Основи на Програмирането

Лекция 5

Итерация. Команди и контексти на използване. Вградени функции и методи. Писане на качествен код.

Лекция 5

Какво ще научите

- Итерация, цикъл, многократно изпълнение
- Клаузи и блокове от команди
- Особености на командите **for** и **while**
- Синтаксис на командите
- Вградени команди **break, continue, pass**
- Клауза **else**
- Филтър **if** в команди за итерация
- Създаване на контейнери с оператор **for**
- Използване на **enumerate** и **next** за итерация
- Динамичен модел за представяне на данните
- Опасности при присвояване на mutable обекти
- Вградена функция print
- Вградена функция input
- Препоръки за писане на качествен код

Команди за итерация

- ✓ Това са вградени команди, които позволяват в зависимост от различни условия, да се изпълни многократно блок от други команди
- ✓ Всяка такава команда включва няколко различни клаузи, отговарящи на различните условия и ситуации от контекста
- ✓ Всяко изпълнение на блока от команди се нарича единична итерация (цикъл)
- ✓ Командите за итерация се наричат и команди за циклично изпълнение

Съставни блокови команди

- ✓ Блок започва след основна клауза, завършваща с “:”
- ✓ Всяка команда от блока започва на нов ред и е изместена с няколко интервала в дясно в сравнение с основната клауза.
- ✓ Стандартният брой интервали за изместване е четири.
- ✓ Първата команда на нов ред със същото изместване като това на основната клауза (или с по-малко) сигнализира за край на блока
- ✓ Допуска се команда от блока да бъде съставна блокова команда, което води до още по-големи отмествания на следващите блокове в дясно
- ✓ Примери – for ; if ; while …

Пример за команда while

```
a , b = 0 , 1
while  b < 10:
    print(b)
    a, b = b, a + b
```

```
1
1
2
3
5
8
```

Друг пример за цикъл while

```
>>> x = 0
>>> while x < 100:
        print(x)          # Извежда числата от 0 до 99
        x += 1
>>> N=5     # ще пресметнем N! за N>0
>>> Fact = 1        # ще го получим във Fact
>>> while N > 0:            # условие за повторение
        Fact = Fact * N     # умножаваме поредното
        N -= 1              # и го намаляваме с 1-ца

>>> print(Fact)    # извеждаме резултата
120
```

Примери за цикъл **for**

```
>>> for x in range(10):   #0-9
        print(x)
0
…
9

>>> plodove = ["Ябълка", "Портокал"]
>>> for plod in plodove:
        print(plod)
Ябълка
Портокал
```

Цикъл: while

```
while <test>: # Проверка на клауза <test>
  <block>         # Ако true: изпълнява блока с команди
else:             # Незадължителна клауза else
  <block>         # Изпълнява се след нормален край
```

Цикъл: for

```
for <var> in <object>:
    # Присвоява на променлива var последователно
    # във всеки цикъл по един елемент от
    # контейнера <object>
  <block>
      # Докато има елементи за присвояване:
       # изпълнява блока с команди
else:          # Незадължителна клауза else
  <block> # Изпълнява се след нормален край
```

Цикъл for – обект итератор

for <var> in <*object*>:

…..

- <object> е специален обект от тип итератор
- Тези обекти представляват наредени контейнери (контейнер, за който е дефинирана подредба)
- За всеки такъв обект е известно кой е 1-я елемент
- За произволен елемент е известно кой е следващият елемент, достъпен с вградената функция next()
- Когато няма повече елементи, next() връща съобщение за грешка

Вградени команди за цикли

- ✓ break – моментален изход от текущия цикъл
- ✓ continue – моментален скок към началната клауза на цикъла (пропускат се другите команди в блока)
- ✓ pass – команда която не прави нищо
- ✓ клауза else – изпълнява се само ако цикъла завърши нормално (без команда break)

Цикъл: while

```
while <test>: # Проверка на клауза <test>
    <block>           # Ако true: изпълнява блока с команди
    if <test1>: break # ненормален изход
    if <test2>: continue      # игнорира следващи команди
 # от блока, ако има, и отива в клауза <test>
else:             # Незадължителна клауза else
    <block>
        # Изпълнява се след нормален край на
        #  цикъла, т.е. без изход с команда break
```

