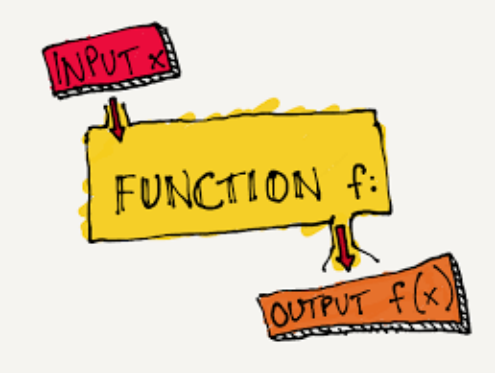
Jawaharlal Nehru Technological University Hyderabad

Kukatpally, Hyderabad - 500 085, Telangana, India



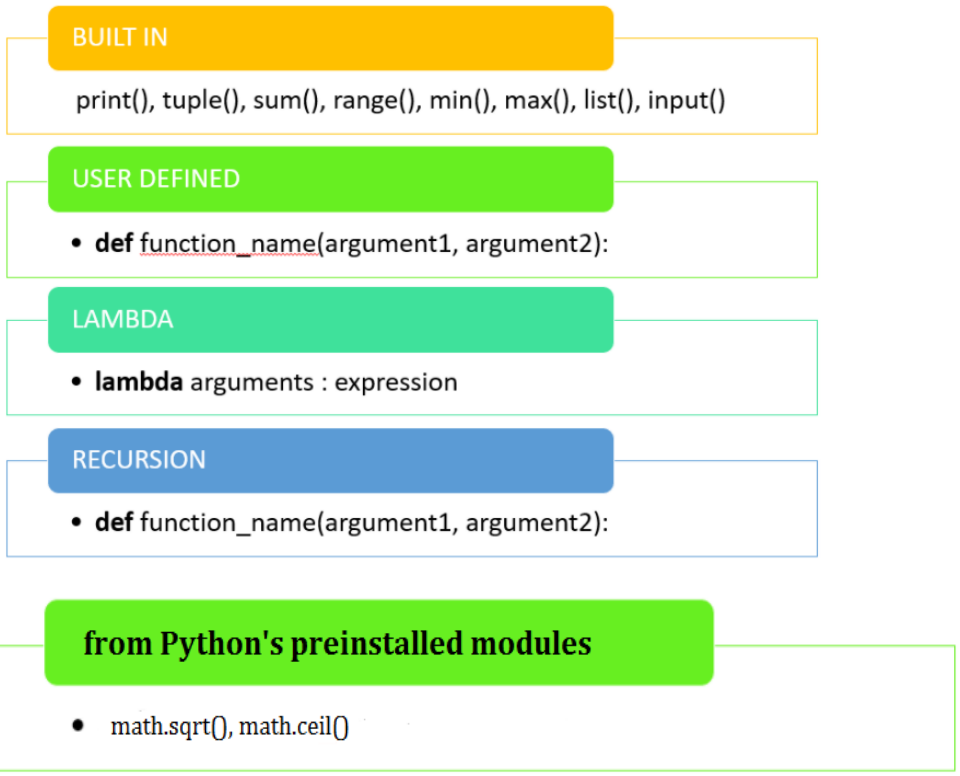
PYTHON PROGRAMMING

Functions Part 2

Session 7 , 16 May 2023

Dr N V Ganapathi Raju
Professor and HOD of IT
GRIET

Various kinds of Functions in Python



In general, Python supports 4 kinds of functions

- **Built-in functions**: which are an integral part of Python (print(), input()). Always available without any additional effort on behalf of the programmer.
- **User-defined functions** which are written by users for users - you can write your own functions and use them freely in your code.
- A **lambda function** is a small function containing a single expression. Helpful when we have to perform small tasks with less code.
- **from Python's preinstalled modules** - a lot of functions, very useful used significantly less often than built-in ones, are available in several modules installed together with Python.

Built-in functions

Built-in Functions

- The Python interpreter has a number of functions that are built into it and are always available.

