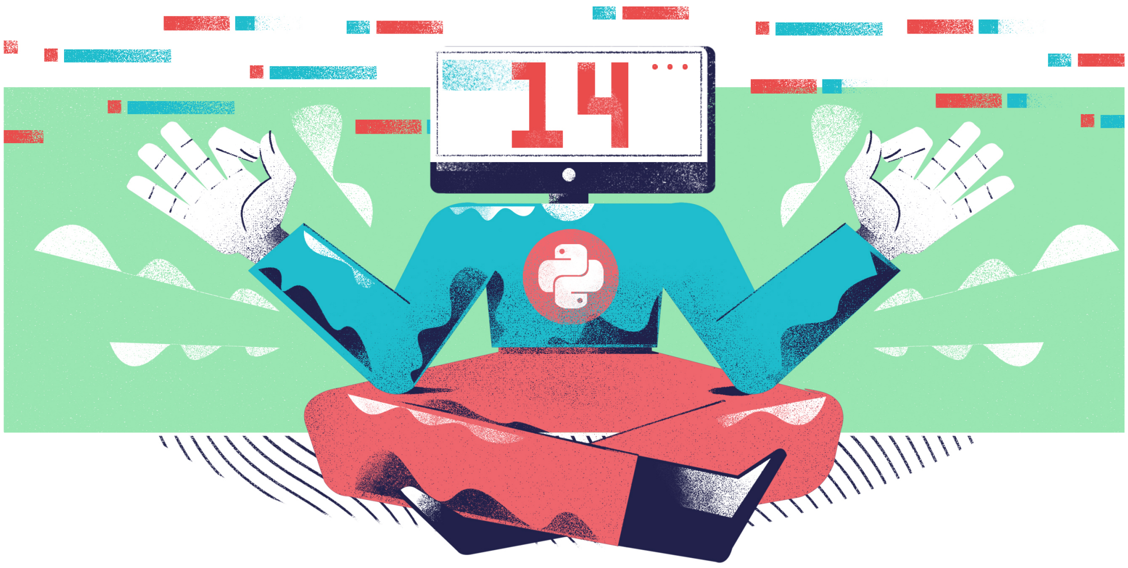
SPLITTING OUR CODE

Day 14: Working with Files



Welcome to day 14 of the 30 Days of Python series. Today is an exciting one, because we're going to learn how to manipulate files.

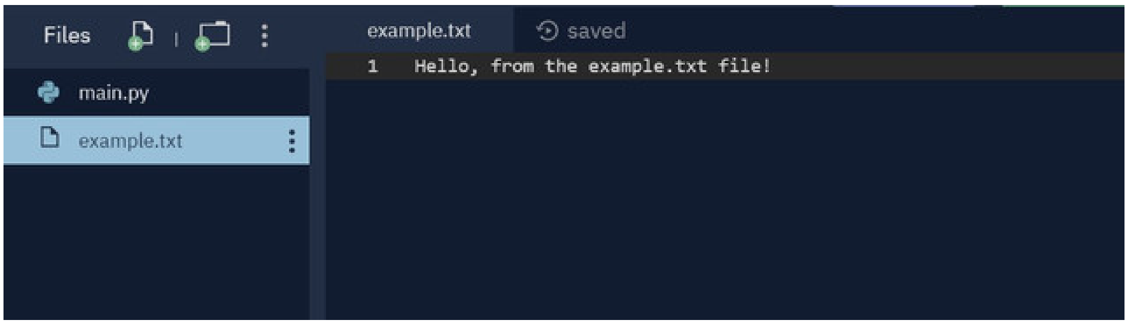
This opens up a lot of doors for us, because by the end of this post, you'll be able to permanently store information. Up until now, anything we do in our programs is lost the second the program closes. Let's fix that!

The `open` function

The first tool we need to start working with files is the `open` function. When we call `open`, it's going to gives us back a means of accessing the data inside a file we specify.

We can actually pass a lot of arguments to `open`, which change its behaviour, but we're going to start with just one: the name of the file we want to access.

