Session 3 File operations and functions

Learning outcomes

- Understand what functions are
- Write a function
- Understand how files and file path works
- Open and close a file
- Read and write to a file

Recap from last session

- What is a list?
- What is an index?
- How to access to the last index of a list?
- How do you evaluate if two variables are equal?
- How to use a for loop to iterate over a list?
- How to use conditional (if/if else) statements?

Introduction to functions

During this course you might hear the words "function". A function is a block of code with a name assigned to it. Functions are really handy, because you only have to write them out in full once, and then you can use them again and again anywhere in your program by "calling" them (more on what that means in a bit).

The concept is a little bit like a paragraph. If you think back to English class in school, a paragraph is "a distinct section of a piece of writing, usually dealing with a single theme and indicated by a new line, indentation, or numbering" (Google Dictionary). In Python, a function is a really similar idea – it's a distinct block of our code that deals with a single task/theme, and is formatted in a particular way. Just like a paragraph, a function can be one statement or many statements long.

Here's an example of a very simple function:

```
def hello_world():
    print("Hello World!")
```

Any time you write a function in Python, it requires the following:

- `def` at the very beginning, so that Python knows the indented code below is part of a function (you're "defining" your function below this line).
- A unique name, with no spaces in it. It's important that you don't have two functions with the exact same name in your code. You also don't want to name your function "function", "sum" or "python" etc. If you name your function after a Python library or package or built-in function, Python will get really confused and your code won't run. Fortunately most IDEs are be able to pick up on the use of reserved words and will highlight the offending lines for you to edit.
- You'll also need to put brackets and a colon right after the name.

Notice how, after the first line of code in a function, the other lines are indented. In Python, this is very important – it's how the code interpreter in your computer knows what's part of your function and what isn't.

Calling a function

A function by itself doesn't actually do anything. You have to "call" it by its name, to run the code inside it. Calling a function is really easy! All you have to do is type the name of your function with `()` after it, like this:

```
def hello_world():
    print("Hello World!")

hello_world()
```

Notice that the function call is **not** indented, because it's not part of the function definition!

Task:

1. Create a new file called `my_first_function.py` and write a function that prints your name to the terminal, instead of `Hello World!`
2. Add another line of code so that your function also prints the answer to `2 + 2`

Arguments

We've all had those moments when our computer says "no" to our code and we say "but my code looks good! Why isn't it working?!" – but that's not the kind of argument we're talking about here! Every function you write in Python will have `()` after its name.

Sometimes, you might want or need to put some values/variables/parameters between those brackets, so that they can be used inside your function. The values inside those brackets are called "arguments". Sometimes you might seem them referred to as "args". Depending on the function you're using, you might have no arguments, one argument, or more than one argument. If you're writing your own function, you get to define this!

Let's use the built-in Python function `max()` as an example. We don't have to write the code for it because the lovely people who came up with Python have done that for us already. From Python's documentation, we know that `max()` can take 2, 3, or more arguments (you can read more about this function and how it works here).

Task:

1. Create a new file called `my_arguments.py`, paste the code below into it, and run it. What do you think `max()` is doing when it's given 2, 3 and 4 arguments?

```
print(max(1, 2))       # two arguments
print(max(1, 2, 3))    # three arguments
print(max(1, 2, 3, 4)) # four arguments
```

More fun with functions

Here's a function that adds two numbers together and prints out the answer. Remember, it doesn't actually do anything until we call the function using the statement `add_two_numbers()`.

```
def add_two_numbers():
    number1 = 1
    number2 = 2
    answer = number1 + number2
    print("{} plus {} is {}".format(number1, number2, answer))

add_two_numbers()
```

Here's another function that adds two numbers together. Notice how this function takes two arguments: `number1` and `number2`.

```
def add_two_numbers_from_args(number1, number2): # These can only be variable names
    answer = number1 + number2
    print("{} plus {} is {}".format(number1, number2, answer))
```

```
add_two_numbers_from_args(5, 10) # These can be variable names or actual numbers
```

Task:

1. Copy the code immediately above into your `my_arguments.py` file, and run it.
2. What happens when you change the values of the arguments you used when you called the function?
3. What do you think happens if you don't put 2 arguments in when you call the function `add_two_numbers_from_args` ?

You must use variable names for your arguments when you're defining your function (you can't put the actual values of the numbers or strings there). This is so you can re-use your functions, because the actual values for the arguments are given when you call the function, just like you see in `add_two_numbers_from_args(5, 10)` . Python is smart enough to know that when you call that function, the first value inside the bracket is number1 and the second value is number2.

Returning stuff from functions

You can use functions to do things like crunch numbers or manipulate data, and then "return" information from your function so that you can use it somewhere else in your program. Functions can return a number, a string, a list, or even another function. Today, we're going to learn how to return a simple number value from your function.

