**Зачем нужны модули**

* Разработка большого объема кода
* командная разработка (каждый правит свой файл)
* повторное использование кода (функций)
* DEA! можно переписать *реализацию* модуля, оставив только прежние названия и аргументы функций (и классов)

**Программы и модули**

Программа - любой файл с расширением .py   
Предназначена, чтобы ее запускали.

Модуль - файл с расширением .py   
Могут быть модули, написанные на других языках (например, С).   
Предназначены для import (или запуска как отдельного модуля)

**Как работает import**

Напишем программу module\_using\_sys.py

import sys

print('The command line arguments are:')

for i in sys.argv:

print(i)

print('\n\nThe PYTHONPATH is', sys.path, '\n')

Запустим ее:

$ python module\_using\_sys.py we are arguments # each arg is separated by white space

The command line arguments are:

module\_using\_sys.py

we

are

arguments

The PYTHONPATH is ['/tmp/py',

# many entries here, not shown here

'/Library/Python/2.7/site-packages',

'/usr/local/lib/python2.7/site-packages']

* При выполнении import ищется запись о том, не был ли он импортирован.
* Если не был, то ищется модуль sys. Так как это встроенный модуль (build-in), то известно где его искать.
  + Ищем в sys.path:
    - Сначала в каталоге, где находится программа, даже если запускали ее из другого места.
    - Потом из переменной окружения PYTHONPATH (если есть).
    - пути к каталогам стандартной библиотеки python
  + IP Переменную sys.path можно изменять во время выполнения программы.
* При **первом** import этого модуля в программе происходит инициация модуля. Больше этот модуль не будет инициироваться. Инициализация это:
  + Python выполняет скомпилированный *байт-код* модуля, создавая тем самым переменные, функции

и другие объекты модуля,

* + добавляет во внутреннюю структуру запись о том, что модуль был импортирован

LERT! не создавайте программы или модули, имена которых совпадают с именами модулей или каталогов верхнего уровня в библиотеке, если только вы не пытаетесь подставить свою собственную реализацию и ваше переопределение преднамеренно.

Чтобы проверить, есть ли уже в системе модуль или пакет с именем, например Music, запустите

python -c "import Music"

Если модуля нет, то будет ошибка ImportError[?](http://acm.mipt.ru/twiki/bin/edit/Cintro/ImportError?topicparent=Cintro.PythonModules).

IP Проверка *сейчас* не гарантирует, что мы НЕ поставим такой модуль потом.

**Байт-код**

Чтобы программа выполнялась быстрее, ее можно заранее скомпилировать в байт-код (.pyc) и оптимизировать (.pyo)

Байт-код в питоне создается по необходимости. В других языках (Java), его нужно создавать явно.

Когда интерпретатору нужен байт-код для файла myname.py:

* ищет файл myname.pyo
* если не находит или находит, но старый, то ищет файл myname.pyc
* иначе скомпилирует и загрузит myname.pyc

Чтобы получить оптимизированный байт-код запустите питон с ключом -O (буква О, а не ноль!) или установите переменную среды PYTHONOPTIMIZE в O (директория должна иметь права на запись).

Чтобы НЕ сохранять байт-код в файле, запустите питон с ключом -B или установите переменную среды окружения PYTHONDONTWRITEBYTECODE

**Формы import**

Чтобы использовать функцию из другого файла myfile.py нужно просто написать import myfyle

import пишут вверху файла, по одному пакету на строчку.

Хорошо:

import os

import os.path

import sys

Плохо:

import os, os.path, sys

Порядок импорта:

* стандартные библиотеки (sys, os)
* внешние библиотеки (turtle)
* свои пакеты

**import имя\_модуля as короткое\_имя**

Импортируем имя модуля и обращаемся к его функциям и переменным по имени модуль.функция

import math

import numpy as np

a = math.sqrt(2)

b = np.sqrt(2)

* Плюсы:
  + нет конфликта имен с локальными переменными  
    в примере выше используем и не путаемся переменную sys.argv и argv. Это разные переменные.
  + нет конфликта имен с функциями с одинаковыми именами из разных пакетов.   
    math.sqrt и numpy.sqrt
  + когда читаешь код, легко понять, что это импортированная функция из какого-то (быть может внешнего) пакета.  
    Полезно, когда мы хотим перейти с одного внешнего пакета на другой или оценить количество работы по переходу.
* Минусы:
  + Длиннее имена, труднее читать.

