*Spyder* is an Integrated Development Environment (IDE) for scientific computing using the Python programming language. It comes with an Editor to write code, a Console to evaluate it and see its results at any time, a Variable Explorer to see what variables have been defined during evaluation, and several other facilities to help you to effectively develop the programs you need as a scientist.

This tutorial is authored by Hans Fangohr from the University of Southampton (UK) (see historical note for more detail).

## Outline

## First steps with Spyder

This section is aimed at Python and Spyder beginners. If you find it too simple, please continue to the next section.

### Execute a given program

- We are going to use this program as a first example:

```
# Demo file for Spyder Tutorial
# Hans Fangohr, University of Southampton, UK

def hello():
    """Print "Hello World" and return None"""
    print("Hello World")

# main program starts here
hello()
```

- To use this program, please create a new file in the Spyder editor pane. Then copy and paste the code inside the box above on the file, and the save it with the name hello.py.
- To execute the program, select Run > Run from the menu (or press F5), and confirm the Run settings if required.

If this is your first time, you should see an output like this:

```
In [1]: runfile('/Users/fangohr/Desktop/hello.py', wdir=r'/Users/fangohr/Desktop')
Hello World

In [2]:
```

If so, then you have just run your first Python program – well done.

> **Note**
>
> The particular path shown next to `runfile` will depend on where you have saved the file, but this is inserted by Spyder automatically.

#### Use the IPython Console

Before we proceed, we recommend you to use the IPython console. This console can do a little more than the standard Python console, and we suggest to use it as the default console here.

#### What happens when you execute the program?

- Python reads the file line by line, ignoring comments (i.e. lines starting with the `#` symbol).
- When it comes across the `def` keyword, it knows that a function is DEFined in this and the next (one or more) lines. All *indented* lines following `def hello():` belong to the function body.

  Note that the function object is just created at this point in the file, but the function is not yet called (i.e. not executed).

- When Python comes across commands (other than `def ...` and a few other keywords) that are written in the left-most column, it will execute these immediately. In the `hello.py` file this is only the line reading `hello()` which will actually call (i.e. *execute*) the function with name `hello`.

  If you comment or remove the line `hello()` from the program and run the whole file again (by pressing F5, or selecting `Run > Run`), nothing will be printed (because the function `hello` is defined, but not called, i.e. not executed).

Now you should know how to execute a Python program that you have in the editor pane in Spyder using the IPython console.

If you are just starting to learn Python, this is probably a good point to return to your text book / course and look at more basic examples.

The next section gives more detailed information how you can execute *parts* of the code in the editor in the IPython console, and thus

update parts of your definitions in the editor. This is a more advanced technique but can be very useful. (You may also be interested in the option to execute chunks (so-called "cells") of code that are separated by delimiters – see Shortcuts for useful functions.)

#### Call existing functions in the console

Once you have executed the hello.py program, the function object hello is defined and known to the IPython console. We can thus call the function from the console like this:

- Type hello() in the console (next to In [?] prompt, where the question mark can be any positive integer number), and press the Enter key.

  You should find that the hello() function is executed again, i.e. Hello World is printed again. Your function call at the console together with the output should look like this:

```
In [ ]: hello()
Hello World
```

- Can you see how this differs from executing the whole program again?

  When we execute the whole program (by pressing F5), Python goes through the file, creates the hello function object (overriding the previous object), reaches the hello() line and calls the function.

  When we call hello() in the console, we only call the function object hello that has been defined in the IPython console when we executed the whole hello.py file earlier (by pressing F5).

  This will become clearer over time and also when we work with slightly larger examples. You may want to return to this tutorial at a slightly later stage.

#### Inspecting objects defined in the console

- Python provides a function that displays all known objects in the current name space of the console. It is called dir(): when you type dir() at the console, you get a list of known objects. Ignore everything starting with an underscore for now. Can you see hello in the list?

> **Note**
>
> If you get a long list of defined objects, then Spyder may have done some convenience imports for you already. To address this you may want to:
>
> - Reset the name space
> - Execute hello.py again by pressing F5
>
> Then run dir() as suggested above.