Let's have a look at the code below:

```
def add_two_numbers_and_return_value():
    number1 = 1
    number2 = 2
    answer = number1 + number2
    return answer # answer = 3

returned_value = add_two_numbers_and_return_value()
print(returned_value) # 3
```

Task:

1. What do you think `returned_value` will be if you don't have a return line in your function? Copy the code from the example above into your `my_arguments.py` file and run the file. Then remove only the line return answer and see what happens.

Best practice tips for writing functions:

- Try to give your functions descriptive names so it's easier to understand what they do just by looking at the name (you can also use comments to help with this).
- Try not to write giant functions! It's best to keep each function related to a specific task – this makes them easier to reuse.
- Sometimes it helps to sketch out on paper what you're trying to do before you start coding.

What is a file?

Data scientists often work with data through files and it is important to understand what a file is. A file is a set of bytes used to store data. The data is organised in a specific format and can be anything as simple as a text file or a program executable.

Files are usually composed of 3 main parts:

1. **Header**: metadata about the contents of the files (e.g file name, size, type etc.)
2. **Data**: contents of the file
3. **End of file (EOF)**: special character to indicate the end of the file.

The data representation is typically specified by an extension. You may have seen file extensions such as `.csv` or `.txt` , and there are many other file extensions.

Task:

1. Create a text file called `hello.txt`, add some text (e.g. `hello world!`) to the file and save it.
2. Create a Python file called `file_practice.py` in the same directory, and add the following code:

```
import os
print(os.stat("hello.txt"))
```

3. Run your python file and see what is printed in your terminal.

File Paths

When you access a file from Python, you need to know the path to the file, which is typically a string which represents the location of a file (e.g `/Users/user_name/Desktop/hello.txt`). The file path consists of 3 parts.

1. **Folder Path**: the file folder location on the file system where subsequent folders are separated by a forward slash `/` (Unix) or backslash `\` (Windows)
2. **File Name**: the actual name of the file
3. **Extension**: the end of the file path with a period (.) to indicate the file type

There are mainly two types of paths: relative paths and absolute paths.

- An *absolute* path specifies the location of a file or directory from the root directory, which is a complete path from the start of the actual file system. For example, `/Users/user_name/Desktop/hello.txt` is an absolute path.
- A *relative* path specifies the location of a file relative to the current working directory. For example, if your current working directory is in `/Users/user_name/`, a relative path to `hello.txt` would be `./Desktop/hello.txt`. You sometimes see `.` or `..` in part of a file path. `.` represents the current working directory, and `..` represents the parent directory.

Task:

1. In your python file, write the following code and run it to see the output.

```
import os
print(os.getcwd())
```

where `getcwd` means "get the current working directory".

Opening and closing a file in Python

Before running any examples in this section, you'll need to complete the following task.

Task:

1. Create a file called `dog_breeds.txt` in the same directory that your python file is located in.
2. Insert the following content into the file (by copy and pasting):

```
Pug
Jack Russell Terrier
English Springer Spaniel
German Shepherd
Staffordshire Bull Terrier
Cavalier King Charles Spaniel
Golden Retriever
West Highland White Terrier
Boxer
Border Terrier
```

When you want to work with a file, the first thing to do is to open it. This is done by invoking the `open()` built-in function. `open()` has a single required argument that is the path to the file. `open()` has a single return, the file object

```
file = open('dog_breeds.txt')
```

After you open a file, the next thing to learn is how to close it. You should **always** make sure that an open file is properly closed!

It's important to remember that it's your responsibility to close the file. In most cases, upon termination of an application or script, a file will be closed eventually. However, there is no guarantee when exactly that will happen. This can lead to unwanted behavior including resource leaks. It's also a best practice to make sure that your code behaves in a way that is well defined and reduces any unwanted behavior.

The recommended way to close a file is to use the `with` statement:

```
with open('dog_breeds.txt') as reader:
    # Further file processing goes here
```

The `with` statement automatically takes care of closing the file once it leaves the `with` block, even in cases of error. It is highly recommended that you use the `with` statement as much as possible, as it allows for cleaner code and makes handling any unexpected errors easier for you.

Most likely, you'll also want to use the second positional argument, called *mode*. This argument is a short string that represents how you want to open the file. The default and most common is `r`, which represents opening the file in read-only mode as a text file:

```
with open('dog_breeds.txt', 'r') as reader:
    # Further file processing goes here
```

Other options for modes can be in Python documentation, but the most commonly used ones are the following:

Character	Meaning
`r`	Open for reading (default)
`w`	Open for writing, overwriting the file first
`rb` or `wb`	Open in binary mode (read/write using byte data)

Reading Opened Files

Once you've opened up a file, you'll want to read or write to the file. First, let's cover reading a file. There are multiple methods that can be called on a file object to help you out:

Method	What It Does
`.read(size=-1)`	This reads from the file based on the number of size bytes. If no argument is passed or None or -1 is passed, then the entire file is read.
`.readline(size=-1)`	This reads at most size number of characters from the line. This continues to the end of the line and then wraps back around. If no argument is passed or None or -1 is passed, then the entire line (or rest of the line) is read.
`.readlines()`	This reads the remaining lines from the file object and returns them as a list.

