Module 04: Functions and modules

Click to add text

Agenda

- What is Function?
- Function Definition
- Function Arguments
- Local and Global Variables
- Pass by reference vs value
- Modules in Python
- Import Statement
- The *from* .. Import statement
- Locating Modules
- Packages intro
- Useful built-in statements

What is a Function?

- Function is a small sub program that perform a single task
- Functions provide better modularity and high code reuse
- Python has many build-in functions but user can define his own function
- Function definition starts with *def* keyword, function name and then list of parameters in () parentheses, : at the end
- Functions in python can use return statement.
  - return can come with expression, the one that we want to return
  - return can be empty
- return exits the function.

Functions documentation strings

- First function statement can be the function documentation called *docstring.*
- Python documentation string is used like a comment and provide a convenient way to access this documentation even in run time
- Python docstrings can be placed in functions, classes, modules
- In python docstring can be accessed with *help* function or *obj.__doc__*

```
help(my_func)  # prints my_func function docstring
s = my_func.__doc__  # returns prints my_func function docstring
```

Function Definition

```
def func_name(parameters):
        "docstring"
        command
        .
        .
        command
        [return expr]
```

For example:

```
def simple_print(str):
    "function prints value of parameter"
    print("parameter is:", str)

simple_print("Python functions are great!!")
```

docstring

Function call

Function overloading

- Python does not support function overloading
- When function with the same name defined more than once, they are treated as function redefinition and last one wins
- Python has many different ways to overcome this obstacle, like default arguments, variable-length parameters, etc (will be discussed later)

Function Arguments

- Required arguments:
  - Required arguments must be passed to a function in correct order and exact amount.

For example:

```
def print_data(color, size):
   print ("color: ", color, " and size : ", size)
print_data("Red", 4)      # prints color: Red and size : 4
print_data("Blue")    #generates arguments count error
```

Function Arguments – cont’d

- Keyword arguments:
  - Arguments can be pass by keyword (name). We can pass arguments in any order when passing them by names.
  - Non-keyword arguments are allowed after keyword ones

Example 1:

```
def print_data(size, color):
    print("color:", color, "and size:", size)

print_data(size=6, color="Blue")
print_data(color="Blue", size=6)
```

Example 2:

```
def func(x=1,y=2):
   print(x,y)

func(2,6)     #prints 2,6
func(2)          #prints 2,2
func(x=2,6)   # generates the error
```

Function Arguments – cont’d

- Default Arguments:
  - A default argument is an argument that defines a default value. The argument holds this default value, unless other value is passed
  - If function have both default and formal arguments, formal arguments must be defined first.

```
def print_data(color, size=10):
  print("color: ", color, " and size : ", size)

print_data("Yellow")                    # color: Yellow and size : 10
print_data(size = 3, color="Yellow")       #color: Yellow and size : 3
```

Function Arguments – cont'd

- Variable-Length parameters:
  - The special syntax, *args and **kwargs in function definitions is used to pass a variable number of arguments to a function.

  - The single asterisk form (*args) is used to pass a *non-keyworded*, argument list, that passed as tuple

  - The double asterisk form (**kwargs) is used to pass a *keyworded*, variable-length argument list, that passed as name-value dictionary

Function Arguments – cont’d

Variable-Length *args example:

```
def calc_aver(*args):
    sm = 0
    for elem in args:
        sm += elem
    return float(sm) / len(args)


aver = calc_aver(1, 2, 3, 4);
print(aver)                             # prints 2.5
aver = calc_aver(1, 2, 3, 4, 52);
print(aver)                             # prints 12.4
```

Function Arguments – cont'd

Variable-Length ****kwargs** example:

```
def print_named_variables(title, **dict):
    print ("title: ", title)
    cnt = 1
    for var_name,var_value in dict.items():
        print ("{}) {}={}" .format(cnt, var_name, var_value))
        cnt+=1

print_named_variables("all values", x=1,y="abc",z=-3.5)
```

```
title:  all values
        1) y=abc
        2) x=1
        3) z=-3.5
```

Local and Global Variables

- The scope of a variable determines where it can be accessed.
- Variables that are defined inside a function body have a local scope. Local variables can be accessed only inside the function in which they are declared
- Variables that are defined outside any function have a global scope. Global variables can be accessed throughout the program body by all functions.

Local and Global Variables – cont’d

```
def f():
    s = "I am globally unknown"  # define local variable s
    print(s)                          # prints "I am globally not known“



f()
print(s)                              # generates  error
```

Local and Global Variables – cont’t

Example 1:

```
def func():
    print("in func: ",
glob)


glob = 10

func()
print("in main
space: ", glob)
```

The Output:
in func: 10
in main space : 10

Example 2:

```
glob = 10


def func():
    glob = 11
    print("in func: ", glob)


func()
print("in main space: ", glob)
```

The Output:
in func: 11
in main space : 10 – Why??

Local and Global Variables – cont’d

Lets add one new line code to the *func* function from the previos example:

```
glob = 10


def func():
    print("in func: ", glob)
    glob = 11
    print("in func: ", glob)


func()
print("in main space: ", glob)
```

What will be the output now?

Python "assumes" that we want a local variable due to the assignment to *glob* inside of func(), so the first print statement throws this error message.

Local and Global Variables – cont’d

- Any variable which is changed or created inside of a function is local unless it hasn't been declared as a **global** variable.

```
glob = 10


def func():
    global glob
    print("in func: ", glob)
    glob = 11
    print("in func: ", glob)


func()
print("in main space: ", glob)
```

Pass by reference vs value

- All parameters in the Python language are passed by assignment.
- If we change what a parameter refers to within a function, the change also reflects back in the calling function. For example:

```
def chang_list( lst ):
    "This changes a passed list into this function"
    lst.append(4)
    print ("inside the function: ", lst)

list = [1,2,3]
chang_list(list )
print ("outside the function: ", list)
```

The Output:

```
inside the function:  [1, 2, 3, 4]
outside the function:  [1, 2, 3, 4]
```

Pass by reference vs value – cont’d

```
def chang_list( lst ):
    "This changes a passed list into this function"
    lst = [4]
    print ("inside the function: ", lst )

list = [1,2,3]
chang_list(list )
print ("outside the function: ", list)
```

The Output:

```
inside the function:  [4]
outside the function:  [1, 2, 3]
```

Pass by reference vs value – cont’d

```
def chang_string( s ):
    s = "is fun"
    print ("inside the function: ", s)

s = "python"
chang_string(s)
print ("outside the function: ", s)
```

The Output:

```
inside the function:  is fun
outside the function:  python
```

Modules in Python

- As your program gets longer, you may want to split it into several source files for easier maintenance
- File that contains definitions only can be treated as module and can be used in python programs.
- Module can define functions, classes and global variables
- Definitions from a module can be *imported* into python program and other modules

Import Statement

- *import module_name* imports all the *module_name*'s definitions under namespace

```
import my_module
```

- The default namespace name is module name.
- Namespace name can be changed with *as* statement in import line

```
import my_module as MM
```

- All module's definitions can be accessed through namespace only

```
my_module.my_func(…)
```

Import Statement - example

- Lets say we have utils.py file with 2 function:
  - def do_somthing1():

    ….

  - def do_somthing2():

    ….

```
import utils
utils.do_somthing1()
```

The *from...import* Statement

- Python's *from* statement lets you import specific attributes or all attributes from a module into the current namespace.
- The *from...import* has the following syntax: from module_name import name1, name2, ... nameN

```
from utils import do_something1
from utils import *
```

- Namespaces are omitted using *from...import* add only direct access is possible

```
do_something1()
utils.do_something1()          #generates error
```

Init file

- The __init__.py file, commonly referred to as the "init file," is a special file in a Python package that serves as an initializer for the package. It is executed when the package is imported, allowing you to perform necessary setup and define the contents of the package.

- In addition to its role as an initializer, the init file can also transform a directory into a Python package. Without an init file, the directory would simply be treated as a regular folder without any package-specific functionality.

Init file – cont’d

- **Imports**: You can import other modules or packages that are part of the package. This allows you to organize your code into separate modules and use them within the package.

- **Variable and Constant Definitions:** You can initialize variables, constants, or any other objects that will be used by the package. These definitions can be accessed by other modules within the package.

- **Function Definitions:** You can define functions that are specific to the package or provide utility functions that can be used throughout the package.

- **Initialization Code:** You can include any necessary initialization code that needs to be executed when the package is imported. This can involve setting up configurations, connecting to databases, or performing any other required setup tasks.

Overall, the init file plays a vital role in structuring and initializing a Python package. It allows you to organize code, define package-level variables and functions, and perform necessary setup operations when the package is imported.

Init file

Demo

Init file – demo

- __init__.py

```
from .english import say_hello as hello_english
from .spanish import say_hello as hello_spanish
```

- english.py

```
def say_hello:()
    print("Hello!")
```

- spanish.py

```
def say_hello:()
    print("¡Hola!")
```

Let's consider a package named greetings, which provides simple greeting functions. We can create the following directory structure:

```
greetings/
    __init__.py
    english.py
    spanish.py
```

Output:

```
import greetings

greetings.hello_english()  # Output: Hello!
greetings.hello_spanish()  # Output: ¡Hola!
```

Locating Modules

- When importing modules the interpreter searches:
  - First interpreter searches in the current directory.
  - If the module isn't found, Python then searches each directory in the shell variable PYTHONPATH
  - If the module isn’t found, the interpreter searched in standard installation path. (In Linux/Unix for example in /usr/local/lib/python)
  - If the module is not found in any of the above locations, the interpreter raises an “ImportError” indicating that the module could not be found.

Python packages

- To help organize modules and provide a naming hierarchy, python has a concept of packages

- The __init__.py files are required to make Python treat the directories as package

- Access to modules inside a package through the full name: *package_name.module_name*

- Packages can contains sub-packages

Lab 01

Lab

Questions