HackerFrogs Afterschool
Python Programming Basics: Part 2

Class:
Programming (Python)

Workshop Number:
AS-PRO-PY-02

Document Version:
1.0

Special Requirements:
Completion of AS-PRO-PY-01

What We Learned Before

This workshop is the second intro class to Python programming.

During our last workshop, we learned about a few programming concepts through Python, including the following:

Hello, World!

We wrote the **Hello, World!** program in Python, which is the first program students write for any computer language.

The Print Function

```
>>> print("This is how you print text in Python!")
This is how you print text in Python!
```

We learned how to use the Python print function, which prints information to the console and is common to almost all programs written in Python.

Variables

```
>>> name = "HackerFrogs"
>>> greetings = "Hello, "
>>> print(greetings + name)
Hello, HackerFrogs
```

We learn about programming variables, named storage locations which hold specific values, which can be referenced multiple times in a program.

Data Types

```
>>> pi = 3.14
>>> number = 1337
>>> name = "shyhat"
>>> print(type(pi),type(number),type(name))
(<class 'float'>, <class 'int'>, <class 'str'>)
```

We learned about Data Types, which are sorted into different categories, including **strings** (non-numeric text), **integers** (whole numbers), and **floats** (numbers which include decimal places).

Lists

```
>>> vegetables = ["carrots", "lettuce", "beets"]
>>> print(vegetables[0])
carrots
```

We learned about lists, which are variables with one or more items associated with them. Items in the list can be referenced by a process called **list indexing**.

Basic Operators

```
>>> 7 + 2 / 1 * 3 - 2
11.0
```

The last thing we learned in the previous workshop is Python basic operators, which are used to perform arithmetic operations.

This Workshop's Topics

- Part 1: String Formatting

- Part 2: Basic String Operations

- Part 3: Conditions

This Workshop's Topics

- Part 1: String Formatting

- Part 2: Basic String Operations

- Part 3: Conditions

This Workshop's Topics

- Part 1: String Formatting

- Part 2: Basic String Operations

- Part 3: Conditions

This Workshop's Topics

- Part 1: String Formatting

- Part 2: Basic String Operations

- Part 3: Conditions

Part 1: String Formatting

```
>>> name = "Shyhat"
>>> daily_cups_of_coffee = 3
>>> print("%s drinks %d cups of coffee a day." % (name, daily_cups_of_coffee))
Shyhat drinks 3 cups of coffee a day.
```

String formatting lets us use variables in strings, allowing for dynamic print output.

String Formatting

```
>>> name = "Shyhat"
>>> daily_cups_of_coffee = 3
>>> print("%s drinks %d cups of coffee a day." % (name, daily_cups_of_coffee))
Shyhat drinks 3 cups of coffee a day.
```

When formatting in this way, we use the **%** symbol followed by a letter as a placeholder for the variable in the string.

String Formatting

```
>>> name = "Shyhat"
>>> daily_cups_of_coffee = 3
>>> print("%s drinks %d cups of coffee a day." % (name, daily_cups_of_coffee))
Shyhat drinks 3 cups of coffee a day.
```

After the string, we include the **%** symbol again, then in parentheses, we specify the names of variables in the order they appear in the string.

String Formatting

```
>>> name = "Shyhat"
>>> daily_cups_of_coffee = 3
>>> print("%s drinks %d cups of coffee a day." % (name, daily_cups_of_coffee))
Shyhat drinks 3 cups of coffee a day.
```

Note that inside of the string, we must pair the **%** symbol with a letter, depending on the data type of the variable we're inserting. **%s** for strings, **%d** for integers, and **%f** for floats.

Tuples

```
>>> related_to_bakers_tuple = ("dough", 13, "oven", "flour")
>>> print(related_to_bakers_tuple)
('dough', 13, 'oven', 'flour')
```

Before doing the exercise for this section, let's quickly go over tuples, since they appear in the exercise. In Python, tuples are very similar to lists, with one important difference:

Tuples

```
>>> related_to_bakers_tuple = ("dough", 13, "oven", "flour")
>>> print(related_to_bakers_tuple)
('dough', 13, 'oven', 'flour')
```

Whereas lists in Python can be modified during program execution (mutable), tuples are fixed, and cannot be modified during program execution (immutable).