Using the same dog_breeds.txt file you used above, let's go through some examples of how to use these methods. Here's an example of how to open and read the entire file using `.read()`:

```
with open('dog_breeds.txt', 'r') as reader:
    # Read & print the entire file
    print(reader.read())
```

You should see the following text printed to the screen when running the above program.

```
Pug
Jack Russell Terrier
English Springer Spaniel
German Shepherd
Staffordshire Bull Terrier
Cavalier King Charles Spaniel
Golden Retriever
West Highland White Terrier
Boxer
Border Terrier
```

Here's an example of how to read 5 bytes of a line each time using the Python `.readline()` method:

```
with open('dog_breeds.txt', 'r') as reader:
    # Read & print the first 5 characters of the line 5 times
    print(reader.readline(5))
    print(reader.readline(5))
    print(reader.readline(5))
    print(reader.readline(5))
    print(reader.readline(5))
```

which gives the following output

```
Pug
Jack
Russe
ll Te
rrier
```

Here's an example of how to read the entire file as a list using the Python `.readlines()` method:

```
with open('dog_breeds.txt') as f:
    lines = f.readlines()

print(lines)
```

which should print

```
['Pug\n', 'Jack Russell Terrier\n', 'English Springer Spaniel\n', 'German Shepherd\n', 'Staffordshire Bull
Terrier\n', 'Cavalier King Charles Spaniel\n', 'Golden Retriever\n', 'West Highland White Terrier\n', 'Boxer\n',
'Border Terrier\n']
```

Iterating Over Each Line in the File

A common thing to do while reading a file is to iterate over each line. Here's an example of how to use the Python `.readline()` method to perform that iteration:

```
with open('dog_breeds.txt', 'r') as reader:
    # Read and print the entire file line by line
    line = reader.readline()
    while line != '':  # The EOF char is an empty string
        print(line, end='')
        line = reader.readline()
```

Which will print the following:

```
Pug
Jack Russell Terrier
English Springer Spaniel
```

```
German Shepherd
Staffordshire Bull Terrier
Cavalier King Charles Spaniel
Golden Retriever
West Highland White Terrier
Boxer
Border Terrier
```

Another way you could iterate over each line in the file is to use the Python `.readlines()` method of the file object. Remember, `.readlines()` returns a list where each element in the list represents a line in the file:

```
with open('dog_breeds.txt', 'r') as reader:
    for line in reader.readlines():
        print(line, end='')
```

which will print the following:

```
Pug
Jack Russell Terrier
English Springer Spaniel
German Shepherd
Staffordshire Bull Terrier
Cavalier King Charles Spaniel
Golden Retriever
West Highland White Terrier
Boxer
Border Terrier
```

However, the above examples can be further simplified by iterating over the file object itself:

```
with open('dog_breeds.txt', 'r') as reader:
    # Read and print the entire file line by line
    for line in reader:
        print(line, end='')
```

```
Pug
Jack Russell Terrier
English Springer Spaniel
German Shepherd
Staffordshire Bull Terrier
Cavalier King Charles Spaniel
Golden Retriever
West Highland White Terrier
Boxer
Border Terrier
```

Task:

1. Create a text file and add some texts in multiple lines.
2. Write a Python script to read and print the text.

Writing Files

Now let's dive into writing files. As with reading files, file objects have multiple methods that are useful for writing to a file:

Method	Meaning
`.write(string)`	This writes the string to the file.
	This writes the sequence to the file. No line endings are appended to each sequence item. It's up to you to

`.writelines(seq)`	add the appropriate line ending(s).

Here's a quick example of using `.write()` and `.writelines()`:

```
with open('dog_breeds.txt', 'r') as reader:
    # Note: readlines doesn't trim the line endings
    dog_breeds = reader.readlines()

with open('dog_breeds_reversed.txt', 'w') as writer:
    # Alternatively you could use
    # writer.writelines(reversed(dog_breeds))

    # Write the dog breeds to the file in reversed order
    for breed in reversed(dog_breeds):
        writer.write(breed)
```

Task:

1. Write a Python script to write additional text to a text file.

Homework

1. You now know enough Python to create an interactive program. Let's create a quiz app! The app should provide 3-5 questions (or even more) about anything (e.g how old is the Queen?), and prompts the user to input an answer. Based on the input, your app should tell the user whether their answer is correct or not (hint: if/else statement). Make sure to create your own functions in order to make your code more modular. The app can be as interactive/complicated as you wish.
2. Can you make your quiz app load all the questions and answers from a text file? How would your code recognise which line is a question and which one is an answer?
3. Play with some of the questions in https://www.w3resource.com/python-exercises/python-functions-exercises.php

As mentioned before, you can find more exercises in Codewars.

Reference

The content of this course is developed based on the following references:

- CFG Advanced Python course
- Real Python article
- GeeksforGeeks