4. Function and File

Prof. Vishal Polara

BVM Engineering College

vishal.polara@bvmengineering.com

Outline

1. Need of Function

- Function is defined as of program into smaller part.
- It is difficult to test and debug a large program. So large program is divided into smaller part known as function.
- It makes programmers task easy when error occurs.
- It is used to reduce redundancy of code.
- It is used convert large program into smaller one.
- It makes program easy to understand once the task is over.
- It also provides efficiency in form of outcomes.

Characteristics of function

- It Can be assigned to a variable
- It Can be passed as a parameter.
- It Can be returned from a function.
- Functions are treated like any other variable in Python.
- **d**ef keyword is used to create function.
- Functions are objects.
- The same reference rules hold for them as for other objects.
- Parameters can be assigned default values. They are overridden if a parameter is given for them.
- The type of the default doesn't limit the type of a parameter.
- It is Call by name.
- Any positional arguments must come before named ones in a call.

2. Types of Function

- Function with no argument and no return value.
- Function with argument and no return value.
- Function with no argument and return value.
- Function with argument and return value.

Example-1

- Code:

```
def Add(a,b) :
    c=a+b
    print (c)
def bar(x) :
    return x * x
Add(5,6)
C=bar(6)
Print (c)
```

- Output:

```
»>
11
36
```

Example-2

- Code:

```
def add (x) :
    def ad(y) :
        return x + y
    return ad
f = add(3)
print f
print f(2)
```

- Output:

```
»>
5
```

Example-3

- Code:

```
def foo(x = 3) :
    print x
foo()
foo(10)
foo('hello')
```

- Output:

```
»>
3
10
hello
```

Function with default arguments

- Example:

```
def foo (a,b,c) :
    print a, b, c
foo(c = 10, a = 2, b = 14)
foo(3, c = 2, b = 19)
```

- Output:

```
»>
2 14 10
3 19 2
```

3. Function Inside Function

- Since they are like any other object, you can have functions inside functions.

```
def Add(x,y) :
    def ad(z) :
        return z * 2
    return ad(x) + y
Add(2,3)
```

- Output:

```
»>
7
```

4. Function with Variable Arguments

```
def print_two(*argv):
    arg1, arg2 = argv
print ("arg1: %r, arg2: %r" % (arg1, arg2))
def print_one(arg1):
    print ("arg1: %r" % arg1)
print_two("Zed","Shaw")
print_one("First!")
```

Scope of Variable

- There are two types of variable.
- **Local Variable:**
- Scope is limited to function.
- Same name can be used in multiple. function.
- By default it is local.
- **Global Variable:**
- Scope is limited to program.
- Global keyword is used to create global variable.

5. Anonymous function(Lambda function)

- It is a function without name.
- Normal function can be defined using def keyword.
- Anonymous function are defined by lambda, hence it is called lambda function.
- It is a one line version of function but It is not like inline function.
- It is useful when requires nameless function for short period of time.
- syntax:
  a= lambda argument: expression
  a= lambda x:x*2

Example

- »> a=lambda x,y=10:x+y
  »> a(10)
  20
  »> a(20)
  30
  »> b=lambda *z:z
  »> b(23,'zyx')
  (23, 'zyx')
  »> b(42)
  (42,)
- »> factorial = lambda i:1 if i==0 else i*factorial(i-1)
  »> print(factorial(4))
  »> 4 * 3 * 2 * 1 = 12 * 2 = 24

Example

- »> a=[2, 3, 45, 6, 37]
- »> [i for i in filter(lambda x:x>4,arr)]
- »> a=(1,2,3,4)
- »> b=map(lambda x:x*2,a)
- »> list(b) [2, 4, 6, 8]

6. Modules

- It is the highest level structure of Python.
- Any python source file is module.
- It is used to reduce rewriting of same code.
- Each file with the py suffix is a module.
- Each module has its own namespace.
- It can be access by using **import** keyword.
- When a module is imported, all of the statements in the module execute one after another until the end of the file is reached
- import always executes the entire file.
- Modules are still isolated environments.

Way of importing module

import mymodule	Brings all elements of mymodule in, but must refer to as mymodule.<elem>
from mymodule import x	Imports x from mymodule right into this namespace
from mymodule import *	Imports all elements of mymodule into this namespace

From math import sin, cos

- Allows parts of a module to be used without having to type the module prefix.
  Example:

```
from math import sin, cos
def rectangular(r, theta):
    x = r * cos(theta)
    y = r * sin(theta)
    return x, y
```

Example of import *

- Takes all the symbols from a module and places them into local scope.
- Usually considered bad style (try to avoid).
  Example:

```
from math import *
def rectangular(r, theta):
    x = r * cos(theta)
    y = r * sin(theta)
    return x, y
```