Цикъл: for

```
for <var> in <object>: # Присвоява на променлива
# var елементи от <object>
    <block> # Докато има елементи за присвояване:
# изпълнява блока с команди
    if <test1>: break # ненормален изход
    if <test2>: continue      # игнорира следващи команди
# от блока, ако има, и отива в клауза <test>
    else:          # Незадължителна клауза else
        <block> # Изпълнява се след нормален край на
#  цикъла, т.е. без изход с команда break
```

Цикъл: for

- Променливата в оператор за цикъл for пробягва (последователно се свързва с) всеки един обект от подредената редица (итератор) зададена след in
- Блокът команди се изпълнява за всяка една стойност на променливата получена с next(object) (колкото са получените стойности, толкова пъти се изпълнява)
- Завършва нормално след като няма повече стойности в контейнера (функцията next() е завършила с грешка)

Определени (крайни) цикли

- При обработка на краен брой от елементи в контейнер, използваме цикъл за достъп до всеки един елемент от контейнера. Този тип цикли се наричат крайни (определени).
- За този тип цикли казваме, че променливата пробягва всеки един елемент от крайното множество - контейнер.

Пример за безкраен цикъл

```
>>> while True:
... print('Въведи stop за край!')
... name = input('Въведи име:')
... if name == 'stop': break
... age = input('Въведи възраст: ')
... print('Здравей, ', name, '=>', int(age) ** 2)
...
```

Примери

Операторът break прекратява повторението на цикъла и предава управлението на оператора веднага след блока за повторение. Той е еквивалентен на тестова клауза за изход от цикъла, възможна навсякъде в цикъла

```
while True:
    line = input('> ')
    if line == 'done':
        break
    print(line)
print('Done!')
```

Примери

Операторът continue прекратява текущата итерация и предава управлението в началото на следващ итерация

```
while True:
    line = input('> ')
    if line[0] == '#':
        continue
    if line = 'done':
        break
    print(line)
print('Done!')
```

Примери

```
x = y // 2 # y е произволно цяло число > 1
while x > 1:
   if y % x == 0: # намерен е делител
      print(y, ' има делител ', x)
      break # Напуска цикъла и пропуска клаузата else
   x -= 1
else: # При нормален изход от цикъла
   print(y, ' е просто число')
```

Намиране на най-голям елемент

```
largest-so-far = 10e-10000
for the-num in [9, -41, 12, -3, 74, -15] :
    if the-num > largest-so-far :
        largest-so-far = the-num
print ('Най-голямото: ', largest-so-far)
```

Използваме largest-so-far за съхранение на най-голямата стойност намерена до момента. Ако текущият елемент има по-голяма стойност, го запазваме като нова най-голяма стойност.

Намиране на най-малък елемент

```
smallest-so-far = -10e10000
for thing in [9, 41, -12, 3, -74, 15] :
    if thing < smallest-so-far :
        smallest-so-far = thing
print ('Най-малкото: ', smallest-so-far)
```

Използваме smallest-so-far за съхранение на най-малката стойност намерена до момента. Ако текущият елемент има по-малка стойност, го запазваме като нова най-малка стойност.

Броене в цикъл

```
broy = 0
for thing in [9, 41, 12, 3, 74 , 15] :
    broy += 1
print ('Брой: ', broy)
```

Броим колко пъти е изпълнен цикъл чрез променлива брояч, с начална стойност 0, която се увеличава с 1 при всяко изпълнение на блока команди от цикъла.

Полезно за контейнер без вградена функция len()

Намиране на сума в цикъл

```
suma = 0
for thing in [9, 41, 12, 3, 74 , 15] :
    suma += thing
print ('Сума: ', suma)
```

За намиране на сума от елементите в контейнер използваме променлива suma с начална стойност 0, която се увеличава с поредния елемент от контейнера при всяко изпълнение на блока команди от цикъла.

Полезно за контейнер без вградена функция sum()

Намиране на средна стойност в цикъл

```
broy, suma = 0, 0
for thing in [9, 41, 12, 3, 74 , 15] :
    suma += thing
    broy += 1
print ('Средна стойност: ', broy, suma, suma / broy)
```

За намиране на средна стойност от елементите в контейнер използваме променливи broy и suma както от предните примери, като средната стойност е частното на suma и broy.

