

About Python

Python is one of those rare languages which can claim to be both *simple* and *powerful*. You will find yourself pleasantly surprised to see how easy it is to concentrate on the solution to the problem rather than the syntax and structure of the language you are programming in.

The official introduction to Python is:

Python is an easy to learn, powerful programming language. It has efficient high-level data structures and a simple but effective approach to object-oriented programming. Python's elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms.

I will discuss most of these features in more detail in the next section.

Story behind the name

Guido van Rossum, the creator of the Python language, named the language after the BBC show "Monty Python's Flying Circus". He doesn't particularly like snakes that kill animals for food by winding their long bodies around them and crushing them.

Features of Python

Simple

Python is a simple and minimalistic language. Reading a good Python program feels almost like reading English, although very strict English! This pseudo-code nature of Python is one of its greatest strengths. It allows you to concentrate on the solution to the problem rather than the language itself.

Easy to Learn

As you will see, Python is extremely easy to get started with. Python has an extraordinarily simple syntax, as already mentioned.

Free and Open Source

Python is an example of a *FLOSS* (Free/Libre and Open Source Software). In simple terms, you can freely distribute copies of this software, read its source code, make changes to it, and use pieces of it in new free programs. FLOSS is based on the concept of a community which shares knowledge. This is one of the reasons why Python is so good - it has been created and is constantly improved by a community who just want to see a better Python.

High-level Language

When you write programs in Python, you never need to bother about the low-level details such as managing the memory used by your program, etc.

Portable

Due to its open-source nature, Python has been ported to (i.e. changed to make it work on) many platforms. All your Python programs can work on any of these platforms without requiring any changes at all if you are careful enough to avoid any system-dependent features.

You can use Python on GNU/Linux, Windows, FreeBSD, Macintosh, Solaris, OS/2, Amiga, AROS, AS/400, BeOS, OS/390, z/OS, Palm OS, QNX, VMS, Psion, Acorn RISC OS, VxWorks, PlayStation, Sharp Zaurus, Windows CE and PocketPC!

You can even use a platform like [Kivy](#) to create games for your computer *and* for iPhone, iPad, and Android.

Interpreted

This requires a bit of explanation.

A program written in a compiled language like C or C++ is converted from the source language i.e. C or C++ into a language that is spoken by your computer (binary code i.e. 0s and 1s) using a compiler with various flags and options. When you run the program, the linker/loader software copies the program from hard disk to memory and starts running it.

Python, on the other hand, does not need compilation to binary. You just *run* the program directly from the source code. Internally, Python converts the source code into an intermediate form called bytecodes and then translates this into the native language of your computer and then runs it. All this, actually, makes using Python much easier since you don't have to worry about compiling the program, making sure that the proper libraries are linked and loaded, etc. This also makes your Python programs much more portable, since you can just copy your Python program onto another computer and it just works!

Object Oriented

Python supports procedure-oriented programming as well as object-oriented programming. In *procedure-oriented* languages, the program is built around procedures or functions which are nothing but reusable pieces of programs. In *object-oriented* languages, the program is built around objects which combine data and functionality. Python has a very powerful but simplistic way of doing OOP, especially when compared to big languages like C++ or Java.

Extensible

If you need a critical piece of code to run very fast or want to have some piece of algorithm not to be open, you can code that part of your program in C or C++ and then use it from your Python program.

Embeddable

You can embed Python within your C/C++ programs to give *scripting* capabilities for your program's users.

Extensive Libraries

The Python Standard Library is huge indeed. It can help you do various things involving regular expressions, documentation generation, unit testing, threading, databases, web browsers, CGI, FTP, email, XML, XML-RPC, HTML, WAV files, cryptography, GUI (graphical user interfaces), and other system-dependent stuff. Remember, all this is always available wherever Python is installed. This is called the *Batteries Included* philosophy of Python.

Besides the standard library, there are various other high-quality libraries which you can find at the [Python Package Index](#).

Summary

Python is indeed an exciting and powerful language. It has the right combination of performance and features that make writing programs in Python both fun and easy.

Python 3 versus 2

You can ignore this section if you're not interested in the difference between "Python version 2" and "Python version 3". But please do be aware of which version you are using. This book is written for Python version 3.

Remember that once you have properly understood and learn to use one version, you can easily learn the differences and use the other one. The hard part is learning programming and understanding the basics of Python language itself. That is our goal in this book, and once you have achieved that goal, you can easily use Python 2 or Python 3 depending on your situation.

For details on differences between Python 2 and Python 3, see:

- [The future of Python 2](#)
- [Porting Python 2 Code to Python 3](#)
- [Writing code that runs under both Python2 and 3](#)
- [Supporting Python 3: An in-depth guide](#)

What Programmers Say

You may find it interesting to read what great hackers like ESR have to say about Python:

- *Eric S. Raymond* is the author of "The Cathedral and the Bazaar" and is also the person who coined the term *Open Source*. He says that [Python has become his favorite programming language](#). This article was the real inspiration for my first brush with Python.
- *Bruce Eckel* is the author of the famous 'Thinking in Java' and 'Thinking in C++' books. He says that no language has made him more productive than Python. He says that Python is perhaps the only language that focuses on making things easier for the programmer. Read the [complete interview](#) for more details.
- *Peter Norvig* is a well-known Lisp author and Director of Search Quality at Google (thanks to Guido van Rossum for pointing that out). He says that [writing Python is like writing in pseudocode](#). He says that Python has always been an integral part of Google. You can actually verify this statement by looking at the [Google Jobs](#) page which lists Python knowledge as a requirement for software engineers.

Installation

When we refer to "Python 3" in this book, we will be referring to any version of Python equal to or greater than version [Python 3.6.0](#).

Installation on Windows

Visit <https://www.python.org/downloads/> and download the latest version. At the time of this writing, it was Python 3.5.1 The installation is just like any other Windows-based software.

Note that if your Windows version is pre-Vista, you should [download Python 3.4 only](#) as later versions require newer versions of Windows.

CAUTION: Make sure you check option `Add Python 3.5 to PATH`.

To change install location, click on `Customize installation`, then `Next` and enter `C:\python35` (or another appropriate location) as the install location.

If you didn't check the `Add Python 3.5 PATH` option earlier, check `Add Python to environment variables`. This does the same thing as `Add Python 3.5 to PATH` on the first install screen.

You can choose to install Launcher for all users or not, it does not matter much. Launcher is used to switch between different versions of Python installed.

If your path was not set correctly (by checking the `Add Python 3.5 Path` or `Add Python to environment variables` options), then follow the steps in the next section (`DOS Prompt`) to fix it. Otherwise, go to the `Running Python prompt on Windows` section in this document.

NOTE: For people who already know programming, if you are familiar with Docker, check out [Python in Docker](#) and [Docker on Windows](#).

DOS Prompt

If you want to be able to use Python from the Windows command line i.e. the DOS prompt, then you need to set the PATH variable appropriately.

For Windows 2000, XP, 2003, click on `Control Panel -> System -> Advanced -> Environment Variables`. Click on the variable named `PATH` in the *System Variables* section, then select `Edit` and add `;C:\Python35` (please verify that this folder exists, it will be

different for newer versions of Python) to the end of what is already there. Of course, use the appropriate directory name.

For older versions of Windows, open the file `C:\AUTOEXEC.BAT` and add the line

`PATH=%PATH%;C:\Python35` and restart the system. For Windows NT, use the `AUTOEXEC.NT` file.

