Programming in Python with Full Stack Development (303105257)

Anand Jawdekar

Assistant Professor CSE Department

DIGITAL LEARNING CONTENT

Unit-2

Functions

Unit-2

Functions :

Defining and using functions, including the use of arguments and return values

OOPS Concepts :

Object, class, abstraction, encapsulation, polymorphism, Inheritence.

Exceptions and File handling:

Handling exceptions and working with files

Defining and using functions

A Python function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.

A top-to-down approach towards building the processing logic involves defining blocks of independent reusable functions. A Python function may be invoked from any other function by passing required data (called parameters or arguments). The called function returns its result back to the calling environment.

Types of Python Functions

Sr.No	Type & Description
1	**Built-in functions** Python's standard library includes number of built-in functions. Some of Python's built-in functions are print(), int(), len(), sum(), etc. These functions are always available, as they are loaded into computer's memory as soon as you start Python interpreter.
2	**Functions defined in built-in modules** The standard library also bundles a number of modules. Each module defines a group of functions. These functions are not readily available. You need to import them into the memory from their respective modules.
3	**User-defined functions** In addition to the built-in functions and functions in the built-in modules, you can also create your own functions. These functions are called user-defined functions.

Defining a Python Function

You can define custom functions to provide the required functionality. Here are simple rules to define a function in Python −

Function blocks begin with the keyword def followed by the function name and parentheses ().

Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.

The first statement of a function can be an optional statement; the documentation string of the function or docstring.

The code block within every function starts with a colon (:) and is indented.

The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

Syntax to Define a Python Function

```
def function_name( parameters ):
   "function_docstring"
   function_suite return
   [expression]
```

By default, parameters have a positional behavior and you need to inform them in the same order that they were defined.
Once the function is defined, you can execute it by calling it from another function or directly from the Python prompt.

Example to Define a Python Function

The following example shows how to define a function greetings(). The bracket is empty so there aren't any parameters. Here, the first line is a docstring and the function block ends with return statement.

```
def greetings():
   "This is docstring of greetings function"
   print ("Hello World")
   return
```

When this function is called, **Hello world** message will be printed.

Calling a Python Function

Defining a function only gives it a name, specifies the parameters that are to be included in the function and structures the blocks of code. Once the basic structure of a function is finalized, you can call it by using the function name itself. If the function requires any parameters, they should be passed within parentheses. If the function doesn't require any parameters, the parentheses should be left empty.

Example to Call a Python Function

```
# Function definition is here
def printme( str ):
   "This prints a passed string into this function“
   print (str)
   return; # Now you can call the function
printme("I'm first call to user defined function!")
printme("Again second call to the same function")
```

When the above code is executed, it produces the following output −

```
I'm first call to user defined function! Again second call to the same function
```

Pass by Reference vs Value

In programming languages like C and C++, there are two main ways to pass variables to a function, which are Call by Value and Call by Reference (also known as pass by reference and pass by value). However, the way we pass variables to functions in Python differs from others.

call by value – When a variable is passed to a function while calling, the value of actual arguments is copied to the variables representing the formal arguments. Thus, any changes in formal arguments does not get reflected in the actual argument. This way of passing variable is known as call by value.

call by reference – In this way of passing variable, a reference to the object in memory is passed. Both the formal arguments and the actual arguments (variables in the calling code) refer to the same object. Hence, any changes in formal arguments does get reflected in the actual argument.

Python uses pass by reference mechanism. As variable in Python is a label or reference to the object in the memory, both the variables used as actual argument as well as formal arguments really refer to the same object in the memory. We can verify this fact by checking the id() of the passed variable before and after passing.

Cont..

```
def testfunction(arg):
   print ("ID inside the function:", id(arg))
var = "Hello" print ("ID before passing:", id(var))
testfunction(var)
```

If the above code is executed, the id() before passing and inside the function will be displayed.

```
ID before passing: 1996838294128 ID
inside the function: 1996838294128
```

Types of Python Function Arguments

Based on how the arguments are declared while defining a Python function, they are classified into the following categories −

- Default arguments
- Keyword arguments
- Required arguments
- Variable-length arguments