Филтриране в цикъл

```
for thing in [9, 41, 12, 3, 74 , 15] :
    if thing > 20:
        print ('Голямо число: ', thing)
```

За филтриране на елементите в контейнер използваме оператор if за намиране на елементите от контейнера с дадено свойство.

Намиране на елемент в контейнер с цикъл

```
found, ind = False, -1
for num, thing in enumerate([9, 41, 12, 3, 74 , 15]):
    if thing == 3:
        found, ind = True, num
        break
print ('Среща се: ', found, ', в позиция: ', ind)
```

За намиране на елемент в контейнер използваме оператор if и променливи found (True ако сме намерили елемента, False иначе) и ind (индекс на елемента в контейнера, -1 ако го няма).

Полезна за намиране на индекса (за разлика от in)

Намиране на прости числа в диапазон

```
for n in range(2,10):
   for x in range(2, n//2):
      if n % x == 0:
         print(n, ' е равно на: ', x,'*', n//x)
         break
    else: #цикълът пропада и не намира
            # делител – значи е просто число
      print(n,' е просто')
```

Някои препоръки

- ✓ Не променяйте контейнера използван за итерация
- ✓ Общо правило: използвайте цикъл for вместо цикъл while винаги когато е възможно
  Причина: изпълнението на цикъла for е по-бързо и се консумират по-малко ресурси
- ✓ Не използвайте range в цикъл, ако имате друга възможност
  Причина: отново по-бавно изпълнение и заделяне на големи ресурси за range

Обхващане на списък (List Comprehensions)

[<expression> **for** <value> **in** <collection> **if** <condition>]

Мощно средство за обработка на списъци, защото:

- С един ред изразяваме итерация
- Пресмятаме израза за всеки елемент от контейнера
- Филтрираме елементите от контейнера с условия
- Елегантно и сигурно влагаме цикли един в друг

```
odd_numbers = [1, 5, 7, 9, 3, 11]
even_numbers = [2*n for n in odd_numbers]
```

Създаване на списъци с цикъл for

```
>>> vec  =  [2 ,4 ,6 ]
>>>  [3*x for x in vec ]
[6 , 12, 18]
>>>  vec1 = [2, 4, 6]
>>>  vec2 = [4, 3, -9]
>>>  [x*y for x in vec1  for y in vec2]
[8, 6, -18, 16, 12, -36, 24, 18, -54]
>>>  [x+y for x in vec1 for y in vec2]
[6, 5, -7, 8, 7, -5, 10, 9, -3]
```

Създаване на контейнери с цикъл for

✓ Формула: Създаването на контейнер в цикъл включва две основни части оградени в скоби

✓ <отв.скоба> + <израз> + <контейнер-контекст> + <затв.скоба>

✓ Скобите определят вида на контейнера, който се създава (списък, редица, множество или речник).

✓ Първата основна част е израз, който се пресмята при всяка итерация на цикъла, и породената стойност се записва като пореден елемент в създавания контейнер.

✓ Втората основна част се задава с една или повече клаузи за цикъл for, като за всяка от тях може да има и съответна клауза if. Чрез клаузите for се определя породността и вложеността на всеки нов елемент в контейнера, а с клаузите if може да се задават условия дали породния пресметнат обект да се запише в контейнера или не.

Създаване на контейнери с цикъл for

✓ Когато се създава списък, множество или речник, резултата е обект от този тип – списък, множество или речник.

✓ Когато се създава редица, резултата е обект итератор, който включва като елементи съответните елементи от редицата.

Създаване на множества с цикъл for

```
>>> {x for x in range(10)}
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
>>>

>>> {x % 6 for x in range(10)}
{0, 1, 2, 3, 4, 5}
>>>
```

Създаване на речници с цикъл for

```
>>> D = {x: x*2 for x in range(10)}
>>> D
{0:0, 1: 2, 2: 4, 3: 6, 4: 8, 5: 10, 6: 12, 7: 14, 8: 16, 9: 18}

>>> D = {c: c * 3 for c in 'STRING'}
>>> D
{'S': 'SSS', 'T': 'TTT', 'R': 'RRR', 'I': 'III', 'N': 'NNN', 'G': 'GGG'}
```

Създаване на редици с цикъл for

```
>>> (x for x in range(10))
<generator object <genexpr> at 0x0000000001E48CF0>
>>> print(_)
<generator object <genexpr> at 0x0000000001E48CF0>
>>> for x in _: print(x)
0
1
2
3
4
5
6
7
8
9
```

