Introduction to Programming Using Python

Microsoft Python Certification | **Exam 98-381**

Lecture 5:

Files and Functions

Lecturer: **James W. Jiang**, Ph.D. | Summer, 2018

Microsoft™

SAWY PRO

PROFESSIONAL EDUCATION TRAINING

Read and Write Files:
Introduction

File Operation

When we want to read from or write to a file we need to open it first. When we are done, it needs to be closed, so that resources that are tied with the file are freed.

Hence, in Python, a file operation takes place in the following order.

- Open a file
- Read or write (perform operation)
- Close the file

How to open a file

Python has a built-in function open() to open a file. This function returns a file object, also called a handle, as it is used to read or modify the file accordingly.

```
f = open("test.txt")

# open file in current directory

f = open("C:/Python33/README.txt")

# specifying full path
```

Mode

We can specify the mode while opening a file. In mode, we specify whether we want to read 'r', write 'w' or append 'a' to the file. We also specify if we want to open the file in text mode or binary mode.

The default is reading in text mode. In this mode, we get strings when reading from the file.

On the other hand, binary mode returns bytes and this is the mode to be used when dealing with non-text files like image or exe files.

Python File Modes

Mode	Description
`'r'`	Open a file for reading. (default)
`'w'`	Open a file for writing. Creates a new file if it does not exist or truncates the file if it exists.
`'x'`	Open a file for exclusive creation. If the file already exists, the operation fails.
`'a'`	Open for appending at the end of the file without truncating it. Creates a new file if it does not exist.
`'t'`	Open in text mode. (default)
`'b'`	Open in binary mode.

open()

```
f = open("test.txt")
# equivalent to 'r' or 'rt'
f = open("test.txt",'w')
# write in text mode
f = open("img.bmp",'r+b')
# read and write in binary mode
```

close a file Using Python

When we are done with operations to the file, we need to properly close the file.

Closing a file will free up the resources that were tied with the file and is done using Python close() method.

Python has a garbage collector to clean up unreferenced objects but, we must not rely on it to close the file.

```
f = open("test.txt",encoding = 'utf-8')
# perform file operations
f.close()
```

close a file Using Python (foolproof)

This method is not entirely safe. If an exception occurs when we are performing some operation with the file, the code exits without closing the file. A safer way is to use a try...`finally` block.

```
try:
    f = open("test.txt",encoding = 'utf-8')
    # perform file operations
finally:
    f.close()
```

Read and Write Files:
Directory & Files Management

What is Directory in Python?

If there are a large number of files to handle in your Python program, you can arrange your code within different directories to make things more manageable.

A directory or folder is a collection of files and sub directories. Python has the os module, which provides us with many useful methods to work with directories (and files as well).

Get Current Directory

We can get the present working directory using the getcwd() method.

```
import os

os.getcwd()
```

The extra backslash implies escape sequence. The print() function will render this properly.

```
print(os.getcwd())
```

Changing Directory

We can change the current working directory using the chdir() method.

The new path that we want to change to must be supplied as a string to this method. We can use both forward slash (/) or the backward slash (\) to separate path elements.

It is safer to use escape sequence when using the backward slash.

```
os.chdir('C:\\Users\\James')
print(os.getcwd())
```

Making a New Directory

We can make a new directory using the mkdir() method.

This method takes in the path of the new directory. If the full path is not specified, the new directory is created in the current working directory.

```
os.mkdir('test')
```

```
os.listdir()
```

Renaming a Directory or a File

The rename() method can rename a directory or a file.