For Windows Vista:

- Click Start and choose `Control Panel`
- Click System, on the right you'll see "View basic information about your computer"
- On the left is a list of tasks, the last of which is `Advanced system settings`. Click that.
- The `Advanced` tab of the `System Properties` dialog box is shown. Click the `Environment Variables` button on the bottom right.
- In the lower box titled `System Variables` scroll down to Path and click the `Edit` button.
- Change your path as need be.
- Restart your system. Vista didn't pick up the system path environment variable change until I restarted.

For Windows 7 and 8:

- Right click on Computer from your desktop and select `Properties` or click `Start` and choose `Control Panel -> System and Security -> System`. Click on `Advanced system settings` on the left and then click on the `Advanced` tab. At the bottom click on `Environment Variables` and under `System variables`, look for the `PATH` variable, select and then press `Edit`.
- Go to the end of the line under Variable value and append `;C:\Python35` (please verify that this folder exists, it will be different for newer versions of Python) to the end of what is already there. Of course, use the appropriate folder name.
- If the value was `%SystemRoot%\system32;` It will now become `%SystemRoot%\system32;C:\Python36`
- Click `OK` and you are done. No restart is required, however you may have to close and reopen the command line.

For Windows 10:

Windows Start Menu > `Settings` > `About` > `System Info` (this is all the way over to the right) > `Advanced System Settings` > `Environment Variables` (this is towards the bottom) > (then highlight `Path` variable and click `Edit`) > `New` > (type in whatever your python location is. For example, `C:\Python35\`)

Running Python prompt on Windows

For Windows users, you can run the interpreter in the command line if you have [set the PATH variable appropriately](#).

To open the terminal in Windows, click the start button and click `Run`. In the dialog box, type `cmd` and press `[enter]` key.

Then, type `python` and ensure there are no errors.

Installation on Mac OS X

For Mac OS X users, use [Homebrew](#): `brew install python3`.

To verify, open the terminal by pressing `[Command + Space]` keys (to open Spotlight search), type `Terminal` and press `[enter]` key. Now, run `python3` and ensure there are no errors.

Installation on GNU/Linux

For GNU/Linux users, use your distribution's package manager to install Python 3, e.g. on Debian & Ubuntu: `sudo apt-get update && sudo apt-get install python3`.

To verify, open the terminal by opening the `Terminal` application or by pressing `Alt + F2` and entering `gnome-terminal`. If that doesn't work, please refer the documentation of your particular GNU/Linux distribution. Now, run `python3` and ensure there are no errors.

You can see the version of Python on the screen by running:

```
$ python3 -V
Python 3.6.0
```

NOTE: `$` is the prompt of the shell. It will be different for you depending on the settings of the operating system on your computer, hence I will indicate the prompt by just the `$` symbol.

CAUTION: Output may be different on your computer, depending on the version of Python software installed on your computer.

Summary

From now on, we will assume that you have Python installed on your system.

Next, we will write our first Python program.

First Steps

We will now see how to run a traditional 'Hello World' program in Python. This will teach you how to write, save and run Python programs.

There are two ways of using Python to run your program - using the interactive interpreter prompt or using a source file. We will now see how to use both of these methods.

Using The Interpreter Prompt

Open the terminal in your operating system (as discussed previously in the [Installation](#) chapter) and then open the Python prompt by typing `python3` and pressing `[enter]` key.

Once you have started Python, you should see `>>>` where you can start typing stuff. This is called the *Python interpreter prompt*.

At the Python interpreter prompt, type:

```
print("Hello World")
```

followed by the `[enter]` key. You should see the words `Hello World` printed to the screen.

Here is an example of what you should be seeing, when using a Mac OS X computer. The details about the Python software will differ based on your computer, but the part from the prompt (i.e. from `>>>` onwards) should be the same regardless of the operating system.

```
$ python3
Python 3.6.0 (default, Jan 12 2017, 11:26:36)
[GCC 4.2.1 Compatible Apple LLVM 8.0.0 (clang-800.0.38)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello World")
Hello World
```

Notice that Python gives you the output of the line immediately! What you just entered is a single Python *statement*. We use `print` to (unsurprisingly) print any value that you supply to it. Here, we are supplying the text `Hello World` and this is promptly printed to the screen.

How to Quit the Interpreter Prompt

If you are using a GNU/Linux or OS X shell, you can exit the interpreter prompt by pressing `[ctrl + d]` or entering `exit()` (note: remember to include the parentheses, `()`) followed by the `[enter]` key.

If you are using the Windows command prompt, press `[ctrl + z]` followed by the `[enter]` key.

Choosing An Editor

We cannot type out our program at the interpreter prompt every time we want to run something, so we have to save them in files and can run our programs any number of times.

To create our Python source files, we need an editor software where you can type and save. A good programmer's editor will make your life easier in writing the source files. Hence, the choice of an editor is crucial indeed. You have to choose an editor as you would choose a car you would buy. A good editor will help you write Python programs easily, making your journey more comfortable and helps you reach your destination (achieve your goal) in a much faster and safer way.

One of the very basic requirements is *syntax highlighting* where all the different parts of your Python program are colorized so that you can see your program and visualize its running.

If you have no idea where to start, I would recommend using [PyCharm Educational Edition](#) software which is available on Windows, Mac OS X and GNU/Linux. Details in the next section.

If you are using Windows, *do not use Notepad* - it is a bad choice because it does not do syntax highlighting and also importantly it does not support indentation of the text which is very important in our case as we will see later. Good editors will automatically do this.

If you are an experienced programmer, then you must be already using [Vim](#) or [Emacs](#). Needless to say, these are two of the most powerful editors and you will benefit from using them to write your Python programs. I personally use both for most of my programs, and have even written an [entire book on Vim](#).

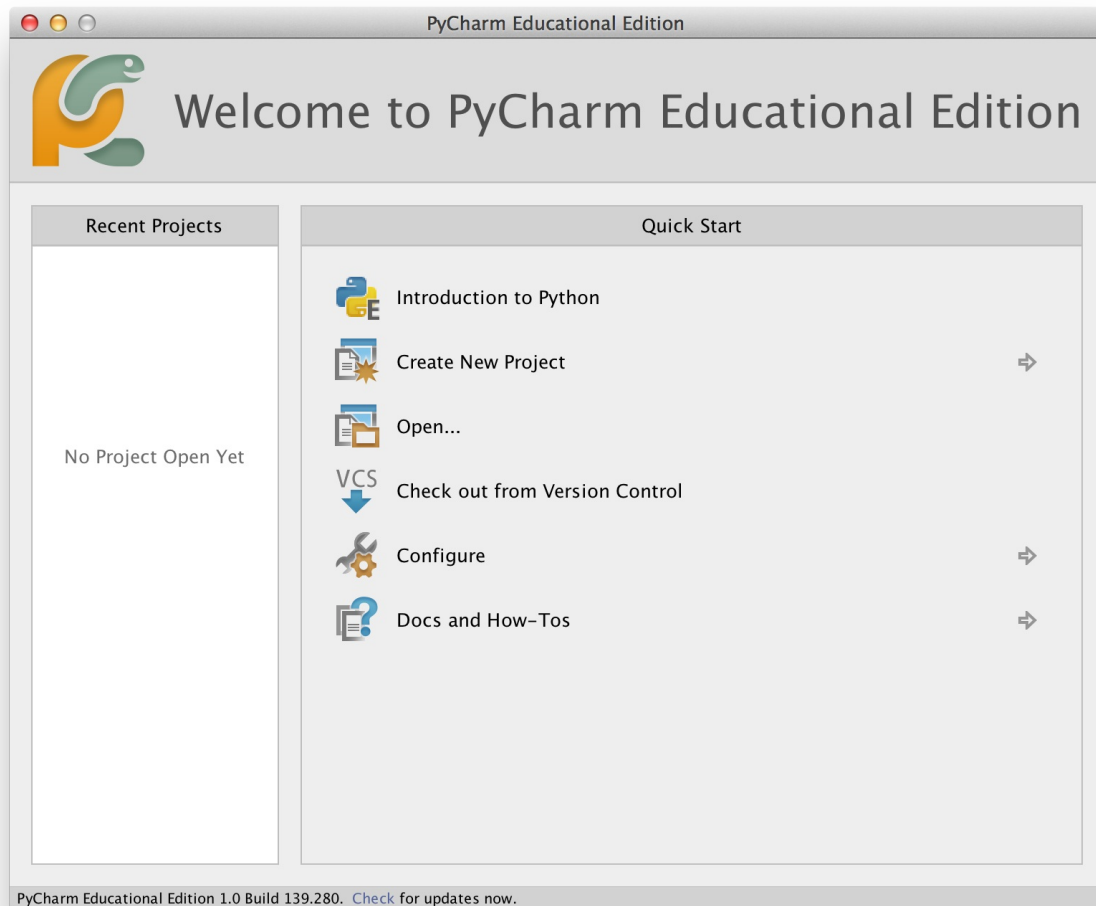
In case you are willing to take the time to learn Vim or Emacs, then I highly recommend that you do learn to use either of them as it will be very useful for you in the long run. However, as I mentioned before, beginners can start with PyCharm and focus the learning on Python rather than the editor at this moment.