Комбиниране на for с if

```
>>> vec = [2,4,6]
>>> [3*x for x in vec if x > 3]
[12, 18]
>>> [3*x for x in vec if x < 2]
[]
>>> [5*x for x in vec if x<5]
[10, 20]
>>>
```

Примери и съвети за използване

Сложни функции в един ред:

```
def list_prod(vec1, vec2):  # умножение на вектори
   return [u*v for u, v in zip(vec1, vec2)]
```

Филтриране:

```
nums = [-1.2, 0.5, 12, 1.8, -9.0, 5.3]
good_nums = [r for r in nums if r > 0]
```

Лош пример за влагане:

```
orig = [15, 30, 78, 91, 25]
finals = [min(s, 100) for s in [f+5 for f in orig]]
```

Добър пример:

```
finals = [min(s+5, 100) for s in orig]
```

Комбиниране на два цикъла for, и на if с else

```
>>> users = ['Петя', 'Зорка', 'Гинка']
>>> pairs = [(x,y) for x in users for y in users if x != y]
>>> pairs
[('Петя', 'Зорка'), ('Петя', 'Гинка'), ('Зорка', 'Петя'),
   ('Зорка', 'Гинка'), ('Гинка', 'Петя'), ('Гинка', 'Зорка')]
>>> even = [x if x%2 == 0 else x-1 for x in range(10)]
>>> even
[0, 0, 2, 2, 4, 4, 6, 6, 8, 8]
>>>
```

Синтаксис на enumerate

enumerate(iterator, count=0)

Резултатът е създаване на специален обект от тип итератор, който е контейнер от двойки (редици с по два елемента).

Първият елемент от обекта резултат е редица с два елемента: първата стойност на брояча count (0 по премълчаване) и първият елемент от аргумента iterator

Всеки следващ елемент от резултата е редица (двойка) от два елемента –следващата стойност на брояча и следващият елемент от iterator.

Използване на enumerate

```
>>> S = 'spam'
>>> for (offset, item) in enumerate(S):
... print(item, 'appears at offset', offset)
...
s appears at offset 0
p appears at offset 1
a appears at offset 2
m appears at offset 3
```

Използване на enumerate и next

```
>>> E = enumerate(S)
>>> E
<enumerate object at 0x0000000002A8B900>
>>> next(E)
(0, 's')
>>> next(E)
(1, 'p')
>>> next(E)
(2, 'a')
```

Използване на enumerate

```
>>> S = 'spam'

>>> [c * i for (i, c) in enumerate(S)]
['', 'p', 'aa', 'mmm']
>>>
```

Динамичен модел в езика Python

- Всяка променлива се създава при присвояване на стойност към нея, и сочи (е свързана с) някакъв обект, който реално е носител на нейната стойност
- Променливите нямат тип
- Когато са в лявата част на оператор за присвояване, те приемат стойност на израза в дясната част и неговия тип
- Когато се срещнат в някакъв израз, те се заменят със стойността на обекта, към който са свързани

Модел на данните в езика Python

- Променливите и обектите, към които те сочат, се намират в различни части на ОП
- В таблицата на променливите, за всяка променлива се съхранява нейното име (низ) и адреса на обекта (в таблицата на обектите в ОП), към който тя сочи (id() на свързания обект)
- В таблицата на обектите, за всеки обект се съхранява неговия тип данни, началния адрес в ОП където се съхранява обекта (неговата стойност), и брояч на променливите (имената) сочещи към обекта. Когато брояча стане 0, обекта може да се изтрие от ОП.

Mutable / immutable обекти

- Всеки обект има един и същи неизменяем начален адрес към ОП, където се съхранява обекта (неговата стойност).
- Ако обекта е immutable, и съдържанието на ОП, сочено от идентификатора на обекта, не може да се изменя (не може да променя своята стойност) .
- Ако обекта е mutable, съдържанието на ОП, сочено от идентификатора на обекта, може да се изменя (да приема различна стойност).

Представяне на контейнер в езика Python

- В ОП, където се съхранява обект контейнер, се съхраняват адресите към обектите, които са елементи от контейнера.
- Конкретните обекти, които са елементи на контейнера, се съхраняват на друго място в ОП.
- Ако контейнера е mutable, можем да променяме броя и адресите на обектите, съхранявани като негови елементи.
- Ако контейнера е immutable, тогава броя на елементите му е постоянен, и адреса сочещ към всеки елемент не може да се променя.

Промяна в стойност на mutable тип данни

```
>>> a = [1, 2, 3]    # a сочи към списъка [1, 2, 3]
>>> b = a            # b сочи към обекта, сочен и от  a
>>> a.append(4)      # Промяна в обекта списък сочен от a
>>> print(b)         # Тъй като b сочи към същия обект:
[1, 2, 3, 4]
# Стойността на променливата b също е променена …
```

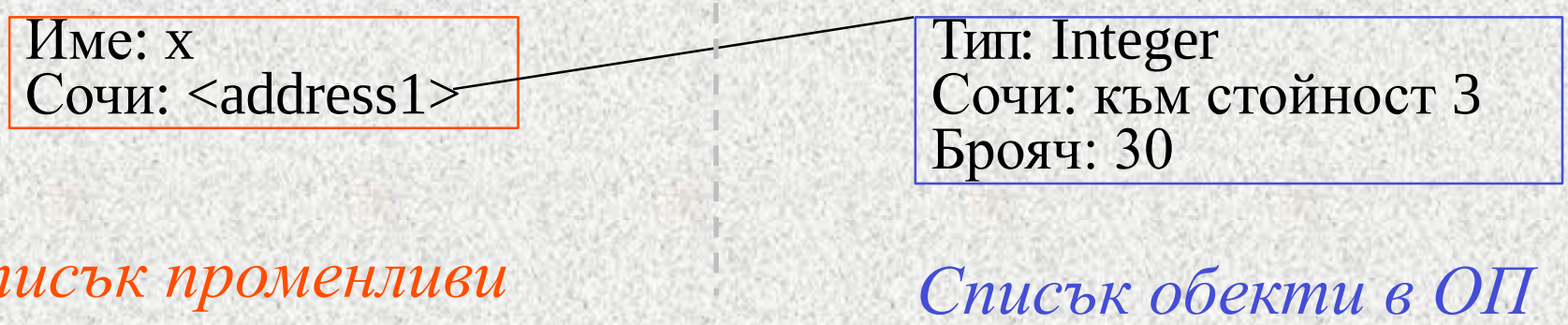
Защо ??

Динамичен модел на обекти и типове данни

Какво се случва когато въведем командата: x = 3

- Създава се обект цяло число *3* в паметта
- Създава се променлива с име *x*
- *Адресът* на паметта, където се съхранява информацията за обекта *3*, се присвоява на променливата с име *x*

Когато казваме че стойността на променливата *x* е *3*, ние имаме в предвид, че променливата *x е свързана с* обект – цяло число *3*



Динамичен модел на обекти и типове данни

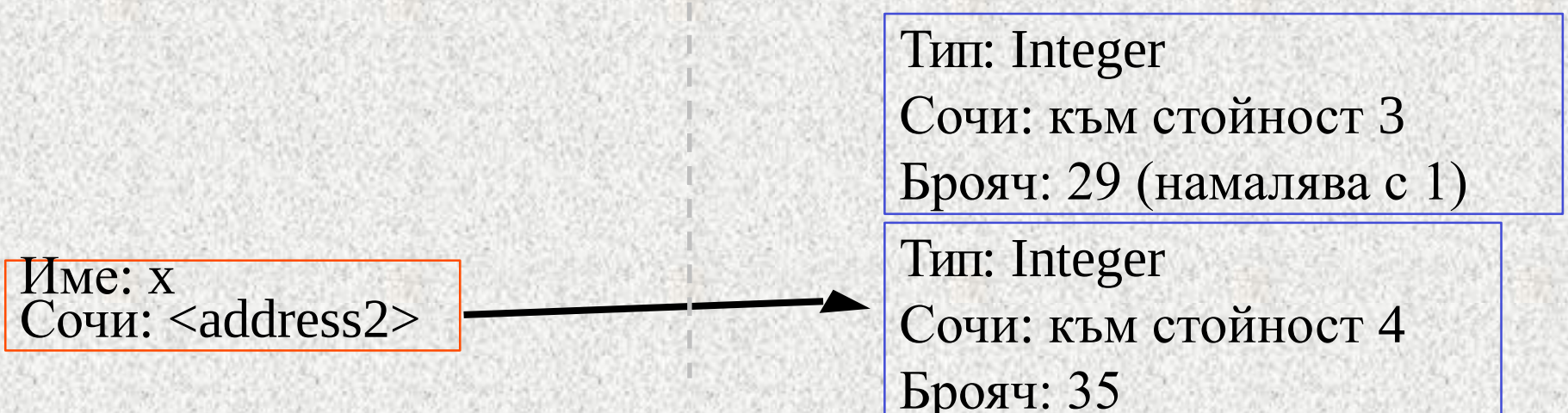
- Обектът 3 който създадохме е от тип цяло число.
- В езика Python типовете данни integer, float, string и tuple са “immutable”
- Това не означава че не можем да променим стойността на x, т.е. *адреса на обекта към който x сочи …*
- Ето пример за промяна в стойността на променливата x:

```
>>> x = 3
>>> x += 1
>>> print(x)
4
```

