2. Python Functions, Modules & Packages

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

2.2. built-in functions

2.3. Arguments Passing, Anonymous Function: lambda

2.4. Decorators and Generators

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

2.6. Package: import basics

2.7. Python namespace packages

2.8. User defined modules and packages

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

Python - Functions

Python includes many built-in functions. These functions perform a predefined task and can be called upon in any program, as per requirement. **However, if you don't find a suitable built-in function to serve your purpose, you can define one**. We will now see how to define and use a function in a Python program.

Defining a Function

A function is a reusable block of programming statements designed to perform a certain task. To define a function, Python provides the **def** keyword. The following is the syntax of defining a function.

Syntax:

```
def function_name(parameters):
    """docstring"""
    statement1
    statement2
    ...
    ...
    return [expr]
```

The keyword **def** is followed by a suitable identifier as the name of the function and parentheses. One or more parameters may be optionally mentioned inside parentheses. The **:** symbol after parentheses starts an indented block.

The first statement in the function body can be a string, which is called the **docstring**. It explains the functionality of the function/class. **The docstring is not mandatory.**

The function body contains one or more statements that perform some actions. **It can also use pass keyword.**

Optionally, the last statement in the function block is the **return statement.** It sends an execution control back to calling the environment. If an expression is added in front of return, its value is also returned to the calling code.

The following example defines the **greet()** function.

Example: User-defined Function

```
def greet():
    """This function displays 'Hello World!'"""
    print('Hello World!')
```

Above, we have defined the **greet()** function. The first statement is a **docstring** that mentions what this function does. The second like is a print method that displays the specified string to the console. Note that it does not have the return statement.

To call a defined function, just use its name as a statement anywhere in the code. For example, the above function can be called using parenthesis, greet().

Example: Calling User-defined Function

```
greet()
```

Output

```
Hello World!
```

By default, all the functions return None if the return statement does not exist.

Example: Calling User-defined Function

```
val = greet()
print(val)
```

Output

```
None
```

The help() function displays the docstring, as shown below.

Example: Calling User-defined Function

```
>>> help(greet)
Help on function greet in module __main__:

    greet()
        This function displays 'Hello World!'
```

Function Parameters / arguments

It is possible to define a function to receive one or more parameters (also called arguments) and use them for processing inside the function block. Parameters/arguments may be given suitable formal names. The greet() function is now defined to receive a string parameter called name. Inside the function, the print() statement is modified to display the greeting message addressed to the received parameter.

Example: Parameterized Function

```
def greet(name):
    print ('Hello ', name)

greet('Steve') # calling function with argument
greet(123)
```

Output

```
Hello Steve
Hello 123
```

The names of the arguments used in the definition of the function are called formal arguments/parameters.

Objects actually used while calling the function are called actual arguments/parameters.

The function parameters can have an annotation to specify the type of the parameter using parameter:type syntax. For example, the following annotates the parameter type string.

Example: Parameterized Function

```
def greet(name:str):
    print ('Hello ', name)

greet('Steve') # calling function with string argument
greet(123) # raise an error for int argument
```

Multiple Parameters

A function can have multiple parameters. The following function takes three arguments.

Example: Parameterized Function

```
def greet(name1, name2, name3):
    print ('Hello ', name1, ' , ', name2 , ', and ', name3)

greet('Steve', 'Bill', 'Yash') # calling function with string argument
```

Output

Hello Steve, Bill, and Yash

Unknown Number of Arguments

A function in Python can have an unknown number of arguments by putting `*` before the parameter if you don't know the number of arguments the user is going to pass.

Example: Parameterized Function

```
def greet(*names):
    print ('Hello ', names[0], ', ', names[1], ', ', names[3])

greet('Steve', 'Bill', 'Yash')
```

Output

Hello Steve, Bill, and Yash

The following function works with any number of arguments.

Example: Parameterized Function

```
 def greet(*names):
            i=0
            print('Hello ', end= “”)
            while len(names) > i:
                        print(names[i], end=', ')
                        i+=1

greet('Steve', 'Bill', 'Yash')
```

```
greet('Steve', 'Bill', 'Yash', 'Kapil', 'John', 'Amir')
```

Output

```
Hello Steve, Bill, Yash,
Hello Steve, Bill, Yash, Kapil, John, Amir,
```

Function with Keyword Arguments

In order to call a function with arguments, the same number of actual arguments must be provided. However, a function can be called by passing parameter values using the parameter names in any order. For example, the following passes values using the parameter names.

```
def greet(firstname, lastname):
    print ('Hello', firstname, lastname)

greet(lastname='Jobs', firstname='Steve') # passing parameters in any order using keyword argument
```

Output

```
Hello Steve Jobs
```

Keyword Argument **kwarg

The function can have a single parameter prefixed with `**`. This type of parameter initialized to a new ordered mapping receiving any excess keyword arguments, defaulting to a new empty mapping of the same type.

Example: Parameterized Function

```
def greet(**person):
        print('Hello ', person['firstname'],  person['lastname'])

greet(firstname='Steve', lastname='Jobs')
greet(lastname='Jobs', firstname='Steve')
greet(firstname='Bill', lastname='Gates', age=55)
greet(firstname='Bill') # raises KeyError
```

Output

```
Hello Steve Jobs
Hello Steve Jobs
```

Hello Bill Gates

When using the ** parameter, the order of arguments does not matter. However, the name of the arguments must be the same. Access the value of keyword arguments using paramter_name['keyword_argument'].

If the function access the keyword argument but the calling code does not pass that keyword argument, then it will raise the KeyError exception, as shown below.

Example: Parameterized Function

```
def greet(**person):
        print('Hello ', person['firstname'],  person['lastname'])

greet(firstname='Bill') # raises KeyError, must provide 'lastname' arguement
```

Output

```
Traceback (most recent call last):
  File "<pyshell#21>", line 1, in <module>
    greet(firstname='Bill')
  File "<pyshell#19>", line 2, in greet
    print('Hello ', person['firstname'],  person['lastname'])
KeyError: 'lastname'
```

Parameter with Default Value

While defining a function, its parameters may be assigned default values. This default value gets substituted if an appropriate actual argument is passed when the function is called. However, if the actual argument is not provided, the default value will be used inside the function.

The following greet() function is defined with the name parameter having the default value 'Guest'. It will be replaced only if some actual argument is passed.

Example: Parameter with Default Value

```
def greet(name = 'Guest'):
    print ('Hello', name)
```

```
greet()
greet('Steve')
```

Output

```
Hello Guest
Hello Steve
```

Function with Return Value

Most of the time, we need the result of the function to be used in further processes. Hence, when a function returns, it should also return a value.

A user-defined function can also be made to return a value to the calling environment by putting an expression in front of the return statement. In this case, the returned value has to be assigned to some variable.

Example: Return Value

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

The above function can be called and provided the value, as shown below.

Example: Parameter with Default Value

```
 total=sum(10, 20)
print(total)
total=sum(5, sum(10, 20))
print(total)
```

Output

```
30
35
```

Variable Scope in Python

In general, a variable that is defined in a block is available in that block only. It is not accessible outside the block. Such a variable is called **a local variable.** Formal argument identifiers also behave as local variables.

The following example will underline this point. An attempt to print a local variable outside its scope will raise the NameError exception.

Example: Local Variable

```
def greet():
    name = 'Steve'
    print('Hello ', name)
```

Here, name is a local variable for the greet() function and is not accessible outside of it.

