Photo by Émile Perron on Unsplash

What happens behind the scenes when we import a module in Python

Importing modules in Python behind the scenes!!

Rachit Tayal Follow
Oct 12, 2019 · 5 min read ★

Using an *import* statement in our Python file is quite regular. Even for experienced Pythonistas, imports can be confusing, since there is no single method for guaranteeing that the imports will always work.

The aim of this article is to understand the internals of how the import statement works so as to develop better understanding and to resolve common importing problems.

Before jumping right in, we need to have some understanding of Python modules to know how import works.

Python Modules: Overview

A Python module is a file that has a `.py` extension. Python modules are pretty straightforward to build. All we need to do is create a file that contains legitimate Python code and give the file a name with `.py` extension. That's it!!

Modules are constructs in Python that promote code modularization. Making use of modules allows us to make our programs more robust and powerful as we're leveraging existing code.

For example, suppose we create a module called `mod.py` containing the following code:

```
name = "Sarah"
age = 26


def greet():
    print("Welcome {}!!".format(name))

class xyz:
    pass
```

The objects in the module `mod.py` can be accessed by importing the module as follows:

```
>>> import mod
>>> print(mod.name)
Sarah
>>> mod.age
26
>>> mod.greet()
'Welcome Sarah!!'
>>> obj = mod.xyz()
>>> obj
<mod.xyz object at 0x10e2173d0>
```

How import works

What happens when Python executes the statement:

```
import mod
```

When the interpreter executes the above `import` statement, it searches for `mod.py` in a list of directories gathered from the following sources:

- The directory from where the input script was run or the current directory if the interpreter is being run interactively.
- The list of directories contained in the `PYTHONPATH` environment variable, if it is set. (The format for `PYTHONPATH` is OS-dependent but should mimic the `PATH` environment variable.)
- An installation-dependent list of directories configured at the time Python is installed.

The resulting search path is accessible in the Python variable `sys.path`, which is obtained from a module named `sys`:

```
>>> import sys
>>> sys.path
['', '/home/roark/personal', '/home/roark/workdir',
'/usr/local/Python/3.7/lib',
'/usr/local/python3.7/site-packages',
'/usr/local/Python/3.7/python37.zip',
'/usr/local/Python/3.7/lib/python3.7']
```

Thus, to ensure our module is found, we need to do one of the following:

- Put `mod.py` in the directory where the input script is located or the current working directory, if interactive
- Modify the `PYTHONPATH` environment variable to contain the directory where `mod.py` is located before starting the interpreter (or put `mod.py` in one of the directories already contained in the `PYTHONPATH` variable)

There is actually one additional option: we can put the module file in any directory of your choice and then modify `sys.path` at run-time so that it contains that directory. For example, in this case, we could put `mod.py` in directory `/home/sarah/` and then issue the following statements:

```
>>> sys.path.append(r'/home/sarah/')
>>> sys.path
['', '/home/roark/personal', '/home/roark/workdir',
'/home/sarah',
'/usr/local/Python/3.7/lib',
'/usr/local/python3.7/site-packages',
'/usr/local/Python/3.7/python37.zip',
'/usr/local/Python/3.7/lib/python3.7']
>>> import mod
```

Once module has been imported, we can determine the location where it was found with the module's `__file__` attribute:

```
>>> import mod
>>> mod.__file__
'/home/sarah/mod.py'

>>> import datetime
>>> datetime.__file__
'/usr/local/Python/3.7/lib/datetime.py'
```

The directory portion of `__file__` should be one of the directories in `sys.path`.

The import statement

Module contents are made available to the caller with the `import` statement. The `import` statement takes many different forms, shown below.

1. import <module_name>

The simplest form is the one, we have already seen above.

`import <module_name>`

Note that it does not make the module contents *directly* accessible to the caller. Each module has its own **private symbol table**, which serves as the global symbol table for all objects defined *in the module*. Thus, a module creates a separate **namespace**, as already noted.

