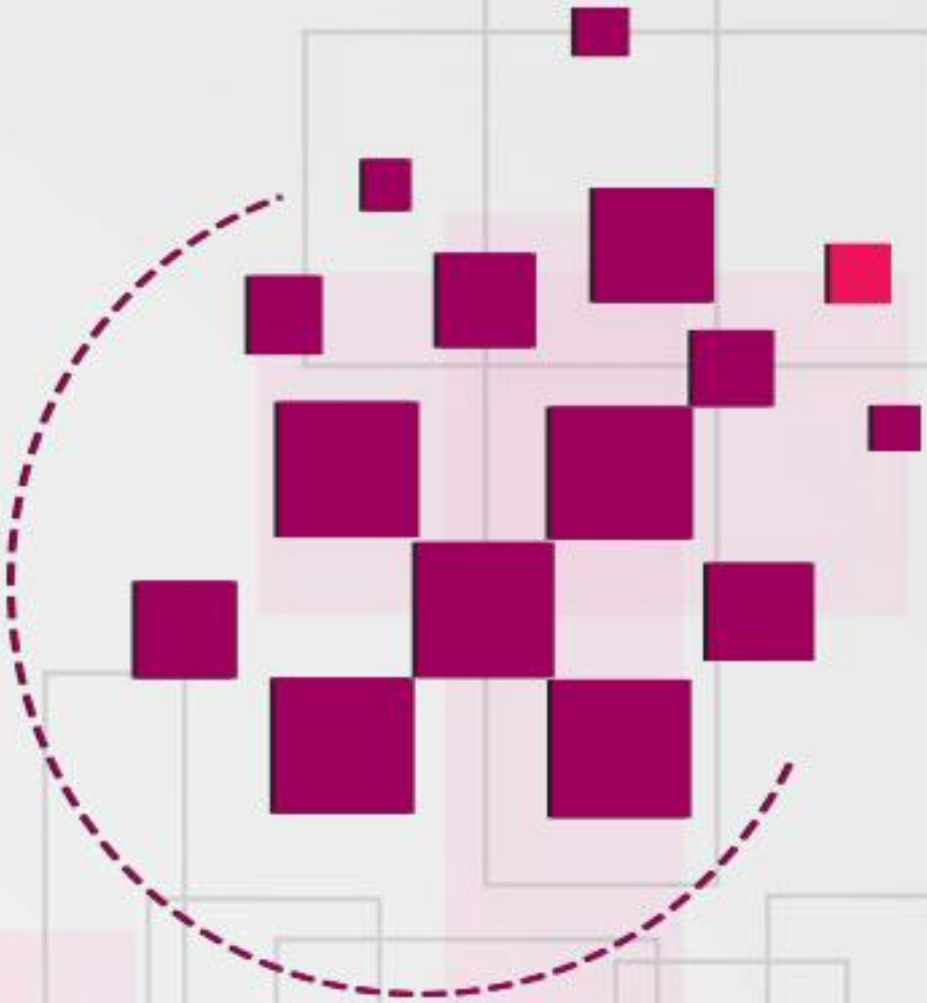


Object Oriented Python

tutorialspoint

SIMPLY EASY LEARNING

www.tutorialspoint.com

https://www.facebook.com/tutorialspointindia https://twitter.com/tutorialspoint

About the Tutorial

Python has been an object-oriented language since it existed. In this tutorial we will try to get in-depth features of OOPS in Python programming.

Audience

This tutorial has been prepared for the beginners and intermediate to help them understand the Python Oops features and concepts through programming.

Prerequisites

Understanding on basic of Python programming language will help to understand and learn quickly. If you are new to programming, it is recommended to first go through "Python for beginners" tutorials.

Table of Contents

1. OOP in Python – Introduction

Programming languages are emerging constantly, and so are different methodologies. Object-oriented programming is one such methodology that has become quite popular over past few years.

This chapter talks about the features of Python programming language that makes it an object-oriented programming language.

Language Programming Classification Scheme

Python can be characterized under object-oriented programming methodologies. The following image shows the characteristics of various programming languages. Observe the features of Python that makes it object-oriented.

Language Classes	Categories	Languages
Programming Paradigm	Procedural	C, C++, C#, Objective-C, Java, Go
	Scripting	CoffeeScript, JavaScript, Python, Perl, Php, Ruby
	Functional	Clojure, Eralang, Haskell, Scala
Compilation Class	Static	C, C++, C#, Objective-C, Java, Go, Haskell, Scala
	Dynamic	CoffeeScript, JavaScript, Python, Perl, Php, Ruby, Clojure, Eralang
Type Class	Strong	C#, Java, Go, Python, Ruby, Clojure, Erlang, Haskell, Scala
	Weak	C, C++, Objective-C, CoffeeScript, JavaScript, Perl, Php
Memory Class	Managed	Others
	Unmanaged	C, C++, Objective-C

What is Object Oriented Programming?

Object Oriented means directed towards objects. In other words, it means functionally directed towards modelling objects. This is one of the many techniques used for modelling complex systems by describing a collection of interacting objects via their data and behavior.

Python, an Object Oriented programming (OOP), is a way of programming that focuses on using objects and classes to design and build applications.. Major pillars of Object Oriented Programming (OOP) are **Inheritance**, **Polymorphism**, **Abstraction**, ad **Encapsulation**.

Object Oriented Analysis(OOA) is the process of examining a problem, system or task and identifying the objects and interactions between them.

Why to Choose Object Oriented Programming?

Python was designed with an object-oriented approach. OOP offers the following advantages:

- Provides a clear program structure, which makes it easy to map real world problems and their solutions.
- Facilitates easy maintenance and modification of existing code.
- Enhances program modularity because each object exists independently and new features can be added easily without disturbing the existing ones.
- Presents a good framework for code libraries where supplied components can be easily adapted and modified by the programmer.
- Imparts code reusability

Procedural vs. Object Oriented Programming

Procedural based programming is derived from structural programming based on the concepts of **functions/procedure/routines**. It is easy to access and change the data in procedural oriented programming. On the other hand, Object Oriented Programming (OOP) allows decomposition of a problem into a number of units called **objects** and then build the data and functions around these objects. It emphasis more on the data than procedure or functions. Also in OOP, data is hidden and cannot be accessed by external procedure.

The table in the following image shows the major differences between POP and OOP approach.

Difference between Procedural Oriented Programming (POP) vs. Object oriented programming (OOP).

	Procedural Oriented Programming	Object Oriented Programming
Based On	In Pop, entire focus is on data and functions	Oops is based on a real world scenarios. Whole program is divided into small parts called object
Reusability	Limited Code reuse	Code reuse
Approach	Top down approach	Object focused Design
Access specifiers	Not any	Public, Private and Protected
Data movement	Data can move freely from functions to function in the system	In oops, objects can move and communicate with each other through member functions
Data Access	In pop, most function uses global data for sharing that can be accessed freely from function to function in the system	In oops, data cannot move freely from method to method, it can be kept in public or private so we can control the access of data.
Data Hiding	In pop, so specific way to hide data, so little bit less secure	It provides data hiding, so much more secure.
Overloading	Not possible	Function and Operator Overloading
Example-Languages	C, VB, Fortran, Pascal	C++, Python, Java, C#
Abstraction	Uses abstraction at procedure level	Uses abstraction at class and object level

Principles of Object Oriented Programming

Object Oriented Programming (OOP) is based on the concept of **objects** rather than actions, and **data** rather than logic. In order for a programming language to be object-oriented, it should have a mechanism to enable working with classes and objects as well as the implementation and usage of the fundamental object-oriented principles and concepts namely inheritance, abstraction, encapsulation and polymorphism.

Four Pillars of Object-Oriented Programming

Let us understand each of the pillars of object-oriented programming in brief:

Encapsulation

This property hides unnecessary details and makes it easier to manage the program structure. Each object's implementation and state are hidden behind well-defined boundaries and that provides a clean and simple interface for working with them. One way to accomplish this is by making the data private.

Inheritance

Inheritance, also called generalization, allows us to capture a hierarchal relationship between classes and objects. For instance, a 'fruit' is a generalization of 'orange'. Inheritance is very useful from a code reuse perspective.

Abstraction

This property allows us to hide the details and expose only the essential features of a concept or object. For example, a person driving a scooter knows that on pressing a horn, sound is emitted, but he has no idea about how the sound is actually generated on pressing the horn.

Polymorphism

Poly-morphism means many forms. That is, a thing or action is present in different forms or ways. One good example of polymorphism is constructor overloading in classes.

Object-Oriented Python

The heart of Python programming is **object** and **OOP**, however you need not restrict yourself to use the OOP by organizing your code into classes. OOP adds to the whole design philosophy of Python and encourages a clean and pragmatic way to programming. OOP also enables in writing bigger and complex programs.

Modules vs. Classes and Objects

Modules are like "Dictionaries"

When working on Modules, note the following points:

- A Python module is a package to encapsulate reusable code.
- Modules reside in a folder with a **__init__.py** file on it.
- Modules contain functions and classes.
- Modules are imported using the **import** keyword.

Recall that a dictionary is a **key-value** pair. That means if you have a dictionary with a key **EmployeID** and you want to retrieve it, then you will have to use the following lines of code:

```
employee = {"EmployeID": "Employee Unique Identity!"}
print (employee ['EmployeID])
```

You will have to work on modules with the following process:

- A module is a Python file with some functions or variables in it.
- Import the file you need.
- Now, you can access the functions or variables in that module with the '**.**' **(dot)** Operator.

Consider a module named **employee.py** with a function in it called **employee**. The code of the function is given below:

```
# this goes in employee.py
def EmployeID():
     print ("Employee Unique Identity!")
```

Now import the module and then access the function **EmployeID**:

```
import employee
employee. EmployeID()
```

You can insert a variable in it named **Age**, as shown:

```
def EmployeID():
     print (“Employee Unique Identity!”)
# just a variable
Age = “Employee age is **”
```

Now, access that variable in the following way:

```
import employee
employee.EmployeID()
print(employee.Age)
```

Now, let’s compare this to dictionary:

```
Employee[‘EmployeID’]    # get EmployeID from employee
Employee.employeID()     # get employeID from the module
Employee.Age             # get access to variable
```

Notice that there is common pattern in Python:

- Take a **key = value** style container
- Get something out of it by the key’s name

When comparing module with a dictionary, both are similar, except with the following:

- In the case of the **dictionary**, the key is a string and the syntax is [key].
- In the case of the **module**, the key is an identifier, and the syntax is .key.

Classes are like Modules

Module is a specialized dictionary that can store Python code so you can get to it with the ‘**.**’ Operator. A class is a way to take a grouping of functions and data and place them inside a container so you can access them with the ‘**.**‘operator.

If you have to create a class similar to the employee module, you can do it using the following code:

```
class employee(object):
     def __init__(self):
          self. Age = “Employee Age is ##”
     def EmployeID(self):
          print (“This is just employee unique identity”)
```

Note: Classes are preferred over modules because you can reuse them as they are and without much interference. While with modules, you have only one with the entire program.

Objects are like Mini-imports

A class is like a **mini-module** and you can import in a similar way as you do for classes, using the concept called **instantiate**. Note that when you instantiate a class, you get an **object**.

You can instantiate an object, similar to calling a class like a function, as shown:

```
this_obj = employee()          # Instantiatethis_obj.EmployeID()            #
get EmployeId from the class
print(this_obj.Age)                  # get variable Age
```

You can do this in any of the following three ways:

```
# dictionary style
Employee[‘EmployeID’]
# module style
Employee.EmployeID()
Print(employee.Age)
# Class style
this_obj = employee()
this_obj.employeID()
Print(this_obj.Age)
```

2. OOP in Python – Environment Setup

This chapter will explain in detail about setting up the Python environment on your local computer.

Prerequisites and Toolkits

Before you proceed with learning further on Python, we suggest you to check whether the following prerequisites are met:

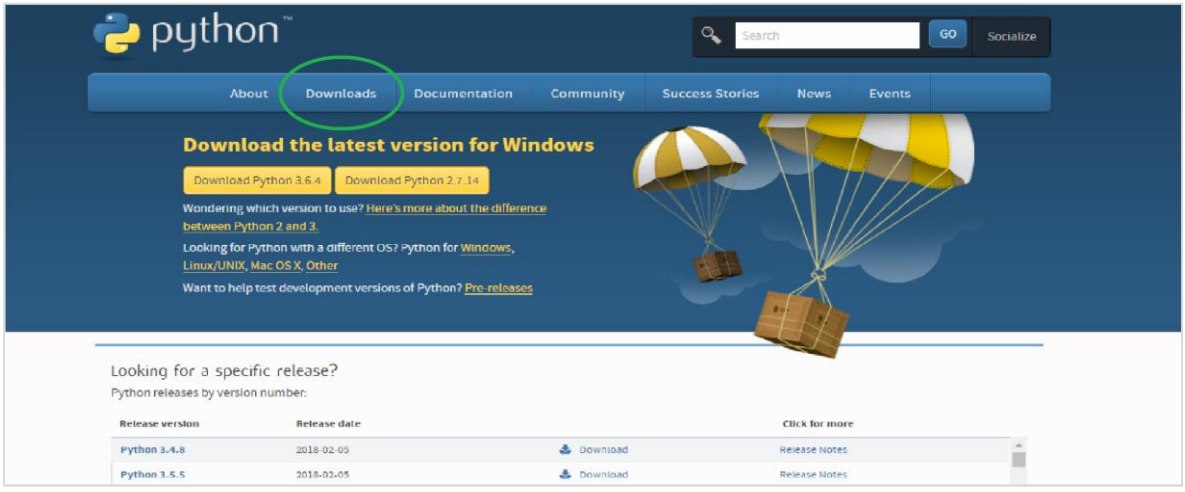
- Latest version of Python is installed on your computer
- An IDE or text editor is installed
- You have basic familiarity to write and debug in Python, that is you can do the following in Python:
    - Able to write and run Python programs.
    - Debug programs and diagnose errors.
    - Work with basic data types.
    - Write **for** loops, **while** loops, and **if** statements
    - Code **functions**

If you don't have any programming language experience, you can find lots of beginner tutorials in Python on TutorialsPoint.

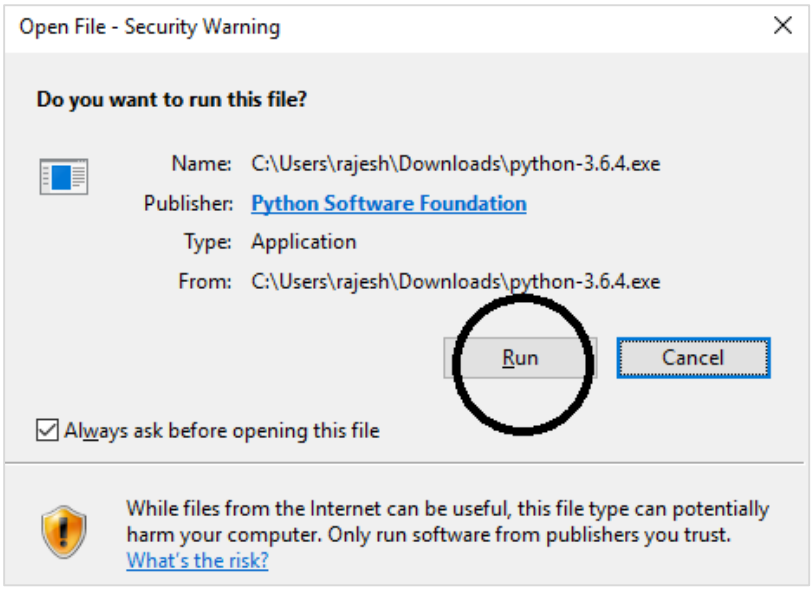
Installing Python

The following steps show you in detail how to install Python on your local computer:

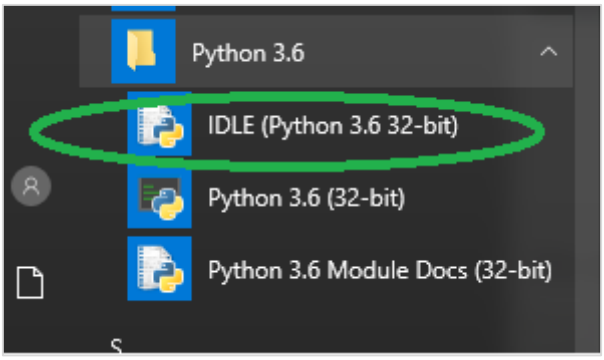
Step 1: Go to the official Python website https://www.Python.org/, click on the **Downloads** menu and choose the latest or any stable version of your choice.



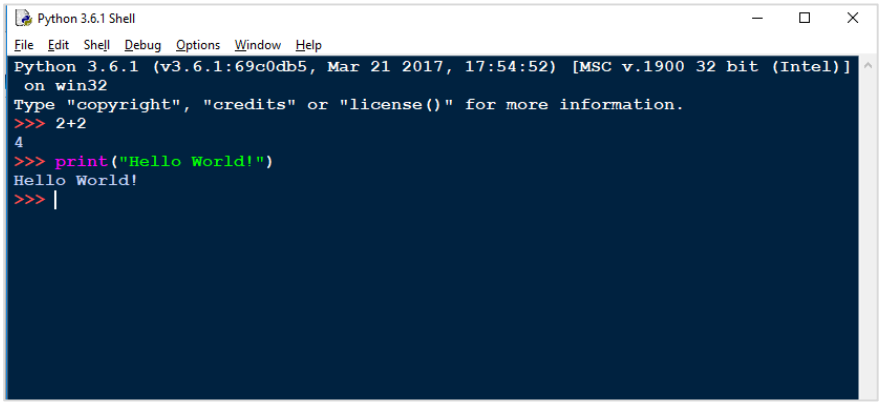
Step 2: Save the Python installer exe file that you're downloading and once you have downloaded it, open it. Click on **Run** and choose **Next** option by default and finish the installation.



Step 3: After you have installed, you should now see the Python menu as shown in the image below. Start the program by choosing IDLE (Python GUI).



This will start the Python shell. Type in simple commands to check the installation.



Choosing an IDE

An Integrated Development Environment is a text editor geared towards software development. You will have to install an IDE to control the flow of your programming and to group projects together when working on Python. Here are some of IDEs avaialable online. You can choose one at your convenience.

- Pycharm IDE
- Komodo IDE
- Eric Python IDE

Note: Eclipse IDE is mostly used in Java, however it has a Python plugin.

Pycharm

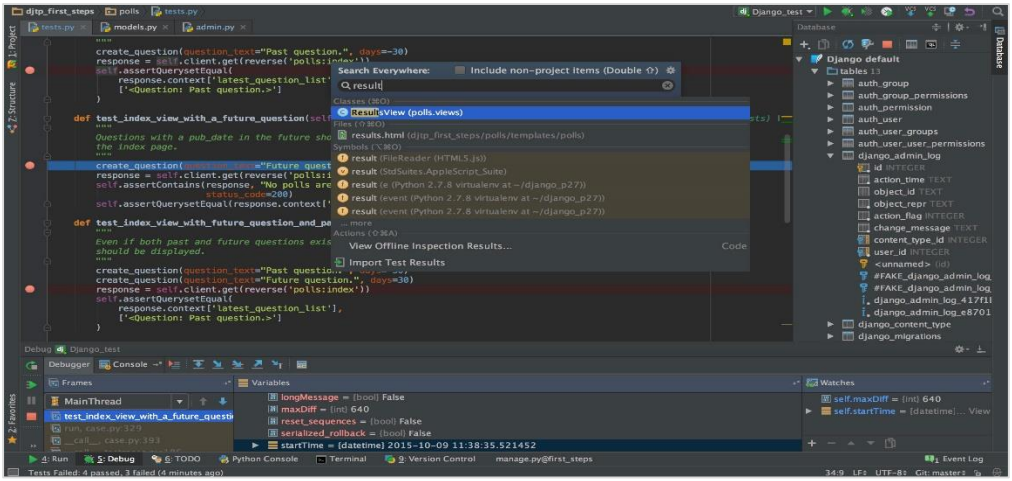
Pycharm, the cross-platform IDE is one of the most popular IDE currently available. It provides coding assistance and analysis with code completion, project and code navigation, integrated unit testing, version control integration, debugging and much more.

