Modules and Packages: Live and Let Die!

David Beazley (@dabeaz)
http://www.dabeaz.com

Presented at PyCon'2015, Montreal

Requirements

- You need Python 3.4 or newer
- No third party extensions
- Code samples and notes

  http://www.dabeaz.com/modulepackage/

- Follow along if you dare!

Hieronymus Bosch
c. 1450 - 1516

Hieronymus Bosch
c. 1450 - 1516

"*The Garden of Earthly Delights*", *c. 1500*

Hieronymus Bosch
c. 1450 - 1516
(Dutch)

"*The Garden of Earthly Delights*", *c. 1500*

"The Garden of Pythonic Delights", c. 2015

"The Garden of Pythonic Delights", c. 2015

- The "creation"
- Python and its standard library
- "Batteries Included"
- import antigravity

Guido?

"The Garden of Pythonic Delights", c. 2015

- PyPI?

There are currently **57642** packages here.

- PyCON?

Everyone is naked, riding around on exotic animals, eating giant berries, etc.
???????????

"The Garden of Pythonic Delights", c. 2015

- Hell
- Package management?
- The future?
- A warning?

"The Garden of Pythonic Delights", c. 2015

Background: I've been using Python since version 1.3. Basic use of modules and packages is second nature. However, I also realize that I don't know that much about what's happening under the covers. I want to correct that.

"The Garden of Pythonic Delights", c. 2015

- This tutorial!
- A fresh take on modules
- Goal is to reintroduce the topic
- Avoid crazy hacks? (maybe)

"*The Garden of Pythonic Delights*", c. 2015

- Target Audience: Myself!
- Understanding import is useful
- Also: Book writing
- Will look at some low level details, but keep in the mind the goal is to gain a better idea of how everything works and holds together

"The Garden of Pythonic Delights", c. 2015

- Perspective: I'm looking at this topic from the point of view of an application developer and how I might use the knowledge to my advantage
- I am not a Python core developer
- Target audience is not core devs

"*The Garden of* Pythonic *Delights*", *c. 2015*

It's not "Modules and Packaging"

The tutorial is not about package managers (setuptools, pip, etc.)

... because "reasons"

Standard Disclaimers

- I learned a lot preparing
- Also fractured a rib while riding my bike on this frozen lake
- Behold the pain killers that proved to be helpful in finishing
- Er... let's start....

Part I

Basic Knowledge

Modules

- Any Python source file is a module

```
# spam.py
def grok(x):
    ...

def blah(x):
    ...
```

- You use import to execute and access it

```
import spam
a = spam.grok('hello')

from spam import grok
a = grok('hello')
```

Namespaces

- Each module is its own isolated world

```
# spam.py

x = 42

def blah():
    print(x)
```

```
# eggs.py

x = 37

def foo():
    print(x)
```

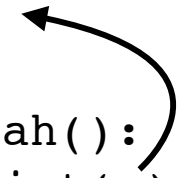
These definitions of x are different

- What happens in a module, stays in a module

Global Variables

- Global variables bind inside the same module

```
# spam.py

x = 42

def blah():
    print(x)
```



- Functions record their definition environment

```
>>> from spam import blah
>>> blah.__module__
'spam'
>>> blah.__globals__
{ 'x': 42, ... }
>>>
```

Module Execution

- When a module is imported, all of the statements in the module execute one after another until the end of the file is reached
- The contents of the module namespace are all of the global names that are still defined at the end of the execution process
- If there are scripting statements that carry out tasks in the global scope (printing, creating files, etc.), you will see them run on import

from module import

- Lifts selected symbols out of a module after importing it and makes them available locally

```
from math import sin, cos

def rectangular(r, theta):
    x = r * cos(theta)
    y = r * sin(theta)
    return x, y
```

- Allows parts of a module to be used without having to type the module prefix

from module import *

- Takes all symbols from a module and places them into local scope

```
from math import *

def rectangular(r, theta):
    x = r * cos(theta)
    y = r * sin(theta)
    return x, y
```

- Sometimes useful
- Usually considered bad style (try to avoid)

Commentary

- Variations on import do not change the way that modules work

```
import math as m
from math import cos, sin
from math import *
...
```

- import always executes the entire file
- Modules are still isolated environments
- These variations are just manipulating names

Module Names

- File names have to follow the rules

Yes

```
# good.py

...
```

No

```
# 2bad.py

...
```

- Comment: This mistake comes up a lot when teaching Python to newcomers
- Must be a valid identifier name
- Also: avoid non-ASCII characters

Naming Conventions

- It is standard practice for package and module names to be concise and lowercase

```
foo.py        not        MyFooModule.py
```

- Use a leading underscore for modules that are meant to be private or internal

```
_foo.py
```

- Don't use names that match common standard library modules (confusing)

```
projectname/
            math.py
```

Module Search Path

- If a file isn't on the path, it won't import

```
>>> import sys
>>> sys.path
['',
 '/usr/local/lib/python34.zip',
 '/usr/local/lib/python3.4',
 '/usr/local/lib/python3.4/plat-darwin',
 '/usr/local/lib/python3.4/lib-dynload',
 '/usr/local/lib/python3.4/site-packages']
```

- Sometimes you might hack it

```
import sys
sys.path.append("/project/foo/myfiles")
```

... although doing so feels "dirty"

Module Cache

- Modules only get loaded once
- There's a cache behind the scenes

```
>>> import spam
>>> import sys
>>> 'spam' in sys.modules
True
>>> sys.modules['spam']
<module 'spam' from 'spam.py'>
>>>
```

- Consequence: If you make a change to the source and repeat the import, nothing happens (often frustrating to newcomers)

Module Reloading

- You can force-reload a module, but you're never supposed to do it

```
>>> from importlib import reload
>>> reload(spam)
<module 'spam' from 'spam.py'>
>>>
```

- Apparently zombies are spawned if you do this
- No, seriously.
- Don't. Do. It.

__main__ check

- If a file might run as a main program, do this

```
# spam.py

...
if __name__ == '__main__':
    # Running as the main program
    ...
```

- Such code won't run on library import

```
import spam              # Main code doesn't execute

bash % python spam.py    # Main code executes
```

Packages

- For larger collections of code, it is usually desirable to organize modules into a hierarchy

```
spam/
    foo.py
    bar/
        grok.py
    ...
```

- To do it, you just add __init__.py files

```
spam/
    __init__.py
    foo.py
    bar/
        __init__.py
        grok.py
    ...
```

Using a Package

- Import works the same way, multiple levels

```
import spam.foo
from spam.bar import grok
```

- The __init__.py files import at each level
- Apparently you can do things in those files
- We'll get to that

Comments

- At a simple level, there's not much to 'import'
- ... except for everything else
- So let's continue

Part 2

Packages

Question

- Which is "better?"
  - One .py file with 20 classes and 10000 lines?
  - 20 .py files, each containing a single class?
- Most programmers prefer the latter
- Smaller source files are easier to maintain

Question

- Which is better?
  - 20 files all defined at the top-level

```
foo.py
bar.py
grok.py
```

  - 20 files grouped in a directory

```
spam/
    foo.py
    bar.py
    grok.py
```

- Clearly, latter option is easier to manage

Question

- Which is better?
  - One module import

```
from spam import Foo, Bar, Grok
```

  - Importing dozens of submodules

```
from spam.foo import Foo
from spam.bar import Bar
from spam.grok import Grok
```

- I prefer the former (although it depends)
- "Fits my brain"

Modules vs. Packages

- Modules are easy--a single file
- Packages are hard--multiple related files
- Some Issues
    - Code organization
    - Connections between submodules
    - Desired usage
- It can get messy

Implicit Relative Imports

- Don't use implicit relative imports in packages

```
spam/
     __init__.py
     foo.py
     bar.py
```

- Example :

```
# bar.py
import foo    # Relative import of foo submodule
```

- It "works" in Python 2, but not Python 3

Absolute Imports

- Alternative: Use an absolute module import

```
spam/
      __init__.py
      foo.py
      bar.py
```

- Example :

```
# bar.py

from spam import foo
```

- Notice use of top-level package name
- I don't really like it (verbose, fragile)

Explicit Relative Imports

- A better approach

```
spam/
      __init__.py
      foo.py
      bar.py
```

- Example:

```
# bar.py

from . import foo      # Import from same level
```

- Leading dots (.) used to move up hierarchy

```
from . import foo          # Loads ./foo.py
from .. import foo         # Loads ../foo.py
from ..grok import foo     # Loads ../grok/foo.py
```

Explicit Relative Imports

- Allow packages to be easily renamed

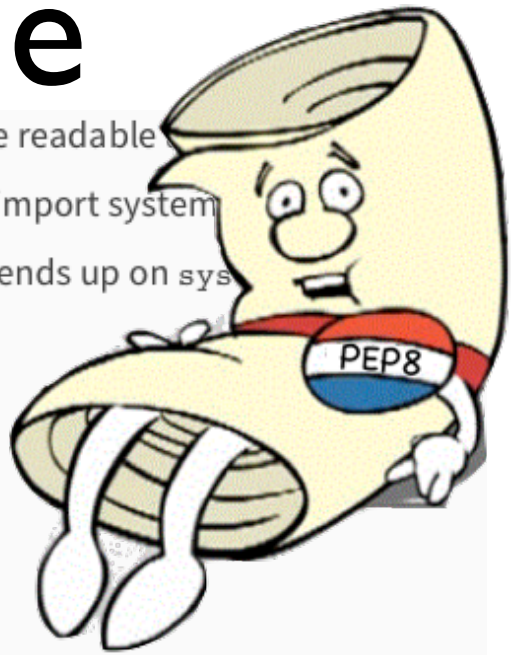
```
spam/                          grok/
     __init__.py                    __init__.py
     foo.py                         foo.py
     bar.py                         bar.py
```

- Explicit relative imports still work unchanged

```
# bar.py

from . import foo       # Import from same level
```

- Useful for moving code around, versioning, etc.

Let's Talk Style



- Absolute imports are recommended, as they are usually more readable
  better behaved (or at least give better error messages) if the import system
  rectly configured (such as when a directory inside a package ends up on sys

```
import mypkg.sibling
from mypkg import sibling
from mypkg.sibling import example
```

However, explicit relative imports are an acceptable alternative to absolute imports, especially when dealing with complex package layouts where using absolute imports would be unnecessarily verbose:

```
from . import sibling
from .sibling import example
```

Standard library code should avoid complex package layouts and always use absolute imports.

Let's Talk Style

NO

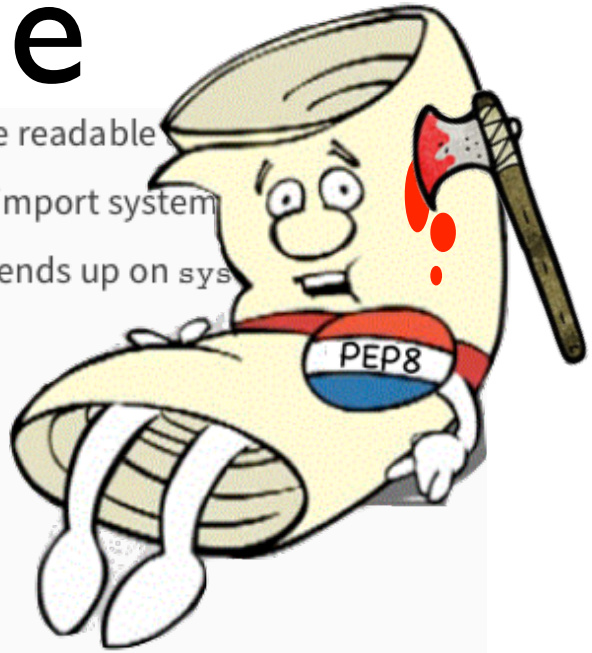
Absolute imports are recommended, as they are usually more readable and tend to be better behaved (or at least give better error messages) if the import system is incorrectly configured (such as when a directory inside a package ends up on sys.path):

```
import mypkg.sibling
from mypkg import sibling
from mypkg.sibling import example
```

However, explicit relative imports are an acceptable alternative to absolute imports, especially when dealing with complex package layouts where using absolute imports would be unnecessarily verbose:

```
from . import sibling
from .sibling import example
```

Standard library code should avoid complex package layouts and always use absolute imports.



Let's Talk Style

NO

Absolute imports are recommended, as they are usually more readable and tend to be better behaved (or at least give better error messages) if the import system is incorrectly configured (such as when a directory inside a package ends up on sys.path):

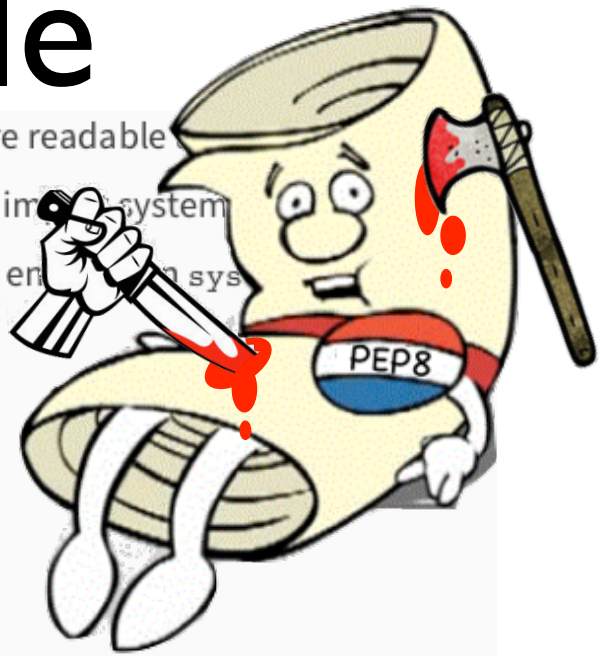
```
import mypkg.sibling
from mypkg import sibling
from mypkg.sibling import example
```

However, explicit relative imports are an acceptable alternative to absolute imports, especially when dealing with complex package layouts where using absolute imports would be unnecessarily verbose:

```
from . import sibling
from .sibling import example
```

NO

Standard library code should avoid complex package layouts and always use absolute imports.

