Лекция 2 "Управляющие конструкции и списки"

часть 1 Управляющие конструкции

Финансовый университет при Правительстве РФ, лектор С.В. Макрушин

v 0.8 27.07.2021

Разделы:

-

- к оглавлению

In [1]:

```
# загружаем стиль для оформления презентации
from IPython.display import HTML
from urllib.request import urlopen
html = urlopen("file:./lec_v1.css")
HTML(html.read().decode('utf-8'))
```

Out[1]:

Булев тип

- к оглавлению

In [6]:

```
t1 = True
t2 = False
```

In [7]:

```
print(t1)
print(t2)
```

```
True
False
```

In [8]:

```
type(t1)
```

Out[8]:

```
bool
```

In [9]:

```
t3 = bool()
t3
```

Out[9]:

```
False
```

Приведение других типов к значениям типа bool:

In [10]:

```
# Значения других типов, интерпретируемые как False:
# число 0
ft1 = bool(0)
ft2 = bool(0.0)

ft3 = bool('') # пустая строка
# последовательность, не содержащая элементов:
ft4 = bool(tuple()) # пустой кортеж
ft5 = bool(list()) # пустой список

ft6 = bool(set()) # пустое множество
ft7 = bool(dict()) # пустой словарь

ft8 = bool(None) # ссылка на объект None

print(ft1, ft2, ft3, ft4, ft5, ft6, ft7, ft8)
```

```
False False False False False False False False
```

Для упрощения можно запомнить, что "объекты-контейнеры", не содержащие элементов (т.е. для которых `len(obj)` возвращает 0), приводятся к `False` .

In [11]:

```
# Примеры значений других типов, интерпретируемых как True:
# число, отличное от 0
tt1 = bool(42)
tt2 = bool(42.42)

tt3 = bool('a b c') # НЕ пустая строка
# последовательность, содержащая элементы:
tt4 = bool((1, 2, 3)) # НЕ пустой кортеж
tt5 = bool([1, 2, 3]) # НЕ пустой список

tt6 = bool({1, 2, 3}) # НЕ пустое множество
tt7 = bool({'a':1, 'b':2, 'c':3}) # не пустой словарь

print(tt1, tt2, tt3, tt4, tt5, tt6, tt7)
```

```
True True True True True True True
```

In [12]:

```
bool('False')
```

Out[12]:

```
True
```

Операции с логическим типом

In [13]:

```
t1
```

Out[13]:

```
True
```

In [14]:

```
t2
```

Out[14]:

```
False
```

In [15]:

```
not t1
```

Out[15]:

```
False
```

In [16]:

```
t1 or t2
```

Out[16]:

```
True
```

In [17]:

```
t1 and t2
```

Out[17]:

```
False
```

Приоритет:

1. not (соответствует унарному -)
2. and (соответствует *)
3. or (соответствует +)

>

Операции сравнения

- к оглавлению

In [18]:

```
# создаем переменные:
a = 2
b = 6
```

Виды сравнений:

In [15]:

```
a == b # проверка на равенство
```

Out[15]:

```
False
```

In [19]:

```
a != b # проверка на неравенство
```

Out[19]:

```
True
```

In [20]:

```
a < b # меньше
```

Out[20]:

```
True
```

In [21]:

```
a <= b # меньше или равно
```

Out[21]:

```
True
```

In [22]:

```
a > b # больше
```

Out[22]:

```
False
```

In [23]:

```
a >= b # больше или равно
```

Out[23]:

```
False
```

In [24]:

```
# Цепочка сравнений:
0 <= a <= 10 # эквивалентно: (0 <= a) and (a <= 10)
```

Out[24]:

```
True
```

In [25]:

```
# Можно даже так:
0 < a < b > 2
```

Out[25]:

```
True
```

In [26]:

```
# сравнение значений различных числовых типов выполняет автоматическое приведение типов
# (автоматическое приведение типов работает в Python только для числовых типов!):
3.0 < 4
```

Out[26]:

```
True
```

In [27]:

```
# автоматического приведения типов при сравнении (кроме случая числовых типов) нет:
int("3") < 4
```

Out[27]:

```
True
```

Сравнения для нечисловых типов:

In [28]:

```
True > False
```

Out[28]:

```
True
```

In [29]:

```
False == False
```

Out[29]:

```
True
```

In [30]:

```
a, b
```

Out[30]:

```
(2, 6)
```

In [29]:

```
(a < b) == True # бессмысленное сравнение с True
```

Out[29]:

```
True
```

In [31]:

```
a < b  # правильная альтернатива!
```

Out[31]:

```
True
```

In [32]:

```
None == None
```

Out[32]:

```
True
```

In [33]:

```
# в случае булева типа и None "is" и "==" эквивалентно
# Но проверка "is" предпочтительна
b1 = True
n1 = None
print('---- ==:')
print(b1 == True)
print(n1 == None)
print('---- is:')
print(b1 is True)
print(n1 is None)

print('---- is not:')
# проверка на несовпадение (предпочтительная):
print(b1 is not True) # is not - это не два оператора, а один!
print(n1 is not None)
```

```
---- ==:
True
True
---- is:
True
True
---- is not:
False
False
```

Сравнение последовательностей:

In [34]:

```
s1 = 'Hello world'
s2 = 'Hello world'
s3 = 'Hello'
s4 = 'Halo'
s5 = 'A'
s6 = '1'
s1 is s2 # совпадают ли объекты?
```

Out[34]:

```
False
```

In [35]:

```
s1 == s2  # совпадает ли содержимое объектов?
```

Out[35]:

```
True
```

In [36]:

```
s1 == s3
```

Out[36]:

```
False
```

Лексикографический порядок используется для операций порядка для объектов, хранящих упорядоченные последовательности:

In [41]:

```
print(f'"{s1}" > "{s3}" is: {s1>s3}') # сравнение в лексикографическом порядке
```

```
"Hello world" > "Hello" is: True
```

In [42]:

```
print(f'"{s3}" > "{s4}" is: {s3 > s4}')
```

```
"Hello" > "Halo" is: True
```

In [43]:

```
print(f'"{s4}" > "{s5}" is: {s4 > s5}')
```

```
"Halo" > "A" is: True
```

In [44]:

```
print(f'"{s6}" < "{s5}" is: {s6 < s5}')
```

```
"1" < "A" is: True
```

In [45]:

```
tu1 = (1, 2, 3)
tu2 = (1, 3, 3)
tu3 = (1, 2)
```

In [46]:

```
print(f'"{tu1}" < "{tu2}" is: {tu1 < tu2}')
```

```
"(1, 2, 3)" < "(1, 3, 3)" is: True
```

In [47]:

```
print(f'"{tu1}" > "{tu3}" is: {tu1 > tu3}')
```

```
"(1, 2, 3)" > "(1, 2)" is: True
```

>

Условные операторы

В языке Python для обозначения блочной структуры используются **отступы** (вместо фигурных скобок (`{` и `}`) в языках с C-образным синтаксисом или блоков `begin - end` в Pascal).

Так как некоторые синтаксические конструкции языка Python требуют наличия блока кода, Python предоставляет ключевое слово `pass` , которое представляет собой инструкцию, не делающую ровным счетом ничего, и которая может использоваться везде, где требуется блок кода (или когда мы хотим указать на наличие особого случая), но никаких действий выполнять не требуется.

```
if x:
    a = b
```

Общий вид синтаксиса условного оператора:

```
if boolean_expression1:
    suite1
elif boolean_expression2:
    suite2
elif boolean_expressionN:
    suiteN
else:
    else_suite
```

- может быть ноль или более предложений `elif`
- блок `else` не обязательный

In [48]:

```
x = 42
```

In [49]:

```
if x:
    print('x is not False')
```

```
x is not False
```

In [50]:

```
if x == 5:
    print('x is not False')
    y = 10
print('result!')
```

```
result!
```

In [51]:

```
x = 0
```

In [52]:

```
if not x:
    print('x is False (or equivalent)')
    y = 10
else:
    print('x is True (or equivalent)')
    y = 5
print(f'y: {y}')
```

```
x is False (or equivalent)
y: 10
```

In [53]:

```
x = -150
```

In [54]:

```
if x >= 0:
    print('x is positve')
elif x >= -10:
    print('x is slightly negative')
elif x >= -100:
    print('x is pretty negative')
else:
    print('x is totally negative')
```

```
x is totally negative
```

In [57]:

```
x = 10
if x >= 0:
    pass # если блок кода обязательный, а ничего выполнять не требуется, то используется on
elif x >= -10:
    print('x is slightly negative')
elif x >= -100:
    print('x is pretty negative')
else:
    print('x is totally negative')
```

