Material @ https://github.com/prodicus/talks

Demystifying how imports work in Python

Tasdik Rahman (@tasdikrahman)

Presented @ ChennaiPy, October'16 meetup

Requirements

- Python 3.4 or newer.
- Material @ https://github.com/prodicus/talks
- No extra 3rd party extensions
- Coming over for this meetup!

Modules

- Any `python` source file would be counted as a module.
- You import a module to execute and access its classes/definitions/variables.

```
>>> import os
>>> os.path.abspath('.')
'/home/tasdik/Dropbox/talks/chennaipy/october/samplecode'
>>>
```

- `posixpath` would be the module name where the method `abspath()` resides.

What happens when you import a module?

- It being a python script, the statements start getting executed from top to bottom of the source file.
- If there are any tasks in the statements (eg: a `print()` statement), then they get executed when the module is being imported.

```
# 'samplecode/basicpackage/'
>>> import basicpackage.bar
inside basicpackage/__init__.py
inside 'basicpackage/bar'
>>>
```

Packages

- Used to orgranize larger number of modules in a systematic manner.
- One way of accessing individual modules can be accessed using the `import foo.bar` import style.

```
# typical package structuring
$ tree basicpackage/
basicpackage
├── bar.py
├── foo.py
└── __init__.py
```

Why packages?

```
bumblebee
├── constants.py
├── core.py
├── exceptions.py
├── helpers
│   ├── __init__.py
│   └── vwo_helpers.py
├── __init__.py
└── vwo
    ├── __init__.py
    └── smart_code.py
```

Looks good?

Different styles for importing modules

from module import foo

- This essentially imports the module first then picks up specific parts from the module to be available locally.

```
>>> from basicpackage import foo
inside basicpackage/__init__.py
inside 'basicpackage/foo.py' with a variable in it
>>>
```

- allows using the parts of the module without giving the full prefix before it.

from module import *

- Brings out all the symbols from the module and makes them available in the namespace.

```
>>> from basicpackage_all import *
inside basicpackage_all/__init__.py
inside 'basicpackage_all/foo.py' with a variable in it
inside 'basicpackage_all/bar.py'
>>>
```

- You can use `__all__` inside your `__init__.py` module to import the modules which you need to import.
- **Generally not a good idea!**

Takeaways so far

- The way you import a module doesn't actually change the working of the module.
- Difference between `import foo.bar` and `from foo import bar` ?
  - the difference is subjective. Pick one style and be consistent with it.
  - doing a `from foo import bar` is more efficient.
    - `python` imports the whole file! period.

Module names

- naming modules follow the general variable naming convention.

```
# Bad choices
$ touch 2foo.py MyAwesomeFoo.py os.py

# Good choices
$ touch foo.py a_large_module_name.py
```

- Don't use Non-ASCII characters while doing so.
- Avoid creating module names which conflict with the standard library modules.

Module lookup

- If it's not in the python path, it just won't import.

```
>>> pprint(sys.path)
['',
 '/usr/lib/python35.zip',
 ...
 '/usr/lib/python3/dist-packages']
```

- Explicitly bring a module inside your path

```
>>> import sys
>>> sys.path.append('/absoule/path/to/module')
```

Modules get imported Only once!

Implicit Relative imports

```
$ rod/
    foo.py
    bar.py
    __init__.py
```

- So want to have some things from `foo.py` inside `bar.py` ? Nothing uncommon.

```
# python 2
# inside "bar.py"
import foo
```

- Don't do it! Works in `python2` but not in Doesn't work in `python3`

How do I fix it?

Absolute relative imports

- One way to fix it would be using the name of it's top level package name `rod`.

```
# relativeimports/foo.py
from relativeimports import bar
```

- This works, but is brittle!
- What if you wanted to change the name of the top level?
- Errors!!!!