Default Arguments

A default argument is a parameter that assumes a default value if a value is not provided in the function call for that argument. The following example illustrates Default arguments to write functions in Python.

```
# Python code to demonstrate the use of default arguments
# defining a function
def function( n1, n2 = 20 ):
   print("number 1 is: ", n1)
   print("number 2 is: ", n2)
# Calling the function and passing only one argument
print( "Passing only one argument" )
function(30)
# Now giving two arguments to the function
print( "Passing two arguments" )
function(50,30)
```

Output:

```
Passing only one argument number 1 is: 30 number 2 is: 20
Passing two arguments number 1 is: 50 number 2 is: 30
```

Keyword Arguments

The idea is to allow the caller to specify the argument name with values so that the caller does not need to remember the order of parameters.

```
# Python program to demonstrate Keyword Arguments
def student(firstname, lastname):
    print(firstname, lastname)
# Keyword arguments
student(firstname=‘Hello', lastname='Pyhton’)
```

Output:

```
Hello Python
```

Positional Arguments

We used the Position argument during the function call so that the first argument (or value) is assigned to name and the second argument (or value) is assigned to age. By changing the position, or if you forget the order of the positions, the values can be used in the wrong places, as shown in the Case-2 example below, where 27 is assigned to the name and Suraj is assigned to the age.

```
def nameAge(name, age):
    print("Hi, I am", name)
    print("My age is ", age)
# You will get correct output because
# argument is given in order
print("Case-1:")
nameAge("Suraj", 27)
# You will get incorrect output because
# argument is not in order
print("\nCase-2:")
nameAge(27, "Suraj")
```

Calling Function With Keyword Arguments

Output:

Case-1:

```
Hi, I am Suraj
My age is 27
```

Case-2:

```
Hi, I am 27
My age is Suraj
```

Arbitrary Keyword Arguments

You may want to define a function that is able to accept **arbitrary** or **variable number of arguments**.
Moreover, the arbitrary number of arguments might be positional or keyword arguments.
An argument prefixed with a single asterisk * for arbitrary positional arguments.
An argument prefixed with two asterisks ** for arbitrary keyword arguments.

```
# sum of numbers
def add(*args):
   s=0
   for x in args:
      s=s+x
   return s
result = add(10,20,30,40)
print (result)
result = add(1,2,3)
print (result)
Output: 100 6
```

Example Arbitrary Keyword Arguments (**kwargs)

```
def addr(**kwargs):
  for k,v in kwargs.items():
    print ("{}:{}".format(k,v))

print ("pass two keyword args")
addr(Name="John", City="Mumbai")
print ("pass four keyword args")

# pass four keyword args
addr(Name="Raam", City="Mumbai", ph_no="9123134567", PIN="400001")
```

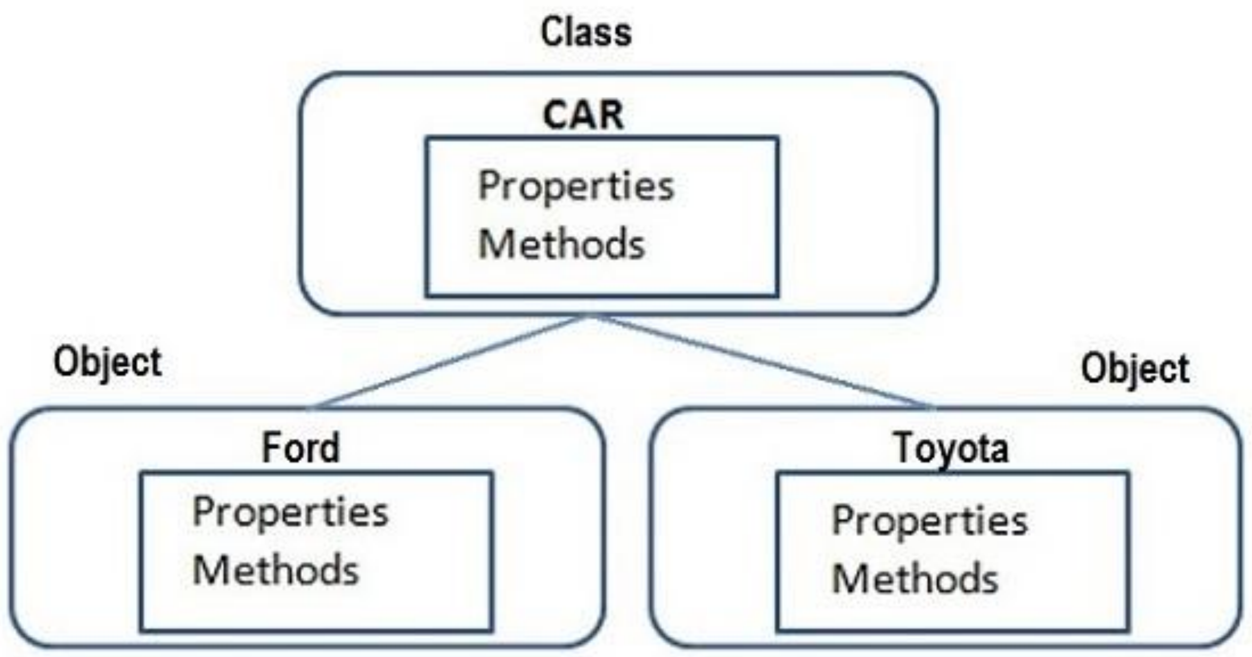
Output:

```
pass two keyword args
Name:John
City:Mumbai
pass four keyword args Name:Raam City:Mumbai ph_no:9123134567 PIN:400001
```