Динамичен модел на обекти и типове данни

Когато увеличим x, се случва следното:

1. Намира се адреса на обекта сочен от ***x (цяло число)***
2. Взима се стойността на този обект от ОП (3)
3. Изчислява се израза 3+1, получавайки нов обект **4, за** който се създава нов ред в таблицата на обектите, и място в ОП където се записва неговата стойност
4. *Променливата* ***x*** *сочи към този нов обект*



Динамични обекти

За други типове данни (списъци, речници), които са изменяеми, присвояването се извършва по различен начин

- Тези типове данни са **“mutable.”**
- Ние ги променяме *на място (в паметта)*
- Не правим ново копие всеки път
- След команда y=x, когато променливата y е свързана с обект от тип данни – mutable, ако променим което и да е от x или y, това води до промяна на обекта на място в ОП, и като следствие както x така и y се променят

Динамични обекти

immutable

```
x = 3
y = x
y = 4
print(x)
3
```

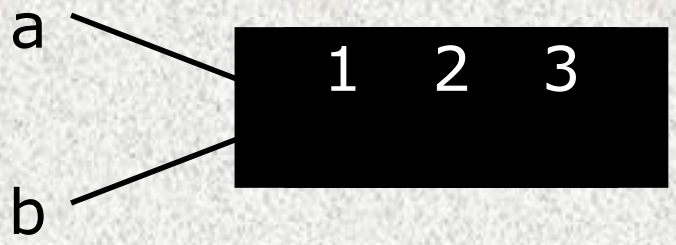
mutable

```
x = [1, 2, 3]
y = x
y.append(4)
print(x)
[1, 2, 3, 4]
```

Защо се получава така?

a = [1, 2, 3]

b = a



a.append(4)

Избягване на този ефект с копиране

```
>>> L1 = [2, 3, 4]
>>> L2 = L1[:]      # Правим копие на списъка L1
                    # (L2=list(L1), L2=L1.copy(), и т.н.)
>>> L1[0] = 24
>>> L1
[24, 3, 4]
>>> L2              # L2 не се променя
[2, 3, 4]
```

Вградена функция print

print([object, ...][, sep=' '][, end='\n'][, file=sys.stdout]
[, flush=False])

- ✓ Всичко поставено в [] е незадължителен елемент
- ✓ Всеки елемент от тип key=val означава, че val е стойност по премълчаване за аргумента key
- ✓ Тази вградена функция извежда в текстуален вид обектите зададени като аргументи, разделени помежду си с низ sep, използвайки end за разделител на редовете, в изходно устройство file с или без използване на буфериране

Аргументи на функцията print

- ✓ Аргументите sep, end, file, flush се задават както елементите в речник от тип key=val
- ✓ sep задава низ, който се поставя между всеки два съседни обекта, зададени като част от 1-и аргумент
- ✓ end задава какво да се сложи като разделител на ред след края на извеждането
- ✓ низа file задава името, където да се запише текстът (в кое изходно устройство или файл)
- ✓ с булевата стойност flush се указва дали В/И буфер да се запише или не след изпълнението на print в изходното устройство (файл)

Действие на функцията print

- ✓ Текстуалното представяне на всеки обект <obj> се получава чрез вградената функция str: str(<obj>)
- ✓ Тя връща в удобен за потребител (“user friendly”) низ стойността на обекта
- ✓ Когато няма аргументи, функцията print извежда символ за край на ред в стандартното изходно устройство, т.е. празен ред на екрана.