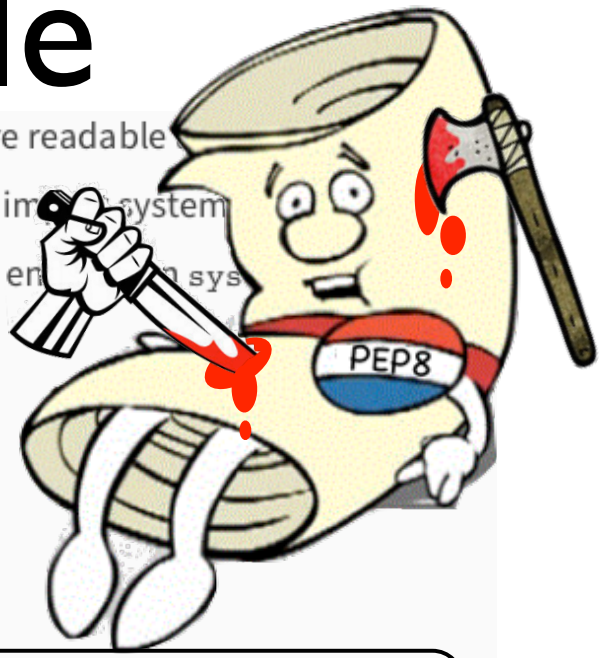


Let's Talk Style

NO

Absolute imports are recommended, as they are usually more readable and tend to be better behaved (or at least give better error messages) if the import system is incorrectly configured (such as when a directory inside a package ends up on sys.path):

```
import mypkg.sibling
from mypkg import sibling
from mypkg.sibling import example
```



YES

However, explicit relative imports are an acceptable alternative to absolute imports, especially when dealing with complex package layouts where using absolute imports would be unnecessarily verbose:

```
from . import sibling
from .sibling import example
```

NO

Standard library code should avoid complex package layouts and always use absolute imports.

Commentary

- PEP-8 predates explicit relative imports
- I think its advice is sketchy on this topic
- Please use explicit relative imports
- They ARE used in the standard library

__init__.py

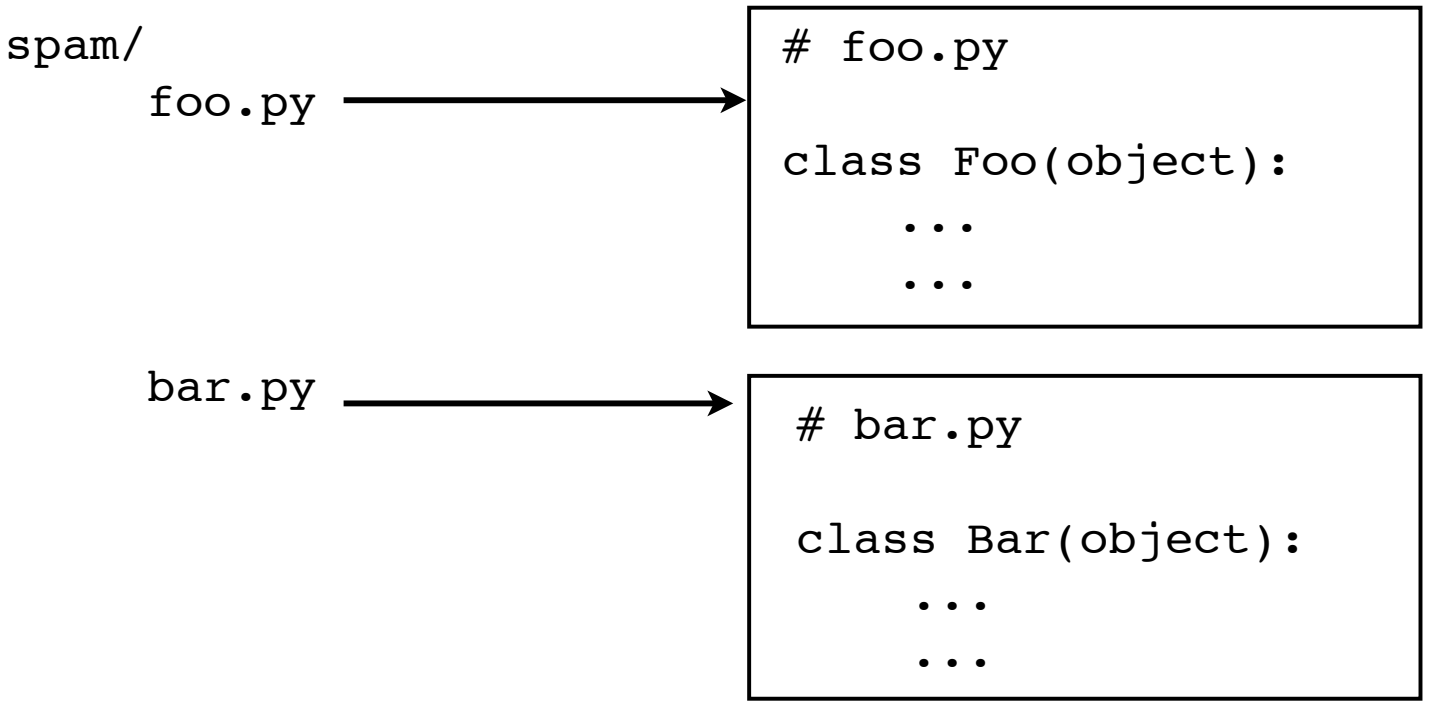
"From hell's heart I stab at thee."

__init__.py

- What are you supposed to do in those files?
- Claim: I think they should mainly be used to stitch together multiple source files into a "unified" top-level import (if desired)
- Example: Combining multiple Python files, building modules involving C extensions, etc.

Module Assembly

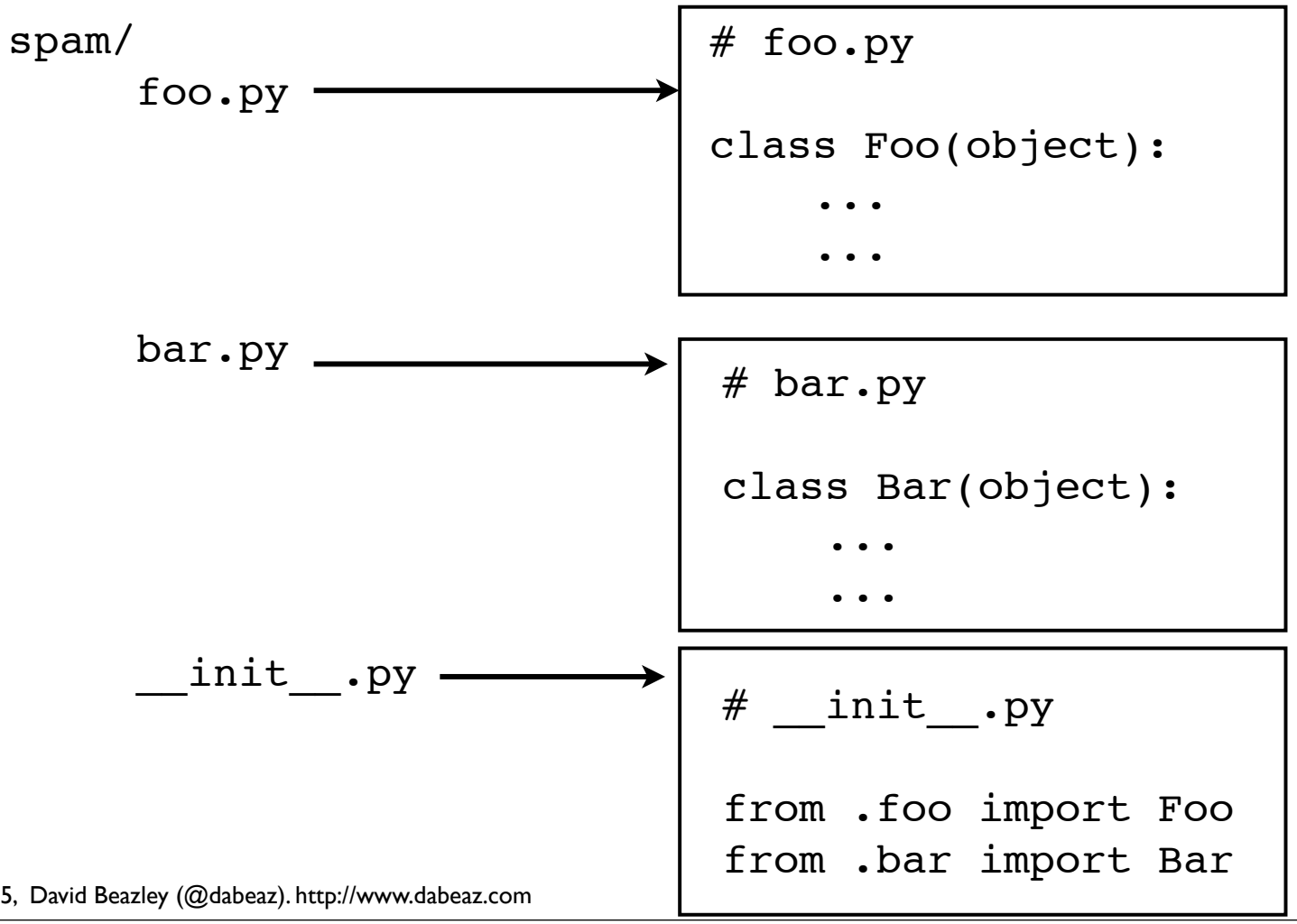
- Consider two submodules in a package



- Suppose you want to combine them

Module Assembly

- Combine in __init__.py



Module Assembly

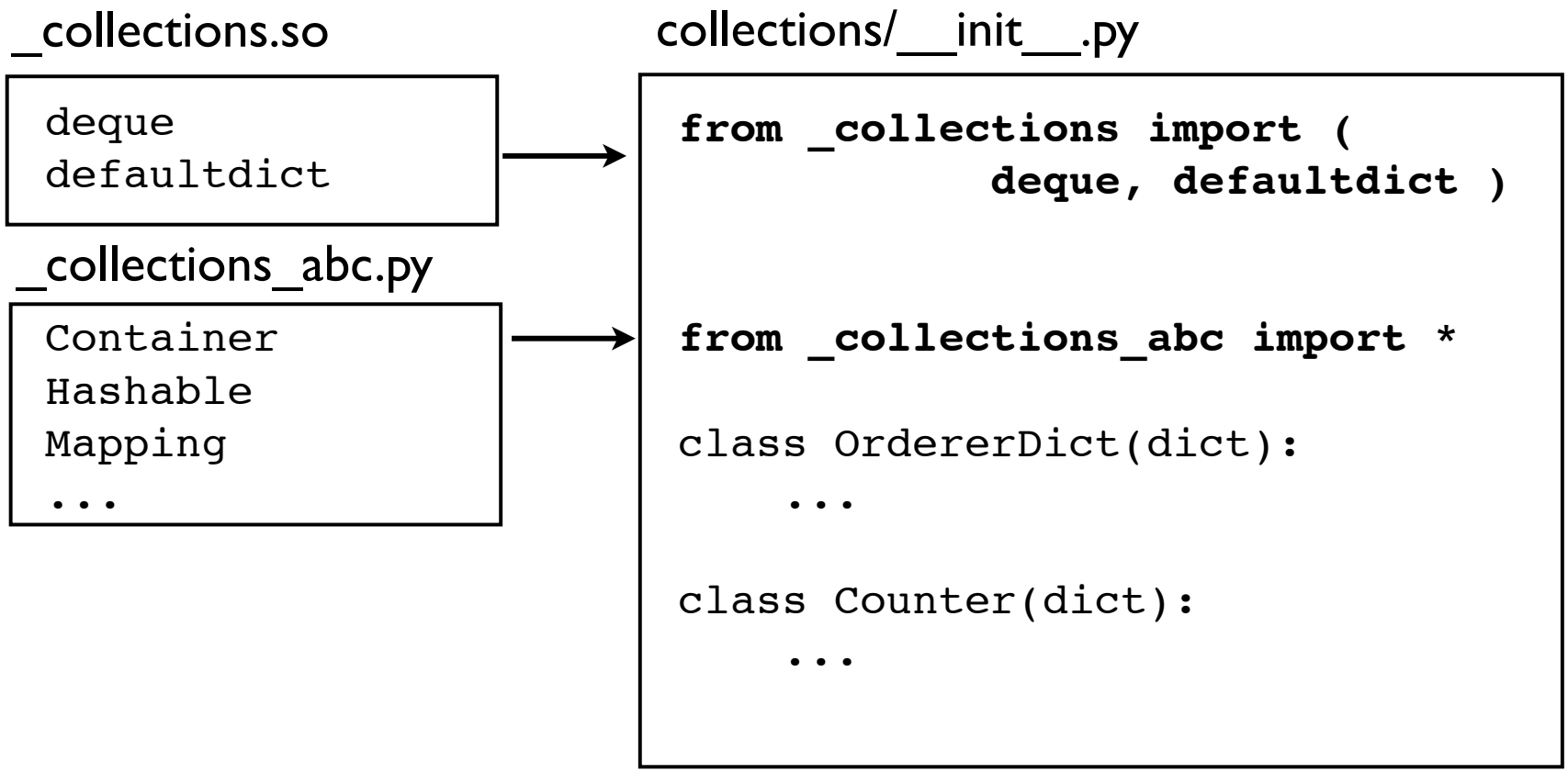
- Users see a single unified top-level package

```
import spam

f = spam.Foo()
b = spam.Bar()
...
```

- Split across files is an implementation detail

Case Study

- The collections "module"
- It's actually a package with a few components



Controlling Exports

- Each submodule should define __all__

```
# foo.py

__all__ = [ 'Foo' ]

class Foo(object):
    ...
```

```
# bar.py

__all__ = [ 'Bar' ]

class Bar(object):
    ...
```

- Controls behavior of 'from module import *'
- Allows easy combination in __init__.py

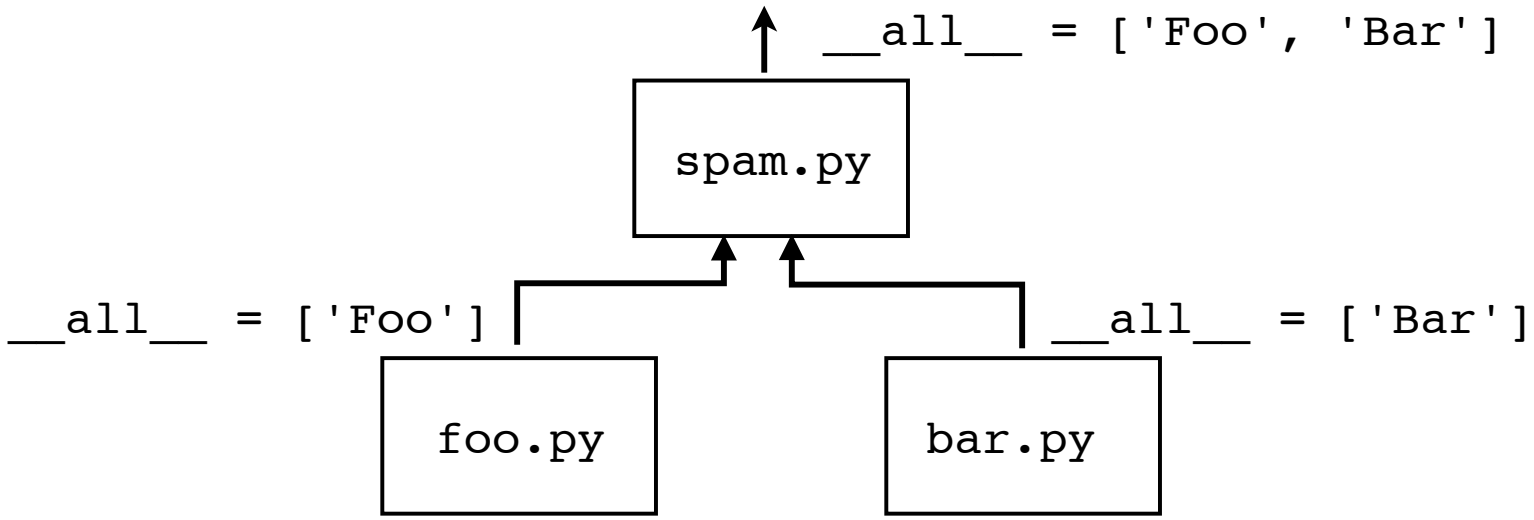
```
# __init__.py
from .foo import *
from .bar import *

__all__ = (foo.__all__ + bar.__all__)
```

