

Unit 2: Functions, OOP, Exceptions, Files

Python Functions

- Once defined, Python functions can be called multiple times and from any location in a program.
- Our Python program can be broken up into numerous, easy-to-follow functions if it is significant.
- The ability to return as many outputs as we want using a variety of arguments is one of Python's most significant achievements.
- However, Python programs have always incurred overhead when calling functions.

However, calling functions has always been overhead in a Python program.

Syntax

1. `# An example Python Function`
2. `def function_name(parameters):`
3. `# code block`

The accompanying components make up to characterize a capability, as seen previously.

- The start of a capability header is shown by a catchphrase called `def`.
- `function_name` is the function's name, which we can use to distinguish it from other functions. We will utilize this name to call the capability later in the program. Name functions in Python must adhere to the same guidelines as naming variables.
- Using parameters, we provide the defined function with arguments. Notwithstanding, they are discretionary.
- A colon (`:`) marks the function header's end.
- Several valid Python statements make up the function's body. The entire code block's indentation depth-typically four spaces-must be the same.
- A return expression can get a value from a defined function.

Illustration of a User-Defined Function

We will define a function that returns the argument number's square when called.

1. `# Example Python Code for User-Defined function`
2. `def square(num):`
3. `"""`
4. `This function computes the square of the number.`
5. `"""`
6. `return num**2`
7. `object_ = square(6)`
8. `print("The square of the given number is: ", object_)`

Output:

```
The square of the given number is:  36
```

Calling a Function

Calling a Function To define a function, use the `def` keyword to give it a name, specify the arguments it must receive, and organize the code block.

When the fundamental framework for a function is finished, we can call it from anywhere in the program. An illustration of how to use the `a_function` function can be found below.

1. `# Example Python Code for calling a function`
2. `# Defining a function`
3. `def a_function(string):`
4. `"This prints the value of length of string"`
5. `return len(string)`
6.
7. `# Calling the function we defined`
8. `print("Length of the string Functions is: ", a_function("Functions"))`
9. `print("Length of the string Python is: ", a_function("Python"))`

Output:

```
Length of the string Functions is:  9
Length of the string Python is:    6
```

Pass by Reference vs. Pass by Value

In the Python programming language, all parameters are passed by reference.

Code

```
1. # Example Python Code for Pass by Reference vs. Value
2. # defining the function
3. def square( item_list ):
4.     """This function will find the square of items in the list"""
5.     squares = [ ]
6.     for l in item_list:
7.         squares.append( l**2 )
8.     return squares
9.
10. # calling the defined function
11. my_list = [17, 52, 8];
12. my_result = square( my_list )
13. print( "Squares of the list are: ", my_result )
```

Output:

```
Squares of the list are:  [289, 2704, 64]
```

Function Arguments

The following are the types of arguments that we can use to call a function:

- 1. Default arguments**
- 2. Keyword arguments**
- 3. Required arguments**
- 4. Variable-length arguments**

1) Default Arguments

Code

```
1. # Python code to demonstrate the use of default arguments
2. # defining a function
3. def function( n1, n2 = 20 ):
```

```
4. print("number 1 is: ", n1)
5. print("number 2 is: ", n2)
6.
7.
8. # Calling the function and passing only one argument
9. print( "Passing only one argument" )
10. function(30)
11.
12. # Now giving two arguments to the function
13. print( "Passing two arguments" )
14. 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
```

```
def fun(a,x=30,y,b=10):
    print("a=",a,"b=",b)
```

```
fun(1,20)
```

Output:

ERROR!

File "<string>", line 1

```
def fun(a,x=30,y,b=10):
```

^

SyntaxError: non-default argument follows default argument

>

Below works fine:

```
def fun(a,b=10,c=20):  
    print("a=",a,"b=",b,"c=",c)
```

fun(1,20)

Below works fine:

```
def fun(a=1,b=1,c=1):  
    print("a=",a,"b=",b,"c=",c)
```

fun()

2) Keyword Arguments

Code

1. # Python code to demonstrate the use of keyword arguments
2. # Defining a function
3. **def** function(n1, n2):
4. **print**("number 1 is: ", n1)
5. **print**("number 2 is: ", n2)
- 6.
7. # Calling function and passing arguments without using keyword

8. `print("Without using keyword")`
9. `function(50, 30)`
- 10.
11. `# Calling function and passing arguments using keyword`
12. `print("With using keyword")`
13. `function(n2 = 50, n1 = 30) #param names must match`

Output:

```
Without using keyword
number 1 is:  50
number 2 is:  30
With using keyword
number 1 is:  30
number 2 is:  50
```

Below works fine:

```
def fun(a=1,b=1,c=1):
    print("a=",a,"b=",b,"c=",c)
```

```
fun(b=10,a=20)
```

Below works fine:

```
def fun(a=10,b=1,c=1):
    print("a=",a,"b=",b,"c=",c)
```

```
fun(c=30,b=20)
```

3) Required Arguments

Code

```
1.  
2. # Defining a function  
3. def function( n1, n2 ):  
4.     print("number 1 is: ", n1)  
5.     print("number 2 is: ", n2)  
6.  
7. # Calling function and passing two arguments out of order, we need num1 to be 20 and num2 to be 30  
8. print( "Passing out of order arguments" )  
9. function( 30, 20 )  
10.  
11. # Calling function and passing only one argument  
12. print( "Passing only one argument" )  
13. try:  
14.     function( 30 )  
15. except:  
16.     print( "Function needs two positional arguments" )
```

Output:

```
Passing out of order arguments  
number 1 is:  30  
number 2 is:  20  
Passing only one argument  
Function needs two positional arguments
```

4) Variable-Length Arguments

"args" and "kwargs" refer to arguments not based on keywords.

Arbitrary Keyword Arguments, **kwargs

If you do not know how many keyword arguments that will be passed into your function, add two asterisk: ** before the parameter name in the function definition.

This way the function will receive a ***dictionary of arguments***, and can access the items accordingly:

If the number of keyword arguments is unknown, add a double ****** before the parameter name:

```
def my_function(**kid): #getting dictionary
    print("His last name is " + kid["lname"]) #printing from Dictionary
```

```
my_function(fname = "amit", lname = "patel")
```

Arbitrary Kword Arguments are often shortened to ****kwargs** in Python documentations.

To help you understand arguments of variable length, here's an example.

Code

```
1. # Python code to demonstrate the use of variable-length arguments
2. # Defining a function
3. def function( *args_list ):
4.     ans = []
5.     for l in args_list:
6.         ans.append( l.upper() )
7.     return ans
8. # Passing args arguments
9. object = function('Python', 'Functions', 'tutorial')
10. print( object )
11.
12. # defining a function
13. def function( **kwargs_list ):
14.     ans = []
15.     for key, value in kwargs_list.items():
16.         ans.append([key, value])
17.     return ans
18. # Paasing kwargs arguments
19. object = function(First = "Python", Second = "Functions", Third = "Tutorial")
20. print(object)
```


Output:

```
['PYTHON', 'FUNCTIONS', 'TUTORIAL']  
[['First', 'Python'], ['Second', 'Functions'], ['Third', 'Tutorial']]
```

```
def fun(**a):  
    print(a) #it will print Dictionary
```

```
fun(c=30,b=20)
```

Output:

```
{'c': 30, 'b': 20}
```

Example:

```
def fun(c,**a,d):  
    print(a)  
    print(c)
```

```
fun(10,a=10,d=30)
```

Error:

ERROR!

File "<string>", line 1

```
def fun(c,**a,d):
```

^

SyntaxError: arguments cannot follow var-keyword argument

Example

```
def fun(c=10,**a):
```

```
    print(a)
```

```
    print(c)
```

```
fun(a=10,d=30)
```

Output:

```
{'a': 10, 'd': 30}
```

```
10
```

Example:

```
def fun(b,c=10,**a):
```

```
    print(a)
```

```
    print(c)
```

```
fun(40,30)
```

Output:

```
{}
```

```
30
```

Example
`def fun(**a):`

```
    print(a)
```

```
    # print(c)
```

```
fun(40,30)
```

ERROR!

Traceback (most recent call last):

```
File "<string>", line 5, in <module>
```

TypeError: fun() takes 0 positional arguments but 2 were given

Example:

```
def fun(**a):
```

```
print(a)
```

```
print(c)
```

```
fun(40)
```

Output:

ERROR!

Traceback (most recent call last):

File "<string>", line 5, in <module>

TypeError: fun() takes 0 positional arguments but 1 was given

Example:

```
def fun(c,**a):
```

```
    print(a)
```

```
    print(c)
```

```
fun(40)
```

Output:

```
{}
```

40

Example:

```
def fun(c,**a):  
    print(a)  
    print(c)
```

```
fun(40,a=10,b=20)
```

Output:

```
{'a': 10, 'b': 20}
```

40

Example:

```
def fun(**b,**a):  
    print(a)  
    print(c)
```

```
fun(b=50,a=500)
```

Example:

ERROR!

File "<string>", line 1

```
def fun(**b,**a):
```

```
    ^^
```

SyntaxError: arguments cannot follow var-keyword argument

Example

```
def fun(**a):
```

```
    print(a)
```

```
    #print(c)
```

```
fun()
```

Output:

```
{}
```

Example

```
def fun(**a):
```

```
    print(a)
```

```
    #print(c)
```

```
list=[1,2,3]
```

```
fun(b=list)
```

Output:

```
{'b': [1, 2, 3]}
```

Example:

```
def fun(**a):
```

```
    print(a)
```

```
    #print(c)
```

```
list=[1,2,3]
```

```
fun(list)
```

ERROR

Example:

```
def fun(c,**a):
```

```
    print(a)
```

```
    print(c)
```

```
list=[1,2,3]
```

```
fun(list,b=[4,5,6],x=[10,20,30],y="amit")
```

Output:

```
{'b': [4, 5, 6], 'x': [10, 20, 30], 'y': 'amit'}
```

```
[1, 2, 3]
```

Example:

```
def fun(c,x,**a):
```

```
    print(a)
```

```
list=[1,2,3]
```

```
fun(list,10,d=20,a=50,x=100)
```

Error:

Traceback (most recent call last):

```
File "/home/cg/root/657ed8daed710/main.py", line 5,  
in <module>
```

```
fun(list,10,d=20,a=50,x=100)
```

TypeError: fun() got multiple values for argument 'x'


```
def fun(p,a,x):
```

```
    print(a)
```

```
list=[1,2,3]
```

```
fun(20,a=50,p=100)
```

Traceback (most recent call last):

File "/home/cg/root/657ed8daed710/main.py", line 5,
in <module>

```
fun(20,a=50,p=100)
```

TypeError: fun() got multiple values for argument 'p'

Example:

```
def fun(p,a,l):
```

```
    print(a)
```

```
list=[1,2,3]
```

```
fun(20,a=50,90)
```

Output:

File "/home/cg/root/657ed8daed710/main.py", line 5

```
fun(20,a=50,90)
```

^

SyntaxError: positional argument follows keyword argument

```
def fun(l):
```

```
    return l*2
```

```
list=[1,2,3]
```

```
print(fun(list))
```

Output:

```
[1, 2, 3, 1, 2, 3]
```

```
def fun(**a):
```

```
    return a
```

```
print(fun(a=10,b=20))
```

Output:

```
{'a': 10, 'b': 20}
```

Example:

```
def fun(**a):  
    a['a']=1000 #changing value in dictionary  
    return a  
  
print(fun(a=10,b=20))
```

Output:

```
{'a': 1000, 'b': 20}
```

*args

```
def fun(*a):  
    print(*a)  
    print(a)
```

```
print(fun(10))
```

Output:

10

(10,)

None

Example:

```
def fun(*a):
```

```
    print(*a)
```

```
    print(a)
```

```
print(fun(10,20,30))
```

Output:

10 20 30

(10, 20, 30)

None

Example:

```
def fun(*a,c,d,b=1000):  
    print(*a)  
    print(f'a={a},b={b},c={c},d={d}')
```

print(a)

```
print(fun(10,d=110,c=20))
```

Output:

10

a=(10,),b=1000,c=20,d=110

(10,)

None

Example:

```
def fun(a=10,b=50,*c):  
    print(*c)  
    print(f'a={a},b={b},c={c}')
```

print(c)

```
print(fun(20,30))
```

Output:

```
a=20,b=30,c=()
```

```
()
```

```
None
```

Example:

```
def fun(a,b,**c,d):
```

```
    print(c)
```

```
    print(c)
```

```
print(fun(20,30,c=40,d=50))
```

Output:

File `"/home/cg/root/657ed8daed710/main.py"`, line 1

```
def fun(a,b,**c,d):
```

```
    ^
```

SyntaxError: invalid syntax

Example:

```
def fun(*t):
```

```
    print(t)
```

```
print(fun(111,222,[10,20,30],(40,50),{60,70},{"name":  
amit"}))
```

Output:

```
(111, 222, [10, 20, 30], (40, 50), {60, 70}, {'name':  
'amit'})
```

None

When a defined function is called, a return statement is written to exit the function and return the calculated value.

Arbitrary Arguments, *args

If you do not know how many arguments that will be passed into your function, add a `*` before the parameter name in the function definition.

This way the function will receive *a tuple* of arguments, and can access the items accordingly:

If the number of arguments is unknown, add a `*` before the parameter name:

```
def my_function(*kids):  
    print("The youngest child is " + kids[2])
```

```
my_function("one", "two", "three")
```

Arbitrary Arguments are often shortened to **args* in Python documentations.

Related Pages

return Statement

When a defined function is called, a return statement is written to exit the function and return the calculated value.

Syntax:

1. **return** < expression to be returned as output >

The return statement can be an argument, a statement, or a value, and it is provided as output when a particular job or function is finished. A declared function will return an empty string if no return statement is written.

A return statement in Python functions is depicted in the following example.

Code

1. # Python code to demonstrate the use of return statements
2. # Defining a function with return statement
3. **def** square(num):
4. **return** num**2
- 5.
6. # Calling function and passing arguments.
7. **print**("With return statement")
8. **print**(square(52))
- 9.
10. # Defining a function without return statement
11. **def** square(num):
12. num**2
- 13.
14. # Calling function and passing arguments.
15. **print**("Without return statement")
16. **print**(square(52))

Output:

```
With return statement
2704
Without return statement
None
```

The Anonymous Functions (like lambda)

Since we do not use the def keyword to declare these kinds of Python functions, they are unknown. The lambda keyword can define anonymous, short, single-output functions.