Вградена функция print - примери

```
>>> x = 'spam'
>>> y = 99
>>> z = ['eggs']
>>> print(x, y, z, sep='...', file=open('data.txt', 'w'))
# Изход във файл
>>> print(x, y, z) # Изход на екран
spam 99 ['eggs']
>>> print(open('data.txt').read())
# Извеждане съдържанието на файла
spam...99...['eggs']
```

Вградена функция input()

- ✓ Чете и връща следващия ред от стандартното входно устройство. Функцията изчаква потребителя да въведе един ред и символите въведени в реда се присвояват като низ на някаква променлива: I = input()
- ✓ Може да има незадължителен аргумент от тип низ, който се извежда. Логиката е да подскаже на потребителя какво се иска от него да въведе.
- ✓ Може да работи с произволни В/И устройства
- ✓ В Windows се поставя функция input() в края на всеки файл с програма, така че при изпълнение в команден прозорец, той да не се затвори веднага, а да остане докато потребителя разгледа какво се е получило.

Правила за оформяне на програмите

- ✓Правила на общността от автори и поддръжници на езика Python:
- ✓https://www.python.org/dev/peps/pep-0008/

- ✓Правила на Гугъл за програмиране и оформяна на програмите на езика Python:
- ✓http://google.github.io/styleguide/pyguide.html

Съглашения за имената

Python общността използва следните съглашения за имената на обектите в езика:

- ✓Малки букви със '_' за имена на променливи, функции, методи и атрибути
- ✓'_' разделя отделни смислени думи: малък_член
- ✓При имена на класове отделните думи се отделят с голяма първа буква за всяка дума: MyFirstClass
- ✓Големи букви (със '_ ') за имена на константи: PI
- ✓Имена на атрибути: интерфейс, _вътрешни, __частни

Редове и блокове в Python

- **Използва се специалния символ newline (/n) за обозначаване на край на ред (команда)**
- **Не се използват {} за обозначаване на блок команди а се използват измествания с интервали**
  - Първият ред с по-малко изместване *е извън блока*
  - Първият ред с *по-голямо* изместване започва нов вътрешен блок
- **Често започването на нов блок с команди се обозначава с двоеточие (:) – например в условните команди, командите за повторение и други**

Препоръки за начинаещи с Python

- Не забравяйте двоеточието в края на клаузи от сложни команди
- Започвайте всеки нов ред, който не е вложен в блок от команди, в началната позиция (колона 1) - независимо дали пишете във файл или в интерактивния прозорец на интерпретатора (IDLE)
- Празните редове имат значение в интерактивния режим на работа, защото указват край на ред (и от там на команда).
- Празните редове във файл с програма отделят дефинициите на модулите и функциите (обикновено с два празни реда)

Препоръки за начинаещи с Python

- Ако не сте въвели всичко, не натискайте Enter при празен ред, а продължете да въвеждате до края
- Използвайте един и същ начин на изместване във всички вложени блокове команди. Не смесвайте интервали и табулации.
- Интервалите са важни в езика Python: особено за изместванията (4 интервала) и нови редове
- Не използвайте заграждащи скоби () около тестовите клаузи в команди като if, while …

Препоръки за по-ефективен код

- Използвайте команда за многократно изпълнение for вместо while или range – така се получава по-ясен и по-ефективен код
- Внимавайте с присвояване при mutable обекти! Многозначните и късите форми променят тези обекти което може да има нежелан страничен ефект при други променливи или изрази

Препоръки за по-ефективен код

- Внимавайте при обръщение към вградени методи – функции, които променят аргумента (mutable) но не връщат друг резултат освен None. Не присвоявайте резултата от изпълнението им!
- Не забравяйте при обръщение към функция да сложите () – пишете file.close() вместо file.close (последното е валидно обръщение, но не затваря файла!)

Заключение

- Итерация, цикъл, многократно изпълнение
- Клаузи и блокове от команди
- Особености на командите **for** и **while**
- Синтаксис на командите
- Вградени команди **break, continue, pass**
- Клауза **else**
- Филтър **if** в команди за итерация
- Създаване на контейнери с оператор **for**
- Използване на **enumerate** и **next** за итерация
- Динамичен модел за представяне на данните
- Опасности при присвояване на mutable обекти
- Вградена функция print
- Вградена функция input
- Препоръки за писане на качествен код