Tuples

```
>>> my_list = ["oranges", "Shyhat", "trees"]
>>> my_tuple = (1, "bowling ball", "bees")
```

Visually, we can distinguish between lists and tuples by the use of square brackets or parentheses, respectively.

String Formatting Exercise

Let's practice using string formatting with Python at the following URL:

https://learnpython.org/en/String_Formatting

Part 2: Basic String Operations

```
>>> city = "Seattle"
>>> city.count("e")
2
```

Strings in Python are considered an object **class**, and classes in Python have special operations, called **methods**, that can be applied to those class objects.

Basic String Operations

```
city.count('t')
```

Use of methods in Python code can be identified by the name of the object, followed by a . (dot), then the name of the method, then parentheses, and any arguments the method requires inside of the parentheses.

Basic String Operations

```
len('HackerFrogs')
```

Use of functions in Python code can be identified by the name of the function, followed by parentheses, and any arguments the function requires inside of the parentheses.

Basic String Operations

```
len('HackerFrogs')
```

Note that, in both functions and methods, inclusion of parentheses are required, even if there are no arguments required for the function or method to execute.

The Len Function

```
>>> len('hotdog')
6
```

The **len** function is used to count the number of indexes contained in an object. In the case of strings, it returns the number of characters, and in the case of a list, it returns the number of items.

The Index Method

```
>>> food = "pizza"
>>> food.index("a")
4
```

The **index** method is used to locate the first instance of the argument within the object. The output is the number of characters into the string where the argument appears (counting from zero).

The Index Method

```
>>> food = "pizza"
>>> food.index("a")
4
```

Here, the output of the method is 4, because we're starting our count from zero.

The Count Method

```
>>> greeting = "Hello"
>>> greeting.count("l")
2
```

The **count** method returns the number of times the argument appears in the object. Note that we don't count from zero in this case.

String Index Slicing

```
>>> pet = "dog"
>>> print(pet[0])
d
```

Just as we can index lists to retrieve the item contained at a specific position, we can index strings, with each character in the string counting as a separate item.

String Index Slicing

```
>>> pet = "dog"
>>> print(pet[0])
d
```

Remember, when indexing in Python, counting starts from zero instead of one.

String Index Slicing

```
>>> pet = "python"
>>> print(pet[2:4])
th
```

What's more interesting is the ability to select specific slices of the string. Here, we've isolated the third and fourth letters from the string 'python'.

String Index Slicing

```
>>> pet = "python"
>>> print(pet[2:4])
th
```

Inside the brackets, we indicate the character to start from (counting from zero), then a colon, then the character that marks the stopping point (which is not returned).

String Index Slicing

```
>>> pet = "python"
>>> print(pet[2:4])
th
```

So `2:4` indicates to start from the third character, and stop at the fifth character, not including it in the output.

String Index Slicing

```
>>> bug = "centipede"
>>> bug[0:9:2]
cniee
```

If a second colon is included in index slicing, the number to the right of it indicates the number of characters to count by when slicing.

String Index Slicing

```
>>> bug = "centipede"
>>> bug[0:9:2]
cniee
```

In this output, we see that the slice starts from the first character, ends before the tenth character, and skips every second character.

String Index Slicing

```
>>> fruit = "grapes"
>>> print(fruit[-1])
s
```

The last part of string index slicing we'll cover is using negative index numbers. If a negative index number is used, the index starts counting from the end of the string instead of the beginning.

String Index Slicing

```
>>> fruit = "grapes"
>>> print(fruit[-1])
s
```

So here we're printing out the string index for “grapes” at the -1 position, which is the last letter.

String Index Slicing

```
>>> fruit = "grapes"
>>> print(fruit[::-1])
separg
```

When indexing, if we include a colon (:) without a number to its left, the first colon will have a default value of 0 (the start of the string), and the second colon will have a default value of the number of characters plus one (the end of the string).

String Index Slicing

```
>>> fruit = "grapes"
>>> print(fruit[::-1])
separg
```

Here, the whole string is in range, due to the inclusion of two colons with no numbers to the left of them, and we're stepping through the characters in reverse order, one by one (-1).