The first argument is the old name and the new name must be supplies as the second argument.

```
os.listdir()
['test']
os.rename('test','new_one')
os.listdir()
```

Read and Write Files:

Write to .txt Files

.write()

Use the write function

```
fileName = "GuestList.txt"
accessMode = "w"
myFile = open(fileName, accessMode)
myFile.write("Hi there!")
myFile.write("How are you?")
```

.write()

Think back to the print statement.
How did we specify to display text over multiple lines?
\n

```
fileName = "GuestList.txt"
accessMode = "w"
myFile = open(fileName, accessMode)
myFile.write("Hi there!\n")
myFile.write("How are you?")
```

.close()

When you are finished you should always close the file
Use the close method

```
fileName = "GuestList.txt"
accessMode = "w"
myFile = open(fileName, accessMode)
myFile.write("Hi there!\n")
myFile.write("How are you?")
myFile.close()
```

Read and Write Files:
Read .txt Files

open()

Use the open function

```
myFile = open(fileName, accessMode)
```

Look familiar? Yes, it's the same method we use to write to a file. So how does the program know whether to read or write? The access mode

Access mode	Action
r	Read the file
w	Write to the file
a	Append to the existing file content
b	Open a binary file

.read()

Use the read method

```
fileContent= myFile.read()
```

The read method will return the entire contents of the file into the specified string variable

read one line

Use the readline method

```
fileContent= myFile.readline()
```

The readline method will return one line from the file

Read and Write Files:
CSV Files

CSV Files

A common format for storing information in a file is Comma Separated (CSV)
A CSV file contains data separated by a character (usually a comma).
Each row represents one record of data
It is sometimes called a Character Separated Values file because the separating character could be a different character such as a semi colon ';'

import csv

If you are reading a CSV file, there is a csv library that will help you!

To access the features in the csv library you must import it

```
import csv
```


csv.reader

Now you can use the reader function to return all the rows from the file into a list. The reader function will take an open csv file and return each row from the file into a list.

```
dataFromFile = csv.reader(myCSVfile)
```

If your file is not using a comma to separate the values, you can tell the reader function what character is used as a delimiter

```
dataFromFile = csv.reader(myCSVFile, delimiter=",")
```

csv.reader()

```
fileName = "GuestList.txt"
accessMode = "r"
with open(fileName, accessMode) as myCSVFile:
        #Read the file contents
        dataFromFile = csv.reader(myCSVFile)
```

with open

Why do we have a 'with' and ':' ?

```
with open(fileName, accessMode) as myCSVFile:
```

Programs should always open a file, and close it when they are done
If they don't sometimes the code crashes when you try to re-open a file that wasn't closed last time you ran your code
The 'with' ':' syntax is used for certain methods to make sure clean up code such as close file runs even if there is an error.

with open

Use a for loop to loop through the values in the list
Each row will be one value

```
with open(fileName, accessMode) as myCSVFile:
      #Read the file contents
      dataFromFile = csv.reader(myCSVFile)
      #For loop that will run once per row
      for row in dataFromFile :
         print(row)
```

Functions:

Why We Need Functions

You have already used function

Function: A reusable section of code with a name that does something

Sometimes called a method

You have already used functions!

- print
- open
- write
- close

Why create functions?

- Code reuse
- You are doing the same thing over and over again
- Simplify your code
- Functions have names to define what they do
- Breakdown complex blocks of code
- Easier to make changes
- If it's only been written once, you only have to update it once

Types of Functions

Types of Functions

Basically, we can divide functions into the following two types:

- Built-in functions - Functions that are built into Python.
- User-defined functions - Functions defined by the users themselves.

Functions:
Define a Function

Define a Function

- The first line of the function definition is called the **header**; the rest is called the body.
- The header has to end with a colon and the body has to be indented.
- By convention, indentation is always four spaces.
- The body can contain any number of statements.
- To end the function, you have to enter an empty line.