Arguments can be accepted in any number by lambda expressions; However, the function only produces a single value from them. They cannot contain multiple instructions or expressions.

Lambda functions can only refer to variables in their argument list and the global domain name because they contain their distinct local domain.

lambda expressions appear to be one-line representations of functions.

Syntax

Lambda functions have exactly one line in their syntax:

1. **lambda** [argument1 [,argument2... .argumentn]] : expression

Below is an illustration of how to use the lambda function:

A lambda function is a small anonymous function.

A lambda function can take any number of arguments, but can only have one expression.

Syntax

`lambda arguments : expression`

The expression is executed and the result is returned:

Add 10 to argument `a`, and return the result:

```
x = lambda a : a + 10
print(x(5))
```

Lambda functions can take any number of arguments:

Example

Multiply argument `a` with argument `b` and return the result:

```
x = lambda a, b : a * b
print(x(5, 6))
```

Example

Summarize argument `a`, `b`, and `c` and return the result:

```
x = lambda a, b, c : a + b + c
print(x(5, 6, 2))
```

EXAMPLE

```
x = lambda **a : print(a)
print(x(a=6,b=8))
```

Output:

```
{'a': 6, 'b': 8}
None
```

EXAMPLE

```
x = lambda b,c,*a : print(a,b,c)
print(x(1,2,3,4))
```

Output:

```
(3, 4) 1 2
None
```

Example:

```
x = lambda b,*a,c=10 : print(a,b,c)
print(x(1,2,3,4))
```

Output:

```
(2, 3, 4) 1 10
None
```

Example:

```
x = lambda b,*a,c : print(a,b,c)
print(x(1,2,3,4))
```

Output:

Traceback (most recent call last):

File "/home/cg/root/657ed8daed710/main.py", line 2, in <module>

print(x(1,2,3,4))

TypeError: <lambda>() missing 1 required keyword-only argument: 'c'

Why Use Lambda Functions?

The power of lambda is better shown when you use them as an anonymous function inside another function.

Say you have a function definition that takes one argument, and that argument will be multiplied with an unknown number:

```
def myfunc(n):  
    return lambda a : a * n
```

Example

```
def myfunc(n):  
    return lambda a : a * n
```

```
mydoubler = myfunc(2)
```

```
print(mydoubler(11))
```

Example

```
def myfunc(n):  
    return lambda a : a * n  
    #it returns lambda function, not any value, returns lambda a:a*2 (i.e.  
    value of n)
```

```
mydoubler = myfunc(2)
```

```
# myfunc() returns lambda function that is saved in mydoubler, so my  
doubler is now lambda function
```

```
print(mydoubler) #prints memory address of lambda function
```

```
print(mydoubler(11)) #calling lambda function
```

Or, use the same function definition to make a function that always *triples* the number you send in:

Example

```
def myfunc(n):  
    return lambda a : a * n #returns lambda a: a*3
```

```
mytripler = myfunc(3) #mytripler is a lambda function
```

```
print(mytripler(11)) #calling lambda function
```

Or, use the same function definition to make both functions, in the same program:

Example

```
def myfunc(n):  
    return lambda a : a * n
```

```
mydoubler = myfunc(2)  
mytripler = myfunc(3)
```

```
print(mydoubler(11))  
print(mytripler(11))
```

Use lambda functions when an anonymous function is required for a short period of time.

Code

1. # Python code to demonstrate anonymous functions
2. # Defining a function
3. lambda_ = **lambda** argument1, argument2: argument1 + argument2;
- 4.
5. # Calling the function and passing values
6. **print**("Value of the function is : ", lambda_(20, 30))
7. **print**("Value of the function is : ", lambda_(40, 50))

Output:

```
Value of the function is :  50  
Value of the function is :  90
```

Scope and Lifetime of Variables

A variable's scope refers to the program's domain wherever it is declared. The lifespan of a function is the same as the lifespan of its internal variables. When we exit the function, function variables are taken away from us. As a result, the value of a variable in a function does not persist from previous executions.

Code

1. `# Python code to demonstrate scope and lifetime of variables`
2. `#defining a function to print a number.`
3. `def number():`
4. `num = 50`
5. `print("Value of num inside the function: ", num)`
- 6.
7. `num = 10`
8. `number()`
9. `print("Value of num outside the function:", num)`

Output:

```
Value of num inside the function:  50
Value of num outside the function: 10
```

Here, we can see that the initial value of num is 10. Even though the function number() changed the value of num to 50, the value of num outside of the function remained unchanged.

Is it really pass by Reference in function calls?

```
def fun(a):
```

```
    print(id(a)) #both 'a' have same id
```

```
    a=20 #after assignment id(a) changed to new values
```

```
    print(id(a)) #prints new id(a)
```

```
a=0
```

```
print(id(a))
```

```
fun(a)
```

```
print(a)
```

```
print(id(a))
```

Output:

140200934097104

140200934097104

140200934097744 #local 'a' after assignment in function, address of 'a' in function changed

0

140200934097104 #'a' outside function

Function inside a Function

A function defined within another is called an "inner" or "nested" function. The parameters of the outer scope are accessible to inner functions.

Code

1. # Python code to show how to access variables of a nested functions
2. # defining a nested function
3. **def** word():
4. string = 'Python functions tutorial'
5. x = 5
6. **def** number():
7. **print**(string)
8. **print**(x)
- 9.
10. number()
11. word()

Output:

```
Python functions tutorial
5
```

Nested (or inner) functions are functions defined within other functions that allow us to directly access the variables and names defined in the enclosing function.

Defining an inner/nested Function

Simply use the **def** keyword to initialize another function within a function to define a **nested** function.

The following program is to demonstrate the inner function in Python –

Example

```
# creating an outer function
def outerFunc(sample_text):
    sample_text = sample_text
# creating an inner function
    def innerFunc():
# Printing the variable of the parent class(outer class)
        print(sample_text)
# calling inner function
        innerFunc()
# Calling the outer Function by passing some random name
outerFunc('Hello Amit')
innerFunc() #This will give error, you cannot call inner function outside outer
function
```

Output

On executing, the above program will generate the following output –

Hello Amit

Here, **innerFunc()** is declared inside the **outerFunc()**, making it an inner function. We must first call **outerFunc()** before we can call **innerFunc()**. The **outerFunc()** will then call the **innerFunc()**, which is defined within it.

It is essential that the outer function be called so that the inner function can execute.

Below will give not output as we have not called inner function

```
# creating an outer function
```

```
def outerFunc(sample_text):  
  
    # creating an inner function  
  
    def innerFunc():  
  
        # Printing the variable of the parent class(outer class)  
  
        print(sample_text)  
  
    # calling inner function  
  
  
    # Calling the outer Function by passing some random name  
  
outerFunc('Hello Amit')
```

Without Calling Outer Function

Example

```
# creating an outer function  
def outerFunc(sample_text):  
  
    # creating an inner function  
    def innerFunc():  
        print(sample_text)  
    # calling inner function  
    innerFunc()
```

Output

On executing, the above program will generate the following output –

When run, the above code will return no results.

Scope of Variable in Nested Function

The scope of a variable is the location where we can find a variable and access it if necessary.

It is well-understood how to access a global variable within a function, but what about accessing the variable of an outside function?

The following program demonstrates the scope of the nested functions –

Example

```
# creating an outer function
def outerFunc():
    sample_str = 'Hello Amit'

    # creating an inner function
    def innerFunc():
        print(sample_str)

    # calling an inner function inside the outer function
    innerFunc()
# calling outer function
outerFunc()
```

Output

On executing, the above program will generate the following output –

Hello Amit

It is clear from the above example that it is equivalent to accessing a global variable from a function.

Assume you wish to **modify the variable** of the outer function.

Example

The following program changes the value of a variable inside the nested function(inner function) –

```
# creating an outer function
def outerFunc():
    sample_str = 'Hello Amit'
    # creating an inner function
    def innerFunc():
```

```
# modifying the sample string
sample_str = 'Welcome'
print(sample_str)
# calling an inner function inside the outer function
innerFunc()
print(sample_str)
# calling outer function
outerFunc()
```

Output

On executing, the above program will generate the following output –

```
Welcome
Hello Amit
```

Here, the value of the outer function's variable remains unchanged. However, the value of the outer function's variable can be modified. There are several methods for changing the value of the outer function's variable.

Printing IDs

```
# creating an outer function

def outerFunc():

    sample_str = 'Hello Amit with outer ID'

    print(id(sample_str))

# creating an inner function

def innerFunc():

    # modifying the sample string

    sample_str = 'Welcome with Inner ID'

    print(sample_str)

    print(id(sample_str))
```

```
# calling an inner function inside the outer function
```

```
    innerFunc()
```

```
    print(sample_str)
```

```
    print(id(sample_str))
```

```
# calling outer function
```

```
outerFunc()
```

Output:

```
140591567492224
```

```
Welcome with Innder ID
```

```
140591565048688
```

```
Hello Amit with outer ID
```

```
140591567492224
```

```
>
```

Using an iterable of inner functions

The Following program demonstrates how to use modify iterables in inner functions –

```
# Creating outer function
def outerFunc():
    # Creating an iterable
    sample_str = ['Hello Amit python']
    # Creating inner Function/Nested Function
    def innerFunc():
        # using an iterable to modify the value
```

```

    sample_str[0] = 'Welcome'
    print(sample_str)
    # Calling the inner function
    innerFunc()
    # Printing variable of the outer function
    print(sample_str)
# Calling the outer function
outerFunc()

```

Output

On executing, the above program will generate the following output –

```

['Welcome']
['Welcome']

```

Example:

```

# Creating outer function
def outerFunc():
    # Creating an iterable
    sample_str = ['Hello Amit python']
    # Creating inner Function/Nested Function
    def innerFunc():
        # using an iterable to modify the value
        sample_str=["Hi PU","Hello students"]
        print(sample_str)
        sample_str[0] = 'Welcome to'
        print(sample_str)
    # Calling the inner function
    innerFunc()
    # Printing variable of the outer function
    print(sample_str)
# Calling the outer function
outerFunc()

```

Output:

```

['Hi PU', 'Hello students']
['Welcome to', 'Hello students']

```

```
['Hello Amit python']
```

Why should you use nested functions?

Data encapsulation

There are cases when you wish to encapsulate a function or the data it has access to in order to prevent it from being accessed from other parts of your code.

When you nest a function like this, it becomes hidden to the global scope. Because of this characteristic, data encapsulation is also known as data hiding or data privacy.

```
# creating an outer function
def outerFunc():
    print("This is outer function")
    # creating an inner function
    def innerFunc():
        print("This is inner function")

    # calling an inner function inside the outer function
    innerFunc()
# calling inner function outside the outer function
innerFunc()
```

Output

On executing, the above program will generate the following output –

Traceback (most recent call last):

File "main.py", line 11, in <module>

innerFunc()

NameError: name 'innerFunc' is not defined

The inner function in the code above is only accessible from within the outer function. If you attempt to call inner from outside the function, you will see the error shown above.

Instead, you must call the outer function as follows.

```
# creating an outer function
def outerFunc():
    print("This is outer function")
    # creating an inner function
    def innerFunc():
        print("This is inner function")

    # calling an inner function inside the outer function
    innerFunc()
# calling outer function
outerFunc()
```

Output

On executing, the above program will generate the following output –

```
This is outer function
This is inner function
```

Python Lambda Functions

Lambda Functions are anonymous functions in Python. A lambda function can take n number of arguments at a time. **But it returns only one argument at a time.**

What are Lambda Functions in Python?

Lambda Functions in Python are anonymous functions, implying they don't have a name. The def keyword is needed to create a typical function in Python, as we already know. **We can also use the lambda keyword in Python to define an unnamed function.**

Syntax

The syntax of the Lambda Function is given below -

1. **lambda** arguments: expression

This function accepts any count of inputs but only evaluates and returns one expression. **That means it takes many inputs but returns only one output.**

Lambda functions are limited to a single statement.

Example

1. `# Code to demonstrate how we can use a lambda function for adding 4 numbers`
2. `add = lambda num: num + 4` #add variable/object points to the Lambda Function
3. `print(add(6))`

Output:

```
10
```

Here we explain the above code. The lambda function is "lambda num: num+4" in the given programme. The parameter is num, and the computed and returned equation is num * 4.

There is no label for this function. It generates a function object associated with the "add" identifier. We can now refer to it as a standard function.

Output:

Now we compile the above code in Python, and after successful compilation, we run it. Then the output is given below -

```
10
```

Program Code

Now we gave an example of a lambda function that multiply 2 numbers and return one result. The code is shown below -

1. `a = lambda x, y : (x * y)`
2. `print(a(4, 5))`

Output:

```
20
```

Program Code

1. `a = lambda x, y, z : (x + y + z)`
2. `print(a(4, 5, 5))`

Output:

```
14
```

What's the Distinction Between Lambda and Def Functions?

This program calculates the reciprocal of a given number:

Program Code:

1. `# Python code to show the reciprocal of the given number to highlight the difference between def() and lambda().`
2. `def reciprocal(num):`
3. `return 1 / num`
- 4.
5. `lambda_reciprocal = lambda num: 1 / num`
- 6.
7. `# using the function defined by def keyword`
8. `print("Def keyword: ", reciprocal(6))`
- 9.
10. `# using the function defined by lambda keyword`
11. `print("Lambda keyword: ", lambda_reciprocal(6))`

Output:

```
Def keyword:  0.16666666666666666
Lambda keyword:  0.16666666666666666
```

Explanation:

Using Lambda: Instead of a "return" statement, Lambda definitions always include a statement given at output. The beauty of lambda functions is their convenience..

Using Lambda Function with filter()

The filter() method accepts two arguments in Python: a function and an iterable such as a list.

The function is called for every item of the list, and a new iterable or list is returned that holds just those elements that returned True when supplied to the function.