Python - OOP Concepts

OOP is an abbreviation that stands for **Object-oriented programming** paradigm. It is defined as a programming model that uses the concept of **objects** which refers to real-world entities with state and behavior. Python is a programming language that supports object-oriented programming. This makes it simple to create and use classes and objects.
In the real world, we deal with and process objects, such as student, employee, invoice, car, etc. Objects are not only data and not only functions, but combination of both. Each real-world object has attributes and behavior associated with it.

Python - OOP Concepts



Python - OOP Concepts

Attributes

- Name, class, subjects, marks, etc., of student
- Name, designation, department, salary, etc., of employee
- Invoice number, customer, product code and name, price and quantity, etc., in an invoice
- Registration number, owner, company, brand, horsepower, speed, etc., of car

Each attribute will have a value associated with it. Attribute is equivalent to data.

Behavior

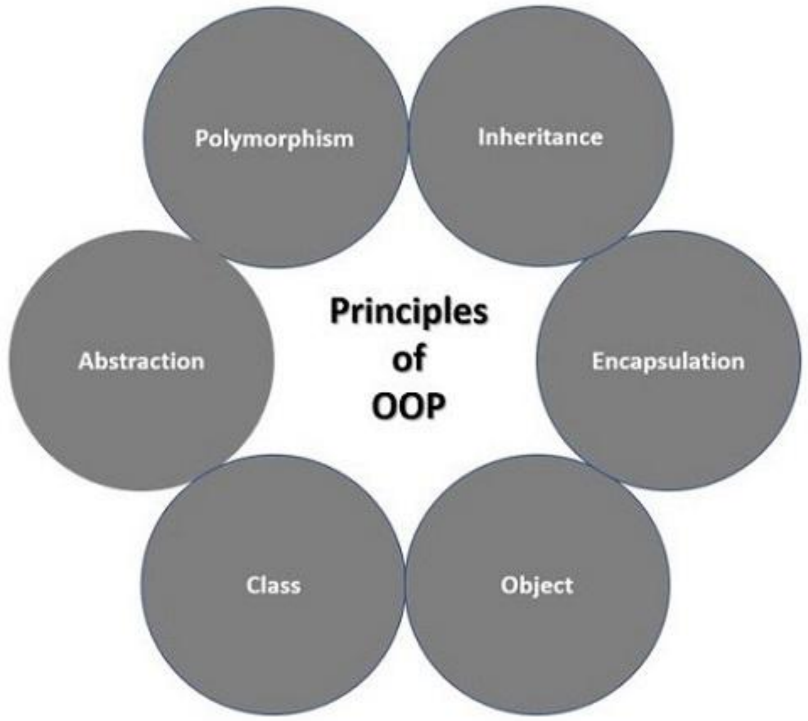
- Processing attributes associated with an object.
- Compute percentage of student's marks
- Calculate incentives payable to employee
- Apply GST to invoice value
- Measure speed of car

Behavior is equivalent to function. In real life, attributes and behavior are not independent of each other, rather they co-exist.