Example: Local Variable

```
>>> greet()
Hello Steve
>>> name
Traceback (most recent call last):
File "<pyshell#4>", line 1, in <module> name
NameError: name 'name' is not defined
```

Any variable present outside any function block is called a global variable. Its value is accessible from inside any function. In the following example, the **name** variable is initialized before the function definition. Hence, it is a global variable.

Example: Global Variable

```
name='John'
def greet():
    name = ‘Steve’
    print ("Hello ", name)
```

Now, you can access the global variable name because it has been defined out of a function.

```
>>> greet()
Hello Steve
```

```
>>> name
'Steve'
```

However, if we assign another value to a globally declared variable inside the function, a new local variable is created in the function's namespace. This assignment will not alter the value of the global variable. For example:

Example: Local and Global Variables

```
name = 'Steve'
def greet():
    name = 'Bill'
    print('Hello ', name)
```

Now, changing the value of global variable `name` inside a function will not affect its global value.

```
>>> greet()
Hello Bill
>>> name
'Steve'
```

If you need to access and change the value of the global variable from within a function, this permission is granted by the `global` keyword.

Example: Access Global Variables

```
name = 'Steve'
def greet():
    global name
    name = 'Bill'
    print('Hello ', name)
```

The above would display the following output in the Python shell.

```
>>> name
'Steve'
>>> greet()
Hello Bill
>>> name
'Bill'
```

It is also possible to use a global and local variable with the same name simultaneously. Built-in function globals() returns a dictionary object of all global variables and their respective values. Using the name of the variable as a key, its value can be accessed and modified.

Example: Global Variables

```
name = 'Steve'
def greet():
    globals()['name'] = 'James'
    name='Steve'
    print ('Hello ', name)
```

The result of the above code shows a conflict between the global and local variables with the same name and how it is resolved.

```
>>> name
'Steve'
>>> greet()
Hello Steve
>>> name
'James'
```

if __name__ == “__main__”:

statement(s)

The special variable, __name__ with “__main__”, is the entry point to your program. When Python interpreter reads the *if* statement and sees that __name__ does equal to "__main__", it will execute the block of statements present there.

```
def greet():
   print(“Hello world”)

def main():
    greet()

if __name__ == “__main__”:
      main()
```

```
# Example 1: Nested functions

def talk(phrase):
    def say(word):
        print(word)

    words = phrase.split(' ')
    for word in words:
        say(word)

talk('I am going to buy the milk')
```

Recursion in Python

A function that calls itself is a recursive function. This method is used when a certain problem is defined in terms of itself. Although this involves iteration, using an iterative approach to solve such a problem can be tedious. The recursive approach provides a very concise solution to a seemingly complex problem. It looks glamorous but can be difficult to comprehend!

The most popular example of recursion is the calculation of the factorial. Mathematically the factorial is defined as: n! = n * (n-1)!

We use the factorial itself to define the factorial. Hence, this is a suitable case to write a recursive function. Let us expand the above definition for the calculation of the factorial value of 5.

5! = 5 X 4!
    5 X4 X 3!
    5 X4 X 3 X 2!
    5 X4 X 3 X 2 X 1!
    5 X4 X 3 X 2 X 1
  = 120

While we can perform this calculation using a loop, its recursive function involves successively calling it by decrementing the number until it reaches 1. The following is a recursive function to calculate the factorial.

Example: Recursive Function

```
def rec_factorial(n):
  if n == 1:
    return n
  else:
     return n*rec_factorial(n-1)
# take input from the user
num = int(input("Enter a number: "))
# check is the number is negative
if num < 0:
```

```
    print("Sorry, factorial does not exist for negative numbers")
elif num == 0:
    print("The factorial of 0 is 1")
else:
    print("The factorial of", num ,"is", rec_factorial(num))
```

When the factorial function is called with 5 as argument, successive calls to the same function are placed, while reducing the value of 5. Functions start returning to their earlier call after the argument reaches 1. The return value of the first call is a cumulative product of the return values of all calls.

Fibonacci sequence Using Recursion

Fibonacci sequence:

A Fibonacci sequence is a sequence of integers in which first two terms are 0 and 1 and all other terms of the sequence are obtained by adding their preceding two numbers.

For example: 0, 1, 1, 2, 3, 5, 8, 13 and so on...

See this example:

```
def recur_fibo(n):
    if n <= 1:
        return n
    else:
        return(recur_fibo(n-1) + recur_fibo(n-2))
# take input from the user
nterms = int(input("How many terms? "))
# check if the number of terms is valid
if nterms <= 0:
    print("Plese enter a positive integer")
else:
    print("Fibonacci sequence:")
    for i in range(nterms):
        print(recur_fibo(i))
```

2.2. built-in functions

Python - Built-in Modules

The Python interactive shell has a number of built-in functions. They are loaded automatically as a shell starts and are always available, such as print() and input() for I/O, number conversion functions int(), float(), complex(), data type conversions list(), tuple(), set(), etc.

In addition to built-in functions, a large number of pre-defined functions are also available as a part of libraries bundled with Python distributions. These functions are defined in modules are called built-in modules.

Built-in modules are written in C and integrated with the Python shell. Each built-in module contains resources for certain system-specific functionalities such as OS management, disk IO, etc. The standard library also contains many Python scripts (with the .py extension) containing useful utilities.

To display a list of all available modules, use the following command in the Python console:

Python IDLE

```
>>>help('modules')
```

Enter any module name to get more help. Or, type "modules spam" to search for modules whose name or summary contain the string “spam”.

2.3. Arguments Passing, Anonymous Function: lambda

Lambda Functions & Anonymous Functions in Python

The `lambda` keyword is used to define anonymous functions in Python. Usually, such a function is meant for **one-time use.**

Syntax:

```
lambda [arguments] : expression
```

The lambda function can have zero or more arguments before the **:** symbol. When this function is called, the expression after **:** is executed.

Example: Lambda Function

```
square = lambda x : x * x
```

Above, the lambda function starts with the `lambda` keyword followed by parameter `x`. An expression `x * x` after `:` returns the value of `x * x` to the caller. The whole lambda function **`lambda x : x * x`** is assigned to a variable **`square`** in order to call it like a named function. The **variable name** becomes the **function name** so that we can call it as **a regular function**, as shown below.

Example: Calling Lambda Function

```
>>> square(5)
25
```

The above lambda function definition is the same as the following function:

```
def square(x):
    return x * x
```

The expression does not need to always return a value. The following lambda function does not return anything.

Example: Lambda Function

```
>>> greet = lambda name: print('Hello ', name)
>>> greet('Steve')
Hello Steve
```

Note:

The lambda function can have only one expression. Obviously, it cannot substitute a function whose body may have conditionals, loops, etc.

The following lambda function contains multiple parameters:

Example: Lambda Function

```
>>> sum = lambda x, y, z : x + y + z
>>> sum(5, 10, 15)
30
```

The following lambda function can take any number of parameters:

Example: Lambda Function

```
>>> sum = lambda *x: x[0]+x[1]+x[2]+x[3]
>>> sum(5, 10, 15, 20)
50
```

Parameter less Lambda Function

The following is an example of the parameter less lambda function.

Example: parameter less Lambda Function

```
>>> greet = lambda : print('Hello World!')
>>> greet()
Hello World!
```

Anonymous Function

We can declare a lambda function and call it as an anonymous function, without assigning it to a variable.