Controlling Exports

- The last step is subtle

```
__all__ = (foo.__all__ + bar.__all__)
```

- Ensures proper propagation of exported symbols to the top level of the package



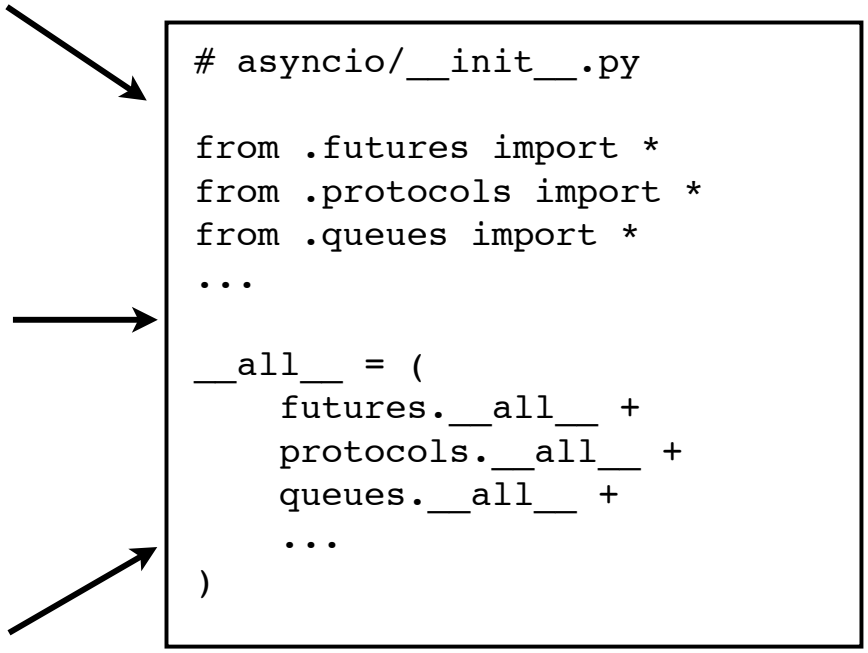
Case Study

- Look at implementation of asyncio (stdlib)

```
# asyncio/futures.py
__all__ = ['CancelledError',
           'TimeoutError',
           'InvalidStateError',
           'Future',
           'wrap_future']
```

```
# asyncio/protocols.py
__all__ = ['BaseProtocol',
           'Protocol',
           'DatagramProtocol',
           'SubprocessProtocol']
```

```
# asyncio/queues.py
__all__ = ['Queue',
           'PriorityQueue',
           'LifoQueue',
           'JoinableQueue',
           'QueueFull',
           'QueueEmpty']
```



```
# asyncio/__init__.py

from .futures import *
from .protocols import *
from .queues import *
...

__all__ = (
    futures.__all__ +
    protocols.__all__ +
    queues.__all__ +
    ...
)
```

An Export Decorator

- I sometimes use an explicit export decorator

```
# spam/__init__.py

__all__ = []

def export(defn):
    globals()[defn.__name__] = defn
    __all__.append(defn.__name__)
    return defn

from . import foo
from . import bar
```

- Will use it to tag exported definitions
- Might use it for more (depends)

An Export Decorator

- Example usage

```
# spam/foo.py

from . import export

@export
def blah():
    ...

@export
class Foo(object):
    ...
```

- Benefit: exported symbols are clearly marked in the source code.

Performance Concerns

- Should __init__.py import the universe?
- For small libraries, who cares?
- For large framework, maybe not (expensive)
- Will return to this a bit later
- For now: Think about about it

__init__.py Revisited

- Should __init__.py be used for other things?
  - Implementation of a "module"?
  - Path hacking?
  - Package upgrading?
  - Other weird hacks?

Implementation in __init__

- Is this good style?

```
spam/
     __init__.py
```



```
# __init__.py

class Foo(object):
    ...

class Bar(object):
    ...
```

- A one file package where everything is put inside __init__.py
- It feels sort of "wrong"
- __init__ connotes initialization, not implementation

__path__ hacking

- Packages define an internal __path__ variable

```
>>> import xml
>>> xml.__path__
['/usr/local/lib/python3.4/xml']
>>>
```

- It defines where submodules are located

```
>>> import xml.etree
>>> xml.etree.__file__
'/usr/local/lib/python3.4/xml/etree/__init__.py'
>>>
```

- Packages can hack it (in __init__.py)

```
__path__.append('/some/additional/path')
```

Package Upgrading

- A package can "upgrade" itself on import

```
# xml/__init__.py

try:
    import _xmlplus
    import sys
    sys.modules[__name__] = _xmlplus
except ImportError:
    pass
```

- Idea: Replace the sys.modules entry with a "better" version of the package (if available)
- FYI: xml package in Python2.7 does this

Other __init__.py Hacks

- Monkeypatching other modules on import?
- Other initialization (logging, etc.)
- My advice: Stay away. Far away.
- Simple __init__.py == good __init__.py

Part 3

__main__

Main Modules

- python -m module
- Runs a module as a main program

```
spam/
      __init__.py
      foo.py
      bar.py

bash % python3 -m spam.foo          # Runs spam.foo as main
```

- It's a bit special in that package relative imports and other features continue to work as usual

Main Modules

```
bash % python3 -m spam.foo
```

- Execution steps (pseudocode)

```
>>> import spam
>>> __package__ = 'spam'
>>> exec(open('spam/foo.py').read())
```

- Makes sure the enclosing package is imported
- Sets __package__ so relative imports work

Main Modules

- I like the -m option a lot
- Makes the Python version explicit

```
bash % python3 -m pip install package
```

vs

```
bash % pip install package
```

Rant: I can't count the number of times I've had to debug someone's Python installation because they're running some kind of "script", but they have no idea what Python it's actually attached to. The -m option avoids this.

Main Packages

- __main__.py designates main for a package
- Also makes a package directory executable

```
spam/
      __init__.py
      __main__.py                  # Main program
      foo.py
      bar.py

bash % python3 -m spam              # Run package as main
```

- Explicitly marks the entry point (good)
- Useful for a variety of other purposes

Executable Submodules

- Example

```
spam/
    __init__.py
    core/
        __init__.py          ⟶  import spam.core
        foo.py
        bar.py
    test/
        __init__.py
        __main__.py          ⟶  python3 -m spam.test
        foo.py
        bar.py
    server/
        __init__.py          ⟶  python3 -m spam.server
        __main__.py
        ...
```

- A useful organizational tool

Writing a Main Wrapper

- Make a tool that wraps around a script
- Examples:

```
bash % python3 -m profile someprogram.py
bash % python3 -m pdb someprogram.py
bash % python3 -m coverage run someprogram.py
bash % python3 -m trace --trace someprogram.py
...
```

- Many programming tools work this way

Writing a Script Wrapper

- Sample implementation

```
import sys
import os.path

def main():
    ...
    sys.argv[:] = sys.argv[1:]
    progname = sys.argv[0]
    sys.path.insert(0, os.path.dirname(progname))
    with open(progname, 'rb') as fp:
         code = compile(fp.read(), progname, 'exec')
    globs = {
         '__file__' : progname,
         '__name__' : '__main__',
         '__package__' : None,
         '__cached__' : None
    }
    exec(code, globs)
```

Must rewrite the command line arguments

Provide a new execution environment

Executable Directories

- Variant, Python can execute a raw directory
- Must contain __main__.py

```
spam/
    foo.py
    bar.py
    __main__.py

bash % python3 spam
```

- This also applies to zip files

```
bash % python3 -m zipfile -c spam.zip spam/*
bash % python3 spam.zip
```

Executable Directories

- Obscure fact: you can prepend a zip file with #! to make it executable like a script (since Py2.6)

```
spam/
    foo.py
    bar.py
    __main__.py

bash % python3 -m zipfile -c spam.zip spam/*
bash % echo -e '#!/usr/bin/env python3\n' > spamapp
bash % cat spam.zip >>spamapp
bash % chmod +x spamapp
bash % ./spamapp
```

- See PEP-441 for improved support of this

Part 4

sys.path

Let's Talk ImportError

- Almost every tricky problem concerning modules/packages is related to sys.path

```
>>> import sys
>>> sys.path
['',
 '/usr/local/lib/python34.zip',
 '/usr/local/lib/python3.4',
 '/usr/local/lib/python3.4/plat-darwin',
 '/usr/local/lib/python3.4/lib-dynload',
 '/usr/local/lib/python3.4/site-packages']
```

- Not on sys.path? Won't import. End of story.
- Package managers/install tools love sys.path

sys.path

- It's a list of strings
  - Directory name
  - Name of a .zip file
  - Name of an .egg file
- Traversed start-to-end looking for imports
- First match wins

.zip Files

- .zip files added to sys.path work as if they were normal directories
- Example: Creating a .zip file

```
% python3 -m zipfile -c myfiles.zip blah.py foo.py
%
```

- Using a .zip file

```
>>> import sys
>>> sys.path.append('myfiles.zip')
>>> import blah      # Loads myfiles.zip/blah.py
>>> import foo       # Loads myfiles.zip/foo.py
>>>
```

.egg Files

- .egg files are actually just directories or .zip files with extra metadata (for package managers)

```
% python3 -m zipfile -l blah-1.0-py3.4.egg
blah.py
foo.py
EGG-INFO/zip-safe
EGG-INFO/top_level.txt
EGG-INFO/SOURCES.txt
EGG-INFO/PKG-INFO
EGG-INFO/dependency_links.txt
...
```

- Associated with setuptools

Types of Modules

- Python looks for many different kinds of files

```
>>> import spam
```

- What it looks for (in each path directory)

`spam/`	Package directory

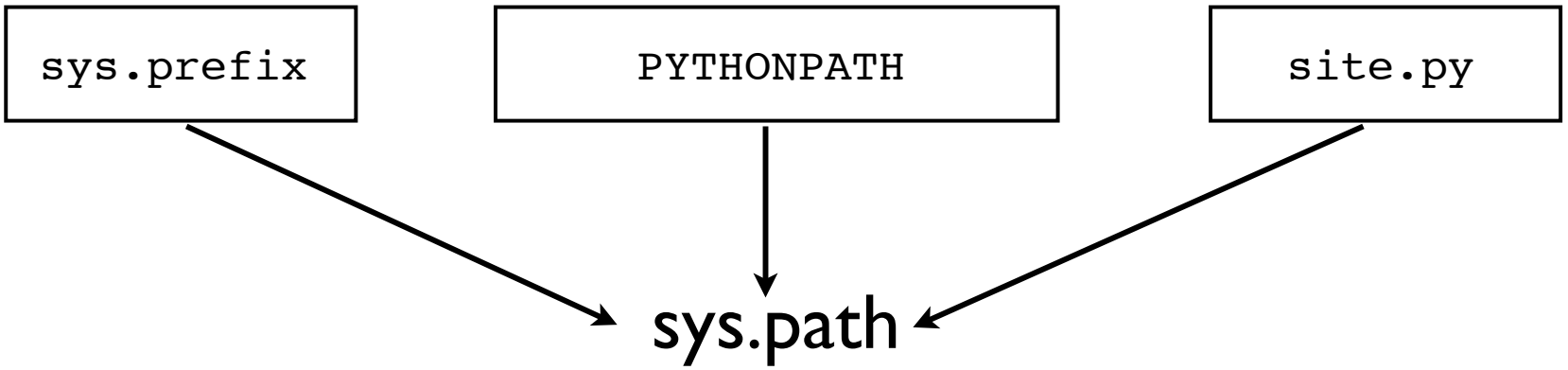
`spam.cpython-34m.so`
`spam.abi3.so`
`spam.so`
	C Extensions
	(not allowed in .zip/.egg)

`spam.py`	Python source file
`__pycache__/spam.cpython-34.pyc` `spam.pyc`	Compiled Python

- Run python3 -vv to see verbose output

Path Construction

- sys.path is constructed from three parts



- Let's deconstruct it

Path Construction

- Path settings of a base Python installation

```
bash % python3 -S          # -S skips site.py initialization
>>> sys.path
[
 '',
 '/usr/local/lib/python34.zip',
 '/usr/local/lib/python3.4/',
 '/usr/local/lib/python3.4/plat-darwin',
 '/usr/local/lib/python3.4/lib-dynload'
]
>>>
```

- These define the location of the standard library

sys.prefix

- Specifies base location of Python installation

```
>>> import sys
>>> sys.prefix
'/usr/local'
>>> sys.exec_prefix
'/usr/local'
>>>
```

- exec_prefix is location of compiled binaries (C)
- Python standard libraries usually located at

```
sys.prefix + '/lib/python3X.zip'
sys.prefix + '/lib/python3.X'
sys.prefix + '/lib/python3.X/plat-sysname'
sys.exec_prefix + '/lib/python3.X/lib-dynload'
```

sys.prefix Setting

- Python binary location determines the prefix

```
bash % which python3
/usr/local/bin/python3
bash %
```

```
sys.prefix = '/usr/local'
```

- However, it's far more nuanced than this
  - Environment variable check
  - Search for "installation" landmarks
  - Virtual environments

PYTHONHOME

- PYTHONHOME environment overrides location

```
bash % env PYTHONHOME=prefix[:execprefix] python3
```