Here's a simple illustration of using the filter() method to return only odd numbers from a list.

Program Code:

1. `# This code used to filter the odd numbers from the given list`
2. `list_ = [35, 12, 69, 55, 75, 14, 73]`
3. `odd_list = list(filter(lambda num: (num % 2 != 0) , list_))`
4. `print('The list of odd number is:',odd_list)`

Output:

```
The list of odd number is: [35, 69, 55, 75, 73]
```

Python map() Function

Example

Calculate the length of each word in the tuple:

```
def myfunc(n):  
    return len(n)
```

```
x = map(myfunc, ('apple', 'banana', 'cherry'))
```

Example:

```
def myfunc(a):  
    print(len(a))  
    return len(a) # returns len 5, then 6, then 6
```

```
x = map(myfunc, ('apple', 'banana', 'cherry'))  
#passing one element at a time to myfunc() and  
#creating map object with 5,6,6
```

```
print(x)#prints map object, not any values
```

```
#convert the map into a list, for readability:  
print(list(x))
```

Definition and Usage

The `map()` function executes a specified function for each item(element) in an iterable (like lists). The item is sent to the function as a parameter.

Syntax

```
map(function, iterables)
```

Parameter Values

| Parameter | Description |
|-----------------|--|
| <i>function</i> | Required. The function to execute for each item |
| <i>iterable</i> | Required. A sequence, collection or an iterator object. You can send as many iterables as you like, just make sure the function has one parameter for each iterable. |

More Examples

Example

Make new fruits by sending two iterable objects into the function:

```
def myfunc(a, b):
    print("a=",a)
    print("b=",b,"\n")
    return a + b

x = map(myfunc, ('apple', 'banana', 'cherry'), ('orange', 'lemon', 'pineapple'))

print(x)#prints map object , not values

#convert the map into a list, for readability:
print(list(x))
```

```
<map object at 0x154fe889f130>
a= apple
b= orange

a= banana
b= lemon

a= cherry
b= pineapple

['appleorange', 'bananalemon', 'cherrypineapple']
```

Using Lambda Function with map()

A method and a list are passed to Python's map() function.

The function is executed for all of the elements within the list, and a new list is produced with elements generated by the given function for every item.

The map() method is used to square all the entries in a list in this example.

Program Code:

Here we give an example of lambda function with map() in Python. Then code is given below -

1. #Code to calculate the square of each number of a list using the map() function

2. `numbers_list = [2, 4, 5, 1, 3, 7, 8, 9, 10]`
3. `squared_list = list(map(lambda num: num ** 2 , numbers_list))`
4. `print('Square of each number in the given list:' ,squared_list)`

Output:

```
Square of each number in the given list: [4, 16, 25, 1, 9, 49, 64, 81, 100]
```

List Comprehension

List comprehension offers a shorter syntax when you want to create a new list based on the values of **an existing list**.

Example:

Based on a list of fruits, you want a new list, containing only the fruits with the letter "a" in the name.

Without list comprehension you will have to write a **for** statement with a conditional test inside:

Example

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]  
newlist = []
```

```
for x in fruits:  
    if "a" in x:  
        newlist.append(x)
```

```
print(newlist)
```

With list comprehension you can do all that with only one line of code:

Example

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
```

```
newlist = [x for x in fruits if "a" in x]
```

```
print(newlist)
```

The Syntax

```
newlist = [expression for item in iterable if condition == True]
```

The return value is a new list, leaving the old list unchanged.

Condition

The *condition* is like a filter that only accepts the items that valuate to **True**.

Example

Only accept items that are not "apple":

```
newlist = [x for x in fruits if x != "apple"]
```

The condition `if x != "apple"` will return **True** for all elements other than "apple", making the new list contain all fruits except "apple".

The *condition* is optional and can be omitted:

Example

With no **if** statement:

```
newlist = [x for x in fruits]
```

Iterable

The *iterable* can be any iterable object, like a list, tuple, set etc.

Example

You can use the `range()` function to create an iterable:

```
newlist = [x for x in range(10)]
```

Same example, but with a condition:

Example

Accept only numbers lower than 5:

```
newlist = [x for x in range(10) if x < 5]
```

Expression

The *expression* is the current item in the iteration, but it is also the outcome, which you can manipulate before it ends up like a list item in the new list:

Example

Set the values in the new list to upper case:

```
newlist = [x.upper() for x in fruits]
```

You can set the outcome to whatever you like:

Example

Set all values in the new list to 'hello':

```
newlist = ['hello' for x in fruits]
```

Using Lambda Function with if-else

We will use the lambda function with the if-else block. In the program code below, we check which number is smaller than the given two numbers using the if-else block.

Program Code:

1. `# Code to use lambda function with if-else`
2. `Minimum = lambda x, y : x if (x < y) else y`
3. `print('The smaller number is:', Minimum(35, 74))`

Output:

```
The smaller number is: 35
```

A lambda function can take n number of arguments at a time. But it returns only one argument at a time.

Python File Handling

Python File Handling

Introduction:

The file handling plays an important role when the data needs to be stored permanently into the file. **A file is a named location on disk to store related information.** We can access the stored information (non-volatile) even after the program termination.

In Python, files are treated in two modes as text or binary. The file may be in the text or binary format, and each line of **a file is ended with the special character like a comma (,) or a newline character.** Python executes the code line by line. So, it works in one line and then asks the interpreter to start the new line again. This is a continuous process in Python.

Hence, a file operation can be done in the following order.

- Open a file
- Read or write or append to a file
- Close the file

Opening a file

A file operation starts with the file opening. At first, open the File then Python will start the operation. File opening is done with **the open()** function in Python. **This function will accepts two arguments, file name and access mode** in which the file is accessed. **When we use the open() function, that time we must be specified the mode for which the File is opening. The function returns a file object which can be used to perform various operations like reading, writing, etc.**

Syntax:

The syntax for opening a file in Python is given below -

1. file object = open(<file-name>, <access-mode>, <buffering>)

The files can be accessed using various modes like read, write, or append. The following are the details about the access mode to open a file.

| SN | Access mode | Description |
|----|-------------|---|
| 1 | r (default) | r means to read. So, it opens a file for read-only operation. The file pointer exists at the beginning. The file is by default open in this mode if no access mode is passed. |
| 2 | rb | It opens the file to read-only in binary format. The file pointer exists at the beginning of the file. |
| 3 | r+ | It opens the file to read and write both. The file pointer exists at the beginning of the file. |
| 4 | rb+ | It opens the file to read and write both in binary format. The file pointer exists at the beginning of the file. |
| 5 | w | It opens the file to write only. It overwrites the file if previously exists or creates a new one if no file exists with the same name. The file pointer exists at the beginning of the file. |
| 6 | wb | It opens the file to write only in binary format. It overwrites the file if it exists previously or creates a new one if no file exists. The file pointer exists at the beginning of the file. |
| 7 | w+ | It opens the file to write and read both. It is different from r+ in the sense that it overwrites the previous file if one exists whereas r+ doesn't overwrite the previously written file. It creates a new file if no file exists. The file pointer exists at the beginning of the file. |
| 8 | wb+ | It opens the file to write and read both in binary format. The file pointer exists at the beginning of the file. |
| 9 | a | It opens the file in the append mode. The file pointer exists at the end of the previously written file if exists any. It creates a new file if no file exists with the same name. |
| 10 | ab | It opens the file in the append mode in binary format. The pointer exists at the end of the previously written file. It creates a new file in binary format if no file exists with the same name. |
| 11 | a+ | It opens a file to append and read both. The file pointer remains at the end of the file if a file exists. It creates a new file if no file exists with the same name. |

| | | |
|----|-----|--|
| 12 | ab+ | It opens a file to append and read both in binary format. The file pointer remains at the end of the file. |
|----|-----|--|

Let's look at the simple example to open a file named "file.txt" (stored in the same directory) in read mode and printing its content on the console.

Program code for read mode:

It is a read operation in Python. We open an existing file with the given code and then read it. The code is given below -

1. #opens the file file.txt in read mode
2. fileptr = open("file.txt", "r")
- 3.
4. if fileptr:
5. print("file is opened successfully")

Output:

Now we compile the above code in Python, and after successful compilation, we run it. Then the output is given below -

```
file is opened successfully
```

In the above code, we have passed **filename** as a first argument and opened file in read mode as we mentioned **r** as the second argument. **The fileptr holds the file object and if the file is opened successfully, it will execute the print statement**

Program code for Write Mode:

It is a write operation in Python. We open an existing file using the given code and then write on it. The code is given below -

1. file = open('file.txt', 'w')
2. file.write("Here we write a command")
3. file.write("Hello users of PARUL MIS")
4. file.close()

The close() Method

The close method used to terminate the program. Once all the operations are done on the file, we must close it through our Python script using the **close()** method. Any

unwritten information gets destroyed once the **close()** method is called on a file object.

We can perform any operation on the file externally using the file system which is the currently opened in Python; hence it is good practice to close the file once all the operations are done. Earlier use of the close() method can cause the of destroyed some information that you want to write in your File.

The syntax to use the **close()** method is given below.

Syntax

The syntax for closing a file in Python is given below -

1. fileobject.close()

Consider the following example.

Program code for Closing Method:

Here we write the program code for the closing method in Python. The code is given below -

1. `# opens the file file.txt in read mode`
2. `fileptr = open("file.txt","r")`
- 3.
4. `if fileptr:`
5. `print("The existing file is opened successfully in Python")`
- 6.
7. `#closes the opened file`
8. `fileptr.close()`

After closing the file, we cannot perform any operation in the file. The file needs to be properly closed. If any exception occurs while performing some operations in the file then the program terminates without closing the file.

We should use the following method to overcome such type of problem.

1. `try:`
2. `fileptr = open("file.txt")`
3. `# perform file operations`
4. `finally:`

5. `fileptr.close()`

The with statement

The **with** statement was introduced in python 2.5. The with statement is useful in the case of manipulating the files.

Syntax:

The syntax of with statement of a file in Python is given below -

1. `with open(<file name>, <access mode>) as <file-pointer>:`
2. `#statement suite`

The advantage of using with statement is that it provides the guarantee to close the file regardless of how the nested block exits.

It is always suggestible to use the with statement in the case of files because, if the break, return, or exception occurs in the nested block of code then it automatically closes the file, we don't need to write the close() function. It doesn't let the file to corrupt.

Program code 1 for with statement:

Here we write the program code for with statement in Python. The code is given below:

1. `with open("file.txt",'r') as f:`
2. `content = f.read();`
3. `print(content)`

Program code 2 for with statement:

Here we write the program code for with statement in Python. The code is given below -

1. `with open("file.txt", "w") as f:`
2. `A = f.write("Hello Coders")`
3. `Print(A)`

```
with open('demoText2.txt','w') as filePtr:
    fileData=filePtr.write("hi\nhello\ngood morning\ngood")
```

```
bye")
print(fileData)
```

file.write() returns number of characters written in the file

Writing the file

To write some text to a file, we need to open the file using the open method and then we can use the write method for writing in this File. If we want to open a file that does not exist in our system, it creates a new one. On the other hand, if the File exists, then erase the past content and add new content to this File. It is done by the following access modes.

w: It will overwrite the file if any file exists. The file pointer is at the beginning of the file.

a: It will append the existing file. The file pointer is at the end of the file. It creates a new file if no file exists.

Program code 1 for Write Method:

1. **# open the file.txt in append mode. Create a new file if no such file exists.**
2. `fileptr = open("file2.txt", "w")`
- 3.
4. **# appending the content to the file**
5. `fileptr.write('Python is the modern programming language. It is done any kind of program in shortest way.')`
- 6.
7. **# closing the opened the file**
8. `fileptr.close()`

Output:

Now we compile the above code in Python, and after successful compilation, we run it. Then the output is given below -

```
File2.txt
Python is the modern programming language. It is done any kind of program
in shortest way.
```

We have opened the file in w mode. The file1.txt file doesn't exist, it created a new file and we have written the content in the file using the write() function

Program code 2 for Write Method:

```
with open('demoText.txt', 'w') as file2:  
    file2.write('Hello coders\n')  
file2.write('Welcome to parul MIS')
```

Error:

Traceback (most recent call last):

File
"C:\Users\dell\PycharmProjects\pythonProject2\GUIpythonProject\FilesNonGUI\open
File.py", line 3, in <module>

file2.write('Welcome to parul MIS')

ValueError: I/O operation on closed file.

Note: At the end of the block of WITH, file is automatically closed. So, you cannot access that file outside the WITH block.

Here we write the program code for write method in Python. The code is given below

-

1. with open(test1.txt', 'w') as file2:
2. file2.write('Hello coders\n')
3. file2.write('Welcome to parul MIS')

Output:

Now we compile the above code in Python, and after successful compilation, we run it. Then the output is given below -

```
Hello coders  
Welcome to parul MIS
```

Program code 3 for Write Method:

Here we write the program code for write method in Python. The code is given below

-

1. `#open the file.txt in write mode.`
2. `fileptr = open("file2.txt","a")`
- 3.
4. `#appending the content of the file`
5. `fileptr.write(" Python has an easy syntax and user-friendly interaction.")`
- 6.
7. `#closing the opened file`
8. `fileptr.close()`

We can see that the content of the file is appended/modified. We have opened the file in **a** mode and it appended the content in the existing **file2.txt**.

Read(count=number of characters/bytes)

The Python provides the **read(no. of characters)** method. It returns string. It can read the data in the text as well as a binary format.

Syntax:

1. `fileobj.read(<count>)`

Here, the count is the number of bytes/characters to be read from the file starting from the beginning of the file. If the count is not specified, then it may read the content of the file until the end.

Program code for read() Method:

```
fileptr = open("demoText.txt","r")
    #stores all the data of the file into the variable content

allData=fileptr.read()
print(type(allData)) # prints string object
    #prints the content of the file
print(allData)

firstTenBytes = fileptr.read(10)
print(firstTenBytes) #fileptr at end so will not print
anything
fileptr.seek(0) #reset fileptr to first character in the file
firstTenBytes=fileptr.read(10)

print(firstTenBytes)
#prints first 10 characters
```

```
#closes the opened file
fileptr.close()
```

Output:

<class 'str'>

my name is amit

some students talk too much during lectures/labs

some use mobile phones

i need to punish them as demo students to set them as examples

my name is

We have passed count value as ten which means it will read the first ten characters/bytes from the file.

