Module 08	ITFO 106 Introduction to Python Programming & Django Framework

Like that of other languages such as C++, Java, Tcl, and JS, Python is also a high-level, interactive, object-oriented scripting language. It is easy to learn, read and understand. The syntactical constructions are easy and simpler as compared to other programming languages. Furthermore, it has few other key features and characteristics which make this language unique from other languages. In this chapter, you will get to know about what Python is and its advantages, features, difference with respect to Python 2 and why Python is preferred over other languages.

Characteristic Features of Python –

- ❖ This language supports practical and planned programming methods along with OOP.
- ❖ This language can be used as a lightweight scripting language or can be written so that it can be compiled in byte-code in order to build large applications.
- ❖ This language offers very high-level, dynamic data types with real-time type checking.
- ❖ This language is used for supporting automatic garbage collection.
- ❖ This language has the capability to easily integrate with C, C++, ActiveX languages, COM, Java and CORBA.

Installations and versions 2.x and 3.x

To start coding the Python language, you have to setup its environment so that there you can write the Python code. So, in this chapter, you will learn about how to set up your Python environment in different OS and environment as Python is available in a wide variety of platforms.

Python Supporting Systems –

Python can be used on any of the following –

- ❖ Unix (Solaris, Linux, FreeBSD, AIX, HP/UX, SunOS, IRIX, etc.)
- ❖ Win 9x/NT/2000

- Macintosh (Intel, PPC, 68K)
- OS/2
- DOS (multiple versions)
- PalmOS
- Nokia mobile phones
- Windows CE
- Acorn/RISC OS
- BeOS
- Amiga
- VMS/OpenVMS
- QNX
- VxWorks
- Psion

Looking for Python?

Almost all updated and current source code, as well as binaries, documentation, or news will be available on the official website of Python which is: https://www.python.org/. Also, the documentation is available in here: https://www.python.org/doc/. You have to go to this link https://www.python.org/downloads/ for downloading the latest release of Python. In this setup process, you will find Python for different OS, as well as choose specific release of Python to install this in your system.

Double-click the executable file which is downloaded; the following window will open. Select Customize installation and proceed. The following window shows all the optional features. See if all the features are required to be installed or not and then checked it by default; you need to click next for continuing the installation.

Installing Python for Linux –

The steps are:

i) First you have to update your APT Repository. You have to type the command in your terminal:

```
$ apt-get update
```

ii) Now, install Python by typing the command –

```
$ apt-get install python3.6
```

iii) Now, verify Python by typing

```
$ python
```

For Python3 type the command goes something like this.

```
$ python3
```

Installing Python in Macintosh systems –

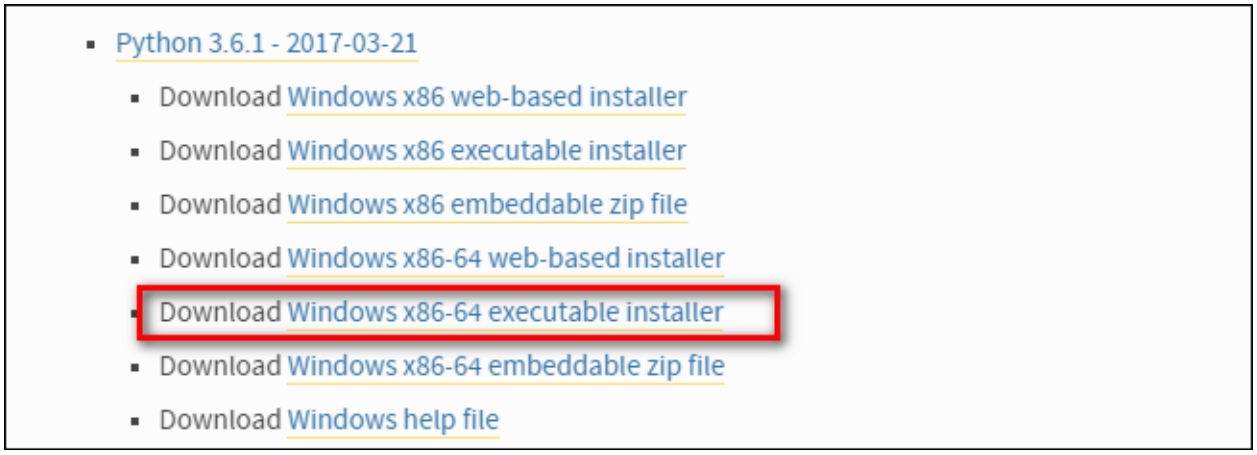
You do not need to install or configure anything else for using Python 2. These instructions are documented in the installation of Python 3. The version of Python which ships with OS-X becomes good for learning.

But if you want to have a stable release for Python, you will need to install GCC first. GCC can be obtained by downloading Xcode (https://developer.apple.com/xcode/), the smaller Command Line Tools (https://developer.apple.com/downloads/) (for this you must have an Apple account) or even the smaller OSX-GCC-Installer (https://github.com/kennethreitz/osx-gcc-installer#readme) package.

Installing Python in Windows 10

Python doesn't come pre-packaged with Windows, but that doesn't mean Windows users won't find the flexible programming language useful. For this go to the official website of Python (https://www.python.org/downloads/windows/).

From there, you have to download your system compatible latest Python released version. It comes with both Windows x86-64 web-based installer as well as Windows x86 embeddable zip file / Windows x86 executable installer. Once the file is downloaded, you have to double click to install it.



- Python 3.6.1 - 2017-03-21
  - Download Windows x86 web-based installer
  - Download Windows x86 executable installer
  - Download Windows x86 embeddable zip file
  - Download Windows x86-64 web-based installer
  - Download Windows x86-64 executable installer
  - Download Windows x86-64 embeddable zip file
  - Download Windows help file

As soon as you run the setup file of the Python installer, you will see a screen where you will have to choose a radio button either “Install for all users” or “Install just for me”. Click Next > Next and let it install Python interpreter into your system.

Now to check whether Python has been installed in your system, you have to open command prompt in your system and type the following command –

```
python -v
```

Python 2 vs. Python 3 –

Even though Python 3 is the most recent version and generation of this language, many programmers still exercise Python 2.7 which is the final update to Python version 2, and was released in 2010. Programming languages continually grow as developers extend its functionality as well as iron out quirks which will cause problems for the developers. Python 3 was introduced in the year 2008 with the aim of creation of a programming language that is easier to apply and change the easy to handle strings for matching the demands placed on the language today.

Python 3.0 is basically unlike to its previous releases because it is the first Python release which is not well-suited with its older versions. Programmers typically do not have to be concerned regarding the minor updates (e.g. from 2.6 to 2.7) as they frequently change the internal workings of Python and do not need programmers for changing their syntax. The modification between Python 2.7 and Python 3.0 is much more noteworthy — syntaxes that work in Python 2.7 may require to be written in a changed way to work in Python 3.0.

History of python:

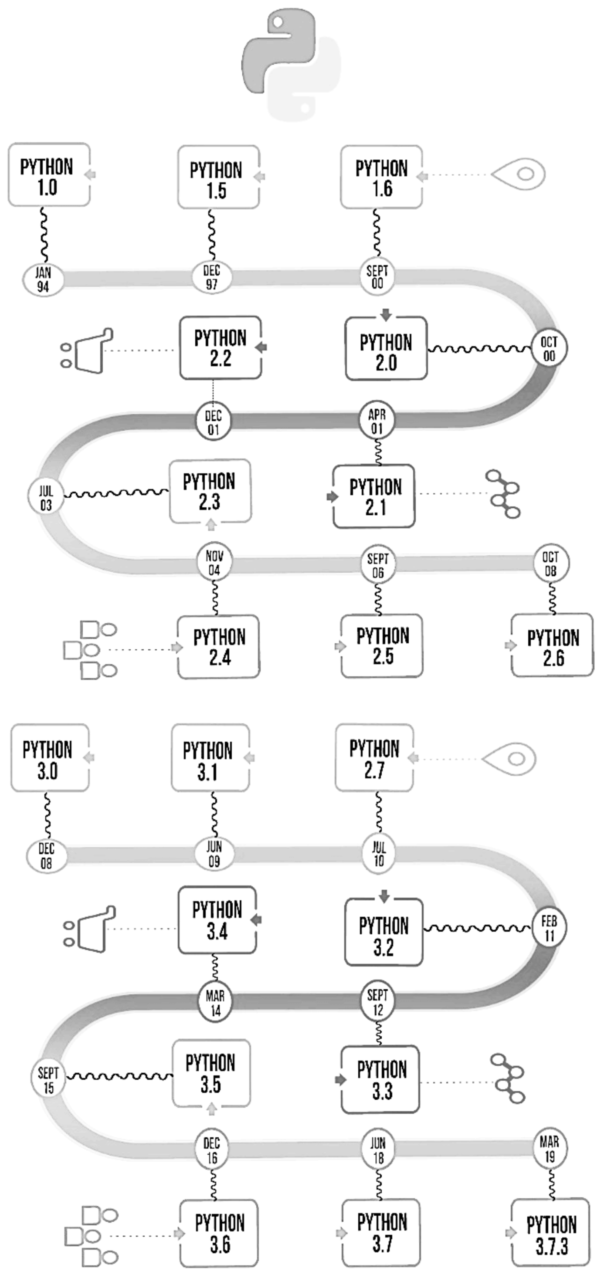
Python is a widely used general-purpose, high-level programming language. It was initially designed by Guido van Rossum in 1991 and developed by Python Software Foundation. It was mainly developed for emphasis on code readability, and its syntax allows programmers to express concepts in fewer lines of code.

Let us explain it in details –

In the late 1980s, history was about to be written. It was that time when working on Python started. Soon after that, Guido Van Rossum began doing its application-based work in December of 1989 by at Centrum Wiskunde & Informatica (CWI) which is situated in Netherland. It was started firstly as a hobby project because he was looking for an interesting project to keep him occupied during Christmas. The programming language which Python is said to have succeeded is ABC Programming Language, which had the interfacing with the Amoeba Operating System and had the feature of exception handling. He had already helped to create ABC earlier in his career and he had seen some issues with ABC but liked most of the features. After that what he did as really very clever. He had taken the syntax of ABC, and some of its good features. It came with a lot of complaints too, so he fixed those issues completely and had created a good

scripting language which had removed all the flaws. The inspiration for the name came from BBC’s TV Show – ‘Monty Python’s Flying Circus’, as he was a big fan of the TV show and also he wanted a short, unique and slightly mysterious name for his invention and hence he named it Python! He was the “Benevolent dictator for life” (BDFL) until he stepped down from the position as the leader on 12th July 2018. For quite some time he used to work for Google, but currently, he is working at Dropbox.
The language was finally released in 1991. When it was released, it used a lot fewer codes to express the concepts, when we compare it with Java, C++ & C. Its design philosophy was quite good too. Its main objective is to provide code readability and advanced developer productivity. When it was released it had more than enough capability to provide classes with inheritance, several core data types exception handling and functions.

Following are the illustrations of different versions of Python along with the timeline.



Why use Python / Benefits –

i. Presence of 3rd party modules: Python Package Index (PyPI) has a number of third-party modules which helps in making Python capable of cooperating with the majority of the other languages as well as platforms.
ii. Extensive Support Libraries: This language offers a wide standard of libraries that includes areas such as internet protocols, string operations, web services tools as well as OS interfaces.
iii. Open Source and Community Development: Python has been developed under an OSI-approved open source license that makes it free to use and distribute for personal as well as commercial purpose.
iv. Easy to learn and understand.
v. Python has built-in list and dictionary as data structures which makes it very user friendly for constructing runtime data structure.
vi. Python has a clean object-oriented design approach, and enhanced process management capabilities, along with easy text processing capabilities.
vii. Python can be used in different technical domains such as data science, machine learning, robotics, game development, application development, embedded system development etc.

Why Use Python?

Python's expansive library of open source data analysis tools, web frameworks, and testing instruments make its ecosystem one of the largest out of any programming community. Python is an accessible language for new programmers because the community provides many introductory resources. The language is also widely taught in universities and used for working with beginner-friendly devices such as the Raspberry Pi.

If you're learning about why to use Python you should also take a look at the best Python resources and read more about web development.

Python's programming language popularity

Several programming language popularity rankings exist. While it's possible to criticize that these guides are not exact, every ranking shows Python as a top programming language within the top ten, if not the top five of all languages.

The IEEE ranked Python as the #1 programming language in 2019, which continued its hot streak after ranking it #1 in 2018, #1 in 2017 and #3 top programming language in 2016. RedMonk's June 2019 ranking had Python at #3, which held consistent from previous years' rankings in 2018 and 2017.

Stack Overflow's community-created question and answer data confirms the incredible growth of the Python ecosystem and tries to determine why it growing so quickly with their own analysis. In the 2018 Stack Overflow developer survey the data indicated that Python was the fastest growing major programming language and that there is a close alignment between the languages and tools that developers choose to learn and the usage in developers' professional work.

The TIOBE Index a long-running language ranking, has Python moving up the charts to #3, climbing from just a few years ago.

The Popularity of Programming Language (PYPL), based on leading indicators from Google Trends search keyword analysis, shows Python at #1.

These rankings provide a rough measure for language popularity. They are not intended as a precise measurement tool to determine exactly how many developers are using a language. However, the aggregate view shows that Python remains a stable programming language with a growing ecosystem.

Why does the choice of programming language matter?

Programming languages have unique ecosystems, cultures and philosophies built around them. You will find friction with a community and difficulty in learning if your approach to programming varies from the philosophy of the programming language you've selected.

Python's culture values open source software, community involvement with local, national and international events and teaching to new programmers. If those values are also important to you and/or your organization then Python may be a good fit. The philosophy for Python is so strongly held that it's even embedded in the language as shown when the interpreter executes "import this" and displays The Zen of Python.

Invoking the Interpreter

The Python interpreter is usually installed as /usr/local/bin/python3.8 on those machines where it is available; putting /usr/local/bin in your Unix shell's search path makes it possible to start it by typing the command:

```
python3.8
```

to the shell. Since the choice of the directory where the interpreter lives is an installation option, other places are possible; check with your local Python guru or system administrator. (E.g., /usr/local/python is a popular alternative location.)

On Windows machines where you have installed Python from the Microsoft Store, the python3.8 command will be available. If you have the py.exe launcher installed, you can use the py command. See Excursus: Setting environment variables for other ways to launch Python.

Typing an end-of-file character (Control-D on Unix, Control-Z on Windows) at the primary prompt causes the interpreter to exit with a zero exit status. If that doesn't work, you can exit the interpreter by typing the following command: quit().

If you've installed Python in Windows using the default installation options, the path to the Python executable wasn't added to the Windows **Path variable**. The Path variable lists the directories that will be searched for executables when you type a command in the command prompt. By adding the path to the Python executable, you will be able to access **python.exe** by typing the **python** keyword (you won't need to specify the full path to the program).

