Тема 3. ОПЕРАТОРЫ ЯЗЫКА

Оглавление

Тема 3. ОПЕРАТОРЫ ЯЗЫКА

3.1 Ветвление и условные операторы

Использование условных операторов позволяет реализовать алгоритмическую конструкцию ветвления в ходе выполнения программы. Благодаря условиям некоторые инструкции могут быть опущены, в то время как другие – выполнены.

3.1.1 Конструкция if-else

Базовый вариант условного оператора предполагает выбор одного из двух наборов инструкций:

```
if выражение:
    инструкция_1
    инструкция_2
    ...
    инструкция_n
else:
    инструкция_1
    инструкция_2
    ...
    инструкция_n
```

Оператор *if* используется для проверки условий: если оно верно, выполняется *if*-блок выражений, иначе выполняется *else*-блок выражений. Следует отметить, *else*-блок не обязателен. Условный оператор в коде программы:

Сокращенная форма

```
flag = 0
x = int(input())
if x%2 == 0:
   flag = 1
print("flag = ", flag)
```

Данные и результат:

```
34
flag =  1
```

Полная форма

```
x = int(input())
if x%2 == 0:
   print("Число четное")
else:
   print("Число нечетное")
```

Данные и результат:

```
 45
Число нечетное
```

Инструкции, приведенные для сокращенной формы, демонстрируют изменение значения переменной *flag* в зависимости от значения переменной *x*. Если в переменной хранится четное число, значение *flag* изменяется и становится равным 1, для нечетно-

го числа условие $x\%2 == 0$ генерирует *False* и *flag* сохраняет исходное значение.

В примере для условного оператора в полной форме выполняется проверка вводимого числа на четность/нечетность.

3.1.2 Вложенные условия

Внутри условного оператора можно использовать любые другие операторы. Если в качестве исполняемой инструкции используется другой условный оператор, возникает ситуация вложенного ветвления:

```
if условие_1:
    блок кода
else:
    if условие_2:
        блок кода
    else:
        if условие_3:
            блок кода
...
```

Пример 1. Написать код для нахождения наибольшего из трех чисел $d = \max(a, b, c)$. Фрагмент блок-схемы алгоритма для решения этой задачи приведена на рис. 3.1.

