Lesson 2 - Scripting with Python 3 - Basic Scripting

February 11, 2020

Lesson 2: Scripting with Python 3 - Basic Scripting

2.1 Reading User Input

Our scripts become more powerful when they can take in inputs and don't just do the same thing every time.

Let's learn how to prompt the user for input.

- 2.2.1 Accepting User Input Using `input()`

Let's build a script that requests three pieces of information from the user after the script runs.

Let's collect the following data:

- name - The user's name as a string
- birthdate - The user's birthdate as a string
- age - The user's age as an integer (we'll need to this to int)

Open a text editor and create a new file named `lab_age.py`

Add the code below:

[ ]:
```
name = input("What is your name? ")
birthdate = input("What is your birthdate? ")
age = int(input("How old are you? "))

print(f"{name} was born on {birthdate}")
print(f"Half of your age is {age / 2}")
```

Then run the script

(command-line)

```
$ python3 lab_age.py
What is your name? Monty
What is your birthdate Jan 1, 1991
How old are you? 29
Monty was born on Jan 1, 1991
Half of your age is 14.5
```

2.2 Function Basics

Being able to write code that we can call multiple times without repeating ourselves is one of the most powerful things that we can do.

We can do this using Functions.

- 2.2.1 Creating Functions

We can create functions in Python using the following:

- The `def` keyword
- The function name - lowercase starting with a letter or underscore (`_`)
- Left parenthesis (`(`)
- 0 or more argument names
- Right parenthesis (`)`)
- A colon `:`
- An indented function body

Here's an example of a function without an argument:

[2]:
```
def say_hello():
    print("Hello World!")
```

Let's call the function:

[3]:
```
say_hello()
```

```
Hello World!
```

- 2.2.2 Adding Function Arguments

If we want to define an argument, we need to put a variable name within the parentheses:

[4]:
```
def say_hello(name):
    print(f"Hello {name}!")
```

And then call the function passing the varaible name we specified - `name`:

[5]:
```
say_hello("Matt")
```

```
Hello Matt!
```

- 2.2.3 Returning a Value

We will usually want to have a return value from for our function unless the function is our "main" function or carries out a "side-effect" like printing.

If we don't explicitly declare a return value, then the result will be None.

We can declare a return value from a function using the `return` keyword:

[6]:
```
def add_numbers(a, b):
    return a + b
```

[7]:
```
add_numbers(1, 1)
```

[7]: 2

2.3 Using Functions in Scripts

Now that we know the structure of functions, let's utilize them in a script.

- 2.3.1 Encapsulating Behavior with Functions

To dig into functions, we're going to write a script that prompts the user for some information and calculates the user's Body Mass Index (BMI). That isn't a common problem, but it's something that makes sense as a function and doesn't require us to use language features that we haven't learned yet.

Here's the formula for BMI:

`BMI = (weight in kg / height in meters squared )`

For Imperial systems, it's the same formula except you multiply the result by `703`.

We want to prompt the user for their information, gather the results, and make the calculations if we can.

If we can't understand the measurement system, then we need to prompt the user again after explaining the error.

- 2.3.2 Gathering Info

Since we want to be able to prompt a user multiple times, we're going to package up our calls to `input` within a single function that returns a tuple with the user given information:

[23]:
```
def gather_info():
    height = float(input("What is your height? (inches or meters) "))
    weight = float(input("What is your weight? (pounds or kilograms) "))
    system = input("Are your measurements in metric or imperial units? ").
↪lower().strip()
    return (height, weight, system)
```

We're converting the `height` and `weight` into `float` values, and we're okay with a potential error if the user inputs an invalid number.

For the `system`, we're going to standardize things by calling `lower` to lowercase the input and then calling `strip` to remove the whitespace from the beginning and the end.

The most important thing about this function is the `return` statement that we added to ensure that we can pass the `height`, `weight`, and `system` back to the caller of the function.

- 2.3.3 Processing The Information (Calculating the BMI and Returning the Result)

Once we've gathered the information, we need to use that information to calculate the BMI.

Let's write a function that can do this:

[25]:
```
def calculate_bmi(weight, height, system='metric'):
    """
    Return the Body Mass Index (BMI) for the
    given weight, height, and measurement system.
    """
    if system == 'metric':
        bmi = (weight / (height ** 2))
    else:
        bmi = 703 * (weight / (height ** 2))
    return bmi
```

This function will return the calculated value, and we can decide what to do with it in the normal flow of our script.

The triple-quoted string we used at the top of our function is known as a "documentation string" or "doc string" and can be used to automatically generated documentation for our code using tools in the Python ecosystem.

