Лекция 7

Модули и пакеты

Модуль в языке Python - это обычный файл с расширением .py. Модуль может содержать любой программный код на языке Python.

Каждая программа, которую мы писали до сих пор, находилась в отдельном файле .py, который можно считать не только программой, но и модулем. Основное различие между модулем и программой состоит в том, что программа предназначена для того, чтобы ее запускали, тогда как модуль предназначен для того, чтобы его импортировали и использовали в программах.

Не все модули располагаются в файлах с расширением .py, например, модуль sys встроен в Python, а некоторые модули написаны на других языках программирования (чаще всего на языке C). Однако большая часть библиотеки языка Python написана именно на языке Python.

Импортирование может выполняться несколькими синтаксическими конструкциями, например:

```
 import importable
import importablel, importable2 importableN
import importable as preferred_name
```

In [1]:
```
# пример:
import math
```

In [6]:
```
# одновременный импорт нескольких модулей:
import math, collections
```

In [5]:
```
# для импортируемого модуля задается произвольное имя:
import numpy as np
```

In [7]:
```
# использование импортированных модулей:
math.cos(0.0)
```

Out[7]:
```
1.0
```

In [9]:
```
nt = collections.namedtuple('Person', ['name', 'age'])
nt
```

Out[9]:
```
__main__.Person
```

In [10]:
```
np.arange(10)
```

Out[10]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Теоретически последний способ может привести к конфликтам имен, но на практике синтаксис as обычно используется, чтобы как раз избежать их. Подобное переименование, в частности, удобно использовать при экспериментировании с различными реализациями одного и того же модуля.

Например, допустим, что у нас имеется два модуля MyModuleA и MyModuleB, которые имеют один и тот же API (Application Programming Interface - прикладной программный интерфейс). Мы могли бы в программе записать инструкцию import MyModuleA as MyModule, а позднее легко переключиться на использование import MyModuleB as MyModule.

** Расположение и порядок выполнения **

Обычно все инструкции import помещаются в начало файла .py после описания модуля. Рекомендуется сначала импортировать модули стандартной библиотеки, затем модули сторонних разработчиков и в последнюю очередь свои собственные модули.

Альтернативный вариант использования инструкции import:

```
 from importable import object as preferred_name
from importable import objecth object2 objectN
from importable import (objectl, object2, object3, object4, objects,
object6 objectN)
from importable import *
```

Эти синтаксические конструкции могут приводить к конфликтам имен, поскольку они обеспечивают непосредственный доступ к импортируемым объектам (переменным, функциям, типам данных или модулям). Если для импортирования большого числа объектов необходимо использовать синтаксис from ... import, мы можем расположить инструкцию импорта в нескольких строках, либо экранируя каждый символ перевода строки, кроме последнего, либо заключая список имен объектов в круглые скобки, как показано в третьем примере.

In [11]:
```
from math import sin
```

In [15]:
```
# использование импортированного объекта без указания имени модуля:
sin(0.0)
```

Out[15]: 0.0

In [13]:
```
from math import (sin, cos, tan)
```

In [14]:
```
tan(0.0)
```

Out[14]: 0.0

В синтаксической конструкции:

```
from importable import *
```

символ «*» означает «импортировать все имена, которые не являются частными». На практике это означает, что будут импортированы все объекты из модуля за исключением тех, чьи имена начинаются с символа подчеркивания, либо, если в модуле определена глобальная переменная *_all_* со списком имен, будут импортированы все объекты, имена которых перечислены в переменной *_all_*.

Синтаксис import * потенциально опасен появлением конфликтов имен.

In [ ]:
```
```

In [18]:
```
filename = 'C:\Python\Ipynb\2016\lec_6'
```

In [3]:
```
import os

print(os.path.basename(filename)) # безопасный доступ по полным квалифицированным именам
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-3-39418f1febf1> in <module>()
      1 import os
      2
----> 3 print(os.path.basename(filename)) # безопасный доступ по полным квалифицированным именам

NameError: name 'filename' is not defined
```

In [21]:
```
import os.path as p

print(p.basename(filename))
```

```
lec_6
```

In [22]:
```
from os import path

print(path.basename(filename))
```

```
lec_6
```

In [23]:
```
from os.path import basename

print(basename(filename))
```

```
lec_6
```

In [24]:
```
from os.path import *

print(basename(filename)) # есть риск множественных конфликтов имен
```

```
lec_6
```

__ Порядок поиска файлов содержащих модули __

Порядок следующий:

1. каталог, где находится сама программа, даже если она вызывается из другого каталога;
2. пути к каталогам из переменной окружения PYTHONPATH, если она определена;
3. пути к каталогам стандартной библиотеки языка Python - они определяются на этапе установки Python.

Программа может импортировать некоторые модули, которые в свою очередь импортируют другие модули, включая те, что уже были импортированы. Это не является проблемой. Всякий раз, когда выполняется попытка импортировать модуль, интерпретатор Python сначала проверяет - не был ли импортирован требуемый модуль ранее.