Example: Parameter less Lambda Function

```
>>> (lambda x: x*x)(5)
25
```

Above, **lambda x: x*x** defines an anonymous function and call it once by passing arguments in the parenthesis **(lambda x: x*x)(5).**

In Python, **functions are the first-class citizens**, which mean that just as **literals, functions can also be passed as arguments.**

The **lambda functions** are useful when we want **to give** or **pass the function as one of the arguments to another function**. We can pass **the lambda function** without assigning it to **a variable**, as **an anonymous function** as **an argument** to another function.

Example: Parameter less Lambda Function

```
>>> def dosomething(fn):
        print('Calling function argument:')
        fn()
>>> dosomething(lambda : print('Hello World')) # passing anonymous function
Calling function argument:
Hello World
>>> myfn = lambda : print('Hello World')
>>> dosomething(myfn) # passing lambda function
```

Above, the dosomething() function is defined with the fn parameter which is called as a function inside dosomething(). The dosomething(lambda : print('Hello World')) calls the dosomething() function with **an anonymous lambda function as an argument**.

Python has built-in functions that take other functions as arguments.

The map(), filter() and reduce() functions are important functional programming tools. All of them take a function as their argument. The argument function can be a normal function or a lambda function.

Example: Pass Lambda Function to map()

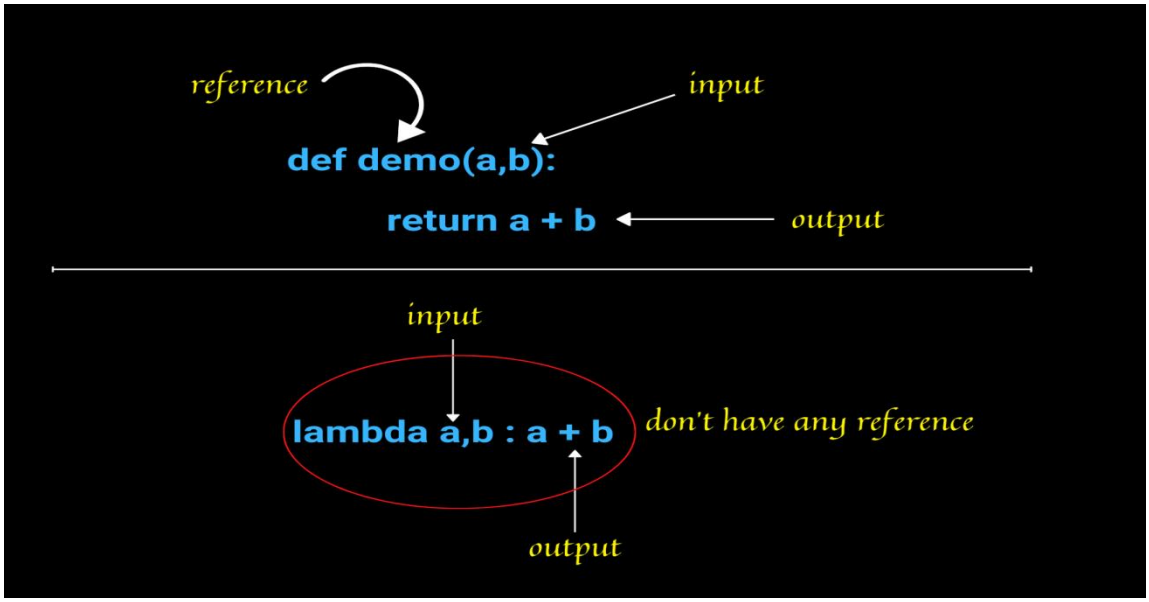
```
>>> sqrList = map(lambda x: x*x, [1, 2, 3, 4, 5]) # passing anonymous function
>>> next(sqrList)
1
>>> next(sqrList)
4
>>> next(sqrList)
9
>>> next(sqrList)
16
>>> next(sqrList)
25
```

Lambda Function

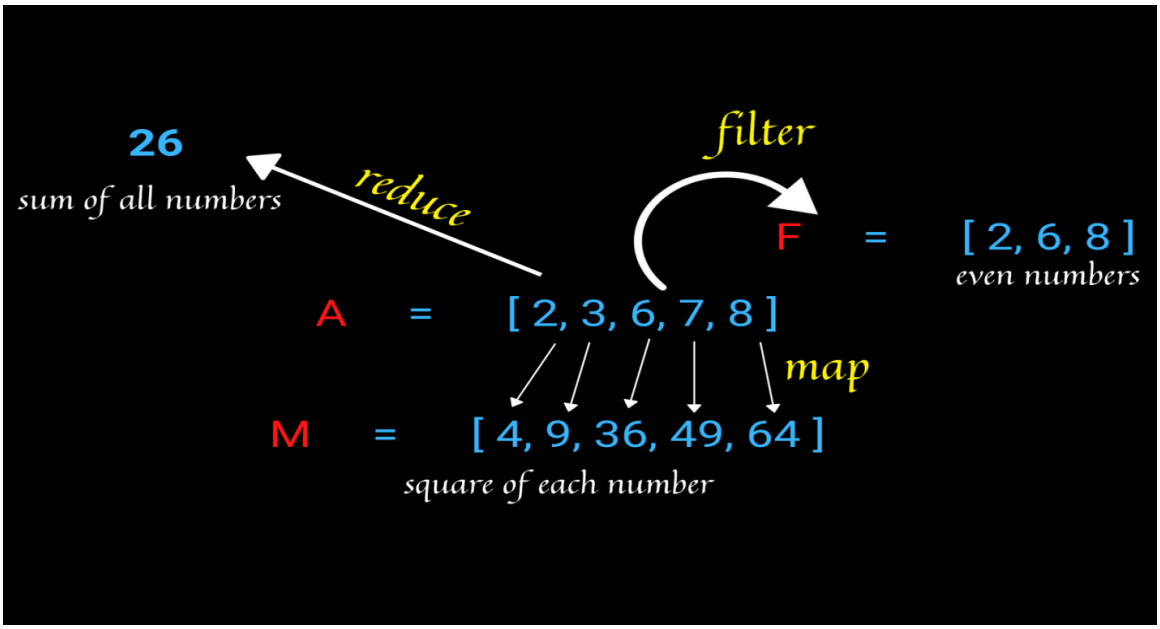
Normal function resides in memory for the lifetime of a program but what if we want to use a function for a single time, for that purpose there is a concept known as a **lambda function**.

It is a single line function and after execution, it gets deleted automatically from **RAM.** That's why it is also known as an **anonymous function**.

Let's see **normal function vs lambda function.**



Let's see how we can use the lambda function in map, filter, and reduce.



map:

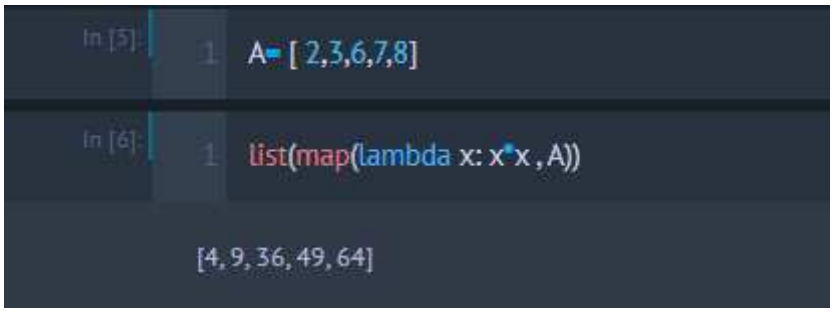
Map applies a function to all the items in a given iterable object.