In [58]:

```
tv = x > 5
# работающий, но некрасивый способ:
if tv == False:
    print('win!')

# правильный способ:
if not tv:
    print('win! (correct)')
```

Общий вид синтаксиса тернарного условного оператора:

```
true_result if boolean_expression1 else false_result
```

In [59]:

```
# пример:
a = 5
b = 7
print(a if a > 5 else b)
```

```
7
```

In [60]:

```
# тернарный оператор хорошо подходит для присвоения значения:
c = a if a > 0 else 0
c
```

Out[60]:

```
5
```

In [62]:

```
max(a, 0)A
```

Out[62]:

```
5
```

>

Циклы

- к оглавлению

Цикл while

Синтаксис оператора цикла `while` :

```
while boolean_expression:
    suite
else:
    else_suite
```

- блок `else` не обязательный

In [69]:

```
i = 0
while i < 10:
    print(i)
    i += 1
```

```
0
1
2
3
4
5
6
7
8
9
```

Операторы `break` , `continue` , блок `else` :

In [73]:

```
st = 'Hello world'
target = 'l'

# поиск индекса первого вхождения символа target в строку st
index = 0
while index < len(st):
    if st[index] == target:
        break # преждевременный выход из цикла
    index += 1
else: # блок else выполняется, если выход из цикла произошел не в результате выполнения one
    index = -1

print(index)
# print(f'resilt: {result}')
```

```
2
```

In [71]:

```
len(st)
```

Out[71]:

```
11
```

In [66]:

```
index
```

Out[66]:

```
-1
```

In [74]:

```
st = 'Hello world'
target = 'l'
st_wo_t = ''

# создание строки, в которой из строки st исключен символ target
index = 0
while index < len(st):
    cur = st[index]
    index += 1
    if cur == target:
        continue # преждевременное окончание итерации цикла и переход к очередной проверке
    st_wo_t += cur

st_wo_t
```

Out[74]:

```
'Heo word'
```

In [ ]:

```
# Пример использования "вечного цикла"
while True: # вечный цикл
    item = get_next_item()
    if correct(item): # условие выхода из цикла
        break
    process_item(item)
```

>

Цикл for

- к оглавлению

Синтаксис оператора цикла `for`:

```
for expression in iterable:
    for_suite
else:
    else_suite
```

- В качестве iterable может использоваться любой тип данных, допускающий выполнение итераций по нему, включая строки (где итерации выполняются по символам строки), списки, кортежи и другие типы коллекций языка Python.
- блок `else` не обязательный
- внутри цикла могут использоваться операторы `break` и `continue`
- в практике программирования на Python циклы `for` используются намного чаще чем `while`

In [75]:

```
st = 'Hello world'
for symb in st:
    print(symb)
```

```
H
e
l
l
o

w
o
r
l
d
```

Итерируемый тип данных - это такой тип, который может возвращать свои элементы по одному. Для возвращения элементов по одному у объекта итерируемого типа данных нужно запросить итератор. Любой объект, имеющий метод `__iter__()`, или любая последовательность (то есть объект, имеющий метод `__getitem__()`, принимающий целочисленный аргумент со значением от 0 и выше), является итерируемым и может предоставлять итератор.

Итератор - это объект, имеющий метод `__next__()`, который при каждом вызове возвращает очередной элемент и возбуждает **исключение StopIteration** после исчерпания всех элементов.

In [80]:

```
li1 = [1, 2, 3, 4]
```

In [81]:

```
i1 = iter(li1)
```

In [82]:

```
type(i1)
```

Out[82]:

```
list_iterator
```

In [83]:

```
next(i1)
```

Out[83]:

```
1
```

In [84]:

```
next(i1)
```

Out[84]:

```
2
```

In [85]:

```
next(i1)
```

Out[85]:

```
3
```

In [86]:

```
next(i1)
```

Out[86]:

```
4
```

In [87]:

```
next(i1)
```

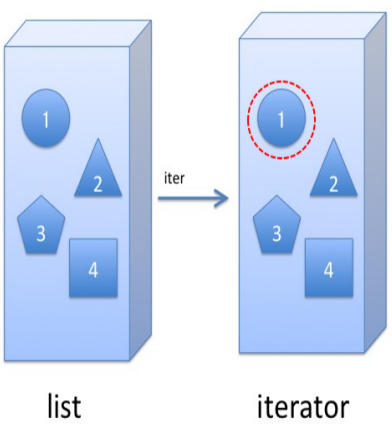
```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-87-cc9ef6da1ea7> in <module>
----> 1 next(i1)

StopIteration:
```

In [ ]:

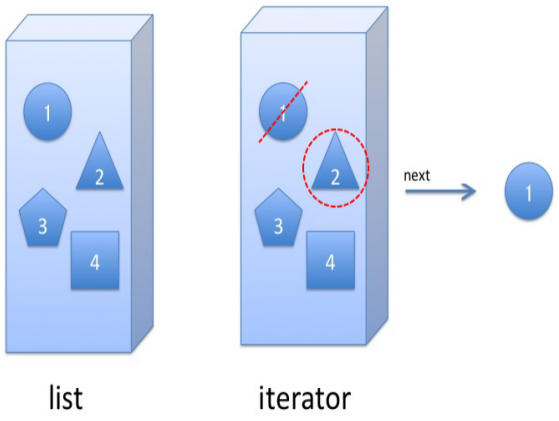
```

```

Работа итерируемого типа данных и итератера

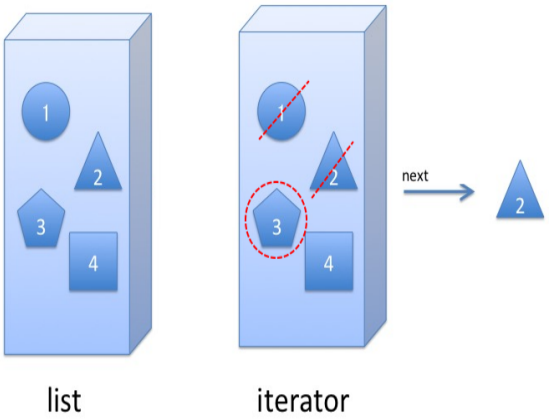
Создание итерируемым объектом итератера



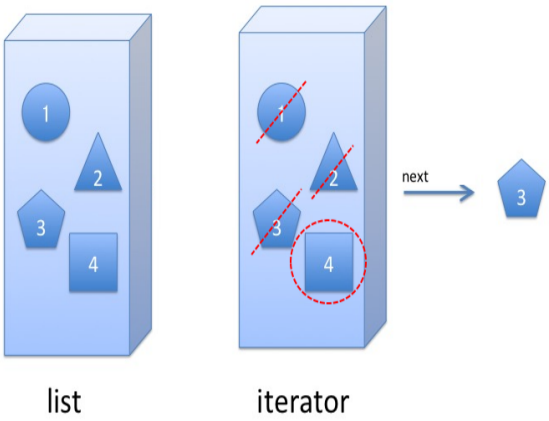
Возвращение итератором 1го объекта



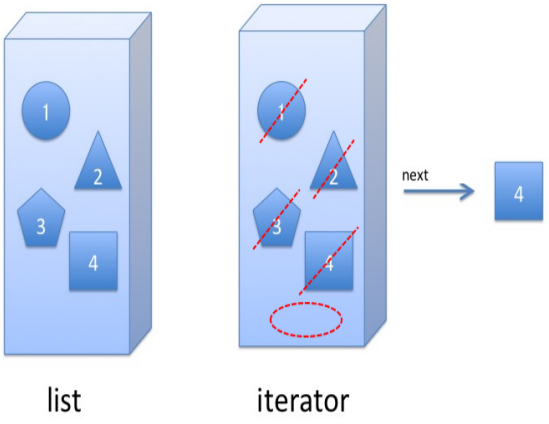
Возвращение итератором 2го объекта



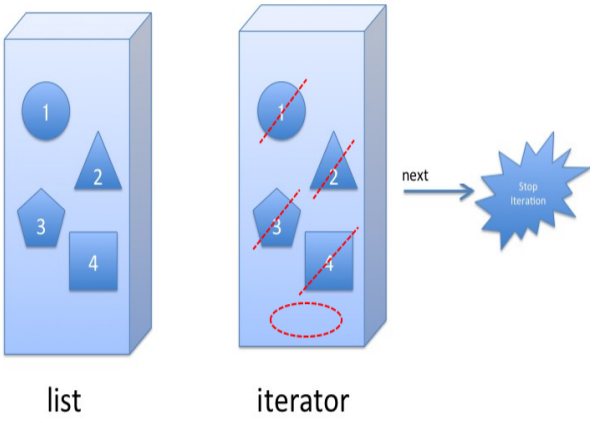
Возвращение итератором 3го объекта



Возвращение итератором 4го объекта



Возвращение итератором исключительной ситуации StopIteration



Порядок, в котором возвращаются элементы, зависит от итерируемого объекта.