- Once an object is visible in the current name space (as is hello in this example), we can use the help function as follows to learn about it: Typing help(hello) at the console prompt, you should see an output like this:

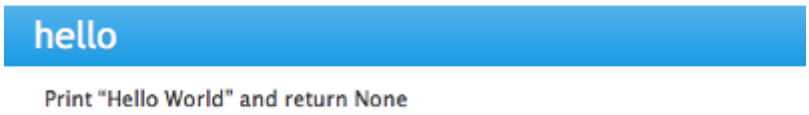
```
In [ ]: help(hello)
Help on function hello in module __main__:

hello()
    Print "Hello World" and return None
```

Where does Python take the information from? Some of it (like the number of input arguments and names of those variables; here we have no input arguments) Python can find through inspecting its objects, additional information comes from the documentation string provided for the function object hello. The documentation string is the first string immediately below the line def hello():.

This strings are special, and they are called *docstrings* which is short for *documentation strings*. As they usually extend over multiple lines, there are enclosed by triple single quotes (''') or triple double quotes (""").

- The Spyder environment also provides the Object inspector which by default is located in the top right corner.

  While the cursor is on the name of an object, press CTRL+i (or CMD+i on Mac), and you should find that the same information as we obtained from help(hello) is provided automatically in the object inspector:




This works in the console and in the editor.

### Updating objects

#### Simple strategy: re-execute whole program

- In the Editor window, change the function `hello` so that it prints `Good Bye World` rather than `Hello World`.
- Press F5 (to execute the whole program) and check that the output of the program is now:

```
Good Bye World
```

What has happened when you pressed F5 is this: Python has gone through the `hello.py` file and created a new function object `hello` (overriding the function object `hello` we had defined before) and then executed the function.

#### Looking at the details

We need to start with a clearly defined state. To do this, please change the function `hello()` back so that it prints `Hello World`, then press F5 to run the whole program and check that it prints `HelloWorld`.

- Call the function `hello()` from the command prompt (as described in Call existing functions in the console). You should see `Hello World` printed.
- Now change the function definition so that it would print `Laters World`, and save the file (but do NOT execute the program, i.e. do NOT press F5 yet).
- Call the function `hello()` in the console again. You should find that the text printed reads `Hello World`, like here

```
In [ ]: hello()
Hello World
```

Why is this so? Because the hello function object in the console is the old one which prints Hello World. So far, we have changed the file hello.py (and replaced Hello World in there with LatersWorld) in the editor but this has not affected the objects that have previously been created in the console.

Here are two possibilities to use our modified version of the hello function:

- Option 1: execute the whole file hello.py again by pressing F5: this creates a new function object hello (and overrides the old one). You should find that if you press F5, and then call hello() at the prompt, the new text Laters World is printed.
- Option 2: select the region you have changed (in this case the whole function hello, starting from the line def hello(): down to print("Laters Wold"), and then select Run > Run selection.

This will update the hello object in the console without having to execute the whole hello.py file:

```
In [ ]: def hello():
   ...:     """Print "Hello World" and return None"""
   ...:     print("Laters world")
   ...:
```

If we now type hello(), we see the update response:

```
In [ ]: hello()
Laters world
```

The ability to execute *parts of the code* to update some objects in the console (in the example above, we updated the function object hello), is of great use when developing and debugging more complex codes, and when creating objects/data in the console session take time. For example, by modifying only the functions (or classes/objects, etc) that we are actually developing or debugging, we can keep re-using the data and other objects that are defined in the console session.

To teach and learn Python programming, we recommend here to use IPython instead of the normal Python console. This accepts IPython as the de-facto standard in the scientific Python community.

### Switch to an IPython console

If you already have an IPython console active, you can ignore this section, and make it visible by clicking on the "IPython console" rider.

In the console window (lower right corner by default), you see by default a prompt with three greater than signs, i.e. >>>. This shows that we are using the console - basically a normal Python console session (with some added functionality from Spyder).

Instead, we would like to use an *Interactive Python* console, short *IPython* from the IPython project. To do this, select Consoles > Open an IPython Console.

You should see in the consolse window a new shell appearing, and the IPython prompt In [1]: should be displayed.

### Reset the name space

The name space (i.e. the collection of objects defined in the console at any given time) can be cleared in IPython using the %reset command. Type %reset and press return, then confirm with y:

```
In [1]: %reset

Once deleted, variables cannot be recovered. Proceed (y/[n])? y

In [2]:
```

That's all.

