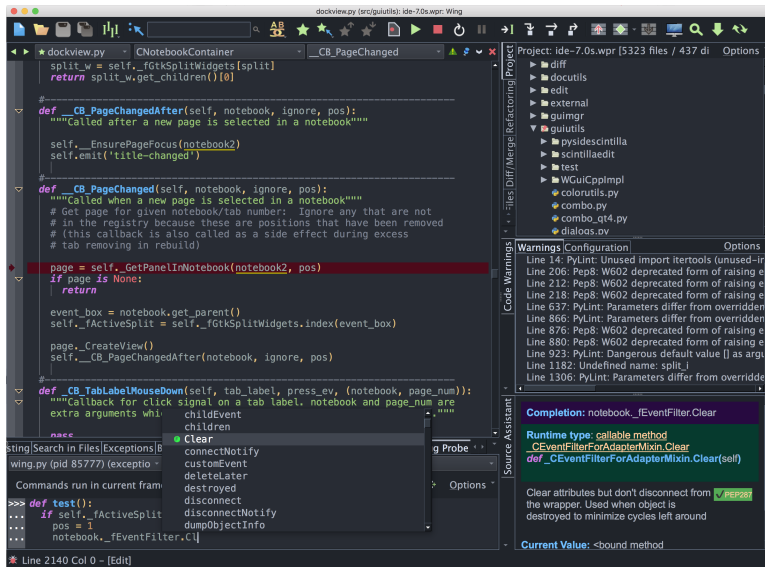





Wing Pro Tutorial



This tutorial introduces Wing Pro by taking you through its feature set with a small coding example. For a faster introduction, see the [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 101 to teach programming with Python.

Our [How-Tos](#) show how to use Wing with 3rd party web development frameworks, GUI toolkits, scientific data visualization tools, Python-based modeling, rendering & compositing systems, and other Python frameworks and toolkits. A collection of [Wing Tips](#), available on our website and by weekly email subscription, provides additional tips and tricks for using Wing productively.

To get started, press the  **Next Page** icon in the toolbar immediately above this page.


Tutorial: Why Wing?

Wing Pro is a light-weight but powerful integrated development environment that was designed from the ground up for Python. As you get up to speed with Wing Pro you should find that:

- Wing speeds up your development of new code
- Wing makes it easier to understand and work with existing code
- Wing reveals errors earlier in the development process
- Wing makes it easier to find and fix bugs
- Wing adapts to your needs and style

This is made possible through deep code analysis (both static and runtime), a focus on interactive development in the live runtime, high-level editing operations and refactoring, continuous early error detection, support for test-driven development, powerful always-on debugger, seamless support for remote and containerized development, and extreme configurability.

There is a lot to learn about the IDE, but don't be intimidated. You can get started with a subset of the functionality, adding in other features and tools over time as you become more comfortable. The user interface can be customized in color and layout, and you can easily remove tools that you are not yet interested in.

Let's get started! To get to the next page in the tutorial, use the  **Next Page** icon in the toolbar immediately above this page.

Tutorial: Getting Started

To get started using this tutorial, you will need to:

Install Python

If you don't already have it on your system, install Python from python.org. Wing also supports Python provided by Anaconda ActivePython, MacPorts, Fink, Homebrew, cygwin, and others. See [Supported Python Versions](#) for details.

Install Wing

Then install [Wing](#). For detailed instructions, see [Installing Wing](#).

Start Wing

Wing can be started from a menu, desktop, or tray icon, or by using the command line executable. For detailed instructions, see [Running Wing](#). If you are using Wing Pro and don't have a license, you can obtain a free 30-day trial (divided into three 10-day periods) the first time you start Wing.

Switch to the Integrated Tutorial

Once Wing is running, you should switch to using the **Tutorial** listed in Wing's **Help** menu because it contains links directly into the IDE's functionality. This includes the next step below.

Copy the Tutorial Directory

Copy the entire **tutorial** directory out of the **Install Directory** listed in Wing's **About box** to another location on disk. You can do this manually or use the following link, which will prompt you to select the target directory:

Copy Tutorial Now

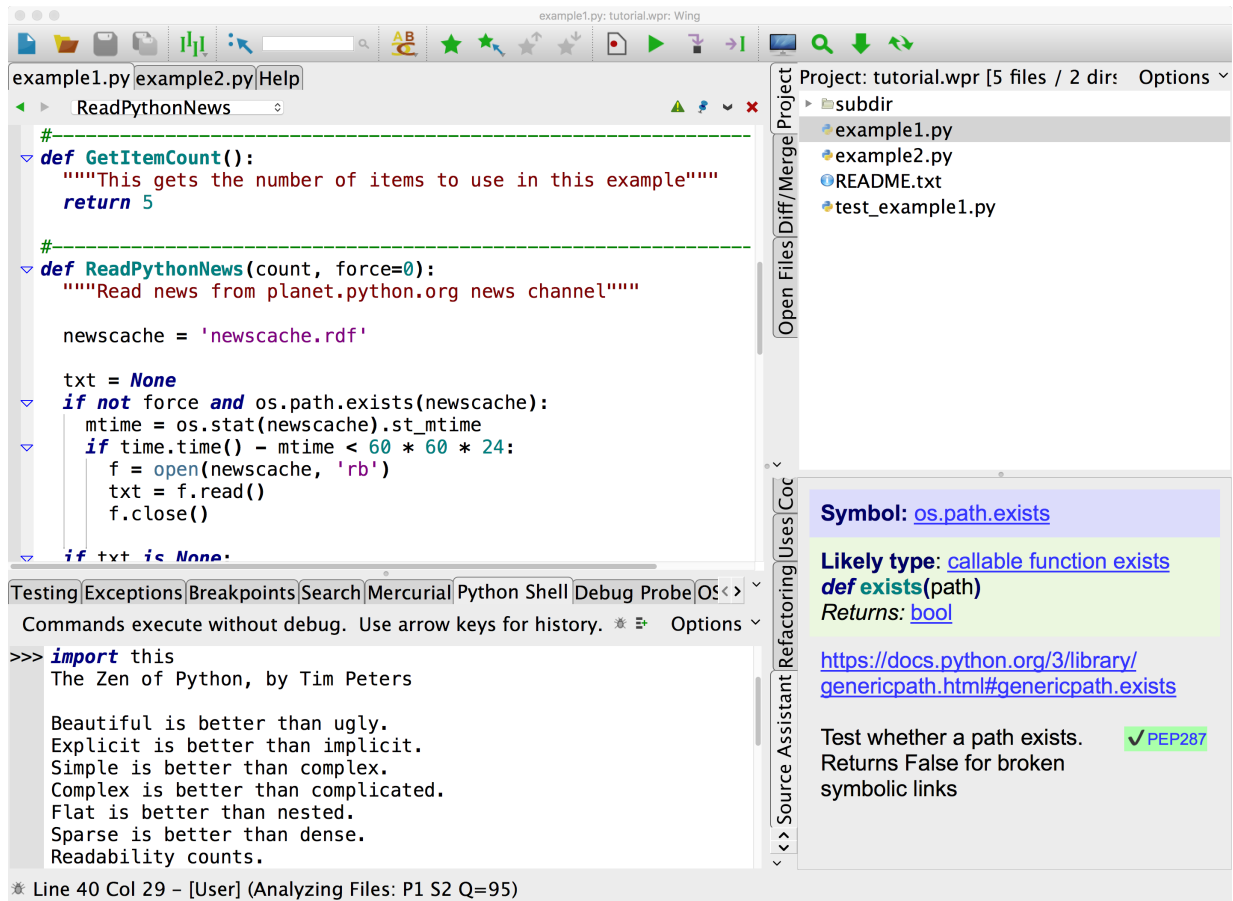
Note

We welcome feedback, which can be submitted with **Submit Feedback** in Wing's **Help** menu or by emailing support@wingware.com

Tutorial: Getting Around Wing

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

Wing'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 or moving them to a separate window. 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 display.

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.

See [User Interface Layout](#) for details.

Context Menus

In general, right-clicking provides a menu for interacting with or configuring a part of the user interface. On some systems you may need to configure your track pad to allow right-clicking, or use a keyboard modifier to emulate a right mouse click.

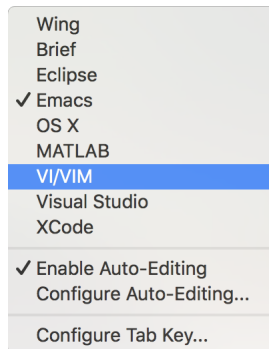
Splitting the Editor Area

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 selecting **Move Wing Help to New Window**.

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.

Configuring the Keyboard

Use the **Edit > Keyboard Personality** menu to tell Wing to emulate another editor, such as Visual Studio, VI/Vim, Emacs, Eclipse, XCode, MATLAB, or Brief.



The **Configure Tab Key** item in the **Edit > Keyboard Personality** menu 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**, **Tab** acts differently according to context. For example, if lines are selected, repeated presses of **Tab** moves the lines among syntactically valid indent positions. And, when the caret is at the end of a line, pressing **Tab** adds one indent level.

See [Keyboard Personalities](#) for details.

Accessing Preferences

Preferences for controlling how Wing Pro looks and behaves are available from the **Edit > Preferences** menu item (or **Wing Pro >> Preferences** on macOS). Try this now so you will remember how to bring up the preferences dialog as you work through the rest of this tutorial. However, it's best not to delve into all of the available options right away. The next few sections will highlight some of the more important ones that are worth looking at now.

Auto-Editing

Wing Pro implements a variety of auto-editing operations, which are designed to speed up typing and reduce common errors. A subset of the available operations that does not require learning different keystrokes is enabled by default. For example, when **(** is typed Wing will enter the closing **)** automatically. If the closing **)** is pressed anyway, Wing just skips over it. Auto-editing can be disabled as a whole using the **Editor > Auto-editing > Enable Auto-Editing** preference or individual operations can be selected.

- Auto-Editing Enabled
- Auto-Close Characters
- Auto-Enter Invocation Args
- Auto-wrap Arguments
- Invoke After Completion
- Apply Quotes to Selection
- Apply Comment Key to Selection
- Apply (), [], and {} to Selection
- Apply Colon to Selection
- Auto-Enter Spaces
- Auto-Space After Keywords
- Enforce PEP 8 Style Spacing
- Spaces Around : in Type Annotations
- Manage Blocks on Repeated Colon Key Presses
- Continue Comment or String on New Line
- Correct Out-of-Order Typing

This topic will be covered in more detail later in the tutorial.

Auto-Completion

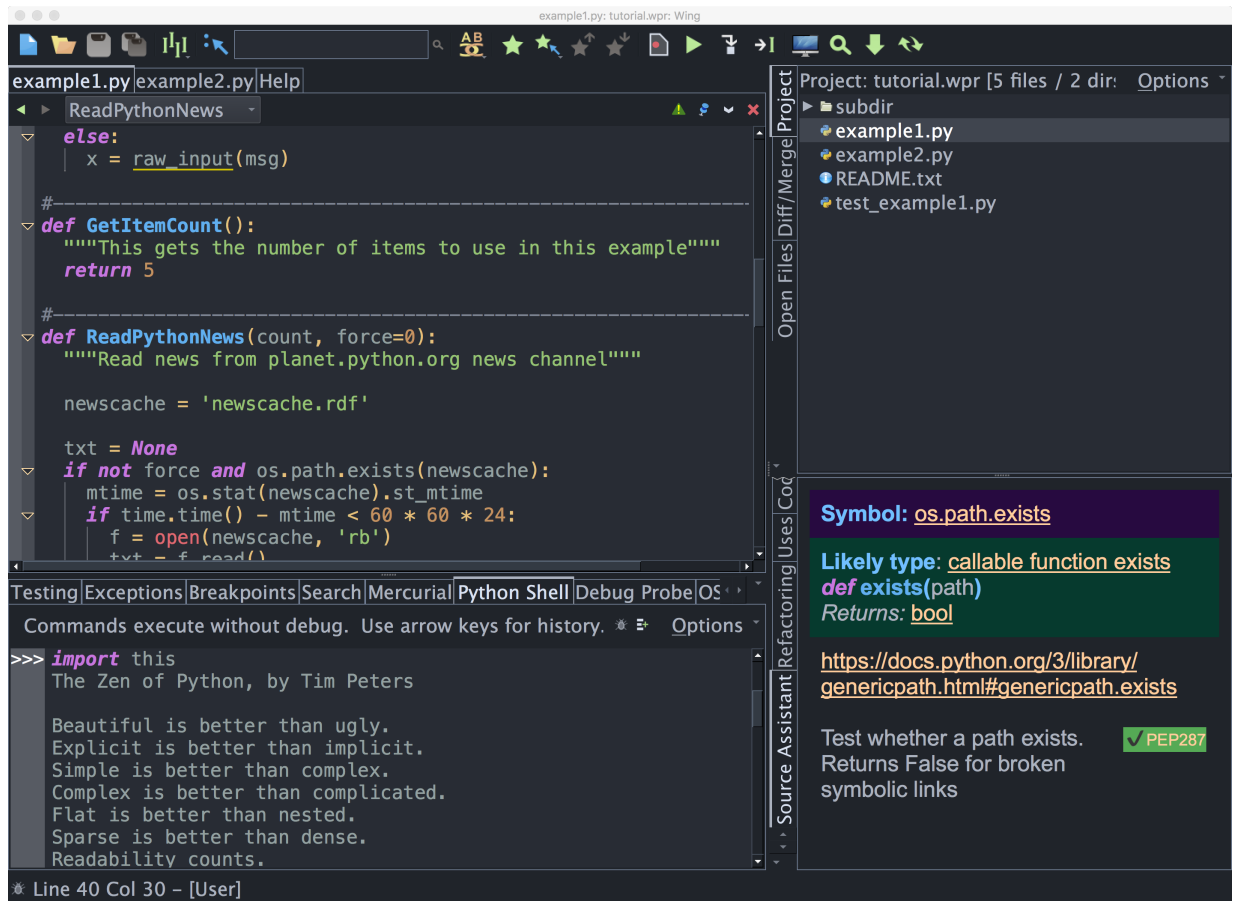
There are many options for Wing's auto-completer. These are set in the **Editor > Auto-completion** preferences group. For example, if you want to use the **Enter** key for completion, you may wish to select that now in the **Editor > Auto-completion > Completion Keys** preference.

