top

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.

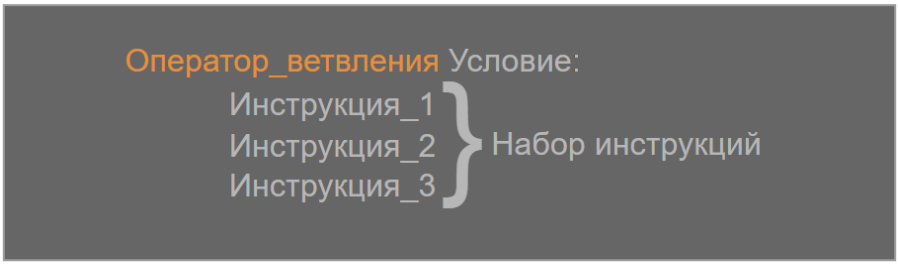

Рисунок 1

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.

Исключения

Исключения — один из двух основных типов ошибок в программировании. В отличие от синтаксических ошибок, которые возникают во время написания, исключения могут появиться во время выполнения программы. Примером такого различия может служить автомобиль: он может быть неисправен, что сделает путешествие на нем невозможным (подобно синтаксическим ошибкам), кроме того, водитель может не справиться с управлением, что приведет к ДТП уже во время поездки (подобно исключениям). В программировании же исключения приводят не к ДТП, а к полному прекращению или к неверному выполнению программ. Для избегания прекращения работы или получения дополнительной информации об ошибке используют конструкции обработки исключений.

Типы исключений

У каждого исключения есть собственный тип, который определяется тем, какая ошибка его вызвала, и дает возможность по-разному реагировать на различные виды ошибок.

Рассмотрим некоторые типы исключений, которые могут возникнуть во время прохождения этого курса:

- BaseException — базовый тип, из которого происходят все остальные, в том числе системные:
  - Exception — базовый тип для «стандартных» и пользовательских исключений:
    - ArithmeticError — арифметическая ошибка:

        - OverflowError — возникает, когда результат арифметической операции слишком велик для представления;

На сегодняшний день практически невозможно получить OverflowError используя стандартную библиотеку Python, т.к. целые числа имеют динамическую длину и скорее вызовут MemoryError, а числа с плавающей запятой при пересечении граничных значений заменяются на 0.0, либо inf. Однако, эта ошибка может быть получена при работе со сторонними библиотеками или модулями, написанными на языках программирования с жесткой типизацией (числовые типы имеют четкие границы, выход за которые — ошибка).

    - ZeroDivisionError — деление на ноль.
- ImportError — импортировать модуль или его атрибут не удалось;
- LookupError — некорректный индекс или ключ:
    - IndexError — индекс не входит в диапазон элементов;
    - KeyError — несуществующий ключ (например, в словаре).
- NameError — не найдено переменной с указанным именем;
- RuntimeError — возникает, когда исключение не попадает ни под одну из других категорий;
- SyntaxError — синтаксическая ошибка:
    - IndentationError — неправильные отступы:
        - TabError — смешивание в отступах табуляции и пробелов.

- TypeError — операция применена к объекту несоответствующего типа;
- ValueError — функция получает аргумент правильного типа, но некорректного значения.

Список предусмотренных самим языком программирования («встроенных») типов исключений гораздо больше и приведен в документации к языку. Кроме того, пользователи сами могут определять новые типы исключений в своих программах и библиотеках.

Перехват исключений

Для обработки исключений в Python используют конструкцию «try … except». В общем случае для построения этой конструкции необходимо:

- открыть блок «try», введя соответствующую инструкцию и двоеточие;
- указать набор инструкций, в результате работы которых может возникнуть исключение;
- выделить набор инструкций отступом;
- открыть блок «except», введя соответствующую инструкцию и двоеточие;
- указать набор инструкций, которые нужно выполнить в случае возникновения исключения;
- выделить набор инструкций отступом.

