

# Информационные технологии

## ОСНОВЫ программирования на Python 3



# Лекция 4.

## Инструкции и операторы

- Операторы языка Python
- Условные операторы
- Циклы
- Последовательности

# Инструкции

Инструкция	Роль	Пример
Присваивание	Создание ссылок	<i>a, b = "раз", "два"</i>
Вызовы	Запуск функций	<i>log.write("текст, ввёл\n")</i>
<i>print</i>	Вывод объектов	<i>print("Печать", joke)</i>
<i>raise</i>	Возбуждение исключений	<i>raise endSearch(location)</i>
<i>try/except/finally</i>	Обработка исключений	<i>try:     action() except:     print("исключение")</i>
<i>assert</i>	Отладочные проверки	<i>assert x &lt; 0, "x мал"</i>
<i>import</i>	Доступ к модулям	<i>import sys</i>
<i>from</i>	Доступ к атрибутам модуля	<i>from sys import stdin</i>
<i>class</i>	Создание объектов	<i>class Subclass(Superclass):     staticData = []     def method(self): pass</i>

# Инструкции

Инструкция	Роль	Пример
<i>if/elif/else</i>	Операция выбора	<i>if "python" in text: print(text)</i>
<i>for/else</i>	Обход последовательности в цикле	<i>for x in mylist: print(x)</i>
<i>while/else</i>	Циклы общего назначения	<i>while x &gt; y: y+=2</i>
<i>break</i>	Прерывание цикла	<i>while True: if flexit: break</i>
<i>continue</i>	Переход в начало цикла	<i>while True: if flrun: continue</i>
<i>def</i>	Создание функций и методов	<i>def f(a, b, c=1, *d): return a + b + c + d[0]</i>
<i>return</i>	Возврат результата	
<i>yield</i>	Функции-генераторы	<i>def gen(n): for i in n, yield i*2</i>
<i>pass</i>	Пустая инструкция заполнитель	<i>while True: pass</i>

# Инструкции

Инструкция	Роль	Пример
<i>global</i>	Пространства имен	<i>x = "old"</i> <i>def function():</i> <i>    global x, y; x = "new"</i>
<i>nonlocal</i>		<i>x = "old"</i> <i>def function():</i> <i>    nonlocal x, y; x = "new"</i>
<i>del</i>	Удаление ссылок	<i>del data[k]</i> <i>del data[i:j]</i> <i>del obj.attr</i> <i>del variable</i>
<i>exec</i>	Запуск фрагментов программного кода	<i>exec "import " + modName</i> <i>exec code in gdict, ldict</i>
<i>with/as</i>	Менеджеры контекста	<i>with open("data") as myfile:</i> <i>    process(myfile)</i>

# Операторы Python

- Арифметические операторы
- Операторы сравнения (реляционные)
- Операторы присваивания
- Побитовые операторы
- Логические операторы
- Операторы членства (Membership operators)
- Операторы тождественности (Identity operators)

# Арифметические операторы

<b>+</b>	Сложение	$15 + 5 = 20$ $20 + -3 = 17$ $13.4 + 7 = 20.4$
<b>-</b>	Вычитание	$15 - 5 = 10$ $20 - -3 = 23$ $13.4 - 7 = 6.4$
<b>*</b>	Умножение	$5 * 5 = 25$ $7 * 3.2 = 22.4$ $-3 * 12 = -36$
<b>/</b>	Деление	$5 / 2 = 2.5$ (В Python 2.x версии результат будет 2) $5.0 / 2 = 2.5$ (Хотя бы один операнд должен быть <i>float</i> )
<b>%</b>	Остаток	$6 \% 2 = 0$ $7 \% 2 = 1$ $13.2 \% 5 = \sim 3.2$
<b>**</b>	Возведение в степень	$5 ** 2 = 25$ $2 ** 3 = 8$ $-3 ** 2 = -9$ (степень приоритетнее знака)
<b>//</b>	Целочисленное деление	$12 // 5 = 2$ $4 // 3 = 1$ $25 // 6 = 4$

# По вычислению корней

□ Корень кубический вычисляется исходя из

Например, вычислим  $\sqrt[3]{-1}$ .

