|  |  |
| --- | --- |
|  |  |
| **PYTHON** |  |
|  |  |
|  | DATE – 07/11/2022  PYTHON |
|  | Zodex |

Table of Contents

[1. Basic 4](#_Toc177644321)

[2. Python Data Types 5](#_Toc177644322)

[2.1 Data Types 5](#_Toc177644323)

[2.1.1 Strings 5](#_Toc177644324)

[2.1.2 Numbers - Integer, Float (look use of decimal.Decimal), Complex 5](#_Toc177644325)

[2.1.3 Booleans 5](#_Toc177644326)

[2.1.4 Python List 5](#_Toc177644327)

[2.1.5 Python Tuples 6](#_Toc177644328)

[2.1.6 Python Sets (Frozen Sets) 6](#_Toc177644329)

[2.1.7 Python Dictionary 6](#_Toc177644330)

[2.1.8 Python Arrays 7](#_Toc177644331)

[2.2 Type Casting 7](#_Toc177644332)

[3. Python Operators 7](#_Toc177644333)

[4. Python Conditional Statement 7](#_Toc177644334)

[5. Python Loops 7](#_Toc177644335)

[6. Python Functions 8](#_Toc177644336)

[6.1 Python Function syntax 8](#_Toc177644337)

[6.2 Python Function Global and Local Scope Variables ( Uses global keyword to modify global 'a' within a function) 8](#_Toc177644338)

[6.3 Use of pass Statement in Function 8](#_Toc177644339)

[6.4 Return statemen in Python Function 8](#_Toc177644340)

[6.5 Python range() function 8](#_Toc177644341)

[6.6 \*args and \*\*kwargs in Python Function 8](#_Toc177644342)

[6.7 Python ‘Self’ as Default Argument 9](#_Toc177644343)

[6.8 Decorators in Python 9](#_Toc177644344)

[6.8.1 First Class Objects 9](#_Toc177644345)

[6.8.2 Decorators 10](#_Toc177644346)

[6.8.3 Chaining Decorators 12](#_Toc177644347)

[6.9 Python closures 12](#_Toc177644348)

[6.10 Lambda Function 12](#_Toc177644349)

[6.11 Map Function 13](#_Toc177644350)

[6.12 Filter Function 14](#_Toc177644351)

[6.13 Reduce Function 14](#_Toc177644352)

[6.14 reduce() vs accumulate() 15](#_Toc177644353)

[7. Python OOPs Concepts 15](#_Toc177644354)

[7.1 Python Classes and Objects 15](#_Toc177644355)

[7.1.1 \_\_init\_\_() method 16](#_Toc177644356)

[7.1.2 \_\_str\_\_() method 16](#_Toc177644357)

[7.1.3 Class and Instance Variables 17](#_Toc177644358)

[7.2 Polymorphism 18](#_Toc177644359)

[7.2.1 Types of Polymorphism Supported by Different Languages 18](#_Toc177644360)

[7.3 Inheritance 20](#_Toc177644361)

[7.3.1 Subclassing (Calling constructor of parent class) 21](#_Toc177644362)

[7.3.2 The super() Function 23](#_Toc177644363)

[7.3.3 Different types of Python Inheritance 23](#_Toc177644364)

[7.3.4 Private members of the parent class 23](#_Toc177644365)

[7.4 Abstract 24](#_Toc177644366)

[7.4.1 Concrete (Normal) Methods in Abstract Base Classes 25](#_Toc177644367)

[7.4.2 Abstract Properties in Python 25](#_Toc177644368)

[7.4.3 Abstract Class Instantiation 26](#_Toc177644369)

[7.5 Encapsulation 26](#_Toc177644370)

[7.5.1 How is Encapsulation Achieved in Python? 27](#_Toc177644371)

[7.5.2 Access modifiers in Python 27](#_Toc177644372)

[7.5.3 Using @property decorator in Python classes: 29](#_Toc177644373)

[8. Python Exception Handling 30](#_Toc177644374)

[8.1.1 Raising Exception 31](#_Toc177644375)

[9. Python File Handling 32](#_Toc177644376)

[9.1 with statement in Python 32](#_Toc177644377)

[9.2 Implementing all the functions in File Handling 32](#_Toc177644378)

# Basic

The key difference is that with primitive values, each variable holds its own copy of the value. Changes to one variable do not affect others. In contrast, with objects (including arrays), variables hold references to the same object, so changes to one reference are reflected in all references to that object.

In most modern programming languages like JS, Java, C++,etc. it holds true.

* 1. **Primitive values** (integers, strings) are immutable in Python, meaning changes to one variable do not affect another, as each variable holds its own copy of the value.
  2. **Objects** (lists, dictionaries, custom classes) are mutable, so variables referring to the same object will reflect changes made through any of those references.

**1. Primitive Values (Integers)**

```python

a = 10

b = a # b gets a copy of the value of a

a += 5 # Modify the value of a

print(a) # Output: 15

print(b) # Output: 10

```

- `a` and `b` hold copies of the integer value. Changing `a` does not affect `b`.

**2. Lists (Mutable Objects)**

```python

a = [1, 2, 3]

b = a # b is a reference to the same list as a

a.append(4) # Modify the list

print(a) # Output: [1, 2, 3, 4]

print(b) # Output: [1, 2, 3, 4]

```

- Both `a` and `b` refer to the same list object. Changes to `a` affect `b` because they are references to the same object.

**3. Custom Objects**

```python

class MyClass:

def \_\_init\_\_(self, value):

self.value = value

a = MyClass(10)

b = a # b is a reference to the same MyClass instance as a

a.value += 5 # Modify the attribute of the instance

print(a.value) # Output: 15

print(b.value) # Output: 15

```

# Python Data Types

## Data Types

### [Strings](https://www.geeksforgeeks.org/python-string/)

### [Numbers](https://www.geeksforgeeks.org/python-numbers/) - Integer, Float (look use of decimal.Decimal), Complex

### [Booleans](https://www.geeksforgeeks.org/boolean-data-type-in-python/)

###### *There are eight kinds of types supported by PyTables:*

**bool**: Boolean (true/false) types. Supported precisions: 8 (default) bits.

**int**: Signed integer types. Supported precisions: 8, 16, 32 (default) and 64 bits.

**uint**: Unsigned integer types. Supported precisions: 8, 16, 32 (default) and 64 bits.

**float**: Floating point types. Supported precisions: 16, 32, 64 (default) bits and extended precision floating point (see [note on floating point types](https://www.pytables.org/usersguide/datatypes.html#floating-point-note)).

**complex**: Complex number types. Supported precisions: 64 (32+32), 128 (64+64, default) bits and extended precision complex (see [note on floating point types](https://www.pytables.org/usersguide/datatypes.html#floating-point-note)).

**string**: Raw string types. Supported precisions: 8-bit positive multiples.

**time**: Data/time types. Supported precisions: 32 and 64 (default) bits.

**enum**: Enumerated types. Precision depends on base type.

### [Python List](https://www.geeksforgeeks.org/python-list/)

**Shallow Copy and Deep Copy**

A deep copy is a copy of a list, where we add an element in any of the lists, only that list is modified.

