KARPOV.COURSES >>>
КОНСПЕКТ

> Конспект > 2 урок > Основы программирования Python. Условные конструкции и циклы

> Оглавление 2 урока

Скачать Ноутбук

Скачать в PDF

> Условные конструкции

Условные конструкции — логические структуры, которые в зависимости от истинности
или ложности суждений выполняют определённую операцию. По сути это структуры для реализации
механизма принятия решений.

Они позволяют задавать условие, благодаря которому выполняется тот или иной заданный код.

Немного отвлечёмся и поговорим про отступы в Python, которые задают структуру кода:— весь синтаксис построен на работе с блоками;

— блоки выделяются отступами ;

— отступы однородны внутри блока.

«Классическая» конструкция часто имеет вид:

```
if какое-то условие:
    Код, который выполняется, если условие верно
else:
    Код, который выполняется, если условие не верно
```

Где после оператора `if` следует **какое-то условие**.

При соблюдении условия выполняется много полезного или не очень кода. Если какое-то условие не соблюдается, выполняется другой не менее полезный код, стоящий перед оператором `else`.

Пример исполнения

Пример:

```
r2 = 0.8
if r2 > 0.9:
    print('Да, все верно')
else:
    print('R^2 меньше или равно 0.9')
```

Output:

```
R^2 меньше или равно 0.9
```

Если условие маленькое, то его можно записать в одну строчку (но лучше не злоупотреблять такой записью).

Структура следующая: в начале пишется код, который должен быть выполнен, если условие в `if` истинно. Далее задаётся условие и код, который должен быть выполнен,
если это условие неверно.

```
print('Да, все верно') if r2 > 0.9 else print('R^2 меньше или ра
```

Output:

```
R^2 меньше или равно 0.9
```

Можно задавать множественные условия (согласно классическим логическим правилам):

```
if 0.5 < r2 < 0.9:
    print('R^2 в диапазоне (0.5, 0.9)')
else:
    print('Иначе')
```

Output:

```
R^2 в диапазоне (0.5, 0.9)
```

Если требуется задать более чем 2 варианта действий, то необходимо воспользоваться следующей конструкцией:

```
if условие №1:
    Код, который выполняется, если условие №1 верно
elif условие №2:
    Код, который выполняется, если условие №2 верно, а предыдущ
    ...
else:
    Код, который выполняется, если ни одно из условий не верно
```

Операторов `elif` может быть сколько угодно в зависимости от задачи.

Пример

```
user = 'vasya96'

actress_page = {'ne_tvoya', 'nogotochki', 'igor_kot'}
surfing = {'healthy_guy', 'cheap_tan'}

if user in actress_page:
    print('Вася подписан на актрису')
elif user in surfing:
    print('Вася интересуется серфингом')
else:
    print('Подписок Васи не нашли')
```

Output:

```
Подписок Васи не нашли
```

Ключевое слово pass

Ключевое слово `pass` необходимо использовать, если по каким-то причинам нужно, чтобы при выполнении условия ничего не происходило. Это связано с тем, что данная конструкция
не может быть пустой:

```
if expressions:
    code
```

Пример

```
cars = {'igor_kot', 'vasya96', '50cent'}

if user in actress_page:
    print('Вася подписан на актрису')
```

```
elif user in cars:
    pass
```

isinstance(variable, type)

Для проверки того, какой тип данных у переменной, использование `type` является не лучшим решением. Для таких ситуаций лучше использовать функцию `isinstance(variable, type)`, которая возвращает `True`, если переменная `variable` имеет тип `type.`

Пример

```
print(isinstance(1, int))
print(isinstance('str', list))
```

Output:

```
True
False
```

Проверка нахождения элемента в структуре

Для проверки того, что в некоторой структуре или строке содержится элемент, можно использовать следующую конструкцию:

`element in structure`

Пример

```
if 'a' in 'Hello':
    print('Да, оно тут')
else:
    print('Не повезло')
```

Output:

```
Не повезло
```

> Циклы

В Python существует 2 вида циклов:

1. `while` (до тех пор, пока).
2. `for` (для всех элементов из переданного множества).

Цикл while

`while` — выполняет заданные действия до тех пор, пока `условие` является истинным.

```
while условие:
    Выполняемый код
```

Пример:

```
tasks = [] #Создание пустого списка
while len(tasks) < 3:
    tasks.append('new task') #добавляем задачи, пока кол-во зада
    print(len(tasks), tasks)
```

Output:

```
1 ['new task']
2 ['new task', 'new task']
3 ['new task', 'new task', 'new task']
```

Цикл for

`for` — выполняет заданные действия для каждого элемента итерируемой переменной. Переменная цикла меняет своё значение при итерации.