Применение данной конструкции рассмотрим на следующем примере: на склад доставили 10 ящиков, также на складе есть бочки. Программа учета хранимых на складе предметов просит пользователя указать, сколько и какие именно предметы были доставлены:

```
amount = int(input("Enter the amount of received
                   items: "))
items_type = input("Specify the type of received
                   items: ")
```

Т. к. просьба указать тип полученных элементов отображается только после введения их числа, пользователи, которые не знакомы с программой, могут допустить ошибку:

```
Enter the amount of received items: 10 boxes
```

Часть введенной строки «boxes» нельзя привести к целому числу, вследствие чего будет вызвано исключение «ValueError».

```
Enter the amount of received items: 10 boxes
Traceback (most recent call last):
  File "C:/Users/[redacted]/PycharmProjects/Module 2/main.py", line 331, in <module>
    amount = int(input("Enter the amount of received items: "))
ValueError: invalid literal for int() with base 10: '10 boxes'

Process finished with exit code 1
```

Рисунок 2

Для обработки этого исключения применим конструкцию «try … except»:

```
try:
    amount = int(input("Enter the amount of received
                       items: "))
    items_type = input("Specify the type of received
                       items: ")
except:
    print("Amount should be an integer")
```

Результат работы нашего кода в консоли:

```
Enter the amount of received items: 10 boxes
Amount should be an integer
```

Так, если пользователь введет строку, которую нельзя привести к целому числу, выполнение блока «try» будет прервано (все последующие инструкции блока выполнения будут пропущены), и выполнение перейдет к блоку «except». Допустим, программа учета разделяет полученные предметы на несколько равных частей, чтобы их можно было расположить на разных уровнях склада. Добавим в блок «try» следующие инструкции:

```
parts_number = int(input("Enter the number of parts: "))
parts = amount / parts_number
```

Если пользователь в качестве «parts_number» укажет «0», произойдет ошибка, которая соответствует типу исключения «ZeroDivisionError» (деление на ноль невозможно). Однако сообщение об ошибке, показанное пользователю, останется прежним. Для того чтобы при разных ошибках выполнялись разные инструкции, необходимо создать несколько блоков «except» и указать типы исключений, при которых они будут выполняться:

```
try:
    amount = int(input("Enter the amount of received
                        items: "))
    items_type = input("Specify the type of received
                        items: ")
```

```
    parts_number = int(input("Enter the number of
                                parts: "))
    parts_amount = amount / parts_number
    print("Supply of", amount, items_type, "saved")
    print("Each of", parts_number, "parts consists of",
          parts_amount, items_type)
except ValueError:
    print("Amount should be an integer")
except ZeroDivisionError:
    print("You cannot divide the delivery into 0 parts")
```

Результат работы нашего кода в консоли:

```
Enter the amount of received items: 10
Specify the type of received items: boxes
Enter the number of parts: 0
You cannot divide the delivery into 0 parts
```

Кроме того, конструкцией «try … except» предусмотрен блок «finally», инструкции которого будут выполнены вне зависимости от того, возникнет исключение или нет:

```
try:
    amount = int(input("Enter the amount of received
                        items: "))
    items_type = input("Specify the type of received
                        items: ")
    parts_number = int(input("Enter the number of
                        parts: "))
    parts_amount = amount / parts_number
    print("Supply of", amount, items_type, "saved")
    print("Each of", parts_number, "parts consists of",
          parts_amount, items_type)
except ValueError:
    print("Amount should be an integer")
```

```
except ZeroDivisionError:
    print("You cannot divide the delivery into 0 parts")
finally:
    print("The program has finished")
```

Блок «finally» используется в том случае, когда нам необходимо гарантировать окончание работы программы. Например, мы можем быть подключены к удаленному серверу по сети, работать с файлом или графическим интерфейсом пользователя (GUI). Во всех этих случаях мы должны освободить используемые программой ресурсы до того, как программа завершиться вне зависимости от того, успешно она выполнилась или нет. Такие действия (освобождение ресурсов) выполняются в блоке «finally». Рассмотрим общий случай применения на примере работы с файлом:

```
try:
   f = open("test.txt",'w')
   # perform file operations
finally:
   f.close()
```