In list copy() method, changes made to the copied list are not reflected in the original list. The changes made to one list are not reflected on other lists except for in nested elements (like a list within a list).

We can use the copy.deepcopy() from the copy module to avoid this problem.

Techniques to deep copy:

Using copy.deepcopy()

Techniques to shallow copy:

Using copy.copy()

Using list.copy()

Using slicing

import copy

# Initializing list

list1 = [1, [2, 3], 4]

print("list 1 before modification:\n", list1)

# all changes are reflected

list2 = list1

# shallow copy - changes to

# nested list is reflected,

# same as copy.copy(), slicing

list3 = list1.copy()

list4 = list1[:]

# deep copy - no change is reflected

list5 = copy.deepcopy(list1)

list1.append(5)

list1[1][1] = 999

print("list 1 after modification:\n", list1)

print("list 2 after modification:\n", list2)

print("list 3 after modification:\n", list3)

print("list 4 after modification:\n", list4)

print("list 4 after modification:\n", list5)

**Output**

list 1 before modification:

[1, [2, 3], 4]

list 1 after modification:

[1, [2, 999], 4, 5]

list 2 after modification:

[1, [2, 999], 4, 5]

list 3 after modification:

[1, [2, 999], 4]

list 4 after modification:

[1, [2, 999], 4]

list 4 after modification:

[1, [2, 3], 4]

### [Python Tuples](https://www.geeksforgeeks.org/python-tuples/)

### [Python Sets](https://www.geeksforgeeks.org/python-sets/) (Frozen Sets)

### [Python Dictionary](https://www.geeksforgeeks.org/python-dictionary/)

|  |  |  |  |
| --- | --- | --- | --- |
| **List** | **Tuple** | **Set** | **Dictionary** |
| A list is a non-homogeneous data structure that stores the elements in columns of a single row or multiple rows. | A Tuple is a non-homogeneous data structure that stores elements in columns of a single row or multiple rows. | The set data structure is non-homogeneous but stores the elements in a single row. | A dictionary is also a non-homogeneous data structure that stores key-value pairs. |
| The list can be represented by [ ] | A tuple can be represented by  ( ) | The set can be represented by { } | The dictionary can be represented by { } |
| The list allows duplicate elements | Tuple allows duplicate elements | The Set will not allow duplicate elements | The dictionary doesn’t allow duplicate keys. |
| The list can be nested among all | A tuple can be nested among all | The set can be nested among all | The dictionary can be nested among all |
| Example: [1, 2, 3, 4, 5] | Example: (1, 2, 3, 4, 5) | Example: {1, 2, 3, 4, 5} | Example: {1: “a”, 2: “b”, 3: “c”, 4: “d”, 5: “e”} |
| A list can be created using the list() function | Tuple can be created using the tuple() function. | A set can be created using the set() function | A dictionary can be created using the dict() function. |
| A list is mutable i.e we can make any changes in the list. | A tuple is immutable i.e we cannot make any changes in the tuple. | A set is mutable i.e we can make any changes in the set, its elements are not duplicated. | A dictionary is mutable, its Keys are not duplicated. |
| List is ordered | Tuple is ordered | Set is unordered | Dictionary is ordered (Python 3.7 and above) |
| Creating an empty list  l=[] | Creating an empty Tuple  t=() | Creating a set  a=set()  b=set(a) | Creating an empty dictionary  d={} |

### [Python Arrays](https://www.geeksforgeeks.org/python-arrays/)

The main difference between lists and arrays in Python is that lists can hold items of different data types, while arrays are designed to hold items of the same data type. Arrays are more memory efficient and provide better performance for numerical operations. However, lists are more flexible and easier to use for general-purpose programming.

## Type Casting

* 1. Python Implicit Type Conversion (int + float = float)
  2. Python Explicit Type Conversion (Using keywords like float(), int())

# Python Operators

* 1. [Arithmetic operators](https://www.geeksforgeeks.org/python-arithmetic-operators/) (+, -, \*, /, %, //, \*\*)
  2. [Comparison Operators](https://www.geeksforgeeks.org/relational-operators-in-python/) (==, !=, >, <, >=, <=)
  3. [Logical Operators](https://www.geeksforgeeks.org/python-logical-operators-with-examples-improvement-needed/) (and, or, not)
  4. [Bitwise Operators](https://www.geeksforgeeks.org/python-bitwise-operators/) (&(AND), |(OR), ^(XOR), ~(NOT), >>(RS), <<(LS))
  5. [Assignment Operators](https://www.geeksforgeeks.org/assignment-operators-in-python/) (=, +=, -=, \*=, /=, %=, //=, \*\*=, &=, |=, ^=, >>=, <<=, :=)
  6. [Membership & Identity Operators | Python “in”, and “is” operator](https://www.geeksforgeeks.org/python-membership-identity-operators-not-not/)

# Python Conditional Statement

* 1. [Python If else](https://www.geeksforgeeks.org/python-if-else/)
  2. [Nested if statement](https://www.geeksforgeeks.org/nested-if-statement-in-python/)
  3. [Python if-elif-else Ladder](https://www.geeksforgeeks.org/python3-if-if-else-nested-if-if-elif-statements/)
  4. [Python If Else on One Line](https://www.geeksforgeeks.org/one-liner-for-python-if-elif-else-statements/)
  5. [Ternary Condition in Python](https://www.geeksforgeeks.org/ternary-operator-in-python/)
  6. [Match Case Statement](https://www.geeksforgeeks.org/python-match-case-statement/)

# Python Loops

* 1. [For Loop](https://www.geeksforgeeks.org/python-for-loops/)
  2. [While Loop](https://www.geeksforgeeks.org/python-while-loop/)
  3. [Loop control statements (break, continue, pass)](https://www.geeksforgeeks.org/break-continue-and-pass-in-python/)
  4. [Python List Comprehension](https://www.geeksforgeeks.org/python-list-comprehension/) (list comprehensions are quite faster than for loop)
  5. [Python Dictionary Comprehension](https://www.geeksforgeeks.org/python-dictionary-comprehension/)

# Python Functions

## Python Function syntax

*def function\_name(parameter: data\_type) -> return\_type:*

*"""Docstring"""*

*# body of the function*

*return expression*

Example:

**def** add(num1: int, num2: int) -> int:  
 *"""Add two numbers"""*  
 num3 = num1 + num2

**return** num3

## [Python Function Global and Local Scope Variables](https://www.geeksforgeeks.org/global-local-variables-python/) ( Uses global keyword to modify global 'a' within a function)

## [Use of pass Statement in Function](https://www.geeksforgeeks.org/how-to-write-an-empty-function-in-python-pass-statement/)

## [Return statemen in Python Function](https://www.geeksforgeeks.org/python-return-statement/)

## [Python range() function](https://www.geeksforgeeks.org/python-range-function/)

## [\*args and \*\*kwargs in Python Function](https://www.geeksforgeeks.org/args-kwargs-python/)

**Note:** “We use the “wildcard” or “\*” notation like this – \*args OR \*\*kwargs – as our function’s argument when we have doubts about the number of arguments we should pass in a function.”