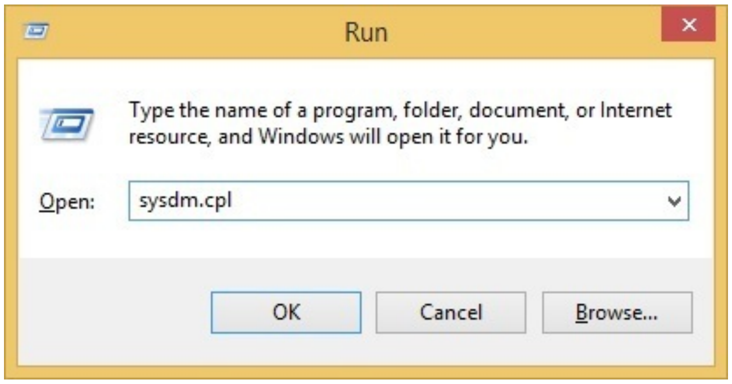
Consider what happens if we enter the **python** command in the command prompt and the path to that executable is not added to the Path variable:

```
C:\>python
'python' is not recognized as an internal or external command,
operable program or batch file.
```

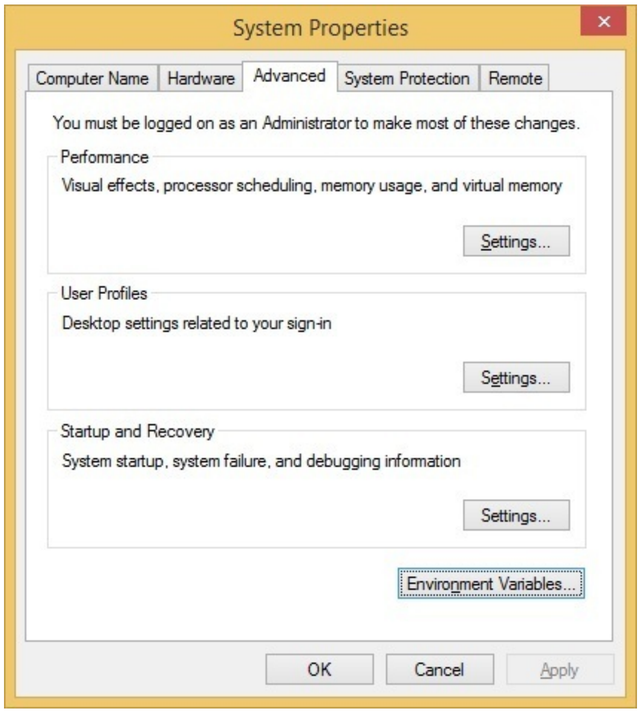
As you can see from the output above, the command was not found. To run **python.exe**, you need to specify the full path to the executable:

```
C:\>C:\Python34\python --version
Python 3.4.3
```

To add the path to the **python.exe** file to the Path variable, start the **Run** box and enter **sysdm.cpl**:


Run
Type the name of a program, folder, document, or Internet resource, and Windows will open it for you.
Open: sysdm.cpl
OK
Cancel
Browse...

This should open up the **System Properties** window. Go to the **Advanced** tab and click the **Environment Variables** button:



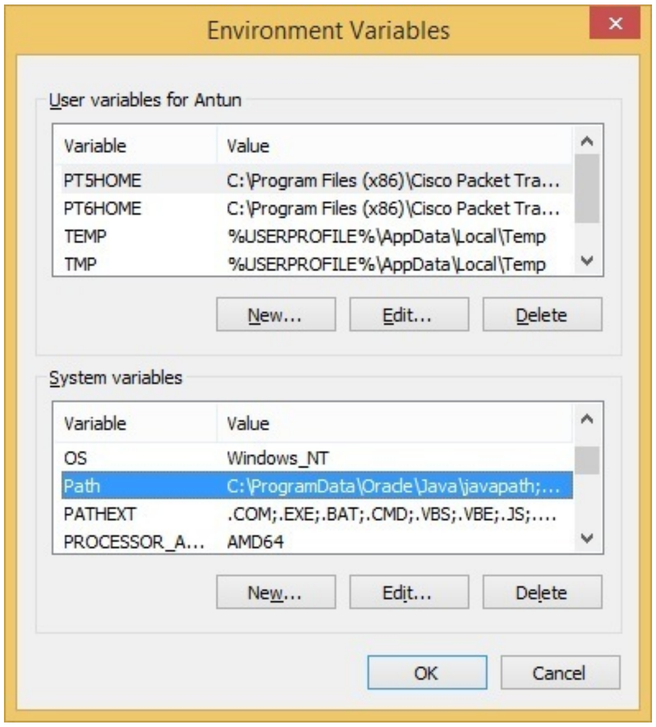
You must be logged on as an Administrator to make most of these changes.

Visual effects, processor scheduling, memory usage, and virtual memory

Desktop settings related to your sign-in

System startup, system failure, and debugging information

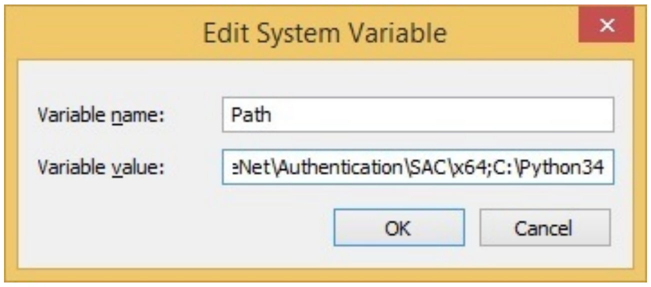
In the **System variable** window, find the **Path** variable and click **Edit**:



Variable	Value
PT5HOME	C:\Program Files (x86)\Cisco Packet Tra...
PT6HOME	C:\Program Files (x86)\Cisco Packet Tra...
TEMP	%USERPROFILE%\AppData\Local\Temp
TMP	%USERPROFILE%\AppData\Local\Temp

Variable	Value
OS	Windows_NT
Path	C:\ProgramData\Oracle\Java\javapath;...
PATHEXT	.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;....
PROCESSOR_A...	AMD64

Position your cursor at the end of the **Variable value** line and add the path to the **python.exe** file, preceeded with the semicolon character (**;**). In our example, we have added the following value: **;C:\Python34**



Path

:Net\Authentication\SAC\x64;C:\Python34

Close all windows. Now you can run **python.exe** without specifying the full path to the file:

```
C:>python --version

Python 3.4.3
```

Using the Python Interpreter

1 Invoking the Interpreter

The Python interpreter is usually installed as /usr/local/bin/python on those machines where it is available; putting /usr/local/bin in your Unix shell's search path makes it possible to start it by typing the command

```
python
```

to the shell. Since the choice of the directory where the interpreter lives is an installation option, other places are possible; check with your local Python guru or system administrator. (E.g., /usr/local/python is a popular alternative location.)

Typing an end-of-file character (Control-D on Unix, Control-Z on Windows) at the primary prompt causes the interpreter to exit with a zero exit status. If that doesn't work, you can exit the interpreter by typing the following commands: "import sys; sys.exit()".

The interpreter's line-editing features usually aren't very sophisticated. On Unix, whoever installed the interpreter may have enabled support for the GNU readline library, which adds more elaborate interactive editing and history features. Perhaps the quickest check to see whether command line editing is supported is typing Control-P to the first Python prompt you get. If it beeps, you have command line editing; see Appendix A for an introduction to the keys. If nothing appears to happen, or if P is echoed, command line editing isn't available; you'll only be able to use backspace to remove characters from the current line.

The interpreter operates somewhat like the Unix shell: when called with standard input connected to a tty device, it reads and executes commands interactively; when called with a file name argument or with a file as standard input, it reads and executes a script from that file.

A second way of starting the interpreter is "python -c command [arg] ...", which executes the statement(s) in command, analogous to the shell's -c option. Since Python statements often contain spaces or other characters that are special to the shell, it is best to quote command in its entirety with double quotes.

Some Python modules are also useful as scripts. These can be invoked using "python -m module [arg] ...", which executes the source file for module as if you had spelled out its full name on the command line.

Note that there is a difference between "python file" and "python <file". In the latter case, input requests from the program, such as calls to input() and raw_input(), are satisfied from file. Since this file has already been read until the end by the parser before the program starts executing, the program will encounter end-of-file immediately. In the former case (which is usually what you want) they are satisfied from whatever file or device is connected to standard input of the Python interpreter.

When a script file is used, it is sometimes useful to be able to run the script and enter interactive mode afterwards. This can be done by passing -i before the script. (This does not work if the script is read from standard input, for the same reason as explained in the previous paragraph.)

1.1 Argument Passing

When known to the interpreter, the script name and additional arguments thereafter are passed to the script in the variable sys.argv, which is a list of strings. Its length is at least one; when no script and no arguments are given, sys.argv[0] is an empty string. When the script name is given as '-' (meaning standard input), sys.argv[0] is set to '-'. When -c command is used, sys.argv[0] is set to '-c'. When -m module is used, sys.argv[0] is set to the full name of the located module. Options found after -c command or -m module are not consumed by the Python interpreter's option processing but left in sys.argv for the command or module to handle.

1.2 Interactive Mode

When commands are read from a tty, the interpreter is said to be in interactive mode. In this mode it prompts for the next command with the primary prompt, usually three greater-than signs (">>> "); for continuation lines it prompts with the secondary prompt, by default three dots

("... "). The interpreter prints a welcome message stating its version number and a copyright notice before printing the first prompt:

```
python
Python 1.5.2b2 (#1, Feb 28 1999, 00:02:06)  [GCC 2.8.1] on sunos5
Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam
>>>
```

Continuation lines are needed when entering a multi-line construct. As an example, take a look at this if statement:

```
>>> the_world_is_flat = 1
>>> if the_world_is_flat:
...     print "Be careful not to fall off!"
...
Be careful not to fall off!
```

2 *The Interpreter and Its Environment*

2.1 Error Handling

When an error occurs, the interpreter prints an error message and a stack trace. In interactive mode, it then returns to the primary prompt; when input came from a file, it exits with a nonzero exit status after printing the stack trace. (Exceptions handled by an except clause in a try statement are not errors in this context.) Some errors are unconditionally fatal and cause an exit with a nonzero exit; this applies to internal inconsistencies and some cases of running out of memory. All error messages are written to the standard error stream; normal output from executed commands is written to standard output.

Typing the interrupt character (usually Control-C or DEL) to the primary or secondary prompt cancels the input and returns to the primary prompt.2.1Typing an interrupt while a command is executing raises the KeyboardInterrupt exception, which may be handled by a try statement.

2.2 Executable Python Scripts

On BSD'ish Unix systems, Python scripts can be made directly executable, like shell scripts, by putting the line

```
#! /usr/bin/env python
```

(assuming that the interpreter is on the user's PATH) at the beginning of the script and giving the file an executable mode. The "#!" must be the first two characters of the file. On some platforms, this first line must end with a Unix-style line ending ("\n"), not a Mac OS ("\r") or Windows

("\r\n") line ending. Note that the hash, or pound, character, "#", is used to start a comment in Python.

The script can be given a executable mode, or permission, using the chmod command:

```
$ chmod +x myscript.py
```

How to Run Python Code Interactively

A widely used way to run Python code is through an interactive session. To start a Python interactive session, just open a command-line or terminal and then type in python, or python3 depending on your Python installation, and then hit Enter.

Here’s an example of how to do this on Linux:

```
$ python3
Python 3.6.7 (default, Oct 22 2018, 11:32:17)
[GCC 8.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The standard prompt for the interactive mode is >>>, so as soon as you see these characters, you’ll know you are in.

Now, you can write and run Python code as you wish, with the only drawback being that when you close the session, your code will be gone.

When you work interactively, every expression and statement you type in is evaluated and executed immediately:

```
>>>
>>> print('Hello World!')
Hello World!
>>> 2 + 5
7
>>> print('Welcome to Real Python!')
Welcome to Real Python!
```

An interactive session will allow you to test every piece of code you write, which makes it an awesome development tool and an excellent place to experiment with the language and test Python code on the fly.

To exit interactive mode, you can use one of the following options:

- quit() or exit(), which are built-in functions
- The Ctrl + Z and Enter key combination on Windows, or just Ctrl + D on Unix-like systems

Note: The first rule of thumb to remember when using Python is that if you're in doubt about what a piece of Python code does, then launch an interactive session and try it out to see what happens.

If you've never worked with the command-line or terminal, then you can try this:

- On Windows, the command-line is usually known as command prompt or MS-DOS console, and it is a program called cmd.exe. The path to this program can vary significantly from one system version to another.

  A quick way to get access to it is by pressing the Win + R key combination, which will take you to the *Run* dialog. Once you're there, type in cmd and press Enter.

- On GNU/Linux (and other Unixes), there are several applications that give you access to the system command-line. Some of the most popular are xterm, Gnome Terminal, and Konsole. These are tools that run a shell or terminal like Bash, ksh, csh, and so on.

  In this case, the path to these applications is much more varied and depends on the distribution and even on the desktop environment you use. So, you'll need to read your system documentation.

- On Mac OS X, you can access the system terminal from *Applications → Utilities → Terminal.*

How Does the Interpreter Run Python Scripts?

When you try to run Python scripts, a multi-step process begins. In this process the interpreter will:

1. Process the statements of your script in a sequential fashion
2. Compile the source code to an intermediate format known as bytecode

This bytecode is a translation of the code into a lower-level language that's platform-independent. Its purpose is to optimize code execution. So, the next time the interpreter runs your code, it'll bypass this compilation step.

Strictly speaking, this code optimization is only for modules (imported files), not for executable scripts.

3. Ship off the code for execution

   At this point, something known as a Python Virtual Machine (PVM) comes into action. The PVM is the runtime engine of Python. It is a cycle that iterates over the instructions of your bytecode to run them one by one.

   The PVM is not an isolated component of Python. It's just part of the Python system you've installed on your machine. Technically, the PVM is the last step of what is called the Python interpreter.

The whole process to run Python scripts is known as the **Python Execution Model**.

Python Editor and IDEs –

Writing Python using IDLE or the Python Shell is great for simple things, but those tools quickly turn larger programming projects into frustrating pits of despair. Using an IDE, or even just a good dedicated code editor, makes coding fun—but which one is best for you?

Fear not, Gentle Reader! We are here to help explain and demystify the myriad of choices available to you. We can't pick what works best for you and your process, but we can explain the pros and cons of each and help you make an informed decision.

To make things easier, we'll break our list into two broad categories of tools: the ones built exclusively for Python development and the ones built for general development that you can use for Python. We'll call out some Whys and Why Nots for each. Lastly, none of these options are mutually exclusive, so you can try them out on your own with very little penalty.

But first…

What Are IDEs and Code Editors?

An IDE (or Integrated Development Environment) is a program dedicated to software development. As the name implies, IDEs integrate several tools specifically designed for software development. These tools usually include:

- An editor designed to handle code (with, for example, syntax highlighting and auto-completion)
- Build, execution, and debugging tools
- Some form of source control

Most IDEs support many different programming languages and contain many more features. They can, therefore, be large and take time to download and install. You may also need advanced knowledge to use them properly.

In contrast, a dedicated code editor can be as simple as a text editor with syntax highlighting and code formatting capabilities. Most good code editors can execute code and control a debugger. The very best ones interact with source control systems as well. Compared to an IDE, a good dedicated code editor is usually smaller and quicker, but often less feature rich.

Requirements for a Good Python Coding Environment

So what things do we really need in a coding environment? Feature lists vary from app to app, but there are a core set of features that makes coding easier:

- Save and reload code files
  If an IDE or editor won't let you save your work and reopen everything later, in the same state it was in when you left, it's not much of an IDE.
- Run code from within the environment
  Similarly, if you have to drop out of the editor to run your Python code, then it's not much more than a simple text editor.
- Debugging support
  Being able to step through your code as it runs is a core feature of all IDEs and most good code editors.
- Syntax highlighting
  Being able to quickly spot keywords, variables, and symbols in your code makes reading and understanding code much easier.
- Automatic code formatting
  Any editor or IDE worth it's salt will recognize the colon at the end of a while or for statement, and know the next line should be indented.

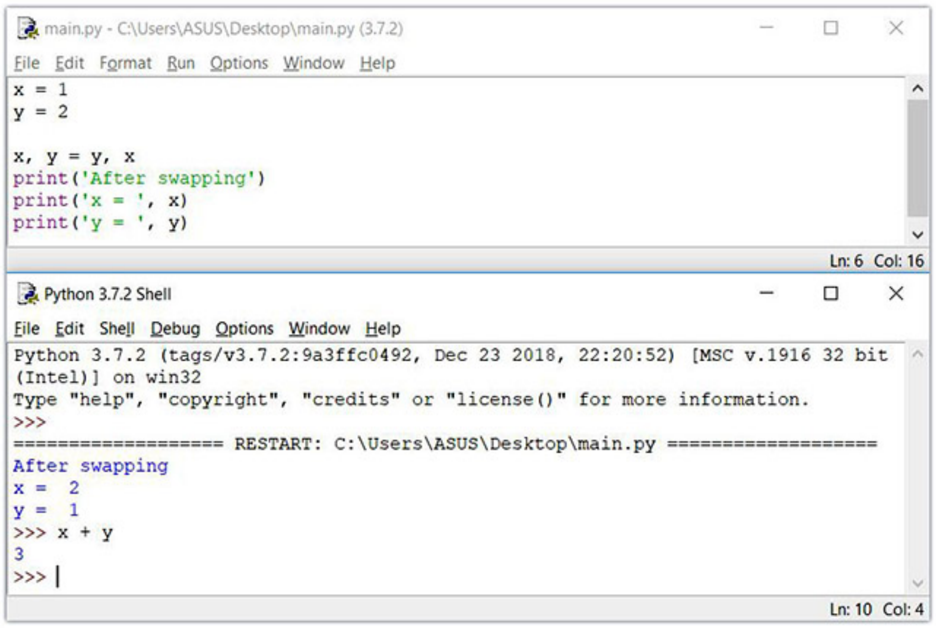
Of course, there are lots of other features you might want, like source code control, an extension model, build and test tools, language help, and so on. But the above list is what I'd see as "core features" that a good editing environment should support.

With these features in mind, let's take a look at some general-purpose tools we can use for Python development.

IDLE

When you install Python, IDLE is also installed by default. This makes it easy to get started in Python. Its major features include the Python shell window(interactive interpreter), auto-completion, syntax highlighting, smart indentation, and a basic integrated debugger.

IDLE is a decent IDE for learning as it's lightweight and simple to use. However, it's not for optimum for larger projects.


main.py - C:\Users\ASUS\Desktop\main.py (3.7.2)
File Edit Format Run Options Window Help
Ln: 6 Col: 16
Python 3.7.2 Shell
File Edit Shell Debug Options Window Help
Ln: 10 Col: 4

```
x = 1
y = 2

x, y = y, x
print('After swapping')
print('x = ', x)
print('y = ', y)
```

```
Python 3.7.2 (tags/v3.7.2:9a3ffc0492, Dec 23 2018, 22:20:52) [MSC v.1916 32 bit
(Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
=================== RESTART: C:\Users\ASUS\Desktop\main.py ===================
After swapping
x =  2
y =  1
>>> x + y
3
>>>
```

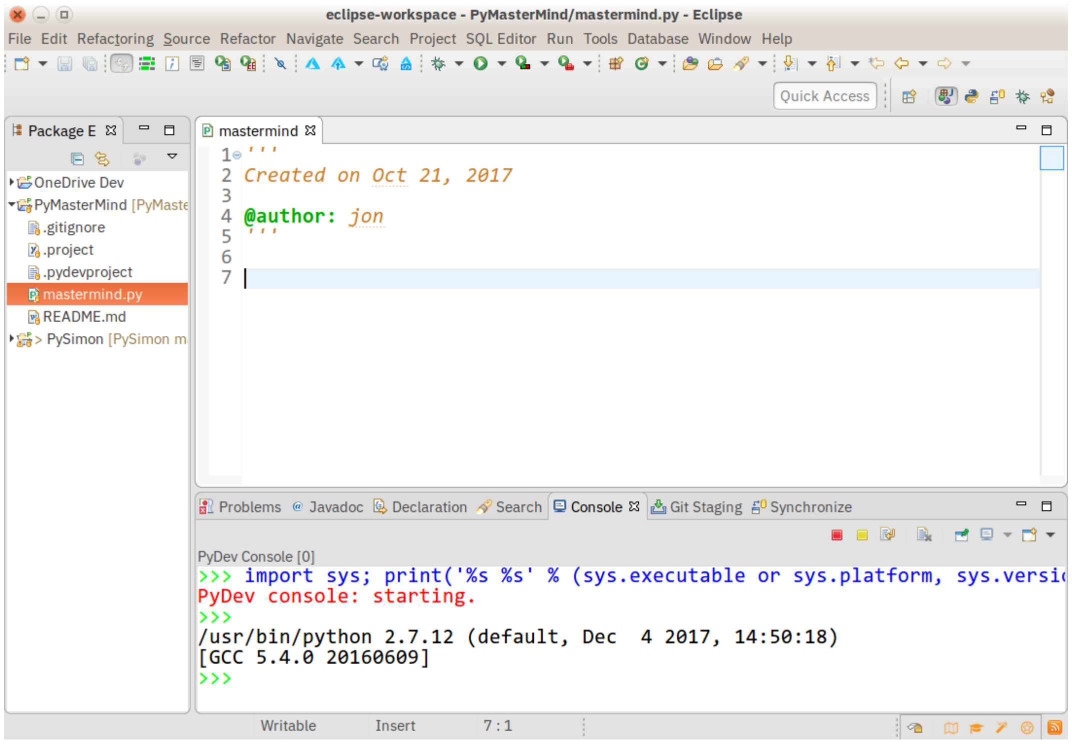
General Editors and IDEs with Python Support

Eclipse + PyDev

Category: IDE
Website: www.eclipse.org
Python tools: PyDev, www.pydev.org

If you've spent any amount of time in the open-source community, you've heard about Eclipse. Available for Linux, Windows, and OS X at, Eclipse is the de-facto open-source IDE for Java development. It has a rich marketplace of extensions and add-ons, which makes Eclipse useful for a wide range of development activities.

One such extension is PyDev, which enables Python debugging, code completion, and an interactive Python console. Installing PyDev into Eclipse is easy: from Eclipse, select Help, Eclipse Marketplace, then search for PyDev. Click Install and restart Eclipse if necessary.



```
'''
Created on Oct 21, 2017

@author: jon
'''

```

```
PyDev Console [0]
>>> import sys; print('%s %s' % (sys.executable or sys.platform, sys.versi
PyDev console: starting.
>>>
/usr/bin/python 2.7.12 (default, Dec  4 2017, 14:50:18)
[GCC 5.4.0 20160609]
>>>
```

Pros: If you've already got Eclipse installed, adding PyDev will be quicker and easier. PyDev is very accessible for the experienced Eclipse developer.

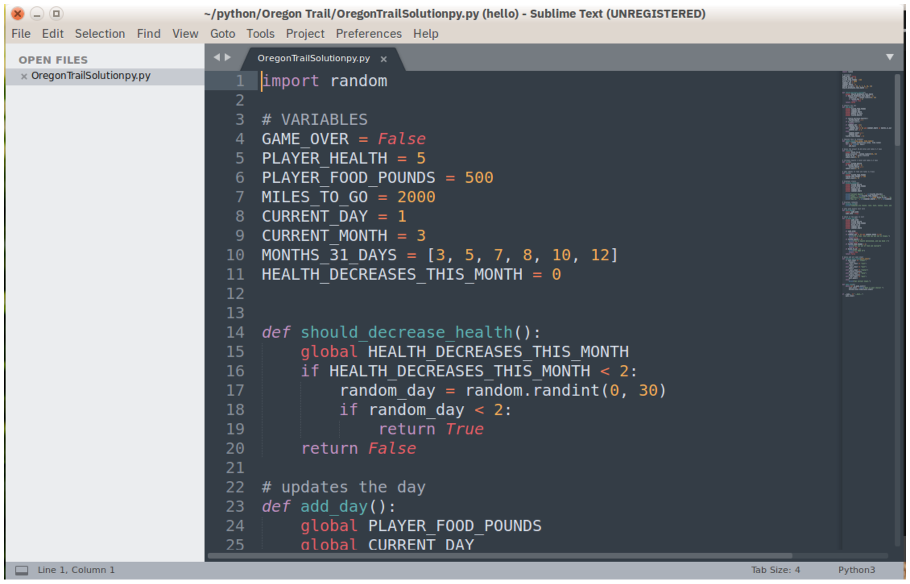
Cons: If you're just starting out with Python, or with software development in general, Eclipse can be a lot to handle. Remember when I said IDEs are larger and require more knowledge to use properly? Eclipse is all that and a bag of (micro)chips.

Sublime Text

Category: Code Editor
Website: http://www.sublimetext.com

Written by a Google engineer with a dream for a better text editor, Sublime Text is an extremely popular code editor. Supported on all platforms, Sublime Text has built-in support for Python code editing and a rich set of extensions (called packages) that extend the syntax and editing features.

Installing additional Python packages can be tricky: all Sublime Text packages are written in Python itself, and installing community packages often requires you to execute Python scripts directly in Sublime Text.


~/python/Oregon Trail/OregonTrailSolutionpy.py (hello) - Sublime Text (UNREGISTERED)
File Edit Selection Find View Goto Tools Project Preferences Help
OPEN FILES
OregonTrailSolutionpy.py
Line 1, Column 1
Tab Size: 4
Python3

```
import random

# VARIABLES
GAME_OVER = False
PLAYER_HEALTH = 5
PLAYER_FOOD_POUNDS = 500
MILES_TO_GO = 2000
CURRENT_DAY = 1
CURRENT_MONTH = 3
MONTHS_31_DAYS = [3, 5, 7, 8, 10, 12]
HEALTH_DECREASES_THIS_MONTH = 0


def should_decrease_health():
    global HEALTH_DECREASES_THIS_MONTH
    if HEALTH_DECREASES_THIS_MONTH < 2:
        random_day = random.randint(0, 30)
        if random_day < 2:
            return True
    return False

# updates the day
def add_day():
    global PLAYER_FOOD_POUNDS
    global CURRENT_DAY
```

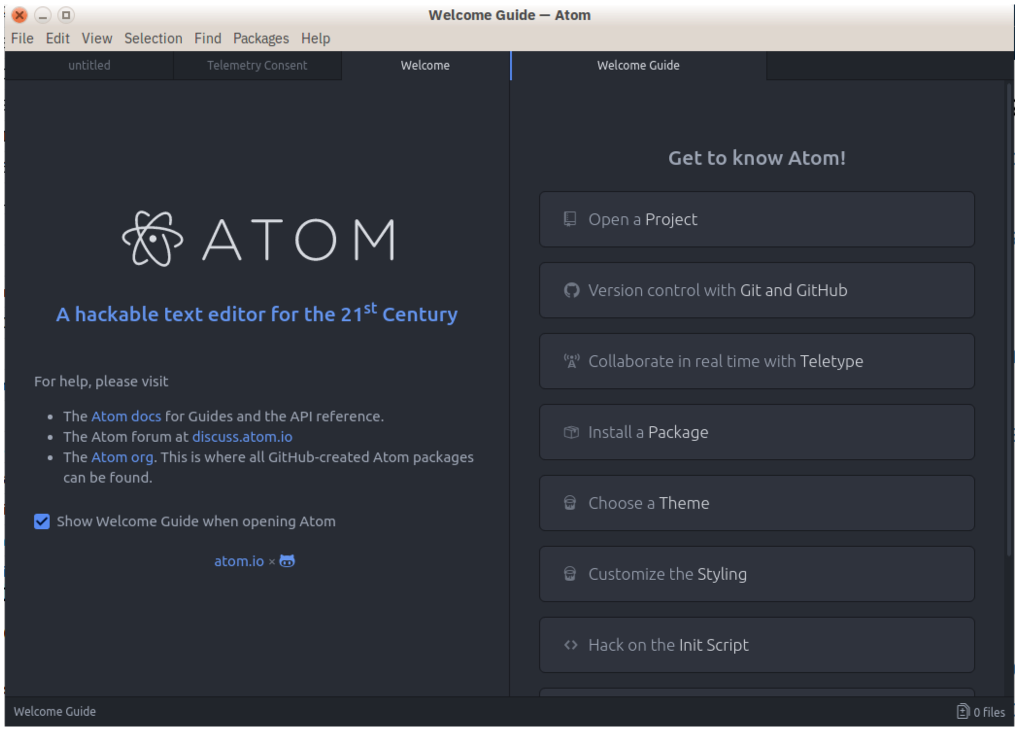
Pros: Sublime Text has a great following in the community. As a code editor, alone, Sublime Text is fast, small, and well supported.

Cons: Sublime Text is not free, although you can use the evaluation version for an indefinite period of time. Installing extensions can be tricky, and there's no direct support for executing or debugging code from within the editor.

To make the most of your Sublime Text setup, read our Python + Sublime Text setup guide and consider our in-depth video course that shows you how to craft an effective Python development setup with Sublime Text 3.

Atom
Category: Code Editor
Website: https://atom.io/

Available on all platforms, Atom is billed as the "hackable text editor for the 21st Century." With a sleek interface, file system browser, and marketplace for extensions, open-source Atom is built using Electron, a framework for creating desktop applications using JavaScript, HTML, and CSS. Python language support is provided by an extension that can installed when Atom is running.



A hackable text editor for the 21st Century

For help, please visit

- The Atom docs for Guides and the API reference.
- The Atom forum at discuss.atom.io
- The Atom org. This is where all GitHub-created Atom packages can be found.

✓ Show Welcome Guide when opening Atom

atom.io ×

Pros: It has broad support on all platforms, thanks to Electron. Atom is small, so it downloads and loads fast.

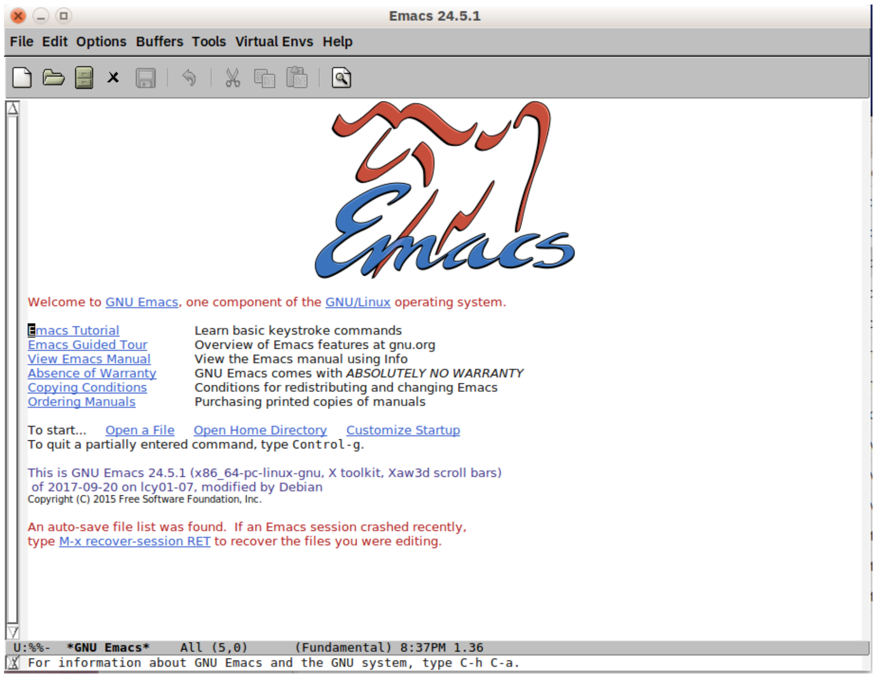
Cons: Build and debugging support aren't built-in but are community provided add-ons. Because Atom is built on Electron, it's always running in a JavaScript process and not as a native application.

GNU Emacs

Category: Code Editor
Website: https://www.gnu.org/software/emacs/

Back before the iPhone vs Android war, before the Linux vs Windows war, even before the PC vs Mac war, there was the Editor War, with GNU Emacs as one of the combatants. Billed as "the extensible, customizable, self-documenting, real-time display editor," GNU Emacs has been around almost as long as UNIX and has a fervent following.

Always free and available on every platform (in one form or another), GNU Emacs uses a form of the powerful Lisp programming language for customization, and various customization scripts exist for Python development.



Welcome to GNU Emacs, one component of the GNU/Linux operating system.

Emacs Tutorial	Learn basic keystroke commands
Emacs Guided Tour	Overview of Emacs features at gnu.org
View Emacs Manual	View the Emacs manual using Info
Absence of Warranty	GNU Emacs comes with *ABSOLUTELY NO WARRANTY*
Copying Conditions	Conditions for redistributing and changing Emacs
Ordering Manuals	Purchasing printed copies of manuals

To start... Open a File Open Home Directory Customize Startup
To quit a partially entered command, type `Control-g`.

This is GNU Emacs 24.5.1 (x86_64-pc-linux-gnu, X toolkit, Xaw3d scroll bars)
of 2017-09-20 on lcy01-07, modified by Debian
Copyright (C) 2015 Free Software Foundation, Inc.

An auto-save file list was found. If an Emacs session crashed recently,
type M-x recover-session RET to recover the files you were editing.