Имеем

$$\sqrt[3]{-1} = \sqrt[3]{1 \cdot (\cos \pi + i \sin \pi)} = \sqrt[3]{1} \cdot \left( \cos \frac{\pi + 2k\pi}{3} + i \sin \frac{\pi + 2k\pi}{3} \right).$$

Давая  $k$  значения 0, 1, 2, получим три значения  $\sqrt[3]{-1}$

$$\text{при } k = 0 \quad \omega_1 = \frac{1}{2} + i \frac{\sqrt{3}}{2},$$

$$\text{при } k = 1 \quad \omega_2 = -1,$$

$$\text{при } k = 2 \quad \omega_3 = \frac{1}{2} - i \frac{\sqrt{3}}{2}.$$



# Пример корня

$$-27^{**} (1/3) \quad \# \quad -3.0$$

$$(-27)^{**} (1/3) \quad \# \quad (1.5+2.598076211353316j)$$

Проверка:

$$(1.5 + 2.598076211353316j)^{**}3 \quad \# \quad (-27+0j)$$

Сопряжённый корень проверка:

$$(1.5 - 2.598076211353316j)^{**}3 \quad \# \quad (-27+0j)$$

Аналогично

$$a = -27$$

$$a^{**} (1/3) \quad \# \quad (1.5+2.598076211353316j)$$

# Операторы сравнения

<b>==</b>	Проверяет равны ли оба операнда. Если да, то условие становится истинным.	5 == 5 даёт <b>True</b> True == False даёт <b>False</b> "hello" == "hello" даёт <b>True</b>
<b>!=</b>	Проверяет равны ли оба операнда. Если нет, то условие становится истинным*.	12 != 5 даёт <b>True</b> False != False даёт <b>False</b> "hi" != "Hi" даёт <b>True</b>
<b>&gt;</b>	Проверяет больше ли значение левого операнда, чем значение правого. Если да, то условие становится истинным.	5 > 2 даёт <b>True</b> . True > False даёт <b>True</b> . "A" > "B" даёт <b>False</b> .
<b>&lt;</b>	Проверяет меньше ли значение левого операнда, чем значение правого. Если да, то условие становится истинным.	3 < 5 даёт <b>True</b> . True < False даёт <b>False</b> . "A" < "B" даёт <b>True</b> .
<b>&gt;=</b>	Проверяет больше или равно значение левого операнда, чем значение правого. Если да, то условие становится истинным.	1 >= 1 даёт <b>True</b> . 23 >= 3.2 даёт <b>True</b> . "C" >= "D" даёт <b>False</b> .
<b>&lt;=</b>	Проверяет меньше или равно значение левого операнда, чем значение правого. Если да, то условие становится истинным.	4 <= 5 даёт <b>True</b> . 0 <= 0.0 даёт <b>True</b> . -0.001 <= -36 даёт <b>False</b> .

\* – оператор неравенства "**<>**" больше не применяется

# Составные операторы сравнения

$x = 5$

$2 < x < 10$       *# True*

$10 < x < 20$       *# False*

$x < 10 < x * 10 < 100$       *# True*

$10 > x <= 9$       *# True*

$5 == x > 4$       *# True*

# Операторы присваивания

<b>=</b>	Присваивает значение правого операнда левому.	$d = 24$
<b>+=</b>	Прибавит значение правого операнда к левому и присвоит эту сумму левому операнду.	$c = 5$ $a = 2$ $c += a$ то же: $c = c + a$ # $c=7$
<b>-=</b>	Отнимает значение правого операнда от левого и присваивает результат левому операнду.	$c = 5$ $a = 2$ $c -= a$ то же: $c = c - a$ # $c=3$
<b>*=</b>	Умножает правый операнд с левым и присваивает результат левому операнду.	$c = 5$ $a = 2$ $c *= a$ то же: $c = c * a$ # $c=10$
<b>/=</b>	Делит левый операнд на правый и присваивает результат левому операнду.	$c = 10$ $a = 2$ $c /= a$ то же: $c = c / a$ # $c=5$

