Материалы занятия

Курс: Разработка программного обеспечения
Дисциплина: Основы программирования на Python

Тема занятия №5: Управляющие выражения. Блоки, условия, циклы

Введение

Циклы являются такой же важной частью структурного программирования, как условные операторы. С помощью циклов можно организовать повторение выполнения участков кода. Потребность в этом возникает довольно часто. Например, пользователь последовательно вводит числа, и каждое из них требуется добавлять к общей сумме. Или нужно вывести на экран квадраты ряда натуральных чисел и тому подобные задачи.

Цикл while

"While" переводится с английского как "пока". Но не в смысле "до свидания", а в смысле "пока имеем это, делаем то".

Рассмотрим синтаксис:

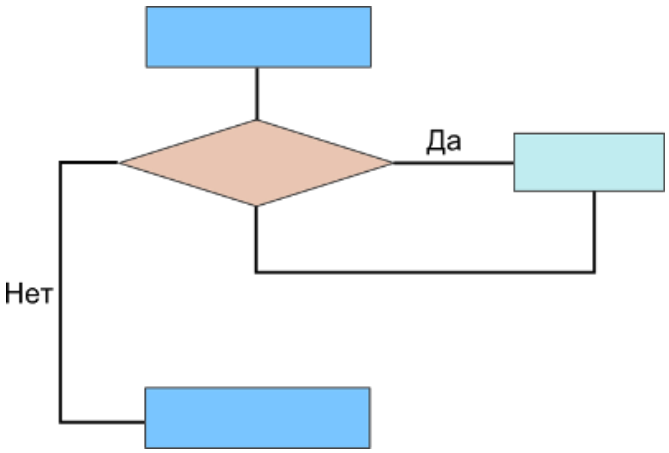
```
while логическое_выражение {
   выражение 1;
   …
   выражение n;
}
```

Это похоже на условный оператор if. Однако в случае циклических операторов их тела могут выполняться далеко не один раз. В случае if, если логическое выражение в заголовке возвращает истину, то тело выполняется единожды. После этого поток выполнения программы возвращается в основную ветку и выполняет следующие выражения, расположенные ниже всей конструкции условного оператора.

В случае while, после того как его тело выполнено, поток возвращается к заголовку цикла и снова проверяет условие. Если логическое выражение возвращает истину, то тело снова выполняется. Потом снова возвращаемся к заголовку и так далее.

Цикл завершает свою работу только тогда, когда логическое выражение в заголовке возвращает ложь, то есть условие выполнения цикла больше не соблюдается. После этого поток выполнения перемещается к выражениям, расположенным ниже всего цикла. Говорят, "происходит выход из цикла".

Рассмотрим блок-схему цикла while.



Ярко-голубыми прямоугольниками обозначена основная ветка программы, ромбом – заголовок цикла с логическим выражением, бирюзовым прямоугольником – тело цикла.

С циклом while возможны две исключительные ситуации:

Если при первом заходе в цикл логическое выражение возвращает False, то тело цикла не выполняется ни разу. Эту ситуацию можно считать нормальной, так как при определенных условиях логика программы может предполагать отсутствие необходимости в выполнении выражений тела цикла.

Если логическое выражение в заголовке while никогда не возвращает False, а всегда остается равным True, то цикл никогда не завершится, если только в его теле нет оператора принудительного выхода из цикла (break) или вызовов функций выхода из программы – quit(), exit() в случае Python. Если цикл повторяется и повторяется бесконечное количество раз, то в программе происходит зацикливание. В это время она зависает и самостоятельно завершиться не может.

Рассмотрим пример из урока про исключения. Пользователь должен ввести целое число. Поскольку функция input() возвращает строку, то программный код должен преобразовать введенное к целочисленному типу с помощью функции int(). Однако, если были введены символы, не являющиеся цифрами, то возникает исключение ValueError, которое обрабатывается веткой except. На этом программа завершается.

Другими словами, если бы программа предполагала дальнейшие действия с числом (например, проверку на четность), а она его не получила, то единственное, что программа могла сделать, это закончить свою работу досрочно.

Но ведь можно просить и просить пользователя корректно вести число, пока он его не введет. Вот как может выглядеть реализующий это код:

```
n = input("Введите целое число: ")

while type(n) != int:
    try:
        n = int(n)
    except ValueError:
        print("Неправильно ввели!")
        n = input("Введите целое число: ")

if n % 2 == 0:
    print("Четное")
else:
    print("Нечетное")
```

Примечание 1. Не забываем, в языке программирования Python в конце заголовков сложных инструкций ставится двоеточие.