Особенности работы с try … except

Вызов исключений

В Python мы можем не только перехватывать исключения, но и напрямую вызывать их. Для этого необходимо использовать ключевое слово «raise». С помощью «raise» мы можем указать, к какому типу будет принадлежать вызванное исключение:

```
try:
    apples = int(input("Enter the amount of apples
                       you have: "))
    if apples < 0:
        raise Exception
    print("You have", apples, "apples")

except Exception:
   print("You can't have -10 apples")
```

Результат работы нашего кода в консоли:

```
Enter the amount of apples you have: -10
You can't have -10 apples
```

Вызов исключений может потребоваться в тех случаях, когда логика программы требует дополнительных условий. Например, в приведенном выше примере для программы не важно, будет введено 10 или -10, т.к. это целые числа, но мы с вами знаем что для обозначения количества предметов используются числа натуральные (1, 2, 3…) и ноль, следовательно, яблок никак не может быть -10.

Порядок размещения блоков exept

Одной из главных особенностей работы с «try … except» является правильное расположение блоков «except». В предыдущем примере с «ValueError» и «ZeroDivisionError» расположение не играло роли, поскольку ни один из них не является подтипом другого. Из таблицы типов исключений, приведенной в соответствующем разделе, видно, что «ZeroDivisionError» является подтипом «ArithmeticError», а «ValueError» — подтипом «Exception». Полную структуру

иерархии встроенных типов исключений можно посмотреть в документации к языку. Рассмотрим влияние расположения блоков на работу программы на примере пары «ValueError» и «Exception»:

```
try:
    raise Exception
except Exception:
    print("Hmm... Something went wrong")
except ValueError:
    print("Improper value was obtained")
```

Результат работы нашего кода в консоли:

```
Hmm... Something went wrong
```

В данном примере внутри блока try возникает исключение Exception, которое обрабатывается соответствующим блоком except. Изменим тип вызываемого исключения:

```
try:
    raise ValueError
except Exception:
    print("Hmm... Something went wrong")
except ValueError:
    print("Improper value was obtained")
```

Результат работы нашего кода в консоли:

```
Hmm... Something went wrong
```

«ValueError» является подтипом «Exception», возникшее исключение будет обработано первым блоком, а второй, предназначенный для обработки исключений такого типа, выполнен не будет. Для того, чтобы исправить это,

необходимо изменить порядок следования блоков except. Общие типы исключений должны быть расположены ниже частных:

```
try:
    raise ValueError
except ValueError:
    print("Improper value was obtained")
except Exception:
    print("Hmm... Something went wrong")
```

Результат работы нашего кода в консоли:

```
Improper value was obtained
```

Комбинирование общего и конкретных блоков exept

Мы так же можем комбинировать использование блоков «exept» для конкретных типов исключений и общий блок «exept». В таком случае общий блок должен быть размещен последним, он выполниться если тип полученного исключения не совпадает ни с одним из типов блоков «exept»:

```
try:
    f = open("Some_file.txt")
except ZeroDivisionError:
    print("You cannot divide the delivery into 0
parts")
except:
    print("Hmm... Something went wrong")
```

Результат работы нашего кода в консоли:

```
Hmm... Something went wrong
```

В этом примере мы пытаемся открыть файл «Some_file.txt», который не был создан ранее, что приводит к исключению типа «FileNotFoundError», однако для этого типа не предусмотрен отдельный блок «exept» и оно обрабатывается общим блоком.

