Chapter 9 - Importing

Python comes with lots of pre-made code baked in. These pieces of code are known as modules and packages. A module is a single importable Python file whereas a package is made up of two or more modules. A package can be imported the same way a module is. Whenever you save a Python script of your own, you have created a module. It may not be a very useful module, but that's what it is. In this chapter, we will learn how to import modules using several different methods. Let's get started!

import this

Python provides the **import** keyword for importing modules. Let's give it a try:

```
import this
```

If you run this code in your interpreter, you should see something like the following as your output:

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

You have found an "Easter egg" in Python known as the "Zen of Python". It's actually a sort of an unofficial best practices for Python. The **this** module doesn't actually do anything, but it provided a fun little way to show how to import something. Let's actually import something we can use, like the **math** module:

```
>>> import math
>>> math.sqrt(4)
2.0
```

Here we imported the **math** module and then we did something kind of new. We called one of its functions, **sqrt** (i.e. square root). To call a method of an imported module, we have to use the following syntax: **module_name.method_name(argument)**. In this example, we found the square root of 4. The **math** module has many other functions that we can use, such as **cos** (cosine), **factorial**, **log** (logarithm), etc. You can call these functions in much the same way you did **sqrt**. The only thing you'll need to check is if they accept more arguments or not. Now let's look at another way to import.

Using from to import

Some people don't like having to preface everything they type with the module name. Python has a solution for that! You can actually import just the functions you want from a module. Let's pretend that we want to just import the **sqrt** function:

```
>>> from math import sqrt
>>> sqrt(16)
4.0
```

This works pretty much exactly how it is read: **from the math module, import the sqrt function**. Let me explain it another way. We use Python's **from** keyword to import the **sqrt** function **from** the **math** module. You can also use this method to import multiple functions from the math function:

```
>>> from math import pi, sqrt
```

In this example, we import both **pi** and **sqrt**. If you tried to access **pi** you may have noticed that it's actually a value and not a function that you can call. It just returns the value of pi. When you do an import, you may end up importing a value, a function or even another module! There's one more way to import stuff that we need to cover. Let's find out how to import everything!

Importing Everything!

Python provides a way to import **all** the functions and values from a module as well. This is actually a **bad** idea as it can contaminate your **namespace**. A namespace is where all your variables live during the life of the program. So let's say you have your own variable named **sqrt**, like this:

```
>>> from math import sqrt
>>> sqrt = 5
```

Now you have just changed the **sqrt** function into a variable that holds the value of 5. This is known as **shadowing**. This becomes especially tricky when you import everything from a module. Let's take a look:

```
>>> from math import *
>>> sqrt = 5
>>> sqrt(16)
Traceback (most recent call last):
  File "<string>", line 1, in <fragment>
TypeError: 'int' object is not callable
```

To import everything, instead of specifying a list of items, we just use the "*" wildcard which means we want to import everything. If we don't know what's in the **math** module, we won't realize that we've just clobbered one of the functions we imported. When we try to call the **sqrt** function after reassigning it to an integer, we find out that it no longer works.

Thus it is recommended that in most cases, you should import items from modules using one of the previous methods mentioned in this chapter. There are a few exceptions to this rule. Some modules are made to be imported using the "*" method. One prominent example is Tkinter, a toolkit included with Python that allows you to create desktop user interfaces. The reason that it is supposedly okay to import Tkinter in this way is that the modules are named so that it is unlikely you would reuse one yourself.

Wrapping Up

Now you know all about Python imports. There are dozens of modules included with Python that you can use to give extra functionality to your programs. You can use the builtin modules to query your OS, get information from the Windows Registry, set up logging utilities, parse XML, and much, much more. We will be covering a few of these modules in Part II of this book.

In the next chapter, we will be looking at building our own functions. I think you'll find this next topic to be very helpful.