Colors and Dark Mode

colors for the user interface can be selected with the **User Interface > Display Mode**, **User Interface > Light Theme** and **User Interface > Dark Theme** preferences.

The colors used in editor areas and some of the tools, including the Python Shell, can be set independently with the **User Interface > Light Editor** and **User Interface > Dark Editor** preferences.

The following screenshot was created by setting **Display Mode** to **Use Dark Theme** and selecting **One Dark** for the **Dark Display Theme**. This display mode will be used in the rest of this tutorial:



Other Configuration Options

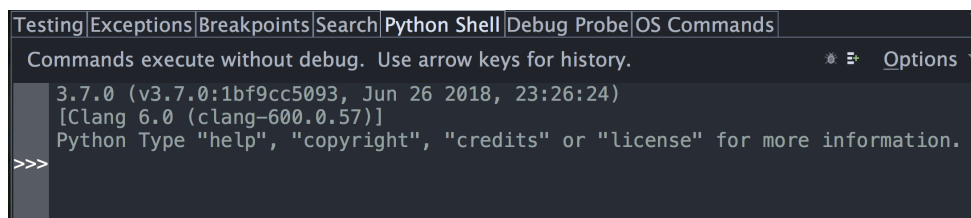
To set the fonts in the user interface and editor, change the **User Interface > Fonts > Display Font/Size** and **User Interface > Fonts > Editor Font/Size** preferences.

The size and type of tools used in the toolbar at the top of Wing's 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 [Customization](#).

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 **Python Executable** in **Project Properties**, accessed from the **Project** menu.

An easy way to determine the value to use for **Python Executable** is to start the Python you wish to use with Wing and type the following at Python's **>>>** prompt:

```
import sys
print(sys.executable)
```

You can also use a virtualenv or Anaconda environment by selecting the **Activated Env** option here, but for now let's just use the base Python installation.

You will need to **Restart Shell** from the **Options** menu in the **Python Shell** tool after altering **Python Executable**.

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 into the shell, such as:

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

Note that Wing offers auto-completion as you type and shows call signature and documentation in the **Source Assistant** tool. Use the **Tab** key to enter a selected completion. Other keys can be set up as completion keys with the **Editor > Auto-completion > Completion Keys** preference.

You can create as many instances of the **Python Shell** tool as you wish by right-clicking on a tool tab and selecting **Insert Tool**. Each one will run in its own process space.

Tutorial: Set Up a Project

Now we're ready to get started with some coding. The first step 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 installation, please do so now as described in [Tutorial: Getting Started](#).

Wing starts up initially with the default project. Instead of using that, create a new project now with **New Project** in the **Project** menu. Select **Create Blank Project** on the local host and then click on **OK**:

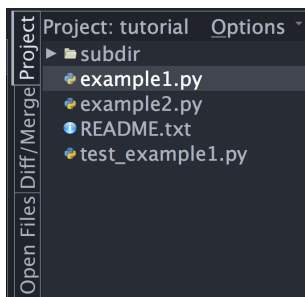


Wing will display a confirmation dialog after creating the new blank project:



Click **Save Now** to save the new project. 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.



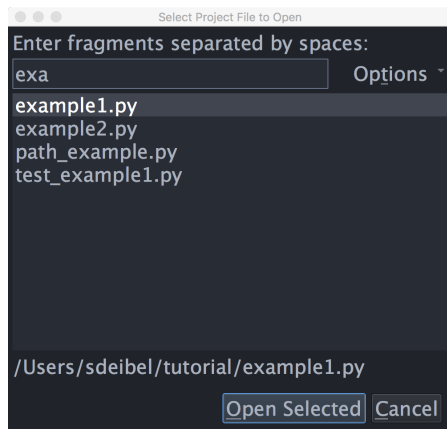
Note

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 **Open From Project** now, from the **File** menu. Type **ex** as the file name fragment and then use the arrow keys and press **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, so you'll probably want to learn the key binding listed for this command in the **File** menu. The binding varies according to which keyboard personality you have chosen to use.



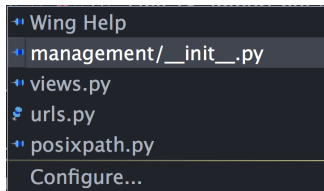
Transient, Sticky, and Locked Files

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, click on **os** in **import os** at the top of **example1.py** and press **F4** to go to the definition of **os**. The file **os.py** will be opened in non-sticky transient mode, so that it is automatically closed in least-recently-used order when you navigate away from it to other files.

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 may be closed when it is no longer visible. When a non-sticky transient file is edited, it immediately converts to sticky.
-  - The file is locked in the editor, so that the editor split will not be used 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. This contains both non-sticky transient and sticky files, while the **Recent** list in the **File** menu contains only sticky files.



The number of non-sticky transient editors to keep open, in addition to those that are visible, is set with the **Editor > Advanced > Maximum Non-Sticky Editors** preference (default=5).

This mechanism is also used in multi-file searching, debugging, 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.

Shared Project Files

Wing Pro writes two files for each project, for example **tutorial.wpr** and **tutorial.wpu**. If you plan to use Wing projects with a revision control system such as **Git**, **Mercurial**, or **Perforce**, you should check in only the ***.wpr** file.

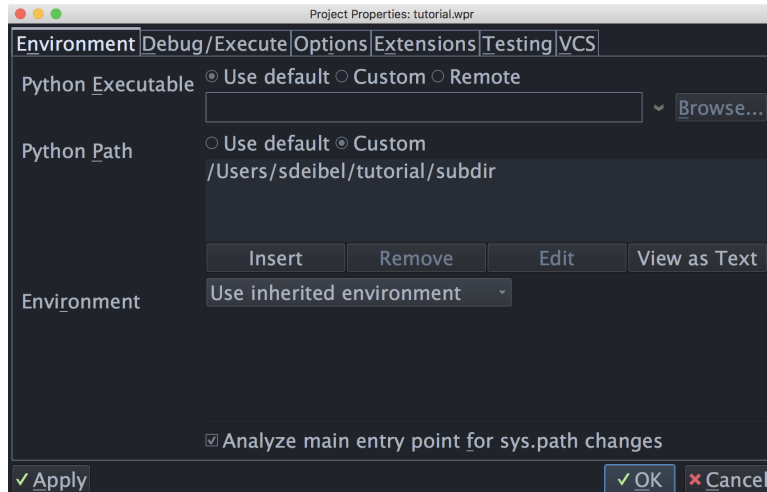
See the [Sharing Projects](#) documentation page for details.

Tutorial: Setting Python Path

Python uses a search path referred to as the Python Path to find modules that are imported into code with the **import** statement. Most code only imports modules that are already on the default path, for example modules in the Python standard library, or modules installed into Python by **pip**, **poetry**, **pipenv**, **conda**, or some other package manager.

However, in some cases code will depend on a different path provided either by setting the environment variable **PYTHONPATH** before starting Python, or by modifying **sys.path** at runtime before importing modules.

If the **Python Path** is changed by one of these methods, you may also need to tell Wing about this change. This is done with **Python Path** in **Project Properties**, accessed from the **Project** menu:



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 during debugging, if 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 the **Python Path** by moving the **path_example.py** file to the same location as the example scripts.

Startup Environment

Wing uses its startup environment as the default environment for your Python code. As a result, if **PYTHONPATH** is set when you start Wing, it will also be used with your code. If this inherited path matches the needs of your code, then you don't need to set **Python Path** in Wing. However, if you have different Python environments on your system or code with different path expectations, then you should set **Python Path** in the project so that switching projects will also switch to the correct environment.

Virtualenv and Anaconda Environments

If you are using **virtualenv**, Anaconda environments, Poetry, or **pipenv** to set up your Python environment, you don't need to set **Python Path**. Instead, set **Python Executable** to **Activated Env** and enter the command that activates your environment. This causes Wing to pick up the correct path and other environment needed to run code in the environment. In this case, Python is launched by running **python** in that environment.

You can also create a new virtualenv or Anaconda environment at the same time as creating a Wing project by selecting the **Create New Virtualenv** or **Create New Anaconda Environment** project types in the **New Project** dialog, accessed from the **Project** menu.

But don't do this now; you'll need the current project as you work through this tutorial.

Python Path Analysis

If your main entry point alters `sys.path`, and the file is set as the **Main Entry Point** in **Project Properties** then Wing may be able to automatically determine the correct path to use.

When in doubt, compare the value of `sys.path` at runtime in your code with the value reported by **Show Python Environment** in the **Source** menu. If they match then there is no need to set **Python Path** in your project.

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 to provide auto-completion, call tips, goto-definition, code warnings, and other editing and navigation 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 12 Col 16 - [Edit] (Analyzing Files: P2 S1 Q=2)
```

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

Editing with Wing

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** from the Tools menu or by clicking on its tab. This is where Wing shows documentation, call signature, and other information as you move around in your source code or work with other tools.

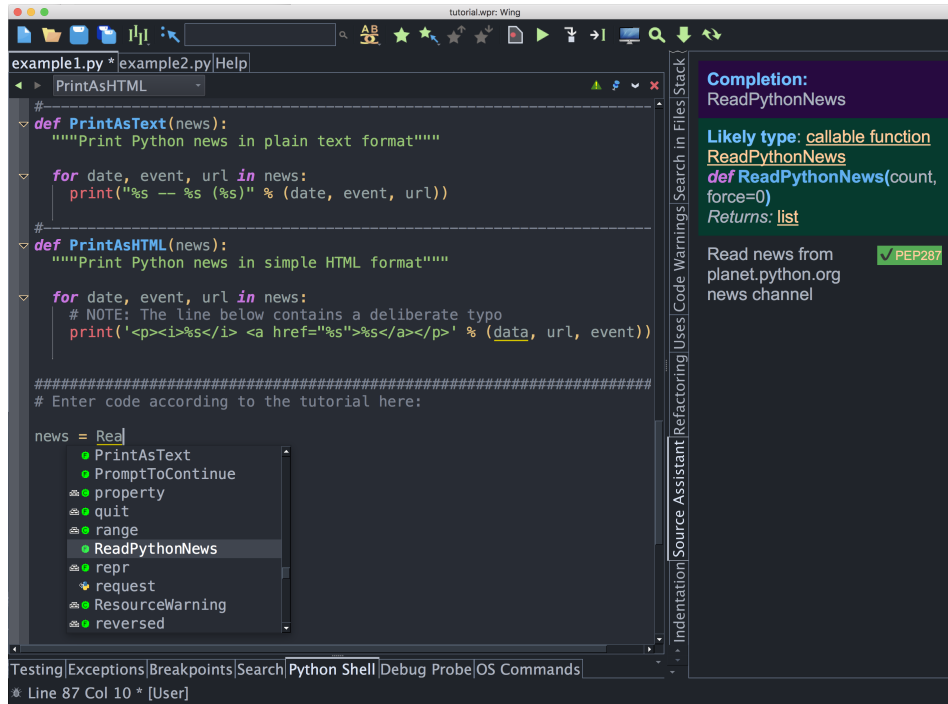
Scroll down to the bottom of `example1.py` and enter the following code by typing (not pasting) it into the file:

```
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.

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 symbol is selected in the auto-completer:



Next, press **Tab** to enter the completion of **ReadPythonNews** and type **(** (left parenthesis). In Wing Pro, you should now see the following code in your editor because Wing auto-enters the argument list and closing parenthesis:

```
news = ReadPythonNews(count, force=0)
```

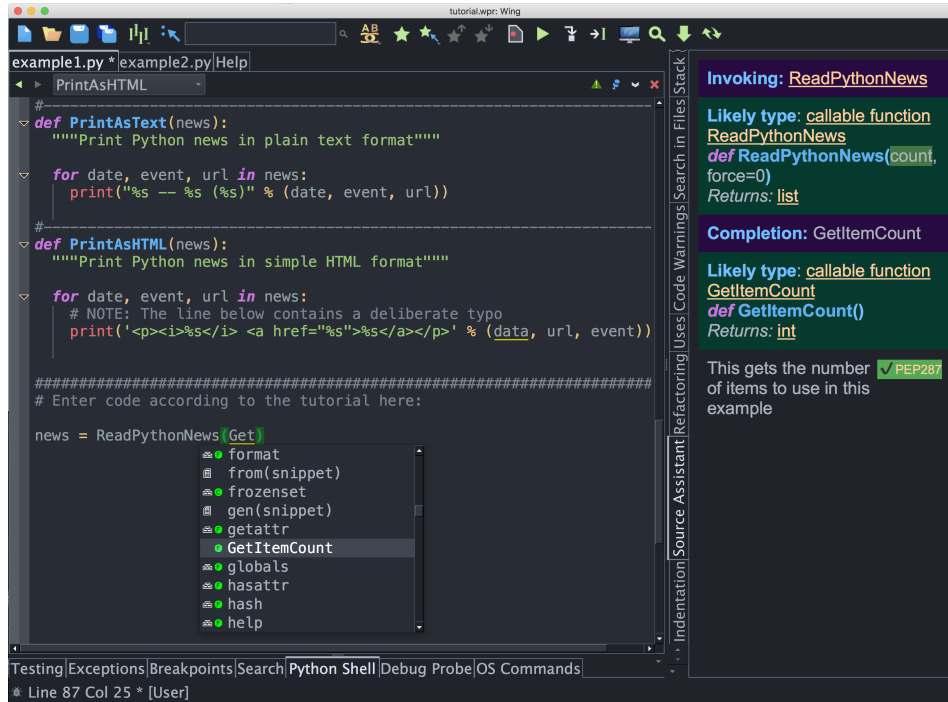
Notice that when Wing Pro auto-enters arguments, it starts with all arguments selected so you have the option of simply typing over them. Alternatively, the **Tab** key can be used to move between and replace arguments or just the default value in keyword arguments (like **force** in this example). When argument entry is completed by pressing **)** at the end of the list or by moving the caret out of the list, Wing automatically removes any keyword arguments with unaltered defaults.

