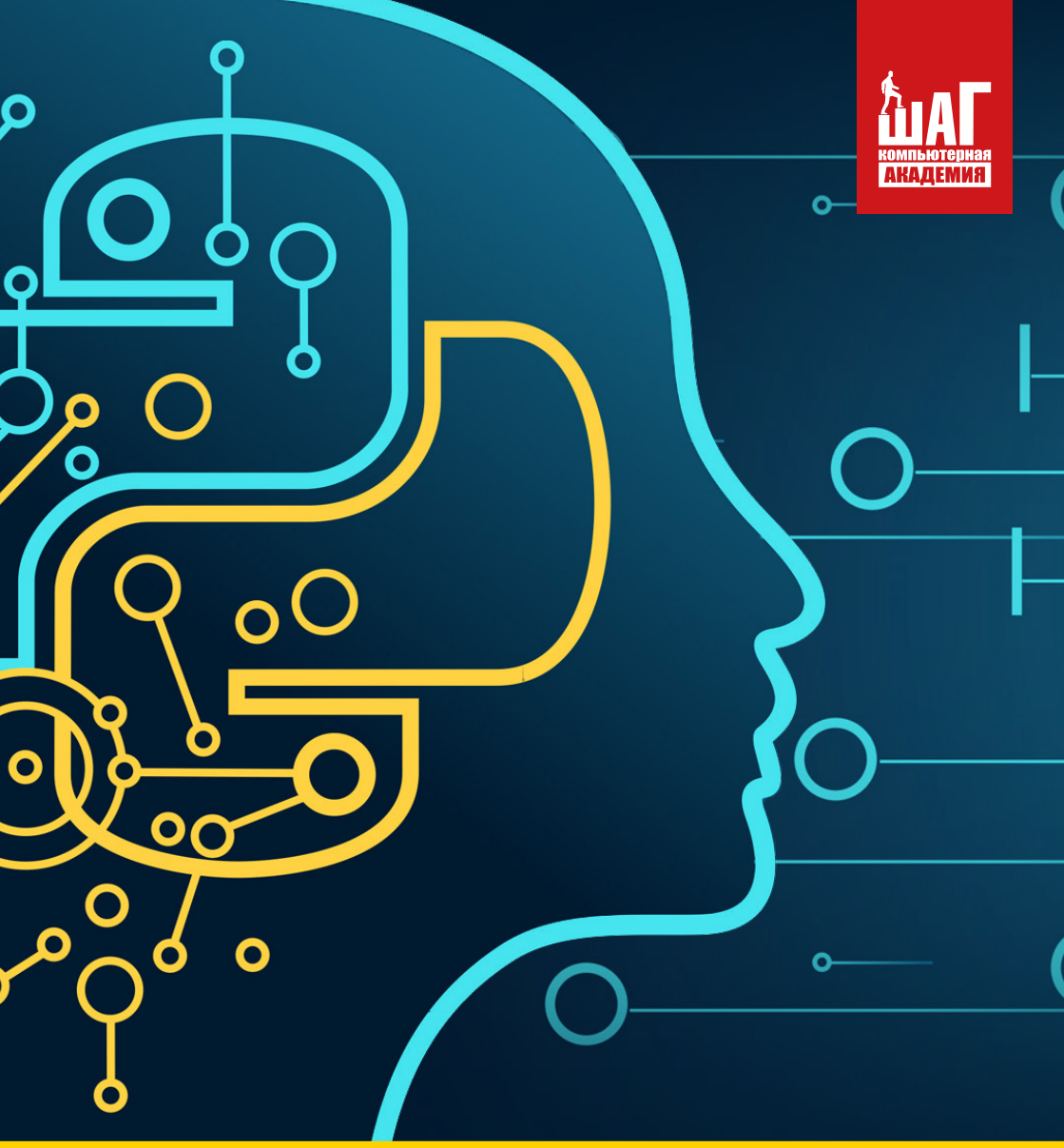

ШАГ
компьютерная
АКАДЕМИЯ

# Основы программирования на языке Python

# Урок 2

## Операторы ветвлений, циклы, исключения

## Содержание

# Введение

В предыдущем модуле рассматривались примеры, в которых инструкции выполнялись последовательно, однако иногда требуется пропустить некоторые инструкции или выбрать, какая инструкция будет выполнена. В программировании для предоставления вариативности использования и многофункциональности применяются условные конструкции, конструкции для обработки исключений и циклы.

