Python import, sys.path, and PYTHONPATH

Table of Contents

Introduction

The import statement is usually the first thing you see at the top of any Python file. We use it all the time, yet it is still a bit mysterious to many people. This tutorial will walk through how import works and how to view and modify the directories used for importing.

Modules versus packages

First, let's clarify the difference between modules and packages. They are very closely related, and often confused. They both serve the same purpose which is to organize code, but they each provide slightly different ways of doing that.

- A *module* is a single .py file with Python code.
- A *package* is a directory that can contains multiple Python modules.

A module can be thought of as a self-contained package, and a package is like a module that is separated out across multiple files. It really depends on how you want to organize your code and how large your project is. I always start with a module and turn it in to a package if needed later.

How import works

The import keyword in Python is used to load other Python source code files in to the current interpreter session. This is how you re-use code and share it among multiple files or different projects.

There are a few different ways to use import. For example, if we wanted to use the function join() that lives in the path module of the os package. Its full name would be os.path.join(). We have a few ways of importing and using the function. Read more about the os package at https://docs.python.org/3/library/os.path.html.

IMPORT VERSUS FROM

There are a few different ways you can import a package or a module. You can directly call import or use from x import y format. The from keyword tells Python what package or module to look in for the name specified with import. Here are a few example that all accomplish the same end goal.

Different ways to import and execute os.path.join():

```
import os
os.path.join()
```

```
from os import path
path.join()
```

```
from os import *
path.join()
```

```
from os.path import join
join()
```

As you can see, you can import the whole package, a specific module within a package, a specific function from within a module. The * wildcard means load all modules and functions. I do not recommend using the wildcard because it is too ambiguous. It is better to explicitly list each import so you can identify where it came from. A good IDE like PyCharm will help you manage these easily.

When you call import in the Python interpreter searches through a set of directories for the name provided. The list of directories that it searches is stored in sys.path and can be modified during run-time. To modify the paths before starting Python, you can modify the PYTHONPATH environment variable. Both sys.path and PYTHONPATH are covered more below.

IMPORT BY STRING

If you want to import a module programmatically, you can use importlib.import_module(). This function is useful if you are creating a plugin system where modules need to be loaded at run-time based on string names.

```
from importlib import import_module

# String should match the same format you would normally use to import
my_module = import_module("my_package.my_module")

# Then you can use it as if you did `import my_package.my_module`
my_module.my_function()
```

This method is not commonly used, and is only useful in special circumstances. For example, if you are building a plugin system where you want to load every file in a directory as a module based on the filepath string.

How __init__ and __main__ work

Names that start and end with double underscores, often called 'dunders', are special names in Python. Two of them are special names related to modules and packages: __init__ and __main__. Depending on whether you are organizing your code as a package or a module, they are treated slightly differently.

We will look at the difference between a module and a package in a moment, but the main idea is this:

- When you import a package it runs the __init__.py file inside the package directory.
- When you execute a package (e.g. python -m my_package) it executes the __main__.py file.
- When you import a module it runs the entire file from top to bottom.
- When you execute a module ir runs the entire file from top-to-bottom and sets the __name__ variable to the string "__main__".

IN A PACKAGE

In a Python package (a directory), you can have a module named __init__.py and another named __main__.py.

Here is an example structure of a package:

```
# Directory structure of my_package/
my_package/
my_package/__init__.py
my_package/__main__.py
my_package/some_other_module.py
```

If a package is invoked directly (e.g. python -m my_package), the __main__.py module is executed. The __init__.py file is executed when a package is imported (e.g. import my_package).

IN A MODULE

In the previous section, we saw how a package can have separate files for __init__.py and __main__.py. In a module (a single .py file) the equivalent of __init__ and __main__ are contained in the single file. The entire itself essentially becomes both the __init__.py and the __main__.py.

When a module is imported, it runs the whole file, loading any functions defined.

When a module is invoked directly, for example, python my_module.py or python -m my_module, then it does the same thing as importing it, but also sets the __name__ variable to the string "__main__".

You can take advantage of this and execute a section of code only if the module is invoked directly, and not when it is imported. To do this, you need to explicitly check

the __name__ variable, and see if it equals __main__. If it is set to the string __main__, then you know the module was invoked directly, and not simply imported.

Take this example. Create a file named my_module.py with the following contents:

```
# my_module.py
print('This will run when the file is imported.')

def my_function():
    print('Executing function. This will only run when the function is called.')

if __name__ == '__main__':
    print('This will get executed only if')
    print('the module is invoked directly.')
    print('It will not run when this module is imported')
    my_function()
```

Try out a few different things to understand how it works:

- Run the file directly with Python: python my_module.py
- Invoke the module with -m flag: python -m my_module
- Import the module from another Python file: python -c "import my_module"
- Import and call the function defined: python -c "import my_module; my_module.my_function()"

Manage import paths

SYS.PATH

When you start a Python interpreter, one of the things it creates automatically is a list that contains all of directories it will use to search for modules when importing. This list is available in a variable named sys.path. Here is an example of printing out sys.path. Note that the empty '' entry means the current directory.

```
>>> import sys
>>> sys.path
['',
'C:\\opt\\Python36\\python36.zip',
'C:\\opt\\Python36\\DLLs',
'C:\\opt\\Python36\\lib',
'C:\\opt\\Python36',
'C:\\Users\\NanoDano\\AppData\\Roaming\\Python\\Python36\\site-packages',
'C:\\opt\\Python36\\lib\\site-packages',
'C:\\opt\\Python36\\lib\\site-packages\\win32',
'C:\\opt\\Python36\\lib\\site-packages\\win32\\lib',
'C:\\opt\\Python36\\lib\\site-packages\\Pythonwin']
```

You are allowed to modify sys.path during run-time. Just be sure to modify it before you call import. It will search the directories in order stopping at the first place it finds the specified modules.

