Введение в Jupyter и Python

Что есть блокнот?

Блокнот/нотбук --- это файл с разделениями на ячейки. Ячейки бывают двух типов --- для записей и комментариев (markdown) и для кода на Python (вообще говоря, можно писать и на других языках, но в нашем курсе будет рассматриваться только Python).

Markdown (md)

Markdown --- легковесный язык разметки текста. Это означает, что можно небольшими усилиями красиво оформить текст.

Для редактирования/просмотра исходного текста --- нажмите дважды левой кнопкой мыши по ячейке. Для входа в режим просмотра --- нажмите галочку в верхнем правом углу блока или Ctrl+Enter (или Ctrl+Alt+Enter).

Для заголовка в начале строке используйте символ "#". С помощью него можно разбивать текст на логические блоки, для подраздела используйте на один символ больше ("##"). Максимальная глубина разделения --- 6, аналогично `<h1>` - `<h6>` в HTML. Здесь можно использовать разметку HTML, в основном она используется только для оформления таблиц (но не обязательно, как вы увидете дальше), остальное, как раскраска текста используется крайне редко.

Для создания нового абзаца нужна (как минимум) одна пустая строка. Если начать писать одну строку под другой они будут объединены в один абзац.

Несколько пустых строк будут проигнорированы, разрыва текста не будет.

Если нужен разрыв, то начните писать в новом блоке (между двумя блоками может быть один или несколько пустых, но лучше так не делать).

Примеры оформления текста:

- `*` слово `*` или `_` слово `_` --- *слово*
- `**` слово `**` или `__` слово `__` --- **слово**
- `***` слово `***` или `___` слово `___` --- ***слово***
- `~~` слово `~~` --- ~~слово~~
- `` ` `` слово `` ` `` --- `слово`

Последнее делает шрифт моноширинным и выделяет фон текста. Используется для того, чтобы выделить название переменной/класса/метода/etc в тексте. Например: "Переменная `cat_legs` равна 4". Символ находится на кнопке, где тильда (`~`)/ `ё`

на английской раскладке, только без зажатия Shift. Для крупных фрагментов/листингов можно использовать окружение тремя апострофами (кстати говоря, эту и некоторые другие вещи поддерживает телеграм).

Кстати, здесь вы увидели как сделать маркированный список (кроме `*`, можно использовать `+` и `-`). Нумерованный делается схожим образом:

1. Вот
2. Так
3. Вот

При этом вам не обязательно следить за правильностью нумерации в исходном тексте.

1. What
2. Tak
3. What тоже можно

LaTeX

Читается "латех", а **не** "латекс".

LaTeX предназначен для вёрстки математических текстов (и не только). Чтобы освоить его в полной мере нужно потратить не один десяток часов, но, благо, нам это и не нужно.

Вот простые правила. Чтобы вставить формулу в текст, окружите её знаками $. Например: `$a^2 + b^2 = c^2$` даёт $a^2 + b^2 = c^2$. Для выделения формулы в отдельную строку и центрирования её используйте следующую запись:

```
$$
a^2 + b^2 = c^2
$$
```

что даст:

$$a^2 + b^2 = c^2$$

Таблица-шпаргалка

Вот таблица, ориентируясь на которую можно намного красивее оформлять свои формулы. Обратите внимание, что если в индексе (верхнем или нижнем), в дроби, под корнем и в других местах содержится более одного символа необходимо заключать эти символы в фигурные скобочки.

Обозначение	Код LaTeX	Отображение
Греческие буквы	`\alpha` `\beta` `\gamma` `\dots`	$\alpha \beta \gamma \dots$

Обозначение	Код LaTeX	Отображение
Прописные греческие	`\Gamma` `\Delta` `\Omega` `\dots`	$\Gamma\ \Delta\ \Omega\ \dots$
Индексы и степени	`a_i^2`, `a_{i+1}^{n+1}`	$a_i^2,\ a_{i+1}^{n+1}$
Дробь	`\frac{1}{2}`	$\frac{1}{2}$
Корень	`\sqrt{x}`, `\sqrt[n]{x}`	$\sqrt{x},\ \sqrt[n]{x}$
Скобки	`\left( \frac{a}{b} \right)`	$\left(\frac{a}{b}\right)$
Сумма	`\sum_{k=1}^{n}`	$\sum_{k=1}^{n}$
Произведение	`\cdot`, `\times`, `\prod_{i=1}^{m}`	$\cdot,\ \times,\ \prod_{i=1}^{m}$
Интеграл	`\int_{0}^{1} x^2\,dx`	$\int_0^1 x^2\,dx$
Матем. функции	`\sin x`, `\log x`, `\lim_{x\to 0}`	$\sin x, \log x,$ $\lim_{x\to 0}$
Стрелки	`\to`, `\rightarrow`, `\leftarrow`, `\mapsto`, `\Rightarrow`	$\to, \rightarrow, \leftarrow, \mapsto,$ $\Rightarrow$

Красивые скобки

Перед скобкой не обязательно писать `\left` или `\right`, но такой способ позволяет подгонять размер скобочки под содержание. Сравните:

```
(a + (b + (c + d)^4)^3)^2
```

$$(a + (b + (c + d)^4)^3)^2$$

и

```
\left(a + \left(b + \left(c + d\right)^4\right)^3\right)^2
```

$$\left(a + \left(b + (c + d)^4\right)^3\right)^2$$

Важно, при использовании `\left` и `\right` скобки должны составлять валидную пару, иначе будет ошибка. Сравните:

```
a)( %работает
```

$$a)($$

и

```
a\right)\left( % не работает.
```

a\right)\left(% не работает.

В предыдущем примере было показано как писать комментарии для LaTeX.

Gathered

При написании особенно длинных выражений стоит использовать блок `\gathered`, позволяющей переносить строки с помощью `\\`:

```
$$
\begin{gathered}
  F(x) = \sqrt[n]{\frac{x^n - 1}{\sqrt{x + \frac{1}{x}}}}
  + \sum_{k=1}^n \binom{n}{k}  \Gamma\left(k^2 + 1\right)
\sin\left(\alpha  \beta^k\right)
  - \int_{0}^{\frac{\pi}{2}} \ln\left(1 + t^2\right) dt = \\
  = 2\left(\sqrt[n]{\frac{x^n - 1}{\sqrt{x + \frac{1}{x}}}}
  + \sum_{k=1}^n \binom{n}{k}  \Gamma\left(k^2 + 1\right)
\sin\left(\alpha  \beta^k\right)
  - \int_{0}^{\frac{\pi}{2}} \ln\left(1 + t^2\right)
dt\right)\cdot0.5
\end{gathered}
$$
```

$$
\begin{gathered}
  F(x) = \sqrt[n]{\frac{x^n - 1}{\sqrt{x + \frac{1}{x}}}}
  + \sum_{k=1}^n \binom{n}{k} \Gamma\left(k^2 + 1\right)
\sin\left(\alpha \beta^k\right)
  - \int_{0}^{\frac{\pi}{2}} \ln\left(1 + t^2\right) dt = \\
  = 2\left(\sqrt[n]{\frac{x^n - 1}{\sqrt{x + \frac{1}{x}}}}
  + \sum_{k=1}^n \binom{n}{k} \Gamma\left(k^2 + 1\right)
\sin\left(\alpha \beta^k\right)
  - \int_{0}^{\frac{\pi}{2}} \ln\left(1 + t^2\right)
dt\right)\cdot0.5
\end{gathered}
$$

