Additional Exercises for Python Non-Programmers

The following is an excerpt from the book: The Python Workshop - Second Edition: Write Python code to solve challenging real-world problems 2nd ed. Edition

Published by Packt> available from Amazon and the publisher: https://www.packtpub.com/en-us/product/the-python-workshop-second-edition-9781804610619

Exercise 1 – getting to know the order of operations

The goal of this exercise is to work with the primary math operations in Python and understand their order of execution. This exercise can be performed on the Python terminal, your Jupyter Notebook, or any other IDE:

1. Subtract 5 to the 3rd power from 100 and divide the result by 5:

```
(100 - 5 ** 3) / 5
```

The output is as follows:

```
–5.0
```

2. Add 6 to the remainder of 15 divided by 4:

```
6 + 15 % 4
```

The output is as follows:

```
9
```

3. Add 2 to the 2nd power, which is 4, to the integer division of 24 and 4:

```
2 ** 2 + 24 // 4
```

The output is as follows:

```
10
```

In this exercise you used Python to perform basic math using the order of operations.

Exercise 2 – integer and float types

1. Begin by explicitly determining the type of 6 using the following code:

type(6)

The output is as follows:

int

2. Now, enter type(6.0) in the next cell of your notebook:

type(6.0)

The output is as follows:

float

3. Now, add 5 to 3.14. Infer the type of their sum:

5 + 3.14

The output is as follows:

8.14

It's clear from the output that combining an int and a float gives us a float. This makes sense. If Python returned 8, you would lose information. When possible, Python converts types to preserve information.

You can, however, change types by using the type keyword. 4. Now, convert 7.999999999 into an int:

int(7.999999999)

The output is as follows:

7

5. Convert 6 into a float:

float(6)

The output is as follows:

6.0

In this exercise, you determined types using the type() keyword, and you changed types between integers and floats.

Exercise 3 – assigning variables

The goal of this exercise is to assign values to variables.

1. Set x equal to the number 2, then add 1 to the x variable:

11

x = 2

x + 1

The output is as follows:

3

Once we add 1 to x, we get the output of 3, because the x variable has had 1 added to it. 2. Check the value of x by entering x in a coding cell and running the code:

x

The output is as follows:

2

Note that the value of x has not changed.

3. Change x to 3.0, set x equal to x+1, and check the value of x:

x = 3.0 x = x + 1 x

The output is as follows:

4.0

In this step, we changed the value of x by setting x equal to x+1. This is permissible in Python because of the right to left order in variable assignment. On the right-hand side, x+1 has a value of 4.0; this value may be assigned to any variable, including x.

Incrementing a variable by 1 is so common in computer programming that there's a shortcut for it.

Instead of writing

x=x+1,

you can write

x+=1

Exercise 4 – naming variables

The goal of this exercise is to learn standard ways to name variables by considering good and bad practices. This exercise can be performed in a Jupyter Notebook:

1. Create a variable called 1st_number and assign it a value of 1:

1st_number = 1

Let's see the output:

Figure 1.4 – Output throwing a syntax error

You'll get the error mentioned in the preceding screenshot because you cannot begin a variable with a number.

2. Now, let's try using letters to begin a variable:

first_number = 1

3. Now, use special characters in a variable name, as in the following code:

my_$ = 1000.00

The output is as follows:

Exercise 4 – naming variables

The goal of this exercise is to learn standard ways to name variables by considering good and bad practices. This exercise can be performed in a Jupyter Notebook:

1. Create a variable called 1st_number and assign it a value of 1:

1st_number = 1

Let's see the output:

Figure 1.4 – Output throwing a syntax error

You'll get the error mentioned in the preceding screenshot because you cannot begin a variable with a number.

2. Now, let's try using letters to begin a variable:

first_number = 1

3. Now, use special characters in a variable name, as in the following code:

my_$ = 1000.00

File "<python-input-0>", line 1my_$ = 1000.00

You'll receive a error:

>> my_$ = 1000.00

File "<python-input-1>", line 1
my_$ = 1000.00
^
SyntaxError: invalid syntax

4. Now, use letters again instead of special characters for the variable name:

my_money = 1000.00

In this exercise, you learned how to use underscores to separate words when naming variables, and not to start variables' names with numbers or include any symbols.

Exercise 5 – assigning multiple variables

In this exercise, you will perform mathematical operations using more than one variable. This exercise can be performed in your Jupyter Notebook:

1. Assign 5 to x and 2 to y:

x = 5

y = 2

2. Add x to x and subtract y to the second power:

x + x - y ** 2

This will give you the following output:

6

Python has a lot of cool shortcuts, and multiple variable assignment is one of them. Let's look at the Pythonic way of declaring two variables.

3. Assign 8 to x and 5 to y in one line:

```
x, y = 8, 5
```

4. Find the integer division of x and y:

```
x // y
```

This will give you the following output:

```
1
```

In this exercise, you practiced working with multiple variables, and you even learned the Pythonic way to assign values to multiple variables in one line.

Exercise 6 – comments in Python

In this exercise, you will learn two different ways to display comments in Python. This exercise can be performed in a Jupyter Notebook:

1. Write a comment that states This is a comment:

```
# This is a comment
```

When you execute this cell, nothing should happen.

