



Python Fundamentals

Introduction

- Python is a very popular highly object oriented language
- Easy to learn due to its simple structure and syntax
- Very versatile language with application in many domains like AI/ ML, Data Science etc.
- Selenium is a very popular browser automation tool
- Combined with Python Selenium is fast becoming the choice of language for web application automation testing



Course Overview - Python

- Python Fundamentals
- Installation and Setup (Python, PyCharm IDE)
- Python Variables, Data Types and Data Structures
- Python Loops and Conditionals
- Python Functions
- Python Packages and Modules






Python Fundamentals

Installation

- Go to <https://www.python.org/downloads/> and download the installer depending on your OS
- Double click the installer and follow instructions mentioned in the wizard



Setting PATH

- After installation open command prompt and type python
 - If you see any error message like “python is not recognised as a command or program” it means that system is unable to find the python executable
 - Usually system looks into environment variable called path to find executables of different programs
 - To set path, open environment variables
 - Under system settings section find a variable called path
 - Double click it and at the end add location of python home folder
 - e. g. path = %path%;C:\Python
- 

Python IDEs

- While there are many IDEs available for coding in Python, most popular ones are Visual Studio Code and PyCharm
- You can download either of them using the links given below
- PyCharm
 - <https://www.jetbrains.com/pycharm/download/download-thanks.html?platform=windows&code=PCC>
- Visual Studio Code
 - <https://code.visualstudio.com/download>



Writing your first code

You can write code in python in 2 ways

- Interactive Mode -- Writing code in Python CLI or Shell or Interpreter
- Script Mode -- Writing code in IDE and saving them as files with .py extension



Interactive Mode - CLI

- For interactive mode on Desktop, open command prompt and type python. You should see something like below.

```
C:\Users\nravi>python

Python 3.10.0 (tags/v3.10.0:b494f59, Oct  4 2021, 19:00:18) [MSC v.1929
64 bit (AMD64)] on win32

Type "help", "copyright", "credits" or "license" for more information.

>>> print('Hello World')

Hello World

>>>
```

Interactive Mode - Jupyter Notebook

- Jupyter is a third party Python Library which offers interactive way to work with Python using Jupyter Notebook
- We can install Jupyter using Python Package Manager (PIP)
- Files written in Python are saved as .ipynb



Interactive Mode - Jupyter Lab

- Jupyter Lab is similar but better compared to Jupyter Notebook and comes along with the Jupyter Library
- This can also be used to write Python code as .ipynb files
- Has some additional features compared to Notebook



Interactive Mode - Google Colab

- Google has extended Jupyter Lab and provided Google Colab
- All you need is a Google account to access Colab
- You can access it using the below link
 - <https://colab.research.google.com/>
- Python code written in Colab is stored in Google Drive under your account and executes on servers which are allocated dynamically by Google



Script Mode

- For script mode, open any text editor or Python IDE like Visual Studio Code or PyCharm Community Edition
- Open a new file and type the below content and save the file as helloworld.py

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



Script Mode (Cont.)

- From command prompt go to the folder where the files is stored and execute the program as follows

```
D:\Nagaraj\work\training\py_sel>python helloworld.py  
  
hello world  
  
D:\Nagaraj\work\training\py_sel>
```



Variables

- Variables in Python are memory reserved memory location in which we can store something
- So when we create a variable, we are telling the system to keep some storage aside
- How much storage to allocate is defined by the type of the variable we create



Creating Variables

- Typically in other programming languages creating a variable is a 2 step process
- First you declare a variable and then you initialise it as follows
- But in Python we have already seen that we do not need to explicitly declare and initialize our variables.
- Variable gets created and is ready to use as soon as we assign some value to it

```
name = "John"    #string variable
age = 25         #integer variable
height = 6.1     #floating point variable
print(name , " is ", age, " years of age and is a good ", height, " in
height.")
```


Data Types

- As the name suggests Data Type defines the type of the data stored in them
- Depending on the data type, the operations that can be performed and the storage mechanism varies

Text Type	str (String)
Numeric Type	int, float, complex
Sequence Type	list, tuple, range
Mapping Type	dict (dictionary)
Set Type	set

Python Strings

- String in Python or any language for that matter are a contiguous set of characters represented in quotes.
- So in a way string in Python becomes an array or a list of characters
- Characters in String can be accessed using the slicing operators [] or [:]
- String index starts from 0



