Управляющие структуры и функции

С.В. Лемешевский (sergey.lemeshevsky@gmail.com)

Институт математики НАН Беларуси

Mar 18, 2020

Рассматриваться управляющие структуры языка Python: условные инструкции и циклы, инструкции обработки исключительных ситуаций. Кроме того рассматривается создание собственных функций, и здесь будет подробно рассматриваться чрезвычайно гибкий механизм работы с аргументами функций.

Содержание

1. Управляющие структуры

В языке Python условное ветвление реализуется с помощью инструкции `if`, а циклическая обработка – с помощью инструкций `while` и `for ... in`. В языке Python имеется также такая конструкция, как условное выражение – вариант инструкции `if`, аналог трехместного оператора (`?:`), имеющегося в C-подобных языках.

1.1. Условное ветвление

Общий синтаксис инструкции условного ветвления в языке Python имеет следующий вид:

```
if boolean_expression1:
    suite1
elif boolean_expression2:
    suite2
...
elif boolean_expressionN:
    suiteN
else:
    else_suite
```

Инструкция может содержать ноль или более предложений `elif`. Заключительное предложение `else` также является необязательным. Если необходимо предусмотреть ветку для какого-то особого случая, который не требует никакой обработки, в качестве блока кода этой ветки можно использовать инструкцию `pass` (она ничего не делает и просто является инструкцией-заполнителем, используемой там, где должна находиться хотя бы одна инструкция).

В некоторых случаях можно сократить инструкцию `if ... else` до единственного условного выражения. Ниже приводится синтаксис условных выражений:

```
expression1 if boolean_expression else expression2
```

Если логическое выражение `boolean_expression` возвращает значение `True`, результатом всего условного выражения будет результат выражения `expression1`, в противном случае – результат выражения `expression2`.

Об использовании скобок

Предположим, что нам необходимо записать в переменную `width` значение 100 и прибавить к нему 10, если переменная margin имеет значение `True`. Мы могли бы написать такое выражение:

```
width = 100 + 10 if margin else 0    # ОШИБКА!
```

Особенно неприятно, что эта строка программного кода работает правильно, когда переменная `margin` имеет значение `True`, записывая значение 110 в переменную width. Но когда переменная `margin` имеет значение `False`, в переменную `width` вместо 100 будет записано значение 0. Это происходит потому, что интерпретатор Python воспринимает выражение 100 + 10 как часть `expression1` условного выражения. Решить эту проблему можно с помощью круглых скобок:

```
width = 100 + (10 if margin else 0)
```

Кроме того, круглые скобки делают программный код более понятным для человека.

Условные выражения могут использоваться для видоизменения сообщений, выводимых для пользователя. Например, при выводе числа обработанных файлов, вместо того чтобы печатать «0 файл(ов)», «1 файл(ов)» или что-то подобное, можно было бы использовать пару условных выражений:

```
print("{0} файл{1}".format((count if count != 0 else "нет"), ("ов" if count % 10 not in [1, 2, 3, 4] else ("а" if cou
```

1.2. Циклы

В языке Python есть две инструкции циклов – `while` и `for ... in`.

Циклы `while`. Ниже приводится полный синтаксис цикла `while`:

```
while boolean_expression:
    while_suite
else:
    else_suite
```

Предложение `else` является необязательным. До тех пор, пока выражение `boolean_expression` возвращает значение `True`, в цикле будет выполняться блок `while_suite`. Если выражение `boolean_expression` вернет значение `False`, цикл завершится, и при наличии предложения `else` будет выполнен блок `else_suite`. Если внутри блока `while_suite` выполняется инструкция `continue`, то управление немедленно передается в начало цикла и выражение `boolean_expression` вычисляется снова. Если цикл не завершается нормально, блок предложения `else` не выполняется.

Необязательное предложение `else` имеет несколько сбивающее с толку название, поскольку оно выполняется во всех в случаях, когда цикл нормально завершается. Если цикл завершается в результате выполнения инструкции `break` или `return`, когда цикл находится внутри функции или метода, или в результате исключения, то блок `else_suite` предложения `else` не выполняется. (При возникновении исключительной ситуации интерпретатор Python пропускает предложение `else` и пытается отыскать подходящий обработчик исключения, о чем будет рассказываться в следующем разделе.) Плюсом такой реализации является одинаковое поведение предложения `else` в циклах `while`, в циклах `for ... in` и в блоках `try ... except`.