2. Set the pi variable equal to 3.14. Add a comment above the line stating what you did:

```
# Set the variable pi equal to 3.14 pi = 3.14
```

Adding this comment clarifies what follows.

3. Now, try setting the pi variable equal to 3.14 again, but add the comment stating what you did on the same line:

```
pi = 3.14  # Set the variable pi equal to 3.14
```

Although it's less common to provide comments on the same line of code, it's acceptable and often appropriate.

Docstrings

Docstrings, short for document strings, state what a given document, such as a program, a function, or a class, actually does. The primary difference in syntax between a docstring and a

comment is that docstrings are intended to be written over multiple lines, which can be accomplished with triple quotes, """. They also introduce a given document, so they are placed at the top.

Here is an example of a docstring:

"""

This document will explore why comments are particularly useful when writing and reading code.

"""

Docstrings, like comments, are designed as information for developers reading and writing code.

Docstrings

Docstrings, short for document strings, state what a given document, such as a program, a function, or a class, actually does. The primary difference in syntax between a docstring and a comment is that docstrings are intended to be written over multiple lines, which can be accomplished with triple quotes, """. They also introduce a given document, so they are placed at the top.

Here is an example of a docstring:

"""

This document will explore why comments are particularly useful when writing and reading code.

"""

Docstrings, like comments, are designed as information for developers reading and writing code.

Exercise 7 – string error syntax

The goal of this exercise is to learn appropriate string syntax: 1. Open a Jupyter Notebook.

2. Enter a valid string:

bookstore = 'City Lights'

3. Now, enter an invalid string:

bookstore = 'City Lights"

The output is as follows:

```
File "<python-input-2>", line 1
  bookstore = 'City Lights"
              ^
SyntaxError: unterminated string literal (detected at line 1)
```

If you start with a single quote, you must end with a single quote. Since the string has not been completed, you receive a syntax error.

4. Now, you need to enter a valid string format again, as shown in the following code snippet:

bookstore = "Moe's"

This is okay. The string starts and ends with double quotes. Anything can be inside the quotation marks, except for more of the same quotation marks.

5. Now, add an invalid string again:

bookstore = 'Moe's'

Let’s look at the output:

```
File "<python-input-3>", line 1
  bookstore = 'Moe's'
                  ^
SyntaxError: unterminated string literal (detected at line 1)
```

This is a problem. You started and ended with single quotes, and then you added another single quote and an s.

This is a problem. You started and ended with single quotes, and then you added another single quote and an s.

A couple of questions arise. The first is whether single or double quotes should be used. The answer is that it depends on developer preference. Double quotes are more traditional, and they can be used to avoid potentially problematic situations such as the aforementioned Moe's example. Single quotes save space and require one less keystroke.

A couple of questions arise. The first is whether single or double quotes should be used. The answer is that it depends on developer preference. Double quotes are more traditional, and they can be used to avoid potentially problematic situations such as the aforementioned Moe's example. Single quotes save space and require one less keystroke.

Exercise 8 – displaying strings

In this exercise, you will learn about different ways to display strings:

1. Open a new Jupyter Notebook.

2. Define a greeting variable with a value of 'Hello'. Display the greeting using the print() function:

greeting = 'Hello' print(greeting)

The output is as follows:

Hello

Hello, as shown in the display, does not include single quotes. This is because the print() function is generally intended for the user to print the output.

3. Display the value of greeting without using the print() function in a Jupyter Notebook:

greeting

The output is as follows:

’Hello’

When we input greeting without the print() function, we are obtaining the encoded value, hence the quotes.

4. Consider the following sequence of code in a single cell in a Jupyter Notebook:

spanish_greeting = 'Hola.'

spanish_greeting

arabic_greeting = 'Ahlan wa sahlan.'

When the preceding cell is run, the output does not display spanish_greeting. If the code were run as three separate cells, it would display Hola., the string assigned to spanish_greeting. For consistency, it’s useful to use print() any time information should be displayed.

5. Now, display the Arabic greeting message defined in Step 4 using the global print() function:

print(arabic_greeting)

We will see the following output:

Ahlan wa sahlan.

In this exercise, you learned different ways to display strings, including using the print() function. You will use the print() function very frequently. The output is as follows:

Hello

Hello, as shown in the display, does not include single quotes. This is because the print() function is generally intended for the user to print the output.

3. Display the value of greeting without using the print() function in a Jupyter Notebook:

greeting

The output is as follows:

'Hello'

When we input greeting without the print() function, we are obtaining the encoded value, hence the quotes.

4. Consider the following sequence of code in a single cell in a Jupyter Notebook:

spanish_greeting = 'Hola.'

spanish_greeting

arabic_greeting = 'Ahlan wa sahlan.'

When the preceding cell is run, the output does not display spanish_greeting. If the code were run as three separate cells, it would display Hola., the string assigned to spanish_greeting. For consistency, it's useful to use print() any time information should be displayed.

5. Now, display the Arabic greeting message defined in Step 4 using the global print() function:

print(arabic_greeting)

We will see the following output:

Ahlan wa sahlan.

In this exercise, you learned different ways to display strings, including using the print() function. You will use the print() function very frequently.

