Chris Yeh

The Definitive Guide to Python import Statements

Posted: Aug 8, 2017. Last updated: Sep 30, 2020.
Tags: python

I've almost never been able to write correct Python `import` statements on the first go. Behavior is inconsistent between Python 2.7 and Python 3.6 (the two versions that I test here), and there is no single method for guaranteeing that imports will always work. This post is my dive into how to resolve common importing problems. Unless otherwise stated, all examples here work with both Python 2.7 and 3.6.

Contents

- Summary / Key Points
- Basic Definitions
- Example Directory Structure
- What is an `import`?
- Basics of the Python `import` and `sys.path`
  - More on `sys.path`
- All about `__init__.py`
  - Converting a folder of scripts into an importable package of modules
  - Running package initialization code
- Using Objects from the Imported Module or Package
  - Use `dir()` to examine the contents of an imported module
  - Importing Packages
- Absolute vs. Relative Import
- Case Examples
  - Case 1: `sys.path` is known ahead of time
  - Case 2: `sys.path` could change
  - Case 3: Importing from Parent Directory
- Python 2 vs. Python 3
- Appendix: List of Always Built-in Modules
- Miscellaneous topics and readings not covered here, but worth exploring

Summary / Key Points

- `import` statements search through the list of paths in `sys.path`
- `sys.path` always includes the path of the script invoked on the command line and is agnostic to the working directory on the command line.

- importing a package is conceptually the same as importing that package's `__init__.py` file

Basic Definitions

- **module**: any `*.py` file. Its name is the file name.
- **built-in module**: a "module" (written in C) that is compiled into the Python interpreter, and therefore does not have a `*.py` file.
- **package**: any folder containing a file named `__init__.py` in it. Its name is the name of the folder.
  - in Python 3.3 and above, any folder (even without a `__init__.py` file) is considered a package
- **object**: in Python, almost everything is an object - functions, classes, variables, etc.

Example Directory Structure

```
test/                          # root folder
    packA/                     # package packA
        subA/                  # subpackage subA
            __init__.py
            sa1.py
            sa2.py
        __init__.py
        a1.py
        a2.py
    packB/                     # package packB (implicit namespace package)
        b1.py
        b2.py
    math.py
    random.py
    other.py
    start.py
```

Note that we do not place a `__init__.py` file in our root `test/` folder.

What is an `import`?

When a module is imported, Python runs all of the code in the module file. When a package is imported, Python runs all of the code in the package's `__init__.py` file, if such a file exists. All of the objects defined in the module or the package's `__init__.py` file are made available to the importer.

Basics of the Python `import` and `sys.path`

According to Python documentation, here is how an `import` statement searches for the correct module or package to import:

> When a module named `spam` is imported, the interpreter first searches for a built-in module with that name. If not found, it then searches for a file named `spam.py` in a list of directories

> given by the variable `sys.path`. `sys.path` is initialized from these locations:
>
> - The directory containing the input script (or the current directory when no file is specified).
> - `PYTHONPATH` (a list of directory names, with the same syntax as the shell variable PATH).
> - The installation-dependent default.
>
> After initialization, Python programs can modify `sys.path`. The directory containing the script being run is placed at the beginning of the search path, ahead of the standard library path. This means that scripts in that directory will be loaded instead of modules of the same name in the library directory. *Source: Python 2 and 3*

Technically, Python's documentation is incomplete. The interpreter will not only look for a *file* (i.e., module) named `spam.py`, it will also look for a *folder* (i.e., package) named `spam`.

Note that the Python interpreter first searches through the list of *built-in modules*, modules that are compiled directly into the Python interpreter. This list of built-in modules is installation-dependent and can be found in `sys.builtin_module_names` (Python 2 and 3). Some modules that are commonly built-in include `sys`, `math`, `itertools`, and `time`, among others. See the appendix below for a list of always-included built-in modules.