It is a **one-to-one** relation.

Syntax: map(function, iterable object)

Suppose, we have **a list A= [2, 3, 6, 7, 8]** and we want to create **a new list** that will have **a square of numbers** from **list A**.

In such a situation, we can use the map function.



Here we have used a lambda function to calculate the square of a number. This function iterates over each element in A.

filter:

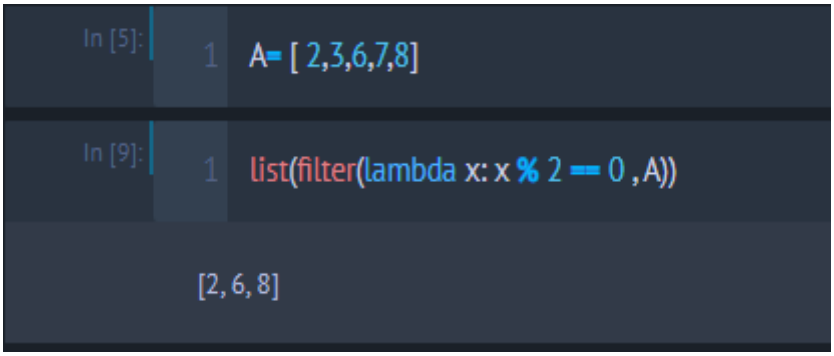
Filter creates a list of elements for which a function returns true.

Syntax: filter(function, iterable object)

Always remember when there is a situation where we want **m** outputs from **n** inputs where **m ≤ n,** in that case, we can use the filter function.

As the name suggests, it filters the values.

Suppose, we want values from A which are even.



If a value is True then and then it will be added to the list.

reduce:

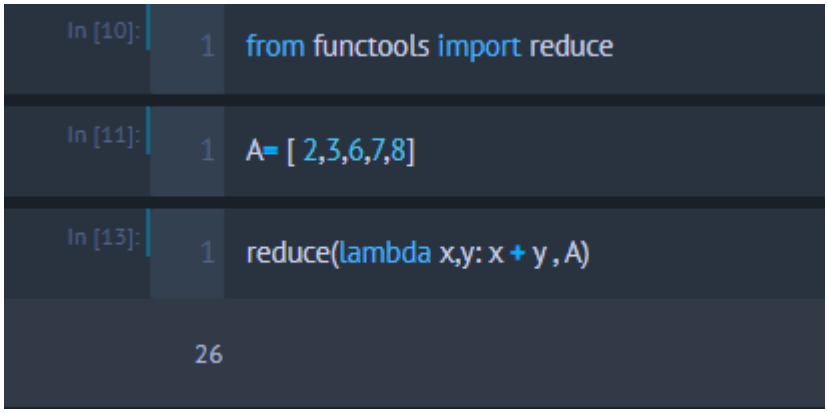
reduce always returns a single value.

It is just a many-to-one relation.

First, we have to import reduce module from functools.

Syntax: reduce(function, iterable object)

Let's see how we can use reduce to calculate the sum of all elements in a list.



Now you can try to calculate the maximum element from a list using reduce function.

2.4. Decorators and Generators

Python - Generator Functions

Python provides a generator to create your own iterator function. A generator is a special type of function which does not return a single value, instead, it returns an iterator object with a sequence of values. In a generator function, a yield statement is used rather than a return statement. The following is a simple generator function.

Example: Generator Function

```
def mygenerator():
    print('First item')
    yield 10

    print('Second item')
    yield 20
```

```
    print('Last item')
    yield 30
```

In the above example, the **mygenerator()** function is a generator function. It uses **yield** instead of return keyword. So, this will return the value against the **yield** keyword each time it is called. However, you need to create an iterator for this function, as shown below.

Example: next()

```
>>> gen = mygenerator()
>>> next(gen)
First item
10
>>> next(gen)
Second item
20
>>> next(gen)
Last item
30
```

The generator function cannot include the return keyword. If you include it, then it will terminate the function. The difference between yield and return is that yield returns a value and pauses the execution while maintaining the internal states, whereas the return statement returns a value and terminates the execution of the function.

The following generator function includes the return keyword.

Example: return in Generator Function

```
def mygenerator():
    print('First item')
    yield 10
    return
    print('Second item')
    yield 20
    print('Last item')
    yield 30
```

Now, execute the above function as shown below.

Example: Generator Function

```
>>> gen = mygenerator()
>>> next(gen)
First item
10
>>> next(gen)
Traceback (most recent call last):
    File "<pyshell#13>", line 1, in <module>

        it.__next__()
        StopIteration
```

As you can see, the above generator stops executing after getting the first item because the return keyword is used after yielding the first item.

Using for Loop with Generator Function

The generator function can also use the for loop.

Example: Use For Loop with Generator Function

```
def get_sequence_upto(x):
  for i in range(x):
    yield i
```

As you can see above, the get_sequence_upto function uses the yield keyword. The generator is called just like a normal function. However, its execution is paused on encountering the yield keyword. This sends the first value of the iterator stream to the calling environment. However, local variables and their states are saved internally.

The above generator function get_sequence_upto() can be called as below.

Example: Calling Generator Function

```
>>> seq = get_sequence_upto(5)
>>> next(seq)
0
>>> next(seq)
```

```
1
>>> next(seq)
2
>>> next(seq)
3
>>> next(seq)
4
>>> next(seq)
Traceback (most recent call last):
    File "<pyshell#13>", line 1, in <module>

        it.__next__()
        StopIteration
```

The function resumes when next() is issued to the iterator object. The function finally terminates when next() encounters the StopIteration error.

In the following example, function square_of_sequence() acts as a generator. It yields the square of a number successively on every call of next().

Example:

```
def square_of_sequence(x):
  for i in range(x):
    yield i*i
```

The following script shows how to call the above generator function.

```
gen=square_of_sequence(5)
while True:
  try:
    print ("Received on next(): ", next(gen))
  except StopIteration:
    break
```

The above script uses the try..except block to handle the StopIteration error. It will break the while loop once it catches the StopIteration error.

Output

Received on next(): 0

Received on next(): 1

Received on next(): 4

Received on next(): 9

Received on next(): 16

We can use the for loop to traverse the elements over the generator. In this case, the next() function is called implicitly and the StopIteration is also automatically taken care of.

Example: Generator with the For Loop

```
squres = square_of_sequence(5)
for sqr in squres:
    print(sqr)
```

Output

0

1

4

9

16

Note:

One of the advantages of the generator over the iterator is that elements are generated dynamically. Since the next item is generated only after the first is consumed, it is more memory efficient than the iterator.

Generator Expression

Python also provides a generator expression, which is a shorter way of defining simple generator functions. The generator expression is an anonymous generator function. The following is a generator expression for the square_of_sequence() function.

Example: Generator Expression

```
>>> squres = (x*x for x in range(5))
>>> print(next(squres))
0
```

```
>>> print(next(squres))
1
>>> print(next(squres))
4
>>> print(next(squre))
9
>>> print(next(squres))
16
```

In the above example, (x*x for x in range(5)) is a generator expression. The first part of an expression is the yield value and the second part is the for loop with the collection.

The generator expression can also be passed in a function. It should be passed without parentheses, as shown below.

Example: Passing Generator Function

```
>>> import math
>>> sum(x*x for x in range(5))
30
```

In the above example, a generator expression is passed without parentheses into the built-in function sum.

Decorators in Python

In programming, **decorator is a design pattern that adds additional responsibilities to an object dynamically.** In Python, **a function is the first-order object**. So, **a decorator in Python adds additional responsibilities / functionalities to a function dynamically without modifying a function.**

In Python, a function can be passed as an argument to another function. It is also possible to define a function inside another function, and a function can return another function.

So, **a decorator in Python is a function that receives another function as an argument. The behavior of the argument function is extended by the decorator without actually modifying it. The decorator function can be applied over a function using the @decorator syntax.**

Let's understand the decorator in Python step-by-step.

Consider that we have the greet() function, as shown below.

Example: A Function

```
def greet():
        print('Hello! ', end='')
```

Now, we can extend the above function's functionality without modifying it by passing it to another function, as shown below.

Example: A Function with Argument

```
def mydecorator(fn):
        fn()
        print('How are you?')
```

Above, the mydecorator() function takes a function as an argument. It calls the argument function and also prints some additional things. Thus, it extends the functionality of the greet() function without modifying it. However, it is not the actual decorator.

Example: Calling Function in Python Shell

```
>>> mydecorator(greet)
Hello! How are you?
```

The mydecorator() is not a decorator in Python. The decorator in Python can be defined over any appropriate function using the @decorator_function_name syntax to extend the functionality of the underlying function.

The following defines the decorator for the above greet() function.

Example: A Decorator Function

```
def mydecorator(fn):
   def inner_function():
      fn()
      print('How are you?')
   return inner_function
```

The mydecorator() function is the decorator function that takes a function (any function that does not take any argument) as an argument. The inner function inner_function() can access the outer function's argument, so it executes some

code before or after to extend the functionality before calling the argument function. The mydecorator function returns an inner function.

Now, we can use mydecorator as a decorator to apply over a function that does not take any argument, as shown below.

Example: Applying Decorator

```
@mydecorator
def greet():
        print('Hello! ', end=' ')
```

Now, calling the above greet() function will give the following output.

Example: Calling a Decorated Function

```
>>> greet()
Hello! How are you?
```

The mydecorator can be applied to any function that does not require any argument. For example:

Example: Applying Decorator

```
 @mydecorator
def dosomething():
        print('I am doing something.', end='')
```

Example: Calling Decorated Function in Python Shell

```
>>> dosomething()
I am doing something. How are you?
```

The typical decorator function will look like below.

Decorator Function Syntax

```
def mydecoratorfunction(some_function): # decorator function
   def inner_function():
      # write code to extend the behavior of some_function()
      some_function() # call some_function
      # write code to extend the behavior of some_function()
   return inner_function # return a wrapper function
```

Built-in Decorators

Python library contains many built-in decorators as a shortcut of defining properties, class method, static methods, etc.

Decorator	Description
@property	Declares a method as a property's setter or getter methods.
@classmethod	Declares a method as a class's method that can be called using the class name.
@staticmethod	Declares a method as a static method.

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

2.6. Package: import basics

2.8. User defined modules and packages

Python Modules

Any text file with the .py extension containing Python code is basically a module. Different Python objects such as functions, classes, variables, constants, etc., defined in one module can be made available to an interpreter session or another Python script by using the import statement. Functions defined in built-in modules need to be imported before use. On similar lines, a custom module may have one or more user-defined Python objects in it. These objects can be imported in the interpreter session or another script.

If the programming algorithm requires defining a lot of functions and classes, they are logically organized in modules. One module stores classes, functions and other resources of similar relevance. Such a modular structure of the code makes it easy to understand, use and maintain.

Creating a Module

Shown below is a Python script containing the definition of sum() function. It is saved as calc.py.

calc.py

```
def sum(x, y):
   return x + y
```

Importing a Module

We can now import this module and execute the sum() function in the Python shell.

Example: Importing a Module

```
>>> import calc
>>> calc.sum(5, 5)
```

```
10
```

In the same way, to use the above calc module in another Python script, use the import statement.

Every module, either built-in or custom made, is an object of a module class. Verify the type of different modules using the built-in type() function, as shown below.

Example: Module Type

```
>>> import math
>>> type(math)
<class 'module'>
>>> import calc
>>> type(calc)
<class 'module'>
```

Renaming the Imported Module

Use the as keyword to rename the imported module as shown below.

Example:

```
>>> import math as cal
>>> cal.log(4)
1.3862943611198906
```

from .. import statement

The above import statement will load all the resources of the module in the current working environment (also called namespace). It is possible to import **specific objects** from **a module** by using this syntax. For example, the following module calc.py has three functions in it.

calc.py

```
def sum(x,y):
    return x + y
def average(x, y):
    return (x + y)/2
def power(x, y):
    return x**y
```

Now, we can import one or more functions using the from...import statement. For example, the following code imports only two functions in the test.py.

Example: Importing Module's Functions

```
>>> from calc import sum, average
>>> sum(10, 20)
30
>>> average(10, 20)
15
>>> power(2, 4)
```

The following example imports only one function - sum.

Example: Importing Module's Function

```
>>> from functions import sum
>>> sum(10, 20)
30
>>> average(10, 20)
```

You can also import all of its functions using the **from...import *** syntax.

Example: Import Everything from Module

```
>>> from calc import *
>>> sum(10, 20)
30
>>> average(10, 20)
15
>>> power(2, 2)
4
```

Module Search Path

When the import statement is encountered either in an interactive session or in a script:

- **First, the Python interpreter tries to locate the module in the current working directory.**
- **If not found, directories in the PYTHONPATH environment variable are searched.**
- **If still not found, it searches the installation default directory.**

As the Python interpreter starts, it put all the above locations in a list returned by the sys.path attribute.

Example: Module Attributes

```
>>> import sys
>>> sys.path
['','C:\\python36\\Lib\\idlelib', 'C:\\python36\\python36.zip',
'C:\\python36\\DLLs', 'C:\\python36\\lib', 'C:\\python36',
'C:\\Users\\acer\\AppData\\Roaming\\Python\\Python36\\site-packages',
'C:\\python36\\lib\\site-packages']
```

If the required module is not present in any of the directories above, the message `ModuleNotFoundError` is thrown.

```
>>> import MyModule
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
    ModuleNotFoundError: No module named 'MyModule'
```

Reloading a Module

Suppose you have already imported a module and using it. However, the owner of the module added or modified some functionalities after you imported it. So, you can reload the module to get the latest module using the `reload()` function of the `imp` module, as shown below.

Example: Reloading Module

```
>>> import importlib
>>> importlib.reload(calc)
<module 'calc' from 'D:\\Python_programs\\calc.py'>
```

Getting Help on Modules

Use the help() function to know the methods and properties of a module. For example, call the `help("math")` to know about the math module. If you already imported a module, then provide its name, e.g. `help(math)`.

Getting Help on Module

As shown above, you can see the method names and descriptions. It will not display pages of help ending with --More--. Press Enter to see more help.

You can also use the dir() function to know the names and attributes of a module.

Know Module Attributes and Methods

Python Module Attributes: name, doc, file, dict

Python module has its attributes that describes it. Attributes perform some tasks or contain some information about the module. Some of the important attributes are explained below:

__name__ Attribute

The __name__ attribute returns the name of the module. By default, the name of the file (excluding the extension .py) is the value of __name__attribute.

Example: __name__ Attribute

```
>>> import math
>>> math.__name__
'math'
```

In the same way, it gives the name of your custom module.

Example: __name__ Attribute

```
>>> hello.__name__
'hello'
```

However, this can be modified by assigning different strings to this attribute. Change hello.py as shown below.

Example: Set __name__

```
def SayHello(name):
    print ("Hi {}! How are you?".format(name))
__name__="SayHello"
```

And check the __name__ attribute now.

```
>>> import hello
>>> hello.__name__
'SayHello'
```

The value of the __name__ attribute is __main__ on the Python interactive shell.

```
>>> __name__
'__main__'
```

When we run any Python script (i.e. a module), its __name__ attribute is also set to __main__. For example, create the following welcome.py in IDLE.

Example: welcome.py

```
print("__name__ = ", __name__)
```

Run the above welcome.py in IDLE by pressing F5. You will see the following result.

Output in IDLE:

```
>>> __name__ =  __main__
```

However, when this module is imported, its __name__ is set to its filename. Now, import the welcome module in the new file test.py with the following content.

Example: test.py

```
import welcome
print("__name__ = ", __name__)
```

Now run the test.py in IDLE by pressing F5. The __name__ attribute is now "welcome".

Example: test.py

```
 __name__ =  welcome
```

This attribute allows a Python script to be used as an executable or as a module.

Visit __main__ in Python for more information.

__doc__ Attribute

The __doc__ attribute denotes the documentation string (docstring) line written in a module code.

Example:

```
>>> import math
>>> math.__doc__
'This module is always available.  It provides access to the mathematical functions defined by the C standard.'
```

Consider the the following script is saved as test.py module.

test.py

```
 """This is docstring of test module"""
def SayHello(name):
```

```
    print ("Hi {}! How are you?".format(name))
    return
```

The __doc__ attribute will return a string defined at the beginning of the module code.

Example:

```
>>> import test
>>> test.__doc__
'This is docstring of test module'
```

__file__ Attribute

__file__ is an optional attribute which holds the name and path of the module file from which it is loaded.

Example: __file__ Attribute

```
 >>> import io
>>> io.__file__
'C:\\python37\\lib\\io.py'
```

__dict__ Attribute

The __dict__ attribute will return a dictionary object of module attributes, functions and other definitions and their respective values.

Example: __dict__ Attribute

```
 >>> import math
>>> math.__dict__
{'__name__': 'math', '__doc__': 'This module is always available.  It provides a
ccess to the\nmathematical functions defined by the C standard.', '__package__':
 '', '__loader__': <class '_frozen_importlib.BuiltinImporter'>, '__spec__': Modu
leSpec(name='math', loader=<class '_frozen_importlib.BuiltinImporter'>, origin='
built-in'), 'acos': <built-in function acos>, 'acosh': <built-in function acosh>
, 'asin': <built-in function asin>, 'asinh': <built-in function asinh>, 'atan':
<built-in function atan>, 'atan2': <built-in function atan2>, 'atanh': <built-in
 function atanh>, 'ceil': <built-in function ceil>, 'copysign': <built-in functi
on copysign>, 'cos': <built-in function cos>, 'cosh': <built-in function cosh>,
'degrees': <built-in function degrees>, 'erf': <built-in function erf>, 'erfc':
```

<built-in function erfc>, 'exp': <built-in function exp>, 'expm1': <built-in fun
ction expm1>, 'fabs': <built-in function fabs>, 'factorial': <built-in function
factorial>, 'floor': <built-in function floor>, 'fmod': <built-in function fmod>
, 'frexp': <built-in function frexp>, 'fsum': <built-in function fsum>, 'gamma':
<built-in function gamma>, 'gcd': <built-in function gcd>, 'hypot': <built-in f
unction hypot>, 'isclose': <built-in function isclose>, 'isfinite': <built-in fu
nction isfinite>, 'isinf': <built-in function isinf>, 'isnan': <built-in functio
n isnan>, 'ldexp': <built-in function ldexp>, 'lgamma': <built-in function lgamm
a>, 'log': <built-in function log>, 'log1p': <built-in function log1p>, 'log10':
<built-in function log10>, 'log2': <built-in function log2>, 'modf': <built-in
function modf>, 'pow': <built-in function pow>, 'radians': <built-in function ra
dians>, 'remainder': <built-in function remainder>, 'sin': <built-in function si
n>, 'sinh': <built-in function sinh>, 'sqrt': <built-in function sqrt>, 'tan': <
built-in function tan>, 'tanh': <built-in function tanh>, 'trunc': <built-in fun
ction trunc>, 'pi': 3.141592653589793, 'e': 2.718281828459045, 'tau': 6.28318530
7179586, 'inf': inf, 'nan': nan}

dir() is a built-in function that also returns the list of all attributes and functions in a module.

2.7. Python namespace packages

How to create python namespace packages in Python 3?

In Python, a namespace package allows you to spread Python code among several projects. This is useful when you want to release related libraries as separate downloads. For example, with the directories Package-1 and Package-2 in PYTHONPATH,

```
Package-1/namespace/__init__.py
Package-1/namespace/module1/__init__.py
Package-2/namespace/__init__.py
Package-2/namespace/module2/__init__.py
the end-user can import namespace.module1 and import namespace.module2.
```

On Python 3.3, you don't have to do anything, just don't put any __init__.py in your namespace package directories and it will just work. This is because Python 3.3 introduces implicit namespace packages.

On older versions, there's a standard module, called pkgutil, with which you can 'append' modules to a given namespace. You should put those two lines in both Packages.

```
1/namespace/__init__.py and Package-2/namespace/__init__.py:
from pkgutil import extend_path
__path__ = extend_path(__path__, __name__)
```

This will add to the package's __path__ all subdirectories of directories on sys.path named after the package. After this you can distribute the 2 packages separately.

Python - Math Module

Some of the most popular mathematical functions are defined in the math module. These include trigonometric functions, representation functions, logarithmic functions, angle conversion functions, etc. In addition, two mathematical constants are also defined in this module.

Pi is a well-known mathematical constant, which is defined as the ratio of the circumference to the diameter of a circle and its value is 3.141592653589793.

Example: Getting Pi Value

```
>>> import math
>>>math.pi
3.141592653589793
```

Another well-known mathematical constant defined in the math module is **e**. It is called **Euler's number** and it is a base of the natural logarithm. Its value is 2.718281828459045.

Example: e Value

```
>>> import math
>>> math.e
2.718281828459045
```

The math module contains functions for calculating various trigonometric ratios for a given angle. The functions (sin, cos, tan, etc.) need the angle in radians as an argument. We, on the other hand, are used to express the angle in degrees. The math module presents two angle conversion functions: degrees() and radians(), to convert the angle from degrees to radians and vice versa. For example, the following statements convert the angle of 30 degrees to radians and back (Note: π radians is equivalent to 180 degrees).

Example: Math Radians and Degrees

```
>>> import math
>>> math.radians(30)
0.5235987755982988
>>> math.degrees(math.pi/6)
29.999999999999996
```

The following statements show sin, cos and tan ratios for the angle of 30 degrees (0.5235987755982988 radians):

Example: sin, cos, tan Calculation

```
>>> import math
>>> math.sin(0.5235987755982988)
0.49999999999999994
>>> math.cos(0.5235987755982988)
0.8660254037844387
>>> math.tan(0.5235987755982988)
0.5773502691896257
```

You may recall that sin(30)=0.5, cos(30)=32 (which is 0.8660254037844387) and tan(30)= 13 (which is 0.5773502691896257).

math.log()

The math.log() method returns the natural logarithm of a given number. The natural logarithm is calculated to the base e.

Example: log

```
 >>> import math
>>>math.log(10)
2.302585092994046
```

math.log10()

The math.log10() method returns the base-10 logarithm of the given number. It is called the standard logarithm.

Example: log10

```
>>> import math
>>>math.log10(10)
1.0
```

math.exp()

The math.exp() method returns a float number after raising e to the power of the given number. In other words, exp(x) gives e**x.

Example: Exponent

```
>>> import math
>>>math.exp(10)
22026.465794806718
```

This can be verified by the exponent operator.

Example: Exponent Operator **

```
 >>> import math
>>>math.e**10
22026.465794806703
```

math.pow()

The math.pow() method receives two float arguments, raises the first to the second and returns the result. In other words, pow(4,4) is equivalent to 4**4.

Example: Power

```
>>> import math
>>> math.pow(2,4)
16.0
>>> 2**4
16
```

math.sqrt()

The math.sqrt() method returns the square root of a given number.

Example: Square Root

```
 >>> import math
>>> math.sqrt(100)
10.0
>>> math.sqrt(3)
1.7320508075688772
```

The following two functions are called representation functions. The **ceil()** function approximates the given number to the smallest integer, greater than or equal to the given floating point number. The floor() function returns the largest integer less than or equal to the given number.

Example: Ceil and Floor

```
>>> import math
>>> math.ceil(4.5867)
5
>>> math.floor(4.5687)
4
```

Python - Random Module

The random module is a built-in module to generate the pseudo-random variables. It can be used perform some action randomly such as to get a random number, selecting a random elements from a list, shuffle elements randomly, etc.

Generate Random Floats

The random.random() method returns a random float number between 0.0 to 1.0. The function doesn't need any arguments.

Example: random()

```
>>> import random
>>> random.random()
0.645173684807533
```

Generate Random Integers

The random.randint() method returns a random integer between the specified integers.

Example: randint()

```
>>> import random
>>> random.randint(1, 100)
95
>>> random.randint(1, 100)
49
```

Generate Random Numbers within Range

The random.randrange() method returns a randomly selected element from the range created by the start, stop and step arguments. The value of start is 0 by default. Similarly, the value of step is 1 by default.

Example:

```
>>> random.randrange(1, 10)
2
>>> random.randrange(1, 10, 2)
5
>>> random.randrange(0, 101, 10)
80
```

Select Random Elements

The random.choice() method returns a randomly selected element from a non-empty sequence. An empty sequence as argument raises an IndexError.

Example:

```
 >>> import random
>>> random.choice('computer')
't'
>>> random.choice([12,23,45,67,65,43])
45
>>> random.choice((12,23,45,67,65,43))
67
```

Shuffle Elements Randomly

The random.shuffle() method randomly reorders the elements in a list.

Example:

```
>>> numbers=[12,23,45,67,65,43]
>>> random.shuffle(numbers)
>>> numbers
[23, 12, 43, 65, 67, 45]
```

```
>>> random.shuffle(numbers)
>>> numbers
[23, 43, 65, 45, 12, 67]
```

Python Datetime

Python Dates

A date in Python is not a data type of its own, but we can import a module named datetime to work with dates as date objects.

Example

Import the datetime module and display the current date:

```
import datetime
x = datetime.datetime.now()
print(x)
```

Date Output

When we execute the code from the example above the result will be:

```
2021-06-20 18:59:34.897317
```

The date contains year, month, day, hour, minute, second, and microsecond.

The datetime module has many methods to return information about the date object.

Here are a few examples, you will learn more about them later in this chapter:

Example

Return the year and name of weekday:

```
import datetime
x = datetime.datetime.now()
print(x.year)
print(x.strftime("%A"))
```

Creating Date Objects

To create a date, we can use the datetime() class (constructor) of the datetime module.

The datetime() class requires three parameters to create a date: year, month, day.

Example

Create a date object:

```
import datetime

x = datetime.datetime(2020, 5, 17)
print(x)
```

The datetime() class also takes parameters for time and timezone (hour, minute, second, microsecond, tzone), but they are optional, and has a default value of 0, (None for timezone).

The strftime() Method

The datetime object has a method for formatting date objects into readable strings.

The method is called strftime(), and takes one parameter, format, to specify the format of the returned string:

Example

Display the name of the month:

```
import datetime
x = datetime.datetime(2018, 6, 1)
print(x.strftime("%B"))
```

A reference of all the legal format codes:

Directive	Description	Example
%a	Weekday, short version	Wed
%A	Weekday, full version	Wednesday
%w	Weekday as a number 0-6, 0 is Sunday	3
%d	Day of month 01-31	31
%b	Month name, short version	Dec

%B	Month name, full version	December
%m	Month as a number 01-12	12
%y	Year, short version, without century	18
%Y	Year, full version	2018
%H	Hour 00-23	17
%I	Hour 00-12	05
%p	AM/PM	PM
%M	Minute 00-59	41
%S	Second 00-59	08
%f	Microsecond 000000-999999	548513
%z	UTC offset	+0100
%Z	Timezone	CST
%j	Day number of year 001-366	365
%U	Week number of year, Sunday as the first day of week, 00-53	52
%W	Week number of year, Monday as the first day of week, 00-53	52
%c	Local version of date and time	Mon Dec 31 17:41:00 2018
%x	Local version of date	12/31/18

%X	Local version of time	17:41:00
%%	A % character	%
%G	ISO 8601 year	2018
%u	ISO 8601 weekday (1-7)	1
%V	ISO 8601 weeknumber (01-53)	01

```
import datetime
dt = datetime.datetime.now()
print(dt)
import pytz
dt = datetime.datetime.now(pytz.timezone('Asia/Kolkata'))
print(dt)
print(f'Weekday short version - {dt.strftime("%a")}')
print(f'Weekday full version - {dt.strftime("%A")}')
print(f'Weekday as a number - {dt.strftime("%w")}')
print(f'Day of Month - {dt.strftime("%d")}')
print(f'Month name short version - {dt.strftime("%b")}')
print(f'Month name full version - {dt.strftime("%B")}')
print(f'Month as a number - {dt.strftime("%m")}')
print(f'Year short version - {dt.strftime("%y")}')
print(f'Year full version - {dt.strftime("%Y")}')
print(f'Hour 00-23 - {dt.strftime("%H")}')
print(f'Hour 00-12 - {dt.strftime("%I")}')
print(f'AM/PM - {dt.strftime("%p")}')
print(f'Minute 00-59 - {dt.strftime("%M")}')
print(f'Second 00-59 - {dt.strftime("%S")}')
print(f'Microsecond 000000-999999 - {dt.strftime("%f")}')
print(f'UTC offset - {dt.strftime("%z")}')
```

```
print(f'Timezone - {dt.strftime("%Z")}')
print(f'Day number of year 001-366 - {dt.strftime("%j")}')
print(f'Week number of year - Sunday as the first day of week 00-53 - {dt.strftime("%U")}')
print(f'Week number of year - Monday as the first day of week 00-53 - {dt.strftime("%W")}')
print(f'Local version of date and time - {dt.strftime("%c")}')
print(f'Local version of date - {dt.strftime("%x")}')
print(f'Local version of time - {dt.strftime("%X")}')
print(f'ISO 8601 year - {dt.strftime("%G")}')
```

Output:

```
2021-07-27 14:40:30.558934+05:30
Weekday short version - Tue
Weekday full version - Tuesday
Weekday as a number - 2
Day of Month - 27
Month name short version - Jul
Month name full version - July
Month as a number - 07
Year short version - 21
Year full version - 2021
Hour 00-23 - 14 Hour 00-12 - 02
AM/PM - PM
Minute 00-59 - 40
Second 00-59 - 30
Microsecond 000000-999999 - 558934
UTC offset - +0530 Timezone - IST
Day number of year 001-366 - 208
Week number of year - Sunday as the first day of week 00-53 - 30
Week number of year - Monday as the first day of week 00-53 - 30
Local version of date and time - Tue Jul 27 14:40:30 2021
Local version of date - 07/27/21
Local version of time - 14:40:30
ISO 8601 year - 2021
```

Python - OS Module

It is possible to automatically perform many operating system tasks. The OS module in Python provides functions for creating and removing a directory (folder), fetching its contents, changing and identifying the current directory, etc.