Матрицы и векторы

Для создания векторов и матриц используется блок `pmatrix`, где `&` разделяет столбцы, а `\\` строки:

```
$$
\begin{pmatrix}
  1 & 2 & 3 \\
  4 & 5 & 6 \\
  7 & 8 & 9
\end{pmatrix}
$$
```

$$
\begin{pmatrix}
  1 & 2 & 3 \\
  4 & 5 & 6 \\
  7 & 8 & 9
\end{pmatrix}
$$

Другие полезные вещи

Из полезного стоит упоминуть `\limits`, которое позволяет избегать сплющивания. Сравните сумму в таблице и здесь: $\sum\limits_{k=1}^{n}$.

Если используемой многобуквенной функции (такой как $\sin$) латех "не знает", то такие функции окружаются с помощью `\mathrm{}`: `$\mathrm{cov}(a, b)$` --- $\mathrm{cov}(a, b)$. Букву T, указывающую на транспонированность матрицы/вектора также принято писать через `\mathrm`: `$A^\mathrm{T}$` --- A^T.

Производные записываются с помощью `^\prime`: `F^\prime = f` --- $F^\prime = f$

% --- для комментариев в LaTeX. Если он вам нужен в формуле, то его нужно экранировать с помощью бэкслеша: `\%` --- $\%$.

Импорт изображений

Для импорта изображений можно воспользоваться конструкцией:

```
![%альтернативное название%](%путь%)
```

например:

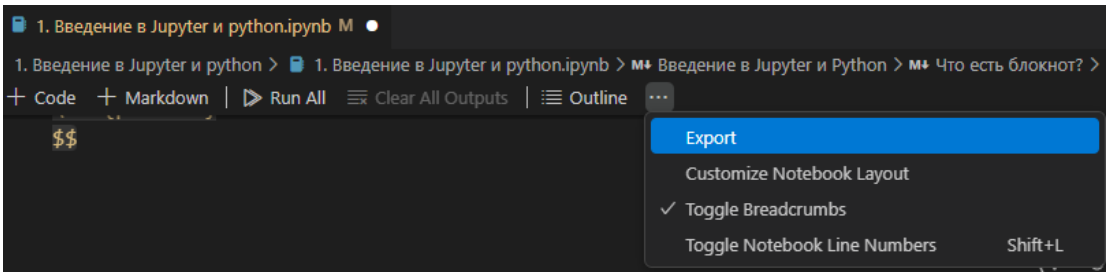
```
![Image](terminal.png)
```

Важно, в пути не должно быть пробелов. Если они есть, то при указании пути их нужно заменить на `%20` (но для человекочитебельности чаще используют относительный путь и замену пробелов на `_` в пути/имени файла).

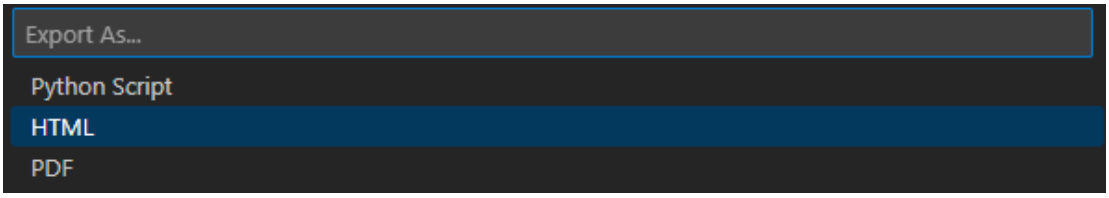
Экспорт

Одна из главных фишек блокнота --- экспорт в html страницу (или pdf, но в html легче). Для этого необходимо найти кнопку `export` (см. изображение 1) и выбрать экспорт (см. изображение 2)

Изображение 1

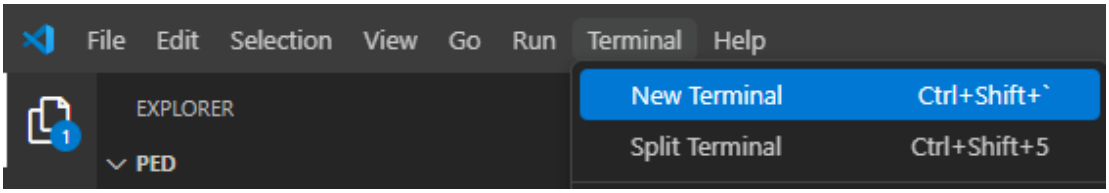


Изображение 2



Если возникла ошибка, то необходимо открыть терминал (например, встроенный как на следующем изображении)

Изображение 3



и выполнить следующие комманды:

```
pip install ipykernel
```

```
pip install -U jupyter_server
```

Python

Основы

Переменные, их инициализация

Итак, -- Python! Python -- это интерпретируемый язык программирования, код выполняется построчно. Python язык со строгой динамической типизацией. Что это означает на практике? В языке есть типы, как и во многих других: `int`, `float`, `string`, `bool`, классы (где класс типа `type`, а объект класса типа самого класса, но нам не обязательно углубляться в такие дебри).

Переменые объявляются без указания типа, он присваивается в зависимости от присваемого значения. В блоке ниже мы объявим 3 переменных типа `int`, `float` и `bool` соответственно. Чтобы выполнить ячейку необходимо нажать `Ctrl+Enter` или треугольник в верхнем левом углу блока кода. В случае успеха слева внизу появится зелёная галочка.

In [37]:
```
x = 5
price = 19.99
is_active = True
```

Важно! Булевые значения true и false в Python пишутся с большой буквы! `True`, `False`.

После выполнения ячейки переменные помещаются в память -- их не нужно будет исполнять каждый раз, когда вы хотите запустить какую-то другую ячейку (в

отличии от скрипта, который каждый раз выполняется с самого начала) и не пропадут, пока не будут удалены или пока ядро не перезагрузится.

Если ячейка содержит неинициализированное имя (будь то переменная, класс или функция), то появится ошибка.