To reiterate, please choose a proper editor - it can make writing Python programs more fun and easy.

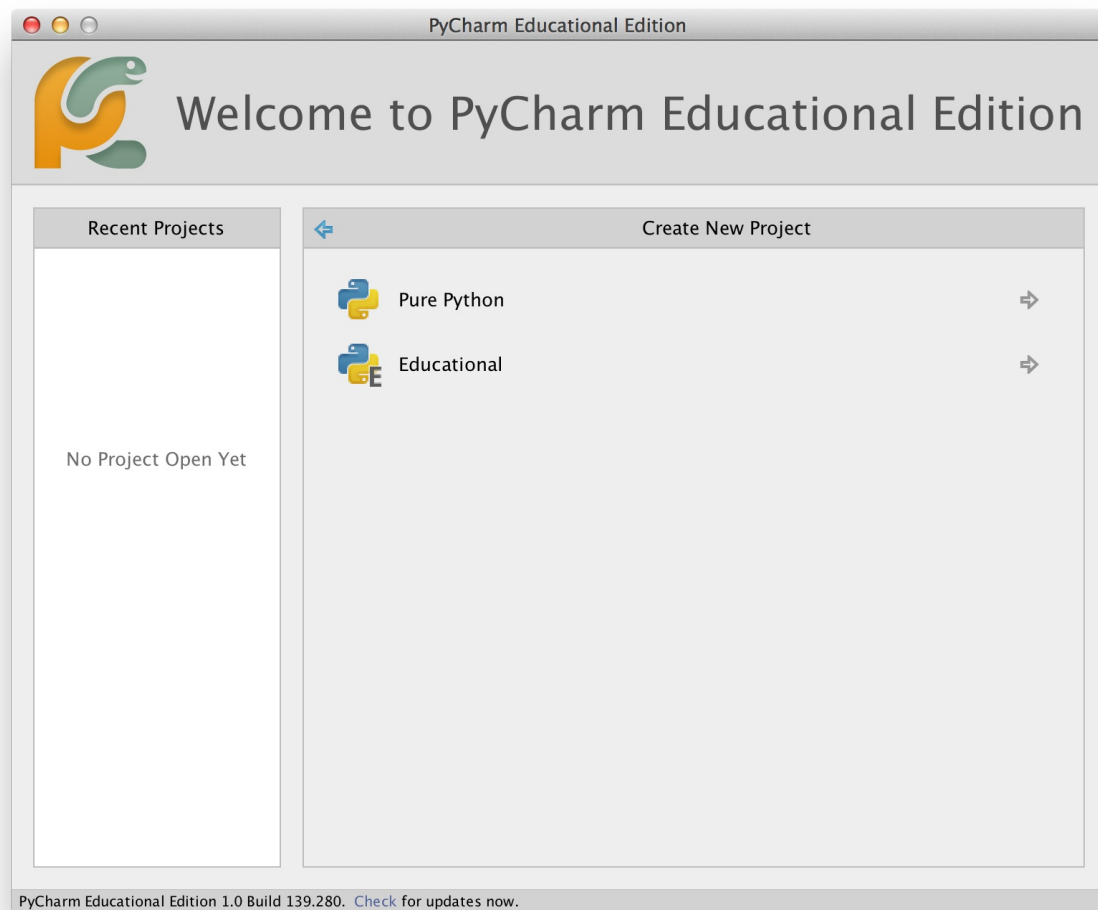
PyCharm

[PyCharm Educational Edition](#) is a free editor which you can use for writing Python programs.

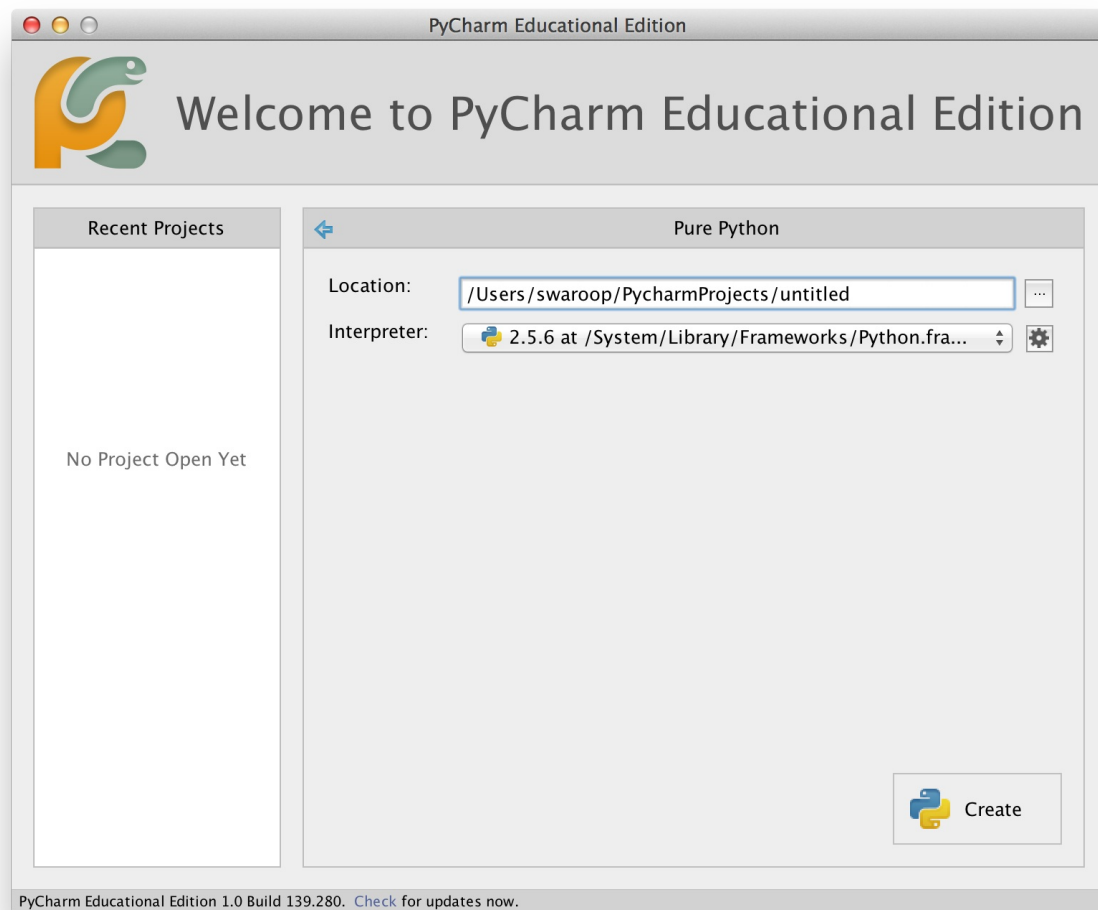
When you open PyCharm, you'll see this, click on `Create New Project` :