We discuss this a little further, but you can skip the following if you are not interested: After issuing the %reset command, we should have only a few objects defined in the name space of that session. We can list all of them using the dir() command:

```
In [2]: dir()
Out[2]:
['In',
 'Out',
```

```
 '__builtin__',
 '__builtins__',
 '__name__',
 '_dh',
 '_i',
 '_i2',
 '_ih',
 '_ii',
 '_iii',
 '_oh',
 '_sh',
 'exit',
 'get_ipython',
 'help',
 'quit']
```

Finally, if you like to skip the confirmation step of the reset command, use can use %reset -f instead of %reset.

#### Strive for PEP8 Compliance

In addition to the syntax that is enforced by the Python programming language, there are additional conventions regarding the layout of the source code, in particular the Style Guide for Python source code known as "PEP8". By following this guide and writing code in the same style as almost all Python programmers do, it becomes easier to read, and thus easier to debug and re-use – both for the original author and others.

You should change Spyders settings to Warn if PEP8 coding guidelines are violated.

## Selected Preferences

#### Where are the preferences?

A lot of Spyder's behaviour can be configured through it's Preferences. Where this is located in the menu depends on your operating system:

- On Windows and Linux, go to Tools > Preferences
- On Mac OS, go to Python/Spyder > Preferences

#### Warn if PEP8 coding guidelines are violated

Go to Preferences > Editor > Code Introspection/Analysis and select the tickbox next to Style analysis (PEP8)

### Automatic Symbolic Python

Through Preferences > IPython console > Advanced Settings > Use symbolic math we can activate IPython's SYMbolic PYthon (sympy) mode that is provided by the sympy module. This mode in Spyder allows nicely rendered mathematical output (latex style) and also imports some sympy objects automatically when the IPython console starts, and reports what it has done.

```
These commands were executed:
>>> from __future__ import division
>>> from sympy import *
>>> x, y, z, t = symbols('x y z t')
>>> k, m, n = symbols('k m n', integer=True)
>>> f, g, h = symbols('f g h', cls=Function)
```

We can then use the variables x, y, for example like this:

## Shortcuts for useful functions

- F5 executes the current file
- F9 executes the currently highlighted chunk of code: this is very useful to update definitions of functions (say) in the console session without having to run the whole file again. If nothing is selected F9executes the current line.
- Tab auto-completes commands, function names, variable names, methods in the Console (both Python and IPython) and in the Editor. This feature is very useful, and should be used routinely. Do try it now if auto-completion is new to you. Assume you have defined a variable:

```
mylongvariablename = 42
```

Suppose we need to write code that computes mylongvariablename + 100, we can simply type my and then press the Tab key. The full variable name will be completed and inserted at the cursor position if the name is unique, and then we can carry on and type + 100. If the name is not uniquely identifiable given the letters my, a list field will be displayed from which the right variable can be chosen. Choosing from the list can be done with the <Arrow up> key and <Arrow down> key and the Enter key to select, or by typing more letters of the name in question (the selection

will update automatically) and confirming by pressing Enter when the right name is identified.

- Ctrl+Enter executes the current cell (menu enty Run > Run cell). A cell is defined as the code between two lines which start with the agreed tag #%%.
- Shift+Enter executes the current cell and advances the cursor to the next cell (menu entry Run > Run cell and advance).

  Cells are useful to execute a large file/code segment in smaller units. (It is a little bit like a cell in an IPython notebook, in that chunks of code can be run independently.)

- Alt+<Up Arrow> moves the current line up. If multiple lines are highlighted, they are moved up together. Alt+<Down arrow> works correspondingly moving line(s) down.
- Ctrl+Left Mouse Click on a function/method in the source, opens a new editor windows showing the definition of that function.
- Shift+Ctrl+Alt+M maximizes the current window (or changes the size back to normal if pressed in a maximized window)
- Ctrl+Shift+F activates the search pane across all files.
- Cmd + + (On MacOS X) or Ctrl + + (otherwise) will increase the font size in the Editor, whereas Cmd + - (Ctrl + -) will decrease it. Also works in the IPython Console.

  The font size for the Object Inspector, the Python console etc can be set individually via Preferences > Object inspector etc.

  I couldn't find a way of changing the font size in the variable explorer.

- Cmd+S (on MacOS X) and Ctrl+S (otherwise) *in the Editor* pane saves the file currently being edited. This also forces various warning triangles in the left column of the Editor to be updated (otherwise they update every 2 to 3 seconds by default).
- Cmd+S (on MacOS X) and Ctrl+S (otherwise) *in the IPython console* pane saves the current IPython session as an HTML file, including any figures that may be displayed inline. This is useful as a quick way of recording what has been done in a session.