In [3]:
```
c
d = 'AA'
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_20924/3235490055.py in <module>
----> 1 c

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

In [4]:
```
c = 3
c
```

Out[4]: 3

Если ячейка, в которой инициализируется имя, не была выполнена, то инициализация не произойдёт, даже если эта ячейка находится раньше исполняемой (сравните случай `c` и `d`).

In [5]:
```
d
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_20924/3161387801.py in <module>
----> 1 d

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

Константы

Константы в Python объявляются так:

(т.е. никак, их здесь нет). Тут можно написать переменную большими буквами и надеяться, что другие пользователи кода не будут её изменять в своих целях (поскольку из-за соглашения, о котором позже, будут относиться к ней как к константе).

In [30]:
```
IMPORTANT_CONSTANT = 42
```

Строки

Строки (`string`) объявляются с помощью одинарных `''` или двойных кавычек `""`. Здесь нет смыслового деления на "символ" и "строку", оба варианта используются взаимозаменяемо.

In [23]:
```
message = 'Hello '
next_message = "world!"
```

Их комбинацией можно экранировать другие кавычки:

In [16]:
```
horray = 'Он закричал: "Ура, товарищи!"'
print(horray)
CSh = "В C# для инициализации символа используется одинарнарные " \
        "кавычки: char b = 'c';"
print(CSh)
```

```
Он закричал: "Ура, товарищи!"
В C# для инициализации символа используется одинарнарные кавычки: char b = 'c';
```

Выберите один способ и придерживайтесь одного стиля во всём проекте.

Также обратите внимание на перенос строки с помощью `\`, такой способ позволяет не занимать много (горизонтального) места, соблюдая соглашение (о котором позже). Длинную строку можно разбить на несколько подстрок для более комфортной их записи, т.е.

In [14]:
```
actually_not_very_long_string = 'a' 'b'
print(actually_not_very_long_string)
```

```
ab
```

Это был пример для наглядности и в действительности прошлая строка не нуждалась в разбиении, в отличии от `CSh`.

Для разбиения на подстроки, как вы уже могли догадаться, необходимо написать две строки рядом, **не разделяя их запятой**, в случае необходимости перенести строку с помощью `\`.

Escape-последовательности

- `\n` --- перевод строки
- `\r` --- возврат коретки
- `\t` --- таб
- `\"` --- экранирование двойных кавычек
- `\'` --- экранирование одинарной кавычки

In [2]:
```
mayak = '-- Маяковский, а вам правда платят построчно?\n-- H\n\t    E\n\t\t
print(mayak)
```

```
-- Маяковский, а вам правда платят построчно?
-- H
            E
                    T
```

Форматированные строки

Для вывода переменных в строке (с некоторым описанием) можно использовать форматированный вывод:

In [22]:
```
print(f'Это переменная x и она равна {x}')
```

```
Это переменная x и она равна 5
```

Как вы уже заметили, в начале форматированной строки ставится буква `f`, а выводимая переменная находится в фигурных скобочках: `{x}`.

Возвращаемся к переменным

В начале мы объявили, не считая строк, три переменных, давайте посмотрим на их тип:

In [24]:
```
type(x)
```

Out[24]:
```
int
```

In [25]:
```
type(price)
```

Out[25]:
```
float
```

In [26]:
```
type(is_active)
```

Out[26]:
```
bool
```

Как видите, нам если последней строкой стоит возвращаемое выражение (или просто переменная), то она отобразится в выводе, но это работает только с последней строкой:

In [27]:
```
type(price)
type(is_active)
```

Out[27]:
```
bool
```

Наша переменная `x` типа `int`. Прибавим к ней 0.1 и посмотрим что получится.

In [38]:
```
x += 0.1
type(x)
```

Out[38]:
```
float
```

`x` стала типом `float`. Теперь же попробуем прибавить к ней строку:

In [30]:
```
x += 'a'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_20924/249547710.py in <module>
----> 1 x += 'a'

TypeError: unsupported operand type(s) for +=: 'float' and 'str'
```

Получили ошибку. Она возникла из-за того, что, как уже было сказано, Python типизированный язык, и он проверяет допустимость выражений. Сложить целое число с дробным допустимо, но результат получится также дробным, а целое число (или дробное) со строкой уже нельзя --- поскольку они разных типов и для интерпретатора "не очевидно", что вы хотите получить. Для того, чтобы получить строку, нужно конвертировать `x` в строку с помощью `str`:

In [39]:
```
x = str(x)
print(x)
```

```
5.1
```

После этого `x` становится строкой со значением `5.1`, но это строковое, а не численное. Поэтому если вы опять попытаетесь **добавить строку к `x` в позапрошлой ячейке, то ошибки не возникнет**, в отличии от следующего:

In [40]:
```
x += 0.1
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_20924/3942638572.py in <module>
----> 1 x += 0.1

TypeError: can only concatenate str (not "float") to str
```

Обратите внимание на жирную часть в прошлой md-ячейке. Здесь я хотел подчеркнуть, что значения в ячейках берутся из памяти, а не "из истории по порядку". Представим, что есть три ячейки:

1. `c = 4`
2. `c += 2`
3. `c += 4`

Выполнив их по порядку вы получите, что `c = 10`, но если вы после этого снова выполните вторую ячейку, то вы получите `c = 12`, а не `c = 6`, как было в первый раз. Я обращаю на это ваше внимание, поскольку подобные ситуации могут привести к многочасовой отладке, в то время как проблема была на поверхности.

Приведение типов

Чтобы привести строку к числу нужно использовать оператор приведения (на самом деле это конструктор, но для упрощения назовём его так): число в строку с помощью `str()`, строку (или дробное число) в целое число с помощью `int()`, строку/целое число в дробное с помощью `float()`. Обратим внимание, в качестве разделителя дробного числа в строке должна стоять точка, а не запятая. Также нельзя сразу из строки с дробным числом сделать целое число, для этого необходимы дополнительные действия

In [56]:
```
x_float = float(x)
print(f"x_float: {x_float}, type: {type(x_float)}")
x_int = int(x_float)
print(f"x_int: {x_int}, type: {type(x_int)}")
str_from_float = str(x_float)
print(f"str_from_float: {str_from_float}, type: {type(str_from_float)}")
```

```
x_float: 5.1, type: <class 'float'>
x_int: 5, type: <class 'int'>
str_from_float: 5.1, type: <class 'str'>
```

Операторы

Арифметические операторы

- Сложение: `+`,
- Вычитание: `-`,
- Умножение: `*`,
- Деление: `/`,
- Целочисленное деление: `//`
- Остаток от деления: `%`
- Степень: `**`

Важно, в отличии от многих ЯП, в Python при делении двух целочисленных переменных результат **не** целочисленный, поэтому приведения к типу не нужны.

In [41]:
```
div = 5
diver = 7
res = div/diver
res
```

Out[41]:
```
0.7142857142857143
```

Задача. Найти значение `i`

```
i = 5
i = ++i + ++i
```

Ответ: в Python нет инкремента и декремента, поэтому будет ошибка.

