Логические операторы и операторы ветвлений

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

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.

```
Оператор_ветвления Условие:
        Инструкция_1
        Инструкция_2   } Набор инструкций
        Инструкция_3
```

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

Вопросы и ответы, логические выражения

Программист пишет программу, а **программа задает вопросы. Компьютер выполняет программу и предоставляет ответы. Программа должна уметь реагировать в соответствии с полученными ответами.**

Как мы знаем из прошлых занятий компьютер знает **только 2 вида ответов:**

- **да** (True, логическая 1)
- **нет** (False, логический 0)

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.

Вопросы и ответы, логические выражения

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, как вы уже знаете 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.

Все это можно легко проверить при помощи функции print()

Операции сравнения примеры

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

>>>

```
1 == 1: True
1 == 2: False
1 != 1: False
1 != 2: True
1 > 1: False
1 > 2: False
2 > 1: True
1 < 1: False
1 < 2: True
2 < 1: False
1 >= 1: True
1 >= 2: False
2 >= 1: True
1 <= 1: True
1 <= 2: True
2 <= 1: False
```

В примере представлен код, который демонстрирует результаты работы операций сравнения с целыми числами. Сам код программы находится слева, справа отображается результат работы
нашего кода в консоли.

Напомню что если в задании указано вывести в консоль какой-то результат или программа выдает следующий результат, я жду что я его увижу именно в консоли используемой мной IDE

Использование значений в качестве условий

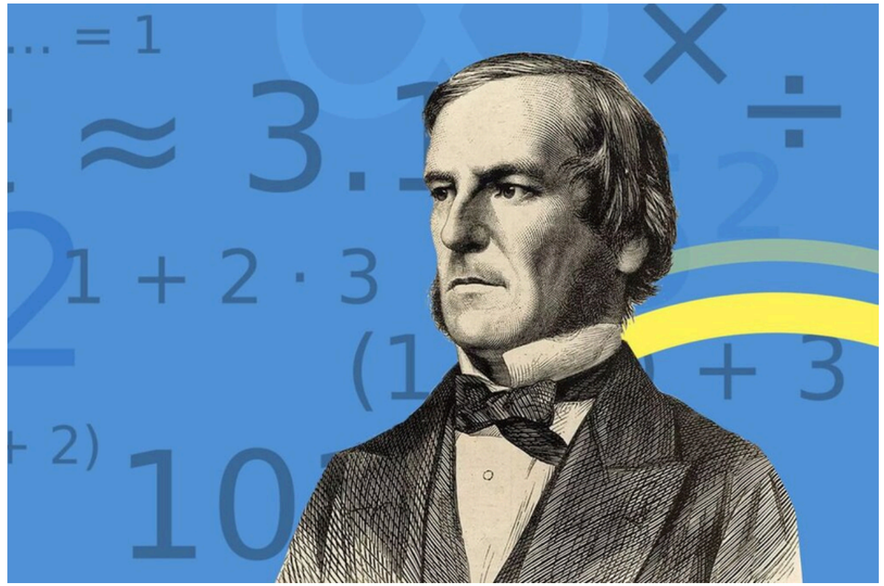
В Python существуют правила, согласно которым можно привести любое значение любого типа к логическому. Если вместо условия написать строку или число, программа заменит его на True или False.

Значения, которые будут заменены на False:

- структуры, не содержащие элементы:
  - строки без символов;
  - пустые списки, словари, массивы и т.д.
- структуры, не содержащие элементы:
  - 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... .



Использование значений в качестве условий пример

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

```
print(bool(""))  # строка                                          >>> False
print(bool([]))  # список                                          >>> False
print(bool(()))  # кортеж                                          >>> False
print(bool({}))  # множество/словарь                              >>> False
print(bool(0))  # целое число                                     >>> False
print(bool(0.0))  # число с плавающей точкой                      >>> False
print(bool(None))  # Ничто                                        >>> False
print(bool("IT Step Academy"))  # строка                          >>> True
print(bool(["item_1", "item_2"]))  # список                       >>> True
print(bool(("item_1", "item_2")))  # кортеж                       >>> True
print(bool({"item_1", "item_2"}))  # множество                    >>> True
print(bool({"key_1": "value_1", "key_2": "value_2"}))  # словарь  >>> True
print(bool(1))  # целое число                                     >>> True
print(bool(0.5))  # число с плавающей точкой                      >>> True
```

В примере представлен код, который демонстрирует результаты работы операций преобразования с различными типами данных.