Select `Pure Python` :

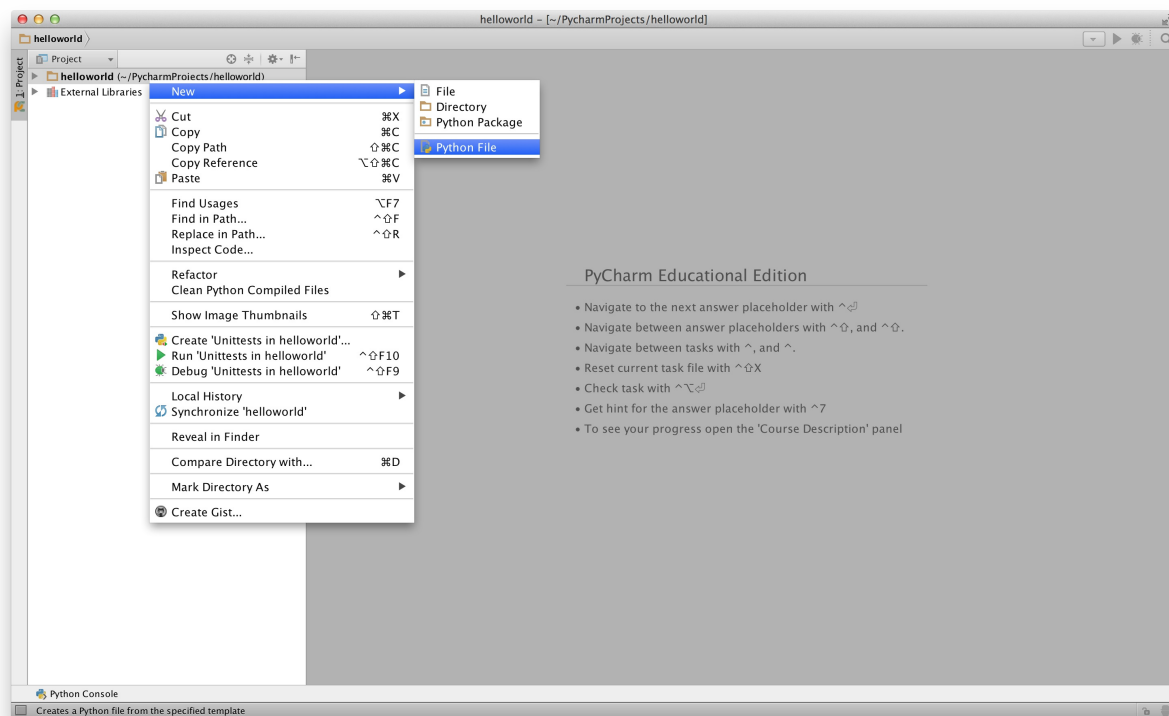


Change `untitled` to `helloworld` as the location of the project, you should see details similar to this:

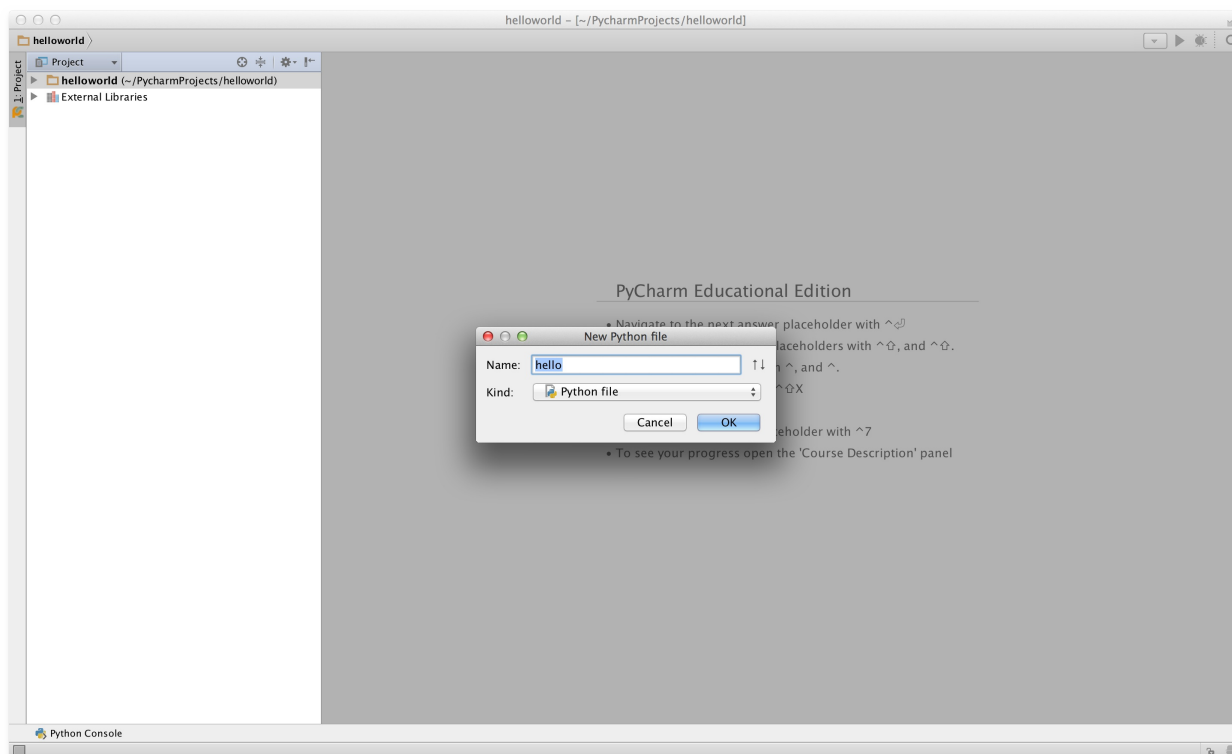


Click the `create` button.