В случае необходимости нужно использовать `i += 1`, `i -= 1`.

Операторы сравнения

- Равно: `==`
- Не равно: `!=`
- Больше: `>`
- Меньше: `<`
- Не меньше (больше или равно): `>=`
- Не больше (меньше или равно): `<=`

Логические операторы

- **И**: `and`
- **ИЛИ**: `or`
- **НЕ**: `not`

Рассмотрим их применение:

In [45]:
```
apples = 7
bananas = 3

if bananas > apples:
    print('Бананов больше')
else:
    print('Бананов меньше')
```

```
Бананов меньше
```

In [46]:
```
apples = 6
bananas = 3

if (bananas % 2 == 0) and (apples % 2 == 0):
    print('Можно поделиться с другим')
elif bananas % 2 == 0:
    print('Можно поделиться с другим бананами')
elif apples % 2 == 0:
    print('Можно поделиться с другим яблоками')
else:
    print('Всё наше, и ананасы, и рябчики')
```

```
Можно поделиться с другим яблоками
```

Отступ

Обратите внимание на отступы. По соглашению (о котором позже) необходимо делать четыре пробела (не таб) при начале нового "тела". Гугл-стандарт предпочитает два пробела. Без отступов код работать не будет, отступы в Python вместо фигурных скобочек в других ЯП. Нужно выбрать одно количество пробелов для объявления нового тела и следовать ему во всём проекте.

In [48]:
```
# не будет работать
if bananas > apples:
print('Бананов больше')
```

```
  File "C:\Users\ShkuratD\AppData\Local\Temp/ipykernel_20924/1224007781.py", line 3
    print('Бананов больше')
    ^
IndentationError: expected an indented block after 'if' statement on line 2
```

Базовые структуры данных

Важным отличием структур данных в Python'е от других языков является возможность хранить несколько типов данных в одной структуре. Есть структуры, в которых этого делать нельзя (например, массивы и да, они в Python есть, но чаще используются другие структуры), но мы их касаться не будем. Если знакомы с языком C#, то можете думать о этих структурах, как о коллекциях типа `object`, только без boxing-unboxing.

Список (List)

Список --- это изменяемая коллекция.

Создание

In [117…
```
my_list = [1, 2, 3, 4]
empty_list = []

new_list = [3] * 5 # 5 раз повторить 3
```

In [118…
```
new_list
```

Out[118…
```
[3, 3, 3, 3, 3]
```

Индексация

Индексация начинается с нуля. Отрицательные индексы ведут отсчёт с конца. Индексы с двоеточиями, как `[2:5]`, --- срез начиная от 2 (включая) до 5 (не включая). Эта индексация справедлива для многих других структур данных, в том числе кортежей и строк.

In [41]:
```
print(my_list[0])   # первый элемент
print(my_list[-1])  # последний элемент
print(my_list[1:3]) # От второго до четвёртого, не включая (т.е. второй и третий
print(my_list[::-1]) # В обратном порядке
print(my_list[3:1:-1]) # От четвёртого до второго, не включая (т.е. четвёртый и
print(my_list[::2]) # Каждый второй элемент (начиная с 0)
print(my_list[::-2]) # Каждый второй элемент (начиная с последнего)
```

```
1
4
[2, 3]
[4, 3, 2, 1]
[4, 3]
[1, 3]
[4, 2]
```

Методы

In [68]:
```
my_list.append(5)     # Добавить 5 в конец
print(f'Добавили 5 в конец:\t\t {my_list}')
my_list.remove(2)     # Удалить первый встреченный элемент со значением 2
print(f'Удалили 2:\t\t\t {my_list}')
my_list.pop()         # Удалить и вернуть последний элемент
print(f'Удалили последний элемент:\t {my_list}')
my_list.insert(1, 99) # Вставить значение 99 на позицию 1
print(f'Вставили 99 на позицию 1:\t {my_list}')
my_list.sort()        # Отсортировать список
print(f'Отсортировали список:\t\t {my_list}')
my_list[2] = 7        # Поменяли третий элемент на 7 (изменяемость)
print(f'Поменяли третий элемент на 7:\t {my_list}')
```

```
Добавили 5 в конец:             [1, 2, 3, 4, 5]
Удалили 2:                      [1, 3, 4, 5]
Удалили последний элемент:      [1, 3, 4]
Вставили 99 на позицию 1:       [1, 99, 3, 4]
Отсортировали список:           [1, 3, 4, 99]
Поменяли третий элемент на 7:   [1, 3, 7, 99]
```

Кортеж

Кортеж, в отличии от списка, неизменяемая коллекция. Индексация и срезы работают так же, как и у списка. К кортежам **нельзя** добавить, удалить или изменить элементы после создания. Нужны они для хранения (как ни странно) неизменяемых данных, а также для возвращения из функции сразу нескольких значений.

Создание

Кортежи объявляются в круглых скобках, обратите внимание на случай с одним элементом в кортеже.

In [1]:
```
my_tuple = (1, 2, 3, 4)
single_element_tuple = (1, ) # запятая необходима, иначе скобочки будут приняты
                             # за математический оператор и переменной будет
                             # присвоено целое число 1.
empty_tuple = ()
```

Неизменяемость

Демонстрация неизменяемости:

In [2]:
```
my_tuple[0] = 4
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_19728/1583458910.py in <module>
----> 1 my_tuple[0] = 4

TypeError: 'tuple' object does not support item assignment
```

Строки

Строки тоже не изменяемы. Можете проверить это сами.

Многострочные строки

Мы не рассмотрели ещё один способ создания строки, а если быть точнее --- многострочной строки.

In [3]:
```
multi_line = """Многострочная
строка"""
print(multi_line)
```

```
Многострочная
строка
```

В основном они используются для создании документирования функций и в этом случае называются докстрингами.

Конкатенация

In [33]:
```
s1 = 'A'
s2 = 'b'
s3 = s1+s2
print(f'{s3}\n')

s1 = 'long string'
```

```
s2 = 'gnirts gnol'
s3 = s1[:5] + s2[-6::-1]
print(s3)
```

```
Ab

long string
```

Методы

Рассмотрим несколько методов над строками (вызов метода возвращает новую строку).