Примечание 2. В выражении type(n) != int с помощью функции type() проверяется тип переменной n. Если он не равен int, то есть значение n не является целым числом, а является в данном случае строкой, то выражение возвращает истину. Если же тип n равен int, то данное логическое выражение возвращает ложь.

Примечание 3. Оператор % в языке Python используется для нахождения остатка от деления. Так, если число четное, то оно без остатка делится на 2, то есть остаток будет равен нулю. Если число нечетное, то остаток будет равен единице.

Проследим алгоритм выполнения этого кода. Пользователь вводит данные, они имеют строковый тип и присваиваются переменной n. В заголовке while проверяется тип n. При первом входе в цикл тип n всегда строковый, то есть он не равен int., следовательно, логическое выражение возвращает истину, что позволяет зайти в тело цикла.

Здесь в ветке try совершается попытка преобразования строки к целочисленному типу. Если она была удачной, то ветка except пропускается, и поток выполнения снова возвращается к заголовку while.

Теперь n связана с целым числом, следовательно, ее тип int, который не может быть не равен int. Он ему равен. Таким образом логическое выражение type(n) != int возвращает False, и весь цикл завершает свою работу. Далее поток выполнения переходит к оператору if-else, находящемуся в основной ветке программы. Здесь могло бы находиться что угодно, не обязательно условный оператор.

Вернемся назад. Если в теле try попытка преобразования к числу была неудачной, и было выброшено исключение ValueError, то поток выполнения программы отправляется в ветку except и выполняет находящиеся здесь выражения, последнее из которых просит пользователя снова ввести данные. Переменная n теперь имеет новое значение.

После завершения except снова проверяется логическое выражение в заголовке цикла. Оно даст True, так как значение n по-прежнему строка.

Выход из цикла возможен только тогда, когда значение n будет успешно конвертировано в число.

Рассмотрим следующий пример:

```
total = 100

i = 0
while i < 5:
    n = int(input())
    total = total - n
    i = i + 1

print("Осталось", total)
```

Сколько раз "прокрутится" цикл в этой программе, то есть сколько итераций он сделает? Ответ: 5.

Сначала переменная i равна 0. В заголовке цикла проверяется условие i < 5, и оно истинно. Тело цикла выполняется. В нем меняется значение i, путем добавления к нему единицы.

Теперь переменная i равна 1. Это меньше пяти, и тело цикла выполняется второй раз. В нем i меняется, ее новое значение 2.

Два меньше пяти. Тело цикла выполняется третий раз. Значение i становится равным трем.

Три меньше пяти. На этой итерации i присваивается 4.

Четыре по-прежнему меньше пяти. К i добавляется единица, и теперь ее значение равно пяти.

Далее начинается шестая итерация цикла. Происходит проверка условия i < 5. Но поскольку теперь оно возвращает ложь, то выполнение цикла прерывается, и его тело не выполняется.

"Смысловая нагрузка" данного цикла – это последовательное вычитание из переменной total вводимых чисел. Переменная i в данном случае играет только роль счетчика итераций цикла. В других языках программирования для таких случаев предусмотрен цикл for, который так и называется: "цикл со счетчиком". Его преимущество заключается в том, что в теле цикла не надо изменять переменную-счетчик, ее значение меняется автоматически в заголовке for.

В языке Python тоже есть цикл for. Но это не цикл со счетчиком. В Питоне он предназначен для перебора элементов последовательностей и других сложных объектов. Данный цикл и последовательности будут изучены в последующих уроках.

Для while наличие счетчика не обязательно. Представим, что надо вводить числа, пока переменная total больше нуля. Тогда код будет выглядеть так:

```
total = 100

while total > 0:
    n = int(input())
    total = total - n

print("Ресурс исчерпан")
```

Сколько раз здесь выполнится цикл? Неизвестно, все зависит от вводимых значений. Поэтому у цикла со счетчиком известно количество итераций, а у цикла без счетчика – нет.

Самое главное для цикла while – чтобы в его теле происходили изменения значений переменных, которые проверяются в его заголовке, и чтобы хоть когда-нибудь наступил случай, когда логическое выражение в заголовке возвращает False. Иначе произойдет зацикливание.