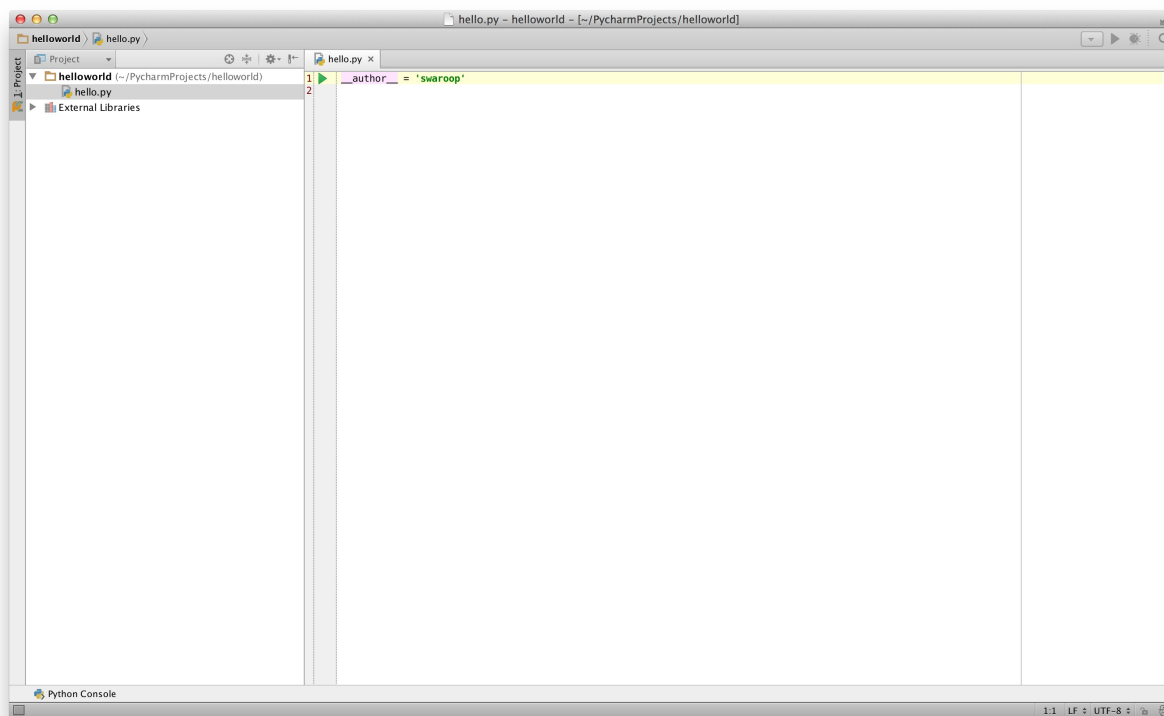
Right-click on the `helloworld` in the sidebar and select `New` -> `Python File` :



You will be asked to type the name, type `hello` :



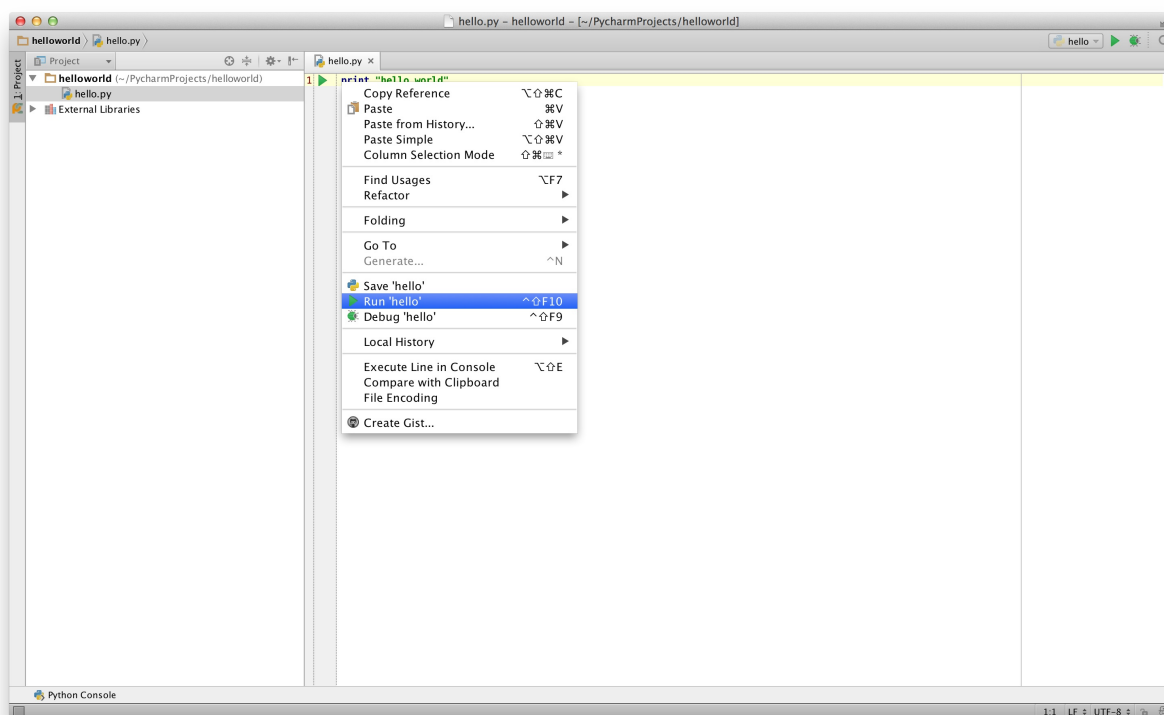
You can now see a file opened for you:



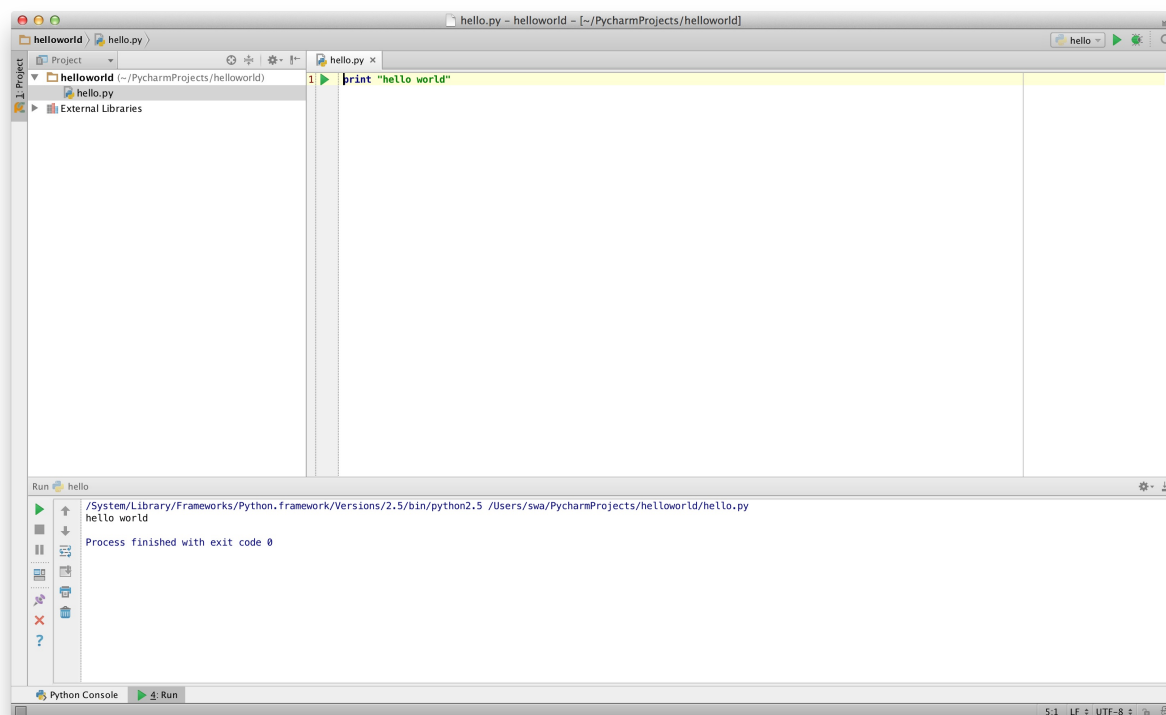
Delete the lines that are already present, and now type the following:

```
print("hello world")
```

Now right-click on what you typed (without selecting the text), and click on `Run 'hello'`.



You should now see the output (what it prints) of your program:



Phew! That was quite a few steps to get started, but henceforth, every time we ask you to create a new file, remember to just right-click on `helloworld` on the left -> `New` -> `Python File` and continue the same steps to type and run as shown above.

You can find more information about PyCharm in the [PyCharm Quickstart](#) page.