- 2.3.4 Setting Up The Script's Flow

Our functions don't do us any good if we don't call them.

Now it's time for us to set up our scripts flow.

We want to be able to re-prompt the user, so we want to utilize an intentional infinite loop that we can break out of.

Depending on the `system`, we'll determine how we should calculate the BMI or prompt the user again. Here's our flow:

[ ]:
```
while True:
    height, weight, system = gather_info()
    if system.startswith('i'):
        bmi = calculate_bmi(weight, system='imperial', height=height)
        print(f"Your BMI is {bmi}")
        break
    elif system.startswith('m'):
        bmi = calculate_bmi(weight, height)
        print(f"Your BMI is {bmi}")
        break
```

```
    else:
        print("Error: Unknown measurement system. Please use imperial or metric.
↪")
```

- 2.3.5 Full Script

[ ]:
```
def gather_info():
    height = float(input("What is your height? (inches or meters) "))
    weight = float(input("What is your weight? (pounds or kilograms) "))
    system = input("Are your mearsurements in metric or imperial systems? ").
↪lower().strip()
    return (height, weight, system)

def calculate_bmi(weight, height, system='metric'):
    if system == 'metric':
        bmi = (weight / (height ** 2))
    else:
        bmi = 703 * (weight / (height ** 2))
    return bmi

while True:
    height, weight, system = gather_info()
    if system.startswith('i'):
        bmi = calculate_bmi(weight, system='imperial', height=height)
        print(f"Your BMI is {bmi}")
        break
    elif system.startswith('m'):
        bmi = calculate_bmi(weight, height)
        print(f"Your BMI is {bmi}")
        break
    else:
        print("Error: Unknown measurement system. Please use imperial or metric.
↪")
```

- 2.3.6 Run The Script

Now, let's save our code in a file named `lab_bmi.py` and run the script:

(command-line)

```
$ python3 lab_bmi.py
What is your height? (inches or meters) 71
What is your weight? (pounds or kilograms) 165
Are your mearsurements in metric or imperial systems? imperial
Your BMI is 23.01031541360841
```

2.4 Using Standard Library Packages

Up to this point, we've only used functions and types that are always globally available, but there are a lot of functions that we can use if we import them from the standard library.

- 2.4.1 Importing a Module

Here's how we can `import` the `time` package for use:

[8]:
```
import time
```

Importing the package allows us to access functions and classes that it defines.

We can do that by chaining off of the package name.

Let's call the `localtime` function provided by the `time` package:

[9]:
```
now = time.localtime()
now
```

[9]:
```
time.struct_time(tm_year=2020, tm_mon=1, tm_mday=28, tm_hour=20, tm_min=36,
tm_sec=2, tm_wday=1, tm_yday=28, tm_isdst=0)
```

Calling this function returns a `time.struct_time` to use that has some attributes that we can interact with using a period (`.`):

[10]:
```
now.tm_hour
```

[10]:
```
20
```

- 2.4.2 Importing Specifics from a Module

Since we're only using a subset of the functions from the time package, and it's a good practice to only import what we need.

We can import a subset of a module using the from statement combined with our import. The usage will look like this:

Format:

`from MODULE import FUNC1, FUNC2, etc...`

Example:

[14]:
```
from time import localtime, mktime, strftime
```

2.5 Working with Environment Variables

A common way to configure a script or CLI is to use environment variables.

Let's figure out how we can access environment variables from inside of our Python scripts.

- 2.5.1 The `os` package

By importing the `os` package, we're able to access a lot of miscellaneous operating system level attributes and functions, not the least of which is the `environ` object.

This object behaves like a dictionary, so we can use the subscript operation to read from it.

- 2.5.2 The `os.environ` attribute

Let's create a simple script that will read a `'STAGE'` environment variable and print out what stage we're currently running in:

[ ]:
```
import os

stage = os.environ["STAGE"].upper()

output = f"We're running in {stage}"

if stage.startswith("PROD"):
    output = "DANGER!!! - " + output

print(output)
```

We can set the environment variable when we run the script to test the differences:

- for staging:

(command-line)

```
$ export STAGE="staging"
$ python3 lab_check_stage.py
We're running in STAGING
```

- for production:

(command-line)

```
$ export STAGE="production"
$ python3 lab_check_stage.py
DANGER!!! - We're running in PRODUCTION
```

- 2.5.3 The `os.getenv` function (Handling a Missing Environment Variable)

Let's unset the `'STAGE'` environment variable

(command-line)

```
$ unset STAGE
$ echo $STAGE

$
```

Try running the script again:

(command-line)

```
$ python3 lab_check_stage.py
Traceback (most recent call last):
  File "lab_check_stage.py", line 3, in <module>
    stage = os.environ["STAGE"].upper()
  File "/usr/lib/python3.6/os.py", line 669, in __getitem__
    raise KeyError(key) from None
KeyError: 'STAGE'
```

This particular scenario happens when the `'STAGE'` environment variable is not set.

