

Технологии программирования на Python



Тема 3. Управляющие конструкции



python

Условная конструкция, if, else

Очень часто нам нужно, чтобы код выполнялся при определённых условиях. В таком случае используется **условная конструкция**.

```
if логическое_выражение:  
    инструкции  
[elif логическое выражение:  
    инструкции]  
[else:  
    инструкции]
```

Условная конструкция if

```
language = "russian"  
if language == "english":  
    print("Hello")  
    print("World")  
else:  
    print("Привет")  
    print("мир")
```

В Python каждый вложенный блок выделяется отступами. Они играют ту же роль, что фигурные скобки в других языках программирования. Рекомендуется использовать отступы длиной в 4 пробела на каждый уровень вложенности.

Логическое выражение

<	and
>	or
==	not
in	

Конструкция match/case

Начиная с версии 3.10 в python

```
match subject:  
    case <pattern_1>:  
        <action_1>  
    case <pattern_2>:  
        <action_2>  
    case <pattern_3>:  
        <action_3>  
    case _:  
        <action_wildcard>
```

- Конструкция принимает некоторые данные subject, например, переменную.
- subject проверяется на соответствие шаблону pattern, указанному при объявлении case сверху вниз, пока не будет подтверждено соответствие.
- Если соответствие подтверждено - выполняется действие action, указанное после case.
- Если совпадение не подтверждено - выполняется действие wildcard_action, указанное после последнего case _, если такой блок существует.

```
season = 'spring'

match season:
    case 'winter':
        print('зима')
    case 'spring':
        print('весна')
    case 'summer':
        print('лето')
    case 'autumn':
        print('осень')
    case _:
        print('неизвестное время года')

#>>> весна
```

```
season = 'spring'

match season:
    case 'winter' | 'spring' | 'summer' | 'autumn':
        print('известное время года')
    case _:
        print('неизвестное время года')

#>>> известное время года
```

Ещё одним распространённым шаблоном является **шаблон захвата**, используемый для сохранения значения в переменную:

```
transport = 'трамвай'

match transport:
    case 'автомобиль':
        print('еду на автомобиле')
    case 'велосипед':
        print('еду на велосипеде')
    case other_transport:
        print('еду на чём-то другом')

#>>> еду на чём-то другом

print(other_transport)

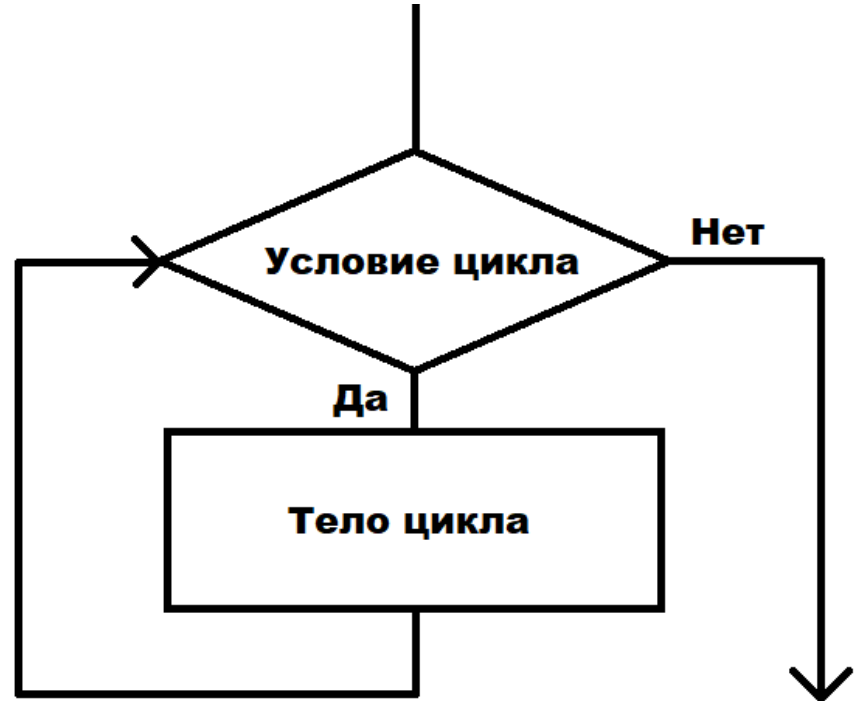
#>>> трамвай
```

Если переменная **transport** соответствует одному из заранее известных значений, будет выполнен соответствующий блок **case**. В случае несоответствия выполнится последний блок **case** с указанием переменной ***other_transport***, а само значение сохранится в эту переменную.

