Компьютерная математика II

Дисциплина для студентов 1-го курса специальности «Компьютерная математика и системный анализ» ММФ БГУ

Тема 7. Модуль в Python

доц. Лаврова О.А., кафедра дифференциальных уравнений и системного анализа (ауд. 329)

апрель, 2024

7.1 Понятие модуля, пакета, расширения

Модуль в Python, который не является модулем верхнего уровня, — это текстовый файл, который содержит операторы явных присваиваний, определения функций и определения классов.

Модуль предназначен для группирования инструментов/функциональности в отдельном файле с целью дальнейшего *многократного использования* его содержимого в различных программах. Использование модуля в других модулях осуществляется посредством импортирования модуля.

При именовании модулей рекомендовано следовать правилам именования переменных. Например, имя модуля рекомендовано начинать с буквы нижнего регистра.

Модули могут быть написаны как на языке Python, так и на других языках программирования.

Модуль, написанный на Python, имеет расширение *.py*.

Модули, написанные на Python, транслируются в *байт-код*.

Байт-код -- низкоуровневое и независимое от платформы представление исходного кода. Байт-код не является машинным кодом (инструкции для процессора). Это представление специфично для Python.

Инструкции байт-кода затем выполняются *интерпретатором* **виртуальной машиной Python** (Python Virtual Machine, PVM) для различных реализаций Python: CPython, PyPy, Jython, Pyston, Pyjion, Cinder, GraalIPython и др.

Модули Python легко создавать. Для этого операторы Python (например, операторы присваивания, операторы `def`, `class`) нужно поместить в отдельный файл. Модули — это файлы программного кода Python без специального синтаксиса.

Расширение — это модуль, написанный и скомпилированный НЕ на Python, а, например, на C, Java, C#. Модуль-расширение не транслируется в байт-код и выполняется в интерпретаторе со скоростью кода, скомпилированного НЕ на Python. Расширения имен модулей: `.so`, `.sl`, `.dll`, `.pyd`.

Пакет — это каталог, в котором хранятся модули.

Стандартная библиотека Python

Стандартная библиотека — это большая коллекция модулей и пакетов, которая поставляется вместе с базовым языком Python. Стандартная библиотека состоит из более чем 300 модулей.

Программирование прикладных задач осуществляется с применением инструментов из *стандартной библиотеки*.

Модули из *стандартной библиотеки*, которые мы применяли/упоминали: `builtins`, `decimal`, `fractions`, `math`, `cmath`, `random`, `re`, `functools`, `itertools`, `sys`.

Модули НЕ из *стандартной библиотеки*, которые мы применяли/упоминали: `sympy`.

Расширения, которые мы применяем: `numpy`.

Пакеты, которые мы применяли/упоминали: `matplotlib` (модули `pyplot`, `animation`), `scipy` (модуль `constants`).

Модульная структура программы на Python

Программа на Python — это система, элементами которой являются модули. Код программы всегда находится внутри какого-то модуля. **Программа на Python имеет модульную структуру.**

Один из модулей предназначен быть главным файлом либо файлом верхнего уровня, или *сценарием* — файлом, запускаемым для старта программы, которая выполняется строка за строкой обычным образом. Модуль верхнего уровня является основным документом программы.

Блокнот *Jupyter Notebook* всегда является модулем верхнего уровня.

Модуль верхнего уровня использует инструменты, определенные в других модулях, а модули могут применять инструменты, определенные в других модулях. Цепочки импортирования модулей могут иметь любую желаемую глубину.

7.2 Импортирование модуля

Инструменты модуля — это объекты, которые создаются на верхнем уровне модуля при его выполнении. Чаще всего объектами модуля являются объекты встроенных типов, объекты функций, объекты классов. Для использования в программе инструментов модуля его необходимо *импортировать*.

Доступ к объектам, созданным внутри модуля, будет осуществляться через переменные, которые ссылаются на объекты в коде модуля. Переменные модуля определяются явными присваиваниями с помощью оператора присваивания (`=`) или неявными присваиваниями при определении функций (оператор `def`) и классов (оператор `class`).

`import` и `from/import` — это операторы импортирования модуля. Оператор `import` импортирует ВСЕ инструменты модуля. Оператор `from/import` импортирует НЕКОТОРЫЕ инструменты из модуля.

Операторы импортирования связывают модули в программную систему *во время выполнения программы*.