**\*args (Non-Keyword Arguments):**

The special syntax \*args**in** function definitions in Python is used to pass a variable number of arguments to a function. It is used to pass a non-keyworded, variable-length argument list.

**def** myFun(arg1, \*argv):  
 print("First argument :", arg1)  
 **for** arg **in** argv:  
 print("Next argument through \*argv :", arg)

myFun('Hello', 'Welcome', 'to', 'GeeksforGeeks')

Output:

First argument : Hello

Next argument through \*argv : Welcome

Next argument through \*argv : to

Next argument through \*argv : GeeksforGeeks

**\*\*kwargs (Keyword Arguments):**

The special syntax \*\*kwargs in function definitions in Python is used to pass a keyworded, variable-length argument list. We use the name kwargs with the double star. The reason is that the double star allows us to pass through keyword arguments (and any number of them).

**def** myFun(arg1, \*\*kwargs):  
 **for** key, value **in** kwargs.items():  
 print("**%s** == **%s**" % (key, value))

*# Driver code*  
myFun("Hi", first='Geeks', mid='for', last='Geeks')

Output:

first == Geeks

mid == for

last == Geeks

## [Python ‘Self’ as Default Argument](https://www.geeksforgeeks.org/why-python-uses-self-as-default-argument/)

In Python, the ‘self‘ is used to reference the instance of a class within its methods. Unlike some other programming languages, Python does not implicitly pass the instance to the method; instead, it requires the explicit use of ‘self.’

No, ‘ self ‘ ( You can use any other name instead of it) is not a keyword in Python. Self is just a parameter name used in instance methods to refer to the instance itself.

## [Decorators in Python](https://www.geeksforgeeks.org/decorators-in-python/)

### First Class Objects

In Python, functions are first class objects which means that functions in Python can be used or passed as arguments.

Properties of first class functions:

* 1. A function is an instance of the Object type.
  2. You can store the function in a variable.
  3. You can pass the function as a parameter to another function.
  4. You can return the function from a function.
  5. You can store them in data structures such as hash tables, lists, …

***# Python program to illustrate functions can be treated as objects***   
**def** shout(text):   
 **return** text.upper()

print(shout('Hello'))

yell = shout

print(yell('Hello'))

Output:

HELLO

HELLO

***# Python program to illustrate functions can be passed as arguments to other functions***   
**def** shout(text):   
 **return** text.upper()

def whisper(text):   
 return text.lower()

def greet(func):   
 *# storing the function in a variable*   
 greeting = func("""Hi, I am created by a function passed as an argument.""")   
 print (greeting)

greet(shout)   
greet(whisper)

Output:

HI, I AM CREATED BY A FUNCTION PASSED AS AN ARGUMENT.

hi, i am created by a function passed as an argument.

***# Python program to illustrate functions Functions can return another function***

**def** create\_adder(x):   
 **def** adder(y):   
 **return** x+y

**return** adder

add\_15 = create\_adder(15)

print(add\_15(10))

Output:

25

### Decorators

Decorators are used to modify the behaviour of function or class. In Decorators, functions are taken as the argument into another function and then called inside the wrapper function.

**Syntax for Decorator:**

@gfg\_decorator

def hello\_decorator():

print("Gfg")

'''Above code is equivalent to -

def hello\_decorator():

print("Gfg")

hello\_decorator = gfg\_decorator(hello\_decorator)'''

*# defining a decorator*  
**def** hello\_decorator(func):

*# inner1 is a Wrapper function in which the argument is called*  
   
 *# inner function can access the outer local*  
 *# functions like in this case "func"*  
 **def** inner1():  
 print("Hello, this is before function execution")

*# calling the actual function no inside the wrapper function.*  
 func()

print("This is after function execution")  
   
 **return** inner1

*# defining a function, to be called inside wrapper*  
**def** function\_to\_be\_used():  
 print("This is inside the function !!")

*# passing 'function\_to\_be\_used' inside the decorator to control its behaviour*  
function\_to\_be\_used = hello\_decorator(function\_to\_be\_used)

*# calling the function*  
function\_to\_be\_used()

Output:

Hello, this is before function execution

This is inside the function !!

This is after function execution

*# importing libraries*  
**import time**  
**import math**

*# decorator to calculate duration taken by any function.*  
**def** calculate\_time(func):  
   
 *# added arguments inside the inner1, if function takes any arguments, can be added like this.*  
 **def** inner1(\*args, \*\*kwargs):

*# storing time before function execution*  
 begin = time.time()  
 func(\*args, \*\*kwargs)

*# storing time after function execution*  
 end = time.time()  
 print("Total time taken in : ", func.\_\_name\_\_, end - begin)

**return** inner1

*# this can be added to any function present in this case to calculate a factorial*  
@calculate\_time  
**def** factorial(num):

*# sleep 2 seconds because it takes very less time so that you can see the actual difference*  
 time.sleep(2)  
 print(math.factorial(num))

*# calling the function.*  
factorial(10)

Output:

3628800

Total time taken in : factorial 2.0061802864074707

*# if a function returns something or an argument is passed to the function*

**def** hello\_decorator(func):  
 **def** inner1(\*args, \*\*kwargs):  
 print("before Execution")  
 *# getting the returned value*  
 returned\_value = func(\*args, \*\*kwargs)  
 print("after Execution")  
 *# returning the value to the original frame*  
 **return** returned\_value  
 **return** inner1

*# adding decorator to the function*  
@hello\_decorator  
**def** sum\_two\_numbers(a, b):  
 print("Inside the function")  
 **return** a + b

a, b = 1, 2

*# getting the value through return of the function*  
print("Sum =", sum\_two\_numbers(a, b))

Output:

before Execution

Inside the function

after Execution

Sum = 3

### Chaining Decorators

In simpler terms [chaining decorators](https://www.geeksforgeeks.org/chain-multiple-decorators-in-python/) means decorating a function with multiple decorators.