Циклы

Часто задача требует определённое количество раз повторить определённый блок инструкций. Для этого в языках программирования, в том числе в python, существуют специальные конструкции - **циклы**.

Цикл – форма организации действий **исполнителю** многократно повторять указанную последовательность операторов.



В python циклы устроены так же, как в большинстве языков программирования. В теле цикла задан блок инструкций, в условии - число повторений. Одно повторение тела цикла называется итерацией.

Виды циклов

```
graph TD; A[Виды циклов] --> B[С условием]; A --> C[С параметром];
```

С условием

С параметром

Цикл while

Цикл while (пока) выполняет код в теле цикла, пока условие истинно. Обычно цикл while используется в тех случаях, когда точное число итераций заранее неизвестно.

```
while условие:  
    ... <body cycle>
```

При выполнении цикла while сначала проверяется условие. Если оно ложно, то выполнение цикла прекращается. Если условие истинно, то выполняется код в теле цикла, после чего условие проверяется снова. Как только условие станет ложно, работа цикла завершится.

Программа последовательно принимает у пользователя целые числа и выводит их квадраты, пока пользователь не передаст число 0.

```
n = int(input())

while n != 0:
    print(n * n)
    n = int(input())
```

Выражение **n != 0** - это условие цикла. Пока пользователь вводит числа, отличные от 0 - это выражение возвращает True и цикл while продолжает своё выполнение. Как только переменная n становится равной нулю, условие становится ложным и выполнение цикла прекращается.

Цикл for

Другой вид цикла - цикл for (для). В python цикл for используется для перебора элементов итерируемых объектов (подробно с механизмом итерируемых объектов вы познакомитесь позже. Сейчас достаточно знать, что это объекты, состоящие из множества перебираемых элементов. В этом уроке для удобства будем называть их **последовательностями**).

```
for item in collection:  
    <body cycle>
```

На каждой итерации переменной, заданной в условии, присваивается следующий элемент последовательности, а затем выполняется код в теле цикла. Когда закончится последовательность - завершится и работа цикла.

```
for n in "python":  
    print(n)
```

```
#>>> p
```

```
#>>> y
```

```
#>>> t
```

```
#>>> h
```

```
#>>> o
```

```
#>>> n
```

```
for n in [2, 46, 15, 5, 62]:  
    print(n)
```

```
#>>> 2
```

```
#>>> 46
```

```
#>>> 15
```

```
#>>> 5
```

```
#>>> 62
```

break и continue

Иногда выполнение цикла нужно прервать при выполнении определённого условия. Для этого в python существуют операторы **break** (прервать) и **continue** (продолжить).

break прерывает цикл, в котором он объявлен.

```
for symbol in "python":  
    if symbol == "o":  
        break  
    print(symbol)
```

```
#>>> p  
#>>> y  
#>>> t  
#>>> h
```

continue прерывает не весь цикл, а только текущую его итерацию, сразу же переходя к следующей.

```
for symbol in "python":  
    if symbol == "o":  
        continue  
    print(symbol)
```

```
#>>> p  
#>>> y  
#>>> t  
#>>> h  
#>>> n
```

else в циклах

Блок else в циклах - это действия, которые выполняются, когда (и если) выполняется условие выхода из цикла.

Если цикл был прерван как-то иначе, например с помощью break, блок else выполнен не будет.

```
for n in range(5, 0, -1):  
    if n == 2:  
        break  
    print(n)  
else:  
    print("Поехали!")
```

```
#>>> 5
```

```
#>>> 4
```

```
#>>> 3
```


Функция range()

```
for k in [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]:  
    print(k)
```

Реализация цикла **for** в **python** отличается от классической. В **python** цикл **for** пробегает значения последовательности, что соответствует циклу **foreach** в других языках. Но иногда требуется полный функционал классического **for**.

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]



range(11)

```
>>> collect = range(11)
>>> type(collect)
<class 'range'>
>>> print(collect)
range(0, 11)
>>>
```

Объект range – итерируемый объект. Т.е. объект который может итерироваться. Вся последовательность не хранится в памяти. Новый элемент последовательности создается каждый раз когда цикл for обращается к объекту, пытаясь перебрать очередной элемент из последовательности. (Ленивые коллекции).

```
range (end)
```

[0, end)

```
range (start, end)
```

[start, end)

```
range (start, end, step)
```

step – шаг

Функция enumerate()

Функция `enumerate()` конструирует генератор по переданной в нее (через аргумент) объект. Она предоставляет кортежи, состоящие из двух элементов, первый из которых – индекс, а второй – значение, извлекаемое из объекта.