In [14]:
```
s1 = 'ОБЪЯСНИТЕ СВОЮ ВЕЛИКОСТЬ'
print(f'Было: {s1}')
s1 = s1.lower()
print(f'Стало: {s1}\n')

s2 = 'объясните свою маленькость'
print(f'Было: {s2}')
s2 = s2.upper()
print(f'Стало: {s2}\n')

s3 = '*o*'
print(f'Было: {s3}')
s3 = s3.replace('*', 'T')
print(f'Стало: {s3}')
s4 = 'Можем изменить не только символ'
print(f'Было: {s4}')
s4 = s4.replace('не только символ', 'ещё и целую подстроку')
print(f'Было: {s4}\n')

str_list = ['This', 'is', 'list', 'of', 'strings']
combine1 = ' '.join(str_list) # объединяем лист строк через пробел
print(combine1)
combine2 = '_'.join(str_list) # объединяем лист строк через _
print(combine2)

str_list2 = combine1.split() # разделитель по умолчанию пробел
print(str_list2) # получаем лист строк
str_list2 = combine2.split() # получаем лист строк из одного элемента
                             # т.к. разделителя здесь нет
print(str_list2)
str_list2 = combine2.split('_') # указываем разделитель, получаем лист строк
print(str_list2)
```

```
Было: ОБЪЯСНИТЕ СВОЮ ВЕЛИКОСТЬ
Стало: объясните свою великость

Было: объясните свою маленькость
Стало: ОБЪЯСНИТЕ СВОЮ МАЛЕНЬКОСТЬ

Было: *o*
Стало: ToT
Было: Можем изменить не только символ
Было: Можем изменить ещё и целую подстроку

This is list of strings
This_is_list_of_strings
['This', 'is', 'list', 'of', 'strings']
['This_is_list_of_strings']
['This', 'is', 'list', 'of', 'strings']
```

Словарь

Структура ключ-значение. Ключи сохраняются в порядке добавления (начиная с Python 3.7), поэтому если вы видите, что словарь неупорядоченная структура, то знайте --- это устаревшая информация.

In [51]:
```
my_dict = {
    'name': 'Алиса',
    'age': 30,
    'city': 'Питер',
    'hobby': 'music',
    7: 'nice', # просто для демонстрации, что можно ключём и число назначить
    36.6: True
}
empty_dict = {}

print(my_dict)
print(empty_dict)
```

```
{'name': 'Алиса', 'age': 30, 'city': 'Питер', 'hobby': 'music', 7: 'nice', 36.6:
True}
{}
```

In [57]:
```
new_dict = {'sample' : 's'}
new_dict[(1, 2, 3)] = 4 # кортеж может быть ключом, потому что он неизменяемый
print(new_dict[(1, 2, 3)])

new_dict[[2, 3, 4]] = 7 # а список не может, потому что он изменяемый
```

```
4
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_19728/700816510.py in <module>
      3 print(new_dict[(1, 2, 3)])
      4
----> 5 new_dict[[2, 3, 4]] = 7 # а список не может, потому что он изменяемый

TypeError: unhashable type: 'list'
```

Работа со словарём

In [52]:
```
print(my_dict['age']) # прочитать

my_dict['age'] += 1   # изменить
print(my_dict['age'])

my_dict['job'] = 'девопс' # Алиса теперь работает девопсом
print(my_dict['job'])


print(my_dict)
# del my_dict['hobby'] # у Алисы больше нет хобби, при повторном вызове выкинет
# my_dict.pop('hobby') # также удаляет ключ, при повторном вызывает ошибку, но!
# hobby = my_dict.pop('hobby') # если ключ есть, то перед удалением возвращается
_ = my_dict.pop('hobby', None) # можно настроить значение по умолчанию, в случае
                               # в этом случае ошибка не происходит. _ -- означ
                               # удаления, так мы показываем, что мы знаем, что
                               # значение, но оно нас не интересует


print(my_dict)

has_job = 'job' in my_dict
has_hobby = 'hobby' in my_dict
print(f'У Алисы есть работа: {has_job}')
print(f'У Алисы есть хобби: {has_hobby}')
```

```
30
31
девопс
{'name': 'Алиса', 'age': 31, 'city': 'Питер', 'hobby': 'music', 7: 'nice', 36.6:
True, 'job': 'девопс'}
{'name': 'Алиса', 'age': 31, 'city': 'Питер', 7: 'nice', 36.6: True, 'job': 'дево
пс'}
У Алисы есть работа: True
У Алисы есть хобби: False
```

In [55]:
```
print(my_dict.keys()) # все ключи
print('\n')

print(my_dict.values()) # все значения
print('\n')

print(my_dict.items()) # все пары
print('\n')

print(my_dict.get('city')) # вернуть город, если есть
print(my_dict.get('hobby')) # вернуть хобби, если есть (его нет, поэтому вернёт
print('\n')
```

```
dict_keys(['name', 'age', 'city', 7, 36.6, 'job'])


dict_values(['Алиса', 31, 'Питер', 'nice', True, 'девопс'])


dict_items([('name', 'Алиса'), ('age', 31), ('city', 'Питер'), (7, 'nice'), (36.
6, True), ('job', 'девопс')])


Питер
None
```

Обход словаря:

In [127…
```
for key, item in my_dict.items():
    print(f'{key}: {item}')
```

```
name: Алиса
age: 31
city: Питер
7: nice
36.6: True
job: девопс
```

Множество

Работает точно так, как и множество в математике --- набор элементов.

Создание

In [65]:
```
my_set = {1, 2, 3}
empty_set = set() # {} -- создаёт пустой словарь, не путайте
```

In [60]:
```
my_set
```

Out[60]:
```
{1, 2, 3}
```

Добавление/удаление

In [66]:
```
print(f'Исходный: {my_set}')
my_set.add(4)
print(f'Добавили 4: {my_set}')
my_set.remove(2)
print(f'Убрали 2: {my_set}')
my_set.add(4)
print(f'Снова добавили 4: {my_set} (ничего не изменилось, множество же)')
```

```
Исходный: {1, 2, 3}
Добавили 4: {1, 2, 3, 4}
Убрали 2: {1, 3, 4}
Снова добавили 4: {1, 3, 4} (ничего не изменилось, множество же)
```

Нельзя добавлять нехэшируемые элементы.

In [63]:
```
my_set.add((1, 2, 3)) # кортеж можно
print(f'Добавили кортеж: {my_set}')
```

```
print(f'При попытке добавить список появится ошибка:')
my_set.add([1, 2, 3]) # список нельзя
print(f'Этой строки не будет')
```

```
Добавили кортеж: {1, 3, 4, (1, 2, 3)}
При попытке добавить список появится ошибка:
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_19728/2309722673.py in <module>
      3
      4 print(f'При попытке добавить список появится ошибка:')
----> 5 my_set.add([1, 2, 3]) # список нельзя
      6 print(f'Этой строки не будет')

TypeError: unhashable type: 'list'
```

Операции над множествами

In [67]:
```
set1 = {1, 2, 3}
set2 = {3, 4, 5}

union_set = set1 | set2
print(f'Объединение: {union_set}')

inter_set = set1 & set2
print(f'Пересечение: {inter_set}')

diff_set = set1 - set2
print(f'Разность: {diff_set}')

sym_diff_set = set1 ^ set2
print(f'Симметрическая разность: {sym_diff_set}')
```

```
Объединение: {1, 2, 3, 4, 5}
Пересечение: {3}
Разность: {1, 2}
Симметрическая разность: {1, 2, 4, 5}
```

Суть множества в быстрой проверке наличия элемента во множестве:

In [68]:
```
print(3 in union_set)
print(3 in sym_diff_set)
```

```
True
False
```

`zip` /unzip

`zip` позволяет итерироваться по нескольким [итерируемым] объектам сразу.

