Python: Getting Started

Simple Calculations

In addition to all the more complex things you can do in Python, it is also useful for completing simple mathematical operations.

Addition and Subtraction

Add numbers together like this:

In [ ]:
```
1 + 1
```

Notice that you don't need to include = in order to make the operation work.

Subtract numbers like this:

In [ ]:
```
2 - 1
```

Division

Divide numbers like this:

In [ ]:
```
4 / 2
```

Remember that Python will return an error if you try to divide by 0!

Modulo Division

Perform modulo division like this:

In [ ]:
```
5 % 2
```

Multiplication

Multiply numbers like this:

In [ ]:
```
4 * 2
```

Add **exponents** to numbers like this:

In [ ]:
```
4 ** 2
```

Order of Operations

Just like everywhere else, Python works on a specific order of operations. That order is:

Parentheses Exponents Multiplication Addition Subtraction

If you remember being taught PEMDAS, then this is the same thing! All the operations in Python work with levels of parentheses, so if you wanted to add two numbers together, and then add another number to the result, the code would look like this:

In [ ]:
```
(6 + 4) + 4
```

Things get more complicated when we add more terms to the equation, but the principle remains the same: if you open a set of parentheses, make sure you close it too. Here's an example that uses all the PEMDAS possibilities:

In [ ]:
```
(((5 ** 3 + 7) * 4) / 16 + 9 - 2)
```

Data Types

There are lots of different types of data in the world, and Python groups that data into several categories.

Boolean (`bool`)**:**

- Any data which can be expressed as either `True` or `False` .
- Used when comparing two values. For example, if you enter `10 > 9` , Python will return `True` .

String (`str`)**:**

- Data that involves text — either letters, numbers, or special characters.
- Strings are enclosed in either single- or double-quotation marks: `"string-1"` or `'string-2'` .

Numeric (`int` , `float` , `complex`)**:**

- Data that can be expressed numerically.
- An integer, or `int` , is a whole number, positive or negative, without decimals, of unlimited length: `123` .
- A floating-point number, or `float` , is a number, positive or negative, containing one or more decimals: `123.01` .
- A complex number, or `complex` , are imaginary numbers, designated by a `j` : `(3 + 6j)` .

Sequence (`list` , `tuple` , `set`)**:**

- Data that is a collection of discrete items.
- A `list` is collection that is ordered and changeable. It's designated using square brackets `[]` , and items can be of different data types: `["red", 1, 1.03, 1]` .
- A `tuple` is a collection which is ordered and unchangeable. It's designated using parentheses `()` : `("red", 1, 1.03, 1)` .

- A `set` is a collection which is unordered, unchangeable, and does not permit duplicate items. It's designated using curly brackets `{}`: `{"red", 1, 1.03}`.

Mapping (`dict`):

- Dictionaries store data in *key-value* pairs. They're designated using curly brackets `{}`, like a `set`, but notice that keys and values are associated with each other using a colon `:`. Each pair is separated from the next using a comma `,`.

```
dict1 = {
    "department": "quindio",
    "property_type": "house",
    "price_usd": 330899.98
}
```

Binary (`bytes`, `bytearray`, `memoryview`):

- Used to manipulate and display binary data. That is, data that can be expressed with integers represented with base 2.
- Unlike the other data types described above, `binary` types are not human-readable.

Lists

In Python, a **list** is a collection of data that stores multiple items in a single variable. These items must be ordered, able to be changed, and can be duplicated. A list can store data of multiple types; not all the items in the list need to be the same type.

Creating Lists

Lists can be as long or as short as you like. Let's create a short list based on data from the Colombian real estate market to give us something to work with.

Lists are written with square brackets. Code for a short list that shows the price of houses in US dollars looks like this:

In [ ]:
```
price_usd = [97919.38, 300511.20, 293758.14]
print(price_usd)
```

Working with Lists

After you've created a list, you can **access** any item on the list by referring to the item's index number. Keep in mind that in Python, the first item in a list is always 0.

Let's access the second item of our `price_usd` list.

In [ ]:
```
print(price_usd[1])
```

Practice

Try it yourself! Create and print a list that shows the area of the houses, called `area_m2`. Include the items `187.0`, `82.0`, and `235.0`.

In [ ]:
```
area_m2 = ...
print(area_m2)
```

If we want to access the an item at the end of the list, we can use **negative indexing**. In negative indexing, -1 refers to the last item, -2 to the second to last, and so on.

Let's access the last item in our `department` list.

In [ ]:
```
print(price_usd[-1])
```

Practice

Try accessing the second item in your `area_m2` list.

In [ ]:
```

```

Try accessing the last item in the same list.

In [ ]:
```

```

Appending Items

It's also possible to add an item to a list that already exists using the `append` method like this:

In [ ]:
```
price_usd.append(540244.86)
```

Practice

Add the item `195.0` to your `area_m2` list.

In [ ]:
```
print(area_m2)
```

Aggregating Items

We can also **aggregate** items on a list to make analyzing the list more useful. For example, if we wanted to know the total value in US dollars of the houses on our `price_usd` list, we could use the `sum` method.