**from модуль import функция1, функция2, функция3**

Импортируем только нужные функции и переменные. Обращаемся прямо по их именам.

from math import sin, cos, pi

a = sin(pi) + cos(pi)

* Плюсы:
  + Проще читать алгоритм, записанный в коде.
* Минусы:
  + при чтении кода не понятно, какая часть - использование внешних модулей. (Возможно понадобится при рефакторинге).

**from модуль import \***

Импортировать все функции и методы из этого модуля.

LERT! не делайте так. Много лишнего импорта. Возможен конфликт имен функций из разных модулей.

from math import \*

a = sin(pi) + cos(pi)

Будут импортированы:

* если есть глобальная переменная \_\_all\_\_ , то только то, что перечислено в ней;
* иначе все объекты из модуля
  + кроме тех, чьи имена начинаются с символа подчеркивания (\_),

**Конфликты имен**

import os

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

# квалифицированным именам

import os.path as path

print(path.basename(filename)) # есть риск конфликта имен с модулем path

from os import path

print(path.basename(filename)) # есть риск конфликта имен с модулем path

from os.path import basename

print(basename(filename)) # есть риск конфликта имен с модулем basename

from os.path import \*

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

**Пакеты**

Пакет – это каталог, содержащий множество модулей и файл с именем \_\_init\_\_.py

Пусть мы написали функции, которые работают с изображениями, например, save и load.

Методы для каждого формата (bmp, png, jpeg) поместили в отдельный файл Bmp.py, Png.py, Jpeg.py

Все эти файлы поместили в директорию Graphics. Туда добавили *пустой* файл \_\_init\_\_.py

Директория превратилась в пакет Graphics.

Graphics/

\_\_init\_\_.py

Bmp.py

Jpeg.py

Png.py

Tiff.py

Xpm.py

Дальше мы можем работать с этим пакетом, как привыкли:

import Graphics

**Переменная \_\_all\_\_**

IP Иногда удобнее загружать все модули пакета одной инструкцией. Для этого в файле \_\_init\_\_.py добавьте в переменную \_\_all\_\_ какие модули нужно загрузить

Файл \_\_init\_\_.py:

\_\_all\_\_ = ["Bmp", "Jpeg", "Png", "Tiff", "Xpm"]

Можно в \_\_init\_\_.py добавлять любой другой код.

Теперь можно импортировать все модули пакета, используя \*.

from Graphics import \*

image = Xpm.load("sleepy.xpm")

**Вложенные пакеты**

Пусть в директории Graphics есть директория Vector с модулями Eps.py и Svg.py:

Graphics/

\_\_init\_\_.py

Bmp.py

Jpeg.py

Png.py

Tiff.py

Vector/

\_\_init\_\_.py

Eps.py

Svg.py

Xpm.py

Чтобы директория Vector стала пакетом, поместим в него файл \_\_init\_\_.py (пустой или с \_\_all\_\_)

import Graphics.Vector.Eps

image = Graphics.Vector.Eps.load("sneezy.eps")

**Пример своего модуля**

Пример взят из книги Саммерфильда, Глава 5 Модули / Модули и пакеты / Пакеты.

Пример файла TextUtil[?](http://acm.mipt.ru/twiki/bin/edit/Cintro/TextUtil?topicparent=Cintro.PythonModules).py

#!/usr/bin/env python3

# Copyright (c) 2008 Qtrac Ltd. All rights reserved.

"""

Этот модуль предоставляет несколько функций манипулирования строками.

>>> is\_balanced("(Python (is (not (lisp))))")

True

>>> shorten("The Crossing", 10)

'The Cro...'

>>> simplify(" some text with spurious whitespace ")

'some text with spurious whitespace'

"""

import string

def simplify(text, whitespace=string.whitespace, delete=""):

r"""Возвращает текст, из которого удалены лишние пробелы.

Параметр whitespace - это строка символов, каждый из которых

считается символом пробела.Если параметр delete не пустой,

он должен содержать строку, и тогда все символы, входящие

в состав строки delete, будут удалены из строки результата.

>>> simplify(" this and\n that\t too")

'this and that too'

>>> simplify(" Washington D.C.\n")

'Washington D.C.'

>>> simplify(" Washington D.C.\n", delete=",;:.")

'Washington DC'

>>> simplify(" disemvoweled ", delete="aeiou")

'dsmvwld'

"""

result = []

word = ""

for char in text:

if char in delete:

continue

elif char in whitespace:

if word:

result.append(word)

word = ""

else:

word += char

if word:

result.append(word)

return " ".join(result)

def shorten(text, length=25, indicator="..."):

"""Возвращает text или усеченную его копию с добавлением indicator в конце

text &#8211; любая строка;

length &#8211; максимальная длина возвращаемой строки string (включая indicator);

indicator &#8211; строка, добавляемая в конец результата, чтобы показать, что текст аргумента text был усечен

>>> shorten("The Road")

'The Road'

>>> shorten("No Country for Old Men", 20)

'No Country for Ol...'

>>> shorten("Cities of the Plain", 15, "\*")

'Cities of the \*'

"""

if len(text) > length:

text = text[:length - len(indicator)] + indicator

return text

def is\_balanced(text, brackets="()[]{}<>"):

"""

Проверяет баланс открывающих и закрывающих скобок в text.

"""

for left, right in zip(brackets[::2], brackets[1::2]):

assert left != right, "the bracket characters must differ"

.... дальше напишете сами, через стек на основе списка, а не через тупой подсчет количества открывающих и закрывающих :)

if \_\_name\_\_ == "\_\_main\_\_":

import doctest # import, который нужен только для тестирования, пишем прямо тут

doctest.testmod() # выполним примеры (как - расскажем дальше)

Разберем код по строкам.

#!/usr/bin/env python3

Позволяет запускать файл (если есть права доступа) как исполняемый, без указания python3 в командной строке.

./TextUtil.py

Пользователь может не думать, на чем написана ваша программа и чем ее запускать.

Далее в """...""" - документация по модулю. Она доступна в других модулях через TextUtil.\_\_doc\_\_

В модуле может не быть кода вне функций. Если мы запустим такой файл как программу, то ничего не будет выполняться (только определения функций).

**Модуль doctest - выполнение тестовых примеров из описаний функций**

Разберем подробнее код

if \_\_name\_\_ == "\_\_main\_\_":

import doctest

doctest.testmod()

(Вставка из Саммерфильда:)

Функция doctest.testmod() с помощью механизма интроспекции Python выявляет все функции в модуле и их строки документирования, после чего пытается выполнить все фрагменты программного кода, которые приводятся в строках документирования. При запуске модуля таким способом вывод на экране появится только при наличии ошибок. Сначала это может привести в замешательство, так как создается впечатление, будто вообще ничего не происходит; но если интерпретатору передать ключ командной строки –v, на экране может появиться примерно следующее:

Trying:

is\_balanced("(Python (is (not (lisp))))")

Expecting:

True

ok

...

Trying:

simplify(" disemvoweled ", delete="aeiou")

Expecting:

'dsmvwld'

ok

4 items passed all tests:

3 tests in \_\_main\_\_

5 tests in \_\_main\_\_.is\_balanced

3 tests in \_\_main\_\_.shorten

4 tests in \_\_main\_\_.simplify

15 tests in 4 items.