В этом модуле будут подробно рассмотрены:

- особенности работы с операторами ветвления, их комбинирование;
- особенности обработки исключений и их типизация;
- циклы, их типы и особенности использования.

# Условные инструкции и их синтаксис

Операторы ветвлений (или условные инструкции) позволяют строить простые конструкции, перенаправляющие выполнение программы подобно лифту, который определяет общий вес пассажиров и на основании этой информации начинает перемещаться или сообщает о перевесе. Также, примером такой конструкции служит сканер отпечатков пальцев, который дает или отказывает в доступе, в зависимости от их совпадения с отпечатками зарегистрированных пользователей.

В общем случае, для построения условной конструкции нужно:

- указать оператор ветвления;
- разделить оператор и условие пробелом;
- указать условие;
- поставить двоеточие в конце условия;
- указать набор инструкций;
- выделить набор инструкций отступом.

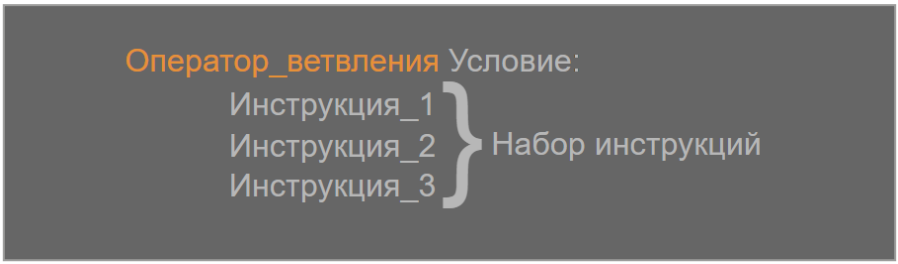


*Рисунок 1*

В программировании условия так же называют логическими выражениями. Для начала выясним, что представляют собой логические выражения и наборы инструкций.

## Логические выражения и операторы

Наряду с арифметическими и прочими операциями, в программировании повсеместно используются операции сравнения, которые позволяют создавать логические выражения на основании сравнений, и логические операции, позволяющие комбинировать несколько логических выражений в одно.

Логические выражения — любые конструкции, результатом выполнения которых является True или False.

Человеческое доверие демонстрирует работу логических выражений. Каждый раз, когда мы слышим какое-либо высказывание, мы или верим ему (т. е. считаем истинным), или относимся к нему с недоверием (т. е. считаем ложным). Например, когда ваш друг позвал вас на прогулку и сказал, что на улице хорошая погода, может произойти две ситуации:

- вы посмотрите на улицу, убедитесь, что погода хорошая (т. е. утверждение друга истинно), и вы пойдете на прогулку;
- вы посмотрите на улицу, увидите, что погода плохая (т. е. утверждение друга ложно), и вы останетесь дома.

Конечно, в реальности мы можем сомневаться, однако в программировании оценка выражений всегда сводится к одному из этих двух вариантов. Для представления истинности и ложности используются ключевые слова True и False соответственно.