Vim

1. Install [Vim](#)
 - Mac OS X users should install `macvim` package via [HomeBrew](#)
 - Windows users should download the "self-installing executable" from [Vim website](#)
 - GNU/Linux users should get Vim from their distribution's software repositories, e.g. Debian and Ubuntu users can install the `vim` package.
2. Install [jedi-vim](#) plugin for autocompletion.
3. Install corresponding `jedi` python package : `pip install -U jedi`

Emacs

1. Install [Emacs 24+](#).
 - Mac OS X users should get Emacs from <http://emacsformacosx.com>
 - Windows users should get Emacs from <http://ftp.gnu.org/gnu/emacs/windows/>

- GNU/Linux users should get Emacs from their distribution's software repositories, e.g. Debian and Ubuntu users can install the `emacs24` package.

2. Install [ELPY](#)

Using A Source File

Now let's get back to programming. There is a tradition that whenever you learn a new programming language, the first program that you write and run is the 'Hello World' program - all it does is just say 'Hello World' when you run it. As Simon Cozens¹ says, it is the "traditional incantation to the programming gods to help you learn the language better."

Start your choice of editor, enter the following program and save it as `hello.py`.

If you are using PyCharm, we have already [discussed how to run from a source file](#).

For other editors, open a new file `hello.py` and type this:

```
print("hello world")
```

Where should you save the file? To any folder for which you know the location of the folder. If you don't understand what that means, create a new folder and use that location to save and run all your Python programs:

- `/tmp/py` on Mac OS X
- `/tmp/py` on GNU/Linux
- `c:\py` on Windows

To create the above folder (for the operating system you are using), use the `mkdir` command in the terminal, for example, `mkdir /tmp/py`.

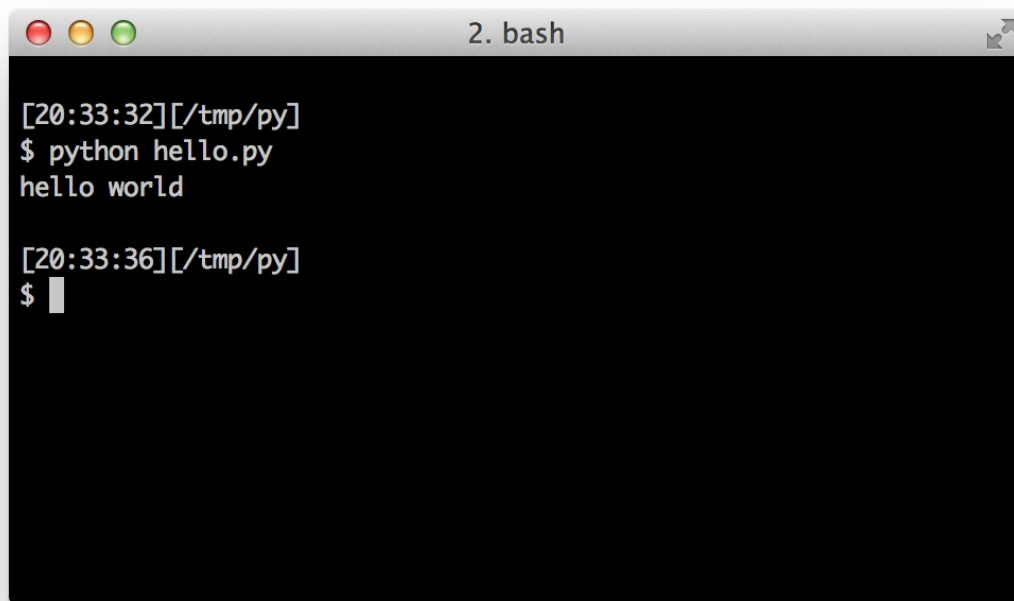
IMPORTANT: Always ensure that you give it the file extension of `.py`, for example,

```
foo.py
```

To run your Python program:

1. Open a terminal window (see the previous [Installation](#) chapter on how to do that)
2. **Change directory** to where you saved the file, for example, `cd /tmp/py`
3. Run the program by entering the command `python hello.py`. The output is as shown below.

```
$ python hello.py
hello world
```

A terminal window titled "2. bash" with a dark background. It shows two prompts. The first prompt is "[20:33:32] [/tmp/py]" followed by "\$ python hello.py" and the output "hello world". The second prompt is "[20:33:36] [/tmp/py]" followed by "\$" and a cursor.

```
[20:33:32] [/tmp/py]
$ python hello.py
hello world

[20:33:36] [/tmp/py]
$
```

If you got the output as shown above, congratulations! - you have successfully run your first Python program. You have successfully crossed the hardest part of learning programming, which is, getting started with your first program!

In case you got an error, please type the above program *exactly* as shown above and run the program again. Note that Python is case-sensitive i.e. `print` is not the same as `Print` - note the lowercase `p` in the former and the uppercase `P` in the latter. Also, ensure there are no spaces or tabs before the first character in each line - we will see [why this is important](#) later.

How It Works

A Python program is composed of *statements*. In our first program, we have only one statement. In this statement, we call the `print` *statement* to which we supply the text "hello world".

Getting Help

If you need quick information about any function or statement in Python, then you can use the built-in `help` functionality. This is very useful especially when using the interpreter prompt. For example, run `help('len')` - this displays the help for the `len` function which is used to count number of items.

TIP: Press `q` to exit the help.

Similarly, you can obtain information about almost anything in Python. Use `help()` to learn more about using `help` itself!

In case you need to get help for operators like `return`, then you need to put those inside quotes such as `help('return')` so that Python doesn't get confused on what we're trying to do.

Summary

You should now be able to write, save and run Python programs at ease.

Now that you are a Python user, let's learn some more Python concepts.

¹. the author of the amazing 'Beginning Perl' book [↩](#)

Basics

Just printing `hello world` is not enough, is it? You want to do more than that - you want to take some input, manipulate it and get something out of it. We can achieve this in Python using constants and variables, and we'll learn some other concepts as well in this chapter.

Comments

Comments are any text to the right of the `#` symbol and is mainly useful as notes for the reader of the program.

For example:

```
print('hello world') # Note that print is a function
```

or:

```
# Note that print is a function  
print('hello world')
```

Use as many useful comments as you can in your program to:

- explain assumptions
- explain important decisions
- explain important details
- explain problems you're trying to solve
- explain problems you're trying to overcome in your program, etc.

Code tells you how, comments should tell you why.

This is useful for readers of your program so that they can easily understand what the program is doing. Remember, that person can be yourself after six months!

Literal Constants

An example of a literal constant is a number like `5` , `1.23` , or a string like `'This is a string'` or `"It's a string!"` .

It is called a literal because it is *literal* - you use its value literally. The number `2` always represents itself and nothing else - it is a *constant* because its value cannot be changed. Hence, all these are referred to as literal constants.

Numbers

Numbers are mainly of two types - integers and floats.

An example of an integer is `2` which is just a whole number.

Examples of floating point numbers (or *floats* for short) are `3.23` and `52.3E-4`. The `E` notation indicates powers of 10. In this case, `52.3E-4` means `52.3 * 10-4`.

Note for Experienced Programmers

There is no separate `long` type. The `int` type can be an integer of any size.

Strings

A string is a *sequence* of *characters*. Strings are basically just a bunch of words.

You will be using strings in almost every Python program that you write, so pay attention to the following part.

Single Quote

You can specify strings using single quotes such as `'Quote me on this'`.

All white space i.e. spaces and tabs, within the quotes, are preserved as-is.

Double Quotes

Strings in double quotes work exactly the same way as strings in single quotes. An example is `"What's your name?"`.

Triple Quotes

You can specify multi-line strings using triple quotes - (`"""` or `'''`). You can use single quotes and double quotes freely within the triple quotes. An example is:

```
'''This is a multi-line string. This is the first line.
This is the second line.
"What's your name?," I asked.
He said "Bond, James Bond."
'''
```

Strings Are Immutable

This means that once you have created a string, you cannot change it. Although this might seem like a bad thing, it really isn't. We will see why this is not a limitation in the various programs that we see later on.