```
U:%%-  *GNU Emacs*    All (5,0)      (Fundamental) 8:37PM 1.36
For information about GNU Emacs and the GNU system, type C-h C-a.
```

Pros: You know Emacs, you use Emacs, you love Emacs. Lisp is a second language, and you know the power it gives you means you can do anything.

Cons: Customization means writing (or copy/pasting) Lisp code into various script files. If it's not already provided, you may have to learn Lisp to figure out how to do it.

Plus, you know that Emacs would be a great operating system, if it only had a good text editor…

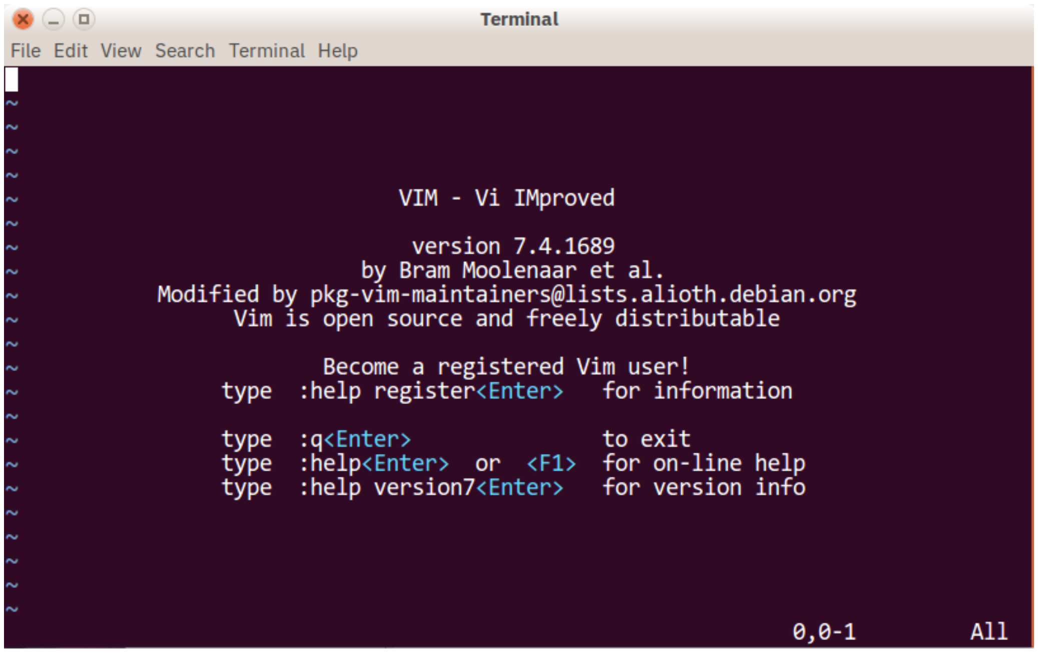
Be sure to consult our Python + Emacs setup guide to make the most of this setup.

Vi / Vim

Category: Code Editor
Website: https://www.vim.org/

On the other side of the Text Editor War stands VI (aka VIM). Included by default on almost every UNIX system and Mac OS X, VI has an equally fervent following.

VI and VIM are modal editors, separating the viewing of a file from the editing of a file. VIM includes many improvements on the original VI, including an extensibility model and in-place code building. VIMScripts are available for various Python development tasks.



```
~
~
~
~
~
~                         VIM - Vi IMproved
~
~                          version 7.4.1689
~                      by Bram Moolenaar et al.
~        Modified by pkg-vim-maintainers@lists.alioth.debian.org
~              Vim is open source and freely distributable
~
~                     Become a registered Vim user!
~             type  :help register<Enter>   for information
~
~             type  :q<Enter>               to exit
~             type  :help<Enter>  or  <F1>  for on-line help
~             type  :help version7<Enter>   for version info
~
~
~
~
~
                                                  0,0-1         All
```

Pros: You know VI, you use VI, you love VI. VIMScripts don't scare you, and you know you bend it to your will.

Cons: Like Emacs, you're not comfortable finding or writing your own scripts to enable Python development, and you're not sure how a modal editor is supposed to work.

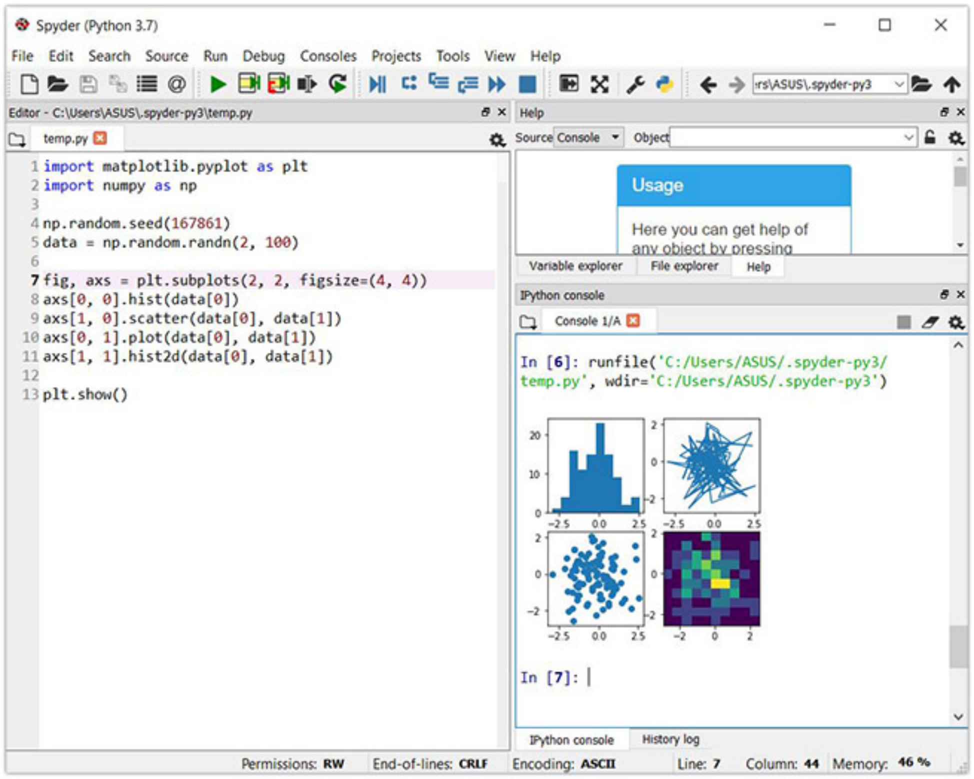
Plus, you know that VI would be a great text editor, if only it had a decent operating system.

If you're going with this combination, check out our Python + VIM setup guide with tips and plugin recommendations.

Spyder: Spyder is an open-source IDE usually used for scientific development.

The easiest way to get up and running up with Spyder is by installing Anaconda distribution. If you don't know, Anaconda is a popular distribution for data science and machine learning. The Anaconda distribution includes hundreds of packages including NumPy, Pandas, scikit-learn, matplotlib and so on.

Spyder has some great features such as autocompletion, debugging and iPython shell. However, it lacks in features compared to PyCharm.


Spyder (Python 3.7)
File Edit Search Source Run Debug Consoles Projects Tools View Help
Editor - C:\Users\ASUS\.spyder-py3\temp.py
temp.py
Help
Source Console Object
Variable explorer File explorer Help
IPython console
Console 1/A
IPython console History log

```
import matplotlib.pyplot as plt
import numpy as np

np.random.seed(167861)
data = np.random.randn(2, 100)

fig, axs = plt.subplots(2, 2, figsize=(4, 4))
axs[0, 0].hist(data[0])
axs[1, 0].scatter(data[0], data[1])
axs[0, 1].plot(data[0], data[1])
axs[1, 1].hist2d(data[0], data[1])

plt.show()
```

Usage

Here you can get help of any object by pressing

```
In [6]: runfile('C:/Users/ASUS/.spyder-py3/
temp.py', wdir='C:/Users/ASUS/.spyder-py3')

In [7]:
```

Permissions: RW End-of-lines: CRLF Encoding: ASCII Line: 7 Column: 44 Memory: 46 %

Built-in Python Functions –

The Python interpreter has a number of functions that are always available for use. These functions are called built-in functions. For example, print() function prints the given object to the standard output device (screen) or to the text stream file.

In Python 3.6 (latest version), there are 68 built-in functions. They are listed below alphabetically along with brief description.

Method	Description
• Python abs()	returns absolute value of a number
• Python all()	returns true when all elements in iterable is true
• Python any()	Checks if any Element of an Iterable is True
• Python ascii()	Returns String Containing Printable Representation
• Python bin()	converts integer to binary string
• Python bool()	Converts a Value to Boolean
• Python bytearray()	returns array of given byte size
• Python bytes()	returns immutable bytes object
• Python callable()	Checks if the Object is Callable

Python chr()	Returns a Character (a string) from an Integer
Python classmethod()	returns class method for given function
Python compile()	Returns a Python code object
Python complex()	Creates a Complex Number
Python delattr()	Deletes Attribute From the Object
Python dict()	Creates a Dictionary
Python dir()	Tries to Return Attributes of Object
Python divmod()	Returns a Tuple of Quotient and Remainder
Python enumerate()	Returns an Enumerate Object
Python eval()	Runs Python Code Within Program
Python exec()	Executes Dynamically Created Program
Python filter()	constructs iterator from elements which are true
Python float()	returns floating point number from number, string
Python format()	returns formatted representation of a value
Python frozenset()	returns immutable frozenset object
Python getattr()	returns value of named attribute of an object
Python globals()	returns dictionary of current global symbol table
Python hasattr()	returns whether object has named attribute
Python hash()	returns hash value of an object
Python help()	Invokes the built-in Help System
Python hex()	Converts to Integer to Hexadecimal
Python id()	Returns Identify of an Object
Python input()	reads and returns a line of string
Python int()	returns integer from a number or string
Python isinstance()	Checks if a Object is an Instance of Class
Python issubclass()	Checks if a Object is Subclass of a Class
Python iter()	returns an iterator
Python len()	Returns Length of an Object
Python list()	creates a list in Python
Python locals()	Returns dictionary of a current local symbol table
Python map()	Applies Function and Returns a List
Python max()	returns the largest item
Python memoryview()	returns memory view of an argument
Python min()	returns the smallest value
Python next()	Retrieves next item from the iterator

- Python object()		creates a featureless object
- Python oct()	returns the octal representation of an integer
- Python open()	Returns a file object
- Python ord()	returns an integer of the Unicode character
- Python pow()	returns the power of a number
- Python print()	Prints the Given Object
- Python property()		returns the property attribute
- Python range()return sequence of integers between start and stop
- Python repr()	returns a printable representation of the object
- Python reversed()		returns the reversed iterator of a sequence
- Python round()		rounds a number to specified decimals
- Python set()	constructs and returns a set
- Python setattr()		sets the value of an attribute of an object
- Python slice()	returns a slice object
- Python sorted()		returns a sorted list from the given iterable
- Python staticmethod()transforms a method into a static method
- Python str()	returns the string version of the object
- Python sum()	Adds items of an Iterable
- Python super()Returns a proxy object of the base class
- Python tuple()	Returns a tuple
- Python type()	Returns the type of the object
- Python vars()	Returns the __dict__ attribute
- Python zip()	Returns an iterator of tuples
- Python __import__()	Function called by the import statement

Strings –

What is String in Python?

A string is a sequence of characters. A character is simply a symbol. For example, the English language has 26 characters. Computers do not deal with characters, they deal with numbers (binary). Even though you may see characters on your screen, internally it is stored and manipulated as a combination of 0's and 1's.

This conversion of character to a number is called encoding, and the reverse process is decoding. ASCII and Unicode are some of the popular encoding used.

In Python, string is a sequence of Unicode character. Unicode was introduced to include every character in all languages and bring uniformity in encoding. You can learn more about Unicode from here.

How to create a string in Python?

Strings can be created by enclosing characters inside a single quote or double quotes. Even triple quotes can be used in Python but generally used to represent multiline strings and docstrings.

script.py

```
# all of the following are equivalent
my_string = 'Hello'
print(my_string)

my_string = "Hello"
print(my_string)

my_string = '''Hello'''
print(my_string)

# triple quotes string can extend multiple lines
my_string = """Hello, welcome to
        the world of Python"""
print(my_string)
```

When you run the program, the output will be:

```
Hello
Hello
Hello
Hello, welcome to
        the world of Python
```

How to access characters in a string?

We can access individual characters using indexing and a range of characters using slicing. Index starts from 0. Trying to access a character out of index range will raise an IndexError. The index must be an integer. We can't use float or other types, this will result into TypeError.

Python allows negative indexing for its sequences.

The index of -1 refers to the last item, -2 to the second last item and so on. We can access a range of items in a string by using the slicing operator (colon).