Define a Function

```
def print_quote(): # header
    print("all work no play makes James?")
    print("a dull boy.")

def print_quote():
....print("all work no play makes James?")
....print("a dull boy.")
....
```

Functions:
Call A Function

Call a Function

```
print_quote()

all work no play makes James?

a dull boy.
```

Use Pre-defined Function in Another Function

Once you have defined a function, you can use it inside another function. For example, to repeat the previous refrain, we could write the following equation

```
def repeat_quotes():
    print_quote()
    print_quote()
def repeat_quotes():
....print_quote()
....print_quote()
....
```

Call a Function

```
repeat_quotes()
all work no play makes James?
a dull boy.
all work no play makes James?
a dull boy.
```

Functions:
Arguments

Arguments for a Function

Inside the function, the arguments are assigned to variables called parameters. Here is a definition for a function that takes an argument:

Argument in a Function

```
def print_your_quote(Name): # header
    first = "all work no play makes "
    second = Name
    third = "?"
    print(first + second + third)
    print("a dull boy.")
```

Argument in a Function

```
def print_your_quote(Name): # header
....first = "all work no play makes "
....second = Name
....third = "?"
....print(first + second + third)
....print("a dull boy.")
....
```

Argument in a Function

```
print_your_quote('James')

all work no play makes James?

a dull boy.
```

Argument in a Function

```
name = 'James'
print_your_quote(name)
all work no play makes James?
a dull boy.
```

Multiple Arguments

If you have multiple arguments, simply add them in, separated by commas

```
def displayMessage(greeting, name):
    message = greeting + ', ' + name
    print(message)
    return

displayMessage('Hi', 'Christopher')
```

Functions:
Default Argument

Default Argument

Function arguments can have default values in Python.

We can provide a default value to an argument by using the assignment operator (=). Here is an example.

Default Argument

```
def greet(name, msg = "Good morning!"):
    """
    This function greets to the person with the
    provided message. If message is not provided,
    it defaults to "Good morning!"
    """
    print("Hello",name + ', ' + msg)

greet("Kate")
greet("Bruce","How do you do?")
```

Default Argument

```
def greet(name, msg = "Good morning!"):
    """
    This function greets to the person with the
    provided message. If message is not provided,
    it defaults to "Good morning!"
    """
    print("Hello",name + ', ' + msg)

greet("Kate")
greet("Bruce","How do you do?")
```

Default Argument

In this function, the parameter name does not have a default value and is required (mandatory) during a call.

On the other hand, the parameter msg has a default value of "Good morning!". So, it is optional during a call. If a value is provided, it will overwrite the default value.

Any number of arguments in a function can have a default value. But once we have a default argument, all the arguments to its right must also have default values.

This means to say, non-default arguments cannot follow default arguments.

Default Argument

For example, if we had defined the function header above as:

```
def greet(msg = "Good morning!", name):
    """
    This function greets to the person with the
    provided message. If message is not provided,
    it defaults to "Good morning!"
    """
    print("Hello",name + ', ' + msg)
greet("Kate")
```

Functions:
Arbitrary Arguments

Arbitrary Arguments

Sometimes, we do not know in advance the number of arguments that will be passed into a function. Python allows us to handle this kind of situation through function calls with arbitrary number of arguments.

In the function definition we use an asterisk (*) before the parameter name to denote this kind of argument. Here is an example.

Arbitrary Arguments

```
def greet(*names):
    """This function greets all
    the person in the names tuple."""

    # names is a tuple with arguments
    for name in names:
        print("Hello",name)

greet("Monica","Luke","Steve","John")
```

Functions:
Docstring

Docstring

The first string after the function header is called the docstring and is short for documentation string. It is used to explain in brief, what a function does.

Although optional, documentation is a good programming practice. Unless you can remember what you had for dinner last week, always document your code.

In the above example, we have a docstring immediately below the function header. We generally use triple quotes so that docstring can extend up to multiple lines. This string is available to us as `__doc__` attribute of the function.

Docstring

```
def greet(name):
        """This function greets to
        the person passed in as
        parameter"""
        print("Hello, " + name + ". Good morning!")
```

In the above example, we have a docstring immediately below the function header. We generally use triple quotes so that docstring can extend up to multiple lines. This string is available to us as __doc__ attribute of the function.

