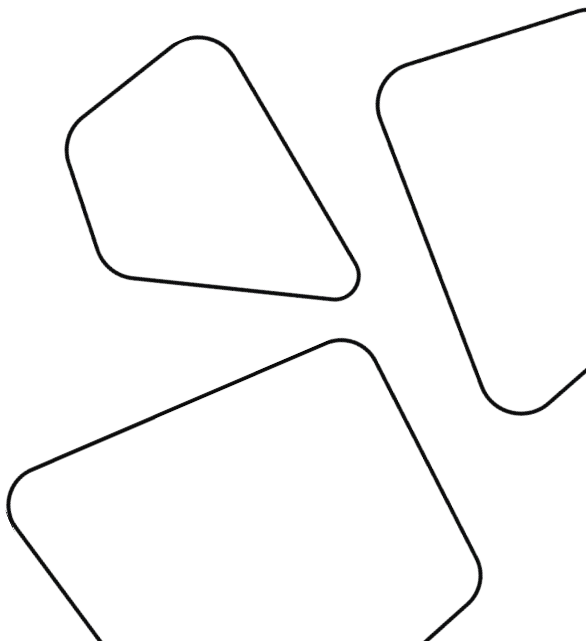
Software Development with Python

Lecture 1. Modules, Packages and Import system

Solovev Aleksandr,

March 2021



Topics

1. Modules
2. Packages
3. Resource imports
4. Dynamic imports
5. Import system

Modules

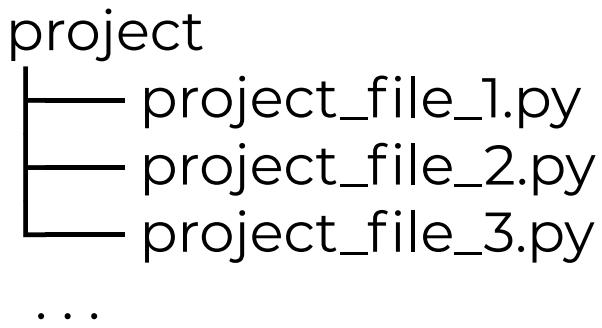
Modules

An object that serves as an organizational unit of Python code.

Modules have a namespace containing arbitrary Python objects.

Modules are loaded into Python by the process of importing.

Modules



Modules

```python
# project_file_1.py
import itertools

def project_file_func():
    print("Simple project file func")

SIMPLE_CONSTANT = 10
```

Modules

What is a module in terms of code?

Modules

```python
# project_file_2.py
import project_file_1

print(project_file_1)
# <module 'project_file_1' from
'/Users/nightingale/project/project_file_1.py'>

print(type(project_file_1))
# <class 'module'>
```

Modules

In this context each file in *project directory* is a module

We can import it and reuse it's code

Modules

```python
# project_file_2.py
import project_file_1


project_file_1.project_file_func()
# Simple project file func
print(project_file_1.SIMPLE_CONSTANT)
# 10
```

Modules

```
# project_file_3.py
import project_file_2

# Simple project file func
# 10
```

Modules

```
# project_file_3.py

import project_file_2


print(dir(project_file_2))

# ['__builtins__', '__cached__', '__doc__',
'__file__', '__loader__', '__name__',
'__package__', '__spec__', 'project_file_1']
```

Modules

```
# project_file_3.py
...
from project_file_2 import project_file_1
print(dir(project_file_1))
# ['SIMPLE_CONSTANT', '__builtins__', '__cached__',
'__doc__', '__file__', '__loader__', '__name__',
'__package__', '__spec__', 'itertools',
'project_file_func']
print(type(project_file_1))
# <class 'module'>
```

Modules

```python
# project_file_4.py
import math as m

print(m)
# <module 'math' from
'/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.
7/lib-dynload/math.cpython-37m-darwin.so'>

print(type(m))
# <class 'module'>

print(dir())
# ['__annotations__', '__builtins__', ..., 'm']
```

Modules

It's almost the same as this but ***without math in dir***

```
import math

m = math

print(dir())
# ['__annotations__', '__builtins__',
'__cached__', '__doc__', '__file__',
'__loader__', '__name__', '__package__',
'__spec__', 'm', 'math']
```

Modules

```python
# project_file_5.py
from sys import (
   getrecursionlimit as get_rec_lim,
   setrecursionlimit as set_rec_lim,
)

get_rec_lim()
set_rec_lim(10000)
```

Modules

When the interpreter executes the above import statement, it searches for ***project_file_n.py*** in a list of directories assembled from the following sources:

- The directory from which the input script was run or the current directory if the interpreter is being run interactively
- The list of directories contained in the PYTHONPATH environment variable, if it is set
- An installation-dependent list of directories configured at the time Python is installed

Modules

```
# project_file_2.py
import project_file_1
...

import sys

print(sys.path)

# [
'/Users/nightingale/project',
'/Users/nightingale/project',      …,
'/Users/nightingale/Library/Python/3.7/lib/python/site-packages',
'/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/site-packages']
```

Modules

```python
# project_file_6.py
PROJECT_VAR = 1

# project_file_7.py
PROJECT_VAR = 2

# project_file_8.py
from project_file_6 import *

print(dir())
# ['PROJECT_VAR', '__annotations__',
'__builtins__', '__cached__', '__doc__',
'__file__', '__loader__', '__name__',
'__package__', '__spec__']
```

Modules

```python
# project_file_8.py
from project_file_7 import *
from project_file_6 import *

print(PROJECT_VAR)
# 1
```

```python
# project_file_8.py
from project_file_6 import *
from project_file_7 import *

print(PROJECT_VAR)
# 2
```

Modules

```
# project_file_9.py
__all__ = ['function']


INTERNAL_CONSTANT = 42


def _function():
    return INTERNAL_CONSTANT


def function():
    return _function()
```

Modules

```
# project_file_9.py
__all__ = ['function']


INTERNAL_CONSTANT = 42


def _function():
    return INTERNAL_CONSTANT


def function():
    return _function()
```

Modules

```
# project_file_10.py
from project_file_9 import *

print(dir())
# [
# '__annotations__', '__builtins__',
'__cached__',
# '__doc__', '__file__', '__loader__',
'__name__',
# '__package__', '__spec__', 'function']
```

Modules

```
# project_file_10.py
...
from project_file_9 import _function

print(_function())
# 42
```

Modules

```python
# script_module.py
def sum(a, b):
    return a + b

print(f'Running demo script for
script_module.py: {sum(1, 2)}')


# project_file_10.py
...
import script_module
# Running demo script for script_module.py: 3
```

Modules

```python
# script_module.py
def sum(a, b):
    return a + b

print(f'Running demo script for
script_module.py: {sum(1, 2)}')


# project_file_10.py
...
import script_module
# Running demo script for script_module.py: 3
```

Modules

```python
# script_module.py
def sum(a, b):
    return a + b


if __name__ == "__main__":
    print(
        f'Running demo script for script_module.py:
          {sum(1, 2)}'
    )

# project_file_10.py
...
import script_module
```

Modules

```
~/project » python3 script_module.py
Running demo script for script_module.py: 3
```

Modules Cache

To speed up loading modules, Python caches the compiled version of each module in the `__pycache__`directory under the name module.*version*.pyc, where the version encodes the format of the compiled file; it generally contains the Python version number. For example, in CPython release 3.3 the compiled version of spam.py would be cached as `__pycache__/spam.cpython-33.pyc`. This naming convention allows compiled modules from different releases and different versions of Python to coexist.

Package

Package

A Python module which can contain submodules or recursively, subpackages. Technically, a package is a Python module with an `__path__` attribute.

See also regular package and namespace package.

Package

In practice, a package typically corresponds to a file directory containing Python files and other directories.

To create a Python package yourself, you create a directory and a file named __init__.py inside it.

The *__init__.py* file contains the contents of the package when it's treated as a module. It can be left empty.

__init__.py file can be a facade for your own package.

Package

It’s important to keep in mind that all packages are modules, but not all modules are packages. Or put another way, packages are just a special kind of module.

Specifically, any module that contains a `__path__` attribute is considered a package.

Package

All modules have a name.

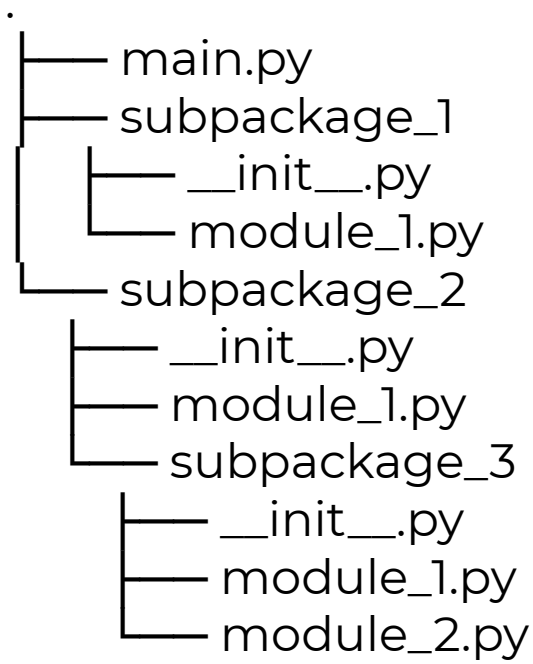
Subpackage names are separated from their parent package name by a dot, akin to Python's standard attribute access syntax.