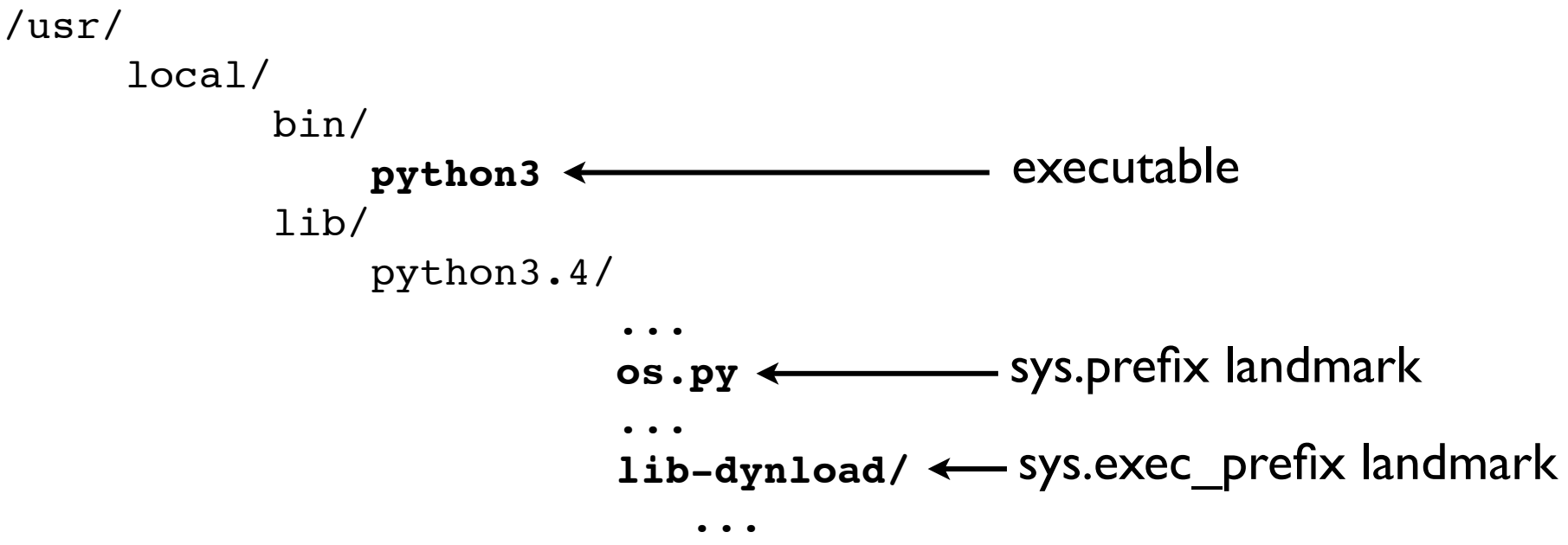
- Example: make a copy of the standard library

```
bash % mkdir -p mylib/lib
bash % cp -R /usr/local/lib/python3.4 mylib/lib
bash % env PYTHONHOME=mylib python3 -S
>>> import sys
>>> sys.path
['', 'mylib/lib/python34.zip', 'mylib/lib/python3.4/', ...]
>>>
```

- Please, don't do that though...

sys.prefix Landmarks

- Certain library files must exist



- Python searches for them and sets sys.prefix

sys.prefix Landmarks

- Suppose Python3 is located here

```
/Users/beazley/software/bin/python3
```

- sys.prefix checks (also checks .pyc files)

```
/Users/beazley/software/lib/python3.4/os.py
/Users/beazley/lib/python3.4/os.py
/Users/lib/python3.4/os.py
/lib/python3.4/os.py
```

- sys.exec_prefix checks

```
/Users/beazley/software/lib/python3.4/lib-dynload
/Users/beazley/lib/python3.4/lib-dynload
/Users/lib/python3.4/lib-dynload
/lib/python3.4/lib-dynload
```

Last Resort

- sys.prefix is hard-coded into python (getpath.c)

```
/* getpath.c */
#ifndef PREFIX
#define PREFIX "/usr/local"
#endif

#ifndef EXEC_PREFIX
#define EXEC_PREFIX PREFIX
#endif
```

- This is set during compilation/configuration

Commentary

- Control of sys.prefix is a major part of tools that package Python in custom ways
- Historically: virtualenv (Python 2)
- Modern: pyvenv (Python 3, in standard library)
- Of possible use in other settings (embedding, etc.)

Path Construction

- PYTHONPATH environment variable

```
bash % env PYTHONPATH=/foo:/bar python3 -S
>>> sys.path
['',
 '/foo',
 '/bar',
 '/usr/local/lib/python34.zip',
 '/usr/local/lib/python3.4/',
 '/usr/local/lib/python3.4/plat-darwin',
 '/usr/local/lib/python3.4/lib-dynload'
]
>>>
```

notice addition of the environment paths

- Paths in PYTHONPATH go first!

Path Construction

- site.py adds third-party module directories

```
bash % env PYTHONPATH=/foo:/bar python3
>>> sys.path
['',
 '/foo',
 '/bar',
 '/usr/local/lib/python34.zip',
 '/usr/local/lib/python3.4',
 '/usr/local/lib/python3.4/plat-darwin',
 '/usr/local/lib/python3.4/lib-dynload',
 '/Users/beazley/.local/lib/python3.4/site-packages',
 '/usr/local/lib/python3.4/site-packages']
>>>
```

- This is where packages install

notice addition of two site-packages directories

site-packages

- Default settings
- System-wide site-packages (pip install)

```
'/usr/local/lib/python3.4/site-packages'
```

- User site-packages (pip install --user)

```
'/Users/beazley/.local/lib/python3.4/site-packages'
```

- Sometimes, linux distros add their own directory

```
'/usr/local/lib/python3.4/dist-packages
```

Virtual Environments

- Makes a Python virtual environment

```
bash % python3 -m venv spam  ⟶  spam/
                                    pyvenv.cfg
                                    bin/
                                       activate
                                       easy_install
                                       pip
                                       python3
                                    include/
                                       ...
                                    lib/
                                        python3.4/
                                              site-packages/
```

- A fresh "install" with no third-party packages
- Includes python, pip, easy_install for setting up a new environment
- I prefer 'python3 -m venv' over the script 'pyvenv'

venv site-packages

- Suppose you have a virtual environment

```
/Users/
     beazley/
             mypython/
                 pyvenv.cfg
                 bin/
                     python3
                 lib/
                     python3.4/
                               ...
                               site-packages/
```

- venv site-packages gets used instead of defaults

venv site-packages

- Example

```
bash % python3 -m venv mypython
bash % mypython/bin/python3
>>> import sys
>>> sys.path
['',
 '/usr/local/lib/python34.zip',
 '/usr/local/lib/python3.4',
 '/usr/local/lib/python3.4/plat-darwin',
 '/usr/local/lib/python3.4/lib-dynload',
 '/Users/beazley/mypython/lib/python3.4/site-packages']
>>>
```

a single site-packages directory

venv site-packages

- Variant: Include system site-packages

```
bash % python3 -m venv --system-site-packages mypython
bash % mypython/bin/python3
>>> import sys
>>> sys.path
['',
 '/usr/local/lib/python34.zip',
 '/usr/local/lib/python3.4',
 '/usr/local/lib/python3.4/plat-darwin',
 '/usr/local/lib/python3.4/lib-dynload',
 '/Users/beazley/mypython/lib/python3.4/site-packages',
 '/Users/beazley/.local/lib/python3.4/site-packages',
 '/usr/local/lib/python3.4/site-packages']
>>>
```

Get the system site-packages and that of the virtual environment

.pth Files

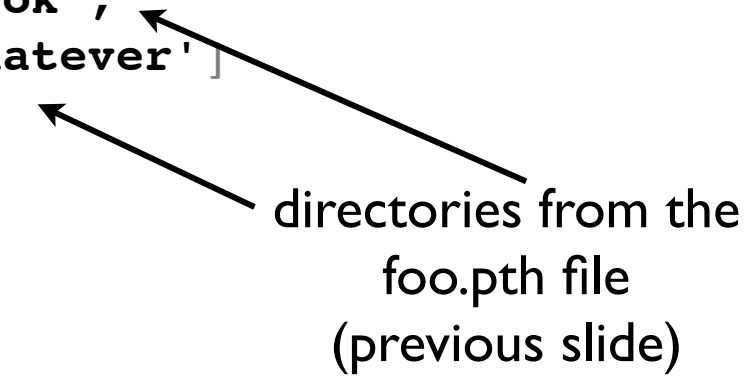
- A further technique of extending sys.path
- Make a file with a list of additional directories

```
# foo.pth
./spam/grok
./blah/whatever
```

- Copy this file to any site-packages directory
- All directories that exist are added to sys.path

.pth Files

```
bash % env PYTHONPATH=/foo:/bar python3
>>> sys.path
['',
 '/foo',
 '/bar',
 '/usr/local/lib/python34.zip',
 '/usr/local/lib/python3.4',
 '/usr/local/lib/python3.4/plat-darwin',
 '/usr/local/lib/python3.4/lib-dynload',
 '/Users/beazley/.local/lib/python3.4/site-packages',
 '/usr/local/lib/python3.4/site-packages',
 '/usr/local/lib/python3.4/spam/grok',
 '/usr/local/lib/python3.4/blah/whatever']
>>>
```



.pth Files

- .pth files mainly used by package managers to install packages in additional directories
- Example: adding '.egg' files to the path

```
>>> sys.path
['',
 '/usr/local/lib/python3.4/site-packages/ply-3.4-py3.4.egg',
 '/usr/local/lib/python34.zip',
 '/usr/local/lib/python3.4',
 '/usr/local/lib/python3.4/plat-darwin',
 '/usr/local/lib/python3.4/lib-dynload',
 ...
]
```

- But, it gets even better!

.pth "import" hack

- Any line starting with 'import' is executed
- Example: setuptools.pth

```
import sys; sys.__plen = len(sys.path)
./ply-3.4-py3.4.egg
import sys; new=sys.path[sys.__plen:]; del sys.path \
[sys.__plen:]; p=getattr(sys,'__egginsert',0); \
sys.path[p:p]=new; sys.__egginsert = p+len(new)
```

- Package managers and extensions can use this to perform automagic steps upon Python startup
- No patching of other files required

.pth "import" hack

[Python-Dev] Extending startup code: PEP needed?

Martin v. Loewis martin@loewis.home.cs.tu-berlin.de
Sun, 7 Jan 2001 19:45:15 +0100

```
Authors of extension packages often find the need to auto-import some
of their modules. This is often needed for registration, e.g. a codec
author (like Tamito KAJIYAMA, who wrote the JapaneseCodecs package)
may need to register a search function with codecs.register. This is
currently only possible by writing into sitecustomize.py, which must
be done by the system administrator manually.

To enhance the service of site.py, I've written the patch

http://sourceforge.net/patch/?func=detailpatch&patch_id=103134&group_id=5470

which treats lines in PTH files which start with "import" as
statements and executes them, instead of appending these lines to
sys.path.
```

(site|user)customize.py

- Final steps of site.py initialization
  - import sitecustomize
  - import usercustomize
- ImportError silently ignored (if not present)
- Both imports may further change sys.path

Script Directory

- First path component is same directory as the running script (or current working directory)
- It gets added last

```
bash % python3 programs/script.py
>>> import sys
>>> sys.path
['/Users/beazley/programs/',
 '/usr/local/lib/python34.zip',
 '/usr/local/lib/python3.4',
 '/usr/local/lib/python3.4/plat-darwin',
 '/usr/local/lib/python3.4/lib-dynload',
 ...
]
```


Added last

Locking Out Users

- You can lock-out user customizations to the path

```
python3 -E ...          # Ignore environment variables
python3 -s ...          # Ignore user site-packages
python3 -I ...          # Same as -E -s
```

- Example:

```
bash % env PYTHONPATH=/foo:/bar python3 -I
>>> import sys
>>> sys.path
['',
 '/usr/local/lib/python34.zip',
 '/usr/local/lib/python3.4',
 '/usr/local/lib/python3.4/plat-darwin',
 '/usr/local/lib/python3.4/lib-dynload',
 '/usr/local/lib/python3.4/site-packages']
>>>
```

- Maybe useful in #! scripts

Package Managers

- easy_install, pip, conda, etc.
- They all basically work within this environment
- Installation into site-packages, etc.
- Differences concern locating, downloading, building, dependencies, and other aspects.
- Do I want to discuss further? Nope.

Part 5

Namespace Packages

Die __init__.py Die!

- Bah, you don't even need it!

```
spam/
      foo.py
      bar.py
```

- It all works fine without it! (No, Really)

```
>>> import spam.foo
>>> import spam.bar
>>> spam.foo
<module 'spam.foo' from 'spam/foo.py'>
>>>
```

- Wha!?!??? (Don't try in Python 2)

Namespace Packages

- Omit __init__.py and you get a "namespace"

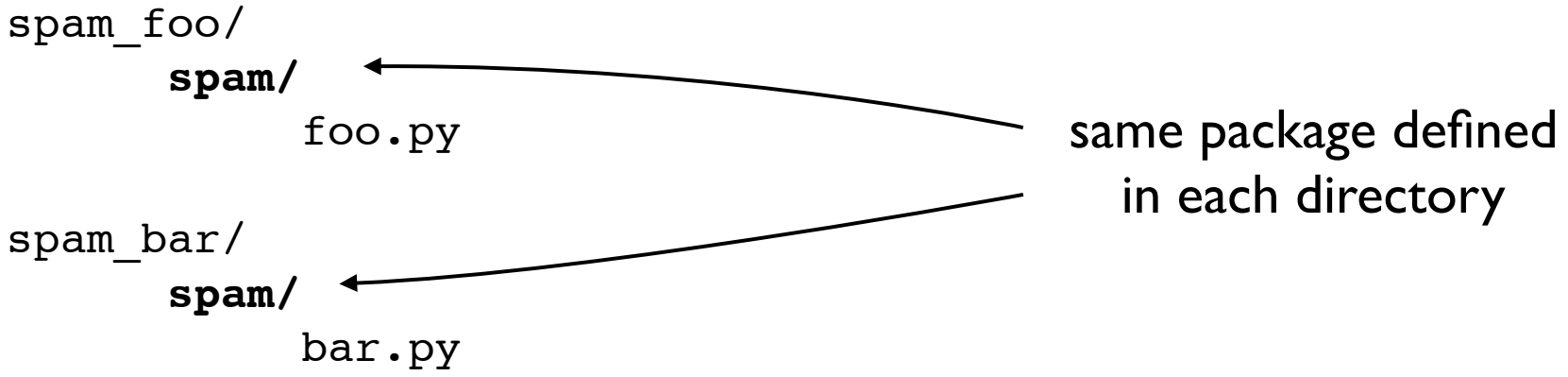
```
spam/
     foo.py
     bar.py
```

```
>>> import spam
>>> spam
<module 'grok' (namespace)>
>>>
```

- A namespace for what?
- For building an extensible library of course!

Namespace Packages

- Suppose you have two directories like this



- Both directories contain the same top-level package name, but different subparts

Namespace Packages

- Put both directories on sys.path.

```
>>> import sys
>>> sys.path.extend(['spam_foo','spam_bar'])
>>>
```

- Now, try some imports--watch the magic!

```
>>> import spam.foo
>>> import spam.bar
>>> spam.foo
<module 'spam.foo' from 'spam_foo/spam/foo.py'>
>>> spam.bar
<module 'spam.bar' from 'spam_bar/spam/bar.py'>
>>>
```

- Two directories become one!

