2_python

October 2, 2023

1 SC207 Python Fundamentals

1.1 Loops, Lists and Strings, oh my!

- Our main priorities in this course will be to generate and analyse data.
- When we can rely on a tried and trusted toolset, we will.
- However, in order to do that we must understand the basics of how Python works.

2 Basics

2.1 Simple Expressions

Programming is fundamentally about creating and manipulating different types of data to get some sort of outcome. A really basic example…

[ ]:
```
# Addition
1 + 2
```

[ ]:
```
3
```

[ ]:
```
# Subtraction
10 - 5
```

[ ]:
```
# Multiplication
5 * 10
```

[ ]:
```
# Divison
10 / 2
```

[ ]:
```
# Exponents ('...to the power of')
5 ** 2
```

[ ]:
```
# We can connect these operators together
((5 + 1 - 2) * 3) / 2
```

[ ]:
```
6.0
```

2.2 Text (Strings)

We're not limited to just numbers. Text in Python is called a **String** as it is a collection of characters 'strung' together.

- Strings are defined by enclosing text in either 'single' or "double" quotes.
- Make sure you always 'open' and 'close' your strings with the same type of quote mark.

```
[ ]: 'I am a string'
```

```
[ ]: 'I am a string'
```

```
[ ]: "I am also a string!"
```

```
[ ]: 'I am also a string!'
```

```
[ ]: "We have both types 'so we can still have quote marks' in our strings."
```

```
[ ]: "We have both types 'so we can still have quote marks' in our strings."
```

```
[ ]: 'I am broken and will throw a Syntax Error"
```

```
  File "<ipython-input-13-2dc857b980ac>", line 1
    'I am broken and will throw a Syntax Error"
                                               ^
SyntaxError: EOL while scanning string literal
```

We can also do some basic operations with strings - Addition will join strings together - Multiplication will replicate strings

```
[ ]: "Hello my name is " + "Javier the FANTASTIC!"
```

```
[ ]: 'Hello my name is Javier the FANTASTIC!'
```

```
[ ]: 'Time to say a lot! ' * 5 # Note the space on the end
```

```
[ ]: 'Time to say a lot! Time to say a lot! Time to say a lot! Time to say a lot!
     Time to say a lot! '
```

2.3 Print

By default, Jupyter will display the results of an operation, but only the last line.

```
[ ]: 1 + 1 + 2
     'There can only be one!'
```

```
[ ]: 'There can only be one!'
```

We can force it to display something using `print()`

[ ]:
```python
print(1 + 1 + 2)
print('There can only be one!')
```

```
4
There can only be one!
```

2.4 Assigning Variables

Assigning variables allows us to create a label, and then assign a value to that label. This then allows us to manipulate or use that value in our code.

Assigning values to variables is a key building block of the Python environment. Variables are assigned using the = operator.

[ ]:
```python
my_number = 10
print(my_number)
```

```
10
```

Variables store whatever you assign to it allowing you to then use that value in your code just like you would use the original value.

[ ]:
```python
part_a = 'For now I am ' # note the extra space at the end of the first string.
part_b = 'whole again'

part_a + part_b
```

[ ]:
```
'For now I am whole again'
```

[ ]:
```python
first_name = 'Scooby'
surname = 'Doo'
job = 'Detective'

first_name + ' ' + surname + ' is a ' + job
```

[ ]:
```
'Scooby Doo is a Detective'
```

2.5 Object Methods

Everything in Python is an ‘object’ and objects come in many different types.

[ ]:
```python
type('Hello')
```

[ ]:
```
str
```

[ ]:
```python
type(10)
```

[ ]:
```
int
```

```
[ ]: # Note that numbers with decimals are an entire different type
     type(1.5)
```

```
[ ]: float
```

```
[ ]: # even that print function is an object
     type(print)
```

```
[ ]: builtin_function_or_method
```

Different types of object behave differently and have different methods associated with them. Methods are like built-in tools that we can use to interact with an object (or apply to other objects!)

2.5.1 String Methods

Strings have a range of different methods associated with them.

```
[ ]: 'We Need to Keep a LOW Profile'.lower()
```

```
[ ]: 'we need to keep a low profile'
```

```
[ ]: 'I am not shouting!'.upper()
```

```
[ ]: 'I AM NOT SHOUTING!'
```

```
[ ]: 'This is our last method example'.startswith('This')
```

```
[ ]: True
```

```
[ ]: # Finally a new addition to Python is 'F-Strings'. F-Strings provide a nice␣
      ↪easy way of achieving what
     # `.format` does above but cleaner...

     age= 21
     name = 'Jeb'
     years_work_left = 50

     test_fstring = f'Meet {name}. He is currently {age} years old, and will retire␣
      ↪at the age of {age+years_work_left}.'
```

```
Meet Jeb. He is currently 21 years old, and will retire at the age of 71.
```

2.6 Exercises 1

Take a look at section 1 of the exercises sheet. Complete the tasks before moving on.

2.7 Lists

Lists are an important data structure because they allow us to collect values together in a way that retains their order. We can then access the values in the list, run through them one by one etc.