### *Операции сравнения*

Операции сравнения используются для создания условий на основании сравнения элементов. Элементы, к которым применяется операция, называют операндами. Python поддерживает следующие операции сравнения:

- `==` — «равно»: истинно (True), если оба операнда равны, иначе — ложно False;
- `!=` — «не равно»: истинно (True), если оба операнда не равны, иначе — ложно False;
- `>` — «больше чем»: истинно (True), если первый операнд больше второго, иначе — ложно False;
- `<` — «меньше чем»: истинно (True), если первый операнд меньше второго, иначе — ложно False;
- `>=` — «больше или равно»: истинно (True), если первый операнд больше или равен второму, иначе — ложно False;
- `<=` — «меньше или равно»: истинно (True), если первый операнд меньше или равен второму, иначе — ложно False.

```
print("1 == 1:", 1 == 1)      1 == 1: True
print("1 == 2:", 1 == 2)      1 == 2: False
print("1 != 1:", 1 != 1)      1 != 1: False
print("1 != 2:", 1 != 2)      1 != 2: True
print("1 > 1:", 1 > 1)        1 > 1: False
print("1 > 2:", 1 > 2)        1 > 2: False
print("2 > 1:", 2 > 1)        2 > 1: True
print("1 < 1:", 1 < 1)        1 < 1: False
print("1 < 2:", 1 < 2)        1 < 2: True
print("2 < 1:", 2 < 1)        2 < 1: False
```

```
print("1 >= 1:", 1 >= 1)      1 >= 1: True
print("1 >= 2:", 1 >= 2)      1 >= 2: False
print("2 >= 1:", 2 >= 1)      2 >= 1: True
print("1 <= 1:", 1 <= 1)      1 <= 1: True
print("1 <= 2:", 1 <= 2)      1 <= 2: True
print("2 <= 1:", 2 <= 1)      2 <= 1: False
```

На сером фоне представлен код, который демонстрирует результаты работы операций сравнения с целыми числами. На чёрном фоне отображается результат работы нашего кода в консоли.

### *Использование значений в качестве условий*

В Python существуют правила, согласно которым можно привести любое значение любого типа к логическому. Так, если вместо условия написать строку или число, программа заменит его на True или False.

Значения, которые будут заменены на False:

- структуры, не содержащие элементы:
  - строки без символов;
  - пустые списки, словари, массивы и т.д.
- нули любых численных типов:
  - 0;
  - 0.0.
- константа None.

Значения, которые будут заменены на True:

- структуры, содержащие элементы:
  - строки любой длины, отличной от нуля;
  - списки, словари, массивы и т.д. с элементами;

- ненулевые значения любых численных типов:
  - 1, 2, 3…;
  - 0.1, 1.2, 2.3… .

С помощью функции bool() можно проверить результат любого такого преобразования:

```
print(bool(""))                    False
print(bool(0.0))                   False
print(bool(None))                  False
print(bool("IT Step Academy"))     True
print(bool(1))                     True
```

На сером фоне представлен код, который демонстрирует результаты работы операций сравнения с целыми числами. На чёрном фоне отображается результат работы нашего кода в консоли.

### *Логические операции*

Логические операции позволяют комбинировать несколько логических выражений в одно. В Python имеются следующие логические операторы:

- and

«Логическое умножение»: возвращает True, если оба выражения равны True. Например, работодатель, заинтересованный в хорошем сотруднике, возьмет человека на работу, если он компетентный И ответственный:

```
competent = True
responsible = True
print(competent and responsible)
```

Результат работы нашего кода в консоли:

```
True
```

Если же человек компетентен но не ответственен:

```
competent = True
responsible = False
print(competent and responsible)
```

Результат работы нашего кода в консоли:

```
False
```

- or

«Логическое сложение»: возвращает True, если хотя бы одно из выражений равно True. Например, работодатель, заинтересованный в найме сотрудника в кратчайшие сроки, возьмет человека на работу, если он компетентный ИЛИ ответственный:

```
competent = True
responsible = False
print(competent or responsible)
```

Результат работы нашего кода в консоли:

```
True
```

Если же человек не компетентен и не ответственен:

```
competent = False
responsible = False
print(competent or responsible)
```