Блок exept для нескольких типов исключений

Python так же позволяет один блок «exept» для разных типов исключений. Для этого их необходимо разместить через запятую внутри круглых скобок. Для рассмотрения этой особенности немного видоизменим один из предыдущих примеров:

```
try:
    amount = int(input("Enter the amount of received
                        items: "))
    items_type = input("Specify the type of received
                        items: ")
    parts_number = int(input("Enter the number of parts:"))
    parts_amount = amount / parts_number
    print("Supply of", amount, items_type, "saved")
    print("Each of", parts_number, "parts consists of",
          parts_amount, items_type)
except (ValueError, ZeroDivisionError):
    print("Improper value was obtained")
```

В силу того, что ошибки являются объектами соответствующих типов (подробнее об определении объекта будет рассказано в следующих модулях), их можно присвоить переменным с помощью ключевого слова «as»:

```
try:
    x = 1/0
```

```
except Exception as ex:
    print(ex)
```

В этом примере объект ошибки присваивается переменной «ex». Результатом выполнения данного кода будет вывод дополнительной информации об ошибке, хранящейся в переменной. Используя ключевое слово «raise», можно вручную указать, какое сообщение будет храниться в объекте исключения:

```
raise ValueError("You made a mistake entering a value")
```

Кроме того, используя переменную, в которой хранится объект исключения, можно получить точную информацию о типе исключения:

```
try:
    raise ValueError
except BaseException as ex:
    print(type(ex).__name__)
```

Приведенный код демонстрирует, что при выполнении инструкций блока «except», на консоль будет выведена строка «ValueError», которая представляет тип возникшего исключения.

Подытожим все полученные знания и совместим их в одном примере:

```
try:
    apples = int(input("Enter the amount of apples
                        you have: "))
```

```
    if apples < 0:
        raise Exception("You can't have -10 apples")
    parts_number = int(input("Enter the number of parts:"))
    parts_amount = apples / parts_number

    print("You have " + str(apples) + " apples \n")
    print("Each of " + str(parts_number) +
          " parts consists of " +
          str(parts_amount) + " apples")

except (ZeroDivisionError, ValueError):
    print("Improper value was obtained")
except Exception as ex:
    print(ex)
except:
    print("Hmm... Something went wrong")

finally:
    print("The program has finished")
```

Циклы

Ежедневно каждый из нас сталкивается с циклами — процессом выполнения однотипных действий. Хорошим примером цикла является тренировка, в которой спортсмену необходимо выполнить определенное движение несколько раз для успешного выполнения упражнения. Даже подъем по лестнице так же является циклом, так как для этого необходимо несколько раз подняться на следующую ступеньку. Цикличность имеет свойство заканчиваться, так достигнув нужного этажа человек не станет подниматься на следующую ступеньку, а спортсмен закончит упражнение, когда выполнит необходимое количество повторений.

Понятие итерации

Итерация — отдельное повторение однотипного действия, под которым понимается блок выполнения цикла («тела цикла»). Возвращаясь, к рассмотренному примеру с лестницей, итерацией будет считаться подъем на новую ступеньку.

Цикл while

В первую очередь рассмотрим цикл «while», синтаксис которого идентичен конструкции «if». Этот цикл повторяет выполнение тела цикла до тех пор, пока соответствующее условие не окажется ложным:

```
number = 1
while number < 3:
    print(number, "is greater than 0 and less than 3")
    number += 1
```

Результат работы нашего кода в консоли:

```
1 is greater than 0 and less than 3
2 is greater than 0 and less than 3
```

Рассмотрим работу цикла по шагам:

- Создание переменной «number» со значением 1;
- Проверка условия цикла «number < 3»: поскольку условие истинно, выполняется тело цикла;
- Выполняется тело цикла: на консоль выводится строка, значение «number» увеличивается на 1;
- Проверка условия цикла «number < 3»: поскольку условие истинно, выполняется тело цикла;
- Выполняется тело цикла: на консоль выводится строка, значение «number» увеличивается на 1;
- Проверка условия цикла «number < 3»: поскольку условие ложно, выполнение цикла и программы закончено.