If we use the following line, then it will print all content of the file.

1. `content = fileptr.read()`
2. `print(content)`

Read file through for loop

We can read the file using for loop. Consider the following example.

Program code 1 for Read File using For Loop:

Here we give an example of read file using for loop. The code is given below -

1. `#open the file.txt in read mode. causes an error if no such file exists.`
2. `fileptr = open("file2.txt","r")`
3. `#running a for loop`
4. `for i in fileptr:`
5. `print(i) # i contains each line of the file`

Output:

Python is the modern day language.

It makes things so simple.

Python has easy syntax and user-friendly interaction.

```
fileptr = open("demoText.txt","r")
#running a for loop
print(fileptr)
for i in fileptr.read():
    print(i) # prints character by character

fileptr.seek(0)
for i in fileptr:
    print(i) #print line by line
```

```
<_io.TextIOWrapper name='demoText.txt' mode='r' encoding='cp1252'>
```

I

a

m

A

m

i

t

I am Amit

Program code 2 for Read File using For Loop:

1. `A = ["Hello\n", "Coders\n", "Parul MIS\n"]`
2. `f1 = open('myfile.txt', 'w')`
3. `f1.writelines(A)`
4. `f1.close()`
5. `f1 = open('myfile.txt', 'r')`
6. `Lines = f1.read()`
7. `count = 0`
8. `for line in Lines:`
9. `count += 1`
10. `print("Line{}: {}".format(count, line))`

Output:

Line1: H
Line2: e
Line3: l
Line4: l
Line5: o
Line6:
Line7: C
Line8: o
Line9: d
Line10: e
Line11: r
Line12: s
Line13:

```
A = ["Hello\n", "Coders\n", "Parul MIS\n"]

print("A=",A)
f1 = open('demoText.txt', 'w')
f1.writelines(A)
f1.close()
f1 = open('demoText.txt', 'r')
Lines = f1.read()
print(type(Lines)) #prints string object
print("data=",Lines) #prints all lines in file

for myChar in Lines:
    print(myChar)

print(f1) #file IO object
count = 0
f1.seek(0)
for line in f1:
    count += 1
    print("Line{}: {}".format(count, line))
```

Output:

A= ['Hello\n', 'Coders\n', 'Parul MIS\n']
<class 'str'>
data= Hello
Coders
Parul MIS

H
e

l
l
o

C
o
d
e
r
s

P
a
r
u
l

M
I
S

```
<_io.TextIOWrapper name='demoText.txt' mode='r' encoding='cp1252'>  
Line1: Hello
```

```
Line2: Coders
```

```
Line3: Parul MIS
```

Read Lines of the file

Python facilitates to read the file line by line by using a function **readline()** method. The **readline()** method reads the lines of the file from the beginning, i.e., if we use the **readline()** method two times, then we can get the first two lines of the file.

Consider the following example which contains a function **readline()** that reads the first line of our file **"file2.txt"** containing three lines. Consider the following example.

1. #open the file.txt in read mode. causes error if no such file exists.
2. fileptr = open("file2.txt","r");
3. #stores first line of the file into the variable content
4. content = fileptr.readline()

5. `content1 = fileptr.readline()` #stores second line of the file
6. `#prints the content of the file`
7. `print(content)` #prints first line
8. `print(content1)` #prints second line
9. `#closes the opened file`
10. `fileptr.close()`

Output:

Python is the modern day language.

It makes things so simple.

```
#open the file.txt in read mode. causes error if no such file exists.
fileptr = open("myfile.txt","r");
#stores all the data of the file into the variable content
oneLine = fileptr.readline()
print("oneLine=",oneLine)
secondLine = fileptr.readline()
#prints the content of the file

print(secondLine)
#closes the opened file
fileptr.close()
```

oneLine= Hello

Coders

```
#open the file.txt in read mode. causes error if no such file exists.
fileptr = open("myfile.txt","r")
#stores all the data of the file into the variable content
for line in fileptr.readlines():
    print(line)
```

H
e
l
l
o

We called the **readline()** function two times that's why it read two lines from the file. That means, if you called readline() function n times in your program, then it read n number of lines from the file. This is the uses of readline() function in Python.

```
#open the file.txt in read mode. causes error if no such file exists.
fileptr = open("myfile.txt","r")
#stores all the data of the file into the variable content
line=fileptr.readline()

while(line):
    print(line)
    line=fileptr.readline()
```

this is first line

this is line two

this is third line

Python provides **readlines()** method which is used for the reading lines. It returns the list of the lines till the end of **file(EOF)** is reached.

Example 2:

1. #open the file.txt in read mode. causes error if no such file exists.
2. fileptr = open("file.txt","r");
- 3.
4. #stores all the data of the file into the variable content
5. listOfLines= fileptr.readlines()
- 6.
7. #prints the content of the file
8. print(listOfLines)
- 9.
10. #closes the opened file
11. fileptr.close()

Output:

```
['Python is the modern day language.\n', 'It makes things so simple.\n',  
'Python has easy syntax and user-friendly interaction.']
```

Example 3:

Here we give the example of reading the lines using the `readlines()` function in Python. The code is given below -

1. `A = ["Hello\n", "Coders\n", "Parul MIS\n"]` #string of list
2. `f1 = open('myfile.txt', 'w')`
3. `f1.writelines(A)`
4. `f1.close()`
5. `f1 = open('myfile.txt', 'r')`
6. `Lines = f1.readlines()` #Lines is a list of Lines in the file
7. `count = 0`
8. `for line in Lines:` #Lines= list of Lines in the file
9. `count += 1`
10. `print("Line{}: {}".format(count, line))`

Output:

```
Line1: Hello  
Line2: Coders  
Line3: Parul MIS
```

Creating a new file

The new file can be created by using one of the following access modes with the function `open()`. The `open()` function used so many parameters. The syntax of it is given below -

```
file = open(path_to_file, mode)
```

x, a and w is the modes of `open()` function. The uses of these modes are given below

x: it creates a new file with the specified name. It causes an error if a file exists with the same name.

a: It creates a new file with the specified name if no such file exists. It appends the content to the file if the file already exists with the specified name.

w: It creates a new file with the specified name if no such file exists. It overwrites the existing file.

Consider the following example.

Program code1 for Creating a new file:

Here we give an example for creating a new file in Python. For creates a file, we have to use the open() method. The code is given below -

1. `#open the file.txt in read mode. causes error if no such file exists.`
2. `fileptr = open("file2.txt","x")`
3. `print(fileptr)`
4. `if fileptr:`
5. `print("File created successfully")`

Output:

Now we compile the above code in Python, and after successful compilation, we run it. Then the output is given below -

```
<_io.TextIOWrapper name='file2.txt' mode='x' encoding='cp1252'>
File created successfully
```

Program code2 for creating a new file with try-except:

```
try:
    fileptr=open("demoText.txt",'x')
except FileExistsError as error:
    print(str(error)+"\nFile already exists!")
```

[Errno 17] File exists: 'demoText.txt'

File already exists!

File Pointer positions

Python provides the tell() method which is used to print the byte number at which the file pointer currently exists. The tell() methods returns the position of read or write pointer in this file. The syntax of tell() method is given below -

1. `fileobject.tell()`

Program code1 for File Pointer Position:

1. `# open the file file2.txt in read mode`
2. `fileptr = open("file2.txt", "r")`
- 3.
4. `#initially the filepointer is at 0`
5. `print("The filepointer is at byte :",fileptr.tell())`
- 6.
7. `#reading the content of the file`
8. `content = fileptr.read();`
- 9.
10. `#after the read operation file pointer modifies. tell() returns the location of the fileptr.`
- 11.
12. `print("After reading, the filepointer is at:",fileptr.tell())`

Output:

```
The filepointer is at byte : 0
After reading, the filepointer is at: 117
```

Program code2 for File Pointer Position:

1. `file = open("File2.txt", "r")`
2. `print("The pointer position is: ", file.tell())`

Output:

```
The pointer position is: 0
```

Modifying file pointer position (seek())

In real-world applications, sometimes we need to change the file pointer location externally since we may need to read or write the content at various locations.

For this purpose, the Python provides us the `seek()` method which enables us to modify the file pointer position externally. That means, using `seek()` method we can easily change the cursor location in the file, from where we want to read or write a file.

Syntax:

The syntax for `seek()` method is given below -

1. <file-ptr>.seek(offset[, **from**])

The seek() method accepts two parameters:

Offset (MANDATORY): It refers to the new position of the file pointer within the file.

from: It indicates the reference position from where the bytes are to be moved. If it is set to 0, the beginning of the file is used as the reference position. **If it is set to 1, the current position of the file pointer is used as the reference position. If it is set to 2, the end of the file pointer is used as the reference position.**

Consider the following example.

Here we give the example of how to modifying the pointer position using seek() method in Python. The code is given below -

1. # open the file file2.txt in read mode
2. fileptr = open("file2.txt","r")
- 3.
4. #initially the filepointer is at 0
5. **print**("The filepointer is at byte :",fileptr.tell())
- 6.
7. #changing the file pointer location to 10.
8. fileptr.seek(10);
- 9.
10. #tell() returns the location of the fileptr.
11. **print**("After reading, the filepointer is at:",fileptr.tell())

Output:

Now we compile the above code in Python, and after successful compilation, we run it. Then the output is given below -

```
The filepointer is at byte : 0
After reading, the filepointer is at: 10
```

```
file = open("demoText.txt", "r")
print(file.read())

file.seek(0)
print(file.read(1))
print("The pointer position is: ", file.tell())
```



```
file.seek(11,0)
print(file.read(1))
print("The pointer position is: ", file.tell())

file.seek(22,0)
print(file.read(1))
print("The pointer position is: ", file.tell())
```

```
123456789
abcdefghi
ABCDEFGHI
1
The pointer position is:  1
a
The pointer position is:  12
A
The pointer position is:  23
```

Python OS module:

Renaming the file

The Python **os** module enables interaction with the operating system. It comes from the Python standard utility module. The os module provides a portable way to use the operating system-dependent functionality in Python. The os module provides the functions that are involved in file processing operations like renaming, deleting, etc. It provides us the `rename()` method to rename the specified file to a new name. Using the `rename()` method, we can easily rename the existing File. This method has not any return value. The syntax to use the **rename()** method is given below.

Syntax:

The syntax of rename method in Python is given below -

1. `rename(current-name, new-name)`

The first argument is the current file name and the second argument is the modified name. We can change the file name by passing these two arguments.

Program code 1 for `rename()` Method:

Here we give an example of the renaming of the files using rename() method in Python. The current file name is file2.txt, and the new file name is file3.txt. The code is given below -

1. **import** os
- 2.
3. **#rename file2.txt to file3.txt**
4. os.rename("file2.txt", "file3.txt")

```
from os import *  
  
#rename file2.txt to file3.txt  
rename("demoText.txt", "mydemoText.txt")
```

Removing the file

The os module provides the **remove()** method which is used to remove the specified file.

Syntax:

The syntax of remove method is given below -

1. remove(file-name)

Program code 1 for remove() method:

1. **import** os;
2. **#deleting the file named file3.txt**
3. os.remove("file3.txt")

| Method | Description |
|----------------------------|---|
| file.close() | It closes the opened file. The file once closed, it can't be read or write anymore. |
| File.next() | It returns the next line from the file. |
| File.read([size]) | It reads the file for the specified size. |
| File.readline([size]) | It reads one line from the file and places the file pointer to the beginning of the new line. |
| File.readlines([sizehint]) | It returns a list containing all the lines of the file. It reads the file until the EOF occurs using readline() function. |
| File.seek(offset[,from]) | It modifies the position of the file pointer to a specified offset with the specified reference. |
| File.tell() | It returns the current position of the file pointer within the file. |
| File.truncate([size]) | It truncates the file to the optional specified size or the current cursor position, remaining data will be truncated. |
| File.write(str) | It writes the specified string to a file |
| File.writelines(seq) | It writes a sequence of the strings to a file. |

Python Exceptions

What is an Exception?

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. Consider the following scenario:

Code

1. `#Python code after removing the syntax error`
2. `string = "Python Exceptions"`
- 3.
4. `for s in string:`
5. `if (s != o:`
6. `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:

Code

1. `#Python code after removing the syntax error`
2. `string = "Python Exceptions"`
- 3.
4. `for s in string:`
5. `if (s != o): #this is o not zero`
6. `print(s)`

Output:

```
2 string = "Python Exceptions"
4 for s in string:
----> 5     if (s != o):
      6         print( s )
```

```
NameError: name 'o' is not defined
```

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.

Code

1. **# Python code to catch an exception and handle it using try and except code blocks**
- 2.
3. **a = ["Python", "Exceptions", "try and except"]**
4. **try:**
5. **#looping through the elements of the array a, choosing a range that goes beyond the length of the array**
6. **for i in range(4):**
7. **print("The index and element from the array is", i, a[i])**
8. **#if an error occurs in the try block, then except block will be executed by the Python interpreter**
9. **except:**
10. **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
```

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

we can intentionally raise an exception using the `raise` keyword. If we wish to use `raise` to generate an exception when a given condition happens, we may do so as follows:

Code

1. `#Python code to show how to raise an exception in Python`
2. `num = [3, 4, 5, 7]`
3. `if len(num) > 3:`
4. `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
```

The implementation stops and shows our exception in the output, providing indications as to what went incorrect.

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.

Try with Else Clause

Python also supports the else clause, which should come after except clauses, 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.