Try this a few times now to get a feel for how the tab order works. **Undo** can be used to easily undo all changes made during argument entry. If you prefer not to use this feature, it can be turned off with the **Editor > Auto-Editing > Auto-Enter Invocation Args** preference. The same preferences page can be used to disable auto-editing entirely or to enable and disable other operations. The default set of enabled auto-editing operations are those that should not interfere significantly with finger memory. The other operations will be described later.

Now edit the code you have entered so it reads as follows and the caret is inside the **()**:

```
news = ReadPythonNews()
```

Then 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** right-click context menu.

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

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

Notice that typing a close parenthesis at the end of the invocation in Wing Pro skips over the close parenthesis that was previously auto-entered.

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.

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. This includes links to the point of definition of symbols. 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.

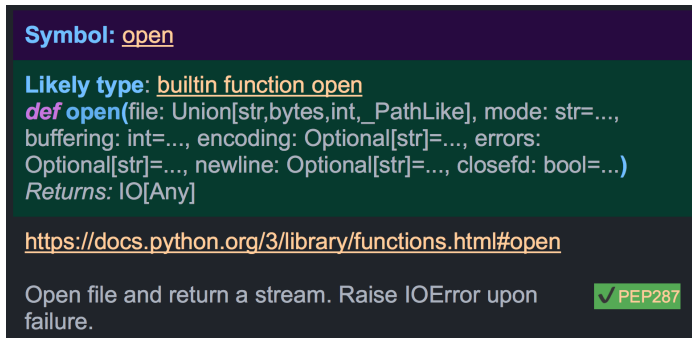
After visiting the point of definition with one of these links, use the green back arrow at the top left of the editor to return from the value or type definition:



The link after **Symbol:** goes to the point of definition of that variable, while any links after **Type:** or **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 and attributes. For example, for **news** the point of definition is the line where **news** 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 <https://docs.python.org> link. The documentation will be opened in your default web browser.



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

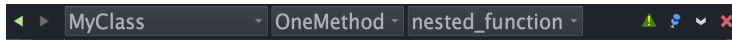
Goto-Definition

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**.

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.

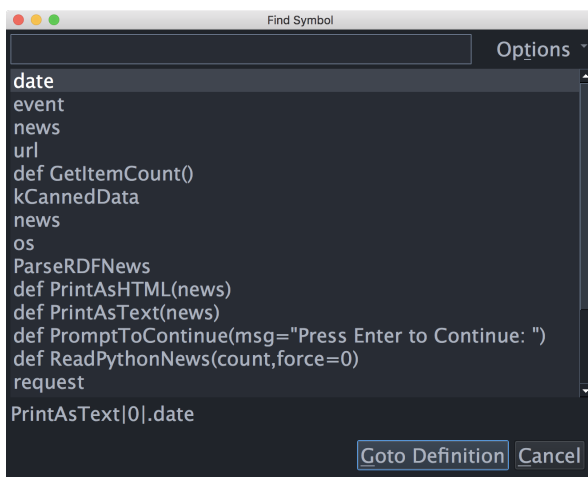


Try these now to navigate to **CHandler** in **path_example.py**, and then use the second menu to navigate to **endElement**.

Then 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

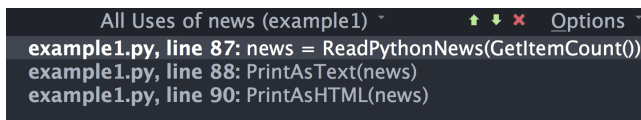
If you are looking for a symbol defined in the current scope, use **Find Symbol** in the **Source** menu. This displays a dialog where you can type a fragment matching the symbol name. Use the arrow keys to traverse the matches and press **Enter** to visit the symbol's point of definition.



Find Symbol in Project in Wing Pro functions in the same way but searches all files in the project for a symbol.

Find Points of Use

In Wing Pro it is also possible to enumerate and visit all points of use of a symbol. Try this now by right-clicking on **news** and selecting **Find Points of Use**. Wing will display the **Uses** tool with a list of all the points of use for that symbol. Click on the uses to visit them in the editor.




Note that Wing distinguishes between the **news** that is defined at the top level of **example1.py**, in the code that you typed, and the like-named but independent variables **news** inside the various functions

here. For an example, use **F4** to go to the definition of **ReadPythonNews** and run **Find Uses** on the variable **news** defined at the bottom of the function. The results are distinct from those returned for the top-level **news**.


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.

Tutorial: Debugging

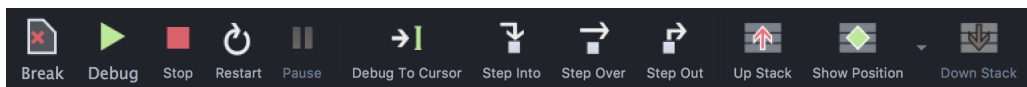
The **example1.py** program you have just created connects to **python.org** via HTTP, reads and parses the Python-related news feed in RDF format, and then prints the most recent five items as text and HTML. Don't worry if you are working offline. The script has canned data it will use when it cannot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

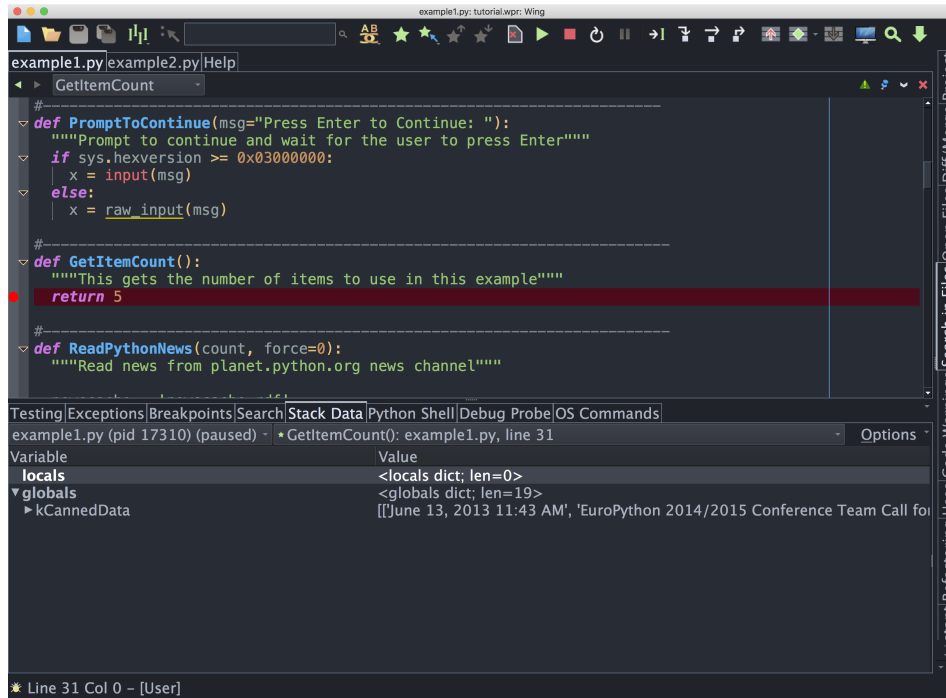
Next start the debugger with  **Debug** in the toolbar or the **Start/Continue** item in the **Debug** menu. Wing will show the **Debug Properties** dialog with the properties that will be used during the debug run. Just ignore this for now, uncheck the **Show this dialog before each run** checkbox at 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 items do when you hover the 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 with  **Up Stack** and  **Down Stack** in the toolbar or from the **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. Hover the 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` in the toolbar or `Debug` menu. Two clicks on `Step Over` also work. `ReadPythonNews` is a good function to step through in order to try out the basic debugger features described above.

Try stepping or running to a breakpoint on the last line of this function, which reads `return news[:count]`. In this context, right-clicking on `news` under `locals` in `Stack Data` allows viewing the value in textual form or as an array. The latter loads data incrementally for only the visible portion of the value, which is useful with `numpy` arrays, pandas `DataFrames`, `sqlite` query results, and other larger data sets.

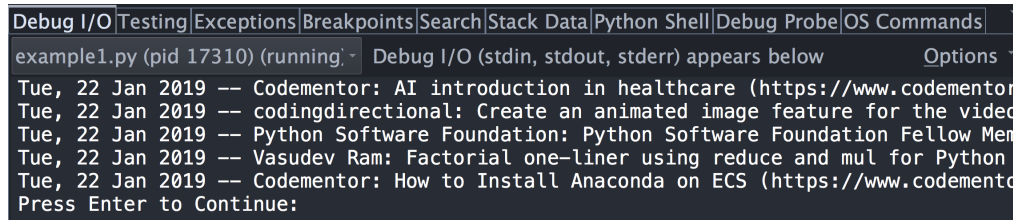
Data can also be viewed in tooltips on the editor by hovering the mouse over a value. Try this with `count` to see the value `5`. In Wing Pro, pressing `Shift-Space` displays tooltips for all values visible in the editor, if they are defined in the current stack frame. The last line of `ReadPythonNews` is also a good place to try that.

Finally, try `Step Over` to reach the return event in `ReadPythonNews`, which is indicated by a change from the solid debug line marker to an underline. Notice that hovering the mouse over `return` in the editor displays the value that is being returned from the function. Similarly, `<return value>` is added to the `locals` shown in the `Stack Data` tool.

9.1. 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 it.

Once you step over the line **PrintAsText(news)** you should see output similar to the following:



```
Debug I/O | Testing | Exceptions | Breakpoints | Search | Stack Data | Python Shell | Debug Probe | OS Commands
example1.py (pid 17310) (running) - Debug I/O (stdin, stdout, stderr) appears below Options
Tue, 22 Jan 2019 -- Codementor: AI introduction in healthcare (https://www.codementor.com/...)
Tue, 22 Jan 2019 -- codingdirectional: Create an animated image feature for the video...
Tue, 22 Jan 2019 -- Python Software Foundation: Python Software Foundation Fellow Merr...
Tue, 22 Jan 2019 -- Vasudev Ram: Factorial one-liner using reduce and mul for Python...
Tue, 22 Jan 2019 -- Codementor: How to Install Anaconda on ECS (https://www.codementor.com/...)
Press Enter to Continue:
```

For code that reads from **stdin** or uses **input()** or Python 2.x's **raw_input()**, the **Debug I/O** tool is where you would type input to your program. Try this now by stepping over the **PromptToContinue** call with **Step Over** in the toolbar. 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 keyboard focus is in the **Debug I/O** tool.

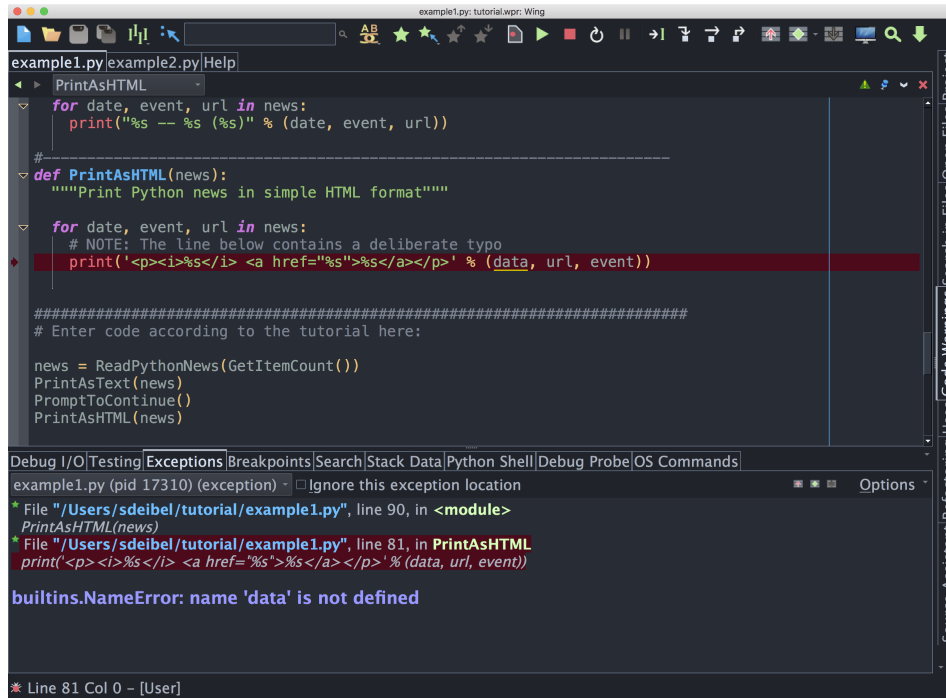
You can also configure Wing to use an external console from the **Options** menu in the **Debug I/O** tool. This is useful for programs that requires a more complete console implementation to run correctly, for example those that use the **curses** module.