Built-in Functions				
abs()	delattr()	hash()	memoryview()	set()
all()	dict()	help()	min()	setattr()
any()	dir()	hex()	next()	slice()
ascii()	divmod()	id()	object()	sorted()
bin()	enumerate()	input()	oct()	staticmethod()
bool()	eval()	int()	open()	str()
breakpoint()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	

https://docs.python.org/3/library/functions.html

Examples of built-in functions

```
print(abs(-3))
print(min(1, 2, 3, 4, 5))
print(max(4, 5, 6, 7, 8))
print(pow(3, 2))
print(len("Conduira Online"))
```

```
1 print(abs(-3))
2 print(min(1, 2, 3, 4, 5))
3 print(max(4, 5, 6, 7, 8))
4 print(pow(3, 2))
5 print(len("Conduira Online"))
```

```
3
1
8
9
15
```

Built-In functions

- **The Python core library has three methods called**
  - zip()
  - map()
  - filter()
  - sorted()
  - reduce()
  - enumerate()

enumerate()

- An enumerator built-in-function ***adds a counter of iterable numbers*** to the provided data structure of integers, characters or strings and many more.
- The data structure might be any list, tuple, dictionary or sets.
- If the counter is not provided by the user, then it starts from 0 by default.
- Based on the number provided the enumerator function iterates.
- **Syntax: enumerate(iterable, start)**
- The return type of an enumerate function is an ***object*** type.
- So the enumerate function returns an object by adding the iterating counter value to it. You can also convert the enumerator object into a list(), tuple(), set() and many more.

zip() built-in function

- zip() : function take iterables (can be zero or more), makes iterator that aggregates elements based on the iterables passed, and returns an iterator of tuples.

  zip(*iterables)

- The zip() function returns an iterator of tuples based on the iterable object.

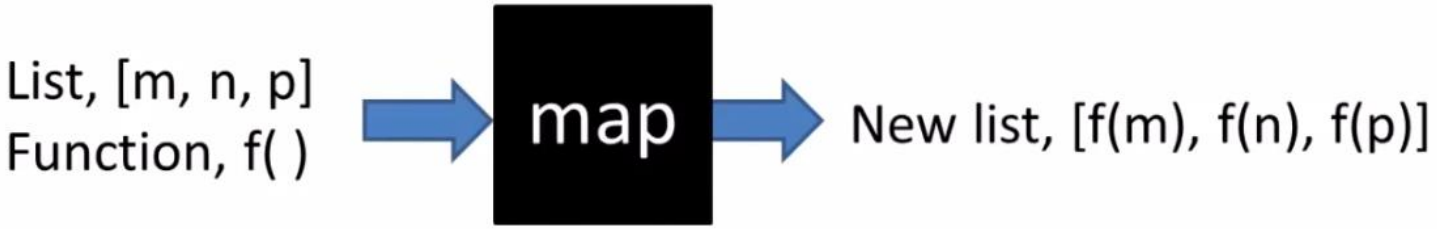
```
name = ["Akshay", "Dravid", "Sachin"]
roll_no = [10, 20, 30]
marks = [90, 88, 75]

mapped = zip(name, roll_no, marks)

print(list(mapped))
```

```
1  name = ["Akshay", "Dravid", "Sachin"]
2  roll_no = [10, 20, 30]
3  marks = [90, 88, 75]
4
5  mapped = zip(name, roll_no, marks)
6
7  print(list(mapped))
```

```
[('Akshay', 10, 90), ('Dravid', 20, 88), ('Sachin', 30, 75)]
```

map() built in function

- **map(fun, iter, ...)** function applies a given function to each element of an iterable.
- ***fun :*** *It is a function to which map passes each element of given iterable.* ***iter :*** *It is a iterable which is to be mapped.*
- The returned value from map() (map object) then can be passed to functions like list(), set().




```
nums = [1, 2, 3, 4, 5]
def sq(n):
    return n*n

square = list(map(sq, nums))
print(square)
```

```
1  nums = [1, 2, 3, 4, 5]
2
3  def sq(n):
4      return n*n
5
6  square = list(map(sq, nums))
7
8  print(square)
```

```
[1, 4, 9, 16, 25]
```

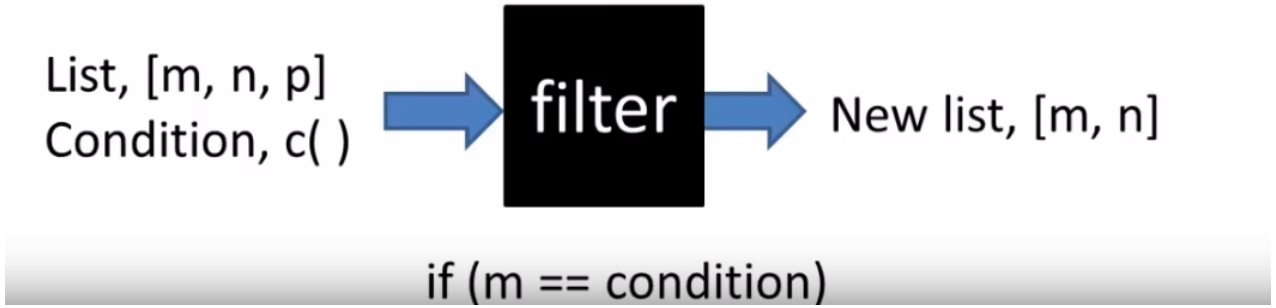
square of all elements in list

filter() built-in function

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

filter(fun, Iter)

- **fun:** function that tests if each element of a sequence true or not.
- **Iter:** Iterable which needs to be filtered.



filter() built-in function

- **Function to filter out vowels from list**

```
alphabets = ['a', 'b', 'd', 'e', 'i', 'j', 'o']

def filterVowels(alphabet):
    vowels = ['a', 'e', 'i', 'o', 'u']

    if(alphabet in vowels):
        return True
    else:
        return False

filteredVowels = filter(filterVowels, alphabets)

print('The filtered vowels are:')
for vowel in filteredVowels:
    print(vowel,end=" ")
```

```
1  alphabets = ['a', 'b', 'd', 'e', 'i', 'j', 'o']
2
3  def filterVowels(alphabet):
4      vowels = ['a', 'e', 'i', 'o', 'u']
5
6      if(alphabet in vowels):
7          return True
8      else:
9          return False
10
11 filteredVowels = filter(filterVowels, alphabets)
12
13 print('The filtered vowels are:')
14 for vowel in filteredVowels:
15     print(vowel,end=" ")
```

```
The filtered vowels are:
a e i o
```

filter()

- It takes a function and applies it to each item in the list to create a new list with only those items that cause the function to return True.

```
def checkAge(age):
    if age > 18:
        return True
    else:
        return False

lst = [10,14,18,22,24]
adults = filter(checkAge, lst)
print(list(adults))
```

```
age = [10,14,18,22,24]

adults = filter(lambda x: x > 18, age)

print(list(adults))
```

```
1  age = [10,14,18,22,24]
2  adults = filter(lambda x: x > 18, age)
3  print(list(adults))
```

```
[22, 24]
```

sorted()

names = ['Guido van Rossum', 'Bjarne Stroustrup' , 'James Gosling']

print(sorted(names, key= lambda name: name.split()[-1]))

```
1  names = ['Guido van Rossum', 'Bjarne Stroustrup' , 'James Gosling']
2
3  print(sorted(names, key= lambda name: name.split()[-1]))
```

```
['James Gosling', 'Guido van Rossum', 'Bjarne Stroustrup']
```

reduce()

- The reduce(fun,seq) function is used to apply a particular function passed in its argument to all of the list elements mentioned in the sequence.
- This function is defined in “functools” module.

```
from functools import reduce
reduce(lambda x,y: x+y, [1,2,3,4])
```

```
1  from functools import reduce
2  reduce(lambda x,y: x+y, [1,2,3,4])
```

```
10
```

max(), min()

studmarks = [('ABC', 35), ('CDE', 25), ('XYZ', 30),('PQR', 20),]

maxlst = max(studmarks, key=lambda student: student[1])

minlst = min(studmarks, key=lambda student: student[1])

print(maxlst)

print(minlst)

```
1  studmarks = [('ABC', 35), ('CDE', 25), ('XYZ', 30),('PQR', 20), ]
2  maxlst = max(studmarks, key=lambda student: student[1])
3  minlst = min(studmarks, key=lambda student: student[1])
4  print(maxlst)
5  print(minlst)
```

```
('ABC', 35)
('PQR', 20)
```

Lambda functions

lambda functions

- An anonymous function is a function that is defined **without a name**.

- While normal functions are defined using the def keyword in Python, *anonymous functions are defined using the **lambda** keyword.*

Syntax for lambda functions: **lambda arguments: expression**

  - Lambda functions can have any **number of arguments** but return only one **expression**. The **expression** is **evaluated** and **returned**.
  - Lambda functions are syntactically restricted to return a single expression
  - We can use lambda functions as an anonymous functions inside other functions
  - Lambda functions can be used **wherever function objects are required**.

Examples of lambda functions

```
int = lambda x: x * 2
print(int(5))     # 10

float = lambda x: x * 2
print(float(5.0))  # 10.0

add = lambda x, y: x + y
print (add (5,10))  # 15

x="Conduira Online"
(lambda x : print(x))(x)  # Conduira Online

Name = lambda first, second: first +' '+ second
Name('Conduira', 'Online')   # 'Conduira Online'
```

```
# Python lambda function, as Immediately
#    Invoked Function Expression (IIFE)

(lambda x, y: x + y)(2, 3)  # 5
```

higher-order functions

- **Lambda functions** are frequently used with **higher-order functions**, which take one or more functions as arguments or return one or more functions.
- A **lambda function** can be a higher-order function by taking a function (normal or lambda) as an argument.

```
high_ord_func = lambda x, func: x + func(x)

print(high_ord_func(2, lambda x: x + 3))   # 7
```

=> 2 + fun(2+3)
=> 2 + 5
=> 7

Arguments

- Like a normal function object defined with def, Python lambda expressions support all the different ways of passing arguments. This includes:
  - Positional arguments
  - Named arguments (keyword arguments)
  - Variable list of arguments (often referred to as var-args)
  - Variable list of keyword arguments
  - Keyword-only arguments

Arguments

```
print((lambda x, y, z: x + y + z)(1, 2, 3))          #  6

print((lambda x, y, z=3: x + y + z)(1, 2))           #  6

print((lambda x, y, z=3: x + y + z)(1, y=2))         #  6

print((lambda *args: sum(args))(1,2,3))              #  6

print((lambda **kwargs: sum(kwargs.values()))(one=1, two=2, three=3)) #  6
```

Limitations

- Since we can evaluate single expressions, features like

  iteration,

  conditionals,

  exception handling cannot be specified.

- But very useful in the place of one-line functions that evaluate single expressions.

Addition , multiplication and power operations

```
add = lambda a,b,c : a+b+c
print(add(5,3,2))

multiply = lambda x,y:x * y
print(multiply(3,7))

power = lambda m,n: m**n
print(power(6,2))
```

```
1  add = lambda a,b,c : a+b+c
2  print(add(5,3,2))
3
4  multiply = lambda x,y:x * y
5  print(multiply(3,7))
6
7  power = lambda m,n: m**n
8  print(power(6,2))
```

```
10
21
36
```

Preinstalled modules

Preinstalled modules

- Modules in Python are reusable libraries of code having *.py* extension, which implements a group of methods and statements. Python comes with many built-in modules as part of the standard library.
- To use a module in your program, import the module using *import* statement. All the *import* statements are placed at the beginning of the program.

  import module_name where import is a keyword
- Example : import math
- The ***math* module** is part of the Python standard library which provides access to various **mathematical** functions and is always available to the programmer
- The syntax for using a function defined in a module is, ***module_name.function_name()***

math module

```
import math
print(math.ceil(5.4))
print(math.sqrt(4))
print(math.pi)
print(math.cos(1))
print(math.factorial(6))
print(math.pow(2, 3))
```

```
1 import math
2 print(math.ceil(5.4))
3 print(math.sqrt(4))
4 print(math.pi)
5 print(math.cos(1))
6 print(math.factorial(6))
7 print(math.pow(2, 3))
```

```
6
2.0
3.141592653589793
0.5403023058681398
720
8.0
```

random module

- Another useful module in the Python standard library is the *random* module which generates random numbers.

```
import random
print(random.random())
print(random.randint(5,10))
```

```
1  import random
2  print(random.random())
3  print(random.randint(5,10))
```

```
0.8787676695752806
9
```

- *random()* function generates a random floating-point number between 0 and 1 and it produces a different value each time.
- *random randint(start, stop)* which generates a integer number between start and stop argument numbers (including both).

Creating our own Module

Packages

Packages in Python

- Suppose we have developed a very large application that includes many modules.
- As the number of modules grows, it becomes difficult to keep track of them.
- So we need to group them based on similar functionality and organize them.

Packages in Python

- Packages allow for a hierarchical structuring of the module namespace using dot notation.
- In the same way that modules help avoid collisions between global variable names, packages help avoid collisions between module names.
- To create a package, makes use of the operating system's inherent hierarchical file structure.
- Create a directory named pkg that contains two modules, mod1.py and mod2.py and a blank __init__.py
- Each package in Python is a directory which MUST contain a special file called __init__.py.