Download link

https://www.jetbrains.com/pycharm/download/

Languages Supported: Python, HTML, CSS, JavaScript, Coffee Script, TypeScript, Cython,AngularJS, Node.js, template languages.

Screenshot



Why to Choose?

PyCharm offers the following features and benefits for its users:

- Cross platform IDE compatible with Windows, Linux, and Mac OS
- Includes Django IDE, plus CSS and JavaScript support
- Includes thousands of plugins, integrated terminal and version control
- Integrates with Git, SVN and Mercurial

- Offers intelligent editing tools for Python
- Easy integration with Virtualenv, Docker and Vagrant
- Simple navigation and search features
- Code analysis and refactoring
- Configurable injections
- Supports tons of Python libraries
- Contains Templates and JavaScript debuggers
- Includes Python/Django debuggers
- Works with Google App Engine, additional frameworks and libraries.
- Has customizable UI, VIM emulation available

Komodo IDE

It is a polyglot IDE which supports 100+ languages and basically for dynamic languages such as Python, PHP and Ruby. It is a commercial IDE available for 21 days free trial with full functionality. ActiveState is the software company managing the development of the Komodo IDE. It also offers a trimmed version of Komodo known as Komodo Edit for simple programming tasks.

This IDE contains all kinds of features from most basic to advanced level. If you are a student or a freelancer, then you can buy it almost half of the actual price. However, it's completely free for teachers and professors from recognized institutions and universities.

It got all the features you need for web and mobile development, including support for all your languages and frameworks.

Download link

The download links for Komodo Edit(free version) and Komodo IDE(paid version) are as given here:

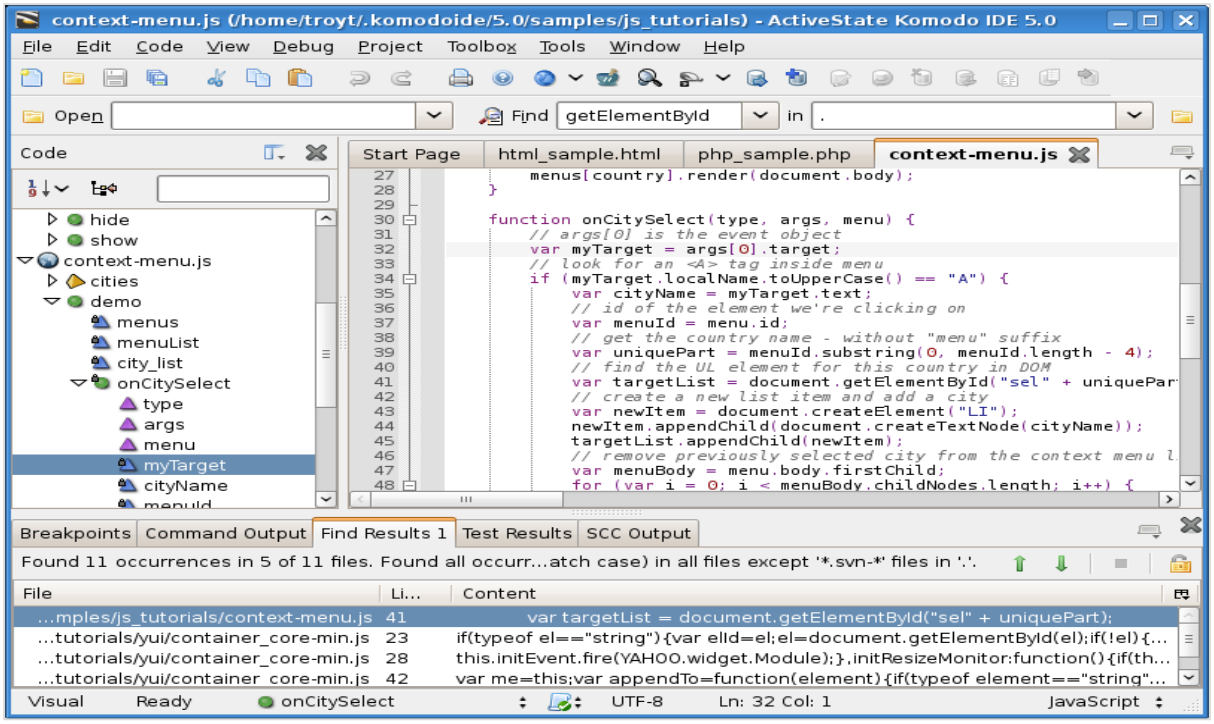
Komodo Edit (free)

https://www.activestate.com/komodo-edit

Komodo IDE (paid)

https://www.activestate.com/komodo-ide/downloads/ide

Screenshot



Why to Choose?

- Powerful IDE with support for Perl, PHP, Python, Ruby and many more.
- Cross-Platform IDE.

It includes basic features like integrated debugger support, auto complete, Document Object Model(DOM) viewer, code browser, interactive shells, breakpoint configuration, code profiling, integrated unit testing. In short, it is a professional IDE with a host of productivity-boosting features.

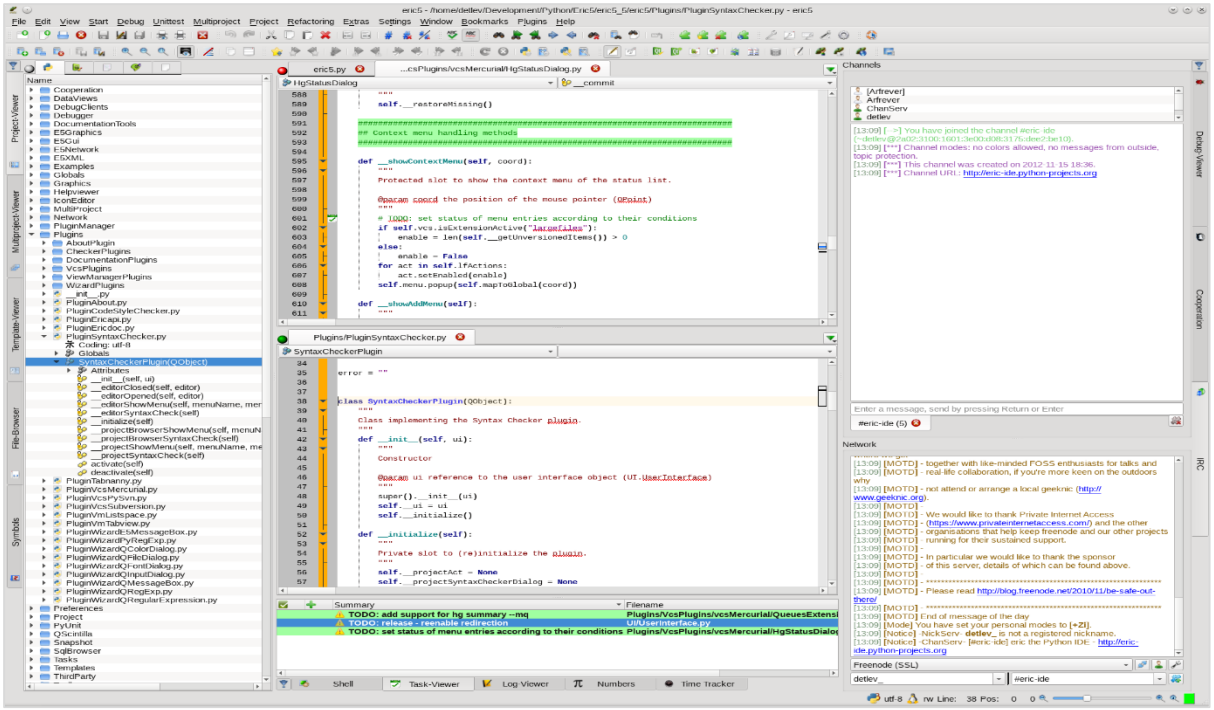
Eric Python IDE

It is an open-source IDE for Python and Ruby. Eric is a full featured editor and IDE, written in Python. It is based on the cross platform Qt GUI toolkit, integrating the highly flexible Scintilla editor control. The IDE is very much configurable and one can choose what to use and what not. You can download Eric IDE from below link: https://eric-ide.Python-projects.org/eric-download.html

Why to Choose

- Great indentation, error highlighting.
- Code assistance
- Code completion
- Code cleanup with PyLint
- Quick search
- Integrated Python debugger.

Screenshot



Choosing a Text Editor

You may not always need an IDE. For tasks such as learning to code with Python or Arduino, or when working on a quick script in shell script to help you automate some tasks a simple and light weight code-centric text editor will do. Also many text editors offer features such as syntax highlighting and in-program script execution, similar to IDEs. Some of the text editors are given here:

- Atom
- Sublime Text
- Notepad++

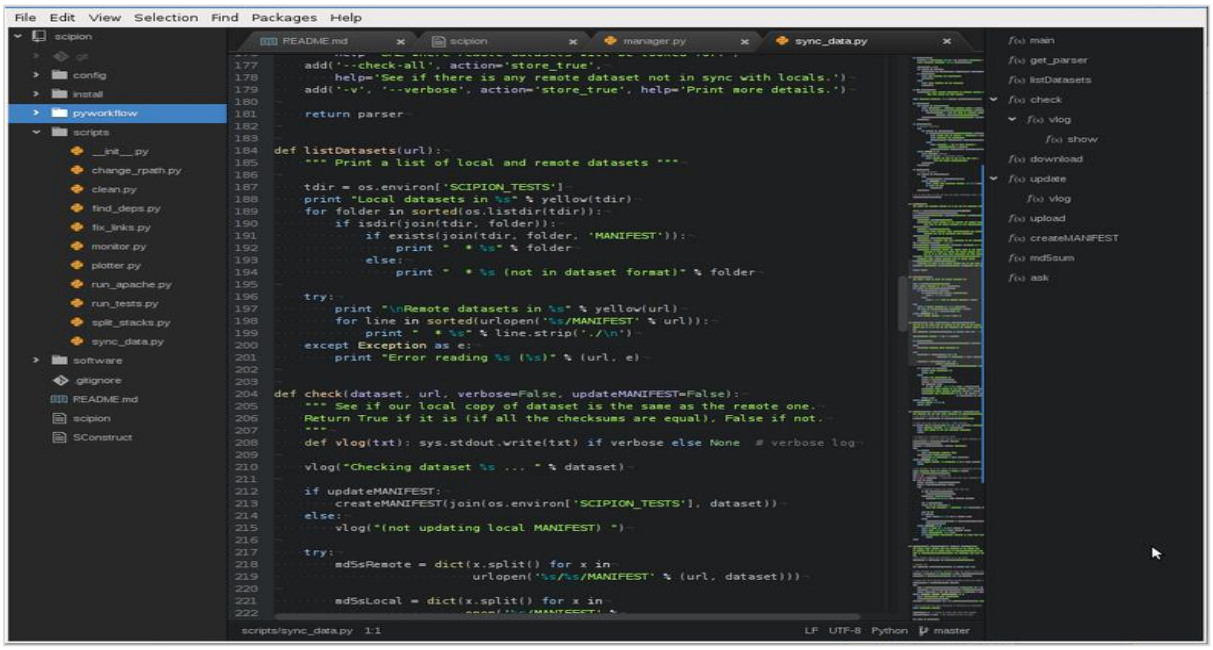
Atom Text Editor

Atom is a hackable text editor built by the team of GitHub. It is a free and open source text and code editor which means that all the code is available for you to read, modify for your own use and even contribute improvements. It is a cross-platform text editor compatible for macOS, Linux, and Microsoft Windows with support for plug-ins written in Node.js and embedded Git Control.

Download link

https://atom.io/

Screenshot



Languages Supported

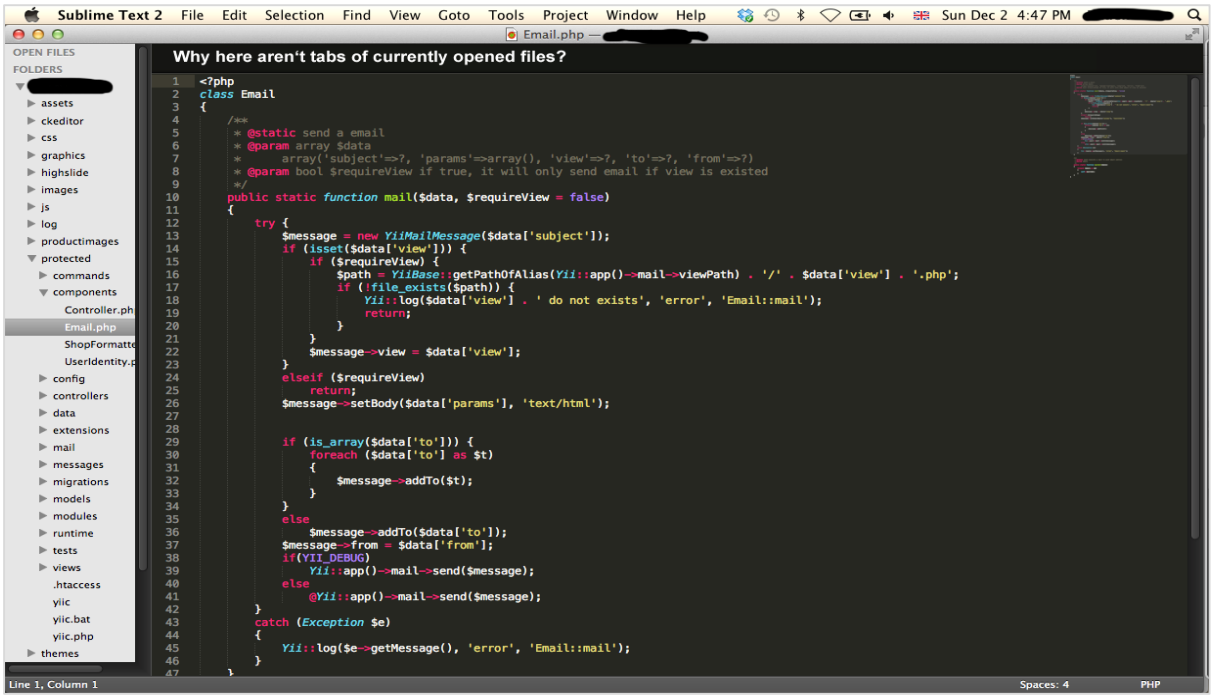
C/C++, C#, CSS, CoffeeScript, HTML, JavaScript, Java, JSON, Julia, Objective-C, PHP, Perl, Python, Ruby on Rails, Ruby, Shell script, Scala, SQL, XML, YAML and many more.

Sublime Text Editor

Sublime text is a proprietary software and it offers you a free trial version to test it before you purchase it. According to Stackoverflow's 2018 developer survey, it's the fourth most popular Development Environment.

Some of the advantages it provides is its incredible speed, ease of use and community support. It also supports many programming languages and mark-up languages, and functions can be added by users with plugins, typically community-built and maintained under free-software licenses.

Screenshot



Language supported

- Python, Ruby, JavaScript etc.

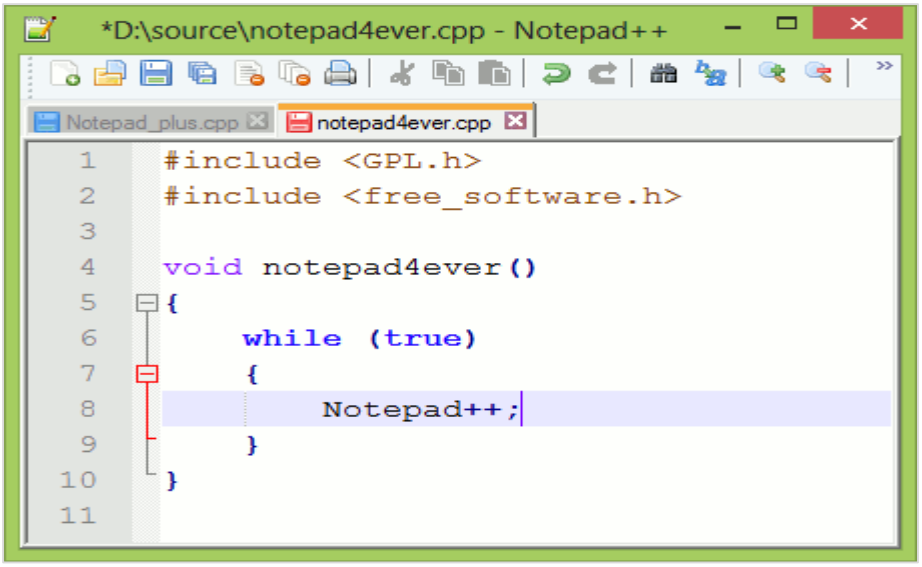
Why to Choose?

- Customize key bindings, menus, snippets, macros, completions and more.
- Auto completion feature
- Quickly Insert Text & code with sublime text snippets using snippets, field markers and place holders
- Opens Quickly
- Cross Platform support for Mac, Linux and Windows.
- Jump the cursor to where you want to go
- Select Multiple Lines, Words and Columns

Notepad ++

It's a free source code editor and Notepad replacement that supports several languages from Assembly to XML and including Python. Running in the MS windows environment, its use is governed by GPL license.
In addition to syntax highlighting, Notepad++ has some features that are particularly useful to coders.

Screenshot



Key Features

- Syntax highlighting and syntax folding
- PCRE (Perl Compatible Regular Expression) Search/Replace.
- Entirely customizable GUI
- Auto completion
- Tabbed editing
- Multi-View
- Multi-Language environment
- Launchable with different arguments.

Language Supported

- Almost every language (60+ languages) like Python, C, C++, C#, Java etc.

3. OOP in Python – Data Structures

Python data structures are very intuitive from a syntax point of view and they offer a large choice of operations. You need to choose Python data structure depending on what the data involves, if it needs to be modified, or if it is a fixed data and what access type is required, such as at the beginning/end/random etc.

Lists

A List represents the most versatile type of data structure in Python. A list is a container which holds comma-separated values (items or elements) between square brackets. Lists are helpful when we want to work with multiple related values. As lists keep data together, we can perform the same methods and operations on multiple values at once. Lists indices start from zero and unlike strings, lists are mutable.

Data Structure - List

```
>>>
>>> # Any Empty List
>>> empty_list = []
>>>
>>> # A list of String
>>> str_list = ['Life', 'Is', 'Beautiful']
>>> # A list of Integers
>>> int_list = [1, 4, 5, 9, 18]
>>>
>>> #Mixed items list
>>> mixed_list = ['This', 9, 'is', 18, 45.9, 'a', 54, 'mixed', 99, 'list']
>>> # To print the list
>>>
>>> print(empty_list)
[]
>>> print(str_list)
['Life', 'Is', 'Beautiful']
>>> print(type(str_list))
<class 'list'>
>>> print(int_list)
[1, 4, 5, 9, 18]
```

```
>>> print(mixed_list)
['This', 9, 'is', 18, 45.9, 'a', 54, 'mixed', 99, 'list']
```

Accessing Items in Python List

Each item of a list is assigned a number – that is the index or position of that number. Indexing always start from zero, the second index is one and so forth. To access items in a list, we can use these index numbers within a square bracket. Observe the following code for example:

```
>>> mixed_list = ['This', 9, 'is', 18, 45.9, 'a', 54, 'mixed', 99, 'list']
>>>
>>> # To access the First Item of the list
>>> mixed_list[0]
'This'
>>> # To access the 4th item
>>> mixed_list[3]
18
>>> # To access the last item of the list
>>> mixed_list[-1]
'list'
```

Empty Objects

Empty Objects are the simplest and most basic Python built-in types. We have used them multiple times without noticing and have extended it to every class we have created. The main purpose to write an empty class is to block something for time being and later extend and add a behavior to it.