Примечание 1. Не обязательно в выражениях total = total - n и i = i + 1 повторять одну и ту же переменную. В Python допустим сокращенный способ записи подобных выражений: total -= n и i += 1.

Примечание 2. При использовании счетчика он не обязательно должен увеличиваться на единицу, а может изменяться в любую сторону на любое значение. Например, если надо вывести числа кратные пяти от 100 до 0, то изменение счетчика будет таким i = i - 5, или i -= 5.

Примечание 3. Для счетчика не обязательно использовать переменную с идентификатором i. Можно назвать переменную-счетчик, как угодно. Однако так принято в программировании, что счетчики обозначают именами i и j (иногда одновременно требуются два счетчика).

Цикл for

Цикл for в языке программирования Python предназначен для перебора элементов структур данных и некоторых других объектов. Это не цикл со счетчиком, каковым является for во многих других языках.

Что значит перебор элементов? Например, у нас есть список, состоящий из ряда элементов. Сначала берем из него первый элемент, затем второй, потом третий и так далее. С каждым элементом мы выполняем одни и те же действия в теле for. Нам не надо извлекать элементы по их индексам и заботиться, на каком из них список заканчивается, и следующая итерация бессмысленна. Цикл for сам переберет и определит конец.

```
>>> spisok = [10, 40, 20, 30]
```

```
>>> for element in spisok:
...     print(element + 2)
...
12
42
22
32
```

После ключевого слова for используется переменная под именем element. Имя здесь может быть любым. Нередко используют i. На каждой итерации цикла for ей будет присвоен очередной элемент из списка spisok. Так при первой прокрутке цикла идентификатор element связан с числом 10, на второй – с числом 40, и так далее. Когда элементы в spisok заканчиваются, цикл for завершает свою работу.

С английского "for" переводится как "для", "in" как "в". Перевести конструкцию с языка программирования на человеческий можно так: для каждого элемента в списке делать следующее (то, что в теле цикла).

В примере мы увеличивали каждый элемент на 2 и выводили его на экран. При этом сам список конечно же не изменялся:

```
>>> spisok
[10, 40, 20, 30]
```

Нигде не шла речь о перезаписи его элементов, они просто извлекались и использовались. Однако бывает необходимо изменить сам список, например, изменить значение каждого элемента в нем или только определенных, удовлетворяющих определенному условию. И тут без переменной, обозначающей индекс элемента, в большинстве случаев не обойтись:

```
>>> i = 0
>>> for element in spisok:
...     spisok[i] = element + 2
...     i += 1
...
>>> spisok
[12, 42, 22, 32]
```

Но если мы вынуждены использовать счетчик, то выгода от использования цикла for не очевидна. Если знать длину списка, то почему бы не воспользоваться while. Длину можно измерить с помощью встроенной в Python функции len().

```
>>> i = 0
>>> while i < len(spisok):
...     spisok[i] = spisok[i] + 2
...     i = i + 1 # или i += 1
...
>>> spisok
[14, 44, 24, 34]
```

Кроме того, с циклом while мы избавились от переменной element.

Функция range()

Теперь пришло время познакомиться со встроенной в Python функцией range(). "Range" переводится как "диапазон". Она может принимать один, два или три аргумента. Их

назначение такое же как у функции randrange() из модуля random. Если задан только один, то генерируются числа от 0 до указанного числа, не включая его. Если заданы два, то числа генерируются от первого до второго, не включая его. Если заданы три, то третье число – это шаг.

Однако, в отличие от randrange(), функция range() генерирует не одно случайное число в указанном диапазоне. Она вообще не генерирует случайные числа. Она генерирует последовательность чисел в указанном диапазоне. Так, range(5, 11) сгенерирует последовательность 5, 6, 7, 8, 9, 10. Однако это будет не структура данных типа "список". Функция range() производит объекты своего класса – диапазоны:

```
>>> a = range(-10, 10)
>>> a
range(-10, 10)
>>> type(a)
<class 'range'>
```

Несмотря на то, что мы не видим последовательности чисел, она есть, и мы можем обращаться к ее элементам:

```
>>> a[0]
-10
>>> a[5]
-5
>>> a[15]
5
>>> a[-1]
9
```

Хотя изменять их нельзя, так как, в отличие от списков, объекты range() относятся к группе неизменяемых:

```
>>> a[10] = 100
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'range' object does not support
item assignment
```

Циклы

Вложенные циклы

Циклы while и for можно как использовать по отдельности, так и комбинировать. Можно вложить цикл for внутрь внешнего цикла while и наоборот, а также вкладывать циклы одного вида друг в друга.