Lists are created using square brackets [] with values inside the brackets being seperated by a comma ,.

```
[ ]: list_of_numbers = [1,2,3]
     list_of_numbers
```

```
[ ]: [1, 2, 3]
```

```
[ ]: list_of_strings = ['Hello','Goodbye','Farewell']
     list_of_strings
```

```
[ ]: ['Hello', 'Goodbye', 'Farewell']
```

```
[ ]: simple_list = ['red','green','blue','yellow','black']
```

List indexing always begins at 0, which can be very confusing!

If we want the first item in the list we ask for the item at index 0

```
[ ]: simple_list[0]
```

```
[ ]: 'red'
```

Reverse list indexes always start at -1… because you can't have a -0

```
[ ]: simple_list[-1]
```

```
[ ]: 'black'
```

We can also access a range of values in a list by providing the start and end indexes.

Note that this can be tricky as it does not include the value of the end index. Think of it as [from:up to but not including].

```
[ ]: simple_list[1:4]
```

```
[ ]: ['green', 'blue', 'yellow']
```

Importantly after a list has been created we can change the values in it, for example we could change an item at a specific position

```
[ ]: simple_list[2] = 'monkey'
     simple_list
```

```
[ ]: ['red', 'green', 'monkey', 'yellow', 'black']
```

We can also add to a list using one of its built in methods `.append`

```
[ ]: simple_list.append('lion')
     simple_list
```

```
[ ]: ['red', 'green', 'monkey', 'yellow', 'black', 'lion']
```

Lists also have other methods such as sort. Note that sort does not create a new value, it simply sorts the original list.

[ ]:
```
simple_list.sort()
simple_list
```

[ ]:
```
['black', 'green', 'lion', 'monkey', 'red', 'yellow']
```

If we want to clear it, we simply overwrite the variable with a blank list

[ ]:
```
simple_list = []
simple_list
```

[ ]:
```
[]
```

2.8 Comparison and control flow

Python can 'test' whether particular expressions are `True` or `False`.

'Comparison' operators allow us to compare values, and Python will tell us if the statement is `True` or `False`

- `==` Equal to

- `!=` Not Equal to
- `>` Greater than

- `<` Less than

- `>=` Greater than or equal to

- `<=` Less than or equal to

2.8.1 Some examples of conditionals

[ ]:
```
3 * 5 == 15
```

[ ]:
```
True
```

[ ]:
```
"Fred" == "George"
```

[ ]:
```
False
```

[ ]:
```
"Fred" != "George"
```

[ ]:
```
True
```

Conditionals are critical in controlling the flow of your code. Flow control means that certain code is only run under certain conditions. - The main relevant keywords are… - `if` If a statement evaluates to `True`… do something. - `else` If your above statement evaluated to `False` do this instead. - `elif` 'Else if' - If your above `if` statement evaluated to `False`, check this statement instead

Flow control works using **blocks**. Normally a block begins with a conditional statement, with indented code beneath it indicating what code should run if the conditional evaluates to `True`.

A usual conditional block looks like this...

```
if True_statement:
    Do something
```

[ ]:
```python
# if statements execute if a condition is True and do nothing if False

test_value = 'Hello!'

if test_value == 'Hello!':
    print("This string says 'Hello'")
```

```
This string says 'Hello'
```

`else` statements tell the code what to do if the condition is `False`

[ ]:
```python
age = 35
retirement_age = 85

if age >= retirement_age:
    print('Time to RETIRE!')
else:
    print('Get back to work slacker!')
```

```
Get back to work slacker!
```

`elif` statements stand for 'else if' and only run if the above statement was False. They are useful for checking a value against different criteria.

[ ]:
```python
name = 'HAROLD'

if name == 'John':
    print('Hello John')
elif name.isupper():
    print('The name is upper case')
elif name.startswith('H'):
    print('Name starts with an H')
else:
    print('No criteria matched')
```

```
The name is upper case
```

2.9 Loops

Looping is a foundational concept in programming that allows your code to do a lot of useful things. The basic structure of a loop is...

```
for item in iterable:
    do something with item
```

Key Things - An `iterable` is a variable that can be broken down into individual items, often a list, but you can also iterate over strings and many other types of object. - In our example `item` refers to each individual object in our iterable. The word item could be anything, it is simply the name given to refer to the object in our code.

```
[ ]: the_gang = ['Scooby', 'Shaggy','Velma','Daphne']

     for item in the_gang:
         print(item)
```

```
Scooby
Shaggy
Velma
Daphne
```

We can use flow control to filter output

```
[ ]: # More importantly we can do things in the loop such as filter based on␣
      ↪conditions
     for item in the_gang:
         if item.startswith('S'):
             print(item)
```

```
Scooby
Shaggy
```

```
[ ]: # Or do some sort of transformation

     for item in the_gang:
         print('Team ' + item)
```

```
Team Scooby
Team Shaggy
Team Velma
Team Daphne
```

One key use of this is to filter through data and put the results we want into a list for later

```
[ ]: s_characters = []

     for item in the_gang:
         if item.startswith('S'):
             s_characters.append(item)

     s_characters
```

```
[ ]: ['Scooby', 'Shaggy']
```

2.10 Exercises 2

Take a look at section 2 of the exercises sheet. Complete the tasks. If there is time: - Work through Chapter 2 of the McLevey textbook (See Moodle for this week's Required Reading) OR - Work through the first two sections of Datacamp's Introduction to Python course.

```
[ ]:
```