Другими словами, тело цикла, которое представлено строками 3 и 4, повторится два раза, т.к. после второго повторения переменная «number» будет иметь значение 3 и условие «number < 3» окажется ложным. Обратите внимание, что проверка истинности условия цикла выполняется только после выполнения последней инструкции его тела. Если поменять местами строки 3 и 4, цикл

также выполнится два раза, но представленная пользователю информация окажется ложной, т.к. на момент выполнения «print()» значение «number» уже не будет удовлетворять условию:

```
number = 1
while number < 3:
    number += 1
    print(number, "is greater than 0 and less than 3")
```

Результат работы нашего кода в консоли:

```
2 is greater than 0 and less than 3
3 is greater than 0 and less than 3
```

Попробуем найти сумму всех целых чисел расположенных между «x1» и «x2». Для этого используем цикл «while» следующим образом:

```
x1 = int(input("Enter x1: "))
x2 = int(input("Enter x2: "))
x = x1 + 1
sum = 0
while x < x2:
   sum += x
   x += 1
print("The sum of all integers between",
      x1, "and", x2, "is", sum)
```

Результат работы нашего кода в консоли:

```
Enter x1: -13
Enter x2: 17
The sum of all integers between -13 and 17 is 58
```

В этом примере мы создаем переменную «x», которой присваиваем первое целое число после «x1» («x1 + 1»), и «sum», для хранения промежуточного результата суммирования. Цикл начнется только в том случае, если в промежутке от «x1» до «x2» есть хотя бы одно целое число. В противном случае условие «x < x2» не будет выполнено, цикл будет пропущен, и сумма так и останется равной 0. В каждой итерации цикла мы прибавляем к промежуточному результату(«sum») «x» (одно из чисел в промежутке от «x1» до «x2»), после чего заменяем его на следующее («x + 1»). Когда к «sum» будет прибавлено последнее значение промежутка от «x1» до «x2» его дальнейшее увеличение («x + 1») приведет к тому, что условие «x < x2» окажется ложным и цикл будет закончен. Таким образом в переменной «sum» будут последовательно просуммированы все числа в указанном пользователем промежутке.

Цикл for

Циклы типа «for», в отличие от «while», повторяются не в зависимости от выполнения условия, а для каждого элемента в списке, множества, кортежа или другой совокупности элементов. Другими словами, для каждой итерации цикла используется один из элементов совокупности. К примеру, чтобы проверить пригодность продуктов питания в магазине, человек должен по очереди взять каждый продукт из своей корзины и убедиться, что срок годности еще не истек. Так, корзина с продуктами выступает в качестве совокупности элементов, продукты

являются самими элементами, а проверка срока годности одного продукта — итерацией цикла.

В качестве совокупности элементов может выступать строка, поскольку она является совокупностью символов. Для примера попробуем вывести все элементы строки отдельно:

```
line = input("Enter some string: ")
for c in line:
   print(c)
```

Результат работы нашего кода в консоли:

```
Enter some string: ItStep
I
t
S
t
e
p
```

Цикл может пройти по строке частично. Для этого необходимо её разделить при помощи оператора среза «[::]». «line[start, stop, step]» возвращает все элементы строки «line» от «start» (включая) до «stop» (не включая). Шаг определяет, как на сколько будет увеличиваться индекс. Другими словами, шаг указывает на то, какой символ строки будет принят за следующий. Значение «start» по умолчанию — 0, «step» — 1, «stop» по умолчанию соответствует индексу последнего элемента строки. Детальнее этот оператор будет рассмотрен в следующих модулях, поэтому остановимся на тех его свойствах, что

понадобятся нам для демонстрации работ. Ниже приведен пример кода, который выполняет итерацию по первым шести буквам строки:

```
line = input("Enter some string: ")
for c in line[0 : 6 : 1]:
   print(c)
```

Результат работы нашего кода в консоли:

```
Enter some string: ItStepAcademy
I
t
S
t
e
p
```

Здесь в «[0 : 6 : 1]» 0 обозначает, что срез начинается с начала строки, 6 обозначает, что индекс последнего элемента в срезе не превышает 6 (в нашем случае 6-й символ — первая «s»), а 1 обозначает, что символы в срез будут добавляться слева направо без пропусков. 0 и 1 в примере выше — значения, которые используются оператором по умолчанию, поэтому в дальнейшем значения по умолчанию будут пропущены. Изменяя значение шага, можно вывести все символы, которые находятся на четных позициях:

```
line = input("Enter some string: ")
for c in line[ : : 2]:
   print(c)
```