Module Names

- File names have to follow the rules: ex. Good.py , # invalid 2bad.py
- Must be a valid identifier name.
- Also: avoid non-ASCII characters.
- It is standard practice for package and module names to be concise and lowercase. ex. foo.py not MyFooModule.py
- Use a leading underscore for modules that are meant to be private or internal .
  _foo.py
- Don't use names that match common standard library modules (confusing) projectname/ math.py

7. Packages

- It is nothing but the collection of modules.
- For larger collections of code, it is usually desirable to organize modules into a hierarchy.
- To do it, you just add __init__.py files

```
spam/
    __init__.py
    foo.py
    bar/
        __init__.py
        grok.py ...
```

- Import works the same way, multiple levels

```
import spam.foo
from spam.bar import grok
```

- The __init__.py files import at each level

way of importing module

- One module import
  from spam import Foo, Bar, Grok
- Importing dozens of submodules
  from spam.foo import Foo
  from spam.bar import Bar
  from spam.grok import Grok

Modules Vs Packages

- Modules are easy–a single file.
- Packages are hard–multiple related files
- Some Issues:
  - Code organization
  - Connections between submodules
  - Desired usage

Absolute Vs Explicit relative

- **Absolut import** spam/ __init__.py
  foo.py
  bar.py write: from spam import foo
- **Relative imports**
  write: from . import foo
  Leading dots (.) used to move up hierarchy
  from . import foo # Loads ./foo.py
  from .. import foo # Loads ../foo.py
  from ..grok import foo # Loads ../grok/foo.py

Continue..

- Explicit relative import allows packages to renamed. Like it is possible to change name from spam to grok

  spam/

  __init__.py

  foo.py

  bar.py

- Absolute imports are recommended, as they are usually more readable better behaved.

__init__ file contains

- # __init__.py
  from .foo import Foo
  from .bar import Bar

8. File

- File is a collection of related information.
- Data stored in the program are temporary; they are lost when the program terminates. To permanently store the data created in a program, you need to save them in a file on a disk or other permanent storage.
- The file can be transported and can be read later by other programs. There are two types of files text and binary. Text files are essentially strings on disk. file object = open(file_name [, access_mode][, buffering])
- buffering If the buffering value is set to 0, no buffering takes place. If the buffering value is 1, line buffering is performed while accessing a file.

File input method

inflobj = open('data', 'r')	Open the file 'data' for input
S = inflobj.read()	Read whole file into one String
S = inflobj.read(N)	Reads N bytes (N >= 1)
L = inflobj.readlines()	Returns a list of line strings

File Output method

inflobj = open('data', 'W')	Open the file 'data' for Writing
outflobj.write(S)	Writes the string S to file
outflobj.writelines(L)	Writes each of the strings in list L to file
outflobj.close()	Closes the file

File mode usage

Symbol Name	Description
r	Open file in read mode
rb	Open file in read mode for binary format
r+	Open file for both reading and writing
rb+	Open file for reading and writing in binary format
w, wb, w+, wb+	Same like read this for writing
a	Open file for appending
a+	Open file for both appending and reading
ab+	Opens a file for both appending and reading in binary format

File mode usage

Method name	Description
Close()	Closing the file
Write()	Writing to file
Read(size)	Reading from file
Tell()	Tell pointer position (p=fo.tell())
seek(offset[, from])	Move file pointer (p = fo.seek(0, 0);
Rename()	Rename file(os.rename("test1.txt", "test2.txt"))
Remove()	Remove file(os.remove("text2.txt"))
truncate([size])	Delete content of file
Next()	Use to go on next line while iterating(line = fo.next())

Reading Example

```
filename = open('hello.txt')
print ("Here's your file %r:" % filename)
print (filename.read())
print ("Type the filename again:")
file_again = input("> ")
txt_again = open(file_again)
print (txt_again.read())
```

Writing Example

```
filename=input("what is your file name?")
print ("Opening the file...")
target = open(filename, 'w')
print ("Now I'm going to ask you for three lines.")
line1 = input("line 1: ")
line2 = input("line 2: ")
line3 = input("line 3: ")
print ("I'm going to write these to the file.")
target.write(line1)
target.write("\n")
target.write(line2)
target.write("\n")
target.write(line3)
target.write("\n")
print ("And finally, we close it.")
target.close()
```

Copy File1 to File2

```
from os.path import exists
from_file=open('hello.txt')
to_file=open('copy.txt')
print ("Copying from %s to %s" % (from_file, to_file))
in_file = open('hello.txt')
indata = in_file.read()
print ("The input file is %d bytes long" % len(indata))
print ("Does the output file exist? %r" % exists(to_file))
print ("Ready, hit Enter to continue, CTRL- C to abort.")
input()
out_file = open(to_file, 'w')
out_file.write(indata)
print ("Alright, all done.")
out_file.close()
in_file.close()
```


Thank you!