В цикле рекомендуется придумывать специфические имена, чтобы не переопределять переменные, объявленные ранее.

```
for переменная in «последовательность»:
    выполняется столько раз, сколько элементов в последовательно
```

В качестве `iterable_variable` можно использовать:

1. `str`
2. `dict`
3. `set`
4. `tuple`
5. `list`

Итерация по элементам в str

```
upper_letters = []
for letter in 'word':
    upper_letters.append(letter.upper()) #записываем в список бу
upper_letters
```

Output:

```
['W', 'O', 'R', 'D']
```

Этот результат можно "склеить" в одну строку:

```
''.join(upper_letters)
print(upper_letters)
```

Output:

```
WORD
```

Итерация по символам в string

```
for letter in 'word':
     print(letter.upper())
```

Output:

```
W
O
R
D
```

Итерация по элементам в dict

Если итерироваться по одному элементу словаря (`e`), мы получим только ключи.

```
# Со словарями нужно быть внимательнее
demo_dict = {
    'key_0': 'value_0',
    'key_1': 'value_1',
    'key_2': 'value_2'
}
for e in demo_dict:
    print(e)
```

Output:

```
key_0
key_1
key_2
```

Чтобы вывести ключи и значения, нужно задать две переменные и воспользоваться функцией `items()`. В данном случае `k, v`, где `k` — ключи, `v` — значения.

```
demo_dict = {
    'key_0': 'value_0',
    'key_1': 'value_1',
    'key_2': 'value_2'
}
for k, v in demo_dict.items():
    print(k, v)
```

Output:

```
key_0 value_0
key_1 value_1
key_2 value_2
```

Итерация по элементам в tuple

В ходе итерации будет перебираться каждый элемент (аналогично итерации по списку):

```
for element in (0, 2, 10):
     print(element)
```

Output:

```
0
2
10
```

> Управление циклами

Это необходимо, если вы хотите выполнять цикл до определённого события и остановиться после него или пропустить текущую итерацию.

1. `break` — прерывает выполнение цикла, в котором написан.
2. `continue` — завершает текущую итерацию и переводит цикл на следующую.

Break

Останавливает выполнение цикла.

```
sentence = 'Он шел по дороге и смотрел по сторонам'
for word in sentence.split():
    if word == 'и': #Не надо проверять условие на равенство с Т
        break
    print(word)
```

Output:

```
Он
шел
по
дороге
```

Continue

Основной смысл — продолжить выполнение итерации, если условие верно. Это может потребоваться, когда у вас нет необходимости что-то делать с объектом.

```
for index in range(10):
    if index == 0:
        continue
    print(index)
```

Output:

```
1
2
3
4
5
6
```

```
7
8
9
```

Пример цикла, который будет отправлять некоторое сообщение пользователю, пока пользователь не введёт слово «отмена», чтобы прекратить получать сообщение:

```
while True:
    user_input = input()
    if user_input == 'отмена':
        break
    print('прогноз')
```

Конструкция `while True:` будет всегда истинной, поэтому, используя её, очень легко получить бесконечный цикл.

Полезные функции для циклов

1. `enumerate(structure)` — на каждой итерации создаёт пару (индекс, элемент последовательности), есть возможность опустить обращение к элементам по индексу.

```
for index, char in enumerate('String'):
    print(f'{index}: {char}')
```

Output:

```
0: S
1: t
2: r
3: i
4: n
5: g
```

2. `zip(structure_1, structure_2,...)` — позволяет итерироваться сразу по нескольким структурам. Важно, чтобы они были одного размера.

```
users = ['user1', 'user2', 'user3']
ages = [33, 29, 32]
weights = (102, 62, 93)
for user, age, weight in zip(users, ages, weights):
    print(user, age, weight)
```

Output:

```
user1 33 102
user2 29 62
user3 32 93
```

> Comprehensions

Это способ, позволяющий получить список/словарь/сет на основе другого списка/словаря/сета с помощью более короткой записи.

List comprehension

```
newlist = [expression for item in iterable_value if condition ==
```

Dict comprehension

Если `iterable_value` — это словарь, то в качестве `item` должна быть пара `ключ, значение`

```
newdict = {key_expression: value_expression for item in iterable
```

Set comprehension

```
newset = {expression for item in iterable_value if condition ==
```

Пример

Получим список из нечётных цифр с помощью цикла `for`

```
digits = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
# Хотим получить список из нечетных цифр
odd_digits = []
for digit in digits:
    if digit % 2 == 1: # если цифра нечетная, то добавляем в спи
        odd_digits.append(digit)
print(odd_digits)
```

Output:

```
[1, 3, 5, 7, 9]
```

А теперь выполним то же самое с помощью list comprehension

```
digits = [0,1,2,3,4,5,6,7,8,9]
odd_digits = [digit for digit in digits if digit % 2 == 1]
print(odd_digits)
```

Output:

```
[1, 3, 5, 7, 9]
```

`expression` может содержать в себе условия, но не в качестве фильтра, а в качестве способа взаимодействия с элементом `iterable_value`

```
digits = [0,1,2,3,4,5,6,7,8,9]
odd_digits = [digit if digit % 2 == 1 else -1 * digit for digit
print(odd_digits)
```

Output:

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

> Дополнительные материалы

1. Python if else
2. Ещё про оператор if else в Python
3. Таблица истинности
4. Циклы и петли
5. Статьи про цикл while и for
6. List comprehension