Unlike built-in modules which are first in the search path, the rest of the modules in Python's standard library (not built-ins) come after the directory of the current script. This leads to confusing behavior: it is possible to "replace" some but not all modules in Python's standard library. For example, on my computer (Windows 10, Python 3.6), the `math` module is a built-in module, whereas the `random` module is not. Thus, `import math` in `start.py` will import the `math` module from the standard library, NOT my own `math.py` file in the same directory. However, `import random` in `start.py` will import my `random.py` file, NOT the `random` module from the standard library.

Also, **Python imports are case-sensitive.** `import Spam` is not the same as `import spam`.

The function `pkgutil.iter_modules` (Python 2 and 3) can be used to get a list of all importable modules from a given path:

```python
import pkgutil
search_path = ['.'] # set to None to see all modules importable from sys.path
all_modules = [x[1] for x in pkgutil.iter_modules(path=search_path)]
print(all_modules)
```

Sources

- How to get a list of built-in modules in python?
- Thank you etene for pointing out the difference between built-in modules and other modules in Python's standard library (Issue 2)

More on `sys.path`

To see what is in `sys.path`, run the following in the interpreter or as a script:

```
import sys
print(sys.path)
```

Python's documentation for `sys.path` describes it as…

> A list of strings that specifies the search path for modules. Initialized from the environment variable `PYTHONPATH`, plus an installation-dependent default.
>
> As initialized upon program startup, the first item of this list, `path[0]`, is the directory containing the script that was used to invoke the Python interpreter. If the script directory is not available (e.g. if the interpreter is invoked interactively or if the script is read from standard input), `path[0]` is the empty string, which directs Python to search modules in the current directory first. Notice that the script directory is inserted before the entries inserted as a result of `PYTHONPATH`.
>
> *Source: Python 2 and 3*

The documentation for Python's command line interface adds the following about running scripts from the command line. Specifically, when running `python <script>.py`, then…

> If the script name refers directly to a Python file, the directory containing that file is added to the start of sys.path, and the file is executed as the **main** module.
>
> *Source: Python 2 and 3*

Let's recap the order in which Python searches for modules to import:

1. built-in modules from the Python Standard Library (e.g. `sys`, `math`)
2. modules or packages in a directory specified by `sys.path`:
    1. If the Python interpreter is run interactively, `sys.path[0]` is the empty string `''`. This tells Python to search the current working directory from which you launched the interpreter, i.e., the output of `pwd` on Unix systems.

        If we run a script with `python <script>.py`, `sys.path[0]` is the path to `<script>.py`.

    2. directories in the `PYTHONPATH` environment variable
    3. default `sys.path` locations, including remaining Python Standard Library modules which are not built-in

Note that **when running a Python script, `sys.path` doesn't care what your current "working directory" is. It only cares about the path to the script**. For example, if my shell is currently at the `test/` folder and I run `python ./packA/subA/subA1.py`, then `sys.path` includes `test/packA/subA/` but NOT `test/`.

Additionally, `sys.path` is shared across all imported modules. For example, suppose we call `python start.py`. Let `start.py` import `packA.a1`, and let `a1.py` print out `sys.path`. Then `sys.path` will include `test/` (the path to `start.py`), but NOT `test/packA/` (the path to `a1.py`). What this means is that `a1.py` can call `import other` since `other.py` is a file in `test/`.

All about `__init__.py`

An `__init__.py` file has 2 functions.

1. convert a folder of scripts into an importable package of modules (before Python 3.3)
2. run package initialization code

Converting a folder of scripts into an importable package of modules

In order to import a module or package from a directory that is not in the same directory as the script we are writing (or the directory from which we run the Python interactive interpreter), that module needs to be in a package.

As defined above, any directory with a file named `__init__.py` is a Python package. This file can be empty. For example, when running Python 2.7, `start.py` can import the package `packA` but not `packB` because there is no `__init__.py` file in the `test/packB/` directory.

This does NOT apply to Python 3.3 and above, thanks to the adoption of implicit namespace packages. Basically, Python 3.3+ treats all folders as packages, so empty `__init__.py` files are no longer necessary and can be omitted.

