Introduction to Programming in Python Workshop Part 1b

By William Chan

Objectives

In this lecture, you will learn the general syntax of Python for the general programming constructs

1. Variables and data types
2. Conditionals
3. Loops and flow control
4. Functions

Python: the Interactive Console

Python has an interactive console to test python expressions. If Python is installed on your computer, you can access the interactive console by doing the following

1. Opening the terminal on your Mac or MS-DOS on Windows (terminal can be opened via spotlight on the Mac and MS-DOS can be open via Run and typing in “cmd” without the quotes)
2. Type “python” without the quotes in the terminal and now you should be able to type and try out python statements

Another (more traditional) way of trying out Python code

Running the Python program from other python files

1. After creating or editing a file with the .py extension in Sublime Text 2 or 3, run the file by opening the terminal on Mac or MS-DOS on Windows
2. Navigate to the appropriate directory for which the .py file is saved and type `python <name_of_the_python_file>.py`

Common Python Data Types

In programming, there are different data types. Some of the common data types include:

1. int representing integers
2. float representing decimal numbers (without getting into too much detail)
3. string representing list of characters, characters can be thought of as a single letter or a symbol
4. boolean representing either `True` or `False`
5. list is simply a list of values and because Python is dynamic, the list can hold any type
6. dictionary is a key-value data structure where you access a value by a key in the dictionary
7. object representing any type

The Other Not-So Common Data Types

1. long representing integers larger than 9223372036854775807
2. complex representing imaginary numbers
3. set is a list of unique values
4. tuples are immutable lists

For more information on built-in types, feel free to refer to the following links:

1. https://docs.python.org/2.7/library/stdtypes.html
2. https://docs.python.org/2.7/tutorial/datastructures.html

Comments in Python

Comments are code that are not executed in Python and usually are used to tell other developers what the code is doing.

Good comments usually puts emphasis on code that is not quite clear to other developers but for this workshop, comments will be to show the sample output of certain expressions.

Typical comments are preceded with a pound sign like the following.

```
# describes the below code
...
```

Operating on Values

Values can be operated on. As an example, the simplest values that could be operated on are numerical values. Try the following mathematical operations in the interactive console.

```
7 + 7     # 14
10 - 7    # 3
7 * 7     # 49
3.14 * 6 # 18.84
7 / 5       # 1 (because it uses integer division)
7 % 5       # 2 (because it gets the remainder)
```

Initializing Python Variables

Variables store values of previously mentioned types. When naming variables, try to use a name that communicates what kind of value its storing. Try the following in the Python interactive console in the terminal.

```
lucky_number = 7
sample_string = 'hello world'
pi = 3.14
list_of_values = [1, 2, 'hello world']
sample_dict = {'key': 'value'}

type(lucky_number)    # <type 'int'>
type(sample_string)   # <type 'str'>
type(pi)              # <type 'float'>
type(list_of_values)  # <type 'list'>
type(sample_dict)     # <type dict'>
```

Operating on Python Variables

Operating on variables are the same as operating on values but with the value replaced with a variable name. Remember the variable now stores the value. Try the below expressions in the interactive console.

```
lucky_number = 7


lucky_number / 5     # 1 (same as 7 / 5)
lucky_number % 5     # 2 (same as 7 % 2)
```

A Simple Variable Exercise

What are the results of the sample statements? Figure out the value and then try them in the interactive console to confirm.

```
lucky_number = 8
lucky_number % 2   # ?
lucky_number / 3   # ?

lucky_number = 7
lucky_number % 2   # ?
lucky_number / 3   # ?
```

Conditionals and Flow Control

In order to understand conditionals and flow control, it is important to understand booleans (`True` or `False`). As a side note, anything that requires nesting, requires to be tabbed in. This organizes the code in other languages, but is required for Python. It helps to see how the logic is nested and what "belongs" to what.

```
if True:
    # prints to the console
    print('its true so i print')


if False:
    # does not print to the console
    print('its false so i refuse to print')
```

Comparing Values to Create Boolean Values

Sample comparison operators. For more details, you can also reference: https://docs.python.org/2/library/stdtypes.html#comparisons

Operation	Meaning
<	strictly less than
<=	less than or equal
>	strictly greater than
>=	greater than or equal
==	equal
!=	not equal
is	object identity
is not	negated object identity

Sample Code with the Comparison Operators

```
lucky_number = 7

lucky_number == 7 # True
lucky_number > 3  # True
lucky_number >= 7 # True
lucky_number < 10 # True
lucky_number <= 7 # True

lucky_number != 7 # False
lucky_number == 6 # False
lucky_number != 7 # False
lucky_number > 10 # False
lucky_number < 3  # False
```

Combining Comparison Operators in Conditionals to Control Flow

```
# imports a standard library called random
import random

lucky_number = 7
random_int = random.randint(0, 10)     # generate a number from 0 to 10

if random_int == lucky_number:
    # executes the code here if the expression is True
    print('i have good luck!')
else:
    # executes the code here if the expression returns False
    print('i have bad luck!')
```

