Модули

```
"""I'm a useful module."""

some_variable = "foobar"


def boo():
    return 42
```

- Модулем называется файл с расширением `py`.
- Каждый модуль задаёт новое пространство имён, атрибуты которого соответствуют именам, определённым в файле:

```
>>> import useful
>>> dir(useful)
[..., '__cached__', '__doc__', '__file__', '__name__',
 'boo', 'some_variable']
```

- Кроме явно определённых имён в модуле содержатся:

```
>>> useful.__name__
'useful'
>>> useful.__doc__
"I'm a useful module."
>>> useful.__file__
'./useful.py'
>>> useful.__cached__  # и другие
'./__pycache__/useful.cpython-34.pyc'
```

- Модуль можно выполнить, передав его в качестве аргумента интерпретатору.
- В этом случае переменная `__name__` внутри модуля будет иметь специальное значение `"__main__"`.
- Пример:

```
# useful.py

def test():
    assert boo() == 4

if __name__ == "__main__":
    print("Running tests ... ")
    test()
    print("OK")
```

```
$ python ./useful.py
Running tests ...
OK
```

- Оператор **import** “импортирует” модуль с указанным именем и создаёт на него ссылку в текущей области видимости:

```
>>> import useful  # исполняет модуль сверху вниз
>>> useful
<module 'useful' from './useful.py'>
```

- С помощью оператора **as** можно изменить имя переменной, в которую будет записана ссылка на модуль:

```
>>> import useful as alias
>>> alias
<module 'useful' from './useful.py'>
```

- Чтобы модуль был доступен для импорта, содержащая его директория должна присутствовать в списке `sys.path`:

```
>>> import sys
>>> sys.path
['', '/usr/local/lib/python3.4', ...]
# ^                             ^
# текущая директория   стандартная
#                      библиотека
```

- Оператор **from ... import** импортирует имя из другого модуля в текущую область видимости:

```
>>> from useful import boo
>>> boo()
42
```

- Синтаксис оператора позволяет перечислить несколько имен через запятую и, возможно, переименовать некоторые из них:

```
>>> from useful import boo as foo, some_variable
>>> foo()
42
>>> some_variable
'foobar'
```

- Оператор **`from ... import`** можно однозначно переписать через оператор **`import`**:

```
>>> from useful import boo as foo, some_variable
# HARDCORE REWRITING MAGIC
>>> import useful
>>> foo = useful.boo
>>> some_variable = useful.some_variable
>>> del useful  # Зачем это нужно?
```

- Всё сказанное про оператор **`import`** релевантно и для оператора **`from ... import`**.

Оператор `from...import *`

- В качестве второго аргумента оператора **`from ... import`** можно указать `*`.
- Если в модуле определена глобальная переменная `__all__`, то будут импортированы только те имена, которые в ней перечислены.
- Иначе — все имена из `globals()` модуля.

```
>>> from useful import *
>>> some_variable
'foobar'
```

- На практике оператор **`from ... import`** `*` используют редко, потому что он затрудняет чтение кода.

- Модуль в Python — это просто файл с расширением `py`.
- Модуль можно импортировать целиком или выборочно с помощью операторов **`import`** и **`from ... import`**.
- В момент импорта байт-код модуля выполняется интерпретатором сверху вниз.
- Три правила импортирования модулей:
  - размещайте все импорты в начале модуля,
  - сортируйте их в лексикографическом порядке,
  - располагайте блок **`import`** перед **`from ... import`**.
- Пример:

```
import os
import sys
from collections import OrderedDict
from itertools import islice
```

Пакеты

Пакет == директория с модулями

- Пакеты позволяют структурировать код на Python.
- Любая директория, содержащая файл `__init__.py`, автоматически становится пакетом.
- В качестве примера рассмотрим

```
useful
├── __init__.py  # !
├── bar.py
└── foo.py
```

- Импортируем пакет `useful`:

```
>>> import useful
>>> useful
<module 'useful' from './useful/__init__.py'>
>>> useful.foo
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'module' object has no attribute 'foo'
```

