

# Экспресс введение в Python 3.6

Муромцев Никита Андреевич  
Выпускник ВМК МГУ,  
аспирант геологического факультета МГУ  
*2017 год*

# Занятие 1

Основные операции, типы данных, ввод-вывод, циклы, условия, создание и использование функции, битовые операции, импорт библиотек.

# Две версии Python 2 и 3

## 1. Изменили стандарт:

- Что-то упростили
- Сделали более логичным (к примеру, оператор *print* стал функцией *print()*)
- Усилили форматы данных под 21 век (появился текст, другой подход к бинарным и т.п.)
- Чуть удобнее читается, что хорошо в больших проектах
- Добавили больше встроенных функций

## 2. Язык разделился на два лагеря

- Много проектов на языке версии 2 потребовали переработки
- Олдфаги

## 3. Активно поддерживаются оба стандарта

# Возможности языка Python

Вот лишь некоторые из них:

- Работа с xml/html/txt/бинарными файлами
- Работа с http запросами
- GUI (графический интерфейс)
- Создание веб-сценариев
- Работа с изображениями, аудио и видео файлами
- Робототехника
- Программирование математических и научных вычислений

Другим языком:

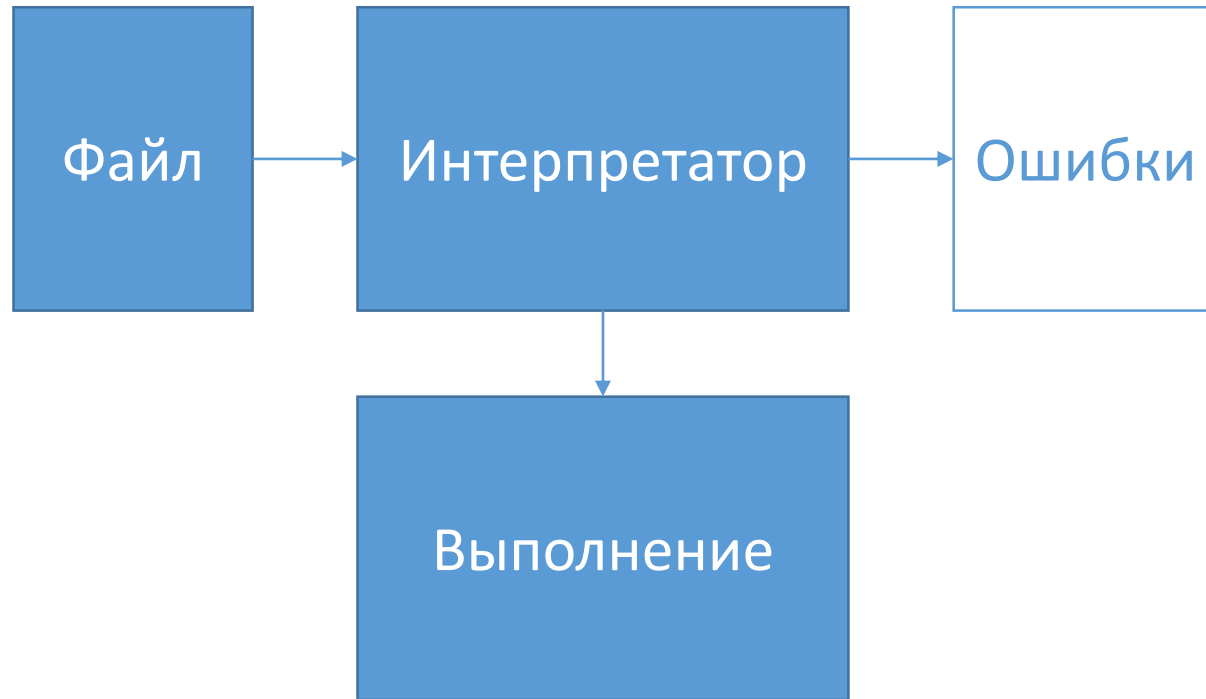
- Работа с файлами научных пакетов, файлами Excel
- Данные из интернета
- Приложения с кнопками/графикой
- Сайт / деталь сайта
- Сгенерировать график, анимацию
- Автоматизировать работу
- Автоматизировать свои вычисления  
объемные/рутинные задачи

Грубо говоря: ВСЁ

# Интерпретатор Python

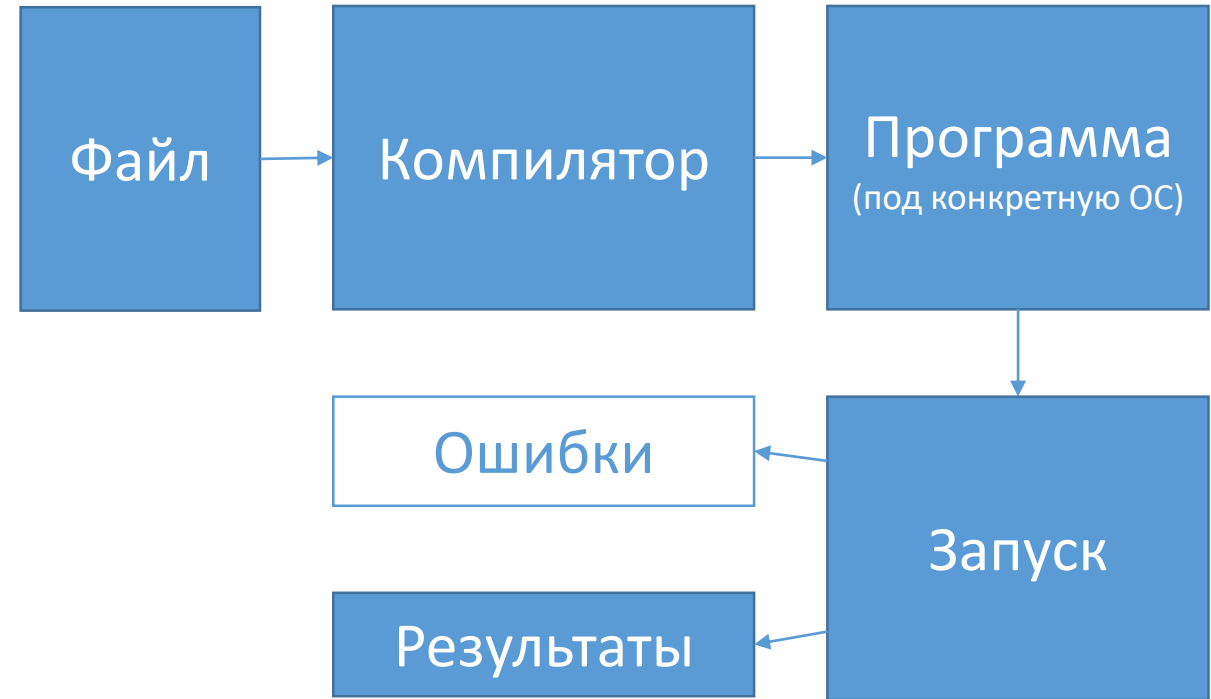
Python - интерпретируемый язык, но может быть и скомпилирован

## Интерпретируемые языки



- Т.е. запускается везде, где есть интерпретатор
- Правится в текстовом редакторе
- После правки готово к повторному запуску

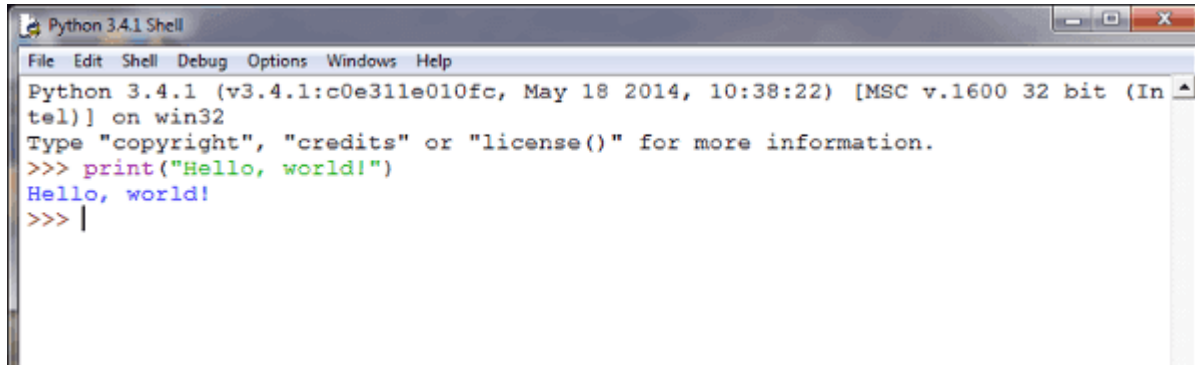
## Компилируемые языки



- Запускается на любой ОС, которую поддерживает
- Для изменений нужен исходный код
- Может не хватать библиотек и программа не соберётся