Рассмотрим пример, демонстрирующий предложение `else` в действии. Методы `str.index()` и `list.index()` возвращают индекс заданной подстроки или элемента или возбуждают исключение `ValueError`, если подстрока или элемент не найдены. Метод `str.find()` делает то же самое, но в случае неудачи он не возбуждает исключение, а возвращает значение `-1`. Для списков не существует эквивалентного метода, но при желании мы могли бы создать такую функцию, использующую цикл `while`:

```
def list_find(lst, target):
    index = 0
    while index < len(lst):
        if lst[index] == target:
```

```
            break
        index += 1
    else:
        index = -1
    return index
```

Эта функция просматривает список в поисках заданного элемента `target`. Если искомый элемент будет найден, инструкция `break` завершит цикл и вызывающей программе будет возвращен соответствующий индекс. Если искомый элемент не будет найден, цикл достигнет конца списка и завершится обычным способом. В случае нормального завер- шения цикла будет выполнен блок в предложении `else`, индекс получит значение `-1` и будет возвращен вызывающей программе.

1.3. Циклы `for`

Подобно циклу `while`, полный синтаксис цикла `for ... in` также включает необязательное предложение `else`:

```
for expression in iterable:
    for_suite
else:
    else_suite
```

В качестве выражения `expression` обычно используется либо единственная переменная, либо последовательность переменных, как правило, в форме кортежа. Если в качестве выражения `expression` используется кортеж или список, каждый элемент итерируемого объекта `iterable` распаковывается в элементы `expression`.

Если внутри блока `for_suite` встретится инструкция `continue`, управление будет немедленно передано в начало цикла и будет начата новая итерация. Если цикл завершается по выполнении всех итераций и в цикле присутствует предложение `else`, выполняется блок `else_suite`. Если выполнение цикла прерывается принудительно (инструкцией `break` или `return`), управление немедленно передается первой инструкции, следующей за циклом, а дополнительное предложение `else` при этом пропускается. Точно так же, когда возбуждается исключение, интерпретатор Python пропускает предложение `else` и пытается отыскать подходящий обработчик исключения (о чем будет рассказываться в следующем разделе).

Ниже приводится версия функции `list_find()`, реализованная на базе цикла `for ... in`, которая так же, как и версия на базе цикла `while`, демонстрирует предложение `else` в действии:

```
def list_find(lst, target):
    for index, x in enumerate(lst):
        if x == target:
        break
    else:
            index = -1
    return index
```

Как видно из этого фрагмента, переменные, созданные в выражении `expression` цикла `for ... in`, продолжают существовать после завершения цикла. Как и любые локальные переменные, они прекращают свое существование после выхода из области видимости, включающей их.

2. Обработка исключений

Об ошибках и исключительных ситуациях интерпретатор Python сообщает посредством возбуждения исключений, хотя в некоторых библиотеках сторонних разработчиков еще используются устаревшие приемы, такие как возврат «ошибочного» значения.

2.1. Перехват и возбуждение исключений

Перехватывать исключения можно с помощью блоков `try ... except`, которые имеют следующий синтаксис:

```
try:
    try_suite
except exception_group1 as variable1:
    except_suite1
...
except exception_groupN as variableN:
    except_suiteN
else:
    else_suite
finally:
    finally_suite
```

Эта конструкция должна содержать хотя бы один блок except, а блоки `else` и `finally` являются необязательными. Блок `else_suite` выполняется, только если блок `try_suite` завершается обычным способом, и не выполняется в случае возникновения исключения. Если блок `finally` присутствует, он выполняется всегда и в последнюю очередь.

Каждая группа `exception_group` в предложении except может быть единственным исключением или кортежем исключений в круглых скобках. Часть `as variable` в каждой группе является необязательной. В случае ее использования в переменную variable записывается ссылка на исключение, которое возникло, благодаря этому к нему можно будет обратиться в блоке `except_suite`.

Если исключение возникнет во время выполнения блока `try_suite`, интерпретатор поочередно проверит каждое предложение `except`. Если будет найдена соответствующая группа `exception_group`, будет выполнен соответствующий блок `except_suite`. Соответствующей считается группа, в которой присутствует исключение того же типа, что и возникшее исключение, или возникшее исключение является подклассом одного из исключений, перечисленных в группе.

На рис. 1 демонстрируется порядок выполнения типичной конструкции `try ... except ... finally`.