In [1]:
```
numbers = [1, 2, 3]
letters = ['a', 'b', 'c']

for num, let in zip(numbers, letters):
    print(num, let)
```

```
1 a
2 b
3 c
```

Можно добавлять не два списка, а больше. Да и не обязательно, чтобы это были списки.

In [10]:
```
coins = (0.01, .02, .05)
for num, let, coin in zip(numbers, letters, coins):
    print(num, let, coin)
```

```
1 a 0.01
2 b 0.02
3 c 0.05
```

Списки разной длины

Рассмотрим случай, когда у нас списки разной длины:

In [19]:
```
numbers = [1, 2, 3, 4]
letters = ['a', 'b']

zipped = zip(numbers, letters)
print(list(zipped))
```

```
[(1, 'a'), (2, 'b')]
```

Как видим `zip` остановился, когда закончился самый маленький список.

Unzip

Мы можем распаковать то, что упаковали.

In [21]:
```
zipped = zip(numbers, letters)    # зачем заново объявляем -- объясним уже в сле
numbers_, letters_ = zip(*zipped) # _ в конце имени просто для наглядности
                                  # что означает * -- рассмотрим позже
print(numbers_)
print(letters_)
```

```
(1, 2)
('a', 'b')
```

Итераторы (опционально)

`zip` создаёт итератор кортежей, который можно преобразовать в нужную структуру данных. (Итераторы встречаются не только здесь, но для базового уровня, пожалуй, о них достаточно просто знать)

In [14]:
```
zipped = zip(numbers, letters)
print(type(zipped))
print(list(zipped))
```

```
<class 'zip'>
[(1, 'a'), (2, 'b'), (3, 'c')]
```

Обратите внимание на то, что если из этого же итератора мы попытаемся сделать кортеж:

In [15]:
```
print(tuple(zipped))
```

()

то у нас ничего не получится, потому что итератор "закончился". "Обновить" его нельзя, только создать заново.

In [17]:
```
zipped = zip(numbers, letters)
print(list(zipped))
```

```
[(1, 'a'), (2, 'b'), (3, 'c')]
```

Управляющие конструкции

`if`

Смотрите блок с логическими операторами (ссылка не работает в VS Code, поэтому ориентируетесь либо по outline (оглавлению) слева, либо можете открыть html файл в браузере и нажать там).

Тернарный оператор сокращает запись, вместо:

In [ ]:
```
bike = True
if bike:
    wheels = 2
else:
    wheels = 4
```

пишут:

In [ ]:
```
wheels = 2 if bike else 4
print(wheels)
```

```
2
```

Кстати, это хороший повод поговорить об области видимости и особенностях интерпретатора.

Область видимости

In [74]:
```
if True:
    cab = True
else:
    print(non_defined_var)

cab
```

Out[74]:
```
True
```

В примере выше мы видим две вещи.

1. В компилируемых (и может в Go и некоторых интерпретируемых, где перед запуском проходит проверка кода) обращение к `non_defined_var` вызвало бы ошибку, но поскольку мы никогда не попадём в другую ветку, то Python это "пропускает мимо". `if True` было выбрано для наглядности, но всё равно об этом важно знать, поскольку если у вас есть условие, которое в 99.99% случаев истинно, то вы можете случайно сделать ошибку в альтернативном случае и не

узнаете о ней, пока не наткнётесь. Код ниже --- демонстрация такого случая. Можете выполнить эту ячейку (с кодом) несколько раз и может быть вам повезёт наткнуться на баг (за 1000 "круток" шанс словить баг будет несколько больше 9.5%).

2. Не смотря на то, что `cab` объявлена в блоке if, после этого блока мы всё равно её видим (но не до), поскольку блок `if` выполнился. В случае ниже `cab2` не выкинет ошибку, если блок `if` не исполнится, но только потому что скрипт сломается раньше. Если заменить код в `else`-блоке на валидный, то тогда `cab2` скажет, что её не определили. Может быть дальше мы ещё раз вернёмся к теме области видимости.

In [75]:
```
import random

if random.random() < 0.9999:
    cab2 = True
else:
    print(non_defined_var)

cab2
```

```
True
```

В блоке выше вы также увидели *импорт модуля* `random`. Об этом мы ещё поговорим потом.

Циклы

`range`

Для начала нужно разобраться с тем, что такое `range`, а это неизменяемая последовательность, причём ленивая (т.е. вычисления происходят не сразу, а на лету), что помогает оптимизировать использование памяти.

In [2]:
```
range(5) # создаст последовательность от 0 до 5, не включая 5 [0;5)
```

Out[2]:
```
range(0, 5)
```

In [3]:
```
range(3, 5) # Можно задать начальный элемент
```

Out[3]:
```
range(3, 5)
```

In [4]:
```
range(1, 11, 2) # Также можно задать шаг
```

Out[4]:
```
range(1, 11, 2)
```

Рассмотрим теперь на практике, вместе с циклом

`for`

Самый простой аналог `for(int i = 0; i < N; i++)` в Python:

In [78]:
```
N = 5
for i in range(N): # не дойдёт до 5
    print(i)
```

```
i
```

```
0
1
2
3
4
```

Out[78]: 4

In [5]:
```
for i in range(3, 5):
    print(i)
```

```
3
4
```

In [6]:
```
for i in range(1, 11, 2):
    print(i)
```

```
1
3
5
7
9
```

In [11]:
```
for i in range(-3, 2): # отрицательные числа также можно использовать
    print(i)
```

```
-3
-2
-1
0
1
```

In [13]:
```
for i in range(3, -2, -1): # и в обратную сторону тоже можно идти
    print(i)
```

```
3
2
1
0
-1
```

" `foreach` "

Аналог `foreach(var c in collection)` .

In [79]:
```
fruits = ['apple', 'banana', 'cherry'] # не обязательно список, можно использова
                                       # и другие структуры
for fruit in fruits:
    print(fruit)

fruit # заметьте, `fruit` продолжает существовать вне цикла, как и i
```

```
apple
banana
cherry
```

Out[79]: 'cherry'

Опять обратим внимание на `range` и его ленивые вычисления. Если сама последовательность нам не нужна, то обход цикла

`while`

Тело выполняется пока верно условие.