This potential `KeyError` is the biggest downfall of using `os.environ`, which is why we will usually use `os.getenv`

If the `'STAGE'` environment variable isn't set, we want to default to the value `'dev'`.

We can do that by using the `os.getenv` function:

[ ]:
```
import os

stage = os.getenv("STAGE", "dev").upper()

output = f"We're running in {stage}"

if stage.startswith("PROD"):
    output = "DANGER!!! - " + output

print(output)
```

Now if we run our script without a `'STAGE'` environment variable set, we won't have an error:

(command-line)

```
$ export STAGE="staging"
$ python3 lab_check_stage.py
We're running in DEV
```

2.6 Interacting with Files

It's pretty common to need to read the contents of a file in a script and it's easy to do that in Python.

Let's create a text file that we can read from called `xmen_base.txt` and the following lines:

```
Storm
Wolverine
Cyclops
Bishop
Nightcrawler
```

Now that we have a file to work with, let's experiment from the REPL before writing scripts that utilize files.

- 2.6.1 Opening and Reading a File

Before we can read a file, we need to open a connection to the file.

Let's open the `xmen_base.txt` file to see what a file object can do

— **2.6.1.1 The `open()` function** (command-line)

```
$ python3
Python 3.6.8 (default, Oct  7 2019, 12:59:55)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> xmen_file = open('xmen_base.txt', 'r')
>>> xmen_file
<_io.TextIOWrapper name='xmen_base.txt' mode='r' encoding='UTF-8'>
>>>
```

The `open` function allows us to connect to our file by specifying the path and the mode.

We can see that our `xmen_file` object is an `_io.TextIOWrapper` so we can look at the documentation to see what we can do with that type of object.

— **2.6.1.2 The `read()` function** There is a `read` function so let's try to use that:

(REPL)

```
>>> xmen_file.read()
'Storm\nWolverine\nCyclops\nBishop\nNightcrawler'
>>> xmen_file.read()
''
>>>
```

`read` gives us all of the content as a single string, but notice that it gave us an empty string when we called the function as second time.

That happens because the file maintains a cursor position and when we first called `read`, the cursor was moved to the very end of the file's contents.

— **2.6.1.3 The `seek()` function** If we want to reread the file we'll need to move the beginning of the file using the seek function like so:

(REPL)

```
>>> xmen_file.read()
'Storm\nWolverine\nCyclops\nBishop\nNightcrawler'
>>> xmen_file.read()
''
>>> xmen_file.seek(0)
0
>>> xmen_file.read()
'Storm\nWolverine\nCyclops\nBishop\nNightcrawler'
>>> xmen_file.seek(6)
6
>>> xmen_file.read()
'Wolverine\nCyclops\nBishop\nNightcrawler'
>>>
```

By seeking to a specific point of the file, we are able to get a string that only contains what is after our cursor's location.

— **2.6.1.4 Reading through file content using a `for` loop** Another way that we can read through content is by using a `for` loop:

(REPL)

```
>>> xmen_file.seek(0)
0
>>> for line in xmen_file:
...     print(line, end="")
...
Storm
Wolverine
Cyclops
Bishop
Nightcrawler
>>>
```

Notice that we added a custom end to our printing because we knew that there were already newline characters (`\n`) in each line.

— **2.6.1.5 The `close()` function (Closing file connection)** Once we're finished working with a file, it is import that we close our connection to the file using the `close` function:

(REPL)

```
>>> xmen_file.close()
>>> xmen_file.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.
>>>
```

- 2.6.2 Creating a New File and Writing to it

Now we know the basics of reading a file but we're also going to need to know how to write content to files.

— **2.6.2.1 Creating a copy of a file** Let's create a copy of our xmen file that we can add additional content to:

(REPL)

```
>>> xmen_base = open('xmen_base.txt')
>>> new_xmen = open('new_xmen.txt', 'w')
>>>
```

We have to reopen our previous connection to the `xmen_base.txt` so that we can read it again.

We then create a connection to a file that doesn't exist yet and set the mode to w, which stands for "write".