Ниже приводится окончательная версия функции `list_find()`, на этот раз она использует механизм обработки исключения:

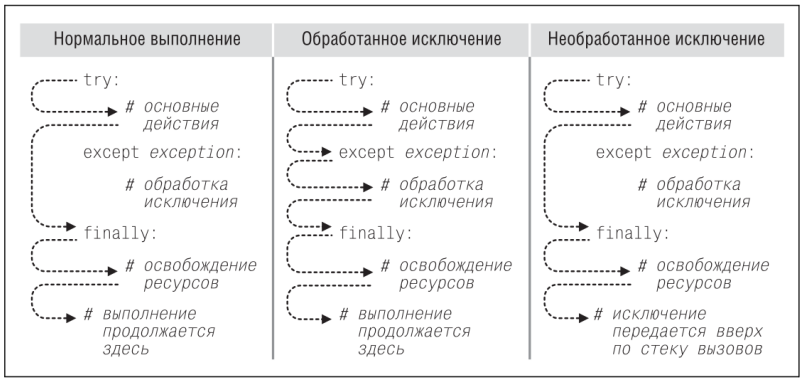

Рис. 1: Порядок выполнения конструкции `try ... except ... finally`

```
def list_find(lst, target):
    try:
        index = lst.index(target)
    except ValueError:
        index = -1
    return index
```

Здесь мы использовали конструкцию `try ... except` для преобразования исключения в возвращаемое значение. Аналогичный подход можно использовать для перехвата одних исключений и возбуждения других – с этим приемом мы познакомимся очень скоро.

3. Возбуждение исключений

Исключения представляют собой удобное средство управления потоком выполнения. Мы можем воспользоваться этим, используя либо встроенные исключения, либо создавая свои собственные и возбуждая нужные нам, когда это необходимо. Возбудить исключение можно одним из двух способов:

```
raise exception(args)
raise
```

В первом случае, то есть когда явно указывается возбуждаемое исключение, оно должно быть либо встроенным, либо нашим собственным, наследующим класс `Exception`. Если исключению в виде аргумента передается некоторый текст, этот текст будет выведен на экран, если исключение не будет обработано программой. Во втором случае, то есть когда исключение не указывается, инструкция `raise` повторно возбудит текущее активное исключение, а в случае отсутствия активного исключения будет возбуждено исключение `TypeError`.

4. Собственные функции

Функции представляют собой средство, дающее возможность упаковывать и параметризовать функциональность. В языке Python можно создать четыре типа функций: глобальные функции, локальные функции, лямбда-функции и методы.

Все функции, которые мы создавали до сих пор, являются *глобальными* функциями. Глобальные объекты (включая функции) доступны из любой точки программного кода в том же модуле (то есть в том же самом файле `.py`), которому принадлежит объект. Глобальные объекты доступны также и из других модулей, как будет показано в следующей главе.

Локальные функции (их еще называют вложенными функциями) – это функции, которые объявляются внутри других функций. Эти функции видимы только внутри тех функций, где они были объявлены – они особенно удобны для создания небольших вспомогательных функций, которые нигде больше не используются.

Лямбда-функции – это выражения, поэтому они могут создаваться непосредственно в месте их использования; они имеют множество ограничений по сравнению с обычными функциями. Методы – это те же функции, которые ассоциированы с определенным типом данных и могут использоваться только в связке с этим типом данных.

Синтаксис создания функции (глобальной или локальной) имеет следующий вид:

```
def functionName(parameters):
    suite
```

Параметры `parameters` являются необязательными и при наличии более одного параметра записываются как последовательность идентификаторов через запятую или в виде последовательности пар `identifier=value`, о чем вскоре будет говориться подробнее. Например, ниже приводится функция, которая вычисляет площадь треугольника по формуле Герона:

```
def heron(a, b, c):
    s = (a + b + c) / 2
    return math.sqrt(s * (s - a) * (s - b) * (s - c))
```

Внутри функции каждый параметр, `a`, `b` и `c`, инициализируется соответствующими значениями, переданными в виде аргументов. При вызове функции мы должны указать все аргументы, например, `heron(3, 4, 5)`. Если передать слишком мало или слишком много аргументов, будет возбуждено исключение `TypeError`. Производя такой вызов, мы говорим, что используем позиционные аргументы, потому что каждый переданный аргумент становится значением параметра в соответствующей позиции. То есть в данном случае при вызове функции параметр `a` получит значение `3`, параметр `b` – значение `4` и параметр `c` – значение `5`.

Все функции в языке Python возвращают какое-либо значение, хотя вполне возможно (и часто так и делается) просто игнорировать это значение. Возвращаемое значение может быть единственным значением или кортежем значений, а сами значения могут быть коллекциями, поэтому практически не существует никаких ограничений на то, что могут возвращать функции. Мы можем покинуть функцию в любой момент, используя инструкцию

return. Если инструкция return используется без аргументов или если мы вообще не используем инструкцию return, функция будет возвращать значение None.