PYTHONPATH

PYTHONPATH is related to sys.path very closely. PYTHONPATH is an environment variable that you set before running the Python interpreter. PYTHONPATH, if it exists, should contain directories that should be searched for modules when using import. If PYTHONPATH is set, Python will include the directories in sys.path for searching. Use a semicolon to separate multiple directories.

Here is an example of setting the environment variable in Windows and listing the paths in Python:

```
set PYTHONPATH=C:\pypath1\;C:\pypath2\
python -c "import sys; print(sys.path)"
# Example output
['',    'C:\\pypath1',    'C:\\pypath2',    'C:\\opt\\Python36\\python36.zip',
'C:\\opt\\Python36\\DLLs',    'C:\\opt\\Python36\\lib',    'C:\\opt\\Python36',
'C:\\Users\\NanoDano\\AppData\\Roaming\\Python\\Python36\\site-packages',
'C:\\opt\\Python36\\lib\\site-packages',        'C:\\opt\\Python36\\lib\\site-
packages\\win32',        'C:\\opt\\Python36\\lib\\site-packages\\win32\\lib',
'C:\\opt\\Python36\\lib\\site-packages\\Pythonwin']
```

And in Linux and Mac you can do the equivalent like this:

```
export PYTHONPATH='/some/extra/path'
python -c "import sys; print(sys.path)"
# Example output
['',   '/some/extra/path',   '/usr/lib/python2.7',   '/usr/lib/python2.7/plat-x86_64-linux-gnu',  '/usr/lib/python2.7/lib-tk',  '/usr/lib/python2.7/lib-old',  '/usr/lib/python2.7/lib-dynload',    '/usr/local/lib/python2.7/dist-packages',  '/usr/lib/python2.7/dist-packages']
```

So, in order to import modules or packages, they need to reside in one of the paths listed in sys.path. You can modify the sys.path list manually if needed from within Python. It is just a regular list so it can be modified in all the normal ways. For example, you can append to the end of the list using sys.path.append() or to insert in an arbitrary position using sys.path.insert(). For more help, refer to https://docs.python.org/3/tutorial/datastructures.html

THE SITE MODULE

You can also use the site module to modify sys.path. See more at https://docs.python.org/3/library/site.html.

```
import site
import sys

site.addsitedir('/the/path')  # Always appends to end
print(sys.path)
```

Example output:

```
['', '/usr/lib/python2.7', '/usr/lib/python2.7/plat-x86_64-linux-gnu',
'/usr/lib/python2.7/lib-tk', '/usr/lib/python2.7/lib-old',
'/usr/lib/python2.7/lib-dynload', '/usr/local/lib/python2.7/dist-packages',
'/usr/lib/python2.7/dist-packages', '/the/path']
```

You can also direclty invoke the site module to get a list of default paths:

```
python3 -m site
```

Example run:

```
nanodano@localhost:~$ python3 -m site
sys.path = [
    '/home/nanodano',
    '/usr/lib/python37.zip',
    '/usr/lib/python3.7',
    '/usr/lib/python3.7/lib-dynload',
    '/usr/local/lib/python3.7/dist-packages',
    '/usr/lib/python3/dist-packages',
]
```

PYTHONHOME

The PYTHONHOME environment variable is similar to PYTHONPATH except it should define where the standard libraries are. If PYTHONHOME is set, it will assume some default paths relative to the home, which can be supplemented with PYTHONPATH.

This is particularly relevant if you embedded Python in to a C application and it is trying to determine the path of Python using the PYTHONHOME environment variable.

Just for reference, here is a quick example of how you would build a C application with Python embedded in it.

```
// pytest.c

// Assuming you built Python from source and installed in the `~/py` directory.
// Reference: https://www.devdungeon.com/content/how-build-python-source

// Compile with:
// gcc pytest.c -I ~/py/include/python3.9 -L ~/py/lib -lpython3.9 -lpthread -
ldl -lm -lutil

// Run with:
// PYTHONHOME=~/py ./a.out

// C code adapted from https://docs.python.org/3.9/extending/embedding.html
#define PY_SSIZE_T_CLEAN
#include <Python.h>

int main(int argc, char *argv[]){
    wchar_t *program = Py_DecodeLocale(argv[0], NULL);
    if (program == NULL) {
        fprintf(stderr, "Fatal error: cannot decode argv[0]\n");
        exit(1);
    }
    Py_SetProgramName(program);  /* optional but recommended */
    Py_Initialize();
    PyRun_SimpleString("import sys\n"
                       "print(sys.path)\n");
    if (Py_FinalizeEx() < 0) {
        exit(120);
    }
    PyMem_RawFree(program);
    return 0;
}
```

Conclusion

After reading this, you should have a better understanding of how Python's import statement works and how the PYTHONPATH environment variable and sys.path affect import. You should also understand the differences and similarities between modules and packages, the dunders __init__ and __main__ and how to use them effectively.