To add a behavior to a class means to replace a data structure with an object and change all references to it. So it is important to check the data, whether it is an object in disguise, before you create anything. Observe the following code for better understanding:

```
>>> #Empty objects
>>>
>>> obj = object()
>>> obj.x = 9
Traceback (most recent call last):
 File "<pyshell#3>", line 1, in <module>
obj.x = 9
AttributeError: 'object' object has no attribute 'x'
```

So from above, we can see it's not possible to set any attributes on an object that was instantiated directly. When Python allows an object to have arbitrary attributes, it takes a certain amount of system memory to keep track of what attributes each object has, for storing both the attribute name and its value. Even if no attributes are stored, a certain amount of memory is allocated for potential new attributes.

So Python disables arbitrary properties on object and several other built-ins, by default.

```
>>> # Empty Objects
>>>
>>> class EmpObject:
     pass
>>> obj = EmpObject()
>>> obj.x = 'Hello, World!'
>>> obj.x
'Hello, World!'
```

Hence, if we want to group properties together, we could store them in an empty object as shown in the code above. However, this method is not always suggested. Remember that classes and objects should only be used when you want to specify both data and behaviors.

Tuples

Tuples are similar to lists and can store elements. However, they are immutable, so we cannot add, remove or replace objects. The primary benefits tuple provides because of its immutability is that we can use them as keys in dictionaries, or in other locations where an object requires a hash value.

Tuples are used to store data, and not behavior. In case you require behavior to manipulate a tuple, you need to pass the tuple into a function(or method on another object) that performs the action.

As tuple can act as a dictionary key, the stored values are different from each other. We can create a tuple by separating the values with a comma. Tuples are wrapped in parentheses but not mandatory. The following code shows two identical assignments .

```
>>> stock1 = 'MSFT', 95.00, 97.45, 92.45
>>> stock2 = ('MSFT', 95.00, 97.45, 92.45)
>>> type (stock1)
<class 'tuple'>
>>> type(stock2)
<class 'tuple'>
>>> stock1 == stock2
True
>>>
```

Defining a Tuple

Tuples are very similar to list except that the whole set of elements are enclosed in parentheses instead of square brackets.

Just like when you slice a list, you get a new list and when you slice a tuple, you get a new tuple.

```
>>> tupl = ('Tuple','is', 'an','IMMUTABLE', 'list')
>>> tupl
('Tuple', 'is', 'an', 'IMMUTABLE', 'list')
>>> tupl[0]
'Tuple'
>>> tupl[-1]
'list'
>>> tupl[1:3]
('is', 'an')
```

Python Tuple Methods

The following code shows the methods in Python tuples:

```
>>> tupl
('Tuple', 'is', 'an', 'IMMUTABLE', 'list')
>>> tupl.append('new')
Traceback (most recent call last):
  File "<pyshell#148>", line 1, in <module>
    tupl.append('new')
AttributeError: 'tuple' object has no attribute 'append'
>>> tupl.remove('is')
Traceback (most recent call last):
  File "<pyshell#149>", line 1, in <module>
    tupl.remove('is')
AttributeError: 'tuple' object has no attribute 'remove'
>>> tupl.index('list')
4
>>> tupl.index('new')
Traceback (most recent call last):
  File "<pyshell#151>", line 1, in <module>
    tupl.index('new')
```

```
ValueError: tuple.index(x): x not in tuple
>>> "is" in tupl
True
>>> tupl.count('is')
1
```

From the code shown above, we can understand that tuples are immutable and hence:

- You **cannot** add elements to a tuple.
- You **cannot** append or extend a method.
- You **cannot** remove elements from a tuple.
- Tuples have **no** remove or pop method.
- Count and index are the methods available in a tuple.

Dictionary

Dictionary is one of the Python's built-in data types and it defines one-to-one relationships between keys and values.

Defining Dictionaries

Observe the following code to understand about defining a dictionary:

```
>>> # empty dictionary
>>> my_dict = {}
>>>
>>> # dictionary  with integer keys
>>> my_dict = { 1:'msft', 2: 'IT'}
>>>
>>> # dictionary with mixed keys
>>> my_dict = {'name': 'Aarav', 1: [ 2, 4, 10]}
>>>
>>> # using built-in function dict()
>>> my_dict = dict({1:'msft', 2:'IT'})
>>>
>>> # From sequence having each item as a pair
>>> my_dict = dict([(1,'msft'), (2,'IT')])
>>>
>>> # Accessing elements of a dictionary
>>> my_dict[1]
```

```
'msft'
>>> my_dict[2]
'IT'
>>> my_dict['IT']
Traceback (most recent call last):
  File "<pyshell#177>", line 1, in <module>
    my_dict['IT']
KeyError: 'IT'
>>>
```

From the above code we can observe that:

- First we create a dictionary with two elements and assign it to the variable **my_dict**. Each element is a key-value pair, and the whole set of elements is enclosed in curly braces.
- The number **1** is the key and **msft** is its value. Similarly, **2** is the key and **IT** is its value.
- You can get values by key, but not vice-versa. Thus when we try **my_dict['IT']** , it raises an exception, because **IT** is not a key.

Modifying Dictionaries

Observe the following code to understand about modifying a dictionary:

```
>>> # Modifying a Dictionary
>>>
>>> my_dict
{1: 'msft', 2: 'IT'}
>>> my_dict[2] = 'Software'
>>> my_dict
{1: 'msft', 2: 'Software'}
>>>
>>> my_dict[3] = 'Microsoft Technologies'
>>> my_dict
{1: 'msft', 2: 'Software', 3: 'Microsoft Technologies'}
```

From the above code we can observe that:

- You cannot have duplicate keys in a dictionary. Altering the value of an existing key will delete the old value.
- You can add new key-value pairs at any time.

- Dictionaries have no concept of order among elements. They are simple unordered collections.

Mixing Data types in a Dictionary

Observe the following code to understand about mixing data types in a dictionary:

```
>>> # Mixing Data Types in a Dictionary
>>>
>>> my_dict
{1: 'msft', 2: 'Software', 3: 'Microsoft Technologies'}
>>> my_dict[4] = 'Operating System'
>>> my_dict
{1: 'msft', 2: 'Software', 3: 'Microsoft Technologies', 4: 'Operating System'}
>>> my_dict['Bill Gates'] = 'Owner'
>>> my_dict
{1: 'msft', 2: 'Software', 3: 'Microsoft Technologies', 4: 'Operating System',
'Bill Gates': 'Owner'}
```

From the above code we can observe that:

- Not just strings but dictionary value can be of any data type including strings, integers, including the dictionary itself.

- Unlike dictionary values, dictionary keys are more restricted, but can be of any type like strings, integers or any other.

Deleting Items from Dictionaries

Observe the following code to understand about deleting items from a dictionary:

```
>>> # Deleting Items from a Dictionary
>>>
>>> my_dict
{1: 'msft', 2: 'Software', 3: 'Microsoft Technologies', 4: 'Operating System',
'Bill Gates': 'Owner'}
>>>
>>> del my_dict['Bill Gates']
>>> my_dict
{1: 'msft', 2: 'Software', 3: 'Microsoft Technologies', 4: 'Operating System'}
>>>
>>> my_dict.clear()
>>> my_dict
{}
```

From the above code we can observe that:

- **del** lets you delete individual items from a dictionary by key.
- **clear** deletes all items from a dictionary.

Sets

Set() is an unordered collection with no duplicate elements. Though individual items are immutable, set itself is mutable, that is we can add or remove elements/items from the set. We can perform mathematical operations like union, intersection etc. with set.

Though sets in general can be implemented using trees, set in Python can be implemented using a hash table. This allows it a highly optimized method for checking whether a specific element is contained in the set

Creating a set

A set is created by placing all the items (elements) inside curly braces **{}**, separated by comma or by using the built-in function **set()**. Observe the following lines of code:

```
>>> #set of integers
>>> my_set = {1,2,4,8}
>>> print(my_set)
{8, 1, 2, 4}
>>>
>>> #set of mixed datatypes
>>> my_set = {1.0, "Hello World!", (2, 4, 6)}
>>> print(my_set)
{1.0, (2, 4, 6), 'Hello World!'}
>>>
```

Methods for Sets

Observe the following code to understand about methods for sets:

```
>>> >>> #METHODS FOR SETS
>>>
>>> #add(x) Method
>>> topics = {'Python', 'Java', 'C#'}
>>> topics.add('C++')
>>> topics
{'C#', 'C++', 'Java', 'Python'}
>>>
```

```
>>> #union(s) Method, returns a union of two set.
```

```
>>> topics
{'C#', 'C++', 'Java', 'Python'}
>>> team = {'Developer', 'Content Writer', 'Editor','Tester'}
>>> group = topics.union(team)
>>> group
{'Tester', 'C#', 'Python', 'Editor', 'Developer', 'C++', 'Java', 'Content
Writer'}
>>> # intersets(s) method, returns an intersection of two sets
>>> inters = topics.intersection(team)
>>> inters
set()
>>>
>>> # difference(s) Method, returns a set containing all the elements of
invoking set but not of the second set.
>>>
>>> safe = topics.difference(team)
>>> safe
{'Python', 'C++', 'Java', 'C#'}
>>>
>>> diff = topics.difference(group)
>>> diff
set()
>>> #clear() Method, Empties the whole set.
>>> group.clear()
>>> group
set()
>>>
```

Operators for Sets

Observe the following code to understand about operators for sets:

```
>>> # PYTHON SET OPERATIONS
>>>
>>> #Creating two sets
>>> set1 = set()
>>> set2 = set()
>>>
>>> # Adding elements to set
```

```
>>> for i in range(1,5):
      set1.add(i)
>>> for j in range(4,9):
      set2.add(j)

>>> set1
{1, 2, 3, 4}
>>> set2
{4, 5, 6, 7, 8}
>>>
>>> #Union of set1 and set2
>>> set3 = set1 | set2 # same as set1.union(set2)
>>> print('Union of set1 & set2: set3 = ', set3)
Union of set1 & set2: set3 =  {1, 2, 3, 4, 5, 6, 7, 8}
>>>
>>> #Intersection of set1 & set2
>>> set4 = set1 & set2  # same as set1.intersection(set2)
>>> print('Intersection of set1 and set2: set4 = ', set4)
Intersection of set1 and set2: set4 =  {4}
>>>
>>> # Checking relation between set3 and set4
>>> if set3 > set4:  # set3.issuperset(set4)
      print('Set3 is superset of set4')
elif set3 < set4:    #set3.issubset(set4)
      print('Set3 is subset of set4')
else:   #set3 == set4
      print('Set 3 is  same as set4')

Set3 is superset of set4
>>>
>>> # Difference between set3 and set4
>>> set5 = set3 - set4
>>> print('Elements in set3 and not in set4: set5 = ', set5)
Elements in set3 and not in set4: set5 =  {1, 2, 3, 5, 6, 7, 8}
>>>
>>> # Check if set4 and set5 are disjoint sets
>>> if set4.isdisjoint(set5):
```

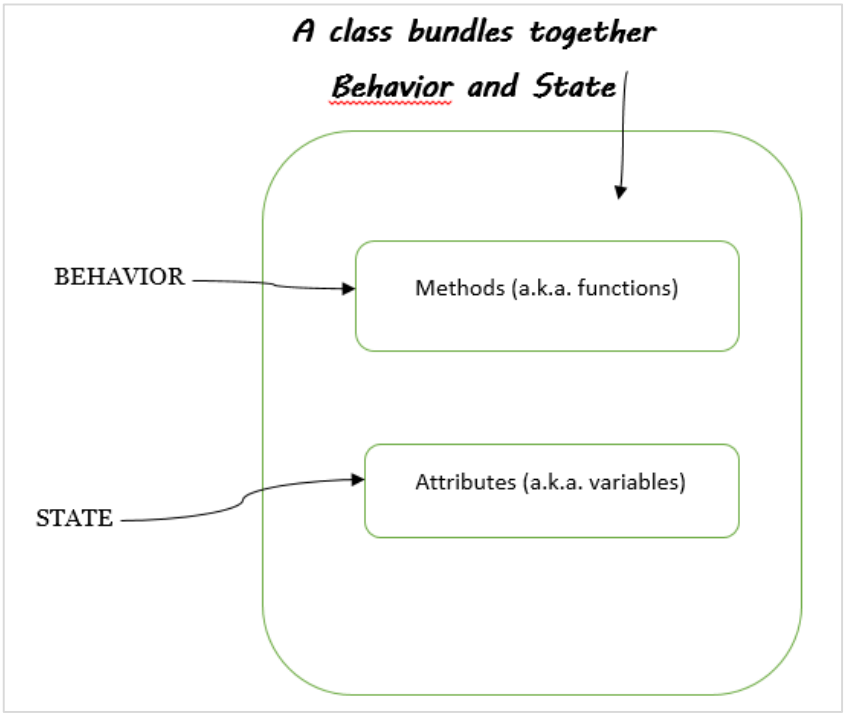
```
     print('Set4 and set5 have nothing in common\n')
Set4 and set5 have nothing in common

>>> # Removing all the values of set5
>>> set5.clear()
>>> set5 set()
```

4. OOP in Python – Building Blocks

In this chapter, we will discuss object oriented terms and programming concepts in detail. Class is a just a factory for an instance. This factory contains the blueprint which describes how to make the instances. An instances or object are constructed from the class. In most cases, we can have more than one instances of a class. Every instance has a set of attribute and these attributes are defined in a class, so every instance of a particular class is expected to have the same attributes.

Class Bundles : Behavior and State

A class will let you bundle together the behavior and state of an object. Observe the following diagram for better understanding:



The following points are worth notable when discussing class bundles:

- The word **behavior** is identical to **function** – it is a piece of code that does something (or implements a behavior)
- The word **state** is identical to **variables** – it is a place to store values within a class.
- When we assert a class behavior and state together, it means that a class packages functions and variables.

Classes have methods and attributes

In Python, creating a method defines a class behavior. The word **method** is the OOP name given to a function that is defined within a class. To sum up:

- **Class functions** is synonym for **methods**
- **Class variables** is synonym for **name attributes**.
- **Class** – a blueprint for an instance with exact behavior.
- **Object** – one of the instances of the class, perform functionality defined in the class.
- **Type** – indicates the class the instance belongs to
- **Attribute** – Any object value: object.attribute
- **Method** – a "callable attribute" defined in the class

Observe the following piece of code for example:

```
var = "Hello, John"
print( type (var))       # < type 'str'> or <class 'str'>
print(var.upper())       # upper() method is called, HELLO, JOHN
```

Creation and Instantiation

The following code shows how to create our first class and then its instance.

```
class MyClass(object):
    pass
# Create first instance of MyClass
this_obj = MyClass()
print(this_obj)
# Another instance of MyClass
that_obj = MyClass()
print (that_obj)
```

Here we have created a class called **MyClass** and which does not do any task. The argument **object** in **MyClass** class involves class inheritance and will be discussed in later chapters. **pass** in the above code indicates that this block is empty, that is it is an empty class definition.

Let us create an instance **this_obj** of **MyClass()** class and print it as shown:

```
<__main__.MyClass object at 0x03B08E10>
<__main__.MyClass object at 0x0369D390>
```

Here, we have created an instance of **MyClass**. The hex code refers to the address where the object is being stored. Another instance is pointing to another address.

Now let us define one variable inside the class **MyClass()** and get the variable from the instance of that class as shown in the following code:

```
class MyClass(object):
    var = 9

# Create first instance of MyClass
this_obj = MyClass()
print(this_obj.var)

# Another instance of MyClass

that_obj = MyClass()
print (that_obj.var)
```

Output

You can observe the following output when you execute the code given above:

```
9
9
```

As instance knows from which class it is instantiated, so when requested for an attribute from an instance, the instance looks for the attribute and the class. This is called the **attribute lookup**.

Instance Methods

A function defined in a class is called a **method**. An instance method requires an instance in order to call it and requires no decorator. When creating an instance method, the first parameter is always **self**. Though we can call it (self) by any other name, it is recommended to use self, as it is a naming convention.

```
class MyClass(object):
    var=9
    def firstM(self):
        print("hello, World")
obj = MyClass()
print(obj.var)
obj.firstM()
```

Output

You can observe the following output when you execute the code given above:

```
9
hello, World
```

Note that in the above program, we defined a method with **self** as argument. But we cannot call the method as we have not declared any argument to it.

```
class MyClass(object):
    def firstM(self):
        print("hello, World")
        print(self)
obj = MyClass()
obj.firstM()
print(obj)
```

Output

You can observe the following output when you execute the code given above:

```
hello, World
<__main__.MyClass object at 0x036A8E10>
<__main__.MyClass object at 0x036A8E10>
```

Encapsulation

Encapsulation is one of the fundamentals of OOP. OOP enables us to hide the complexity of the internal working of the object which is advantageous to the developer in the following ways:

- Simplifies and makes it easy to understand to use an object without knowing the internals.
- Any change can be easily manageable.

Object-oriented programming relies heavily on encapsulation. The terms encapsulation and abstraction (also called data hiding) are often used as synonyms. They are nearly synonymous, as abstraction is achieved through encapsulation.

Encapsulation provides us the mechanism of restricting the access to some of the object's components, this means that the internal representation of an object can't be seen from outside of the object definition. Access to this data is typically achieved through special methods: **Getters** and **Setters**.

This data is stored in instance attributes and can be manipulated from anywhere outside the class. To secure it, that data should only be accessed using instance methods. Direct access should not be permitted.

```
class MyClass(object):
    def setAge(self, num):
        self.age = num

    def getAge(self):
        return self.age

zack = MyClass()
zack.setAge(45)
print(zack.getAge())

zack.setAge("Fourty Five")
print(zack.getAge())
```

Output

You can observe the following output when you execute the code given above:

```
45
Fourty Five
```

The data should be stored only if it is correct and valid, using Exception handling constructs. As we can see above, there is no restriction on the user input to **setAge()** method. It could be a string, a number, or a list. So we need to check onto above code to ensure correctness of being stored.

```
class MyClass(object):
    def setAge(self, num):
        self.age = num

    def getAge(self):
        return self.age
```

```
zack = MyClass()
zack.setAge(45)
```

```
print(zack.getAge())
zack.setAge("Fourty Five")
print(zack.getAge())
```

Init Constructor

The **__init__** method is implicitly called as soon as an object of a class is instantiated. This will initialize the object.

```
x = MyClass()
```

The line of code shown above will create a new instance and assigns this object to the local variable **x**.

The instantiation operation, that is **calling a class object**, creates an empty object. Many classes like to create objects with instances customized to a specific initial state. Therefore, a class may define a special method named ` __init__() ` as shown:

```
def __init__(self):
     self.data = []
```

Python calls __init__ during the instantiation to define an additional attribute that should occur when a class is instantiated that may be setting up some beginning values for that object or running a routine required on instantiation. So in this example, a new, initialized instance can be obtained by:

```
x = MyClass()
```

The __init__() method can have single or multiple arguments for a greater flexibility. The init stands for initialization, as it initializes attributes of the instance. It is called the constructor of a class.

```
 class myclass(object):
    def __init__(self,aaa, bbb):
        self.a = aaa
        self.b = bbb

x = myclass(4.5, 3)
print(x.a, x.b)
```

Output

```
4.5 3
```

Class Attributes

The attribute defined in the class is called "class attributes' and the attributes defined in the function is called 'instance attributes'. While defining, these attributes are not prefixed by self, as these are the property of the class and not of a particular instance.