First, we need to create a file to access. The file we're going to be working with to start is a simple text file which I've added to my repl alongside the `main.py` file.



```
Hello, from the example.txt file!
```

The file is called `example.txt`, and its content is a short message in plain text:

```
Hello, from the example.txt file!
```

Accessing this file is actually fairly simple. All we need to do is write the following in in our `main.py` file:

```python
example_file = open("example.txt")
```

Because the `example.txt` file is in the same directory (folder) as our `main.py` file, we can just write the name of the file, along with its file extension.

We could have also written this:

```python
example_file = open("./example.txt")
```

Here the `.` means "the current directory", so we're saying the `example.txt` file in the current directory.

We could also have specified a complete path to this file if we wanted to. In my case, the full path to the file on repl it is:

```
/home/runner/AnotherHuskySyntax/example.txt
```

Each of these versions has its uses, so it's good to know you have options for identifying the files you want to work with.

By default, when access a file with `open` we're going to be given read access to the file, which means we can get data out of the file, but we can't modify the file.

We can see the file's contents by calling the `read` method on the thing `open` returned for us.

```
example_file = open("example.txt")
print(example_file.read())
```

If we run this code, we'll see the little greeting we wrote in `example.txt` printed to the console!

We now need to do one final thing, and this is *really* important. We need to close the file.

```
example_file = open("example.txt")
print(example_file.read())
example_file.close()
```

You can see a few of the reasons for this in this StackOverflow post.

Opening files in different modes

Now that we're able to read some data from a file, let's talk about another one of the arguments we can pass to `open` : the mode.

By default, `open` is going to open our files in read mode, which is why we only have read access. This mode is denoted by the string, `"r"` .

If we wanted to be explicit (generally a good thing), we could rewrite our code above like this:

```python
example_file = open("example.txt", "r")
print(example_file.read())
example_file.close()
```

We could also use a keyword argument if we wanted:

```python
example_file = open("example.txt", mode="r")
print(example_file.read())
example_file.close()
```

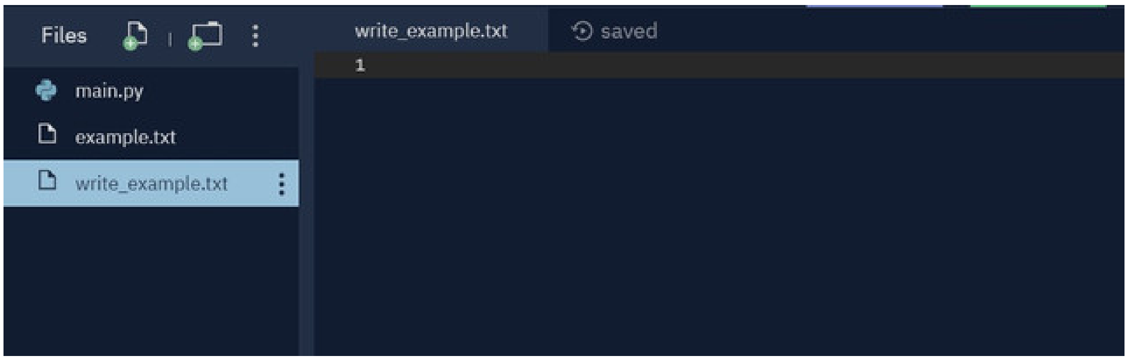
These are all functionally equivalent.

However, as you might expect, read mode is not the only way we can access files. We also have write mode, denoted by the string, `"w"` .

Let's test this out by creating a new file. When we use write mode, Python is going to create the file for us if it doesn't already exist, so we can write something like this:

```python
write_file = open("write_example.txt", "w")
write_file.close()
```

If we run this code, we can see we have a new empty file called `write_example.txt` .



We can write information to this file if we want to by calling the `write` method.

```python
write_file = open("write_example.txt", "w")
```

```
write_file.write("Welcome to the world, write_example.

txt!")
write_file.close()
```

There are plenty of other modes we can use, and you can find information about those in [the official documentation]().