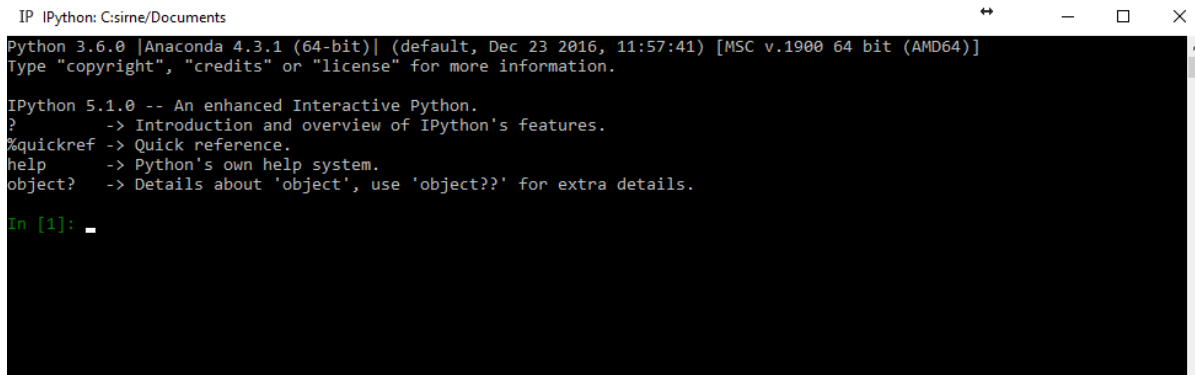
# Консоль python

В любом интерпретируемом языке можно писать программу «налету», т.е. подавать команды и сразу получать результат.



```
Python 3.4.1 Shell
File Edit Shell Debug Options Windows Help
Python 3.4.1 (v3.4.1:c0e311e010fc, May 18 2014, 10:38:22) [MSC v.1600 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> print("Hello, world!")
Hello, world!
>>> |
```

Python консоль в стандартном пакете

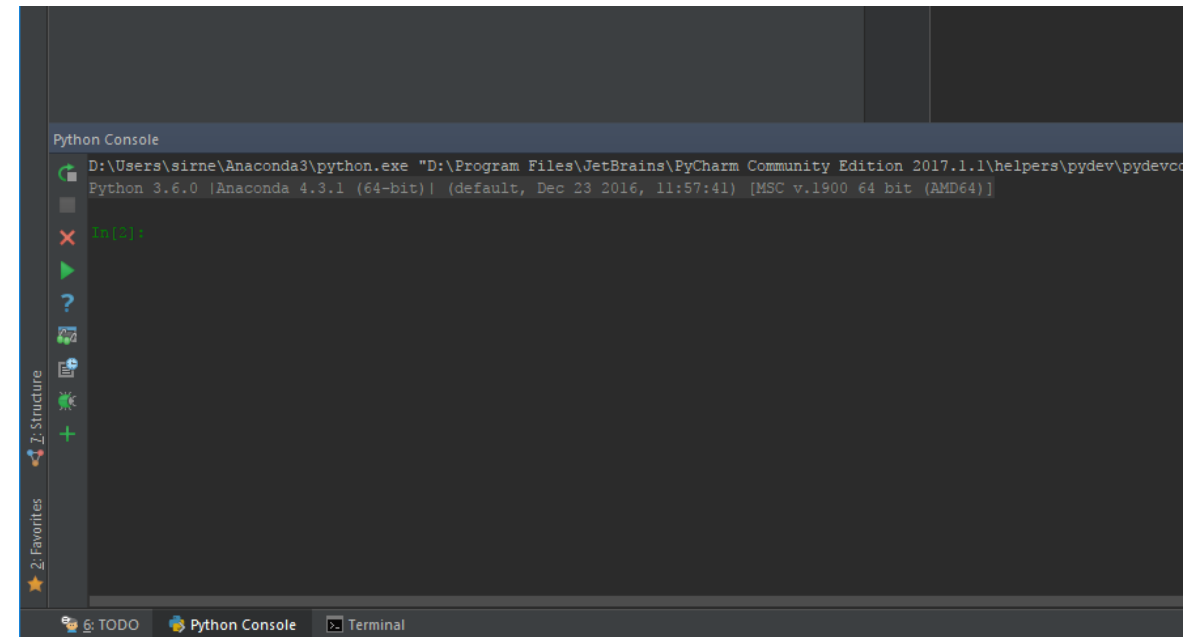


```
IP IPython: C:\sirne\Documents
Python 3.6.0 [Anaconda 4.3.1 (64-bit)] (default, Dec 23 2016, 11:57:41) [MSC v.1900 64 bit (AMD64)]
Type "copyright", "credits" or "license" for more information.

IPython 5.1.0 -- An enhanced Interactive Python.
? -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help -> Python's own help system.
object? -> Details about 'object', use 'object??' for extra details.

In [1]:
```

Python консоль Anaconda



```
Python Console
D:\Users\sirne\Anaconda3\python.exe "D:\Program Files\JetBrains\PyCharm Community Edition 2017.1.1\helpers\pydev\pydevco
Python 3.6.0 [Anaconda 4.3.1 (64-bit)] (default, Dec 23 2016, 11:57:41) [MSC v.1900 64 bit (AMD64)]

In [3]:
```

Python консоль в PyCharm

# Базовые команды

*Все команды ни раз описаны в интернете, поэтому здесь будет приведён список, чтобы вы знали что искать, если забудете*

Простая арифметика (операторы)	$+$ , $-$ , $*$ , $/$
Остаток от деления и целая часть от деления	$\%$ и $//$
Пара ( $x // y$ , $x \% y$ )	<code>divmod(x,y)</code>
Возведение в степень	$x^{**}y$
Модуль числа и смена знака	<code>abs(x)</code> , $-x$

*Конечно, работают скобочки для придания приоритета*

Пример: Ещё можно применить  
>>> `1+1` операцию к самому себе:  
2 `x += 1` # увеличить на 1

**Результат** – целое число, если  
операнды (цифры, с которыми вы  
работаете) – целые числа

# Битовые операции

*Числа в компьютере представлены в двоичном виде. Так  $5_{10} = 101_2$ .*

Побитовое «или», «исключающее или»

|, ^ (это на клавише 6)

Побитовое «и» и инверсия битов

& (клавиша 7) и ~x

Битовый сдвиг влево и вправо на n бит

x<<n, n>>y

Перевод в другие системы счисления числа n происходит с помощью функций:

- `int(n, [x])` – перевод в x-ричную систему счисления; если x не указан, то 10-ричную
- `bin(n)` – в двоичную строку
- `hex(n)` – в шестнадцатеричную строку
- `oct(n)` – в восьмеричную строку



# Типы данных

*Некоторые операции работают только с определёнными(ым) типом данных*

Целые числа (для перевода/создания - <code>int(x)</code> )	1, 2345, 12334124124124153123123
--	----------------------------------

Вещественные числа (для перевода/создания - <code>float(x)</code> )	0.1, 234.1231234123
---	---------------------

Рациональные (для перевода/создания - <code>Fraction(x, y)</code> )	<code>Fraction(2, 6) = Fraction(1, 3)</code>
---	--

Вещественные с максимальной точностью ( <code>Decimal(x * y)</code> )	<code>Decimal(7 * 4.9)</code>
---	-------------------------------

Комплексные (для перевода/создания - <code>complex(x, y)</code> )	<code>complex(1, 2)</code>
---	----------------------------

`int(0.11) = 0`

`float(«0.1») = 0.1`

`x = complex(1, 2)`

`print(x + y)`

`int(1.1) = 1`

`int(«0.1») = error`

`print(x)`

`>>> (4+6j)`

`float(1) = 1.0`

`int(«1») = 1`

`>>> (1+2j)`

`print(x * y)`

`y = complex(3, 4)`

`>>> (-5+10j)`

# Импорт библиотек

*Для python 3 существует очень много сторонних библиотек (модулей) с функциями для большинства потребностей. Аналогично и для второй версии 2. При желании можно добиться совместимости библиотек 2 и 3 версий.*

Импорт всей библиотеки	<code>import &lt;название&gt;</code>
------------------------	--------------------------------------

Импорт библиотеки под другим именем	<code>import &lt;название&gt; as &lt;своё название&gt;</code>
-------------------------------------	---

Импортировать отдельную функцию	<code>from &lt;название библиотеки&gt; import &lt;название&gt;</code>
---------------------------------	---

Импортировать под другим именем	<code>from &lt;библ.&gt; import &lt;название&gt; as &lt;название&gt;</code>
---------------------------------	---

<code>import math</code>	<code>math.pi</code>
<code>import math as m</code>	<code>m.sqrt(85)</code>
<code>from math import sqrt</code>	<code>sqrt(85)</code>
<code>from math import sqrt as p</code>	<code>p</code>
<code>import random</code>	<code>random.random()</code>