*# code for testing decorator chaining*   
**def** decor1(func):   
 **def** inner():   
 x = func()   
 **return** x \* x   
 **return** inner

**def** decor(func):   
 **def** inner():   
 x = func()   
 **return** 2 \* x   
 **return** inner

@decor1  
@decor  
**def** num():   
 **return** 10

@decor  
@decor1  
**def** num2():  
 **return** 10

print(num())   
print(num2())

**Output:**

400

200

**The above example is similar to calling the function as –**

decor1(decor(num))

decor(decor1(num2))

## [Python closures](https://www.geeksforgeeks.org/python-closures/)

## [Lambda Function](https://www.geeksforgeeks.org/python-lambda-anonymous-functions-filter-map-reduce/)

***Syntax:****lambda arguments : expression*

* 1. *This function can have any number of arguments but only one expression, which is evaluated and returned.*
  2. *One is free to use lambda functions wherever function objects are required.*
  3. *You need to keep in your knowledge that lambda functions are syntactically restricted to a single expression.*
  4. *It has various uses in particular fields of programming, besides other types of expressions in functions.*

str1 = 'GeeksforGeeks'

upper = **lambda** string: string.upper()  
print(upper(str1))

Output:

GEEKSFORGEEKS

**Condition Checking Using Python lambda function**

format\_numeric = **lambda** num: f"**{**num**:**e**}**" **if** isinstance(num, int) **else** f"**{**num**:**,.2f**}**"

print("Int formatting:", format\_numeric(1000000))  
print("float formatting:", format\_numeric(999999.789541235))

Output:

Int formatting: 1.000000e+06

float formatting: 999,999.79

|  |  |
| --- | --- |
| With lambda function  lambda\_cube = lambda y: y\*y\*y | Without lambda function  def cube(y):  return y\*y\*y |
| Supports single-line sometimes statements that return some value. | Supports any number of lines inside a function block |
| Good for performing short operations/data manipulations. | Good for any cases that require multiple lines of code. |
| Using the lambda function can sometime reduce the readability of code. | We can use comments and function descriptions for easy readability. |

## [Map Function](https://www.geeksforgeeks.org/python-map-function/)

**Time complexity**: O(n), where n is the number of elements in the input list l.

**Auxiliary space**: O(n)

***Syntax****: map(fun, iter)*

***Parameters:***

* 1. ***fun:****It is a function to which map passes each element of given iterable.*
  2. ***iter:****It is iterable which is to be mapped.*

***NOTE:****You can pass one or more iterable to the map() function.*

***Returns:****Returns a list of the results after applying the given function to each item of a given iterable (list, tuple etc.)*

***NOTE :****The returned value from map() (map object) then can be passed to functions like list() (to create a list), set() (to create a set) .*

*# Python program to demonstrate working of map.*

*# Return double of n*  
**def** addition(n):  
 **return** n + n

# We double all numbers using map()  
numbers = (1, 2, 3, 4)  
result = map(addition, numbers)  
print(list(result))

Output

[2, 4, 6, 8]

*# List of strings*  
l = ['sat', 'bat', 'cat', 'mat']

*# map() can listify the list of strings individually*  
test = list(map(list, l))  
print(test)

Output

[['s', 'a', 't'], ['b', 'a', 't'], ['c', 'a', 't'], ['m', 'a', 't']]

## [Filter Function](https://www.geeksforgeeks.org/filter-in-python/)

**Time complexity**: O(n), where n is the number of elements in the input list l.

***Syntax:****filter(function, sequence)*

***Parameters:***

* 1. ***function:****function that tests if each element of a sequence is true or not.*
  2. ***sequence:****sequence which needs to be filtered, it can be sets, lists, tuples, or containers of any iterators.*

***Returns:****an iterator that is already filtered.*

*# a list contains both even and odd numbers.*   
seq = [0, 1, 2, 3, 5, 8, 13]

# result contains odd numbers of the list  
result = filter(lambda x: x % 2 != 0, seq)  
print(list(result))

# result contains even numbers of the list  
result = filter(lambda x: x % 2 == 0, seq)  
print(list(result))

Output:

[1, 3, 5, 13]

[0, 2, 8]

## [Reduce Function](https://www.geeksforgeeks.org/reduce-in-python/)

**Working:**

* 1. At first step, first two elements of sequence are picked and the result is obtained.
  2. Next step is to apply the same function to the previously attained result and the number just succeeding the second element and the result is again stored.
  3. This process continues till no more elements are left in the container.
  4. The final returned result is returned and printed on console.

*# python code to demonstrate working of reduce() importing functools for reduce()*  
**import functools**

*# initializing list*  
lis = [1, 3, 5, 6, 2]

*# using reduce to compute sum of list*  
print("The sum of the list elements is : ", end="")  
print(functools.reduce(**lambda** a, b: a+b, lis))

*# using reduce to compute maximum element from list*  
print("The maximum element of the list is : ", end="")  
print(functools.reduce(**lambda** a, b: a **if** a > b **else** b, lis))

Output

The sum of the list elements is : 17  
The maximum element of the list is : 6

## reduce() vs accumulate()

Both reduce() and accumulate() can be used to calculate the summation of a sequence elements. But there are differences in the implementation aspects in both of these.

* 1. reduce() is defined in “functools” module, accumulate() in “itertools” module.
  2. reduce() stores the intermediate result and only returns the final summation value. Whereas, accumulate() returns an iterator containing the intermediate results. The last number of the iterator returned is summation value of the list.
  3. reduce(fun, seq) takes function as 1st and sequence as 2nd argument. In contrast accumulate(seq, fun) takes sequence as 1st argument and function as 2nd argument.

*# python code to demonstrate summation using reduce() and accumulate()*

*# importing itertools for accumulate()*  
**import itertools**

*# importing functools for reduce()*  
**import functools**

*# initializing list*  
lis = [1, 3, 4, 10, 4]

*# printing summation using accumulate()*  
print("The summation of list using accumulate is :", end="")  
print(list(itertools.accumulate(lis, **lambda** x, y: x+y)))

*# printing summation using reduce()*  
print("The summation of list using reduce is :", end="")  
print(functools.reduce(**lambda** x, y: x+y, lis))

Output

The summation of list using accumulate is :[1, 4, 8, 18, 22]  
The summation of list using reduce is :22

# Python OOPs Concepts

In this section of [Python OPPs](https://www.geeksforgeeks.org/python-oops-concepts/), we’ll explore the core principles of object-oriented programming (OOP) in Python. From encapsulation to inheritance, polymorphism, abstract classes, and iterators, we’ll cover the essential concepts that empower you to build modular, reusable, and scalable code.

## [Python Classes and Objects](https://www.geeksforgeeks.org/python-classes-and-objects/)

A class is a user-defined blueprint or prototype from which objects are created. Classes provide a means of bundling data and functionality together. Creating a new class creates a new type of object, allowing new instances of that type to be made. Each class instance can have attributes attached to it to maintain its state. Class instances can also have methods (defined by their class) for modifying their state.

An object consists of:

* 1. State: It is represented by the attributes of an object. It also reflects the properties of an object.
  2. Behavior: It is represented by the methods of an object. It also reflects the response of an object to other objects.
  3. Identity: It gives a unique name to an object and enables one object to interact with other objects.