The class attributes can be accessed by the class itself (className.attributeName) as well as by the instances of the class (inst.attributeName). So, the instances have access to both the instance attribute as well as class attributes.

```
>>> class myclass():
     age = 21
>>> myclass.age
21
>>> x = myclass()
>>> x.age
21
>>>
```

A class attribute can be overridden in an instance, even though it is not a good method to break encapsulation.

There is a lookup path for attributes in Python. The first being the method defined within the class, and then the class above it.

```
>>> class myclass(object):
     classy = 'class value'
>>> dd = myclass()
>>> print (dd.classy)    # This should return the string 'class value'
class value
>>>
>>> dd.classy = "Instance Value"
>>> print(dd.classy)     # Return the string "Instance Value"
Instance Value
>>>
>>> # This will delete the value set for 'dd.classy' in the instance.
>>> del dd.classy
>>> >>> # Since the overriding attribute was deleted, this will print 'class
value'.


>>> print(dd.classy)
class value
>>>
```

We are overriding the 'classy' class attribute in the instance **dd**. When it's overridden, the Python interpreter reads the overridden value. But once the new value is deleted with 'del', the overridden value is no longer present in the instance, and hence the lookup goes a level above and gets it from the class.

Working with Class and Instance Data

In this section, let us understand how the class data relates to the instance data. We can store data either in a class or in an instance. When we design a class, we decide which data belongs to the instance and which data should be stored into the overall class.

An instance can access the class data. If we create multiple instances, then these instances can access their individual attribute values as well the overall class data.

Thus, a class data is the data that is shared among all the instances. Observe the code given below for better undersanding:

```
class InstanceCounter(object):
    count = 0                   # class attribute, will be accessible to all
instances
    def __init__(self, val):
        self.val = val
        InstanceCounter.count +=1   # Increment the value of class attribute,
accessible through class name
# In above line, class ('InstanceCounter') act as an object
    def set_val(self, newval):
        self.val = newval

    def get_val(self):
        return self.val

    def get_count(self):
        return InstanceCounter.count
a = InstanceCounter(9)
b = InstanceCounter(18)
c = InstanceCounter(27)

for obj in (a, b, c):
    print ('val of obj: %s' %(obj.get_val()))    # Initialized value ( 9, 18,
27)
    print ('count: %s' %(obj.get_count()))        # always 3
```

Output

```
val of obj: 9
count: 3
val of obj: 18
count: 3
val of obj: 27
count: 3
```

In short, class attributes are same for all instances of class whereas instance attributes is particular for each instance. For two different instances, we will have two different instance attributes.

```
class myClass:
    class_attribute = 99

    def class_method(self):
        self.instance_attribute = 'I am instance attribute'

print (myClass.__dict__)
```

Output

You can observe the following output when you execute the code given above:

```
{'__module__': '__main__', 'class_attribute': 99, 'class_method': <function
myClass.class_method at 0x04128D68>, '__dict__': <attribute '__dict__' of
'myClass' objects>, '__weakref__': <attribute '__weakref__' of 'myClass'
objects>, '__doc__': None}
```

The instance attribute **myClass.__dict__** as shown:

```
>>> a = myClass()
>>> a.class_method()
>>> print(a.__dict__)
{'instance_attribute': 'I am instance attribute'}
```

5. OOP in Python – Object Oriented Shortcuts

This chapter talks in detail about various built-in functions in Python, file I/O operations and overloading concepts.

Python Built-in Functions

The Python interpreter has a number of functions called built-in functions that are readily available for use. In its latest version, Python contains 68 built-in functions as listed in the table given below:

BUILT-IN FUNCTIONS				
abs()	dict()	help()	min()	setattr()
all()	dir()	hex()	next()	slice()
any()	divmod()	id()	object()	sorted()
ascii()	enumerate()	input()	oct()	staticmethod()
bin()	eval()	int()	open()	str()
bool()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	
delattr()	hash()	memoryview()	set()	

This section discusses some of the important functions in brief:

len() function

The len() function gets the length of strings, list or collections. It returns the length or number of items of an object, where object can be a string, list or a collection.

```
>>> len(['hello', 9 , 45.0, 24])
4
```

len() function internally works like **list.__len__()** or **tuple.__len__()**. Thus, note that len() works only on objects that has a **__len__()** method.

```
>>> set1
{1, 2, 3, 4}
>>> set1.__len__()
4
```

However, in practice, we prefer **len()** instead of the **__len__()** function because of the following reasons:

- It is more efficient. And it is not necessary that a particular method is written to refuse access to special methods such as __len__.
- It is easy to maintain.
- It supports backward compatibility.

Reversed(seq)

It returns the reverse iterator. seq must be an object which has __reversed__() method or supports the sequence protocol (the __len__() method and the __getitem__() method). It is generally used in **for** loops when we want to loop over items from back to front.

```
>>> normal_list = [2, 4, 5, 7, 9]
>>>
>>> class CustomSequence():
     def __len__(self):
           return 5
     def __getitem__(self,index):
           return "x{0}".format(index)
>>> class funkyback():
     def __reversed__(self):
           return 'backwards!'
>>> for seq in normal_list, CustomSequence(), funkyback():
           print('\n{}: '.format(seq.__class__.__name__), end="")
           for item in reversed(seq):
                 print(item, end=", ")
```

The for loop at the end prints the reversed list of a normal list, and instances of the two custom sequences. The output shows that **reversed()** works on all the three of them, but has a very different results when we define **__reversed__**.

Output

You can observe the following output when you execute the code given above:

```
list: 9, 7, 5, 4, 2,
CustomSequence: x4, x3, x2, x1, x0,
funkyback: b, a, c, k, w, a, r, d, s, !,
```

Enumerate

The **enumerate ()** method adds a counter to an iterable and returns the enumerate object.

The syntax of enumerate () is:

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

Here the second argument **start** is optional, and by default index starts with **zero (0)**.

```
>>> # Enumerate
>>> names = ['Rajesh', 'Rahul', 'Aarav', 'Sahil', 'Trevor']
>>> enumerate(names)
<enumerate object at 0x031D9F80>
>>> list(enumerate(names))
[(0, 'Rajesh'), (1, 'Rahul'), (2, 'Aarav'), (3, 'Sahil'), (4, 'Trevor')]
>>>
```

So **enumerate()** returns an iterator which yields a tuple that keeps count of the elements in the sequence passed. Since the return value is an iterator, directly accessing it is not much useful. A better approach for enumerate() is keeping count within a for loop.

```
>>> for i, n in enumerate(names):
      print('Names number: ' + str(i))
      print(n)
Names number: 0
Rajesh
Names number: 1
Rahul
Names number: 2
Aarav
Names number: 3
Sahil
Names number: 4
           Trevor
```

There are many other functions in the standard library, and here is another list of some more widely used functions:

- **hasattr**, **getattr**, **setattr** and **delattr**, which allows attributes of an object to be manipulated by their string names.

- **all** and **any**, which accept an iterable object and return **True** if all, or any, of the items evaluate to be true.

- **nzip**, which takes two or more sequences and returns a new sequence of tuples, where each tuple contains a single value from each sequence.

File I/O

The concept of files is associated with the term object-oriented programming. Python has wrapped the interface that operating systems provided in abstraction that allows us to work with file objects.

The **open()** built-in function is used to open a file and return a file object. It is the most commonly used function with two arguments:

```
open(filename, mode)
```

The open() function calls two argument, first is the filename and second is the mode. Here mode can be 'r' for read only mode, 'w' for only writing (an existing file with the same name will be erased), and 'a' opens the file for appending, any data written to the file is automatically added to the end. 'r+' opens the file for both reading and writing. The default mode is read only.

On windows, 'b' appended to the mode opens the file in binary mode, so there are also modes like 'rb', 'wb' and 'r+b'.

```
>>> text = 'This is the first line'
>>> file = open('datawork','w')
>>> file.write(text)
22
>>> file.close()
```

In some cases, we just want to append to the existing file rather then over-writing it, for that we could supply the value 'a' as a mode argument, to append to the end of the file, rather than completely overwriting existing file contents.

```
>>> f = open('datawork','a')
>>> text1 = ' This is second line'
>>> f.write(text1)
20
>>> f.close()
```

Once a file is opened for reading, we can call the read, readline, or readlines method to get the contents of the file. The read method returns the entire contents of the file as a str or bytes object, depending on whether the second argument is 'b'.

For readability, and to avoid reading a large file in one go, it is often better to use a for loop directly on a file object. For text files, it will read each line, one at a time, and we can process it inside the loop body. For binary files however it's better to read fixed-sized chunks of data using the read() method, passing a parameter for the maximum number of bytes to read.

```
>>> f = open('fileone','r+')
>>> f.readline()
'This is the first line. \n'
>>> f.readline()
'This is the second line. \n'
```

Writing to a file, through write method on file objects will writes a string (bytes for binary data) object to the file. The writelines method accepts a sequence of strings and write each of the iterated values to the file. The writelines method does not append a new line after each item in the sequence.

Finally the close() method should be called when we are finished reading or writing the file, to ensure any buffered writes are written to the disk, that the file has been properly cleaned up and that all resources tied with the file are released back to the operating system. It's a better approach to call the close() method but technically this will happen automatically when the script exists.

An alternative to method overloading

Method overloading refers to having multiple methods with the same name that accept different sets of arguments.

Given a single method or function, we can specify the number of parameters ourself. Depending on the function definition, it can be called with zero, one, two or more parameters.

```
class Human:
    def sayHello(self, name=None):
        if name is not None:
            print('Hello ' + name)
        else:
            print('Hello ')

#Create Instance
```

```
obj = Human()

#Call the method, else part will be executed
obj.sayHello()

#Call the method with a parameter, if part will be executed
obj.sayHello('Rahul')
```

Output

```
Hello
Hello Rahul
```

Default Arguments

Functions Are Objects Too

A callable object is an object can accept some arguments and possibly will return an object. A function is the simplest callable object in Python, but there are others too like classes or certain class instances.

Every function in a Python is an object. Objects can contain methods or functions but object is not necessary a function.

```
def my_func():
    print('My function was called')
my_func.description = 'A silly function'
def second_func():

    print('Second function was called')

second_func.description = 'One more sillier function'

def another_func(func):
    print("The description:", end=" ")
    print(func.description)
    print('The name: ', end=' ')
    print(func.__name__)
                print('The class:', end=' ')
```

```
    print(func.__class__)
    print("Now I'll call the function passed in")
    func()



another_func(my_func)
another_func(second_func)
```

In the above code, we are able to pass two different functions as argument into our third function, and get different output for each one:

```
The description: A silly function
The name:  my_func
The class: <class 'function'>
Now I'll call the function passed in
My function was called
The description: One more sillier function
The name:  second_func
The class: <class 'function'>
Now I'll call the function passed in
Second function was called
```

callable objects

Just as functions are objects that can have attributes set on them, it is possible to create an object that can be called as though it were a function.

In Python any object with a __call__() method can be called using function-call syntax.

6. OOP in Python – Inheritance and Polymorphism

Inheritance and polymorphism – this is a very important concept in Python. You must understand it better if you want to learn.

Inheritance

One of the major advantages of Object Oriented Programming is re-use. Inheritance is one of the mechanisms to achieve the same. Inheritance allows programmer to create a general or a base class first and then later extend it to more specialized class. It allows programmer to write better code.

Using inheritance you can use or inherit all the data fields and methods available in your base class. Later you can add you own methods and data fields, thus inheritance provides a way to organize code, rather than rewriting it from scratch.

In object-oriented terminology when class X extend class Y, then Y is called super/parent/base class and X is called subclass/child/derived class. One point to note here is that only data fields and method which are not private are accessible by child classes. Private data fields and methods are accessible only inside the class.

Syntax to create a derived class is:

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

Inheriting Attributes

Now look at the below example:

```
# Inheriting_Attributes

class Date(object):                # Inherits from the 'object' Class
    def get_date(self):
        return '2018-02-02'


class Time(Date):                # Inherits from the 'Date' Class
    def get_time(self):
        return '09:00:00'

dt = Date()
print("Get date from the Date class: ",dt.get_date())

tm = Time()
print("Get time from the Time class: ",tm.get_time())
print('Get date from time class by inheriting or calling Date class method: ',tm.get_date())          # Found this method in the 'Date' class
```

Output

```
Get date from the Date class:  2018-02-02
Get time from the Time class:  09:00:00
Get date from time class by inheriting or calling Date class method:  2018-02-02
```

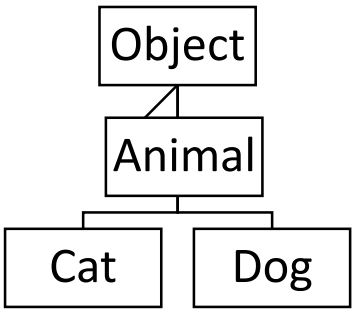
We first created a class called Date and pass the object as an argument, here-object is built-in class provided by Python. Later we created another class called time and called the Date class as an argument. Through this call we get access to all the data and attributes of Date class into the Time class. Because of that when we try to get the get_date method from the Time class object tm we created earlier possible.

Object.Attribute Lookup Hierarchy

- The instance
- The class
- Any class from which this class inherits

Inheritance Examples

Let's take a closure look into the inheritance example:



Let's create couple of classes to participate in examples:

- Animal: Class simulate an animal
- Cat: Subclass of Animal
- Dog: Subclass of Animal

In Python, constructor of class used to create an object (instance), and assign the value for the attributes.

Constructor of subclasses always called to a constructor of parent class to initialize value for the attributes in the parent class, then it start assign value for its attributes.

```
#Inheritance Example
class Animal(object):

    def __init__(self, name):
        self.name = name
    def eat(self, food):          # Common method(or property) of both subclass
        print ('%s is eating %s. ' %(self.name, food))

class Dog(Animal):

    def fetch(self, thing):
        print('%s goes after the %s!' %(self.name, thing))

class Cat(Animal):

    def swatstring(self):
        print('%s shreds the string!' %(self.name))

d = Dog('Ranger')       # Created Dog object, d
c = Cat('MeOw')         # Created Cat object, c

d.fetch('ball')        # Rover goes after the paper!=
c.swatstring()          # Fluffy shreds the string!
d.eat('Dog Food')       # Rover is eating dog Food
c.eat('Cat Food')       #Fluffy is eating cat food.
d.swatstring()          #Attribute Error: 'Dog' object has no Attribute 'swatstring'
```

Output

```
Ranger goes after the ball!
MeOw shreds the string!
Ranger is eating Dog Food.
MeOw is eating Cat Food.
Traceback (most recent call last):
  File "animal.py", line 27, in <module>
    d.swatstring()          #Attribute Error: 'Dog' object has no Attribute 'swatstring'
AttributeError: 'Dog' object has no attribute 'swatstring'
```

In the above example, we see the command attributes or methods we put in the parent class so that all subclasses or child classes will inherits that property from the parent class.

If a subclass try to inherits methods or data from another subclass then it will through an error as we see when Dog class try to call swatstring() methods from that cat class, it throws an error(like AttributeError in our case).

Polymorphism ("MANY SHAPES")

Polymorphism is an important feature of class definition in Python that is utilized when you have commonly named methods across classes or subclasses. This permits functions to use entities of different types at different times. So, it provides flexibility and loose coupling so that code can be extended and easily maintained over time.

This allows functions to use objects of any of these polymorphic classes without needing to be aware of distinctions across the classes.

Polymorphism can be carried out through inheritance, with subclasses making use of base class methods or overriding them.

Let understand the concept of polymorphism with our previous inheritance example and add one common method called show_affection in both subclasses:

From the example we can see, it refers to a design in which object of dissimilar type can be treated in the same manner or more specifically two or more classes with method of the same name or common interface because same method(show_affection in below example) is called with either type of objects.

```
#Polymorphism Example
class Animal(object):

    def __init__(self, name):
        self.name = name
    def eat(self, food):           # Common method(or property) of both subclass
        print ('{0} eats {1}'.format(self.name, food))

class Dog(Animal):

    def fetch(self, thing):
        print('{0} goes after the {1}!'.format(self.name, thing))

    def show_affection(self):      # Same method is called in Dog class
        print('{0} wags tail'.format(self.name))

class Cat(Animal):

    def swatstring(self):
        print('{0} shreds the string!'.format(self.name))

    def show_affection(self):      # Same method is called in Cat class
        print('{0} purrs'.format(self.name))


for a in (Dog('Rover'), Cat('Fluffy'), Cat('Precious'), Dog('Scout')): a.show_affection()      # Same method is called with different attribute
```

Output

```
Rover wags tail
Fluffy purrs
Precious purrs
Scout wags tail
```

So, all animals show affections (show_affection), but they do differently. The "show_affection" behaviors is thus polymorphic in the sense that it acted differently depending on the animal. So, the abstract "animal" concept does not actually "show_affection", but specific animals(like dogs and cats) have a concrete implementation of the action "show_affection".

Python itself have classes that are polymorphic. Example, the len() function can be used with multiple objects and all return the correct output based on the input parameter.

```
>>> len('hello')   # passing an string to len function
5
>>> len([1, 2, 3]) # passing a list of 3 elements
3
>>> len(('x', 'y', 'z')) # passing a tuple of 3 elements
3
>>> len({'a':9, 'b':18}) # passing a dictionary of 2 pairs
2
>>> # illustrate it further,
>>> dir('hello')
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '
__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__
mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__
', '__str__', '__subclasshook__', 'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find', 'format',
'format_map', 'index', 'isalnum', 'isalpha', 'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace'
, 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartit
ion', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
>>>
>>> dir([1,2,3])
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__get
attribute__', '__getitem__', '__gt__', '__hash__', '__iadd__', '__imul__', '__init__', '__init_subclass__', '__iter__', '__le__', '__
len__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__', '__setattr__
', '__setitem__', '__sizeof__', '__str__', '__subclasshook__', 'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop'
, 'remove', 'reverse', 'sort']
>>> dir({'a':9})
['__class__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__
', '__getitem__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__ne__', '__new_
_', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__setitem__', '__sizeof__', '__str__', '__subclasshook__', 'clear', 'c
opy', 'fromkeys', 'get', 'items', 'keys', 'pop', 'popitem', 'setdefault', 'update', 'values']
>>>
```

Overriding

In Python, when a subclass contains a method that overrides a method of the superclass, you can also call the superclass method by calling
Super(Subclass, self).method instead of self.method.

Example

```
class Thought(object):
    def __init__(self):
        pass
    def message(self):
        print("Thought, always come and go")


class Advice(Thought):
    def __init__(self):
        super(Advice, self).__init__()
    def message(self):
        print('Warning: Risk is always involved when you are dealing with
market!')
```

Inheriting the Constructor

If we see from our previous inheritance example, __init__ was located in the parent class in the up 'cause the child class dog or cat didn't've __init__ method in it. Python used the inheritance attribute lookup to find __init__ in animal class. When we created the child class, first it will look the __init__ method in the dog class, then it didn't find it then looked into parent class Animal and found there and called that there. So as our class design became complex we may wish to initialize a instance firstly processing it through parent class constructor and then through child class constructor.

```
import random

class Animal(object):

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

class Dog(Animal):

    def __init__(self, name):
        super(Dog, self).__init__(name)
        self.breed = random.choice(['Doberman', 'German shepherd', 'Beagle'])

    def fetch(self, thing):
        print('%s goes after the %s!' %(self.name, thing))

d = Dog('dogname')

print(d.name)
print(d.breed)
```

Output

```
dogname
German shepherd
```

In above example- all animals have a name and all dogs a particular breed. We called parent class constructor with super. So dog has its own __init__ but the first thing that happen is we call super. Super is built in function and it is designed to relate a class to its super class or its parent class.