Сам код программы находится слева, справа отображается результат работы нашего кода в консоли.

Логические операции

Логические операции оператор and

Логические операции позволяют комбинировать несколько логических выражений в одно. В Python имеются следующие логические операторы **and**, or, not:

- **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                    >>>   True
print(competent and responsible)
```

Если же человек компетентен но не ответственен:

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

Действительно, как видно из примеров мы получаем True только если все выражения истинны если хотя бы одно выражение ложно то мы получим False, это не зависит от количества выражений для True они все должны быть истинными. Этот момент необходимо учитывать когда вы пытаетесь составить с ними какое-либо условие, чтобы не задавать заведомо невозможных условий

Логические операции оператор or

Логические операции позволяют комбинировать несколько логических выражений в одно. В Python имеются следующие логические операторы and, **or**, not:

- **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                    >>>   True
print(competent or responsible)
```

Если же человек не компетентен и не ответственен:

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

Когда мы запускаем данный код, мы получаем True только если все если хотя бы одно выражение истинно, если все выражения ложны то мы получим False, это не зависит от количества выражений для True хотя бы одно должно быть истинным. Этот момент необходимо учитывать когда вы пытаетесь составить с ними какое-либо условие, чтобы не задавать заведомо невозможных условий

Логические операции оператор not

Логические операции позволяют комбинировать несколько логических выражений в одно. В Python имеются следующие логические операторы and, or, **not**:

not - «Логическое отрицание»: возвращает значение, обратное операнду. Например, работодатель НЕ возьмет человека, который ранее был уволен:

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

```
>>>  False
```

В противном случае:

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

```
>>>  True
```

*В этом примере, мы видим что если условие о предыдущем выполнении **True** (то есть увольнение было), то выражение с оператором **not** принимает значение **False** и наоборот если условие **False** (увольнения не было) выражение с оператором **not** принимает значение **True.***

Логические операции пример

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("Введите текущее время в часах: ")) % 24
luggage = False
ticket = False
money = True

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

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

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

1й запуск

```
>>> Введите текущее время в часах: 10
True
True
True
```

2й запуск

```
>>> Введите текущее время в часах: 3
True
False
False
```

В коде были реализованы все наши условия и при первом запуске у нас все отлично, а вот при втором запуске у нас есть проблема:
по времени мы не попадаем, но 1й вариант реализации условия выдает нам результат True, данный результат в рамках задачи и поставленных условий не верен.

Это связано с **приоритетами условий логических операторов**:

1. **сначала выполняется not;**
2. **затем and;**
3. **затем or;**

Именно поэтому выполняющийся в последнюю очередь or вернет **True**.
Для переопределения приоритетов можно либо применить скобки (как в математике), либо изменить саму структуру записи, как бы связав при помощи and условия времени и багажа с условием денег или билета а не наоборот. На практике используйте то в чем больше уверены но еще лучше использовать условные конструкции которые мы рассмотрим далее.

Блоки выполнения

Понятие «блока» выполнения

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

У нас есть следующий код и ранее мы говорили про оператор ветвления и наборы инструкций. Однако, каким образом определить, что инструкция принадлежит к этому набору или, другими словами, «блоку выполнения»?

Как определить блок 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. Разберемся на примере:

```
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
```

```
>>>  It is not part of the execution block
```

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 по 5.

Понятие «блока» выполнения, отступы.

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».

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

- **циклах**

```
while condition:
    print("Inside Cycle")

for i in range(5):
    print("Inside Cycle")

for item in some_collection:
    print(f"{item} inside")
```

- **исключениях**

```
try:
    print("Example")
except:
    print("Some Exception")
finally:
    print("Handling Exceptions")
```

- **функциях**

```
def some_function():
    print("Inside Function")
```

- **классах**

```
class ExampleClass:
    def __init__(self, attr):
        self.attr = attr
        print("Inside init")

    def some_method(self):
        print(self.attr)
```

Условия и условные операторы, оператор if

Вы уже знаете, как задавать вопросы Python, но вы все еще не знаете, как разумно использовать ответы. У вас должен быть механизм, который позволит вам что-то делать, если условие выполняется, и не делать этого, если оно не выполняется.