Если модуль еще не был импортирован, Python выполняет скомпилированный байт-код модуля, создавая тем самым переменные, функции и другие объекты модуля, после чего добавляет во внутреннюю структуру запись о том, что модуль был импортирован.

Когда интерпретатору требуется скомпилированный байт-код модуля, он генерирует его автоматически - этим Python отличается от таких языков программирования, как Java, где компилирование в байт-код должно выполняться явно. Сначала интерпретатор попытается отыскать файл, имя которого совпадает с именем файла, имеющего расширение .py, но имеющий

расширение .pyo - это оптимизированный байт-код скомпилированной версии модуля. Если файл с расширением .pyo не будет найден (или он более старый, чем файл с расширением .py), интерпретатор попытается отыскать одноименный файл с расширением .pyc - это неоптимизированный байт-код скомпилированной версии модуля. Если интерпретатор обнаружит актуальную скомпилированную версию модуля, он загрузит ее; в противном случае Python загрузит файл с расширением .py и скомпилирует его в байт-код. В любом случае интерпретатор загрузит в память модуль в виде скомпилированного байт-кода.

При любых последующих попытках импортировать этот модуль интерпретатор будет обнаруживать, что модуль уже импортирован и не будет выполнять никаких действий.

Пакеты

Пакет - это простой каталог, содержащий множество модулей и файл с именем *_init_*.py.

Например, допустим, что у нас имеется некоторое множество файлов модулей, предназначенных для чтения и записи графических файлов различных форматов. Если поместить эти модули в каталог Graphics вместе с пустым файлом *_init_*.py, то этот каталог превратится в пакет:

```
Graphics/
    __init__.py
    Bmp.py
    Jpeg.py
    Png.py
    Tiff.py
```

Пока каталог Graphics является подкаталогом каталога с программой или находится в пути поиска Python, мы будем иметь возможность импортировать любой из этих модулей и использовать их. Мы должны сделать все возможное, чтобы гарантировать несовпадение имени нашего модуля верхнего уровня (Graphics) с каким-либо из имен верхнего уровня в стандартной библиотеке - с целью избежать конфликтов имен.

In [26]:
```
import Graphics.Bmp

image = Graphics.Bmp.g_load("bashful.bmp")
```

```
file bashful.bmp loaded as BMP
```

In [28]:
```
import Graphics.Jpeg as Jpeg

image = Jpeg.g_load("doc.jpeg")
```

```
file doc.jpeg loaded as JPEG
```

In [30]:
```
from Graphics import Png

image = Png.g_load("dopey.png")
```

```
file dopey.png loaded as PNG
```

In [32]:
```
from Graphics import Tiff as picture

image = picture.g_load("grumpy.tiff")
```

```
file grumpy.tiff loaded as TIFF
```

В некоторых ситуациях бывает удобно загружать все модули пакета одной инструкцией. Для этого необходимо в файле *_init_*.py необходимо задать переменную *_all_* , которая указывала бы, какие модули должны загружаться.

Например, ниже приводится необходимая строка для файла Graphics/*_init_*.py:

```
all = ["Bmp", "Jpeg", "Png", "Tiff"]
```

Этим ограничивается необходимое содержимое файла, но помимо этого, мы можем поместить в него любой программный код, какой только пожелаем.

In [34]:
```
# Теперь мы можем использовать другую разновидность инструкции import:
# Синтаксис from package import * напрямую импортирует все имена  модулей, упомянутые в списке all .

from Graphics import *

image = Tiff.g_load("sleepy.tiff")
```

```
file sleepy.tiff loaded as TIFF
```

Python позволяет организовать произвольное количество уровней вложенности пакетов. Пример:

```
Graphics/
    __init__.py
    Bmp.py
    Jpeg.py
    Png.py
    Tiff.py
    Vector/
```

```
__init__.py
Eps.py
Svg.py
```

Повторная загрузка модулей

In [42]:
```
image = Graphics.Bmp.g_load("bashful.bmp")
```

```
file bashful.bmp loaded as BMP
```

In [45]:
```
from imp import reload

reload(Bmp)
```

Out[45]:
```
<module 'Graphics.Bmp' from 'E:\\YandexDisk\\Python\\Ipynb\\fa1_2016\\lec_6\\Graphics\\Bmp.py'>
```

In [46]:
```
image = Graphics.Bmp.g_load("bashful.bmp")
```

```
file bashful.bmp loaded as BMP v2
```

Запуск программы

Всякий раз, когда файл с расширением .py запускается как программа, интерпретатор Python создает в программе переменную с именем *_name_* и записывает в нее строку "*_main_*". Это можно использовать для того чтобы выполнять некоторый программный код только если файл запускается как программа.

