teclado

WORKING WITH MULTIPLE FILES

# Day 18: Imports

Welcome to day 18 of the 30 Days of Python series! Today we're going to be talking about the very **import**ant topic of imports.

So far we've been working with the core Python syntax, and we've been using the so called built-in functions. However, there's a lot more to the Python standard library than just these functions, but to get access to it, we need to be able to import this functionality.

Importing will also let us use 3rd party libraries, which we'll cover later in the series.

## Basic imports

We're going to start off by importing the `math` module, which, as the name implies, contains a lot of useful tools for performing mathematical operations. It also includes several useful constants. You can find documentation here.

To import a module we need to write the `import` keyword, followed by the name of the module we want to import. To import the `math` module, we'd write this at the top of our code:

```
import math
```

Just like that, we have access to all of the functionality defined in the `math` module. In order to use these functions and variables, we just need to put `math.` before the function or variable name. This tells Python that we're requesting something located in the `math` module.

Let's see this in action with an example.

All the way back in day one of the series, we had the following exercise question:

Calculate and print the area of a circle with a radius of 5 units.

This would be a perfect application for the `math` module, as we can make use of the `math` module's `pi` constant:

```
import math

print(math.pi * 5**2)  # 78.53981633974483
```

This is a great deal more accurate than my original solution, which used `3.141` as the value for pi.

Now let's try a function instead. The `math` module has a function called `fsum` which is for adding together floating point numbers. You can certainly use regular old `sum` for this, but things get a little bit weird for certain values, because there's an inherent inaccuracy in floating point numbers.

Take the following example:

```
numbers = [0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1]
print(sum(numbers))  # 0.9999999999999999
```

That's right. Adding ten lots of `0.1` is not `1`. You'll get similarly dodgy results if you try adding `0.2` over and over again as well.

However, if we use the `fsum` function, mathematics goes right back to normal:

```
import math

numbers = [0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1]
print(math.fsum(numbers))  # 1.0
```

If you're interested to know why `0.1` produces this unusual result, I recommend you check out my post on the Decimal module. At the start I talk about how decimal numbers are represented in a computer, and why this leads to problems with floats.

## Style note

Make sure to always put your imports at the top of the file! This is not only good style, there are also good practical reasons for this, which we'll get to shortly.

It's also a good idea to put a space after you're done with your imports to separate them from the rest of your code.

## Importing specific things from modules

Let's say we're working on a project and I really only need the `pi` constant from the `math` module, and I plan to use it extensively in my code. There's no real problem with using `import math` and then referring to the constant using `math.pi`, but this is going to get a little tedious.

In this situation it would be a lot better if we could just use `pi` in our code, and we can do this by importing just the `pi` constant from the `math` module. The syntax for this is as follows:

```
from math import pi
```

We can import several things from a module by separating the names with commas. Let's say I also want the tau constant:

```
from math import pi, tau
```

Now we can refer to `pi` or `tau` directly instead of using `math.pi` and `math.tau`. However, we no longer have access to the rest of the `math` module using this approach, so we can't call `math.fsum` unless we also import `math`.

In some modules there are also things we can only access through this specific import syntax.

For example, let's say we want to work with `tkinter`, because we've decided we want to make an application with a graphical user interface (a GUI).

The `tkinter` library includes a set of themed widgets that we can use in our applications, but in order to use them, we have to import `ttk` from the tkinter module.

```
from tkinter import ttk
```

## Modules and namespaces

Now that we've seen two different types of import, it's worth talking about what happens when we import a module, and why we access things in different ways depending on how we import them. For example, why do we need to put this `math.` in front of `pi` when importing it one way, but not the other?

Let's use the same technique as we did in day 13 and look at what gets defined in the global scope when we use the two different import methods.

First, let's try `import math`. We can see what is in the global namespace like so:

```
import math

print(globals())
```

What we get is a dictionary like this:

```
{
        '__name__': '__main__',
        '__doc__': None,
        '__package__': None,
        '__loader__': <_frozen_importlib_external.SourceFileL
oader object at 0x7f23b99febb0>,
        '__spec__': None,
        '__annotations__': {},
        '__builtins__': <module 'builtins' (built-in)>,
        '__file__': 'main.py',
        '__cached__': None,
        'math': <module 'math' from '/usr/local/lib/python3.
8/lib-dynload/math.cpython-38-x86_64-linux-gnu.so'>,
8/lib-dynload/math.cpython-38-x86_64-linux-gnu.so'>
}
```

Just like before we have a lot of stuff defined here when we start our program, but right at the end we have a name called `math`. Associated with this name `math` we have something called a module object, and in this case we can see that `Python` has found the code for this module somewhere in our Python installation.

When we import a module, Python runs the module and creates this module object so that we can access the things inside of it. In order to indicate we want to do something with the module, we have to first access the module using the name it's assigned to (in this case, `math`), and then we can access the things inside using the dot notation.