See [Debug Process I/O](#) for details.

9.2. 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 icon or **Start / Continue** in the **Debug** menu. Wing will stop on an incorrect line of code in **PrintAsHTML** and report the problem in the **Exceptions** tool:



The **Exceptions** tool highlights the current stack frame as you move up and down the stack. You can click on frames to navigate the exception backtrace, showing the source code for each frame.

Whenever you are stopped on an exception, the debugger status indicator in the lower left of Wing's main window turns red.

After reaching an exception in the debugger, you can correct your code, stop the debugger with the **Stop** icon in the toolbar, and then start debugging again. But don't do this yet, since we'll be working within the current debug context in the next section.

Exception Handling Options

In Wing Pro and Wing Personal, the debugger provides several exception handling modes, which differ in how they determine when exceptions should be reported. It is also possible to ignore specific exceptions and to specify exception types to always report or never report. Most users will not need to alter these options, but being aware of them is useful.

See [Managing Exceptions](#) for details.

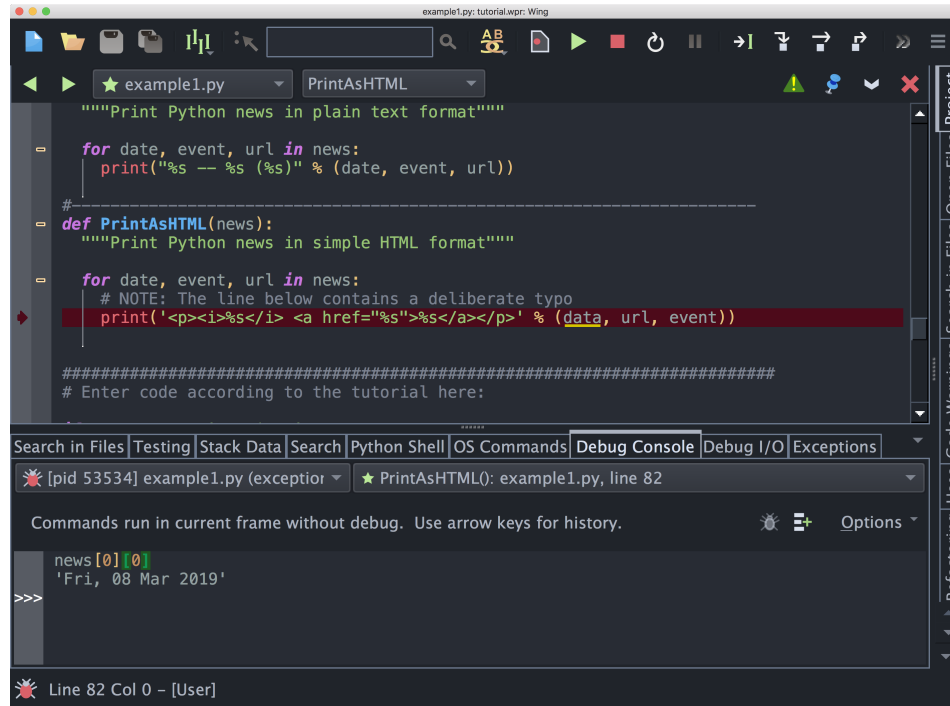
9.3. Tutorial: Interactive Debugging

Wing Pro's **Debug Console** provides a powerful way to find and fix bugs, and to try out new code interactively in the live runtime state. This works much like the **Python Shell** tool 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:



The screenshot shows the Wing IDE interface. The main editor window displays a Python script named 'example1.py' with the following code:

```
"""Print Python news in plain text format"""  
for date, event, url in news:  
    print("%s -- %s (%s)" % (date, event, url))  
#-----  
def PrintAsHTML(news):  
    """Print Python news in simple HTML format"""  
    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>' % (date, url, event))  
#####  
# Enter code according to the tutorial here:
```

The 'PrintAsHTML' function call is highlighted in red. Below the editor, the 'Debug Console' is open, showing the output of the function call:

```
news[0][0]  
'Fri, 08 Mar 2019'
```

Wing offers auto-completion as you type and shows call signature and documentation information in the **Source Assistant**, just as it does when you work in the editor.

Next, try this:

```
news[0][0] = '2013-06-15'
```

This is one way to change program state while debugging, which can 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 Console** using the modified value for **news**.

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

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

This replaces the buggy definition of `PrintAsHTML` found in the `example1.py` source file for the life of the debug process, so that you can now execute it without errors as follows:

```
PrintAsHTML(news)
```

The **Debug Console** is useful in designing fixes for bugs that depend on lots of program state, or that happen in a context that is hard to reproduce outside of a debugger.


See [Interactive Debug Console](#) for details.

Conditional Breakpoints

Since the **Debug Console** is all about working in a selected runtime context, now is a good time to take a look at conditional breakpoints, which are a good way to get the debugger to stop in the context you want to work with.

To set a conditional breakpoint, right-click on the breakpoint margin to the left of the editor and select **Set Conditional Breakpoint**. This brings up a dialog in which you can enter any Python expression. If the expression's truth value is **True**, or if it raises an exception, then the debugger will stop on it. If the expression is not **True** then the debugger will continue running.

Try this now by first selecting **Remove All Breakpoints** from the **Debug** menu and then setting a conditional breakpoint on the `print` within the `for` loop in `PrintAsText`. Use a conditional such as **'beta' in event**. You will need to replace the word `beta` with some other word or fragment to get the debugger to stop here, since this depends on the news items that are currently listed on [python.org](#). Look at the output from your previous runs of `example1.py` to find a word that appears in only one of the news items.

Once this is done, press the  **Restart Debug** icon in the toolbar or select **Restart Debugging** in the **Debug** menu. Wing should stop on your conditional breakpoint in the loop iteration where it is **True**. In more complex code, this would be a quick way to get to the program state that is causing a bug or for which you want to write some new code.

See [Setting Breakpoints](#) for details.

Working in the Editor While Debugging

When the debugger is active, Wing uses both its static analysis of your code and introspection of the live runtime state to offer auto-completion, call tips, and goto-definition in the editor, whenever you are working in code that is active on the debug stack.

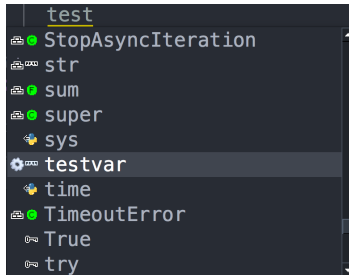
Try this now by typing the following in the **Debug Console**:

```
testvar = 'test'
```

Then switch to `example1.py` and in `PrintAsText` (where you are currently stopped on a conditional breakpoint) create a new line and type this:

```
test
```

Notice that the newly created variable **testvar** shows up in the completer, with a cog icon to indicate that it was found in the runtime state:



This is a handy way to get correct auto-completion in dynamic code where static analysis is not able to find all the symbols that will be defined when code is executed.

9.4. 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.

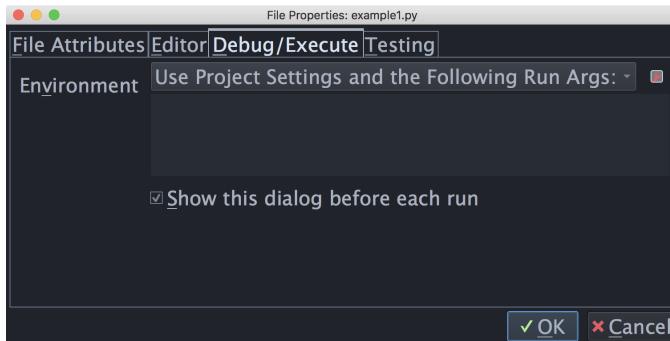
In most cases, **Project Properties** is where you will make changes to the runtime environment for all the project code that you execute and debug.

Try this out now by adding an environment variable **TESTPROJECT=1** to **Environment** in **Project Properties**. Then restart debugging and look at **os.environ** in the **Debug Console** 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 debugged, and optionally to override the project-defined environment on a file by file basis.

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 the 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 Console** 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**. This sets up an environment like that which can be specified in **Project Properties** and pairs it with a particular set of command line arguments. A launch configuration can be reused with multiple files or in **Named Entry Points** (see below).

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 appears, 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 Console**:

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

The output should be:

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

See [File Properties](#) and [Launch Configurations](#) for details.

Main Entry Point

You can specify one file in your project as the main entry point for debugging and execution. When this is set, debugging will always start there unless you use **Debug Current File** in the **Debug** menu.

To set a main entry point use **Set Current as Main Entry Point** in the **Debug** menu, right click on the **Project** tool and select **Set as Main Entry Point**, or use the **Main Entry Point** property under the **Debug** tab of the **Project Properties** dialog.

Try this now by setting **example1.py** as the main entry point. After doing so, it is no longer necessary to bring **example1.py** to front in order to start debugging it.

Whether or not you set a main entry point depends on the nature of your project.

See [Specifying Main Entry Points](#) for details.

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.

See [Named Entry Points](#) for details.

9.5. Tutorial: Debugging from the Python Shell

In addition to launching code to debug from Wing's menu bar and **Debug** menu, it is also possible to debug code that is entered into the **Python Shell** and **Debug Console**.

Enable this now by clicking on the bug icon in the top right of the **Python Shell**. Once this is done, the status message at the top of the **Python Shell** should change to include **Commands will be debugged** and an extra margin is shown in which you can set breakpoints. Wing will reach those breakpoints, as well as any breakpoints in editors for code that is invoked. Any exceptions will be reported in the debugger.

Let's try this out. First stop any running debug process with the **■ Stop** icon in the toolbar. Then paste the following into the **Python Shell** and press **Enter** so that you are returned to the **>>>** prompt:

```
def test_function():
    x = 10
    print(x)
    x += 5
    y = 20
    print(x+y)
```

Next place a breakpoint on the line that reads **print(x)** by clicking in the breakpoint margin to the left of the prompt on that line.

Then type this into the **Python Shell** and press **Enter**:

```
test_function()
```

Wing should reach the breakpoint on **print(x)**.

You can now work with the debugger in the same way that you would if you had launched code from the toolbar or **Debug** menu. Try stepping and viewing the values of **x** and **y** as they change, either in the **Stack Data** tool or by hovering the mouse over the variable names.

Take a look at the stack in the **Call Stack** or **Stack Data** tool to see how stack frames that occur within the **Python Shell** are listed. You can move up and down the stack just as you would if your stack frames were in an editor.

Notice that if you step off the end of the call, you will return to the shell prompt. If you press the **■ Stop** item in the toolbar or select **Stop Debugging** from the **Debug** menu, Wing will complete execution of the code without debug and return you to the **>>>** prompt. Note that the code is still executed to completion in this case because there is no way to simply abandon a number of stack frames in the Python interpreter.

Recursive Debugging

By default Wing will not return you to the **>>>** prompt until your code has finished executing. In Wing Pro, it is possible to enable recursive debugging. This is disabled by default because it can be confusing for users that don't understand it.

To try this out, check the **Enable Recursive Debug** item in the **Options** menu in the **Python Shell**. Then type **test_function()** again in the **Python Shell**, or use the up arrow to retrieve it from command history.

You will see that the shell returns immediately to the **>>>** prompt even though you are now at the breakpoint you set earlier on **print(x)**. The message area in the **Python Shell** indicates that you are debugging recursively and gives you the level to which you have recursed. For example **Debugging recursively (R=2)** indicates two levels of recursive debugging.

Now enter `test_function()` again and then press **Enter**. This is essentially the same thing as invoking `test_function()` from the line at which the debugger is currently paused, in this case within `test_function` itself.

Try doing this several times. Each time, another level of recursive debugging will be entered. Look at the **Call Stack** tool and go up and down the stack to better understand what is happening.