Результат работы нашего кода в консоли:

```
Enter some string: ItStepAcademy
I
S
e
A
a
e
y
```

Здесь «[: : 2]» означает, что в срезе будет пропущен каждый второй символ начальной строки. Кроме того, если в качестве шага использовать отрицательное значение, перебор будет происходить в обратную сторону:

```
line = input("Enter some string: ")

for c in line[ : : -1]:
    print(c)
```

Результат работы нашего кода в консоли:

```
Enter some string: ItStep
p
e
t
S
t
I
```

Так «[: : -1]» означает, что строка будет инвертирована: её конец станет началом, а начало — концом.

Бесконечные циклы

Определение цикла «while» гласит: «этот цикл повторяет выполнение тела цикла до тех пор, пока соответствующее условие не окажется ложным». Нетрудно представить ситуацию, в которой условие так и не окажется ложным. В таком случае цикл будет выполняться бесконечно:

```
while True:
    print("Hello!")
```

Результат работы нашего кода в консоли:

```
Hello!
Hello!
Hello!
Hello!
...
```

Зачастую бесконечные циклы являются результатом ошибки написания программы (программа составлена таким образом, что предусмотренное условие выхода из цикла никогда не выполнится), например:

```
number = 0
while number != 15:
    number += 2
    print(number, "is still not 15")
```

Результат работы нашего кода в консоли:

```
...
10 is still not 15
12 is still not 15
```

```
14 is still not 15
16 is still not 15
18 is still not 15
...
```

Поскольку «number» изначально равняется 0 и в ходе выполнения программы остается четным, он никогда не станет равен 15 и не сделает условие ложным.

Бесконечные циклы могут применяться для обработки серверами клиентских запросов или, например, для того чтобы продемонстрировать бесконечные величины:

```
import time
number = 1
while True:
    print(number, "is natural number")
    number += 1
    time.sleep(0.1)
```

Результат работы нашего кода в консоли:

```
1 is natural number
2 is natural number
3 is natural number
4 is natural number
5 is natural number
...
```

В данном примере инструкция «import time» необходима для использования «time.sleep()», которая, в свою очередь, останавливает выполнение программы на 0.1 секунду, чтобы пользователь успевал увидеть выводимые на экран числа. Вы можете приблизительно оценить скорость выполнения инструкций программой, убрав эти две инструкции.

Вложенные конструкции

Подобно условным конструкциям, циклы так же могут содержать в себе другие циклы и конструкции. Рассмотрим на примере вывода таблицы умножения:

```
for i in range(1, 10):
    for j in range(1, 10):
        print(i * j, end="\t")
    print("\n")
```

Результат работы нашего кода в консоли:

```
1    2    3    4    5    6    7    8    9

2    4    6    8    10   12   14   16   18

3    6    9    12   15   18   21   24   27

4    8    12   16   20   24   28   32   36

5    10   15   20   25   30   35   40   45

6    12   18   24   30   36   42   48   54

7    14   21   28   35   42   49   56   63

8    16   24   32   40   48   56   64   72

9    18   27   36   45   54   63   72   81
```

Здесь внутренний цикл выполняется 9 раз на каждой итерации внешнего цикла, которых тоже 9. Таблица умножения получается, поскольку в теле внутреннего цикла мы выводим произведение элементов обоих циклов.

Рассмотрим еще один пример: человеку необходимо подняться на 5-й этаж. Сейчас он находится на первом этаже, между каждыми двумя этажами по 20 ступенек. Пройдя 70 ступенек подряд, человек устает и останавливается, чтобы отдохнуть:

```
floor = 1
energy = 70
print("I'm on the", floor, "floor")
while floor != 5:
    step = 0
    while step != 20:
        step += 1
        energy -= 1
        if energy == 0:
            print("I'm tired, I will rest a little")
            energy += 70
    floor += 1
    print("Now I'm on the", floor, "floor")
print("I've got to the right floor")
```

Результат работы нашего кода в консоли:

```
I'm on the 1 floor
Now I'm on the 2 floor
Now I'm on the 3 floor
Now I'm on the 4 floor
I'm tired, I will rest a little
Now I'm on the 5 floor
I've got to the right floor
```

Приведенный код демонстрирует, что каждая итерация цикла «while floor != 5:» выполняет вложенный цикл, а вложенный цикл выполняет конструкцию «if».

