teclado

WORKING WITH MULTIPLE FILES

Day 21: Splitting Code Into Multiple Files

←

→

Welcome to day 21 of the 30 Days of Python series! Today we'll learn about how to (and why to) split our Python code into multiple files.

First though, we'll quickly recap how imports work in Python. Let's get started!

Recap of imports in Python

Doing a quick recap is worthwhile in this case, because it's important we understand a few points before going off and creating lots of files for our programs!

If you'd like something a little more comprehensive, feel free to re-read the day 18 post where we first covered imports.

Importing adds a name to `globals()`

If you have a Python file and you type this:

```
import json

print(globals())
```

Then that displays the *names* currently in the global namespace. The module name will be in the global namespace, ready for you to use.

Importing allows us to access elements of the imported module

After importing, we can do something like this to access something inside a module:

```
import math

print(math.pi)  # 3.14
```

We've accessed the `pi` *name* inside the `math` module. In this case, that is the value of the mathematical constant, pi.

You can use the \`as\` keyword to give the imported module a different name in your code. I don't recommend you do this, although you'll see it done every now and then!

You can also use `*` to add *almost everything* from a module to your global namespace. This can "pollute" the global namespace, filling it with variables. Most of the time, it's strongly discouraged.

Why split your code into files?

Now that we've quickly recapped the topic of imports, let's talk about *why* we may want to split our code into multiple files!

The first question we get when we start discussing this is: "isn't it easier to keep the code in one file?".

After all, then you don't have to deal with imports!

Keeping your code in one file may make it easier to write (to begin with!), but as you write more, it'll make it much more difficult to read and modify.

In programming, readability and maintainability trump speed of writing any day!

Separating concerns and ease of organization

When we separate code into files, it's important we have a reason for putting some code in a certain file. Usually, we go by *concerns*. Code that does one thing goes into one file, and code that does something different goes into a different file.

For example, we might have one file for user interaction (prints and inputs), and another file for data storage (saving and retrieving things from a file).

At the moment the different concerns of a Python application may not be obvious to you. That's normal! Over time, you'll learn more about this. With more experience, it'll become clearer when things will benefit from getting split into files.

Separating files by concerns, assuming we give the files good names, also helps us find code more easily. If you have two fils called `data_storage.py` and `user_menu.py`, you know what you're going to find in each!

Using files and folders also does wonders for organisation. You could put files related to working with different types of data storage into one folder, for example.

Improved readability

As your files grow, they start getting more difficult to read. You may have many functions and variables, and finding things may require a lot of scrolling.

Modern editors have features to make it easier to find where things are defined, but this isn't a perfect solution.

Splitting your code into multiple files is a great way to produce smaller, more focused files. Navigating theses smaller files will be easier and so will understanding the content of each of these files.

Easier to reuse code

When you have multiple smaller, focused files, it's easier to reuse the contents of one file in multiple other files by importing.

It's also possible to have *too many* files! Don't aim to make your files as small as possible. Instead, make them focused.

If each file is concerned with one aspect of your application and every file has a good name, navigating your project will be easy!

Working with two files

Let's start off by going into a new Python project (or repl.it repl) and creating two files. Let's call them `main.py` and `myfile.py`.

Remember that repl.it will always run `main.py` when you press the "Run" button, as we mentioned in the day 0 post.

Because both files are in the same folder, you can import one from the other.

Let's type this in `main.py` :

```
import myfile
```

Note that we don't import `myfile.py` , but just `myfile` . Python does the rest!

What happens when you import

When we import a file, Python *runs* the file. It has to do that in order to determine what names exist in that file.

Then it makes that file available to us in `main.py` by putting a reference to the module in the global namespace.

Let's add something into `myfile.py` :

```
print("Hello, world!")
```

Now let's import the file from `main.py` :

```
import myfile

print("What's going on?")
```

Try running that.

You'll see that we get two things printed out:

```
Hello, world!
```

```
What's going on?
```

That's because the `myfile.py` file was executed. It contains a `print` call, so therefore Python printed stuff out!

It's rare that you'll write code like this. Instead, the files you import will normally contain **variables** and **functions**, so that when you import them nothing actually happens until you actually *use* those variables and functions.