Thus you might have a module called sys and a package called email, which in turn has a subpackage called email.mime and a module within that subpackage called email.mime.text

Package

```
sound/                          Top-level package
      __init__.py               Initialize the sound package
      formats/                  Subpackage for file format conversions
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      effects/                  Subpackage for sound effects
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      filters/                  Subpackage for filters
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ...
```

Package


.
main.py
subpackage_1
__init__.py
module_1.py
subpackage_2
__init__.py
module_1.py
subpackage_3
__init__.py
module_1.py
module_2.py

Package

```
# subpackage_1/__init__.py
from .module_1 import module_func as
pckg_1_module_func

__all__ = ['pckg_1_module_func']

# subpackage_1/module_1.py
import os

PATH = os.path.dirname(os.path.abspath(__file__))


def module_func():
    return PATH
```

Package

```python
# subpackage_3/__init__.py
from .module_1 import module_func as pckg_3_module_func
from .module_2 import module_2_func as pckg_3_module_2_func

__all__ = ['pckg_3_module_func', 'pckg_3_module_2_func']

# subpackage_3/module_1.py
import os

PATH = os.path.dirname(os.path.abspath(__file__))

def module_func():
    return PATH

# subpackage_3/module_2.py
import os

PATH = os.path.dirname(os.path.abspath(__file__))

def module_2_func():
    return PATH
```

Package

```python
# subpackage_2/__init__.py
from . import subpackage_3
from .module_1 import module_func as
pckg_2_module_func
from .subpackage_3 import *

__all__ = ['pckg_2_module_func'] +
subpackage_3.__all__


# subpackage_2/module_1.py
import os

PATH = os.path.dirname(os.path.abspath(__file__))


def module_func():
    return PATH
```

Package

```
from subpackage_1 import *
from subpackage_2 import *

print(pckg_1_module_func())
print(pckg_2_module_func())
print(pckg_3_module_func())
print(pckg_3_module_2_func())


# /Users/nightingale/PycharmProjects/package_project/subpackage_1
# /Users/nightingale/PycharmProjects/package_project/subpackage_2
#
/Users/nightingale/PycharmProjects/package_project/subpackage_2/subpackage_3
#
/Users/nightingale/PycharmProjects/package_project/subpackage_2/subpackage_3
```

Creating local package

When you install a package from PyPI, that package is available to all scripts in your environment. However, you can also install packages from your local computer using setuptools, and they'll also be made available in the same way.

Creating local package

```
face_cropper
├── README.md
├── face_cropper
│   ├── __init__.py
│   ├── core
│   │   ├── __init__.py
│   │   ├── face_extractor.py
│   │   └── model.py
│   ├── models
│   │   ├── deploy.prototxt
│   │   ├── opencv_face_detector.pbtxt
│   │   ├── opencv_face_detector_uint8.pb
│   │   └── res10_300x300_ssd_iter_140000_fp16.caffemodel
│   └── utils
│       ├── __init__.py
│       └── utils.py
├── requirements.txt
├── setup.py
└── version.py
```

Creating local package

```
# setup.py
from setuptools import setup, find_packages
. . .
setup(
    name='face_cropper',
    version=get_version(),
    packages=find_packages(),
    package_data={'face_cropper': ['models/*.caffemodel',
'models/*.pb', 'models/*.pbtxt', 'models/*.prototxt']},
    url='https://github.com/galeNightIn/face_cropper',
    description='Face cropper module',
    long_description=readme(),
    install_requires=get_requirements(),
    zip_safe=False
)
```

Creating local package

```
pip install PycharmProjects/face_cropper/

Processing ./PycharmProjects/face_cropper
…

Successfully installed face-cropper-0.1.0
```

```
from face_cropper import FaceExtractor
```

Package

```
subpackage_1
├── __init__.py
├── __main__.py
└── module_1.py
```

Package

```python
# subpackage_1/__main__.py
from .module_1 import module_func

print(f"Result for module func is:
{module_func()}")
```

Package

```
~ package_project » python3 -m subpackage_1

Result for module func is:
/Users/nightingale/PycharmProjects/package_project/sub
package_1
```

Resource imports

Resource imports

This module leverages Python's import system to provide access to *resources* within *packages*.