```
str = 'programming'
print('str = ', str)

#first character
print('str[0] = ', str[0])

#last character
print('str[-1] = ', str[-1])

#slicing 2nd to 5th character
print('str[1:5] = ', str[1:5])

#slicing 6th to 2nd last character
print('str[5:-2] = ', str[5:-2]
```

Python Literals

Literals can be defined as a data that is given in a variable or constant. Python support the following literals:

I. String literals: String literals can be formed by enclosing a text in the quotes. We can use both single as well as double quotes for a String.

Eg:
"Aman" , '12345'

Types of Strings:

There are two types of Strings supported in Python:

a) Single line String- Strings that are terminated within a single line are known as Single line Strings.

Eg:

```
>>>                                                    text1='hello'
```

b) Multi line String- A piece of text that is spread along multiple lines is known as Multiple line String.

There are two ways to create Multiline Strings:

1). Adding black slash at the end of each line.

Eg:

```
>>> text1='hello\
user'
>>> text1
'hellouser'
>>>
```

2)Using triple quotation marks:-

Eg:

```
>>> str2=''''''welcome
to
SSSIT'''
>>> print str2
welcome
to
SSSIT
>>>
```

Numeric literals:

Numeric Literals are immutable. Numeric literals can belong to following four different numerical types.

int(signed integers) Long(long integers) float(floating point) Complex(complex)

Numbers(can be both positive and negative) with no fractional part.eg: 100 Integers of unlimited size followed by lowercase or uppercase L eg: 87032845L Real numbers with both integer and fractional part eg: -26.2 In the form of a+bj where a forms the real part and b forms the imaginary part of complex number. eg: 3.14j

III. Boolean literals:

A Boolean literal can have any of the two values: True or False.

Special literals:

Python contains one special literal i.e., None.

None is used to specify to that field that is not created. It is also used for end of lists in Python.

```
Eg:
>>> val1=10
>>> val2=None
>>> val1
10
>>> val2
>>> print val2
None
>>>
```

Literal Collections.

Collections such as tuples, lists and Dictionary are used in Python.

List:

List contain items of different data types. Lists are mutable i.e., modifiable.
The values stored in List are separated by commas(,) and enclosed within a square brackets([]). We can store different type of data in a List.
Value stored in a List can be retrieved using the slice operator([] and [:]).
The plus sign (+) is the list concatenation and asterisk(*) is the repetition operator.

Eg:

```
>>> list=['aman',678,20.4,'saurav']
>>> list1=[456,'rahul']
>>> list
['aman', 678, 20.4, 'saurav']
>>> list[1:3]
[678, 20.4]
>>> list+list1
['aman', 678, 20.4, 'saurav', 456, 'rahul']
```

```
>>> list1*2
[456, 'rahul', 456, 'rahul']
>>>
```

Math Operators in Python –

Python supports all of the math operations that you would expect. The basic ones are addition, subtraction, multiplication, and division. Other ones include the exponentiation and modulo operators, which you will see in a moment.

Addition & Subtraction

```
a = 4
b = 2
c = a + b
print(c)
```

Even though we used variables for the addition, we could just as well have entered numbers instead.

As I am sure you can imagine, the above prints out the number 6.

Division

```
print(c / 2)
```

Notice how I printed out the result of the division — being an expression — directly, instead of storing the result within a variable first. When printing out values, we are using a function named print. We haven’t talked about functions yet, so we will get back to that later. The point is that this function takes an argument. For this argument, we can pass in a value directly, such as “Hello”, 22, or a mathematical expression as in this case. We will get back to all of this later, so don’t worry if that went over your head.

Either way, the above line of code prints out the number 3.0. Note that this number is actually of the type float — being numbers with decimals — instead of int. In this case we are dividing two integers (i.e. whole numbers), but that might not necessarily be the case. So the fact that divisions always result in floating point numbers is just convenient for us in case we need to perform some further operations on the number.

Multiplication

There is nothing special about multiplication in Python; we just use an asterisk (*) in the same way we did with addition, subtraction, and division.

```
print(2 * 2)
```

It is to be noted that if we multiply by at least one decimal number, the result is a float value.

Exponential Operator

Apart from the most common mathematical operators — being +, –, * and /) — Python also provides a handy operator for working with exponents. Consider the following example.

```
print(2 ** 10)
```

This raises 2 to the power of 10, also noted as 210, where 10 is the exponent. Without getting too much into math, this is the equivalent of multiplying 2 by itself 10 times, i.e. 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2. As you can see, using the exponential operator is much more convenient than typing all of that out manually. The result of the above is 1024.

Modulo Operator

Another math operator, is the modulo operator. What this operator does, is that it performs a division and then returns the remainder of the division. Consider the following examples.

```
print(4 % 2)
print(4 % 3)
```

The first line outputs 0. What happens is that 4 is divided by 2, leaving a remainder of 0. The second line divides 4 by 3, which leaves a remainder of 1. Likewise, the expression 105 % 10 gives us the integer 5. Hopefully that makes sense.

So what is the point of this operator? Typically you are not interested in the remainder itself, but just determining whether one number is divisible by another number. We will get back to examples of this at a later point, but a simple example could be to highlight every second row of a table by checking if the current row number is divisible by two (i.e. the remainder is zero). Again, more on that later.

Floor Division

The last math operator that we will look at, is the floor division operator. Suppose that we divide 10 by 3, for instance. This division yields a floating point number (i.e. a decimal number) of 3.33 (I reduced the number of decimals for brevity). When using floor division, the result of a division is rounded down to the nearest integer value, i.e. to a whole number. Consider the following example, where the floor division is denoted by two slashes, i.e. //.

```
print(10 // 3)
```

The division itself results in the decimal number 3.33 (again, the actual result produces more decimals). This number is then rounded down to the nearest full number, being 3. So the result of the above is the integer 3.

But what if the division results in a negative number? Suppose that we divide -11 by 3, which gives -3.66. What would the result of the following then be?

```
print(-11 // 3)
```

The result would actually be -4. That’s because the number is rounded down, and since -3.66 is below zero, rounding down to the nearest integer (or full number) means rounding down to -4. So if the result of the division is less than zero, we round away from zero, so to speak.

String Formatting –

Python uses C-style string formatting to create new, formatted strings. The "%" operator is used to format a set of variables enclosed in a "tuple" (a fixed size list), together with a format string, which contains normal text together with "argument specifiers", special symbols like "%s" and "%d".

Let's say you have a variable called "name" with your user name in it, and you would then like to print(out a greeting to that user.)

```
# This prints out "Hello, John!"
name = "John"
print("Hello, %s!" % name)
```

To use two or more argument specifiers, use a tuple (parentheses):

```
# This prints out "John is 23 years old."
name = "John"
age = 23
print("%s is %d years old." % (name, age))
```

Command line parameters –

Accessing Command Line Arguments

The Python sys module provides access to any of the command-line arguments via sys.argv. It solves two purposes:

- sys.argv: is the list of command line arguments
- len(sys.argv): is the number of command line arguments that you have in your command line
- sys.argv[0]: is the program, i.e. script name

You can execute Python in this way:

```
$python Commands.py inp1, inp2 inp3
```

```
import sys
print ‘Number of arguments:’, len (sys.argv), ‘arguments.’
print ‘Argument List:’, str(sys.argv)
```

It will produce the following output:

```
Number of arguments: 4 arguments.

Argument List: [‘sample.py’, ‘inp1’, ‘inp2’, ‘inp3’]
```

FLOW CONTROL

What is Control Flow?

In computer programming, control flow or flow of control is the order function calls, instructions, and statements are executed or evaluated when a program is running. Many programming languages have what are called control flow statements, which are used to determine what section of code is run in a program at any time.

Python Indentation

Most of the programming languages like C, C++, Java use braces { } to define a block of code. Python uses indentation.

A code block (body of a function, loop etc.) starts with indentation and ends with the first un-indented line. The amount of indentation is up to you, but it must be consistent throughout that block.

Generally, four whitespaces are used for indentation and is preferred over tabs. Here is an example.

```
for i in range(1,11):
   print(i)
   if i == 5:
      break
```

The enforcement of indentation in Python makes the code look neat and clean. This results into Python programs that look similar and consistent. Indentation can be ignored in line continuation. But it's a good idea to always indent. It makes the code more readable. For example:

```
if True:
   print('Hello')
   a = 5
```

and

```
if True: print('Hello'); a = 5
```

if and elif Statements –

In Python, If Statement is used for decision making. It will run the body of code only when IF statement is true. When you want to justify one condition while the other condition is not true, then you use "if statement".

Syntax:

```
if expression
 Statement
else
```

Statement

Decision making is required when we want to execute a code only if a certain condition is satisfied. The if...elif...else statement is used in Python for decision making.

Python **if Statement Syntax**

```
if test expression:

  statement(s)
```

Here, the program evaluates the test expression and will execute statement(s) only if the text expression is True.
If the text expression is False, the statement(s) is not executed.
In Python, the body of the if statement is indicated by the indentation. Body starts with an indentation and the first un-indented line marks the end.
Python interprets non-zero values as True. None and 0 are interpreted as False.

Python if Statement Flowchart

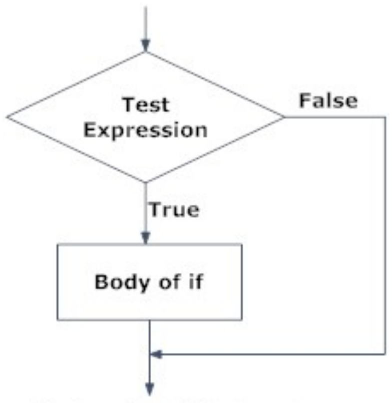


Fig: Operation of if statement

Example: Python if Statement

```
num = 3
if num > 0:
  print(num, "is a positive number.")
print("This is always printed.")

num = -1
if num > 0:
    print(num, "is a positive number.")
print("This is also always printed.")
```

Python if...else Statement

Syntax of if...else

```
if test expression:

    Body of if

else:

    Body of else
```

The if..else statement evaluates test expression and will execute body of if only when test condition is True.
If the condition is False, body of else is executed. Indentation is used to separate the blocks.
Python if..else Flowchart

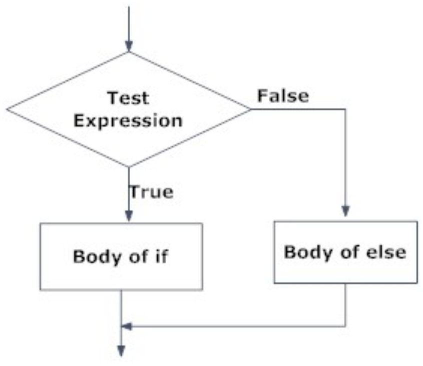


Fig: Operation of if...else statement

```
num = 3

# Try these two variations as well.
# num = -5
# num = 0

if num >= 0:
   print("Positive or Zero")
else:
   print("Negative number")
```

Python Nested if statements –

We can have a if...elif...else statement inside another if...elif...else statement. This is called nesting in computer programming. Any number of these statements can be nested inside one another. Indentation is the only way to figure out the level of nesting.

```
num = float(input("Enter a number: "))
if num >= 0:
   if num == 0:
      print("Zero")
   else:
      print("Positive number")
else:
   print("Negative number")
```

Loops:

The while loop in Python is used to iterate over a block of code as long as the test expression (condition) is true.

We generally use this loop when we don't know beforehand, the number of times to iterate.

Syntax of while Loop in Python

```
while test_expression:
    Body of while
```

In while loop, test expression is checked first. The body of the loop is entered only if the test_expression evaluates to True. After one iteration, the test expression is checked again. This process continues until the test_expression evaluates to False.

In Python, the body of the while loop is determined through indentation.

Body starts with indentation and the first un-indented line marks the end.

Python interprets any non-zero value as True. None and 0 are interpreted as False.

Flowchart of while Loop

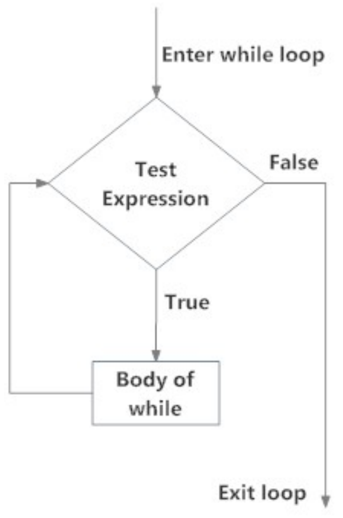


Fig: operation of while loop

Example: Python while Loop

```
n = 10

# initialize sum and counter
sum = 0
i = 1

while i <= n:
    sum = sum + i
    i = i+1    # update counter

# print the sum
print("The sum is", sum)
```

while loop with else –

Same as that of for loop, we can have an optional else block with while loop as well. The else part is executed if the condition in the while loop evaluates to False.

The while loop can be terminated with a break statement. In such case, the else part is ignored. Hence, a while loop's else part runs if no break occurs and the condition is false.

```
counter = 0

while counter < 3:
    print("Inside loop")
    counter = counter + 1
else:
    print("Inside else")
```

While loop with List –

```
a = ["fizz", "baz", "buzz"]
while a:
    print(a.pop(-1))
```

What is for loop in Python?

The for loop in Python is used to iterate over a sequence (list, tuple, string) or other iterable objects. Iterating over a sequence is called traversal.

Syntax of for Loop

```
for val in sequence:

        Body of for
```

Here, val is the variable that takes the value of the item inside the sequence on each iteration. Loop continues until we reach the last item in the sequence. The body of for loop is separated from the rest of the code using indentation.
Flowchart of for Loop

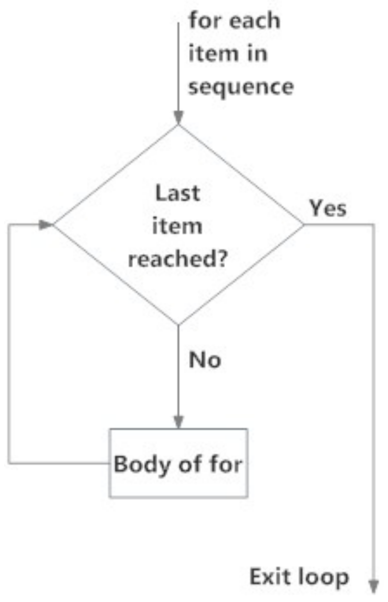


Fig: operation of for loop

Example: Python for Loop

```
numbers = [6, 5, 3, 8, 4, 2, 5, 4, 11]
# variable to store the sum
sum = 0
# iterate over the list
for val in numbers:
    sum = sum+val
# Output: The sum is 48
print("The sum is", sum)
```

The range() function: -

We can generate a sequence of numbers using range() function. range(10) will generate numbers from 0 to 9 (10 numbers).

We can also define the start, stop and step size as range(start,stop,step size). step size defaults to 1 if not provided. This function does not store all the values in memory, it would be inefficient. So it remembers the start, stop, step size and generates the next number on the go.

To force this function to output all the items, we can use the function list().
The following example will clarify this.