# Операторы присваивания

<b>%=</b>	Делит по модулю операнды и присваивает результат левому.	$c = 5$ $a = 2$ $c \% = a$ то же: $c = c \% a$ # $c=1$
<b>**=</b>	Возводит в левый операнд в степень правого и присваивает результат левому операнду.	$c = 3$ $a = 2$ $c ** = a$ то же: $c = c ** a$ # $c=9$
<b>//=</b>	Производит целочисленное деление левого операнда на правый и присваивает результат левому операнду.	$c = 11$ $a = 2$ $c //= a$ то же: $c = c // a$ # $c=5$

# Побитовые операторы

$a = 0b0011\ 1100$  ( $60_{10}$ )

$b = 0b0000\ 1101$  ( $13_{10}$ )

<b>&amp;</b>	Бинарное "И". Копирует бит в результат только если бит присутствует в обоих операндах.	$a \ \& \ b = 0000\ 1100$ ( $12_{10}$ )
<b> </b>	Бинарный "ИЛИ" Копирует бит, если тот присутствует в хотя бы в одном операнде.	$a \   \ b = 0011\ 1101$ ( $61_{10}$ )
<b>^</b>	Бинарный "Исключающий ИЛИ" оператор копирует бит только если бит присутствует в одном из операндов, но не в обоих сразу.	$a \ ^ \ b = 0011\ 0001$ ( $49_{10}$ )

# Побитовые операторы

$a = 0b0011\ 1100$  ( $60_{10}$ )

$b = 0b0000\ 1101$  ( $13_{10}$ )

<b>~</b>	Бинарный комплиментарный оператор. Является унарным. Меняет биты на обратные	$\sim a = 1100\ 0011$ ( $-61_{10}$ )
<b>&lt;&lt;</b>	Побитовый сдвиг влево. Значение левого операнда "сдвигается" влево на количество бит указанных в правом операнде.	$a \ll 2$ даст 1111 0000 ( $240_{10}$ ) $a \ll 5$ даст 111 1000 0000 ( $1920_{10}$ )
<b>&gt;&gt;</b>	Побитовый сдвиг вправо. Значение левого операнда "сдвигается" вправо на количество бит указанных в правом операнде.	$a \gg 2$ даст 0000 1111 ( $15_{10}$ ) $a \gg 4$ 11 ( $3_{10}$ )

# Логические операторы

<b>and</b>	Логический оператор "И". Условие будет истинным, если оба операнда истина.
<b>or</b>	Логический оператор "ИЛИ". Если хотя бы один из операндов истинный, то и все выражение будет истинным.
<b>not</b>	Логический оператор "НЕ". Изменяет логическое значение операнда на противоположное.



# Операторы членства

<b>in</b>	Возвращает истину, если элемент присутствует в последовательности, иначе возвращает ложь.	<i>"cad" in "cadillac" (True).</i> <i>1 in [2,3,1,6] (True).</i> <i>"hi" in {"hi":2, "bye":1} (True)</i> <i>2 in {"hi":2, "bye":1} (False)</i> (в словарях проверяется наличие в ключах).
<b>not in</b>	Возвращает истину, если элемента нет в последовательности.	Результаты противоположны результатам оператора <i>in</i> .

# Операторы тождественности

<b>is</b>	Возвращает истину, если оба операнда указывают на один объект.	$x \text{ is } y$ вернет истину, если $\text{id}(x)$ будет равно $\text{id}(y)$ .
<b>is not</b>	Возвращает ложь, если оба операнда указывают на один объект.	$x \text{ is not } y$ , вернет истину если $\text{id}(x)$ не равно $\text{id}(y)$ .