При выполнении оператора `import <имя_файла>` осуществляется *загрузка* файла с именем `<имя_файла>` (**Этап 1**), *создание* объекта модуля (**Этап 2**) и *неявное присваивание* объекта модуля переменной `<имя_файла>` (**Этап 3**).

Загрузка модуля (**Этап 1**) состоит из трех шагов:

- **Этап 1.1:** поиск файла
- **Этап 1.2:** трансляция кода модуля в байт-код (при необходимости)
- **Этап 1.3:** ВЫПОЛНЕНИЕ байт-кода интерпретатором

Имена каталогов для поиска файлов хранятся в атрибуте `path` модуля `sys` из стандартной библиотеки. С помощью атрибута `path` можно просмотреть пути поиска для импортируемых файлов

In [1]:
```
import sys
sys.path
```

Out[1]:
```
['C:\\Users\\Olga\\Documents\\Python Scripts\\MyNotebooks\\Materials2024',
 'C:\\Users\\Olga\\anaconda3\\python311.zip',
 'C:\\Users\\Olga\\anaconda3\\DLLs',
 'C:\\Users\\Olga\\anaconda3\\Lib',
 'C:\\Users\\Olga\\anaconda3',
 '',
 'C:\\Users\\Olga\\anaconda3\\Lib\\site-packages',
 'C:\\Users\\Olga\\anaconda3\\Lib\\site-packages\\win32',
 'C:\\Users\\Olga\\anaconda3\\Lib\\site-packages\\win32\\lib',
 'C:\\Users\\Olga\\anaconda3\\Lib\\site-packages\\Pythonwin']
```

Этап 1.2 Трансляция кода при загрузке модуля

Байт-код импортированных модулей сохраняется в файлах с расширением `.pyc` для ускорения будущих операций импортирования. Файлы байт-кода хранятся в отдельном подкаталоге `__pycache__` и содержат в своих именах название Python (CPython, Jython) и версию Python во избежание конфликтов и повторной трансляции в байт-код. Например, для модуля `test.py` в подкаталоге `__pycache__` создается файл байт-кода `test.cpython-39.pyc`

Байт-код для модуля верхнего уровня не сохраняется на диске.

Повторная трансляция в байт-код не выполняется, если Python обнаруживает в пути поиска файл байт-кода `.pyc`, который не старше соответствующего файла исходного кода `.py` и который создан текущей версией Python.

Если в процессе импортирования модуля, Python не находит файл исходного кода `.py` в пути поиска, но находит файл байт-кода `.pyc`, то файл байт-кода загружается напрямую. Это позволяет предоставлять модули без исходного кода, а только в виде файлов байт-кода для их использования другими модулями и программами.

Объекты импортированных модулей хранятся в атрибуте `modules` модуля `sys`

In [2]:
```
type(sys.modules), len(sys.modules)
```

Out[2]:
```
(dict, 958)
```

Этап 1.3. Выполнение байт-кода при загрузке

`<имя_файла>` при выполнении оператора импортирования `import <имя_файла>` служит двум целям: оно идентифицирует файл для загрузки и определяет переменную для ссылки на объект модуля. `<имя_файла>` необходимо записывать без указания пути к файлу и расширения файла

In [3]:
```
import math
```

Переменная `math` ссылается на объект модуля

In [4]:
```
type(math), type(math).__bases__
```

Out[4]:
```
(module, (object,))
```

В операторе `import` можно указать несколько модулей для импортирования. Модули будут импортироваться последовательно

```
import <имя_файла1>,...,<имя_файлаN>
```

In [5]:
```
import decimal, fractions
```

In [6]:
```
decimal.Decimal(1/3), fractions.Fraction(1,3)
```

Out[6]:
```
(Decimal('0.33333333333333331482961625624739099293947219848632812 5'),
 Fraction(1, 3))
```

При выполнении оператора `import` с *расширением* `as` в виде

```
import <имя_файла> as <имя_объекта_модуля>
```

осуществляется *загрузка* модуля с именем `<имя_файла>` , *создание* объекта модуля и *неявное присваивание* объекта модуля переменной `<имя_объекта_модуля>`

In [7]:
```
import matplotlib.pyplot as plt
```

In [8]:
```
?plt
```

```
Type:        module
String form: <module 'matplotlib.pyplot' from 'C:\\Users\\Olga\\anaconda3\\Lib\\sit
e-packages\\matplotlib\\pyplot.py'>
File:        c:\users\olga\anaconda3\lib\site-packages\matplotlib\pyplot.py
Docstring:
`matplotlib.pyplot` is a state-based interface to matplotlib. It provides
an implicit,  MATLAB-like, way of plotting.  It also opens figures on your
screen, and acts as the figure GUI manager.

pyplot is mainly intended for interactive plots and simple cases of
programmatic plot generation::

    import numpy as np
    import matplotlib.pyplot as plt

    x = np.arange(0, 5, 0.1)
    y = np.sin(x)
    plt.plot(x, y)

The explicit object-oriented API is recommended for complex plots, though
pyplot is still usually used to create the figure and often the axes in the
figure. See `.pyplot.figure`, `.pyplot.subplots`, and
`.pyplot.subplot_mosaic` to create figures, and
:doc:`Axes API </api/axes_api>` for the plotting methods on an Axes::

    import numpy as np
    import matplotlib.pyplot as plt

    x = np.arange(0, 5, 0.1)
    y = np.sin(x)
    fig, ax = plt.subplots()
    ax.plot(x, y)


See :ref:`api_interfaces` for an explanation of the tradeoffs between the
implicit and explicit interfaces.
```

При выполнении оператора `from/import` :

```
from <имя_файла> import <переменная1>, <переменная2>, ..., <переменнаяN>
```

осуществляется *загрузка* файла (**Этап 1**) с именем `<имя_файла>` с выполнением ВСЕГО кода модуля и *создание* только тех объектов модуля, которые связаны с переменными `<var1>` , `<var2>` , ..., `<verN>` . Переменные `<ver1>` , `<ver2>` , ..., `<verN>` создаются в *пространстве имен импортирующего модуля*. Доступ к именам импортированного модуля осуществляется напрямую без указания объекта модуля, так как объект модуля с именем `<имя_файла>` не создается (**Этап 2 и Этап 3 отсутствуют**).

Использование оператора `from/import` вместо оператора `import` не дает выигрыша в использовании памяти!

При выполнении оператора `from/import` с *расширением* `as` в виде

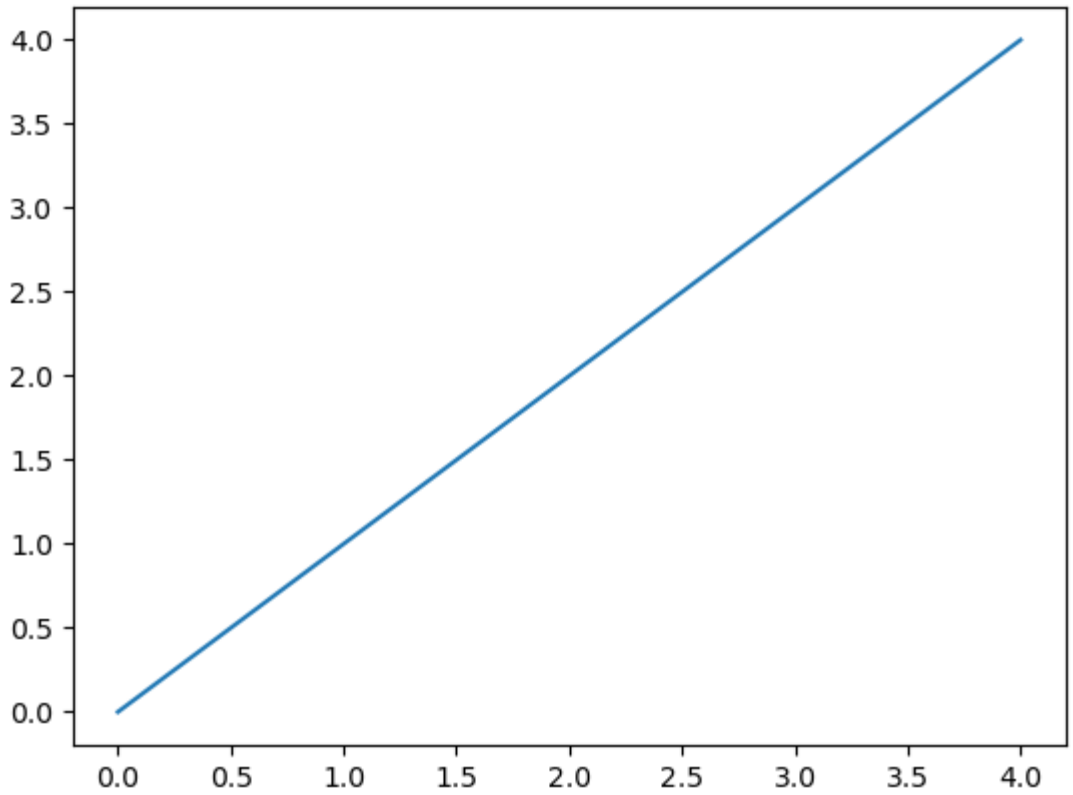
```
from <имя_файла> import <переменная1> as <name1>, ..., <переменнаяN> as <nameN>
```