Для принятия таких решений Python предлагает специальную инструкцию, она называется: **условной инструкцией** (или условным выражением).
Существует несколько вариантов условной инструкции, самый простой вариант:

- инструкция **if,** например**:**

```
if true_or_not:
    do_this_if_true
```

Это условное выражение, оно состоит из следующих, строго необходимых, элементов в этом и только в этом порядке:

- ключевое слово **if**;
- один или несколько пробелов;
- **выражение** значение которого будет исключительно **True** или **False**;
- **двоеточие** с последующим переводом строки;
- **инструкция** с отступом или набор инструкций (**минимум одна инструкция**);

Как работает оператор if?

- Если выражение true_or_not принимает значение **True** (т.е. его значение не равно нулю), оператор(ы) с отступом будет(ут) выполнен(ы);
- если выражение true_or_not принимает значение **False** (т.е. его значение равно нулю), оператор(ы) с отступом будет(ут) опущен(ы) (проигнорированы), и следующая выполненная инструкция будет той, которая находится после исходного уровня отступа.

Например:

```
if the_weather_is_good:
    go_for_a_walk()
    ....

have_lunch()
```

Это выражение означает если будет хорошая погода, пойдем гулять;
затем пообедаем. Причем пообедаем мы в любом случае. Это работает и на несколько условий, главное они должны находится внутри данного условия.

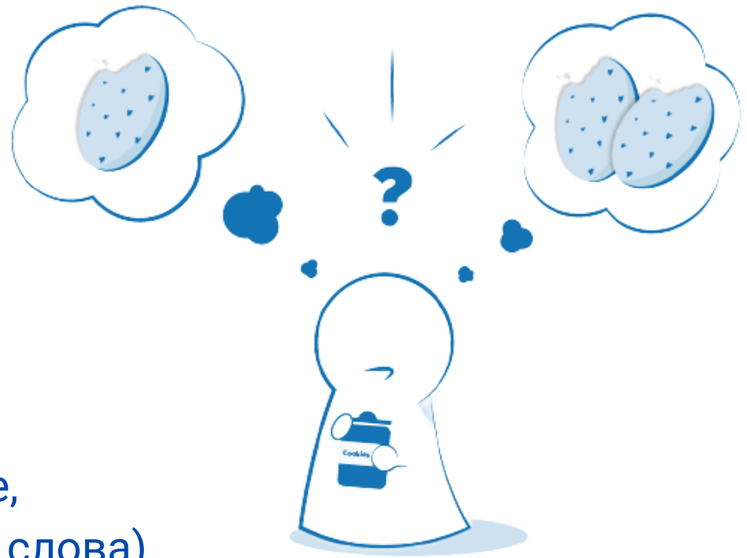
if - else

А что если мы можем рассмотреть 2 возможности, как для хорошей так и для плохой погоды?
Например, идем гулять при хорошей погоде и в кино при плохой?
Тогда наше условное выражение будет выглядеть следующим образом:

```
if the_weather_is_good:
    go_for_a_walk()
else:
    go_to_a_cinema()

have_lunch()
```

Это выражение означает если будет хорошая погода, пойдем гулять;
если нет то пойдем в кино;
пообедаем мы в любом случае.



Как мы видим появилось новое слово: else — это ключевое слово.
Часть кода, которая начинается с **else**, говорит, что делать, если условие,
указанное для **if**, не выполнено (обратите внимание на двоеточие после слова).