Note for C/C++ Programmers

There is no separate `char` data type in Python. There is no real need for it and I am sure you won't miss it.

Note for Perl/PHP Programmers

Remember that single-quoted strings and double-quoted strings are the same - they do not differ in any way.

The format method

Sometimes we may want to construct strings from other information. This is where the `format()` method is useful.

Save the following lines as a file `str_format.py` :

```
age = 20
name = 'Swaroop'

print('{0} was {1} years old when he wrote this book'.format(name, age))
print('Why is {0} playing with that python?'.format(name))
```

Output:

```
$ python str_format.py
Swaroop was 20 years old when he wrote this book
Why is Swaroop playing with that python?
```

How It Works

A string can use certain specifications and subsequently, the `format` method can be called to substitute those specifications with corresponding arguments to the `format` method.

Observe the first usage where we use `{0}` and this corresponds to the variable `name` which is the first argument to the `format` method. Similarly, the second specification is `{1}` corresponding to `age` which is the second argument to the `format` method. Note that Python starts counting from 0 which means that first position is at index 0, second position is at index 1, and so on.

Notice that we could have achieved the same using string concatenation:

```
name + ' is ' + str(age) + ' years old'
```

but that is much uglier and error-prone. Second, the conversion to string would be done automatically by the `format` method instead of the explicit conversion to strings needed in this case. Third, when using the `format` method, we can change the message without having to deal with the variables used and vice-versa.

Also note that the numbers are optional, so you could have also written as:

```
age = 20
name = 'Swaroop'

print('{} was {} years old when he wrote this book'.format(name, age))
print('Why is {} playing with that python?'.format(name))
```

which will give the same exact output as the previous program.

We can also name the parameters:

```
age = 20
name = 'Swaroop'

print('{name} was {age} years old when he wrote this book'.format(name=name, age=age))
print('Why is {name} playing with that python?'.format(name=name))
```

which will give the same exact output as the previous program.

Python 3.6 introduced a shorter way to do named parameters, called "f-strings":

```
age = 20
name = 'Swaroop'

print(f'{name} was {age} years old when he wrote this book') # notice the 'f' before
the string
print(f'Why is {name} playing with that python?') # notice the 'f' before the string
```

which will give the same exact output as the previous program.

What Python does in the `format` method is that it substitutes each argument value into the place of the specification. There can be more detailed specifications such as:

```
# decimal (.) precision of 3 for float '0.333'
print('{0:.3f}'.format(1.0/3))
# fill with underscores (_) with the text centered
# (^) to 11 width '___hello___'
print('{0:_^11}'.format('hello'))
# keyword-based 'Swaroop wrote A Byte of Python'
print('{name} wrote {book}'.format(name='Swaroop', book='A Byte of Python'))
```

Output:

```
0.333
___hello___
Swaroop wrote A Byte of Python
```

Since we are discussing formatting, note that `print` always ends with an invisible "new line" character (`\n`) so that repeated calls to `print` will all print on a separate line each. To prevent this newline character from being printed, you can specify that it should `end` with a blank:

```
print('a', end='')
print('b', end='')
```

Output is:

```
ab
```

Or you can `end` with a space:

```
print('a', end=' ')
print('b', end=' ')
print('c')
```

Output is:

```
a b c
```

Escape Sequences

Suppose, you want to have a string which contains a single quote (`'`), how will you specify this string? For example, the string is `"What's your name?"` . You cannot specify `'What's your name?'` because Python will be confused as to where the string starts and ends. So, you will have to specify that this single quote does not indicate the end of the string. This can be done with the help of what is called an *escape sequence*. You specify the single quote as `\'` : notice the backslash. Now, you can specify the string as `'What\'s your name?'` .

Another way of specifying this specific string would be `"What's your name?"` i.e. using double quotes. Similarly, you have to use an escape sequence for using a double quote itself in a double quoted string. Also, you have to indicate the backslash itself using the escape sequence `\\` .

What if you wanted to specify a two-line string? One way is to use a triple-quoted string as shown [previously](#) or you can use an escape sequence for the newline character - `\n` to indicate the start of a new line. An example is:

```
'This is the first line\nThis is the second line'
```

Another useful escape sequence to know is the tab: `\t` . There are many more escape sequences but I have mentioned only the most useful ones here.

One thing to note is that in a string, a single backslash at the end of the line indicates that the string is continued in the next line, but no newline is added. For example:

```
"This is the first sentence. \  
This is the second sentence."
```

is equivalent to

```
"This is the first sentence. This is the second sentence."
```

Raw String

If you need to specify some strings where no special processing such as escape sequences are handled, then what you need is to specify a *raw* string by prefixing `r` or `R` to the string. An example is:

```
r"Newlines are indicated by \n"
```

Note for Regular Expression Users

Always use raw strings when dealing with regular expressions. Otherwise, a lot of backwhacking may be required. For example, backreferences can be referred to as

```
'\\1' or r'\1'.
```

Variable

Using just literal constants can soon become boring - we need some way of storing any information and manipulate them as well. This is where *variables* come into the picture. Variables are exactly what the name implies - their value can vary, i.e., you can store anything using a variable. Variables are just parts of your computer's memory where you store some information. Unlike literal constants, you need some method of accessing these variables and hence you give them names.

Identifier Naming

Variables are examples of identifiers. *Identifiers* are names given to identify *something*.

There are some rules you have to follow for naming identifiers:

- The first character of the identifier must be a letter of the alphabet (uppercase ASCII or lowercase ASCII or Unicode character) or an underscore (`_`).
- The rest of the identifier name can consist of letters (uppercase ASCII or lowercase ASCII or Unicode character), underscores (`_`) or digits (0-9).
- Identifier names are case-sensitive. For example, `myname` and `myName` are *not* the same. Note the lowercase `n` in the former and the uppercase `N` in the latter.
- Examples of *valid* identifier names are `i` , `name_2_3` . Examples of *invalid* identifier names are `2things` , `this is spaced out` , `my-name` and `>a1b2_c3` .

Data Types

Variables can hold values of different types called *data types*. The basic types are numbers and strings, which we have already discussed. In later chapters, we will see how to create our own types using [classes](#).

Object

Remember, Python refers to anything used in a program as an *object*. This is meant in the generic sense. Instead of saying "the *something*", we say "the *object*".

Note for Object Oriented Programming users:

Python is strongly object-oriented in the sense that everything is an object including numbers, strings and functions.

We will now see how to use variables along with literal constants. Save the following example and run the program.