Explicit relative imports

- A better way would be to

```
# explicitimports/bar.py
from . import foo
```

- Works even when you rename the root level package for whatever the reason be (eg: `explicitimportsv1`)

```
$ mv explicitimports/ newimports/
```

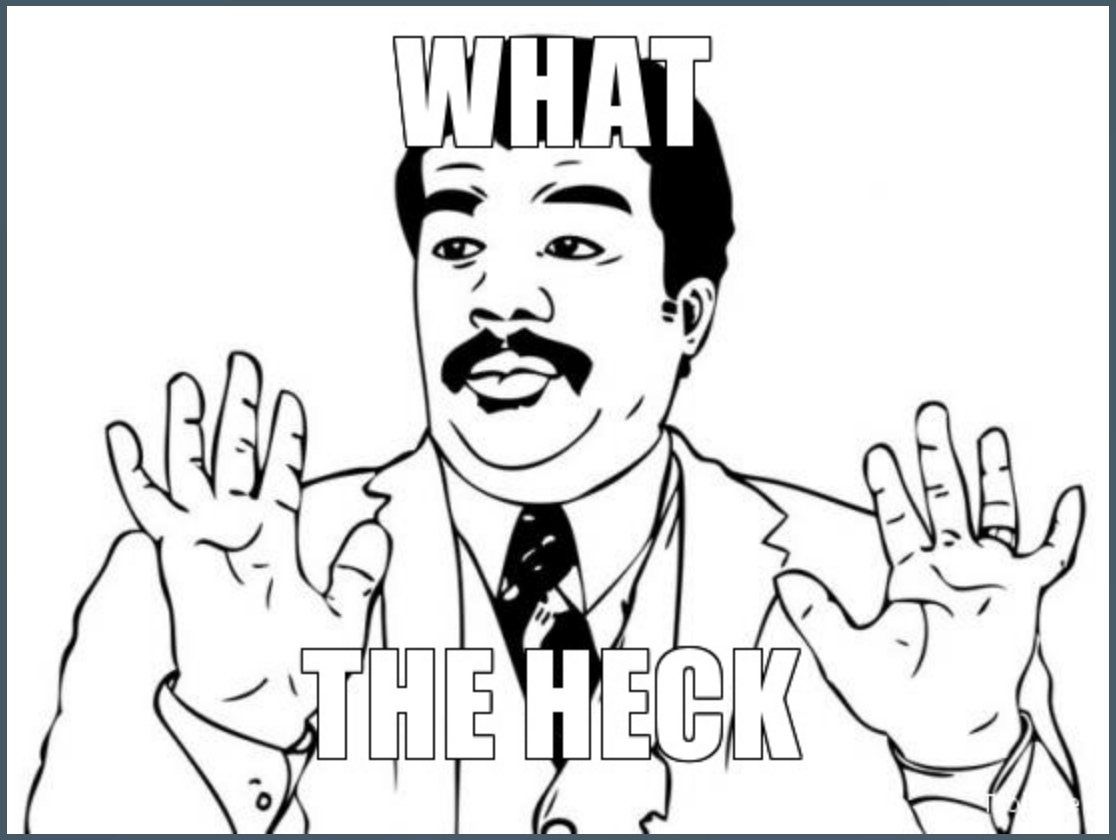
The leading **(.)** would be used to move up a directory.

```
# look for foo.py in the same level
from . import foo

# go a dir up and import foo.py
from .. import foo

# go a dir up and enter plino/ and look for bar.py
from ..plino import bar
```

`__init__.py`



What should you put into it?

- Most of the time, it's empty!
- Stiching together submodules:

```
# minions/foo.py
class Foo(object):
        pass

# minions/bar.py
class Bar(object):
        pass

# minions/__init__.py
from .foo import Foo
from .bar import Bar
```

Advantage?

- Headache free imports for small modules

```
>>> import minions
inside minions/__init__.py
inside 'minions/foo.py' with a variable in it
inside 'minions/bar.py'
>>> a = minions.Foo()
>>> b = minions.Bar()
```

- controlling import behaviour of `from foo import *` using the `__all__` variable inside `__init__.py`

References

- https://docs.python.org/3/tutorial/modules.html
- https://docs.python.org/3/reference/import.html
- https://docs.python.org/3/reference/executionmodel.html
- https://docs.python.org/3/library/distribution.html

Questions? Would be happy to answer

tasdikrahman.me

Twitter (tasdikrahman)

Github (@prodicus)