Principles of OOPs Concepts

Object-oriented programming paradigm is characterized by the following principles –

- Class
- Object
- Encapsulation
- Inheritance
- Polymorphism



Class & Object

A **class** is an user-defined prototype for an object that defines a set of attributes that characterize any object of the class. The attributes are data members (class variables and instance variables) and methods, accessed via dot notation.

An **object** refers to an instance of a certain class. For example, an object named **obj** that belongs to a class **Circle** is an instance of that class. A unique instance of a data structure that is defined by its class. An object comprises both data members (class variables and instance variables) and methods

Example

```
# defining class
class Smartphone:
   # constructor
   def __init__(self, device, brand):
      self.device = device
      self.brand = brand
   # method of the class
   def description(self):
      return f"{self.device} of {self.brand} supports Android 14"
# creating object of the class
phoneObj = Smartphone("Smartphone", "Samsung")
print(phoneObj.description())
```

Output:

```
Smartphone of Samsung supports Android 14
```

Encapsulation

Data members of class are available for processing to functions defined within the class only. Functions of class on the other hand are accessible from outside class context. So object data is hidden from environment that is external to class. Class function (also called method) encapsulates object data so that unwarranted access to it is prevented.

Example

```
class Desktop:
  def __init__(self):
    self.__max_price = 25000
  def sell(self):
    return f"Selling Price: {self.__max_price}"
  def set_max_price(self, price):
    if price > self.__max_price:
      self.__max_price = price
# Object
desktopObj = Desktop()
print(desktopObj.sell())
# modifying the price directly
desktopObj.__max_price = 35000
print(desktopObj.sell())
# modifying the price using setter function
desktopObj.set_max_price(35000)
print(desktopObj.sell())
Output: Selling Price: 25000 Selling Price: 25000 Selling Price: 35000
```

Inheritance

A software modelling approach of OOP enables extending capability of an existing class to build new class instead of building from scratch. In OOP terminology, existing class is called **base or parent class**, while new class is called **child or sub class**.

Child class inherits data definitions and methods from parent class. This facilitates reuse of features already available. Child class can add few more definitions or redefine a base class function.

Syntax

Derived classes are declared much like their parent class; however, a list of base classes to inherit from is given after the class name −

```
class SubClassName (ParentClass1[, ParentClass2, ...]):
    'Optional class documentation string'
   class_suite
```

Example

```
class Animal:
    def speak(self):
        print("Animal Speaking")
#child class Dog inherits the base class Animal
class Dog(Animal):
    def bark(self):
        print("dog barking")
d = Dog()
d.bark()
d.speak()
```

Output:

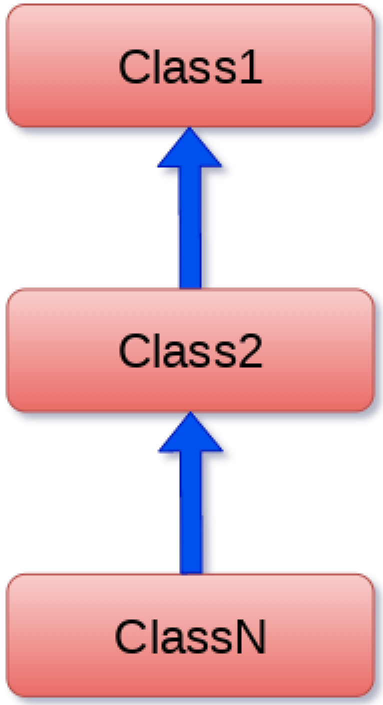
```
dog barking
Animal Speaking
```

Python Multi-Level inheritance

Multi-Level inheritance is possible in python like other object-oriented languages. Multi-level inheritance is archived when a derived class inherits another derived class. There is no limit on the number of levels up to which, the multi-level inheritance is archived in python.

Syntax

```
class class1:
    <class-suite>
class class2(class1):
    <class suite>
class class3(class2):
    <class suite
```



Example

```
class Animal:
    def speak(self):
        print("Animal Speaking")
#The child class Dog inherits the base class Animal
class Dog(Animal):
    def bark(self):
        print("dog barking")
#The child class Dogchild inherits another child class Dog
class DogChild(Dog):
    def eat(self):
        print("Eating bread...")
d = DogChild()
d.bark()
d.speak()
d.eat()
```