In [ ]:
```
# Пример, файл Bmp.py:

def g_load(file):
    print("file {} loaded as BMP v2".format(file))
    return None

if __name__ == '__main__':
    print("Test  Bmp.py")
    print(g_load("test.bmp"))
```

In [47]:
```
ls
```

```
 '®¬ ў гбва®©бвўЃ Е Ё¬ГЃв ¬ГвЄг Data
 'ГаЁ©л© ®¬Ґа в®¬ : 1C8C-28F1

 '®¤Ґа¦Ё¬®Ґ Ї ЇЄЁ E:\YandexDisk\Python\Ipynb\fa1_2016\lec_6

05.12.2016  22:41    <DIR>          .
05.12.2016  22:41    <DIR>          ..
05.12.2016  22:24    <DIR>          Graphics
05.12.2016  22:41            22я545 lec6_v2016s2_v1.ipynb
               1 д ©«®ў          22я545 Ў ©в
               3 Ї Ї®Є  110я528я200я704 Ў ©в бў®Ў®¤®
```

In [49]:
```
%run Graphics/Bmp.py
```

```
Test  Bmp.py
file test.bmp loaded as BMP v2
None
```

Модуль sys позволяет также получить те слова, которые были введены в команде, запустившей сценарий на языке Python. Эти слова обычно называются аргументами командной строки и находятся во встроенном списке строк sys.argv. Программисты на C могут заметить сходство с массивом argv в языке C (массивом строк).

In [ ]:
```
# Jpeg.py

import sys

def g_load(file):
    print('file {} loaded as JPEG'.format(file))
    return None

if __name__ == '__main__':
    print('Test  Jpeg.py')
    print('Command line arguments: ', sys.argv)
    print(g_load('test.jpeg'))
```

In [51]:
```
%run Graphics/Jpeg.py arg1 my_value 11
```

```
Test  Jpeg.py
Command line arguments:  ['Graphics/Jpeg.py', 'arg1', 'my_value', '11']
file test.jpeg loaded as JPEG
None
```

Для серьезной работы с аргументами командной строки подходит модуль argparse:

https://docs.python.org/3.5/library/argparse.html (https://docs.python.org/3.5/library/argparse.html)

https://docs.python.org/3.5/library/argparse.html#argparse.ArgumentParser.add_argument (https://docs.python.org/3.5/library/argparse.html#argparse.ArgumentParser.add_argument)

https://pymotw.com/3/argparse/index.html (https://pymotw.com/3/argparse/index.html)

In [56]:
```
import argparse

parser = argparse.ArgumentParser(description='Short sample app')

parser.add_argument('-a', action="store_true", default=False)
parser.add_argument('-b', action="store", dest="b")
parser.add_argument('-c', action="store", dest="c", type=int)

ns = parser.parse_args(['-a', '-bval', '-c', '3'])
print(ns)
```

```
Namespace(a=True, b='val', c=3)
```

In [58]:
```
ns.a
```

Out[58]:
```
True
```

In [59]:
```
ns.b
```

Out[59]:
```
'val'
```

In [61]:
```
ns.c
```

Out[61]:
```
3
```

In [60]:
```
vars(ns)
```

Out[60]:
```
{'a': True, 'b': 'val', 'c': 3}
```

In [55]:
```
import argparse

parser = argparse.ArgumentParser(description='Process some integers.')
parser.add_argument('integers', metavar='N', type=int, nargs='+',
                    help='an integer for the accumulator')
parser.add_argument('--sum', dest='accumulate', action='store_const',
                    const=sum, default=max,
                    help='sum the integers (default: find the max)')

# $ python prog.py 1 2 3 4 --sum
args = parser.parse_args(['1', '2', '3', '--sum'])
print(args.accumulate(args.integers))
```

```
6
```

Установка модулей из глобального репозитария

pip - это система управления пакетами, которая используется для установки и управления программными пакетами, написанными на Python. Начиная с Python версии 3.4, pip поставляется вместе с интерпретатором python.

pip очень легко использовать для загрузки модулей из PyPI - the Python Package Index (https://pypi.python.org/pypi (https://pypi.python.org/pypi)). На 05.12.2016 в этом репозитории содержалось 94 220 пакетов!

Для установки пакетов в командной строке нужно выполнить (для выполнения команды могут потребоваться права администратора):

pip install package_name

pip help - помощь по доступным командам.

pip install package_name - установка пакета(ов).

pip uninstall package_name - удаление пакета(ов).

pip list - список установленных пакетов.

pip show package_name - показывает информацию об установленном пакете.

pip search - поиск пакетов по имени.

pip --proxy user:passwd@proxy.server (mailto:passwd@proxy.server):port - использование с прокси.

pip install -U - обновление пакета(ов).

pip install --force-reinstall - при обновлении, переустановить пакет, даже если он последней версии.

Модули Python:

https://docs.python.org/3/library/index.html (https://docs.python.org/3/library/index.html)

Хорошие учебные материалы по встроенным модулям:

https://pymotw.com/3/ (https://pymotw.com/3/)

Встроенные функции

https://docs.python.org/3/library/functions.html (https://docs.python.org/3/library/functions.html)

http://python-reference.readthedocs.io/en/latest/docs/functions/ (http://python-reference.readthedocs.io/en/latest/docs/functions/)

---

https://pymotw.com/3/collections/index.html (https://pymotw.com/3/collections/index.html)

https://docs.python.org/3.5/library/collections.html (https://docs.python.org/3.5/library/collections.html)

In [ ]: