Python Programming

Dr. Cao Tien Dung

School of Engineering - Tan Tao University

Vicohub

https://vicohub.com

Lesson 4 - Outline

□ Function

□ Anonymous function

□ Built-in function: map, filter, reduce

□ Generator

□ Recursive function

Function

- Function is a set of instructions that you want to use repeatedly or a sub program and called when needed.
- There are three types of functions in Python
  - Built-in functions: **`min(), help(), print(),...`**
  - User-Defined Functions: defined by yourself
  - Anonymous functions (called lambda functions) they are not declared with the standard way (i.e., **`def`** keyword).

Define a function

```
def function_name (parameters):
    statement
    return values   #optional
```

- Parameters and return statement are optional
  - Without the return statement, your function will return an object **None.** Usually use to print out something.
- Examples

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

def hi_someone(name):
  print(“Hi ”, name)
```

```
def plus(a,b):
  s = a + b
  return s
```

Define a Function

- Return multiple values
  - Values are constructed with a tuple

```
def plus_sub(a,b):
  sum = a + b
  sub = a-b
  #return multiple values
  return sum, sub
```

```
r = plus_sub(5,2) #call function
print(r)
#output:  (7,3)

sum, sub = plus_sub(5,2) #call function
print(sum, ‘ -- ’, sub)
```

Define a function

- functions immediately exit when they come across a **`return`** statement, even if it means that they won’t return any value.

```
def run():
  for x in range(10):
     if x > 2:
       return
     print("Run!")

run()
```

- Four type of arguments
  - Default arguments
  - Required arguments
  - Keyword arguments
  - Variable number of arguments

Type of arguments

```
def plus(a=1,b=2):
    return a + b

plus() #default arguments
plus(2, 4) # still OK
```

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

plus(a=2,b=4)#keyword arguments
```

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

plus(2,4)#required arguments
```

```
#function to accept a variable number of arguments
def plus(*args):
  total = 0
  for i in args:
    total += i
  return total

plus(20,30,40,50, 60)
```

Global vs Local Variables

- Variables that are defined inside a function can only be accessed inside that function (i.e., local scope), and those defined outside can be accessed by all functions that might be in your script (i.e., global scope).

```
gvar = -10 # global variable
def plus(*arg):
    total = 0 #local variable
    for x in arg:
        total += x
    return total + gvar #use global variable is OK

print(total) #error (total is outside its scope)
plus(1,2,3,4) #output: 0
```

Anonymous Functions

- A function is defined use **`lambda`** keyword instead of use **`def`** keyword.
  - Function with the arguments, and an expression or instruction that gets evaluated and returned, but without function name.

```
double = lambda x: x*2

double(5)
```

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

double(5)
```

- You use anonymous functions when you require a nameless function for a short period of time and that is created at runtime.

map() function

- **map()** applies the function **myfunc** to all the elements of the sequence seq.

```
map(myfunc, seq)
```

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

t = (1,2,3,4,5)
double = map(myfunc, t) # applying myfunc to each
element of tuple t
#use with lambda: double = map(lambda x: x*2, t)
for x in double:
    print(x)
```

map() function: more...

```
a = [1,2,3,4,5]
b = [10,9,8,7,6]
#create a list which is sum of elements in a and b
list(map(lambda x,y: x+y, a,b))
```

- □ Mapping a list of function

```
#define a function to map a list of function to a value
from math import sqrt,exp, log #import library
def map_myfunc(x, funcs):
    return [f(x) for f in funcs]
#apply list of functions to number 2
list(map_myfunc(2,[sqrt,exp,log]))
#output
[1.4142135623730951, 7.38905609893065, 0.6931471805599453]
```

filter() function

- offers an elegant way to filter out all the elements of a sequence "sequence", for which the function **`testfunc`** returns **`True.`**
  - Usually **`testfunc`** is used **`lambda`** function

```
filter(testfunc, seq)
```

```
fib = [0,1,1,2,3,5,8,13,21,34,55]
odd_numbers = list(filter(lambda x: x % 2, fib))
```

```
from random import randint
#random 10 number on interval (-10,10)
li = [randint(-10,10) for i in range(10)]
list(filter(lambda x: x % 2 and x > 0, li))
```

reduce() function

- it applies a rolling computation to sequential pairs of values in a sequence.
  - It is a function in **functools** module

```
from functools import reduce
reduce(myfunc, seq)
#myfunc requires 2 parameter and return 1 value
```

```
from functools import reduce
from random import randint
fib = [randint(-10,10) for i in range(10)]
#reduce, sum of all elements
reduce(lambda x, y: x+y, fib)
```

Generator

- Generators are iterators, but you can only iterate over them once (value is not stored in memory). Generators are implemented as function but do not **return** a value, they **yield** it.

```
def mygenerator():
    for i in range(10):
        yield I

gen = mygenerator()
print(next(gen))
print(next(gen))
#use for loop
for x in gen:
    print(x)
```

```
#output
0
1

#output of for loop
2
3
…
```

Recursive function

- In Python, a function can call other functions. It í possible for function to call itself. There type of construct are termed as recursive function.
- Example: factorial of a number

```
6! = 1 * 2 * 3 * 4 * 5 * 6
    = (1 * 2 * 3 * 4 * 5) * 6 = 5! * 6 = (6-1)! * 6
    = ((1 * 2 * 3 * 4) * 5) * 6 = ((4!) * 5) * 6
    = ….
    = 1! * 2 * 3 * 4 * 5 * 6
    = 1 * 2 * 3 * 4 * 5 * 6
```

Recursive function

```
def factorial(n):
    # 1! = 1 (base case)
    if n ==1:
        return n
    # recursive case n! = (n-1)! * n
    else:
        return factorial(n-1) * n

#call function
print(factorial(6)) ➔ 720
```

- Multiple stop condition cases and/or multiple recursive expressions are also possible