Вложенные циклы работают по следующей схеме: программа сначала сталкивается с внешним циклом и начинает выполнять его условия. Затем запускается внутренний вложенный цикл, который выполняется до своего завершения. Программа будет завершать внутренний цикл и возвращаться к началу внешнего до тех пор, пока последовательность не будет завершена или другой оператор не нарушит этот процесс.

Вложенный цикл for

Рассмотрим на примере вложенного цикла for, как это работает на практике:

```
num_list = [1, 2, 3]
alpha_list = ['a', 'b', 'c']
for number in num_list:
print(number)
for letter in alpha_list:
print(letter)
```

Результат:

```
1
a
b
c
2
a
b
c
3
a
b
c
```

По результату выполнения кода видно, что программа завершает первую итерацию внешнего цикла на цифре 1 и затем запускает завершение внутреннего цикла, печатая a, b, c. Как только внутренний цикл завершен, программа возвращается к началу внешнего цикла и печатает цифру 2, а после снова воспроизводит вложенный цикл.

Вложенный цикл while

Вложенный цикл **while** выглядит так:

```
i=1
while i<=3 :
print(i,"Outer loop is executed only once")
j=1
while j<=3:
print(j,"Inner loop is executed until to completion")
j+=1
i+=1;
```

Результат:

```
1 Outer loop is executed only once
1 Inner loop is executed until to completion
2 Inner loop is executed until to completion
3 Inner loop is executed until to completion
2 Outer loop is executed only once
1 Inner loop is executed until to completion
2 Inner loop is executed until to completion
3 Inner loop is executed until to completion
3 Outer loop is executed only once
1 Inner loop is executed until to completion
2 Inner loop is executed until to completion
```

3 Inner loop is executed until to completion

Инструкции if в цикле for

Внутри цикла for также можно использовать инструкции if.

В качестве примера можно привести известное упражнение, которое предлагают junior-специалистам в сфере data science:

Переберите числа до 99. Выводите «fizz» для каждого числа, которое делится на 3, «buzz» — для тех, что делятся на 5 и «fizzbuzz» — для тех, что делятся на 3 и на 5! Если число не делится, выводите тире (’-’)

Вот решение:

```
for i in range(100):
    if i % 3 == 0 and i % 5 == 0:
        print('fizzbuzz')
    elif i % 3 == 0:
        print('fizz')
    elif i % 5 == 0:
        print('buzz')
    else:
        print('-')
```

Функция enumerate

Функция enumerate() применяется для так называемых итерируемых объектов (список относится к таковым) и создает объект-генератор, который генерирует кортежи, состоящие из двух элементов – индекса элемента и самого элемента.

```
>>> spisok = [16, 46, 26, 36]
>>> for i in enumerate(spisok):
...     print(i)
...
(0, 16)
(1, 46)
(2, 26)
(3, 36)
```

```
>>> b = "hello"
>>> for i in enumerate(b):
...     print(i)
...
(0, 'h')
(1, 'e')
(2, 'l')
(3, 'l')
(4, 'o')
```

Функция enumerate() используется для упрощения прохода по коллекциям в цикле, когда кроме самих элементов требуется их индекс:

```
>>> a = [10, 20, 30, 40]
>>> for id, item in enumerate(a):
...     a[id] = item + 5
...
```

```
>>> a
[15, 25, 35, 45]
```

В данном случае на каждой итерации цикла из объекта, полученного от вызова функции enumerate, извлекается очередной кортеж. Этот кортеж состоит из индекса очередного элемента списка и значения этого элемента. Элементы кортежа связываются с идентификаторами id и item.

Без использования enumerate в цикл пришлось бы вводить переменную-счетчик:

```
>>> a = [10, 20, 30, 40]
>>> id = 0 # используется счетчик
>>> for num in a:
...     a[id] = num + 5
...     id += 1
...
>>> a
[15, 25, 35, 45]
```

Или извлекать элементы по индексу:

```
>>> a = [10, 20, 30, 40]
>>> for i in range(len(a)): # перебор по индексам
...     a[i] += 5
...
>>> a
[15, 25, 35, 45]
```