

# Язык программирования



## Лекция № 2

Владимир Владимирович Руцкий  
[rutsky.vladimir@gmail.com](mailto:rutsky.vladimir@gmail.com)



# План занятия

- Повторение
- Модули, области видимости, полезные функции
- Практика

# Повторение

# Модули

- Программы на Python пишутся внутри \*.py файлов — **модулей**
- Модули можно выполнять (`python.exe hello.py`):

```
def my_hello():
    print("Hello, world!")

if __name__ == "__main__":
    my_hello()
```

- Объекты из модулей можно использовать в других модулях (или в интерактивном режиме) с помощью команды `import`:

```
>>> import hello
>>> hello.my_hello()
'Hello, World!'
>>>
```

# Стандартная библиотека (1/2)

- Стандартная библиотека Python состоит из большого числа модулей: <http://docs.python.org/3/library/index.html>
  - Работа со строками
  - Работа с двоичными данными
  - Структуры данных
  - Математические функции и типы
  - Объекты для функционального программирования
  - Работа с файлами и директориями
  - СерIALIZАЦИЯ и сохранение данных, БД
  - Сжатие данных и работа с архивами
  - Работа с файлами определённых типов
  - Криптографические функции
  - Работа с ОС
  - Параллельное выполнение кода
  - ...

# Стандартная библиотека (2/2)

- продолжение:
  - Межпроцессорное взаимодействие
  - Работа с типами Internet
  - Работа с HTML/XML
  - Интернет протоколы
  - Мультимедиа
  - Интернационализация
  - Фреймворки для программ
  - Графический интерфейс с помощью Tk
  - Средства для разработки
  - Средства отладки и профилирования

# Модуль random

```
>>> import random
>>> random.random() # Случайное float число в диапазоне [0, 1)
0.5827532718821743
>>> random.randint(15, 30) # Случайное int число в диапазоне [15, 30]
29
>>> random.uniform(3.5, 10.0) # Случайное float число из диапазона [3.5, 10.0]
7.332874018578613
>>> # Есть некоторые популярные распределения: равномерное, Бета, экспоненциальное,
... # Гамма, Гаусса, Нормальное, Парето
... random.gauss(mu=0, sigma=1)
0.512533053672749
>>> a = ['Alice', 'Bob', 'Valery']
>>> random.choice(a) # Вернёт произвольный элемент последовательности
'Valery'
>>> random.shuffle(a) # Случайным образом перемешает последовательность
>>> a
['Alice', 'Valery', 'Bob']
>>>
```

# Способы импортирования (1/2)

```
>>> # При импортировании модуля его можно "переименовать"
... import random as rnd
>>> rnd.random()
0.5827532718821743
>>> # Можно импортировать только некоторые имена в локальную область видимости
... from random import choice
>>> choice(['a', 'b', 'c', 'd'])
'b'
>>> # Можно импортировать имя из модуля, и переименовать его:
... from random import uniform as unf
>>> unf(-3, 3)
-2.519376744749322
>>> # Можно импортировать несколько модулей в одной команде (не рекомендуется)
... import random, os, sys
>>> sys.platform
'linux'
>>> # Можно импортировать несколько имён из модуля:
... from random import gammavariate as G, choice, uniform as unf
>>> # При импортировании имён из модулей в текущей области видимости создаются
... # ссылки на объекты из импортируемых модулей, копирования не происходит:
... rnd is random
True
>>> rnd.choice is choice
True
>>>
```

# Способы импортирования (2/2)

```
>>> # Можно импортировать все имена из модуля в текущую область видимости:  
... from random import *  
>>> choice(['a', 'b', 'c', 'd'])  
'b'  
>>> random()  
0.5827532718821743  
>>> dir()  
[..., 'betavariate', 'choice', 'expovariate', 'gammavariate', 'gauss', 'getrandbits', ...]  
>>>
```