— **2.6.2.2 The `read()` and `write()` functions (Populating a New File)** The opposite of the read function is the write function, and we can use both of those to populate our new file:

(REPL)

```
>>> new_xmen.write(xmen_base.read())
43
>>> new_xmen.close()
>>> new_xmen = open(new_xmen.name, 'r+')
>>> new_xmen.read()
'Storm\nWolverine\nCyclops\nBishop\nNightcrawler'
>>>
```

We did quite a lot there, let's break that down:

1. We `read` from the base file and used the return value as the argument to `write` for our new file.
2. We closed the new file.
3. We reopened the new file, using the `r+` mode which will allow us to read and write content to the file.
4. We `read` the content from the new file to ensure that it wrote properly.

— **2.6.2.3 Overwriting Contents of a File** Now that we have a file that we can read and write from let's add some more names:

(REPL)

```
>>> new_xmen.seek(0)
0
>>> new_xmen.write("Beast\n")
6
>>> new_xmen.write("Phoenix\n")
```

```
8
>>> new_xmen.seek(0)
0
>>> new_xmen.read()
'Beast\nPhoenix\ne\nCyclops\nBishop\nNightcrawler'
>>>
```

What happened there?

Since we are using the `r+` we are overwriting the file on a per character basis since we used `seek` to go back to the beginning of the file.

If we reopen the file in the `w` mode, the pre-existing contents will be truncated.

- 2.6.3 Appending to a File

A fairly common thing to want to do is to append to a file without reading its current contents.

This can be done with the `a` mode.

Let's `close` the `xmen_base.txt` file and reopen it in the `a` mode to add another name without worrying about losing our original content.

This time, we're going to use the `with` statement to temporarily open the file and have it automatically closed after our code block has executed:

(REPL)

```
>>> xmen_file.close()
>>> with open('xmen_base.txt', 'a') as f:
...     f.write('Professor Xavier\n')
...
17
>>> f = open('xmen_base.txt', 'a')
>>> with f:
...     f.write("Something\n")
...
10
>>> exit()
```

To test what we just did, let's `cat` out the contents of this file:

(command-line)

```
$ cat xmen_base.txt
Storm
Wolverine
Cyclops
Bishop
NightcrawlerProfessor Xavier
Something
```

2.7 File Parsing Libraries (csv, json, yaml, xml)

- 2.7.1 What is data serialization?

Data serialization is the process of converting structured data to a format that allows sharing or storage of the data in a form that allows recovery of its original structure.

In some cases, the secondary intention of data serialization is to minimize the data's size which then reduces disk space or bandwidth requirements.

- 2.7.2 Flat vs. Nested data

Before beginning to serialize data, it is important to identify or decide how the data should be structured during data serialization - flat or nested.

The differences in the two styles are shown in the below examples.

— 2.7.2.1 Flat style:

```
{ "Type" : "A", "field1": "value1", "field2": "value2", "field3": "value3" }
```

— 2.7.2.2 Nested style:

```
{
    "A": { "field1": "value1", "field2": "value2", "field3": "value3" }
}
```

- 2.7.3 CSV file (flat data)

The CSV module in Python implements classes to read and write tabular data in CSV format.

Simple example for Reading:

[ ]:
```python
# Reading CSV content from a file
import csv
with open('/tmp/file.csv', newline='') as f:
    reader = csv.reader(f)
    for row in reader:
        print(row)
```

Simple example for Writing:

[ ]:
```python
# Writing CSV content to a file
import csv
with open('/temp/file.csv', 'w', newline='') as f:
    writer = csv.writer(f)
    writer.writerows(iterable)
```

- 2.7.4 JSON file (nested data)

Python's JSON module can be used to read and write JSON files. Example code is below.

Reading:

```
[ ]: # Reading JSON content from a file
import json
with open('/tmp/file.json', 'r') as f:
    data = json.load(f)
```

Writing:

```
[ ]: # Writing JSON content to a file using the dump method
import json
with open('/tmp/file.json', 'w') as f:
    json.dump(data, f, sort_keys=True)
```

- 2.7.5 YAML (nested data)

There are many third party modules to parse and read/write YAML file structures in Python.

One such example is below.

```
[ ]: # Reading YAML content from a file using the load method
import yaml
with open('/tmp/file.yaml', 'r', newline='') as f:
    try:
        print(yaml.load(f))
    except yaml.YAMLError as ymlexcp:
        print(ymlexcp)
```

- 2.7.6 XML (nested data)

XML parsing in Python is possible using the xml package.

Example:

```
[ ]: # reading XML content from a file
import xml.etree.ElementTree as ET
tree = ET.parse('country_data.xml')
root = tree.getroot()
```