#9: Functions

Reading: Chapter 6
Contents

Functions
• Pass by reference versus pass by value
• Anonymous functions (lambda)
• Map, filter, reduce
• Variable-length arguments
• Command Line Arguments
Pass by object reference

- **Passing** input parameters (arguments) **by object reference** means that if you change what a parameter refers to within a function, **the change** also **reflects back** in the calling function.

Mutable objects (like lists) **can** be passed by reference.

```python
def changemeRef(mylist):
    mylist.append([1,2,3,4])
    print("Values inside the function: ", mylist)
    return

# Now you can call changemeRef function
mylist = [10,20,30]
changemeRef( mylist )
print("Values outside the function: ", mylist)
```

Values inside the function: [10, 20, 30, [1, 2, 3, 4]]
Values outside the function: [10, 20, 30, [1, 2, 3, 4]]
Pass by value

Objects like int, string, tuple (immutables) can be passed by value, which means that if you change what a parameter refers to within a function, the change does not reflect back in the calling function.

Mutable objects (like lists) can be passed by value.

def changemeVal(mylist):
    mylist = [1,2,3,4]  # This would assign a new reference
    print("Values inside the function: ", mylist)
    return

# Now you can call changeme function
mylist = [10,20,30]
changemeVal(mylist)
print("Values outside the function: ", mylist)

Values inside the function: [1, 2, 3, 4]
Values outside the function: [10, 20, 30]
Pass by value vs. reference

def ref_demo(x):
    print("x=",x," id=",id(x))  # x = 9, id= 41902552
    x=42  # x = 42
    print("x=",x," id=",id(x))  # x = 42, id= 41903752

# main
ref_demo(9)

id(obj) returns the "identity" of the object "obj", which is a unique and constant integer for this object during its lifetime. Two different objects with non-overlapping lifetimes may have the same id() value.
Anonymous functions (lambda)

These functions are called *anonymous* because they are functions without a name and they are just needed where they have been created.

Characteristics:

- Lambda forms can **take any number of arguments** but **return just one value** in the form of an expression. They cannot contain commands or multiple expressions.
- An anonymous function *cannot* be a direct call to `print` because lambda requires an expression.
- Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.
Anonymous functions (lambda)

Syntax:

\[
\text{lambda } [\text{arg1 [,arg2,.....argn]}] : \text{expression}
\]

```
sum = lambda arg1, arg2 : arg1 + arg2
# Now you can call sum as a function
print("Value of total : ", sum( 10, 20 ))
print("Value of total : ", sum( 20, 20 ))
```

Value of total : 30
Value of total : 40

It is used primarily to write very short functions that are a hassle to define in the normal way.
Lambda

Lambda is often used as an argument to other functions that expects a function object, such as sorted()'s 'key' argument.

```
listnested = [[3, 4], [3, 5], [1, 2], [7, 3]]
print(sorted(listnested, key=lambda x: x[1]))
[[1, 2], [7, 3], [3, 4], [3, 5]]
print(sorted(listnested, key=lambda x: x[0]))
[[1, 2], [3, 4], [3, 5], [7, 3]]
```
filter

`filter(function, sequence)` offers an elegant way to filter out all the elements of a sequence, for which the function `function` returns True. Only if `function` returns True will the element of the sequence be included in the result list.

It returns an iterator of a list in Python3, and a list in Python2.

```python
fib = [0,1,1,2,3,5,8,13,21,34,55]
result = list(filter(lambda x: x % 2, fib))
print(result)
[1, 1, 3, 5, 13, 21, 55]
print(list(filter(lambda x: x % 2 == 0, [1,2,3,4,5])))
[2, 4]
```
map

map(function, list)
calls for every item in the list a function function

It returns an iterator of a list in Python3. It returns a list in Python2.

Ex.

Write the code that doubles every integer from the list.
filter

# in Python 3
print(list(map(lambda x: x*2, [1,2,3,4,5])))
[2, 4, 6, 8, 10]

# in Python 2
print(map(lambda x: x*2, [1,2,3,4,5]))
[2, 4, 6, 8, 10]
reduce

reduce(function, list)
continually applies the function function to the list. It returns a single value.

from functools import reduce  # in Python3
reduce(lambda x,y: x+y, [47,11,42,13])
113
reduce

Ex.2: Determining the maximum of a list of numerical values by using reduce:

\[ f = \text{lambda } a, b: a \text{ if } (a > b) \text{ else } b \]

\[ \text{reduce}(f, [47, 11, 42, 102, 13]) \]

102

Ex.3: Calculating the sum of the numbers from 1 to 100:

\[ \text{reduce}(\text{lambda } x, y: x+y, \text{range}(1, 101)) \]

5050
Variable-length arguments

• You may need to process a function for more arguments than you specified while defining the function. These arguments are called variable-length arguments.

```python
def functionname([[formal_args,] *var_args_tuple ]):
    function_statements
    return [expression]
```

The tuple remains empty if no additional arguments are specified during the function call.

* used in Python to define a variable number of arguments, any actual parameters after the `formal_args` parameters will be packed into a tuple and bound to "remaining."
Variable-length arguments

def printinfo(arg1, *vartuple):
    print("Output is: ")
    print(arg1)
    for var in vartuple:
        print(‘>’, var)
    return

# Now you can call printinfo function
printinfo(10)
Output is:
10
printinfo(70, 60, 50)
Output is:
70  >60  >50
Variable-length arguments

Sometimes, it's necessary to use positional parameters followed by an arbitrary number of parameters in a function definition. This is possible, but the positional parameters always have to precede the arbitrary parameters.

```python
def locations(city, *other_cities):
    print(city, other_cities)

# main
locations("Paris")
Paris ()
locations("Paris", "Strasbourg", "Lyon", "Dijon")
Paris ('Strasbourg', 'Lyon', 'Dijon')
```
Variable-length arguments

** is for arbitrary number of keyword parameters

def f(**args):
    print(args)

#main
f()
f()
{}
f(de="Germnan",en="English",fr="French")
{'fr': 'French', 'de': 'Germnan', 'en': 'English'}
Command Line Arguments

It's possible to give the parameters as command line arguments to Python script.

If you call a Python script from a shell, the arguments (separated by spaces) are placed after the script name.

Inside of the script these arguments are accessible through the list variable `sys.argv`. The name of the script is included in this list `sys.argv[0]`. `sys.argv[1]` contains the first parameter, `sys.argv[2]` the second and so on.
Command Line Arguments

# Module sys has to be imported:
import sys

# Iteration over all arguments:
for eachArg in sys.argv:
    print(eachArg)

Example call to this script in shell:
python3 argumente.py python course for beginners
argumente.py
python
course
for
beginners
Example Program: Averaging a List

def getNums():  # creates a list from user input
    nums = []
    while 1:
        xStr = input("Enter a number: ")
        if xStr == 'q':
            break
        nums.append(eval(xStr))
    return nums

def average(lst):
    sum = 0.0
    for num in lst:
        sum += num
    return sum / len(lst)

#main
data = getNums()
print("Average =", average(data))
def reverse(list, num):
    size = len(list)
i=0
    while i<num:
        temp = list[size-1-i]
        list[size-1-i]=list[i]
        list[i]=temp
        i=i+1
    num=num+1

# main
mylist=[1,2,3,4]
num=3
reverse(mylist, num)
print(mylist)
print(num)
Next lecture ...

• Midterm