15 passed and 0 failed.

Test passed.

Мы использовали многоточия, чтобы показать, что было опущено множество строк. Если в модуле имеются функции (или классы, или методы), не имеющие тестов, при запуске интерпретатора с ключом –v они будут перечислены. Обратите внимание, что модуль doctest обнаружил тесты как в строке документирования модуля, так и в строках до кументирования функций. Примеры в строках документирования, которые могут выполняться как тесты, называют доктестами (doctests). Обратите внимание, что при написании доктестов мы вызываем функцию simplify(), не используя полное квалифицированное имя (поскольку доктесты находятся непосредственно в самом модуле). За пределами модуля, после выполнения инструкции import TextUtil[?](http://acm.mipt.ru/twiki/bin/edit/Cintro/TextUtil?topicparent=Cintro.PythonModules), мы должны использовать квалифицированные имена, например, TextUtil[?](http://acm.mipt.ru/twiki/bin/edit/Cintro/TextUtil?topicparent=Cintro.PythonModules).is\_balanced().

**Как сделать пакет доступным другим программам?**

Несколько вариантов:

Поместите файл TextUtils[?](http://acm.mipt.ru/twiki/bin/edit/Cintro/TextUtils?topicparent=Cintro.PythonModules).py в директорию

* где находится программа.
  + Плохо. А если нужно его использовать в нескольких разных местах? Раскладывать линки?
* в директорию site-packages , где **установлен питон**, обычно это:
  + C:\Python30\Lib\site-packages - Windows
  + ... - Linux, MAC
  + Не хорошо. Пользователь без полномочия администратора это не может сделать. Обновляем питон - сносим пакеты?
* в локальную директорию **пользователя** site-packages , обычно это:
  + %APPDATA%/Python/Python30/site-packages - Windows
  + ~/.local/lib/python3.0/site-packages - Linux, MAC
  + можно поставить локально одним пользователем
* в любую директорию. Путь к этой директории добавляем в переменную PYTHONPATH
  + можно поставить локально одним пользователем или для всех пользователей
  + echo %PYTHONPATH% - посмотреть содержимое переменной, Windows   
    PYTHONPATH=%PYTHONPATH%;c:\python20\lib
  + echo $PYTHONPATH - посмотреть содержимое переменной, Linux, Mac  
    set PYTHONPATH=$PYTHONPATH:/usr/local/lib/python
* *на лету* изменяем значение переменной sys.path

**Переменная \_\_name\_\_. Запуск модуля как модуля и как программы.**

Если мы импортируем модуль то переменная \_\_name\_\_ этого модуля равна имени модуля (без .py).

Если мы запускаем файл как программу, то переменная \_\_name\_\_ принимает значение "\_\_main\_\_"

Для чего нужно это знание? Мы можем, запуская файл *только* как отдельную программу, протестировать его функции. Полезно для отладки. Смотри последние строки в

Сохраните этот код в файле t1.py

if \_\_name\_\_ == '\_\_main\_\_':

print('This program is being run by itself, \_\_name\_\_ = ', \_\_name\_\_)

else:

print('I am being imported from another module, my ***name*** =', \_\_name\_\_)

Запустим файл, как отдельную программу:

$python3 t1.py

This program is being run by itself, \_\_name\_\_ = \_\_main\_\_

Импортируем этот модуль.

$ python3

Python 3.6.1 (default, Mar 24 2017, 12:50:34) .....

>>> import t1

I am being imported from another module, my \_\_name\_\_ = t1

**Функция dir()**

(Вставка из A bit of python, глава 11 Модули)

Вы можете использовать встроенную функцию dir, чтобы получить список идентификаторов, которые объект определяет. Так в число идентификаторов модуля входят функции, классы и переменные, определённые в этом модуле.

Когда вы передаёте функции dir() имя модуля, она возвращает список имён, определённых в этом модуле. Если никакого аргумента не передавать, она вернёт список имён, определённых в текущем модуле.