Управляющие операторы continue, break и else

Для управления выполнением цикла используются операторы «continue», «break» и «else», которые позволяют пропустить итерацию или выполнить набор инструкций после успешного завершения цикла соответственно. Рассмотрим примеры их использования.

break

Оператор «break» используется для прерывания выполнения цикла.

```
x = 1
while x < 10:
    print(x)
    x += 1
    break
```

Результат работы нашего кода в консоли:

```
1
```

В данном примере на консоль выведено только «1», поскольку в первой итерации выполнился оператор «break», и цикл был прерван.

Если в примере с лестницей поднявшись на 3-й этаж человек воспользуется лифтом, код можно изменить следующим образом:

```
floor = 1
energy = 70
print("I'm on the", floor, "floor")
while floor != 5:
    step = 0
```

```
    if floor == 3:
        print("I will take an elevator")
        break
    while step != 20:
        step += 1
        energy -= 1
        if energy == 0:
            print("I'm tired, I will rest a little")
            energy += 70
    floor += 1
    print("Now i'm on the", floor, "floor")
print("I've got to the right floor")
```

Результат работы нашего кода в консоли:

```
I'm on the 1 floor
Now i'm on the 2 floor
Now i'm on the 3 floor
I will take an elevator
I've got to the right floor
```

Здесь по достижению 3-го этажа используется оператор «break», который прерывает цикл подъема по лестнице. Обратите внимание, что при использовании оператора «break» во вложенном цикле, внешний цикл прерван не будет:

```
number = 0
while number < 5:
    print(number)
    while True:
        number += 1
        break
        number -= 1
```

Результат работы нашего кода в консоли:

```
0
1
2
3
4
```

В приведенном примере так же видно, что при помощи «break» можно прервать бесконечный цикл. Эта особенность может быть полезна для создания циклов с постусловием. Вы уже могли заметить, что цикл «while» выполняет проверку условия перед выполнением тела цикла, поэтому возможна ситуация, когда условие будет изначально ложно и тело цикла не выполнится ни разу. В других языках для избежания подобной ситуации можно использовать цикл «do … while» («цикл с постусловием»), в котором сначала выполняется тело цикла, а потом проверяется условие. В Python такого цикла нет, но его несложно составить самому. Допустим, мы хотим сделать цикл с постусловием из другого, ранее созданного цикла «while»:

```
number = 1
while number < 5:
    print(number)
    number += 1
```

Результат работы нашего кода в консоли:

```
1
2
3
4
```

Для этого заменим условие цикла на «while», что сделает его бесконечным, и поместим оператор «break» в условную конструкцию, использующую условие, противоположное изначальному, в конце тела цикла:

```
number = 1

while True:
    print(number)
    number += 1

    if number >= 5:
        break
```

Результат работы нашего кода в консоли:

```
1
2
3
4
```

Если бы мы использовали изначальное условие завершения цикла «number < 5», оператор «break» прекращал бы выполнение цикла для всех значений «number», которые нам подходят (1, 2, 3, 4, т. е. < 5), и делал бы цикл бесконечным для всех неподходящих значений (>= 5). Поэтому условие необходимо заменить на противоположное. Другими словами, в изначальном цикле условие указывало на необходимость продолжения повторений, а в цикле с постусловием нам нужно, чтобы оно указывало на необходимость их прекращения. Хорошим примером для понимания

подобного поведения является сам язык. Одно и тоже мы можем сказать двумя способами:

- Я буду продолжать радоваться пока мне не исполнится 100 лет.
- Я прекращу радоваться, когда мне исполнится 100 лет.

