FTDS //PYTHON: FUNCTIONS, MODULE & PACKAGES

Hacktiv8 DS Curriculum Team

Phase 0
Day 3 AM
2021

Phase 0

2021

Contents

WEEK 1

Python:

Functions

Python Function

A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing. Function blocks begin with the keyword `def` followed by the function name and parentheses (`( )`). For example:

```
def function_name( parameters ):
   "docstring"
   statement(s)
```

By default, parameters have a positional behavior and you need to inform them in the same order that they were defined.

The docstring is short for documentation string. It is used to explain in brief, what a function does.

Calling a Function

Defining a function only gives it a name, specifies the parameters that are to be included in the function and structures the blocks of code. The code below is the example to call `printme()` function:

```
# Function definition is here
def printme( str ):
   "This prints a passed string into this function"
   print(str)
   return

# Now you can call printme function
printme("I'm first call to user defined function!")
printme("Again second call to the same function")
```

Pass by Reference vs Value

All parameters (arguments) in the Python language are passed by reference. For example:

```
# Function definition is here
def changeme( mylist ):
   "This changes a passed list into this function"
   mylist.append([1,2,3,4])
   print("Values inside the function: ", mylist)
   return

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

Function Arguments

You can call a function by using the following types of formal arguments:

- **Required arguments**
- **Keyword arguments**
- **Default arguments**
- **Variable-length arguments**

Required arguments

Required arguments are the arguments passed to a function in correct positional order. Here, the number of arguments in the function call should match exactly with the function definition. To call the function `printme()`, you definitely need to pass one argument, otherwise it gives a syntax error as follows:

```
# Function definition is here
def printme( str ):
   "This prints a passed string into this function"
   print(str)
   return;

# Now you can call printme function
printme()
```

Keyword arguments

Keyword arguments are related to the function calls. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name. You can also make keyword calls to the `printme()` function in the following ways:

```
# Function definition is here
def printme( str ):
   "This prints a passed string into this function"
   print(str)
   return;

# Now you can call printme function
printme(str = "Hacktiv8")
```

Default arguments

A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument. For example:

```
# Function definition is here
def printinfo( name, age = 26 ):
   "This prints a passed info into this function"
   print("Name: ", name)
   print("Age: ", age)
   return;

# Now you can call printinfo function
printinfo( age=50, name="hacktiv8" )
printinfo( name="hacktiv" )
```

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 and are not named in the function definition, unlike required and default arguments. here is the syntax:

```
def functionname([formal_args,] *var_args_tuple ):
   "function_docstring"
   function_suite
   return [expression]
```

An asterisk (*) is placed before the variable name that holds the values of all non-keyword variable arguments.

The Anonymous Functions

These functions are called anonymous because they are not declared in the standard manner by using the `def` keyword. You can use the `lambda` keyword to create small anonymous functions. The code below is the example to show how `lambda` form of function work:

```
# Function definition is here
sum = lambda arg1, arg2: arg1 + arg2;
def sum(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 ))
```

The return Statement

A return statement with no arguments is the same as return `None`. You can return a value from a function as follows:

```
# Function definition is here
def sum(arg1, arg2):
    # Add both the parameters and return them."
    total = arg1 + arg2
    total2 = total + arg1
    print("Inside the function : ", total)
    return total2

# Now you can call sum function
total = sum(10, 20)
print("Outside the function : ", total)
```

Scope of Variables

All variables in a program may not be accessible at all locations in that program. This depends on where you have declared a variable.
The scope of a variable determines the portion of the program where you can access a particular identifier. There are two basic scopes of variables in Python:

- **Global variables**
- **Local variables**

Global vs. Local variables

Variables that are defined inside a function body have a local scope, and those defined outside have a global scope.