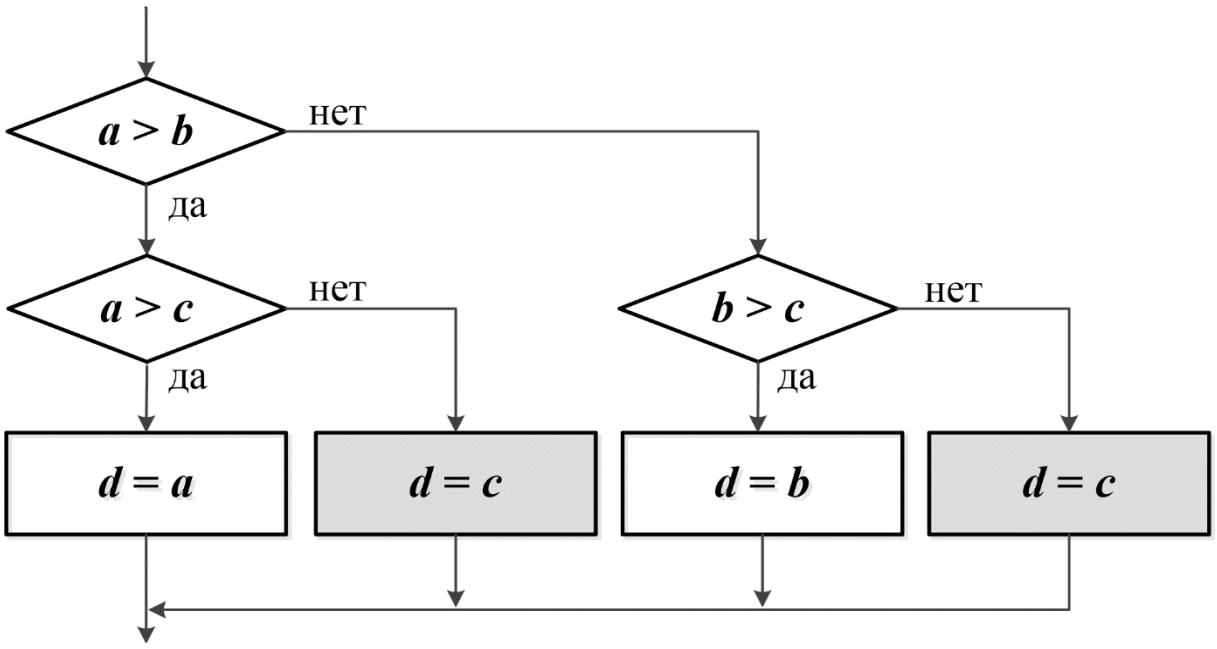

Рис. 3.1. Алгоритм выбора максимального из трех чисел

Соответствующий этой блок-схеме код:

```
a = int(input('Введите a: '))
b = int(input('Введите b: '))
c = int(input('Введите c: '))
if a > b:
    if a > c:
        d = a
    else:
        d = c
else:
    if b>c:
        d = b
    else:
        d = c
print('max =',d)
```

Усовершенствуем алгоритм, убрав повтор оператора *d = c* (рис. 3.2).

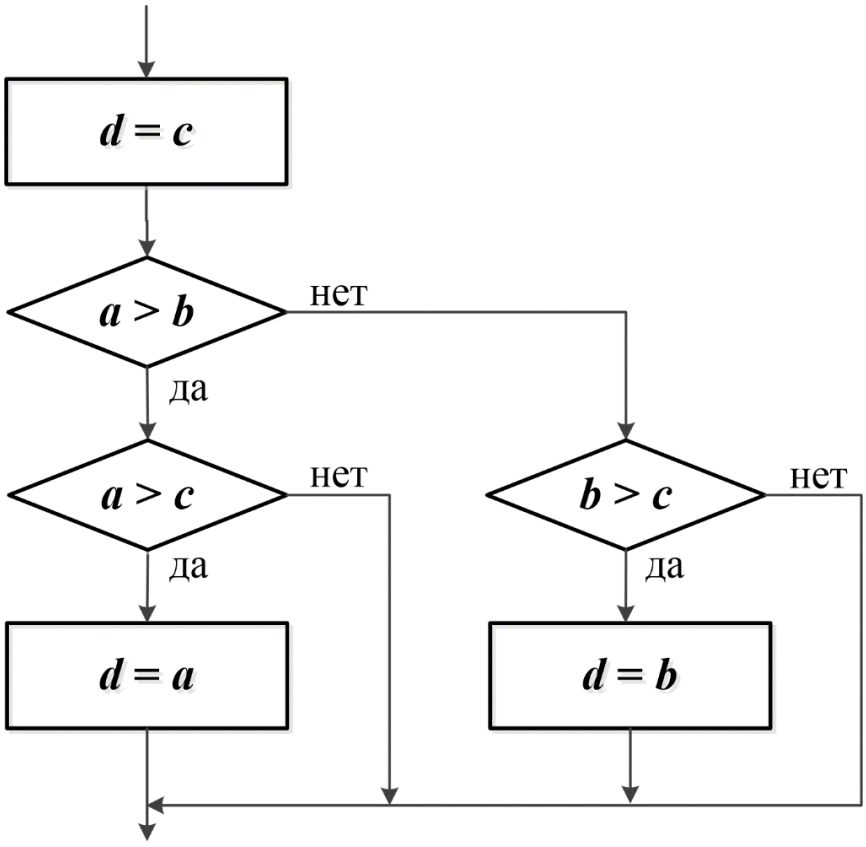

Рис. 3.2. Оптимизированный алгоритм выбора максимального из трех чисел

Код, соответствующий оптимизированному алгоритму выбора максимального из трех чисел:

```
a = int(input('Введите a: '))
```

```
b = int(input('Введите b: '))
c = int(input('Введите c: '))
d = c
if a > b:
    if a > c:
        d = a
else:
    if b>c:
        d = b
print('max =',d)
```

В обоих вариантах решения задачи уровень вложенности равен двум, поэтому код достаточно легко читается до и после оптимизации. Рассмотрим иллюстрацию более глубокого уровня вложенности.

