Programming in Python: Takeaways

Syntax

- Displaying the output of a computer program:

```
print(1 + 2)

print(5   10)
```

- Ignoring certain lines of code by using code comments:

```
# print(1 + 2)

print(5   10)

# This program will only print 50
```

- Performing arithmetical operations:

```
1 + 2

4 - 5

30   1

20 / 3

4**3

(4   18)**2 / 10
```

Concepts

- When we give a computer a set of instructions, we say that we're **programming** it. To program a computer, we need to write the instructions in a special language, which we call a **programming language**.
- Python has **syntax** rules, and each line of instruction has to comply with these rules. For example, `print(23   + 7) print(10   -  6) print(12   + 38)` doesn't comply with Python's syntax rules and raises a **syntax error**.
- The instructions we send to the computer are collectively known as **code**. Each line of instruction is known as a **line of code**.

- When we write code, we *program* the computer to do something. For this reason, we also call the code we write a **computer program**, or a **program**.
- The code we write serves as **input** to the computer. The result of executing the code is called **output**.
- The sequence of characters that follows the `#` symbol is called a **code comment**. We can use code comments to stop the computer executing a line of code or add information about the code we write.

Variables and Data Types: Takeaways

Syntax

- Storing values to variables:

```
twenty = 20

result = 43 + 2**5

currency = 'USD'
```

- Updating the value stored in a variable:

```
x = 30

x += 10 # this is the same as x = x + 10
```

- Rounding a number:

```
round(4.99) # the output will be 5
```

- Using quotation marks to create a string:

```
app_name = "Clash of Clans"

app_rating = '3.5'
```

- Concatenating two or more strings:

```
print('a' + 'b') # prints 'ab'

print('a' + 'b' + 'c') # prints 'abc'
```

- Converting between types of variables:

```
int('4')

str(4)

float('4.3')

str(4.3)
```

- Finding the type of a value:

```
type(4)

type('4')
```

Concepts

- We can store values in the computer memory. Each storage location in the computer's memory is called a **variable**.
- There are two syntax rules we need to be aware of when we're naming variables:
    - We must use only letters, numbers, or underscores (we can't use apostrophes, hyphens, whitespace characters, etc.).
    - Variable names can't start with a number.
- Whenever the syntax is correct, but the computer still returns an error for one reason or another, we say we got a **runtime error**.
- In Python, the `=` operator tells us that the value on the right is **assigned** to the variable on the left. It doesn't tell us anything about equality. We call `=` an **assignment operator**, and we read code like `x = 5` as "five is assigned to x" or "x is assigned five," but not "x equals five."
- In computer programming, values are classified into different **types**, or **data types**. The type of a value offers the computer the required information about how to handle that value. Depending on the type, the computer will know how to store a value in memory, or what operations can and can't be performed on a value.
- In this mission, we learned about three data types: integers, floats, and strings.
- The process of linking two or more strings together is called **concatenation**.

Resources

- More on Strings in Python.

Lists and For Loops: Takeaways

Syntax

- Creating a list of data points:

```
row_1   ['Facebook', 0.0, 'USD', 2974676, 3.5]

row_2   ['Instagram', 0.0, 'USD', 2161558, 4.5]
```

- Creating a list of lists:

```
data = [row_1, row_2]
```

- Retrieving an element of a list:

```
first_row = data[0]

first_element_in_first_row = first_row[0]

first_element_in_first_row = data[0][0]

last_element_in_first_row = first_row[-1]

last_element_in_first_row = data[0][-1]
```

- Retrieving multiple list elements and creating a new list:

```
row_1 = ['Facebook', 0.0, 'USD', 2974676, 3.5]

rating_data_only = [row_1[3], row_1[4]]
```

- Performing list slicing:

```
python

row_1 = ['Facebook', 0.0, 'USD', 2974676, 3.5]

second_to_fourth_element = row_1[1:4]
```

- Opening a data set file and using it to create a list lists:

```
opened_file = open('AppleStore.csv')

from csv import reader

read_file   reader(opened_file)

apps_data   list(read_file)
```

- Repeating a process using a for loop:

```
row_1 = ['Facebook', 0.0, 'USD', 2974676, 3.5]

for data_point in row_1:

    print(data_point)
```

Concepts

- A **data point** is a value that offers us some information.
- A set of data points make up a **data set**. A table is an example of a data set.
- **Lists** are data types which we can use to store data sets.
- Repetitive process can be automated using **for loops**.