Output:

```
dog barking Animal Speaking Eating bread...
```

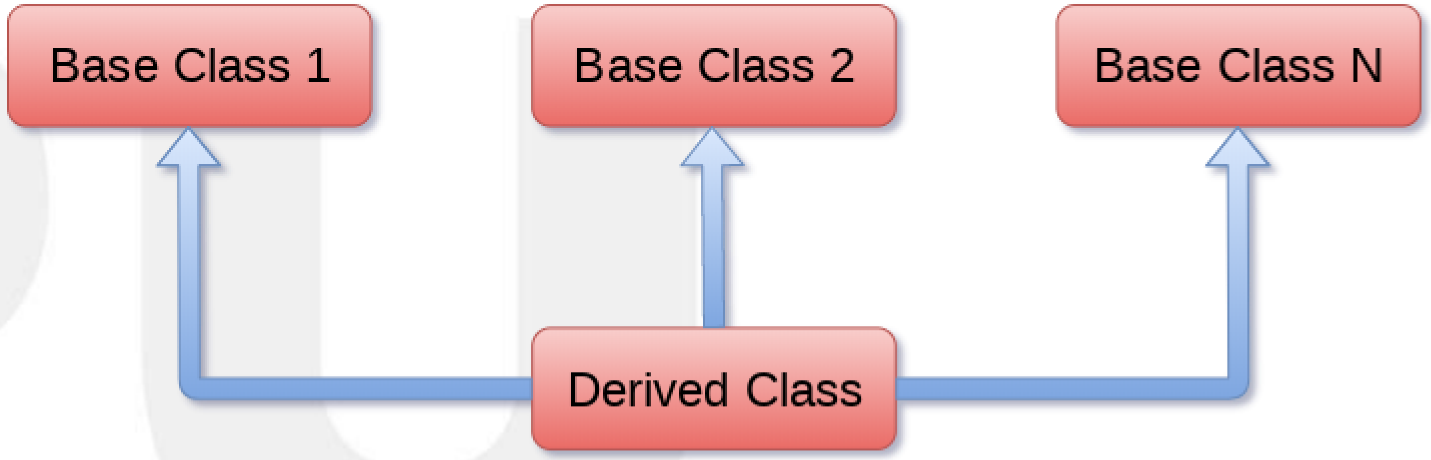
Python Multiple inheritance

Python provides us the flexibility to inherit multiple base classes in the child class.

```
class Base1:
    <class-suite>

class Base2:
    <class-suite>
.
.
.
class BaseN:
    <class-suite>

class Derived(Base1, Base2, ...... BaseN):
    <class-suite>
```



Example

```
class Calculation1:
    def Summation(self,a,b):
        return a+b;
class Calculation2:
    def Multiplication(self,a,b):
        return a*b;
class Derived(Calculation1,Calculation2):
    def Divide(self,a,b):
        return a/b;
d = Derived()
print(d.Summation(10,20))
print(d.Multiplication(10,20))
print(d.Divide(10,20))
```

Output:

```
30 200 0.5
```

Polymorphism

Polymorphism is a Greek word meaning having multiple forms. In OOP, polymorphism occurs when each sub class provides its own implementation of an abstract method in base class.
You can always override your parent class methods. One reason for overriding parent's methods is because you may want special or different functionality in your subclass.

Example

```
# define parent class
class Parent:
   def myMethod(self):
      print ("Calling parent method")

# define child class
class Child(Parent):
   def myMethod(self):
      print ("Calling child method")

# instance of child
c = Child()
# child calls overridden method
c.myMethod()
Output: Calling child method
```

Data abstraction in python

Abstraction is an important aspect of object-oriented programming. In python, we can also perform data hiding by adding the double underscore (___) as a prefix to the attribute which is to be hidden. After this, the attribute will not be visible outside of the class through the object.

Example

```
class Employee:
    __count = 0;
    def __init__(self):
        Employee.__count = Employee.__count+1
    def display(self):
        print("The number of employees",Employee.__count)
emp = Employee()
emp2 = Employee()
try:
    print(emp.__count)
finally:
    emp.display()
```

Output:

```
The number of employees 2 AttributeError: 'Employee' object has no attribute '__count'
```

Abstraction in Python

Abstraction is used to hide the internal functionality of the function from the users. The users only interact with the basic implementation of the function, but inner working is hidden. User is familiar with that **"what function does"** but they don't know **"how it does."**

In simple words, we all use the smartphone and very much familiar with its functions such as camera, voice-recorder, call-dialing, etc., but we don't know how these operations are happening in the background. Let's take another example - When we use the TV remote to increase the volume. We don't know how pressing a key increases the volume of the TV. We only know to press the "+" button to increase the volume.