Пример 2. Написать код для перевода стобалльной оценки в пятибалльную.

```
grade = int(input('Введите Ваш балл: '))
if  grade >= 75:
    print('Отлично')
else:
    if grade >= 60:
        print('Хорошо')
    else:
        if grade >= 35:
            print('Удовлетворительно')
        else:
            if grade >= 25:
                print('Неудовлетворительно')
            else:
                print('Неявка')
```

В этом примере вложенные условия значительно усложняют понимание кода. Написать пять независимых инструкций *if* в сокращенной форме нельзя, так как будет напечатано сразу несколько значений пятибалльной оценки

В случае необходимости проверки нескольких условий на разных уровнях вложенности следует использовать ***оператор множественного выбора***.

3.1.3 Множественный выбор

Оператор множественного или каскадного выбора *if-elif-else* позволяет организовать более двух ветвей выполнения программы без вложенных условных операторов.

Синтаксис каскадного условного оператора имеет следующий вид:

```
if условие_1:
    блок кода
elif условие_2:
    блок кода
...
else:
    блок кода
```

Заключительный блок *else* в операторе является необязательным.

При исполнении каскадного условного оператора сначала проверяется `условие_1`. Если оно является истинным, то исполняется блок кода, который следует сразу за ним, до выражения *elif*. Остальная часть конструкции игнорируется. Однако если `условие_1` является ложным, то программа перескакивает к следующему выражению *elif* и проверяет `условие_2`. Если оно истинное, то исполняется соответствующий ему блок кода до следующего выражения *elif*, и остальные инструкции условного оператора игнорируются. Этот процесс продолжается до тех пор, пока не будет найдено условие, которое является истинным, либо пока больше не останется выражений *elif*. Если ни одно условие не является истинным, то исполняется блок кода после выражения *else*.

Пример 1. Написать код для перевода стобалльной оценки в пятибалльную с использованием каскадного условного оператора:

```
grade = int(input('Введите Ваш балл: '))
if  grade >= 75:
    print('Отлично')
elif grade >= 60:
    print('Хорошо')
elif grade >= 35:
    print('Удовлетворительно')
```

```
elif grade >= 25:
    print('Неудовлетворительно')
else:
    print('Неявка')
```

Приведенный код работает аналогично решению, предложенному в пункте 2.6.2. Однако логика инструкции *if-elif-else* прослеживается легче, чем длинная серия вложенных инструкций *if-else*. В том числе, за счет выраженной вертикали выравнивания выражений *if, elif* и *else* и выделения исполняемых по условию блоков отступом.

Начиная с версии 3.10, в *Python* появился оператор ***match***, который можно использовать как аналог оператора ***switch***, присутствующего в других языках.

В *match* множественное ветвление организуется с помощью веток *case*:

```
match element:
    case значение_1:
         действия
    case значение_2:
        действия
    case _:
        действия
```

Оператор *match* принимает выражение *element* и сравнивает его значение с последовательными шаблонами, заданными как один или несколько блоков *case*. В отличие от *if-elif-else* в этом операторе нельзя использовать логические выражения. После `case` должен находиться литерал, конкретное значение или выражение, возвращающее однозначный результат. Ветвь `case _` несет такой же смысл, как и *else* в операторе каскадного выбора, и является необязательной.

Пример 2. Написать код, вычисляющий результат одной из арифметических операций. Арифметический оператор (+, -, *, /) и операнды вводятся пользователем с клавиатуры. Фрагмент блок-схемы алгоритма для решения этой задачи приведена на рис. 3.3.