Print Docstring

Try running the following into the Python shell to see the output.

```
print(greet.__doc__)
```

Functions:
Recursion

Recursion

It is legal for one function to call another; it is also legal for a function to call itself. It may not be obvious why that is a good thing, but it turns out to be one of the most magical things a program can do.

```
def countdown(n):
    if n <= 0:
        print('Blastoff!')
    else:
        print(n)
        countdown(n-1)
```

find the factorial of a number

```
def calc_factorial(x):
    """This is a recursive function
    to find the factorial of an integer"""
    if x == 1:
        return 1
    else:
        return (x * calc_factorial(x-1))
num = 4
print("The factorial of", num, "is",calc_factorial(num))
```

Keyboard Input

Python provides a built-in function called input that stops the program and waits for the user to type something.

```
text = input()
What are you waiting for?
text
What are you waiting for?

name = input('What...is your name?\n')
```

Functions:
Return Results

Return Values

Calling the function generates a return value, which we usually assign to a variable or use as part of an expression.

```
def area(radius):
    a = math.pi * radius**2
    return a
```

Factorial

$0! = 1$

$n! = n(n - 1)!$

```
def factorial(n):
    if n == 0:
        return 1
    else:
        recurse = factorial(n-1)
        result = n * recurse
        return result
```

Functions:
Variable Scope

Scope of a Variable

Scope of a variable is the portion of a program where the variable is recognized. Parameters and variables defined inside a function is not visible from outside. Hence, they have a local scope.

Lifetime of a variable is the period throughout which the variable exits in the memory. The lifetime of variables inside a function is as long as the function executes.

They are destroyed once we return from the function. Hence, a function does not remember the value of a variable from its previous calls.

Here is an example to illustrate the scope of a variable inside a function.

Scope of a Variable

```
def my_func():
        x = 10
        print("Value inside function:",x)


x = 20
my_func()
print("Value outside function:",x)
```

Scope

Although there are various unique namespaces defined, we may not be able to access all of them from every part of the program. The concept of scope comes into play.

Scope is the portion of the program from where a namespace can be accessed directly without any prefix.

At any given moment, there are at least three nested scopes.

- Scope of the current function which has local names
- Scope of the module which has global names
- Outermost scope which has built-in names

Global Variables

In Python, a variable declared outside of the function or in global scope is known as global variable. This means, global variable can be accessed inside or outside of the function.

```
x = "global"

def foo():

    print("x inside :", x)

foo()

print("x outside:", x)
```

Global Variables

```
x = "global"
def foo():
    x = x * 2
    print(x)
foo()
```

The output shows an error because Python treats x as a local variable and x is also not defined inside foo().

Local Variables

```
def foo():
    y = "local"


foo()
print(y)
```

When we run the code, the will output be:

```
NameError: name 'y' is not defined
```

Global and local variables

```
x = "global"
def foo():
    global x
    y = "local"
    x = x * 2
    print(x)
    print(y)
foo()
```

Global and local variables

```
x = 5
def foo():
    x = 10
    print("local x:", x)
foo()
print("global x:", x)
local x: 10
global x: 5
```

Global and local variables

```
def outer_function():
    a = 20
    def inner_function():
        a = 30
        print('a =',a)
    inner_function()
    print('a =',a)
a = 10
outer_function()
print('a =',a)
```

Global and local variables

In this program, three different variables a are defined in separate namespaces and accessed accordingly. While in the following program,

```
def outer_function():
    global a; a = 20
    def inner_function():
        global a; a = 30
        print('a =',a)
    inner_function()
    print('a =',a)
a = 10
outer_function(); print('a =',a)
```

add()

```
c = 0 # global variable


def add():
    global c
    c = c + 2 # increment by 2
    print("Inside add():", c)


add()
print("In main:", c)
```

Global Variables Across Python Modules

In Python, we create a single module config.py to hold global variables and share information across Python modules within the same program.

Here is how we can share global variable across the python modules.