How it Works

- Packages have a magic __path__ variable

```
>>> import xml
>>> xml.__path__
['/usr/local/lib/python3.4/xml']
>>>
```

- It's a list of directories searched for submodules
- For a namespace, all matching paths get collected

```
>>> spam.__path__
_NamespacePath(['spam_foo/spam', 'spam_bar/spam'])
>>>
```

- Only works if no __init__.py in top level

How it Works

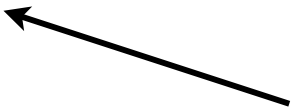
- Namespace __path__ is dynamically updated

```
spam_grok/
      spam/
           grok.py
```

```
>>> sys.path.append('spam_grok')
```

- Watch it update

```
>>> spam.__path__
_NamespacePath(['spam_foo/spam', 'spam_bar/spam'])
>>> import spam.grok
>>> spam.__path__
_NamespacePath(['spam_foo/spam', 'spam_bar/spam',
'spam_grok/spam'])
>>>
```



Notice how the new path is added

Applications

- Namespace packages might be useful for framework builders who want to have their own third-party plugin system
- Example: User-customized plugin directories

Challenge

Build a user-extensible framework

"Telly"

Telly In a Nutshell

- There is a framework core

```
telly/
    __init__.py
    ...
```

- There is a plugin area ("Tubbytronic Superdome")

```
telly/
    __init__.py
    ...
    tubbytronic/
        laalaa.py
        ...
```

Telly Plugins

- Telly allows user-specified plugins (in $HOME)

```
~/.telly/
    telly-dipsy/
        tubbytronic/
            dipsy.py

    telly-po/
        tubbytronic/
            po.py
```

- Not installed as part of main package

Our Task

- Figure out some way to unify all of the plugins in the same namespace

```
>>> from telly.tubbytronic import laalaa
>>> from telly.tubbytronic import dipsy
>>> from telly.tubbytronic import po
>>>
```

- Even though the plugins are coming from separately installed directories

Illustrated

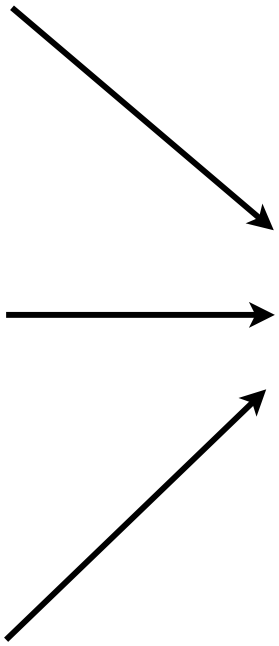
File System Layout

```
telly/
    __init__.py
    tubbytronic/
        laalaa.py
```

```
~/.telly/
    telly-dipsy/
        tubbytronic/
            dipsy.py
```

```
~/.telly/
    telly-po/
        tubbytronic/
            po.py
```

Logical Package



```
telly/
    __init__.py
    tubbytronic/
        laalaa.py
        dipsy.py
        po.py
```

Strategy

- Create a namespace package subcomponent

```
# telly/__init__.py

...
__path__ = [
  '/usr/local/lib/python3.4/site-packages/telly/tubbytronic',
  '/Users/beazley/.telly/telly-dipsy/tubbytronic',
  '/Users/beazley/.telly/telly-po/tubbytronic'
]
```

- Again: merging a system install with user-plugins

Implementation

- Just a bit of __path__ hacking

```
# telly/__init__.py

import os
import os.path

user_plugins = os.path.expanduser('~/.telly')
if os.path.exists(user_plugins):
    plugins = os.listdir(user_plugins)
    for plugin in plugins:
        __path__.append(os.path.join(user_plugins, plugin))
```

- Does it work?

Example

- Try it:

```
>>> from telly import tubbytronic
>>> tubbytronic.__path__
_NamespacePath([
   '/usr/local/lib/python3.4/site-packages/telly/tubbytronic',
   '/Users/beazley/.telly/telly-dipsy/tubbytronic',
   '/Users/beazley/.telly/telly-po/tubbytronic'])
>>> from telly.tubbytronic import laalaa
>>> from telly.tubbytronic import dipsy
>>> laalaa.__file__
'.../python3.4/site-packages/telly/tubbytronic/laalaa.py'
>>> dipsy.__file__
'/Users/beazley/.telly/telly-dipsy/tubbytronic/dipsy.py'
>>>
```

- Cool!

Thoughts

- Namespace packages are kind of insane
- Only thing more insane: Python 2 implementation of the same thing (involving setuptools, etc.)
- One concern: Packages now "work" if users forget to include __init__.py files
- Wonder if they know how much magic happens

Part 6

The Module

What is a Module?

- A file of source code
- A namespace
- Container of global variables
- Execution environment for statements
- Most fundamental part of a program?

Module Objects

- A module is an object (you can make one)

```
>>> from types import ModuleType
>>> spam = ModuleType('spam')
>>> spam
<module 'spam'>
>>>
```

- It wraps around a dictionary

```
>>> spam.__dict__
{'__loader__': None, '__doc__': None,
'__name__': 'spam', '__spec__': None,
'__package__': None}
>>>
```

Module Attributes

- Attribute access manipulates the dict

```
spam.x              # return spam.__dict__['x']
spam.x = 42         # spam.__dict__['x'] = 42
del spam.x          # del spam.__dict__['x']
```

- That's it!
- A few commonly defined attributes

```
__name__            # Module name
__file__            # Associated source file (if any)
__doc__             # Doc string
__path__            # Package path
__package__         # Package name
__spec__            # Module spec
```

Modules vs. Packages

- A package is just a module with two defined (non-None) attributes

```
__package__          # Name of the package
__path__             # Search path for subcomponents
```

- Otherwise, it's the same object

```
>>> import xml
>>> xml.__package__
'xml'
>>> xml.__path__
['/usr/local/lib/python3.4/xml']
>>> type(xml)
<class 'module'>
>>>
```

Import Explained

- import creates a module object
- Executes source code inside the module
- Assigns the module object to a variable

```
>>> import spam
>>> spam
<module 'spam' from 'spam.py'>
>>>
```

- Creation is far more simple than you think

Module Creation

- Here's a minimal "implementation" of import

```
import types

def import_module(modname):
    sourcepath = modname + '.py'
    with open(sourcepath, 'r') as f:
        sourcecode = f.read()
    mod = types.ModuleType(modname)
    mod.__file__ = sourcepath
    code = compile(sourcecode, sourcepath, 'exec')
    exec(code, mod.__dict__)
    return mod
```

- It's barebones: But it works!

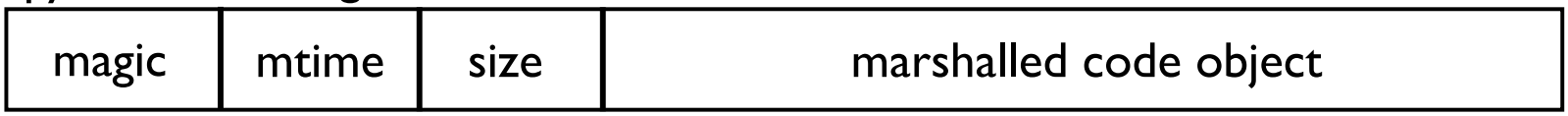
```
>>> spam = import_module('spam')
>>> spam
<module 'spam' from 'spam.py'>
>>>
```

Module Compilation

- Modules are compiled into '.pyc' files

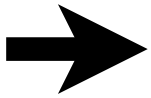
```
import marshal, os, importlib.util, sys

def import_module(modname):
    ...
    code = compile(sourcecode, sourcepath, 'exec')
    ...
    with open(modname + '.pyc', 'wb') as f:
        mtime = os.path.getmtime(sourcepath)
        size = os.path.getsize(sourcepath)
        f.write(importlib.util.MAGIC_NUMBER)
        f.write(int(mtime).to_bytes(4, sys.byteorder))
        f.write(int(size).to_bytes(4, sys.byteorder))
        marshal.dump(code, f)
```

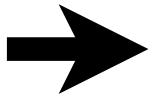
.pyc file encoding

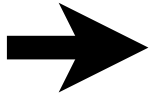


Module Cache

- Modules are cached. This is checked first







```
import sys, types

def import_module(modname):
    if modname in sys.modules:
        return sys.modules[modname]
    ...
    mod = types.ModuleType(modname)
    mod.__file__ = sourcepath

    sys.modules[modname] = mod

    code = compile(sourcecode, sourcepath, 'exec')
    exec(code, mod.__dict__)
    return sys.modules[modname]
```

- New module put in cache prior to exec

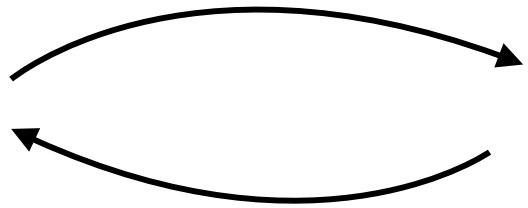
Module Cache

- The cache is a critical component of import
- There are some tricky edge cases
- Advanced import-related code might have to interact with it directly

Corner Case: Cycles

- Cyclic imports

```
# foo.py
import bar
...
```



```
# bar.py
import foo
...
```

- Repeated import picks module object in cache
- Python won't crash. It's fine
- Caveat: Module is only partially imported

Corner Case: Cycles

- Definition/import order matters

```
# foo.py

import bar

def spam():
    ...
```

```
# bar.py

import foo

x = foo.spam()
```

- Fail!

```
>>> import foo
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/beazley/.../foo.py", line 3, in <module>
    import bar
  File "/Users/beazley/.../bar.py", line 5, in <module>
    x = foo.spam()
AttributeError: 'module' object has no attribute 'spam'
>>>
```

Corner Case: Cycles

- Definition/import order matters

```
# foo.py

import bar

def spam():
    ...
```

```
# bar.py

import foo

x = foo.spam()
```

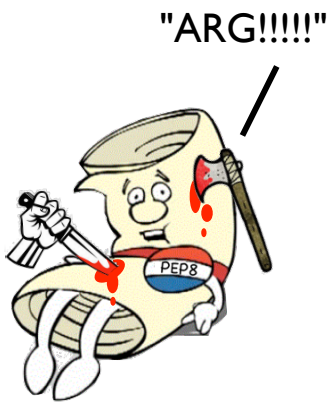
(Not Defined!)

- Follow the control flow
- A possible "fix" (move the import)

swap

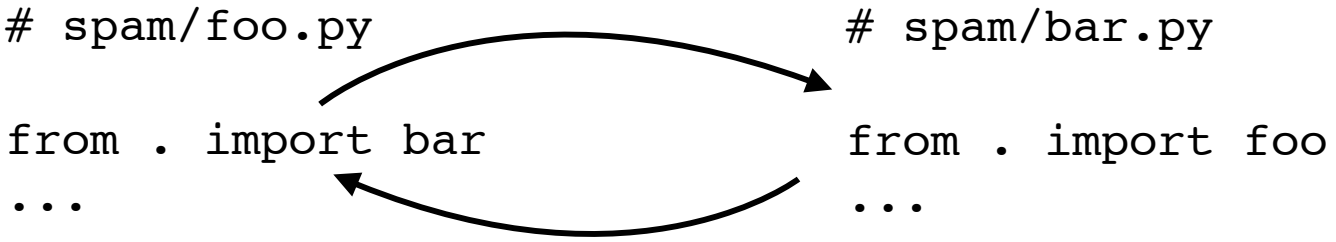
```
# foo.py

def spam():
    ...
import bar
```

```
# bar.py

import foo

x = foo.spam()
```



Evil Case: Package Cycles

- Cyclic imports in packages

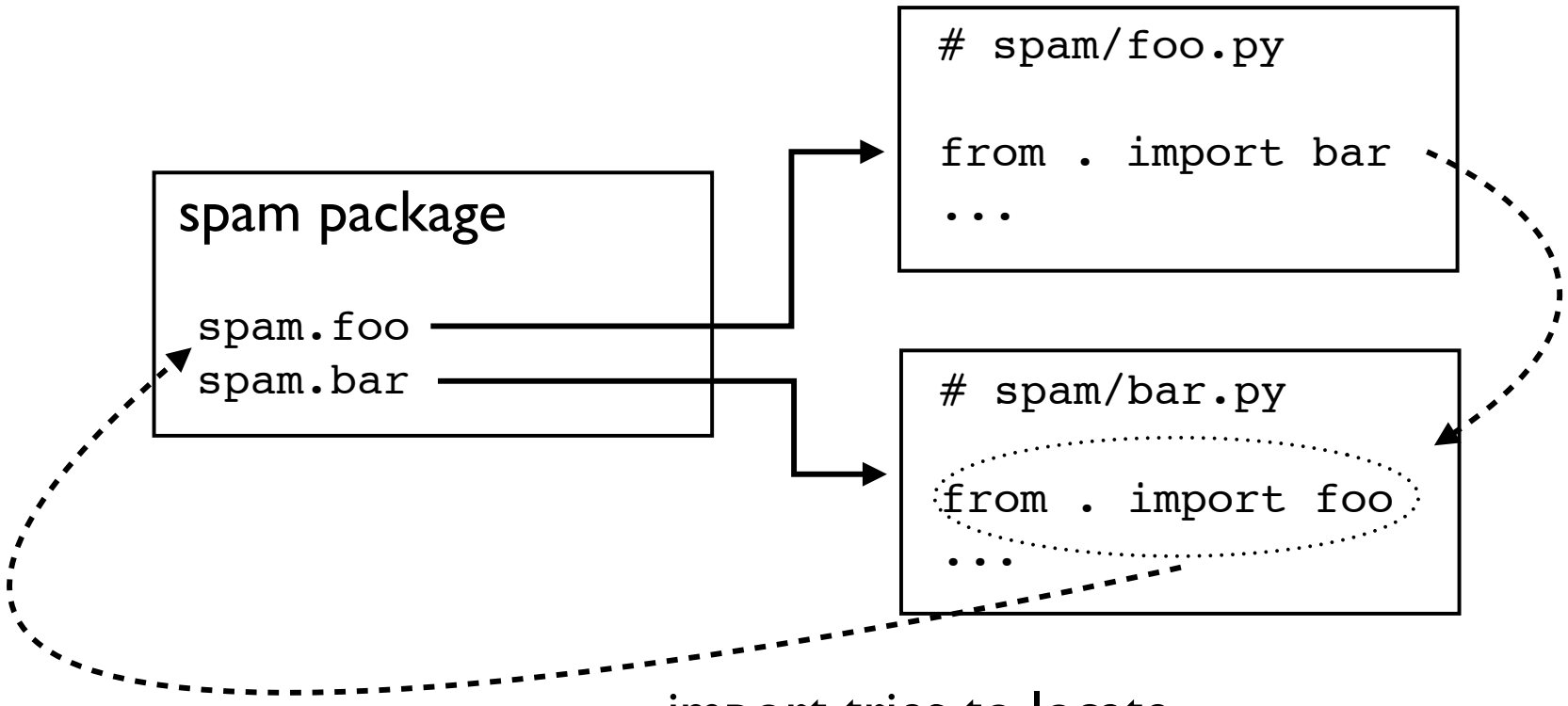


- This crashes outright

```
>>> import spam.foo
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "...spam/foo.py", line 1, in <module>
    from . import bar
  File "...spam/bar.py", line 1, in <module>
    from . import foo
ImportError: cannot import name 'foo'
>>>
```

Evil Case: Package Cycles

- Problem: Reference to a submodule only get created after the entire submodule imports



Evil Case: Package Cycles

- Can "fix" by realizing that sys.modules holds submodules as they are executing

```
# spam/bar.py

try:
   from . import foo
except ImportError:
   import sys
   foo = sys.modules[__package__ + '.foo']
```

- Commentary: This is a fairly obscure corner case--try to avoid import cycles if you can. That said, I have had to do this once in real-world production code.

