PMDS508L - Python Programming
Functions and Modules

Dr. B.S.R.V. Prasad
Department of Mathematics
School of Advanced Sciences
Vellore Institute of Technology
Vellore

srvprasad.bh@gmail.com (Personal)
srvprasad.bh@vit.ac.in (Official)
+91-8220417476

Functions

- A function is a block of code which can be run when it is called.
- Functions can be re-used and reduce the programming complexity.
- In other words, a function is a piece of code written to carry out a specified task.
- To carry out that specific task, the function might or might not need multiple inputs (arguments).
- When the task is carried out, the function can or can not return one or more values.

Python Functions

- There are three types of functions in Python:
  - **Built-in functions**, such as `help()` to ask for help, `min()` to get the minimum value, `print()` to print an object to the terminal,... You can find an overview with more of these functions here.
  - **User-Defined Functions (UDFs)**, which are functions that users create to help them out; And
  - **Anonymous functions**, which are also called lambda functions because they are not declared with the standard def keyword.

Parameters vs Arguments

- The **parameters** are the variables that we can define in the function declaration.
- The **arguments** are the variables given to the function for execution.
- In other words, **parameters** are the names used when defining a function or a method, and into which **arguments** will be mapped.
- **Arguments** are the things which are supplied to any function, while the function code refers to the arguments by their **parameters**.
- **Parameters** are local variables which are assigned values of the arguments when the function is called.
- **Parameters** are known as *Formal Parameters* and **arguments** are known as *Actual Parameters*.

User-Defined Functions

In Python a user-defined function is defined using the following four-steps:

1. Use the keyword `def` to declare the function and follow this up with the function name.
2. Add parameters to the function: they should be within the parentheses of the function. End your line with a colon.
3. Add statements that the functions should execute.
4. End your function with a return statement if the function should output something. Without the return statement, your function will return an object None.

User-Defined Functions


```python
def my_first_function():
    print("Hellooo.. This is my first function")

my_first_function()
```

Python Functions
The return Statement

- If we want to continue to work with the result of your function and try out some operations on it, we will need to use the return statement to actually return a value, such as a String, an integer, etc.
- If we are only printing a message then we don’t need to return any value.

```python
def hello():
  print("Hello World")
  return("hello")

def hello_noreturn():
  print("Hello World")

hello() * 2 # Multiply the output of `hello()` with 2

hello_noreturn() * 2 # (Try to) multiply the output of `hello_noreturn()` with 2
```

Python Funcitons
The return Statement

Functions immediately exit when they come across a return statement, even if it means that they won't return any value:

```python
def test_fun():
  for x in range(10):
      if x == 5:
         return
  print("Run!")

test_fun()
```

Python Funcitons
The return Statement

Another thing that is worth mentioning when we're working with the `return` statement, we can use it to return multiple values.

```python
# Define 'plus()'
def plus(a,b):
  sumab = a + b
  return (sumab, a)

# Call 'plus()'
sumab, a = plus(3,4)

print(sumab) # Print 'sumab'
print(sumab,' ',a) # Print 'sumab' and 'a'
print(sumab,' ',b) # Error
```

Python Functions
Function Arguments

We can pass arguments to functions and can evaluate them inside the function block.

```python
def addnums(num1,num2):
    print("The num1 is: ",num1)
    print("The num2 is: ",num2)
    print("The sum of the two numbers is: ",num1+num2)

addnums(2,3)
addnums(-5,6)
addnums(10.5, 2.6)
addnums(10.5,2)
```

Function Arguments

- By default, a function must be called with the correct number of arguments.
- If you try to call the function with lesser or greater arguments, the Python returns an error.

Python Functions
Variable No.of Arguments

- We can even pass the arbitrary number of arguments into a function.
- For this we add * before the parameter name in the function definition
- There are two types of variable arguments in Python.
  - `*args` for Non-Keyword Arguments
  - `**kwargs` for Keyword Arguments

Python Functions
Variable No.of Arguments

- The `*args` in Python function definitions is used to pass a non-key worded, variable number of arguments to a function.

Python Functions
Variable No.of Arguments

- The `*args` in Python function definitions is used to pass a non-key worded, variable number of arguments to a function.

```
def my_function(*args):
    l = len(args)
    print("The number of arguments passed are: ",l)
    for i in range(0,l):
        print('Argument ',i+1,'is: ',args[i])

my_function(2,6)
my_function("a")
my_function(2,"a",3.4)
my_function(2,"a",[3,5.7,"Test"])
```

Python Functions
Variable No.of Arguments

- The `**kwargs` in function definitions in python is used to pass a keyworded, variable-length argument list.
- We use the name `kwargs` with the double star.
- A keyword argument is where you provide a name to the variable as you pass it into the function.
- One can think of the `kwargs` as being a dictionary that maps each keyword to the value that we pass alongside it.
- As a result, we can iterate over the `kwargs` and perform the necessary operations in a function call.

Python Functions
Variable No.of Arguments


```python
def my_function(**kwargs):
    for key,value in kwargs.items():
        print('Key = ',key,'; Value = ',value)

my_function(First="One", Second=34.5, Third=True)
my_function(First=1)
my_function(Name="Prasad",Degree="PhD")
my_function(Key1=2,Key2="a",Key3=[3,5.7,"Test"])
```

Python Functions
Use of *args and **kwargs


```python
def my_function(*args, **kwargs):
    print("args: ", args)
    print("kwargs: ", kwargs)

my_function('one', 2, 3.14, first="Prasad", last="
   Bhuvanagiri", degree="PhD")
```

Python Functions
Default Parameter Values

We can use default parameter values for a function as follows:

```
def addnums(num1=0,num2=0):
    print("The num1 is: ",num1)
    print("The num2 is: ",num2)
    print("The sum of the two numbers is: ",num1+num2)

addnums(2,3)
addnums(2)
addnums(num2=3)
addnums(num1=5.6)
addnums()
```

Python Functions
List of Arguments

We can even pass a List as an argument to Python function

```
def  addnums(myNums):
     numsum = 0
     for i in myNums:
          numsum += i
     print("The sum of the numbers is: ",numsum)

addnums([1, 20, 43, 52])
```

Python Functions
Returning a Value

Functions can return a value to the user:

```
def addnums(myNums):
    numsum = 0
    for i in myNums:
        numsum += i
    return numsum

print("The sum of numbers is: ",addnums([1, 20, 43, 52]))
```

Scope of the Variables in Python Functions

- In general, variables that are defined inside a function body have a local scope, and those defined outside have a global scope.
- That means that local variables are defined within a function block and can only be accessed inside that function, while global variables can be obtained by all functions that might be in our script.

Global vs Local Variables


```python
# Global variable init
init = 1

# Define plus() function to accept a variable number of arguments
def plus(*args):
  # Local variable total
  total = 0
  print('The initial value is: ',init)
  for i in args:
    total += i
  return total

# Access the global variable
print("this is the initialized value " + str(init))

# (Try to) access the local variable
print("this is the sum " + str(total))
```

Recursive Functions

- Recursion is a mathematical and programming concept, in which a function calls itself.
- This has the benefit that one can loop through data to reach a result.
- Recursion is a very efficient and mathematically-elegant approach to programming
- Developer/Programmer should be very careful while designing a recursion.
- It can be quite easy to slip into writing a recursion which never terminates, or uses excess amounts of memory or processor power.

Recursive Functions


```
def my_recursion(num):
    if(num > 0):
        result = num + my_recursion(num-1)
        print(result)
    else:
        result = 0
    return result

print("\nMy Recursion Example Results:")
finalAns = my_recursion(6)
print("\nThe Final Answer is: ",finalAns)
```

Lambda Function

- We can write our very own Python functions using the def keyword, function headers, docstrings, and function bodies.
- However, there’s a quicker way to write functions on the fly, and these are called lambda functions because you use the keyword lambda.
- A lambda function is a small anonymous function.
- A lambda function can take any number of arguments, but can only have one expression.

Lambda Function

Syntax

```
fn = lambda arguments : expression
```

Lambda Function


```python
raise_to_power = lambda x, y: x ** y

raise_to_power(2, 3)
```

```python
myfunc = lambda a, b, c : a+b*c

print(myfunc(2,3,5))
```

map() function

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

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

```
function: (Required) The function to execute on each item.
iterables: (Required) A sequence, collection or an iterator object.
We can send as many iterables as we like. But need to make sure
that the function has one parameter for each iterable.
```

map() function

```python
def myFunc(a, b):
  return a + b

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

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

map() function


```
def myFunc(x):
  return x**2

X = map(myFunc, [1,5,-2,3])

print(X) #returns a map object
print(list(X)) #convert the map into a list, for
    readability
```

map() function with lambda function

We can pass lambda function to the map() without even naming them, and in this case, we refer to them as anonymous functions.

```
nums = [48, 6, 9, 21, 1]

square_all = map(lambda num: num ** 2, nums)

print(square_all) # Returns a map object

print(list(square_all)) # To see what the above map object returns, we use list to turn into a list
```

`filter()` function

The `filter()` function returns an iterator where the items are filtered through a function to test if the item is accepted or not.

```
filter(function, iterable)
```

```
function: (Required) The function to execute on each item.
iterables: (Required) A sequence, collection or an iterator
object.
```

filter() function


```python
ages = [5, 12, 17, 18, 24, 32]

def myFunc(x):
  if x < 18:
    return False
  else:
    return True

adults = filter(myFunc, ages)

for x in adults:
  print(x)
```

filter() function


```python
# function that filters vowels
def fun(variable):
    letters = ['a', 'e', 'i', 'o', 'u']
    if (variable in letters):
        return True
    else:
        return False

sequence = ['b', 'e', 'i', 'l', 'k', 's', 'p', 'a']
filtered = filter(fun, sequence) # using filter function

print('The filtered letters are:')
for s in filtered:
    print(s)
```

filter() function with lambda function


```python
my_list = [1,2,3,4,5,6,7,8,9,10]

# Use lambda function with `filter()`
filtered_list = list(filter(lambda x: (x*2 > 10),
    my_list))

print(filtered_list)
```

Functions with function arguments
Higher order functions

- Python functions are nothing but objects and names we define are simply identifiers bound to these objects.

```
def first(msg):
    print(msg)

first("Hello")

second = first
second("Hello")
```

- When we run the above code both functions `first` and `second` Return the same output as both refers to same object.

Functions with function arguments
Higher order functions

- We can even pass a function as argument to other function and these functions are called **higher order functions**

```python
def inc(x):
    return x + 1

def dec(x):
    return x - 1

def operate(func, x):
    result = func(x)
    return result

operate(inc, 3)
operate(dec, 3)
```

Nested Functions

- A function can return another function

```
def fun_called():
    def fun_returned():
        print("Hello!.. from inner function")
    return fun_returned

new = fun_called() # Create the fun_called object

new() # Call the function new()

#Output: Hello!.. from inner function
```

Nested Functions

- Nested functions can access variables of the enclosing scope.
- Following is an example of a nested function accessing a non-local variable (a variable which is read-only by default)

```
def outer_fn(msg): #Outer function
    def inner_fn():
        print("Printing from inner function with argument passed to outer function\n"+msg)
    inner_fn()

outer_fn("Hello!.. Argument passed to outer function")
```

Closure Functions

- Consider the following code, which is modification of above code replacing `inner_fn()` call with a `return inner_fn`

```
def outer_fn(msg): #Outer function
    def inner_fn():
        print("Printing from inner function with argument passed to outer function\n"+msg)
    return inner_fn

fun = outer_fn("Hello!.. Argument passed to outer function")
fun()
```

Closure Functions

- Here, we called the `outer_fn()` with string "Hello!.." and the returned function was bound to the name `fun`.
- On calling `fun()`, the message was still remembered although we had already finished executing `outer_fn()` function.
- This technique by which some data (in our case the msg) gets attached to the code is called **closure in Python**.
- This enclosing value is remembered even when the variable goes out of scope or the function itself is removed from the current namespace.

Closure Functions


```
del outer_fn

fun()

outer_fn("Hello!..")
```

Criteria for Closure Function Creation in Python

- The following criteria must be met to create closure in Python:
  - We must have a nested function (function inside a function)
  - The nested function must refer to a value defined in the enclosing (outer) function.
  - The enclosing function must return the nested function.

Use of Closures in Python

- Closures can avoid the use of global values and provide some form of data hiding.
- They can also provide an object oriented solution to the problem.
- When there are few methods (one method in most cases) to be implemented, closures can provide an alternative and more elegant solution.

Use of Closures in Python


```python
def multiplier_of(n):
    def multiplier(x):
        return x * n
    return multiplier

# Multiplier of 3
times3 = multiplier_of(3)

# Multiplier of 5
times5 = multiplier_of(5)

print(times3(9)) # Output: 27
print(times5(3)) # Output: 15
print(times5(times3(2))) # Output: 30
```

Python Decorators

- A decorator takes in a function, adds some functionality and returns it.
- To understand the decorators in Python, we must recall that:
  - Functions are objects of Python and various difference names can be bound to the same function object.
  - Functions can also take other functions as arguments and this type of functions are called higher order functions.
  - A nested function can also act as a closure function.

Python Decorators


```python
def make_pretty(func):
    def inner():
        print("I got decorated")
        func()
    return inner

def ordinary():
    print("I am ordinary")
```

Python Decorators


```
>>> ordinary()
I am ordinary

>>> pretty = make_pretty(ordinary)
>>> pretty()
I got decorated
I am ordinary
```

In this example, `make_pretty()` is a decorator.

The assignment step `pretty = make_pretty(ordinary)` decorates the function `ordinary()` and returns a function with name `pretty`.

This decorator function added some new functionality to the original function.

Python Decorators

Generally, we decorate a function and reassign it as

```
ordinary = make_pretty(ordinary)
```

This is a common construct for decorator functions. The above can be simplified as:

```
@make_pretty
def ordinary():
    print("I am ordinary")
```

which is equivalent to

```
def ordinary():
    print("I am ordinary")
ordinary = make_pretty(ordinary)
```

Python Decorators Illustration

Closure

```python
def deco(f):
    def g(*args, **kwargs):
        return f(*args, **kwargs)
    return g

def func(x):
    return 2*x

func = deco(func)
```

Decorated function

Decorator function

https://towardsdatascience.com/closures-and-decorators-in-python-2551abbc6eb6

```python
def deco(f):
    def g(*args, **kwargs):
        return f(*args, **kwargs)
    return g
def func(x):
    return 2*x
func = deco(func)
func(2)  # Output is 4
```

After the assigning the result, `func` refers to the closure `g`. So calling `func(a)` is like calling `g(a)`.

Python Decorators Illustration

Before decoration

func

```python
def func(x):
    return 2*x
```

After decoration

```python
func = deco(func)
```

func

```python
def deco(f):
    def g(*args, **kwargs):
        return f(*args, **kwargs)
    return g
```

```python
def func(x):
    return 2*x
```

Python Decorators Illustration

Before decoration

```
func  →  def func(x):
             return 2*x
```

After decoration

```
func = deco(func)
```

```
func  →  def deco(f):
             def g(*args, **kwargs):
                 return f(*args, **kwargs)
             return g
```