In this case we saying that get the super class of dog and pass the dog instance to whatever method we say here the constructor __init__. So in another words we are calling parent class Animal __init__ with the dog object. You may ask why we won't just say Animal __init__ with the dog instance, we could do this but if the name of animal class were to change, sometime in the future. What if we wanna rearrange the class hierarchy,

so the dog inherited from another class. Using super in this case allows us to keep things modular and easy to change and maintain.

So in this example we are able to combine general __init__ functionality with more specific functionality. This gives us opportunity to separate common functionality from the specific functionality which can eliminate code duplication and relate class to one another in a way that reflects the system overall design.

Conclusion

- __init__ is like any other method; it can be inherited
- If a class does not have a __init__ constructor, Python will check its parent class to see if it can find one.
- As soon as it finds one, Python calls it and stops looking
- We can use the super () function to call methods in the parent class.
- We may want to initialize in the parent as well as our own class.

Multiple Inheritance and the Lookup Tree

As its name indicates, multiple inheritance is Python is when a class inherits from multiple classes.
For example, a child inherits personality traits from both parents (Mother and Father).

Python Multiple Inheritance Syntax

To make a class inherits from multiple parents classes, we write the the names of these classes inside the parentheses to the derived class while defining it. We separate these names with comma.
Below is an example of that:

```
>>> class Mother:
      pass


>>> class Father:
     pass


>>> class Child(Mother, Father):
     pass


>>> issubclass(Child, Mother) and issubclass(Child, Father)
True
```

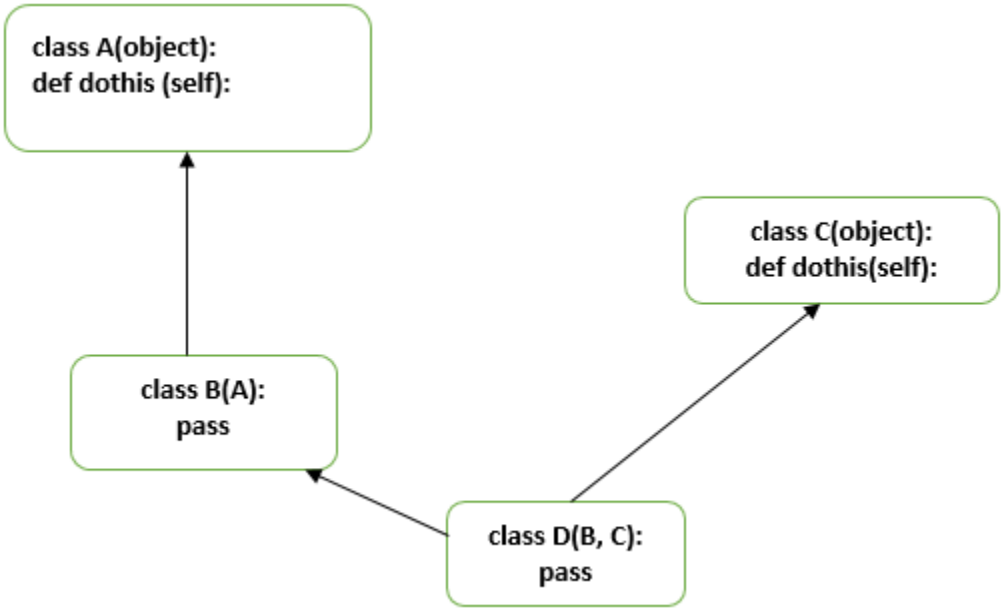
Multiple inheritance refers to the ability of inheriting from two or more than two class. The complexity arises as child inherits from parent and parents inherits from the grandparent class. Python climbs an inheriting tree looking for attributes that is being requested to be read from an object. It will check the in the instance, within class then

parent class and lastly from the grandparent class. Now the question arises in what order the classes will be searched - breath-first or depth-first. By default, Python goes with the depth-first.

That's is why in the below diagram the Python searches the dothis() method first in class A. So the method resolution order in the below example will be

Mro- D->B->A->C

Look at the below multiple inheritance diagram:



Let’s go through an example to understand the “mro” feature of an Python.

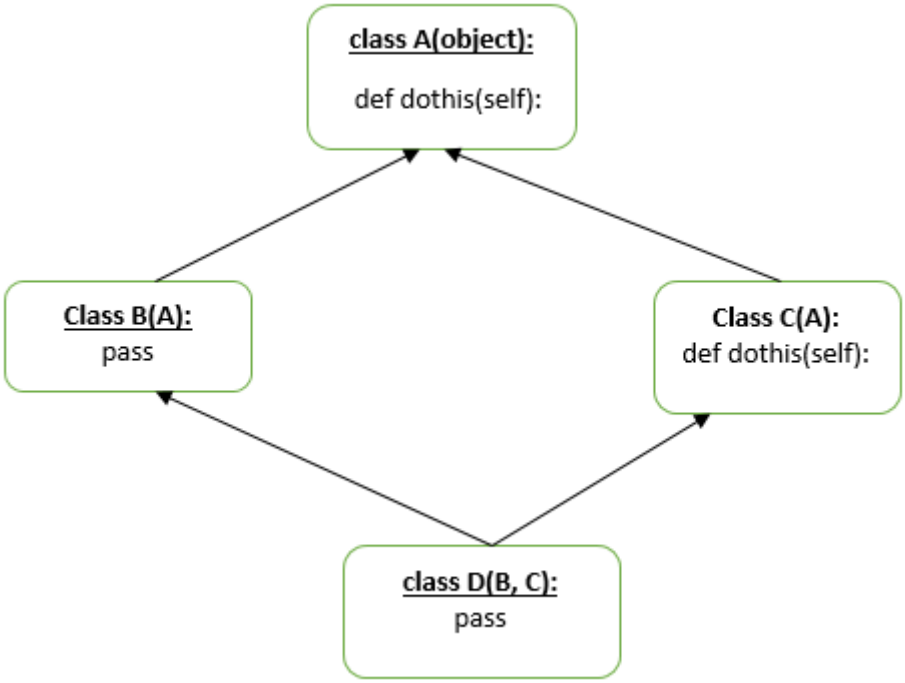
```
class A(object):
    def dothis(self):
        print('doing this in A')

class B(A):
    pass

class C(object):
    def dothis(self):
        print('do this in C')

class D(B,C):
    pass

d_instance = D()

d_instance.dothis()

print(D.mro())
```

Output

```
doing this in A
[<class '__main__.D'>, <class '__main__.B'>, <class '__main__.A'>, <class '__main__.C'>, <class 'object'>]
```

Example 3

Let's take another example of "diamond shape" multiple inheritance.



Above diagram will be considered ambiguous. From our previous example understanding "method resolution order" .i.e. mro will be D->B->A->C->A but it's not. On getting the second A from the C, Python will ignore the previous A. so the mro will be in this case will be D->B->C->A.

Let's create an example based on above diagram:

```
class A(object):
    def dothis(self):
        print('doing this in A')

class B(A):
    pass

class C(A):
    def dothis(self):
        print('do this in C')

class D(B,C):
    pass

d_instance = D()
d_instance.dothis()

print(D.mro())
```

Output

```
do this in C
[<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A'>, <class 'object'>]
```

Simple rule to understand the above output is- if the same class appear in the method resolution order, the earlier appearances of this class will be remove from the method resolution order.

In conclusion:

- Any class can inherit from multiple classes
- Python normally uses a "depth-first" order when searching inheriting classes.
- But when two classes inherit from the same class, Python eliminates the first appearances of that class from the mro.

Decorators, Static and Class Methods

Functions(or methods) are created by def statement.

Though methods works in exactly the same way as a function except one point where method first argument is instance object.

We can classify methods based on how they behave, like

- **Simple method**: defined outside of a class. This function can access class attributes by feeding instance argument:

```
def outside_func(():
```

- **Instance method**:

```
def func(self,)
```

- **Class method**: if we need to use class attributes

```
                @classmethod
          def cfunc(cls,)
```

- **Static method**: do not have any info about the class

```
          @staticmethod
          def sfoo()
```

Till now we have seen the instance method, now is the time to get some insight into the other two methods,

Class Method

The @classmethod decorator, is a builtin function decorator that gets passed the class it was called on or the class of the instance it was called on as first argument. The result of that evaluation shadows your function definition.

Syntax

```
class C(object):
    @classmethod
    def fun(cls, arg1, arg2, ...):
        ....
fun: function that needs to be converted into a class method
returns: a class method for function
```

They have the access to this cls argument, it can't modify object instance state. That would require access to self.

- It is bound to the class and not the object of the class.
- Class methods can still modify class state that applies across all instances of the class.

Static Method

A static method takes neither a self nor a cls(class) parameter but it's free to accept an arbitrary number of other parameters.

Syntax

```
class C(object):
    @staticmethod
    def fun(arg1, arg2, ...):
        ...

returns: a static method for function funself.
```

- A static method can neither modify object state nor class state.
- They are restricted in what data they can access.

When to use what

- We generally use class method to create factory methods. Factory methods return class object (similar to a constructor) for different use cases.
- We generally use static methods to create utility functions.

7. OOP in Python –Python Design Pattern

Overview

Modern software development needs to address complex business requirements. It also needs to take into account factors such as future extensibility and maintainability. A good design of a software system is vital to accomplish these goals. Design patterns play an important role in such systems.
To understand design pattern, let's consider below example-

- Every car's design follows a basic design pattern, four wheels, steering wheel, the core drive system like accelerator-break-clutch, etc.

So, all things repeatedly built/ produced, shall inevitably follow a pattern in its design.. it cars, bicycle, pizza, atm machines, whatever…even your sofa bed.

Designs that have almost become standard way of coding some logic/mechanism/technique in software, hence come to be known as or studied as, Software Design Patterns.

Why is Design Pattern Important?

Benefits of using Design Patterns are:

- Helps you to solve common design problems through a proven approach
- No ambiguity in the understanding as they are well documented.
- Reduce the overall development time.
- Helps you deal with future extensions and modifications with more ease than otherwise.
- May reduce errors in the system since they are proven solutions to common problems.

Classification of Design Patterns

The GoF (Gang of Four) design patterns are classified into three categories namely creational, structural and behavioral.

Creational Patterns

Creational design patterns separate the object creation logic from the rest of the system. Instead of you creating objects, creational patterns creates them for you. The creational patterns include Abstract Factory, Builder, Factory Method, Prototype and Singleton.

Creational Patterns are not commonly used in Python because of the dynamic nature of the language. Also language itself provide us with all the flexibility we need to create in a sufficient elegant fashion, we rarely need to implement anything on top, like singleton or Factory.

Also these patterns provide a way to create objects while hiding the creation logic, rather than instantiating objects directly using a new operator.

Structural Patterns

Sometimes instead of starting from scratch, you need to build larger structures by using an existing set of classes. That's where structural class patterns use inheritance to build a new structure. Structural object patterns use composition/ aggregation to obtain a new functionality. Adapter, Bridge, Composite, Decorator, Façade, Flyweight and Proxy are Structural Patterns. They offers best ways to organize class hierarchy.

Behavioral Patterns

Behavioral patterns offers best ways of handling communication between objects. Patterns comes under this categories are: Visitor, Chain of responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy and Template method are Behavioral Patterns.
Because they represent the behavior of a system, they are used generally to describe the functionality of software systems.

Commonly used Design Patterns

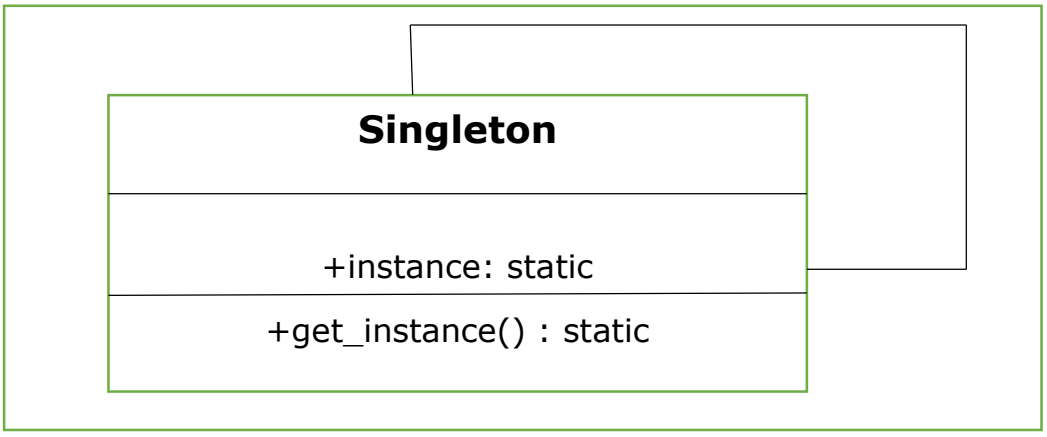
Singleton

It is one of the most controversial and famous of all design patterns. It is used in overly object-oriented languages, and is a vital part of traditional object-oriented programming.

The Singleton pattern is used for,

- When logging needs to be implemented. The logger instance is shared by all the components of the system.
- The configuration files is using this because cache of information needs to be maintained and shared by all the various components in the system.
- Managing a connection to a database.

Here is the UML diagram,




```
class Logger(object):
    def __new__(cls, *args, **kwargs):
        if not hasattr(cls, '_logger'):
```

```
            cls._logger = super(Logger, cls).__new__(cls, *args, **kwargs)
        return cls._logger
```

In this example, Logger is a Singleton.

When __new__ is called, it normally constructs a new instance of that class. When we override it, we first check if our singleton instance has been created or not. If not, we create it using a super call. Thus, whenever we call the constructor on Logger, we always get the exact same instance.

```
>>>
>>> obj1 = Logger()
>>> obj2 = Logger()
>>> obj1 == obj2
True
>>>
>>> obj1
<__main__.Logger object at 0x03224090>
>>> obj2
<__main__.Logger object at 0x03224090>
```

8. OOP in Python – Advanced Features

In this we will look into some of the advanced features which Python provide

Core Syntax in our Class design

In this we will look onto, how Python allows us to take advantage of operators in our classes. Python is largely objects and methods call on objects and this even goes on even when its hidden by some convenient syntax.

```
>>> var1 = 'Hello'
>>> var2 = ' World!'
>>> var1 + var2
'Hello World!'
>>>
>>> var1.__add__(var2)
'Hello World!'
>>> num1 = 45
>>> num2 = 60
>>> num1.__add__(num2)
105
>>> var3 = ['a', 'b']
>>> var4 = ['hello', ' John']
>>> var3.__add__(var4)
['a', 'b', 'hello', ' John']
```

So if we have to add magic method __add__ to our own classes, could we do that too. Let's try to do that.

We have a class called Sumlist which has a contructor __init__ which takes list as an argument called my_list.

```
class SumList(object):
    def __init__(self, my_list):
        self.mylist = my_list
    def __add__(self, other):
```

```
        new_list = [ x + y for x, y in zip(self.mylist, other.mylist)]

        return SumList(new_list)

    def __repr__(self):
        return str(self.mylist)

aa = SumList([3,6, 9, 12, 15])

bb = SumList([100, 200, 300, 400, 500])
cc = aa + bb        # aa.__add__(bb)
print(cc)           # should gives us a list ([103, 206, 309, 412, 515])
```

Output

```
[103, 206, 309, 412, 515]
```

But there are many methods which are internally managed by others magic methods. Below are some of them,

```
'abc' in var    # var.__contains__('abc')

var == 'abc'    # var.__eq__('abc')

var[1]          # var.__getitem__(1)

var[1:3]        # var.__getslice__(1, 3)

len(var)        # var.__len__()

print(var)      # var.__repr__()
```

Inheriting From built-in types

Classes can also inherit from built-in types this means inherits from any built-in and take advantage of all the functionality found there.

In below example we are inheriting from dictionary but then we are implementing one of its method __setitem__. This (setitem) is invoked when we set key and value in the dictionary. As this is a magic method, this will be called implicitly.

```
class MyDict(dict):

    def __setitem__(self, key, val):
        print('setting a key and value!')
        dict.__setitem__(self, key, val)

dd = MyDict()
dd['a'] = 10
dd['b'] = 20

for key in dd.keys():
    print('{0}={1}'.format(key, dd[key]))
```

Output:

```
setting a key and value!
setting a key and value!
a=10
b=20
```

Let's extend our previous example, below we have called two magic methods called __getitem__ and __setitem__ better invoked when we deal with list index.

```
# Mylist inherits from 'list' object but indexes from 1 instead for 0!
class Mylist(list):         # inherits from list
    def __getitem__(self, index):
        if index == 0:
            raise IndexError
        if index > 0:
            index = index - 1
            return list.__getitem__(self, index)    # this method is called
when

# we access a value with subscript like x[1]
    def __setitem__(self, index, value):
        if index == 0:
            raise IndexError
        if index > 0:
                        index = index - 1
```

```
            list.__setitem__(self, index, value)

x = Mylist(['a', 'b', 'c'])     # __init__() inherited from builtin list

print(x)            # __repr__() inherited from builtin list

x.append('HELLO');  # append() inherited from builtin list

print(x[1])         # 'a' (Mylist.__getitem__ cutomizes list superclass
                          # method. index is 1, but reflects 0!

print (x[4])        # 'HELLO' (index is 4 but reflects 3!
```

Output

```
['a', 'b', 'c']
a
HELLO
```

In above example, we set a three item list in Mylist and implicitly __init__ method is called and when we print the element x, we get the three item list ([‘a’,’b’,’c’]). Then we append another element to this list. Later we ask for index 1 and index 4. But if you see the output, we are getting element from the (index-1) what we have asked for. As we know list indexing start from 0 but here the indexing start from 1 (that’s why we are getting the first item of the list).

Naming Conventions

In this we will look into names we’ll used for variables especially private variables and conventions used by Python programmers worldwide. Although variables are designated as private but there is not privacy in Python and this by design. Like any other well documented languages, Python has naming and style conventions that it promote although it doesn’t enforce them. There is a style guide written by **“Guido van Rossum” the originator of Python, that describe the best practices and use of name and is called PEP8. Here is the link for this,**

https://www.Python.org/dev/peps/pep-0008/

PEP stands for Python enhancement proposal and is a series of documentation that distributed among the Python community to discuss proposed changes. For example it is recommended all,

- Module names : all_lower_case
- Class names and exception names: CamelCase
- Global and local names: all_lower_case
- Functions and method names: all_lower_case
- Constants: ALL_UPPER_CASE

These are just the recommendation, you can vary if you like. But as most of the developers follows these recommendation so might me your code is less readable.

Why conform to convention?

We can follow the PEP recommendation we it allows us to get,

- More familiar to the vast majority of developers
- Clearer to most readers of your code.
- Will match style of other contributers who work on same code base.
- Mark of a professional software developers
- Everyone will accept you.