# Приоритет операторов

<b>**</b>	Возведение в степень
<b>~ + -</b>	Комплиментарный оператор
<b>* / % //</b>	Умножение, деление, деление по модулю, целочисленное деление.
<b>+ -</b>	Сложение и вычитание.
<b>&gt;&gt; &lt;&lt;</b>	Побитовый сдвиг вправо и побитовый сдвиг влево.
<b>&amp;</b>	Бинарный "И".
<b>^  </b>	Бинарный "Исключительное ИЛИ" и бинарный "ИЛИ"
<b>&lt;= &lt; &gt; &gt;=</b>	Операторы сравнения
<b>&lt;&gt; == !=</b>	Операторы равенства
<b>= %= /= //=-= += *= **=</b>	Операторы присваивания
<b>is is not</b>	Тождественные операторы
<b>in not in</b>	Операторы членства
<b>not or and</b>	Логические операторы

# Условный оператор

- В Python инструкция *if* выбирает, какое действие следует выполнить.
- Это основной инструмент выбора в Python, который отражает большую часть логики программы.
- Синтаксис:

*if* <логическое условие1>:

    < Инструкции-1 >

*elif* <логическое условие2>:

    < Инструкции-2 >

*else*: < Инструкции-3 >

# Пример

```
a = int(input())  
if a < -3:  
    print("Мало")  
elif -3 <= a <= 3:  
    print("Средне")  
else:  
    print("Много")
```

- Любое число, не равное 0, или непустой объект – истина.
- Числа, равные 0, пустые объекты и значение *None* – ложь
- Операции сравнения применяются к структурам данных рекурсивно

# Рекомендации по использованию

- Пользуйтесь *.startswith()* и *.endswith()* вместо обработки срезов строк для проверки суффиксов или префиксов.

*if s.startswith("pfx"):*           # *правильно*

*if s[:3] == "pfx":*           # *неправильно*

- Сравнение типов объектов делайте с помощью *isinstance()*, а не прямым сравнением типов:

*if isinstance(obj, int):*       # *правильно*

*if type(obj) is type(1):*       # *неправильно*

- Переключатель *True/False*

*x = y > 0*                   # *правильно*

*if y > 0:*                   # *неправильно*

*x = True*

*else:*

*x = False*

# Рекомендации, продолжение

□ Не сравнивайте логические типы с *True* и *False* с помощью **==**

*if condit:* # **правильно**

*if condit == True:* # **неправильно**

*if condit is True:* # **неправильно**

□ Для последовательностей (строк, списков, кортежей) используйте то, что пустая последовательность есть *false*

*if not seq:* # **правильно**

*if seq:* # **правильно**

*if len(seq)* # **неправильно**

*if not len(seq)* # **неправильно**

# Трехместное выражение if/else

Пример инструкции вида

*if*  $x > 5$ :

$x = y$

*else*:

$x = z$

Можно заменить на более короткий вариант

$x = \text{truepart}$  *if*  $\langle \text{условие} \rangle$  *else*  $\text{falsepart}$

Теперь пример можно переписать как:

$x = y$  *if*  $x > 5$  *else*  $z$

**Примечание:** аналог в языке C/C++:

$\langle \text{условие} \rangle ? \langle \text{truepart} \rangle : \langle \text{falsepart} \rangle$



# Замена `switch-case` через `elif`

□ В Python нет конструкций множественного выбора типа *switch-case*.

□ Один из вариантов замены – использование *elif*:

# Производится последовательное сравнение переменной *n*.

# Если *n > 70* выполняется код *code70* и выполнение переходит на строку *final*, иначе выполняется дальнейшая проверка.

*if n > 70:*

*print("code70")*

# Если *n > 50* – выполняется код *code50* и выполнение переходит на строку *final*, иначе продолжаем...

*elif n > 50:*

*print("code50")*

*elif n > 20:*

*print("code20")*

# Если результат всех проверок оказался ложным выполняется блок *code0*, после чего переходим на строку *final*

*else:*

*print("code0")*

*print("final")*

# Другие замены switch-case

□ Существует множество рекомендаций по замене.

□ С использованием словаря:

```
choices = {"a": 1, "b": 2}
```

```
result = choices.get(key, "default")
```

Другие способы см.

<http://qaru.site/questions/10714/replacements-for-switch-statement-in-python>

# ЦИКЛЫ

□ В Python существуют следующие два типа циклических выражений:

- ✓ Цикл *while* (цикл типа "пока")
- ✓ Цикл *for* (цикл типа "для")

# Цикл типа **while**

- **while** – один из самых универсальных циклов в Python, поэтому довольно медленный
- Инструкция **while** повторяет указанный блок кода до тех пор, пока указанное в цикле условие **ИСТИННО**.