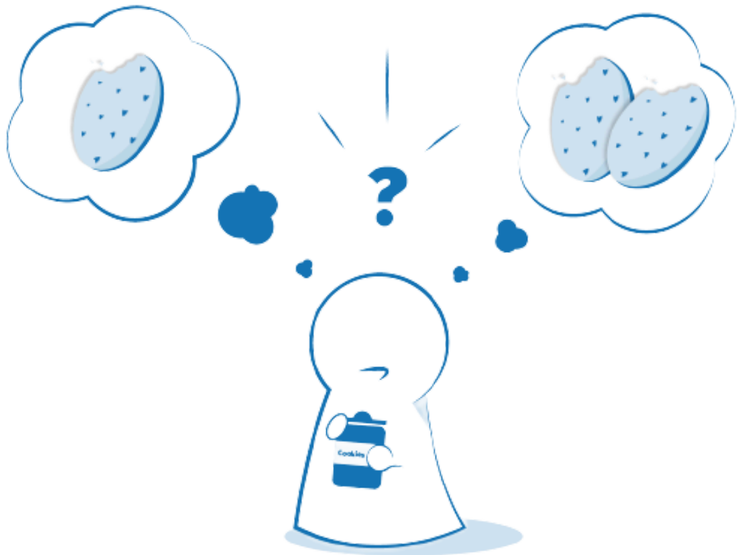
Как работает if - else

Выполнение if-else происходит следующим образом:

- если условие оценивается как **True** (его значение не равно нулю), выполняется оператор **perform_if_condition_true**, и условный оператор прекращает действие;
- если условие оценивается как **False** (оно равно нулю), выполняется оператор **perform_if_condition_false**, и условный оператор прекращает действие.

Оператор if - else: больше условий выполнения

if the_weather_is_sunny:	если будет солнечная погода:
go_for_a_walk()	• пойдем гулять;
have_fun()	• получим удовольствие.
else:	если нет **(во всех прочих случаях):**
go_to_a_cinema()	• пойдем в кино
buy_popcorn()	• купим попкорн
enjoy_the_movie()	• насладимся фильмом
have_lunch()	покушаем в любом случае))



Вложенность if - else

Рассмотрим первый частный случай применения условных операторов.
Инструкция, помещенная после **if**, является другой **if**.

```
if the_weather_is_sunny:
    go_for_a_walk()
    if nice_restaurant_is_found:
        eat_a_steak()
    else:
        eat_a_street_food()
else:
    go_to_a_cinema()
    buy_popcorn()
    enjoy_the_movie()
    have_lunch()
```

если будет солнечная погода:
- пойдем гулять
- если найдем хороший ресторан:
- - съедим стейк
- если не найдем:
- - съедим что найдем

если нет **(во всех прочих погодных случаях):**
- пойдем в кино
- купим попкорн
- насладимся фильмом
- покушаем

- **такое использование оператора if** известно как **вложение;** помните, что **каждое else относится к if, который находится на том же уровне отступа**;
- учтите, как **отступ улучшает читабельность и облегчает понимание и отслеживание кода**.

Вложенность elif (else if)

Второй частный случай применения условных операторов.
Для него водится еще одно новое ключевое слово Python: **elif**. Как понятно из заголовка, это более короткая форма для **else if**.

elif используется для проверки более чем одного условия и остановки, когда найдено первое условие, которое является истинным.

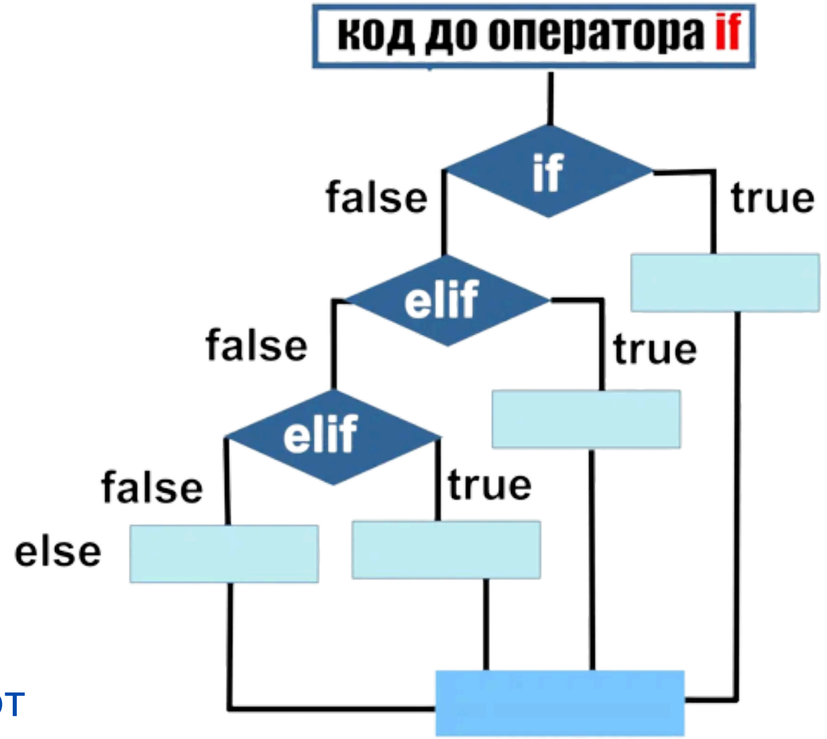
```
if the_weather_is_sunny:
    go_for_a_walk()
elif tickets_are_available:
    go_to_the_cinema()
elif table_is_available:
    go_for_eat_a_steak()
else:
    order_a_pizza()
    play_board_games_at_home()
```

