2. PYTHON FUNCTIONS, MODULES & PACKAGES.

2.1 Function Basics-Scope, nested function, non-local statements

Python Functions:

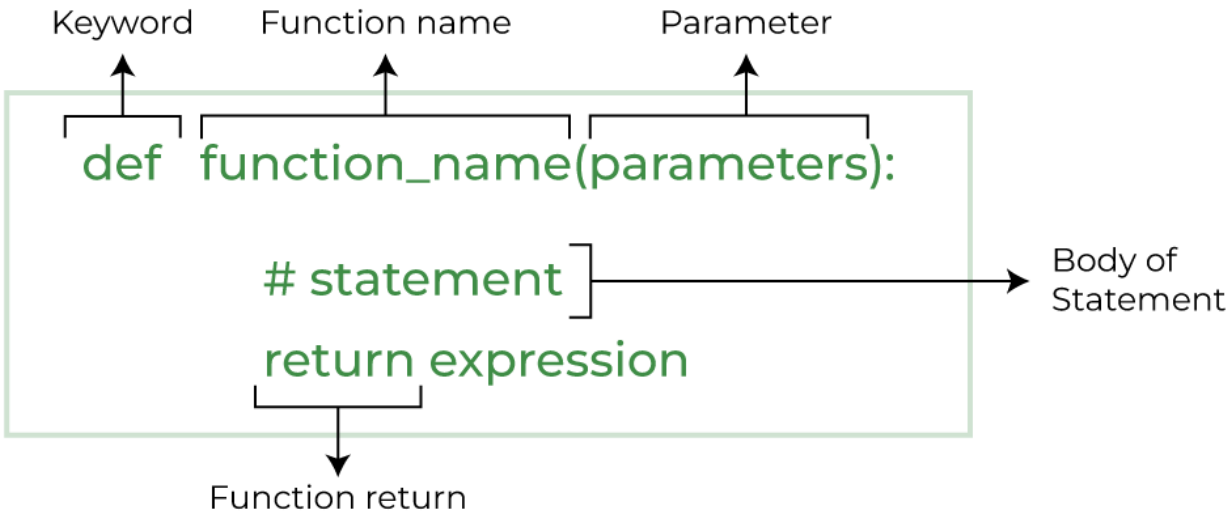
Python Functions is a block of statements that return the specific task.

The idea is to put some commonly or repeatedly done tasks together and make a function so that instead of writing the same code again and again for different inputs, we can do the function calls to reuse code contained in it over and over again. Some Benefits of Using Functions

- Increase Code Readability
- Increase Code Reusability

Python Function Declaration

The syntax to declare a function is:



Types of Functions in Python

1. **Built-in library function.**
2. **User-defined function**.

1. **Built-in library function.**

   These are Standard functions in Python that are available to use. Python provides a lot of built-in functions that eases the writing of code.

Function Name	Description
Python abs()	Return the absolute value of a number
Python all()	Return true if all the elements of a given iterable(List, Dictionary, Tuple, set, etc) are True else it returns False
Python any()	Returns true if any of the elements of a given iterable(List, Dictionary, Tuple, set, etc) are True else it returns False
Python ascii()	Returns a string containing a printable representation of an object
Python bin()	Convert integer to binary string
Python bool()	Return or convert a value to a Boolean value i.e., True or False
Python bytearray()	Returns a bytearray object which is an array of given bytes
Python bytes()	Converts an object to immutable byte represented object of given size and data
Python callable()	Returns True if the object passed appears to be callable
Python chr()	Returns a string representing a character whose Unicode code point is an integer
Python classmethod()	Returns a class method for a given function
Python compile()	Returns a Python code object
Python complex()	Creates Complex Number
Python delattr()	Delete the named attribute from the object
Python dict()	Creates a Python Dictionary
Python dir()	Returns list of the attributes and methods of any object
Python divmod()	Takes two numbers and returns a pair of numbers consisting of their quotient and remainder
Python enumerate()	Adds a counter to an iterable and returns it in a form of enumerating object
Python eval()	Parses the expression passed to it and runs python expression(code) within the program
Python exec()	Used for the dynamic execution of the program
Python filter()	Filters the given sequence with the help of a function that tests each element in the sequence to be true or not
Python float()	Return a floating-point number from a number or a string

Python format()	Formats a specified value
Python frozenset()	Returns immutable frozenset
Python getattr()	Access the attribute value of an object
Python globals()	Returns the dictionary of current global symbol table
Python hasattr()	Check if an object has the given named attribute and return true if present
Python hash()	Encode the data into unrecognizable value
Python help()	Display the documentation of modules, functions, classes, keywords, etc
Python hex()	Convert an integer number into its corresponding hexadecimal form
Python id()	Return the identity of an object
Python input()	Take input from user as a string
Python int()	Converts a number in given base to decimal
Python isinstance()	Checks if the objects belong to certain class or not
Python issubclass()	Check if a class is a subclass of another class or not
Python iter()	Convert an iterable to iterator
Python len()	Returns the length of the object
Python list()	Creates a list in Python
Python locals()	Returns the dictionary of the current local symbol table
Python map()	Returns a map object(which is an iterator) of the results after applying the given function to each item of a given iterable
Python max()	Returns the largest item in an iterable or the largest of two or more arguments
Python memoryview()	Returns memory view of an argument
Python min()	Returns the smallest item in an iterable or the smallest of two or more arguments
Python next()	Receives the next item from the iterator
Python object()	Returns a new object

Python oct()	returns octal representation of integer in a string format.
Python open()	Open a file and return its object
Python ord()	Returns the Unicode equivalence of the passed argument
Python pow()	Compute the power of a number
Python print()	Print output to the console
Python property()	Create property of a class
Python range()	Generate a sequence of numbers
Python repr()	Return intable version of the object
Python reversed()	Returns an iterator that accesses the given sequence in the reverse order
Python round()	Rounds off to the given number of digits and returns the floating-point number
Python set()	Convert any of the iterable to sequence of iterable elements with distinct elements
Python setattr()	Assign the object attribute its value
Python slice()	Returns a slice object
Python sorted()	Returns a list with the elements in sorted manner, without modifying the original sequence
Python staticmethod()	Converts a message into static message
Python str()	Returns the string version of the object
Python sum()	Sums up the numbers in the list
Python super()	Returns a temporary object of the superclass
Python tuple()	Creates a tuple in Python
Python type()	Returns the type of the object
Python vars()	Returns the __dict__ attribute for a module, class, instance, or any other object
Python zip()	Maps the similar index of multiple containers
Python __import__()	Imports the module during runtime

2. **User-defined function**.

We can create our own functions based on our requirements.