- Синтаксис:

```
while <условие>:      # Условное выражение  
    <инструкции>     # тело цикла  
else:                # необязательная часть  
    <инструкции>
```

необязательная часть **else** выполняется, если выход из цикла был произведён не инструкцией **break**

# Инструкции цикла **while**

- *break* – производит выход из цикла.
- *continue* – производит переход к началу цикла.
- *pass* – пустая инструкция-заполнитель.

Общий вид цикла **while** можно тогда записать как:

*while* <условие1>:

<инструкции>

*if* <условие2>: *break* # Выйти из цикла, пропустив *else*

*if* <условие3>: *continue* # Перейти в начало цикла

*else*:

<инструкции> # Выполняется, если не была использована инструкция "*break*"

Пара *else/break* часто позволяет избавиться от необходимости сохранять флаг штатного выхода из цикла по *условию1*. (см. примеры ниже)

Блок *else* выполняется ещё и в том случае, когда тело цикла ни разу не выполнялось.

# Пример цикла **while** с флагом

Поиск некоторого значения

```
found = False # флаг найденного значения
```

```
while x and not found:
```

```
# Пока x не пустой и не найдено значение
```

```
if match(x[0]): # Искомое значение является  
первым? (match(x) – некоторая функция,  
устанавливающая соответствие x
```

```
критериям поиска)
```

```
print("Нашли")
```

```
found = True
```

```
else:
```

```
x = x[1:] # Вырезать первое значение и  
повторить (это медленный способ)
```

```
if not found:
```

```
print("Не нашли")
```

# Пример с **else** без флага

```
while x: # Выйти, когда x опустеет  
    if match(x[0]): # (match(x) – некоторая  
        функция, устанавливающее соответствие x  
        критериям поиска)  
        print("Нашли")  
        break # Выход, в обход блока else  
    x = x[1:]  
else:  
    print("Не нашли") # Этот блок отработает,  
        только если строка x исчерпана
```

Эта версия более компактна по сравнению с предыдущей.

# Цикл типа **for**

- Цикл *for* – универсальный итератор последовательностей.
- Он выполняет обход элементов в любых упорядоченных объектах.
- *for* может работать со списками, кортежами, строками и другими встроенными итерируемыми объектами, в т. ч. и с новыми объектами, созданными с помощью классов.
- Циклы *for* могут применяться даже к объектам, которые не являются последовательностями, таким как файлы и словари.



# Общий формат циклов **for**

*for* <цель> *in* <объект>: # Связывает элементы  
объекта с переменной цикла  
<инструкции> # тело цикла  
*else*:  
< инструкции> # произведён штатный выход  
из цикла без "break"

## Полная форма:

*for* <цель> *in* <объект>:  
<инструкции> # тело цикла  
*if* <условие1>: *break* # Выход из цикла  
*if* <условие2>: *continue* # Переход в начало  
цикла  
*else*:  
<инструкции> # произведён штатный выход  
из цикла без "break"

# Примеры

Простейший перебор элементов списка:

```
for x in ["a", "b", "c"]:
```

```
    print(x, end=" ") # (в одну строчку) – a b c
```

Обход строки:

```
for x in "строка":
```

```
    print(x, end=" ") # с т р о к а
```

А вот так со списком не пройдёт:

```
L = [1, 2, 3, 4, 5]
```

```
for x in L:
```

```
    x += 1 # Элемент списка это не изменит!
```

```
print(L) # [1, 2, 3, 4, 5]
```

# Обход кортежа:

```
for (a, b) in [(1, 2), (3, 4), (5, 6)]: # создаём  
кортеж (a, b)  
print((a, b), "/", end=" ") # (1, 2) / (3, 4) / (5, 6) /
```

В результате в каждой итерации автоматически выполняется операция присваивания кортежа.