$ python3

>>> import sys # получим список атрибутов модуля 'sys'

>>> dir(sys)

['\_\_displayhook\_\_', '\_\_doc\_\_', '\_\_excepthook\_\_', '\_\_name\_\_', '\_\_package\_\_', '\_\_s

tderr\_\_', '\_\_stdin\_\_', '\_\_stdout\_\_', '\_clear\_type\_cache', '\_compact\_freelists',

'\_current\_frames', '\_getframe', 'api\_version', 'argv', 'builtin\_module\_names', '

byteorder', 'call\_tracing', 'callstats', 'copyright', 'displayhook', 'dllhandle'

, 'dont\_write\_bytecode', 'exc\_info', 'excepthook', 'exec\_prefix', 'executable',

'exit', 'flags', 'float\_info', 'getcheckinterval', 'getdefaultencoding', 'getfil

esystemencoding', 'getprofile', 'getrecursionlimit', 'getrefcount', 'getsizeof',

'gettrace', 'getwindowsversion', 'hexversion', 'intern', 'maxsize', 'maxunicode

', 'meta\_path', 'modules', 'path', 'path\_hooks', 'path\_importer\_cache', 'platfor

m', 'prefix', 'ps1', 'ps2', 'setcheckinterval', 'setprofile', 'setrecursionlimit

', 'settrace', 'stderr', 'stdin', 'stdout', 'subversion', 'version', 'version\_in

fo', 'warnoptions', 'winver']

>>> dir() # получим список атрибутов текущего модуля

['\_\_builtins\_\_', '\_\_doc\_\_', '\_\_name\_\_', '\_\_package\_\_', 'sys']

>>> a = 5 # создадим новую переменную 'a'

>>> dir()

['\_\_builtins\_\_', '\_\_doc\_\_', '\_\_name\_\_', '\_\_package\_\_', 'a', 'sys']

>>> del a # удалим имя 'a'

>>> dir()

['\_\_builtins\_\_', '\_\_doc\_\_', '\_\_name\_\_', '\_\_package\_\_', 'sys']

>>>

Как это работает?

Сперва мы видим результат применения dir к импортированному модулю sys. Видим огромный список атрибутов, содержащихся в нём.

Затем мы вызываем функцию dir, не передавая ей параметров. По умолчанию, она возвращает список атрибутов текущего модуля. Обратите внимание, что список импортированных модулей также входит туда.

Чтобы пронаблюдать за действием dir, мы определяем новую переменную a и присваиваем ей значение, а затем снова вызываем dir. Видим, что в полученном списке появилось дополнительное значение. Удалим переменную/атрибут из текущего модуля при помощи оператора del, и изменения вновь отобразятся на выводе функции dir.

Замечание по поводу del: этот оператор используется для удаления переменной/имени, и после его выполнения, в данном случае – del a, к переменной a больше невозможно обратиться – её как будто никогда и не было.

Обратите внимание, что функция dir() работает для любого объекта. Например, выполните dir('print'), чтобы увидеть атрибуты функции print, или dir(str), чтобы увидеть атрибуты класса str.

**Относительные пути в import**

[PEP-328](https://docs.python.org/2.5/whatsnew/pep-328.html)

Есть пакет pkg, состоящий из файлов:

pkg/

pkg/\_\_init\_\_.py

pkg/main.py

pkg/string.py

pkg/sub1

pkg/sub1/\_\_init\_\_.py # каждый уровень вложенности - пишем свой \_\_init\_\_.py

pkg/sub1/hello.py

Как в файле main.py импортировать файл string.py?

Можно написать import string

Но тогда вы можете перепутать имя модуля со стандартным модулем string.

Чтобы явно указать, что хотите именно string.py в той же директории, используйте относительную адресацию. Начинается с точки.

# Import names from pkg.string

from .string import name1, name2

# Import pkg.string

from . import string

Знак .. обозначает "директорией выше".