In [80]:
```
n = 5
while n > 0:
    print(n)
    n -= 1
```

```
5
4
3
2
1
```

`break` и `continue`

`break` --- закончить выполнение цикла, `continue` --- начать следующую итерацию.

In [81]:
```
for i in range(10):
    if i == 3:
        break # останавливаем когда `i` равен трём
    print(i)
```

```
0
1
2
```

In [82]:
```
for i in range(5):
    if i == 2:
        continue # пропускаем 2
    print(i)
```

```
0
1
3
4
```

Генератор списков (list conpehensions) и не только

Пишем красиво, вместо:

In [84]:
```
numbers = []
for i in range(10):
    numbers.append(i**2)

numbers
```

Out[84]: `[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]`

пишем:

In [86]:
```
numbers = [i**2 for i in range(10)] # а ещё это быстрее
numbers
```

Out[86]:
```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

- `[]` --- квадратные скобки означают создание списка
- `i**2` --- возведение в квадрат
- `for i in range(10)` --- по сути проход цикла. Можем вместо генератора использовать список:

In [90]:
```
numbers = [i/2 for i in numbers]
numbers
```

Out[90]:
```
[0.0, 0.5, 2.0, 4.5, 8.0, 12.5, 18.0, 24.5, 32.0, 40.5]
```

Можем добавить условие:

In [87]:
```
even_squares = [i**2 for i in range(10) if i%2 == 0]
even_squares
```

Out[87]:
```
[0, 4, 16, 36, 64]
```

Случай с **ИЛИ** пишем через тернарное выражение

In [89]:
```
new_powers = [i**2 if i%2 == 0 else i**3 for i in range(10)]
new_powers
```

Out[89]:
```
[0, 1, 4, 27, 16, 125, 36, 343, 64, 729]
```

Также можем создавать кортежи:

In [97]:
```
numbers = tuple(i**2 for i in range(10)) # нужно указывать, что это кортеж
numbers
```

Out[97]:
```
(0, 1, 4, 9, 16, 25, 36, 49, 64, 81)
```

Словари:

In [98]:
```
numbers = {i: i**2 for i in range(10)} # используем {} и :
numbers
```

Out[98]:
```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```

Множества:

In [99]:
```
numbers = {i**2 for i in range(10)} # используем {}
numbers
```

Out[99]:
```
{0, 1, 4, 9, 16, 25, 36, 49, 64, 81}
```

Функции

Синтаксис:

```
def имя_функции(параметры):
    Тело функции
    ...
    return результат
```

Если результат возвращать не нужно (`void`), то `return` можно не писать, в таком случае ф-ция вернёт `None` .

In [103…
```
def my_final_message(name, message):                        # объвление функции
    print(f'{message} my final message. Goodbye, {name}') # тело функции

my_final_message('%player_name%', 'Change da world')      # вызов функции
```

```
Change da world my final message. Goodbye, %player_name%
```

Именованные аргументы

Именованные аргументы позволяют явно указать какому параметру какое значение передаётся (в библиотеках, которые мы будем использовать, есть функции с очень большим количеством аргументов, атк что это довольно важный момент).

In [104…
```
my_final_message(message='Change da world', # обратите внимание, здесь = не окру
                 name='Jojo')
```

```
Change da world my final message. Goodbye, Jojo
```

Аргументы по умолчанию

Аргументы по умолчанию стоят после позиционных аргументов.

```
def имя_функции(арг1, ..., аргN, умолч_арг1=%значение%, ...,
умолч_аргN=%значение_N%)
```

In [106…
```
def my_final_message(name, message='Change da world'):
    print(f'{message} my final message. Goodbye, {name}')

my_final_message('%player_name%') # передаём только один аргумент
```

```
Change da world my final message. Goodbye, %player_name%
```

Кортеж аргументов (`*args`)

Объединение аргументов в кортеж. Все неименнованные аргументы были собраны в кортеж (см. ниже).

In [108…
```
def tuple_ex(*args):
    print(args)

tuple_ex('a', 2, True)
```

```
('a', 2, True)
```

Здесь первый аргумент позиционный, остальные неименнованные собираются в кортеж.

In [109…
```
def tuple_ex(arg1, *args):
    print(arg1)
    print(args)

tuple_ex('a', 2, True)
```

```
a
(2, True)
```

Смешивать кортеж и именнованные так просто нельзя:

```
tuple_ex('a', 2, arg1=True)
```

выдаст ошибку о том, что аргументу arg1 пытались присвоить несколько значений,

```
tuple_ex(arg1=True, 'a', 2)
```

также не сработает, поскольку здесь позиционные аргументы идут после именнованных.

Пример использований:

In [116…
```
def power(pow_to, *args):
    for arg in args:
        print(arg**pow_to)

power(2, 4, 5, 6, 7)
```

```
16
25
36
49
```

Да, это можно было бы сделать и так:

In [120…
```
def power2(pow_to, args):
    for arg in args:
        print(arg**pow_to)

list_ = [4, 5, 6, 7]
power2(2, list_)
```

```
16
25
36
49
```

Но и так мы можем тоже сделать:

In [121…
```
power(2, *list_)
```

```
16
25
36
49
```

Звёздочка перед списком "раскрывает" его и он передаётся как несколько аргументов.

Так делать **нельзя**:

```
def foo(*args1, *args2)
```

Передача именнованых аргументов с помощью словаря/ `**kwargs` .

In [124…
```
def tuple_ex(arg1, **kwargs):
    print(arg1)
    print(kwargs)

tuple_ex('a', arg2=2, arg3=True) # два последних аргумента были переданы как сло
tuple_ex(arg1='a', arg2=2, arg3=True) # первый был именнованным, 2 и 3 -- словар
tuple_ex(arg2=2, arg3=True, arg1='a') # последний именнованный, 1 и 2 -- словарь
```

```
a
{'arg2': 2, 'arg3': True}
a
{'arg2': 2, 'arg3': True}
a
{'arg2': 2, 'arg3': True}
```

Использование:

In [136…
```
def set_settings(**settings):
    if 'default' in settings:
        if settings['default']:
            print('Settings were set to default')
            return

    for key, value in settings.items():
        print(f'{key} was set to {value}')

set_settings(resolution='1920x1080',
             graphics='high',
             level='hard')

print()

settings = {
    'resolution': '640x480',
    'graphics': 'low',
    'level': 'easy'
}

set_settings(**settings)

print()

set_settings(default=True)
```

```
resolution was set to 1920x1080
graphics was set to high
level was set to hard

resolution was set to 640x480
graphics was set to low
level was set to easy

Settings were set to default
```

Объединение `*args` и `**kwargs`

Мы можем объединять их в сигнатуре функции.