Найти в строке первое вхождение символа 'o' и вывести номер его позиции.

```
msg = "hello!"
i = 0
for sym in msg:
    if sym == 'o':
        print("index = ", i)
        break
    i += 1

# index = 4
```

```
msg = "hello!"
for tp in enumerate(msg):
    if 'o' in tp:
        print("index = ", tp[0])

# index = 4
```

В процессе работы цикла `for` из объекта, созданного функцией `enumerate()`, будут последовательно извлекаться следующие кортежи:

(0, 'h')
(1, 'e')
(2, 'l')
(3, 'l')
(4, 'o')

Функция map()

Функция map() предоставляет возможность применить указанную функцию к каждому элементу объекта. В результате получим список из модифицированных элементов исходного объекта.

```
>>> collect = [2,5,7]
>>> s = map(str, collect)
>>> type(s)
<class 'map'>
>>> print(s)
<map object at 0x000001DBD1AF9750>
>>> print(list(s))
['2', '5', '7']
>>>
```

Функция zip()

Функция zip() позволяет в одном цикле for производить параллельную обработку данных. Это очень мощный инструмент! Zip принимает в качестве аргументов объекты, элементы которых будут объединены в кортежи, полученную структуру можно превратить в список кортежей, если это необходимо.

```
a = [1, 3, 5, 7, 9]
b = [2, 4, 6, 8, 10]
print(list(zip(a, b)))

# [(1, 2), (3, 4), (5, 6), (7, 8), (9, 10)]
```

С помощью функции zip() можно создавать словари.

```
a = [1, 3, 5, 7, 9]
keys = ['a', 'b', 'c', 'd', 'e']
print(dict(zip(keys, a)))

# {'d': 7, 'c': 5, 'a': 1, 'b': 3, 'e': 9}
```

Генераторы списков List comprehension

В python существует специальная конструкция - **генератор списков**, позволяющая создавать заполненные списки по определённым правилам.

```
[action for element in sequence]
```

Генератор возвращает список, каждый элемент которого - результат применения действия action к соответствующему элементу element последовательности sequence.

С фильтром:

```
[action for element in sequence [if<условие>]]
```

stepik > ex1.py

Project ex1.py

stepik D:\Python\Projects\stepik

ex1.py

> External Libraries

Scratches and Consoles

```
1 N = 6
2 # a = [0] * N
3 #
4 # for i in range(N):
5 #     a[i] = i ** 2
6
7 a = [x ** 2 for x in range(N)]
8 print(a)
9
```

Run: ex1

C:\Python39\python.exe D:/Python/Projects/stepik/ex1.py

[0, 1, 4, 9, 16, 25]

Process finished with exit code 0

Stepik D:\Python\Projects\stepik
ex1.py
ernal Libraries
atches and Consoles

```
1 d_inp = input("Целые числа через пробел: ")  
2  
3 a = [int(d) for d in d_inp.split()]  
4 print(a)  
5
```

ex1 X

C:\Python39\python.exe D:/Python/Projects/stepik/ex1.py

Целые числа через пробел: 1 2 3 4

[1, 2, 3, 4]

Process finished with exit code 0



```
>>> b = [1] * N
```

```
>>> b
```

```
[1, 1, 1, 1, 1, 1, 1]
```

```
>>> a = [x % 4 for x in range(N)]
```

```
>>> a
```

```
[0, 1, 2, 3, 0, 1, 2]
```

```
>>> a = [x % 2 == 0 for x in range(N)]
```

```
>>> a
```