Resources

- Python Lists
- Python For Loops
- More on CSV files
- A list of keywords in Python `for` and `in` are examples of keywords (we used `for` and `in` to write for loops)

Conditional Statements: Takeaways

Syntax

- Using an if statement to control your code:

```
if True:
    print(1)
if 1 == 1:
    print(2)
    print(3)
```

- Combining multiple conditions:

```
if 3 > 1 and 'data == data':
    print('Both conditions are true!')
if 10 < 20 or 4 <= 5:
    print('At least one condition is true.')
```

- Building more complex if statements:

```
if (20 > 3 and 2 != 1) or 'Games' == 'Games':
    print('At least one condition is true.')
```

- Using the else clause:

```
if False:
    print(1)
else:
    print('The condition above was false.')
```

- Using the elif clause:

```
if False:
    print(1)
elif 30 > 5:
    print('The condition above was false.')
```

Concepts

- We can use an `if` **statement** to implement a condition in our code.
- An `elif` clause is executed if the preceding `if` statement (or the other preceding `elif` clauses) resolves to `False` *and* the condition specified after the `elif` keyword evaluates to `True`
- `True` and `False` are **Boolean values**.
- `and` and `or` are **logical operators**, and they bridge two or more Booleans together.
- We can compare a value `A` to value `B` to determine whether:
    - `A` is **equal** to `B` and vice versa (`B` is equal to `A`)
    - `A` is **not equal** to `B` and vice versa `!=`
    - `A` is **greater** than `B` or vice versa — `>` .
    - `A` is **greater than or equal to** `B` or vice versa — `>=`
    - `A` is **less** than `B` or vice versa `<` .
    - `A` is **less than or equal to** `B` or vice versa `<=`

Resources

- If Statements in Python

Dictionaries and Frequency Tables: Takeaways

Syntax

- Creating a dictionary:

```
# First way

dictionary = {'key_1': 1, 'key_2': 2}

# Second way

dictionary = {}

dictionary['key_1']   1

dictionary['key_2']   2
```

- Retrieving individual dictionary values:

```
dictionary = {'key_1': 100, 'key_2': 200}

dictionary['key_1']  # Outputs 100

dictionary['key_2']  # Outputs 200
```

- Checking whether a certain value exist in the dictionary as a key:

```
dictionary = {'key_1': 100, 'key_2': 200}

'key_1' in dictionary  # Outputs True

'key_5' in dictionary  # Outputs False

100 in dictionary  # Outputs False
```

- Updating dictionary values:

```
dictionary = {'key_1': 100, 'key_2': 200}

dictionary['key_1'] += 600  # This will change the value to 700
```

- Creating a frequency table for the unique values in a column of a data set:

```
frequency_table = {}

for row in a_data_set:

    a_data_point = row[5]

    if a_data_point in frequency_table:

        frequency_table[a_data_point] += 1

    else:

        frequency_table[a_data_point]   1
```

Concepts

- The index of a dictionary value is called a **key**. In `'4+': 4433`, the dictionary key is `'4+'`, and the dictionary value is `4433`. As a whole, `'4+': 4433` is a **key-value pair**.
- Dictionary values can be of any data type: strings, integers, floats, Booleans, lists, and even dictionaries. Dictionary keys can be of almost any data type we've learned so far, except for lists and dictionaries. If we use lists or dictionaries as dictionary keys, the computer raises an error.
- We can check whether a certain value exist in the dictionary as a key using an the `in` operator. An `in` expression always returns a Boolean value.
- The number of times a unique value occurs is also called **frequency**. Tables that map unique values to their frequencies are called **frequency tables**.
- When we iterate over a dictionary with a `for` loop, the looping is done by default over the dictionary keys.

Resources

- Dictionaries in Python

Functions: Fundamentals: Takeaways

Syntax

- Creating a function with a single parameter:

```
def square(number):
    return number**2
```

- Creating a function with more than one parameter:

```
def add(x, y):
    return x + y
```

- Reusing a function within another function's definition:

```
def add_to_square(x):
    return square(x) + 1000  # we defined square() above
```

Concepts

- Generally, a function displays this pattern:
  - It takes in an input.
  - It does something to that input.
  - It gives back an output.
- In Python, we have **built-in functions** like `sum()`, `max()`, `min()`, `len()`, and `print()`, and functions that we create ourselves.
- Structurally, a function is composed of a header (which contains the `def` statement), a body, and a `return` statement.
- Input variables are called **parameters**, and the various values that parameters take are called **arguments**. In `def square(number)`, the `number` variable is a parameter. In `square(number=6)`, the value `6` is an argument that is passed to the parameter `number`

- Arguments that are passed by name are called **keyword arguments** (the parameters give the name). When we use multiple keyword arguments, the order we use doesn't make any practical difference.
- Arguments that are passed by position are called **positional arguments**. When we use multiple positional arguments, the order we use matters.
- **Debugging** more complex functions can be a bit more challenging, but we can find the **bugs** by reading the **traceback**.