Variable Naming: 'Public' and 'Private'

In Python, when we are dealing with modules and classes, we designate some variables or attribute as private. In Python, there is no existence of "Private" instance variable which cannot be accessed except inside an object. Private simply means they are simply not intended to be used by the users of the code instead they are intended to be used internally. In general, a convention is being followed by most Python developers i.e. a name prefixed with an underscore for example. _attrval (example below) should be treated as a non-public part of the API or any Python code, whether it is a function, a method or a data member. Below is the naming convention we follow,

- Public attributes or variables (intended to be used by the importer of this module or user of this class): **regular_lower_case**
- Private attributes or variables (internal use by the module or class): **_single_leading_underscore**
- Private attributes that shouldn't be subclassed: **__double_leading_underscore**
- Magic attributes: **__double_underscores__** (use them, don't create them)

```
class GetSet(object):

    instance_count = 0      # public

    __mangled_name = 'no privacy!'  # special variable
```

```
    def __init__(self, value):
        self._attrval = value       # _attrval is for internal use only
        GetSet.instance_count += 1


    @property
    def var(self):
        print('Getting the "var" attribute')
        return self._attrval


    @var.setter
    def var(self, value):
        print('setting the "var" attribute')
        self._attrval = value


    @var.deleter
    def var(self):
        print('deleting the "var" attribute')
        self._attrval = None

cc = GetSet(5)
cc.var = 10     # public name
print(cc._attrval)
print(cc._GetSet__mangled_name)
```

Output

```
setting the "var" attribute
10
no privacy!
```

Strings

Strings are the most popular data types used in every programming language. Why? Because we, understand text better than numbers, so in writing and talking we use text and words, similarly in programming too we use strings. In string we parse text, analyse text semantics, and do data mining – and all this data is human consumed text.
The string in Python is immutable.

String Manipulation

In Python, string can be marked in multiple ways, using single quote (`), double quote(") or even triple quote ('") in case of multiline strings.

```
>>> # String Examples
>>> a = "hello"
>>> b = ''' A Multi line string,
Simple!'''
>>> e = ('Multiple' 'strings' 'togethers')
```

String manipulation is very useful and very widely used in every language. Often, programmers are required to break down strings and examine them closely.

Strings can be iterated over (character by character), sliced, or concatenated. The syntax is the same as for lists.

The str class has numerous methods on it to make manipulating strings easier. The dir and help commands provides guidance in the Python interpreter how to use them.

Below are some of the commonly used string methods we use.

METHODS	DESCRIPTION
isalpha()	Checks if all characters are Alphabets
isdigit()	Checks Digit Characters
isdecimal()	Checks decimal Characters
isnumeric()	checks Numeric Characters
find()	Returns the Highest Index of substrings.
istitle()	Checks for Titlecased strings
join()	Returns a concatenated string.
lower()	returns lower cased string
upper()	returns upper cased string
partion()	Returns a tuple
bytearray()	Returns array of given byte size.

enumerate()	Returns an enumerate object.
isprintable()	Checks printable character.

Let's try to run couple of string methods,

```
>>> str1 = 'Hello World!'
>>> str1.startswith('h')
False
>>> str1.startswith('H')
True
>>> str1.endswith('d')
False
>>> str1.endswith('d!')
True
>>> str1.find('o')
4
>>> #Above returns the index of the first occurence of the character/substring.
>>> str1.find('lo')
3
>>> str1.upper()
'HELLO WORLD!'
>>> str1.lower()
'hello world!'
>>> str1.index('b')
Traceback (most recent call last):
  File "<pyshell#19>", line 1, in <module>
    str1.index('b')
ValueError: substring not found
>>> s = ('hello How Are You')
>>> s.split(' ')
['hello', 'How', 'Are', 'You']
>>> s1=s.split(' ')
>>> '*'.join(s1)
'hello*How*Are*You'
>>> s.partition(' ')
         ('hello', ' ', 'How Are You')
```

```
>>>
```

String Formatting

In Python 3.x formatting of strings has changed, now it more logical and is more flexible. Formatting can be done using the format() method or the % sign(old style) in format string.

The string can contain literal text or replacement fields delimited by braces {} and each replacement field may contains either the numeric index of a positional argument or the name of a keyword argument.

Syntax

```
str.format(*args, **kwargs)
```

Basic Formatting

```
>>> '{} {}'.format('Example', 'One')
'Example One'
>>> '{} {}'.format('pie', '3.1415926')
'pie 3.1415926'
```

Below example allows re-arrange the order of display without changing the arguments.

```
>>> '{1} {0}'.format('pie', '3.1415926')
'3.1415926 pie'
```

Padding and aligning strings

A value can be padded to a specific length.

```
>>> #Padding Character, can be space or special character
>>> '{:12}'.format('PYTHON')
'PYTHON      '
>>> '{:>12}'.format('PYTHON')
'      PYTHON'
>>> '{:<{}s}'.format('PYTHON',12)
'PYTHON      '
>>> '{:*<12}'.format('PYTHON')
'PYTHON******'
```

```
>>> '{:*^12}'.format('PYTHON')
'***PYTHON***'
>>> '{:.15}'.format('PYTHON OBJECT ORIENTED PROGRAMMING')
'PYTHON OBJECT O'
>>> #Above, truncated 15 characters from the left side of a specified string
>>> '{:.{}}'.format('PYTHON OBJECT ORIENTED',15)
'PYTHON OBJECT O'
>>> #Named Placeholders
>>> data = {'Name':'Raghu', 'Place':'Bangalore'}
>>> '{Name} {Place}'.format(**data)
'Raghu Bangalore'
>>> #Datetime
>>> from datetime import datetime
>>> '{:%Y/%m/%d.%H:%M}'.format(datetime(2018,3,26,9,57))
'2018/03/26.09:57'
```

Strings are Unicode

Strings as collections of immutable Unicode characters. Unicode strings provide an opportunity to create software or programs that works everywhere because the Unicode strings can represent any possible character not just the ASCII characters.

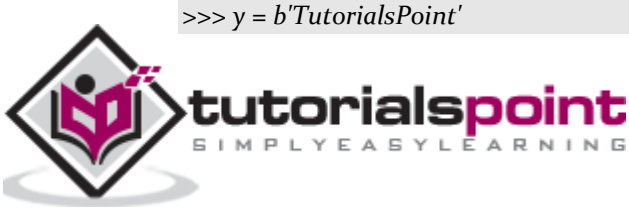
Many IO operations only know how to deal with bytes, even if the bytes object refers to textual data. It is therefore very important to know how to interchange between bytes and Unicode.

Converting text to bytes

Converting a strings to byte object is termed as encoding. There are numerous forms of encoding, most common ones are: PNG; JPEG, MP3, WAV, ASCII, UTF-8 etc. Also this(encoding) is a format to represent audio, images, text, etc. in bytes.

This conversion is possible through encode(). It take encoding technique as argument. By default, we use 'UTF-8' technique.

```
>>> # Python Code to demonstrate string encoding
>>>
>>> # Initialising a String
>>> x = 'TutorialsPoint'
>>>
>>> #Initialising a byte object
>>> y = b'TutorialsPoint'
```



```
>>>
>>> # Using encode() to encode the String
>>> # encoded version of x is stored in z using ASCII mapping
>>> z = x.encode('ASCII')
>>>
>>> # Check if x is converted to bytes or not
>>>
>>> if(z==y):
        print('Encoding Successful!')
else:
        print('Encoding Unsuccessful!')


Encoding Successful!
```

Converting bytes to text

Converting bytes to text is called the decoding. This is implemented through decode(). We can convert a byte string to a character string if we know which encoding is used to encode it.

So Encoding and decoding are inverse processes.

```
>>>
>>> # Python code to demonstrate Byte Decoding
>>>
>>> #Initialise a String
>>> x = 'TutorialsPoint'
>>>
>>> #Initialising a byte object
>>> y = b'TutorialsPoint'
>>>
>>> #using decode() to decode the Byte object
>>> # decoded version of y is stored in z using ASCII mapping
>>> z = y.decode('ASCII')
>>>
```

```
>>> #Check if y is converted to String or not
>>> if (z == x):
        print('Decoding Successful!')
else:
        print('Decoding Unsuccessful!')


Decoding Successful!
>>>
```

File I/O

Operating systems represents files as a sequence of bytes, not text.

A file is a named location on disk to store related information. It is used to permanently store data in your disk.

In Python, a file operation takes place in the following order.

- Open a file
- Read or write onto a file (operation).
- Close the file.

Python wraps the incoming (or outgoing) stream of bytes with appropriate decode (or encode) calls so we can deal directly with str objects.

Opening a file

Python has a built-in function open() to open a file. This will generate a file object, also called a handle as it is used to read or modify the file accordingly.

```
>>> f = open(r'c:\users\rajesh\Desktop\index.webm','rb')
>>> f
<_io.BufferedReader name='c:\\users\\rajesh\\Desktop\\index.webm'>
>>> f.mode
'rb'
>>> f.name
'c:\\users\\rajesh\\Desktop\\index.webm'
```

For reading text from a file, we only need to pass the filename into the function. The file will be opened for reading, and the bytes will be converted to text using the platform default encoding.

10. OOP in Python – Exception and Exception Classes

In general, an exception is any unusual condition. Exception usually indicates errors but sometimes they intentionally puts in the program, in cases like terminating a procedure early or recovering from a resource shortage. There are number of built-in exceptions, which indicate conditions like reading past the end of a file, or dividing by zero. We can define our own exceptions called custom exception.

Exception handling enables you handle errors gracefully and do something meaningful about it. Exception handling has two components: "throwing" and 'catching'.

Identifying Exception (Errors)

Every error occurs in Python result an exception which will an error condition identified by its error type.

```
>>> #Exception
>>> 1/0
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    1/0
ZeroDivisionError: division by zero
>>>
>>> var=20
>>> print(ver)
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    print(ver)
NameError: name 'ver' is not defined
>>> #Above as we have misspelled a variable name so we get an NameError.
>>>
>>> print('hello)

SyntaxError: EOL while scanning string literal
>>> #Above we have not closed the quote in a string, so we get SyntaxError.
>>>
>>> #Below we are asking for a key, that doen't exists.
```

```
>>> mydict = {}
>>> mydict['x']
Traceback (most recent call last):
  File "<pyshell#15>", line 1, in <module>
    mydict['x']
KeyError: 'x'
>>> #Above keyError
>>>
>>> #Below asking for a index that didn't exist in a list.
>>> mylist = [1,2,3,4]
>>> mylist[5]
Traceback (most recent call last):
  File "<pyshell#20>", line 1, in <module>
    mylist[5]
IndexError: list index out of range
>>> #Above, index out of range, raised IndexError.
```

Catching/Trapping Exception

When something unusual occurs in your program and you wish to handle it using the exception mechanism, you 'throw an exception'. The keywords try and except are used to catch exceptions. Whenever an error occurs within a try block, Python looks for a matching except block to handle it. If there is one, execution jumps there.

Syntax:

```
try:
    #write some code
    #that might throw some exception
except <ExceptionType>:
    # Exception handler, alert the user
```

The code within the try clause will be executed statement by statement.

If an exception occurs, the rest of the try block will be skipped and the except clause will be executed.

```
try:
     some statement here
except:
           exception handling
```

Let's write some code to see what happens when you not use any error handling mechanism in your program.

```
number = int(input('Please enter the number between 1 & 10: '))
print('You have entered number',number)
```

Above programme will work correctly as long as the user enters a number, but what happens if the users try to puts some other data type(like a string or a list).

```
Please enter the number between 1 & 10: 'Hi'
Traceback (most recent call last):
  File "C:/Python/Python361/exception2.py", line 1, in <module>
    number = int(input('Please enter the number between 1 & 10: '))
ValueError: invalid literal for int() with base 10: "'Hi'"
```

Now ValueError is an exception type. Let's try to rewrite the above code with exception handling.

```
import sys

print('Previous code with exception handling')

try:
    number = int(input('Enter number between 1 & 10: '))

except(ValueError):
    print('Error..numbers only')
    sys.exit()

print('You have entered number: ',number)
```

If we run the program, and enter a string (instead of a number), we can see that we get a different result.

```
Previous code with exception handling
Enter number between 1 & 10: 'Hi'
Error..numbers only
```

Raising Exceptions

To raise your exceptions from your own methods you need to use raise keyword like this

```
raise ExceptionClass('Some Text Here')
```

Let's take an example

```
def enterAge(age):
    if age<0:
        raise ValueError('Only positive integers are allowed')
    if age % 2 ==0:
        print('Entered Age is even')
    else:
        print('Entered Age is odd')
try:
    num = int(input('Enter your age: '))
    enterAge(num)
except ValueError:
    print('Only positive integers are allowed')
```

Run the program and enter positive integer.

Expected Output

```
Enter your age: 12
Entered Age is even
```

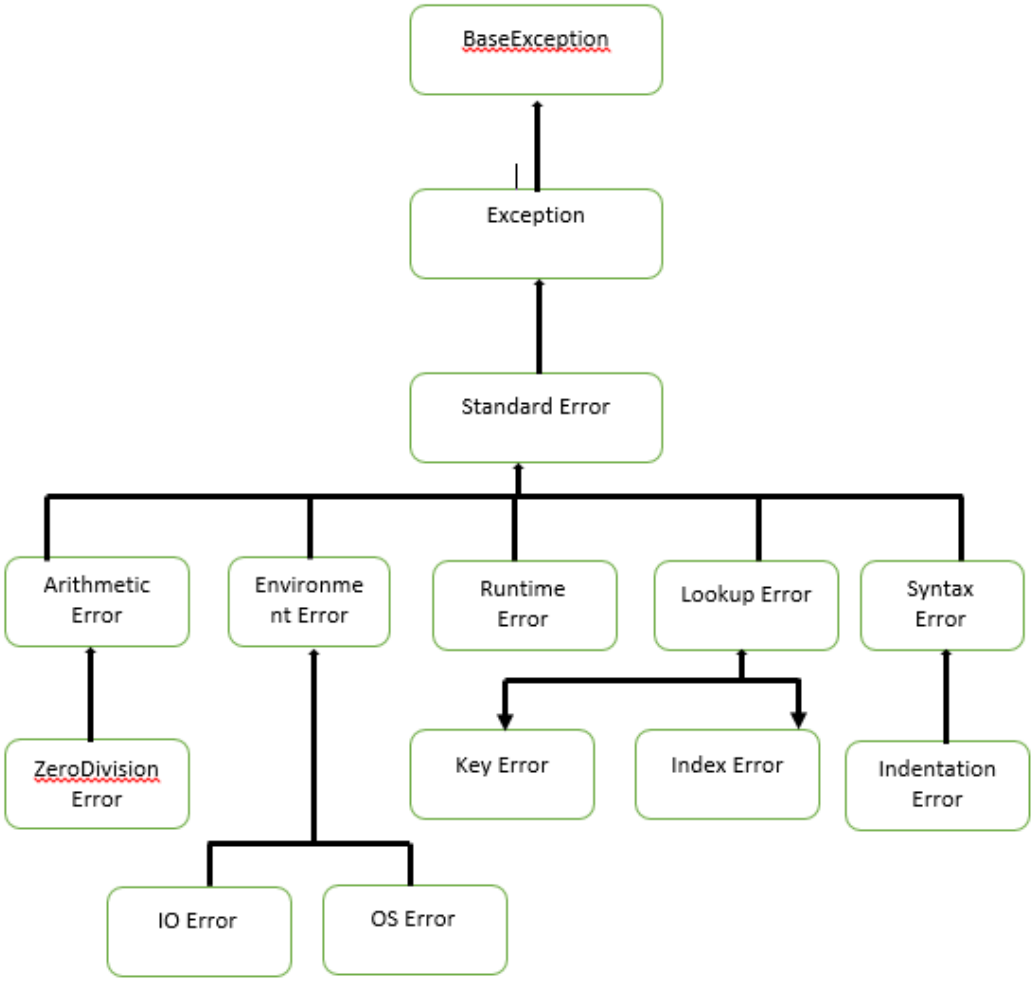
But when we try to enter a negative number we get,

Expected Output

```
Enter your age: -2
Only positive integers are allowed
```

Creating Custom exception class

You can create a custom exception class by Extending BaseException class or subclass of BaseException.



From above diagram we can see most of the exception classes in Python extends from the BaseException class. You can derive your own exception class from BaseException class or from its subclass.

Create a new file called NegativeNumberException.py and write the following code.

```
class NegativeNumberException(RuntimeError):
    def __init__(self, age):
        super().__init__()
        self.age = age
```

Above code creates a new exception class named NegativeNumberException, which consists of only constructor which call parent class constructor using super()__init__() and sets the age.

Now to create your own custom exception class, will write some code and import the new exception class.

```
from NegativeNumberException import NegativeNumberException
def enterage(age):
    if age < 0:
        raise NegativeNumberException('Only positive integers are allowed')

    if  age % 2 == 0:
        print('Age is Even')

    else:
        print('Age is Odd')

try:
    num = int(input('Enter your age: '))
    enterage(num)
except NegativeNumberException:
    print('Only positive integers are allowed')
except:
    print('Something is wrong')
```

Output:

```
Enter your age: -2
Only positive integers are allowed
```

Another way to create a custom Exception class.

```
class customException(Exception):
    def __init__(self, value):
        self.parameter = value

    def __str__(self):
        return repr(self.parameter)
try:
    raise customException('My Useful Error Message!')
except customException as instance:
            print('Caught: ' + instance.parameter)
```

Output

```
Caught: My Useful Error Message!
```

Exception hierarchy

The class hierarchy for built-in exceptions is:

```
BaseException
+-- SystemExit
+-- KeyboardInterrupt
+-- GeneratorExit
+-- Exception
+-- StopIteration
+-- StopAsyncIteration
+-- ArithmeticError
| +-- FloatingPointError
| +-- OverflowError
| +-- ZeroDivisionError
+-- AssertionError
+-- AttributeError
+-- BufferError
+-- EOFError
+-- ImportError
+-- LookupError
| +-- IndexError
| +-- KeyError
+-- MemoryError
+-- NameError
| +-- UnboundLocalError
+-- OSError
| +-- BlockingIOError
| +-- ChildProcessError
| +-- ConnectionError
| | +-- BrokenPipeError
| | +-- ConnectionAbortedError
| | +-- ConnectionRefusedError
| | +-- ConnectionResetError
| +-- FileExistsError
| +-- FileNotFoundError
| +-- InterruptedError
| +-- IsADirectoryError
| +-- NotADirectoryError
| +-- PermissionError
| +-- ProcessLookupError
| +-- TimeoutError
+-- ReferenceError
+-- RuntimeError
| +-- NotImplementedError
| +-- RecursionError
+-- SyntaxError
                | +-- IndentationError
```

```
| +-- TabError
+-- SystemError
+-- TypeError
+-- ValueError
| +-- UnicodeError
| +-- UnicodeDecodeError
| +-- UnicodeEncodeError
| +-- UnicodeTranslateError
+-- Warning
+-- DeprecationWarning
+-- PendingDeprecationWarning
+-- RuntimeWarning
+-- SyntaxWarning
+-- UserWarning
+-- FutureWarning
+-- ImportWarning
+-- UnicodeWarning
+-- BytesWarning
+-- ResourceWarning
```

11. OOP in Python – Object Serialization

In the context of data storage, serialization is the process of translating data structures or object state into a format that can be stored (for example, in a file or memory buffer) or transmitted and reconstructed later.

In serialization, an object is transformed into a format that can be stored, so as to be able to deserialize it later and recreate the original object from the serialized format.

Pickle

Pickling is the process whereby a Python object hierarchy is converted into a byte stream (usually not human readable) to be written to a file, this is also known as Serialization. Unpickling is the reverse operation, whereby a byte stream is converted back into a working Python object hierarchy.

Pickle is operationally simplest way to store the object. The Python Pickle module is an object-oriented way to store objects directly in a special storage format.

What can it do?

- Pickle can store and reproduce dictionaries and lists very easily.
- Stores object attributes and restores them back to the same State.

What pickle can't do?

- It does not save an objects code. Only it's attributes values.
- It cannot store file handles or connection sockets.

In short we can say, pickling is a way to store and retrieve data variables into and out from files where variables can be lists, classes, etc.