Exercise 9 – string concatenation

In this exercise, you will learn how to combine strings using string concatenation:

1. Open a new Jupyter Notebook.

2. Combine spanish_greeting, which we used in Exercise 8 – displaying strings, with 'Senor.' using the + operator and display the results:

23

spanish_greeting = 'Hola' print(spanish_greeting + 'Senor.')

The output is as follows:

HolaSenor.

Notice that there are no spaces between the greeting and the name. If we want spaces between strings, we need to explicitly add them.

3. Now, combine spanish_greeting with 'Senor.' using the + operator, but this time, include a space:

spanish_greeting = 'Hola ' print(spanish_greeting + ' Senor.')

The output is as follows:

Hola Senor.

4. Display the greeting variable five times using the * multiplication operator:

greeting = 'Hello'

print(greeting * 5)

The output is as follows:

HelloHelloHelloHelloHello

By completing this exercise successfully, you have concatenated strings using the + and * operators.

Comma separators

Variables may be interpolated into strings using commas to separate clauses. It's similar to the + operator, except it adds spacing for you.

Look at the following example, where we add Ciao within a print statement:

italian_greeting = 'Ciao'

print('Should we greet people with', italian_greeting, 'in North Beach?')

The output is as follows:

Should we greet people with Ciao in North Beach?

f-strings

Perhaps the most effective way to combine variables with strings is with f-strings. Introduced in Python 3.6, f-strings are activated whenever the f character is followed by quotations. The advantage is that any variable inside curly brackets will automatically be converted into a string.

Here is an example:

poet = 'Amanda Gorman'

age = 22

f'At age {age}, {poet} became the youngest inaugural poet in US history.'

The output is as follows:

'At age 22, Amanda Gorman became the youngest inaugural poet in US history.'

Exercise 10 – implementing string methods

In this exercise, you will learn how to implement string methods: 1. Set a new variable, called name, to any name that you like:

name = 'Josephine'

Note

In Jupyter Notebooks, you can access string methods by pressing the Tab button after the variable name and dot (.).

You can scroll down the list to obtain all available string methods.

2. Now, convert the name variable into lowercase letters using the lower() method:

name.lower()

The output is as follows:

’josephine’

3. Now, capitalize the name variable using the capitalize() method:

name.capitalize()

The output is as follows:

’Josephine’

4. Convert the name variable into uppercase letters using upper():

name.upper()

The output is as follows:

’JOSEPHINE’

5. Finally, count the number of e instances in the name variable:

name.count('e')

The output is as follows for name=Josephine:

2

In this exercise, you learned about a variety of string methods, including lower(), capitalize(), upper(), and count().

Methods may only be applied to their representative types. For instance, the lower() method only works on strings, not integers or floats. By contrast, global functions such as len() and print() can be applied to a variety of types.

Exercise 11 – types and casting

In this exercise, you will learn how types and casting work together: 1. Open a new Jupyter Notebook.

2. Determine the type of '5':

type('5')

The output is as follows:

str

3. Now, add '5' and '7':

'5' + '7'

The output is as follows:

’57’

The answer is not 12 because, here, 5 and 7 are of the string type, not of the int type. Recall that the + operator concatenates strings. If we want to add 5 and 7, we must convert them first.

4. Convert the '5' string into an int using the following line of code:

int('5')

The output is as follows:

5

Now that 5 is a number, it can be combined with other numbers via standard mathematical operations.

5. Add '5' and '7' by converting them into the int type first:

int('5') + int('7')

The output is as follows:

12

In this exercise, you learned how strings may be cast as ints, and how ints may be cast as strings via a general procedure that will work for all transferable types.

Exercise 12 – using the input() function

In this exercise, you will utilize the input() function to obtain information from the user. Note that in computer programming, the user generally refers to the person or entity using the program that you are writing:

1. Open a new Jupyter Notebook.

2. Ask a user for their name using the input() function, as follows:

name = input('What is your name?')

The output is as follows:

What is your name?

The text following input is displayed, and the computer is waiting for the user's response. Enter a response and then press Enter:

Now that a name has been provided to the input() function, it has been stored as the name variable and may be referenced later.

Access the provided name using the global input() function, as follows:

print(f'Hello, {name}.')

The output will be as follows:

Hello, Alenna.

input() can be finicky in Jupyter Notebooks. If an error arises when you're entering the code, try restarting the kernel using the Runtime: Restart Runtime tab. Restarting the kernel will erase the current memory and start each cell afresh. This is advisable if the notebook stalls.

String indexing and slicing

Indexing and slicing are crucial parts of programming. Indexing and slicing are regularly used in lists, a topic that we will cover in Chapter 2, Python Data Structures. In data analysis, indexing and slicing DataFrames is essential to keep track of rows and columns, something you will practice in Chapter 10, Data Analytics with pandas and NumPy.

Indexing

The characters in strings exist in specific locations. In other words, their order counts. The index is a numerical representation of where each character is located. The first character is at index 0, the second character is at index 1, the third character is at index 2, and so on.

Consider the following string:

destination = 'San Francisco'

'S' is in the 0th index, 'a' is in the 1st index, 'n' is in the 2nd index, and so on.

