i

OOP in Python

**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.

ii

OOP in Python

**Table of Contents**

**About the Tutorial ...........................................................................................................................................ii Audience..........................................................................................................................................................ii Prerequisites....................................................................................................................................................ii**

OOP IN PYTHON – INTRODUCTION ............................................................................................. 1 **Language Programming Classification Scheme ................................................................................................1 What is Object Oriented Programming? ..........................................................................................................2 Why to Choose Object-oriented programming?...............................................................................................2 Procedural vs. Object Oriented Programming..................................................................................................2 Principles of Object Oriented Programming.....................................................................................................3 Object-Oriented Python...................................................................................................................................5 Modules vs. Classes and Objects......................................................................................................................5**

OOP IN PYTHON – ENVIRONMENT SETUP ................................................................................... 8 **Prerequisites and Toolkits ...............................................................................................................................8 Installing Python..............................................................................................................................................8 Choosing an IDE .............................................................................................................................................10 Pycharm.........................................................................................................................................................10 Komodo IDE...................................................................................................................................................11 Eric Python IDE ..............................................................................................................................................12 Choosing a Text Editor...................................................................................................................................13 Atom Text Editor ...........................................................................................................................................13 Screenshot of Atom text................................................................................................................................14 Sublime Text Editor .......................................................................................................................................14 Notepad ++....................................................................................................................................................15**

OOP IN PYTHON – DATA STRUCTURES ...................................................................................... 17

**Lists.....................................................................................................................................17**

OOP in Python

**Accessing Items in Python List .......................................................................................................................18 Empty Objects ...............................................................................................................................................18 Tuples............................................................................................................................................................19 Dictionary ......................................................................................................................................................21 Sets................................................................................................................................................................24**

OOP IN PYTHON – BUILDING BLOCKS ........................................................................................ 28 **Class Bundles : Behavior and State ................................................................................................................28 Creation and Instantiation.............................................................................................................................29 Instance Methods..........................................................................................................................................30 Encapsulation ................................................................................................................................................31 Init Constructor..............................................................................................................................................33 Class Attributes..............................................................................................................................................34 Working with Class and Instance Data ...........................................................................................................35**

OOP IN PYTHON – OBJECT ORIENTED SHORTCUT ..................................................................... 37 **Python Built-in Functions...............................................................................................................................37 Default Arguments ........................................................................................................................................42**

OOP IN PYTHON – INHERITANCE AND POLYMORPHISM ........................................................... 44 **Inheritance ....................................................................................................................................................44 Inheriting Attributes......................................................................................................................................44 Inheritance Examples.....................................................................................................................................45 Polymorphism (“MANY SHAPES”)..................................................................................................................47 Overriding......................................................................................................................................................48 Inheriting the Constructor .............................................................................................................................49 Multiple Inheritance and the Lookup Tree.....................................................................................................50 Decorators, Static and Class Methods............................................................................................................54**

OOP IN PYTHON –PYTHON DESIGN PATTERN............................................................................ 57

i

OOP in Python

**Overview .......................................................................................................................................................57 Why is Design Pattern Important? .................................................................................................................57 Classification of Design Patterns....................................................................................................................57 Commonly used Design Patterns ...................................................................................................................58**

OOP IN PYTHON – ADVANCED FEATURES ................................................................................. 60 **Core Syntax in our Class design......................................................................................................................60 Inheriting From built-in types........................................................................................................................61 Naming Conventions......................................................................................................................................63**

OOP IN PYTHON – FILES AND STRINGS ...................................................................................... 65 **Strings............................................................................................................................................................66 File I/O...........................................................................................................................................................71**

OOP IN PYTHON – EXCEPTION AND EXCEPTION CLASSES.......................................................... 72 **Identifying Exception (Errors) ........................................................................................................................72 Catching/Trapping Exception.........................................................................................................................73 Raising Exceptions .........................................................................................................................................75 Creating Custom exception class....................................................................................................................76**

OOP IN PYTHON – OBJECT SERIALIZATION ................................................................................ 80 **Pickle .............................................................................................................................................................80 Methods........................................................................................................................................................81 Unpickling......................................................................................................................................................82 JSON ..............................................................................................................................................................82 YAML .............................................................................................................................................................85 Installing YAML..............................................................................................................................................85 PDB – The Python Debugger ..........................................................................................................................89 Logging ..........................................................................................................................................................91 Benchmarking................................................................................................................................................93**

****ii

OOP in Python

12. OOP IN PYTHON – PYTHON LIBRARIES ................................................................................ 96 **Requests: Python Requests Module ..............................................................................................................96 Making a GET Request...................................................................................................................................96 Making POST Requests..................................................................................................................................97 Pandas: Python Library Pandas......................................................................................................................97 Indexing DataFrames.....................................................................................................................................98 Pygame..........................................................................................................................................................99 Beautiful Soup: Web Scraping with Beautiful Soup......................................................................................102**

****iii

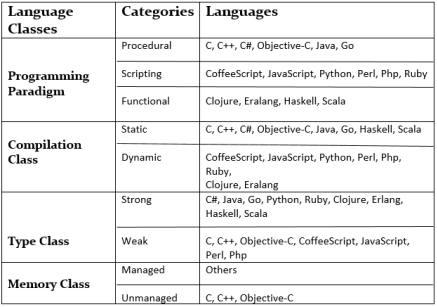
1. OOP in Python – Introduction OOP in Python

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.