Python Strings (Cont.)

```
str = 'Hello World!'

print (str)           # Prints complete string

print(len(str))       # Prints length of the string

print (str[0])        # Prints first character of the string

print (str[2:6])      # Prints characters starting from 3rd to 5th

print (str[2:])       # Prints string starting from 3rd character

print (str * 2)       # Prints string two times

print (str + " TEST") # Prints concatenated string

print(str + str)
```

Python Lists

- Lists are very versatile compound data type in Python
- List contains items wrapped in [] separated by commas
- Lists are similar to Arrays in other programming languages like C or Java
- But there is a major difference here
- There is no restriction that list should consist of same data types in Python



Python Lists (Cont.)

```
player_list = ["John", 25, 6.1]
tiny_list = ["John", 25]

print (player_list)           # Prints complete list
print (player_list[0])        # Prints first element of the list
print (player_list[1:3])      # Prints elements starting from 2nd till
3rd
print (player_list[2:])        # Prints elements starting from 3rd element
print (tiny_list * 2)          # Prints list two times
print (player_list + tiny_list) # Prints concatenated lists
```

Python Tuples

- Tuples, similar to Lists are another sequence data type
- So basically tuple also consists of a number of values seperated by commas
- Unlike lists tuples are denoted enclosed in () and not []
- Another major difference is tuples are IMMUTABLE
- Means once they are created their state cannot be changed
- So it's a kind of read-only list



Python Tuples (Cont.)

```
player_tuple = ("John", 25, 6.1)
tiny_tuple = ("John", 25)

print (player_tuple)           # Prints complete tuple
print (player_tuple[0])        # Prints first element of the tuple
print (player_tuple[1:3])      # Prints elements starting from 2nd till
3rd
print (player_tuple[2:])       # Prints elements starting from 3rd
element
print (tiny_tuple * 2)         # Prints tuple two times
print (player_tuple + tiny_tuple) # Prints concatenated tuple
```

Python Tuples (Cont.)

- Next year John grew an year older and say he has grown an inch taller too
- Lets try updating the list and tuple objects corresponding to him

```
player_list[1] = 26
player_list[2] = 6.2
print(player_list)

player_tuple[1] = 26
player_tuple[2] = 6.2
print(player_tuple)
```


Python Dictionaries

- Dictionaries in Python are like hash-table type objects, again used to store sequence of values but along with something called a key
- So dictionaries store data as key-value pairs
- If you are familiar with databases, key is something like a primary key in a table used to identify each row in a table uniquely
- There is no restriction on what types of objects you can choose as keys and values
- But usually keys are numbers or strings
- Values can be any Python object
- Dictionary stores values inside curly braces { }
- Items in Dictionary can be accessed or added using []



Python Dictionaries (Cont.)

```
player_dict = {"name" : "John", "age" : 25, "height" : 6.1}
player_age = player_dict["age"]
player_dict["game"] = "basket ball"

print(player_dict)
print(player_age)
print(player_dict["game"])
```



Data Type Conversions

- Since we are talking about data types, we should see how we can convert between data types
- In Python, we can convert from one type to the other by simply using the type name as a function and passing the value to be converted as an argument



Data Type Conversions (Cont.)

```
anint = 10  
  
afloat = float(anint)  
  
anint1 = int(afloat)  
  
astr = str(anint)  
  
print(anint)
```

```
print(afloat)  
  
print(anint1)  
  
print(astr)  
  
print(astr+astr)  
  
print(anint+anint)
```