Other Boolean Operations

Booleans can be created using the comparison operators but they can also be compounded using the and and or operators. Feel free to refer to: https://docs.python.org/2/library/stdtypes.html#boolean-operations-and-or-not

```
# in “and” both sides have to be True
True and True        # True
True and False       # False
False and True       # False

# in “or” only one has to be True
False or True        # True
True or False        # True
False or False       # False
```

The Else-If Conditional

“elif” is the keyword to determine multiple conditionals. It works its way down, first checking the if the expression under the “if” is true and then move down to the “elif”. If it ever reaches a `True` value, it will stop going through other branches. As a side note, the more specific

```
import random


random_num = random.randint(0, 50)
if random_num % 5 == 0 and random_num % 3 == 0:
     print('fizzbuzz ' + random_num)
elif random_num % 5 == 0:
     print('fizz ' + random_num)
elif random_num % 3 == 0:
     print('buzz ' + random_num)
else:
     print(random_num)
```

Loops to Control Flow

Loops are to execute until the expression becomes true. It generally requires the following:

1. Initializing a variable
2. Compare the variable in an expression that returns `True` or `False`
3. Update the variable initialized (this will eventually update the variable to be `False` or else it would be a never ending loop)

The “while” loop

While the expression is `True`, continue executing until the expression is `False`. The order goes like the following:

```
# step 1: set the value for the expression
while <step 2: expression>:
    # do something...
    # step 3: update expression
```

A “while” loop example

```
import random

# step 1: initializing the value to be used to compare in the while statement
lucky_number = random.randint(0, 10)

# step 2: compare the value to see if it would be True
while lucky_number != 7:
    print('not so lucky')
    # 3. updates lucky_number to be compared again
    lucky_number = random.randint(0, 10)
```

The “for” loop

The for loop is usually for iterating over values in an iterable which are generally lists, sets, or tuples. It iterates over each value to the last value and exits.

List example:

```
some_list = [1, 2, 3]
for val in some_list:
    print(val)
```

Defining Functions

Functions are defined using the “def” keyword and are called using the name of the function. A simple template looks like the following:

```
# defining a function
def function_name():
    # do something

function_name() # call the function
```

A Simple Function Exercise

Define a function named `fizzbuzz` that iterates from 0 to 50 and prints fizzbuzz if the number is both divisible by 5 and 3, fizz if it is only divisible by 5, and buzz if it is only divisible by 3, else it just prints the number.

Defining Functions that Take Parameter Values

Functions can also take parameter values to use. It follows a very similar template. The below function takes two parameters with the variable name of parameter_1 and parameter_2.

```
def function_name(parameter_1, parameter_2):
    # parameter_1 and parameter_2 can now be accessed as
    # individual variables inside the function

# calling the function with two sample parameter values, 1 and 2
function_name(1, 2)
```

A Function with Parameters Exercise

Using the fizzbuzz function written in a previous exercise, rewrite the `fizzbuzz` function to take a parameter value to iterate up until. Previously the value was hardcoded to 50 but it can be made more flexible to have it accept any value when called.

For example, the call would now be the following:

```
fizzbuzz(25)          # iterating from 0 to 25 printing fizzbuzz, fizz,
                      # buzz, and the number where appropriate
fizzbuzz(100)         # now iterating from 0 to 100
```

Defining Functions that Take Default Parameter Values

Functions can take default parameters by following, another yet similar template.

```
def function_name(default_variable=10):
    # do something with the variable default_variable that is passed
    # into the function

function_name()       # will use the default variable with value 10
function_name(20)     # will use the default variable with the value 20
```

A Function with Default Parameters Exercise

Again with a small change to the `fizzbuzz` function set the default parameter to 50 so that if a value is not passed to the fizzbuzz function, it will iterate from 0 to 50 and print out fizzbuzz, fizz, buzz, and the number accordingly.

It should be called like the following:

```
fizzbuzz()      # iterate from 0 to 50
fizzbuzz(30)    # iterate from 0 to 30
```

Function Return Values

While a function can take parameters as “input”, it can also return values to provide an output to eventually to be stored or used by functions or operating code.

```
# a function with a return value
def multiply_by_5(num):
    return num * 5

# value is stored
fifty = multiply_by_5(10)
print(fifty) # outputs `50`
```

```
# a function with no return value
def divide_by_5(num):
    print(num / 5)

# value is not stored
ten = divide_by_5(50)
print(ten) # outputs ‘None’
```

Function Return Values Cont'd

When a `return` is reached, the function returns the value and concludes the function. It will no longer execute any other code in the function. Below is an example. In addition to returning to exit the function, the return statement also returns nothing.

```
def some_func():
    return
    print('this print statement is never reached')


some_func() # will do nothing
```

Summary

1. Variables allow values to be stored and used later
2. Conditionals and loops control the flow of the program, checking whether code should execute and whether it should be repeated based on an expression that evaluates to a True or False value
3. Functions modularizes and allow code to be reused by executing according to parameter values
4. Functions can return values to their callers and provide a computed value to be used in future operations