Why Abstraction is Important?

In Python, an abstraction is used to hide the irrelevant data/class in order to reduce the complexity. It also enhances the application efficiency.

Abstraction classes in Python

A class that consists of one or more abstract method is called the abstract class. Abstract methods do not contain their implementation. Abstract class can be inherited by the subclass and abstract method gets its definition in the subclass. Abstraction classes are meant to be the blueprint of the other class. An abstract class can be useful when we are designing large functions. An abstract class is also helpful to provide the standard interface for different implementations of components. Python provides the **abc** module to use the abstraction in the Python program. Let's see the following syntax.

```
from abc import ABC
class ClassName(ABC):
```

We import the ABC class from the **abc** module.

Working of the Abstract Classes

Unlike the other high-level language, Python doesn't provide the abstract class itself. We need to import the abc module, which provides the base for defining Abstract Base classes (ABC). The ABC works by decorating methods of the base class as abstract. It registers concrete classes as the implementation of the abstract base. We use the ***@abstractmethod*** decorator to define an abstract method or if we don't provide the definition to the method, it automatically becomes the abstract method. Let's understand the following example.

Example -

```
# Python program demonstrate
# abstract base class work
from abc import ABC, abstractmethod
class Car(ABC):
    def mileage(self):
        pass
class Tesla(Car):
    def mileage(self):
        print("The mileage is 30kmph")
class Suzuki(Car):
    def mileage(self):
        print("The mileage is 25kmph ")
class Duster(Car):
     def mileage(self):
          print("The mileage is 24kmph ")
class Renault(Car):
    def mileage(self):
            print("The mileage is 27kmph ")
```

Example

```
# Driver code
t= Tesla ()
t.mileage()

r = Renault()
r.mileage()

s = Suzuki()
s.mileage()
d = Duster()
d.mileage()
The mileage is 30kmph The mileage is 27kmph The mileage is 25kmph The mileage is 24kmph
```

Python Exceptions

An exception in Python is an incident that happens while executing a program that causes the regular course of the program's commands to be disrupted. When a Python code comes across a condition it can't handle, it raises an exception. An object in Python that describes an error is called an exception.

When a Python code throws an exception, it has two options: handle the exception immediately or stop and quit.

Exceptions versus Syntax Errors

When the interpreter identifies a statement that has an error, syntax errors occur.

Example

```
#Python code after removing the syntax error
string = "Python Exceptions"
for s in string:
    if (s != o:
        print( s )
```

Output:

```
if (s != o: ^ SyntaxError: invalid syntax
```

The arrow in the output shows where the interpreter encountered a syntactic error. There was one unclosed bracket in this case. Close it and rerun the program:

```
#Python code after removing the syntax error
string = "Python Exceptions"
for s in string:
    if (s != o):
        print( s )
```

Example

Output:

```
2 string = "Python Exceptions"
4 for s in string: ----> 5 if (s != o):
6 print( s ) NameError: name 'o' is not defined
```

We encountered an exception error after executing this code. When syntactically valid Python code produces an error, this is the kind of error that arises. The output's last line specified the name of the exception error code encountered. Instead of displaying just "exception error", Python displays information about the sort of exception error that occurred. It was a NameError in this situation. Python includes several built-in exceptions. However, Python offers the facility to construct custom exceptions.

Try and Except Statement - Catching Exceptions

In Python, we catch exceptions and handle them using try and except code blocks. The try clause contains the code that can raise an exception, while the except clause contains the code lines that handle the exception. Let's see if we can access the index from the array, which is more than the array's length, and handle the resulting exception.

Example

```
# Python code to catch an exception and handle it using try and except code blocks
a = ["Python", "Exceptions", "try and except"]
try:
    #looping through the elements of the array a, choosing a range that goes beyond the length of the array
     for i in range( 4 ):
        print( "The index and element from the array is", i, a[i] )
#if an error occurs in the try block, then except block will be executed by the Python interpreter
except:
    print ("Index out of range")
```

Output:

```
The index and element from the array is 0
Python The index and element from the array is 1 Exceptions
The index and element from the array is 2 try and except
Index out of range
```

Explanation of code

The code blocks that potentially produce an error are inserted inside the try clause in the preceding example. The value of i greater than 2 attempts to access the list's item beyond its length, which is not present, resulting in an exception. The except clause then catches this exception and executes code without stopping it.