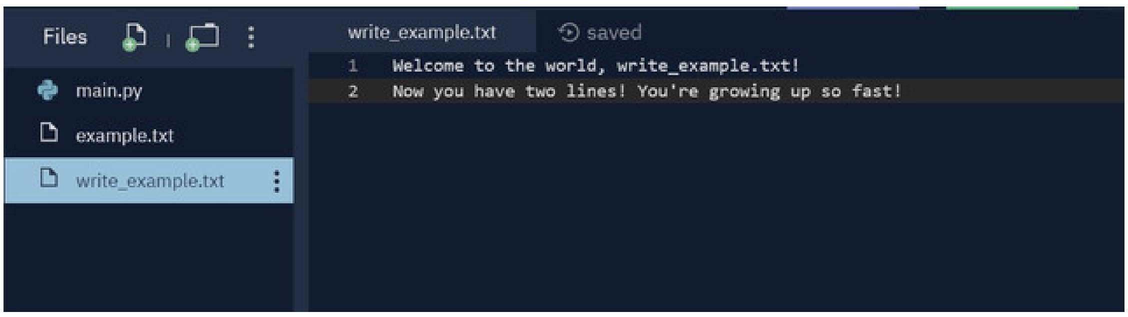
For now, we're just going to learn about one more: append mode. Append mode is another type of write mode, and is denoted by the string, `"a"` .

One thing we have to be aware of is that write mode truncates the file as soon as we open it. That means it deletes all of the data in the file. Append mode lets us write to the end of the file instead, so this can be useful when we just want to extend the file's contents.

We can write to a file opened in append mode in exactly the same way as with write mode.

```
write_file = open("write_example.txt", "a")
write_file.write("\nNow you have two lines! You're gro
wing up so fast!")
write_file.close()
```

Now our `write_example.txt` file looks like this:



```
Welcome to the world, write_example.txt!
Now you have two lines! You're growing up so fast!
```

Context managers for working with files

We've only accessed a few files so far, but always having to close the file after we're done working with it is already getting a little tedious.

Because this is something we have to do every single time we open a file, Python gives us a handy tool called a context manager, which handles these repetitive actions for us.

The syntax looks like this:

```
with open("example.txt", "r") as example_file:
        print(example_file.read())
```

This may look a little weird at first, but it's functionally the same as this:

```
example_file = open("example.txt", "r")
print(example_file.read())
example_file.close()
```

First we have the `with` keyword, which indicates to Python we're using a context manager. Next we call the `open` function, just like we did before, but in order to assign the result to a variable, we need to use this `as` keyword.

Everything we want to do while the file is open is placed in an indented block underneath this first line, and then when there's no more code in this block to run, Python closes the file for us.

It's important that we first learn and understand the longer version of the syntax, but in practice, you really should always be using the context manager syntax when working with files. Even the best of us can sometimes forget to close files when we're working with them manually, and context managers make that impossible.

Here are a few more examples so you can see the long version alongside the context manager equivalent.

This:

```
write_file = open("write_example.txt", "w")
write_file.write("Welcome to the world, write_example.
txt!")
write_file.close()
```

Can be rewritten as this:

```
with open("write_example.txt", "w") as write_file:
        write_file.write("Welcome to the world, write_
example.txt!")
```

And this version with the append mode:

```
write_file = open("write_example.txt", "a")
write_file.write("\nNow you have two lines! You're gro
wing up so fast!")
write_file.close()
```

Can be rewritten as this:

```
with open("write_example.txt", "a") as write_file:
        write_file.write("\nNow you have two lines! Yo
u're growing up so fast!")
```

CSV data

Now that we're familiar with how to do some basic file operations, let's talk about how to work with data in a common format: CSV.

CSV stands for comma separated values, and it's one of the simplest ways we can store data as plain text. As the name would imply, values are separated by commas, and usually this data is arranged like a table, where each row of values is on a different line in the file.

There are many different styles or "dialects" of CSV, but we're going to keep things simple and work with a version that just uses commas and line breaks.