The characters of each index are accessed using bracket notation, as follows:

destination[0]

The output is as follows:

’S’

To access the data from the first index, enter the following:

destination[1]

The output is as follows:

’a’

Now, try adding -1 as the index value to access the last character of the string:

destination[-1]

The output is as follows:

’o’

Negative numbers start at the end of the string. (It makes sense to start with -1 since -0 is the same as 0.)

To access the data from the end of any string, always use the negative sign:

destination[-2]

The output is as follows:

’c’

Slicing

A slice is a subset of a string or other element. A slice could be the whole element or one character, but it’s more commonly a group of adjoining characters.

Let’s say you want to access the fifth through eleventh letters of a string. So, you start at index 4 and end at index 10, as was explained in the Indexing section. When slicing, the colon symbol (:) is inserted between indices, like so: [4:10].

There is one caveat: the lower bound of a slice is always included, but the upper bound is not. So, in the preceding example, if you want to include the 10th index, you must use [4:11].

Now, let’s have a look at the following example for slicing.

Retrieve the fifth through eleventh letters of the destination variable, which you used in the Indexing section:

destination[4:11]

The output is as follows:

'Francis'

Retrieve the first three letters of destination:

destination[0:3]

The output is as follows:

'San'

There is a shortcut for getting the first n letters of a string. If the first numerical character is omitted, Python will start at the 0th index.

Now, to retrieve the first eight letters of destination using the shortcut, use the following code:

destination[:8]

The output is as follows

'San Fran'

Finally, to retrieve the last nine letters of destination, use the following code:

destination[-9:]

The output is as follows:

'Francisco'

The negative sign, -, means that we start at the end. So, the -9 parameter means start at the ninth- to-last letter, and the colon means we end at the last letter.

Exercise 13 – Boolean variables

In this short exercise, you will use, assign, and check the types of Boolean variables:

1. Open a new Jupyter Notebook.
2. Now, use a Boolean to classify someone as being over 18 using the following code snippet:

over_18 = True

type(over_18)

The output is as follows:

bool

The given output is bool, which is short for Boolean.

Exercise 14 – comparison operators

In this exercise, you will practice using comparison operators. You will start with some basic mathematical examples:

1. Open a new Jupyter Notebook.

2. Now, set income equal to 80000 and include a comparison operator to check whether income is less than 75000:

income = 80000

income < 75000

The output is as follows:

False

3. Using the following code snippet, you can check whether income is greater than or equal to 80000 and less than or equal to 100000:

income >= 80000 and income <= 100000

The output is as follows:

True

4. Now, check whether income is not equivalent to 10000:

income != 100000

The output is as follows:

True

5. Now, check whether income is equivalent to 90000:

income == 90000

The output is as follows:

False

The double equals sign, or the equivalent operator, ==, is very important in Python. It allows us to determine whether two objects are equal. You can now address the question of whether 6 and 6.0 are the same in Python.

6. Is 6 equivalent to 6.0 in Python? Let's find out:

```
6 == 6.0
```

The output is as follows:

```
True
```

This may come as a bit of a surprise. 6 and 6.0 are different types, but they are equivalent. Why would that be?

Since 6 and 6.0 are equivalent mathematically, it makes sense that they would be equivalent in Python, even though the types are different. Consider whether 6 should be equivalent to 42/7. The mathematical answer is yes. Python often conforms to mathematical truths, even with integer division. From this, you can conclude that different types can have equivalent objects.

7. Now, find out whether 6 is equivalent to the '6' string:

```
6 == '6'
```

This will result in the following output:

```
False
```

Different types usually do not have equivalent objects. In general, it's a good idea to cast objects as the same type before testing for equivalence.

8. Next, let's find out whether someone who is 29 is in their 20s or 30s:

```
age=29
(20 <= age < 30) or (30 <= age < 40)
```

Now, the output will be as follows:

```
True
```

Although the parentheses in the preceding code line are not strictly required, they make the code more readable. A good rule of thumb is to use parentheses for clarity. When using more than two conditions, parentheses are generally a good idea.

Exercise 15 – practicing comparing strings

In this exercise, you will be comparing strings using Python: 1. Open a new Jupyter Notebook.

2. Let's compare single letters:

39

'a' < 'c'

Let's see the output:

True

3. Now, let's compare 'New York' and 'San Francisco':

'New York' > 'San Francisco'

Now, the output changes:

False

This is False because 'New York' < 'San Francisco'. 'New York' does not come later in the dictionary than 'San Francisco'.

In this exercise, you learned how to compare strings using comparison operators.

Exercise 16 – using the if syntax

In this exercise, you will be using conditionals using the if clause:

1. Open a new Jupyter Notebook.

2. Now, run multiple lines of code where you set the age variable to 20 and add an if clause, as mentioned in the following code snippet:

age = 20

if age >= 18 and age < 21:

print('At least you can vote.') print('US Poker will have to wait.')

The output is as follows:

At least you can vote.

US Poker will have to wait.

There is no limit to the number of indented statements. Each statement will run in order, provided that the preceding condition is True.