Соответствующий этой блок-схеме код:

```
sign = input('Знак оператора: ')
```

```
a = int(input('Число 1: '))
b = int(input('Число 2: '))
match sign:
    case '+':
        print(a + b)
    case '-':
        print(a - b)
    case '*':
        print(a * b)
    case '/':
        if b != 0:
            print(a / b)
        else:
            print('Ошибка! Деление на ноль ')
    case _:
        print('Неверный знак ')
```

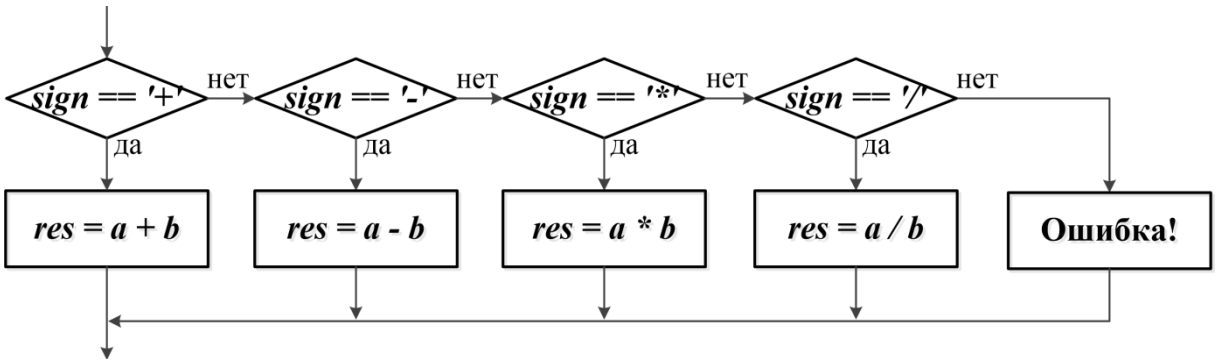

Рис. 3.3. Алгоритм выбора одного из нескольких вариантов

В ветвях *case* оператора *match* не требуется указывать оператор *break* для выхода за пределы оператора по завершению инструкций выбранного *case*-блока. При необходимости, в операторе *match* может использоваться оператор-заглушка *pass*.

Можно определить блок *case*, в котором проверяется сразу несколько значений. В этом случае они разделяются вертикальной чертой:

```
language = input('Язык: ')
match language:
    case 'russian':
        print('Привет')
    case 'français'|'french':
        print('Bonjour')
    case _:
        print('Undefined')
```

В данном случае шаблон *case 'français' | 'french'* соответствует сразу двум значениям.

3.1.4 Тернарный условный оператор

Во многих языках программирования есть ***тернарный условный оператор***, который позволяет компактно записывать условные выражения.

В *Python* подобный оператор тоже существует, но его синтаксис отличается от большинства других языков программирования:

```
<значение 1> if <условие> else <значение 2>
```

Он возвращает значение 1, если условие истинно, а иначе – значение 2.

Пример 1. Выполнить проверку вводимого числа на четность/нечетность:

```
x = int(input('x: '))
res = 'четное' if x%2 == 0 else 'нечетное'
print('Число', res)
```

Тернарный оператор автоматически возвращает результат. В данном примере – это слово 'четное' или 'нечетное'.

Следует отметить, в данном операторе нет внутренних блоков, в которых можно записывать несколько операторов. Вместо <значение 1> и <значение 2> можно прописывать только одну какую-либо конструкцию, в том числе, еще один тернарный оператор.

Пример 2. Написать код для нахождения наибольшего из трех чисел $d = \max(a, b, c)$:

```
a = int(input('Введите a: '))
b = int(input('Введите b: '))
c = int(input('Введите c: '))
d = (a if a > c else c) if a > b else (b if b > c else c)
print(d)
```

В приведенном коде круглые скобки использованы для упрощения восприятия инструкции и выделения первого условия в цепочке проверок.

Следует отметить, с помощью тернарного условного оператора можно записать только полное условие с *if*- и *case*-блоками, поэтому приведенное выше решение соответствует неоптимизированному варианту алгоритма из п. 3.2.2.

3.2 Циклы

3.2.1 Оператор цикла while

Цикл ***while*** проверяет истинность некоторого условия, и если условие истинно, то выполняет инструкции цикла. Он имеет следующее формальное определение:

```
while условное_выражение:
    инструкция_1
    инструкция_2
    ...
    инструкция_n
```

После ключевого слова *while* указывается условное выражение, и пока это выражение (заголовок цикла) возвращает значение *True*, будет выполняться идущий далее блок инструкций, называемый телом цикла.