Результат работы нашего кода в консоли:

```
False
```

- not

«Логическое отрицание»: возвращает значение, обратное операнду. Например, работодатель НЕ возьмет человека, который ранее был уволен:

```
previously_fired = True
print(not previously_fired)
```

Результат работы нашего кода в консоли:

```
False
```

В противном случае:

```
previously_fired = False
print(not previously_fired)
```

Результат работы нашего кода в консоли:

```
True
```

Обратите внимание, что если один из операндов оператора and возвращает False, то другой операнд не оценивается, так как результатом выполнения оператора в любом случае будет False. Аналогично, если один из операндов оператора or возвращает True, то второй операнд не оценивается, так как результатом выполнения оператора в любом случае будет True.

Рассмотрим работу логических операций на простом примере: возвращаясь домой, человек захотел воспользоваться

метро. Он помнит, что метро работает с 6 утра и до 24 (или 0), для проезда ему необходимо предъявить билет или оплатить поездку, кроме того, его не пустят с большим багажом. На часы он еще не смотрел, билет забыл дома, денег на проезд у него хватает, и он без багажа:

```
time = int(input("Enter the current time in hours: "))% 24
ticket = False
money = True
luggage = False
print(money or ticket and not luggage and time > 6)
```

На первый взгляд наше условие верно. Однако, оно истинно вне зависимости от значений time и luggage.

Классический пример из математики гласит , потому как сначала нужно умножить, а потом прибавить. Такой специфический порядок вычислений определен приоритетом операций. То же самое есть и у логических операций: сначала выполняется not, затем and, затем or. Именно поэтому выполняющийся в последнюю очередь or вернет True. Как и с математическими операциями, для переопределения порядка выполнения можно использовать скобки:

```
print((money or ticket) and not luggage and time > 6)
```

На практике простейшим примером применения логических выражений является if: Если условие истинно — выполняется набор инструкций:

```
car_speed = 100
if car_speed > 50:
   print("Car is faster than 50 km/h")
```

В этом случае результатом выполнения кода будет вывод на консоль строки «You have at least 3 flowers», т.к. выражение 2 < 3 истинно.

```
passenger_weight = 400
if passenger_weight < 300:
   print("The elevator can go")
```

В этом примере лифт не сможет поехать, и строка «The elevator can go» не будет выведена на консоль, т.к. выражение 400 < 300 ложно.

Рассмотрим пример применения логических операций с if:

```
car_speed = 100
if car_speed > 50 and car_speed < 150:
  print("Car speed is between 50 km/h and 150 km/h")
```

В данном примере строка «Car speed is between 50 km/h and 150 km/h» будет выведена в случае, если переменная «car_speed» находиться в промежутке от 50 до 150 (не включительно). Python позволяет записать такое условие подобно двойному неравенству в математике:

```
if 50 < car_speed < 150:
```

Известно, что високосным годом считается тот год, который кратен 4, не кратен 100 или кратен 400. Запишем условное выражение, определяющее, является ли год високосным:

```
year = 2000
if year % 4 == 0 and year % 100 != 0 or year % 400 == 0:
   print("Year", year, "is leap")
```

## Понятие «блока» выполнения

Ранее был рассмотрен общий случай использования оператора ветвления, в котором упоминался «набор инструкций». Однако, каким образом определить, что инструкция принадлежит к этому набору или, другими словами, «блоку выполнения»?

### *Определение блока выполнения*

В Python, в отличие от многих других языков, инструкции, принадлежащие к одному блоку выполнения, достаточно выделить одинаковым отступом. Концом блока выполнения считается последняя инструкция, которая будет выделена соответствующим отступом.