You first need to import the os module to interact with the underlying operating system. So, import it using the import os statement before using its functions.

Getting Current Working Directory

The getcwd() function confirms returns the current working directory.

Example: Get Current Working Directory

```
>>> import os
>>> os.getcwd()
'C:\\Python37'
```

Creating a Directory

We can create a new directory using the os.mkdir() function, as shown below.

Example: Create a Physical Directory

```
>>> import os
>>> os.mkdir("C:\MyPythonProject")
```

A new directory corresponding to the path in the string argument of the function will be created. If you open the C:\ drive, then you will see the MyPythonProject folder has been created.

By default, if you don't specify the whole path in the mkdir() function, it will create the specified directory in the current working directory or drive. The following will create MyPythonProject in the C:\Python37 directory.

Example: Create a Physical Directory

```
>>> import os
>>> os.getcwd()
'C:\Python37'
>>> os.mkdir("MyPythonProject")
```

Changing the Current Working Directory

We must first change the current working directory to a newly created one before doing any operations in it. This is done using the chdir() function. The following change current working directory to C:\MyPythonProject.

Example: Change Working Directory

```
>>> import os
>>> os.chdir("C:\MyPythonProject") # changing current workign directory
>>> os.getcwd()
'C:\MyPythonProject'
```

You can change the current working directory to a drive. The following makes the C:\ drive as the current working directory.

Example: Change Directory to Drive

```
>>> os.chdir("C:\\")
>>> os.getcwd()
'C:\\'
```

In order to set the current directory to the parent directory use ".." as the argument in the chdir() function.

Example: Change CWD to Parent

```
 >>> os.chdir("C:\\MyPythonProject")
>>> os.getcwd()
'C:\\MyPythonProject'
>>> os.chdir("..")
>>> os.getcwd()
'C:\\'
```

Removing a Directory

The rmdir() function in the OS module removes the specified directory either with an absolute or relative path. Note that, for a directory to be removed, it should be empty.

Example: Remove Directory

```
>>> import os
>>> os.rmdir("C:\\MyPythonProject")
```

However, you can not remove the current working directory. To remove it, you must change the current working directory, as shown below.

Example: Remove Directory

```
>>> import os
>>> os.getcwd()
'C:\\MyPythonProject'
>>> os.rmdir("C:\\MyPythonProject")
PermissionError: [WinError 32] The process cannot access the file because it is being used by another process: 'd:\\MyPythonProject'
>>> os.chdir("..")
>>> os.rmdir("MyPythonProject")
```

Above, the `MyPythonProject` will not be removed because it is the current directory. We changed the current working directory to the parent directory using `os.chdir("..")` and then remove it using the `rmdir()` function.

List Files and Sub-directories

The `listdir()` function returns the list of all files and directories in the specified directory.

Example: List Directories

```
>>> import os
>>> os.listdir("c:\python37")
['DLLs', 'Doc', 'fantasy-1.py', 'fantasy.db', 'fantasy.py', 'frame.py',
'gridexample.py', 'include', 'Lib', 'libs', 'LICENSE.txt', 'listbox.py', 'NEWS.txt',
'place.py', 'players.db', 'python.exe', 'python3.dll', 'python36.dll', 'pythonw.exe',
'sclst.py', 'Scripts', 'tcl', 'test.py', 'Tools', 'tooltip.py', 'vcruntime140.dll',
'virat.jpg', 'virat.py']
```

If we don't specify any directory, then list of files and directories in the current working directory will be returned.

Example: List Directories of CWD

```
>>> import os
>>>os.listdir()
['.config', '.dotnet', 'python']
```

Python - sys Module

The sys module provides functions and variables used to manipulate different parts of the Python runtime environment. You will learn some of the important features of this module here.

sys.argv

sys.argv returns a list of command line arguments passed to a Python script. The item at index 0 in this list is always the name of the script. The rest of the arguments are stored at the subsequent indices.

Here is a Python script (test.py) consuming two arguments from the command line.

test.py

```
import sys
print("You entered: ",sys.argv[1], sys.argv[2], sys.argv[3])
```

This script is executed from command line as follows:

```
C:\python36> python test.py Python C# Java
You entered: Python C# Java
```

Above, sys.argv[1] contains the first argument 'Python', sys.argv[2] contains the second argument 'Python', and sys.argv[3] contains the third argument 'Java'. sys.argv[0] contains the script file name test.py.

sys.exit

This causes the script to exit back to either the Python console or the command prompt. This is generally used to safely exit from the program in case of generation of an exception.

sys.maxsize

Returns the largest integer a variable can take.

Example: sys.maxsize

```
>>> import sys
>>>sys.maxsize
9223372036854775807
```

sys.path

This is an environment variable that is a search path for all Python modules.

Example: sys.path

```
>>> import sys
>>>sys.path
['', 'C:\\python36\\Lib\\idlelib', 'C:\\python36\\python36.zip',
'C:\\python36\\DLLs', 'C:\\python36\\lib', 'C:\\python36',
'C:\\Users\\acer\\AppData\\Roaming\\Python\\Python36\\site-packages',
'C:\\python36\\lib\\site-packages']
```

sys.version

This attribute displays a string containing the version number of the current Python interpreter.

Example: sys.version

```
>>> import sys
>>>sys.version
'3.7.0 (v3.7.0:f59c0932b4, Mar 28 2018, 17:00:18) [MSC v.1900 64 bit
(AMD64)]'
```

Python - Statistics Module

The statistics module provides functions to mathematical statistics of numeric data. The following popular statistical functions are defined in this module.

Mean

The `mean()` method calculates the **arithmetic mean** of the numbers in a list.

Example:

```
>>> import statistics
>>> statistics.mean([2,5,6,9])
5.5
```

Median

The `median()` method returns the **middle value** of numeric data in a list.

Example:

```
>>> import statistics
>>> statistics.median([1,2,3,8,9])
3
>>> statistics.median([1,2,3,7,8,9])
5.0
```

Mode

The mode() method returns **the most common data point** in the list.

Example:

```
>>> import statistics
>>> statistics.mode([2,5,3,2,8,3,9,4,2,5,6])
2
```

Standard Deviation

The stdev() method calculates the standard deviation on a given sample in the form of a list.

Example:

```
>>> import statistics
>>> statistics.stdev([1,1.5,2,2.5,3,3.5,4,4.5,5])
```

Output - 1.3693063937629153