В модуле A.B.C напишем код:

from . import D # Imports A.B.D

from .. import E # Imports A.E

from ..F import G # Imports A.F.G

**Запуск модуля из командной строки (ключ -m), файл \_\_main\_\_.py**

* Для запуска файла как модуля, а не как программы:
  + пишут ключ python -m (модуль)
  + пишут имя файла БЕЗ расширения .py
* Чтобы запустить пакет из командной строки, в него помещают файл \_\_main\_\_.py   
  При запуске python -m директория выполняется именно \_\_main\_\_.py из этой директории.

**C-модуль**

[Intermediate Python, C-расширения](https://lancelote.gitbooks.io/intermediate-python/content/book/python_c_extension.html)

**Задачи**

**Задача 0**

Напишите свою реализацию функции is\_balanced из файла TestUtils[?](http://acm.mipt.ru/twiki/bin/edit/Cintro/TestUtils?topicparent=Cintro.PythonModules).py, добавляя очередную открывающую скобку в стек (на основе списка) и убирающую верхнюю открывающую скобку из стека, при получении закрывающей скобки.

**Задача 1**

Файл t1.py должен запускаться как программа python t1.py и как модуль python -m t1

**Задача 2**

Пакет pkg должен запускаться из командной строки python -m pkg

**Задача 3**

В модуле pkg/sub1/hello.py использовать функцию из модуля pkg/string.py

**Задача 4**

Для создания задач в системе ejudge написать:

1) пакет scripts/testgen.py, который по единому файлу тестов в формате

input1

---

output1

===

input2

---

output2

===

Создает директорию tests в директории с задачей и там создает нужные файлы 001.dat, 001.ans, 002.dat, 002.ans

Входные и выходные данные могут быть на нескольких строках.

Дополнительно: выходные данные можно задать как ? и тогда они должны быть созданы по решению. Решение можно класть в виде файла имязадачи\_solution.py

2) Взять пример задачи на принадлежность точки области и сделать свою задачу на принадлежность точки области. В задаче должно быть:

* ровно 23 теста, которые должны выявлять типичные ошибки при решении задачи;
* решение на С через float на котором проходят все тесты;
* решение на С через double на котором проходят все тесты;
* решение на С через int на котором НЕ проходят некоторые тесты тесты;
* решение на С через float и int на котором НЕ проходят некоторые тесты тесты;
* программа для рисования картинки - графиков с заливкой и легендой.

Пакет из п.1 лежит на том же уровне, что и директория с задачей:

funk\_kr

+- scripts

+- testgen.py - генератор тестов

+- func\_A - директория с задачей

+- func\_A.xml - условие задачи на функции

+- if\_A.xml - условие задачи на if

+- func\_A\_begin.c - часть файла, которая прикрепляется системой к решению задачи на функцию

+- func\_A\_solution.c - решение задачи на функцию

+- if\_A\_solution.c - решение задачи на if через float (проходит все тесты)

+- if\_A\_double\_solution.c - решение задачи на if через double (проходит все тесты)

+- if\_A\_int\_fail.c - решение задачи на if через int (часть тестов не проходит)

+- if\_A\_floatint\_fail.c - решение задачи на if через float и int (часть тестов не проходит)

+- test.txt - единый файл тестов для генерации

+- make\_pict.py - программа на питоне для рисования картинки

+- tests - директория с тестами

+- 001.dat - input 1-го теста

+- 001.ans - эталонный output 1-го теста

+- 002.dat

+- 002.dat

+- img - директория с картинками (создается)

+- func\_A.png - картинка (создается)

* При запуске с ключом -d файл make\_pict.py должен рисовать график, область и наносить все точки:
  + красным - лежат вне области (NO)
  + зеленым - лежат в области (YES)
  + (дополнительно) голубым - лежат в области, (YES), ответ сгенерен программой-решением
  + (дополнительно) оранжевым - лежат вне области, (NO), ответ сгенерен программой-решением

Скопировать файлы примера к себе в текущую директорию