В играх подобным образом определяются комбинации или «комбо»: успешные действия, идущие друг за другом, образуют «комбо», которое прерывается, как только действие будет выполнено неудачно. Рассмотрим пример блока выполнения:

```
if 1 > 4:                                    # line 1
    print("This is the start
          of an execution block")            # line 2
    print("This is part
          of the execution block")           # line 3
    print("This is still part
          of the execution block")           # line 4
    print("This is the end
          of an execution block")            # line 5
    print("It is not part
          of the execution block")           # line 6
```

В этом случае результатом выполнения кода будет вывод на консоль строки «It is not part of the execution

block», т. к. выражение 1 > 4 ложное и инструкции, принадлежащие к блоку выполнения конструкции, не будут выполнены. В частности, к нему принадлежат строки от 2 до 5.

Отступы должны состоять из пробелов и, согласно руководству по оформлению кода, их количество желательно делать кратным 4 (т. е. 4 пробела, 8, 16…). Хотя если блок будет выделен отступом в 2 пробела, программа также будет работать. Кроме того, в последних версиях языка для отступа нельзя использовать табуляцию, однако современные среды разработки (в сокращенном варианте «IDE») автоматически размещают 4 пробела при нажатии на клавиатуре кнопки «Tab».

### *Применение*

Определение блоков выполнения необходимо не только при использовании условных конструкций. Они также используются в:

- циклах;

```
while condition:
    print("Cycles")
```

- обработке исключений;

```
try:
    print("Some code")
except:
    print("Error processing")
finally:
    print("Handling Exceptions")
```

- функциях;

```
def function():
    print("Functions")
```

- классах.

```
class Class:
    def __init__(self):
        print("Classes")
```

С этими конструкциями вы познакомитесь позже.

## Операторы ветвления if … else

Теперь, когда вы научились выделять инструкции в блоки, составлять и комбинировать выражения, можно приступить к рассмотрению условных конструкций. Ранее был приведен простейший пример использования условной конструкции if:

```
car_speed = 100
if car_speed > 50:
  print("Car is faster than 50 km/h")
```

Давайте попробуем расширить его на основании полученных знаний. Введем две переменные, обозначающие скорость (в км/ч) машины и мотоцикла соответственно:

```
car_speed = 150
motorcycle_speed = 100
```

В таком случае наш пример можно изменить с использованием переменных:

```
if car_speed > motorcycle_speed:
    print("Car is faster than motorcycle")
```

Результат работы нашего кода в консоли:

```
Car is faster than motorcycle
```

Если изменить скорость так, что машина окажется быстрее, наше условие окажется ложным и пользователь не получит никакой информации. В таком случае удобно будет использовать else — блок, который выполняется только если последнее условие оказалось ложным. Добавим его в наш пример:

```
car_speed = 100
motorcycle_speed = 150
if car_speed > motorcycle_speed:
   print("Car is faster than motorcycle")
else:
   print("Motorcycle is faster than car")
```

Результат работы нашего кода в консоли:

```
Motorcycle is faster than car
```

Скорости могут уравняться и, хотя блок else выполнится, представленная пользователю информация будет неверна. Исправить это можно, добавив еще одно условие:

```
car_speed = 100
motorcycle_speed = 100
```

```
if car_speed > motorcycle_speed:
   print("Car is faster than motorcycle")
if motorcycle_speed > car_speed:
   print("Motorcycle is faster than car")
else:
   print("Car and motorcycle are equally fast")
```

Результат работы нашего кода в консоли:

```
Car and motorcycle are equally fast
```

Здесь условные конструкции выполняются поочередно: сначала проверяется условие motorcycle_speed < car_speed, затем motorcycle_speed > car_speed. Обратите внимание, что второе условие будет проверено вне зависимости от истинности первого. Полезность подобного поведения ситуативная: если эти конструкции самостоятельны и предполагают как независимое, так и совместное выполнение — это полезно, однако если они связаны и предполагается выполнение только одной конструкции — это может привести к ошибкам. Данный пример хорошо справляется со сравнением постоянных скоростей. Если же за время сравнения скорость мотоцикла изменится, мы получим два противоречивых результата:

```
car_speed = 130
motorcycle_speed = 100
if car_speed > motorcycle_speed:
    print("Car is faster than motorcycle")
    motorcycle_speed += 50
if motorcycle_speed > car_speed:
    print("Motorcycle is faster than car")
    motorcycle_speed += 50
```

```
else:
    print("Car and motorcycle are equally fast")
    motorcycle_speed += 50
```