осуществляется *загрузка* файла (**Этап 1**) с именем `<имя_файла>` с выполнением ВСЕГО кода модуля и *создание* только тех объектов модуля, которые связаны с переменными `<var1>`, `<var2>`, ..., `<varN>`. В пространстве имен импортирующего модуля создаются переменные `<name1>`, `<name2>`, ..., `<nameN>` для ссылки на импортированные объекты. **Этап 2 и Этап 3 отсутствуют**.

In [9]:
```
from matplotlib.pyplot import plot as PLOT, figure as FIGURE
```

In [10]:
```
FIGURE()
PLOT(range(5))
```

Out[10]:
```
[<matplotlib.lines.Line2D at 0x2a7fc073410>]
```



При выполнении оператора `from/import` с использованием `*`:

```
from <имя_файла> import *
```

осуществляется *загрузка* файла (**Этап 1**) с именем `<имя файла>` и *создание* ВСЕХ объектов из загружаемого модуля. ВСЕ переменные импортируемого модуля создаются в *пространстве имен импортирующего модуля*. Доступ к переменным импортированного модуля осуществляется напрямую без указания объекта модуля, так как объект модуля с именем `<имя_файла>` не создается (**Этап 2 и Этап 3 отсутствуют**).

`from <имя_файла> import *` не рекомендовано использовать в Python! Импортированная переменная *конфликтует* с переменной импортирующего модуля, если обе переменные имеют одинаковые имена.

Импортирование модуля является затратной операцией. По умолчанию **Python импортирует модуль только один раз в программе**. Повторные выполнения оператора `import` или `from/import` используют объект уже загруженного модуля и не перезагружают модуль. Повторного выполнения кода модуля не происходит.

Доступ к переменным импортированного модуля

При выполнении оператора `import <имя_файла>` осуществляется не только *загрузка* модуля (**Этап 1**), но и *создание* объекта модуля с именем `<имя_файла>` (**Этап 2 и Этап 3**). Модуль, как внешний файл, становится объектом модуля, расположенным в памяти.

Переменная, созданная на верхнем уровне модуля явным или неявным присваиванием, имеет *глобальную область видимости* и является *глобальной переменной* внутри модуля.

После импортирования модуля с помощью оператора `import` переменные из глобальной области видимости модуля становятся переменными в *пространстве имен объекта модуля* и доступны в коде через атрибуты объекта модуля

In [11]:
```
math.e, math.exp(1)
```

Out[11]:
```
(2.718281828459045, 2.718281828459045)
```

Каждый модуль, импортированный с помощью оператора `import`, ассоциируется с *изолированным* пространством имен. Переменные одного модуля не могут конфликтовать с переменными другого модуля, даже если их имена одинаковые. Обращение к переменным разных модулей осуществляется через явное указание имен объектов модулей, которым переменные принадлежат

In [12]:
```
import module1, module2
module1.x, module2.x
```

Out[12]:
```
('module1', 'module2')
```

Переменные, импортированные с помощью оператора `from/import`, переносятся внутрь пространства имен импортирующего модуля. Так как при выполнении оператора `from/import` не создается объект модуля, то может возникнуть конфликт имен для переменных импортированного и импортирующего модулей. *Это не соответствует идее модульной структуры программы с изолированными пространствами имен для каждого модуля.*

Так как для модуля, импортированного с помощью оператора `from/import`, не создается объект модуля (**Этап 2 и Этап 3 отсутствуют**), то его нельзя перезагрузить.