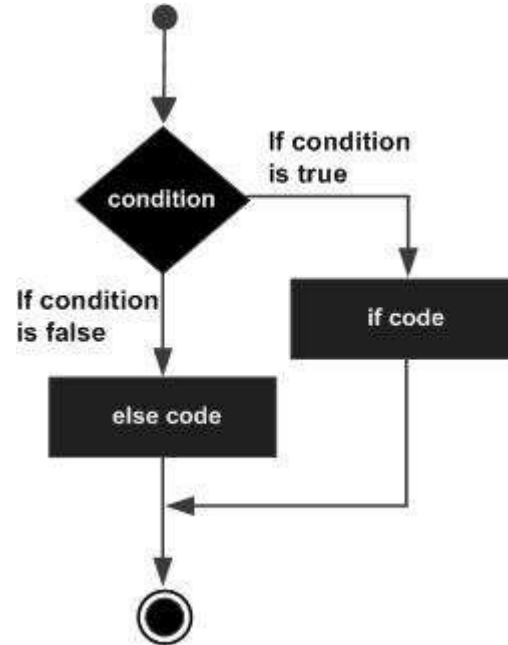
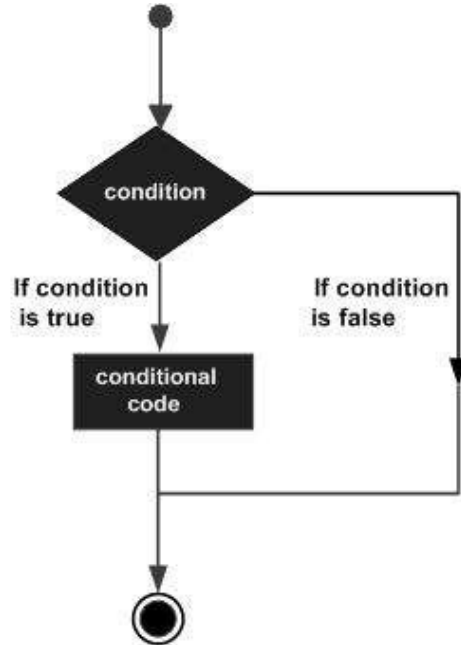


Decision Making

- Normal execution flow of the program statements is top to bottom
- But there could be situations where we need to alter this sequential flow of execution
- One such situation is Decision Making
- We would need to decide which code block to execute depending on satisfying certain conditions



Decision Making (Cont.)



Decision Making (Cont.)

- So decision making is anticipation of conditions during the program execution and taking the specified actions
- Decision structures evaluate multiple condition and produce True or False as the outcome



Decision Making in Python

- In Python any non-zero or non-null values are evaluated as True
- Zero and null values are evaluated as False
- null in Python is represented by the None
- Following types of conditional statements are available in Python
 - If
 - If else
 - If elif else



Decision Making in Python (Cont.)

- If
 - Tells the program what to do when the condition is evaluated to True

```
if expression:  
    statement 1  
    statement 2  
    ...  
    statement n
```



Decision Making in Python (Cont.)

- If else

- As a good programming practice, we should never leave a conditional statement hanging in between
- In the above code block we told the program what to do if condition is true
- But what if it is not?
- So usually there is a else following if

```
if expression:  
    statement(s)  
  
else:  
    statement(s)
```



Decision Making in Python (Cont.)

- If ... elif ... else
 - The above condition works for amount < 1000 or > 1000. So just 2 slabs
 - What if we want to have multiple discount slabs?
 - Other programming languages provide us a conditional construct called switch ... case statement
 - In Python we do not have switch case but we can simulate the same behavior using if ... elif ... else



Decision Making in Python (Cont.)

```
if expression1:  
    statement(s)  
  
elif expression2:  
    statement(s)  
  
elif expression3:  
    statement(s)  
  
else:  
    statement(s)
```



Conditionals Hands-on

- Write an if ... elif ... else block to provide a discount of:
 - 5% for amount less than 1000
 - 10% for amount less than 5000
 - 15% for any other amount
- Give an additional 5% discount if the customer is above 70 yrs
- Give additional 5% discount if the customer is a privilege customer
- Take customer age and whether privilege customer or not as input



Nested conditionals

- In some situation we would like to check for another condition after one condition is satisfied
- In such cases we can make use of nested if
- In nested if inside if ... elif ... else construct we can have one or more if ... elif ... else constructs



Nested conditionals (Cont.)

```
if expression1:  
    statement(s)  
    if expression2:  
        statement(s)  
    else  
        statement(s)  
else:  
    statement(s)
```