```
def func(x):
    return 2*x
```

- After decoration, the variable `func` refers to the closure `g`

Python Decorators Illustration

Before decoration

```
func  ───►  def func(x):
                return 2*x
```

After decoration `func = deco(func)`

```
            def deco(f):                      ──►  def func(x):
func  ───►      def g(*args, **kwargs):                return 2*x
                    return f(*args, **kwargs)
                return g
```

- After decoration, the variable `func` refers to the closure `g`
- Inside `g`, `f` refers to the `func(x)` definition.

Python Decorators Illustration

Before decoration

```
func ──→ def func(x):
             return 2*x
```

After decoration

```
func = deco(func)
```

```
         def deco(f):                          def func(x):
func ──→     def g(*args, **kwargs):               return 2*x
                 return f(*args, **kwargs)
             return g
```

- After decoration, the variable `func` refers to the closure `g`
- Inside `g`, `f` refers to the `func(x)` definition.

- In fact, `g` is now acting as an interface for the original function `func(x)` which was decorated.

Python Decorators Illustration

Before decoration

func →
```python
def func(x):
    return 2*x
```

After decoration
```python
func = deco(func)
```

func →
```python
def deco(f):
    def g(*args, **kwargs):
        return f(*args, **kwargs)
    return g
```
→
```python
def func(x):
    return 2*x
```

- After decoration, the variable `func` refers to the closure `g`
- Inside `g`, `f` refers to the `func(x)` definition.
- In fact, `g` is now acting as an interface for the original function `func(x)` which was decorated.
- We cannot directly call `func(x)` ourside `g`.

Python Decorators Illustration

Before decoration

```
func          def func(x):
                  return 2*x
```

After decoration   `func = deco(func)`

```
           def deco(f):                          def func(x):
func           def g(*args, **kwargs):               return 2*x
                   return f(*args, **kwargs)
               return g
```

- After decoration, the variable `func` refers to the closure `g`
- Inside `g`, `f` refers to the `func(x)` definition.
- In fact, `g` is now acting as an interface for the original function `func(x)` which was decorated.
- We cannot directly call `func(x)` ourside `g`.
- Instead, we first call `func` to call `g`, and then inside `g` we can call `f` to call the original function `func(x)`.

Python Decorators Illustration

Before decoration

func

```python
def func(x):
    return 2*x
```

After decoration

```python
func = deco(func)
```

func

```python
def deco(f):
    def g(*args, **kwargs):
        return f(*args, **kwargs)
    return g
```

```python
def func(x):
    return 2*x
```

- After decoration, the variable `func` refers to the closure `g`
- Inside `g`, `f` refers to the `func(x)` definition.

- In fact, `g` is now acting as an interface for the original function `func(x)` which was decorated.
- We cannot directly call `func(x)` ourside `g`.
- Instead, we first call `func` to call `g`, and then inside `g` we can call `f` to call the original function `func(x)`.
- So, we are calling the original function `func(x)` using the closure `g`.

Python Decorators Examples


```python
def smart_divide(func):
    def inner(a, b):
        print("I am dividing", a, "by", b)
        if b == 0:
            print("Whoops! Denominator is Zero. Cannot divide")
            return

        return func(a, b)
    return inner

@smart_divide
def divide(a, b):
    print(a/b)
```

Python Decorators
Examples


```
>>> divide(2,5)
I am dividing 2 by 5
0.4

>>> divide(2,0)
I am dividing 2 by 0
Whoops! Denominator is Zero. Cannot divide
```

Stacked Decorators


```
def deco1(f):
    def g1(*args, **kwargs):
        print("Calling ", f.__name__, "using deco1")
        return f(*args, **kwargs)
    return g1
def deco2(f):
    def g2(*args, **kwargs):
        print("Calling ", f.__name__, "using deco2")
        return f(*args, **kwargs)
    return g2
def func(x):
    return 2*x
func = deco2(deco1(func))
func(2)
```

Stacked Decorators