If you can import a package, you can access resources within that package.

Resources can be opened or read, in either binary or text mode.

Resource imports

importlib.resources gives access to resources within packages.

In this context, a resource is any file located within an importable package.

The file may or may not correspond to a physical file on the file system.

Resource imports

importlib.resources

since python 3.7

Resource imports

```
resource_package
├── __init__.py
└── resource_file.txt
```

Resource imports

```
# resource_example.py
from importlib import resources as r
import resource_package

with r.open_text(resource_package, "resource_file.txt") as
res:
   print(res.read())

# Hello, this is resource text file!
# Have a nice day!
```

Resource imports

```python
# zip_example.py
import sys
from importlib import resources

sys.path.append('zip_package.zip')

import zip_package

print(resources.read_binary(zip_package, "resource.bin"))
# b'binary'
```

Resource imports

```python
# zip_example.py
import sys
from importlib import resources

sys.path.append('zip_package.zip')

import zip_package

print(resources.read_binary(zip_package, "resource.bin"))
# b'binary'
```

Resource imports

```
importlib.resources.open_binary(package, resource)

importlib.resources.open_text(package, resource, encoding='utf-8', errors='strict')

importlib.resources.read_binary(package, resource)

importlib.resources.read_text(package, resource, encoding='utf-8', errors='strict')

importlib.resources.path(package, resource)

importlib.resources.is_resource(package, name)

importlib.resources.contents(package)
```

Dynamic imports

Dynamic imports

```
# dynamic_imports.py
f = __import__(name='functools', globals=globals(), locals=locals())

f = f.partial(lambda x, y: x * y, y=42)
print(f(10))
```

Dynamic imports

```
# dynamic_imports.py

...
def _import(name, *args, imp=__import__):
    print(f"Importing module {name!r}")
    return imp(name, *args)


import builtins
builtins.__import__ = _import


from subpackage_2 import *
```

Dynamic imports

```
# Importing module 'subpackage_2'
# Importing module ''
# Importing module 'module_1'
# Importing module 'os'
# Importing module 'module_2'
# Importing module 'os'
# Importing module 'module_1'
# Importing module 'os'
# Importing module 'subpackage_3'
```

Dynamic imports

```
import importlib

module = importlib.import_module("name")
```

Dynamic imports

```python
# dynamic_import_2.py
import importlib
from types import ModuleType
from typing import List, Union, Optional


def search_import(attr: str, from_modules: List[Union[str, ModuleType]]) ->
Optional[object]:
    for module in modules:
        try:
            if isinstance(module, ModuleType):
                mod = module
            elif isinstance(module, str):
                mod = importlib.import_module(module)
            else:
                raise TypeError('Must be list of strings or ModuleType')
            met = getattr(mod, attr, None)
            if met:
                return met
        except ImportError:
            continue
    return None
```