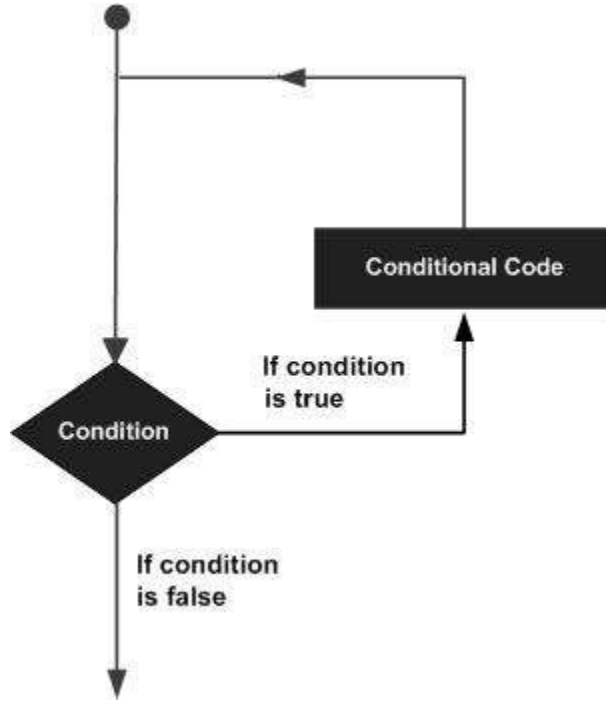


Python Loops

- As we have already seen program statements are executed in sequence in the absence of any other interruption
- In some situations we would like to execute a block of code multiple times based on a condition or for a specified number of times
- We can use loops in such cases
- The below diagram shows use of loops

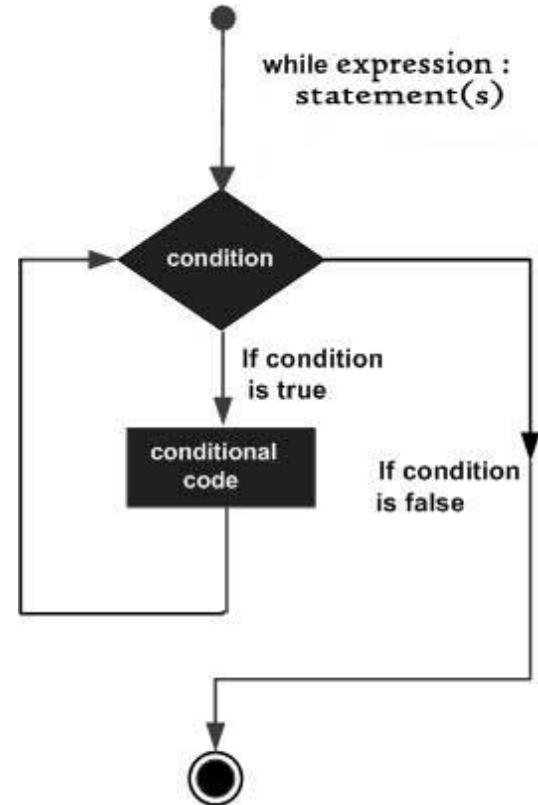


Python Loops (Cont.)



Python Loops - While Loop

- The condition is checked for each iteration and statements are executed
- This goes on and on till the condition becomes false
- The while loop runs forever when the condition never evaluates to False which is termed as an infinite loop
- So we must be cautious while using while loop to choose the condition carefully



While Loop Syntax

```
while condition:  
    statement(s)  
    terminating logic
```

```
num = int(input("Enter a number:"))  
  
while num > 0 :  
    print(num)  
    num -= 1
```



Else Statement in While Loop

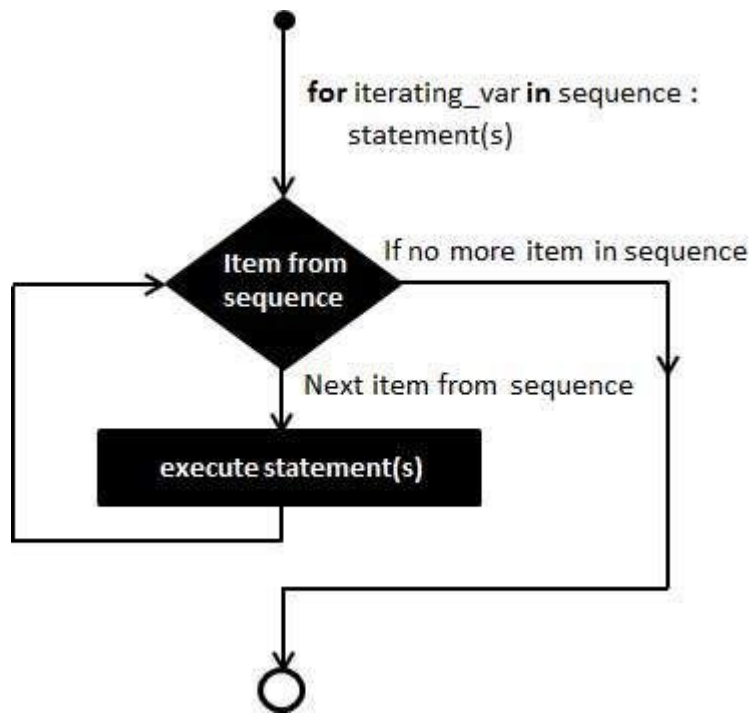
- Interesting thing with Python is that you can also use else statement with while loop unlike other languages
- Statements mentioned in else part of the while loop are executed when the condition becomes false