To Pickle something you must:

- import pickle
- Write a variable to file, something like

```
pickle.dump(mystring, outfile, protocol),
```

where 3rd argument protocol is optional

To unpickling something you must:

Import pickle

Write a variable to a file, something like

```
myString = pickle.load(inputfile)
```

Methods

The pickle interface provides four different methods.

- dump() : The dump() method serializes to an open file (file-like object).
- dumps(): Serializes to a string
- load(): Deserializes from an open-like object.
- loads() : Deserializes from a string.

Based on above procedure, below is an example of "pickling".

```
import pickle
class Animal:
   def __init__(self, number_of_legs, color):
       self.number_of_legs = number_of_legs
       self.color = color
class Cat(Animal):
   def __init__(self, color):
       Animal.__init__(self, 4, color)

pussy = Cat("White")
print (str.format("My Cat pussy is {0} and has {1} legs", pussy.color, pussy.number_of_legs))
pickled_pussy = pickle.dumps(pussy)
print ("Would you like to see her pickled? Here she is!")
print (pickled_pussy)
```

Output

```
My Cat pussy is White and has 4 legs

Would you like to see her pickled? Here she is!

b'\x80\x03c__main__\nCat\nq\x00)\x81q\x01}q\x02(X\x0e\x00\x00\x00number_of_legs
q\x03K\x04X\x05\x00\x00\x00colorq\x04X\x05\x00\x00\x00Whiteq\x05ub.'
```

So, in the example above, we have created an instance of a Cat class and then we've pickled it, transforming our "Cat" instance into a simple array of bytes.

This way we can easily store the bytes array on a binary file or in a database field and restore it back to its original form from our storage support in a later time.

Also if you want to create a file with a pickled object, you can use the dump() method (instead of the dumps*()* one) passing also an opened binary file and the pickling result will be stored in the file automatically.

```
[….]

binary_file = open(my_pickled_Pussy.bin', mode='wb')

my_pickled_Pussy = pickle.dump(Pussy, binary_file)

binary_file.close()
```

Unpickling

The process that takes a binary array and converts it to an object hierarchy is called unpickling.

The unpickling process is done by using the load() function of the pickle module and returns a complete object hierarchy from a simple bytes array.

Let's use the load function in our previous example.

```
import pickle
class Animal:
   def __init__(self, number_of_legs, color):
       self.number_of_legs = number_of_legs
       self.color = color
class Cat(Animal):
   def __init__(self, color):
       Animal.__init__(self, 4, color)

# Step 1: Let's create the Cat Pussy
Pussy = Cat("white")
# Step 2: Let's pickle Pussy
my_pickled_Pussy = pickle.dumps(Pussy)
# Step 3: Now, let's unpickle our Cat Pussy creating another instance, another Cat... MeOw!
MeOw = pickle.loads(my_pickled_Pussy)
# MeOw and Pussy are two different objects, in fact if we specify another color for MeOw
# there are no conseguencies for Pussy
MeOw.color = "black"
print (str.format("MeOw is {0} ", MeOw.color))
print (str.format("Pussy is {0} ", Pussy.color))
```

Output

```
MeOw is black

Pussy is white
```

JSON

JSON(JavaScript Object Notation) has been part of the Python standard library is a lightweight data-interchange format. It is easy for humans to read and write. It is easy to parse and generate.

Because of its simplicity, JSON is a way by which we store and exchange data, which is accomplished through its JSON syntax, and is used in many web applications. As it is in human readable format, and this may be one of the reasons for using it in data transmission, in addition to its effectiveness when working with APIs.

An example of JSON-formatted data is as follow:

```
{"EmployID": 40203, "Name": "Zack", "Age":54, "isEmployed": True}
```

Python makes it simple to work with Json files. The module sused for this purpose is the JSON module. This module should be included (built-in) within your Python installation.

So let's see how can we convert Python dictionary to JSON and write it to a text file.

JSON to Python

Reading JSON means converting JSON into a Python value (object). The json library parses JSON into a dictionary or list in Python. In order to do that, we use the loads() function (load from a string), as follow:

```
import json

jsonData = '{"EmployId":402040, "EmployeName":"Zack", "Department":"Financial Services"}'

jsonToPython = json.loads(jsonData)

print(jsonToPython)
```

Output

```
{'EmployId': 402040, 'EmployeName': 'Zack', 'Department': 'Financial Services'}
```

Below is one sample json file,

data1.json

```
{"menu": {
  "id": "file",
  "value": "File",
  "popup": {
    "menuitem": [
      {"value": "New", "onclick": "CreateNewDoc()"},
      {"value": "Open", "onclick": "OpenDoc()"},
      {"value": "Close", "onclick": "CloseDoc()"}
    ]
  }
}}
```

Above content (Data1.json) looks like a conventional dictionary. We can use pickle to store this file but the output of it is not human readable form.

JSON(Java Script Object Notification) is a very simple format and that's one of the reason for its popularity. Now let's look into json output through below program.

```
import json

with open('data1.json') as fh:
    conf = json.load(fh)

print(conf)
print("++++++++++++++++++")
print(type(conf))
print("+++++++++++++++")
conf['neykey'] = 9.04050


with open('data1.json', 'w') as fh:
    json.dump(conf, fh)


print(conf)
```

Output

```
{'menu': {'id': 'file', 'value': 'File', 'popup': {'menuitem': [{'value': 'New',
 'onclick': 'CreateNewDoc()'}, {'value': 'Open', 'onclick': 'OpenDoc()'}, {'valu
e': 'Close', 'onclick': 'CloseDoc()'}]}}
++++++++++++++++++
<class 'dict'>
++++++++++++++
{'menu': {'id': 'file', 'value': 'File', 'popup': {'menuitem': [{'value': 'New',
 'onclick': 'CreateNewDoc()'}, {'value': 'Open', 'onclick': 'OpenDoc()'}, {'valu
e': 'Close', 'onclick': 'CloseDoc()'}]}}, 'neykey': 9.0405}
>>>
```

Above we open the json file (data1.json) for reading, obtain the file handler and pass on to json.load and getting back the object. When we try to print the output of the object, its same as the json file. Although the type of the object is dictionary, it comes out as a Python object. Writing to the json is simple as we saw this pickle. Above we load the json file, add another key value pair and writing it back to the same json file. Now if we see out data1.json, it looks different .i.e. not in the same format as we see previously.

To make our output looks same (human readable format), add the couple of arguments into our last line of the program,

```
json.dump(conf, fh, indent=4, separators = (',',  ': '))
```

Similarly like pickle, we can print the string with dumps and load with loads. Below is an example of that,

```
>>> import json
>>>
>>> x = json.dumps({'x1': [ 2, 4, 6], 'y1':[3, 6, 9], 'z1': [4, 8, 12]})
>>>
>>> print(x)
{"x1": [2, 4, 6], "y1": [3, 6, 9], "z1": [4, 8, 12]}
>>>
>>> mystruct = json.loads(x)
>>> for key in mystruct:
        print(key)
        for x in mystruct[key]:
                print ("  {0}".format(x))


x1
  2
  4
  6
y1
  3
  6
  9
z1
  4
  8
  12
```

YAML

YAML may be the most human friendly data serialization standard for all programming languages.

Python yaml module is called pyaml

YAML is an alternative to JSON:

- Human readable code: YAML is the most human readable format so much so that even its front-page content is displayed in YAML to make this point.
- Compact code: In YAML we use whitespace indentation to denote structure not brackets.
- Syntax for relational data: For internal references we use anchors (&) and aliases (*)
- One of the area where it is used widely is for viewing/editing of data structures: for example configuration files, dumping during debugging and document headers.

Installing YAML

As yaml is not a built-in module, we need to install it manually. Best way to install yaml on windows machine is through pip. Run below command on your windows terminal to install yaml,

```
pip install pyaml (Windows machine)

sudo pip install pyaml (*nix and Mac)
```

On running above command, screen will display something like below based on what's the current latest version.

```
Collecting pyaml
```

```
Using cached pyaml-17.12.1-py2.py3-none-any.whl
Collecting PyYAML (from pyaml)
Using cached PyYAML-3.12.tar.gz
Installing collected packages: PyYAML, pyaml
Running setup.py install for PyYAML ... done
Successfully installed PyYAML-3.12 pyaml-17.12.1
```

To test it, go to the Python shell and import the yaml module,

import yaml, if no error is found, then we can say installation is successful.

After installing pyaml, let's look at below code,

```
script_yaml1.py
```

```
import yaml

mydict = {'a':2, 'b':4, 'c': 6}
mylist = [1, 2, 3, 4, 5]
mytuple = ('x', 'y', 'z')

print(yaml.dump(mydict, default_flow_style=False))
#print(loaded_yaml)

print(yaml.dump(mylist, default_flow_style = False))

print(yaml.dump(mytuple, default_flow_style = False))
```

Above we created three different data structure, dictionary, list and tuple. On each of the structure, we do yaml.dump. Important point is how the output is displayed on the screen.

Output

```
a: 2
b: 4
c: 6

- 1
- 2
- 3
- 4
- 5

!!python/tuple
- x
- y
- z
```

Dictionary output looks clean .ie. key: value.

White space to separate different objects.

List is notated with dash (-)

Tuple is indicated first with !!Python/tuple and then in the same format as lists.

Loading a yaml file

So let's say I have one yaml file, which contains,

```
---
# An employee record
name: Raagvendra Joshi
job: Developer
skill: Oracle
employed: True
foods:
    - Apple
    - Orange
    - Strawberry
    - Mango
languages:
    Oracle: Elite
    power_builder: Elite
    Full Stack Developer: Lame
education:
    4 GCSEs
    3 A-Levels
    MCA in something called com
```

Now let's write a code to load this yaml file through yaml.load function. Below is code for the same.

```
import yaml


with open('eRecord.yaml') as fh:
    struct = yaml.load(fh)

print(struct)

print ("========================================")
print('To make the output in more readable format, add the json module!')


import json
print(json.dumps(struct, indent = 4, separators = (',', ': ')))
```

As the output doesn't looks that much readable, I prettify it by using json in the end. Compare the output we got and the actual yaml file we have.

Output

```
{'name': 'Raagvendra Joshi', 'job': 'Developer', 'skill': 'Oracle', 'employed':
True, 'foods': ['Apple', 'Orange', 'Strawberry', 'Mango'], 'languages': {'Oracle
': 'Elite', 'power_builder': 'Elite', 'Full Stack Developer': 'Lame'}, 'educatio
n': '4 GCSEs 3 A-Levels MCA in something called computer'}
========================================
To make the output in more readable format, add the json module!
{
    "name": "Raagvendra Joshi",
    "job": "Developer",
    "skill": "Oracle",
    "employed": true,
    "foods": [
        "Apple",
        "Orange",
        "Strawberry",
        "Mango"
    ],
    "languages": {
        "Oracle": "Elite",
        "power_builder": "Elite",
        "Full Stack Developer": "Lame"
    },
    "education": "4 GCSEs 3 A-Levels MCA in something called computer"
}
```

One of the most important aspect of software development is debugging. In this section we'll see different ways of Python debugging either with built-in debugger or third party debuggers.

PDB – The Python Debugger

The module PDB supports setting breakpoints. A breakpoint is an intentional pause of the program, where you can get more information about the programs state.

To set a breakpoint, insert the line

```
pdb.set_trace()
```

Example

```
pdb_example1.py
import pdb
x=9
y=7
pdb.set_trace()
total = x + y
pdb.set_trace()
```

We have inserted a few breakpoints in this program. The program will pause at each breakpoint (pdb.set_trace()). To view a variables contents simply type the variable name.

```
c:\Python\Python361>Python pdb_example1.py
> c:\Python\Python361\pdb_example1.py(8)<module>()
-> total = x + y
(Pdb) x
9
(Pdb) y
7
(Pdb) total
*** NameError: name 'total' is not defined
(Pdb)
```

Press c or continue to go on with the programs execution until the next breakpoint.

```
(Pdb) c
--Return--
> c:\Python\Python361\pdb_example1.py(8)<module>()->None
-> total = x + y
(Pdb) total
```

```
16
```

Eventually, you will need to debug much bigger programs – programs that use subroutines. And sometimes, the problem that you're trying to find will lie inside a subroutine. Consider the following program.

```
import pdb
def squar(x, y):
    out_squared = x^2 + y^2
    return out_squared
if __name__ == "__main__":
    #pdb.set_trace()
    print (squar(4, 5))
```

Now on running the above program,

```
c:\Python\Python361>Python pdb_example2.py
> c:\Python\Python361\pdb_example2.py(10)<module>()
-> print (squar(4, 5))
(Pdb)
```

We can use **?** to get help, but the arrow indicates the line that's about to be executed. At this point it's helpful to hit s to **s** to step into that line.

```
(Pdb) s
--Call--
> c:\Python\Python361\pdb_example2.py(3)squar()
-> def squar(x, y):
```

This is a call to a function. If you want an overview of where you are in your code, try l:

```
(Pdb) l
  1     import pdb
  2
  3     def squar(x, y):
  4  ->     out_squared = x^2 + y^2
  5
  6         return out_squared
  7
  8     if __name__ == "__main__":
  9         pdb.set_trace()
 10         print (squar(4, 5))
                [EOF]
```

```
(Pdb)
```

You can hit n to advance to the next line. At this point you are inside the out_squared method and you have access to the variable declared inside the function .i.e. x and y.

```
(Pdb) x
4
(Pdb) y
5
(Pdb) x^2
6
(Pdb) y^2
7
(Pdb) x**2
16
(Pdb) y**2
25
(Pdb)
```

So we can see the ^ operator is not what we wanted instead we need to use ** operator to do squares.

This way we can debug our program inside the functions/methods.

Logging

The logging module has been a part of Python's Standard Library since Python version 2.3. As it's a built-in module all Python module can participate in logging, so that our application log can include your own message integrated with messages from third party module. It provides a lot of flexibility and functionality.

Benefits of Logging

- Diagnostic logging: It records events related to the application's operation.
- Audit logging: It records events for business analysis.

Messages are written and logged at levels of "severity":

- DEBUG (debug()): diagnostic messages for development.
- INFO (info()): standard "progress" messages.
- WARNING (warning()): detected a non-serious issue.
- ERROR (error()): encountered an error, possibly serious
- CRITICAL (critical()): usually a fatal error (program stops)

Let's looks into below simple program,

logging1.py

```
import logging


logging.basicConfig(level=logging.INFO)


logging.debug('this message will be ignored')      # This will not print
logging.info('This should be logged')               # it'll print
logging.warning('And this, too')                    # It'll print
```

Above we are logging messages on severity level. First we import the module, call basicConfig and set the logging level. Level we set above is INFO. Then we have three different statement: debug statement, info statement and a warning statement.

Output of logging1.py

```
INFO:root:This should be logged
WARNING:root:And this, too
```

As the info statement is below debug statement, we are not able to see the debug message. To get the debug statement too in the output terminal, all we need to change is the basicConfig level.

```
logging.basicConfig(level=logging.DEBUG)
```

And in the output we can see,

```
DEBUG:root:this message will be ignored
INFO:root:This should be logged
WARNING:root:And this, too
```

Also the default behavior means if we don't set any logging level is warning. Just comment out the second line from the above program and run the code.

```
#logging.basicConfig(level=logging.DEBUG)
```

Output

```
WARNING:root:And this, too
```

Python built in logging level are actually integers.

```
>>> import logging
>>>
>>> logging.DEBUG
10
>>> logging.CRITICAL
```

```
50
>>> logging.WARNING
30
>>> logging.INFO
20
>>> logging.ERROR
40
>>>
```

We can also save the log messages into the file.

```
logging.basicConfig(level=logging.DEBUG, filename = 'logging.log')
```

Now all log messages will go the file (logging.log) in your current working directory instead of the screen. This is a much better approach as it lets us to do post analysis of the messages we got.

We can also set the date stamp with our log message.

```
logging.basicConfig(level=logging.DEBUG, format = '%(asctime)s
%(levelname)s:%(message)s')
```

Output will get something like,

```
2018-03-08 19:30:00,066  DEBUG:this message will be ignored
2018-03-08 19:30:00,176  INFO:This should be logged
2018-03-08 19:30:00,201  WARNING:And this, too
```

Benchmarking

Benchmarking or profiling is basically to test how fast is your code executes and where the bottlenecks are? The main reason to do this is for optimization.

timeit

Python comes with a in-built module called timeit. You can use it to time small code snippets. The timeit module uses platform-specific time functions so that you will get the most accurate timings possible.

So, it allows us to compare two shipment of code taken by each and then optimize the scripts to given better performance.

The timeit module has a command line interface, but it can also be imported.

There are two ways to call a script. Let's use the script first, for that run the below code and see the output.

```
import timeit
```

```
print ( 'by index: ', timeit.timeit(stmt = "mydict['c']", setup="mydict =
{'a':5, 'b':10, 'c':15}", number = 1000000))


print ( 'by get: ', timeit.timeit(stmt = 'mydict.get("c")', setup = 'mydict =
{"a":5, "b":10, "c":15}', number = 1000000))
```

Output

```
by index:  0.1809192126703489

by get:  0.6088525265034692
```

Above we use two different method .i.e. by subscript and get to access the dictionary key value. We execute statement 1 million times as it executes too fast for a very small data. Now we can see the index access much faster as compared to the get. We can run the code multiply times and there will be slight variation in the time execution to get the better understanding.

Another way is to run the above test in the command line. Let's do it,

```
c:\Python\Python361>Python -m timeit -n 1000000 -s "mydict = {'a': 5, 'b':10,
'c':15}" "mydict['c']"

1000000 loops, best of 3: 0.187 usec per loop


c:\Python\Python361>Python -m timeit -n 1000000 -s "mydict = {'a': 5, 'b':10,
'c':15}" "mydict.get('c')"

1000000 loops, best of 3: 0.659 usec per loop
```

Above output may vary based on your system hardware and what all applications are running currently in your system.

Below we can use the timeit module, if we want to call to a function. As we can add multiple statement inside the function to test.

```
import timeit


def testme(this_dict, key):

    return this_dict[key]


print (timeit.timeit("testme(mydict, key)", setup ="from __main__ import
testme; mydict = {'a':9, 'b':18, 'c':27}; key='c'", number=1000000))
```

Output

```
0.7713474590139164
```

12. 12. OOP in Python – Python Libraries

Requests: Python Requests Module

Requests is a Python module which is an elegant and simple HTTP library for Python. With this you can send all kinds of HTTP requests. With this library we can add headers, form data, multipart files and parameters and access the response data.

As Requests is not a built-in module, so we need to install it first.

You can install it by running the following command in the terminal:

```
pip install requests
```

Once you have installed the module, you can verify if the installation is successful by typing below command in the Python shell.

```
import requests
```

If the installation has been successful, you won't see any error message.

Making a GET Request

As a means of example we'll be using the "pokeapi"

```
import requests
import json

def main():
    req = requests.get('http://pokeapi.co/api/v2/pokemon/1/')
    print('HTTP Status Code: ' + str(req.status_code))
    print(req.headers)
    json_response = json.loads(req.content)
    print('Pokemon Name: ' + json_response['name'])

if __name__ == '__main__':
    main()
```

Output:

```
HTTP Status Code: 200
{'Date': 'Thu, 01 Mar 2018 05:49:39 GMT', 'Content-Type': 'application/json', 'Transfer-Encoding': 'chunked', 'Connect
ion': 'keep-alive', 'Set-Cookie': '__cfduid=dfdc3b64c65430b43d1c24d1951e66d2b1519883377; expires=Fri, 01-Mar-19 05:49:
37 GMT; path=/; domain=.pokeapi.co; HttpOnly; Secure', 'Vary': 'Accept-Encoding, Cookie', 'X-Frame-Options': 'SAMEORIG
IN', 'Allow': 'GET, HEAD, OPTIONS', 'X-XSS-Protection': '1; mode=block', 'Content-Encoding': 'gzip', 'Expect-CT': 'max
-age=604800, report-uri="https://report-uri.cloudflare.com/cdn-cgi/beacon/expect-ct"', 'Server': 'cloudflare', 'CF-RAY
': '3f495767490b0805-SIN'}
Pokemon Name: bulbasaur
```

Making POST Requests

The requests library methods for all of the HTTP verbs currently in use. If you wanted to make a simple POST request to an API endpoint then you can do that like so:

```
req = requests.post('http://api/user', data=None, json=None)
```

This would work in exactly the same fashion as our previous GET request, however it features two additional keyword parameters:

- data which can be populated with say a dictionary, a file or bytes that will be passed in the HTTP body of our POST request.