# ВВОД - ВЫВОД

*Способов считывания и вывода информации программой большое количество*

Вывод	<code>print(&lt;что вывести&gt;)</code>
Считывание	<code>&lt;куда считать&gt; = input (&lt;сообщение пользователю&gt;)</code>

```
print("Hello world!")
name = input("Как Вас зовут? ")
Как вас зовут _
```

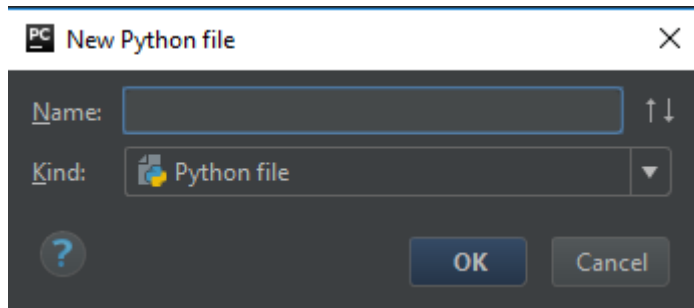
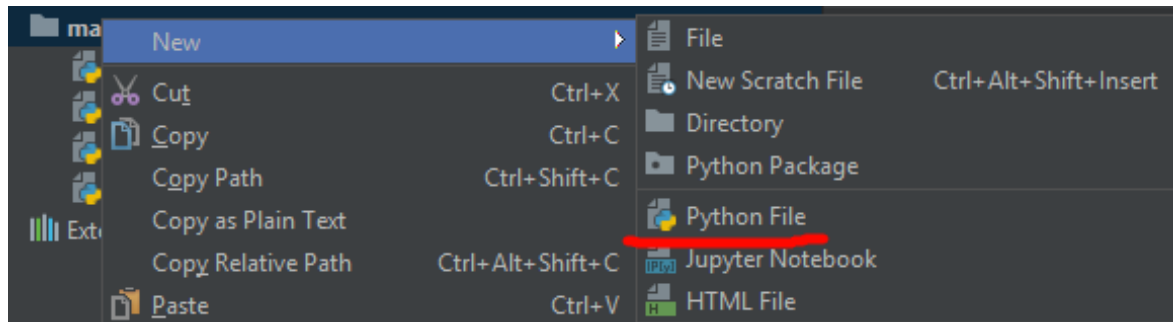
```
x = input()
z = x + 1
>>> error
```

```
x = input()
z = int(x) + 1
>>> error
```

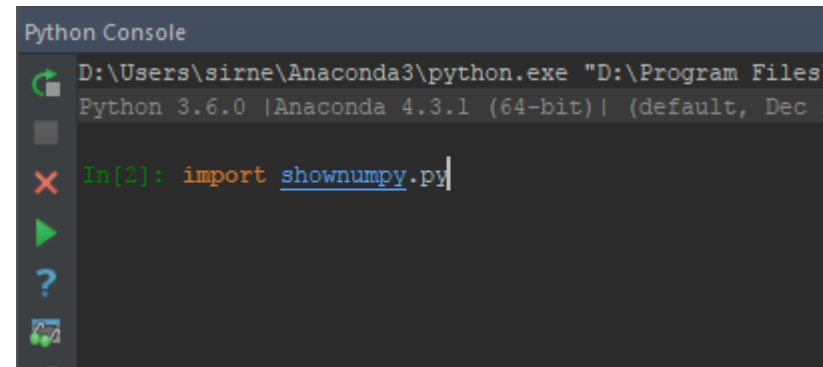
# Создание python файла

*Для интерпретирования можно создать текстовый файл с любым названием и расширением .py*

## Как это делается в PyCharm



## Как загрузить файл в консоль python



`import <путь до файла>`

После импорта будут доступны содержащиеся в нём функции, а также он запустится на исполнение, если какие-либо команды в нём не экранированы

Проверьте свои силы: <http://informatics.mccme.ru/mod/statements/view.php?id=2296#1>

# Синтаксис

*Синтаксис python достаточно прост, но есть несколько важных нюансов*

Разделителем команд может служить либо переход на новую строку, либо «;» (точка с запятой)

`a = 1; b = 2; print(a, b)` # это корректно

---

Некоторые конструкции требуют вложенности. В таком случае первая строка в конструкции является модулятором параметров или флагом запуска выполнения сложных команд, а блок последующий команд являются инструкциями, которые выполняются в контексте запускающей строки.

```
if a > b:  
    print("YEAN")  
    print("21")
```

Здесь `if a>b` - условие запуска, а далее блок команд для выполнения

При этом верна команда `if a > b: print("YEAN")`, т.к. в блоке инструкций всего одна команда

Внутри файла каждый отступ должен состоять из одинаковых символов. Это может быть табуляция, либо 2,3,4 и более пробелов, но каждый раз одно и то же число.

Для каждого следующего уровня вложенности добавляем ещё один отступ.

# УСЛОВИЯ

*Для использования условий необходимо знать логические операции. Не путайте с битовыми операциями, в данном случае результат либо «правда», либо «ложь»*

И, или, отрицание(не)

X and Y, X or Y, not X

Больше, меньше, равно

X > Y, X < Y, X == Y (обратите внимание на два равно)

Не больше, не меньше, не равно X <= Y, X >= Y (равно на втором месте), X != Y или X <> Y

Проверить есть ли элемент во множестве

X in Y аналогично «not in»

Сравнение операндов

x is y вернёт истину, если x и y ссылаются на один объект

Также бывают функции, которые выдают логический результат

```
from math import sqrt as s
from math import sqrt
print(s is sqrt)
>>> True
```

1 == 1

>>> True

1 <= 1

>>> True

1 < 1

>>> False

1 and 1

>>> 1

1 and 2

>>> 2

not 1

>>> False

False and True

>>> False

False or True

>>> False

not 1

>>> False

# УСЛОВИЯ

## Варианты составления условий

### Простое условие

```
if <условие>: <одна операция>
```

### Условие с несколькими операциями

```
if <условие>:  
    <операция 1>  
    <операция 2>
```

### Условие, в котором выполняется один из блоков операций

```
if <условие>:  
    <операция 1>  
    <операция 2>  
else:  
    <операция 1>  
    <операция 2>
```

Серия условий, которые проверяются по очереди. Блок операций срабатывает у первого сработавшего. Если не будет else, то может не выполниться ничего

```
if <условие 1>:  
    <операция 1>  
    <операция 2>  
elif <условие 2>:  
    <операция 1>  
    <операция 2>  
elif <условие 3>:  
    <операция 1>  
    <операция 2>  
else:  
    <операция 1>  
    <операция 2>
```

Трёхместное условие:  $A = Y \text{ if } X \text{ else } Z$   
<операция> if <условие> else <операция в противном случае>

# ЦИКЛЫ

*Цикл позволяет повторять блок операций по заданному условию*

Цикл с предусловием

```
while <условие>:  
    <операция 1>  
    <операция 2>
```

*Пока условие истинно, повторять*

- **break** досрочно прерывает цикл (в блоке операций)
- **continue** начинает следующий проход цикла, минуя оставшееся тело цикла (в блоке операций)
- **else:** операции в блоке **else** выполнятся, если цикл закончился без помощи **break** (на уровне с for/while)

Цикл с итератором по множеству

```
for <переменная> in <множество>:  
    <операция 1>  
    <операция 2>
```

*Операции повторятся столько раз, сколько элементов во множестве. В операциях можно использовать итератор.*

Варианты множеств:

- Тестовая строка: «HELLO»
- Множество натуральных чисел:
  - range(start, stop)
  - range(count)
  - range(start, stop, step)



# ФУНКЦИИ

*Чтобы блок операций можно было использовать многократно, для удобства, для некоторых стилей программирования и т.д. команды объединяют в функции*

`def <название> ([операнды, их может не быть]):`

`<операции>`

`return <результат, т.е. здесь писать переменную или сразу функцию>`

`# return может не быть`

```
a = 1
print("значение до", a)

def first(a):
    if a < 10:
        a += 1
        print("сейчас a =", a)
        a = first(a) # рекурсия
    return a

print("значение после", first(a))
```

# Задачи для практики по занятию 1

Обязательно (1 балл, примерно на 1-2 часа): Дополнительно (1,5 балла):

- [2937 – ввод-вывод](#)
- [2938 – яблоки](#)
- [338 – цифры в обратном порядке](#)
- [74 – a+b](#)
- [596 – пробежка](#)
- [595 – диета](#)
- [2944 – сумма цифр](#)
- [2949 – обмен значений](#)
- [315 – сумма квадратов](#)
- [120 – сумма и факториалы](#)
- [334 – остаток](#)
- [1476 – часовая стрелка](#)
- [341 – делители числа](#)
- [115 – количество нулей](#)
- [3064 – длина последовательности](#)

- [597 – пробежка 2](#)
- [2940 – мкад](#)
- [2945 – четное число](#)
- [2951 – пирожки](#)
- [2955 – улитка](#)
- [352 – степень](#)
- [335 – полные квадраты](#)
- [1433 – кролики](#)

Особые (3,5 балла):

- [248 – шахматные слоны](#)
- [2950 – расписание](#)
- [2967 – найти значение](#)
- [2957 – делимость](#)
- [2958 - максимум](#)
- [321 – сложная сумма](#)

# Дополнение

Если в одной строке при вводе будет несколько чисел, то нужно использовать функцию `split()`

Пример:

```
a, b, c = input().split()
```

a, b, c – получатся текстовыми

Чтобы a, b, c вышли числами необходимо применить следующую конструкцию^

```
a,b,c = map(int, input().split())
```

# Занятие 2

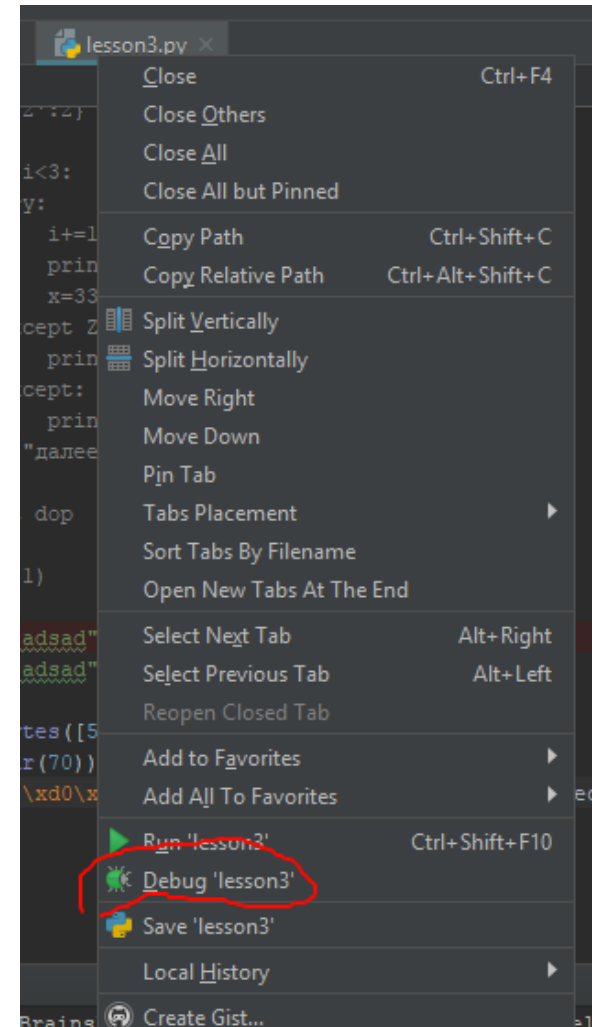
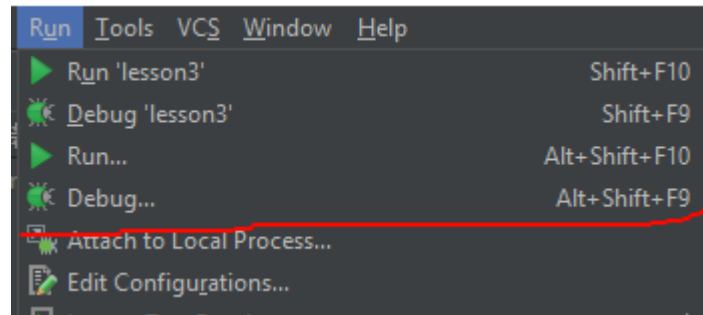
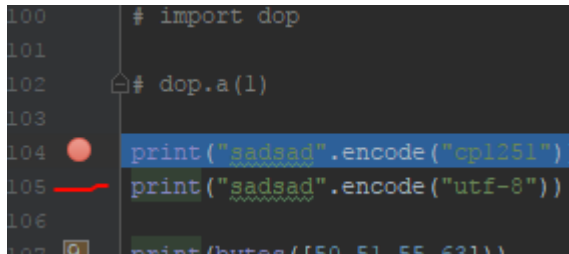
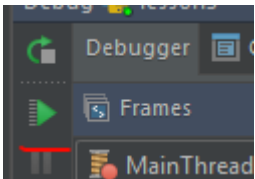
Строки, регулярные выражения, списки, кортежи, словари, множества, собственные модули, работа с файлами

# Debug режим (отладка, профилирование)

*Для использования необходимо стороннее ПО, к примеру IDE PyCharm*

В PyCharm несколько вариантов запуска, три из них на скриншотах.

- В данном режиме можно пошагово выполнить все инструкции программы с разной степенью детализации.
- На всех шагах можно посмотреть текущее значение переменных.
- Можно устанавливать стоп-точки, чтобы быстро выполнить инструкции до точки останова (без пошагового прохода).



# Строки

*Строки в апострофах и кавычках - одно и то же*

Строка в python 3 – список символов.

'spam"s' и "spam's" - верно

"spam"s" - ошибка

*Экранированные последовательности:*

Есть служебные комбинации, которые преобразовываются в другие символы:

- \n – перевод строки
- \t – горизонтальная табуляция
- \v – вертикальная табуляция
- Есть и другие символы (используются сильно реже)

*Для отключения экранирования перед строкой пишем r, к примеру: r'text'*

Можно писать большие блоки текста в тройных апострофах

```
с = '''это очень большая  
строка, многострочный  
блок текста'''
```

```
>>> с
```

```
'это очень большая\nстрока,  
многострочный\nблок текста'
```

```
>>> print(с)
```

```
это очень большая  
строка, многострочный  
блок текста
```

# Строки - операции

*Различных функций очень много, вот наиболее популярные*

Сложение (объединение)

`'test' + 'quest' = 'testquest'`

Дублирование

`'test' * 3 = 'testtesttest'`

Длина

`len('test') = 4`

Индексация и срез

`'test'[2] = 's'; 'test'[0:1:-1] = 'et'`

Привести операнд в строковый тип

`str(x)`

Поиск подстроки

`str.find(temp, st, end); str.index(temp, st, end)`

Разделение строки (результат – список)

`str.split('разделитель')`

Объединить элементы списка в строку

`'символ для промежутков'.join(список)`

Удаление незначащих пробелов

`str.strip()`

# Строки - форматирование

*Часто требуется вывести большое количество данных в читаемом виде*

```
# Вставить символ (по порядку)
>>> '{} , {} , {}'.format('a', 'b', 'c')
'a, b, c'

# Вставить символ в личном порядке
>>> '{0}, {1}, {0}'.format('a', 'b', 'c')
'a, b, a'

# Для передачи списка в качестве аргумента - '*'
>>> c = [1,2,3]
>>> '{2}, {1}'.format(*c)
'3 2'

# Для передачи словаря использовать '**'

# Возможности адрессации
>>> '{second[1]}, {first}'.format(first = 'a', second = ['b','c'])
'c, a'

# Выравнивание - символы '>', '<', '^', '='
# Символ заполнителя до символа выравнивания (в примере '*')
>>> '{:<30}'.format('left aligned')
'left aligned'
>>> '{:>30}'.format('right aligned')
'right aligned'
>>> '{:^30}'.format('centered')
'*****centered*****'
```

```
# Использование знака:
# '+' - все числа
# '-' - только у отрицательных
# ' ' - минусы и пробелы для положительных
>>> '{:+f}; {:+f}'.format(1, -1)
'+1; -1'
>>> '{: +f}; {: +f}'.format(1, -1)
' 1; -1'
>>> '{: +f}; {: +f}'.format(1, -1)
'1; -1'

# Вывод в разных форматах, в том числе с префиксом (через '#')
>>> "int: {0:d}; hex: {0:x}; hex: {0:#x}; oct: {0:#o}; bin: {0:b}".format(42)
'int: 42; hex: 2a; hex: 0x2a; oct: 0o52; bin: 101010'

# Точность
>>> '{0:.3f}'.format(75.765367)
'75.765'

# Отступ
>>> 'aaa{0:4}aaa {1:5}'.format(3, "kkk")
'aaa 4aaa kkk '

# Есть и другие

# Проценты
>>> '{:.2%}'.format(34/67)
'50.75%'

# Возможны комбинации, пример
>>> '{:*>#30.5}'.format(34.34)
'*****-34.340'
```

Для комбинаций используется следующая грамматика (в [] – элементы, которые могут указываться 0-1 раз):

"{[имя поля] [!преобразование (r или s)] [:[символ заполнения]символ выравнивания (>, <, =, ^)][знаковость (пробел, +, -)][#][0][ширина поля (знаков)][,][.точность (обратите внимание на точку, указывается число символов на числе конкретно)][тип (b, c, d, e, E, f, F, g, G, n, o, s, x, X, %)]}"