Python Loops - For Loop

- Very useful and most used
- Useful to iterate over the elements in a sequence like list or string

```
for iterating_var in sequence:  
    statements(s)
```



Types of For Loop

- We can use for loop for iterating through a set of statements certain number of times
- Say we want to execute a statement for 10 times, we can create a range of values upto 10 and iterate over the range
- We can also use for loop to iterate over iterables like lists, dictionaries
- Check the hands-on examples for more



Loop Control Statements

- At times we might need to have some control on how the execution happens when going in a loop
- This can be done using Loop Control statements available in Python
- They are ***break*** and ***continue***
- Break
 - Check for a condition and when the condition is met, break the loop
- Continue
 - check for the condition and when the condition is met, skip the current iteration and go to the next iteration



Break - Hands-on

- Create a list of 10 numbers chosen randomly. Using for loop write code to check if the list contains an even number or not.

```
numbers = [11,33,55,38,55,75,37,21,23,41,13]
for number in numbers:
    if number%2 == 0:
        print("The given list contains an even number")
        break
else:
    print("The given list does not contain an even
number")
```


Continue - Hands-on

- From the given list of numbers, extract all the even numbers into another list

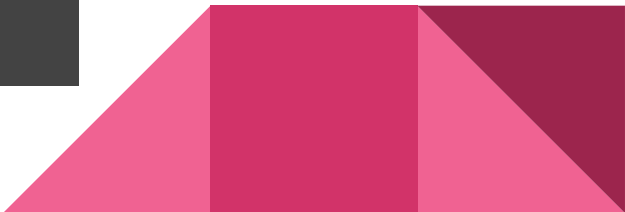
```
numbers = [11,33,55,39,55,28,75,37,21,16,23,41,90,13]
even_list = []
for number in numbers:
    if number%2 != 0:
        continue
    even_list.append(number)
print(even_list)
```



Pass Statement

- When we want the interpreter to not do anything we can use ***pass*** statement

```
numbers = [11,33,55,39,55,28,75,37,21,16,23,41,90,13]
even_list = []
for number in numbers:
    if number%2 != 0:
        pass
    even_list.append(number)
print(even_list)
```



Python Functions

- We use functions to define a block of organized and reusable code that perform related actions
- They provide modularity and make code reusable
- We have been using functions in built in Python for a while now, but we are now talking about user defined functions
- Functions start with a key word `def` followed by a name and `()`
- We pass any arguments to the function within the `()`

```
def printme(str):  
    print(str)
```

Functions With Arguments

- The data we pass to the functions are called as arguments
- If we define a function with arguments, we must pass them while calling the function
- Arguments passed to the function are processed in the order in which they are passed

```
def print_details(x, y):  
    print('Name is ', x)  
    print('Age is ', y)  
print_details('John', 50)
```

Functions With Keyword Arguments

- In case we need to pass the arguments in different order than they are defined we can pass them as keyword arguments

```
def print_details(x, y):  
    print('Name is ', x)  
    print('Age is ', y)  
print_details(y=50, x='John')
```

Functions With Default Arguments

- In some cases we can define functions with default arguments
- If an argument is defined as default and if it is not passed while calling the function, the default value will be used

```
def print_details(x, y=50):  
    print('Name is ', x)  
    print('Age is ', y)  
print_details('John')
```