```
# Output: range(0, 10)
print(range(10))

# Output: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
print(list(range(10)))

# Output: [2, 3, 4, 5, 6, 7]
print(list(range(2, 8)))

# Output: [2, 5, 8, 11, 14, 17]
print(list(range(2, 20, 3)))
```

We can use the range() function in for loops to iterate through a sequence of numbers. It can be combined with the len() function to iterate though a sequence using indexing. Here is an example.

Sequences & File Operations –

Lists:

The most basic data structure in Python is the sequence. Each element of a sequence is assigned a number - its position or index. The first index is zero, the second index is one, and so forth. Python has six built-in types of sequences, but the most common ones are lists and tuples, which we would see in this tutorial.

There are certain things you can do with all sequence types. These operations include indexing, slicing, adding, multiplying, and checking for membership. In addition, Python has built-in functions for finding the length of a sequence and for finding its largest and smallest elements.

Python Lists

The list is a most versatile datatype available in Python which can be written as a list of comma-separated values (items) between square brackets. Important thing about a list is that items in a list need not be of the same type. Creating a list is as simple as putting different comma-separated values between square brackets. For example –

```
list1 = ['physics', 'chemistry', 1997, 2000];
list2 = [1, 2, 3, 4, 5 ];
list3 = ["a", "b", "c", "d"]
```

Similar to string indices, list indices start at 0, and lists can be sliced, concatenated and so on.

Accessing Values in Lists –

To access values in lists, use the square brackets for slicing along with the index or indices to obtain value available at that index. For example –

```
list1 = ['physics', 'chemistry', 1997, 2000];
list2 = [1, 2, 3, 4, 5, 6, 7 ];
print "list1[0]: ", list1[0]
print "list2[1:5]: ", list2[1:5]
```

When the above code is executed, it produces the following result –

```
list1[0]:  physics
list2[1:5]:  [2, 3, 4, 5]
```

Update a Python Lists

You can update single or multiple elements of lists by giving the slice on the left-hand side of the assignment operator, and you can add to elements in a list with the append() method. For example –

```
list = ['physics', 'chemistry', 1997, 2000];
print "Value available at index 2 : "
print list[2]
list[2] = 2001;
print "New value available at index 2 : "
print list[2]
```

Delete List Elements:

To remove a list element, you can use either the del statement if you know exactly which element(s) you are deleting or the remove() method if you do not know. For example –

```
list1 = ['physics', 'chemistry', 1997, 2000];
print list1
del list1[2];
print "After deleting value at index 2 : "
print list1
```

Tuples:

There is another type of sequence data structure that are immutable objects. They are called tuples. The Python tuples are similar to that of lists except that Tuples cannot be changed unlike lists. In this chapter you will get to know about how to use tuples in a Python program.

Tuples –

These are sequence data structures where the values are put as comma separated values. Tuple values are enclosed within parenthesis and they are immutable objects of Python. Example:

```
tupl1 = ('C++', 'Python', 4, 2);
tupl2 = (8, 6, 4, 2 );
tupl3 = "g", "k", "r", "s";
```

An empty tuple is defined with 2 parentheses having nothing inside –

```
tupl = ();
```

For writing a tuple with a single value, you have to make use of a comma, even though there is only a single value within it, like this –

```
tupl = (62,);
```

Like string indices, tuple indices start from 0.

Accessing the Elements of Tuples –

For accessing the different values within a tuple, you have to implement the square brackets to slice it against the index or indices for obtaining the value available at that location. For example –

```
t1 = ('MIT', 'Illinois', 1974, 1982);
t2 = (9, 7, 5, 3, 1, 2, 4 );
print "t1[0]: ", t1[0];
print "t2[1:5]: ", t2[1:5];
```

Updating Tuples –

Tuples are immutable which means you cannot update or change the values of tuple elements. You are able to take portions of existing tuples to create new tuples as the following example demonstrates –

```
t1 = (6, 3.1416);
t2 = ('ghi', 'stu');
# These won't work for tuples
# t1[0] = 42;
# Hence, you have to create a new tuple like this
t3 = t1 + t2; #concatenation
print t3;
```

Delete Tuple Elements –

Removal of independent tuple elements is not likely. But there is a way to explicitly remove an entire tuple simply using the del statement. For example –

```
tpl = ('JavaScript', 'Go', 'ASP', 'CSharp');
print tpl;
del tpl;
print "After deleting tup : ";
print tpl;
```

Indexing and Slicing –

Indexing

To retrieve an element of the list, we use the *index operator* ([]):
my_list[0]'a'

Lists are "zero indexed", so [0] returns the zero-th (*i.e.* the left-most) item in the list, and [1] returns the one-th item (*i.e.* one item to the right of the zero-th item). Since there are 9 elements in our list ([0] through [8]), attempting to access my_list[9] throws an IndexError: list index out of range, since it is actually trying to get the tenth element, and there isn't one.

Python also allows you to index from the *end* of the list using a negative number, where [-1] returns the last element. This is super-useful since it means you don't have to programmatically find out the length of the iterable in order to work with elements at the end of it. The indices and reverse indices of my_list are as follows:

```
 0   1   2   3   4   5   6   7   8

['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']

 -9  -8  -7  -6  -5  -4  -3  -2  -1
```

Slicing

A *slice* is a subset of list elements. In the case of lists, a single slice will always be of contiguous elements. Slice notation takes the form
my_list[*start*:*stop*]

where *start* is the index of the first element to include, and *stop* is the index of the item to stop at *without including it in the slice*. So my_list[1:5] returns ['b', 'c', 'd', 'e']:

```
 0   1    2    3    4    5   6   7   8
 ×   ↓    ↓    ↓    ↓    ×   ×   ×   ×
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']
```

Leaving either slice boundary blank means start from (or go to) the end of the list. For example:
my_list[5:]['f', 'g', 'h', 'i']
my_list[:4]['a', 'b', 'c', 'd']

Using a negative indexer will set the start/stop bounds relative to their position from the *end* of the list, so my_list[-5:-2] returns ['e', 'f', 'g']:

```
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']
 ×   ×   ×   ×   ↑    ↑    ↑    ×   ×
 -9  -8  -7  -6  -5   -4   -3   -2  -1
```

Note that if you try my_list[-2:-5], you'll get an empty list. This was something I got tripped up on, but here's what's happening: in order to be included in the slice, an element must be at or to the right of the *start* boundary AND to the left of the *stop* boundary. Because the -2 is already to the *right* of -5, the slicer stops before populating any value into the slice.

A for loop works exactly the same way; the first loop below has no output, but the second does:

```
for i in range(-2,-5):
   print(i)for i in range(-5,-2):
   print(i)-5
-4
-3
```

Iterating through Sequence (Lists & Tuples) –

```
list = [1, 3, 5, 7, 9]
# Using for loop
for i in list:
    print(i)
```

Iterating though Tuples –

There are different ways to iterate through a tuple object. The for statement in Python has a variant which traverses a tuple till it is exhausted.

```
T = (10,20,30,40,50)
for var in T:
print (T.index(var),var)
```

Another Example:

```
for var in range(len(T)):
  print (var,T[var])
```

Functions for all Sequences –

Built-in List Functions & Methods –

Python lists make use of the below mentioned functions –

a. cmp(list1, list2): It is used for comparing elements of both lists.
b. len(list): It is used for providing the total length of the list.
c. max(list): This function is used for returning the item from the list having maximum value.
d. min(list): This function is used for returning the item from the list having minimum value.
e. list(seq): This function is used for converting a tuple into list.

List methods

a. list.append(obj): This method is used for appending object an object to a list
b. list.count(obj): This method is used for returning the count of how many times obj occurs in list
c. list.extend(seq): This method is used for appending the contents of seq to list
d. list.index(obj): This method is used for returns the lowest index in the list in which object appears
e. list.insert(index, obj): This method is used for inserting object (obj) into list at offset index
f. list.pop(obj=list[-1]): This method is used for removing as well as returning the final object i.e. obj from your list
g. list.remove(obj): This method is used for removing object (obj) from your list
h. list.reverse(): This method is used for reversing objects of list
i. list.sort([func]): This method is used for sorting objects of list, by comparing 'func', when given

Built-in Tuple Functions –

Here is a list that includes the different tuple functions –

a. cmp(tuple1, tuple2): This function is used for comparing the different elements of both tuples.
b. len(tuple): This function is used for giving the total length of the tuple.
c. max(tuple): This function is used for returning the item from the tuple having maximum value.
d. min(tuple): This function is used for returning the item from the tuple having minimum value.
e. tuple(seq): This function helps in converting a list to tuple.

Enumerate() in Python –

A lot of times when dealing with iterators, we also get a need to keep a count of iterations. Python eases the programmers' task by providing a built-in function enumerate() for this task. Enumerate() method adds a counter to an iterable and returns it in a form of enumerate object. This enumerate object can then be used directly in for loops or be converted into a list of tuples using list() method.

Syntax:

```
enumerate(iterable, start=0)
```

```
Ex:
l1 = ["eat","sleep","repeat"]
# printing the tuples in object directly
for ele in enumerate(l1):
   print ele
print
# changing index and printing separately
for count,ele in enumerate(l1,100):
   print count,ele
```

xrange Function:

This function returns the generator object that can be used to display numbers only by looping. Only particular range is displayed on demand and hence called "lazy evaluation". The xrange object allows iteration, indexing, and the len() method. You can have a for loop to traverse it and gets the numbers in every iteration.

```
>>> xr = xrange(1, 100, 1)
>>> type(xr)
<type 'xrange'>
>>> xr[0]
1
>>> for it in xr:
...    print(it)
...
1
2
3
...
99
```

List comprehensions –

List comprehensions provide a concise way to create lists. It consists of brackets containing an expression followed by a for clause, then zero or more for or if clauses. The expressions can be anything, meaning you can put in all kinds of objects in lists.

The result will be a new list resulting from evaluating the expression in the context of the for and if clauses which follow it. The list comprehension always returns a result list.

```
new_list = []
for i in old_list:
  if filter(i):
    new_list.append(expressions(i))
```

You can obtain the same thing using list comprehension:

```
new_list = [expression(i) for i in old_list if filter(i)]
```

Generator Expressions in Python:

In Python, to create iterators, we can use both regular functions and generators. Generators are written just like a normal function but we use yield() instead of return() for returning a result. It is more powerful as a tool to implement iterators. It is easy and more convenient to implement because it offers the evaluation of elements on demand. Unlike regular functions which on encountering a return statement terminates entirely, generators use yield statement in which the state of the function is saved from the last call and can be picked up or resumed the next time we call a generator function. Another great advantage of the generator over a list is that it takes much less memory.

In addition to that, two more functions _next_() and _iter_() make the generator function more compact and reliable. Example :

```
filter_none
edit
play_arrow

brightness_4
# Python code to illustrate generator, yield() and next().
def generator():
  t = 1
  print 'First result is ',t
  yield t
```

```
        t += 1
        print 'Second result is ',t
        yield t

        t += 1
        print 'Third result is ',t
        yield t

    call = generator()
    next(call)
    next(call)
    next(call)
```

Output :

```
    First result is  1
    Second result is  2
    Third result is  3
```

Dictionaries and Sets –

Another Python data structure where each key is divided from its associated data/value using the colon (:). Also, the items within are comma separated values, and the entire thing is bounded within curly braces. You can also create an empty dictionary which can be written with just 2 curly braces - {}.

It is to be noted that the keys need to be unique within the dictionary but the values can be redundant. The values within your dictionary can be of heterogenous type, but their keys need to be of an immutable data type (either of these: numbers, strings, or tuples).

Accessing Elements of Dictionaries –

For accessing the elements in the dictionary, make use of the square brackets along with a key for obtaining its value. Here is an example of how to do –

```
d = {'Name': 'Karlos', 'Salary': 850000, 'Job': 'Researcher'}
print "d['Name'] is: ", d['Name']
print "d['Age'] is: ", d['Salary']
```

Updating Dictionary –

Python dictionary also provides you with an option to update your dictionary by including new entry/entries or a key-value pair, as mentioned in the example below –

```
d = {'Name': 'Karlos', 'Height': 6, 'Job': 'Researcher'}
d['Height'] = 5;          #updating the already assigned value with a new entry
d['University'] = "LPU";          # New dictionary element added
print "d['Height']: ", d['Height']
print "d['University']: ", d['University']
```

Delete Dictionary Elements –

There is also a possibility to remove or eliminate individual dictionary elements i.e. clear the complete contents of your dictionary. For explicitly removing the complete dictionary, you have to implement the del statement. Here is an example showing the same –

```
d = {'Name': 'Karlos', 'Height': 6, 'Job': 'Researcher'}
del d['Name']; # this will remove entry having the key 'Name'
dict.clear();      #  this will remove each and every entry of your dictionary
del dict ;          # this will delete the complete dictionary
print "dict['Age']: ", dict['Age']
print "dict['School']: ", dict['School']
```

Sets in Python –

A Set is an unordered collection data type that is iterable, mutable and has no duplicate elements. Python's set class represents the mathematical notion of a set. The major advantage of using a set, as opposed to a list, is that it has a highly optimized method for checking whether a specific element is contained in the set. This is based on a data structure known as a hash table.

```
# demonstrate sets

# Same as {"a", "b", "c"}
Set = set(["a", "b", "c"])

print("Set: ")
print(Set)
```

```
# Adding element to the set
Set.add("d")

print("\nSet after adding: ")
print(Set)
```
Output:

```
Set:
set(['a', 'c', 'b'])

Set after adding:
set(['a', 'c', 'b', 'd'])
```

Functions in Python: -

A function is a set of statements that take inputs, do some specific computation and produces output. The idea is to put some commonly or repeatedly done task together and make a function, so that instead of writing the same code again and again for different inputs, we can call the function.

Python provides built-in functions like print(), etc. but we can also create your own functions. These functions are called user-defined functions.

```
# A simple Python function to check
# whether x is even or odd
def evenOdd( x ):
   if (x % 2 == 0):
      print "even"
   else:
      print "odd"
 evenOdd(2)
evenOdd(3)
```
Output:

```
even
odd
```

Formal Parameters –

Formal arguments are identifiers used in the function definition to represent corresponding actual arguments. The parameter defined as part of the function definition, e.g., x in:

```
def cube(x):
   return x*x*x
```

Global and Local Variables in Python –

Global variables are the one that are defined and declared outside a function and we need to use them inside a function.