Create a config.py file, to store global variables

```
a = 0

b = "empty"
```

Create a update.py file, to change global variables

Global in Nested Functions

```
def foo():
    x = 20
    def bar():
        global x
        x = 25
    print("Before calling bar: ", x)
    print("Calling bar now")
    bar()
    print("After calling bar: ", x)
foo()
print("x in main : ", x)
```

Functions:
Lambda Functions

Lambda Functions

While normal functions are defined using the def keyword, in Python anonymous functions are defined using the lambda keyword. Hence, anonymous functions are also called lambda functions.

```
double = lambda x: x * 2


# Output: 10
print(double(5))
```

Summary

A function definition consists of following components.

Keyword def marks the start of function header.

A function name to uniquely identify it. Function naming follows the same rules of writing identifiers in Python.

Parameters (arguments) through which we pass values to a function. They are optional.

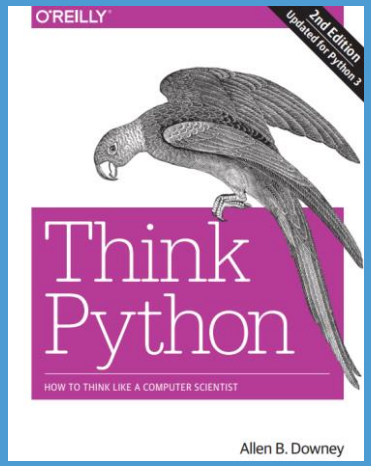
A colon (:) to mark the end of function header.

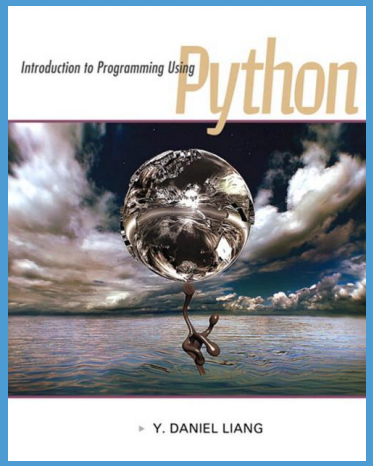
Optional documentation string (docstring) to describe what the function does.

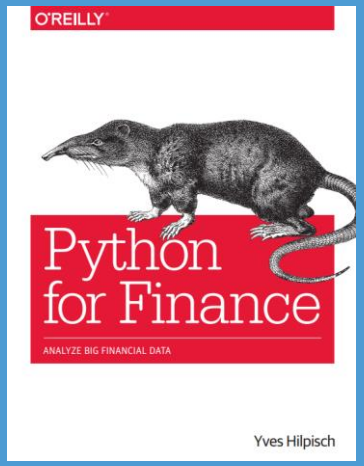
One or more valid python statements that make up the function body. Statements must have same indentation level (usually 4 spaces).

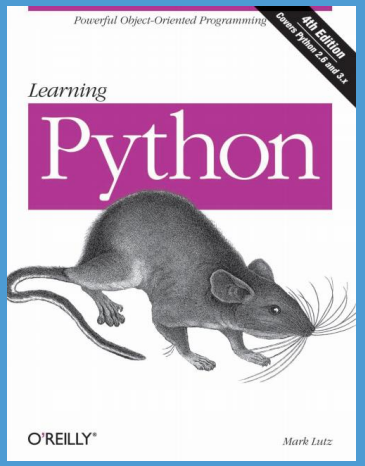
An optional return statement to return a value from the function.

References


O'REILLY
2nd Edition
Updated for Python 3
Think Python
HOW TO THINK LIKE A COMPUTER SCIENTIST
Allen B. Downey


Introduction to Programming Using Python
Y. DANIEL LIANG


O'REILLY
Python for Finance
ANALYZE BIG FINANCIAL DATA
Yves Hilpisch


Powerful Object-Oriented Programming
4th Edition
Learning
Python
O'REILLY
Mark Lutz