3. Now, use nested conditionals:

```
if age >= 18:

print('You can vote.')

if age >= 21:

print('You can play poker in the US.')
```

The output is now as follows:

```
You can vote.
```

In this case, it's true that age >= 18, so the first statement prints You can vote. The second condition, age>= 21, however, is false, so the second statement does not get printed.

In this exercise, you learned how to use conditionals using the if clause. Conditionals will always start with if.

Exercise 17 – using the if-else syntax

In this exercise, you will learn how to use conditionals that have two options – one following if, and one following else:

1. Open a new Jupyter Notebook.
2. Introduce a voting program only to users over 18 by using the following code snippet:

```
age = 20

if age < 18:

print('You aren\'t old enough to vote.')

else:

print('Welcome to our voting program.')
```

The output will be as follows:

```
Welcome to our voting program.
```

Note

Everything after else is indented, just like everything after the if loop.

The elif statement

elif is short for else if. elif does not have meaning in isolation. elif appears in between an if and else clause. Have a look at the following code snippet and copy it into your Jupyter notebook. The explanation is mentioned after the output:

```
if age <= 10:

print('Listen, learn, and have fun.') elif age<= 19:

print('Go fearlessly forward.')

elif age <= 29:

print('Seize the day.')

elif age <= 39:

print('Go for what you want.')

elif age <= 59:

print('Stay physically fit and healthy.') else:

print('Each day is magical.')
```

The output is as follows:

```
Seize the day.
```

Now, let's break down the code for a better explanation:

1. The first line checks if age is less than or equal to 10. Since this condition is false, the next branch is checked.

2. The next branch is elif age <= 19. This line checks if the specified age is less than or equal to 19. This is also not true, so we move to the next branch.

3. The next branch is elif age <= 29. This is true since age = 20. The indented statement that follows will be executed.

4. Once any branch has been executed, the entire sequence is aborted, and none of the subsequent elif or else branches are checked.

5. If none of the if or elif branches were true, the final else branch will automatically be executed.

Exercise 18 – calculating perfect squares

The goal of this exercise is to prompt the user to enter a given number and find out whether it is a perfect square without using square roots.

The following steps will help you with this:

1. Open a new Jupyter Notebook.
2. Prompt the user to enter a number to see if it's a perfect square:

47

```
print('Enter a number to see if it\'s a perfect square.')
```

3. Set a variable equal to input(). In this case, let's enter 64:

```
number = input()
```

4. Ensure the user input is a positive integer:

```
number = abs(int(number))
```

5. Choose an iterator variable:

```
i = -1
```

6. Initialize a Boolean to check for a perfect square:

```
square = False
```

7. Initialize a while loop from -1 to the square root of the number:

```
while i <= number:
```

8. Increment i by 1:

```
i += 1
```

9. Check the square root of number:

```
if i*i == number:
```

10. Indicate that we have a perfect square:

```
square = True
```

11. break out of the loop:

```
break
```

12. If the number is square, print out the result:

if square:

print('The square root of', number, 'is', i, '.')

13. If the number is not a square, print out this result:

else:

print('', number, 'is not a perfect square.')

The output is as follows:

The square root of 64 is 8.

In this exercise, you wrote a program to check whether the user's number is a perfect square.

Exercise 19 – real estate offer

The goal of this exercise is to prompt the user to bid on a house and let them know if and when the bid has been accepted.

The following steps will help you with this: 1. Open a new Jupyter Notebook.

2. Begin by stating a market price:

print('A one bedroom in the Bay Area is listed at $599,000.')

3. Prompt the user to make an offer on the house using input() and convert it into an integer:

offer = int(input('Enter your first offer on the house.'))

4. Prompt the user to enter their highest offer for the house:

highest = int(input('Enter your highest offer on the house.'))

5. Prompt the user to choose increments:

increment = int(input('How much more do you want to offer each time if each time your offer is rejected ?'))

6. Set offer_accepted equal to False:

offer_accepted = False

7. Initialize the for loop from offer to best:

while offer <= best:

8. If offer is greater than 650000, they get the house:

```
if offer >= 650000:
offer_accepted = True
print('Your offer of', offer, 'has been accepted!')
break
```

9. If offer does not exceed 650000, they don't get the house:

```
print(f'We\'re sorry, your offer of {offer} has not been accepted.' )
```

10. Add increment to offer:

```
offer += increment
```

The output is as follows:

```
A one bedroom in the Bay Area is listed at $599,000
Enter your first offer on the house.
600000
Enter your best offer on the house.
690000
How much more do you want to offer each time?
10000
We're sorry, you're offer of 600000 has not been accepted.
We're sorry, you're offer of 610000 has not been accepted.
We're sorry, you're offer of 620000 has not been accepted.
We're sorry, you're offer of 630000 has not been accepted.
We're sorry, you're offer of 640000 has not been accepted.
Your offer of 650000 has been accepted!
```

Exercise 20 – using for loops

In this exercise, you will utilize for loops to print the characters in a string, in addition to a range of numbers:

1. Open a new Jupyter Notebook.
2. Print out the characters of 'Amazing':

for i in 'Amazing': print(i)

The output is as follows:

A
m
a
z
i
n
g

The for keyword often goes with the in keyword. The i variable is known as a dummy variable. The for i in phrase means that Python is going to check what comes next and look at its components. Strings are composed of characters, so Python will do something with each of the individual characters. In this particular case, Python will print out the individual characters, as per the print(i) command. What if we want to do something with a range of numbers? Can for loops be used for that? Absolutely. Python provides another keyword, range, to access a range of numbers. range is often defined by two numbers – the first number and the last number – and it includes all numbers in between. Interestingly, the output of range includes the first number, but not the last number.