```
# This function uses global variable s
def f():
   print s

# Global scope
s = "I love ithubschool"
f()
```

Local Variables

A variable declared inside the function's body or in the local scope is known as local variable.

```
def foo():
   y = "local"
foo()
print(y)
```

Parameter passing and returning values in python: -

Call by Value:

The most common strategy is the call-by-value evaluation, sometimes also called pass-by-value. For example. In call-by-value, the argument expression is evaluated, and the result of this evaluation is bound to the corresponding variable in the function. So, if the expression is a variable, a local copy of its value will be used, i.e. the variable in the caller's scope will be unchanged when the function returns.

Call by Reference:

In call-by-reference evaluation, which is also known as pass-by-reference, a function gets an implicit reference to the argument, rather than a copy of its value. As a consequence, the function can modify the argument, i.e. the value of the variable in the caller's scope can be changed. The advantage of call-by-reference consists in the advantage of greater time- and space-efficiency, because arguments do not need to be copied. On the other hand this harbours the disadvantage that variables can be "accidentally" changed in a function call. So special care has to be taken to "protect" the values, which shouldn't be changed.

```
def mean(a):
   total = 0.0
   for v in a:
      total += v
   return total / len(a)
```

File Handling in Python: -

Python too supports file handling and allows users to handle files i.e., to read and write files, along with many other file handling options, to operate on files. The concept of file handling has stretched over various other languages, but the implementation is either complicated or lengthy, but alike other concepts of Python, this concept here is also easy and short. Python treats file differently as text or binary and this is important. Each line of code includes a sequence of characters and they form text file. Each line of a file is terminated with a special character, called the EOL or End of Line characters like comma {,} or newline character. It ends the current line and tells the interpreter a new one has begun. Let's start with Reading and Writing files.

Working of open() function: We use open () function in Python to open a file in read or write mode. As explained above, open ( ) will return a file object. To return a file object we use open() function along with two arguments, that accepts file name and the mode, whether to read or write. So, the syntax being: open(filename, mode). There are three kinds of mode, that Python provides and how files can be opened:

“ r “, for reading.
“ w “, for writing.
“ a “, for appending.
“ r+ “, for both reading and writing

One must keep in mind that the mode argument is not mandatory. If not passed, then Python will assume it to be “ r ” by default. Let's look at this program and try to analyze how the read mode works:

```
# a file named "geek", will be opened with the reading mode.
file = open('geek.txt', 'r')
# This will print every line one by one in the file
for each in file:
   print (each)
```

The open command will open the file in the read mode and the for loop will print each line present in the file.

Working of read() mode: There is more than one way to read a file in Python. If you need to extract a string that contains all characters in the file then we can use file.read(). The full code would work like this:

```
# Python code to illustrate read() mode
file = open("file.text", "r")
print file.read()
```

Another way to read a file is to call a certain number of characters like in the following code the interpreter will read the first five characters of stored data and return it as a string:

```
# Python code to illustrate read() mode character wise
file = open("file.txt", "r")
print file.read(5)
```

Creating a file using write() mode: Let's see how to create a file and how write mode works:

To manipulate the file, write the following in your Python environment:

```
# Python code to create a file
file = open('geek.txt','w')
file.write("This is the write command")
file.write("It allows us to write in a particular file")
file.close()
```

The close() command terminates all the resources in use and frees the system of this particular program.

Working of append() mode: Let's see how the append mode works:

```
# Python code to illustrate append() mode
file = open('geek.txt','a')
```

```
file.write("This will add this line")
file.close()
```

Binary Data: "Binary" files are any files where the format isn't made up of readable characters. Binary files can range from image files like JPEGs or GIFs, audio files like MP3s or binary document formats like Word or PDF. In Python, files are opened in text mode by default. To open files in binary mode, when specifying a mode, add 'b' to it.

For example:

```
f = open('my_file', 'w+b')
byte_arr = [120, 3, 255, 0, 100]
binary_format = bytearray(byte_arr)
f.write(binary_format)
f.close()
```

Python pickle module is used for serializing and de-serializing a Python object structure. Any object in Python can be pickled so that it can be saved on disk. What pickle does is that it "serializes" the object first before writing it to file. Pickling is a way to convert a python object (list, dict, etc.) into a character stream. The idea is that this character stream contains all the information necessary to reconstruct the object in another python script.

```
import pickle

def storeData():
  # initializing data to be stored in db
  Omkar = {'key' : 'Omkar', 'name' : 'Omkar Pathak',
  'age' : 21, 'pay' : 40000}
  Jagdish = {'key' : 'Jagdish', 'name' : 'Jagdish Pathak',
  'age' : 50, 'pay' : 50000}

  # database
  db = {}
  db['Omkar'] = Omkar
  db['Jagdish'] = Jagdish

  # Its important to use binary mode
  dbfile = open('examplePickle', 'ab')
```

```
    # source, destination
    pickle.dump(db, dbfile)
    dbfile.close()

def loadData():
    # for reading also binary mode is important
    dbfile = open('examplePickle', 'rb')
    db = pickle.load(dbfile)
    for keys in db:
        print(keys, '=>', db[keys])
    dbfile.close()

if __name__ == '__main__':
    storeData()
    loadData()
```

Errors & Exception Handling

A Python program terminates as soon as it encounters an error. In Python, an error can be a syntax error or an exception. In this article, you will see what an exception is and how it differs from a syntax error. After that, you will learn about raising exceptions and making assertions. Then, you'll finish with a demonstration of the try and except block.

Exceptions versus Syntax Errors

Syntax errors occur when the parser detects an incorrect statement. Observe the following example:

```
>>> print( 0 / 0 ))
  File "<stdin>", line 1
    print( 0 / 0 ))
                  ^
SyntaxError: invalid syntax:
```

The arrow indicates where the parser ran into the syntax error. In this example, there was one bracket too many. Remove it and run your code again:

```
>>> print( 0 / 0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

This time, you ran into an exception error. This type of error occurs whenever syntactically correct Python code results in an error. The last line of the message indicated what type of exception error you ran into.

Instead of showing the message exception error, Python details what type of exception error was encountered. In this case, it was a ZeroDivisionError. Python comes with various built-in exceptions as well as the possibility to create self-defined exceptions.

If you want to throw an error when a certain condition occurs using raise, you could go about it like this:

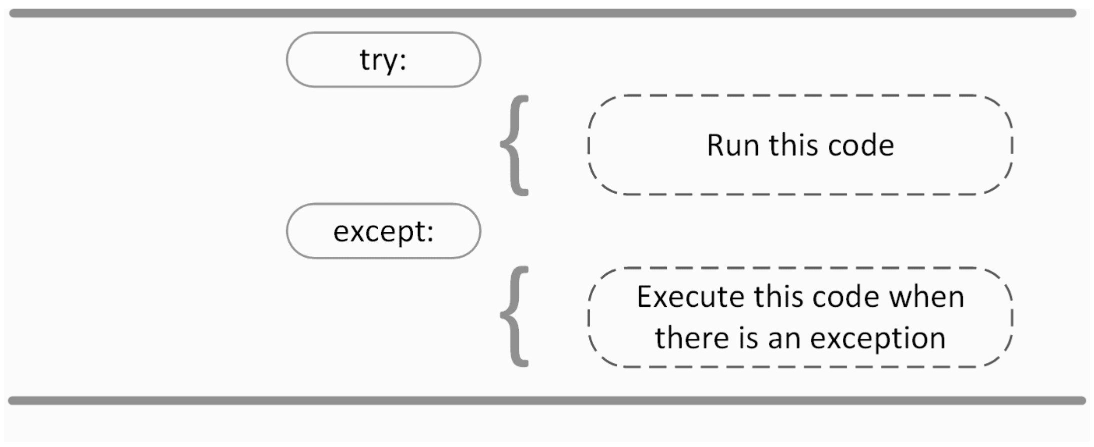
```
x = 10
if x > 5:
   raise Exception('x should not exceed 5. The value of x was: {}'.format(x))
```

When you run this code, the output will be the following:

```
Traceback (most recent call last):
  File "<input>", line 4, in <module>
Exception: x should not exceed 5. The value of x was: 10
```

Handling Exceptions –

The try and except block in Python is used to catch and handle exceptions. Python executes code following the try statement as a “normal” part of the program. The code that follows the except statement is the program’s response to any exceptions in the preceding try clause.



As you saw earlier, when syntactically correct code runs into an error, Python will throw an exception error. This exception error will crash the program if it is unhandled. The except clause determines how your program responds to exceptions.

The following function can help you understand the try and except block:

```
def linux_interaction():
    assert ('linux' in sys.platform), "Function can only run on Linux systems."
    print('Doing something.')
```

The linux_interaction() can only run on a Linux system. The assert in this function will throw an AssertionError exception if you call it on an operating system other then Linux.

You can give the function a try using the following code:

```
try:
    linux_interaction()
except:
    pass
```

When an exception occurs in a program running this function, the program will continue as well as inform you about the fact that the function call was not successful.

What you did not get to see was the type of error that was thrown as a result of the function call. In order to see exactly what went wrong, you would need to catch the error that the function threw.

The following code is an example where you capture the AssertionError and output that message to screen:

```
try:
    linux_interaction()
except AssertionError as error:
    print(error)
    print('The linux_interaction() function was not executed')
```

Running this function on a Windows machine outputs the following:

```
Function can only run on Linux systems.
The linux_interaction() function was not executed
```

Cleaning Up After Using finally: -

Imagine that you always had to implement some sort of action to clean up after executing your code. Python enables you to do so using the finally clause.

Diagram explaining try except else finally statements

Have a look at the following example:

```
try:
    linux_interaction()
except AssertionError as error:
    print(error)
else:
    try:
        with open('file.log') as file:
            read_data = file.read()
    except FileNotFoundError as fnf_error:
        print(fnf_error)
finally:
    print('Cleaning up, irrespective of any exceptions.')
```

In the previous code, everything in the finally clause will be executed. It does not matter if you encounter an exception somewhere in the try or else clauses.

Running the previous code on a Windows machine would output the following:

```
Function can only run on Linux systems.
Cleaning up, irrespective of any exceptions.
```

USING MODULES –

Import module in Python

Import in python is similar to #include header_file in C/C++. Python modules can get access to code from another module by importing the file/function using import. The import statement is the most common way of invoking the import machinery, but it is not the only way.

```
import module_name
```

When import is used, it searches for the module initially in the local scope by calling __import__() function. The value returned by the function are then reflected in the output of the initial code.

```
import math
print(math.pi)
```

The Module Search Path

```
>>> import mod
>>> mod.a
[100, 200, 300]
>>> mod.s
'Computers are useless. They can only give you answers.'
```

When the interpreter executes the above import statement, it searches for mod.py in a list of directories assembled from the following sources:

The directory from which the input script was run, or the current directory if the interpreter is being run interactively. The list of directories contained in the PYTHONPATH environment variable, if it is set. (The format for PYTHONPATH is OS-dependent but should mimic the PATH environment variable.)

An installation-dependent list of directories configured at the time Python is installed

The resulting search path is accessible in the Python variable sys.path, which is obtained from a module named sys:

```
>>> import sys
>>> sys.path
['', '/Library/Frameworks/Python.framework/Versions/3.7/bin',
'/Library/Frameworks/Python.framework/Versions/3.7/lib/python37.zip',
'/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7',
'/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/lib-dynload',
'/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/site-packages']
```

So, to ensure that your module is found, you need to do one of the following:

Put mod.py in the directory where the input script is located, or the current directory if interactive

Modify the PYTHONPATH environment variable to contain the directory where mod.py is located before starting the interpreter. Or put mod.py in one of the directories already contained in the PYTHONPATH variable.

Put mod.py in one of the installation-dependent directories, which you may or may not have write-access to, depending on the OS.

There is also one additional option: You can put the module file in any directory of your choice and then modify sys.path at run-time so that it contains that directory. For example, in this case, you could put mod.py in directory /Users/chris/ModulesAndPackages and then issue the following statements:

```
>>> sys.path
['', '/Library/Frameworks/Python.framework/Versions/3.7/bin',
'/Library/Frameworks/Python.framework/Versions/3.7/lib/python37.zip',
'/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7',
'/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/lib-dynload',
'/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/site-packages',
'/Users/chris/ModulesAndPackages']
>>> import mod
>>> mod.s
'Computers are useless. They can only give you answers.'
```

Install Python packages for personal use

Set up your user environment

To install Python packages, you must have Python added to your user environment. On the command line, enter:

```
1.   module list
```

2. If Python is not among the list of currently loaded modules, use the module load command to add it; for example:
   - To add the default version, on the command line, enter:

```
o    module load python
```

- To add a non-default version:

1. Check which versions are available; on the command line, enter:

```
2.    module avail python
```

3. Load the preferred version; on the command line, enter (replace version_number with the preferred version number):

```
4.    module load python/version_number
```

3. If Python is listed among the currently loaded modules, but you prefer or need to use another version, you must remove the currently loaded module before loading the other version. To do this with one command, use module switch; for example, on the command line, enter (replace current_version with the version number of the currently loaded python module and new_version with the preferred version number):

```
module switch python/current_version python/new_version
```

Install a package using pip

The pip package management tool, one of the standard tools maintained by the Python Package Authority (PyPA), is the recommended tool for installing packages from the Python Package Index (PyPI) repository. To install a package from the PyPI repository (for example, foo), use the pip install command with the --user flag; for example:

To install:	Use the command:
The latest version	pip install foo –user
A particular version (for example, foo 1.0.3)	pip install foo==1.0.3 --user

To install:	Use the command:
A minimum version (for example, foo 2.0)	pip install 'foo>=2.0' --user

The --user option directs pip to download and unpack the source distribution for your package (for example, foo) in the user site-packages directory for the running Python; for example:

```
~/.local/lib/python2.7/site-packages/foo
```

Python automatically searches this directory for modules, so prepending this path to the PYTHONPATH environmental variable is not necessary.

Install a package using its setup.py script

To install a Python package from a source other than the PyPI repository, you can download and unpack the source distribution yourself, and then use its setup.py script to install the package in the user site-packages directory:

1. Set up your user environment (as described in the previous section).
2. Use the wget command to download the distribution archive (for example, foo-1.0.3.gz) from the source (for example, http://pythonfoo.org); for example:

```
3.    wget http://pythonfoo.org/foo-1.0.3.gz
```

4. Use tar to unpack the archive (for example, foo-1.0.3.gz); for example:

```
5.    tar -xzf foo-1.0.3.gz
```

   The distribution should unpack into a similarly-named directory in your home directory (for example, ~/foo-1.0.3).

6. Change (cd) to the new directory, and then, on the command line, enter:

```
7.    python setup.py install --user
```

The --user option directs setup.py to install the package (for example, foo) in the user site-packages directory for the running Python; for example:

```
~/.local/lib/pythonX.Y/site-packages/foo
```

Regular Expression: -

A Regular Expression (RegEx) is a sequence of characters that defines a search pattern. For example,

^a...s$

The above code defines a RegEx pattern. The pattern is: any five-letter string starting with a and ending with s.