For example, `packB` is a namespace package because it doesn't have a `__init__.py` file in the folder. If we start a Python 3.6 interactive interpreter in the `test/` directory, then we get the following output:

```
>>> import packB
>>> packB
<module 'packB' (namespace)>
```

Sources

1. What is **init**.py for?
2. PEP 420: Implicit Namespace Packages

Running package initialization code

The first time that a package or one of its modules is imported, Python will execute the `__init__.py` file in the root folder of the package if the file exists. All objects and functions defined in `__init__.py` are considered part of the package namespace.

Consider the following example.

`test/packA/a1.py`

```
def a1_func():
    print("running a1_func()")
```

`test/packA/__init__.py`

```
## this import makes a1_func directly accessible from packA.a1_func
from packA.a1 import a1_func

def packA_func():
    print("running packA_func()")
```

`test/start.py`

```
import packA  # "import packA.a1" will work just the same

packA.packA_func()
packA.a1_func()
packA.a1.a1_func()
```

output of running `python start.py`:

```
running packA_func()
running a1_func()
running a1_func()
```

Note: If `a1.py` calls `import a2` and we run `python a1.py`, then `test/packA/__init__.py` will NOT be called, even though it seems like `a2` is part of the `packA` package. This is because when Python runs a script (in this case `a1.py`), its containing folder is not considered a package.

Using Objects from the Imported Module or Package

There are 4 different syntaxes for writing import statements.

1. `import <package>`
2. `import <module>`
3. `from <package> import <module or subpackage or object>`
4. `from <module> import <object>`

Let `X` be whatever name comes after `import`.

- If `X` is the name of a module or package, then to use objects defined in `X`, you have to write `X.object`.
- If `X` is a variable name, then it can be used directly.

- If `X` is a function name, then it can be invoked with `X()`

Optionally, `as Y` can be added after any `import X` statement: `import X as Y`. This renames `X` to `Y` within the script. Note that the name `X` itself is no longer valid. A common example is `import numpy as np`.

The argument to the `import` function can be a single name, or a list of multiple names. Each of these names can be optionally renamed via `as`. For example, this would be a valid import statement in `start.py`: `import packA as pA, packA.a1, packA.subA.sa1 as sa1`

Example: `start.py` needs to import the `helloWorld()` function in `sa1.py`

- Solution 1: `from packA.subA.sa1 import helloWorld`
  - we can call the function directly by name: `x = helloWorld()`
- Solution 2: `from packA.subA import sa1` or equivalently `import packA.subA.sa1 as sa1`
  - we have to prefix the function name with the name of the module: `x = sa1.helloWorld()`
  - This is sometimes preferred over Solution 1 in order to make it explicit that we are calling the `helloWorld` function from the `sa1` module.
- Solution 3: `import packA.subA.sa1`.
  - we need to use the full path: `x = packA.subA.sa1.helloWorld()`

Note: The official syntax for import statements can be found in the documentation (Python 2 and 3). I find it difficult to parse the context-free grammar notation, and I have tried my best to summarize the key points here. However, this post does not cover details such as importing multiple identifiers from the same module.

Use `dir()` to examine the contents of an imported module

After importing a module, use the `dir()` function to get a list of accessible names from the module. For example, suppose I import `sa1`. If `sa1.py` defines a `helloWorld()` function, then `dir(sa1)` would include `helloWorld`.

```
>>> from packA.subA import sa1
>>> dir(sa1)
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__pac
```

Importing Packages

Importing a package is conceptually equivalent to importing the package's `__init__.py` file as a module. Indeed, this is what Python treats the package as:

```
>>> import packA
>>> packA
<module 'packA' from 'packA\__init__.py'>
```

Only objects declared in the imported package's `__init__.py` are accessible to the importer. For example, since `packB` lacks a `__init__.py` file, calling `import packB` (in Python 3.3+) has very little use because no objects in the `packB` package are made available. A subsequent call to `packB.b1` would fail because it has not been imported yet.

Absolute vs. Relative Import

An **absolute import** uses the full path (starting from the project's root folder) to the desired module to import.