Creating a function in Python:

We can create a Python function using the **def** keyword.

```
# A simple Python function
def fun():
  print("Welcome to ATESTC")
```

Calling a Python Function:

After creating a function we can call it by using the name of the function followed by parenthesis containing parameters of that particular function.

```
# A simple Python function
def fun():
    print("Welcome to ATESTC")
# Driver code to call a function
fun()
```

Output: Welcome to ATESTC

Python Function with parameters

Parameters:

A parameter is the variable defined within the parentheses during function definition. Simply they are written when we declare a function.

```
# Here a, b are the parameters
def sum(a, b):
  print(a+b)
sum(1,2)
```

Outpt: 3

Arguments:

An argument is a value that is passed to a function when it is called. It might be a variable, value or object passed to a function or method as input. They are written when we are calling the function.

```
def sum(a,b):
 print(a+b)
# Here the values 1,2 are arguments
sum(1,2)
```
Outpt: 3

Python Function Arguments:

Arguments are the values passed inside the parenthesis of the function. A function can have any number of arguments separated by a comma.

```
# A simple Python function to check whether x is even or odd
def evenOdd(x):
   if (x % 2 == 0):
      print("even")
   else:
      print("odd")
# Driver code to call the function
evenOdd(2)
evenOdd(3)
```

Output: even

Odd

Types of Python Function Arguments

Python supports various types of arguments that can be passed at the time of the function call. In Python, we have the following 4 types of function arguments.

- Default argument
- Positional arguments
- Keyword arguments (named arguments)
- Arbitrary arguments (variable-length arguments *args and **kwargs)

Default Arguments:

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

```
# Python program to demonstrate default arguments
def myFun(x, y=50):
    print("x: ", x)
    print("y: ", y)
# Driver code (We call myFun() with only argument)
myFun(10)
```

Output: x: 10
y: 50

Positional Arguments:

Positional Arguments are needed to be included in proper order i.e the first argument is always listed first when the function is called, second argument needs to be called second and so on.

```
def person_name(first_name, second_name):
  print(first_name+second_name)
# First name is Ram placed first
# Second name is Babu place second
person_name("Ram","Babu")
```

Output: RamBabu

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

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

Output: Case-1:

Hi, I am Suraj

My age is 27

Case-2:

Hi, I am 27

My age is Suraj

Keyword Arguments:

Keyword Arguments is an argument passed to a function or method which is preceded by a keyword and an equal to sign. The order of keyword argument with respect to another keyword argument does not matter because the values are being explicitly assigned.

```
def person_name(first_name, second_name):
  print(first_name+second_name)
# Here we are explicitly assigning the values
person_name(second_name="Babu",first_name="Ram")
```

Output: RamBabu

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

Output: Python Programe

Python Programe

Arbitrary Keyword Arguments:

In Python Arbitrary Keyword Arguments, *args, and **kwargs can pass a variable number of arguments to a function using special symbols. There are two special symbols:

- *args in Python (Non-Keyword Arguments)
- **kwargs in Python (Keyword Arguments)

Example 1: Variable length non-keywords argument

```
# Python program to illustrate *args for variable number of arguments
def myFun(*argv):
    for arg in argv:
        print(arg)
myFun('Hello', 'Welcome', 'to', 'Python Class')
```

Output:
Hello
Welcome
to
Python Class

Example 2: Variable length keyword arguments

```
# Python program to illustrate *kwargs for variable number of keyword arguments
def myFun(**kwargs):
    for key, value in kwargs.items():
        print("%s == %s" % (key, value))
# Driver code
myFun(first='Python', mid='for', last='Python')
```

Output:
first == Python
mid == for
last == Python

Docstring:

The first string after the function is called the Document string or Docstring in short. This is used to describe the functionality of the function. The use of docstring in functions is optional but it is considered a good practice.

The below syntax can be used to print out the docstring of a function:

Syntax: **print(function_name.__doc__)**

Example: Adding Docstring to the function

```
# A simple Python function to check whether x is even or odd
def evenOdd(x):
    """Function to check if the number is even or odd"""
```

```
    if (x % 2 == 0):
        print("even")
    else:
        print("odd")
# Driver code to call the function
print(evenOdd.__doc__)
```

Output: Function to check if the number is even or odd

Python Function within Functions:

A function that is defined inside another function is known as the inner function or nested function. Nested functions are able to access variables of the enclosing scope. Inner functions are used so that they can be protected from everything happening outside the function.

```
# Python program to demonstrate accessing of variables of nested functions
def f1():
    s = 'I love Python'
    def f2():
        print(s)
    f2()
# Driver's code
f1()
```

Output: I love Python

Return statement in Python function:

The function return statement is used to exit from a function and go back to the function caller and return the specified value or data item to the caller. The syntax for the return statement is:

return [expression_list]

The return statement can consist of a variable, an expression, or a constant which is returned at the end of the function execution. If none of the above is present with the return statement a None object is returned.

Example: Python Function Return Statement

```
def square_value(num):
```

```
    """This function returns the square
    value of the entered number"""
    return num**2
print(square_value(2))
print(square_value(-4))
```

Output: 4

16

Pass by Reference or pass by value :

One important thing to note is, in Python every variable name is a reference. When we pass a variable to a function, a new reference to the object is created. Parameter passing in Python is the same as reference passing in Java.

```
    # Here x is a new reference to same list lst
    def myFun(x):
        x[0] = 20
    # Driver Code (Note that lst is modified after function call.
    lst = [10, 11, 12, 13, 14, 15]
    myFun(lst)
    print(lst)
```

Output: [20, 11, 12, 13, 14, 15]

When we pass a reference and change the received reference to something else, the connection between the passed and received parameter is broken. For example, consider the below program as follows:

```
    def myFun(x):
        # After below line link of x with previous object gets broken. A new object is ssigned
        # to x.
        x = [20, 30, 40]
    # Driver Code (Note that lst is not modified after function call.
    lst = [10, 11, 12, 13, 14, 15]
    myFun(lst)
    print(lst)
```

Output: [10, 11, 12, 13, 14, 15]

Another example demonstrates that the reference link is broken if we assign a new value (inside the function).

```
def myFun(x):
    # After below line link of x with previous object gets broken. A new object is
    assigned to x.
    x = 20
# Driver Code (Note that x is not modified after function call.
x = 10
myFun(x)
print(x)
```

Output: 10

2.2 Anonymous Function: lambda

Python Lambda Functions are anonymous function means that the function is without a name. As we already know that the def keyword is used to define a normal function in Python. Similarly, the lambda keyword is used to define an anonymous function in Python.

Python Lambda Function Syntax:

lambda arguments: expression

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

Python Lambda Function Example

```
str1 = 'PythonForPython'
# lambda returns a function object
rev_upper = lambda string: string.upper()[::-1]
print(rev_upper(str1))
```

Output: NOHTYPROFNOHTYP

Example 1: 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

Example 2: Difference Between Lambda functions and def defined function

```
def cube(y):
  return y*y*y
lambda_cube = lambda y: y*y*y
# using function defined using def keyword
print("Using function defined with `def` keyword, cube:", cube(5))
# using the lambda function
print("Using lambda function, cube:", lambda_cube(5))
```

Output: Using function defined with `def` keyword, cube: 125
Using lambda function, cube: 125

With lambda function	Without lambda function
Supports single line statements that returns 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 lambda function can sometime reduce the readability of code.	We can use comments and function descriptions for easy readability.