Evil Case: Threads

- Imports can be buried in functions

```
def evil():
    import foo
    ...
    x = foo.spam()
```

- Functions can run in separate threads

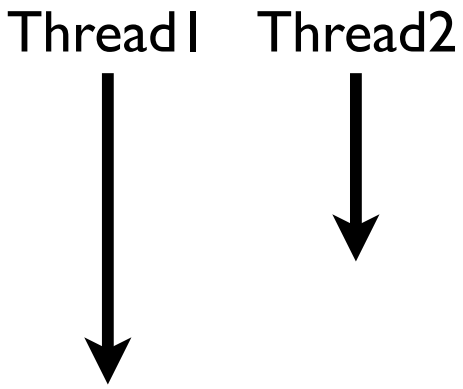
```
from threading import Thread

t1 = Thread(target=evil)
t2 = Thread(target=evil)
t1.start()
t2.start()
```

- Concurrent imports? Yikes!

Evil Case: Threads

- Possibility 1: The module executes twice

```
import sys, types

def import_module(modname):
    if modname in sys.modules:
        return sys.modules[modname]
    ...
    mod = types.ModuleType(modname)
    mod.__file__ = sourcepath

    sys.modules[modname] = mod
    ...
```

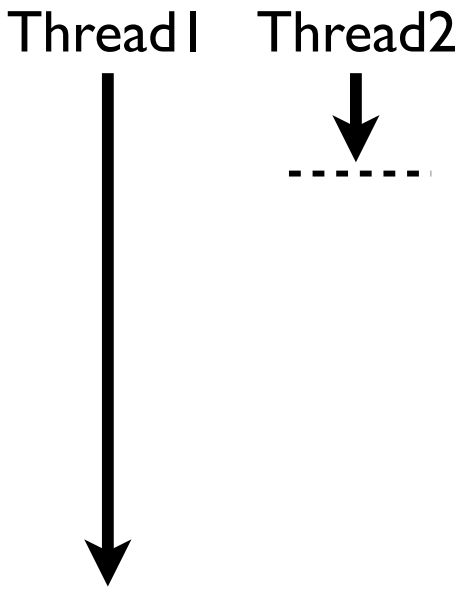


- Race condition related to creating/populating the module cache

Evil Case: Threads

- Possibility 2: One thread gets a partial module

```
import sys, types

def import_module(modname):
    if modname in sys.modules:
        return sys.modules[modname]
    ...
    mod = types.ModuleType(modname)
    mod.__file__ = sourcepath

    sys.modules[modname] = mod
    ...
```



- Thread getting cached copy might crash--module not fully executed yet

Import Locking

- imports are locked

```
from threading import RLock

_import_lock = RLock()

def import_module(modname):
    with _import_lock:
        if modname in sys.modules:
            return sys.modules[modname]
        ...
```

- Such a lock exists (for real)

```
>>> import imp
>>> imp.acquire_lock()
>>> imp.release_lock()
```

- Note: Not the same as the infamous GIL

Import Locking

- Actual implementation is a bit more nuanced
- Global import lock is only held briefly
- Each module has its own dedicated lock
- Threads can import different mods at same time
- Deadlock detection (concurrent circular imports)
- Advice: DON'T FREAKING DO THAT!

The Real "import"

- import is handled directly in bytecode

```
import spam
```

```
  LOAD_CONST               0 (0)
  LOAD_CONST               1 (None)
  IMPORT_NAME              0 (math)
  STORE_NAME               0 (math)
```

implicitly invokes

```
__import__('math', globals(), None, None, 0)
```

- __import__() is a builtin function
- You can call it!

__import__()

- __import__()

```
>>> spam = __import__('spam')
>>> spam
<module 'spam' from 'spam.py'>
>>>
```

- Direct use is possible, but discouraged
- A better alternative: importlib.import_module()

```
# Same as: import spam
spam = importlib.import_module('spam')

# Same as: from . import spam
spam = importlib.import_module('.spam', __package__)
```

Import Tracking

- Just for fun: Monkeypatch __import__ to track all import statements

```
>>> def my_import(modname *args, imp=__import__):
...     print('importing', modname)
...     return imp(modname, *args)
...
>>> import builtins
>>> builtins.__import__ = my_import
>>>
>>> import socket
importing socket
importing _socket
...
```

- Very exciting!

Interlude

- Important points:
  - Modules are objects
  - Basically just a dictionary (globals)
  - Importing is just exec() in disguise
  - Variations on import play with names
  - Tricky corner cases (threads, cycles, etc.)
- Modules are fundamentally simple

Subclassing Module

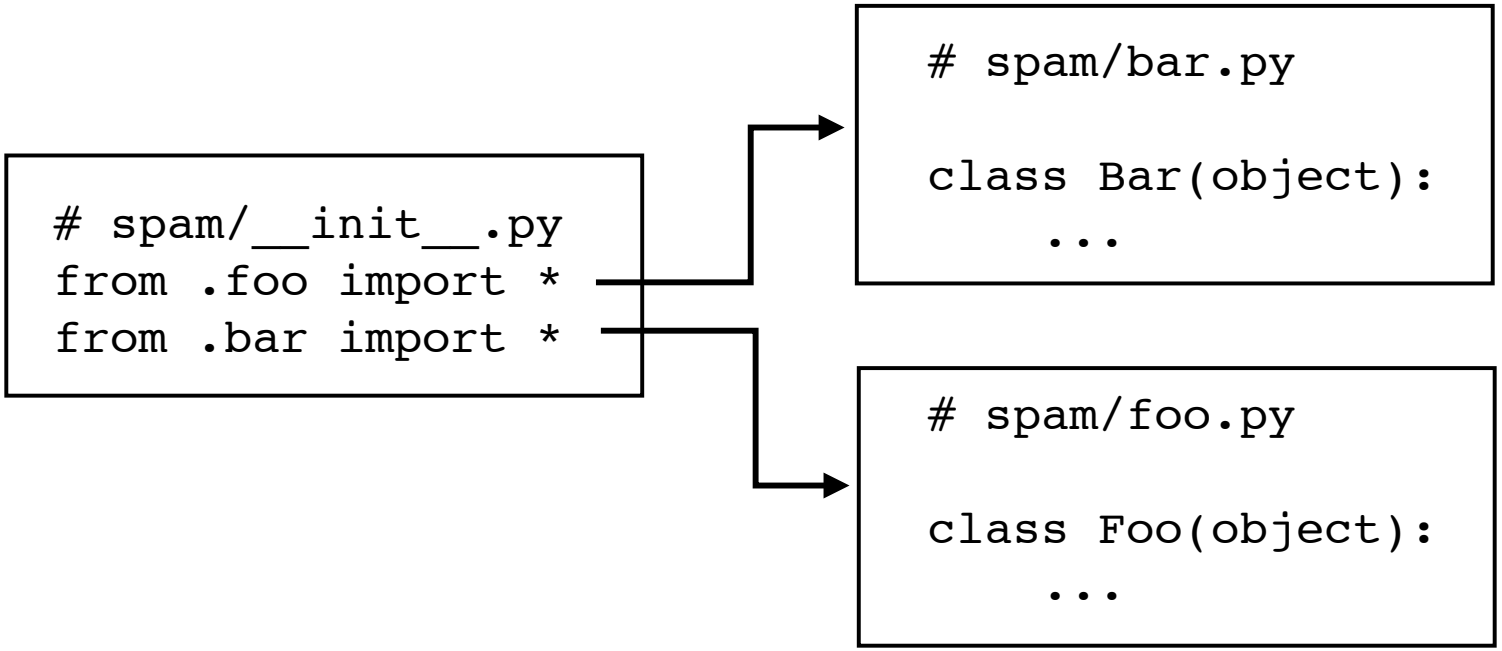
- You can make custom module objects

```
import types

class MyModule(types.ModuleType):
    ...
```

- Why would you do that?
- Injection of "special magic!"

Module Assembly (Reprise)

- Consider: A package that stitches things together



- It imports everything (might be slow)

Thought

- What if subcomponents only load on demand?

```
>>> import spam
>>> f = spam.Foo()
Loaded Foo
>>> f
<spam.foo.Foo object at 0x100656f60>
>>>
```

- No extra imports needed
- Autoload happens behind the scenes

Lazy Module Assembly

- Alternative approach

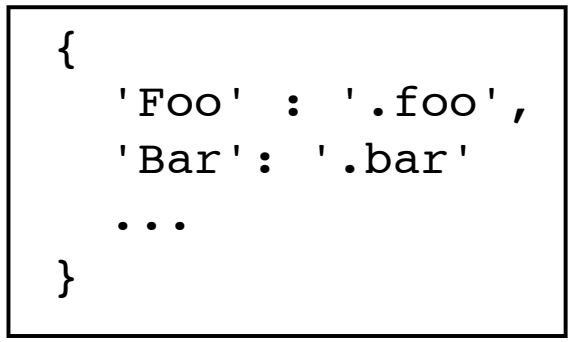
```
# spam/__init__.py

# List the exported symbols by module
_submodule_exports = {
    '.foo' : ['Foo'],
    '.bar' : ['Bar']
}

# Make a {name: modname } mapping
_submodule_by_name = {
     name: modulename
           for modulename in _submodule_exports
           for name in _submodule_exports[modulename] }
```

- This is not actually importing anything...

Lazy Module Assembly

- Alternative approach

```
# spam/__init__.py

# List the exported symbols by module
_submodule_exports = {
    '.foo' : ['Foo'],
    '.bar' : ['Bar']
}

# Make a {name: modname } mapping
_submodule_by_name = {
     name: modulename
           for modulename in _submodule_exports
           for name in _submodule_exports[modulename] }
```



- It builds symbol-module name map

Lazy Module Assembly

```
# spam/__init__.py
...
import types, sys, importlib

class OnDemandModule(types.ModuleType):
    def __getattr__(self, name):
        modulename = _submodule_by_name.get(name)
        if modulename:
            module = importlib.import_module(modulename,
                                             __package__)
            print('Loaded', name)
            value = getattr(module, name)
            setattr(self, name, value)
            return value
        raise AttributeError('No attribute %s' % name)

newmodule = OnDemandModule(__name__)
newmodule.__dict__.update(globals())
newmodule.__all__ = list(_submodule_by_name)
sys.modules[__name__] = newmodule
```

Load symbols on access.

Creates a replacement "module" and inserts it into sys.modules

Example

```
>>> import spam
>>> f = spam.Foo()
Loaded Foo
>>> f
<spam.foo.Foo object at 0x100656f60>
>>> from spam import Bar
Loaded Bar
>>> Bar
<class 'spam.bar.Bar'>
>>>
```

- That's crazy!
- Not my idea: Armin Ronacher
- Werkzeug (http://werkzeug.pocoo.org)

Part 7

The Module Reloaded

Module Reloading

- An existing module can be reloaded

```
>>> import spam
>>> from importlib import reload
>>> reload(spam)
<module 'spam' from 'spam.py'>
>>>
```

- As previously noted: zombies are spawned
- Why?

Reload Undercover

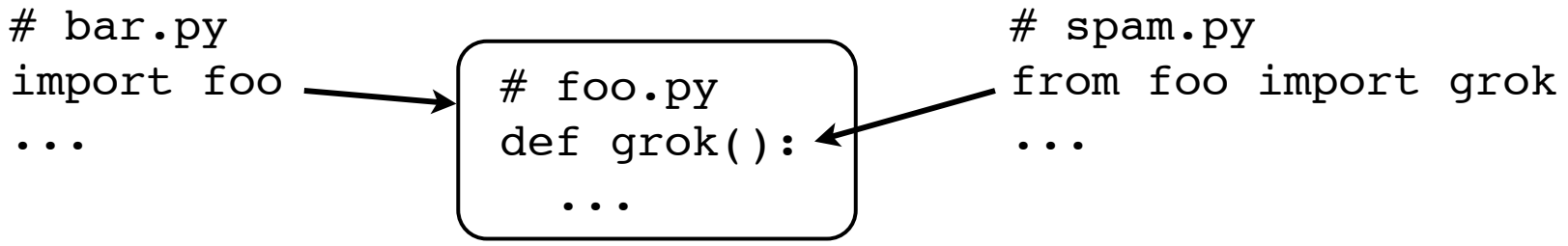
- Reloading in a nutshell

```
>>> import spam
>>> code = open(spam.__file__, 'rb').read()
>>> exec(code, spam.__dict__)
>>>
```

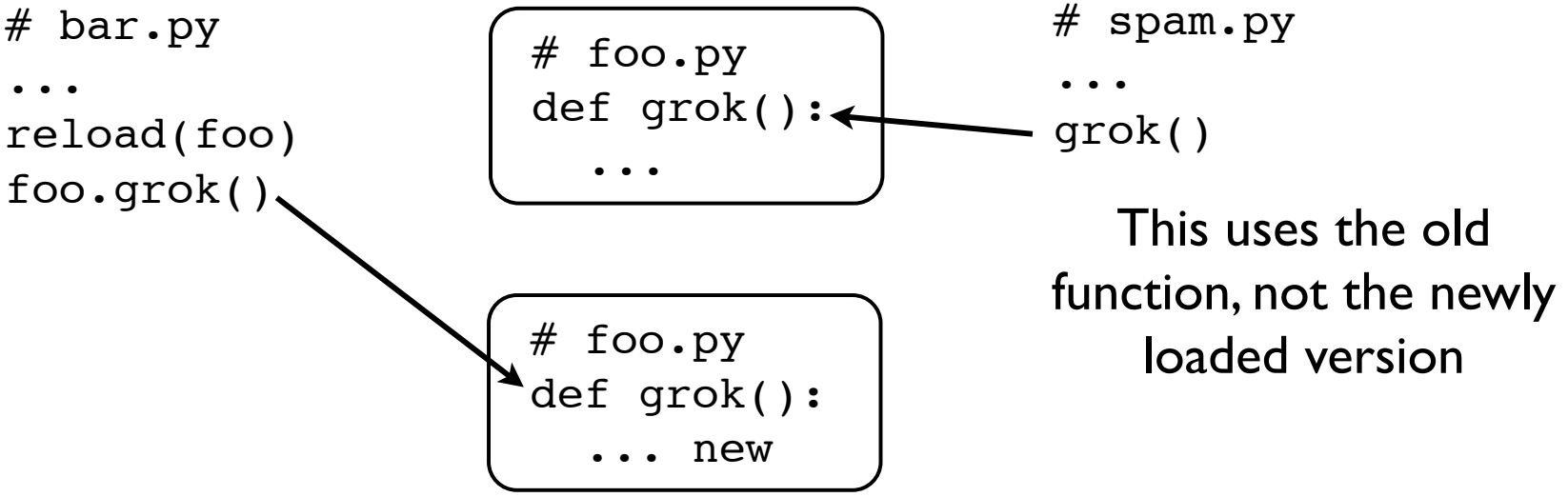
- It simply re-executes the source code in the already existing module dictionary
- It doesn't even bother to clean up the dict
- So, what can go wrong?