The Upper and Lower Methods

```
>>> birthday_message = "Happy Birthday!"
>>> print(birthday_message.upper())
HAPPY BIRTHDAY!
>>> print(birthday_message.lower())
happy birthday!
```

The **upper** and **lower** methods output all of the characters in uppercase or lowercase, respectively.

The Startswith and Endswith Methods

```
>>> group_name = "the HackerFrogs"
>>> group_name.startswith("the")
True
>>> group_name.endswith("Frog")
False
```

The **startswith** and **endswith** methods are used to test whether strings contain the specified argument at the start or end of the string, respectively.

The Split Method

```
>>> groceries = "eggs,milk,bread,lettuce"
>>> groceries_list = groceries.split(",")
>>> print(groceries_list)
['eggs', 'milk', 'bread', 'lettuce']
```

The **split** method converts a string into a list, with the separation between items indicated by the argument given (usually a space or a comma).

Basic String Operations Exercise

Let's practice our basic string operations in Python at the following URL:

https://learnpython.org/en/Basic_String_Operations

Boolean Data Type

```
>>> print(1==1)
True
>>> print(1==2)
False
```

In the previous workshop, we worked with **string**, **float**, and **integer** data types. Now, because we will be learning about conditional statements, we'll learn about a new data type: the **Boolean**.

Boolean Data Type

```
>>> print(1==1)
True
>>> print(1==2)
False
```

The Boolean data type can have either one of two values: **True** or **False**, and which value is returned depends on the statement that is evaluated.

Boolean Data Type

```
>>> print(1==1)
True
>>> print(1==2)
False
```

In the example above, we're using the Boolean operator == (equals) to compare 1 to 1, and 1 to 2, respectively.

Boolean Data Type

```
>>> print(1==1)
True
>>> print(1==2)
False
```

We'll be discussing Booleans more later, but for now, we need to understand that statements equating to True or False is the basis of conditional statements, which leads us to...

Part 3: Conditions

Conditions are how programs make decisions as to which instructions to perform at a specific point in program execution.

Conditions

Generally, conditions trigger when a statement evaluates to **True** or **False**, so let's discuss what this means, exactly.

Conditions

The basic conditional statement in Python is the **if** statement, and **if** statements are written in code blocks like the following:

Conditions

```
password = 'mysecretpassword'

if password == 'mysecretpassword':
    print('password correct')
else:
    print('incorrect password')
```

Here, the program will print one message if the password variable is the string 'mysecretpassword' (Boolean True), and print another message if it is not (Boolean False).

Conditions

```
password = 'mysecretpassword'

if password == 'mysecretpassword':
    print('password correct')
else:
    print('incorrect password')
```

If statements start with if, then a Boolean equation, then a semicolon.

Conditions

```
password = 'mysecretpassword'

if password == 'mysecretpassword':
    print('password correct')
else:
    print('incorrect password')
```

Then an indentation on the next line, followed by instructions if the condition is met, then on the next line else:, then an indent on the next line, followed by instructions if the condition is not met.

Conditions

```
password = 'mysecretpassword'

if password == 'mysecretpassword':
    print('password correct')
else:
    print('incorrect password')

print('Was your password correct?')
```

The **else** portion of the code block is optional. After the **if** statement code block finishes, program execution continues on the next line.

Boolean Operators

Operator	Meaning
== (double equal to)	Equal to
<	Less than
>	Greater than
!=	Not equal to
<=	Less than or equal to
>=	Greater than or equal to

We can use any of the above Boolean operators in If Statements.

The AND Operator

```
birthday = 'October 24th'
todays_date = 'October 24th'

if birthday == 'October 24th' and todays_date == 'October 24th':
  print('Happy birthday!')
```

```
Happy birthday!
```

When using the AND operator, both statements to the right and left of the operator must be True in order for the overall statement to evaluate to True.