если будет солнечная погода:
- пойдем гулять

если же будут билеты:
- пойдем в кино

если же будут свободные столики:
- пойдем есть стейк

если нет **(ничего из этого):**
- закажем пиццу
- играем в настолки дома

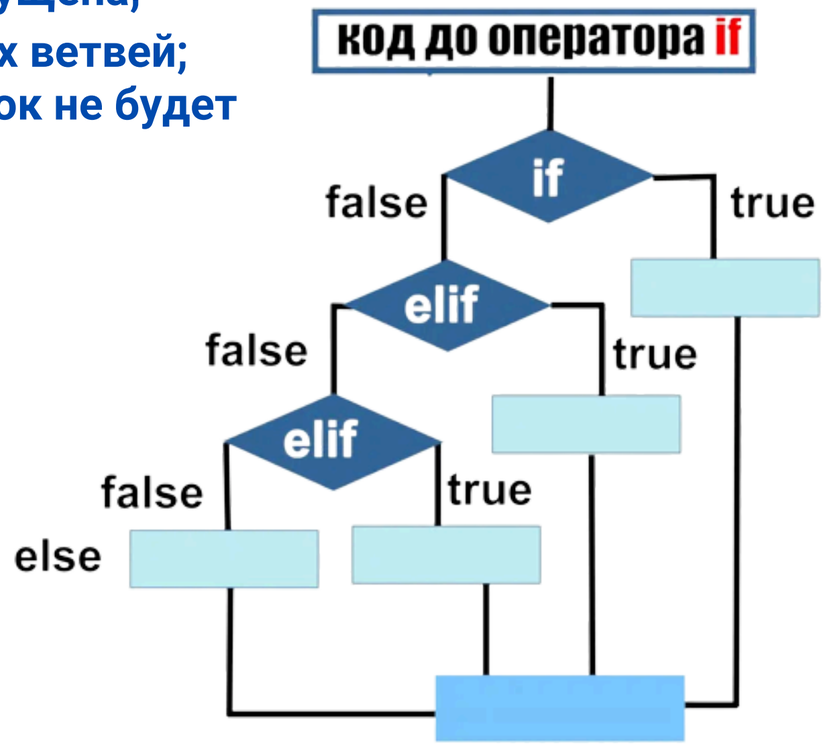


Способ сборки последующих операторов **if-elif-else** иногда называют **каскадом**. Еще раз обратите внимание, что отступ улучшает читаемость кода.

Использование elif (else if)

В случае использования **elif** особое внимание следует уделить **следующему:**

- **вы не должны использовать else без предшествующего if;**
- **else всегда является последней веткой каскада, независимо от того, использовали ли вы elif или нет;**
- **else является необязательной частью каскада и может быть опущена;**
- **если в каскаде есть ветка else, выполняется только одна из всех ветвей;**
- **если нет другой ветки, возможно, что ни одна из доступных веток не будет выполнена.**



Дополнительное разъяснение когда нужно использовать if-if-else, когда if-elif-else.