Let's add something a little more realistic to `myfile.py` . Maybe something like this:

```python
def get_user_age():
    return int(input("Enter your age: "))
```

Now in `main.py` we can use the `get_user_age` function that we defined in `myfile.py` .

```python
import myfile

try:
    myfile.get_user_age()
except ValueError:
    print("That's not a valid value for your age!")
```

Great!

That's the gist of it: separate things into files, and import them!

Don't give your files the same names as built-in modules.

For example, if you create a new file and call it `json.py` , you'll get into trouble!

That's because if another file tries to `import json` , you won't be importing the built-in `json` module that lets us talk to JSON files.

You'd be importing your `json.py` file instead!

Python always looks in the project folder for imports *before* looking at the built-in or installed packages.

Your files work in the same way as modules

Everything we could do with external modules, we can do with our own files:

- Importing the whole file with `import myfile` and then referring to things as `myfile.x` .
- Importing specific things with `from myfile import x` .
- Aliased imports.
- We can do `from myfile mport *` (although it's discouraged).

Using files and folders

You can create folders for your files if you think that'll help with organisation. In my experience, it usually does.

For example, create a folder in your project called `user_interactions` and move `myfile.py` into it. Now your file/folder structure will look like this:

```
  - main.py
- user_interactions/
    | - myfile.py
```

From `main.py` you now have to use slightly different syntax to import `myfile.py` :

```
from user_interactions.myfile import get_user_age
```

```
try:
    get_user_age()
except ValueError:
    print("That's not a valid value for your age!")
```

When importing, the dot (`.`) means something like "inside".

In the example above, we're therefore importing `myfile` from *inside* `user_interactions` .

If you have multiple sub-folders, you will need to use multiple `.` to separate the different levels of folders and files, like this:

```
from folder.subfolder.module import something_in_the_module
```

When you're importing like this, you'll either import specific things like we've done, or you'll use aliased imports. Either of these two are good (the first is generally better):

- `from user_interactions.myfile import get_user_age`
- `import user_interactions.myfile as interactions`

You normally won't be doing anything like in the example below, because it can get quite long to type out:

```
import user_interactions.myfile

user_interactions.myfile.get_user_age()
```

You need to refer to the complete import when you want to use something, so you'd be typing `user_interactions.myfile` a lot in your code!

It's OK to do that if you won't be using the file much though.

Script mode vs. module mode

When we run a file (e.g. in repl.it, that's `main.py`), we say that file is ran in "script mode".

When we import a file, that file runs in "module mode".

At the moment our project structure is like this:

```
- main.py
- user_interactions/
    | - myfile.py
```

I'll delete the contents of both files, and I'll place this in `myfile.py` :

```
print(__name__)
```

And I'll do this in `main.py` :

```
import user_interactions.myfile

print(__name__)
```

You'll see this output:

```
user_interactions.myfile
__main__
```

Remember that when we import, we run the file. Therefore the first line of output belongs to `myfile.py` , and the second line of output belongs to `main.py` .

The file that we run always has a `__name__` variable with a value of `" main "` . That is simply how Python tells us that *we ran that file*.

Any file that doesn't have a `__name__` equal to `"__main__"` was imported.

Try moving things around and see how the output created by `myfile.py` changes!

Running code only in script mode

Sometimes we want to include some code in a file, but we only want that code to run if we executed that file directly—and not if we imported the file.

Since we know that `__name__` must be equal to `"__main__"` for a file to have been run, we can use an if statement.

We could type this in `myfile.py` :

```
def get_user_age():
    return int(input("Enter your age: "))

if __name__ == "__main__":
    get_user_age()
```

That could allow us to run `myfile.py` (we can't do that in repl.it without some extra configuration), and see if the `get_user_age()` function works.

That is one of the key use cases of this construct: to help us see whether the stuff in a file works when we normally don't want it to run.

Another use case is for files which you don't normally run yourself. Sometimes you may write a file that is for use by another program, for example.

Using this construct would allow you to run your file for testing, while not affecting its functionality when it's imported by another program.

Exercises

For today's (only) exercise, we're giving you a bunch of code that is all in one file.

Access the code here.

Your task is to split that code into multiple files. You can choose how many and which files you want to split the code into, but think about why you're putting each piece on code in each file!

Then when you're done, check out our exercise solution to see how we'd split the code, and why.

Project

For today's project, we'll be using a Python library to create some graphs! As part of this project, you'll have to structure your code well so that it doesn't become an unreadable mess.

Check out the project brief for more information on today's project.