Let's start by creating a new repl, and placing a file in our new repl called `iris.csv` , and we're going to put the following data inside:

```
sepal_length,sepal_width,petal_length,petal_width,spec
ies
5.1,3.5,1.4,0.2,Iris-setosa
4.9,3,1.4,0.2,Iris-setosa
4.7,3.2,1.3,0.2,Iris-setosa
4.6,3.1,1.5,0.2,Iris-setosa
5,3.6,1.4,0.2,Iris-setosa
7,3.2,4.7,1.4,Iris-versicolor
6.4,3.2,4.5,1.5,Iris-versicolor
6.9,3.1,4.9,1.5,Iris-versicolor
5.5,2.3,4,1.3,Iris-versicolor
6.5,2.8,4.6,1.5,Iris-versicolor
6.3,3.3,6,2.5,Iris-virginica
5.8,2.7,5.1,1.9,Iris-virginica
7.1,3,5.9,2.1,Iris-virginica
6.3,2.9,5.6,1.8,Iris-virginica
6.5,3,5.8,2.2,Iris-virginica
```

This is a small subset of a very famous data set used in machine learning called the [Iris flower data set](#).

It contains a number of measurements of different parts of various iris flowers, along with the particular species of iris those measurements were taken from.

In the data set above, we have 15 of the 150 items, and the top row contains the headers for our table of data, so that we know what each value refers to.

You can see the data a little better in the following table:

5.1 3.5 1.4 0.2 4.9 3 1.4 0.2 4.7 3.2 1.3 0.2 4.6 3.1 1.5 0.2 5 3.6 1.4 0.2 7 3.2 4.7 1.4 6.4 3.2 4.5 1.5 6.9 3.1 4.9 1.5 5.5 2.3 4 1.3 6.5 2.8 4.6 1.5 6.3 3.3 6 2.5 5.8 2.7 5.1 1.9 7.1 3 5.9 2.1 6.3 2.9 5.6 1.8 6.5 3 5.8 2.2

sepal_length	sepal_width	petal_length	petal_width	species
5.1	3.5	1.4	0.2	Iris-setosa
4.9	3	1.4	0.2	Iris-setosa
4.7	3.2	1.3	0.2	Iris-setosa
4.6	3.1	1.5	0.2	Iris-setosa
5	3.6	1.4	0.2	Iris-setosa
7	3.2	4.7	1.4	Iris-versicolor
6.4	3.2	4.5	1.5	Iris-versicolor
6.9	3.1	4.9	1.5	Iris-versicolor
5.5	2.3	4	1.3	Iris-versicolor
6.5	2.8	4.6	1.5	Iris-versicolor
6.3	3.3	6	2.5	Iris-virginica
5.8	2.7	5.1	1.9	Iris-virginica
7.1	3	5.9	2.1	Iris-virginica
6.3	2.9	5.6	1.8	Iris-virginica
6.5	3	5.8	2.2	Iris-virginica

Our goal in this section is going to be to take this set of CSV data, and create a list of dictionaries from it.

Step 1: Getting the data out of the file

First thing first, we need to open the file and get the CSV data out of it. Using the techniques we've already seen, we can do something like this:

```
with open("iris.csv", "r") as iris_file:
        iris_data = iris_file.read()
```

Step 2: Splitting the data into rows

Now if we were to print `iris_data` , we'd get the file's contents printed to the console.

The question is, how do get the individual lines from the file so we can start working with individual items?

Since the data is displayed on separate lines, we know one very important thing. Every line ends with a newline character to mark the line break. We can therefore split our data based on the `"\n"` character, giving us each line as a list.

```python
with open("iris.csv", "r") as iris_file:
        iris_data = iris_file.read().split("\n")
```

However, because this is a very common operation, Python gives us a tool to do pretty much the same thing. Instead of calling `read`, we can call `readlines`.

```python
with open("iris.csv", "r") as iris_file:
        iris_data = iris_file.readlines()
```

The main difference between the two is that `readlines` is going to preserve this `"\n"` character, so we're going to need to remember to trim it off.

Step 3: Creating our new list and trimming off the header row

Next, I'm going to create an empty list called `irises`, which is where we're going to place our final dictionaries.

```python
with open("iris.csv", "r") as iris_file:
        iris_data = iris_file.readlines()

irises = []
```

Now I'm going to iterate over the list in `iris_data` using a `for` loop. Remember that the first line isn't really data though. It's just the table headers. I'm therefore not going to iterate over all of the iris data, I'm going to iterate over a slice.

```
with open("iris.csv", "r") as iris_file:
        iris_data = iris_file.readlines()

irises = []

for row in iris_data[1:]:
        pass
```

Step 4: Splitting the rows into individual items

For each iteration, I'm going to start by stripping off the `"\n"` character, and splitting the string using a comma as the delimiting string.

To make it easier to refer to the values later on, I'm going to destructure the list we get back from `split`.

```
with open("iris.csv", "r") as iris_file:
        iris_data = iris_file.readlines()

irises = []

for row in iris_data[1:]:
        sepal_length, sepal_width, petal_length, petal
_width, species = row.strip().split(",")
```

Step 5: Creating a dictionary from each row

Now that we have all of our data assigned to these variables, we can construct our dictionary and append it to `irises`:

```
with open("iris.csv", "r") as iris_file:
        iris_data = iris_file.readlines()

irises = []

for row in iris_data[1:]:
        sepal_length, sepal_width, petal_length, petal
        sepal_length, sepal_width, petal_length, petal
_width, species = row.strip().split(",")

        iris_dict = {
                "sepal_length": sepal_length,
                "sepal_width": sepal_width,
                "petal_length": petal_length,
                "petal_width": petal_width,
                "species": species
        }

        irises.append(iris_dict)
```

If we wanted to be a little more succinct, we could define the dictionary when making the `append` call instead:

```
with open("iris.csv", "r") as iris_file:
        iris_data = iris_file.readlines()

irises = []

for row in iris_data[1:]:
        sepal_length, sepal_width, petal_length, petal
_width, species = row.strip().split(",")

        irises.append({
                "sepal_length": sepal_length,
                "sepal_width": sepal_width,
                "petal_length": petal_length,
                "petal_width": petal_width,
                "species": species
        })
```

Using the `dict` function

This is a perfectly good approach, but I want to show you another way using the `dict` function.

`dict` is an alternative method for creating a dictionary, and it's quite versatile in how we can call it. One way we can create a dictionary with dict is to pass it an iterable of iterables, where each of these with dict is to pass it an iterable of iterables, where each of these inner iterables contains a key and a value.

For example, let's say I have a list like this:

```
iris = [
        ("sepal_length", "5.1"),
        ("sepal_width", "3.5"),
        ("petal_length", "1.4"),
        ("petal_width", "0.2"),
        ("species", "Iris-setosa")
]
```

This list contains a number of two-element tuples, and we can think of these tuples as containing a key value pair. The first element in each tuple is the key, and the second element is the associated value.

If we pass this list to `dict`, Python is able to construct a dictionary for us, like this:

```
{
        "sepal_length": "5.1",
        "sepal_width": "3.5",
        "petal_length": "1.4",
        "petal_width": "0.2",
        "species": "Iris-setosa"
}
```

This is very useful for us, because we can create a structure very similar to this by using `zip`.

Instead of throwing away the header row, let's store its values and process them like the different lines of data.

```
with open("iris.csv", "r") as iris_file:
        iris_data = iris_file.readlines()

headers = iris_data[0].strip().split(",")
```

Now let's iterate over the rows again, but this time, let's match each header item to an value in a given row using `zip` .

```
with open("iris.csv", "r") as iris_file:
        iris_data = iris_file.readlines()

headers = iris_data[0].strip().split(",")
irises = []

for row in iris_data[1:]:
        iris = row.strip().split(",")
        iris_dict = dict(zip(headers, iris))

        irises.append(iris_dict)
```

With that, we have our CSV data in dictionary format!

Exercises

Rewrite the following piece of code using a context manager:

```
f = open("hello_world.txt", "w")
f.write("Hello, World!")
f.close()
```

Use append mode to write `"How are you?"` on the second line of the `hello_world.txt` file above.

Take the list of dictionaries we created from the Iris flower data set and write it to a new file in CSV format.

Project

Today is the end of the second week, so that means another end of week project!

This time we're going to be creating a reading list application so that users can store information about books they want to read. This should help solidify the concepts we've been learning about over the past week.

past week.

There are actually two versions of today's project, with different levels of difficulty.

I'd recommend you have a go at the regular version first, because the harder version builds on the regular version, adding some more complex functionality.

Good luck and happy coding!

Additional resources

If you interested in learning more about how to efficiently work with CSV data, we have a video you can watch dedicated to this topic.