Информационные технологии

Основы программирования на Python 3

Каф. ИКТ РХТУ им. Д.И. Менделеева
Ст. преп. Васецкий А.М.

Москва, 2018

Лекция 4.
Инструкции и операторы

- **Операторы языка Python**
- **Условные операторы**
- **Циклы**
- **Последовательности**

Список источников

1. Оригинальная документация https://docs.python.org/3/
2. Изучаем Python. Программирование игр, визуализация данных, веб-приложения. — СПб.: Питер, 2017. — 496 с.: ил. — (Серия «Библиотека программиста»).
3. Рейтц К., Шлюссер Т. Автостопом по Python. — СПб.: Питер, 2017. — 336 с.: ил. — (Серия «Бестселлеры O'Reilly»).
4. Лутц М. Изучаем Python, 4-е издание. – Пер. с англ. – СПб.: Символ-Плюс, 2011. – 1280 с., ил.
5. Прохоренок Н. А., Дронов В. А., «Python 3. Самое необходимое» — СПб.: БХВ-Петербург. – 2016, 464 с
6. Прохоренок Н.А. Дронов В. А., «Python 3 и PyQt. Разработка приложений» — СПб.: БХВ-Петербург. – 2016, 832 с.
7. Любанович Б: Простой Python. Современный стиль программирования. – СПб.: Питер, – 2016, 480 с.
8. http://pythonicway.com
9. http://pythonz.net
10. https://tproger.ru/tag/python/
11. https://pythonworld.ru
12. http://pythontutor.ru

Инструкции

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

Инструкции

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

Инструкции

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

Операторы Python

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

Арифметические операторы

+	Сложение	15 + 5 = 20 20 + -3 = 17 13.4 + 7 = 20.4
–	Вычитание	15 – 5 = 10 20 – -3 = 23 13.4 – 7 = 6.4
*	Умножение	5 * 5 = 25 7 * 3.2 = 22.4 -3 * 12 = -36
/	Деление	5 / 2 = 2.5 (В Python 2.x версии результат будет 2) 5.0 / 2 = 2.5 (Хотя бы один операнд должен быть *float*)
%	Остаток	6 % 2 = 0 7 % 2 = 1 13.2 % 5 = ~3.2
**	Возведение в степень	5 ** 2 = 25 2 ** 3 = 8 -3 ** 2 = -9 **(степень приоритетнее знака)**
//	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 $\sqrt[3]{-1}$:

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

при $k = 1$ $\quad \omega_2 = -1$,

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

Пример корня

```
-27 ** (1/3)    # -3.0
(-27) ** (1/3)      # (1.5+2.598076211353316j)
```

Проверка:

```
(1.5 + 2.598076211353316j)**3  # (-27+0j)
```

Сопряжённый корень проверка:

```
(1.5 – 2.598076211353316j)**3  # (-27+0j)
```

Аналогично

```
a = -27
a ** (1/3)       # (1.5+2.598076211353316j)
```

Операторы сравнения

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

* – 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

=	Присваивает значение правого операнда левому.	d = 24
+=	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
–=	Отнимает значение правого операнда от левого и присваивает результат левому операнду.	c = 5 a = 2 c –= a то же: c = c – a # c=3
*=	Умножает правый операнд с левым и присваивает результат левому операнду.	c = 5 a = 2 c *= a то же: c = c * a # c=10
/=	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

%=	Делит по модулю операнды и присваивает результат левому.	c = 5 a = 2 c %= a то же: c = c % a # c=1
**=	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
//=	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})

&	Бинарное "И". Копирует бит в результат только если бит присутствует в обоих операндах.	**a & b = 0000 1100 (12_{10})**
|	Бинарный "ИЛИ" Копирует бит, если тот присутствует в хотя бы в одном операнде.	**a | b = 0011 1101 (61_{10})**
^	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})

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

Логические операторы

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

Операторы членства

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

Операторы тождественности

is	Возвращает истину, если оба операнда указывают на один объект.	x is y вернет истину, если id(x) будет равно id(y).
is not	Возвращает ложь, если оба операнда указывают на один объект.	x is not y, вернет истину если id(x) не равно id(y).

Приоритет операторов

**	Возведение в степень
~ + –	Комплиментарный оператор
* / % //	Умножение, деление, деление по модулю, целочисленное деление.
+ –	Сложение и вычитание.
>> <<	Побитовый сдвиг вправо и побитовый сдвиг влево.
&	Бинарный "И".
^ |	Бинарный "Исключительное ИЛИ" и бинарный "ИЛИ"
<= < > >=	Операторы сравнения
<> == !=	Операторы равенства
= %= /= //= -= += *= **=	Операторы присваивания
is is not	Тождественные операторы
in not in	Операторы членства
not or and	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 = truepart if <условие> else falsepart

Теперь пример можно переписать как:

```
x = y if x > 5 else z
```

Примечание: аналог в языке C/C++:

```
<условие> ? <truepart> : <falsepart>
```

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 цикличных 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, устанавливающее 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 сработает 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

Использовав 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…9***), если ***num % 2 == 0***

Замена вложенного цикла

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

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

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)]
```

```
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***.

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) # {'т': 2, 'е': 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