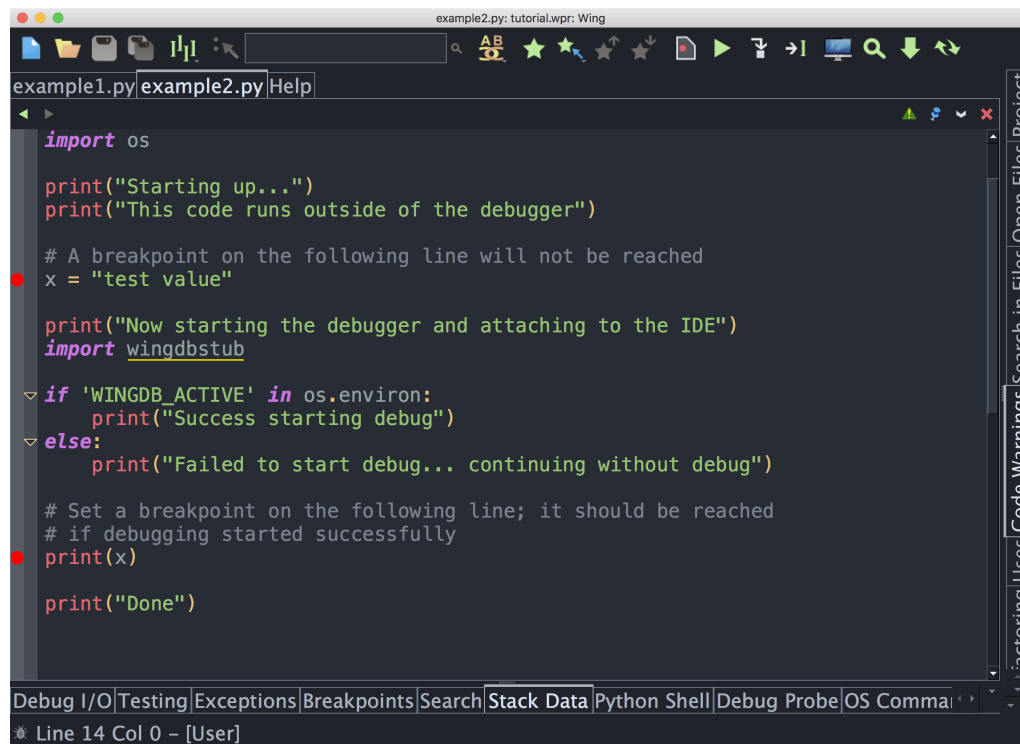
Now if you press **Continue** in the toolbar or use **Start / Continue** in the **Debug** menu you will exit one level of recursion. Similarly, **Stop** exits one level of recursion without debugging the remainder of that recursive invocation.

See [Debugging Code in the Python Shell](#) for details.

9.6. Tutorial: Debugging Code Not Launched by the IDE

So far we've been debugging code launched from inside of Wing. Wing can also debug processes that are running within a web framework, as scripts in a larger application, or that get launched from the command line. These are cases where a debug process cannot be launched from the IDE, so another method is needed to initiate debug.

Let's try this now with `example2.py` in your tutorial directory. First, copy `wingdbstub.py` out of the **Install Directory** listed in Wing's **About** box. Place `wingdbstub.py` in the same directory as `example2.py`. Next, click on the bug icon in the lower left of Wing's main window and select **Accept Debug Connections**. Then set a breakpoint on lines 10 and 22 of `example2.py`:



```
example2.py: tutorial.wpr: Wing
example1.py example2.py Help
import os
print("Starting up...")
print("This code runs outside of the debugger")
# A breakpoint on the following line will not be reached
x = "test value"
print("Now starting the debugger and attaching to the IDE")
import wingdbstub
if 'WINGDB_ACTIVE' in os.environ:
    print("Success starting debug")
else:
    print("Failed to start debug... continuing without debug")
# Set a breakpoint on the following line; it should be reached
# if debugging started successfully
print(x)
print("Done")
Debug I/O Testing Exceptions Breakpoints Search Stack Data Python Shell Debug Probe OS Comma
* Line 14 Col 0 - [User]
```

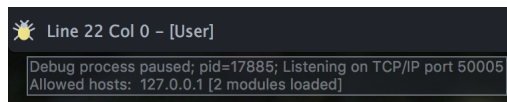
If you are working on macOS, or used the Windows **zip** or Linux **tar** installers for Wing, you will need to edit **wingdbstub.py** in order to set **WINGHOME** to the full path to the **Install Directory** you copied it from. This is done automatically by the other installers. If you are using one 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 macOS to type:

```
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. Notice that the breakpoint on line 10 was not reached because debugging had not yet been initiated at that point. Read through the code and the messages printed to better 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-Tos](#) for help setting this up with specific frameworks and applications.

Remote Debugging

Using the same mechanism, it is also possible to debug Python code launched on another machine, as documented in [Debugging Externally Launched Remote Code](#).

This is part of Wing Pro's ability to work with a remote host through a secure SSH tunnel. This supports all of Wing Pro's features, so you can edit, search, debug, test, and manage remote code in the same way as if it were stored locally, and you can run the **Python Shell** and **OS Commands** on the remote host.

This is the preferred way to work with code on a remote host, although you may still need to use **wingdbstub** to initiate debug if your code cannot be launched from the IDE.

See [Remote Hosts](#) for more information.

9.7. Tutorial: Other Debugger Features

Before moving on, let's look at a few other debugger features that are worth knowing about.

Move Program Counter

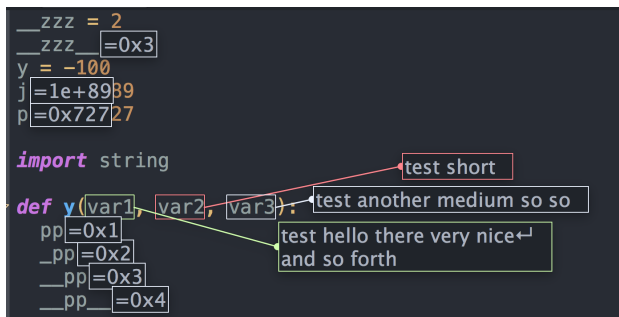
Wing Pro can move the position of the program counter in the innermost stack frame, when the debugger is paused and not at an exception. This is done by right-clicking on the line where the counter should be moved and selecting **Move Program Counter Here**.

This capability allows stepping through already-executed code, by executing it again, so that a problem can be inspected without restarting the debugger.

Due to the way Python is implemented, this is possible only in the innermost stack frame and only if the debugger did not stop at an exception.

Debug Value Tips

Hovering the mouse cursor over symbols in the editor while the debug process is paused will show the value of that symbol, if it is defined in the current stack frame. Similarly, pressing **Shift-Space** will display debug data values for all visible symbols on the editor.



```
__zzz__ = 2
__zzz__ = 0x3
y = -100
j = 1e+89b9
p = 0x72727

import string

def y(var1, var2, var3):
    pp = 0x1
    _pp = 0x2
    __pp = 0x3
    ___pp = 0x4
```

test short

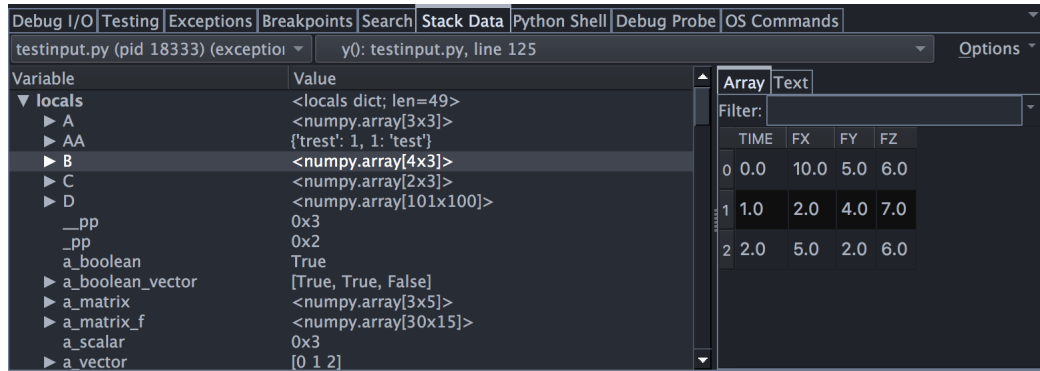
test another medium so so

test hello there very nice
and so forth

See [Viewing Data on the Editor](#) for details.

Viewing Arrays

Selected items in the **Stack Data** tool can be viewed as arrays from the tool's **Options** menu. This works with Python's builtin data types, numpy arrays, pandas DataFrames, and sqlite3 query results, among other things.

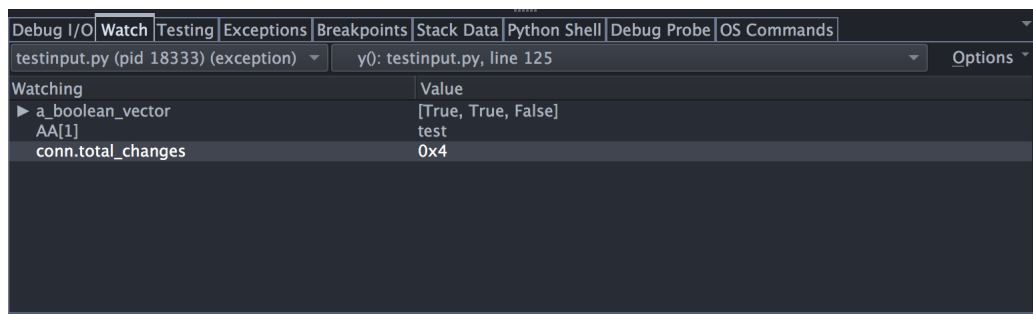


Data is loaded incrementally as needed for display on screen, making this a good way to inspect large datasets without transferring large amounts of data to the IDE.

See [Array and Textual Data Views](#) for details.

Watch Tool

The **Watch** tool lets you watch variables over time by symbolic name or object reference, by right-clicking on them in the **Stack Data**, **Modules**, or **Watch** tools.



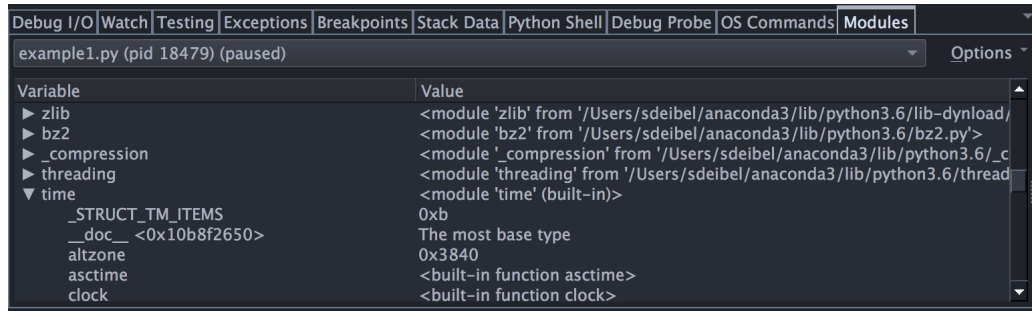
Watching a value by object reference is a great way to inspect an instance while debugging even if you step out of code that contains easily accessible references to it.

You can also watch expressions typed into the **Watching** column of the **Watch** tool.

See [Watching Values](#) for details.

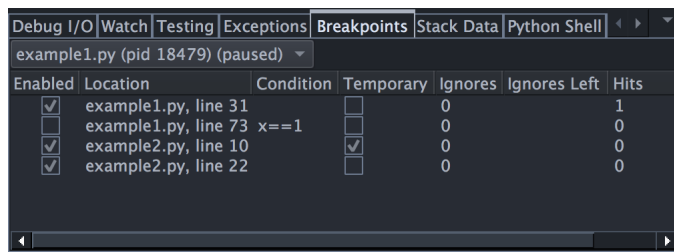
Modules Data View

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 an expandable list of all modules found in `sys.modules`:



Breakpoint Manager

The **Breakpoints** tool shows a list of all defined breakpoints and allows enabling or disabling, editing the breakpoint condition, setting an ignore count, and inspecting the number of times a breakpoint has been reached during the life of a debug process.



See [Setting Breakpoints](#) for details.

Tutorial: Auto-Editing

Let's revisit auto-editing, which was introduced before we tried out the debugger. So far we've seen the editor auto-enter invocation arguments and closing parentheses. There are a number of other auto-editing operations available as well:

Applying Characters to a Selection

If you select a range of text in the editor and press a quote, parenthesis, brace, bracket, or **#**, Wing applies that key stroke to the selection.

For example, try selecting a few lines of non-comment code and press **#**. Wing will comment out those lines using the comment style configured in the **Editor > Block Comment Style** preference. Pressing **#** a second time will remove the comment characters.

Also, selecting some text and pressing **"** (double quote) will surround it with double quotes, or pressing **(** open parenthesis will surround it with parentheses. This also works when typing single quotes, triple quotes, back ticks, brackets, and braces.

Similarly, selecting a string and pressing a different quote character will convert that string to using the type of quote (either single or double quote). This also works if the caret is just after the closing quote of a string, within the opening or closing triple-quote, or one of the quotes is selected.

The `:` colon key can also be applied to a selection, in order to create a new block with one or more selected lines. The `:` is entered, the selected lines are indented following the `:` and the caret is positioned so that the block type can be entered. If `try` is entered, the corresponding `except` is also auto-entered and selected to make it easy to convert it to `finally` or enter the exception type.

These operations are on by default and may be disabled with the **Editor > Auto-editing > Apply Quotes to Selection**, **Editor > Auto-editing > Apply Comment Key to Selection**, **Editor > Auto-editing > Apply [], (), and {} to Selection**, and **Editor > Auto-editing > Apply Colon to Selection** preferences.

Auto-Entering Spacing

Wing can also auto-enter spaces as you type code, optionally enforcing [PEP 8](#) style spacing. This auto-editing operation is off by default but can be turned on with the **Editor > Auto-Editing > Auto-Enter Spaces** preference. Try turning this on now and slowly typing the following into an editor:

```
import os
if os.environ['TEST'] == 'X' * 3:
    pass
```

Notice that Wing is auto-entering a space after the `]`, `=`, and other characters according to the context in the code. If you press the space anyway, it is ignored.

If you also enable the **Editor > Auto-Editing > Enforce PEP 8 Style Spacing** preference, Wing will try to enforce [PEP 8](#) style spacing as you type. For example, typing the following disallows extra spaces around `=`:

```
x = 'test'
```

According to [PEP 8](#), spaces should not be used in argument lists. This is also the default behavior for Wing, whether or not [PEP 8](#) enforcement is on. To override this, enable the **Editor > Auto-Editing > Spaces Around = in Argument Lists** preference.

Similarly, enforcement of spacing around `:` in type annotations can be enabled or disabled with the **Editor > Auto-Editing > Spaces Around : in Type Annotations** preference.

Managing Blocks with the Colon Key

This operation automatically sets up new blocks and allows reindenting existing code under a newly added block. Try this now by typing the following into an editor:

```
if x == 1:
```

Notice that Wing auto-inserts a new line and indentation as soon as the colon is entered, so the contents of the block can be typed right away.

Now try typing the following before `text = None` on line 38 of `example1.py`, inside `ReadPythonNews`:

```
if force:
```

The first time you press `:` Wing shows a status message "Press `:` again to create a new block". This is done because it doesn't know if you are trying to type `if force :=` or just `if force:`. If you press `:` a second time, a new line and indent are added as before.

Next, without moving the caret, press `:` again. Wing will move the first following line `txt = None` under the new block so it looks like this:

```
if force:
    txt = None
if not force and os.path.exists(newscache):
    mtime = os.stat(newscache).st_mtime
    if time.time() - mtime < 60 * 60 * 24:
        f = open(newscache, 'rb')
        txt = f.read()
        f.close()
```

Again without moving the caret press `:` a fourth time. Wing moves the entire following block, up until the next blank line or first line indented less than the current one, so it looks like this:

```
if force:
    txt = None
if not force and os.path.exists(newscache):
    mtime = os.stat(newscache).st_mtime
    if time.time() - mtime < 60 * 60 * 24:
        f = open(newscache, 'rb')
        txt = f.read()
        f.close()
```

If you know you're never going to use `:=` assignments and don't use type hints in block starts then you can enable the **Editor > Auto-Editing > Prefer Block Management Over := and Type Hints** preference and Wing will immediately introduce a new block after you press `:` even in contexts where `:=` or a type hint would be valid.

Line Continuations

If you press **Enter** inside a comment or a string inside `()` and there is text after the caret, Wing auto-continues the line, placing the necessary comment or quote characters. For example, pressing **Enter** after the word `code` on the first line of `example1.py` results in the following:

```
# This is example code
# for use with the Wing tutorial, which
# is accessible from the Help menu of the IDE
```

This is on by default and can be disabled with the **Editor > Auto-Completion > Continue Comment or String on New Line** preference.

Correcting Out-of-Order Typing

Wing also tries to correct out-of-order typing. For example, type the following in an editor:

```
def y(:)
```

Wing figures out that the colon is misplaced and auto-corrects this to read:

```
def y():
```

Similarly, if you type the following:

```
y()x
```

Wing figures out that a `.` is probably missing and auto-corrects this to read:

```
y().x
```

By relying on this, it is possible to save key strokes for caret movement when coding.

This auto-editing operation is on by default and can be disabled with the **Editor > Auto-Completion > Correct Out-of-Order Typing** preference.

See the [Auto-editing](#) documentation page for details.

Tutorial: Turbo Completion Mode

Auto-completion normally requires pressing a completion key, as configured in the **Editor > Auto-completion > Completion Keys** preference, before a completion is entered into the editor.

Wing also provides a **Python Turbo Mode** for auto-completion where where completion occurs on any key that cannot be part of a symbol. It takes some effort to learn to use this feature, but it can greatly reduce typing once you get used to it.

Try it now by enabling the **Editor > Auto-completion > Python Turbo Mode** preference. Then go to the bottom of **example1.py** and press the following keys in order: **R e (G e t (**. You will see the following code in the editor produced by these seven key strokes:

```
ReadPythonNews(GetItemCount())
```

Depending on your Python version you may not need as many keystrokes before **GetItemCount** is selected in the completer. As soon as it is, the final **(** may be pressed.

Turbo completion mode distinguishes between contexts where a new symbol may be defined and those where an existing symbol must be used. For example, try typing the following keystrokes on a new line: **c, =**. Wing knows that the **=** indicates you may be defining a new symbol so it does not place the current selection from the auto-completer. If you did want completion to occur in a defining context, you would still have to press **Tab** or another completion key.

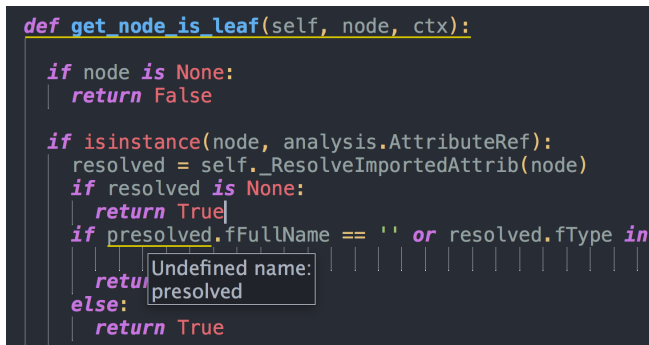
In a context where you are trying to type something other than what is in the completer, pressing **Ctrl**, **Alt** or **Command** briefly by itself will hide the auto-completer and thus disable turbo-completion until you type more symbol characters and the completer is shown again.

Tutorial: Code Warnings

As you probably noticed while working through the tutorial, Wing flags some types of incorrect code by underlining it. This is done for syntax errors, indentation errors, code that can't be reached, undefined variables or attributes, imports that cannot be resolved, and some other types of errors. Hovering the mouse cursor over an indicator on the editor displays details for that warning or error in a tooltip:

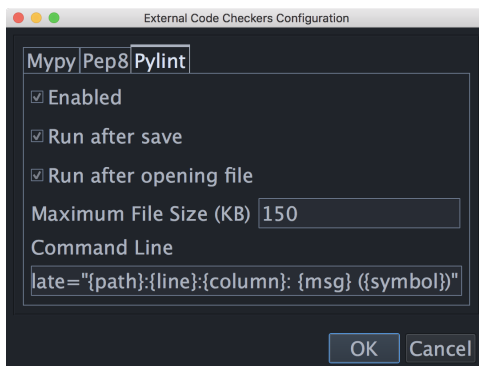
```
def get_node_is_leaf(self, node, ctx):
    if node is None:
        return False

    if isinstance(node, analysis.AttributeRef):
        resolved = self._ResolveImportedAttrib(node)
        if resolved is None:
            return True
        if resolved.fFullName == ' or resolved.fType in
            return resolved
        else:
            return True
```



A **Code Warnings** icon is shown in the top right of any editor that has code warnings, and the **Code Warnings** tool can be used to view and manage the warnings.

The **Code Warnings** tool's **Configuration** tab can be used to set up external sources for code warnings, including ruff, flake8, mypy, pep8, and pylint:



The **Editor > Code Warnings** preference group is used to change the style of the warning indicators on the editor or to globally disable the feature.

See the [Code Warnings](#) documentation for details.

Tutorial: Refactoring

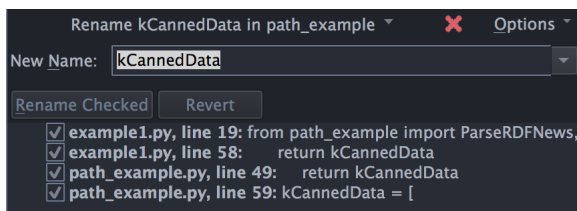
Refactoring is a general term for renaming or restructuring code in a way that does not alter its functionality. It is useful for cleaning up code or to prepare code for easier extension or reuse.

Wing implements a number of refactoring operations. Let's try some of these now in **example1.py**.

Rename Symbol

Click on **kCannedData** in the **import** statement at the top of the file and select **Rename Symbol** from the **Refactor** menu.

Wing will bring up the refactoring tool and enumerate the points of use for the symbol that you have selected:



Now enter **kCannedTuna** as the new name to use and press **Enter** or the **Rename Checked** button. Wing instantly renames all uses of the symbol.

Move Symbol

Now try moving **PromptToContinue** into **subdir/path_example.py** with the **Move Symbol** operation. In the refactoring tool, use **Browse...** to select **subdir/path_example.py** as the target location and leave **Scope** set to **<module global scope>**. Then press **Move & Update Checked**. Wing moves the point of definition into the target file and introduces the necessary **import** so it can still be used from **example1.py**.

Note that the whole module is imported and you would have to manually fix up the import if you instead wished to add the symbol to the existing **from path_example import** statement.

Extract Function/Method

Next select the first larger block in **ReadPythonNews** as follows:

```
newscache = 'newscache.rdf'

txt = None
if not force and os.path.exists(newscache):
    mtime = os.stat(newscache).st_mtime
    if time.time() - mtime < 60 * 60 * 24:
        f = open(newscache, 'rb')
        txt = f.read()
        f.close()

if txt is None:
    try:
        if sys.hexversion >= 0x03000000:
            svc = request.urlopen("http://planet.python.org/
```

Then select the **Extract Function/Method** refactoring operation and enter **ReadNewsCache** as the name for a new top-level function. Wing will create a new function and convert the point of use to a call to that function, as follows, inserting all the necessary arguments and return values:

```
txt = ReadNewsCache(force, newscache)
```

Click on **ReadNewsCache** and use **F4** to visit its point of definition. Then use the history back arrow to get back to the point of use and press **Revert** in the **Refactoring** tool to undo this change.

Try it again now after selecting **Nested Function** instead to see how that operation differs. Then press **Revert** again.

Introduce Variable

Wing can also introduce new variables for an expression. For example, select **time.time() - mtime** in **ReadPythonNews** and use **Introduce Variable** to create a variable called **duration**. Wing inserts the variable and substitutes it into the original expression:

```
txt = None
if not force and os.path.exists(newscache):
    mtime = os.stat(newscache).st_mtime
    duration = time.time() - mtime
    if duration < 60 * 60 * 24:
        f = open(newscache, 'rb')
        txt = f.read()
        f.close()
```

If there had been multiple instances of **time.time() - mtime** in the scope, all of them would have been replaced.

Symbol to *

Several refactoring operations are given to easily convert the name of a symbol between **UpperCamelCase**, **lowerCamelCase**, **under_scored_name**, and **UNDER_SCORED_NAME** naming styles. These work the same way as **Rename Symbol** but prefill the new symbol name field with the selected style of name.

See the [Refactoring](#) documentation for details.

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 **Editor > Indentation > 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.

See [Auto-indent](#) for details.

Block Indentation

One or more selected lines can be increased or reduced in indentation, or adjusted to match indentation according to context, from the **Indentation** toolbar group:



Repeated presses of the **Match Indent** tool will move the selected lines among the possible syntactically correct indent levels for that context. The default action of the **Tab** key does the same thing.

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

Converting Indentation Styles

In Wing Pro and Wing Personal, the **Indentation** tool can be used to analyze and convert the style of indentation found in source files.

See [Indentation Tool](#) for details.

Folding

Unless the feature is disabled with the **Editor > Folding > Enable Folding** preference, Wing Pro and Wing Personal 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's right-click context menu.

Folding acts in such a way that selecting across a fold and copying will copy the text, including its hidden portions.

See [Folding](#) for details.

Other Features

Wing Pro and Wing Personal can also show indentation guides on the editor and set preferred indentation style and policies, among other things. See [Indent Guides, Policies, and Warnings](#) for details.

Tutorial: Other Editor Features

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

Goto-Line

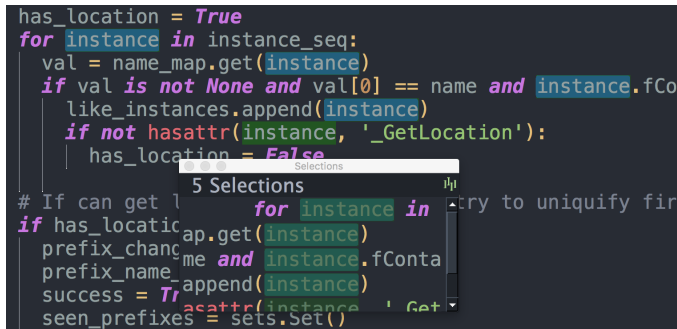
You can navigate quickly to a numbered source line with **Goto Line** in the **Edit** menu, or with the key binding displayed there. Type the line number and then press **Enter** to complete the action.

In Wing Pro and Wing Personal, line numbers can be shown in the editor with the **Show Line Numbers** item in the **Edit** menu.

Multiple Selections

In Wing Pro and Wing Personal, multiple selections can be made on the editor with the **Edit > Multiple Selections** menu items and **⌘ Multiple Selections** in the toolbar.

This provides a quick way to select several identical occurrences of text either sequentially, or within a particular file, class, function, or block:



```
has_location = True
for instance in instance_seq:
    val = name_map.get(instance)
    if val is not None and val[0] == name and instance.fCo
        like_instances.append(instance)
        if not hasattr(instance, '_GetLocation'):
            has_location = False
# If can get l
if has_locatic
    prefix_chang
    prefix_name
    success = Tr
    seen_prefixes = sets.Set()
```

Once there are multiple selections, edits made will be applied to all the selections concurrently.

Multiple selections may also be made by pressing **Ctrl+Alt** (or **Command+Option** on the Mac) while making a selection with the mouse.

Selection Mode and Structural Code Selection

Wing supports character, line, and block mode selection from **Selection Mode** in the **Edit** menu, and the key bindings shown there.

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-Up** will select more code in logical units. Press **Ctrl-Down** 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.

See [Selecting Text](#) for details.

Line Editing

In Wing Pro and Wing Personal, 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 add your own key bindings with the **User Interface > Keyboard > Custom Key Bindings** preference. 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**.

Code Snippets

In Wing Pro, 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.

You may already have noticed that these appear in Wing's auto-completer. Try this now by typing **def** into the top level of a file in the editor. Then select the **def (snippet)** completion choice. Wing will place the snippet into the editor and enter into a data entry mode similar to the mode used for entering arguments when the **Auto-Enter Invocation Arg** auto-editing operation is enabled. Type any text you want in each field within the snippet and press **Tab** to move between the fields. Data-entry mode will end at the last tab stop or if you move out of the snippet body.

Now try it again with **class** and then inside the scope of the class use the **def** snippet again. Notice that the form of snippet in this context differs from the one used at the top level; it includes **self**. Like-named snippets can be defined in this way for the following contexts: **module**, **class**, **function**, **method**, **attribute** (after a period), **comment**, and **string**.

See [Code Snippets](#) for details.

Block Commenting

Lines of code can be commented out or un-commented quickly from the **Source** menu or, in Wing Pro, by pressing the **#** key while several lines of Python code are selected. In Python code, the **Editor > 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 matches as you type, unless this is disabled from the **Editor > Brace Matching > Brace Highlighting** preference. The **Match Braces** item in the **Source** menu causes Wing to select all the code that is contained in the nearest matching braces, as 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 column configured in the preference **Editor > Line Wrapping > Reformatting Wrap Column** with the **Rewrap Text** item in the **Source** menu. This will limit wrapping to a single logical line of code, so it can be used to reformat function or method arguments or long list or tuple without altering surrounding code.

Bookmarks

In Wing Pro, you can define and jump to marked locations in the editor with the bookmarking commands in the **Source** menu, the editor's right-click context menu, and the bookmark toolbar group.

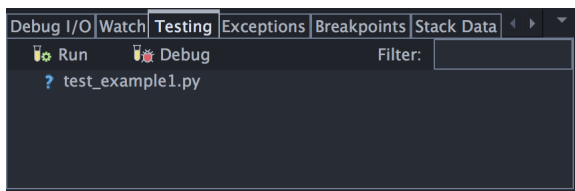
In Python files, these bookmarks are defined relative to the named scope in the file so they move around with the scope as the file is edited outside of Wing. Bookmarks can be categorized and managed in the **Bookmarks** tool in the **Tools** menu.

See [Bookmarks](#) for details.

Tutorial: Unit Testing

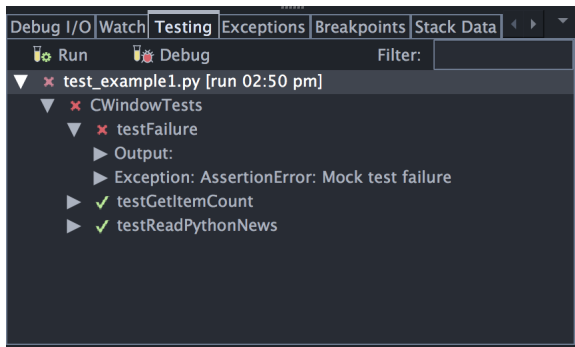
Wing Pro's **Testing** tool makes it easy to run and debug units tests written for the **unittest**, **doctest**, **pytest**, **nose**, and **Django** unit testing frameworks.

Let's try this out now. First, open up **Project Properties** and under the **Testing** tab insert a **Test File Pattern** that is set to **Glob / Wildcard** and **test_*.py**. This tells Wing which of your project files are unit test files. Press **OK** or **Apply** and bring up the **Testing** tool from the **Tools** menu. This should now contain an entry for the file **test_example1.py**:



Next comment out the line that reads **PromptToContinue** in **example1.py** so that the module can be loaded by the tests without prompting.

Then press **Run Tests** in the **Testing** tool. You should see two of the three tests pass, and one will fail:



You can expand the tree to see details of the failed tests, including any output printed by the test and the exception that occurred. Double-clicking on the test results and exception will take you to the relevant code.

Note that you can also run tests from the editor by clicking on the test you want to run and selecting **Run Tests at Cursor** from the **Testing** menu.

Debugging Tests

Now run the failed **testFailure** in the debugger by clicking on it in the **Testing** tool and pressing **Debug Tests** at the top of the tool. Wing should stop at the exception and you can use the debugger on the test as you would for any other Python code.

Environment

When unit tests are run in the **Testing** tool, by default they run in the same environment that is used for debugging and executing code. This can be changed with **Environment** under the **Testing** tab of **Project Properties** or in the **File Properties** for the unit test file.

See [Unit Testing](#) for details.

Tutorial: Version Control Systems

Wing Pro includes revision control integrations for **Git**, **Mercurial**, **Subversion**, **Perforce**, **Bazaar**, and **CVS**. These auto-enable based on the contents of your project and provide the most commonly used operations such as commit, status, log, diff, pull, revert, and update. The set of operations supported varies for each version control system.

If you have a code base that is in revision control you might want to try this out now, by adding your code to the tutorial project with **Add Existing Directory** in the **Project** menu.

Once that is done, Wing should auto-detect the revision control system and add a menu to the menu bar. You can now select **Project Status** from that menu or use the **Tools** menu to bring up the appropriate revision control tool. Use the menu or right-click on the tool to initiate operations.

If you have a project with files in multiple revision control systems, or if you want to keep a particular system active at all times, you can do this from the **Version Control** preference group.

See [Version Control](#) for details.

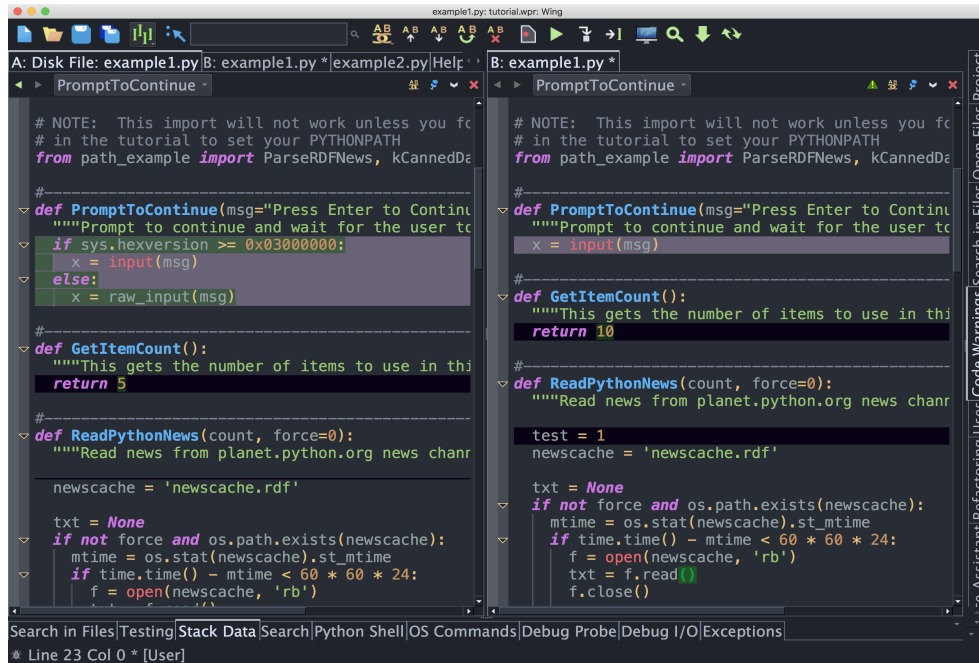
Difference and Merge

When a revision control system is active, you can compare the working copy of your code to the repository revision it is based on by right-clicking on items in the revision control tool or on the editor or **Project** tool.

This tool can also be used to compare two files, two directories, and an unsaved file with the disk.

Try it now by making several changes to **example1.py** without saving them to disk. Then click on the  **Difference/Merge** icon in the toolbar to select **Compare Buffer With Disk**.

Wing splits the editor area to show two editors side by side and shows additional icons in the toolbar to control the difference and merge session:



Use the **Previous Difference** and **Next Difference** items to move through the differences and the **Merge A->B** tool to undo unsaved changes.

When comparing directories, Wing shows the **Diff/Merge** tool while the session is active, and highlights the current file being compared as you move through the session. You can also click on files in this tool to move to a specific comparison.

Difference/Merge is particularly useful for reviewing and undoing any unwanted changes before committing to a revision control repository.

See [Difference and Merge](#) for details.

Tutorial: Searching

Wing Pro provides several different interfaces for searching your code. Which you use depends on what you want to search and how you prefer to interact with the search and replace functionality.

18.1. Tutorial: Toolbar Search

A quick way to search through the current editor or documentation page is to enter your search string into the search area provided by the toolbar:



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

Try this now in **example1.py**: Type **GetItem** into the toolbar search area. Wing will search incrementally, starting when the first letter is typed. 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 in the file from the current editor caret position.

See [Toolbar Quick Search](#) for details.

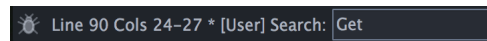
18.2. Tutorial: Keyboard-Driven Search and Replace

If you prefer a more powerful search interface using the keyboard only, try the key bindings listed in the **Mini-search** sub-menu of the **Edit** menu. The bindings vary according to the currently selected [Keyboard Personality](#).

Mini-search supports searching forward and backward in the current editor, documentation, **Python Shell**, **Debug Console** (Wing Pro only), or **Debug I/O** tool, 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 you are using the default keyboard personality, press **Ctrl-U**. For other keyboard personalities, refer to **Mini-search** 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 key press.

Search Behavior

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

While the mini-search area is still active, try pressing the same key combination you used to display it again. Wing will search for the next matching occurrence.

If no match is found **Failed Search** will be displayed. After this, pressing the mini-search key combination again will wrap around and start searching at the top of the file, if there are any matches.

To start searching again using the most recently used search string, press the key combination for search twice, once to display the search entry area, and once again to recall the previous search string.

Search direction can be changed during a search session, by switching to the key bindings assigned to the desired direction.

You can exit from the search by pressing the **Esc** key or **Ctrl-G**, or with arrow keys and other editor commands.

Regular Expression Search

The regular expression search options found in **Mini-search** in the **Edit** menu work similarly but expect regular expressions for the search criteria.

Replace

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.

Two replace operations are available. Both of these operate only on text that follows the caret in the file and do not wrap:

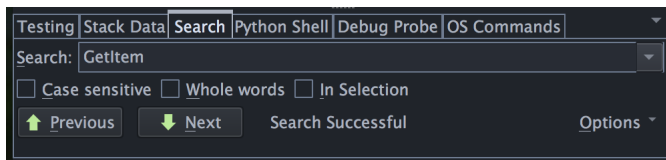
Query/Replace prompts for **Y** and **N** for each replace location

Replace String replaces all following matches in the file without prompting.

See [Keyboard-Driven Search and Replace](#) for details.

18.3. Tutorial: Search Tool

The **Search** tool provides simple search and replace operations on the current editor or documentation page. Key bindings for operations on this tool are given in the **Search and Replace** group in the **Edit** menu.

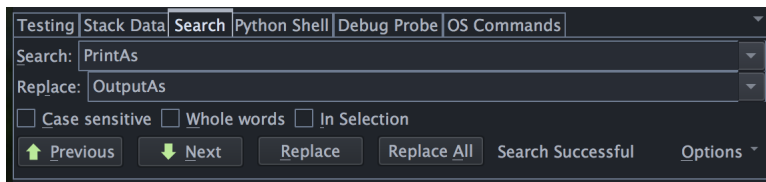


Searches may span the whole file or be constrained to the current selection, can be case sensitive or insensitive, and may optionally be constrained to matching only whole words.

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

Replacing

When the tool is displayed with **Replace**, or when the **Show Replace** item in the **Options** menu 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.

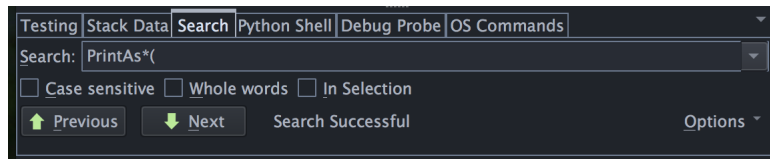
Next, try **Replace All** instead. Wing will simply replace all occurrences in the file at the same time. When this is done, a single undo in the editor will cancel the entire replace operation.

Wildcard Searching

By default, Wing searches for straight text matches on the strings you type. In Wing Pro and Wing Personal, wildcard and regular expression searching are also available in the **Search** tool's **Options** menu.

Wildcard searching 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 [Wildcard Search Syntax](#).

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



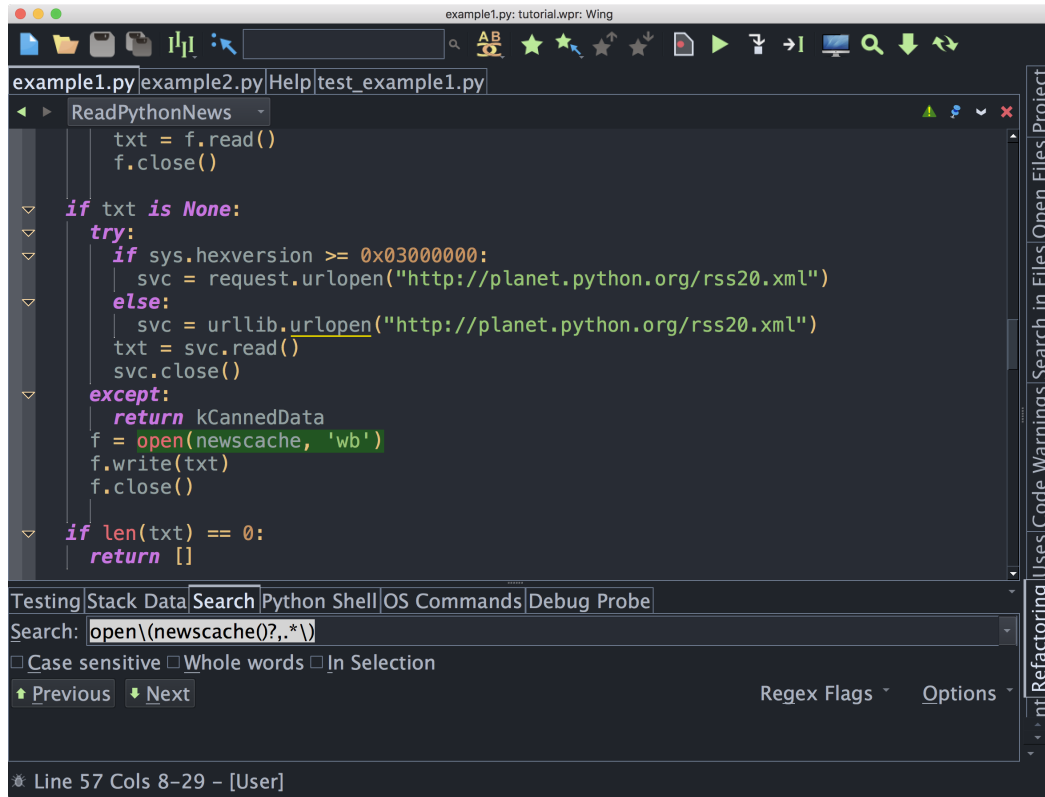
Also try searching on **PrintAs*[A-Z](** with the **Case Sensitive** search option turned on. This matches all strings starting with **PrintAs** followed by zero or more characters, followed by any capital letter from **A** to **Z**, followed by **(**.

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

Regular Expression Search

Regular expressions are most useful for complex 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** after selecting **Regex search** from the **Options** menu then you should get exactly one search match:



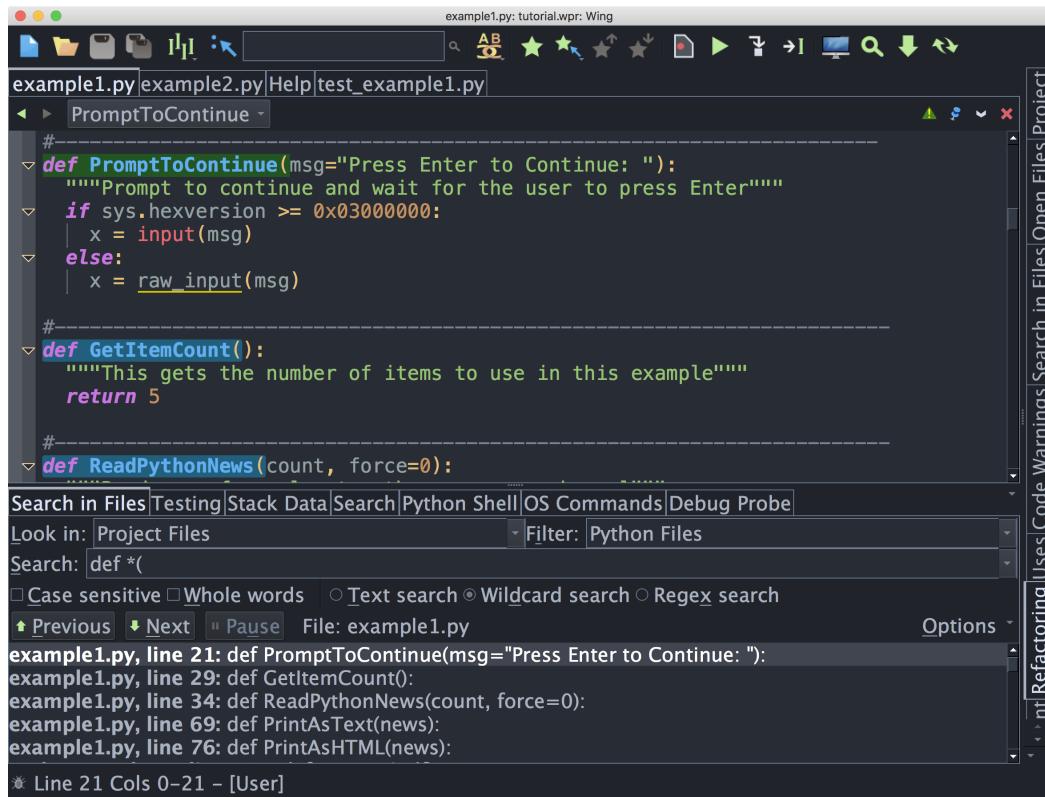
In this mode, the replace string can reference regex match groups with `\1`, `\2`, etc, as in the Python `re.sub()` call.

The details of regular expression syntax and usage see [Regular Expression Syntax](#) in the Python manual.

See also [Search Tool](#) for more information.

18.4. Tutorial: Search in Files

The **Search in Files** tool in Wing Pro and Wing Personal supports multi-file batch search of the disk, project, open editors, documentation, or other sets of files. It can also search and replace using wildcards and regular expressions.



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, or if the files being searched change.

When you are done typing, you should see results similar to those shown in the screen shot above. Click on the first result line to display **example1.py** in the editor with the corresponding search match highlighted.

Next, change the **Look in** selector to **Project Files** and change your search string to **HTML**. This works the same way as searching a single file, but lists the results for all files that have been added to your project.

You can also search all currently open files or within Wing's documentation by instead selecting **Open Files** or **Documentation** from **Look in**.

File Filters

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 Ca
	C/C++ Files	Mime Type: text/x-c-source; Mime Typ
	HTML and XML Files	Mime Type: text/html; Mime Type: text
	Hidden & Temporary Files	Wild Card on File Name: *.pyo; Wild Car
	Key Maps	Wild Card on File Name: *.pyo; Wild Car

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. An example would be to specify ***.js** wildcard for matching Javascript 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 enabling **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.

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 replacing within multiple files, 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, 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 are using a revision control system that can be used to undo mistakes.

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 global replace operation.

See [Multi-File Search and Replace](#) for details.

Tutorial: Other IDE Features

By now you have seen many of the IDE's features. Before we call it a day, let's look at a few other major features that are worth knowing about.

Remote Development

Wing Pro makes it possible to work with Python source code that resides on a remote host, container, or virtual machine. This is done by setting up SSH access to the remote host, then configuring Wing using **Remote Hosts** in the **Project** menu, and pointing **Python Executable** in **Project Properties** at that remote host.

Once this is done, remote files and directories can be added to the project with **Add Existing Directory** in the **Project** menu. Then Wing will be able to edit, debug, test, search, inspect, refactor, and manage remote files, and it can run **Python Shell** and **OS Commands** on the remote host.

See [Remote Hosts](#) for details.

Containers

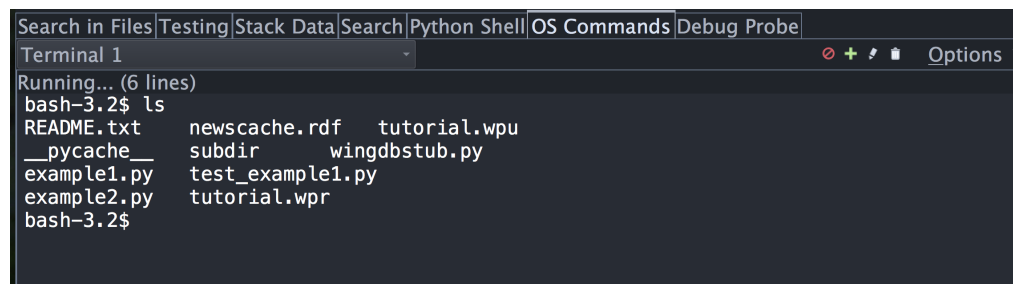
Wing Pro also supports containerized development using Docker and other container systems. This mode of development stores and works with files on the local disk but starts debug, unit tests, and other processes in the container environment.

To work with [Docker](#), use the **New Project** dialog from the **Project** to set up a project for an existing Docker container, or create of a new project together with a new Docker container.

For other containers, set **Python Executable** in **Project Properties** to **Container** and create a container configuration. See [Containers](#) for details.

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 the **Debug/Execute** tab of **Project Properties** can be used to configure and select one command to execute automatically before any debug session begins. OS Commands can also be bound to keys, and the **Start Terminal** item in the **Tools** menu uses it to start command prompt in Wing:

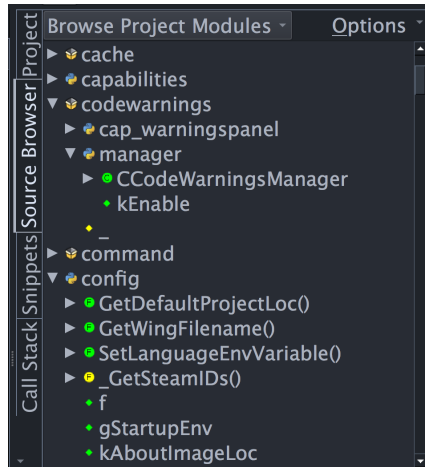