- При импорте пакета импортируется только `__init__.py`.

```
>>> import useful.foo
>>> useful  # !
<module 'useful' from './useful/__init__.py'>
>>> useful.foo
<module 'useful.foo' from './useful/foo.py'>
```

- Остальные модули необходимо импортировать явно:

```
>>> useful.bar
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'module' object has no attribute 'bar'
>>> from useful import bar
>>> bar
<module 'useful.bar' from './useful/bar.py'>
```

- Примеры, которые мы видели ранее, использовали *абсолютный* импорт — вызов оператора **import** содержал имя пакета:

```
import useful.foo
from useful import bar
```

- Можно (и нужно!) использовать *относительный* импорт:

```
from . import foo, bar
#    ^ соответствует имени пакета, в котором
#      вызывается импорт
```

- Почему? Не нужно изменять импорты при переименовании или перемещении пакета.
- Одно но: не работает в интерактивной оболочке:

```
>>> from . import useful
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
SystemError: Parent module '' not loaded, [...]
```

[2] https://www.python.org/dev/peps/pep-0328

- Внутри пакетов могут находиться не только модули, но и другие пакеты. Сделаем модуль `bar` пакетом:

```
useful
├── __init__.py
├── bar
│   ├── __init__.py
│   ├── boo.py
└── foo.py
```

- Синтаксически работа с вложенным пакетом `useful.bar` ничем не отличается от работы с его предшественником:

```
>>> import useful.bar
>>> useful.bar
<module 'useful.bar' from './useful/bar/__init__.py'>
```

- Замечание: в модуле `useful.bar.boo` тоже можно использовать абсолютный импорт:

```
from . import something
from ..foo import something_else
```

Ещё немного об `__init__.py`: фасад

- Задача модуля `__init__.py` — инициализировать пакет, поэтому не стоит реализовывать в нём всю логику.
- Что стоит делать в `__init__.py`?
  - Ничего.
  - Объявить глобальные для пакета переменные (может быть).
  - Оградить пакет фасадом, то есть импортировать имена из вложенных модулей и пакетов и определить `__all__`.
- Фасад для пакета `useful`:

```
useful
├── __init__.py
├── bar
│   ├── __init__.py
│   └── boo.py
└── foo.py
```

```python
# useful/bar/__init__.py
from .boo import *

__all__ = boo.__all__
```

```python
# useful/__init__.py
from .foo import *
from .bar import *

__all__ = foo.__all__ + bar.__all__
```

Плюсы и минусы использования фасада в `__init__.py`

- Плюсы:
  - Пользователю не нужно запоминать внутреннюю структуру пакета и думать, с чем он работает: модулем или пакетом.
    - `from urllib import urlopen` или
    - `from urllib.request import urlopen` или
    - `from urllib.requests import urlopen`?
  - Интерфейс не зависит от деталей реализации — можно перемещать код между внутренними модулями и пакетами.
  - Одним словом, инкапсуляция и инженерное счастье.
- Минусы?

```
$ time python -c "import sympy"
0.59s user 0.14s system 82% cpu 0.883 total
$ time python -c "import theano"
0.83s user 0.21s system 100% cpu 1.032 total
```

- Любой модуль можно выполнить как скрипт, передав его имя в качестве аргумента `-m`:

```
# useful/foo.py
print(__name__)
```

```
$ python -m useful.foo
'__main__'  # !
```

- Чтобы пакет был исполняемым, в нём должен быть файл `__main__.py`[3]:

```
# useful/__main__.py
print("It works!")

# useful/__init__.py
print("useful.__init__")
```

```
$ python -m useful
useful.__init__  # ?
It works!
$ python -m useful.__main__
useful.__init__
It works!
```

[3] https://www.python.org/dev/peps/pep-0338

- Пакеты — это способ группировать код на Python.
- Любая директория, содержащая файл `__init__.py`, задаёт пакет[4].
- Полезные детали, о которых стоит помнить:
  - в пакете можно (и нужно!) использовать относительный импорт вместо абсолютного;
  - с помощью `__init__.py` можно абстрагировать детали реализации пакета от пользователя,
  - а добавив файл `__main__.py` — сделать пакет исполняемым.