Аналогично работает и:

```
for x in [(1, 2), (3, 4), (5, 6)]: # x – кортеж  
print(x, "/", end=" ") # (1, 2) / (3, 4) / (5, 6) /
```

Не обязательно использовать в качестве итератора кортеж. Можно, например, итерировать по новому списку  $[a, b]$

```
for [a, b] in [(1, 2), (3, 4), (5, 6)]: # список [a, b]  
print([a, b], "/", end=" ") # [1, 2] / [3, 4] / [5, 6] /
```

# Обход словаря

```
D = {"a": 1, "b": 2, "c": 3}
for key in D: # Используется итератор
    словаря и операция индексирования
    print(key, ":", D[key], ", ", end=" ")
# на выходе – a : 1 , b : 2 , c : 3 ,
```

```
D = {"a": 1, "b": 2, "c": 3}
for (key, value) in D.items():
    print(key, ":", value, ", ", end=" ")
# Обход ключей и значений одновременно
# a : 1 , b : 2 , c : 3
```

# Многоуровневые данные

```
for ((a, b), c) in [(1, 2), 3), ["XY", 6]]:
```

```
    x = ((a, b), c)
```

```
    print(x, type(x))
```

```
((1, 2), 3) <class 'tuple'>
```

```
(('X', 'Y'), 6) <class 'tuple'>
```

Каждый **x** – это кортеж, сборка которого происходит на каждом шаге цикла.

Можно данный цикл переписать так:

```
for x in [(1, 2), 3), ["XY", 6]]:
```

```
    print(x, type(x))
```

```
(1, 2), 3) <class 'tuple'>
```

```
['XY', 6] <class 'list'>
```

**x** – видоизменяется в зависимости от текущего члена списка

# Вложенные циклы

Поиск пересечений

```
items = ["aaa", 111, (4, 5), 2.01] # Объекты
```

```
tests = [(4, 5), "aaa"] # Ищем ключи
```

```
for key in tests: # Для всех ключей
```

```
    for item in items: # Для всех элементов
```

```
        if item == key: # Проверит
```

```
            print(item, key, "нашли")
```

```
            break
```

```
        else:
```

```
            print(item, key, "не нашли!")
```

aaa (4, 5) не нашли!

111 (4, 5) не нашли!

(4, 5) (4, 5) нашли

aaa aaa нашли

Можно и упростить так:

```
for key in tests: # Для всех ключей
```

```
    s = "нашли" if key in items else "не нашли"
```

```
    print(key, s)
```

(4, 5) нашли

aaa нашли

# Последовательности

- Часто возникают задачи программирования нестандартных обходов последовательностей или параллельного обхода нескольких последовательностей.
- Совместно с циклами зачастую используются специальные *функции-генераторы* последовательностей:

✓ *range*,

✓ *zip*,

✓ *map*,

✓ *enumerate*

# Итерации

- Когда создаётся список, можно считывать его элементы один за другим – это называется **итерацией**
- Всё, то к чему можно применить конструкцию *for... in...*, является **итерируемым объектом**:, строки, файлы, списки, и т.п....

Итерации:

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

```
for i in L:
```

```
    print(i)
```



# Генераторы

□ **Генераторы** – это итерируемые объекты, но прочитать их можно лишь один раз, поскольку они не хранят значения в памяти, а генерируют их на лету.

Пример генератора:

```
mygen = (x*x for x in range(3))
```

```
for i in mygen:
```

```
    print(i, end=" ")    # 0 1 4
```

□ Нельзя применить конструкцию *for i in mygen* второй раз, т.к. генератор может быть использован только единожды:

□ он последовательно вычисляет 0, 1, 4 одно за другим, забывая предыдущие свои значения.

# Генератор **range**

□ **range** возвращает непрерывную последовательность увеличивающихся целых чисел, которые можно задействовать в качестве индексов внутри цикла *for*.

*dir(range)* **#[... 'count', 'index', 'start', 'step', 'stop']**  
Синтаксис: *range(start\_or\_stop, stop[, step])*

✓ *start\_or\_stop* – начальное значение

✓ *stop* – конечное значение (**не включая его!**)

✓ *step* – шаг последовательности. (по умолчанию 1)

Проверка диапазонов на равенство при помощи **==** и **!=** сравнивает их как последовательности. Т.е. два диапазона равны, если они представляют одинаковую последовательность значений.