Однократное выполнение блока инструкций цикла называется итерацией.

Пример. Вывести квадраты целых чисел, кратных 3, в диапазоне от 1 до *n*, где *n* – задается пользователем с клавиатуры:

```
n = int(input('n: '))
i = 1
while i <= n:
    if i % 3 == 0:
        print(i ** 2)
    i += 1
```

В приведенном фрагменте кода инструкция *i* += 1 отвечает за переход к следующему числу в анализируемой последовательности.

Цикл *while* может быть дополнен необязательным оператором *else*, в котором указываются инструкции, выполняющиеся после штатного завершения цикла (условие в заголовке цикла возвращает *False*):

```
while условное_выражение:
    инструкция_1
        ...
    инструкция_n
else:
    инструкция_1
    ...
    инструкция_n
```

Код примера с дополнительным оператором *else*:

```
n = int(input('n: '))
i = 1
while i <= n:
    if i % 3 == 0:
        print(i ** 2)
    i += 1
else:
    print ('Цикл завершен!')
```

Блок *else* полезен в том случае, если условие изначально равно *False*, и данный факт необходимо учитывать в основном вычислительном процессе.

3.2.2 Оператор цикла for

Цикл ***for*** в языке программирования *Python* предназначен для перебора элементов структур данных и других составных объектов. Это не цикл со счетчиком, каковым является *for* во многих других языках, принцип его работы аналогичен оператору *foreach* из других языков программирования.

Формальное определение цикла:

```
for переменная in набор_значений:
    инструкция_1
    ...
    инструкция_n
```

Как правило, циклы *for* используются либо для повторения какой-либо последовательности действий заданное число раз, либо для изменения значения переменной в цикле от некоторого начального значения до некоторого конечного.

В табл. 3.1 представлены примеры совместного использования цикла *for* и функции *range*().

Таблица 3.1 – Перегрузки функции *range*()

Определение функции	Пример использования в коде	Описание

range(*n*)	`sum = 0`
	`n = 4`
	`for i in range(n):`
	`    sum += i`
	`print(sum)`
	Результат: 6
		Переменная *i* принимает значения от 0 до *n*-1 (0, 1, 2, 3). Значение переменной *sum* последовательно увеличивается на указанные значения

range(*n*, *m*)	`sum = 0`
	`m = 5`
	`for i in range(1, m):`
	`    sum += i`
	`print(sum)`
	Результат: 10
		Переменная *i* принимает значения от 1 до *m*-1 (1, 2, 3, 4). Значение переменной *sum* последовательно увеличивается на указанные значения

range(*n*, *m*, *k*)	`sum = 0`
	`m = 10`
	`for i in range(1, m, 2):`
	`    sum += i`
	`print(sum)`
	Результат: 25
		Переменная *i* принимает значения от 1 до *m*-1 с шагом 2 (1, 3, 5, 7, 9). Значение переменной *sum* последовательно увеличивается на указанные значения

Пример 1. В диапазоне чисел от *a* до *b*, включительно, найти сумму четных чисел, кратных 3. Значения *a* и *b* задаются пользователем с клавиатуры:

```
n = int(input('n = '))
m = int(input('m = '))
sum_numbers = 0
for i in range(n, m + 1):
    if i % 2 == 0 and i % 3 == 0:
        sum_numbers += i
print(sum_numbers)
```

Пример 2. Проверить является ли введенное с клавиатуры число простым:

```
number = int(input('Введите целое число: '))
k = 0
for i in range(2, number // 2+1):
```

```
    if (number % i == 0):
        k = k+1
if k <= 0:
    print('Число простое')
else:
    print('Число составное')
```

В цикле подсчитывается количество делителей числа, отличных от единицы и значения самого числа, затем на основе полученной информации определяется его вид.

Параметрами функции *range()* могут быть арифметические операторы. Если первый параметр функции больше второго, будет сгенерирована пустая последовательность.