Creating and invoking a package

mainprg.py must be outside pkg

Steps

- Create a directory with name pkg
- Under pkg directory
  - Create a python program mod1.py
  - Create a python program mod2.py
  - Create a blank python program with __init__.py
- Create a mainprogram.py to invoke both packages
- Run mainprg.py in Idle terminal … executes modules.

mod1.py

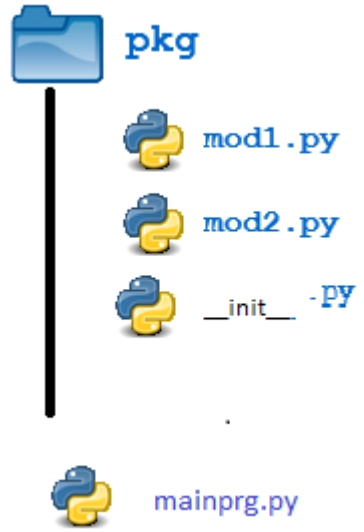
```
def show():
   print("in Show() of mod1")
```

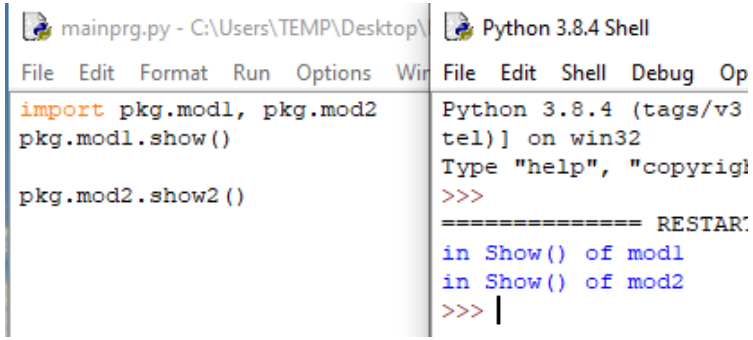
mod2.py

```
def show2():
   print("in Show() of mod2")
```

mainprg.py

```
import pkg.mod1, pkg.mod2
pkg.mod1.show()

pkg.mod2.show2()
```





Package Initialization

- If a file named __init__.py is present in a package directory, it is invoked when the package or a module in the package is imported.
- This can be used for execution of package initialization code, such as initialization of package-level data.

Recursive functions

Recursive Functions

- A **recursive function** is a function defined **in terms of itself via self-referential expressions**.
- The **function will continue to call itself and repeat its behavior until some condition is met to return a result.**
- All recursive functions share a common structure made up of two parts: **base case and recursive case.**

Examples using recursive functions in Python

```
def rec_cout(n):
    if n <= 0:
        print(“hello!')
    else:
        print(n)
        rec_cout(n-1)

rec_cout(3)
```

```
def print_n(s, n):
    if n <= 0:
        return
    print(s)
    print_n(s, n-1)
print_n(“hello",3)
```

Recursive function for factorial of a given number

```
def factorial_recursive(n):
    # Base case: 1! = 1
    if n == 1:
        return 1

    # Recursive case: n! = n * (n-1)!
    else:
        return n * factorial_recursive(n-1)
factorial_recursive(5)
```

```
1  def factorial_recursive(n):
2      # Base case: 1! = 1
3      if n == 1:
4          return 1
5
6      # Recursive case: n! = n * (n-1)!
7      else:
8          return n * factorial_recursive(n-1)
9  factorial_recursive(5)
```

```
120
```

Lambda functions

lambda functions

- An anonymous function is a function that is defined **without a name**.

- While normal functions are defined using the def keyword in Python, *anonymous functions are defined using the **lambda** keyword.*

Syntax for lambda functions: **lambda arguments: expression**

  - Lambda functions can have any **number of arguments** but return only one **expression**. The **expression** is **evaluated** and **returned**.
  - Lambda functions are syntactically restricted to return a single expression
  - We can use lambda functions as an anonymous functions inside other functions
  - Lambda functions can be used **wherever function objects are required**.

Examples of lambda functions

```
v1 = lambda x : x * 2
print(v1(5))

v2 = lambda x: x * 2
print(v2(5.0))

v3 = lambda x, y: x + y
print (v3 (5,10))

x="lambda functions"
(lambda x : print(x))(x)

Name = lambda first, second: first +' '+ second
Name('Lambda', 'Functions')
```

```
10
10.0
15
lambda functions
'Lambda Functions'
```

Limitations

- Since we can evaluate single expressions, features like

  iteration,

  conditionals,

  exception handling cannot be specified.

- But very useful in the place of one-line functions that evaluate single expressions.

Addition , multiplication and power operations

```
add = lambda a,b,c : a+b+c
print(add(5,3,2))

multiply = lambda x,y:x * y
print(multiply(3,7))

power = lambda m,n: m**n
print(power(6,2))
```

```
1  add = lambda a,b,c : a+b+c
2  print(add(5,3,2))
3
4  multiply = lambda x,y:x * y
5  print(multiply(3,7))
6
7  power = lambda m,n: m**n
8  print(power(6,2))
```

```
10
21
36
```