Functions With Variable Number of Arguments

- In some cases the number of arguments passed to the function is not known initially, as they are passed dynamically
- To handle such cases, we use functions with variable number of arguments
- In the function definition variable arguments are prefixed with a '*'
- Python create a tuple with the given name and stores all the arguments passed while calling the function

```
def print_details(*args):  
    print('Name is ', x)  
    print('Age is ', y)  
print_details('John')
```

Higher Order Functions

- A function that accepts another function as a parameter
- A function that returns another function
- Properties
 - A function is an instance of Object type
 - Functions can be stored in a variables
 - Functions can be stored in other data structures like lists
 - Functions accept other functions as parameter
 - Functions return other functions



Lambda Functions

- At times we need nameless functions that live for a short while
- In such cases we can make use of Lambda functions
- Syntax of a lambda function is as follows
 - `lambda arguments: expression`
- In functional programming using higher order functions, usually lambda functions are used to keep the code concise and readable




Functional Programming

- Imperative programming (the usual programming we do) is done through statements
- These statements change the value of variables and hence change the state of the program or system
- E.g. Loops in which for each iteration the variables and hence the state changes



Functional Programming (Cont.)

- In contrast, Functional programming deals with functions to define computation
 - Which means we do computation through functions that accept other functions as parameter or return other functions
 - In short we deal extensively with higher-order functions
 - As we know Python is not a fully functional programming language
 - But we can achieve functional programming when needed
- 

Functional Programming (Cont.)

- Three very useful functions that help us achieve functional programming are
 - `map()`
 - `filter()`
 - `reduce()`
- These functions accept a function and an iterable as arguments
- They return the result of applying the parameter function to each element of the iterable



Python Modules

- A way to organize and reuse code across different programs or files
- Simply put a Python module is a piece of Python code.
- So it can contain functions, classes or variable or simply runnable code
- A module called `my_calc` is stored in a file called `my_calc.py` and is imported using an import statement
- But for the interpreter to find this file when some other file tries to import it, the module has to be available in interpreter's search path



Python Modules (Cont.)

- To check where does Python look for modules to be imported we can use a built in module called `sys`
 - `import sys`
 - `print(sys.path)`
- If we write any custom modules, it should be placed in one of the locations shown in `sys.path` for the module to be available for import
- For now we can place custom modules in the current working directory



Python Modules (Cont.)

- Python modules can be imported in any of the following ways
 - `import module`
 - Variables and methods from the module will be imported
 - Can be used as `module.variable`, `module.method`
 - `import modules as m`
 - Variables and methods from the module will be imported
 - Can be used as `m.variable`, `m.method`
 - `from module import variable`
 - Directly variable can be used
 - `from module import method`
 - Directly method can be used
 - `from module import *` (not recommended)



Python Packages

- In the previous slides we saw that Python modules is nothing but a Python file which can be imported and used by another python file
- So Python module is equivalent to a file in Windows Explorer
- Python package is equivalent to a directory in Windows Explorer
- To differentiate a normal folder from Python package, it must contain a file named `__init__.py`
- This file can be left blank



Python Packages (Cont.)

- Just like directories can have sub directories, package can have sub packages in them
- Importing packages is similar to importing modules except that in case of packages we need to provide the path like
 - `package.subpackage(s).module`
 - E.g. `game`.




Python DB Operations - SQLite3

- While Python can work with many database systems, the underlying concepts remain the same
- As an example, we can explore Python integration with SQLite and MySQL
- Download SQLite from
 - <https://www.sqlite.org/download.html>
- Download DB browser for SQLite form
 - <https://sqlitebrowser.org/dl/>
-



Python OOP

- OOP or Object Oriented Programming is a programming paradigm centered around objects
 - That is we try to model the problem we are trying to solve or the system we are trying to build in terms of real life objects
 - An object has
 - A set of attributes or properties
 - A behavior
 - Python follows OOP approach to create modular and reusable code by following DRY (Don't Repeat Yourself) principle.
- 

Python OOP (Cont.)

- Class is like a blueprint which defines the generic structure of objects
- Class has a constructor or initializer which is used to create different instances of the class called Objects
- Objects are instance of the class
- Objects have attributes which define their state and methods which define their behavior



Python Classes And Objects

- Class is created using the keyword ***class***
- Class consists of an init method which is called when an object is created from the class
- Since init method is internal and should not be explicitly used it is prefixed and suffixed with double underscores (___)



Python Classes And Objects (Cont.)