3.2.3 Операторы break, continue и pass

Оператор ***break*** используется для управления циклом и позволяет досрочно прервать его выполнение. В случае вложенных конструкций повтора данный оператор вызовет завершение только внутреннего цикла, и инструкции внешнего цикла продолжат свое выполнение. Оператор *break* может использоваться в циклах *for* и *while*, однако при попытке разместить его вне цикла, возникнет ошибка.

Пример 1. Проверить является ли введенное с клавиатуры число простым или нет:

```
number = int(input('Введите целое число: '))
flag = True
for i in range(2, number):
    if number % i == 0:
        flag = False
        break
if flag:
    print('Число простое')
else:
    print('Число составное')
```

В представленном решении переменная *flag* играет роль сигнальной метки. Как только будет найден делитель, отличный от единицы и *number*, значение *flag* меняется на противоположное и

цикл прерывается, поскольку дальнейшее его выполнение лишено смысла, так как число гарантированно не является простым.

Пример 2. Вывести на экран заданную строку посимвольно. При наличии в строке ',' завершить работу цикла досрочно:

```
string = 'Числа, строки, списки'
for i in string:
    if i == ',':
        break
    print(i)
else:
    print('Цикл завершился без break')
```

В цикле перебираются символы, образующие строку. В случае нахождения стоп-символа вычислительный процесс завершается. Инструкция *else* относится к циклу *for*, а не к оператору *if*.

Оператор ***continue*** позволяет пропускать одну итерацию тела цикла, отклоняя все оставшиеся после него инструкции и перемещая управления в начало конструкции повтора. Данный оператор, как и *break*, можно использовать в циклах *for* и *while*.

Если в решении примера 2 заменить *break* на *continue*, то на экран будут выведены все символы, образующие строку, за исключением стоп-символов. Ветвь *else* также будет выполнена, поскольку цикл завершится штатным образом.

Пример 3. Для вводимых с клавиатуры целых чисел найти сумму нечетных значений. Подсчет суммы прекращается при вводе нуля:

```
res = 0
number = 1
while number != 0:
    number = int(input('Введите целое число: '))
    if number % 2 == 0:
        continue
    res += number
print('Сумма нечетных чисел = ' + str(res))
```

На каждой итерации цикла осуществляется проверка вводимого числа, если оно четное, оператор *continue* передает управление в начало числа, пропуская вычисление суммы. Цикл продолжает свою работу и переходит к следующей итерации, пока пользователь не введет ноль. После этого выводится итоговая сумма, полученная к этому моменту.

Поскольку синтаксис языка *Python* построен на отступах, и в нем не предусмотрены какие-либо операторы, обозначающие блоки кода, был введен оператор ***pass***, сопоставимый с пустыми фигурными скобками в языках программирования *C* или *JavaScript*.

Оператор *pass* используется в том случае, когда в коде требуется какой-то синтаксис, но никаких действий производить не надо. Например, данный оператор можно использовать в качестве заполнителя для функции или условного блока при написании нового кода и построении общей концепции с сохранением высокого уровня абстракции.

Иногда *pass* называют оператором-заглушкой:

```
number = 0
for number in range(10):
    if number == 5:
        pass
    print(str(number))
else:
    print('Цикл завершен корректно')
```

Оператор *pass*, размещенный после условного оператора *if*, указывает программе на продолжение выполнения цикла, игнорируя равенство переменной *number* пяти во время одной из итераций.

3.2.4 Вложенные циклы

Часто в программировании решаются задачи с использованием вложенных циклов.

Внешний цикл на первой итерации вызывает внутренний, который исполняется до своего завершения, после чего управление передается в тело внешнего цикла. На втором проходе внешний

цикл опять вызывает внутренний. И так до тех пор, пока не завершится внешний цикл.

Глубина вложения циклов (то есть количество вложенных друг в друга циклов) может быть различной. Общее количество итераций равно произведению итераций всех циклов. Для экономии машинного времени во внутреннем цикле необходимо оставить только те инструкции, которые зависят от его параметра.

На рис. 3.4 представлен фрагмент блок-схемы соответствующей алгоритмической конструкции.