Code

```
1. # Python program to show how to use else clause with try and except clauses
2.
3. # Defining a function which returns reciprocal of a number
4. def reciprocal( num1 ):
5.     try:
6.         reci = 1 / num1
7.     except ZeroDivisionError:
8.         print( "We cannot divide by zero" )
9.     else:
10.        print ( reci )
11. # Calling the function and passing values
12. reciprocal( 4 )
13. reciprocal( 0 )
```

Output:

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

| | |
|--|---|
| <pre> 1 def reciprocal(num1): 2 try: 3 reci = 1 / num1 4 5 except ZeroDivisionError: 6 print("We cannot divide by zero") 7 8 except TypeError: 9 print("We cannot divide by character") 10 else: 11 print (reci) 12 reciprocal(4) 13 reciprocal('a') 14 reciprocal(0) </pre> | <pre> 0.25 We cannot divide by character We cannot divide by zero > </pre> |
|--|---|

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:

Code

1. # Python code to show the use of finally clause
- 2.
3. # Raising an exception in try block
4. **try:**
5. div = 4 / 0
6. **print**(div)
7. # this block will handle the exception raised
8. **except** ZeroDivisionError:
9. **print**("Atepting to divide by zero")
10. # this will always be executed no matter exception is raised or not
11. **finally:**
12. **print**('This is code of finally clause')

Output:

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

```
1 def reciprocal( num1 ):  
2     try:  
3         reci = 1 / num1  
4  
5  
6     except ZeroDivisionError:  
7         print( "We cannot divide by zero" )  
8  
9     except TypeError:  
10        print("We cannot divide by character")  
11    else:  
12        print ( reci )  
13    finally:  
14        print("I am in finally")  
15  
16  
17  
18 reciprocal(4)  
19 reciprocal('a')  
20 reciprocal(0)
```

0.25
I am in finally
We cannot divide by character
I am in finally
We cannot divide by zero
I am in finally
>

User-Defined Exceptions

By inheriting classes from the typical built-in exceptions, Python also lets us design our customized exceptions.

HOME WORK FOR STUDENTS:

RAISE A USER-DEFINED EXCPETION (WRITE A PROGRAM FOR THAT)

```

1 def reciprocal( num1 ):
2     try:
3         reci = 1 / num1
4
5
6     except ZeroDivisionError:
7         print( ZeroDivisionError )
8
9     except TypeError:
10        print(TypeError)
11    else:
12        print ( reci )
13    finally:
14        print("I am in finally")
15
16 reciprocal(4)
17 reciprocal('a')
18 reciprocal(0)

```

```

0.25
I am in finally
<class 'TypeError'>
I am in finally
<class 'ZeroDivisionError'>
I am in finally
>

```

```

1 def reciprocal( num1 ):
2     try:
3         reci = 1 / num1
4
5
6     except Exception as ZeroError:
7         print( ZeroError )
8
9     except Exception as AlphaError:
10        print(AlphaError)
11    else:
12        print ( reci )
13    finally:
14        print("I am in finally")
15
16 reciprocal(4)
17 reciprocal('a')
18 reciprocal(0)

```

```

0.25
I am in finally
unsupported operand type(s) for /: 'int' and 'str'
I am in finally
division by zero
I am in finally
>

```

Python Exceptions List

Here is the complete list of Python in-built exceptions.

| Sr.No. | Name of the Exception | Description of the Exception |
|--------|---------------------------|---|
| 1 | Exception | All exceptions of Python have a base class. |
| 2 | StopIteration | If the next() method returns null for an iterator, this exception is raised. |
| 3 | SystemExit | The sys.exit() procedure raises this value. |
| 4 | StandardError | Excluding the StopIteration and SystemExit, this is the base class for all Python built-in exceptions. |
| 5 | ArithmeticError | All mathematical computation errors belong to this base class. |
| 6 | OverflowError | This exception is raised when a computation surpasses the numeric data type's maximum limit. |
| 7 | FloatingPointError | If a floating-point operation fails, this exception is raised. |
| 8 | ZeroDivisionError | For all numeric data types, its value is raised whenever a number is attempted to be divided by zero. |
| 10 | AttributeError | This exception is raised if a variable reference or assigning a value fails. |
| 12 | ImportError | This exception is raised if using the import keyword to import a module fails. |
| 13 | KeyboardInterrupt | If the user interrupts the execution of a program, generally by hitting Ctrl+C, this exception is raised. |
| 14 | LookupError | LookupErrorBase is the base class for all search errors. |
| 15 | IndexError | This exception is raised when the index attempted to be accessed is not found. |
| 16 | KeyError | When the given key is not found in the dictionary to be found in, this exception is raised. |
| 17 | NameError | This exception is raised when a variable isn't located in either local or global namespace. |

| | | |
|----|----------------------------|---|
| 18 | UnboundLocalError | This exception is raised when we try to access a local variable inside a function, and the variable has not been assigned any value. |
| 19 | EnvironmentError | All exceptions that arise beyond the Python environment have this base class. |
| 20 | IOError | If an input or output action fails, like when using the print command or the open() function to access a file that does not exist, this exception is raised. |
| 22 | SyntaxError | This exception is raised whenever a syntax error occurs in our program. |
| 23 | IndentationError | This exception was raised when we made an improper indentation. |
| 24 | SystemExit | This exception is raised when the sys.exit() method is used to terminate the Python interpreter. |
| 25 | TypeError | This exception is raised whenever a data type-incompatible action or function is tried to be executed. For example divide 5/'a' |
| 26 | ValueError | This exception is raised if the parameters for a built-in method for a particular data type are of the correct type but have been given the wrong values. For example, window.geography(300) |
| 27 | RuntimeError | This exception is raised when an error that occurred during the program's execution cannot be classified. |
| 28 | NotImplementedError | If an abstract function that the user must define in an inherited class is not defined, this exception is raised. |

Summary

- We can throw an exception at any line of code using the raise keyword.
- All statements are carried out in the try clause until an exception is found.
- The try clause's exception(s) are detected and handled using the except function.

- If no exceptions are thrown in the try code block, we can write code to be executed in the else code block.

Here is the syntax of try, except, else, and finally clauses.

Syntax:

1. **try:**
2. # Code block
3. # These statements are those which can probably have some error
- 4.
5. **except:**
6. # except is optional if finally is there
7. # If the try block encounters an exception, this block will handle it.
- 8.
9. **else:**
10. # If there is no exception, this code block will be executed by the Python interpreter
- 11.
12. **finally:**
13. # Python interpreter will always execute this code.

EXCEPT BLOCK IS OPTIONAL IF FINALLY IS THERE

```
1 def reciprocal( num1 ):
2     try:
3         reci = 1 / num1
4     finally:
5         print("hello")
6
7 reciprocal(4)
```

hello

> |

OOPS

Python OOPs Concepts

Like other general-purpose programming languages, Python is also an object-oriented language since its beginning. It allows us to develop applications using an Object-Oriented approach. In [Python](#), we can easily create and use classes and objects.

An object-oriented paradigm is to design the program using classes and objects. The object is related to real-world entities such as book, house, pencil, etc. The oops concept focuses on writing the reusable code. Major principles of object-oriented programming system are given below.

- Class
- Object
- Method
- Inheritance
- Polymorphism
- Data Abstraction
- Encapsulation

Class

The class can be defined as a collection of objects. It is a logical entity that has some specific attributes and methods. For example: if you have an employee class, then it should contain an attributes (class variables) and method (class functions), i.e. an employee id, name, age, salary, etc.

Syntax

1. **class** ClassName:
2. <statement-1>
3. .
4. .
5. <statement-N>

Object

It may be any real-world object like the mouse, keyboard, chair, table, pen, etc.

Everything in Python is an object, and almost everything has attributes and methods. Consider the following example.

Example:

1. **class** car:
2. **def** __init__(self,modelname, year):
3. self.modelname = modelname
4. self.year = year
5. **def** display(self):
6. **print**(self.modelname,self.year)
- 7.
8. c1 = car("Toyota", 2016)
9. c1.display()

Output:

```
Toyota 2016
```

In the above example, we have created the class named car, and it has two attributes modelname and year. We have created a c1 object to access the class attributes. The c1 object will allocate memory for these values.

Method

The method is a function that is associated with an object.

Inheritance

Inheritance is the most important aspect of object-oriented programming, which simulates the real-world concept of inheritance. It specifies that the child object acquires all the properties and behaviors of the parent object.

By using inheritance, we can create a class which uses all the properties and behavior of another class. The new class is known as a derived class or child class, and the one whose properties are acquired is known as a base class or parent class.

It provides the re-usability of the code.

Polymorphism

Polymorphism contains two words "poly" and "morphs". Poly means many, and morph means shape. By polymorphism, we understand that one task can be performed in different ways. For example - you have a class animal, and all animals speak. But they speak differently. Here, the "speak" behavior is polymorphic in a sense and depends on the animal. So, the abstract "animal" concept does not actually "speak", but specific animals (like dogs and cats) have a concrete implementation of the action "speak".

Encapsulation

Encapsulation is also an essential aspect of object-oriented programming. It is used to restrict access to methods and variables. In encapsulation, code and data are wrapped together within a single unit from being modified by accident.

Data Abstraction

Data abstraction and encapsulation both are often used as synonyms. Both are nearly synonyms because data abstraction is achieved through encapsulation.

Abstraction is used to hide internal details and show only functionalities.

Object-oriented vs. Procedure-oriented Programming languages and their examples (Home-work for students)

Classes and Objects in Python

Python is an object-oriented programming language that offers classes, which are tools for writing reusable code.

Classes in Python:

In Python, a class is a user-defined data type that contains both the data itself and the methods that may be used to manipulate the data. In a sense, classes serve as a template to create objects. They provide the characteristics and operations that the objects will employ.

Suppose a class is a prototype of a building. A building contains all the details about the floor, rooms, doors, windows, etc. we can make as many buildings as we want, based on these details. Hence, the building can be seen as a class, and we can create as many objects of this class.

Creating Classes in Python

In Python, a class can be created by using the keyword `class`, followed by the class name. The syntax to create a class is given below.

Syntax

1. `class` ClassName:
2. `#statement_suite`

A class contains a statement suite including fields, constructor, function, etc. definition.

Example:

Code:

1. `class` Person:
2. `def` `__init__`(self, name, age):
3. `# This is the constructor method that is called when creating a new Person object`
4. `# It takes two parameters, name and age, and initializes them as attributes of the object`
5. `self.name = name`
6. `self.age = age`
7. `def` `greet`(self):
8. `# This is a method of the Person class that prints a greeting message`
9. `print("Hello, my name is " + self.name)`

Name and age are the two properties of the Person class. Additionally, it has a function called `greet` that prints a greeting.

Objects in Python:

An object is a particular instance of a class with unique characteristics and functions. After a class has been established, you may make objects based on it. By using the class constructor, you may create an object of a class in Python. The object's attributes are initialised in the constructor, which is a special procedure with the name `__init__`.

Syntax:

1. `# Declare an object of a class`
2. `object_name = Class_Name(arguments)`

Example:

Code:

1. **class** Person:
2. **def** __init__(self, name, age):
3. self.name = name
4. self.age = age
5. **def** greet(self):
6. **print**("Hello, my name is " + self.name)
- 7.
8. # Create a new instance of the Person class and assign it to the variable person1
9. person1 = Person("Ayan", 25)
10. person1.greet()

Output:

```
"Hello, my name is Ayan"
```

The self-parameter

The self-parameter refers to the current instance of the class and accesses the class variables. **We can use anything instead of self, but it must be the first parameter of any function which belongs to the class.**

__init__ method

In order to make an instance of a class in Python, a specific function called `__init__` is called. Although it is used to set the object's attributes, it is often referred to as a constructor.

The self-argument is the only one required by the `__init__` method. This argument refers to the newly generated instance of the class. To initialise the values of each attribute associated with the objects, you can declare extra arguments in the `__init__` method.

Class and Instance Variables

All instances of a class share class variables. They function independently of any class methods and may be accessed through the use of the class name. Here's an illustration:

Code:

```
1. class Person:
2.     count = 0 # This is a class variable
3.
4.     def __init__(self, name, age):
5.         self.name = name # This is an instance variable
6.         self.age = age
7.         Person.count += 1 # Accessing the class variable using the name of the class
8. person1 = Person("Ayan", 25)
9. person2 = Person("Bobby", 30)
10. print(Person.count)
```

Output:

```
2
```

Whereas, instance variables are specific to each instance of a class. They are specified using the self-argument in the `__init__` method. Here's an illustration:

Code:

```
1. class Person:
2.     def __init__(self, name, age):
3.         self.name = name # This is an instance variable
4.         self.age = age
5. person1 = Person("Ayan", 25)
6. person2 = Person("Bobby", 30)
7. print(person1.name)
8. print(person2.age)
```

Output:

```
Ayan
30
```

Class variables are created separately from any class methods and are shared by all class copies/objects. Every instance of a class has its own instance variables, which are specified in the `__init__` method utilising the self-argument.

Python Constructor

A constructor is a special type of method (function) which is used to initialize the instance members (object variables) of the class.

In C++ or Java, the constructor has the same name as its class, but it treats constructor differently in Python. It is used to create an object.

Constructors can be of two types.

- 1. Parameterized Constructor**
- 2. Non-parameterized Constructor**

Constructor definition is executed when we create the object of this class.

Creating the constructor in python

In Python, the method the `__init__()` simulates the constructor of the class. This method is called when the object is instantiated. **It accepts the `self`-keyword as a first argument which allows accessing the attributes (variables) or methods of the class.**

We can pass any number of arguments at the time of creating the class object, depending upon the `__init__()` definition. **It is mostly used to initialize the class attributes (variables). Every class must have a constructor, otherwise it simply relies on the default constructor provided by the system/Python.**

Consider the following example to initialize the **Employee** class attributes.

Example

- class** Employee:
- def** `__init__`(self, name, id):
- self.id = id
- self.name = name
-
- def** display(self):
- print**("ID: %d \nName: %s" % (self.id, self.name))
-
-
-
- emp1 = Employee("John", 101)

```
11. emp2 = Employee("David", 102)
12.
13. # accessing display() method to print employee 1 information
14.
15. emp1.display()
16.
17. # accessing display() method to print employee 2 information
18. emp2.display()
```

Output:

```
ID: 101
Name: John
ID: 102
Name: David
```

Counting the number of objects of a class

The constructor is called automatically when we create the object of the class. Consider the following example.