**Примеры:** *range(0) == range(2, 1, 3) # True*

*range(0, 3, 2) == range(0, 4, 2) # True*

# СВОЙСТВА **range**

□ *range* использует класс [collections.abc.Sequence](#) и поддерживает проверку на содержание, индексацию и срезы.

□ `count(элемент)` – количество вхождений элемента

□ `index(элемент)` – индекс элемента, или ошибку, `ValueError`, если такой не найден.

## Примеры:

```
r = range(0, 20, 2) # 0, 2, 4...18
```

```
11 in r # False
```

```
10 in r # True
```

```
r.index(10) # 5
```

```
r[5] # 10
```

```
r[:5] # range(0, 10, 2)
```

```
r[-1] # 18
```

```
r.count(4) # 1
```

# Примеры последовательностей

```
list(range(6))           # [0, 1, 2, 3, 4, 5]
list(range(2, 5))      # [2, 3, 4]
list(range(0, 10, 3))  # [0, 3, 6, 9]
list(range(0, -5, -1)) # [0, -1, -2, -3, -4]
list(range(0))        # []
list(range(1, 0))     # []
```

Использование последовательности в цикле для доступа по индексу:

```
X = ["a", "b", "c"]
for i in range(len(X)):
    print(X[i], end=" ")           # a b c
```

# Инициализация списков

□ Метод 1

```
x = [[1,2,3,4]] * 3
```

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

□ Метод 2

```
y = [[1,2,3,4] for __ in range(3)]
```

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

□ Однако:

```
x[0][2] = 9 # [[1, 2, 9, 4], [1, 2, 9, 4], [1, 2, 9, 4]]
```

```
y[0][2] = 9 # [[1, 2, 9, 4], [1, 2, 3, 4], [1, 2, 3, 4]]
```

# Конкатенация диапазонов

- Операции для списков не работают с диапазонами.
- Попробуем создать несвязанный диапазон:

```
range(0, 2) + range(10, 13) # ОШИБКА
```

Используя функцию *chain* из *itertools*:

```
from itertools import chain  
conc = chain(range(0, 2), range(10, 13))  
for i in conc:  
    print(i)
```



0  
1  
10  
11  
12

Реализация функции *chain*:

```
def chain(*iterables):  
# chain('ABC', 'DEF') --> A B C D E F  
    for it in iterables:  
        for element in it:  
            yield element
```

По *itertools* см.

<https://pythonworld.ru/moduli/modul-itertools.html>

# Включения

- **Включение (comprehension)** – это компактный способ создать структуру данных из одного или более итераторов.
- Включения позволяют объединять циклы с условными проверками, не используя при этом громоздкий синтаксис.
- Это одна из характерных особенностей Python.

# Включение списка

Список может быть сгенерирован как

```
number_list = list(range(1, 5)) # [1, 2, 3, 4]
```

Формат включения списка является более характерным для Python:

```
[<выражение> for <элемент> in  
<итерабельный объект>]
```

Предыдущий пример можно переписать так:

```
number_list = [number for number in  
range(1, 5)]
```

```
# сгенерирован список [1, 2, 3, 4]
```

Читаем: Собрать в список *number\_list* числа *number* из диапазона 1...4 [с шагом 1].



# Пример включения списка

□ Сначала идет выражение, которое будет задавать элементы списка, потом – цикл, с помощью которого можно изменять выражение

*□ Подсчёт квадратов чётных чисел от 2 до 8*

*res = [x\*\*2 for x in range(2, 8, 2)] #[4, 16, 36]*

Читаем: в список *res* собрать все *x\*\*2* для *x* из диапазона от 2 до 7 с шагом 2

# Условное включение списка

□ Включение списка может содержать условное выражение:

```
[<выражение> for <элемент> in  
<итерабельный объект> if <условие>]
```

Создадим список чётных чисел в диапазоне от 1 до 9:  
Традиционно (если не использовать шаг):

```
a_list = []  
for num in range(1, 10):  
    if num % 2 == 0:  
        a_list.append(num)  
print(a_list)           # [1, 3, 5, 7, 9]
```

С условным включением:

```
a_list = [num for num in range(1, 10) if num % 2 == 0]  
print(a_list)           # [1, 3, 5, 7, 9]
```