[4] Начиная с версии Python3.3 можно определять «именованные» пакеты, не требующие наличия `__init__.py`, но они выходят за рамки сегодняшней лекции

Система импорта

- Что происходит в момент исполнения оператора **import**?

```
>>> import dis
>>> dis.dis("import useful")
      0 LOAD_CONST               0 (0)
      3 LOAD_CONST               1 (None)
      6 IMPORT_NAME              0 (useful)
      9 STORE_NAME               0 (useful)
     12 LOAD_CONST               1 (None)
     15 RETURN_VALUE
```

- Инструкция IMPORT_NAME вызывает встроенную функцию __import__:

```
>>> useful = __import__("useful", globals(),
...                     None, None, 0)
<module 'useful' from './useful.py'>
```

- Поиск функции __import__ в builtins происходит **динамически**, а значит можно применить метод исправления обезьяны (*англ.*, monkey patching)!

```
>>> def import_wrapper(name, *args, imp=__import__):
...     print("importing ", name)
...     return imp(name, *args)
...
>>> import builtins
>>> builtins.__import__ = import_wrapper
>>> import collections
importing  collections
importing  _collections_abc
importing  _collections
importing  operator
# ...
```

Вопрос

Это довольно забавно, но что всё-таки делает функция `__import__`[5]?

[5] На практике для импорта модуля по имени следует использовать функцию `import_module` из пакета `importlib`.

- Сначала для модуля создаётся (пустой) объект.

```
>>> import types
>>> mod = types.ModuleType("useful")
```

- Затем байт код модуля вычисляется в пространстве имён созданного объекта:

```
>>> with open("./useful.py") as handle:
...     source = handle.read()
...
>>> code = compile(source, "useful.py", mode="exec")
>>> exec(code, mod.__dict__)
>>> dir(mod)
[..., 'boo', 'some_variable']
```

- В завершение объект присваивается переменной с соответсвующим именем:

```
>>> useful = mod
>>> useful  # ≈ import useful
<module 'useful'>
```

При первом импорте исходный код модуля компилируется в байткод, который кешируется в файле с расширением pyc.

```
>>> def read_int(handle)
...     return int.from_bytes(handle.read(4), "little")
...
>>> import useful
>>> useful.__cached__
'./__pycache__/useful.cpython-34.pyc'
>>> handle = open(useful.__cached__, "rb")
>>> magic = read_int(handle)  # "3310\r\n" для 3.4
>>> mtime = read_int(handle)  # дата последнего изменения
>>> import time
>>> print(time.asctime(time.localtime(c)))
Sun Nov  1 13:30:44 2015
>>> read_int(handle)          # размер файла
71
>>> import marshal
>>> marshal.loads(handle.read())
<code object <module> at [...], file "./useful.py", line 1>
```

Функция `__import__` и `sys.modules`

- Полученный в результате импорта модуль попадает в специальный словарь `sys.modules`.
- Ключом в словаре является имя модуля, то есть значение атрибута `__name__`:

```
>>> import useful
>>> import sys
>>> "useful" in sys.modules
True
```

- Повторный импорт уже загруженного модуля **не приводит** к его перезагрузке:

```
>>> id(sys.modules["useful"])
4329007128
>>> import useful
>>> id(sys.modules["useful"])
4329007128
```

- Мотивирующий пример:

```
# useful/__init__.py
from .foo import *

some_variable = 42
```

```
# useful/foo.py
from . import some_variable

def foo():
    print(some_variable)
```

- Попробуем импортировать пакет `useful`:

```
>>> import useful
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "./useful/__init__.py", line 3, in <module>
    from .foo import *
  File "./useful/foo.py", line 1, in <module>
    from . import some_variable
ImportError: cannot import name 'some_variable'
```

- Очевидный вопрос: что происходит?

Бороться с циклическими импортами можно, как минимум, тремя способами.