```python
def deco1(f):
    def g1(*args, **kwargs):
        print("Calling ", f.__name__, "using deco1")
        return f(*args, **kwargs)
    return g1
def deco2(f):
    def g2(*args, **kwargs):
        print("Calling ", f.__name__, "using deco2")
        return f(*args, **kwargs)
    return g2
@deco2
@deco1
def func(x):
    return 2*x
```

Stacked Decorators Example


```
def star(func):
    def inner(*args, **kwargs):
        print("*" * 30)
        func(*args, **kwargs)
        print("*" * 30)
    return inner

def percent(func):
    def inner(*args, **kwargs):
        print("%" * 30)
        func(*args, **kwargs)
        print("%" * 30)
    return inner
```

Stacked Decorators Example

```
@star
@percent
def printer(msg):
    print(msg)

printer("Hello")
```

```
def printer(msg):
    print(msg)
printer = star(percent(printer))

printer("Hello")
```

Output:

```
******************************
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Hello
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
******************************
```

Stacked Decorators Example

The order in which we chain decorators matter. If we had reversed the order as,

```python
@percent
@star
def printer(msg):
    print(msg)

printer("Hello")
```

the output would be:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
******************************
Hello
******************************
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Generator functions in Python

- A generator is a function that returns an (iterator) object which we can iterate over (one value at a time).
- Generator function in Python is defined in the same way as the functions are defined but with a `yield` statement instead of `return` statement.
- If a function contains at least one `yield` (it may contain other `yield` or `return` statements), it becomes a generator function.
- Both `yield` and `return` will return some value from a function.
- The difference is that while a `return` statement terminates a function entirely, `yield` statement pauses the function saving all its states and later continues from there on successive calls.

Difference between Generator function and Normal function

- Generator function contains one or more `yield` statements.
- When called, it returns an object (iterator) but does not start execution immediately.
- We can iterate through the items using `next()`.
- Once a function yields, the function is paused and the control is transferred to the caller.
- Local variables and their states are remembered between successive calls.
- Finally, when the function terminates `StopIteration` is raised automatically on further calls.

Generator functions in Python Example


```python
# A simple generator function
def my_gen():
    n = 1
    print('This is printed first')
    # Generator function contains yield statements
    yield n

    n += 1
    print('This is printed second')
    yield n

    n += 1
    print('This is printed at last')
    yield n
```

Generator functions in Python Example


```
>>> a = my_gen() # It returns an object but does not start execution immediately.

>>> next(a)
This is printed first
1

>>> next(a)
This is printed second
2

>>> next(a)
This is printed last
3

>>> next(a)
Traceback (most recent call last):
...
StopIteration
```

Generator functions in Python

- Observe that the value of variable *n* is remembered between the calls.
- The local variables are not destroyed when the function yields. This is exactly the opposite to how the normal functions works.
- Furthermore, the generator object can be iterated only once.
- To restart the process we need to create another generator object using some thing like `a = my_gen()`
- We can use generators directly with `for` loops.

Generator functions in Python


```python
for item in my_gen():
    print(item)
```

Generators with a Loop


```python
def rev_str(mystr):
    length = len(mystr)
    for i in range(length-1,-1,-1):
        yield mystr[i]

for ch in rev_str("Hello!"):
    print(ch)
```

Generators with a Loop


```python
def rev_str(mystr):
    length = len(mystr)
    for i in range(length-1,-1,-1):
        yield mystr[i]