Читаем: В список *a\_list* включаем те элементы *num*  
из диапазона ( $1\dots\bar{9}$ ), если  $num \% 2 == 0$

# Замена вложенного цикла

```
1 1
1 2
2 1
2 2
3 1
3 2
```

```
rows = range(1, 4)
cols = range(1, 3)
for row in rows:
    for col in cols:
        print(row, col)
```

Заменяем

```
rows = range(1, 4)
cols = range(1, 3)
cells = [(row, col) for row in rows for col in cols]
for row, col in cells:
```

```
    print(row, col)           # см. справа =>
# в cells [(1, 1), (1, 2), (2, 1), (2, 2), (3, 1), (3, 2)]
```

Читаем: в список ***cells*** собираем кортежи (*row*, *col*), где первый элемент кортежа *row* из диапазона *rows*, второй элемент *col* – из диапазона *cols*.

```
1 1
1 2
2 1
2 2
3 1
3 2
```

# Включение для словаря

□ Синтаксис:

{<выражение\_ключа>:

<выражение\_значения>*for* <выражение> *in*

<итерабельный объект>}

Пример. Проходя по каждой из *letter* букв в строке *wd*, считаем сколько раз появляется эта буква.

```
wd = "текст"
```

```
count = {letter: wd.count(letter) for letter in wd}
```

```
print(count) # {'m': 2, 'e': 1, 'к': 1, 'с': 1}
```

Читаем: включить в словарь *count* с ключами *letter* количество вхождений буквы *letter* в строку *wd*.

# Кортежи и включения

- Для кортежей не существует включений.
- При необходимости можно конвертировать список в кортеж обычным способом:

```
a_list = [num for num in range(1, 10) if num % 2  
== 1]  
a_tup = tuple(a_list)  
print(a_tup)          # (1, 3, 5, 7, 9)
```

# Генерирование индексов и элементов: `enumerate`

□ В некоторых программах требуется получить и элемент, и его индекс.

Обычное решение:

```
S = "текст"
```

```
i = 0
```

```
for item in S:
```

```
    print(item, "индекс", i)
```

```
    i += 1
```

```
т индекс 0
```

```
е индекс 1
```

```
к индекс 2
```

```
с индекс 3
```

```
т индекс 4
```

Но можно короче:

```
for (i, item) in enumerate(S):
```

```
    print(item, "индекс", i)
```

```
т индекс 0
```

```
е индекс 1
```

```
к индекс 2
```

```
с индекс 3
```

```
т индекс 4
```

# Синтаксис `enumerate`

□ `enumerate(iterable[, start=0])` – возвращает кортеж `(index, value)` для каждого элемента списка.

□ Эквивалентная запись:

```
def enumerate(sequence, start=0):
```

```
    n = start
```

```
    for elem in sequence:
```

```
        yield n, elem
```

```
        n += 1
```

□ `yield` – это ключевое слово, которое используется примерно как `return` – отличие в том, что функция вернёт генератор.

# Примеры

```
S = "текст"
```

```
E = enumerate(S)
```

```
print(list(E))
```

```
# [(0, 'т'), (1, 'е'), (2, 'к'), (3, 'с'), (4, 'т')]
```

Генератор поддерживает метод *next*

```
S = "текст" # <enumerate object at  
0x0000000002A5AAB0>
```

```
E = enumerate(S)
```

```
print(E)
```

```
print(next(E)) # (0, 'т')
```

```
print(next(E)) # (1, 'е')
```

```
print(next(E)) # (2, 'к')
```

```
print(next(E)) # (3, 'с')
```

```
print(next(E)) # (4, 'т')
```



# Другие итераторы встроенных типов

- Помимо файлов и фактических последовательностей, таких как списки, удобные итераторы также имеют и другие типы.

- Классический обход словаря:

```
d = {"a":1, "b":2, "c":3}
```

```
for key in d.keys():  
    print(key, d[key])
```

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

*И с помощью итератора **iter***

```
it = iter(d)  
print(next(it))  
print(next(it))  
print(next(it))
```

```
a  
b  
c
```

Спасибо за внимание