В нашем случае очевидно, что противоположное условие можно получить, используя другой оператор сравнения («>=»), но для более сложных условий разумнее изменять условие, используя логический оператор «not»:

```
if not (number < 5):
```

На примере полученного цикла видно, что его тело выполнится один раз, даже если условие «number < 5» изначально ложно:

```
number = 5
while True:
    print(number)
    number += 1

    if not (number < 5):
        break
```

Результат работы нашего кода в консоли:

```
5
```

continue

Оператор «continue» используется для прерывания текущей итерации цикла. Это полезно, если необходимо опустить некоторые итерации, и позволяет избежать

вложенности кода. Рассмотрим обе ситуации. Допустим у нас есть цикл, который выводит пять последовательных чисел:

```
number = 0
count = 0
while count < 5:
    number += 1
    print(number)
    count += 1
```

Результат работы нашего кода в консоли:

```
1
2
3
4
5
```

Этот результат можно изменить таким образом, чтобы на экран было выведено 5 последовательных четных чисел, используя «continue» при условии, что остаток от деления числа на два не равен нулю (т.е. число нечетное):

```
number = 0
count = 0
while count < 5:
    number += 1

    if (number % 2) == 1:
        continue
    print(number)
    count += 1
```

Результат работы нашего кода в консоли:

```
2
4
6
8
10
```

Для рассмотрения избегания вложенности кода предположим, что необходимо определить, какие из чисел от 0 до 300 делятся на 3, на 3 и 5 или на 3, 5 и 7 одновременно. Тогда код будет выглядеть следующим образом:

```
number = 0
while number < 300:
    number += 1
    if number % 3 == 0 and number % 5 != 0:
        print(number, "divides by 3")
    elif number % 3 == 0:
        if number % 5 == 0 and number % 7 != 0:
            print(number, "divides by 3 and 5")
        elif number % 5 == 0 and number % 7 == 0:
            print(number, "divides by 3 and 5 and 7")
```

Здесь последняя инструкция находится на 3-м уровне вложенности, т. е. он вложен одновременно в 3 конструкции. Используя «continue», можно разделить эти конструкции, заканчивая итерацию, если условие ложно:

```
number = 0
while number < 300:
    number += 1
    if number % 3 != 0:
        continue
```

```
    elif number % 5 != 0:
        print(number, "divides by 3")
    elif number % 7 != 0:
        print(number, "divides by 3 and 5")
    else:
        print(number, "divides by 3 and 5 and 7")
```

Преимущество такой конструкции в том, что она позволяет одновременно разделить условия, упростив их, и понизить уровень вложенности кода, тем самым упростить его прочтение. Как и оператор «break», «continue» завершает итерацию только для вложенного цикла, в то время как итерация внешнего цикла продолжается.

else

В отличие от условных конструкций, в которых «else» выполняется только если условие ложно, «else» в циклах выполняется в том случае, когда цикл был успешно завершен. Например, ребенок, посчитавший от 1 до 5, может сказать, что он посчитал от 1 до 5:

```
number = 1
while number <= 5:
    print(number)
    number += 1
else:
    print("I've counted from 1 to 5!")
```

Результат работы нашего кода в консоли:

```
1
2
3
```

```
4
5
I’ve counted from 1 to 5!
```

Неочевидно, зачем использовать этот блок если «print()» можно расположить вне тела цикла и результат будет таким же, однако если ребенок забыл несколько цифр, например 4 и 5, он не сможет посчитать дальше и сказать, что посчитал от 1 до 5:

```
number = 1
while number <= 5:
    if number == 4:
        break
    print(number)
    number += 1
else:
    print("I’ve counted from 1 to 5!")
```

Результат работы нашего кода в консоли:

```
1
2
3
```

В таком случае цикл не будет считаться успешно завершенным и блок «else» не будет выполнен. Кроме того, прервать успешное выполнение цикла могут операторы «raise» и «return» (подробнее о «return» будет рассказано в следующих модулях).