# Скрытие имён в модулях (1/2)

from ... import \* не импортирует имена, начинающиеся с подчёркивания — в Python принято называть "скрытые" функции и имена с подчёркивания

```
# rndcolors1.py
from random import choice

colors = ["red", "green", "blue"]

def random_color():
    return choice(colors)

>>> from rndcolors1 import *
>>> dir()
[..., 'choice', 'colors', 'random_color']
>>>
```

```
# rndcolors2.py
from random import choice

_colors = ["red", "green", "blue"]

def random_color():
    return random.choice(_colors)

>>> from rndcolors2 import *
>>> dir()
[..., 'choice', 'random_color']
```

# Скрытие имён в модулях (2/2)

Если в модуле объявлена глобальная переменная `__all__`, то с помощью `from ... import *` из этого модуля будут импортироваться только имена, перечисленные в списке или кортеже `__all__`

```
# rndcolors3.py
from random import choice

__all__ = ["random_color"]

colors = ["red", "green", "blue"]

def random_color():
    return random.choice(colors)

>>> from rndcolors3 import *
>>> dir()
[..., 'random_color']
```

# Модуль [builtins](#)

```
>>> # Встроенные функции находятся в специальном модуле `builtins'  
... import builtins  
>>> builtins.print('Test!')  
Test!  
>>> builtins.print is print  
True  
>>>
```

# Пакеты

Модули можно объединять в пакеты:

```
sound/          Пакет верхнего уровня
    __init__.py      Инициализация пакета работы со звуком (sound)
    formats/        Подпакет для конвертирования форматов файлов
        __init__.py
        wavread.py     (чтение wav)
        wavwrite.py    (запись wav)
    effects/        Подпакет для звуковых эффектов
        __init__.py
        echo.py        (эхо)
        surround.py   (фон)
        reverse.py    (обращение)
```

Использование:

```
# Импортируем модуль sound/effects/echo.py
import sound.effects.echo
sound.effects.echo.echofilter(input, output, delay=0.7, atten=4)
```

ИЛИ:

```
from sound.effects import echo
echo.echofilter(input, output, delay=0.7, atten=4)
```

# Файл `__init__.py`

- Пакет — директория, содержащая модули и/или пакеты
- Модуль можно превратить в пакет (и наоборот):
  - `sum` как модуль:

`sum.py:`

```
def my_sum(a, b):
    return a + b
```

- `sum` как пакет:

`sum/__init__.py:`

```
def my_sum(a, b):
    return a + b
```

- Использование одинаковое:

```
>>> import sum
>>> sum.my_sum(10, 20)
30
>>>
```

# Функция dir()

```
1 >>> # Функция dir() позволяет получить список членов (атрибутов) объекта
2 ... a = 30
3 >>> dir(a)
4 ['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__', ...
5 >>> a.__add__(3)
6 33
7 >>> def f(a, b):
8 ...     """Вычисляет сумму двух чисел"""
9 ...     c = a + b
10 ...    return c
11 ...
12 >>> dir(f)
13 ['__annotations__', '__call__', '__class__', '__closure__', '__code__', ...
14 >>> f.__name__
15 'f'
16 >>> f.__doc__
17 'Вычисляет сумму двух чисел'
18 >>> f.__code__.co_varnames # интроспекция – доступ к внутренней информации об объектах
19 ('a', 'b', 'c')
20 >>> # Вызванная без аргументов, dir() возвращает список доступных имён в локальной
21 ... # области видимости.
22 ... dir()
23 ['__builtins__', '__doc__', '__loader__', '__name__', '__package__', 'a', 'f']
24 >>> __name__
25 '__main__'
26 >>>
```

# Способы выполнения кода на Python (1/2)

Прямые способы:

- Интерактивный режим:

```
>>> a = 10
```

- Запуск модуля:

```
C:\>C:\Python33\python.exe hello.py  
Hello!  
C:\>
```

- Выполнение команд, непосредственно переданных интерпретатору:

```
C:\>C:\Python33\python.exe -c "a = 10; print(a)"  
10  
C:\>
```

- Запуск модуля, доступного в текущем дистрибутиве Python:

```
C:\>C:\Python33\python.exe -m random  
2000 times random  
0.001 sec, avg 0.500839, stddev 0.287432, min 0.000557505, max 0.999779  
2000 times normalvariate  
0.003 sec, avg 0.0308376, stddev 1.02038, min -3.30629, max 3.85466  
...  
C:\>
```