Example 1: Python Lambda Function with List Comprehension

```
is_even_list = [lambda arg=x: arg * 10 for x in range(1, 5)]
# iterate on each lambda function and invoke the function to get the calculated value
for item in is_even_list:
  print(item())
```

Output: 10

20

30

40

Example 2: Python Lambda Function with if-else

```
# Example of lambda function using if-else
Max = lambda a, b : a if(a > b) else b
print(Max(1, 2))
```

Output: 2

Example 3: Python Lambda with Multiple statements

Lambda functions does not allow multiple statements, however, we can create two lambda functions and then call the other lambda function as a parameter to the first function. Let's try to find the second maximum element using lambda.

```
List = [[2,3,4],[1, 4, 16, 64],[3, 6, 9, 12]]
# Sort each sublist
sortList = lambda x: (sorted(i) for i in x)
# Get the second largest element
secondLargest = lambda x, f : [y[len(y)-2] for y in f(x)]
res = secondLargest(List, sortList)
print(res)
```

Output: [3, 16, 9]

Explanation: In the above example, we have created a lambda function that sorts each sublist of the given list. Then this list is passed as the parameter to the second lambda function, which returns the n-2 element from the sorted list, where n is the length of the sublist.

Lambda functions can be used along with built-in functions like filter(), map() and reduce().

Using lambda() Function with filter():

The filter() method filters the given sequence with the help of a function that tests each element in the sequence to be true or not.

The filter() function in Python takes in a function and a list as arguments. This offers an elegant way to filter out all the elements of a sequence "sequence", for which the function returns True.

filter(function, sequence)

Parameters:

function: function that tests if each element of a sequence true or not.

sequence: sequence which needs to be filtered, it can be sets, lists, tuples, or containers of any iterators.

Returns:

returns an iterator that is already filtered.

Here is a small program that returns the odd numbers from an input list:

Example 1: Filter out all odd numbers using filter() and lambda function

Here, lambda x: (x % 2 != 0) returns True or False if x is not even. Since filter() only keeps elements where it produces **True**, thus it removes all odd numbers that generated **False**.

```
li = [5, 7, 22, 97, 54, 62, 77, 23, 73, 61]
final_list = list(filter(lambda x: (x % 2 != 0), li))
print(final_list)
```

Output: **[5, 7, 97, 77, 23, 73, 61]**

Example 2: Filter all people having age more than 18, using lambda and filter() function

```
# Python 3 code to people above 18 yrs
ages = [13, 90, 17, 59, 21, 60, 5]
adults = list(filter(lambda age: age > 18, ages))
print(adults)
```

Output: **[90, 59, 21, 60]**

Using lambda() Function with map():

map() function returns a map object(which is an iterator) of the results after applying the given function to each item of a given iterable (list, tuple etc.)

map(fun, iter)

Parameters :

fun : It is a function to which map passes each element of given iterable.

iter : It is a iterable which is to be mapped.

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)

The map() function in Python takes in a function and a list as an argument. The function is called with a lambda function and a list and a new list is returned which contains all the lambda modified items returned by that function for each item.

Example 1: Multiply all elements of a list by 2 using lambda and map() function

```
# Python code to illustrate map() with lambda() to get double of a list.
li = [5, 7, 22, 97, 54, 62, 77, 23, 73, 61]
final_list = list(map(lambda x: x*2, li))
print(final_list)
```

Output: **[10, 14, 44, 194, 108, 124, 154, 46, 146, 122]**

Example 2: Transform all elements of a list to upper case using lambda and map() function

```
# Python program to demonstrate use of lambda() function with map() function
animals = ['dog', 'cat', 'parrot', 'rabbit']

# here we intend to change all animal names to upper case and return the same
uppered_animals = list(map(lambda animal: animal.upper(), animals))
print(uppered_animals)
```

Output: **['DOG', 'CAT', 'PARROT', 'RABBIT']**

Using lambda() Function with reduce():

The reduce() function in Python takes in a function and a list as an argument. The function is called with a lambda function and an iterable and a new reduced result is returned. This performs a repetitive operation over the pairs of the iterable. The reduce() function belongs to the functools module.

Example 1: Sum of all elements in a list using lambda and reduce() function

```
# Python code to illustrate reduce() with lambda() to get sum of a list
from functools import reduce
li = [5, 8, 10, 20, 50, 100]
sum = reduce((lambda x, y: x + y), li)
print(sum)
```

Output: **193**

Here the results of the previous two elements are added to the next element and this goes on till the end of the list like (((((5+8)+10)+20)+50)+100).

Example 2: Find the maximum element in a list using lambda and reduce() function

```
# python code to demonstrate working of reduce() with a lambda function importing
#functools for reduce()
import functools
# initializing list
lis = [1, 3, 5, 6, 2, ]
# 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 maximum element of the list is : 6**

2.3 Decorators and Generators

Decorators in Python:

In Python, a decorator is a design pattern that allows you to modify the functionality of a function by wrapping it in another function.
The outer function is called the decorator, which takes the original function as an argument and returns a modified version of it.

Decorators are a very powerful and useful tool in Python since it allows programmers to modify the behaviour of a function or class. Decorators allow us to wrap another function in order to extend the behaviour of the wrapped function, without permanently modifying it. But before diving deep into decorators let us understand some concepts that will come in handy in learning the decorators.

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:

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

Example 1: Treating the functions as objects.

```
# 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

In the above example, we have assigned the function shout to a variable. This will not call the function instead it takes the function object referenced by a shout and creates a second name pointing to it, yell.

Example 2: Passing the function as an argument

```
# 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.

In the above example, the greet function takes another function as a parameter (shout and whisper in this case). The function passed as an argument is then called inside the function greet.

Example 3: Returning functions from another function.

```
# 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**

In the above example, we have created a function inside of another function and then have returned the function created inside.