#### *Syntax: Class Definition*

class ClassName:

# Statement

#### *Syntax: Object Definition*

obj = ClassName()

print(obj.atrr)

*# Python3 program to demonstrate instantiating a class*  
**class Dog**:

# A simple class attribute  
 attr1 = "mammal"  
 attr2 = "dog"

*# A sample method*  
 **def** fun(self):  
 print("I'm a", self.attr1)  
 print("I'm a", self.attr2)

*# Driver code Object instantiation*  
Rodger = Dog()

*# Accessing class attributes and method through objects*  
print(Rodger.attr1)  
Rodger.fun()

Output:

mammal

I'm a mammal

I'm a dog

### \_\_init\_\_() method

The [\_\_init\_\_](https://www.geeksforgeeks.org/__init__-in-python/) method is similar to constructors in [C++](https://www.geeksforgeeks.org/cpp-tutorial/) and [Java](https://www.geeksforgeeks.org/java-tutorial/). Constructors are used to initializing the object’s state. Like methods, a constructor also contains a collection of statements(i.e. instructions) that are executed at the time of Object creation. It runs as soon as an object of a class is instantiated.

### \_\_str\_\_() method

Python has a particular method called \_\_str\_\_(). that is used to define how a class object should be represented as a string. It is often used to give an object a human-readable textual representation, which is helpful for logging, debugging, or showing users object information.

**class GFG**:  
 **def** \_\_init\_\_(self, name, company):  
 self.name = name  
 self.company = company

**def** \_\_str\_\_(self):  
 **return** f"My name is **{**self.name**}** and I work in **{**self.company**}**."

my\_obj = GFG("John", "GeeksForGeeks")  
print(my\_obj)

Output:

My name is John and I work in GeeksForGeeks.

### Class and Instance Variables

Instance variables are for data, unique to each instance and class variables are for attributes and methods shared by all instances of the class.

*# Class for Dog*

**class Dog**:

# Class Variable  
 animal = 'dog'

*# The init method or constructor*  
 **def** \_\_init\_\_(self, breed, color):

# Instance Variable  
 self.breed = breed  
 self.color = color

*# Objects of Dog class*  
Rodger = Dog("Pug", "brown")

print('Rodger details:')  
print('Rodger is a', Rodger.animal)  
print('Breed: ', Rodger.breed)  
print('Color: ', Rodger.color)

*# Class variables can be accessed using class name also*  
print("**\n**Accessing class variable using class name")  
print(Dog.animal)

Output:

Rodger details:

Rodger is a dog

Breed: Pug

Color: brown

Accessing class variable using class name

dog

*# Python3 program to show that we can create instance variables inside methods*  
*# Class for Dog*

**class Dog**:

# Class Variable  
 animal = 'dog'

*# The init method or constructor*  
 **def** \_\_init\_\_(self, breed):

# Instance Variable  
 self.breed = breed

*# Adds an instance variable*  
 **def** setColor(self, color):  
 self.color = color

# Retrieves instance variable  
 **def** getColor(self):  
 **return** self.color

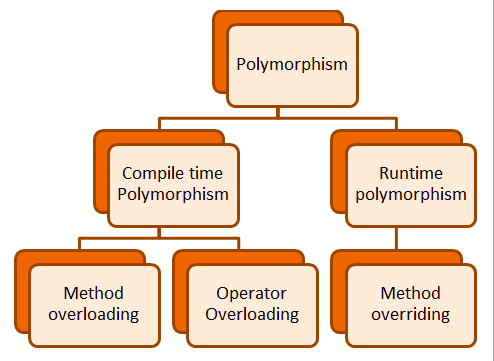
*# Driver Code*  
Rodger = Dog("pug")  
Rodger.setColor("brown")  
print(Rodger.getColor())

Output:

brown

## [Polymorphism](https://www.geeksforgeeks.org/polymorphism-in-python/)

The word polymorphism means having many forms. In programming, polymorphism means the same function name (but different signatures) being used for different types. The key difference is the data types and number of arguments used in function.



### Types of Polymorphism Supported by Different Languages

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Language** | **Compile-Time Polymorphism** | **Run-Time Polymorphism** | **Method Overloading** | **Method Overriding** | **Operator Overloading** |
| C++ | Yes | Yes | Yes | Yes | Yes |
| Java | Yes (Method Overloading) | Yes | Yes | Yes | No |
| JavaScript | No (Explicit Overloading) | Yes (via prototypes) | No | Yes | No |
| Python | No (Manual Emulation) | Yes | No | Yes | Yes |

#### *Note: Python doesn't support method overloading*

#### *Operator Overloading*

class Complex:

def \_\_init\_\_(self, real=0.0, imag=0.0):

self.real = real

self.imag = imag

# Operator overloading for + operator

def \_\_add\_\_(self, other):

return Complex(self.real + other.real, self.imag + other.imag)

# String representation of the object

def \_\_str\_\_(self):

return f"Real: {self.real}, Imaginary: {self.imag}"

# Creating complex number objects

c1 = Complex(3.0, 2.0)

c2 = Complex(1.0, 7.0)

# Adding two complex numbers using overloaded + operator

c3 = c1 + c2

print(c3) # Output: Real: 4.0, Imaginary: 9.0

**In this Python example, we define the \_\_add\_\_ method to overload the + operator, enabling us to add two Complex objects**

#### *Method Overriding*

class Animal:

def sound(self):

print("Animal is making a sound")

class Dog(Animal):

def sound(self):

print("Dog is barking")

# Run-time polymorphism

animal = Dog()

animal.sound() # Output: Dog is barking

**Example of inbuilt polymorphic functions:**

*# Python program to demonstrate in-built polymorphic functions*

*# len() being used for a string*  
print(len("geeks"))

*# len() being used for a list*  
print(len([10, 20, 30]))

Output:

5  
3

**Examples of user-defined polymorphic functions:**

*# A simple Python function to demonstrate Polymorphism*

**def** add(x, y, z = 0):   
 **return** x + y + z

# Driver code   
print(add(2, 3))  
print(add(2, 3, 4))

Output:

5  
9

**Polymorphism with class methods:**

**class India**():  
 **def** capital(self):  
 print("New Delhi is the capital of India.")

**def** language(self):  
 print("Hindi is the most widely spoken language of India.")

**def** type(self):  
 print("India is a developing country.")

**class USA**():  
 **def** capital(self):  
 print("Washington, D.C. is the capital of USA.")

**def** language(self):  
 print("English is the primary language of USA.")

**def** type(self):  
 print("USA is a developed country.")

obj\_ind = India()  
obj\_usa = USA()  
**for** country **in** (obj\_ind, obj\_usa):  
 country.capital()  
 country.language()  
 country.type()

Output:

New Delhi is the capital of India.  
Hindi is the most widely spoken language of India.  
India is a developing country.  
Washington, D.C. is the capital of USA.  
English is the primary language of USA.  
USA is a developed country.