# Способы выполнения кода на Python (2/2)

Косвенные способы — при импортировании модулей:

```
>>> import random # В результате этой команды модуль будет интерпретирован  
>>>
```

module.py:

```
print("This module '__name__' variable is:", __name__)
```

При прямом способе выполнения команды, вводимые в интерактивном режиме, или из интерпретируемого модуля, или из аргументов к `python.exe -c`, выполняются в виртуальном модуле с именем `__main__`:

```
C:\>C:\Python33\python.exe module.py  
This module '__name__' variable is: __main__  
C:\>
```

При интерпретировании модуля в результате импортирования (косвенный способ выполнения) `__name__` будет указывать на имя модуля:

```
>>> import module  
This module '__name__' variable is: module
```

# Отладочный код в библиотеках

- С помощью `__name__` можно определить импортирован ли модуль, либо он выполняется напрямую
- Типичное использование в библиотеках — при запуске библиотечного модуля запускать тесты; `even.py`:

```
1 def is_even(number):
2     """Возвращает True, если number – чётный"""
3     if number % 4 == 0: # Ошибка
4         return True
5     return False
6
7 def _test():
8     assert is_even(4)
9     assert not is_even(3)
10    assert is_even(-20)
11    assert is_even(2)
12
13 if __name__ == "__main__":
14     _test()
```

```
C:\>C:\Python33\python.exe -m even.py
Traceback (most recent call last):
  File "module_test.py", line 17, in <module>
    _test()
  File "module_test.py", line 13, in _test
    assert is_even(2)
AssertionError
C:\>
```

# Шаблон программы на Python (1/2)

```
1 # Шаблон программы на Python
2 #
3 # Это комментарий. Любую программу стоит начинать с описания того,
4 # для чего она предназначена.
5
6 # Импортируем необходимые модули
7 import sys
8
9 def main():
10     """Main program function"""
11
12     # sys.argv содержит список аргументов командной строки.
13     # sys.argv[0] хранит имя запущенного скрипта.
14
15     if len(sys.argv) == 1:
16         print("Ошибка! Слишком мало аргументов!")
17         sys.exit(1)
18
19     else:
20         print("Программа", sys.argv[0], "была запущена с аргументами:")
21         for arg in sys.argv[1:]:
22             print(arg)
23
24 if __name__ == "__main__":
25     # Если скрипт запущен как "python.exe template.py",
26     # то это условие будет выполнено и будет вызвана main().
27     # Впоследствии можно будет написать тесты, которые будут
28     # импортировать этот модуль и вызывать функции из него.
29     main()
```

# Шаблон программы на Python (2/2)

```
C:\>C:\Python33\python.exe template.py
```

```
Ошибка! Слишком мало аргументов!
```

```
C:\>C:\Python33\python.exe template.py Питон 1 2 3
```

```
Программа template.py была запущена с аргументами:
```

```
Питон
```

```
1
```

```
2
```

```
3
```

# Модуль `pprint`

- `pprint.pprint(object, stream=None, indent=1, width=80, depth=None)`

"pretty-print". Выводит на экран текстовое представление объекта с отступами и переносами строк

То же, что и просто `print()`, но форматированное

```
>>> john = {"name": "John", "age": 30, "graduate": ["30", "SPBSTRU", "MIT"]}
>>> kate = {"name": "Kate", "age": 28, "graduate": ["85", "SPBSTRU"]}
>>> people = [john, kate]
>>> print(people)
[{'age': 30, 'graduate': ['30', 'SPBSTRU', 'MIT'], 'name': 'John'}, {'age': ...
>>> import pprint
>>> pprint.pprint(people)
[{'age': 30, 'graduate': ['30', 'SPBSTRU', 'MIT'], 'name': 'John'},
 {'age': 28, 'graduate': ['85', 'SPBSTRU'], 'name': 'Kate'}]
>>>
```

- `pprint.pformat(object, indent=1, width=80, depth=None)`

Возвращает строку с результатом, вместо печати на экран

# Имена (переменные) в Python

- В Python имена — это метки, ссылающиеся на объекты в памяти (*связанные с объектами, binded*)
- Чтобы создать новую метку (или переопределить старую) можно использовать:
  - команду присваивания:

```
a = 30
```

- команду определения функции (или класса):

```
def my_sum(x, y):  
    return x + y
```

- команду import:

```
import random
```

# Области видимости

- Код на Python может выполняться:
  - непосредственно внутри модуля
  - внутри функции
- Перед выполнением кода внутри модуля или функции, интерпретатор создаёт *область видимости* для модуля или функции (далее о.в.)
- Область видимости — это словарь пар:
  - (имя метки, объект)
- При создании новой метки она помещается в словарь текущей области видимости

# Область видимости в модуле

```
>>> import pprint
>>> # Содержимое текущей (локальной) области видимости можно получить
... # с помощью locals() (только на чтение)
... pprint.pprint(locals())
{'__builtins__': <module 'builtins' (built-in)>,
 '__doc__': None,
 '__loader__': <class '_frozen_importlib.BuiltinImporter'>,
 '__name__': '__main__',
 '__package__': None,
 'pprint': <module 'pprint' from '/usr/lib/python3.3/pprint.py'>}
>>> # Создаём метку в текущей области видимости:
... a = 10
>>> pprint.pprint(locals())
{'__builtins__': <module 'builtins' (built-in)>,
 '__doc__': None,
 '__loader__': <class '_frozen_importlib.BuiltinImporter'>,
 '__name__': '__main__',
 '__package__': None,
 'a': 10,
 'pprint': <module 'pprint' from '/usr/lib/python3.3/pprint.py'>}
>>> def my_sum(a, b):
...     return a + b
...
>>> pprint.pprint(locals())
{'__builtins__': <module 'builtins' (built-in)>,
 '__doc__': None,
 '__loader__': <class '_frozen_importlib.BuiltinImporter'>,
 '__name__': '__main__',
 '__package__': None,
 'a': 10,
 'my_sum': <function my_sum at 0x7f1ca3dd38c0>,
 'pprint': <module 'pprint' from '/usr/lib/python3.3/pprint.py'>}
>>>
```

# Область видимости в функции

```
>>> import pprint
>>> def my_sum(a, b):
...     # В функции своя локальная область видимости
...     print("my_sum 1:\n", pprint.pformat(locals()), sep='')
...     sum = a + b
...     print("my_sum 2:\n", pprint.pformat(locals()), sep='')
...     return sum
...
>>> pprint.pprint(locals())
{'__builtins__': <module 'builtins' (built-in)>,
 '__doc__': None,
 '__loader__': <class '_frozen_importlib.BuiltinImporter'>,
 '__name__': '__main__',
 '__package__': None,
 'my_sum': <function my_sum at 0x7f1ca3dd38c0>,
 'pprint': <module 'pprint' from '/usr/lib/python3.3/pprint.py'>}
>>> my_sum(10, 20)
my_sum 1:
{'a': 10, 'b': 20}
my_sum 2:
{'a': 10, 'b': 20, 'sum': 30}
30
>>> # Изменение в локальной области my_sum не меняет содержимое области
... # видимости модуля
... pprint.pprint(locals())
{'__builtins__': <module 'builtins' (built-in)>,
 '__doc__': None,
 '__loader__': <class '_frozen_importlib.BuiltinImporter'>,
 '__name__': '__main__',
 '__package__': None,
 'my_sum': <function my_sum at 0x7f1ca3dd38c0>,
 'pprint': <module 'pprint' from '/usr/lib/python3.3/pprint.py'>}
>>>
```

# Поиск имён

- Все метки в Python определены в каком-то модуле
- Для блока кода определена глобальная область видимости — область видимости модуля, в котором этот блок кода был задан
- Когда в каком-то блоке происходит обращение к метке, интерпретатор смотрит в следующие о.в.:
  - локальную о.в. блока
  - внешние о.в. для текущего блока
    - если, например, функция задана внутри функции
  - глобальную о.в. для текущего блока
    - модуль, в котором блок определён
  - встроенную о.в.
    - содержимое модуля **builtins**

# Глобальная область видимости

```
>>> import pprint
>>> # Глобальную область видимости можно получить с помощью globals()
>>> pprint.pprint(globals())
{'__builtins__': <module 'builtins' (built-in)>,
 '__doc__': None,
 '__loader__': <class '_frozen_importlib.BuiltinImporter'>,
 '__name__': '__main__',
 '__package__': None,
 'pprint': <module 'pprint' from '/usr/lib/python3.3/pprint.py'>}
>>> # В блоке кода модуля глобальная о.в. совпадает с локальной
>>> globals() is locals()
True
>>>
```

# Команда global

```
>>> import pprint
>>> V1 = "global V1" # определим глобальные переменные в модуле
>>> V2 = "global V2"
>>> V3 = "global V3"
>>> def func():
...     print("V1 =", V1) # V1 находится из глобальной о.в.
...     V2 = "func() V2" # метка помещается в локальную о.в., перекрывая глоб. V2
...     print("V2 =", V2)
...     global V3           # указываем, что метка V3 – из глобальной о.в.
...     V3 = "changed in func()"
...     print("V3 =", V3)
...
>>> pprint.pprint(locals())
{'V1': 'global V1',
 'V2': 'global V2',
 'V3': 'global V3',
 ...
 'func': <function func at 0x7f1198dc8680>,
>>> func()
V1 = global V1
V2 = func() V2
V3 = changed in func()
>>> pprint.pprint(locals())
{'V1': 'global V1',
 'V2': 'global V2',
 'V3': 'changed in func()',
 ...
 'func': <function func at 0x7f1198dc8680>,
>>>
```

# Функция enumerate()

- `enumerate(iter, start=0)` — "пересчитать"
  - `iter` — последовательность
  - `start` — опциональный начальный индекс
  - возвращает последовательность пар (индекс, элемент `iter`)

```
>>> a = ["A", "B", "C"]
>>> list(enumerate(a))
[(0, 'A'), (1, 'B'), (2, 'C')]
>>> list(enumerate(a, 10))
[(10, 'A'), (11, 'B'), (12, 'C')]
>>> # Удобно использовать для получения индекса при итерации по последовательности
... for idx, val in enumerate(a):
...     print("idx =", idx, "val =", val)
...
idx = 0 val = A
idx = 1 val = B
idx = 2 val = C
>>>
```

# Функция `zip()` (1/2)

- `zip(iter1[, iter2[, ...]])`
  - от англ. «*zip*» — «застёжка молния» — соединить два списка, как зубцы молнии
  - `iter*` — последовательности
  - возвращает последовательность кортежей:
    - (первые элементы последовательностей)
    - (вторые элементы последовательностей)
    - (третьи элементы последовательностей)
    - ...
  - длина результирующей последовательности равна минимальной длине входной последовательности

## Функция zip() (2/2)

```
>>> a = ["A", "B", "C"]
>>> b = ["a", "b", "c"]
>>> list(zip(a, b))
[('A', 'a'), ('B', 'b'), ('C', 'c')]
>>> # длина результирующей последовательности равна длине минимальной входной посл.
... list(zip(range(10), a, b))
[(0, 'A', 'a'), (1, 'B', 'b'), (2, 'C', 'c')]
>>> # Удобно использовать для одновременной итерации по нескольким посл.:
... for x, y in zip(a, b):
...     print(x, y)
...
A a
B b
C c
>>> for idx, (x, y) in enumerate(zip(a, b)):
...     print(idx, "-", x, y)
...
0 - A a
1 - B b
2 - C c
>>> list(zip(a))
[('A',), ('B',), ('C',)]
>>> list(zip(*a))
[('A', 'B', 'C')]
>>> M = [[11, 12, 13], [21, 22, 23], [31, 32, 33]]
>>> list(zip(*M))
[(11, 21, 31), (12, 22, 32), (13, 23, 33)]
>>>
```

## Функция map() (1/2)

- `map(func, iter1[, iter2[, ...]])` — применить `func` к каждому элем. посл.
- `iter*` — последовательности аргументов
- возвращает последовательность:
  - `func(первые элементы последовательностей)`
  - `func(вторые элементы последовательностей)`
  - `func(третьи элементы последовательностей)`
  - `...`
- длина результирующей последовательности равна минимальной длине входной последовательности

## Функция map() (2/2)

```
>>> a = ["A", "B", "C"]
>>> def prefix(s, prefix="prefix_"):
...     return prefix + s
...
>>> # Результат: последовательность [prefix(a[0]), prefix(a[1]), prefix(a[2]), ...]
... list(map(prefix, a))
['prefix_A', 'prefix_B', 'prefix_C']
>>> b = ["zero_", "one_", "two_"]
>>> # Результат: последовательность [prefix(a[0], b[0]), prefix(a[1], b[1]), ...]
... list(map(prefix, a, b))
['zero_A', 'one_B', 'two_C']
>>> list(map(len, b))
[5, 4, 4]
>>>
```

# Функция filter()

- `filter(func, iter)` — вернуть последовательность только из тех элементов `X`, для которых `func(x)` истина
- `func` может быть `None`, эквивалентно `filter(bool, iter)`

```
>>> def long_enough(s):
...     return len(s) > 3
...
>>> a = ["Alice", "Bob", "Valery", "Nicholas", "I"]
>>> list(filter(long_enough, a))
['Alice', 'Valery', 'Nicholas']
>>> list(filter(None, ['', 'a', 0, 1, True, False]))
['a', 1, True]
>>>
```

# Форматирование строк

Документация: <http://docs.python.org/3/library/string.html#formatstrings>

```
>>> "Hello, {0}".format("Peter")
'Hello, Peter'
>>> "Hello, {0} {1}. Or maybe {1} {0}?".format("John", "Smith")
'Hello, John Smith. Or maybe Smith John?'
>>> "{}, {} {}".format(1, 2, 3)
'Hello, 1 2 3'
>>> "Object: {0} - size is {size}, weight is {w}".format("potato", size=10, w="0.1 kg")
'Object: potato - size is 10, weight is 0.1 kg'
>>> "int: {0:d}; hex: {0:x}; oct: {0:o}; bin: {0:b}".format(30)
'int: 30; hex: 1e; oct: 36; bin: 11110'
>>> "{:.10.3}".format(3.141592653589793)
'            3.14'
>>>
```

# Литералы в других системах счисления

```
>>> # В Python можно записывать целочисленные литералы в разных системах счисления
... 0xA36832      # шестнадцатеричная
10709042
>>> 0o655        # восьмеричная
429
>>> 0b00011110 # двоичная
30
>>> # Строковые представления в разных системах счисления:
... hex(10709042)
'0xa36832'
>>> oct(429)
'0o655'
>>> bin(30)
'0b11110'
>>>
```

# Неименованные функции

- Можно создать неименованную функцию с помощью `lambda`:

```
>>> f = lambda arg1, arg2: arg1**arg2
>>> f(2, 3)
8
>>>
```

- Это удобно для обработки списков:

```
>>> # оставляет только те элементы, у которых длина 3
... filter(lambda s: len(s) == 3, ['one', 'two', 'three'])
['one', 'two']
>>>

>>> map(lambda x: x**3, range(4))
[0, 1, 8, 27]
>>>
```

# List comprehension (1/2)

- List comprehension — способ задания последовательности
  - [выражение for переменная in послед.]
  - [выражение for переменная in послед. if условие]
- В зависимости от скобок генерируются разные типы данных
  - кортеж:
    - (выражение for переменная in послед.)
  - множество:
    - {выражение for переменная in послед.}
  - словарь:
    - { выражение-ключ : выражение-значение for переменная in посл. }
- Удобно использовать для inline-создания последовательностей

# List comprehension (2/2)

```
>>> # квадраты чисел от 0 до 9
... [x**2 for x in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> # квадраты четных чисел от 0 до 9:
... [x**2 for x in range(10) if x % 2 == 0]
[0, 4, 16, 36, 64]
>>> # кортеж
... (x**2 for x in range(10))
(0, 1, 4, 9, 16, 25, 36, 49, 64, 81)
>>> # словарь
... { "K" + str(i): i**2 for i in range(6)}
{'K0': 0, 'K1': 1, 'K2': 4, 'K3': 9, 'K4': 16, 'K5': 25, 'K6': 36}
>>>
```

# Практика