Resources

- Functions in Python

Functions: Intermediate: Takeaways

Syntax

- Initiating parameters with **default arguments**:

```
def add_value(x, constant=3.14):
    return x + constant
```

- Using **multiple return statements**:

```
def sum_or_difference(a, b, do_sum):
    if do_sum:
        return a + b
    return a - b
```

- Returning **multiple variables**:

```
def sum_and_difference(a, b):
    a_sum = a + b
    difference = a - b
    return a_sum, difference
sum_1, diff_1 = sum_and_difference(15, 10)
```

Concepts

- We need to avoid using the name of a built-in function to name a function or a variable because this overwrites the built-in function.
- Each built-in function is well documented in the official Python **documentation**.
- Parameters and return statements are not mandatory when we create a function.

```
def print_constant():
    x = 3.14
    print(x)
```

- The code inside a function definition is executed only when the function is called.
- When a function is called, the variables defined inside the function definition are saved into a temporary memory that is erased immediately after the function finishes running. The temporary memory associated with a function is isolated from the memory associated with the main program (the main program is the part of the program outside function definitions).
- The part of a program where a variable can be accessed is often called scope. The variables defined in the main program are said to be in the global scope, while the variables defined inside a function are in the local scope.
- Python searches the global scope if a variable is not available in the local scope, but the reverse doesn't apply. Python won't search the local scope if it doesn't find a variable in the global scope. Even if it searched the local scope, the memory associated with a function is temporary, so the search would be pointless.

Resources

- Python official documentation
- Style guide for Python code

Project: Learn and Install Jupyter Notebook: Takeaways

Syntax

MARKDOWN SYNTAX

- Adding italics and bold:

  ```
  *Italics*
  ```

  ```
   *Bold*
  ```

- Adding headers (titles) of various sizes:

  ```
  # header one
  ```

  ```
  ## header two
  ```

- Adding hyperlinks and images:

  ```
  [Link](http://a.com)
  ```

- Adding block quotes:

  ```
  > Blockquote
  ```

- Adding lists:

  ```
  *
  ```

  ```
  *
  ```

  ```
  *
  ```

- Adding horizontal lines:

- Adding inline code:

  ```
  `Inline code with backticks`
  ```

- Adding code blocks

  ```
  code
  ```

JUPYTER NOTEBOOK SPECIAL COMMAND

- Displaying the code execution history:

```
%history -p
```

Concepts

- Jupyter Notebook is much more complex than a code editor. Jupyter allows us to:
    - Type and execute code.
    - Add accompanying text to our code (including math equations).
    - Add visualizations.
- Jupyter can run in a browser and is often used to create compelling data science projects that can be easily shared with other people.
- A notebook is a file created using Jupyter notebooks. Notebooks can easily be shared and distributed so people can view your work.
- Types of modes in Jupyter:
    - Jupyter is in edit mode whenever we type in a cell — a small pencil icon appears to the right of the menu bar.
    - Jupyter is in command mode whenever we press `Esc` or whenever we click outside of the cell — the pencil to the right of the menu bar disappears.
- State refers to what a computer remembers about a program.
- We can convert a code cell to a Markdown cell to add text to explain our code. Markdown syntax allows us to use keyboard symbols to format our text.
- Installing the Anaconda distribution will install both Python and Jupyter on your computer.

Keyboard Shortcuts

- Some of the most useful keyboard shortcuts we can use in command mode are:
    - `Ctrl` + `Enter`: run selected cell
    - `Shift` + `Enter`: run cell, select below
    - `Alt` + `Enter`: run cell, insert below
    - `Up`: select cell above
    - `Down`: select cell below
    - `Enter`: enter edit mode
    - `A`: insert cell above
    - `B`: insert cell below
    - `D`, `D` (press D twice): delete selected cell
    - `Z`: undo cell deletion
    - `S`: save and checkpoint
    - `Y`: convert to code cell

- M: convert to Markdown cell
- Some of the most useful keyboard shortcuts we can use in edit mode are:
    - `Ctrl` + `Enter`: run selected cell
    - `Shift` + `Enter`: run cell, select below
    - `Alt` + `Enter`: run cell, insert below
    - `Up`: move cursor up
    - `Down`: move cursor down
    - `Esc`: enter command mode
    - `Ctrl` + `A`: select all
    - `Ctrl` + `Z`: undo
    - `Ctrl` + `Y`: redo
    - `Ctrl` + `S`: save and checkpoint
    - `Tab` : indent or code completion
    - `Shift` + `Tab`: tooltip

Resources

- Jupyter Notebook tutorial
- Jupyter Notebook tips and tricks
- Markdown syntax
- Installing Anaconda