In the next step, you will use range to display the first 9 numbers:

1. Use a lower bound of 1 and an upper bound of 10 with range to print 1 to 9, as follows:

51

for i in range(1,10): print(i)

The output is as follows:

1
2
3
4
5
6
7
8
9

range does not print the number 10.

2. Now, use range with one bound only, the number 10, to print the first 10 numbers:

```
for i in range(10): print(i)
```

The output is as follows:

```
0
1
2
3
4
5
6
7
8
9
```

So, range(10) will print out the first 10 numbers, starting at 0, and ending with 9. By default, range will start with 0, and it will include the number of values provided in parenthesis. Now, let's say that you want to count by increments of 2. You can add a third bound, a step increment, to count up or down by any number desired.

3. Use a step increment to count the odd numbers through 10:

```
for i in range(1, 11, 2): print(i)
```

The output is as follows:

```
1
3
5
7
9
```

Similarly, you can count down using negative numbers, which is shown in the next step.

4. Use a negative step increment to count down from 3 to 1:

```
for i in range(3, 0, -1): print(i)
```

The output is as follows:

```
3
2
1
```

And, of course, you can use nested loops, which are shown in the next step.

5. Now, print each letter of your name three times:

name = 'Alenna' for i in range(3):

for i in name: print(i+'!')

The output is as follows:

A
l
e
n
n
a
!
A
l
e
n
n
a
!
A
l
e
n
n
a
!

In this exercise, you utilized loops to print any given number of integers and characters in a string.

Exercise 21 – basic list operations

In this exercise, you are going to use the basic functions of lists to check the size of a list, combine lists, and duplicate lists:

1. Open a new Jupyter notebook.

2. Type the following code:

61

shopping = ["bread","milk", "eggs"]

3. The length of a list can be found using the global len() function:

print(len(shopping))

Note

The len() function returns the number of items in an object. When the object is a string, it returns the number of characters in the string.

The output is as follows:

3

4. Now, concatenate two lists using the + operator:

list1 = [1,2,3]

list2 = [4,5,6]

final_list = list1 + list2 print(final_list)

You will get the following output:

[1, 2, 3, 4, 5, 6]

As you can see, lists also support many string operations, one of which is concatenation, which involves joining two or more lists together.

5. Now, use the * operator, which can be used for repetition in a list, to duplicate elements:

```
list3 = ['oi']
print(list3*3)
```

It will repeat 'oi' three times, giving us the following output:

['oi', 'oi', 'oi']

Exercise 23 – accessing an item from shopping list data

In this exercise, you will work with lists and gain an understanding of how you can access items from a list. The following steps will enable you to complete this exercise:

1. Open a new Jupyter Notebook.

2. Enter the following code in a new cell:

shopping = ["bread","milk", "eggs"] print(shopping[1])

The output is as follows:

milk

As you can see, the milk value from the shopping list has an index of 1 since the list begins from 0.

3. Now, access the first index and replace it with banana:

shopping[1] = "banana" print(shopping)

The output is as follows:

['bread', 'banana', 'eggs']

4. Type the following code in a new cell and observe the output:

print(shopping[-1])

The output is as follows:

eggs

The output will print eggs – the last item.

Just like with strings, Python lists support slicing with the : notation in the format of list[i:j], where i is the starting element and j is the last element (non-inclusive).

5. Enter the following code to try out a different type of slicing:

print(shopping[0:2])

This prints the first and second elements, producing the following output:

['bread', 'banana']

6. Now, to print from the beginning of the list to the third element, run the following:

print(shopping[:3])

The output is as follows:

['bread', 'banana', 'eggs']

7. Similarly, to print from the second element of the list until the end, you can use the following:

print(shopping[1:])

The output is as follows:

['banana', 'eggs']

Having completed this exercise, you are now able to access items from a list in different ways.

Exercise 22 – adding items to our shopping list

The append method is the easiest way to add a new element to the end of a list. You will use this method in this exercise to add items to our shopping list:

1. In a new cell, type the following code to add a new element, apple, to the end of the list using the append method:

shopping = ["bread","milk", "eggs"] shopping.append("apple") print(shopping)

Let's see the output:

['bread', 'milk', 'eggs', 'apple']

The append method is commonly used when you are building a list without knowing what the total number of elements will be. You will start with an empty list and continue to add items to build the list.

2. Now, create an empty list, shopping, and keep adding items one by one to this empty list:

shopping = []

shopping.append('bread') shopping.append('milk') shopping.append('eggs') shopping.append('apple') print(shopping)

Here's the output:

['bread', 'milk', 'eggs', 'apple']

This way, you start by initializing an empty list, and you extend the list dynamically. The result is the same as the list from the previous code. This is different from some programming languages, which require the array size to be fixed at the declaration stage.