- json which can be populated with a json object that will be passed in the body of our HTTP request also.

Pandas: Python Library Pandas

Pandas is an open-source Python Library providing high-performance data manipulation and analysis tool using its powerful data structures. Pandas is one of the most widely used Python libraries in data science. It is mainly used for data munging, and with good reason: Powerful and flexible group of functionality.

Built on Numpy package and the key data structure is called the DataFrame. These dataframes allows us to store and manipulate tabular data in rows of observations and columns of variables.

There are several ways to create a DataFrame. One way is to use a dictionary. For example:

```
import pandas as pd

dict = {"Bric_country": ["Brazil", "Russia", "India", "China", "South Africa"],
       "Brics_capital": ["Brasília", "Moscow", "New Dehli", "Beijing", "Pretoria"],
       "area": [8.516, 17.10, 3.286, 9.597, 1.221],
       "population": [200.4, 143.5, 1252, 1357, 52.98] }


brics = pd.DataFrame(dict)
print(brics)
```

Output:

```
   Bric_country Brics_capital    area  population
0        Brazil      Brasília   8.516      200.40
1        Russia        Moscow  17.100      143.50
2         India     New Dehli   3.286     1252.00
3         China       Beijing   9.597     1357.00
4  South Africa      Pretoria   1.221       52.98
```

From the output we can see new brics DataFrame, Pandas has assigned a key for each country as the numerical values 0 through 4.

If instead of giving indexing values from 0 to 4, we would like to have different index values, say the two letter country code, you can do that easily as well:

Adding below one lines in the above code, gives

brics.index = ['BR', 'RU', 'IN', 'CH', 'SA']

Output

```
   Bric_country Brics_capital    area  population
BR       Brazil      Brasília   8.516      200.40
RU       Russia        Moscow  17.100      143.50
IN        India     New Dehli   3.286     1252.00
CH        China       Beijing   9.597     1357.00
SA  South Africa      Pretoria   1.221       52.98
```

Indexing DataFrames

```
import pandas as pd

stocks_list = pd.read_csv('stocks_list.csv', index_col = 0)

# print the stock_list.csv file
print(stocks_list)

print(" ")
#print the ISIN column as pandas series
print(stocks_list['ISIN'])


# print the ISIN as pandas DataFrame

print(stocks_list[['ISIN', 'TOTALTRADES']])
```

Output

```
>>>
========== RESTART: C:/Python/Python361/pandas_script2.py ==========
           SERIES     OPEN     HIGH      LOW    CLOSE     LAST  PREVCLOSE  \
SYMBOL
NTPC           EQ   164.00   169.20   163.75   168.75   168.45     167.95
RELIANCE       EQ  1575.00  1597.45  1574.00  1594.50  1595.40    1564.10
HDFC           EQ  1762.00  1780.95  1746.00  1776.90  1779.80    1759.05
INFY           EQ   928.80   928.95   912.90   914.95   915.00     926.65
HINDALCO       EQ   237.80   239.45   235.10   238.00   238.25     236.35
IOC            EQ   454.70   462.95   448.05   454.70   455.50     452.85
RELCAPITAL     EQ   781.30   812.00   775.00   805.30   811.00     779.90
VEDL           EQ   305.35   310.00   300.90   308.90   308.80     305.55
YESBANK        EQ  1750.00  1774.40  1745.05  1753.05  1754.00    1752.10

            TOTTRDQTY    TOTTRDVAL  TIMESTAMP  TOTALTRADES          ISIN
SYMBOL
NTPC         83350065  13902129064  31-Aug-17       157814  INE733E01010
RELIANCE      5738495   9108681150  31-Aug-17       153607  INE002A01018
HDFC          4646322   8220571377  31-Aug-17       179595  INE001A01036
INFY          7511226   6886139481  31-Aug-17       174045  INE009A01021
HINDALCO     23200769   5516181928  31-Aug-17        79864  INE038A01020
IOC          11525950   5260716070  31-Aug-17       124043  INE242A01010
RELCAPITAL    6647282   5252965990  31-Aug-17        78603  INE013A01015
VEDL         14987864   4595411062  31-Aug-17        89794  INE205A01025
YESBANK       2566761   4515288596  31-Aug-17        80122  INE528G01019

                  ISIN  TOTALTRADES
SYMBOL
NTPC       INE733E01010       157814
RELIANCE   INE002A01018       153607
HDFC       INE001A01036       179595
INFY       INE009A01021       174045
HINDALCO   INE038A01020        79864
IOC        INE242A01010       124043
RELCAPITAL INE013A01015        78603
VEDL       INE205A01025        89794
YESBANK    INE528G01019        80122
>>>
```

Pygame

Pygame is the open source and cross-platform library that is for making multimedia applications including games. It includes computer graphics and sound libraries designed to be used with the Python programming language. You can develop many cool games with Pygame.'

Overview

Pygame is composed of various modules, each dealing with a specific set of tasks. For example, the display module deals with the display window and screen, the draw module provides functions to draw shapes and the key module works with the keyboard. These are just some of the modules of the library.

The home of the Pygame library is at http://pygame.org.

To make a Pygame application, you follow these steps:

Import the Pygame library

```
import pygame
```

Initialize the Pygame library

```
pygame.init()
```

Create a window.

```
screen = Pygame.display.set_mode((560,480))
Pygame.display.set_caption('First Pygame Game')
```

Initialize game objects

In this step we load images, load sounds, do object positioning, set up some state variables, etc.

Start the game loop.

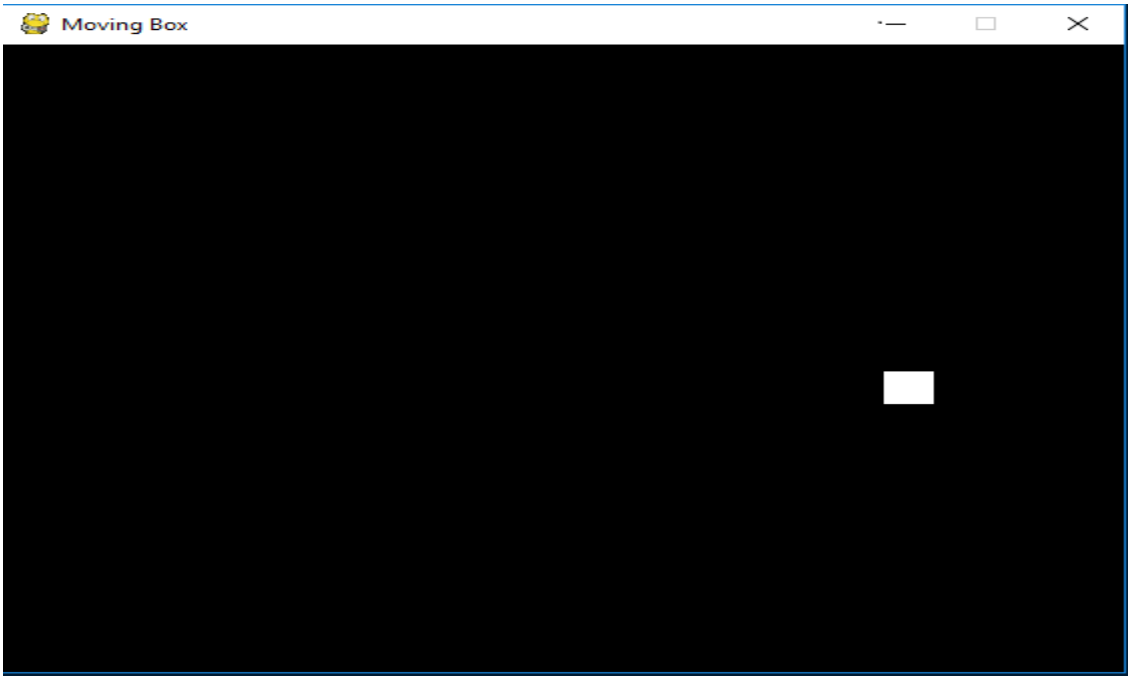
It is just a loop where we continuously handle events, checks for input, move objects, and draw them. Each iteration of the loop is called a frame.

Let’s put all the above logic into one below program,

Pygame_script.py

```
#Import the pygame and the sys module for exiting the window we create
import sys, pygame

BLACK = 0, 0, 0
WHITE = 255, 255, 255
# Initialise the pygame module
pygame.init()
#Create a new window, width=560, height=480
screen = pygame.display.set_mode((560,480))
#Give the window a caption
pygame.display.set_caption("Moving Box")
clock = pygame.time.Clock()
box_x = 300
box_dir = 3
#Loop (repeat) forever
while 1:
  clock.tick(55)
#Get all the users events
  for event in pygame.event.get():
      #if the user wants to quit
    if event.type == pygame.QUIT:
        pygame.quit()
        sys.exit()

  screen.fill(BLACK)

  box_x += box_dir
  if box_x >= 620:
    box_x = 620
    box_dir = -3
  elif box_x <= 0:
    box_x = 0
    box_dir = 3

  pygame.draw.rect(screen, WHITE, (box_x, 250, 25, 25))
  pygame.display.flip()
```

Output



Beautiful Soup: Web Scraping with Beautiful Soup

The general idea behind web scraping is to get the data that exists on a website, and convert it into some format that is usable for analysis.

It's a Python library for pulling data out of HTML or XML files. With your favourite parser it provide idiomatic ways of navigating, searching and modifying the parse tree.

As BeautifulSoup is not a built-in library, we need to install it before we try to use it. To install BeautifulSoup, run the below command

```
$ apt-get install Python-bs4 # For Linux and Python2
$ apt-get install Python3-bs4         # for Linux based system and Python3.

$ easy_install beautifulsoup4         # For windows machine,
Or
$ pip instal beatifulsoup4      # For window machine
```

Once the installation is done, we are ready to run few examples and explores Beautifulsoup in details,

```
#Import Beautiful Soup
from bs4 import BeautifulSoup
from urllib.request import urlopen


r = urlopen('https://en.wikipedia.org/wiki/List_of_countries_by_foreign-exchange_reserves_(excluding_gold)').read()
soup = BeautifulSoup(r, 'html.parser')

print(type(soup))

print(soup.prettify()[0:1000])
```

Output

```
<class 'bs4.BeautifulSoup'>
<!DOCTYPE html>
<html class="client-nojs" dir="ltr" lang="en">
 <head>
  <meta charset="utf-8"/>
  <title>
   List of countries by foreign-exchange reserves (excluding gold) - Wikipedia
  </title>
  <script>
   document.documentElement.className = document.documentElement.className.repla
ce( /(^|\s)client-nojs(\s|$)/, "$1client-js$2" );
  </script>
  <script>
   (window.RLQ=window.RLQ||[]).push(function(){mw.config.set({"wgCanonicalNamesp
ace":"","wgCanonicalSpecialPageName":false,"wgNamespaceNumber":0,"wgPageName":"L
ist_of_countries_by_foreign-exchange_reserves_(excluding_gold)","wgTitle":"List
of countries by foreign-exchange reserves (excluding gold)","wgCurRevisionId":82
8715850,"wgRevisionId":828715850,"wgArticleId":39057473,"wgIsArticle":true,"wgIs
Redirect":false,"wgAction":"view","wgUserName":null,"wgUserGroups":["*"],"wgCate
gories":["Use dmy dates from September 2016","Lists of countries by economic ind
icator","Foreign exchange reserves"],"wgBreakFrames":false,"wgPageContentLangua
```

Below are some simple ways to navigate that data structure:

```
>>> # Simple tricks to navigate different information we can get from data struc
ture.
>>>
>>> soup.title
<title>List of countries by foreign-exchange reserves (excluding gold) - Wikiped
ia</title>


>>> soup.title.name
'title'
>>>
>>> soup.title.string
'List of countries by foreign-exchange reserves (excluding gold) - Wikipedia'
>>>
>>> soup.p
<p>This is a list of the top 33 <a href="/wiki/List_of_sovereign_states" title="
List of sovereign states">sovereign states</a> of the <a href="/wiki/World" titl
e="World">world</a> sorted by their <a href="/wiki/Foreign-exchange_reserves" ti
tle="Foreign-exchange reserves">foreign-exchange reserves</a> <b>excluding</b> <
a href="/wiki/Gold_reserve" title="Gold reserve">gold reserves</a>, <b>but inclu
ding</b> <a href="/wiki/Special_drawing_rights" title="Special drawing rights">s
pecial drawing rights</a> (SDRs) and <a href="/wiki/International_Monetary_Fund"
 title="International Monetary Fund">International Monetary Fund</a> (IMF) reser
ve positions.</p>
>>>
```

One common task is extracting all the URLs found within a page's <a> tags:

```
>>> # Extracting all the URLs found within a page's <a> tags
>>> for link in soup.find_all('a'):
        print(link.get('href'))


None
#mw-head
#p-search
/wiki/List_of_countries_by_foreign-exchange_reserves
/wiki/List_of_sovereign_states
/wiki/World
/wiki/Foreign-exchange_reserves
/wiki/Gold_reserve
/wiki/Special_drawing_rights
/wiki/International_Monetary_Fund
#cite_note-imf-1
/wiki/Sovereign_state
/wiki/Hong_Kong
/wiki/Macau
/wiki/Eurozone
/wiki/United_States_dollar
/wiki/China
#cite_note-2
#cite_note-imf-1
/wiki/Japan
#cite_note-3
#cite_note-imf-1
/wiki/Switzerland
#cite_note-4
/wiki/Saudi_Arabia
#cite_note-imf-1
/wiki/Taiwan
#cite_note-5
```

Another common task is extracting all the text from a page:

```
>>> print(soup.get_text())



List of countries by foreign-exchange reserves (excluding gold) - Wikipedia
document.documentElement.className = document.documentElement.className.replace( /(^|\s)client-nojs(\s|$)/, "$1client-js$2" );
(window.RLQ=window.RLQ||[]).push(function(){mw.config.set({"wgCanonicalNamespace":"","wgCanonicalSpecialPageName":false,"wgNamespaceNumber":0,"wgPageName":"List_of_co
untries_by_foreign-exchange_reserves_(excluding_gold)","wgTitle":"List of countries by foreign-exchange reserves (excluding gold)","wgCurRevisionId":828715850,"wgRevi
sionId":828715850,"wgArticleId":39057473,"wgIsArticle":true,"wgIsRedirect":false,"wgAction":"view","wgUserName":null,"wgUserGroups":["*"],"wgCategories":["Use dmy dat
es from September 2016","Lists of countries by economic indicator","Foreign exchange reserves"],"wgBreakFrames":false,"wgPageContentLanguage":"en","wgPageContentModel
":"wikitext","wgSeparatorTransformTable":["",""],"wgDigitTransformTable":["",""],"wgDefaultDateFormat":"dmy","wgMonthNames":["","January","February","March","April","
May","June","July","August","September","October","November","December"],"wgMonthNamesShort":["","Jan","Feb","Mar","Apr","May","Jun","Jul","Aug","Sep","Oct","Nov","De
c"],"wgRelevantPageName":"List_of_countries_by_foreign-exchange_reserves_(excluding_gold)","wgRelevantArticleId":39057473,"wgRequestId":"WpuBOgpAAEQAAD0746QAAAAV","wg
IsProbablyEditable":true,"wgRelevantPageIsProbablyEditable":true,"wgRestrictionEdit":[],"wgRestrictionMove":[],"wgFlaggedRevsParams":{"tags":{}},"wgStableRevisionId":
null,"wgWikiEditorEnabledModules":[],"wgBetaFeaturesFeatures":[],"wgMediaViewerOnClick":true,"wgMediaViewerEnabledByDefault":true,"wgPopupsShouldSendModuleToUser":tru
e,"wgPopupsConflictsWithNavPopupGadget":false,"wgVisualEditor":{"pageLanguageCode":"en","pageLanguageDir":"ltr","pageVariantFallbacks":"en","usePageImages":true,"useP
ageDescriptions":true},"wgPreferredVariant":"en","wgMFExpandAllSectionsUserOption":true,"wgMFEnableFontChanger":true,"wgMFDisplayWikibaseDescriptions":{"search":false
,"nearby":false,"watchlist":false,"tagline":false},"wgRelatedArticles":null,"wgRelatedArticlesUseCirrusSearch":true,"wgRelatedArticlesOnlyUseCirrusSearch":false,"wgUL
SCurrentAutonym":"English","wgNoticeProject":"wikipedia","wgCentralNoticeCookiesToDelete":[],"wgCentralNoticeCategoriesUsingLegacy":["Fundraising","fundraising"],"wgC
ategoryTreePageCategoryOptions":"{\"mode\":0,\"hideprefix\":20,\"showcount\":true,\"namespaces\":false}","wgWikibaseItemId":"Q17107491","wgScoreNoteLanguages":{"arabi
c":"العربية","catalan":"català","deutsch":"Deutsch","english":"English","espanol":"español","italiano":"italiano","nederlands":"Nederlands","norsk":"norsk","portugues
":"português","suomi":"suomi","svenska":"svenska","vlaams":"West-Vlams"},"wgScoreDefaultNoteLanguage":"nederlands","wgCentralAuthMobileDomain":false,"wgCodeMirrorEnab
led":false,"wgVisualEditorToolbarScrollOffset":0,"wgVisualEditorUnsupportedEditParams":["undo","undoafter","veswitched"],"wgEditSubmitButtonLabelPublish":true});mw.lo
ader.state({"ext.gadget.charinsert-styles":"ready","ext.globalCssJs.user.styles":"ready","ext.globalCssJs.site.styles":"ready","site.styles":"ready","noscript":"ready
","user.styles":"ready","user":"ready","user.options":"ready","user.tokens":"loading","ext.cite.styles":"ready","wikibase.client.init":"ready","ext.visualEditor.deskt
opArticleTarget.noscript":"ready","ext.uls.interlanguage":"ready","ext.wikimediaBadges":"ready","mediawiki.legacy.shared":"ready","mediawiki.legacy.commonPrint":"read
y","mediawiki.sectionAnchor":"ready","mediawiki.skinning.interface":"ready","skins.vector.styles":"ready","ext.globalCssJs.user":"ready","ext.globalCssJs.site":"ready
"});mw.loader.implement("user.tokens@1dqfd71",function($,jQuery,require,module){/*@nomin*/mw.user.tokens.set({"editToken":"+\\","patrolToken":"+\\","watchToken":"+\\"
,"csrfToken":"+\\"});
});mw.loader.load(["ext.cite.a11y","site","mediawiki.page.startup","mediawiki.user","mediawiki.hidpi","mediawiki.page.ready","jquery.tablesorter","mediawiki.searchSug
gest","ext.gadget.teahouse","ext.gadget.ReferenceTooltips","ext.gadget.watchlist-notice","ext.gadget.DRN-wizard","ext.gadget.charinsert","ext.gadget.refToolbar","ext.
gadget.extra-toolbar-buttons","ext.gadget.switcher","ext.centralauth.centralautologin","mmv.head","mmv.bootstrap.autostart","ext.popups","ext.visualEditor.desktopArti
```