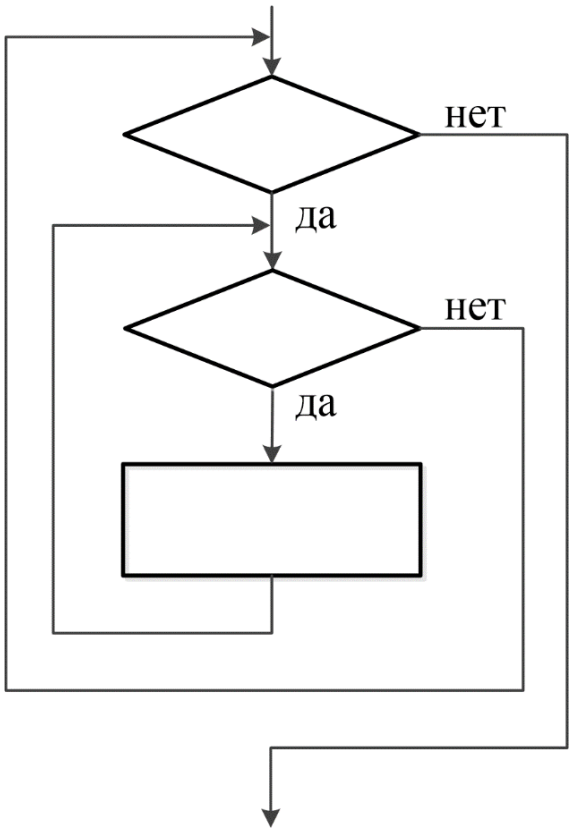

Рис. 3.4. Вложенные циклы

Пример 1. Вывести на консоль таблицу умножения:

```
i = 1
j = 1
while i < 10:
    while j < 10:
        print(i * j, end="\t")
        j += 1
    print("\n")
    j = 1
```

```
    i += 1
```

Внешний цикл срабатывает 9 раз, пока переменная *i* не станет равна 10. Для каждой его итерации выполняется 9 итерации внутреннего цикла, результатом которых является ввод произведения чисел *i* и *j*. Затем значение переменной *j* увеличивается на единицу. По завершению внутреннего цикла происходит переопределение значений счетчиков – *j* устанавливается в единицу, *i* увеличивается на единицу – и переход к следующей итерации внешнего цикла.

Пример 2. Написать программу, которая печатает треугольник высотой *n*:

1
2 3
4 5 6
7 8 9 10
…

Решение:

```
number = int(input('Введите целое число: '))
count = 1
for i in range(1, number + 1):
    for x in range(i):
        print(count, end=' ')
        count += 1 # увеличиваем счетчик
    print()
```

Переменная *count* предназначена для определения чисел, используемых для построения треугольника. Внешний цикл создает диапазон от 1 до введенного числа, включительно. Внутренний цикл отвечает за «построчное» построение треугольника, начиная с вершины. Количество элементов в строке зависит от ее номера, а сама она формируется объединением *count* с *end*.

3.2.5 Бесконечные циклы

Иногда в программах используются циклы, выход из которых не предусмотрен логикой программы. Такие циклы называются безусловными, или бесконечными. Специальных синтаксических средств создания бесконечных циклов языки программирования

не предусматривают, поэтому они создаются с помощью конструкций, предназначенных для создания обычных циклов. Для обеспечения бесконечного повторения проверка условия в таком цикле либо отсутствует (если позволяет синтаксис), либо заменяется константным значением.

Самый простой способ создать бесконечный цикл в *Python* – записать следующий код:

```
while True:
    print('Привет!')
```

Результатом выполнения такого кода будет бесконечное количество строчек 'Привет!'.

Бесконечный цикл выполняет итерации, пока программа не будет прервана. Оператор *break* обеспечивает механизм выхода из бесконечного цикла.

В некоторых языках программирования есть оператор ***цикла с постусловием***, который всегда выполняет одну итерацию и только потом проверяет условие.

В *Python* подобного специального оператора нет, однако используя механизм бесконечного цикла с последующим прерывание, можно получить аналог подобной конструкции:

```
x = 100
while True:
    x += 1
    print(x)
    if x > 5:
        break
```

Каким бы ни было начальное значение переменной *x*, созданный цикл всегда выполнит одну итерацию. Данное поведение соответствует концепции постусловия.