  (It is not possible to load this saved record back into the session - if you need functionality like this, look for the IPython Notebook.)

- Cmd+I (on Mac OS X) and Ctrl+I (otherwise) when pressed while the cursor is on an object, opens documentation for that object in the object inspector.

## Run Settings

These are the settings that define how the code in the editor is executed if we select Run > Run or press F5.

By default, the settings box will appear the first time we try to execute a file. If we want to change the settings at any other time, they can be found under Run > Configure or by pressing F6.

There are three choices for the console to use, of which I'll discuss the first two. Let's assume we have a program hello.py in the editor which reads

```
def hello(name):
    """Given an object 'name', print 'Hello ' and the object."""
    print("Hello {}".format(name))


i = 42
if __name__ == "__main__":
    hello(i)
```

### Execute in current Python or IPython console

This is the default suggestion, and also generally a good choice.

#### Persistence of objects I (after code execution)

#### Persistence of objects II (from before code execution)

### Execute in new dedicated Python console

Choosing Execute in new dedicated Python console under Run > Configure will start *a new Python console everytime* the hello.py program is executed. The major advantage of this mode overExecute in current Python or IPython console is that we can be certain that there are no global objects defined in this console which originate from debugging and repeated execution of our code: every time we run the code in the editor, the python console in which the code runs is restarted.

This is a safe option, but provides less flexibility and cannot use the IPython console.

#### How to double check your code executes correctly "on its own"

Assuming you have chosen for your code to Execute in current Python or IPython console, then you have two options to check that your code does work on its own (i.e. it does not depend on undefined variables, unimported modules and commands etc.)

1. Switch from Execute in current Python or IPython console to Execute in new dedicated Python console, and execute the code in the editor in this dedicated Python console.

   Alternatively, if you want to stay with the current IPython console, you can

2. Use IPython's magic `%reset` command which will remove all objects (such as `i` in the example above) from the current name space, and then execute the code in the editor.

#### Recommendation

My recommendation for beginners would be to Execute in current Python or IPython console, *and* to choose the IPython console for this.

Once you have completed a piece of code, double check that it executes independently using one of the options explained in How to double check your code executes correctly "on its own".

## Other observations

#### Multiple files

When multiple files are opened in the Editor, the corresponding tabs at the top of the window area are arranged in alphabetical order of the filename from left to right.

On the left of the tabs, there is as icon that shows `Browse tabs` if the mouse hovers over it. It is useful to jump to a particular file directly, if many files are open.

#### Environment variables

Environment variables can be displayed from the Python Console window (bottom right window in default layout). Click on the Options icon (the tooltip is Options), then select Environment variables.

#### Reset all customization

All customization saved on disk can be reset by calling spyder from the command line with the switch --reset, i.e. a command like spyder --reset.

#### Objects in the variable explorer

Right-clicking on arrays in the variable explorer gives options to plot and analyze these further.

Double clicking on a dictionary object opens a new window that displays the dictionary nicely.

You can also show and edit the contents of numpy arrays, lists, numbers and strings.

## Documentation string formatting

If you want to add documentation for the code you are developing, we recommend you to write documentation strings (or *docstrings*) for it, using a special format called restructured text (quick reference). This format also needs to follow a set of conventions called the Numpydoc standard

If you follow those guidelines, you can obtain beautifully formated docstrings in Spyder.

For example, to get an average() function look like this in the Spyder Object inspector:



you need to format the documentation string as follows

```
def average(a, b):
```

```
    """
    Given two numbers a and b, return their average value.

    Parameters
    ----------
    a : number
      A number
    b : number
      Another number

    Returns
    -------
    res : number
      The average of a and b, computed using 0.5*(a + b)

    Example
    -------
    >>> average(5, 10)
    7.5

    """

    return (a + b) * 0.5
```

What matters here, is that the word Parameters is used, and underlined. The line a : number shows us that the type of the parameter a is number. In the next line, which is indented, we can write a more extended explanation of what this variable represents, what conditions the allowed types have to fulfill, etc.

The same for all Parameters, and also for the returned value.

Often it is a good idea to include an example too, as shown.