Example

```
1. class Student:
2.     count = 0 #this is class variable
3.     def __init__(self):
4.         Student.count = Student.count + 1 #class variable accessed using class
       name
5. s1=Student()
6. s2=Student()
7. s3=Student()
8. print("The number of students:",Student.count)
```

Output:

```
The number of students: 3
```

Python Non-Parameterized Constructor

The non-parameterized constructor uses when we do not want to manipulate the value or the constructor that has only self as an argument.

NO MULTIPLE CONSTRUCTORS WITH THE SAME NAMES ALLOWED

Python does not support explicit multiple constructors, yet there are some ways using which multiple constructors can be achieved. If multiple `__init__` methods are written for the same class, then the latest one overwrites all the previous constructors and the reason for this can be, python stores all the function names in a class as key in a dictionary so, when a new function is defined with the same name, the key remains the same but the value gets overridden by the new function body.

```
class example:

    def __init__(self):

        print("One")

    def __init__(self):

        print("Two")

    def __init__(self):

        print("Three")

e = example()
```

Output:

Three

```

1 class Student:
2     # Constructor - non parameterized
3     def __init__(self):
4         print("This is non parametrized constructor")
5     def __init__(self,name):
6         print("Hello",name)
7 student = Student()
8 student2=Student("John")

```

```

ERROR!
Traceback (most recent call last):
  File "<string>", line 7, in <module>
TypeError: Student.__init__() missing 1 required positional argument:
    'name'
>

```

Python *args and **kwargs

Suppose, we define a function for addition of 3 numbers.

Example 1: Function to add 3 numbers

```

def adder(x,y,z):
    print("sum:",x+y+z)

adder(10,12,13)

```

When we run the above program, the output will be

```
sum: 35
```

In above program we have `adder()` function with three arguments `x`, `y` and `z`. When we pass three values while calling `adder()` function, we get sum of the 3 numbers as the output.

Lets see what happens when we pass more than 3 arguments in the `adder()` function.


```
def adder(x,y,z):  
    print("sum:",x+y+z)  
  
adder(5,10,15,20,25)
```

When we run the above program, the output will be

```
TypeError: adder() takes 3 positional arguments but 5 were given
```

In the above program, we passed 5 arguments to the `adder()` function instead of 3 arguments due to which we got `TypeError`.

Introduction to *args and **kwargs in Python

In Python, we can pass **a variable number of arguments** to a function using special symbols. There are two special symbols:

1. *args (Non Keyword Arguments)
2. **kwargs (Keyword Arguments)

We use `*args` and `kwargs` as an argument when we are unsure about the number of arguments to pass in the functions.**

Python *args

As in the above example we are not sure about the number of arguments that can be passed to a function. Python has `*args` which allow us to pass the variable number of non keyword arguments to function.

In the function, we should use an asterisk `*` before the parameter name to pass variable length arguments.

Parameter name could be any thing but should have `*` before the name. The arguments are passed as a **tuple** and these passed arguments make tuple inside the function with same name as the parameter excluding asterisk `*`.

Example 2: Using `*args` to pass the variable length arguments to the function

```
def adder(*num): # NOW NUM IS A TUPLE
    sum = 0

    for n in num:
        sum = sum + n

    print("Sum:",sum)

adder(3,5)
adder(4,5,6,7)
adder(1,2,3,5,6)
```

When we run the above program, the output will be

```
Sum: 8
Sum: 22
Sum: 17
```

In the above program, we used `*num` as a parameter which allows us to pass variable length argument list to the `adder()` function. Inside the function, we have a loop which adds the passed argument and prints the result.

Python **kwargs

Python passes variable length non keyword argument to function using `*args` but we cannot use this to pass keyword argument. For this problem Python has got a solution called `**kwargs`, it allows us to pass the variable length of **keyword** arguments to the function.

In the function, we use the double asterisk `**` before the parameter name to denote this type of argument. **The arguments are passed as a dictionary and these arguments make a dictionary inside function with name same as the parameter** excluding double asterisk `**`.

Example 3: Using **kwargs to pass the variable keyword arguments to the function

Note: keyword arguments mean `fun(name="amit")`

Non-keyword arguments mean `fun("amit")`

```
def intro(**data):
    print("\nData type of argument:",type(data))

    for key, value in data.items():
        print("{} is {}".format(key,value))

intro(Firstname="Sita", Lastname="Sharma", Age=22, Phone=1234567890)
intro(Firstname="John", Lastname="Wood", Email="johnwood@nomail.com",
Country="Wakanda", Age=25, Phone=9876543210)
```

When we run the above program, the output will be

```
Data type of argument: <class 'dict'>
Firstname is Sita
Lastname is Sharma
Age is 22
Phone is 1234567890

Data type of argument: <class 'dict'>
Firstname is John
Lastname is Wood
Email is johnwood@nomail.com
Country is Wakanda
```

```
Age is 25
Phone is 9876543210
```

In the above program, we have a function `intro()` with `**data` as a parameter. We have for loop inside `intro()` function which works on the data of passed as dictionary and prints the value of the dictionary.

Things to Remember:

- `*args` and `**kwargs` allow function to take variable length argument.
- `*args` passes variable number of **non-keyworded** arguments and on which operation of the **tuple** can be performed.
- `**kwargs` passes variable number of **keyword** arguments **dictionary** to function on which operation of a dictionary can be performed.

| | |
|---|---|
| <pre>1 class Student: 2 # Constructor - non parameterized 3 def __init__(self, *varArgs): 4 print("This is non parametrized constructor") 5 i=0 6 print("len=",len(varArgs)) 7 while i<len(varArgs): 8 print(varArgs[i]) 9 i=i+1 10 student = Student() 11 student2=Student("John") 12 student3=Student(1,2,3,1.10,"amit",'h',[10,11,12]) 13</pre> | <pre>This is non parametrized constructor len= 0 This is non parametrized constructor len= 1 John This is non parametrized constructor len= 7 1 2 3 1.1 amit h [10, 11, 12]</pre> |
|---|---|

| | |
|--|--|
| <pre> 1 class Student: 2 # Constructor - non parameterized 3 def __init__(self): 4 print("This is non parametrized constructor") 5 def diplay(self): 6 print("display") 7 8 def display(self,number): 9 print(number) 10 11 student = Student() 12 student.display() 13 student.display(10) </pre> | <pre> This is non parametrized constructor ERROR! Traceback (most recent call last): File "<string>", line 12, in <module> TypeError: Student.display() missing 1 required positional argument: 'number' > </pre> |
|--|--|

Python Parameterized Constructor

The parameterized constructor has multiple parameters along with the **self**. Consider the following example.

Example

1. **class** Student:
2. **# Constructor** - parameterized
3. **def** __init__(self, name):
4. **print**("This is parametrized constructor")
5. self.name = name
6. student = Student("John")

Output:

```

This is parametrized constructor
Hello John

```

Python Default Constructor

When we do not include the constructor in the class or forget to declare it, then default constructor is provided by the system. It does not perform any task but initializes the objects. Consider the following example.

Example

1. **class** Student:
2. roll_num = 101




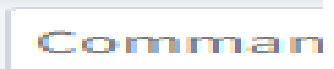
```
3.     name = "Joseph"
4.
5.     def display(self):
6.         print(self.roll_num,self.name)
7.
8. st = Student()
9. st.display()
```

Output:

```
101 Joseph
```

```
1 class Student:
2     roll=10
3
4     def disp(self):
5         print(self.roll)
6     def disp2(self):
7         print(self.roll)
8
9 stud=Student()
10 stud.disp2()
11 stud.disp()
```

Ln: 7, Col: 20

 Run  Share  \$  Command

```
10
10
```

More than One Constructor in Single class

Let's have a look at another scenario, what happen if we declare the two same constructors in the class.

Example

1. `class Student:`
2. `def __init__(self):`
3. `print("The First Constructor")`
4. `def __init__(self):`
5. `print("The second constructor")`
- 6.
7. `st = Student()`

Output:

```
The Second Constructor
```

In the above code, the object **st** called the second constructor whereas both have the same configuration. The first method is not accessible by the **st** object. **Internally, the object of the class will always call the last constructor if the class has multiple constructors.**

Note: The constructor overloading is not allowed in Python.

Python built-in class functions

The built-in functions defined in the class are described in the following table.

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

Example

1. `class Student:`
2. `def __init__(self, name, id, age):`
3. `self.name = name`
4. `self.id = id`
5. `self.age = age`

```
6.
7.     # creates the object of the class Student
8. s = Student("John", 101, 22)
9.
10. # prints the attribute name of the object s
11. print(getattr(s, 'name'))
12.
13. # reset the value of attribute age to 23
14. setattr(s, "age", 23)
15.
16. # prints the modified value of age
17. print(getattr(s, 'age'))
18.
19. # prints true if the student contains the attribute with name as id
20.
21. print(hasattr(s, 'id'))
22. # deletes the attribute age
23. delattr(s, 'age')
24.
25. # this will give an error since the attribute age has been deleted
26. print(s.age)
```

Output:

```
John
23
True
AttributeError: 'Student' object has no attribute 'age'
```

The __str__() Function

The __str__() function controls what should be returned when the class object is represented as a string.

For example, print("hi" + str(StudentObject)) #here __str__(self) is called

If the __str__() function is not set, the string representation of the object is returned:

Example

The string representation of an object WITHOUT the `__str__()` function:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

p1 = Person("John", 36)

print(p1)
```

```
<__main__.Person object at 0x15039e602100>
```

Example

The string representation of an object WITH the `__str__()` function:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __str__(self):
        return f"{self.name}({self.age})"

p1 = Person("John", 36)

print(p1)
```

```
John(36)
```

```
1 class Student:
2     def __str__(self):
3         return "amit"
4
5 stud=Student()
6 print(str(stud))
7
8
```

Ln: 6, Col: 17

 Run

 Share

\$

Com



amit

```
1 class Student:
2     def __str__(self):
3         return "amit"
4
5 stud=Student()
6 print("hi ",stud)
7
8
```

Ln: 6, Col: 18

 Run

 Share

\$

Co



hi amit



```
1 class Student:
2     def __str__(self):
3         return "amit"
4
5 stud=Student()
6 print("hi "+stud)
7
8
```

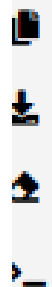
Ln: 6, Col: 13

 Run

 Share

 \$

Command Line Arguments



Traceback (most recent call last):





File "main.py", line 6, in <module>


print("hi "+stud)

TypeError: can only concatenate str (not "Student") to str

```
1 class Student:
2     def __str__(self):
3         return "amit"
4
5 stud=Student()
6 print("hi "+str(stud))
7
```

Ln: 6, Col: 22

 Run  Share  \$  Comm

 hi amit

Built-in class attributes

Along with the other attributes, a Python class also contains some built-in class attributes which provide information about the class.

The built-in class attributes are given in the below table.

| 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>__name__</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. |

Example

```
1. class Student:
2.     def __init__(self,name,id,age):
3.         self.name = name;
4.         self.id = id;
5.         self.age = age
6.     def display_details(self):
7.         print("Name:%s, ID:%d, age:%d"%(self.name,self.id,self.age))
8. s = Student("John",101,22)
9. print(s.__doc__)
10. print(s.__dict__)
11. print(s.__module__)
```

Output:

```
None
{'name': 'John', 'id': 101, 'age': 22}
__main__
```

```
1 class Student:
2     def __init__(self, name):
3         self.name = name
4
5 stud = Student("amit")
6 stud1 = Student("patel")
7 print(stud.__dict__)
8 print(stud1.__dict__)
9
10 print(stud.__doc__)
11 print(stud1.__doc__)
12 print(stud.__module__)
13 print(stud1.__module__)
14
```

n: 11, Col: 25

Run Share \$ Command Line

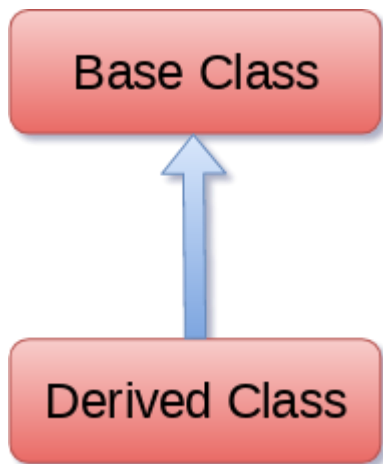
```
{'name': 'amit'}
{'name': 'patel'}
None
None
__main__
__main__
```

Python Inheritance

Inheritance is an important aspect of the object-oriented paradigm. Inheritance provides code reusability to the program because we can use an existing class to create a new class instead of creating it from scratch.

In inheritance, the child class acquires the properties and can access all the data members and functions defined in the parent class. A child class can also provide its specific implementation to the functions of the parent class.

In python, a derived class can inherit base class by just mentioning the base in the bracket after the derived class name. Consider the following syntax to inherit a base class into the derived class.



Syntax

1. **class** derived-**class**(base **class**):
2. <**class**-suite>

A class can inherit multiple classes by mentioning all of them inside the bracket. Consider the following syntax.

Syntax

1. **class** derive-**class**(<base **class** 1>, <base **class** 2>, <base **class** n>):
2. <**class** - suite>

Example 1

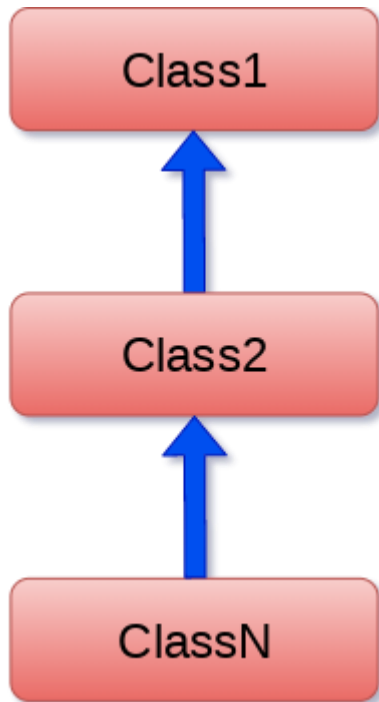
1. **class** Animal:
2. **def** speak(self):
3. **print**("Animal Speaking")
4. *#child class Dog inherits the base class Animal*
5. **class** Dog(Animal):
6. **def** bark(self):
7. **print**("dog barking")
8. d = Dog()
9. d.bark()
10. d.speak()

Output:

```
dog barking
```

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.