Ранее был приведены простейший примеры использования условных конструкций рассмотрим еще несколько примеров:

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

```
>>>  Car is faster than 50 km/h
```

при данных параметрах условие выполняется и мы получаем ожидаемый результат

Расширим его на основании полученных знаний. Введем две переменные, обозначающие скорость (в км/ч) машины и мотоцикла соответственно, вместо значений будем использовать переменные:

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

```
>>>  Car is faster than motorcycle
```

при данных параметрах условие выполняется >>> есть результат, но если параметры не будут удовлетворены мы ничего не получим (также и в 1м примере)

Если изменить скорость так, что мотоцикл окажется быстрее, наше условие окажется ложным и пользователь не получит никакой информации. В таком случае удобно будет использовать else — блок, который выполняется только если все условия оказались ложными. 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")
```

```
>>>  Motorcycle is faster than car
```

при данных параметрах если первое условие не выполняется, то есть альтернативный вариант по которому наша программа и будет действовать.

Дополнительное разъяснение когда нужно использовать if-if-else, когда if-elif-else.

Скорости могут уравняться и, хотя блок else из примера 3 выполнится, представленная пользователю информация будет неверна. Исправить это можно, добавив еще одно условие и изменив вариант действий при условии else:

```
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")
```

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

Здесь условные конструкции выполняются поочередно: сначала проверяется условие motorcycle_speed < car_speed, затем motorcycle_speed > car_speed. ***ВАЖНО! 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**. Пример выше 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")
else:
    print("Car and motorcycle are equally fast")
```

```
>>> 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.

Дополнительное разъяснение когда нужно использовать if-if-else, когда if-elif-else.

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:

```
car_speed = 130
motorcycle_speed = 100
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")
else:
    print("Car and motorcycle are equally fast")
```

```
>>>        Car is faster than motorcycle
```

Если программа выдает больше одного результата по запросу, то эти результаты НЕ ДОЛЖНЫ ПРОТИВОРЕЧИТЬ ДРУГ ДРУГУ, также программа может выдавать на один запрос несколько результатов, например задача написать викторину которая бы могла в зависимости от ответов пользователя на ее вопросы, выдать следующий результат:

1. что отвечено верно
2. что неверно
3. сколько заработал очков
4. какой процент правильных ответов

Как видим 1й и 2й результат опираются на 1 базис но не противоречат друг другу, 3й и 4й получаем в зависимости от первого и второго (но вообще есть разные варианты их реализации)

Дополнительное разъяснение вложенность if-else

Как уже было сказано ранее блоки выполнения if, elif и else могут содержать другие условные конструкции, которые называют «вложенными».

```
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")
```

>>>

```
You have at least 3 flowers
You have less than 5 flowers
```

Блок выполнения вложенных конструкций также должен быть отделен отступом, образующим своего рода новую «ступень»: Также касательно результата вспоминаем пример про викторину

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 но пока будем использовать только if-else:

```
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.")
```

>>>

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

Программа получив ответ пользователя как бы спускается по этим вложенным ступенькам, до тех пор пока не найдет подходящее условие, как только программа его нашла, данное условие выполняется и программа далее выходит из этой конструкции и в данном случае завершается, либо выполняется далее в зависимости от того что мы в ней предусмотрели

Дополнительное разъяснение вложенность if-elif-else

Код из предыдущего примера, 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.")
```

```
>>>
Enter the answer number: 3
You've chosen answer C
```

В if - elif - else выполняется код который находится под первым подходящим условием, важно помнить что если вы строите программу в которой множество условий, их надо строить исходя из того какое условие является наиболее маловероятным и ставить его на первое место, т.к. это может перекрыть доступ к правильному варианту. Также в случае если приложение должно завершить работу по команде пользователя, именно это условие должно быть самым первым!!

Дополнительное разъяснение вложенность копилка

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(f"Here are your {withdrawal}.")
        print(f"There are {account} left.")
    else:
        print(f"There are only {account}.")
else:
    print("There are no money in piggy bank")
```

>>>

```
Enter how much you put: 10000
Enter how much you take: -5000
Here are your 5000.
There are 5000 left.
```

В этом примере пользователь в первой строке сам вводит сумму которую он кладет в копилку в ней же происходит преобразование ее в число т.к. все что введено при помощи ***input() является строкой,*** *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.