```
total = 0  #This is global variable
def sum( arg1, arg2 ):
   total = arg1 + arg2 # Here total is local variable.
   print("Inside the function local total : ", total)
   return total

sum( 10, 20 );
print("Outside the function global total : ", total)
```

Modular programming

Modular programming refers to the process of breaking a large, unwieldy programming task into separate, smaller, more manageable subtasks or modules. Individual modules can then be cobbled together like building blocks to create a larger application. There are several advantages to modularizing code in a large application:

- **Simplicity**
- **Maintainability**
- **Reusability**
- **Scoping**

Python Modules: Overview

There are actually three different ways to define a module in Python:

- **A module can be written in Python itself.**
- **A module can be written in C and loaded dynamically at run-time, like the re (regular expression) module.**
- **A built-in module is intrinsically contained in the interpreter, like the `itertools` module.**

A module's contents are accessed the same way in all three cases: with the `import` statement.

Python Modules: Overview

For example, suppose you have created a file called `mod.py` containing the following:

```
s = "Hacktiv8-PTP Python For Data Science"
a = [100, 200, 300]

def foo(arg):
    print(f'arg = {arg}')

class Foo:
    pass
```

Python Modules: Overview

Assuming `mod.py` is in an appropriate location, which you will learn more about shortly, these objects can be accessed by importing the module as follows:

```
import mod
print(mod.s)
print(mod.a)
mod.foo(['quux', 'corge', 'grault'])
x = mod.Foo()
print(x)
```

The Module Search Path

The resulting search path is accessible in the Python variable `sys.path`, which is obtained from a module named `sys`:

```
import sys
sys.path
```

Thus, to ensure your module is found, you need to do one of the following:

- **Put `mod.py` in the directory where the input script is located or the current directory, if interactive**
- **Modify the `PYTHONPATH` environment variable to contain the directory where `mod.py` is located before starting the interpreter**
- **Or: Put `mod.py` in one of the directories already contained in the `PYTHONPATH` variable**
- **Put `mod.py` in one of the installation-dependent directories, which you may or may not have write-access to, depending on the OS**

The import Statement

Module contents are made available to the caller with the `import` statement. To be accessed in the local context, names of objects defined in the module must be prefixed by `mod`:

```
Import mod
print(mod.s)
```

An alternate form of the `import` statement allows individual objects from the module to be imported directly into the caller's symbol table:

```
from mod import s, foo
print(s)
print(foo('quux'))
```

The import Statement

It is even possible to indiscriminately import everything from a module at one fell swoop:

```
from mod import *
print(s)
print(a)
```

You can also import an entire module under an alternate name:

```
import mod as my_module
print(my_module.a)
```

The `dir()` Function

The `dir()` function can be useful for identifying what exactly has been added to the namespace by an `import` statement:

```
>>> from mod import a, Foo
>>> dir()
['Foo', '__annotations__', '__builtins__', '__doc__', '__loader__',
'__name__',
'__package__', '__spec__', 'a', 'mod']
```

When given an argument that is the name of a module, `dir()` lists the names defined in the module:

```
>>> import mod
>>> dir(mod)
['Foo', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__',
'__name__', '__package__', '__spec__', 'a', 'foo', 's']
```

Python Packages

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.

Creating a package is quite straightforward, since it makes use of the operating system's inherent hierarchical file structure. Consider the following arrangement:

Python Packages

Here, there is a directory named `pkg` that contains two modules, `mod1.py` and `mod2.py`. The contents of the modules are:

- `mod1.py`

```
def foo():
  print('[mod1] foo()')

class Foo:
  pass
```

- `mod2.py`

```
def bar():
  print('[mod2] bar()')

class Bar:
  pass
```

for n in x:
    print(n)

Python Packages

With the previous structure, if the `pkg` directory resides in a location where it can be found (in one of the directories contained in `sys.path`), you can refer to the two modules with dot notation (`pkg.mod1`, `pkg.mod2`) and import them with the syntax you are already familiar with:

```
>>> import pkg.mod1, pkg.mod2
>>> pkg.mod1.foo()
[mod1] foo()

>>> from pkg.mod1 import foo
>>> foo()
[mod1] foo()
```

PIP

`pip` is a package manager for Python. That means it's a tool that allows you to install and manage additional libraries and dependencies that are not distributed as part of the standard library.

You can learn about `pip` supported commands by running it with help:

```
pip help
```

As you can see, `pip` provides an install command to install packages. You can run it to install the requests package:

```
pip install requests
```

You can use the list command to see the packages installed in your environment:

```
pip list
```

Hacktiv8 DS
Curriculum
Team

External References

Colab Link

Visit Here