The AND Operator

```
birthday = 'October 24th'
todays_date = 'October 24th'

if birthday == 'October 24th' and todays_date == 'October 24th':
  print('Happy birthday!')
```

```
Happy birthday!
```

In this case, since the variables **birthday** and **todays_date** are the same day (October 24th), the **if** statement returns True, and the program will print **Happy Birthday!**

The OR Operator

```
favorite_food = 'spaghetti'
hobby = 'watching movies'

if favorite_food == 'spaghetti' or hobby == 'classical painting':
    print("Let's take a vacation to Italy!")
```

```
Let's take a vacation to Italy!
```

When using the OR operator either statement to the left or right of the operator must evaluate True for the overall statement to evaluate to True.

The OR Operator

```
favorite_food = 'spaghetti'
hobby = 'watching movies'

if favorite_food == 'spaghetti' or hobby == 'classical painting':
    print("Let's take a vacation to Italy!")
```

```
Let's take a vacation to Italy!
```

In this example, the if condition would not trigger if **both** statements to the left and right of the OR operator were False.

The IN Operator

```
name = "John"
party_guests = ["John", "Sue", "Bobby"]
if name in party_guests:
    print("You're invited to the party!")
```

```
You're invited to the party!
```

The IN operator is used to check if there is a specific item within a container object, such as a list.

The IN Operator

```
name = "John"
party_guests = ["John", "Sue", "Bobby"]
if name in party_guests:
    print("You're invited to the party!")
```

```
You're invited to the party!
```

Here the print instruction is executed because the string listed in the **name** variable is included as an item in the **party_guests** list.

The IS Operator

```
x = [2, 5, 7]
y = x
z = [2, 5, 7]
print(x == z)
print(x is z)
print(x is y)
```

```
True
False
True
```

The IS operator is used to compare whether an object is exactly referencing another object, as opposed to matching the values contained within.

The NOT Operator

```
telling_the_truth = False

if telling_the_truth == (not False):
  print("You're telling the truth!")
else:
  print("You're lying!")
```

```
You're lying!
```

The NOT operator is essentially an inversion of the whatever Boolean value is paired with it.

The NOT Operator

```
telling_the_truth = False

if telling_the_truth == (not False):
  print("You're telling the truth!")
else:
  print("You're lying!")
```

```
You're lying!
```

NOT paired with True evaluates to False. And NOT paired with False evaluates to True.

Functions Exercise

Let's practice our if conditionals in Python at the following URL:

https://learnpython.org/en/Conditions

Workshop Review Exercise

Let's write a program that uses some of the concepts we learned in this workshop.

We should write the program in one of the Python windows on the learnpython.org webpage.

The program should include one variable that holds the value of an integer (whole number).

Workshop Review Exercise

The program should print out the string

X is an even number!
if the variable is an even number, or

X is an odd number!
if the variable is an odd number.

The value of the variable should substitute in for X in the printed statement. E.g., '7 is an odd number!'

Workshop Review Exercise

The program should be run three times, with the value of the variable set as 0, 13, and 256 for each execution, respectively.

The program should report 0 and 256 as even numbers, and 13 as an odd number.

Workshop Review Exercise

We'll present a possible solution after this, but you should attempt to write your own program now.

Workshop Review Exercise

- a variable named **number**

- an if statement that prints out '**X is an even number!**' if the **number** variable is an even number

- an if statement that prints out '**X is an odd number!**' if the **number** variable is an odd number

- run the program three times, with the **number** variable set to 0, 13, and 256, respectively