for ch in rev_str("Hello!"):
    print(ch)
```

This generator function not only works with strings, but also with other kinds of iterables like list, tuple, etc

Python Generator Expression
Anonymous generator functions

- Simple generators can be easily created on the fly using generator expressions.
- Generator expression create anonymous generator functions. This is similar to lambda function, whcih create anonymous functions.
- The syntax of generator expression similar to that of **list comprehension in Python** but, the square brackets are replaced by round parentheses.

Python Generator Expression
Anonymous generator functions

- Simple generators can be easily created on the fly using generator expressions.
- Generator expression create anonymous generator functions. This is similar to lambda function, whcih create anonymous functions.
- The syntax of generator expression similar to that of **list comprehension in Python** but, the square brackets are replaced by round parentheses.
- The main difference between a list comprehension and a generator expression is that a list comprehension produces list while the generator expression produces one item at a time.
- Generator expressions have lazy execution (i.e., produce items only when needed/asked for)
- As a result, generator expression is much more memory efficient than an equivalent list comprehension.

Python Generator Expressions
Anonymous generator functions


```
my_list = [2,4,6,8,10]
listComprehension = [x**2 for x in my_list]
generator = (x**2 for x in my_list)

print(listComprehension)
print(generator)
```

Python Generator Expressions
Anonymous generator functions


```python
my_list = [2,4,6,8,10]
listComprehension = [x**2 for x in my_list]
generator = (x**2 for x in my_list)

print(listComprehension)
print(generator)
```

- We can use generator expression as function arguments
- While doing so, the round parentheses can be dropped.

```python
sum((x**2 for x in my_list))
max(x**2 for x in my_list)
```

User of Python Generators

1. **Easy of Implement**

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

gen_fn = PowTwoGen(5)

for i in gen_fn:
    print(i)
```

User of Python Generators

2. **Memory Efficient**
   - A normal function return a sequence and create the entire sequence in memory before returning the result.
   - If the sequence is very large then the above process drains our memory.
   - Generator implementation of such sequences is memory friendly as it produces one item at a time.

User of Python Generators

3. **Represent Infinite Stream of Data**
   - A normal function return a sequence which is finite data.
   - 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.
   - In theory, below code generates all even numbers

User of Python Generators

3. **Represent Infinite Stream of Data**
    - A normal function return a sequence which is finite data.
    - 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.
    - In theory, below code generates all even numbers

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

User of Python Generators

4. Pipelining Generators

- Multiple generators can be used to pipeline a series of operations.

```python
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))))
```

This example prints the sum of the squares of the first *n* Fibonacci numbers.

Python Modules

- A module is a file containing Python definitions and statements.
- A file containing Python code, for e.g.: `example.py`, is called a module and its module name would be `example`.
- 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.
- We can define our most used functions in a module and import it, instead of copying their definitions into different programs.

Python Modules

Let us create a module. Type the following and save it as `myCalc.py`.

```python
# Python Module example
def add(a, b):
    return a + b

def subtract(a, b):
    return a - b

def multiply(a, b):
    return a * b

def divide(a, b):
    return a / b
```

Using a Module

Now we can use the module we just created, by using the `import` statement:

```
import myCalc

myCalc.add(2,3)
myCalc.subtract(4,5)
myCalc.multiply(3,5)
myCalc.divide(3,7)
```

Importing a Module with Re-naming

We can import a module by renaming it as follows.

```python
import myCalc as mc

mc.add(2,3)
mc.subtract(4,5)
mc.multiply(3,5)
mc.divide(3,7)
```

Python from…import statement

We can import a specific variable or function from a Module using `from … import` …

```
from myCalc import add

add(2,3)
multiply(2,3) #Returns error
```

Note: When importing using the from keyword, do not use the module name when referring to elements in the module.

Python in-built Modules


```python
import math

print(math.pi)
print(math.e)
```

Python in-built Modules


```python
import math as m

print(m.pi)
print(m.sin(30)) #Prints the sin of 30 radians
```

Python in-built Modules


```python
from math import pi, e

print(pi)
print(e)

from math import *

print(sin(30))
```

The `dir()` function

- The `dir()` built-in function returns a sorted list of strings containing the names defined by a module.
- The list contains the names of all the modules, variables and functions that are defined in a module.

```
import math
dir(math)

import random
dir(random)

import statistics
dir(statistics)
```