# Регулярные выражения

*Зная структуру строки, её проще преобразовывать подобными выражениями*

## import re # не забыть

```
# Поиск совпадений по шаблону с начала строки, возвращает элемент класса, содержимое
выводим через group
result = re.match(r'AV', 'AV Analytics Vidhya AV')
result.group(0) # = AV
result.start() # = 0
result.end() # = 2
# Если будем искать 'Analytics', то не будет ничего найдено
# Поиск '.*Analytics' даст результат
re.match(r'.*Analytics', 'AV Analytics Vidhya AV').group(0)
# = AV Analytics

# Есть метод search, который ищет не с начала строки, он найдёт 'Analytics'

# Поиск всех вхождений
result = re.findall(r'AV', 'AV Analytics Vidhya AV')
result # = ['AV', 'AV']

# re.split(шаблон, строка, [максимальное кол-во разделений=0]) - разделяет по шаблону

# re.sub(шаблон, на что заменять, строка) - делает замены шаблона
re.sub(r'India', 'the World', 'AV is largest Analytics community of India')
# = 'AV is largest Analytics community of the World'

Примеры использования:
re.findall(r'\w', 'AV is largest')
# ['A', 'V', 'i', 's', 'l', 'a', 'r', 'g', 'e', 's', 't']

result = re.findall(r'\w*', 'AV is largest')
# ['AV', '', 'is', '', 'largest']

result = re.findall(r'\w+', 'AV is largest') # ['AV', 'is', 'largest']
result = re.findall(r'^\w+', 'AV is largest') # ['AV']
re.findall(r'\w+$', 'AV is largest') # ['largest']
re.findall(r'\w\w', 'AV is largest') # ['AV', 'is', 'la', 'rg', 'es']
re.findall(r'\b\w.', 'AV is largest') # ['AV', 'is', 'la']
```

## Шпаргалка для составления шаблонов:

.	Один любой символ, кроме новой строки \n.
?	0 или 1 вхождение шаблона слева
+	1 и более вхождений шаблона слева
*	0 и более вхождений шаблона слева
\w	Любая цифра или буква (\W — все, кроме буквы или цифры)
\d	Любая цифра [0-9] (\D — все, кроме цифры)
\s	Любой пробельный символ (\S — любой непробельный символ)
\b	Граница слова
[..]	Один из символов в скобках ([^..] — любой символ, кроме тех, что в скобках)
\	Экранирование специальных символов (\. означает точку или \+ — знак «плюс»)
^ и \$	Начало и конец строки соответственно
{n,m}	От n до m вхождений ({,m} — от 0 до m)
a b	Соответствует a или b
()	Группирует выражение и возвращает найденный текст
\t, \n, \r	Символ табуляции, новой строки и возврата каретки соответственно

# Списки

## Самый популярный элемент

`list(элемент)` – преобразовать в список

`s = []` – создать пустой список

`l = ['s', ['a', 1, 2], 'o', 5]` – список со списком

`l[2] # = ['a', 1, 2]`

`l[2][1] # = 1`

Генераторы списков:

`[c * 3 for c in 'list'] # = ['lll', 'iii', 'sss', 'ttt']`

`[c + d for c in 'list' if c != 'i' for d in 'spam' if d != 'a']`

`# ['ls', 'lp', 'lm', 'ss', 'sp', 'sm', 'ts', 'tp', 'tm']`

`['a']*3 # ['a', 'a', 'a']`

Важно помнить, что операции производятся над самими списками и не возвращают новый элемент:

`a = [1, 2, 3]; b = a; b[1] = 5`

# поменяет и значение в a, т.к. переменные хранят лишь указатель

## Основные методы:

<code>list.append(x)</code>	Добавляет элемент в конец списка
<code>list.extend(L)</code>	Добавить в конец все элементы списка L
<code>list.insert(i, x)</code>	Вставляет на i-ую позицию x
<code>list.remove(x)</code>	Удаляет первый элемент в списке, имеющий значение x. ValueError, если такого элемента не существует
<code>list.pop([i])</code>	Удаляет i-ый элемент и возвращает его. Если индекс не указан, удаляется последний элемент
<code>list.index(x, [start [, end]])</code>	Возвращает положение первого элемента со значением x (при этом поиск ведется от start до end)
<code>list.count(x)</code>	Возвращает количество элементов со значением x
<code>list.sort([key=функция])</code>	Сортирует список, может быть использована сторонняя функция возвращающая значения, которые использовать для сортировки
<code>list.reverse()</code>	Разворачивает список
<code>list.copy()</code>	Поверхностная копия списка
<code>list.clear()</code>	Очищает список

# Словарь

*Удобно для структурированной работы с большим объёмом данных*

Создать словарь:

```
d = {'dict': 1, 'dictionary': 2}
```

```
d = dict(dict=1, dictionary=2)
```

```
d = dict([('dict', 1), ('dictionary', 2)])
```

```
d = dict.fromkeys(['a', 'b'], 100)
```

```
# {'a': 100, 'b': 100}
```

```
d = {a: a ** 2 for a in range(7)}
```

```
# {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36}
```

```
d = {1: 2, 2: 4, 3: 9}
```

```
d[1] # = 2, так обращаемся по ключу
```

```
d[4] = 4 ** 2
```

так можно поменять элемент, если его нет,  
создастся новый

## Методы словарей

<code>dict.clear()</code>	очищает словарь.
<code>dict.copy()</code>	возвращает копию словаря.
<code>dict.get(key[, default])</code>	возвращает значение ключа, если его нет default (по умолчанию None).
<code>dict.items()</code>	возвращает пары (ключ, значение).
<code>dict.keys()</code>	возвращает ключи в словаре.
<code>dict.pop(key[, default])</code>	удаляет ключ и возвращает значение. Если ключа нет, возвращает указанный default / исключение.
	удаляет и возвращает пару (ключ, значение). Если словарь пуст, бросает исключение <code>KeyError</code> .
<code>dict.popitem()</code>	!!! Словари неупорядочены.
	возвращает значение ключа, но если его нет, не бросает исключение, а создает ключ с значением default (по умолчанию None).
<code>dict.setdefault(key[, default])</code>	
	добавляет пары (ключ, значение) из other.
<code>dict.update([other])</code>	Существующие ключи перезаписываются. Возвращает None (не новый словарь!).
<code>dict.values()</code>	возвращает значения в словаре.

# Кортежи (tuple)

*Неизменяемые списки занимают меньше места в памяти и могут быть использованы в качестве ключей словаря*

Создать кортеж:

```
t = ()
```

```
t = tuple()
```

```
t = ('s') # получится строка
```

```
t = ('s',) # получится кортеж
```

```
t = 's', # тоже кортеж
```

Операции над списками, не изменяющие список применимы и к кортежам

Благодаря кортежам  
возможна операция:

**a, b = b, a**

# Множества

## Список из неповторяющихся элементов

Создать множество:

```
s = set()
```

```
s = {1, 2, 3, 4}
```

```
s = {i ** 2 for i in range(10)}
```

```
# {0, 1, 4, 9, 16, 25, 36}
```

```
s = frozenset() # неизменяемое множество
```

## Операции, меняющие множество

```
set.update(other, ...);
```

```
set |= other | ... объединение
```

```
set.intersection_update(
```

```
other, ...); set &= other & ... пересечение
```

```
set.difference_update(
```

```
other, ...); set -= other | ... вычитание
```

```
set.add(elem) добавляет элемент в множество
```

```
set.remove(elem) удаляет элемент из множества
```

```
set.discard(elem) удаляет элемент, если он находится в множестве
```

```
set.pop() удаляет и возвращает некоторый элемент множества
```

```
set.clear() очистить
```

## Операции, возвращающие результат

```
len(s) размер множества
```

```
x in s принадлежность к множеству
```

```
set.isdisjoint(other) истина, если set и other не имеют общих элементов
```

```
set == other множества идентичны
```

```
set.issubset(other) или
```

```
set <= other все элементы set принадлежат other
```

```
set.issuperset(other)
```

```
или set >= other аналогично
```

```
set.union(other, ...) или
```

```
set | other | ... объединение нескольких множеств
```

```
set.intersection(other,
```

```
...) или set & other & ... пересечение
```

```
set.difference(other, ...) множество из всех элементов set, не
```

```
или set - other - ... принадлежащие ни одному из other
```

```
set.symmetric_difference(other); set ^ other множество из элементов, встречающихся в
```

```
одном множестве, но не встречающиеся в обоих
```

```
set.copy() копия множества
```

```
set.symmetric_difference_update(other);
```

```
set ^ other множество из элементов, встречающихся в
```

```
одном множестве, но не встречающиеся в обоих
```

# Файлы

Открыть файл (связать с переменной):

```
f = open(путь до файла, режим)
```

Путь до файла: 'test.txt', 'folder//test.txt',  
'C://folder//test.txt'

Режимы:

'r'	чтение (по умолчанию).
'w'	запись, перезапись
'x'	запись, ошибка, если нет
'a'	запись, дополнение
'+'	запись и чтение

Режимы работы:

'b'	двоичный режим
't'	текстовый (по умолчанию).

В текстовый файл можно писать только текст

В двоичный файл можно писать структуры

После работы с файлом его следует закрыть (особенно при записи):

```
f.close()
```

Читаем с помощью read()

Пишем с помощью write(строка/текст)

```
>>> f = open('text.txt')
>>> f.read(1)
'H'

>>> f.read()
'ello world!\nThe end.\n\n'
>>> f.close()
>>> f = open('text.txt')
>>> for line in f:
...     line

>>> for i in [1,2,3,4,5]:
...     f.write(str(i) + '\n')
```

# Задачи для практики по занятию 2

Список задач опубликован по ссылке:

[https://docs.google.com/spreadsheets/d/1d5UjabePXB9QpW2oTY6QtUkjn7NDMgBMvQzpkW\\_ccZ0/edit#gid=258125635](https://docs.google.com/spreadsheets/d/1d5UjabePXB9QpW2oTY6QtUkjn7NDMgBMvQzpkW_ccZ0/edit#gid=258125635)

На соседней есть результаты работы нашей группы:

[https://docs.google.com/spreadsheets/d/1d5UjabePXB9QpW2oTY6QtUkjn7NDMgBMvQzpkW\\_ccZ0/edit#gid=0](https://docs.google.com/spreadsheets/d/1d5UjabePXB9QpW2oTY6QtUkjn7NDMgBMvQzpkW_ccZ0/edit#gid=0)

Времени на решение всех задач 1, 2 и 3 блока + задачи по моделированию у вас **до конца семестра**.

Для допуска к сдаче задачи по моделированию необходимо решить **45 основных** задач и набрать не менее **60 баллов**.

# Занятие 3

Структуры и файлы: бинарные файлы (pickle), строковые структуры (json), исключения, менеджер контекста, лямбда функции, создание и подключение модулей, баты и байтовые массивы



# Запись структур в файл (pickle)

## *Разбор модуля pickle*

`pickle.dump(obj, file, protocol=None, fix_imports=True)`

Записывает сериализованный объект в file.

protocol указывает используемый протокол (По умолчанию 3 - рекомендован для Python 3, не совместим с предыдущими версиями).

Записывать и загружать надо с одним и тем же протоколом.

`pickle.dumps(obj, protocol=None, fix_imports=True)`

Возвращает сериализованный объект.

Далее можно использовать как хочешь.

`pickle.load(file, fix_imports=True, encoding="ASCII", errors="strict")`

Считывает объект из file, возвращает его.

`pickle.loads(bytes_object, fix_imports=True, encoding="ASCII", errors="strict")`

Загружает объект из потока байт, возвращает.

```
>>> import pickle
>>> data = {
...     'a': [1, 2.0, 3, 4+6j],
...     'b': ("character string", b"byte string"),
...     'c': {None, True, False}
... }
>>>
>>> with open('data.pickle', 'wb') as f:
...     pickle.dump(data, f)
...
>>> with open('data.pickle', 'rb') as f:
...     data_new = pickle.load(f)
...
>>> print(data_new)
{'c': {False, True, None}, 'a': [1, 2.0, 3, (4+6j)], 'b':
('character string', b'byte string')}
```

# Запись структур в файл (json)

*Json работает с текстом, генерирует/разбирает текстовую строку*

## Кодирование

```
>>> json.dumps(['foo', {'bar': ('baz', None, 1.0, 2)}})
'["foo", {"bar": ["baz", null, 1.0, 2]}]'
```

## Сортировка ключей

```
>>> print(json.dumps({"c": 0, "b": 0, "a": 0},
sort_keys=True))
{"a": 0, "b": 0, "c": 0}
```

## Нестандартные сепараторы

```
>>> json.dumps([1,2,3,{'4': 5, '6': 7}],
separators=(',', ':'))
'[1,2,3,{"4":5,"6":7}]'
```

## Красивый вывод

```
>>> print(json.dumps({'4': 5, '6': 7},
sort_keys=True, indent=4))
{
    "4": 5,
    "6": 7
}
```

## Декодирование

```
>>> json.loads('["foo", {"bar":["baz", null, 1.0, 2]}]')
['foo', {'bar': ['baz', None, 1.0, 2]}]
>>> json.loads('""\\"foo\\bar"')
'foo\bar'
```

# Исключения

*Чтобы программа не прерывалась из-за любой ошибки, вы можете их ловить. Также этот механизм может помочь вам в отладке и обслуживании*

**try:**

Инструкции

**except** тип исключения:

Что делать

К примеру, закрыть файл

**except:**

Что делать при любом  
ещё не пойманном исключении  
в этом блоке

**else:**

Что делать, если не было  
исключений

**finally:**

Что делать в любом случае после блока

```
>>> f = open('1.txt')
>>> ints = []
>>> try:
...     for line in f:
...         ints.append(int(line))
... except ValueError:
...     print('Это не число. Выходим.')
... except Exception:
...     print('Это что ещё такое?')
... else:
...     print('Всё хорошо.')
... finally:
...     f.close()
...     print('Я закрыл файл.')
...     # Именно в таком порядке: try, группа
...     except, затем else, и только потом finally.
...
Это не число. Выходим.
Я закрыл файл.
```

# Менеджер контекста

*Иногда удобнее и нагляднее выполнить блок инструкций в рамках контекста, использующегося только в данном блоке*

"with" expression ["as" target] ["," expression ["as" target]] ":"

- Конструкция гарантирует выполнение критической функций, к примеру закрытие файла
- Несколько выражений = несколько уровней вложенности

```
with open('newfile.txt', 'w', encoding='utf-8') as g:
    d = int(input())
    print('1 / {} = {}'.format(d, 1 / d), file=g)
```

# Функции - часть 2

Функция без параметров

```
def func():  
    инструкции
```

Функция с параметрами

для с задано значение по умолчанию

```
def func(a, b, c = '11'):  
    инструкции
```

```
func(2, 3) # c = 11
```

```
func(2, 3, 4) # c = 4
```

`pass` — ничего не делать (иногда нужно)

Переменное количество аргументов (\*)

```
def func(*args):
```

`args` — кортеж

Функция с переменным количеством именованных аргументов (\*\*)

```
def func(**args):  
    return args
```

```
func(a=1, b=2, c=3)
```

```
{'a': 1, 'c': 3, 'b': 2}
```

Лямбда функции — полезны для использования в качестве аргумента

```
func = lambda x, y: x + y
```

```
func(1, 2) # = 3
```

```
(lambda x, y: x + y)('a', 'b') # = 'ab'
```

```
map(lambda x: x*x, [1, 2, 3, 4, 5])
```

```
# = [1, 4, 9, 16, 25]
```

```
filter(lambda x: x < 4, [1, 2, 3, 4, 5])
```

```
# = [1, 2, 3]
```

# МНОГОМОДУЛЬНОСТЬ

*Код получается читабельней и удобнее в последующей использовании (даже в разработке), если вы большую задачу делите максимально на подзадачи. Т.е. стоит повторяющиеся куски кода делать в виде функций, а комплекты функций, сходные по смыслу выносить в отдельные модули, которые можно будет развивать в дальнейшем в других проектах.*

Подключение модуля, в т.ч. своего

```
import test
import test/test
```

Название модуля либо занесено в переменную окружения, либо указывается путь до файл, либо название, если из той же папки

При импорте выполняются все инструкции в файле, в т.ч. импорт библиотек, т.е. код добавляется в код вашего файла.

Защититься от выполнения кода для единоличного запуска модуля можно через проверку на главенство модуля

```
if __name__ == "__main__":
```

# Многомодульность — часть 2

*Ещё кое-что о подключениях*

Можно подключить отдельные объекты из библиотеки (модуля), т.е виртуально добавить код функции в свой файл

```
from <Название модуля> import <Атрибут 1  
(название функции, к пр.)> [ as <Псевдоним  
1> ], [<Атрибут 2> [ as <Псевдоним 2> ] ...]
```

Пример:

```
from math import pi as p, e  
from math import *
```

Есть некоторые правила хорошего тона при обозначении своих библиотек:

- Использовать нижний регистр
- По желанию использовать верхний регистр только для отделения слов
- Начинать название с маленькой буквы
- Не использовать дефисы и нижние подчёркивания
- Называть лаконично

Пример:

getTables

# Байты и байтовые массивы

*Иногда может возникнуть проблема с кодировкой, и вам придётся узнать о том, что такое байты*

Посмотрим некоторые примеры:

```
>>> 'Байты'.encode('utf-8')
```

```
b'\xd0\x91\xd0\xb0\xd0\xb9\xd1\x82\xd1\x8b'
```

```
>>> bytes('bytes', encoding = 'utf-8')
```

```
b'bytes'
```

```
>>> bytes([50, 100, 76, 72, 41])
```

```
b'2dLH)
```

```
>>> chr(50)
```

```
'2'
```

```
>>> ord('f')
```

```
102
```

```
>>> b'\xd0\x91\xd0\xb0\xd0\xb9\xd1\x82\xd1\x8b'.decode('utf-8')
```

```
'Байты'
```

```
>>> b = bytearray(b'hello world!')
```

```
>>> b[0]
```

```
104
```



# Задачи для практики по занятию 3

Список задач опубликован по ссылке:

[https://docs.google.com/spreadsheets/d/1d5UjabePXB9QpW2oTY6QtUkjn7NDMgBMvQzpkW\\_ccZ0/edit#gid=258125635](https://docs.google.com/spreadsheets/d/1d5UjabePXB9QpW2oTY6QtUkjn7NDMgBMvQzpkW_ccZ0/edit#gid=258125635)

На соседней есть результаты работы нашей группы:

[https://docs.google.com/spreadsheets/d/1d5UjabePXB9QpW2oTY6QtUkjn7NDMgBMvQzpkW\\_ccZ0/edit#gid=0](https://docs.google.com/spreadsheets/d/1d5UjabePXB9QpW2oTY6QtUkjn7NDMgBMvQzpkW_ccZ0/edit#gid=0)

Времени на решение всех задач 1, 2 и 3 блока + задачи по моделированию у вас **до конца семестра**.

Для допуска к сдаче задачи по моделированию необходимо решить **45 основных** задач и набрать не менее **60 баллов**.

# Занятие 4

Консоль Windows, bat-скрипты, функциональное программирование, map, reduce, zip, filter, all, any, декораторы, документирование, классы (наследование, перегрузка, инкапсуляция)

# Получение параметров командой строки

*Программу можно запускать не только с помощью IDE PyCharm*

Python скрипт можно запустить минуя запуск IDE. В случае Unix систем .py скрипт можно сделать исполняемым, и он будет считаться программой. Но во всех системах можно через консоль отдать команду выполнить .py скрипт интерпретатору pythonю

`python <путь до файла> [<параметры командной строки>]`

Получить список аргументов, с которыми был запущен скрипт, можно с помощью атрибута `argv` из модуля `sys` (т.е. `sys.argv` – список параметров, начиная с названия запускаемого файла)

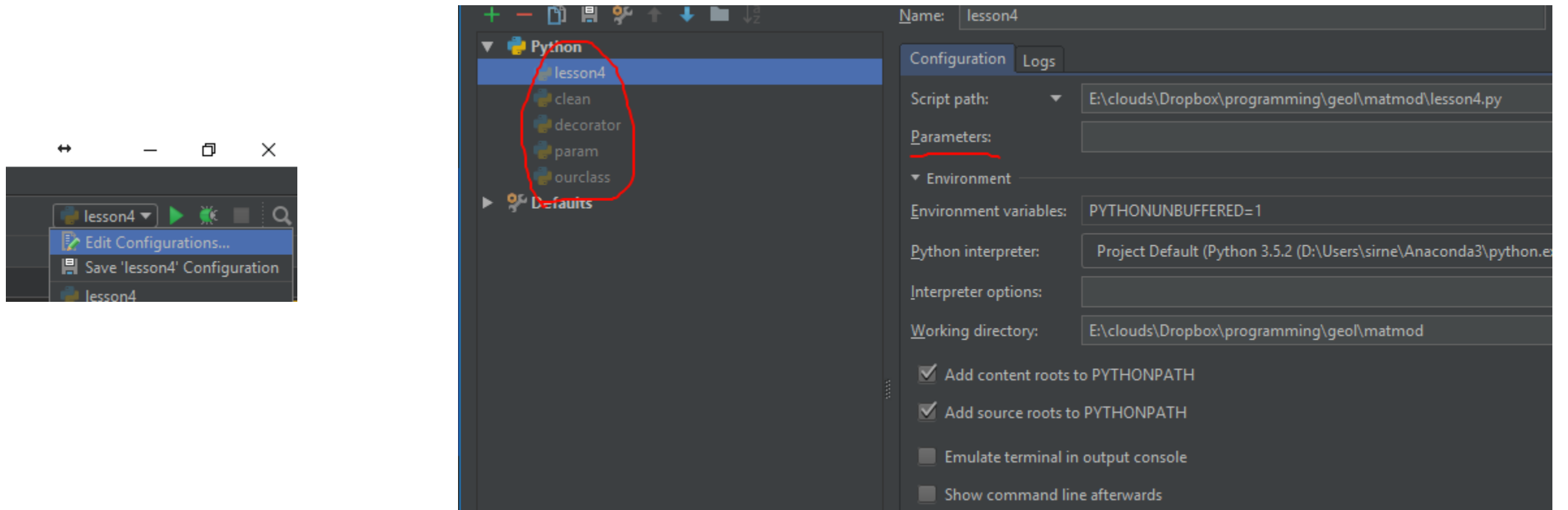
```
import sys
if __name__ == "__main__":
    for param in sys.argv:
        print (param)
```

Команду можно сохранить в текстовый файл с расширением `.bat`, положить в одну папку с нашим скриптом и запускать выполнение скрипта через этот файл

После выполнения команд в BAT-скрипте, окно консоли (CMD) закроется, чтобы отсрочить закрытие используйте команду `pause(<количество секунд>)`

# Параметры командной строки в PyCharm

Потребуется во время разработки, чтобы упростить запуск и отладку



# Функциональное программирование

*Функциональный подход упростит повторное использование кода*

Главное в функциональном коде – отсутствие побочных эффектов.  
Т.е. программа не полагается на данные вне контекста функции.

```
a = 0
def increment1():
    global a
    a += 1
```

Нефункциональный код

```
def increment2(a):
    return a + 1
```

Функциональный код

# Map

*Применяется для выполнения операции над каждым элементом списка*

`map(<функция>, <набор данных (список)>)`

Результат – map элемент, используем list для преобразования

```
name_lengths = list(map(len, ['Маша', 'Петя', 'Юля'])) # => [4, 4, 3]
```

```
squares = list(map(lambda x: x * x, [0, 1, 2, 3, 4])) # => [0, 1, 4, 9, 16]
```

```
new_list = list(map(int, ['1', '2', '3', '4', '5', '6', '7'])) # => [1, 2, 3, 4, 5, 6, 7]
```

Если же количество элементов в списках совпадать не будет, то выполнение закончится на минимальном списке:

```
l1 = [1, 2, 3]
```

```
l2 = [4, 5]
```

```
new_list = list(map(lambda x, y: x + y, l1, l2)) # => [5, 7]
```

# Reduce

*Перерабатывает список в итоговый результат (комбинацию элементов)*

`reduce(<функция>, <набор данных (список)>, [<стартовое значение>] )`

Результат - значение

```
s = sum(lambda x,y: x + y, [0, 1, 2, 3, 4]) # => 10
```

```
sentences = ['капитан джек воробей',  
             'капитан дальнего плавания',  
             'ваша лодка готова, капитан']
```

```
cap_count = reduce(lambda a, x: a + x.count('капитан'),  
                  sentences,  
                  0) # => 3
```

# Zip

*Объединяет списки*

```
zip(<список> , <список>, [<список>])
```

Объединение закончится на минимальном списке

```
a = [1,2,3]
```

```
b = "xyz"
```

```
c = (None, True)
```

```
res = list(zip(a, b, c))
```

```
print (res)
```

```
# => [(1, 'x', None), (2, 'y', True)]
```



# Filter

*Объединяет списки*

**filter**(<функция с булевым результатом (True/False)> , <список>)

Возвращает отфильтрованный список в соответствии с условием

```
mixed = ['мак', 'просо', 'мак', 'мак', 'просо', 'мак', 'просо', 'просо', 'просо', 'мак']
```

```
zolushka = list(filter(lambda x: x == 'мак', mixed)) # => ['мак', 'мак', 'мак', 'мак', 'мак']
```

# All и Any

*Проверка выполнения условия для всех/хотя бы одного элементов списка*

**all**(<генератор списка/список с логическими элементами (False/True)>)

Применяет ко всем элементам списка одновременно операцию «И»

**any**(<генератор списка/список с логическими элементами (False/True)>)

Применяет ко всем элементам списка одновременно операцию «ИЛИ»

```
b = [1,2,3,4,5,5,6,7,8]
```

```
print(all(x<9 for x in b)) # => True
```

```
print(all(x<3 for x in b)) # => False
```

```
print(any(x<3 for x in b)) # => True
```

# Декораторы

*Иногда к выполнению стандартной функции хочется добавить команд*

Декоратор – обертка вокруг функции, с помощью него можно легко добавить вывод отладочной информации для всех используемых функций

```
def decor(k):           # k – функция для декорирования
    def the_wrap(d):    # d – параметр декорируемой функции
        print(type(d))  # команды до декорируемой функции
        r = k(d)         # вызовы декорируемой функции
        pass            # команды после вызова дек. функции
        return r         # возвращаем результат дек. функции
    return the_wrap     # возвращаем обёртку на функцию
```

```
fun = decor(str)        # fun – обернутая str
list = decor(list)      # перегрузили функцию list
```

Другой способ  
оборачивания функции:

```
@decor
def func(s):
    print("test", s)
    return s
```

# Документирование кода

*Помогает понять что делает функция/модуль другим разработчикам*

Для документирования используются тройные кавычки

```
def func():  
    """Документация"""  
    pass
```

- Строки документирования идут сразу после заголовка функции/класса/метода, в начале файла
- Необходимо описываться что функция делает с получаемыми аргументами и что возвращает.