A pattern defined using RegEx can be used to match against a string.

Expression	String	Matched?
^a...s$	abs	No match
	alias	Match
	abyss	Match
	Alias	No match
	An abacus	No match

Python has a module named re to work with RegEx. Here's an example:

```
import re
pattern = '^a...s$'
test_string = 'abyss'
result = re.match(pattern, test_string)
if result:
  print("Search successful.")
else:
  print("Search unsuccessful.")
```

Here, we used re.match() function to search pattern within the test_string. The method returns a match object if the search is successful. If not, it returns None. There are other several functions defined in the re module to work with RegEx. Before we explore that, let's learn about regular expressions themselves.

If you already know the basics of RegEx, jump to Python RegEx.

Specify Pattern Using RegEx

To specify regular expressions, metacharacters are used. In the above example, ^ and $ are metacharacters.

MetaCharacters: -

Metacharacters are characters that are interpreted in a special way by a RegEx engine. Here's a list of metacharacters:

[] . ^ $ * + ? {} () \ |

Square brackets

Square brackets specifies a set of characters you wish to match.

Period

A period matches any single character (except newline '\n').

Caret

The caret symbol `^` is used to check if a string **starts with** a certain character.

Dollar

The dollar symbol `$` is used to check if a string **ends with** a certain character.

Star

The star symbol `*` matches **zero or more occurrences** of the pattern left to it.

Plus

The plus symbol `+` matches **one or more occurrences** of the pattern left to it.

Question Mark

The question mark symbol `?` matches **zero or one occurrence** of the pattern left to it.

Braces

Consider this code: `{n,m}`. This means at least `n`, and at most `m` repetitions of the pattern left to it.

Alternation

Vertical bar `|` is used for alternation (`or` operator). Here, `a|b` match any string that contains either `a` or `b`

Group

Parentheses () is used to group sub-patterns. For example, (a|b|c)xz match any string that matches either a or b or c followed by xz

Pattern Matching –

Steps of Regular Expression Matching: While there are several steps to using regular expressions in Python, each step is fairly simple.

1. Import the regex module with import re.
2. Create a Regex object with the re.compile() function. (Remember to use a raw string.)
3. Pass the string you want to search into the Regex object's search() method. This returns a Match object.
4. Call the Match object's group() method to return a string of the actual matched text.

Matching and Retrieving all data at once –

```
import re
phoneNumRegex = re.compile(r'(\d\d\d)-(\d\d\d-\d\d\d\d)')
mo = phoneNumRegex.search('My number is 415-555-4242.')
print(mo.group(1))
```

Another Example:

```
import re
phoneNumRegex = re.compile(r'(\d\d\d)-(\d\d\d-\d\d\d\d)')
mo = phoneNumRegex.search('My number is 415-555-4242.')
print(mo.groups())
```

Parsing: -

When working with data, it is sometimes necessary to write a regular expression to look for properly entered values. Phone numbers in a dataset is a common field that needs to be checked for validity. Your job in this exercise is to define a regular expression to match US phone numbers that fit the pattern of xxx-xxx-xxxx.

The regular expression module in python is re. When performing pattern matching on data, since the pattern will be used for a match across multiple rows, it's better to compile the pattern first using re.compile(), and then use the compiled pattern to match values.

Substitution & Complex Substitution –

The following rules for $-placeholders apply:

- $$ is an escape; it is replaced with a single $
- $identifier names a substitution placeholder matching a mapping key of "identifier". By default, "identifier" must spell a Python identifier as defined in [2]. The first non-identifier character after the $ character terminates this placeholder specification.
- ${identifier} is equivalent to $identifier. It is required when valid identifier characters follow the placeholder but are not part of the placeholder, e.g. "${noun}ification".

If the $ character appears at the end of the line, or is followed by any other character than those described above, a ValueError will be raised at interpolation time. Values in mapping are converted automatically to strings.

No other characters have special meaning, however it is possible to derive from the Template class to define different substitution rules. For example, a derived class could allow for periods in the placeholder (e.g. to support a kind of dynamic namespace and attribute path lookup), or could define a delimiter character other than $.

Once the Template has been created, substitutions can be performed by calling one of two methods:

- ✓ substitute(): This method returns a new string which results when the values of a mapping are substituted for the placeholders in the Template. If there are placeholders which are not present in the mapping, a KeyError will be raised.

- ✓ safe_substitute(): This is similar to the substitute() method, except that KeyErrors are never raised (due to placeholders missing from the mapping). When a placeholder is missing, the original placeholder will appear in the resulting string.

Here are some examples:

```
>>> from string import Template
>>> s = Template('${name} was born in ${country}')
>>> print s.substitute(name='Guido', country='the Netherlands')
Guido was born in the Netherlands
>>> print s.substitute(name='Guido')
Traceback (most recent call last):
[...]
KeyError: 'country'
>>> print s.safe_substitute(name='Guido')
Guido was born in ${country}
```

The signature of substitute() and safe_substitute() allows for passing the mapping of placeholders to values, either as a single dictionary-like object in the first positional argument, or as keyword arguments as shown above. The exact details and signatures of these two methods is reserved for the standard library documentation.

Special Syntax with Parentheses –

- R(?#comment): Matches "R". All the rest is a comment
- R(?i)uby: Case-insensitive while matching "uby"
- R(?i:uby): Same as above
- rub(?:y|le)): Group only without creating \1 backreference

Object Oriented Programming in Python

Python is an "object-oriented programming language." This means that almost all the code is implemented using a special construct called classes. Programmers use classes to keep related things together. This is done using the keyword "class," which is a grouping of object-oriented constructs.

What is a class?

A class is a code template for creating objects. Objects have member variables and have behavior associated with them. In python a class is created by the keyword class.

An object is created using the constructor of the class. This object will then be called the instance of the class. In Python we create instances in the following manner

```
Instance = class(arguments)
```

How to create a class

The simplest class can be created using the class keyword. For example, let's create a simple, empty class with no functionalities.

```
>>> class Snake:
...     pass
...
>>> snake = Snake()
>>> print(snake)
<__main__.Snake object at 0x7f315c573550>
```

Attributes and Methods in class:

A class by itself is of no use unless there is some functionality associated with it. Functionalities are defined by setting attributes, which act as containers for data and functions related to those attributes. Those functions are called methods.

Attributes:

You can define the following class with the name Snake. This class will have an attribute name.

```
>>> class Snake:
...     name = "python" # set an attribute `name` of the class
...
```

You can assign the class to a variable. This is called object instantiation. You will then be able to access the attributes that are present inside the class using the dot . operator. For example, in the Snake example, you can access the attribute name of the class Snake.

```
>>> # instantiate the class Snake and assign it to variable snake
>>> snake = Snake()

>>> # access the class attribute name inside the class Snake.
>>> print(snake.name)
python
```

Instance Methods in Python

Once there are attributes that “belong” to the class, you can define functions that will access the class attribute. These functions are called methods. When you define methods, you will need to always provide the first argument to the method with a self keyword.

For example, you can define a class Snake, which has one attribute name and one method change_name. The method change name will take in an argument new_name along with the keyword self.

```
>>> class Snake:
...     name = "python"
...
...     def change_name(self, new_name): # note that the first argument is self
...         self.name = new_name # access the class attribute with the self keyword
...
```

Now, you can instantiate this class Snake with a variable snake and then change the name with the method change_name.

```
>>> # instantiate the class
>>> snake = Snake()

>>> # print the current object name
>>> print(snake.name)
python

>>> # change the name using the change_name method
>>> snake.change_name("anaconda")
>>> print(snake.name)
anaconda
```

Instance attributes in python and the init method

You can also provide the values for the attributes at runtime. This is done by defining the attributes inside the init method. The following example illustrates this.

```
class Snake:

    def __init__(self, name):
        self.name = name

    def change_name(self, new_name):
        self.name = new_name
```

Now you can directly define separate attribute values for separate objects. For example,

```
>>> # two variables are instantiated
>>> python = Snake("python")
>>> anaconda = Snake("anaconda")

>>> # print the names of the two variables
>>> print(python.name)
python
>>> print(anaconda.name)
anaconda
```

Practice Example:

```
class Dog:

    # A simple class
    # attribute
    attr1 = "mamal"
    attr2 = "dog"

    # A sample method
    def fun(self):
        print("I'm a", self.attr1)
        print("I'm a", self.attr2)

# Driver code
# Object instantiation
Rodger = Dog()

# Accessing class attributes
# and method through objects
print(Rodger.attr)
Rodger.fun()
```

Properties:

Python has a great concept called property which makes the life of an object oriented programmer much simpler.

An Example To Begin With

Let us assume that you decide to make a class that could store the temperature in degree Celsius. It would also implement a method to convert the temperature into degree Fahrenheit. One way of doing this is as follows.

```
class Celsius:

  def __init__(self, temperature = 0):

    self.temperature = temperature



  def to_fahrenheit(self):

    return (self.temperature * 1.8) + 32
```

We could make objects out of this class and manipulate the attribute temperature as we wished. Try these on Python shell.

```
>>> # create new object
>>> man = Celsius()
>>> # set temperature
>>> man.temperature = 37
>>> # get temperature
>>> man.temperature
37
>>> # get degrees Fahrenheit
>>> man.to_fahrenheit()
98.60000000000001
```

Using Getters and Setters Properties –

Getters (also known as 'accessors') and setters (aka. 'mutators') are used in many object oriented programming languages to ensure the principle of data encapsulation. An obvious solution to the above constraint will be to hide the attribute temperature (make it private) and define new getter and setter interfaces to manipulate it. This can be done as follows.

```
class Celsius:
  def __init__(self, temperature = 0):
    self.set_temperature(temperature)

  def to_fahrenheit(self):
    return (self.get_temperature() * 1.8) + 32

  # new update
```

```
def get_temperature(self):
    return self._temperature


def set_temperature(self, value):
    if value < -273:
        raise ValueError("Temperature below -273 is not possible")
    self._temperature = value
```

Class Methods and Data: -

A data class comes with basic functionality already implemented. For instance, you can instantiate, print, and compare data class instances straight out of the box:

```
>>> queen_of_hearts = DataClassCard('Q', 'Hearts')
>>> queen_of_hearts.rank
'Q'
>>> queen_of_hearts
DataClassCard(rank='Q', suit='Hearts')
>>> queen_of_hearts == DataClassCard('Q', 'Hearts')
True
```

Compare that to a regular class. A minimal regular class would look something like this:

```
class RegularCard
   def __init__(self, rank, suit):
      self.rank = rank
      self.suit = suit
```

Whereas, Instance variables are for data unique to each instance and class variables are for attributes and methods shared by all instances of the class. Instance variables are variables whose value is assigned inside a constructor or method with self whereas class variables are variables whose value is assigned in the class.

```
class Dog:

   # Class Variable

   animal = 'dog'

   # The init method or constructor

   def __init__(self, breed, color):



      # Instance Variable
```

```
        self.breed = breed

        self.color = color

# Objects of CSStudent class

Rodger = Dog("Pug", "brown")

Buzo = Dog("Bulldog", "black")

print('Rodger details:')

print('Rodger is a', Rodger.animal)

print('Breed: ', Rodger.breed)

print('Color: ', Rodger.color)

print('\nBuzo details:')

print('Buzo is a', Rodger.animal)

print('Breed: ', Buzo.breed)

print('Color: ', Buzo.color)

# Class variables can be accessed using class

# name also

print("\nAccessing class variable using class name")

print(Dog.animal)
```

Static Methods in Python –

What is a static method?

Static methods in Python are extremely similar to python class level methods, the difference being that a static method is bound to a class rather than the objects for that class.

This means that a static method can be called without an object for that class. This also means that static methods cannot modify the state of an object as they are not bound to it. Let's see how we can create static methods in Python.

Creating python static methods

Python Static methods can be created in two ways. Let's see each of the ways here:

```
class Calculator:

    def multiplyNums(x, y):
        return x + y
# create addNumbers static method
Calculator.multiplyNums = staticmethod(Calculator.multiplyNums)
print('Product:', Calculator.multiplyNums(15, 110))
```

Advantages of Python static method

Static methods have a very clear use-case. When we need some functionality not w.r.t an Object but w.r.t the complete class, we make a method static. This is pretty much advantageous when we need to create Utility methods as they aren't tied to an object lifecycle usually.

Finally, note that in a static method, we don't need the self to be passed as the first argument.

Private Methods of a class –

Private methods are those methods that should neither be accessed outside the class nor by any base class. In Python, there is no existence of Private methods that cannot be accessed except inside a class. However, to define a private method prefix the member name with double underscore "__".

Note: The __init__ method is a constructor and runs as soon as an object of a class is instantiated.

```
# demonstrate private methods

# Creating a Base class
class Base:

    # Declaring public method
    def fun(self):
        print("Public method")

    # Declaring private method
    def __fun(self):
        print("Private method")

# Creating a derived class
class Derived(Base):
    def __init__(self):

        # Calling constructor of
        # Base class
        Base.__init__(self)

    def call_public(self):

        # Calling public method of base class
        print("\nInside derived class")
        self.fun()

    def call_private(self):

        # Calling private method of base class
        self.__fun()

# Driver code
obj1 = Base()

# Calling public method
obj1.fun()

obj2 = Derived()
obj2.call_public()
```

Python Inheritance –

What is Inheritance?

Inheritance is a powerful feature in object oriented programming. It refers to defining a new class with little or no modification to an existing class. The new class is called derived (or child) class and the one from which it inherits is called the base (or parent) class.

Python Inheritance Syntax

```
class BaseClass:
  Body of base class
class DerivedClass(BaseClass):
  Body of derived class
```

Derived class inherits features from the base class, adding new features to it. This results into re-usability of code.

```
class Polygon:
    def __init__(self, no_of_sides):
        self.n = no_of_sides
        self.sides = [0 for i in range(no_of_sides)]
    def inputSides(self):
        self.sides = [float(input("Enter side "+str(i+1)+" : ")) for i in range(self.n)]
    def dispSides(self):
        for i in range(self.n):
            print("Side",i+1,"is",self.sides[i])
```

Aliasing Modules

It is possible to modify the names of modules and their functions within Python by using the as keyword. You may want to change a name because you have already used the same name for something else in your program, another module you have imported also uses that name, or you may want to abbreviate a longer name that you are using a lot.

The construction of this statement looks like this:

```
import [module] as [another_name]
```

Let's modify the name of the math module in our my_math.py program file. We'll change the module name of math to m in order to abbreviate it. Our modified program will look like this:

```
my_math.py
import math as m
print(m.pi)
print(m.e)
```

Within the program, we now refer to the pi constant as m.pi rather than math.pi.

For some modules, it is commonplace to use aliases. The matplotlib.pyplot module's official documentation calls for use of plt as an alias:

```
import matplotlib.pyplot as plt
```

This allows programmers to append the shorter word plt to any of the functions available within the module, as in plt.show()