From the caller, objects in the module are only accessible when prefixed with `<module_name>` via **dot notation** as illustrated below.

After the following `import` statement, `mod` is placed into the local symbol table. Thus, `mod` has meaning in the caller's local context:

```
>>> import mod
mod
```

```
<module 'mod' from '/Users/z003fxh/personal/mod.py'>
```

The statement `import <module_name>` only places `<module_name>` in the caller's symbol table. The *objects* that are defined in the module *remain in the module's private symbol table*.

Therefore, `age`, `name`, `greet` and `xyz` remain in the module's private symbol table and are not meaningful in the caller's context. To be accessed in the caller's context, names of objects defined in the module must be prefixed by `mod`:

```
>>> greet()
NameError: name greet is not defined
>>> mod.greet()
'Welcome Sarah!!'
>>> mode.age
26
```

2. from <module_name> import <name(s)>

An alternate form of the `import` statement allows individual objects from the module to be imported *directly into the caller's symbol table*.

`from <module_name> import <name(s)>`

Using this syntax, `<name(s)>` can be referenced in the caller's environment without the `<module_name>` prefix:

```
>>> from mod import name, greet
>>> name
'Sarah'
>>> greet()
'Welcome Sarah!!'

>>> from mod import xyz
>>> obj = xyz()
>>> obj
<mod.xyz object at 0x102d46b90>
```

Because this form of `import` places the object names directly into the caller's symbol table, any objects that already exist with the same name will be **overwritten**:

```
>>> name = 'Roark'
>>> age = 25


>>> name, age
('Roark', 25)


>>> from mod import name, age
>>> name, age
('Sarah', 26)
```

3. from <module_name> import <name> as <alt_name>

It is also possible to `import` individual objects but enter them into the local symbol table with alternate names.

`from <module_name> import <name> as <alt_name>`

This makes it possible to place names directly into the local symbol table but avoid conflicts with previously existing names:

```
>>> name = 'Roark'
>>> age = 25

>>> from mod import name as k_name, age as k_age
>>> name
'Roark'
>>> k_name
'Sarah'
>>> age
25
>>> k_age
26
```

4. import <module_name> as <alt_name>

We can also import an entire module under an alternate name:

`import <module_name> as <alt_name>`

```
>>> import mod as my_module
>>> my_module.age
26
>>> my_module.greet()
'Welcome Sarah!!'
```

Lastly, a `try` statement with an `except ImportError` clause can be used to guard against unsuccessful `import` attempts:

```
>>> try:
...     # Non-existent module
...     import def
... except ImportError:
...     print('Module not found')
...

Module not found
```

Conclusions:

In this article, we have covered as to:

- How we can create a Python module
- Paths where the Python interpreter searches for a module
- How to obtain access to the objects defined in a module with the `import` statement

This will hopefully allow us to better understand and gain access to the functionality available in the many third-party and built-in modules available in Python. For deep divers, checkout out the official Python documentation for import statement and modules.

Python Python3 Programming

558 2

WRITTEN BY

Rachit Tayal

Follow

Sports Enthusiast | Senior Deep Learning Engineer. Python Blogger @ medium. Background in Machine Learning & Python. Linux and Vim Fan

Python Features

Follow

It covers important Python aspects with code snippets

More From Medium

November 30: Bug fixes and improvements

Brita Ulf in Streak

Good food, sunshine, and robots

Andrew Millar

Software Developer: Best Advice

Jigar Patel

How should CSS being setup (Part 2)

42KM in A Tiny Vault

Decoding The Performance NFRs

PritamR

How Cost Management Can Tank Your Cloud Migration

Scottie Bryan in HashmapInc

In-Depth Comparison of Time-Tracking Software as a Service Products Based on Market Segmentation

Samuel Harker

Just simple weather station ...

CHATCHAWAN KARAWAN