python_basics

January 25, 2025

1 Python Basics

1.1 Variables and Math

Python can be used like a calculator, with the ability to store multiple named variables.

[1]:
```
3 + 5 * 4
```

[1]:
```
23
```

We can store variables using a name and an equal sign.

[2]:
```
weight_kg = 60.3
height_m = 1.9
```

There are different types of data that we can store in variables, including: * Integer numbers * Floating-point numbers (decimals) * Strings

[3]:
```
name = "John Doe"
height = 5.9
age = 23
```

In Jupyter, we can display the value of a variable just by writing its name at the end of a cell. Outside of Jupyter, we need to use `print`. The output is a little different between these methods, but they both show the value of the variable.

[4]:
```
name
```

[4]:
```
'John Doe'
```

[5]:
```
print(name)
```

```
John Doe
```

`print` is an example of a function. In Python (and other programming languages), functions take some sort of input and produce some sort of output. We can run them like so:

`output1, output2, ... = function_name(input1, input2, ...)`

`print` is a simple function that takes some variable and displays a text version of it. It's often helpful when you need to see the value of a variable, and we'll use it in examples.

Sometimes, it's helpful to construct a more complicated string from individual variables. We can use a feature called *f-strings* to do this.

[25]:
```
greeting = "Hello"
user = "Mary"
message = f"{greeting} {user}"
print(message)
```

```
Hello Mary
```

You make an f-string by adding an "f" before the quotes that make up a string. Variable names can be placed in curly braces ({}) to place their contents at that point in the string.

There are a lot of options for formatting variables as strings, making this method very flexible. We won't get into this much now, but it's helpful to know about. See the Python tutorial for more information.

[28]:
```
name = "John Doe"
height = 5.9
age = 23
print(f"Patient {name}: {height=}, {age=}")
```

```
Patient John Doe: height=5.9, age=23
```

There are many built-in functions in Python. For example, if I want to round a number, I can use the `round` function.

[6]:
```
height_precise = 5.912
height_rounded = round(height_precise, 2)
height_rounded
```

[6]:
```
5.91
```

Unlike some other languages, we don't have to declare what type of data we're using. After making a variable, we can check the data type using the `type` function.

[24]:
```
weight_kg = 60.3
weight_kg_rounded = 60

# get type of variables
type_kg = type(weight_kg)
type_kg_rounded = type(weight_kg_rounded)
print(type_kg)
print(type_kg_rounded)
```

```
<class 'float'>
<class 'int'>
```

We can use

[8]:
```
print(f"With a decimal ({weight_kg}), we get: {type_kg}")
```

```
With a decimal (60.3), we get: <class 'float'>
```

[9]:
```
print(f"with no decimal ({weight_kg_rounded}), we get: {type_kg_rounded}")
```

```
with no decimal (60), we get: <class 'int'>
```

Variable names can be almost anything, but they cannot start with a number (for example, `4vars` is invalid as a variable name).

There's an art to naming variables to make them informative but not too hard to type. In Python, the usual style is to separate words with underscores (_).

[23]:
```
partid = '001'  # hard to read and understand
participant_id = '001'  # clearer, though it will take longer to type
ParticipantID = '001'  # not bad, but goes against standard Python style␣
↪guidelines
```

1.1.1 Review: Variables and Math

Make variables to represent a participant's identifier, `sub-002`, their accuracy on a test where they got 16 out of 24 items correct, and their age in years (23). Check the types of your variables, which should be `str`, `float`, and `int`. Round their accuracy to two decimal places.

1.2 Lists

A common way to organize data is to place elements into a list. We can put any kind of data into a list.

[10]:
```
participant_ids = ["001", "002", "003"]
mixed_data = ["John Doe", 5.9, 23]
```

After we create a list, we can access elements of that list using *indexing*. Indexing lets us access data based on its position in the list.

A slightly confusing thing about Python (and many programming languages) is that the first index is 0, and it counts up from there. So, the first element is at 0, the second element is at 1, etc.

[11]:
```
print(participant_ids)
```

```
['001', '002', '003']
```

[12]:
```
print(participant_ids[0])
```

```
001
```

[13]:
```
print(participant_ids[1])
```

```
002
```

Besides accessing one element at a time, we can also access a range of elements (called a *slice*) using the colon operator (`:`).

Slices usually just have a start index and a finish index, like this:

```
my_list[start:finish]
```

The finish is *non-inclusive*. For example, `[0:2]` will get the first two elements.

[14]:
```
print(participant_ids[0:2])
```

```
['001', '002']
```

This picture summarizes how things work:

```
list:  [  1,  2,  3,  4,  5  ]
index: [  0   1   2   3   4  ]
slice: [0   1   2   3   4   5]
```

The index row shows the index to get individual elements of the list. For example, if we access index 2, we'll get 3.

[8]:
```
l = [1, 2, 3, 4, 5]
print(l[2])
```

```
3
```

The slice row shows how slicing works. You can think of the slice indices as being sort of between the elements of the list. When you slice a list, you'll get back everything between the slice indices.

[9]:
```
print(l[0:3])
print(l[3:5])
```

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

If either the start or finish of a slice is omitted, the corresponding end of the list will be used.

[15]:
```
print(participant_ids[:2])  # from the start until 2
```

```
['001', '002']
```

[16]:
```
print(participant_ids[1:])  # from 1 until the end
```

```
['002', '003']
```

The way indexing and slicing work might seem weird, but it has some nice properties. For example, if we have a list of 4 elements:

[1]:
```
new_list = [1, 2, 4, 8]
```

We can split it in half in an intuitive way, where the index 2 marks the center of the list:

[3]:
```
print(new_list[:2])  # first half
print(new_list[2:])  # second half
```

```
[1, 2]
[4, 8]
```

Lists can hold any type of data. This means we can even make lists of lists.

[11]:
```
participant_groups = [["001", "003", "005"], ["002", "004", "006"]]
```

We can then put together multiple indices. For example, to get the third entry in the second group:

[12]:
```
print(participant_groups[1][2])
```

```
006
```

Slicing and indexing work the same way no matter what type of data we have in the list, so we can use it for lists of lists too.

[13]:
```
print(participant_groups[0])
print(participant_groups[0][1:])
```

```
['001', '003', '005']
['003', '005']
```

Finally, in addition to the usual indices, which are non-negative integers (0, 1, 2, etc.), we can use negative indices instead, which count from the *end* of the list instead of the start.

```
list:  [   1,  2,  3,  4,  5   ]
index: [  -5  -4  -3  -2  -1   ]
slice: [-5  -4  -3  -2  -1   0 ]
```

[17]:
```
l = [1, 2, 3, 4, 5]
print(l[-1])
print(l[-5:-3])
```

```
5
[1, 2]
```

[ ]:
```

```