How to Raise an Exception

If a condition does not meet our criteria but is correct according to the Python interpreter, we can intentionally raise an exception using the raise keyword. We can use a customized exception in conjunction with the statement.

If we wish to use raise to generate an exception when a given condition happens, we may do so as follows:

```
#Python code to show how to raise an exception in Python
num = [3, 4, 5, 7]
if len(num) > 3:
    raise Exception( f"Length of the given list must be less than or equal to 3 but is {len(num)}" )
```

Output:

```
1 num = [3, 4, 5, 7] 2 if len(num) > 3: ----> 3 raise Exception( f"Length of the given list must be less than or equal to 3 but is {len(num)}" ) Exception: Length of the given list must be less than or equal to 3 but is 4
```

Assertions in Python

When we're finished verifying the program, an assertion is a consistency test that we can switch on or off. The simplest way to understand an assertion is to compare it with an if-then condition. An exception is thrown if the outcome is false when an expression is evaluated.
Assertions are made via the assert statement, which was added in Python 1.5 as the latest keyword.
Assertions are commonly used at the beginning of a function to inspect for valid input and at the end of calling the function to inspect for valid output.

The assert Statement

Python examines the adjacent expression, preferably true when it finds an assert statement. Python throws an AssertionError exception if the result of the expression is false.

The syntax for the assert clause is –

assert Expressions[, Argument]

Example

```
#Python program to show how to use assert keyword
# defining a function
def square_root( Number ):
    assert ( Number < 0), "Give a positive integer"
    return Number**(1/2)

#Calling function and passing the values
print( square_root( 36 ) )
print( square_root( -36 ) )
```

Output:

```
7 #Calling function and passing the values ----> 8 print( square_root( 36 ) ) 9 print( square_root( -36 )
) Input In [23], in square_root(Number) 3 def square_root( Number ): ----> 4 assert ( Number < 0),
"Give a positive integer" 5 return Number**(1/2) AssertionError: Give a positive integer
```

Try with Else Clause

Python also supports the else clause, which should come after every except clause, in the try, and except blocks. Only when the try clause fails to throw an exception the Python interpreter goes on to the else block.

Here is an instance of a try clause with an else clause.

Example

```
# Python program to show how to use else clause with try and except clauses

# Defining a function which returns reciprocal of a number
def reciprocal( num1 ):
    try:
        reci = 1 / num1
    except ZeroDivisionError:
        print( "We cannot divide by zero" )
    else:
        print ( reci )
# Calling the function and passing values
reciprocal( 4 )
reciprocal( 0 )
```

Output:

```
0.25 We cannot divide by zero
```

Finally Keyword in Python

The finally keyword is available in Python, and it is always used after the try-except block. The finally code block is always executed after the try block has terminated normally or after the try block has terminated for some other reason.
Here is an example of finally keyword with try-except clauses:

Example

```
# Python code to show the use of finally clause

# Raising an exception in try block
try:
   div = 4 // 0
   print( div )
# this block will handle the exception raised
except ZeroDivisionError:
   print( "Atepting to divide by zero" )
# this will always be executed no matter exception is raised or not
finally:
   print( 'This is code of finally clause' )
```

Output:

```
Atepting to divide by zero This is code of finally clause
```

Python - File Handling

File handling in Python involves interacting with files on your computer to read data from them or write data to them. Python provides several built-in functions and methods for creating, opening, reading, writing, and closing files.
Opening a File in Python
To perform any file operation, the first step is to open the file. Python's built-in open() function is used to open files in various modes, such as reading, writing, and appending. The syntax for opening a file in Python is −
file = open("filename", "mode")
Where, **filename** is the name of the file to open and **mode** is the mode in which the file is opened (e.g., 'r' for reading, 'w' for writing, 'a' for appending).