A **relative import** uses the relative path (starting from the path of the current module) to the desired module to import. There are two types of relative imports:

- an *explicit* relative import follows the format `from .<module/package> import X`, where `<module/package>` is prefixed by dots `.` that indicate how many directories upwards to traverse. A single dot `.` corresponds to the current directory; two dots `..` indicate one folder up; etc.
- an *implicit* relative import is written as if the current directory is part of `sys.path`. **Implicit relative imports are only supported in Python 2. They are NOT SUPPORTED IN PYTHON 3.**

The Python documentation says the following about Python 3's handling of relative imports:

> The only acceptable syntax for relative imports is from .[module] import name. All import forms not starting with . are interpreted as absolute imports.
>
> *Source: What's New in Python 3.0*

For example, suppose we are running `start.py` which imports `a1` which in turn imports `other`, `a2`, and `sa1`. Then the import statements in `a1.py` would look as follows:

- absolute imports:

```
import other
import packA.a2
import packA.subA.sa1
```

- explicit relative imports:

```
import other
from . import a2
from .subA import sa1
```

- implicit relative imports (NOT SUPPORTED IN PYTHON 3):

```
import other
import a2
import subA.sa1
```

Note that for relative imports, the dots `.` can go up only up to (but not including) the directory containing the script run from the command line. Thus, `from .. import other` is invalid in `a1.py`. Doing so results in the error `ValueError: attempted relative import beyond top-level package`.

In general, absolute imports are preferred over relative imports. They avoid the confusion between explicit vs. implicit relative imports. In addition, any script that uses explicit relative imports cannot be run directly:

> Note that relative imports are based on the name of the current module. Since the name of the main module is always "**main**", modules intended for use as the main module of a Python application must always use absolute imports.
>
> *Source: Python 2 and 3*

Sources

- How to accomplish relative import in python
- Changes in import statement python3

Case Examples

Case 1: `sys.path` is known ahead of time

If you only ever call `python start.py` or `python other.py`, then it is very easy to set up the imports for all of the modules. In this case, `sys.path` will always include `test/` in its search path. Therefore, all of the import statements can be written relative to the `test/` folder.

Ex: a file in the `test` project needs to import the `helloWorld()` function in `sa1.py`

- Solution: `from packA.subA.sa1 import helloWorld` (or any of the other equivalent import syntaxes demonstrated above)

Case 2: `sys.path` could change

Often, we want to be flexible in how we use a Python script, whether run directly on the command line or imported as a module into another script. As shown below, this is where we run into problems, especially on Python 3.

Example: Suppose `start.py` needs to import `a2` which needs to import `sa2`. Assume that `start.py` is always run directly, never imported. We also want to be able to run `a2` on its own.

Seems easy enough, right? After all, we just need 2 import statements total: 1 in `start.py` and another in `a2.py`.

Problem: This is clearly a case where `sys.path` changes. When we run `start.py`, `sys.path` contains `test/`. When we run `a2.py`, `sys.path` contains `test/packA/`.

The import statement in `start.py` is easy. Since `start.py` it is always run directly and never imported, we know that `test/` will always be in `sys.path` when it is run. Then importing `a2` is simply `import packA.a2`.

The import statement in `a2.py` is trickier. When we run `start.py` directly, `sys.path` contains `test/`, so `a2.py` should call `from packA.subA import sa2`. However, if we instead run `a2.py` directly, then `sys.path` contains `test/packA/`. Now the import would fail because `packA` is not a folder inside `test/packA/`.

Instead, we could try `from subA import sa2`. This corrects the problem when we run `a2.py` directly. But now we have a problem when we run `start.py` directly. Under Python 3, this fails because `subA` is not in `sys.path`. (This is OK in Python 2, thanks to its support for implicit relative imports.)

Let's summarize our findings about the import statement in `a2.py`:

Run	**`from packA.subA import sa2`**	**`from subA import sa2`**
`start.py`	OK	Py2 OK, Py3 fail (`subA` not in `test/`)
`a2.py`	fail (`packA` not in `test/packA/`)	OK

For completeness sake, I also tried using relative imports: `from .subA import sa2`. This matches the result of `from packA.subA import sa2`.