Decorators:

As stated above the 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:

```
    @atestc_decorator
    def hello_decorator():
        print("ATESTC")
    '''Above code is equivalent to -
    def hello_decorator():
        print("ATESTC")
```

hello_decorator = atestc _decorator(hello_decorator)'''

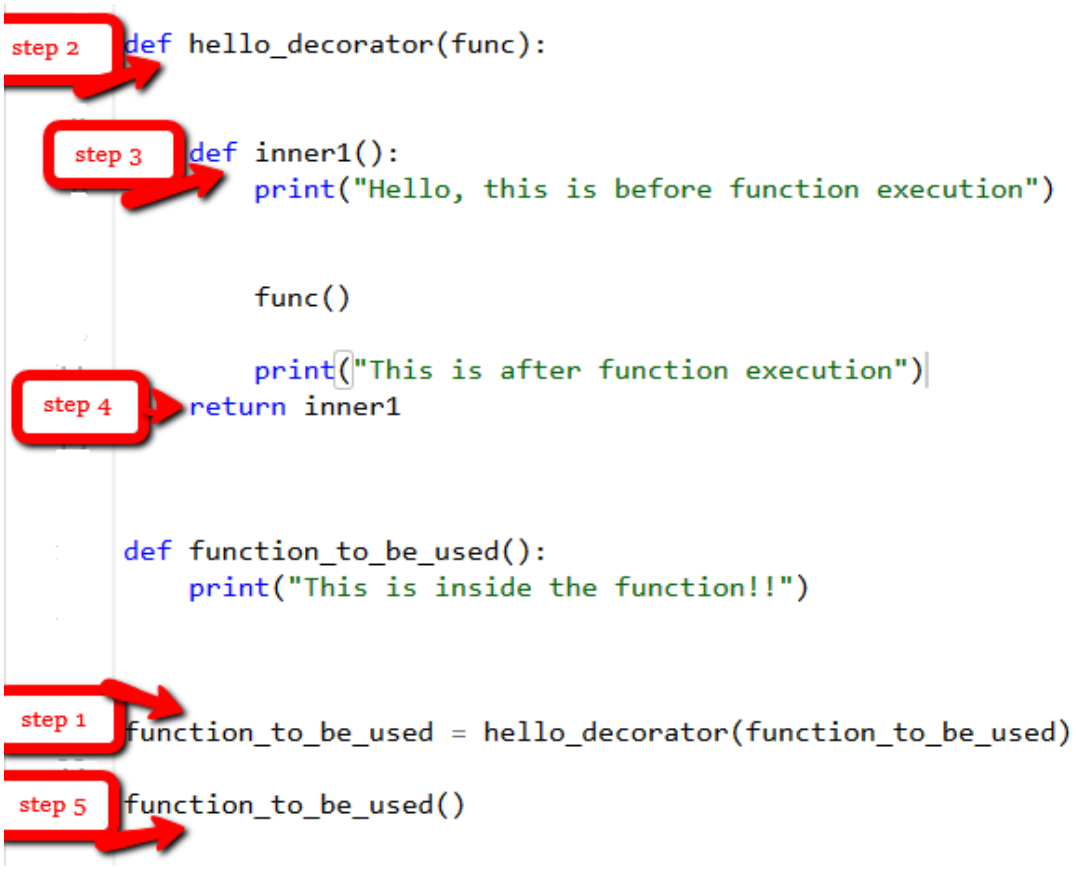
In the above code, atestc _decorator is a callable function, that will add some code on the top of some another callable function, hello_decorator function and return the wrapper function.

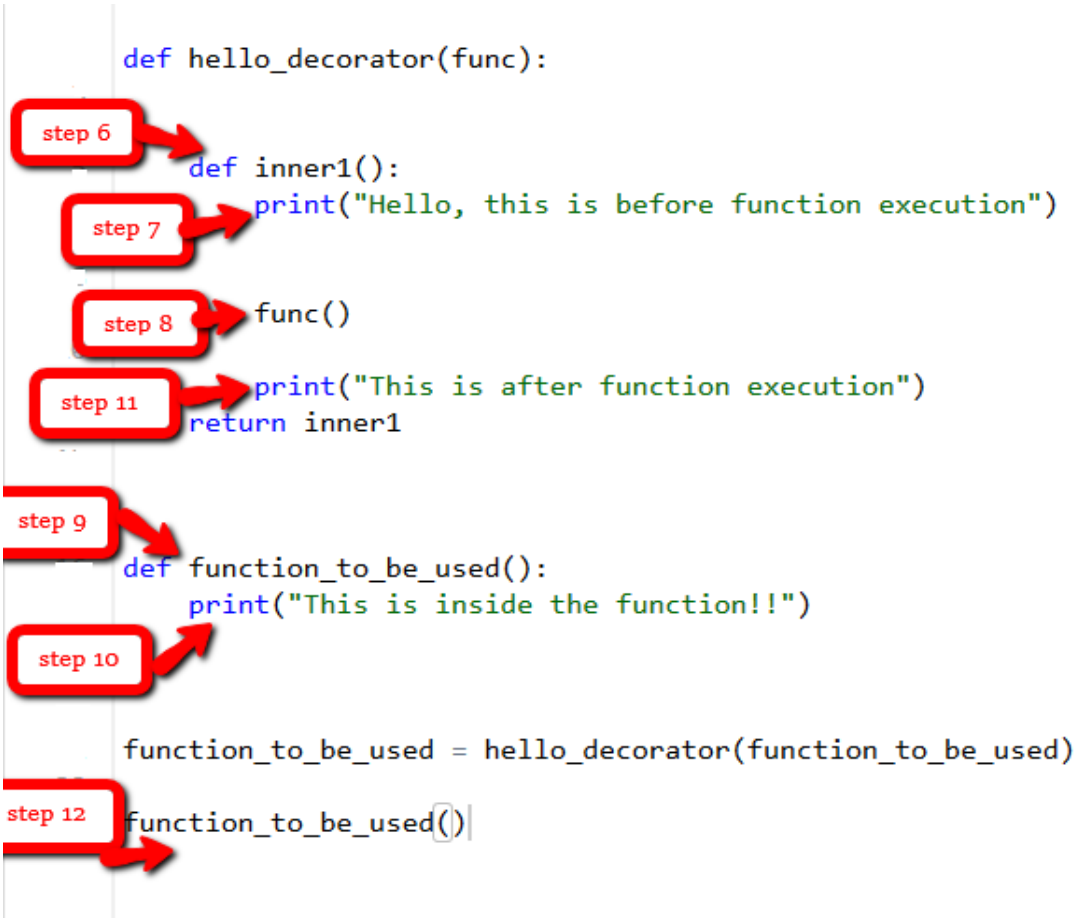
Example:

```
# 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 now 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
```

Let's see the behaviour of the above code and how it runs step by step when the "function_to_be_used" is called.


step 2
def hello_decorator(func):
step 3
def inner1():
print("Hello, this is before function execution")
func()
print("This is after function execution")
step 4
return inner1
def function_to_be_used():
print("This is inside the function!!")
step 1
function_to_be_used = hello_decorator(function_to_be_used)
step 5
function_to_be_used()


def hello_decorator(func):
step 6
def inner1():
step 7
print("Hello, this is before function execution")
step 8
func()
step 11
print("This is after function execution")
return inner1
step 9
def function_to_be_used():
print("This is inside the function!!")
step 10
function_to_be_used = hello_decorator(function_to_be_used)
step 12
function_to_be_used()

Another example where we can easily find out the execution time of a function using a decorator.

```
# 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
```

Example:

```
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
```

In the above example, you may notice a keen difference in the parameters of the inner function. The inner function takes the argument as *args and **kwargs which means that a tuple of positional arguments or a dictionary of keyword arguments can be passed of any length. This makes it a general decorator that can decorate a function having any number of arguments.