1. Вынести общую функциональность в отдельный модуль.

```
# useful/_common.py
some_variable = 42
```

2. Сделать импорт локальным для использующих его функций или методов:

```
# useful/foo.py
def foo():
    from . import some_variable
    print(some_variable)
```

3. Пойти наперекор PEP-8 (!) и изменить модуль `__init__.py` так, чтобы импорт происходил в конце модуля:

```
# useful/__init__.py
some_variable = 42

from .foo import *
```

Подробнее о sys.path

- Напоминание:
  - sys.path — список “путей”, в которых Python ищет модули и пакеты при импорте;
  - “путь” — произвольная строка, например, директория или zip-архив;
  - импортировать можно только то, что доступно через sys.path.
- При импорте модуля обход sys.path происходит слева направо до тех пор, пока модуль не будет найден:

```
>>> import sys
>>> sys.path
['', '/usr/lib/python3.4', ...]
# ^                        ^
# ищем здесь          потом здесь
>>> open("collections.py", "w").close()
>>> import collections
>>> collections  # локальный модуль!
<module 'collections' from './collections.py'>
```

- Мораль: **никогда** не называйте свои модули как модули стандартной библиотеки.

Построение sys.path*

- При старте интерпретатора в `sys.path` находятся текущая директория и директории стандартной библиотеки:

```
$ python3 -S -c 'import sys; print(sys.path)'
['', '/usr/lib/python3.4/', ...]
```

- Затем к ним добавляются директории с пакетами, установленными пользователем:

```
$ python3 -c 'import sys; print(sys.path)'
[..., '/usr/lib/python3.4/site-packages']
```

- Директории, перечисленные в переменной окружения `PYTHONPATH`, попадают в начало `sys.path`:

```
$ PYTHONPATH=foo:bar python3 \
>     -c 'import sys; print(sys.path)'
['', './foo', './bar', ...]
```

- Кроме того, `sys.path` можно изменять программно:

```
>>> import sys
>>> sys.path.extend(["foo", "bar"])
```

- Может показаться, что `sys.path` — властелин и повелитель импорта, но это не так.
- Управляет импортом `sys.meta_path`:

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

- Импортер — экземпляр класса, реализующего протоколы искателя *aka* `Finder` и загрузчика *aka* `Loader`[6]:
  - `Finder` любым известным ему способом ищет модуль,
  - а `Loader` загружает то, что `Finder` нашёл.

[6] Если класс реализует оба протокола, он называется импортером.

Finder должен реализовывать метод find_spec, принимающий имя модуля и возвращающий ModuleSpec или **None**, если модуль не был найден:

```
>>> import sys
>>> builtin_finder, _, path_finder = sys.meta_path
>>> builtin_finder.find_spec("itertools")
ModuleSpec(name='itertools',
           loader=<class '_frozen_importlib.BuiltinImporter'>,
           origin='built-in')
>>> builtin_finder.find_spec("enum")
>>> path_finder.find_spec("enum")
ModuleSpec(name='enum',
           loader=<_frozen_importlib.SourceFileLoader [...]>,
           origin='/usr/lib/python3.4/enum.py')
>>> path_finder.find_spec("math")
ModuleSpec(name='math',
           loader=<_frozen_importlib.ExtensionFileLoader [...]>
           origin='/usr/lib/python3.4/lib-dynload/math.so')
```

ModuleSpec содержит всю необходимую для загрузки информацию о модуле:

```
>>> spec = path_finder.find_spec("collections")
>>> spec.name
'collections'
>>> spec.origin
'/usr/lib/python3.4/collections/__init__.py'
>>> spec.cached
'/usr/lib/python3.4/collections/__pycache__/[...].pyc'
>>> spec.parent
'collections'
>>> spec.submodule_search_locations
['/usr/lib/python3.4/collections']
>>> spec.loader
<_frozen_importlib.SourceFileLoader object at 0x101a71f28>
```

- Мотивация: у нескольких модулей одинаковый интерфейс и отличаются они, например, скоростью работы.
- Хочется попробовать импортировать более быстрый и в случае ошибки использовать медленный.

```
try:
    import _useful_speedups as useful
except ImportError:
    import useful  # В чём проблема?
```

- Более надёжный вариант использует функцию `find_spec`[7] из модуля `importlib.util`:

```
from importlib.util import find_spec

if find_spec("_useful_speedups"):
    import _useful_speedups as useful
else:
    import useful
```

[7] Функция `find_spec` обходит `sys.meta_path` и последовательно вызывает одноимённый метод у каждого из импортеров, пока не найдёт модуль.

