Lesson 7: Files

Reading a File

Computers use file systems to store and retrieve data. Each file is an individual container of related information. If you've ever saved a document, downloaded a song, or even sent an email you've created a file on some computer somewhere. Even *script.ipynb*, the Python notebook you're editing in this learning environment, is a file.

So, how do we interact with files using Python? We're going to learn how to read and write different kinds of files using code.

Let's say we had a file called *real_cool_document.txt* with these contents:

In [ ]:
```
# real_cool_document.txt
Wowsers!
```

Python has a wonderful keyword, with. This keyword enables **context management**. Upon entry into a `with` block, variables have certain meaning.

> **Note:** Upon exit, certain operations take place. For file objects created by opening them, the file is automatically closed upon exit, even if there is an error. This is important. If your program raises an exception before you have a chance to close the file, it won't get closed and you could be in trouble. If you use context management, the file will still get closed. So here is an important tip:

Using this knowledge, we could read that file with the following script:

In [ ]:
```
# script.py
with open('real_cool_document.txt') as cool_doc:
  cool_contents = cool_doc.read()
print(cool_contents)
```

This opens a file object called `cool_doc` and creates a new indented block where you can read the contents of the opened file. We then read the contents of the file `cool_doc` using `cool_doc.read()` and save the resulting string into the variable `cool_contents`. Then we print `cool_contents`, which outputs the statement `Wowsers!`.

Iterating Through Lines

When we read a file, we might want to grab the whole document in a single string, like `.read()` would return. But what if we wanted to store each line in its own variable? We can use the `.readlines()` function to read a text file line by line instead of having the whole thing. Suppose

we have a file:

In [ ]:
```
# keats_sonnet.txt
To one who has been long in city pent,
'Tis very sweet to look into the fair
And open face of heaven,--to breathe a prayer
Full in the smile of the blue firmament.
```

In [ ]:
```
# script.py
with open('keats_sonnet.txt') as keats_sonnet:
  for line in keats_sonnet.readlines():
    print(line)
```

The above script creates a *temporary file* object called `keats_sonnet` that points to the file *keats_sonnet.txt*. It then iterates over each line in the document and prints the entire file out.

Reading a Line

Sometimes you don't want to iterate through a whole file. For that, there's a different file method, .readline(), which will only read a single line at a time. If the entire document is read line by line in this way subsequent calls to `.readline()` will not throw an error but will start returning an empty string (`""`). Suppose we had this file:

In [ ]:
```
# millay_sonnet.txt
I shall forget you presently, my dear,
So make the most of this, your little day,
Your little month, your little half a year,
Ere I forget, or die, or move away,
```

In [ ]:
```
# script.py
with open('millay_sonnet.txt') as sonnet_doc:
  first_line = sonnet_doc.readline()
  second_line = sonnet_doc.readline()
  third_line = sonnet_doc.readline()
  print(sonnet_doc.readline())
```

This script also creates a file object called `sonnet_doc` that points to the file *millay_sonnet.txt*. It then reads in the first line using `sonnet_doc.readline()` and saves that to the variable `first_line` . It then saves the second line (*So make the most of this, your little day,*) into the variable `second_line` and then prints it out.

Writing a File

By default, a file when opened with `open()` is only for reading (that is, the file cannot be changed). A second argument `'r'` is passed to it by default. To write to a file, first open the file with write permission via the `'w'` argument. Then use the `.write()` method to write to the file. If the file already exists, all prior content will be overwritten.

In [ ]:
```
with open('diary.txt','w') as diary:
  diary.write('Special events for today')
```

Writing to an opened file with the `'w'` flag overwrites all previous content in the file. To avoid this, we can append to a file instead. Use the `'a'` flag as the second argument to `open()`. If a file doesn't exist, it will be created for append mode.

> Note that we can use the `.write()` method to write strings to a file, but `.write()` *only* takes strings as arguments. You cannot pass numbers. They must be converted to strings first.

In [ ]:
```
with open('shopping.txt', 'a') as shop:
  shop.write('Tomatoes, cucumbers, celery\n')
```

> **Notation:** `\n` means "create new line here".

Here's the other possible strings that are used as flags:

string	meaning
`'r'`	open a text file for reading
`'w'`	create and open a text file for writing
`'a'`	append an existing text file
`'r+'`	open a text file for reading and writing
append `'b'` to any of the above	same as above, except for binary files

We will mostly be working with text files in the bootcamp, so the first three are the most useful. A big warning, though:

Trying to open an existing file with 'w' will wipe it out and create a new file.

User Input

Sometimes, we'll want to gather some user input, sometimes for just variable values or maybe the the name of a file. Either way, here's how to ask a user (a person) for some information via a prompt:

In [ ]:
```
input('What is your name?\n')
```

Just getting a user input usually isn't good enough. Let's save it:

In [ ]:
```
user_name = input("What is your name?\n")
print('Your name is %s, is that correct?' %(user_name))
```

In [ ]:
```
user_name = input("What is your name?\n")
answer = input('Your name is %s, is that correct?' %(user_name))
if answer=='yes':
    print("Excellent! Your name is %s!" %(user_name))
else:
    print("I'm sorry, I must have misheard. Can I call you %s anyways?" %(user_name))
```

In [ ]:
```
user_name = input("What is your name?\n")
with open('user-names.txt','w') as names:
    names.write(user_name+'\n')
```

> **Note:** When the `input()` function executes, program flow stops until the user has given input. The text or message displayed on the output screen to ask a user to enter an input value is optional i.e. the prompt, which will be printed on the screen is optional.

> **Note:** Whatever you enter as input, the `input()` function converts it into a string. Even if an integer value is entered, `input()` function converts it into a string. You need to explicitly convert it into an integer in your code using *typecasting*.

In [16]:
```
num = input("Gimme a number: ")
print(type(num))
blurb = int(num)
print("Is %d your number?" %(blurb))
print(type(blurb))
```

```
<class 'str'>
Is 12 your number?
<class 'int'>
```

Grabbing multiple inputs

Last lesson, we talked about some snazzy string methods. Well, one of them can help us get multiple user inputs at once: `.split()` .

In [18]:
```
num1, num2 = input("Enter two prime numbers (#, #, etc.): ").split()
```

In [23]:
```
num1, num2 = input("Enter two prime numbers (#, #, etc.): ").split()
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[23], line 1
----> 1 num1, num2 = input("Enter two prime numbers (#, #, etc.): ").split()

ValueError: too many values to unpack (expected 2)
```

In [22]:
```
x, y, z = input("Enter three prime numbers (# # #): ").split()
print(x)
print(y)
```

```
1
2
```

We can also just grab a whole list of inputs, like so:

In [26]:
```
some_data = list(map(int, input("Enter prime numbers (# # etc.): ").split()))
print("List of primes: ", some_data)
```

```
List of primes:  [1, 2, 3]
```

> Note the differences in user input: the first two ways required commas, but the last doesn't.

Proceed to Assignment 7.