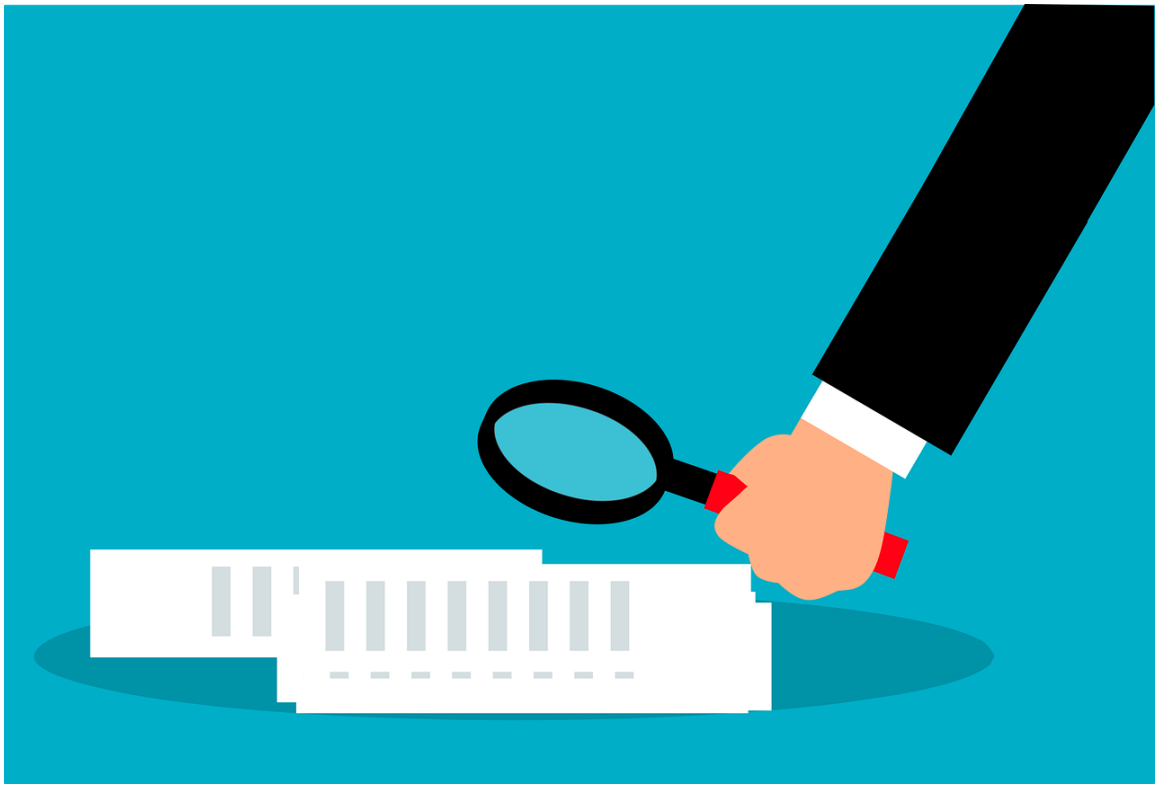
Workshop Review Exercise

A possible solution to this exercise is the following:

```
number = 0

if (number % 2) == 0:
  print('%s is an even number!' % number)
if (number % 2) == 1:
  print('%s is an odd number!' % number)
```

Summary



Let's review the programming concepts we learned in this workshop:

String Formatting

```
>>> name = "Shyhat"
>>> daily_cups_of_coffee = 3
>>> print("%s drinks %d cups of coffee a day." % (name, daily_cups_of_coffee))
Shyhat drinks 3 cups of coffee a day.
```

String formatting is a way for us to include variable values in strings by using C-style syntax.

Basic String Operations

```
>>> city = "Seattle"
>>> city.count("e")
2
```

Strings in Python have a number of different functions and methods that can be useful for obtaining specific details about the strings.

Basic String Operations

```
>>> bug = "centipede"
>>> bug[0:9:2]
cniee
```

We can also output slices of strings through string indexing.

Conditions

```
password = 'mysecretpassword'

if password == 'mysecretpassword':
    print('password correct')
else:
    print('incorrect password')
```

Conditions are used in programs to determine a program's course of action. They rely on the evaluation of Boolean statements to determine if certain instructions are performed or not.

What's Next?

In the next HackerFrogs Afterschool programming workshop, we'll continue learning Python with the learnpython.org website.

What's Next?

Next workshop topics:

- Loops

- Functions

- Classes and Objects

What's Next?

Next workshop topics:

- Loops

- Functions

- Classes and Objects

What's Next?

Next workshop topics:

- Loops

- Functions

- Classes and Objects

What's Next?

Next workshop topics:

- Loops

- Functions

- Classes and Objects

Extra Credit

Looking for more study material on this workshop's topics?

See this video's description for links to supplemental documents and exercises!

Until Next Time, HackerFrogs!

Part 2: Exercise

Line 32 – String should consist of 3 words
Line 29 – String should end with **ome!**
Line 26 – String should start with **Str**

Current guess:

Strings XXX awesome!