Результат работы нашего кода в консоли:

```
Car is faster than motorcycle
Motorcycle is faster than car
```

Так, при первом сравнении условие motorcycle_speed < car_speed будет истинно, скорость мотоцикла изменится, и при втором сравнении motorcycle_speed > car_speed также окажется истинным. Избежать подобной ситуации можно, объединив два условия в одну конструкцию. Для этого заменим второй if на elif — комбинацию блоков else и if, набор инструкций которого выполняется если условия предыдущих блоков ложны, а собственное условие блока — истинно:

```
if car_speed > motorcycle_speed:
    print("Car is faster than motorcycle")
    motorcycle_speed += 50
elif motorcycle_speed > car_speed:
    print("Motorcycle is faster than car")
    motorcycle_speed += 50
else:
    print("Car and motorcycle are equally fast")
    motorcycle_speed += 50
```

Результат работы нашего кода в консоли:

```
Car is faster than motorcycle
```

## Вложенные конструкции

Блоки выполнения if, elif и else могут содержать другие условные конструкции, которые называют «вложенными». Блок выполнения вложенных конструкций также должен быть отделен отступом, образующим своего рода новую «ступень»:

```
flowers_amount = 3
if flowers_amount > 2:
    print("You have at least 3 flowers")

    if flowers_amount < 5:
        print(""You have less than 5 flowers")
```

С помощью вложенных конструкций можно наглядно продемонстрировать принцип работы блока elif. Допустим, пользователь проходит тест, и должен выбрать один из вариантов ответа, введя его номер:

```
number = int(input("Enter the answer number: "))
if number == 1:
   print("You've chosen answer A")
else:
   if number == 2:
       print("You've chosen answer B")
   else:
       if number == 3:
           print("You've chosen answer C")
       else:
           if number == 4:
               print("You've chosen answer D")
           else:
               print("There is no such answer.")
```

Результат работы нашего кода в консоли:

```
Enter the answer number: 4
You've chosen answer D
```

Такой код, построенный с использованием вложенности, можно упростить, используя elif. При этом результат выполнения кода не изменится:

```
number = int(input("Enter the answer number: "))
if number == 1:
   print("You've chosen answer A")
elif number == 2:
   print("You've chosen answer B")
elif number == 3:
   print("You've chosen answer C")
elif number == 4:
   print("You've chosen answer D")
else:
   print("There is no such answer.")
```

Рассмотрим практическое применение вложенных конструкций на примере копилки:

```
account = int(input("Enter how much you put: "))
account = abs(account)

if account > 0:
    withdrawal = int(input("Enter how much you take: "))
    withdrawal = abs(withdrawal)

    if withdrawal < account:
        account -= withdrawal
        print("Here are your", withdrawal, ".")
        print("There are", account, "left.")
```

```
    else:
        print("There are only", account, ".")
else:
    print("There are no money in piggy bank")
```

В этом примере для ввода пользователем суммы, которую он положил в копилку, используют строки:

```
account = int(input("Enter how much you put: "))
account = abs(account)
```

Причем за получение от пользователя строки и приведение её к целому числу отвечает первая строка, а вторая использует функцию abs(), результатом которой является модуль переданного ей числа. Подобная конструкция используется для получения суммы, которую пользователь хочет забрать из копилки.

Из примера видно, что забрать деньги (вне зависимости от того, достаточно их или нет) можно только если они есть в копилке. Этот пример можно упростить, используя рассмотренные ранее правила приведения любых типов к логическому. Так как account — положительное число (благодаря использованию функции abs()), а нулевые значения чисел приводятся к False, мы можем заменить account > 0 на account не изменив при этом поведение программы.