При выполнении оператора `from/import` с использованием `*`:

```
from <имя_файла> import *
```

объекты модуля, которые связаны с переменными вида `<_variable>`, НЕ создаются, а переменная `<_variable>` не добавляется в пространство имен импортирующего модуля. Одиночное подчеркивание в начале переменной не является объявлением *закрытой* переменной. При выполнении оператора `import` создаются ВСЕ объекты импортируемого модуля внутри объекта модуля.

Пространство имен модуля можно просмотреть, используя функцию `dir` для объекта модуля. Результатом является список

In [13]:
```
print(dir(module1))
```

```
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__
package__', '__spec__', 'x']
```

Пространство имен модуля можно просмотреть, используя атрибут `__dict__` для объекта модуля. Результатом является словарный объект

In [14]:
```
# module1.__dict__
```

Атрибут объекта модуля `__name__` возвращает имя файла для объекта модуля в виде строки или строку `'__main__'`, если модуль выполняется как файл верхнего уровня, а не импортируется в другом модуле.

In [15]:
```
plt.__name__, math.__name__
```

Out[15]:
```
('matplotlib.pyplot', 'math')
```

Проверка атрибута `__name__` в программе вида

```
if __name__ == '__main__':
```

может использоваться для размещения кода тестов в том же файле, где расположены тестируемые инструменты. В этом случае *код самотестирования* для проверки корректности и эффективности реализации инструментов модуля *располагают в конце модуля*. Код самотестирования будет выполняться только в том случае, когда модуль будет запущен как модуль верхнего уровня.

In [16]:
```
def func(): pass

# код самотестирования
if __name__ == '__main__':
    print("Test code")
```

```
Test code
```

Размещение кода самотестирования в конце модуля в операторе `if` с условием `__name__ == '__main__'` является часто применяемым и простым способом тестирования модуля в Python.

7.3 Перезагрузка модуля

Если на стадии разработки код модуля изменен и необходимо использовать новую версию модуля, не останавливая и не перезапуская Python, то импортированный модуль нужно *перезагрузить*. Прежде чем модуль можно будет перезагрузить, его необходимо импортировать с помощью оператора `import` .

Для перезагрузки импортированного модуля можно использовать функцию `reload` из модуля `importlib`

In [17]:
```
import importlib
?importlib.reload
```

```
Signature: importlib.reload(module)
Docstring:
Reload the module and return it.

The module must have been successfully imported before.
File:      c:\users\olga\anaconda3\lib\importlib\__init__.py
Type:      function
```

In [18]:
```
module1.x
```

Out[18]:
```
'module1'
```

Изменим значение для переменной `x` внутри модуля `module1` : `x = 'module1New'`

In [21]:
```
print(module1.x)        # модуль не перезагружен
importlib.reload(module1)
print(module1.x)        # модуль перезагружен
```

```
module1
module1New
```

Функция `reload` изменяет атрибуты объекта модуля на месте, не удаляя созданный ранее объект модуля и не создавая новый объект модуля

In [22]:
```
print(id(math))
importlib.reload(math); id(math)
```

Out[22]:
```
2920413836640
2920413836640
```

7.4 Пакет

Хранение модулей может быть организовано по каталогам. Каталог, в котором содержатся модули, называется **пакетом**. Операторы импортирования `import` и `from/import` позволяют импортировать модуль из пакета

```
import <имя_каталога1>.<имя_каталога2>. ... .<имя_каталогаN>.<имя модуля>
from <имя_каталога1>.<имя_каталога2>. ... .<имя_каталогаN> import <имя модуля>
```

Путь, разделенный точками, соответствует пути в иерархии каталогов на диске к модулю с именем `<имя модуля>.`

In [23]:
```
import scipy.constants as const
from scipy import constants as const1
const, const1
```

Out[23]:
```
(<module 'scipy.constants' from 'C:\\Users\\Olga\\anaconda3\\Lib\\site-packages\\scipy\\constants\\__init__.py'>,
 <module 'scipy.constants' from 'C:\\Users\\Olga\\anaconda3\\Lib\\site-packages\\scipy\\constants\\__init__.py'>)
```

Обратите внимание на путь к модулю `constants` .

Если модули организованы в подкаталоги по областям функциональности, тогда импортирование пакетов сделает более очевидной роль, исполняемую импортируемым модулем, что в итоге улучшает читабельность кода.