In simpler terms chaining decorators 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

Python Generators:

In Python, a generator is a function that returns an iterator that produces a sequence of values when iterated over.

Generators are useful when we want to produce a large sequence of values, but we don't want to store all of them in memory at once.

Create Python Generator:

In Python, similar to defining a normal function, we can define a generator function using the def keyword, but instead of the return statement we use the **yield** statement.

```
def generator_name(arg):
    # statements
    yield something
```

Here, the **yield** keyword is used to produce a value from the generator.

When the generator function is called, it does not execute the function body immediately. Instead, it returns a generator object that can be iterated over to produce the values.

Example:

```
def my_generator(n):
    # initialize counter
    value = 0
    # loop until counter is less than n
    while value < n:
        # produce the current value of the counter
        yield value
        # increment the counter
        value += 1
# iterate over the generator object produced by my_generator
for value in my_generator(3):
    # print each value produced by generator
    print(value)
```

Output:

```
0
1
2
```

In the above example, the **my_generator()** generator function takes an integer n as an argument and produces a sequence of numbers from **0 to n-1.**

The **yield** keyword is used to produce a value from the generator and pause the generator function's execution until the next value is requested.

The **for** loop iterates over the generator object produced by **my_generator(),** and the print statement prints each value produced by the generator.

We can also create a generator object from the generator function by calling the function like we would any other function as,

```
generator = my_range(3)
print(next(generator))  # 0
print(next(generator))  # 1
print(next(generator))  # 2
```

Python Generator Expression:

In Python, a generator expression is a concise way to create a generator object.

It is similar to a list comprehension, but instead of creating a list, it creates a generator object that can be iterated over to produce the values in the generator.

Syntax: (expression for item in iterable)

Here, **expression** is a value that will be returned for each item in the **iterable**.

The generator **expression** creates a generator object that produces the values of expression for each item in the **iterable**, one at a time, when iterated over.

Example:

```
# create the generator object
squares_generator = (i * i for i in range(5))
# iterate over the generator and print the values
for i in squares_generator:
  print(i)
```

Output: **0**
1
4

9

16

Here, we have created the generator object that will produce the squares of the numbers **0** through **4** when iterated over.

And then, to iterate over the generator and get the values, we have used the for loop.

Use of Python Generators:

There are several reasons that make generators a powerful implementation.

1. **Easy to Implement:**

   Generators can be implemented in a clear and concise way as compared to their iterator class counterpart. Following is an example to implement a sequence of power of 2 using an iterator class.

```
class PowTwo:
  def __init__(self, max=0):
    self.n = 0
    self.max = max
  def __iter__(self):
    return self
  def __next__(self):
    if self.n > self.max:
      raise StopIteration
    result = 2 ** self.n
    self.n += 1
    return result
```

   The above program was lengthy and confusing. Now, let's do the same using a generator function.

```
def PowTwoGen(max=0):
  n = 0
  while n < max:
    yield 2 ** n
    n += 1
```

   Since generators keep track of details automatically, the implementation was concise and much cleaner.

2. **Memory Efficient:**

   A normal function to return a sequence will create the entire sequence in memory before returning the result. This is an overkill, if the number of items in the sequence is very large. Generator implementation of such sequences is memory friendly and is preferred since it only produces one item at a time.

3. **Represent Infinite Stream**

   Generators are excellent mediums to represent an infinite stream of data. Infinite streams cannot be stored in memory, and since generators produce only one item at a time, they can represent an infinite stream of data.

   The following generator function can generate all the even numbers (at least in theory).

```
def all_even():
    n = 0
    while True:
        yield n
        n += 2
```

4. **Pipelining Generators**

   Multiple generators can be used to pipeline a series of operations. This is best illustrated using an example. Suppose we have a generator that produces the numbers in the Fibonacci series. And we have another generator for squaring numbers. If we want to find out the sum of squares of numbers in the Fibonacci series, we can do it in the following way by pipelining the output of generator functions together.

```
def fibonacci_numbers(nums):
    x, y = 0, 1
    for _ in range(nums):
        x, y = y, x+y
        yield x
def square(nums):
    for num in nums:
        yield num**2
print(sum(square(fibonacci_numbers(10))))
```

   Output: 4895

2.4 Module basic usage, namespaces, reloading modules. – math, random, datetime, etc.

Python Module:

A Python module is a file containing Python definitions and statements. A module can define functions, classes, and variables. A module can also include runnable code. Grouping related code into a module makes the code easier to understand and use. It also makes the code logically organized.

Module is a file that contains code to perform a specific task. A module may contain variables, functions, classes etc.

Create a simple Python module

Let's create a simple calc.py in which we define two functions, one add and another subtract.

```
# A simple module, calc.py
def add(x, y):
    return (x+y)
def subtract(x, y):
    return (x-y)
```

Example: example.py

```
# Python Module addition
def add(a, b):
    result = a + b
    return result
```

Import Module in Python:

We can import the functions, and classes defined in a module to another module using the import statement in some other Python source file.

When the interpreter encounters an import statement, it imports the module if the module is present in the search path. A search path is a list of directories that the interpreter searches for importing a module. For example, to import the module calc.py, we need to put the following command at the top of the script.

Syntax: **import module**

Note: This does not import the functions or classes directly instead imports the module only. To access the functions inside the module the dot(.) operator is used.

```
# importing  module calc.py
import calc
print(calc.add(10, 2))
```

Output: 12

The from-import Statement in Python:

Python's from statement lets you import specific attributes from a module without importing the module as a whole.

Importing specific attributes from the module:

Here, we are importing specific sqrt and factorial attributes from the math module.

```
# importing sqrt() and factorial from the
# module math
from math import sqrt, factorial
# if we simply do "import math", then
# math.sqrt(16) and math.factorial()
# are required.
print(sqrt(16))
print(factorial(6))
```

Output: **4.0**

720

Import all Names:

The * symbol used with the from import statement is used to import all the names from a module to a current namespace.

Syntax: from module_name import *

The use of * has its advantages and disadvantages. If you know exactly what you will be needing from the module, it is not recommended to use *, else do so.

```
# importing sqrt() and factorial from the
# module math
from math import *
```

```
# if we simply do "import math", then
# math.sqrt(16) and math.factorial()
# are required.
print(sqrt(16))
print(factorial(6))
```

Output: **4.0**

720

Locating Python Modules:

Whenever a module is imported in Python the interpreter looks for several locations. First, it will check for the built-in module, if not found then it looks for a list of directories defined in the sys.path. Python interpreter searches for the module in the following manner –

- First, it searches for the module in the current directory.
- If the module isn't found in the current directory, Python then searches each directory in the shell variable PYTHONPATH. The PYTHONPATH is an environment variable, consisting of a list of directories.
- If that also fails python checks the installation-dependent list of directories configured at the time Python is installed.
- Here, sys.path is a built-in variable within the sys module. It contains a list of directories that the interpreter will search for the required module.

```
# importing sys module
import sys
# importing sys.path
print(sys.path)
```

Output: ['C:\\Users\\HOD\\Desktop',
'C:\\Users\\HOD\\AppData\\Local\\Programs\\Python\\Python311\\Lib\\idlelib',
'C:\\Users\\HOD\\AppData\\Local\\Programs\\Python\\Python311\\python311.zip',
'C:\\Users\\HOD\\AppData\\Local\\Programs\\Python\\Python311\\DLLs',
'C:\\Users\\HOD\\AppData\\Local\\Programs\\Python\\Python311\\Lib',
'C:\\Users\\HOD\\AppData\\Local\\Programs\\Python\\Python311',
'C:\\Users\\HOD\\AppData\\Local\\Programs\\Python\\Python311\\Lib\\site-packages']

Renaming the Python module:

We can rename the module while importing it using the keyword.

Syntax: Import Module_name as Alias_name

```
# importing sqrt() and factorial from the
# module math
import math as mt
# if we simply do "import math", then
# math.sqrt(16) and math.factorial()
# are required.
print(mt.sqrt(16))
print(mt.factorial(6))
```

Output: **4.0**

720

The dir() built-in function:

In Python, we can use the **dir()** function to list all the function names in a module. For example, earlier we have defined a function **add()** in the module **example**.

```
dir(example)
['__builtins__',
'__cached__',
'__doc__',
'__file__',
'__initializing__',
'__loader__',
'__name__',
'__package__',
'add']
```

Here, we can see a sorted list of names (along with add). All other names that begin with an underscore are default Python attributes associated with the module (not user-defined).

For example, the **__name__** attribute contains the name of the module.

```
import example
example.__name__
```

Output: example

All the names defined in our current namespace can be found out using the **dir()** function without any arguments.

All the names defined in our current namespace can be found out using the **dir()** function without any arguments.

```
a = 1
b = "hello"
import math
dir()
['__builtins__', '__doc__', '__name__', 'a', 'b', 'math', 'pyscripter']
```

math module in Python:

The math module is a standard module in Python and is always available. To use mathematical functions under this module, you have to import the module using import math. It gives access to the underlying C library functions.

```
# Square root calculation
import math
math.sqrt(4)
```

This module does not support **complex** datatypes. The cmath module is the **complex** counterpart.

Functions in Python Math Module:

List of Functions in Python Math Module	
Function	Description
ceil(x)	Returns the smallest integer greater than or equal to x.
copysign(x, y)	Returns x with the sign of y
fabs(x)	Returns the absolute value of x
factorial(x)	Returns the factorial of x
floor(x)	Returns the largest integer less than or equal to x
fmod(x, y)	Returns the remainder when x is divided by y
frexp(x)	Returns the mantissa and exponent of x as the pair (m, e)

fsum(iterable)	Returns an accurate floating point sum of values in the iterable
isfinite(x)	Returns True if x is neither an infinity nor a NaN (Not a Number)
isinf(x)	Returns True if x is a positive or negative infinity
isnan(x)	Returns True if x is a NaN
ldexp(x, i)	Returns x * (2**i)
modf(x)	Returns the fractional and integer parts of x
trunc(x)	Returns the truncated integer value of x
exp(x)	Returns e**x
expm1(x)	Returns e**x - 1
log(x[, b])	Returns the logarithm of x to the base b (defaults to e)
log1p(x)	Returns the natural logarithm of 1+x
log2(x)	Returns the base-2 logarithm of x
log10(x)	Returns the base-10 logarithm of x
pow(x, y)	Returns x raised to the power y
sqrt(x)	Returns the square root of x
acos(x)	Returns the arc cosine of x
asin(x)	Returns the arc sine of x
atan(x)	Returns the arc tangent of x
atan2(y, x)	Returns atan(y / x)
cos(x)	Returns the cosine of x
hypot(x, y)	Returns the Euclidean norm, sqrt(x*x + y*y)
sin(x)	Returns the sine of x
tan(x)	Returns the tangent of x
degrees(x)	Converts angle x from radians to degrees
radians(x)	Converts angle x from degrees to radians
acosh(x)	Returns the inverse hyperbolic cosine of x
asinh(x)	Returns the inverse hyperbolic sine of x
atanh(x)	Returns the inverse hyperbolic tangent of x
cosh(x)	Returns the hyperbolic cosine of x
sinh(x)	Returns the hyperbolic cosine of x
tanh(x)	Returns the hyperbolic tangent of x

erf(x)	Returns the error function at x
erfc(x)	Returns the complementary error function at x
gamma(x)	Returns the Gamma function at x
lgamma(x)	Returns the natural logarithm of the absolute value of the Gamma function at x
pi	Mathematical constant, the ratio of circumference of a circle to it's diameter (3.14159...)
e	mathematical constant e (2.71828...)

Python Random Module:

You can generate random numbers in Python by using random module. These are pseudo-random number as the sequence of number generated depends on the seed. If the seeding value is same, the sequence will be the same. For example, if you use 2 as the seeding value, you will always see the following sequence.

```
import random
random.seed(2)
print(random.random())
print(random.random())
print(random.random())
```

Output: **0.9560342718892494**
0.9478274870593494
0.05655136772680869

List of Functions in Python Random Module	
Function	Description
seed(a=None, version=2)	Initialize the random number generator
getstate()	Returns an object capturing the current internal state of the generator
setstate(state)	Restores the internal state of the generator
getrandbits(k)	Returns a Python integer with k random bits
randrange(start, stop[, step])	Returns a random integer from the range
randint(a, b)	Returns a random integer between a and b inclusive
choice(seq)	Return a random element from the non-empty sequence

shuffle(seq)	Shuffle the sequence
sample(population, k)	Return a k length list of unique elements chosen from the population sequence
random()	Return the next random floating point number in the range [0.0, 1.0)
uniform(a, b)	Return a random floating point number between a and b inclusive
triangular(low, high, mode)	Return a random floating point number between low and high, with the specified mode between those bounds
betavariate(alpha, beta)	Beta distribution
expovariate(lambd)	Exponential distribution
gammavariate(alpha, beta)	Gamma distribution
gauss(mu, sigma)	Gaussian distribution
lognormvariate(mu, sigma)	Log normal distribution
normalvariate(mu, sigma)	Normal distribution
vonmisesvariate(mu, kappa)	Vonmises distribution
paretovariate(alpha)	Pareto distribution
weibullvariate(alpha, beta)	Weibull distribution

Python datetime Module:

Python has a module named datetime to work with dates and times. It provides a variety of classes for representing and manipulating dates and times, as well as for formatting and parsing dates and times in a variety of formats.

Example 1: Get Current Date and Time:

```
import datetime
# get the current date and time
now = datetime.datetime.now()
print(now)
```

Output: 2023-05-11 15:18:49.212613

Here, we have imported the **datetime** module using the **import datetime** statement. One of the classes defined in the **datetime** module is the **datetime** class. We then used the **now()** method to create a **datetime** object containing the current local date and time.

Example 2: Get Current Date

```
import datetime
```

```
# get current date
current_date = datetime.date.today()
print(current_date)
```

Output: 2023-05-11

In the above example, we have used the **today()** method defined in the **date** class to get a **date** object containing the current local date.

Attributes of datetime Module:

```
import datetime
print(dir(datetime))
```

Output:

['MAXYEAR', 'MINYEAR', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__', 'date', 'datetime', 'datetime_CAPI', 'sys', 'time', 'timedelta', 'timezone', 'tzinfo']

Among all the attributes of datetime module, the most commonly used classes in the datetime module are:

- **datetime.datetime** - represents a single point in time, including a date and a time.
- **datetime.date -** represents a date (year, month, and day) without a time.
- **datetime.time** - represents a time (hour, minute, second, and microsecond) without a date.
- **datetime.timedelta** - represents a duration, which can be used to perform arithmetic with datetime objects.

Example : Print today's year, month and day

```
from datetime import date
# date object of today's date
today = date.today()
print("Current year:", today.year)
print("Current month:", today.month)
print("Current day:", today.day)
```

Output: **Current year: 2023**
Current month: 5
Current day: 11

Python datetime.time Class:

A time object instantiated from the **time** class represents the local time.

```
from datetime import time
# time(hour = 0, minute = 0, second = 0)
a = time()
print(a)
# time(hour, minute and second)
b = time(11, 34, 56)
print(b)
# time(hour, minute and second)
c = time(hour = 11, minute = 34, second = 56)
print(c)
# time(hour, minute, second, microsecond)
d = time(11, 34, 56, 234566)
print(d)
```

Output: **00:00:00**

11:34:56

11:34:56

11:34:56.234566

Example: Print hour, minute, second and microsecond

Once we create the **time** object, we can easily print its attributes such as **hour**, **minute**

```
from datetime import time
a = time(11, 34, 56)
print("Hour =", a.hour)
print("Minute =", a.minute)
print("Second =", a.second)
print("Microsecond =", a.microsecond)
```

Ootput: Hour = 11
Minute = 34

Second = 56

Microsecond = 0

Example: Print year, month, hour, minute and timestamp

```
from datetime import datetime
a = datetime(2022, 12, 28, 23, 55, 59, 342380)
print("Year =", a.year)
print("Month =", a.month)
print("Hour =", a.hour)
print("Minute =", a.minute)
print("Timestamp =", a.timestamp())
```

Output:

Year = 2022

Month = 12

Hour = 23

Minute = 55

Timestamp = 1672251959.34238

Example:

```
# importing built-in module math
import math
# using square root(sqrt) function contained in math module
print(math.sqrt(25))
# using pi function contained in math module
print(math.pi)
# 2 radians = 114.59 degrees
print(math.degrees(2))
# 60 degrees = 1.04 radians
print(math.radians(60))
# Sine of 2 radians
print(math.sin(2))
# Cosine of 0.5 radians
print(math.cos(0.5))
# Tangent of 0.23 radians
print(math.tan(0.23))
# 1 * 2 * 3 * 4 = 24
print(math.factorial(4))
```

```
# importing built in module random
import random
# printing random integer between 0 and 5
print(random.randint(0, 5))
# print random floating point number between 0 and 1
print(random.random())
# random number between 0 and 100
print(random.random() * 100)
List = [1, 4, True, 800, "python", 27, "hello"]
# using choice function in random module for choosing a random element from
a set such as a list
print(random.choice(List))
# importing built in module datetime
import datetime
from datetime import date
import time
# Returns the number of seconds since the Unix Epoch, January 1st 1970
print(time.time())
# Converts a number of seconds to a date object
print(date.fromtimestamp(454554))
```

Output:

```
5.0
3.141592653589793
114.59155902616465
1.0471975511965976
0.9092974268256817
0.8775825618903728
0.23414336235146527
24
3
0.45311797333062176
59.63729354765712
python
1683801210.8296685
```

1970-01-06

Python Namespace:

To simply put it, a namespace is a collection of names. In Python, we can imagine a namespace as a mapping of every name we have defined to corresponding objects. It is used to store the values of variables and other objects in the program, and to associate them with a specific name. This allows us to use the same name for different variables or objects in different parts of your code, without causing any conflicts or confusion.

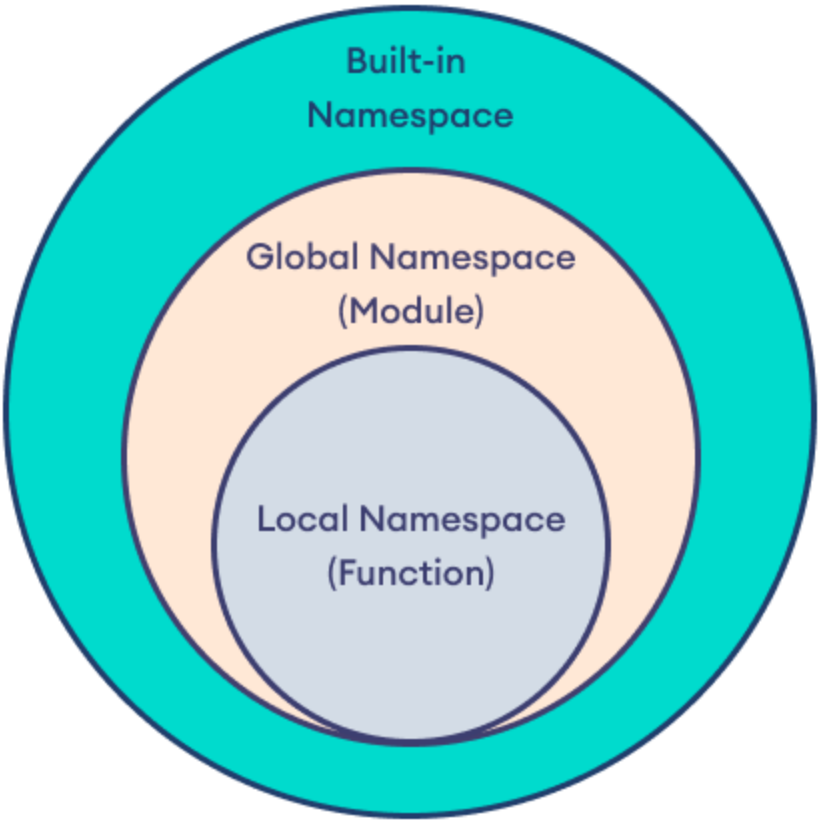
Types of Python namespace:

A namespace containing all the **built-in** names is created when we start the Python interpreter and exists as long as the interpreter runs.

This is the reason that built-in functions like **id(), print()** etc. are always available to us from any part of the program. Each module creates its own **global namespace**.

These different namespaces are isolated. Hence, the same name that may exist in different modules does not collide.

Modules can have various functions and classes. A local namespace is created when a function is called, which has all the names defined in it.



Python Variable Scope:

Although there are various unique namespaces defined, we may not be able to access all of them from every part of the program. The concept of scope comes into play.

A scope is the portion of a program from where a namespace can be accessed directly without any prefix.

At any given moment, there are at least three nested scopes.

1. Scope of the current function which has local names
2. Scope of the module which has global names
3. Outermost scope which has built-in names

When a reference is made inside a function, the name is searched in the local namespace, then in the global namespace and finally in the built-in namespace.

Example 1: Scope and Namespace in Python

```
# global_var is in the global namespace
global_var = 10
def outer_function():
    #  outer_var is in the local namespace
    outer_var = 20
    def inner_function():
        #  inner_var is in the nested local namespace
        inner_var = 30
        print(inner_var)
    print(outer_var)
    inner_function()
# print the value of the global variable
print(global_var)
# call the outer function and print local and nested local variables
outer_function()
```

Output: **10**

20

30

In the above example, there are three separate namespaces: the global namespace, the local namespace within the outer function, and the local namespace within the inner function.

Here,

- **global_var** - is in the global namespace with value **10**
- **outer_val** - is in the local namespace of **outer_function()** with value **20**
- **inner_val** - is in the nested local namespace of **inner_function()** with value **30**

When the code is executed, the **global_var** global variable is printed first, followed by the local variable: **outer_var** and **inner_var** when the outer and inner functions are called.

Example 2: Use of global Keyword in Python

```
# define global variable
global_var = 10
def my_function():
    # define local variable
    local_var = 20
    # modify global variable value
    global global_var
    global_var = 30
# print global variable value
print(global_var)
# call the function and modify the global variable
my_function()
# print the modified value of the global variable
print(global_var)
```

Output: 10
30

Here, when the function is called, the global keyword is used to indicate that **global_var** is a **global** variable, and its value is modified to **30.**

So, when the code is executed, **global_var** is printed first with a value of **10**, then the function is called and the global variable is modified to **30** from the inside of the function.

And finally the modified value of **global_var** is printed again.

2.1 Package: import basics

Python modules may contain several classes, functions, variables, etc. whereas Python packages contain several modules. In simpler terms, Package in Python is a folder that contains various modules as files.

Creating Package

Let's create a package in Python named mypckg that will contain two modules mod1 and mod2. To create this module follow the below steps:

- Create a folder named mypckg.
- Inside this folder create an empty Python file i.e. __init__.py
- Then create two modules mod1 and mod2 in this folder.

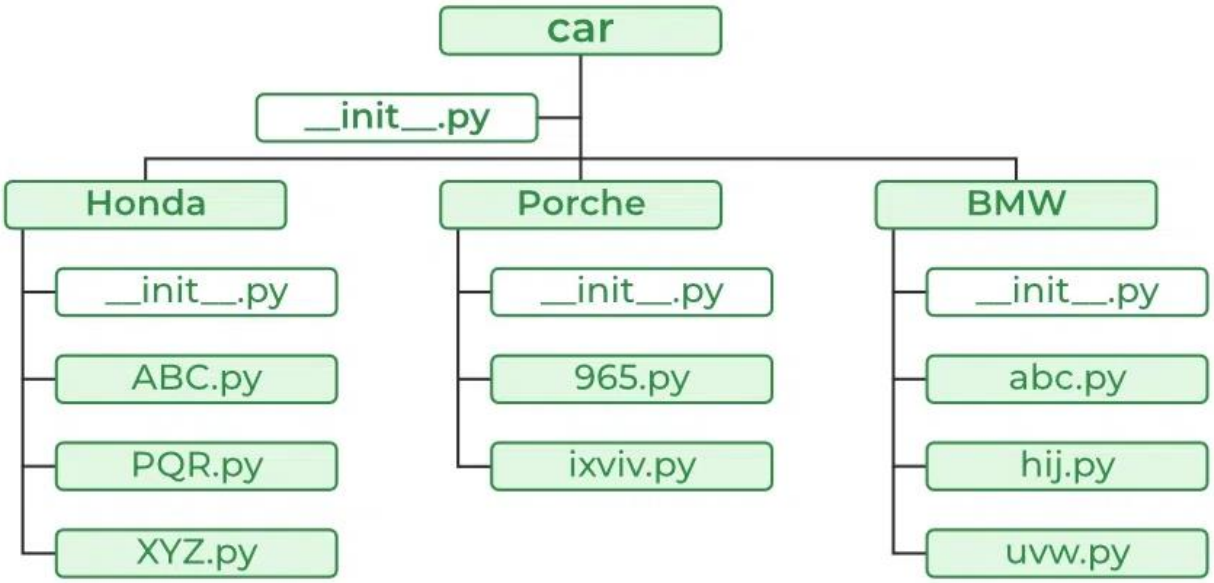
Mod1.py

```
def show():
  print("Welcome to ATESTC")
```

Mod2.py

```
def sum(a, b):
  return a+b
```

The Hierarchy of our Python package looks like this:



Understanding __init__.py

__init__.py helps the Python interpreter recognize the folder as a package. It also specifies the resources to be imported from the modules. If the **__init__.py** is empty this means that all the functions of the modules will be imported. We can also specify the functions from each module to be made available.

For example, we can also create the **__init__.py** file for the above module as:

```
from .mod1 import show
from .mod2 import sum
```

This **__init__.py** will only allow the show and sum functions from the **mod1** and **mod2** modules to be imported.

Import Modules from a Package:

We can import these Python modules using the from…import statement and the **dot(.)** operator.

Syntax: **import package_name.module_name**

Example1: We will import the modules from the above-created package and will use the functions inside those modules.

```
from mypckg import mod1
from mypckg import mod2
mod1.show()
res = mod2.sum(1, 2)
print(res)
```

Output: Welcome to ATESTC

3

Example 2: We can also import the specific function also using the same syntax.

```
from mypckg.mod1 import show
from mypckg.mod2 import sum
show()
res = sum(1, 2)
print(res)
```

Output: **Welcome to GFG**

3

Extra Readings: GUI framework in python