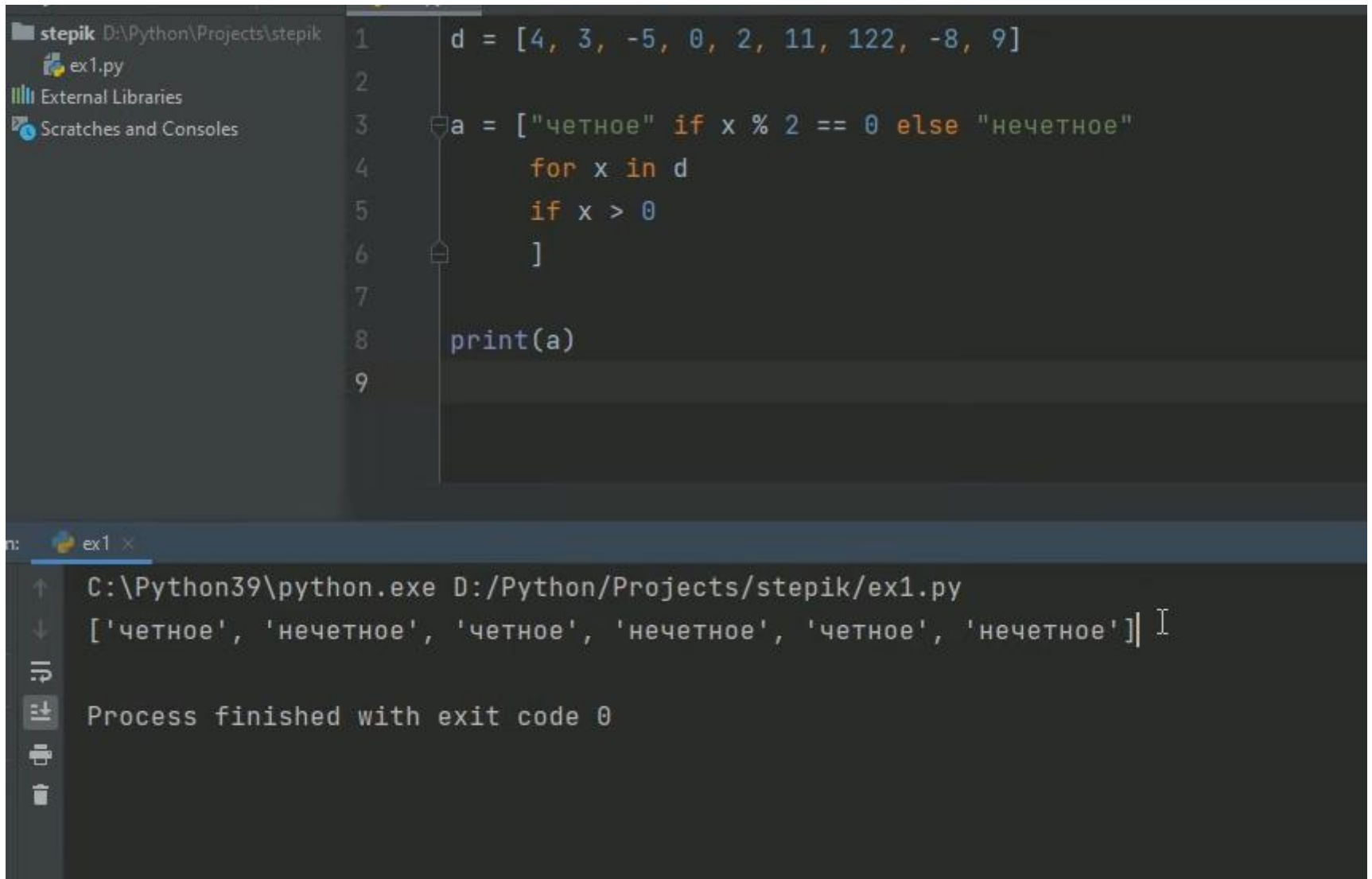
```
[True, False, True, False, True, False, True]
```

```
>>> a = [0.5 * x + 1 for x in range(N)]
```

```
>>> a
```

```
[1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0]
```

```
>>>
```



The image shows a Python IDE with a dark theme. The left sidebar contains a file explorer with 'stepik' (D:\Python\Projects\stepik), 'ex1.py', 'External Libraries', and 'Scratches and Consoles'. The main editor displays a Python script with line numbers 1 through 9. The script defines a list 'd' with values [4, 3, -5, 0, 2, 11, 122, -8, 9], then creates a list 'a' using a list comprehension that checks if each 'x' in 'd' is even or odd. The list 'a' is then printed. The bottom console shows the command to run the script, the resulting output list, and a confirmation that the process finished successfully.

```
1 d = [4, 3, -5, 0, 2, 11, 122, -8, 9]
2
3 a = ["четное" if x % 2 == 0 else "нечетное"
4       for x in d
5       if x > 0
6       ]
7
8 print(a)
9
```

ex1 x

C:\Python39\python.exe D:/Python/Projects/stepik/ex1.py

['четное', 'нечетное', 'четное', 'нечетное', 'четное', 'нечетное']

Process finished with exit code 0