Module Reloading Danger

- Consider



- Effect of reloading



This uses the old function, not the newly loaded version

Reloading and Packages

- Suppose you have a package

```
# spam/__init__.py

print('Loading spam')
from . import foo
from . import bar
```

- What happens to the submodules on reload?

```
>>> import spam
Loading spam
>>> importlib.reload(spam)
Loading spam
<module 'spam' from 'spam/__init__.py'>
>>>
```

- Nothing happens: They aren't reloaded

Reloading and Instances

- Suppose you have a class

```
# spam.py

class Spam(object):
    def yow(self):
        print('Yow!')
```

```
import spam

a = spam.Spam()
```

- Now, you change it and reload

```
# spam.py

class Spam(object):
    def yow(self):
        print('Moar Yow!')
```

```
reload(spam)
b = spam.Spam()

a.yow()       # ????
b.yow()       # ????
```

Reloading and Instances

- Suppose you have a class

```
# spam.py

class Spam(object):
    def yow(self):
        print('Yow!')
```

```
import spam

a = spam.Spam()
```

- Now, you change it and reload

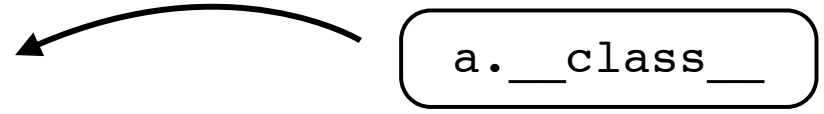
```
# spam.py

class Spam(object):
    def yow(self):
        print('Moar Yow!')
```

```
reload(spam)
b = spam.Spam()

a.yow()       # Yow!
b.yow()       # Moar Yow!
```

Reloading and Instances

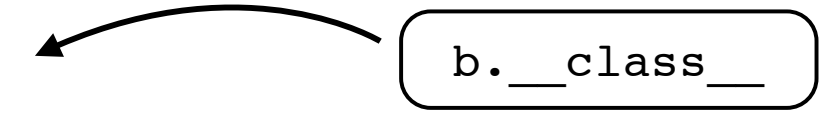
- Existing instances keep their original class

```
class Spam(object):
    def yow(self):
        print('Yow!')
```



- New instances will use the new class

```
class Spam(object):
    def yow(self):
        print('Moar Yow!')
```




```
>>> a.yow()
Yow!
>>> b.yow()
Moar Yow!
>>> type(a) == type(b)
False
>>>
```

Reload Woes

- You might have multiple implementations of the code actively in use at the same time
- Maybe it doesn't matter
- Maybe it causes your head to explode
- No, spawned zombies eat your brain

Detecting Reload

- Modules can detect/prevent reloading

```
# spam.py

if 'foo' in globals():
    raise ImportError('reload not allowed')

def foo():
    ...
```

- Idea: Look for names already defined in globals()
- Recall: module dict is not cleared on reload

Reloadable Packages

- Packages could reload their subcomponents

```
# spam/__init__.py

if 'foo' in globals():
    from importlib import reload
    foo = reload(foo)
    bar = reload(bar)
else:
    from . import foo
    from . import bar
```

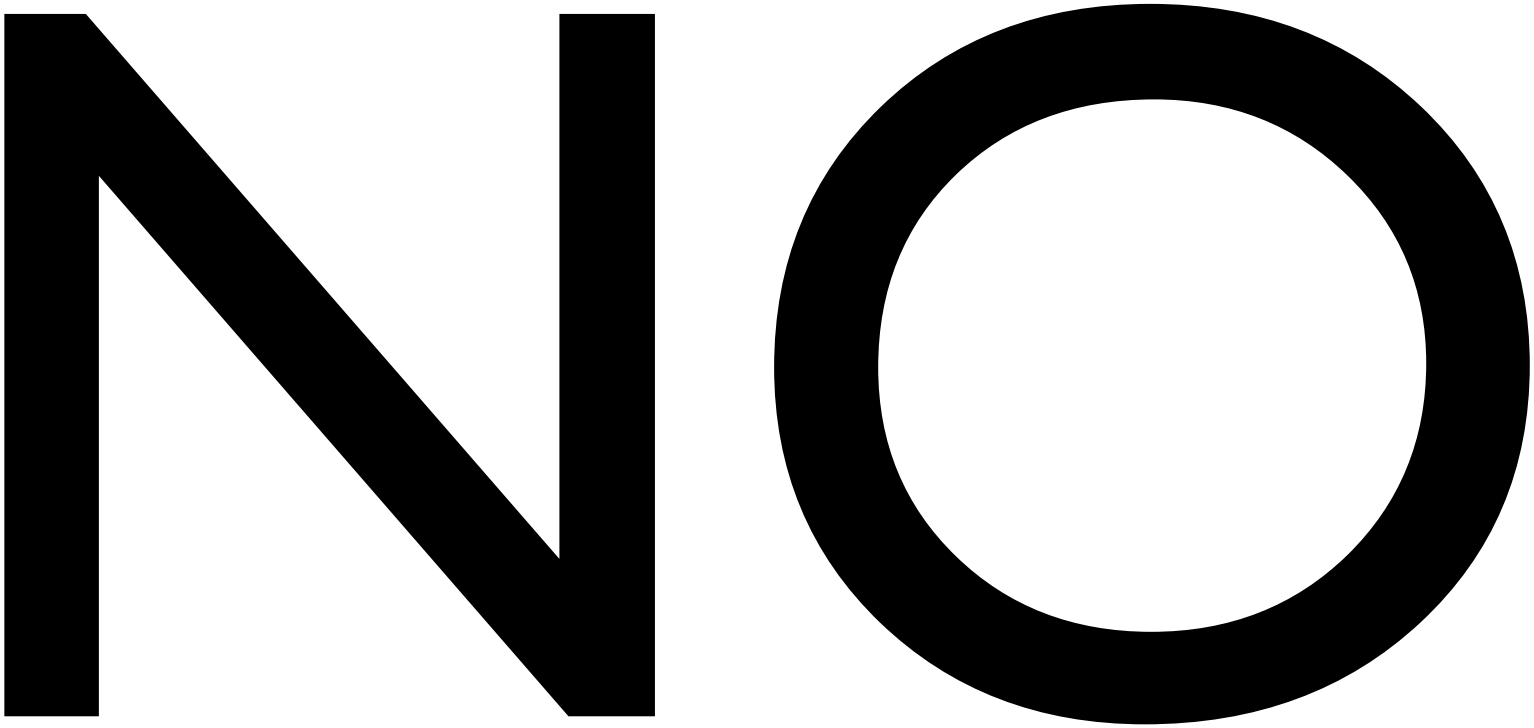
- Ugh. No. Please don't.

"Fixing" Reloaded Instances

- You might try to make it work with hacks

```
import weakref
class Spam(object):
    if 'Spam' in globals():
       _instances = Spam._instances
    else:
       _instances = weakref.WeakSet()

    def __init__(self):
        Spam._instances.add(self)

    def yow(self):
        print('Yow!')

for instance in Spam._instances:
    instance.__class__ = Spam
```

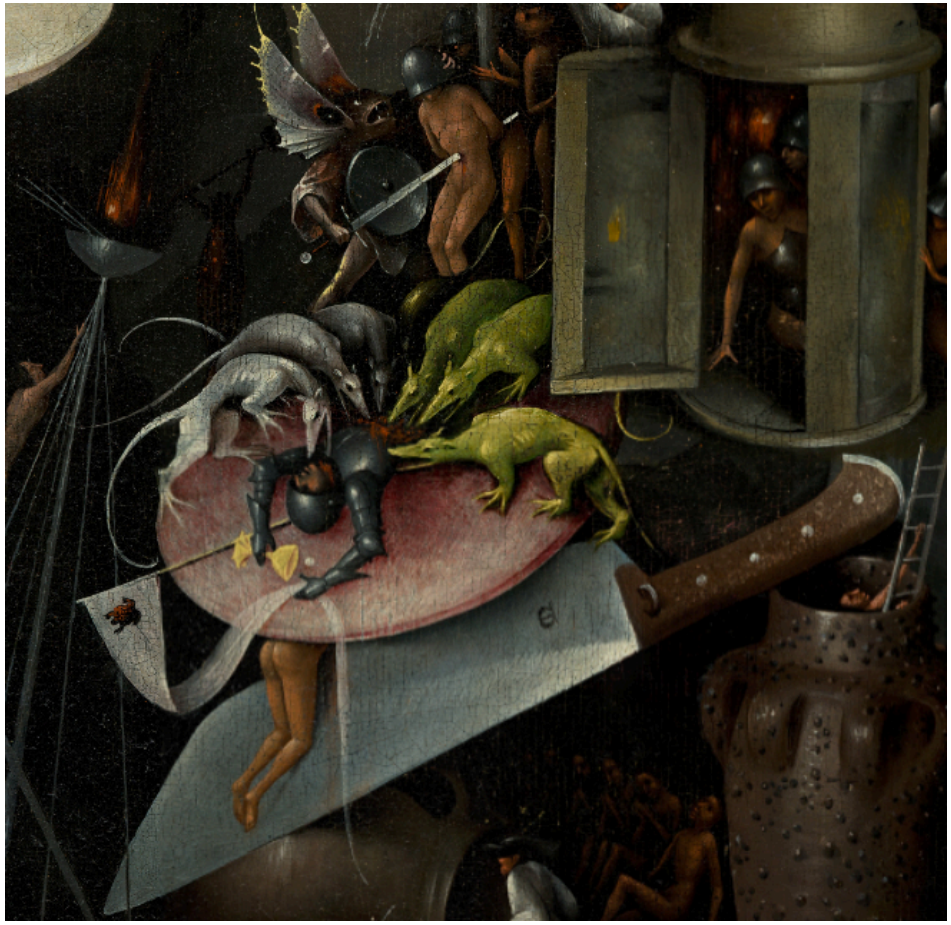
- Will make "code review" more stimulating



Reload/Restarting

- Only safe/sane way to reload is to restart
- Your time is probably better spent trying to devise a sane shutdown/restart process to bring in code changes
- Possibly managed by some kind of supervisor process or other mechanism

Part 8



Import Hooks

WARNING

- What follows has been an actively changing part of Python
- It assumes Python 3.5 or newer
- It might be changed again
- Primary goal: Peek behind the covers a little bit

sys.path Revisited

- sys.path is the most visible configuration of the module/package system to users

```
>>> import sys
>>> sys.path
['',
 '/usr/local/lib/python35.zip',
 '/usr/local/lib/python3.5',
 '/usr/local/lib/python3.5/plat-darwin',
 '/usr/local/lib/python3.5/lib-dynload',
 '/usr/local/lib/python3.5/site-packages']
```

- It is not the complete picture
- In fact, it is a small part of the bigger picture

sys.meta_path

- import is actually controlled by sys.meta_path

```
>>> import sys
>>> sys.meta_path
[<class '_frozen_importlib.BuiltinImporter'>,
 <class '_frozen_importlib.FrozenImporter'>,
 <class '_frozen_importlib.PathFinder'>]
>>>
```

- It's a list of "importers"
- When you import, they are consulted in order

Module Finding

- Importers are consulted for a "ModuleSpec"

```
# importlib.util
def find_spec(modname):
    for imp in sys.meta_path:
        spec = imp.find_spec(modname)
        if spec:
            return spec
    return None
```

- Note: Use importlib.util.find_spec(*modname*)
- Example: Built-in module

```
>>> from importlib.util import find_spec
>>> find_spec('sys')
ModuleSpec(name='sys',
           loader=<class '_frozen_importlib.BuiltinImporter'>,
           origin='built-in')
>>>
```

Module Finding

- Example: Python Source

```
>>> find_spec('socket')
ModuleSpec(name='socket',
           loader=<_frozen_importlib.SourceFileLoader
object at 0x10066e7b8>,
           origin='/usr/local/lib/python3.5/socket.py')
>>>
```

- Example: C Extension

```
>>> find_spec('math')
ModuleSpec(name='math',
           loader=<_frozen_importlib.ExtensionFileLoader
object at 0x10066e7f0>,
           origin='/usr/local/lib/python3.5/lib-dynload/
math.so')
>>>
```

ModuleSpec

- ModuleSpec merely has information about the module location and loading info

```
spec.name                          # Full module name
spec.parent                        # Enclosing package
spec.submodule_search_locations    # Package __path__
spec.has_location                  # Has external location
spec.origin                        # Source file location
spec.cached                        # Cached location
spec.loader                        # Loader object
```

- Example:

```
>>> spec = find_spec('socket')
>>> spec.name
'socket'
>>> spec.origin
'/usr/local/lib/python3.5/socket.py'
>>> spec.cached
'/usr/local/lib/python3.5/__pycache__/socket.cpython-35.pyc'
>>>
```

Using ModuleSpecs

- A module spec can be useful all by itself
- Consider: (Inspired by Armin Ronacher [1])

```
# spam.py
try:
    import foo
except ImportError:
    import simplefoo as foo
```



```
# foo.py
import bar  # Not found
```

Scenario: Code that tests to see if a module can be imported. If not, it falls back to an alternative.

[1] http://lucumr.pocoo.org/2011/9/21/python-import-blackbox/

Using ModuleSpecs

- It's a bit flaky--no error is reported

```
>>> import spam
>>> import spam.foo
<module 'simplefoo' from 'simplefoo.py'>

>>> import os.path
>>> import os.path.exists('foo.py')
True
>>>
```

- User is completely perplexed--the file exists
- Why won't it import?!?!? Much cursing ensues...