```
Search in Files | Testing | Stack Data | Search | Python Shell | OS Commands | Debug Probe
Terminal 1
Running... (6 lines)
bash-3.2$ ls
README.txt      newscache.rdf  tutorial.wpu
__pycache__    subdir        wingdbstub.py
example1.py     test_example1.py
example2.py     tutorial.wpr
bash-3.2$
```

See [OS Commands Tool](#) for details.

Source Browser

The **Source Browser** in Wing Pro and Wing Personal can be used to inspect and navigate the module and class structure of your Python source code.



Double-clicking on items in the **Source Browser** opens them into an editor. When **Follow Selection** is enabled in the **Options** menu, Wing also opens files that are single-clicked or visited by keyboard navigation within the **Source Browser**.

The popup menu at the top left of the **Source Browser** selects whether to browse the current file, all project modules, or all project classes. The **Options** menu in the top right allows sorting and filtering symbols by type.

Notice that the **Source Assistant** tool is integrated with the **Source Browser**, and will update its content as you move around the **Source Browser**.

File Sets

Wing allows you to create named sets of files which you can open as a group or search in the **Search in Files** tool. See [File Sets](#) for details.

File Operations

Files can be created, deleted, moved, and renamed from the **Project** tool by right-clicking, dragging, and clicking on names in the tree. Deleted files are moved to the system's trash or recycling bin.

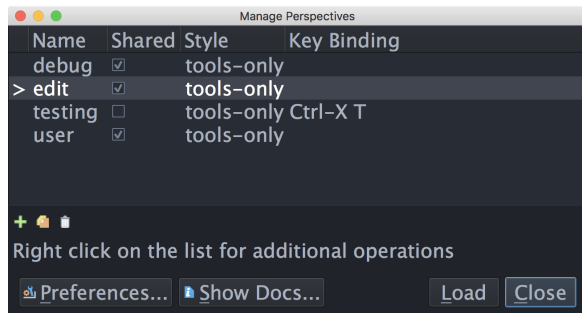
When files are in a revision control system, Wing will also issue the necessary revision control commands to create, delete, move, or rename the file in the repository.

See [File Operations](#) for details.

Perspectives

Perspectives are a way to store and later revisit particular arrangements of the user interface. For example, you may set up one set of visible tools to use when testing, another for working on documentation, and still another for debugging.

Perspectives are accessed from the **Tools** menu.



Wing can optionally switch perspectives automatically whenever debugging starts or stops, so that the user interface differs according to how the tools were left when last editing or debugging. This is done by selecting **Enable Auto-Perspectives** in the **Tools** menu.

See [Perspectives](#) for details.

Command Set and Key Bindings

Wing's complete command set is documented in the [Command Reference](#). Any of these commands can be bound to a key binding with the **User Interface > Keyboard > Custom Key Bindings** preference. A key binding may be a single chord such as **Shift-Ctrl-X** or a sequence like **Ctrl; A**.

To invoke a command directly even if it does not appear in a menu or toolbar item, use the **Command by Name** item in the **Edit** menu.

The default key bindings are documented in [Key Binding Reference](#). You can check what command a key is bound to using the command **describe-key-briefly**, also invoked from **Command by Name** in the **Edit** menu.

Extending the IDE

Wing can be extended by writing Python scripts that call into the IDE's scripting API. This is useful for adding everything from simple editor commands and debugger add-ons to new tools.

Script-defined commands may be bound to keys, added to menus, or invoked from the toolbar just like Wing's built-in commands.

There is a collection of user-contributed scripts for Wing in the [contributed materials](#) area.

For details see [Scripting and Extending Wing](#).

Tutorial: Further Reading

Congratulations, you've finished the tutorial!

As you work with Wing Pro on your own software development projects, the following resources may be useful:

[How-Tos](#) with instructions for using Wing with third party frameworks, applications, and tool, like Django, Jupyter, matplotlib, Autodesk Maya, Raspberry Pi, pygame, and many others.

[Wing Reference Manual](#) which documents all the features in detail.

[Wing Support Website](#) which includes a Q&A support forum, mailing lists, documentation, links to social media, and other information for Wing users.

A collection of [Wing Tips](#), available on our website and by weekly email subscription, provides additional tips and tricks for using Wing productively.

Thanks for using Wing Pro!