1

OOP in Python

**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 ObjectOriented 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

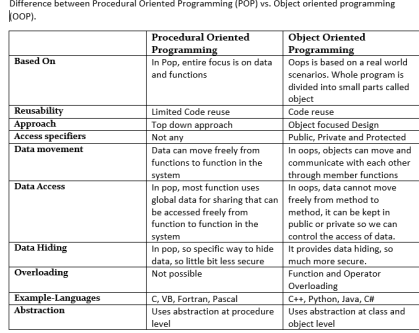
**Procedural vs. Object Oriented Programming**

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.

2

OOP in Python

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

**Principlesof 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.

3

OOP in Python



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.

4

OOP in Python

**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() |
| --- |

5

OOP in Python

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”) |
| --- |

6

OOP in Python

**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) |
| --- |

7

2. OOP in Python – Environment Setup OOP in Python

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:

o Able to write and run Python programs.

o Debug programs and diagnose errors.

o Work with basic data types.

o Write **for** loops, **while** loops, and **if** statements

o 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.

8

OOP in Python

**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.

9

OOP in Python

**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

10

OOP in Python

∙ 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

11

OOP in Python

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.

**EricPythonIDE**

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.

12

OOP in Python

**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/

13

OOP in Python

**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.

14

OOP in Python

**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.

15

OOP in Python

**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.

16

3. OOP in Python – Data Structures OOP in Python

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] |
| --- |

17

OOP in Python

| >>> print(mixed\_list)  ['This', 9, 'is', 18, 45.9, 'a', 54, 'mixed', 99, 'list'] |
| --- |

**Accessing Items in PythonList**

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' |
| --- |

18

OOP in Python

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  >>> |
| --- |

19

**Defining a Tuple**

OOP in Python

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') |
| --- |

20

OOP in Python

| 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]

21

OOP in Python

| '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.

22

OOP in Python

∙ 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  {} |
| --- |

23

OOP in Python

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.

24

OOP in Python

| >>> 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 |
| --- |

25 

OOP in Python

>>> 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):

26

OOP in Python

| 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() |
| --- |

27

4. OOP in Python – Building Blocks OOP in Python

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:BehaviorandState**

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.

28

OOP in Python

**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> |
| --- |

29

OOP in Python

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() |
| --- |

30

OOP in Python

**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**.

31

OOP in Python

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) |
| --- |

32 

OOP in Python

| 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 |
| --- |

33

**Class Attributes**

OOP in Python

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  >>> |
| --- |

34

OOP in Python

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 |
| --- |

35

OOP in Python

**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'} |
| --- |

36

5. OOP in Python – Object Oriented Shortcuts OOP in Python

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

**PythonBuilt-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 |
| --- |

37

OOP in Python

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\_\_**.

38

OOP in Python

**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

39

OOP in Python

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() |
| --- |

40

OOP in Python

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 |
| --- |

41 

OOP in Python

| 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=' ')

42

OOP in Python

| 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.

43

6. OOP in Python – Inheritance and Polymorphism OOP in Python

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:

44

OOP in Python

**Output**

****

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: Object

Animal

Cat Dog

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.

45

OOP in Python



**Output**

****

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).

46

OOP in Python

**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.

**Output**

****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”.

47

OOP in Python

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.



**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!') |
| --- |

48

OOP in Python

**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.

**Output**

****

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,

49

OOP in Python

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

50

OOP in Python

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:

51

OOP in Python

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

**Output**

****52

OOP in Python

**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.

53

OOP in Python

Let’s create an example based on above diagram:



**Output**

****

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.

54

OOP in Python

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 |
| --- |

55

OOP in Python

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.

56

7. OOP in Python –Python Design Pattern OOP in Python

**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.

57

**Structural Patterns**

OOP in Python

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,

**Singleton**

Single

+instance: static

+get\_instance() : static

| class Logger(object):  def \_\_new\_\_(cls, \*args, \*\*kwargs):  if not hasattr(cls, '\_logger'): |
| --- |

58

OOP in Python

| 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> |
| --- |

59

8. OOP in Python – Advanced Features OOP in Python

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): |
| --- |

60

OOP in Python

| 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.

61

OOP in Python

| 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

62

OOP in Python

| 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,

63

OOP in Python

∙ 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

64

9. OOP in Python – Files and Strings OOP in Python

| 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! |
| --- |

65

**Strings**

OOP in Python

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. |

66

OOP in Python

| 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')

67

OOP in Python

| >>> |
| --- |

**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\*\*\*\*\*\*' |
| --- |

68

OOP in 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'*

**69

OOP in Python

*>>>*

*>>> # 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')*

*>>>*

**70

OOP in Python

*>>> #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.

71

10.OOP in Python – Exception and Exception Classes OOP in Python

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. |
| --- |

72 

OOP in Python

| >>> 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 |
| --- |

73

OOP in Python

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 |
| --- |

74