In [ ]:
```python
def func(*args, **kwargs):
    print(args)
    print(kwargs)

func(1, 2, 3, 4, 5, 6)
print()

func(1, 2, 3, var1=4, var2=5, var3=6)
print()

func(var1=1, var2=2, var3=3, var4=4, var5=5, var6=6)
print()

args = (1, 2, 3)
func(*args, var1=4, var2=5, var3=6)
print()

args = [1, 2, 3]
func(*args, var1=4, var2=5, var3=6)
print()

kwargs = {'var1': 4, 'var2': 5, 'var3': 6}
func(*args, **kwargs)
print()

func(*(kwargs.items()))
```

```
(1, 2, 3, 4, 5, 6)
{}

(1, 2, 3)
{'var1': 4, 'var2': 5, 'var3': 6}

()
{'var1': 1, 'var2': 2, 'var3': 3, 'var4': 4, 'var5': 5, 'var6': 6}

(1, 2, 3)
{'var1': 4, 'var2': 5, 'var3': 6}

(1, 2, 3)
{'var1': 4, 'var2': 5, 'var3': 6}

(1, 2, 3)
{'var1': 4, 'var2': 5, 'var3': 6}

(('var1', 4), ('var2', 5), ('var3', 6))
{}
```

Объясните последний результат.

Строгий контроль за аргументами (опционально)

Если объявить сигнатуру функции следующим образом:

```
def foo(a, b, /, c, d=2, *, e, f=4)
```

то аргументы

- `a` и `b` мы *должны* передать только позиционно, мы *не можем* их передать по имени,
- `c` мы можем передать позиционно, можем передать по имени, но в любом случае мы *должны* её передать,
- `e` мы *должны* передать по имени,
- `d` и `f` мы *можем* не передавать.

Можете попробовать потренироваться на функции ниже.

In [137…
```
def foo(a, b, /, c, d=2, *, e, f=4):
    return
```

In [ ]:
```
foo(2, 3, 4, 5, 6, 7)
```

Docstrings

Для того, чтобы все знали, что делает ваша функция к ней необходимо написать документацию. Делается это с помощью докстрингов.

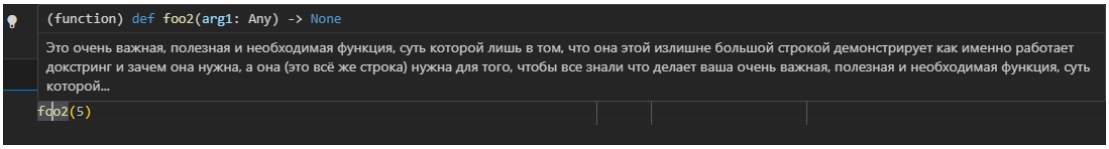
In [154…
```
def foo2(arg1):
    """Это очень важная, полезная и необходимая функция, суть которой
    лишь в том, что она этой излишне большой строкой демонстрирует
    как именно работает докстринг и зачем она нужна, а она (это всё же
    строка) нужна для того, чтобы все знали что делает ваша очень
```

```
    важная, полезная и необходимая функция, суть которой..."""

    print(arg1)
```

Теперь вы будете видеть комментарий к этой функции, когда наведёте на неё мышкой.

In [ ]:
```
foo2(5)
```



λ-функции (опционально)

Синтаксис: `lambda аргументы: выражение`

Пример:

In [22]:
```
square = lambda x: x**2
print(square(5))
```

```
25
```

Используется для объявления и вызовов небольших функций (а также с помощью них можно реализовывать всякие паттерны, например, стратегия)).

`map` (опционально)

Функция `map` применяет переданную функцию к элементам итерируемого объекта, возвращая *итератор* результатов.

Пример:

In [23]:
```
def double(x):
    return x*2

numbers = [1, 2, 3, 4, 5]
res = map(double, numbers) # передаём функцию и списко
print(list(res)) # обращаем внимание на обвёртку в list

res2 = map(lambda x: x*3, numbers) # вариант через lambda-функцию
print(list(res2))
```

```
[2, 4, 6, 8, 10]
[3, 6, 9, 12, 15]
```

`filter` (опционально)

`filter` нужен для, как ни странно, для фильтрации.

Идея та же, что и с прошлым методом, только отображение должно быть в `True` / `False`. Пример:

In [29]:
```
def is_even(x):
    return x % 2 == 0

res = filter(is_even, numbers)
print(list(res))

res2 = filter(lambda x: x % 3 == 0, numbers)
print(list(res2))
```

```
[2, 4]
[3]
```

Соглашение

Я упомянул некоторое соглашение и сказал, что мы вернёмся к нему позже. Это соглашение называется `PEP-8` и ~~его нужно выучить, зазубрить, зарубить на носу, намотать на ус, чтобы вас ночью разбуди --- от зубов отскакивало, а ещё вы обязаны его соблюдать под страхом страшной кары~~ с ним нужно будет ознакомиться и придерживаться так сильно, как только можете. Некоторые моменты (в принципе самые главные и распространённые) вы уже видели по ходу этого документа, а именно:

- именование функций и переменных с маленькой буквы,
- слова в именах разделяются с помощью подчёркивания `_` (`snake_case`),
- "константы" которых в Python нет, пишутся большими буквами
- переменным давать осмысленные имена, не злоупотреблять сокращениями,
- каждый `import` --- это отдельная строка, сначала импорты стандартной библиотеки, потом сторонних проектов, а потом внутри проекта (здесь ещё не было, будет в дальнейшем)
- НЕ ИСПОЛЬЗОВАТЬ `import *` и `from %name% import *` (`*` --- означает всё),
- документировать функции,
- не писать несколько действий в одну строку
- новый отступ --- четыре пробела, вместо таба или любого другого количества пробелов,
- пробелы возле знаков арифметических операций (некоторые позволяют не ставить возле `*` и `**`),
- пробелы не ставятся перед запятыми и двоеточиями, после `(` и перед `)` (в русском языке, кстати, те же правила, не надо ставить пробел перед запятой),
- отсутствие пробелов возле именнованных аргументов/аргументов по умолчанию,
- не меняйте привычки в одном проекте (если используете одинарные кавычки по умолчанию, то только их и используйте, и т.д.),
- строки кода не должны быть больше 79 символов, их нужно разбивать на несколько строк с помощью `\` и/или выносить логические куски в отдельные функции.

Это важно, это нужно соблюдать для всеобщего счастья. Прочитать стандарт за раз будет больно и многого с первого разу не схватишь, но всё же крайне рекомендую

ознакомиться. Я оставляю ссылку на официальный сайт, вы можете найти сами русскую версию, а также краткий пересказ на ваших любимых видеохостингах.

https://peps.python.org/pep-0008/

Для любознательных

Прочитать про итераторы и генераторы. А ещё про области видимости (scope), это важная тема, но по причинам пока её рассмотривать не будет (а может и вообще).