The syntax of multi-level inheritance is given below.

Syntax

1. **class** class1:
2. <**class**-suite>
3. **class** class2(class1):
4. <**class** suite>
5. **class** class3(class2):
6. <**class** suite>
7. .
8. .

Example

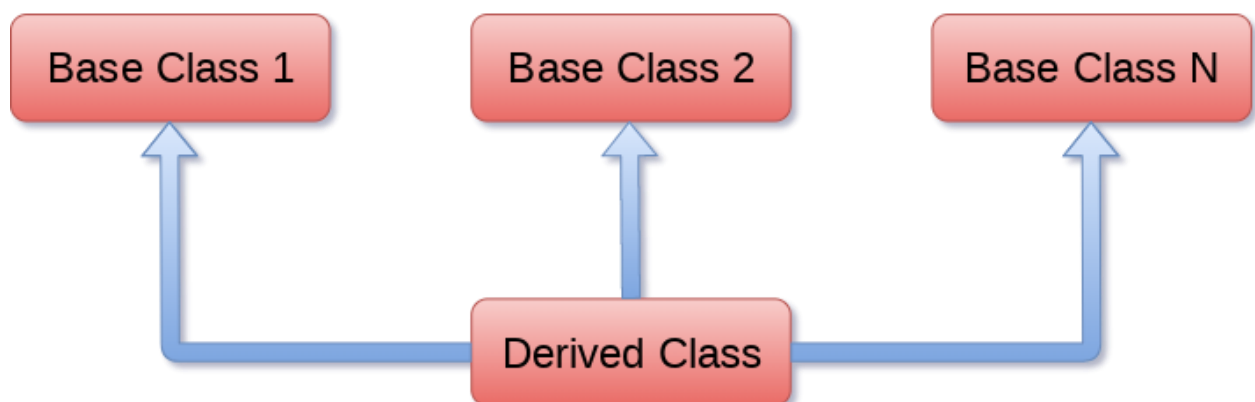
```
1. class Animal:
2.     def speak(self):
3.         print("Animal Speaking")
4. #The child class Dog inherits the base class Animal
5. class Dog(Animal):
6.     def bark(self):
7.         print("dog barking")
8. #The child class Dogchild inherits another child class Dog
9. class DogChild(Dog):
10.    def eat(self):
11.        print("Eating bread...")
12. d = DogChild()
13. d.bark()
14. d.speak()
15. 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.



The syntax to perform multiple inheritance is given below.

Syntax

1. **class** Base1:
2. <**class**-suite>
- 3.
4. **class** Base2:
5. <**class**-suite>
6. .
7. .
8. .
9. **class** BaseN:
10. <**class**-suite>
- 11.
12. **class** Derived(Base1, Base2, BaseN):
13. <**class**-suite>

Example

1. **class** Calculation1:
2. **def** Summation(self,a,b):
3. **return** a+b;
4. **class** Calculation2:
5. **def** Multiplication(self,a,b):
6. **return** a*b;
7. **class** Derived(Calculation1,Calculation2):
8. **def** Divide(self,a,b):
9. **return** a/b;
10. d = Derived()
11. **print**(d.Summation(10,20))
12. **print**(d.Multiplication(10,20))
13. **print**(d.Divide(10,20))

Output:

```
30
200
0.5
```

The issubclass(sub,sup) method

The issubclass(sub, sup) method is used to check the relationships between the specified classes. **It returns true if the first class is the subclass of the second class, and false otherwise.**

Consider the following example.

Example

1. **class** Calculation1:
2. **def** Summation(self,a,b):
3. **return** a+b;
4. **class** Calculation2:
5. **def** Multiplication(self,a,b):
6. **return** a*b;
7. **class** Derived(Calculation1,Calculation2):
8. **def** Divide(self,a,b):
9. **return** a/b;
10. d = Derived()
11. **print**(issubclass(Derived,Calculation2))
12. **print**(issubclass(Calculation1,Calculation2))

Output:

```
True
False
```

The isinstance (obj, class) method

The isinstance() method is used to check the relationship between the objects and classes. **It returns true if the first parameter, i.e., obj is the instance of the second parameter, i.e., class.**

Consider the following example.

Example

1. **class** Calculation1:
2. **def** Summation(self,a,b):

```
3.     return a+b;
4. class Calculation2:
5.     def Multiplication(self,a,b):
6.         return a*b;
7. class Derived(Calculation1,Calculation2):
8.     def Divide(self,a,b):
9.         return a/b;
10. d = Derived()
11. print(isinstance(d,Derived))
```

Output:

```
True
```

Method Overriding

We can provide some specific implementation of the parent class methods in our child class. When the parent class method is defined in the child class with some specific implementation, then the concept is called method overriding. **We may need to perform method overriding in the scenario where the different definition of a parent class method is needed in the child class.**

Consider the following example to perform method overriding in python.

Example

```
1. class Animal:
2.     def speak(self):
3.         print("speaking")
4. class Dog(Animal):
5.     def speak(self):
6.         print("Barking")
7. d = Dog()
8. d.speak()
```

Output:

```
Barking
```

Real Life Example of method overriding

```
1. class Bank:
2.     def getroi(self):
3.         return 10;
4. class SBI(Bank):
5.     def getroi(self):
6.         return 7;
7.
8. class ICICI(Bank):
9.     def getroi(self):
10.        return 8;
11. b1 = Bank()
12. b2 = SBI()
13. b3 = ICICI()
14. print("Bank Rate of interest:",b1.getroi());
15. print("SBI Rate of interest:",b2.getroi());
16. print("ICICI Rate of interest:",b3.getroi());
```

Output:

```
Bank Rate of interest: 10
SBI Rate of interest: 7
ICICI Rate of interest: 8
```

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.**

Consider the following example.

Example

```
1. class Employee:
2.     __count = 0;
3.     def __init__(self):
4.         Employee.__count = Employee.__count+1
5.     def display(self):
6.         print("The number of employees",Employee.__count)
```

7. emp = Employee()
8. emp2 = Employee()
9. **try**:
10. **print(emp.__count)** #error # Note: Except or Finally, at least any one is mandatory after try
11. **finally**:
12. 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.

That is exactly the abstraction that works in the [object-oriented concept](#).

Why Abstraction is Important?

In Python, an abstraction is used to hide the irrelevant data/class in order to reduce the complexity.

Abstraction classes in Python

In [Python](#), abstraction can be achieved by using abstract classes and interfaces.

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. Python provides the `abc` module to use the abstraction in the Python program. Let's see the following syntax.

Syntax

1. from abc **import** ABC
2. **class** ClassName(ABC):
3. **pass**

We import the ABC class from the abc module.

Abstract Base Classes

An abstract base class is the common application program of the interface for a set of subclasses. It can be used by the third-party, which will provide the implementations such as with plugins.

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 -

- 1.
2. from abc **import** ABC, abstractmethod
3. **class** Car(ABC):
4. def mileage(self):
5. pass
- 6.
7. **class** Tesla(Car):
8. def mileage(self):

```
9.     print("The mileage is 30kmph")
10. class Suzuki(Car):
11.     def mileage(self):
12.         print("The mileage is 25kmph ")
13. class Duster(Car):
14.     def mileage(self):
15.         print("The mileage is 24kmph ")
16.
17. class Renault(Car):
18.     def mileage(self):
19.         print("The mileage is 27kmph ")
20.
21. # Driver code
22. t= Tesla ()
23. t.mileage()
24.
25. r = Renault()
26. r.mileage()
27.
28. s = Suzuki()
29. s.mileage()
30. d = Duster()
31. d.mileage()
```

Output:

```
The mileage is 30kmph
The mileage is 27kmph
The mileage is 25kmph
The mileage is 24kmph
```

Explanation -

In the above code, we have imported the **abc module** to create the abstract base class. We created the Car class that inherited the ABC class and defined an abstract empty method named mileage(). We have then inherited the base class from the three different subclasses and implemented the abstract method differently. We created the objects to call the abstract method.

Example -


```
1. # Python program to define
2. # abstract class
3.
4. from abc import ABC
5.
6. class Polygon(ABC):
7.
8.     # abstract method
9.     def sides(self):
10.         pass
11.
12. class Triangle(Polygon):
13.
14.
15.     def sides(self):
16.         print("Triangle has 3 sides")
17.
18. class Pentagon(Polygon):
19.
20.     def sides(self):
21.         print("Pentagon has 5 sides")
22.
23. class Hexagon(Polygon):
24.
25.     def sides(self):
26.         print("Hexagon has 6 sides")
27.
28. class square(Polygon):
29.
30.     def sides(self):
31.         print("I have 4 sides")
32.
33. # Driver code
34. t = Triangle()
35. t.sides()
36.
37. s = square()
```

```
38. s.sides()
39.
40. p = Pentagon()
41. p.sides()
42.
43. k = Hexagon()
44. K.sides()
```

Output:

```
Triangle has 3 sides
Square has 4 sides
Pentagon has 5 sides
Hexagon has 6 sides
```

Explanation -

In the above code, we have defined the abstract base class named Polygon and we also defined the abstract method. This base class inherited by the various subclasses. We implemented the abstract method in each subclass. We created the object of the subclasses and invoke the **sides()** method. The hidden implementations for the **sides()** method inside the each subclass comes into play. The abstract method **sides()** method, defined in the abstract class, is never invoked.

Points to Remember

Below are the points which we should remember about the abstract base class in Python.

- **An Abstract class can contain the both normal methods and abstract methods.**
- **An Abstract cannot be instantiated; we cannot create objects for the abstract class.**

Delete Object Properties

You can delete properties on objects by using the **del** keyword:

Example

Delete the age property from the p1 object:

```
del p1.age
```

Delete Objects

You can delete objects by using the `del` keyword:

Example

Delete the p1 object:

```
del p1
```

The pass Statement

`class` definitions cannot be empty, but if you for some reason have a `class` definition with no content, put in the `pass` statement to avoid getting an error.

Example

```
class Person:  
    pass
```

Add the `__init__()` Function

child class inherits the properties and methods from its parent.

We want to add the `__init__()` function to the child class

Note: The `__init__()` function is called automatically every time the class is being used to create a new object.

Example

Add the `__init__()` function to the `Student` class:

```
class Student(Person):
    def __init__(self, fname, lname):
        #add properties etc.
```

When you add the `__init__()` function, the child class will no longer inherit the parent's `__init__()` function.

Note: The child's `__init__()` function overrides the inheritance of the parent's `__init__()` function.

To keep the inheritance of the parent's `__init__()` function, add a call to the parent's `__init__()` function:

Example

```
class Student(Person):
    def __init__(self, fname, lname):
        Person.__init__(self, fname, lname)
```

Now we have successfully added the `__init__()` function, and kept the inheritance of the parent class, and we are ready to add functionality in the `__init__()` function.

Use the super() Function

Example

```
class Student(Person):
    def __init__(self, fname, lname):
        super().__init__(fname, lname)
```

By using the `super()` function, you do not have to use the name of the parent element

ABSTRACT methods

```
from abc import ABC, abstractmethod

class parent(ABC): #both ABC and @abs method are must to force
    # of abs method by child class
    implementation

    @abstractmethod
    def abMetho(self): #as both ABC and abs method present,
        # now child must implement this method else error
        pass
    def regMethod(self):
        print("reg method")

    #child must implement abMethod else child cannot create
    objects

class child(parent):
    def myMessage(self):
        print("hi")

#p=parent()
#ch=child()
#p.regMethod()
#ch.regMethod()
#p.abMetho()
#ch.abMetho()
```

Notes: Both ABC and @abstract method should be present to force the implementation of abs method by child class.

Class is abs when at least one method is abs and also inherits ABC

Only ABC or @abstract method does not make class abs. Both should be present to make a class abs.

Abs class cannot create objects (i.e. ABC and @abstract method together without providing implementation of abs method by child class will give error.

Directly, abs class and interfaces are not available in python, but we take help of abc module/ABC class.

Interface in Python

Generally, the interface is not part of Python, but we can implement it using the abc module. We will understand how interface works and cautions of Python interface creation.

Interfaces play a crucial role in software engineering as they provide a way to define a set of methods or behaviors that a child class must implement. As an application grows in size and complexity, it becomes more difficult to manage updates and changes to the code base.

Introduction to Interface

An interface is a design template for creating classes that share common methods. **The methods defined in an interface are abstract, meaning they are only outlined and lack implementation. It is unlike classes, which provide a concrete implementation for their methods. Instead, classes that implement an interface will fill in the abstract methods, giving them specific meaning and functionality.**

At least one method is abs, then class is abs class. (ABC+@abs method)

If all methods are abs then it is an interface.

If at least one method is non-abs then it's not interface but abs class.

Interfaces and abs classes cannot be instantiated.

There is no separate concept of abs class and interface. An abs Class with all methods having @abs method is an interface. There is no interface keyword like other prog languages.

Python follows a different approach to implementing the interface than other languages like C++, Java, and Go. **These languages have interface keywords, while Python doesn't have such keywords. Python further deviates from other languages in one other aspect. It doesn't require the class to implement the interface to define all its abstract methods.**

Informal Interfaces

An Informal interface is a class that defines methods that can be overridden, but there is no strict enforcement.

An interface is like a class but its methods just have prototype signature definition without any body to implement. The recommended functionality needs to be implemented by a concrete child class.

In languages like Java, there is interface keyword which makes it easy to define an interface. Python doesn't have it or any similar keyword. Hence the same ABC class and @abstractmethod decorator is used as done in an abstract class and interface both.

An abstract class and interface appear similar in Python. The only difference in two is that the abstract class may have some non-abstract methods, while all methods in interface must be abstract, and the implementing class must override all the abstract methods.

Example

```
from abc import ABC, abstractmethod
class demoInterface(ABC):
    @abstractmethod
    def method1(self):
        print ("Abstract method1")
        return

    @abstractmethod
    def method2(self):
        print ("Abstract method1")
```

```
return
```

The above interface has two abstract methods. **As in abstract class, we cannot instantiate an interface.**

```
obj = demoInterface()  
^^^^^^^^^^^^^^^^^^
```

TypeError: Can't instantiate abstract class demoInterface with abstract methods method1, method2

Let us provide a class that implements both the abstract methods. If doesn't contain implementations of all abstract methods, Python shows following error –

```
obj = concreteclass()  
^^^^^^^^^^^^^^^^^^
```

TypeError: Can't instantiate abstract class concreteclass with abstract method method2

The following class implements both methods –

```
class concreteclass(demoInterface):  
    def method1(self):  
        print ("This is method1")  
        return  
  
    def method2(self):  
        print ("This is method2")  
        return  
  
obj = concreteclass()  
obj.method1()  
obj.method2()
```

Output

When you execute this code, it will produce the following output –

```
This is method1  
This is method2
```


Class Variables

In the above Employee class example, name and age are instance variables, as their values may be different for each object. **A class attribute or variable whose value is shared among all the instances of a in this class. A class attribute represents common attribute of all objects of a class.**

Class attributes are not initialized inside `__init__()` constructor. They are defined in the class, but outside any method. They can be accessed by name of class in addition to object. In other words, a class attribute is available to class as well as its object.

Example

Let us add a class variable called `empCount` in Employee class. For each object declared, the `__init__()` method is automatically called. This method initializes the instance variables as well as increments the `empCount` by 1.