Solutions (Workarounds): I am unaware of a clean solution to this problem. Here are some workarounds:

1. Use absolute imports rooted at the `test/` directory (i.e., middle column in the table above). This guarantees that running `start.py` directly will always work. In order to run `a2.py` directly, run it as an imported module instead of as a script:
   1. change directories to `test/` in the console
   2. `python -m packA.a2`
2. Use absolute imports rooted at the `test/` directory (i.e., middle column in the table above). This guarantees that running `start.py` directly will always work. In order to run `a2.py` directly, we can modify `sys.path` in `a2.py` to include `test/`, before `sa2` is imported.

```
import os, sys
sys.path.append(os.path.dirname(os.path.dirname(os.path.realpath(__file__))))

# now this works, even when a2.py is run directly
from packA.subA import sa2
```

   NOTE: This method usually works. However, under some Python installations, the `__file__` variable might not be correct. In this case, we would need to use the Python built-in `inspect` package. See this StackOverflow answer for instructions.

3. Only use Python 2, and use implicit relative imports (i.e., the right column in the table above).

4. Use absolute imports rooted at the `test/` directory, and add `test/` to the `PYTHONPATH` environment variable.
    - This solution is not portable, so I recommend against it.
    - instructions here: Permanently add a directory to PYTHONPATH

Case 3: Importing from Parent Directory

If we do not modify `PYTHONPATH` and avoid modifying `sys.path` programmatically, then the following is a major limitation of Python imports:

When running a script directly, it is impossible to import anything from its parent directory.

For example, if I were to run `python sa1.py`, then it is impossible for `sa1.py` to import anything from `a1.py` without resorting to a `PYTHONPATH` or `sys.path` workaround.

At first, it may seem that relative imports (e.g. `from .. import a1`) could work around this limitation. However, the script that is being run (in this case `sa1.py`) is considered the "top-level module." Attempting to import anything from a folder above this script results in this error: `ValueError: attempted relative import beyond top-level package`.

My approach is to avoid writing scripts that have to import from the parent directory. In cases where this must happen, the preferred workaround is to modify `sys.path`.

Python 2 vs. Python 3

The most important differences between how Python 2 and Python 3 treat `import` statements have been documented above. They are re-stated again here, along with some other less important differences.

1. Python 2 supports implicit relative imports. Python 3 does not.
2. Python 2 requires `__init__.py` files inside a folder in order for the folder to be considered a package and made importable. In Python 3.3 and above, thanks to its support of implicit namespace packages, all folders are packages regardless of the presence of a `__init__.py` file.
3. In Python 2, one could write `from <module> import *` within a function. In Python 3, the `from <module> import *` syntax is only allowed at the module level, no longer inside functions.

Sources

- Changes in import statement python3
- Python modules documentation for Python 2 and 3
- What's New in Python 3.0

Appendix: List of Always Built-in Modules

Back in Python 1.5, the documentation explicitly labeled each package as either “Built-in” or “Standard.” However, documentation for Python 2 and 3 are less explicit. Here, I aim to track modules that are always built-in (i.e., compiled) into the Python interpreter.

- `__builtin__` : Python 2 only
- `builtins` : Python 3 only
- `math` : Python 2 and 3
- `pwd` : only on Unix systems, Python 2 and 3
- `sys` : Python 2

Miscellaneous topics and readings not covered here, but worth exploring

- using `__all__` variable in `__init__.py` for specifying what gets imported by `from <module> import *`
  - documentation for Python 2 and 3
- using `if __name__ == '__main__'` to check if a script is imported or run directly
  - documentation for Python 2 and 3
- installing a project as a package (in developer mode) with `pip install -e <project>` to add the project root directory to `sys.path`
  - How to run tests without installing package?
- `from <module> import *` does not import names from `<module>` that begin with an underscore `_`
  - documentation for Python 2 and 3

This page was generated by GitHub Pages.