3. Now, use the insert method to add elements to the shopping list:

shopping.insert(2, 'ham') print(shopping)

The output is as follows:

['bread', 'milk', 'ham', 'eggs', 'apple']

As you coded in Step 3, you came across another way to add an element to a list: using the insert method. The insert method requires a positional index to indicate where the new element should be placed. A positional index is a zero-based number that indicates the position in a list. You can use ham to insert an item in the third position.

In the preceding code, you can see that ham has been inserted in the third position and shifts every other item one position to the right.

Exercise 23 – looping through a list

It's common to generate new lists by looping through previous lists. In the following exercise, you will loop through a list of the first 5 primes to generate a list of the squares of the first 5 primes:

1. In a new cell, enter the first 5 primes in a list called primes.

65

primes = [2, 3, 5, 7, 11]

2. Now create an empty list, primes_squared, then loop through the primes list and append the square of each prime, as follows:

primes_squared = []

for i in primes:

primes_squared.append(i**2) print(primes_squared)

The output is as follows:

[4, 9, 25, 49, 121]

This is the standard way to loop through lists to generate new lists.

Exercise 24 – using a nested list to store data from a matrix

In this exercise, you will look at working with a nested list, storing values in it, and accessing it using several methods:

1. Open a new Jupyter notebook.

2. Enter the following code in a new cell:

m = [[1, 2, 3], [4, 5, 6]]

We can store the matrix as a series of lists inside a list, which is called a nested list. We can now access the elements using the [row][column] variable notation.

3. Print the element indexed as the first row and first column:

print(m[1][1])

The output is as follows:

5

It prints the value of row 2, column 2, which is 5 (remember, we are using a zero-based index offset).

4. Now, access each of the elements in the nested list matrix by retaining their reference index with two variables, i and j:

for i in range(len(m)): for j in range(len(m[i])):

print(m[i][j])

The preceding code uses a for loop to iterate twice. In the outer loop (i), we iterate every single row in the m matrix, and in the inner loop (j), we iterate every column in the row. Finally, we print the element in the corresponding position.

The output is as follows:

1
2
3
4
5
6
67

5. Use two for..in loops to print all the elements within the matrix:

for row in m: for col in row:

print(col)

The for loop in the preceding code iterates both row and col. This type of notation does not require us to have prior knowledge of the matrix's dimensions.

The output is as follows:

1
2
3
4
5
6

Exercise 25 – using a dictionary to store a movie record

In this exercise, you will be working with a dictionary to store movie records, and you will also try and access the information in the dictionary using a key. The following steps will enable you to complete this exercise:

1. Open a Jupyter Notebook.

2. Enter the following code in a blank cell:

```
movie = {
"title": "The Godfather",
"director": "Francis Ford Coppola", "year": 1972,
"rating": 9.2
}
```

Here, you have created a movie dictionary with a few details, such as title, director, year, and rating.

3. Access the information from the dictionary by using a key. For instance, you can use 'year' to find out when the movie was first released using bracket notation:

```
print(movie['year'])
```

Here's the output:

```
1972
```

4. Now, update the dictionary value:

```
movie['rating'] = (movie['rating'] + 9.3)/2 print(movie['rating'])
```

The output is as follows:

9.25

As you can see, a dictionary's values can also be updated in place.

5. Construct a movie dictionary from scratch and extend it using key-value assignment:

```
movie = {}
movie['title'] = "The Godfather"
movie['director'] = "Francis Ford Coppola" movie['year'] = 1972
movie['rating'] = 9.2
```

As you may have noticed, similar to a list, a dictionary is flexible in terms of size.

6. You can also store a list inside a dictionary and store a dictionary within that dictionary:

```
movie['actors'] = ['Marlon Brando', 'Al Pacino', 'James Caan']
movie['other_details'] = { 'runtime': 175,
'language': 'English'
}
print(movie)
```

Exercise 26 – accessing a dictionary using dictionary methods

In this exercise, you will learn how to access a dictionary using dictionary methods. The goal of this exercise is to print the order values against the item while accessing dictionary methods:

1. Open a new Jupyter Notebook.
2. Enter the following code in a new cell:

```
album_sales = {'barbara':150, 'aretha':75, 'madonna':300, 'mariah':220}
print( album_sales.values()) print(list( album_sales.values()))
```

The output is as follows:

```
dict_values([150, 75, 300, 220])
 [150, 75, 300, 220]
```

The values() method in this code returns an iterable object. To use the values straight away, you can wrap them in a list directly.

3. Now, obtain a list of keys in a dictionary by using the keys() method:

print(list(album_sales.keys()))

The output is as follows:

['barbara', 'aretha', 'madonna', 'mariah']

4. Although you can't directly iterate a dictionary, you can loop through the dictionary by using the items() method, as in the following code snippet:

for item in album_sales.items(): print(item)

The output is as follows:

('barbara', 150)
aretha75('madonna', 300)
('mariah', 220)
In this exercise, you created a dictionary, accessed the keys and values of the dictionary, and looped through the dictionary.

Exercise 27 – exploring tuple properties in a dance genre list

In this exercise, you will learn about the different properties of a tuple: 1.

Open a Jupyter notebook.