- Once the class is defined, it can be used to create instances or objects
- Each object created has:
 - An identity - the name of the object
 - A state - the attributes or properties
 - A behavior - the methods



Working with Objects

- We can instantiate a Python class or create instances or objects from a Python class as follows
 - `objName = ClassName(<any initial values for instance variables>)`
 - E.g.
 - `person1 = Person("John", "Doe", 45)`
- We can delete an instance attribute or the instance itself using the `del` statement
 - `del person1.firstName`
 - `del person1`



Builtin Class Methods

- Python has the following class methods to modify instance attributes

SN	Function	Description
1	getattr(obj,name,default)	It is used to access the attribute of the object.
2	setattr(obj, name,value)	It is used to set a particular value to the specific attribute of an object.
3	delattr(obj, name)	It is used to delete a specific attribute.
4	hasattr(obj, name)	It returns true if the object contains some specific attribute.

Builtin Class Attributes

- Python has the following class attributes defined

SN	Attribute	Description
1	<code>__dict__</code>	It provides the dictionary containing the information about the class namespace.
2	<code>__doc__</code>	It contains a string which has the class documentation
3	<code>__class__</code>	It is used to access the class name.
4	<code>__module__</code>	It is used to access the module in which, this class is defined.
5	<code>__bases__</code>	It contains a tuple including all base classes.

Inheritance

- Like other OOP languages, Python supports inheritance
- Subclass inherits attributes and methods from super class
- Python also supports multiple as well as multi-level inheritance
- Python cleverly circumvents the diamond problem arising due to multiple inheritance in Java
- For this, Python follows a concept called MRO (method resolution order)
- We can check MRO of a class using the following command
 - `BaseClass.mro()`



Class and Object Relationships

- We have built-in methods to check sub class/ super class and object/ class relationships
- The ***issubclass(sub, sup)*** method can be used to check if ***sub*** is a subclass of ***sup***
- The ***isinstance(obj, class)*** method can be used to check if ***obj*** is an instance of ***class***



Polymorphism


- In OOP polymorphism is achieved in two ways
 - Method overriding
 - Operator overloading
- When subclass defines same method defined in parent class, we say that the subclass overrides the method of the parent class
- When same operator is used on different type of objects where the operation performed is based on the type of object, we call it as operator overloading



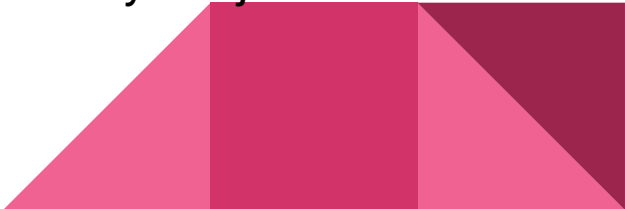
Encapsulation

- Encapsulation is all about how to handle protected and private members of a class
- Python achieves encapsulation using the following convention:
 - Protected members are prefixed with “_” (underscore)
 - Private members are prefixed with “__” (double underscore)
- Protected members are accessible only inside the class and to its subclasses
- Though this is not enforced in Python, as a convention, any instance variable or method prefixed with “_” is not supposed to be accessed directly

Encapsulation (Cont.)

- Similar to protected members, private members of the class are prefixed with “__” (double underscore)
 - These are not straight away accessible outside the class except through object methods
 - However Python is not very strict in implementing it, so we can access them in some other means
 - Again as a convention, any method or variable prefixed with “__” should be treated as private and handled accordingly
- 

Encapsulation (Cont.)

- So the recommended way to get and set the protected and private values are using getter and setter methods
 - But if we had made certain variables as protected or private at a later stage as an afterthought, refactoring the code using getter and setter methods is going to be a huge task and can even break the code
 - To get around this issue, Python provides a built-in decorator called ***property***
 - Property decorator enables us to access methods as if they are just properties
- 

Abstraction

- In Python, Abstraction is achieved using a module called ***abc***
- Module ***abc*** consists of classes and methods for achieving abstraction
 - ABC (Abstract Base Class)
 - ABCMeta (ABC Meta class)
 - abstractmethod (Method which can be used as decorator)
- If a subclass extends an abstract base class, it has to implement the abstract methods of the base class
- Else Python throws error