```
class Employee:
    empCount = 0
    def __init__(self, name, age):
        self.__name = name
        self.__age = age
        Employee.empCount += 1
        print ("Name: ", self.__name, "Age: ", self.__age)
        print ("Employee Number:", Employee.empCount)

e1 = Employee("Bhavana", 24)
e2 = Employee("Rajesh", 26)
e3 = Employee("John", 27)
```

Output

We have declared three objects. Every time, the `empCount` increments by 1.

Name: Bhavana Age: 24
Employee Number: 1

Name: Rajesh Age: 26

Employee Number: 2

Name: John Age: 27

Employee Number: 3

Python - Class Methods

ClassMethod() , @classmethod

An instance method accesses the instance variables of the calling object because it takes the reference to the calling object. **But it can also access the class variable as it is common to all the objects.**

Python has a built-in function classMethod() which transforms an instance method to a class method which can be called with the reference to the class only and not the object.

Instance method can access both class and instance variables

While class method can access only class variables

Syntax

```
classmethod(instance_method)
```

Example

In the Employee class, define a showcount() instance method with the "self" argument (reference to calling object). It prints the value of empCount. Next, transform the method to class method counter() that can be accessed through the class reference.

```
class Employee:
    empCount = 0
    def __init__(self, name, age):
        self.__name = name
```

```

    self.__age = age
    Employee.empCount += 1
def showcount(self):
    print (self.empCount)
counter=classmethod(showcount)
#CONVERTING INSTANCE METHOD INTO CLASS METHOD

e1 = Employee("Bhavana", 24)
e2 = Employee("Rajesh", 26)
e3 = Employee("John", 27)

e1.showcount()
Employee.counter()

```

Output

Call showcount() with object and call counter() with class, both show the value of employee count.

3
3

In above example:

Object.instanceMethod() is okay

Object.classMethod() is okay

Class.classMethod() is okay

Class.instanceMethod() will give you an error.

So, object can call both class and instance methods but class can call only class methods.

@classmethod decorator

Using @classmethod decorator is the prescribed way to define a class method as it is more convenient than first declaring an instance method and then transforming to a class method.

```
@classmethod
def showcount(cls):
    print (cls.empCount)

Employee.showcount()
```

Python - Static Methods

the static method doesn't have a mandatory argument like reference to the object – self or reference to the class – cls.

Python's standard library function staticmethod() returns a static method.

In the Employee class below, a method is converted into a static method. **This static method can now be called by its object or reference of class itself.**

```
class Employee:
    empCount = 0
    def __init__(self, name, age):
        self.__name = name
        self.__age = age
        Employee.empCount += 1

    #@staticmethod
    def showcount():
        print (Employee.empCount)
```

```
        return
    counter = staticmethod(showcount)

e1 = Employee("Bhavana", 24)
e2 = Employee("Rajesh", 26)
e3 = Employee("John", 27)

e1.counter()
e1.showcount() # WILL GIVE YOU AN ERROR, WILL WORK FINE FOR
CLASS METHOD
Employee.counter()
```

Python also has @staticmethod decorator that conveniently returns a static method.

```
@staticmethod
def showcount():
    print (Employee.empCount)
e1.showcount()
Employee.showcount()
```

Notes:

Instance methods can be called by objects only, not by class.

Class method can be called by both objects and classes.

Static methods can be called by both objects and classes.

Instance methods can access both class and instance variables.

Class methods can only access class variables.

Static method cannot access any object or class variables.

The class method in Python is a method, which is bound to the class but not the object of that class. The static methods are also same but there are some basic differences. For class methods, we need to specify @classmethod decorator, and for static method @staticmethod decorator is used.

Syntax for Class Method.

```
class my_class:
    @classmethod
    def function_name(cls, arguments):
        #Function Body
        return value
```

Syntax for Static Method.

```
class my_class:
    @staticmethod
    def function_name(arguments):
        #Function Body
        return value
```

What are the differences between Classmethod and StaticMethod?

| Class Method | Static Method |
|--|--|
| The class method takes cls (class) as first argument. | The static method does not take any specific parameter. |
| Class method can access and modify the class state. | Static Method cannot access or modify the class state. |
| The class method takes the class as parameter to know about the state of that class. | Static methods do not know about class state. These methods are used to do some utility tasks by taking some parameters. |

@classmethod decorator
is used here.

@staticmethod decorator is
used here.

Python - Constructors

A constructor is an instance method in a class, that is automatically called whenever a new object of the class is declared. The constructor's role is to assign value to instance variables as soon as the object is declared.

Python uses a special method called `__init__()` to initialize the instance variables for the object, as soon as it is declared.

The `__init__()` method acts as a constructor. It needs a mandatory argument `self`, which is the reference to the object.

```
def __init__(self):  
    #initialize instance variables
```

The `__init__()` method as well as any instance method in a class has a mandatory parameter, `self`. However, you can give any name to the first parameter, not necessarily `self`.

Let us define the constructor in Employee class to initialize name and age as instance variables. We can then access these attributes of its object.

Example

```
class Employee:  
  
    def __init__(self):  
        self.name = "Bhavana"  
        self.age = 24  
  
e1 = Employee()  
print ("Name: {}".format(e1.name))  
print ("age: {}".format(e1.age))
```

It will produce the following **output** –

Name: Bhavana

age: 24

Parameterized Constructor

Example

In this example, the `__init__()` constructor has two formal arguments. We declare Employee objects with different values –

```
class Employee:
    def __init__(self, name, age):
        self.name = name
        self.age = age

e1 = Employee("Bhavana", 24)
e2 = Employee("Bharat", 25)

print ("Name: {}".format(e1.name))
print ("age: {}".format(e1.age))
print ("Name: {}".format(e2.name))
print ("age: {}".format(e2.age))
```

It will produce the following **output** –

Name: Bhavana

age: 24

Name: Bharat

age: 25

You can assign defaults to the formal arguments in the constructor so that the object can be instantiated with or without passing parameters.

```
class Employee:
    'Common base class for all employees'
    def __init__(self, name="Bhavana", age=24):
        self.name = name
        self.age = age
```



```
e1 = Employee()
e2 = Employee("Bharat", 25)

print ("Name: {}".format(e1.name))
print ("age: {}".format(e1.age))
print ("Name: {}".format(e2.name))
print ("age: {}".format(e2.age))
```

It will produce the following **output** –

```
Name: Bhavana
age: 24
Name: Bharat
age: 25
```

Python - Instance Methods

In addition to the `__init__()` constructor, there may be one or more instance methods defined in a class. **A method with `self` as one of the formal arguments is called instance method, as it is called by a specific object.**

Example

In the following example a `displayEmployee()` method has been defined. It returns the name and age attributes of the `Employee` object that calls the method.

```
class Employee:
    def __init__(self, name="Bhavana", age=24):
        self.name = name
        self.age = age
    def displayEmployee(self):
        print ("Name : ", self.name, ", age: ", self.age)

e1 = Employee()
e2 = Employee("Bharat", 25)

e1.displayEmployee()
```

```
e2.displayEmployee()
```

It will produce the following **output** –

Name : Bhavana , age: 24

Name : Bharat , age: 25

You can add, remove, or modify attributes of classes and objects at any time –

```
emp1.salary = 7000 # Add a 'salary' attribute.  
emp1.name = 'xyz' # Modify 'name' attribute.  
del emp1.salary # Delete 'salary' attribute.
```

Instead of using the normal statements to access attributes, you can use the following functions –

- The **getattr(obj, name[, default])** – to access the attribute of object.
- The **hasattr(obj,name)** – to check if an attribute exists or not.
- The **setattr(obj,name,value)** – to set an attribute. If attribute does not exist, then it would be created.
- The **delattr(obj, name)** – to delete an attribute.

```
print (hasattr(e1, 'salary')) # Returns true if 'salary' attribute exists  
print (getattr(e1, 'name')) # Returns value of 'name' attribute  
setattr(e1, 'salary', 7000) # Set attribute 'salary' to 7000  
delattr(e1, 'age') # Delete attribute 'age'
```

It will produce the following **output** –

False

Bhavana

Python - Access Modifiers

The languages such as C++ and Java, use access modifiers (private, public, etc.) to restrict access to class members (i.e., variables and methods). These languages have keywords public, protected, and private to specify the type of access.

A class member is said to be public if it can be accessed from anywhere in the program. Private members are allowed to be accessed from within the class only.

- Usually, methods are defined as public and instance variable are private. This arrangement of private instance variables and public methods ensures implementation of principle of encapsulation.
- **Protected members are accessible from within the class as well as by classes derived from that class.**

Unlike these languages, Python has no provision to specify the type of access that a class member may have. By default, all the variables and methods in a class are public.

Example

Here, we have Employee class with instance variables name and age. An object of this class has these two attributes. They can be directly accessed from outside the class, because they are public.

```
class Employee:

    def __init__(self, name="Bhavana", age=24):
        self.name = name
        self.age = age

e1 = Employee()
e2 = Employee("Bharat", 25)

print ("Name: {}".format(e1.name))
print ("age: {}".format(e1.age))
```

```
print ("Name: {}".format(e2.name))  
print ("age: {}".format(e2.age))
```

It will produce the following output –

Name: Bhavana

age: 24

Name: Bharat

age: 25

Python doesn't enforce restrictions on accessing any instance variable or method. However, Python prescribes a convention of prefixing name of variable/method with single or double underscore to emulate behavior of protected and private access modifiers.

To indicate that an instance variable is private, prefix it with double underscore (such as "__age"). To imply that a certain instance variable is protected, prefix it with single underscore (such as "_salary")

Example

Let us modify the Employee class. Add another instance variable salary. Make **age** private and **salary** as protected by prefixing double and single underscores respectively.

```
class Employee:  
    def __init__(self, name, age, salary):  
        self.name = name # public variable  
        self.__age = age # private variable with double underscores  
        self._salary = salary # protected variable with single underscore  
    def displayEmployee(self):  
        print ("Name : ", self.name, ", age: ", self.__age, ", salary: ", self._salary)  
  
e1=Employee("Bhavana", 24, 10000)  
  
print (e1.name)  
print (e1._salary)  
print (e1.__age)
```

When you run this code, it will produce the following **output** –

Bhavana

10000

Traceback (most recent call last):

File "C:\Users\user\example.py", line 14, in <module>

print (e1.__age)

^^^^^^

AttributeError: 'Employee' object has no attribute '__age'

Python displays AttributeError because __age is private, and not available for use outside the class.

Mutliple Inheritance and Super() and __init__()

Class child (base1,base2) :

then base1 is first parent and it will overpower base2 by supplying its constructor / variables etc.

```
1 class Employee:
2     x='Employee'
3     def __init__(self):
4         print("in Employee")
5 class emp1:
6     x='emp1'
7     def __init__(self):
8         print("in emp1")
9 class emp2(Employee,emp1): #order of parenst important
10    #constructor of first parent is called
11    def disp():
12        print("hi")
13 e=emp2()
14 print(e.x)
```

in Employee
Employee
>

Using super() with multiple inheritance

Let's look at an example:

```
class Base1:
    def __init__(self):
        print("Base1 constructor")
class Base2:
    def __init__(self):
        print("Base2 constructor")
class Child(Base1, Base2):
    def __init__(self):
        super().__init__()
child = Child()
```

If we run this code, we will see the following output:

```
Base1 constructor
```

Python - Polymorphism

Example

As an example of polymorphism given below, we have **shape** which is an abstract class. It is used as parent by two classes circle and rectangle. Both classes override parent's draw() method in different ways.

```

from abc import ABC, abstractmethod
class shape(ABC):
    @abstractmethod
    def draw(self):
        "Abstract method"
        return

class circle(shape):
    def draw(self):
        super().draw() #calling abs method does not throw any error
        print ("Draw a circle")
        return

class rectangle(shape):
    def draw(self):
        super().draw() #calling abs method does not throw any error!!!
        print ("Draw a rectangle")
        return

shapes = [circle(), rectangle()]
for shp in shapes:
    shp.draw()

```

Output

When you execute this code, it will produce the following output
—

Draw a circle

Draw a rectangle

The variable **shp** first refers to circle object and calls draw() method from circle class. In next iteration, it refers to rectangle object and calls draw() method from rectangle class. Hence draw() method in shape class is polymorphic.

Destructor

`__del__()` and `del obj`

NOTE: if we do not provide our own destructor `__del__()`, the system will provide default destructor.

```
1 class amitclass:
2     def __init__(self):
3         self.x=10
4         print("in const")
5
6     def __del__(self): #destructor is called when del obj
7         print("in dest")
8
9 obj=amitclass()
10 print(obj.x)
11 del obj # __del__() destructor is called
12 print(obj.x) #error as obj is already deleted
13
```

in const
10
in dest
ERROR!
Traceback (most recent call last):
 File "<string>", line 12, in <module>
NameError: name 'obj' is not defined
> |

Method overloading tutpoint