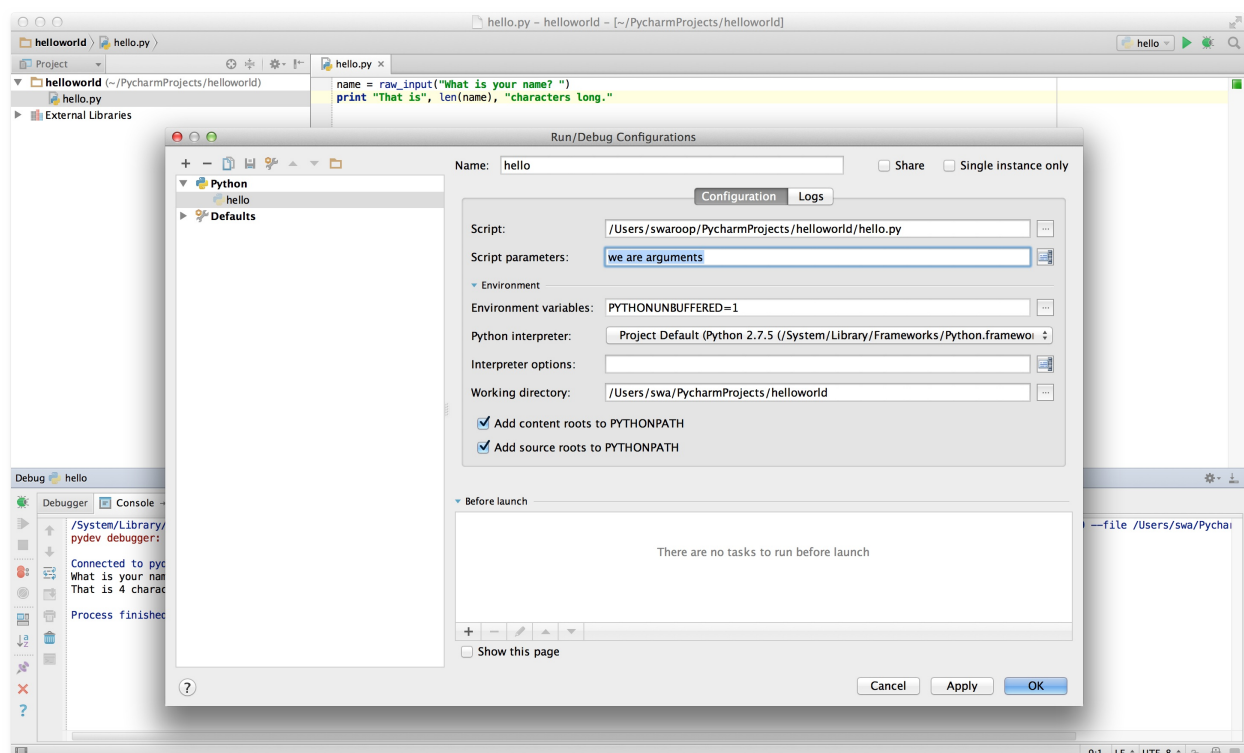
How to write Python programs

Henceforth, the standard procedure to save and run a Python program is as follows:

For PyCharm

1. Open [PyCharm](#).
2. Create new file with the filename mentioned.
3. Type the program code given in the example.
4. Right-click and run the current file.

NOTE: Whenever you have to provide [command line arguments](#), click on `Run` -> `Edit Configurations` and type the arguments in the `Script parameters:` section and click the `OK` button:



For other editors

1. Open your editor of choice.
2. Type the program code given in the example.
3. Save it as a file with the filename mentioned.
4. Run the interpreter with the command `python program.py` to run the program.

Example: Using Variables And Literal Constants

Type and run the following program:

```
# Filename : var.py
i = 5
print(i)
i = i + 1
print(i)

s = '''This is a multi-line string.
This is the second line.'''
print(s)
```

Output:

```
5
6
This is a multi-line string.
This is the second line.
```

How It Works

Here's how this program works. First, we assign the literal constant value `5` to the variable `i` using the assignment operator (`=`). This line is called a statement because it states that something should be done and in this case, we connect the variable name `i` to the value `5`. Next, we print the value of `i` using the `print` statement which, unsurprisingly, just prints the value of the variable to the screen.

Then we add `1` to the value stored in `i` and store it back. We then print it and expectedly, we get the value `6`.

Similarly, we assign the literal string to the variable `s` and then print it.

Note for static language programmers

Variables are used by just assigning them a value. No declaration or data type definition is needed/used.

Logical And Physical Line

A physical line is what you see when you write the program. A logical line is what *Python* sees as a single statement. Python implicitly assumes that each *physical line* corresponds to a *logical line*.

An example of a logical line is a statement like `print('hello world')` - if this was on a line by itself (as you see it in an editor), then this also corresponds to a physical line.

Implicitly, Python encourages the use of a single statement per line which makes code more readable.

If you want to specify more than one logical line on a single physical line, then you have to explicitly specify this using a semicolon (`;`) which indicates the end of a logical line/statement. For example:

```
i = 5
print(i)
```

is effectively same as

```
i = 5;
print(i);
```

which is also same as

```
i = 5; print(i);
```

and same as

```
i = 5; print(i)
```

However, I *strongly recommend* that you stick to *writing a maximum of a single logical line on each single physical line*. The idea is that you should never use the semicolon. In fact, I have *never* used or even seen a semicolon in a Python program.

There is one kind of situation where this concept is really useful: if you have a long line of code, you can break it into multiple physical lines by using the backslash. This is referred to as *explicit line joining*:

```
s = 'This is a string. \
This continues the string.'
print(s)
```

Output:

```
This is a string. This continues the string.
```

Similarly,

```
i = \
5
```

is the same as

```
i = 5
```

Sometimes, there is an implicit assumption where you don't need to use a backslash. This is the case where the logical line has a starting parentheses, starting square brackets or a starting curly braces but not an ending one. This is called *implicit line joining*. You can see this in action when we write programs using [list](#) in later chapters.

Indentation

Whitespace is important in Python. Actually, *whitespace at the beginning of the line is important*. This is called *indentation*. Leading whitespace (spaces and tabs) at the beginning of the logical line is used to determine the indentation level of the logical line, which in turn is used to determine the grouping of statements.

This means that statements which go together *must* have the same indentation. Each such set of statements is called a *block*. We will see examples of how blocks are important in later chapters.

One thing you should remember is that wrong indentation can give rise to errors. For example:

```
i = 5
# Error below! Notice a single space at the start of the line
 print('Value is', i)
print('I repeat, the value is', i)
```

When you run this, you get the following error:

```
File "whitespace.py", line 3
    print('Value is', i)
    ^
IndentationError: unexpected indent
```

Notice that there is a single space at the beginning of the second line. The error indicated by Python tells us that the syntax of the program is invalid i.e. the program was not properly written. What this means to you is that *you cannot arbitrarily start new blocks of statements* (except for the default main block which you have been using all along, of course). Cases where you can use new blocks will be detailed in later chapters such as the [control flow](#).

How to indent

Use four spaces for indentation. This is the official Python language recommendation. Good editors will automatically do this for you. Make sure you use a consistent number of spaces for indentation, otherwise your program will not run or will have unexpected behavior.

Note to static language programmers

Python will always use indentation for blocks and will never use braces. Run `from __future__ import braces` to learn more.

Summary

Now that we have gone through many nitty-gritty details, we can move on to more interesting stuff such as control flow statements. Be sure to become comfortable with what you have read in this chapter.

Operators and Expressions

Most statements (logical lines) that you write will contain *expressions*. A simple example of an expression is `2 + 3`. An expression can be broken down into operators and operands.

Operators are functionality that do something and can be represented by symbols such as `+` or by special keywords. Operators require some data to operate on and such data is called *operands*. In this case, `2` and `3` are the operands.

Operators

We will briefly take a look at the operators and their usage.

Note that you can evaluate the expressions given in the examples using the interpreter interactively. For example, to test the expression `2 + 3`, use the interactive Python interpreter prompt:

```
>>> 2 + 3
5
>>> 3 * 5
15
>>>
```

Here is a quick overview of the available operators:

- `+` (plus)
 - Adds two objects
 - `3 + 5` gives `8`. `'a' + 'b'` gives `'ab'`.
- `-` (minus)
 - Gives the subtraction of one number from the other; if the first operand is absent it is assumed to be zero.
 - `-5.2` gives a negative number and `50 - 24` gives `26`.
- `*` (multiply)
 - Gives the multiplication of the two numbers or returns the string repeated that many times.
 - `2 * 3` gives `6`. `'la' * 3` gives `'lalala'`.
- `**` (power)
 - Returns x to the power of y