[Посмотрите соглашения, которые используются при документировании](#)

Документацию по объекту можно посмотреть с помощью атрибута `__doc__`:

```
print(str.__doc__)
```

# Классы и объекты

*Работаем в программе с объектами, а не списком операций*

## Пример класса

```
class man():
    age = 0 # переменная класса
    name = "" # переменная класса

    def __init__(self, n="", a=0): # конструктор запускается при создании объекта
        self.name = n # self - это объект для которого вызывается конструктор
        self.age = a

    def ages(self):
        print("возраст:", self.age) # вывести значение age для данного объекта

    def names(self):
        print("имя", self.name) # вывести значение name для данного объекта

a = man() # a - объект класса man со значениями по-умолчанию
a.age = 29 # поменяли нашему человеку возраст
a.names() # вывели имя человека
a.height = 185 # добавили в объект данного класса атрибут height
```

# Классы и объекты - наследование

`self` - обязательный аргумент, содержащий в себе экземпляр класса, передающийся при вызове метода, поэтому этот аргумент должен присутствовать во всех методах класса.

```
class worker(man):  
    def __init__(self, n="", a=0, j=None):  
        # Необходимо вызвать метод инициализации родителя  
        # В Python 3.x это делается при помощи функции super()  
        super().__init__(n, a)  
        self.job = j  
  
    def jobs(self):  
        print("возраст:", self.job) # вывести значение job для данного объекта  
  
b = worker ("Вася", 23, "Газпром")  
b.ages() # объект унаследовал методы класса-родителя  
b.jobs() # объект имеет свои методы  
a.jobs() # при вызове этих методов для объектов родительского класса выведется ошибка
```

Концепция наследования в программировании позволяет сократить время разработки, упростить процесс написания программы как сейчас, так и в будущем, когда заказчик захочет внести "небольшие правки" в проект. Правильно спроектированный класс это, кроме прочего, гибкая структура, которую вы свободно сможете изменять, дополнять в будущем. Важно помнить, что иногда проще создать дополнительный базовый класс, наследовать от него и расширять по мере необходимости, чем сразу написать готовый класс.

# Классы и объекты - инкапсуляция

*Строки в апострофах и в кавычках - одно и то же*

```
class A:
    def _private(self):
        # нижнее подчеркивание, чтобы указать, что метод/атрибут
        # приватный, т.е. попросить его не трогать из вне
        print("Это приватный метод!")

    def __veryPrivate(self):
        print("Это очень приватный метод!")
        # двойное подчеркивание в начале имени атрибута даёт большую
        # защиту: атрибут становится недоступным по этому имени
        # полностью это не защищает, так как атрибут всё равно остаётся
        # доступным под именем _ИмяКласса__ИмяАтрибута

a = A()
a._private() # => Это приватный метод!
a.__Private() # ошибка
a.__A__veryPrivate() # работает
```

# Перегрузка операторов и полиморфизм

*Строки в апострофах и в кавычках - одно и то же*

Чтобы вывести список методов класса используем функцию `dir()`

```
class Vector2D:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other): # перегрузили оператор сложения, т.е. +
        return Vector2D(self.x + other.x, self.y + other.y)

    def __str__(self):
        return '({}, {})'.format(self.x, self.y)

class V2D(Vector2D):
    def __init__(self, x, y):
        super().__init__(x, y)

    def __add__(self, other): # перегрузили оператор сложения, т.е. +
        print("Перегрузили метод")
        return super().__add__(other) # super помогает вызывать вообще любой
        # метод родительского класса
```

```
a = Vector2D(1, 1)
b = Vector2D(2, 2)
c = V2D(3, 3)
d = V2D(4, 4)
```

```
print(a + b)
print(c + d)
print(c + b)
```

Вывод:

```
(3, 3)
Перегрузили метод
(7, 7)
Перегрузили метод
(5, 5)
```

[Список доступных для перегрузки стандартных методов](#)



# Задача по математическому моделированию

Поиск пустот в ячейке кристалла в соответствии с заданными условиями

# Задача по матмоделированию

## Обязательно

Необходимо сделать программу для поиска пустот в ячейках кристаллов с целью поиска мест, где могут располагаться дополнительные атомы (нужно, к примеру, чтобы понять что можно в пустоту вставить).

Т.е. требуется написать программу, которая может провести 2 расчёта расстояний в ячейке кристалла по заданному расположению атомов:

1. Найти и вывести с заданным шагом все точки заданной области ячейки, расстояние до которых от центров атомов подпадает под условие заданное диапазоном. К примеру,  $(0.1; 0.4]$  или  $[0.1; +\infty)$
2. Найти и вывести с заданным шагом все точки заданной области ячейки, расстояние до которых от границ атомов подпадает под условие заданное диапазоном. К примеру,  $(0.1; 0.4]$  или  $[0.1; +\infty)$

# Задача по матмоделированию

Пример файла с описанием ячейки (параметры ячейки хранить в файле)

- Рассмотрим ячейку кристалла (к примеру, CaSiO<sub>3</sub>)
- Мы знаем длины сторон ячейки, а также углы между ребрами (a b c alpha beta gamma)
- Также знаем координаты каждого атома и их радиус (name, x, y, z, r)
- Можем сформировать файл со следующим содержимым:

CaSiO<sub>3</sub> (name)

cell (a b c alpha beta gamma)

3.5460 3.5460 3.5460 90 90 90

coordinates (name, x, y, z, r)

Ca	0.50000	0.50000	0.50000	0.990
Si	0.00000	0.00000	0.00000	1.200
O	0.50000	0.00000	0.00000	0.730
O	0.00000	0.50000	0.00000	0.730
O	0.00000	0.00000	0.50000	0.730

# Задача по матмоделированию

## Запуск и ввод данных

- Сделать возможность ввода через консоль после запуска программы, если не были переданы параметры при запуске
- Сделать возможность получение параметров через аргументы командной строки

Зададим крайним углам ячейки координаты 0 0 0 и 1 1 1

Параметры запуска (пример):

- Указание диапазона поиска – 2 точки по 3 значения (координаты в долях, к примеру, 0.1 0.1 0.1 и 1 1 1 )
- Указываем шаг прохода (точность)
- Два значения, задающих условия для первого измерения
- Два значения, задающих условия для второго измерения
- Путь до файла с параметрами ячейки
- 2 пути до файлов результатов

```
./singonia 0.1 0.1 0.1 1 1 1 0.1 =0.3 0.6 0.1 =1.2  
inp1.txt "test/test1.txt" "test/test2.txt" |
```

Для условий (указания диапазона расстояний, подходящих нам) можно использовать следующий синтаксис: '\_' - значит бесконечность, '=' - точная граница.

Т.е.        =0.3        0.6        эквивалент [0.3; 0.6)

Т.е.        0.3        \_        эквивалент [0.3; +∞)

Т.е.        \_        0.6        эквивалент (0; 0.6)

# Задача по матмоделированию

## Результаты

### Пример вывода:

CaSiO<sub>3</sub> - test with radius

Condition:  $d > 0.1$ ,  $d \leq 1.2$

System ( x, y, z) (alpha, beta, gamma)

( 3.546, 3.546, 3.546) ( 90, 90, 90)

Стартовая точка исследования (в долях): ( 0.1000, 0.1000, 0.1000)

Конечная точка исследования (в долях): ( 1.0000, 1.0000, 1.0000)

Для точки: ( 0.0000, 0.3546, 0.3546), в долях ( 0.0000, 0.1000, 0.1000),  
удовлетворяют условию следующие элементы:

Имя	координата	расстояние
Ca	( -1.7730, 1.7730, 1.7730)	0.197571
Si	( -3.5460, 0.0000, 0.0000)	0.977214
O	( -1.7730, 3.5460, 3.5460)	1.09024
O	( 1.7730, 3.5460, 3.5460)	1.01703
O	( 0.0000, 1.7730, 3.5460)	1.09024
O	( 0.0000, 5.3190, 3.5460)	1.01703
O	( 0.0000, 3.5460, 1.7730)	1.09024
O	( 0.0000, 3.5460, 5.3190)	1.01703

- Результаты требуется записать в 2 файла соответственно типу расчётов (пути до них указываются при запуске)
- В файл с результатом должна выводиться информация об измерении (тип измерения, кристалл, параметры ячейки, области и условия измерения)
- Для каждой точки, если есть хоть один подходящий атом, вывести список атомов, расстояние до которых удовлетворяет условию, с указанием их положения и расстоянием.