Loader должен реализовывать два метода create_module для создания пустого модуля и exec_module для его заполнения.

```
>>> from importlib.util import find_spec
>>> spec = find_spec("enum")
>>> mod = spec.loader.create_module(spec)
>>> mod  # None --- используем стандартный загрузчик.
>>> from importlib.util import module_from_spec
>>> mod = module_from_spec(spec)
>>> mod
<module 'enum' from '/usr/lib/python3.5/enum.py'>
>>> dir(mod)
['__cached__', '__doc__', '__file__', '__loader__', ...]
>>> spec.loader.exec_module(mod)
>>> dir(mod)
['DynamicClassAttribute', 'Enum', 'EnumMeta', ...]
```

Пример: автоматическая установка*

```
import sys, subprocess
from importlib.util import find_spec
from importlib.abc import MetaPathFinder

class AutoInstall(MetaPathFinder):
    _loaded = set()

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

sys.meta_path.append(AutoInstall)
```

Знакомьтесь, sys.path_hooks!

- К сожалению, на `sys.meta_path`, искателях и загрузчиках история не заканчивается.
- В игру вступает `sys.path_hooks` — ещё один список, используемый искателем `PathFinder`.
- В `sys.path_hooks` находятся функции, задача которых — подобрать каждому элементу `sys.path` искателя.

```
>>> import sys
>>> sys.path_hooks
[<class 'zipimport.zipimporter'>,
 <function [...].path_hook_for_FileFinder at [...]>]
>>> sys.path_hooks[0]('/usr/lib/python3.4/plat-darwin')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
zipimport.ZipImportError: not a Zip file
>>> sys.path_hooks[1]('/usr/lib/python3.4/plat-darwin')
FileFinder('/usr/lib/python3.4/plat-darwin')
```

Пример: удалённый импорт*

```
import re
import sys
from urllib.request import urlopen

def url_hook(url):
    if not url.startswith(("http", "https")):
        raise ImportError
    with urlopen(url) as page:
        data = page.read().decode("utf-8")
    filenames = re.findall("[a-zA-Z_][a-zA-Z0-0_]*.py",
                           data)
    modnames = {name[:-3] for name in filenames}
    return URLFinder(url, modnames)

sys.path_hooks.append(url_hook)
```

```
from importlib.abc import PathEntryFinder
from importlib.utils import spec_from_loader

class URLFinder(PathEntryFinder):
    def __init__(self, url, available):
        self.url = url
        self.available = available

    def find_spec(self, name, target=None):
        if name in self.available:
            origin = "{}/{}.py".format(self.url, name)
            loader = URLLoader()
            return spec_from_loader(name, loader,
                                    origin=origin)
        else:
            return None
```

```
from urllib.request import urlopen


class URLLoader:
    def create_module(self, target):
        return None

    def exec_module(self, module):
        with urlopen(module.__spec__.origin) as page:
            source = page.read()
        code = compile(source, module.__spec__.origin,
                       mode="exec")
        exec(code, module.__dict__)
```

Удаленный импорт в действии

```
# remote.py
print("It works!")
```

```
$ python -m http.server
Serving HTTP on 0.0.0.0 port 8000 ...
```

```
>>> import remote
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'remote'
>>> import sys
>>> sys.path.append("http://localhost:8000")
>>> import remote
It works!
>>> remote
<module 'remote' (http://localhost:8000/remote.py)>
```

Система импорта: резюме

- Система импорта нетривиальна.
- Импорт контролируется импортерами, задача которых — найти модуль по имени и загрузить его.
- После загрузки интерпретатора в `sys.meta_path` появляются импортеры для работы со встроенными модулями, а также модулями в zip-архивах и “путях”.
- “Путевой” искатель *aka* `PathFinder` можно расширять, добавляя новые “пути” к `sys.path` и функции к `sys.path_hooks`.