Dynamic imports

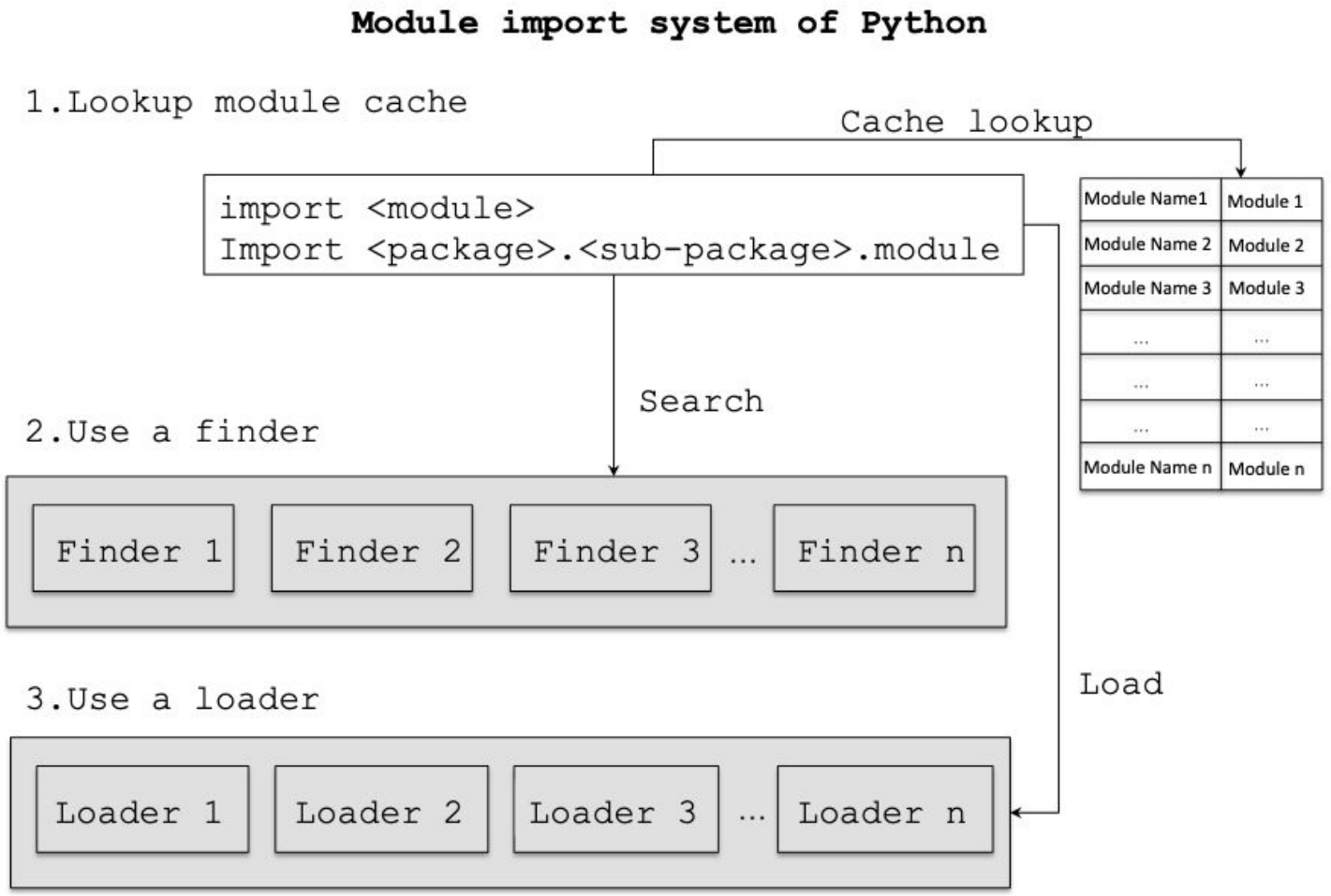
```
print(search_import("__import__", ["builtins"]))
# <built-in function __import__>

print(search_import("nothing", ["builtins"]))
# None

import math, builtins, scipy
print(search_import("sum", [math, builtins, scipy]))
# <built-in function sum>
```

Import system

Import system



Import system

Finding a Python Module:

- A **finder** uses a strategy and finds the given module in the system where Python is installed.
- Upon successfully resolved a given module to a path - a finder returns a module specification, which contains the information about the module to be loaded.

Import system

sys.meta_path controls which finders are called during the import process:

```
import sys

print(sys.meta_path)

# [
#   <class '_frozen_importlib.BuiltinImporter'>,
#   <class '_frozen_importlib.FrozenImporter'>,
#   <class '_frozen_importlib_external.PathFinder'>
# ]
```

Import system

sys.meta_path controls which finders are called during the import process:

```
import sys

print(sys.meta_path)

# [
#   <class '_frozen_importlib.BuiltinImporter'>,
#   <class '_frozen_importlib.FrozenImporter'>,
#   <class '_frozen_importlib_external.PathFinder'>
# ]
```

Import system

Note: built-in modules aren't shadowed by local modules because the built-in finder is called before the import path finder, which finds local modules

You can customize sys.meta_path to your liking

Import system

Note: built-in modules aren't shadowed by local modules because the built-in finder is called before the import path finder, which finds local modules

You can customize sys.meta_path to your liking

Import system

```
# pip_importer.py
from importlib import util
import subprocess
import sys


class PipFinder:
    @classmethod
    def find_spec(cls, name, path, target=None):
        print(f"Module {name!r} not installed.  Attempting to pip
install")
        cmd = f"{sys.executable} -m pip install {name}"
        try:
            subprocess.run(cmd.split(), check=True)
        except subprocess.CalledProcessError:
            return None
        return util.find_spec(name)
```

Import system

```
sys.meta_path.append(PipFinder)

import numpy

print(numpy.zeros(2))
# [0. 0.]
```

Import system

Loading a Python Module:

- Using the module specification a **loader** loads the python module and executes the module. Remember, a Python module contains both executable statements and definitions like function definitions.
- A class can combine the responsibilities of a **finder and a loader**, which is called an importer.

To read

Python tutorial 1, 2

Python doc

Import system in short