In [ ]:
```
total_usd = sum(price_usd)
```

We might also be interested in the average value in US dollars of the houses on the same list. To

find the average, we add the `len` method to the `sum` method.

In [ ]:
```
average_usd = sum(price_usd) / len(price_usd)
```

Practice

Try it yourself! Calculate the total area of the houses on your `area_m2` list, and find the average area of all the houses on the list.

In [ ]:
```
total_area_m2 = ...
print(total_area_m2)
```

In [ ]:
```
average_area_m2 = ...
print(average_area_m2)
```

Zipping Items

Finally, it might be useful to combine -- or **zip** -- two lists together. For example, we might want to create a new list that pairs the values in our `price_usd` list with our `area_m2` list. To do that, we use the `zip` method. The code looks like this:

You might have noticed that the above code involving putting one list (in this case, `new_list`) inside another list (in this case, `list`). This approach is called a **generator**, and we'll come back to what that is and how it works later in the course.

In [ ]:
```
new_list = zip(price_usd, area_m2)
zipped_list = list(new_list)
```

You might have noticed that the above code involving putting one list (in this case, `new_list`) inside another list (in this case, `list`). This approach is called a **generator**, and we'll come back to what that is and how it works later in the course.

Practice

Try it yourself! Create a list called `area_m2` that includes the terms `235.0`, `130.0`, and `137.0`, then create another list called `price_cop` that includes the terms `400000000.0`, `850000000.0`, and `475000000.0`. Then zip them together to create a new list called `area_price`, and `print` the result.

In [ ]:
```
area_m2 = ...
```

Python `for` Loops

A `for` Loop is used for executing a set of statements for each item in a list.

Working with `for` Loops

There can be as many statements as there are items in the list, but to keep things manageable, let's use our list of real estate values in Colombia.

In [ ]:
```
price_usd = [97919.38, 300511.20, 293758.14, 540244.86]
print(price_usd)
```

We might want to see each of the values in the list, so we insert a `for` Loop:

In [ ]:
```
price_usd = [97919.38, 300511.20, 293758.14, 540244.86]
for x in price_usd:
    print(x)
```

Note that the `print` command is indented.

Practice

Try it yourself using the `area_m2` list:

In [ ]:
```
area_m2 = ...
```

In [ ]:
```

```

Python Dictionaries

In Python, a **dictionary** is a collection of data that occurs in an order, is able to be changed, and does not allow duplicates. Data in a dictionary are always presented as **keys** and **values**, and those key-value pairs cannot be duplicated in the dataset.

Creating Dictionaries

Dictionaries can be as big or as small as you like. Let's create a small dictionary based on data from the Colombian real estate market to give us something to work with.

Dictionaries are written with curly brackets, with key-value pairs inside. Code for a small dictionary looks like this:

In [ ]:
```
colomdict = {
    "property_type": "house",
    "department": "quindio",
    "area": 235.0,
}
```

Practice

Try it yourself! Create and `print` a dictionary called `bogota` with the key-value pairs `"price_usd": 121,555.09`, `"area_m2": 82.0`, and `"property_type": "house"`

In [ ]:
```
bogota = ...
print(bogota)
```

Working with Dictionaries

After you've created a dictionary, you can **access** any item by using its key name inside square brackets.

Going back to our example dictionary, let's access the value for "department".

In [ ]:
```
x = colomdict["department"]
print(x)
```

Practice

Try accessing the value for `price_usd` in the Bogotá dictionary you created above.

In [ ]:
```
x = ...
```

You can also use `get` to retrieve a value. That looks like this:

In [ ]:
```
x = colomdict.get("department")
```

Practice

Now try accessing the value for `area_m2` using the `get` method, and print the result.

In [ ]:
```
x = ...
```

JSON

JSON stands for Java Script Object Notation, and it's a text format for storing and transporting data.

Working with JSON

JSON works by creating **key-value pairs**, where the key is data that can be represented by letters (called a **string**). JSON values can be strings, numbers, objects, arrays, boolean data, or null. JSON usually comes as a list of dictionaries, which look like this:

Here's an example from our `colombia-real-estate-1` dataset with two key-value pairs that both include string values:

In [ ]:
```
[
    {"property_type": "house", "department": "bogota"},
    {"property_type": "house", "department": "bogota"},
    {"property_type": "house", "department": "bogota"},
]
```

Here's an simplified example from our `colombia-real-estate-1` dataset with two key-value pairs that both include string values:

In [ ]:
```
{"property_type": "house", "department": "bogota"}
```

JSON pairs with numbers look like this:

In [ ]:
```
{"area_m2": 187.0, "price_usd": 330899.98}
```

When you mix more than one type of value, it looks like this:

In [ ]:
```
{"property_type": "house", "price_usd": 330899.98}
```

References & Further Reading

- A guide to basic math operations in Python
- Python documentation on built-in data types
- Summary of Python data types
- Tutorial on type conversion in Python
- A description of how dictionaries work in Python
- An introduction to JSON
- An introduction to lists in Python
- How to zip lists
- Calculating mean, median, and mode in Python
- A brief tutorial of For Loops