Now let's look at what happens when we import specific items from the `math` module. Once again I'm going to use the `pi` and `tau` constants, so now our code is going to look like

```
from math import pi, tau

print(globals())
```

And we get the following output:

```
{
        '__name__': '__main__',
        '__doc__': None,
        '__package__': None,
        '__loader__': <_frozen_importlib_external.SourceFileL
oader object at 0x7fda798e3bb0>,
        '__spec__': None,
        '__annotations__': {},
        '__builtins__': <module 'builtins' (built-in)>,
        '__file__': 'main.py',
        '__cached__': None,
        'pi': 3.141592653589793,
        'tau': 6.283185307179586
}
```

One thing to notice is that we no longer have this `math` module object, and nothing is assigned to the name `math` in the global scope. This is why we're no longer able to access things like `fsum` using something like `math.fsum` .

What we do have are two new names, `pi` and `tau` , and these names are assigned long float values. This is what we're referring to when we use the names `pi` and `tau` . They're just normal variables that Python went ahead and defined for us when we imported the constants.

## Important

As I mentioned in this section, when we import a module, Python has to run the module. This makes a lot of sense, because some values or functions may rely on other values in the module. No matter what style of import we use, Python *always* has to run the module.

One thing that a lot of Python developers mistakenly assume is that using an import like this:

```
from math import pi, tau
```

Is somehow more efficient, because we're only grabbing a couple of values, and Python therefore didn't have to run the module. This isn't the case, so please don't be tempted to use this specific import syntax because you want to make the module load more efficient.

There is, however, one important way that the syntax above can help with efficiency, and that's when referencing the values. If you're using a name many, many times in a performance sensitive part of your application, referring to `pi` directly will be marginally faster than `math.pi` . This is very rarely something you're going to have to worry about though, and the difference in speed is very minor. You need millions of references for there to be any noticeable difference.

## Aliased imports

Sometimes module names are fairly long, or we just have to refer to them many times, and including the module name whenever we call a function defined in that module can become a little cumbersome. For this reason, we can use an alias for a module when we import it. This allows us to refer to the module using this alias, rather than the module's name.

For example, let's say we're doing some data analysis work and we've decided to use the `numpy` module.

It's common convention when importing numpy to import it using the name `np` , which we can accomplish using the `as` syntax.

```
import numpy as np
```

What we've done here is set an alias for the `numpy` module, which just means we've asked Python to let us refer to it by some other name. In this case, we've asked to refer to it using `np` .

Let's print what ends up in the global namespace in this case:

```
import numpy as np

print(globals())
```

If you're running this code on [repl.it](https://repl.it), you're going to get something like the output below. If you're running this in a local Python installation, the code may not work, because `numpy` is a third party module, and has to be installed.

```
{
        '__name__': '__main__',
        '__doc__': None,
        '__package__': None,
        '__loader__': <_frozen_importlib_external.SourceFileL
oader object at 0x7fa4ee853bb0>,
        '__spec__': None,
        '__annotations__': {},
        '__builtins__': <module 'builtins' (built-in)>,
        '__file__': 'main.py',
        '__cached__': None,
        'np': <module 'numpy' from '/home/runner/.local/shar
e/virtualenvs/python3/lib/python3.8/site-packages/numpy/__ini
t__.py'>
}
```

As we can see, we once again get a `module` object, and the module's name is `'numpy'`, but Python has assigned this module to the name `np`.

This means we can access things in the `numpy` module using `np.`, such as the array type. We'd do this by typing `np.array`.

## Importing using `*`

In addition to the import syntax we've looked at so far, we can also write something like this:

```
from math import *
```

This syntax essentially means, "Give me everything in the `math` module". We don't quite get *everything*, as we're still not going to get access to module contents that require a specific import, like when importing `ttk` from `tkinter`.

I want to make it very clear up front that this is not a style of import you should be using in your code 99% of the time. The reason for this is that all of the names defined in the module we're importing are going to get added to the global namespace.

If you want to see this for yourself, run the following code.

```python
from math import *

print(globals())
```

This is a big problem, because it can very easily lead to name conflicts. For example, we might accidentally use a name of something defined in a module we've imported, and then we've overwritten the value we got from the module. If different modules also use the same names for values, we're also going to run into issues where one is overwriting the other.

For example, both `math` and `numpy` define a value called `pi` , but only one value can be assigned to the name `pi` in our global namespace.

If it's a bad idea to use this style of import, then why does it exist? It's mostly there to facilitate quick testing. If we're not working on a serious application, using the `*` import can save us a lot of typing where we don't need to worry about best practices.

Just don't use it in any serious applications you're writing unless you have a genuine use case and you know what you're doing.

## Exercises

1) Import the `fractions` module and create a `Fraction` from the float `2.25` . You can find information on how to create fractions in the documentation.

2) Import only the `fsum` function from the `math` module and use it to find the sum of the following series of floats:

```
numbers = [1.43, 1.1, 5.32, 87.032, 0.2, 23.4]
```

3) Import the `random` module using an alias, and find a random number between `1` and `100` using the `randint` function. You can find documentation for this function here.

4) Use the `randint` function from the exercise above to create a new version of the guessing game we made in day 8. This time the program should generate a random number, and you should tell the user whether their guess was too high, or too low, until they get the right number.

You can find our solutions to the exercises here.

## Additional Resources

If you want to learn more about GUI development, we have a dedicated course on Udemy, as well as a number of posts on our blog.

We don't have anything on `numpy`, but if you want to learn more about this module, there's a good quickstart tutorial page in the `numpy` documentation.

If you're interested in digging into the problems with floating point numbers, and a solution to those problems, you can check out our post comparing floats to `decimal` objects.