- В случае списков и кортежей элементы обычно возвращаются в предопределенном порядке, начиная с первого элемента (находящегося в позиции с индексом 0).
- Другие итераторы возвращают элементы в произвольном порядке - например, итераторы словарей и множеств.

В качестве выражения `expression` в

```
for expression in iterable:
    for_suite
```

обычно используется:

- единственная переменная которая хранит очередной объект, возвращаемый итератором;
- последовательность переменных, как правило, в форме кортежа. Если в качестве выражения expression используется кортеж или список, каждый элемент, возвращаемый итератором, распаковывается в элементы expression.

Переменные, созданные в выражении `expression` цикла for ... in, **продолжают существовать после завершения цикла**. Как и любые локальные переменные, они прекращают свое существование после выхода из области видимости, включающей их.

In [88]:

```
product = 1
# расчет произведения всех элементов кортежа:
for e in (1, 2, 4):
    product *= e
print(product)
```

```
8
```

In [89]:

```
e
```

Out[89]:

```
4
```

In [78]:

```
product = 1
# расчет произведения всех элементов кортежа:
# (полный аналог цикла for реализованный в цикле while):
tu1 = (1, 2, 4)

itr = iter(tu1) # создание итератора itr
while True:
    try:
        e = next(itr) # получение у итератора очередного объекта
        product *= e # тело цикла (см. цикл for)
    except StopIteration: # обработка события StopIteration
        break
print(product)
```

```
8
```

>

Функции range и enumerate

- к оглавлению

Функция `range` возвращает целочисленный итератор. Способы обращения к функции `range`:

- `range(stop)`

С одним аргументом (stop) итератор представляет последовательность целых чисел от 0 до (stop - 1).

- `range(start, stop)`

С двумя аргументами (start, stop) - последовательность целых чисел от start до (stop - 1).

- `range(start, stop, step)`

С тремя аргументами - последовательность целых чисел от start до (stop - 1) с шагом step.

Функция `range` позволяет удобно реализовать обход целочисленной последовательности с помощью оператора `for`.

In [90]:

```
for i in range(4):
    print(i)
```

```
0
1
2
3
```

вместо:

In [81]:

```
i = 0
while i < 4:
    print(i)
    i += 1
```

```
0
1
2
3
```

In [91]:

```
for i in range(3, 7):
    print(i)
```

```
3
4
5
6
```

In [92]:

```
for i in range(-3, 3):
    print(i)
```

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

In [93]:

```
for i in range(-3, 5, 2):
    print(i)
```

```
-3
-1
1
3
```

Функция `enumerate` обычно используется в циклах `for ... in` , чтобы получить последовательность кортежей `(index, item)` , где значения индексов отсчитывается от 0 или от значения start;

Синтаксис функции `enumerte` :

- `enumerate(i)` генерируется последовательность кортежей `(index, item)` , где значения индексов отсчитывается от 0
- `enumerate (i, start)` генерируется последовательность кортежей `(index, item)` , где значения индексов отсчитывается от значения start

In [95]:

```
st = 'Hello world'
for ind, symb in enumerate(st):
    print(f"Symbol '{symb}' has index {ind}")
```

```
Symbol 'H' has index 0
Symbol 'e' has index 1
Symbol 'l' has index 2
Symbol 'l' has index 3
Symbol 'o' has index 4
Symbol ' ' has index 5
Symbol 'w' has index 6
Symbol 'o' has index 7
Symbol 'r' has index 8
Symbol 'l' has index 9
Symbol 'd' has index 10
```

In [96]:

```
st = 'Hello world'
for num, symb in enumerate(st, 1):
    print(f"Symbol '{symb}' has number {num}")
```

```
Symbol 'H' has number 1
Symbol 'e' has number 2
Symbol 'l' has number 3
Symbol 'l' has number 4
Symbol 'o' has number 5
Symbol ' ' has number 6
Symbol 'w' has number 7
Symbol 'o' has number 8
Symbol 'r' has number 9
Symbol 'l' has number 10
Symbol 'd' has number 11
```

In [ ]:

```

```