Parul® University

DIGITAL LEARNING CONTENT

File Opening Modes

Sr. No.	Modes & Description
1	**r** Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode.
2	**rb** Opens a file for reading only in binary format. The file pointer is placed at the beginning of the file. This is the default mode.
3	**r+** Opens a file for both reading and writing. The file pointer placed at the beginning of the file.
4	**rb+** Opens a file for both reading and writing in binary format. The file pointer placed at the beginning of the file.
5	**w** Opens a file for writing only. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.

Example

```
# Opening a file in read mode
file = open("example.txt", "r")

# Opening a file in write mode
file = open("example.txt", "w")

# Opening a file in append mode
file = open("example.txt", "a")

# Opening a file in binary read mode
file = open("example.txt", "rb")
```

Reading a File in Python

Reading a file in Python involves opening the file in a mode that allows for reading, and then using various methods to extract the data from the file. Python provides several methods to read data from a file –

read() – Reads the entire file.

readline() – Reads one line at a time.

readlines – Reads all lines into a list.

To read a file, you need to open it in read mode. The default mode for the open() function is read mode ('r'), but it's good practice to specify it explicitly.

Example: Using read() method

In the following example, we are using the read() method to read the whole file into a single string –

```
with open("example.txt", "r") as file:
   content = file.read()
   print(content)
```

Following is the output obtained –

```
Hello!!!
Welcome to Parul University!!!
```

Example: Using readline() method

In here, we are using the readline() method to read one line at a time, making it memory efficient for reading large files line by line −

```
with open("example.txt", "r") as file:
   line = file.readline()
   while line:
      print(line, end='')
      line = file.readline()
```

Output of the above code is as shown below −

```
Hello!!!
Welcome to Parul University!!!
```

Example: Using readlines() method

Now, we are using the readlines() method to read the entire file and splits it into a list where each element is a line −

```
with open("example.txt", "r") as file:
   lines = file.readlines()
   for line in lines:
      print(line, end='')
```

We get the output as follows −

```
Hello!!!
Welcome to Parul University!!!
```

Writing to a File in Python

Writing to a file in Python involves opening the file in a mode that allows writing, and then using various methods to add content to the file.
To write data to a file, use the write() or writelines() methods. When opening a file in write mode ('w'), the file's existing content is erased.

Example: Using the write() method

In this example, we are using the write() method to write the string passed to it to the file. If the file is opened in 'w' mode, it will overwrite any existing content. If the file is opened in 'a' mode, it will append the string to the end of the file −

Open Compiler

```
with open("foo.txt", "w") as file:
   file.write("Hello, World!")
   print ("Content added Successfully!!")
```

Output of the above code is as follows −

```
Content added Successfully!!
```

Example: Using the writelines() method

In here, we are using the writelines() method to take a list of strings and writes each string to the file. It is useful for writing multiple lines at once −

Open Compiler

```
lines = ["First line\n", "Second line\n", "Third line\n"]
with open("example.txt", "w") as file:
   file.writelines(lines)
   print ("Content added Successfully!!")
```

The result obtained is as follows −

```
Content added Successfully!!
```

Closing a File in Python

We can close a file in Python using the close() method. Closing a file is an essential step in file handling to ensure that all resources used by the file are properly released. It is important to close files after operations are completed to prevent data loss and free up system resources.

Example

In this example, we open the file for writing, write data to the file, and then close the file using the close() method −

Open Compiler

```
file = open("example.txt", "w")
file.write("This is an example.")
file.close()
print ("File closed successfully!!")
```

The output produced is as shown below −

```
File closed successfully!!
```

Using "with" Statement for Automatic File Closing

The with statement is a best practice in Python for file operations because it ensures that the file is automatically closed when the block of code is exited, even if an exception occurs.
the file is automatically closed at the end of the with block, so there is no need to call close() method explicitly −

Open Compiler

```
with open("example.txt", "w") as file:
   file.write("This is an example using the with statement.")
   print ("File closed successfully!!")
```

Following is the output of the above code −

```
File closed successfully!!
```

Handling Exceptions When Closing a File

When performing file operations, it is important to handle potential exceptions to ensure your program can manage errors gracefully.

In Python, we use a **try-finally** block to handle exceptions when closing a file. The "finally" block ensures that the file is closed regardless of whether an error occurs in the try block −

```
try:
   file = open("example.txt", "w")
   file.write("This is an example with exception handling.")
finally:
   file.close()
   print ("File closed successfully!!")
```

After executing the above code, we get the following output −

```
File closed successfully!!
```