Using ModuleSpecs

- A module spec can be useful all by itself
- A Reformulation

```
# spam.py
from importlib.util import find_spec

if find_spec('foo'):
    import foo
else:
    import simplefoo
```

- If the module can be found, it will import
- A "look before you leap" for imports

Using ModuleSpecs

- Example:

```
>>> import spam
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File ".../spam.py", line 3, in <module>
    import foo
  File ".../foo.py", line 1, in <module>
    import bar
ImportError: No module named 'bar'
>>>
```

- It's a much better error
- Directly points at the problem

Module Loaders

- A separate "loader" object lets you do more

```
>>> spec = find_spec('socket')
>>> spec.loader
<_frozen_importlib.SourceFileLoader object at 0x1007706a0>
>>>
```

- Example: Pull the source code

```
>>> src = spec.loader.get_source(spec.name)
>>>
```

- More importantly: loaders actually create the imported module

Module Creation

- Example of creation

```
module = spec.loader.create_module(spec)
if not module:
    module = types.ModuleType(spec.name)
    module.__file__ = spec.origin
    module.__loader__ = spec.loader
    module.__package__ = spec.parent
    module.__path__ = spec.submodule_search_locations
    module.__spec__ = spec
```

- But don't do that... it's already in the library (py3.5)

```
# Create the module
from importlib.util import module_from_spec
module = module_from_spec(spec)
```

Some Ugly News

- Module creation currently has a split personality
- Legacy Interface: Python 3.3 and earlier

```
module = loader.load_module()
```

- Modern Interface: Python 3.4 and newer

```
module = loader.create_module(spec)
if not module:
    # You're on your own. Make a module object
    # however you want
    ...
sys.modules[spec.name] = module
loader.exec_module(module)
```

- Legacy interface still used for all non-Python modules (builtins, C extensions, etc.)

Module Execution

- Again, creating a module doesn't load it

```
>>> spec = find_spec('socket')
>>> socket = module_from_spec(spec)
>>> socket
<module 'socket' from '/usr/local/lib/python3.5/
socket.py'>
>>> dir(socket)
['__cached__', '__doc__', '__file__', '__loader__',
'__name__', '__package__', '__spec__']
>>>
```

- To populate, the module must be executed

```
>>> sys.modules[spec.name] = socket
>>> spec.loader.exec_module(socket)
>>> dir(socket)
['AF_APPLETALK', 'AF_DECnet', 'AF_INET', 'AF_INET6', ...
```

Commentary

- Modern module loading technique is better
- Decouples module creation/execution
- Allows for more powerful programming techniques involving modules
- Far fewer "hacks"
- Let's see an example

Lazy Imports

- Idea: create a module that doesn't execute itself until it is actually used for the first time

```
>>> # Import the module
>>> socket = lazy_import('socket')
>>> socket
<module 'socket' from '/usr/local/lib/python3.5/socket.py'>
>>> dir(socket)
['__doc__', '__loader__', '__name__', '__package__', '__spec__
```

- Now, access it

```
>>> socket.AF_INET
<AddressFamily.AF_INET: 2>
>>> dir(socket)
['AF_APPLETALK', 'AF_DECnet', 'AF_INET', 'AF_INET6', ... ]
>>>
```

Lazy Imports

- A module that only executes when it gets accessed

```
import types

class _Module(types.ModuleType):
    pass

class _LazyModule(_Module):
    def __init__(self, spec):
        super().__init__(spec.name)
        self.__file__ = spec.origin
        self.__package__ = spec.parent
        self.__loader__ = spec.loader
        self.__path__ = spec.submodule_search_locations
        self.__spec__ = spec

    def __getattr__(self, name):
        self.__class__ = _Module
        self.__spec__.loader.exec_module(self)
        assert sys.modules[self.__name__] == self
        return getattr(self, name)
```

Idea: execute module on first access

Lazy Imports

- A utility function to make the "import"

```
import importlib.util, sys

def lazy_import(name):
    # If already loaded, return the module
    if name in sys.modules:
        return sys.modules[name]

    # Not loaded. Find the spec
    spec = importlib.util.find_spec(name)
    if not spec:
       raise ImportError('No module %r' % name)

    # Check for compatibility
    if not hasattr(spec.loader, 'exec_module'):
       raise ImportError('Not supported')

    module = sys.modules[name] = _LazyModule(spec)
    return module
```

Lazy Imports

```
>>> # Import the module
>>> socket = lazy_import('socket')
>>> socket
<module 'socket' from '/usr/local/lib/python3.5/socket.py'>
>>> dir(socket)
['__doc__', '__loader__', '__name__', '__package__', '__spec__']

>>> # Use the module (notice how it autoloads)
>>> socket.AF_INET
<AddressFamily.AF_INET: 2>
>>> dir(socket)
['AF_APPLETALK', 'AF_DECnet', 'AF_INET', 'AF_INET6', ... ]
>>>
```

- Behold the magic!

That's Crazy!

- Actually a somewhat old (and new) idea
- Goal is to reduce startup time
- Python 2 implementation (Phillip Eby)
  - https://pypi.python.org/pypi/Importing
  - Significantly more "hacky" (involves reload)
- There's a LazyLoader coming in Python 3.5

Importers Revisited

- As noted: Python tries to find a "spec"

```
# importlib.util
def find_spec(modname):
    for imp in sys.meta_path:
        spec = imp.find_spec(modname)
        if spec:
            return spec
    return None
```

- You can also plug into this machinery to do interesting things as well

Watching Imports

- An Importer than merely watches things

```
import sys

class Watcher(object):
    @classmethod
    def find_spec(cls, name, path, target=None):
        print('Importing', name, path, target)
        return None

sys.meta_path.insert(0, Watcher)
```

- Does nothing: simply logs all imports

Watching Imports

- Example:

```
>>> import math
Importing math None None

>>> import json
Importing json None None
Importing json.decoder ['/usr/local/lib/python3.5/json'] None
Importing json.scanner ['/usr/local/lib/python3.5/json'] None
Importing _json None None
Importing json.encoder ['/usr/local/lib/python3.5/json'] None

>>> importlib.reload(math)
Importing math None <module 'math' from '/usr/local/lib/
python3.5/lib-dynload/math.so'>
<module 'math' from '/usr/local/lib/python3.5/lib-dynload/
math.so'>
>>>
```

AutoInstaller

- Idle thought: Wouldn't it be cool if unresolved imports would just automatically download from PyPI?

```
>>> import requests
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'requests'
>>> import autoinstall
>>> import requests
Installing requests
>>> requests
<module 'requests' from '...python3.4/site-packages/requests/
__init__.py'>
>>>
```

- Disclaimer: This is a HORRIBLE idea

AutoInstaller

```
import sys
import subprocess
import importlib.util

class AutoInstall(object):
    _loaded = set()
    @classmethod
    def find_spec(cls, name, path, target=None):
        if path is None and name not in  cls._loaded:
            cls._loaded.add(name)
            print("Installing",  name)
            try:
                out = subprocess.check_output(
                  [sys.executable, '-m', 'pip', 'install', name])
                return importlib.util.find_spec(name)
            except Exception as  e:
                print("Failed")
         return None

sys.meta_path.append(AutoInstall)
```

AutoInstaller

- Example:

```
>>> import requests
Installing requests
Installing winreg
Failed
Installing ndg
Failed
Installing _lzma
Failed
Installing certifi
Installing simplejson
>>> r = requests.get('http://www.python.org')
>>>
```

- Oh, good god. NO! NO! NO! NO! NO!

"Webscale" Imports

- Thought: Could modules be imported from Redis?
- Redis in a nutshell: a key/value server

```
>>> import redis
>>> r = redis.Redis()
>>> r.set('bar', 'hello')
True
>>> r.get('bar')
b'hello'
>>>
```

- Challenge: load code from it?

Redis Example

- Setup (upload some code)

```
>>> import redis
>>> r = redis.Redis()
>>> r.set('foo.py', b'print("Hello World")\n')
True
>>>
```

- Try importing some code

```
>>> import redisloader
>>> redisloader.enable()
>>> import foo
Hello World
>>> foo
<module 'foo' (<redisloader.RedisLoader object at 0x1012dca90>)>
>>>
```

Redis Importer

```
# redisloader.py
import redis
import importlib.util

class RedisImporter(object):
    def __init__(self, *args, **kwargs):
        self.conn = redis.Redis(*args, **kwargs)
        self.conn.exists('test')

    def find_spec(self, name, path, target=None):
        origin = name + '.py'
        if self.conn.exists(origin):
            loader = RedisLoader(origin, self.conn)
            return importlib.util.spec_from_loader(name, loader)
        return None

def enable(*args, **kwargs):
    import sys
    sys.meta_path.insert(0, RedisImporter(*args, **kwargs))
```

Redis Loader

```
# redisloader.py
...
class RedisLoader(object):
    def __init__(self, origin, conn):
        self.origin = origin
        self.conn = conn

    def create_module(self, spec):
        return None

    def exec_module(self, module):
        code = self.conn.get(self.origin)
        exec(code, module.__dict__)
```

Part 9

Path Hooks

sys.path Revisited

- Yes, yes, sys.path.

```
>>> import sys
>>> sys.path
['',
 '/usr/local/lib/python35.zip',
 '/usr/local/lib/python3.5',
 '/usr/local/lib/python3.5/plat-darwin',
 '/usr/local/lib/python3.5/lib-dynload',
 '/usr/local/lib/python3.5/site-packages']
```

- There is yet another piece of the puzzle

sys.path_hooks

- Each entry on sys.path is tested against a list of "path hook" functions

```
>>> import sys
>>> sys.path_hooks
[
  <class 'zipimport.zipimporter'>,
  <function FileFinder..path_hook_for_FileFinder at 0x1003afbf
]
>>>
```

- Functions merely decide whether or not they can handle a particular path

sys.path_hooks

- Example:

```
>>> path = '/usr/local/lib/python3.5'
>>> finder = sys.path_hooks[0](path)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
zipimport.ZipImportError: not a Zip file

>>> finder = sys.path_hooks[1](path)
>>> finder
FileFinder('/usr/local/lib/python3.5')
>>>
```

- Idea: Python uses the path_hooks to associate a module finder with each path entry

Path Finders

- Path finders are used to locate modules

```
>>> finder
FileFinder('/usr/local/lib/python3.5')

>>> finder.find_spec('datetime')
ModuleSpec(name='datetime',
loader=<_frozen_importlib.SourceFileLoader object at
0x10068b7f0>, origin='/usr/local/lib/python3.5/datetime.py')
>>>
```

- Uses the same machinery as before (ModuleSpec)

sys.path_importer_cache

- The path finders get cached

```
>>> sys.path_importer_cache
{
  ...
 '/usr/local/lib/python3.5':
       FileFinder('/usr/local/lib/python3.5'),
 '/usr/local/lib/python3.5/lib-dynload':
       FileFinder('/usr/local/lib/python3.5/lib-dynload'),
 '/usr/local/lib/python3.5/plat-darwin':
       FileFinder('/usr/local/lib/python3.5/plat-darwin'),
 '/usr/local/lib/python3.5/site-packages':
       FileFinder('/usr/local/lib/python3.5/site-packages'),
  ...
>>>
```

sys.path Processing

- What happens during import (roughly)

```
modname = 'somemodulename'
for entry in sys.path:
    finder = sys.path_importer_cache[entry]
    if finder:
        spec = finder.find_spec(modname)
        if spec:
            break
else:
    raise ImportError('No such module')

...
# Load module from the spec
...
```

Customized Paths

- Naturally, you can hook into the sys.path machinery with your own custom code
- Requires three components
  - A path hook
  - A finder
  - A loader
- Example follows

Importing from URLs

- Example: Consider some Python code

```
spam/
    foo.py
    bar.py
```

- Make it available via a web server

```
bash % cd spam
bash % python3 -m http.server
Serving HTTP on 0.0.0.0 port 8000 ...
```

- Allow imports via sys.path

```
import sys
sys.path.append('http://someserver:8000')
import foo
```

A Path Hook

- Step 1: Write a hook to recognize URL paths

```
import re, urllib.request
def url_hook(name):
    if not name.startswith(('http:', 'https:')):
        raise ImportError()
    data = urllib.request.urlopen(name).read().decode('utf-8')
    filenames = re.findall('[a-zA-Z_][a-zA-Z0-9_]*\.py', data)
    modnames = { name[:-3] for name in filenames }
    return UrlFinder(name, modnames)

import sys
sys.path_hooks.append(url_hook)
```

- This makes an initial URL request, collects the names of all .py files it can find, creates a finder.

A Path Finder

- Step 2: Write a finder to check for modules

```
import importlib.util

class UrlFinder(object):
    def __init__(self, baseuri, modnames):
        self.baseuri = baseuri
        self.modnames = modnames

    def find_spec(self, modname, target=None):
        if modname in self.modnames:
            origin = self.baseuri + '/' + modname + '.py'
            loader = UrlLoader()
            return importlib.util.spec_from_loader(modname,
                                        loader, origin=origin)
        else:
            return None
```

A Path Loader

- Step 3: Write a loader

```
class UrlLoader(object):
    def create_module(self, target):
        return None

    def exec_module(self, module):
        u = urllib.request.urlopen(module.__spec__.origin)
        code = u.read()
        compile(code, module.__spec__.origin, 'exec')
        exec(code, module.__dict__)
```

- And, you're done

Example

- Example use:

```
>>> import sys
>>> sys.path.append('http://localhost:8000')
>>> import foo
>>> foo
<module 'foo' (http://localhost:8000/foo.py)>
>>>
```

- Bottom line: You can make custom paths
- Not shown: Making this work with packages

Part 10

Final Comments

Thoughts

- There are a lot of moving parts
- A good policy: Keep it as simple as possible
- It's good to understand what's possible
- In case you have to debug it

References

- https://docs.python.org/3/reference/import.html
- https://docs.python.org/3/library/importlib
- Relevant PEPs

  PEP 273 - Import modules from zip archives
  PEP 302 - New import hooks
  PEP 338 - Executing modules as scripts
  PEP 366 - Main module explicit relative imports
  PEP 405 - Python virtual environments
  PEP 420 - Namespace packages
  PEP 441 - Improving Python ZIP application support
  PEP 451 - A ModuleSpec type for the import system

A Few Related Talks

- "How Import Works" - Brett Cannon (PyCon'13)

  http://www.pyvideo.org/video/1707/how-import-works

- "Import this, that, and the other thing", B. Cannon

  http://pyvideo.org/video/341/pycon-2010--import-this--that--and-the-other-thin

Thanks!

- I hope you got some new ideas
- Please feel free to contact me

@dabeaz (Twitter)
http://www.dabeaz.com

- Also, I teach Python classes

http://www.dabeaz.com/chicago

- Special Thanks:

A. Chourasia, Y. Tymciurak, P. Smith, E. Meschke, E. Zimmerman, JP Bader