## Debugging

### Line by line step execution of code

Activating the debug mode (with the Debug > Debug menu option or Ctrl+F5) starts the Python debugger (Pdb) if the Python console is active, or the IPython debugger (ipdb) if the IPython console is active. After doing that, the Editor pane will highlight the line that is about to be executed, and the Variable Explorer will display variables in the current context of the point of program execution. (It only displays 'numerical' and array type of variables, i.e. not function or class objects)

After entering debug mode, you can execute the code line by line using the Step button of the Debug toolbar:

or the shortcut Ctrl+F10. You can also inspect how a particular function is working by stepping into it with the Step into button

or the shortcut Ctrl+F11. Finally, to get out of a function and continue with the next line you need to use the Step return button

or the shortcut Ctrl+F12.

If you prefer to inspect your program at a specific point, you need to insert a *breakpoint* by pressing F12 on the line on which you want to stop. After that a red dot will be placed next to the line and you can press the Continue button

(after entering debug mode) to stop the execution at that line.

**Note**

You can also control the debugging process by issuing these commands in the console prompt:

- n to move to the Next statement.
- s to Step into the current statement. If this is a function call, step into that function.
- r to complete all statements in the current function and Return from that function before returning control.
- p to print values of variables, for example p x will print the value of the variable x.

At the debugger prompt, you can also *change* values of variables. For example, to modify a variable x at the IPython debugger prompt, you can say ipdb > x = 42 and the debugger will carry on with xbeing bound

to 42. You can also call functions, and do many others things. Try this example:

```
def demo(x):
    for i in range(5):
        print("i={}, x={}".format(i, x))
        x = x + 1

demo(0)
```

If we execute this (Run > Run), we should see the output:

```
i=0, x=0
i=1, x=1
i=2, x=2
i=3, x=3
i=4, x=4
```

Now execute this using the debugger (Debug > Debug), press the Step button until the highlighted line reaches the demo(0) function call, then press the Step into to inspect this function. Keep pressing the Step button to execute the next lines. Then, modify x by typing x=10 in the debugger prompt. You see x changing in the Variable Explorer. You should also see x changing when its value is printed as part of the demo() function. (The printed output appears between your debugger commands and responses.)

This debugging ability to execute code line by line, to inspect variables as they change, and to modify them manually is a powerful tool to understand what a piece of code is doing (and to correct it if desired).

To leave the debugging mode, you can type exit or select from the menu Debug > Debugging Control > Exit

#### Debugging once an exception has occurred with IPython

In the IPython console, we can call %debug straight after an exception has been raised: this will start the IPython debug mode, which allows inspection of local variables at the point where the exception occurred as described above. This is a lot more efficient than adding print statements to the code an running it again.

If you use this, you may also want to use the commands up (i.e. press u at the debugger) and down (i.e. press d) which navigate the

inspection point up and down the stack. (Up the stack means to the functions that have called the current function; down is the opposite direction.)

## Plotting

### Plotting with the IPython console

Assuming we use an IPython console with version >= 1.0.0, we can decide whether figures created with matplotlib/pylab will show

1. *inline*, i.e. inside the IPython console, or whether they should
2. appear inside a new window.

Option 1 is convenient to save a record of the interactive session (section Shortcuts for useful functions lists a shortcut to save the IPython console to an html file).

Option 2 allows to interactively zoom into the figure, manipulate it a little, and save the figure to different file formats via a menu the window it contains has.

The command to get the figures to appear *inline* in the IPython console is:

```
In [3]: %matplotlib inline
```

The command to get figures appear in their own window (which technically is a Qt windown) is:

```
In [4]: %matplotlib qt
```

The Spyder preferences can be used to customize the default behavior (in particular Preferences > IPython Console > Graphics > Activate Support to switch into inline plotting).

Here are two lines you can use to quickly create a plot and test this:

```
In [5]: import pylab
In [6]: pylab.plot(range(10), 'o')
```

### Plotting with the Python console

If we use the Python console, all plots will appear in a new window (there is no way of making it appear inline inside the Python console – this only works for the IPython Console).

Here is a brief example that you can use to create and display a plot:

```
>>> import pylab
>>> pylab.plot(range(10), 'o')
```

If you execute your code in a dedicated console, you need to use matplotlib's or pylab's show() command in your code to make a plot appear, like this: pylab.show().

Note that the show() command will bind the focus to new window that has appeared, and that you will need to close that window before Spyder can accept any further commands or respond to interaction. If you cannot see the new window, check whether it may have appeared behind the Spyder window, or be partly hidden.

## Historical note

This tutorial is based on notes by Hans Fangohr, that are used at the University of Southampton to teach Python for computational modelling to undegraduate engineers and postgraduate PhD students for the Next Generation Computational Modelling doctoral training centre