2. Type the following code in a new cell to initialize a new tuple, t:

t = ('ballet', 'modern', 'hip-hop')

print(len(t))

The output is as follows:

3

Note

Remember, a tuple is immutable; therefore, you can't use the append method to add a new item to an existing tuple. You can't change the value of any existing tuple's elements since both of the following statements will raise an error.

3. Now, as mentioned in the note, enter the following lines of code and observe the error:

t[2] = 'jazz'

The output is as follows:

----→1 t[2] = 'jazz'

TypeError: 'tuple' object does not support item assignment

The only way to get around this is to create a new tuple by concatenating the existing tuple with other new items.

4. Now, use the following code to add two items, jazz and tap, to our tuple, t. This will give us a new tuple. Note that the existing t tuple remains unchanged:

print(t + ('jazz', 'tap')) print(t)

The output is as follows:

('ballet', 'modern', 'hip-hop', 'jazz', 'tap') ('ballet', 'modern', 'hip-hop')

5. Enter the following statements in a new cell and observe the output:

t_mixed = 'jazz', True, 3

print(t_mixed)

t_dance = ('jazz',3), ('ballroom',2), ('contemporary',5)

print(t_dance)

Tuples also support mixed types and nesting, just like lists and dictionaries. You can also declare a tuple without using parentheses, as shown in the code you entered in this step.

The output is as follows:

('jazz', True, 3)

(('jazz', 3), ('ballroom', 2), ('contemporary', 5))

Exercise 28 – writing and importing our first module

In this exercise, as in Exercise 35 – writing and executing our first script, you will find the sum of the factorials of three numbers. However, this time, you will write the code inside of a function

and save the function as a module called my_module.py. Here are the steps to write your first module:

1. Using any text editor, create a new file called my_module.py. You can also use Jupyter (New | Text File).

2. Add a function that returns the result of the computation in Exercise 35 – writing and executing our first script:

```
import math
def factorial_sum(numbers):
total = 0
for n in numbers:
total += math.factorial(n) return total
```

3. Save the file as my_module.py.

4. Open a new Python shell or Jupyter Notebook and execute the following:

python

>>>from my_module import factorial_sum >>>factorial_sum([5, 7, 11])

The output is as follows:

39921960

Exercise 29 – adding a docstring to my_module.py

In this exercise, you extend your my_module.py module from Exercise 36 – writing and importing our first module by adding a docstring. Here are the steps:

1. Open my_module.py in Jupyter or a text editor.

2. Add a docstring to the script (as the first line before beginning with your code, as shown in the following code snippet):

""" This script computes the sum of the factorial of a list of numbers"""

3. Open a Python console in the same directory as your my_module.py file. 4. Import the my_module module by running the following command:

import my_module

5. Call the help function on your my_module script to view the docstring. The help function can be used to obtain a summary of any available information regarding a module, function, or class in Python. You can also call it without an argument—that is, as help()—to start an interactive series of prompts:

help(my_module)

The output is as follows:

```
Help on module my_module:

NAME
    my_module - This script computes the sum of the factorial of a list of numbers

FUNCTIONS
    factorial_sum(numbers)

FILE
    /Users/coreyjwade/my_module.py
```

Exercise 30 – linear search in Python

In this exercise, you will implement the linear search algorithm in Python using a list of numbers. Proceed as follows:

1. Start with a list of numbers:

l = [5, 8, 1, 3, 2]

2. Specify a value to search for:

search_for = 8

3. Create a result variable that has a default value of -1. If the search is unsuccessful, this value will remain -1 after the algorithm is executed:

result = -1

4. Loop through the list. If the value equals the search value, set the result variable equal to the index of the value and exit the loop:

for i in range(len(l)): if search_for == l[i]:

result = i]

break

5. Check the result:

print(result)

The output is as follows:

1

Note

This means that the search found the required value at position 1 in the list (which is the second item in the list, as indices start from 0 in Python).

Exercise 31 – defining a function with keyword arguments

In this exercise, you will use the Python shell to define an add_suffix function that takes an optional keyword argument. The steps for this exercise are as follows:

1. In a Python shell, define an add_suffix function:

def add_suffix(suffix='.com'): return 'google' + suffix

2. Call the add_suffix function without specifying the suffix argument:

add_suffix()

The output is as follows:

'google.com'

3. Call the function with a specific suffix argument:

add_suffix('.co.uk')

The output is as follows:

'google.co.uk'

Exercise 32 – defining a function with positional and keyword arguments

In this exercise, you use the Python shell to define a convert_usd_to_aud function that takes a positional argument and an optional keyword argument, with the following steps:

1. In a Python shell, define a convert_usd_to_aud function:

```
def convert_usd_to_aud(amount, rate=0.75): return amount / rate
```

2. Call the convert_usd_to_aud function without specifying the exchange rate argument:

```
111
convert_usd_to_aud(100)
```

You should get the following output:

```
133.33333333333334
```

3. Call the convert_usd_to_aud function with a specific exchange rate argument:

```
convert_usd_to_aud(100, rate=0.78)
```

The output is as follows:

```
128.2051282051282
```

The rule of thumb is to simply use positional arguments for required inputs that must be provided each time the function is called, and keyword arguments for optional inputs.