**Polymorphism with Inheritance:**

**class Bird**:  
 **def** intro(self):  
 print("There are many types of birds.")  
   
 **def** flight(self):  
 print("Most of the birds can fly but some cannot.")  
   
**class sparrow**(Bird):  
 **def** flight(self):  
 print("Sparrows can fly.")  
   
**class ostrich**(Bird):  
 **def** flight(self):  
 print("Ostriches cannot fly.")  
   
obj\_bird = Bird()  
obj\_spr = sparrow()  
obj\_ost = ostrich()

obj\_bird.intro()  
obj\_bird.flight()

obj\_spr.intro()  
obj\_spr.flight()

obj\_ost.intro()  
obj\_ost.flight()

Output:

There are many types of birds.  
Most of the birds can fly but some cannot.  
There are many types of birds.  
Sparrows can fly.  
There are many types of birds.  
Ostriches cannot fly.

## [Inheritance](https://www.geeksforgeeks.org/inheritance-in-python/)

It is a mechanism that allows you to create a hierarchy of classes that share a set of properties and methods by deriving a class from another class. Inheritance is the capability of one class to derive or inherit the properties from another class.

**Benefits of inheritance are:**

Inheritance allows you to inherit the properties of a class, i.e., base class to another, i.e., derived class. The benefits of Inheritance in Python are as follows:

* 1. It represents real-world relationships well.
  2. It provides the reusability of a code. We don’t have to write the same code again and again. Also, it allows us to add more features to a class without modifying it.
  3. It is transitive in nature, which means that if class B inherits from another class A, then all the subclasses of B would automatically inherit from class A.
  4. Inheritance offers a simple, understandable model structure.
  5. Less development and maintenance expenses result from an inheritance.

**Python Inheritance Syntax**

The syntax of simple inheritance in Python is as follows:

Class BaseClass:

{Body}

Class DerivedClass(BaseClass):

{Body}

*# A Python program to demonstrate inheritance Base or Super class. Note object in bracket. (Generally, object is made ancestor of all classes) In Python 3.x "class Person" is equivalent to "class Person(object)"*

**class Person**(object):

*# Constructor*  
 **def** \_\_init\_\_(self, name):  
 self.name = name

*# To get name*  
 **def** getName(self):  
 **return** self.name

# To check if this person is an employee  
 **def** isEmployee(self):  
 **return False**

*# Inherited or Subclass (Note Person in bracket)*  
**class Employee**(Person):

*# Here we return true*  
 **def** isEmployee(self):  
 **return True**

*# Driver code*  
emp = Person("Geek1") *# An Object of Person*  
print(emp.getName(), emp.isEmployee())

emp = Employee("Geek2") *# An Object of Employee*  
print(emp.getName(), emp.isEmployee())

Output:

Geek1 False

Geek2 True

### Subclassing (Calling constructor of parent class)

The constructor i.e. the ‘\_\_init\_\_’ function of a class is invoked when we create an object variable or an instance of the class.

The variables defined within \_\_init\_\_() are called instance variables or objects. Hence, ‘name’ and ‘idnumber’ are the objects of the class Person. Similarly, ‘salary’ and ‘post’ are the objects of the class Employee. Since the class Employee inherits from class Person, ‘name’ and ‘idnumber’ are also the objects of class Employee.

*# Python code to demonstrate how parent constructors are called.*

*# parent class*  
**class Person**(object):

*# \_\_init\_\_ is known as the constructor*  
 **def** \_\_init\_\_(self, name, idnumber):  
 self.name = name  
 self.idnumber = idnumber

**def** display(self):  
 print(self.name)  
 print(self.idnumber)

*# child class*  
**class Employee**(Person):  
 **def** \_\_init\_\_(self, name, idnumber, salary, post):  
 self.salary = salary  
 self.post = post

*# invoking the \_\_init\_\_ of the parent class*  
 Person.\_\_init\_\_(self, name, idnumber)

*# creation of an object variable or an instance*  
a = Employee('Rahul', 886012, 200000, "Intern")

*# calling a function of the class Person using its instance*  
a.display()

Output:

Rahul

886012

**Python program to demonstrate error if we forget to invoke \_\_init\_\_() of the parent**

If you forget to invoke the \_\_init\_\_() of the parent class then its instance variables would not be available to the child class. The following code produces an error for the same reason.

**class A**:  
 **def** \_\_init\_\_(self, n='Rahul'):  
 self.name = n

**class B**(A):  
 **def** \_\_init\_\_(self, roll):  
 self.roll = roll

object = B(23)  
print(object.name)

Output :

Traceback (most recent call last):

File "/home/de4570cca20263ac2c4149f435dba22c.py", line 12, in

print (object.name)

AttributeError: 'B' object has no attribute 'name'

**Correct Code**

class A:

def \_\_init\_\_(self, n='Rahul'):

self.name = n

class B(A):

def \_\_init\_\_(self, roll):

self.roll = roll

# invoking the \_\_init\_\_ of the parent class

A.\_\_init\_\_(self)

object = B(23)

print(object.name)

### The super() Function

The super() function is a built-in function that returns the objects that represent the parent class. It allows to access the parent class’s methods and attributes in the child class.

*# parent class*  
**class Person**():  
 **def** \_\_init\_\_(self, name, age):  
 self.name = name  
 self.age = age

**def** display(self):  
 print(self.name, self.age)

*# child class*  
**class Student**(Person):  
 **def** \_\_init\_\_(self, name, age):  
 self.sName = name  
 self.sAge = age  
 *# inheriting the properties of parent class*  
 super().\_\_init\_\_("Rahul", age)

**def** displayInfo(self):  
 print(self.sName, self.sAge)

obj = Student("Mayank", 23)  
obj.display()  
obj.displayInfo()

Output:

Rahul 23

Mayank 23

### Different types of Python Inheritance

* 1. Single Inheritance
  2. Multiple Inheritance
  3. Multilevel Inheritance
  4. Hierarchical Inheritance
  5. Hybrid Inheritance:

From <[*https://www.geeksforgeeks.org/types-of-inheritance-python/*](https://www.geeksforgeeks.org/types-of-inheritance-python/)>

### Private members of the parent class

We don’t always want the instance variables of the parent class to be inherited by the child class i.e. we can make some of the instance variables of the parent class private, which won’t be available to the child class.

*# Python program to demonstrate private members of the parent class*

**class C**(object):  
 **def** \_\_init\_\_(self):  
 self.c = 21

*# d is private instance variable*  
 self.\_\_d = 42

**class D**(C):  
 **def** \_\_init\_\_(self):  
 self.e = 84  
 C.\_\_init\_\_(self)

object1 = D()

*# produces an error as d is private instance variable*  
print(object1.c)  
print(object1.\_\_d)

Output :

Here we can see that when we tried to print the variable ‘c’, its value 21 is printed on the console. Whereas when we tried to print ‘d’, it generated the error. This is because the variable ‘d’ is made private by using the underscores. It is not available to the child class ‘D’ and hence the error.

21

File "/home/993bb61c3e76cda5bb67bd9ea05956a1.py", line 16, in

print (object1.d)

AttributeError: type object 'D' has no attribute 'd'

## [Abstract](https://www.geeksforgeeks.org/abstract-classes-in-python/)

A class that contains one or more abstract methods is called an abstract class. An abstract method is a method that has a declaration but does not have an implementation.

By defining an abstract base class, you can define a common Application Program Interface(API) for a set of subclasses. This capability is especially useful in situations where a third party is going to provide implementations, such as with plugins, but can also help you when working in a large team or with a large code base where keeping all classes in your mind is difficult or not possible.

By default, Python does not provide abstract classes. Python comes with a module that provides the base for defining Abstract Base classes(ABC) and that module name is ABC.

ABC works by decorating methods of the base class as an abstract and then registering concrete classes as implementations of the abstract base. A method becomes abstract when decorated with the keyword @abstractmethod.

*# Python program showing abstract base class work*  
**from abc import** ABC, abstractmethod

**class Animal**(ABC):

def move(self):  
 pass

**class Human**(Animal):

def move(self):  
 print("I can walk and run")

**class Snake**(Animal):

def move(self):  
 print("I can crawl")

**class Dog**(Animal):

def bark(self):  
 print("I can bark")

*# Driver code*  
R = Human()  
R.move()

K = Snake()  
K.move()

R = Dog()  
R.bark()

Output

I can walk and run  
I can crawl  
I can bark

### Concrete (Normal) Methods in Abstract Base Classes

Concrete classes contain only concrete (normal) methods whereas abstract classes may contain both concrete methods and abstract methods.

The concrete class provides an implementation of abstract methods, the abstract base class can also provide an implementation by invoking the methods via super(). Let look over the example to invoke the method using super():

*# Python program invoking a method using super()*  
**from abc import** ABC

**class R**(ABC):  
 **def** rk(self):  
 print("Abstract Base Class")

**class K**(R):  
 **def** rk(self):  
 super().rk()  
 print("subclass ")

*# Driver code*  
r = K()  
r.rk()

Output

Abstract Base Class  
subclass

### Abstract Properties in Python

Abstract classes include attributes in addition to methods, you can require the attributes in concrete classes by defining them with @abstractproperty.

*# Python program showing*  
*# abstract properties*

**import abc**  
**from abc import** ABC, abstractmethod

**class parent**(ABC):  
 @abc.abstractproperty  
 **def** geeks(self):  
 **return** "parent class"

**class child**(parent):

@property  
 **def** geeks(self):  
 **return** "child class"

**try**:  
 r = parent()  
 print(r.geeks)  
**except Exception as** err:  
 print(err)

r = child()  
print(r.geeks)

Output

Can't instantiate abstract class parent with abstract methods geeks  
child class

In the above example, the Base class cannot be instantiated because it has only an abstract version of the property-getter method.

### Abstract Class Instantiation

we use an abstract class as a template and according to the need, we extend it and build on it before we can use it. Due to the fact, an abstract class is not a concrete class, it cannot be instantiated. When we create an object for the abstract class it raises an error.

*# Python program showing*  
*# abstract class cannot*  
*# be an instantiation*  
**from abc import** ABC,abstractmethod

**class Animal**(ABC):  
 @abstractmethod  
 **def** move(self):  
 **pass**  
**class Human**(Animal):  
 **def** move(self):  
 print("I can walk and run")

c=Animal()

Output:

Traceback (most recent call last):

File "/home/ffe4267d930f204512b7f501bb1bc489.py", line 19, in

c=Animal()

TypeError: Can't instantiate abstract class Animal with abstract methods move

## [Encapsulation](https://www.geeksforgeeks.org/encapsulation-in-python/)

It protects your classes from accidental changes or deletions and promotes code reusability and maintainability. Consider this simple class definition:

class Smartphone:  
 def \_\_init\_\_(self, brand, os):  
 self.brand = brand  
 self.os = os

iphone = Smartphone("Apple", "iOS 17")

Many Python programmers define classes like this. However, it is far from the best practices that pro Pythonistas follow. The problem with this class is evident when you try to modify its data:

iphone.os = "Android"  
print(iphone.os)

Output:

Android

Imagine an iPhone running on Android — what an outrage! Clearly, we need to set some boundaries within our class so that users can’t change its attributes to whatever they want.

### How is Encapsulation Achieved in Python?

We can only implement encapsulation as a mere convention and expect other Python developers to trust and respect our code.

In other OOP languages such as Java and C++, encapsulation is strictly enforced with access modifiers such as public, private or protected, but Python doesn't have those, Python uses a convention-based approach.

So, most, if not all, encapsulation techniques I am about to show you are Python conventions. They can easily be broken if you decide. But I trust that you respect and follow them in your own development projects.

### Access modifiers in Python

class Tree:  
 def \_\_init\_\_(self, height):  
 self.height = height

pine = Tree(20)  
print(pine.height)

Output:

20

It has a single height attribute that we can print. The problem is that we can also change it to whatever we want:

pine.height = "Grandma"  
pine.height

Output:

'Grandma'

##### Protected Member:

So, how do we tell users that changing height is off-limits? Well, we could turn it into a **protected member** by adding a single preceding underscore:

class Tree:  
 def \_\_init\_\_(self, height):  
 self.\_height = height

pine = Tree(20)  
pine.\_height

Output:

20

Now, people who are aware of this convention will know that they can only access the attribute and that we are strongly discouraging them from using and modifying it. But if they want, they can modify it, oh yes.

class Tree:  
 def \_\_init\_\_(self, height):  
 self.\_height = height

pine = Tree(20)

pine.\_height = 30

print(pine.\_height)

Output:

30

##### Private Member:

So, how do we prevent that too? By using another convention — turn the attribute into a **private member**by adding double preceding underscores:

class Tree:  
 def \_\_init\_\_(self, height):  
 self.\_\_height = height

pine = Tree(20)  
pine.\_\_height

Output:

AttributeError: 'Tree' object has no attribute '\_\_height'

Now, Python will raise an error if someone tries to access the attribute, let alone modify it.

But do you notice what we just did? We hid the only information related to our objects from users. Our class just became useless because it has no public attributes.

So, how do we expose tree height to users but still control how they are accessed and modified? For example, we want tree heights to be within a specific range and only have integer values. How do we enforce that?

At this point, your Java-using friend might chime in and suggest using [getter and setter methods](https://realpython.com/python-getter-setter). So, let’s try that first:

##### Getter and Setter:

class Tree:  
 def \_\_init\_\_(self, height):  
 self.\_\_height = height

def get\_height(self):  
 return self.\_\_height

def set\_height(self, new\_height):  
 if not isinstance(new\_height, int):  
 raise TypeError("Tree height must be an integer")  
 if 0 < new\_height <= 40:  
 self.\_\_height = new\_height  
 else:  
 raise ValueError("Invalid height for a pine tree")

pine = Tree(20)  
pine.get\_height()

Output:

20

pine.set\_height(25)

pine.get\_height()

Output:

25

Before setting a new value, set\_height ensures that the new height is within a certain range and numeric.

pine.set\_height("Password")

Output:

TypeError: Tree height must be an integer

But these methods seem like overkill for a simple operation. Besides, it is ugly to write code like this:

# Increase height by 5  
pine.set\_height(pine.get\_height() + 5)

Wouldn’t it be more beautiful and readable if we could write this code:

pine.height += 5

and still enforce the correct data type and range for height? The answer is yes and we will learn how to do just that in the next section.

### Using @property decorator in Python classes:

class Tree:  
 def \_\_init\_\_(self, height):  
 # First, create a private or protected attribute  
 self.\_\_height = height

@property  
 def height(self):  
 return self.\_\_height

pine = Tree(17)  
pine.height

Output:

17

We want users to access a hidden attribute named \_\_height as if it were a normal attribute called height. To achieve this, we define a method named height that returns self.\_\_height and decorate it with @property.

Now, we can call height and access the private attribute:

pine.height

Output:

17

But the best part is that users can’t modify it:

pine.height = 15

Output:

AttributeError: can't set attribute 'height'

So, we add another method called height(self, new\_height) that is wrapped by a height.setter decorator. Inside this method, we implement the logic that enforces the desired data type and range for height:

class Tree:  
 def \_\_init\_\_(self, height):  
 self.\_\_height = height

@property  
 def height(self):  
 return self.\_\_height

@height.setter  
 def height(self, new\_height):  
 if not isinstance(new\_height, int):  
 raise TypeError("Tree height must be an integer")  
 if 0 < new\_height <= 40:  
 self.\_\_height = new\_height  
 else:  
 raise ValueError("Invalid height for a pine tree")

Now, when a user tries to modify the height attribute, @height.setter is called, thus ensuring the correct value is passed:

pine = Tree(10)

pine.height = 33 # Calling @height.setter  
pine.height = 45 # An error is raised

Output:

ValueError: Invalid height for a pine tree

We can also customize how the height attribute is accessed through dot-notation with @height.getter:

class Tree:  
 def \_\_init\_\_(self, height):  
 self.\_\_height = height

@property  
 def height(self):  
 return self.\_\_height

@height.getter  
 def height(self):  
 # You can return a custom version of height  
 return f"This tree is {self.\_\_height} meters"

pine = Tree(33)

pine.height

Output:

'This tree is 33 meters'

Even though we created pine with an integer height, we could modify its value with @height.getter.

These were examples of how we could promote encapsulation in a Python class. Remember, encapsulation is still a convention because we can still break the internal \_\_height private member:

pine.\_Tree\_\_height = "Gotcha!"

pine.height

Output:

'This tree is Gotcha! meters'

Everything in Python classes is public, and so are [private methods](https://www.datacamp.com/tutorial/python-private-methods-explained). It isn’t a design flaw but an instance of the “We are all adults here” approach.

# Python Exception Handling

Error in Python can be of two types i.e. Syntax errors and Exceptions. Errors are problems in a program due to which the program will stop the execution. On the other hand, exceptions are raised when some internal events occur which change the normal flow of the program.

**Syntax:**

try:

# Some Code....

except:

# optional block

# Handling of exception (if required)

else:

# execute if no exception

finally:

# Some code .....(always executed)

**try**:  
 k = 5//0   
 print(k)

**except ZeroDivisionError**:  
 print("Can't divide by zero")

**finally**:  
 print('This is always executed')

Output:

Can't divide by zero

This is always executed

**def** AbyB(a , b):  
 **try**:  
 c = ((a+b) / (a-b))  
 **except ZeroDivisionError**:  
 print ("a/b result in 0")  
 **else**:  
 print (c)  
AbyB(2.0, 3.0)  
AbyB(3.0, 3.0)

Output:

-5.0

a/b result in 0

### Raising Exception

The [raise statement](https://www.geeksforgeeks.org/python-raising-an-exception-to-another-exception/) allows the programmer to force a specific exception to occur. The sole argument in raise indicates the exception to be raised. This must be either an exception instance or an exception class (a class that derives from Exception).

**try**:   
 **raise NameError**("Hi there")  
**except NameError**:  
 print ("An exception")  
 **raise**

The output of the above code will simply line printed as “An exception” but a Runtime error will also occur in the last due to the raise statement in the last line. So, the output on your command line will look like

Ouput:

An exception

Traceback (most recent call last):

File "/home/main.py", line 2, in <module>

raise NameError("Hi there")

NameError: Hi there

# Python File Handling

## with statement in Python

with statement is used in exception handling to make the code cleaner and much more readable. It simplifies the management of common resources like file streams. Observe the following code example on how the use of with statement makes code cleaner.

*# file handling*

*# 1) without using with statement*  
file = open('file\_path', 'w')  
file.write('hello world !')  
file.close()

*# 2) without using with statement*  
file = open('file\_path', 'w')  
**try**:  
 file.write('hello world')  
**finally**:  
 file.close()

*# using with statement*  
**with** open('file\_path', 'w') **as** file:  
 file.write('hello world !')

Notice that unlike the first two implementations, there is no need to call file.close() when using with statement. The with statement itself ensures proper acquisition and release of resources. An exception during the file.write() call in the first implementation can prevent the file from closing properly which may introduce several bugs in the code, i.e. many changes in files do not go into effect until the file is properly closed. The second approach in the above example takes care of all the exceptions but using the with statement makes the code compact and much more readable. Thus, with statement helps avoiding bugs and leaks by ensuring that a resource is properly released when the code using the resource is completely executed. The with statement is popularly used with file streams, as shown above and with Locks, sockets, subprocesses and telnets etc.

## Implementing all the functions in File Handling

**import os**

**def** create\_file(filename):  
 **try**:  
 **with** open(filename, 'w') **as** f:  
 f.write('Hello, world!**\n**')  
 print("File " + filename + " created successfully.")  
 **except IOError**:  
 print("Error: could not create file " + filename)

**def** read\_file(filename):  
 **try**:  
 **with** open(filename, 'r') **as** f:  
 contents = f.read()  
 print(contents)  
 **except IOError**:  
 print("Error: could not read file " + filename)

**def** append\_file(filename, text):  
 **try**:  
 **with** open(filename, 'a') **as** f:  
 f.write(text)  
 print("Text appended to file " + filename + " successfully.")  
 **except IOError**:  
 print("Error: could not append to file " + filename)

**def** rename\_file(filename, new\_filename):  
 **try**:  
 os.rename(filename, new\_filename)  
 print("File " + filename + " renamed to " + new\_filename + " successfully.")  
 **except IOError**:  
 print("Error: could not rename file " + filename)

**def** delete\_file(filename):  
 **try**:  
 os.remove(filename)  
 print("File " + filename + " deleted successfully.")  
 **except IOError**:  
 print("Error: could not delete file " + filename)

**if** \_\_name\_\_ == '\_\_main\_\_':  
 filename = "example.txt"  
 new\_filename = "new\_example.txt"

create\_file(filename)  
 read\_file(filename)  
 append\_file(filename, "This is some additional text.**\n**")  
 read\_file(filename)  
 rename\_file(filename, new\_filename)  
 read\_file(new\_filename)  
 delete\_file(new\_filename)