Некоторые функции имеют параметры, для которых может существо- вать вполне разумное значение по умолчанию. Например, ниже при- водится функция, которая подсчитывает количество алфавитных сим- волов в строке; по умолчанию подразумеваются алфавитные символы из набора ASCII:

```
def letter_count(text, letters=string.ascii_letters):
    letters = frozenset(letters)
    count = 0
    for char in text:
        if char in letters:
            count += 1
    return count
```

Здесь при помощи синтаксиса parameter=default было определено значение по умолчанию для параметра letters. Это позволяет вызывать функцию letter_count() с единственным аргументом, например, letter_count(”Maggie and Hopey”). В этом случае внутри функции параметр letter будет содержать строку, которая была задана как значение по умолчанию. Но за нами сохраняется возможность изменить значение по умолчанию, например, указав дополнительный позиционный аргумент: letter_count(”Maggie and Hopey”, ”aeiouAEIOU”), или используя именованный аргумент (об именованных аргументах рассказывается ниже): letter_count(”Maggie and Hopey”, letters=”aeiouAEIOU”).

Синтаксис параметров не позволяет указывать параметры, не имеющие значений по умолчанию, после параметров со значениями по умолчанию, поэтому такое определение: def bad(a, b=1, c):, будет вызывать синтаксическую ошибку. С другой стороны, мы не обязаны передавать аргументы в том порядке, в каком они указаны в определении функции – мы можем использовать именованные аргументы и передавать их в виде name=value.

4.1. Распаковывание аргументов и параметров

В предыдущей главе мы видели, что для передачи позиционных аргументов можно использовать оператор распаковывания последовательностей (*). Например, если возникает необходимость вычислить площадь треугольника, а длины всех его сторон хранятся в списке, то мы могли бы вызвать функцию так: heron(sides[0], sides[1], sides[2]), или просто распаковать список и сделать вызов намного проще: heron(*sides). Если элементов в списке (или в другой последовательности) больше, чем параметров в функции, мы можем воспользоваться операцией извлечения среза, чтобы извлечь нужное число аргументов.

Мы можем также использовать оператор распаковывания последовательности в списке параметров функции. Это удобно, когда необходимо создать функцию, которая может принимать переменное число позиционных аргументов. Ниже приводится функция product(), которая вычисляет произведение своих аргументов:

```
def product(*args):
    result = 1
    for arg in args:
        result *= arg
    return result
```

Эта функция имеет единственный аргумент с именем `args`. Наличие символа * перед ним означает, что внутри функции параметр `args` обретает форму кортежа, значениями элементов которого будут значения переданных аргументов. Ниже приводятся несколько примеров вызова функции:

```
product(1, 2, 3, 4) # args == (1, 2, 3, 4); вернет: 24
product(5, 3, 8)    # args == (5, 3, 8); вернет: 120
product(11)           # args == (11,); вернет: 11
```

Мы можем использовать именованные аргументы вслед за позиционными, как в функции, которая приводится ниже, вычисляющей сумму своих аргументов, каждый из которых возводится в заданную степень:

```
def sum_of_powers(*args, power=1):
    result = 0
    for arg in args:
        result += arg ** power
    return result
```

Эта функция может вызываться только с позиционными аргументами, например: `sum_of_powers(1, 3, 5)`, или как с позиционными, так и с именованным аргументами, например: `sum_of_powers(1, 3, 5, power=2)`.

Допускается также использовать символ * в качестве самостоятельного «параметра». В данном случае он указывает, что после символа * не может быть других позиционных параметров, однако указание именованных аргументов допускается. Ниже приводится модифицированная версия функции `heron()`. На этот раз функция принимает точно три позиционных аргумента и один необязательный именованный аргумент.

```
def heron2(a, b, c, *, units="meters"):
    s = (a + b + c) / 2
    area = math.sqrt(s * (s - a) * (s - b) * (s - c))
    return "{0} {1}".format(area, units)
```

Ниже приводятся несколько примеров вызовов функции:

```
heron2(25, 24, 7)          # вернет: '84.0 meters'
heron2(41, 9, 40, units="inches") # вернет: '180.0 inches'
heron2(25, 24, 7, "inches")      # ОШИБКА! Возбудит исключение TypeError
```

В третьем вызове мы попытались передать четыре позиционных аргумента, но оператор * не позволяет этого и вызывает исключение `TypeError`.

Поместив оператор * первым в списке параметров, мы тем самым полностью запретим использование *любых* позиционных аргументов и вынудим тех, кто будет вызывать ее, использовать именованные аргументы. Ниже приводится пример сигнатуры такой (вымышленной) функции:

```
def print_setup(*, paper="Letter", copies=1, color=False):
```

Мы можем вызывать функцию `print_setup()` без аргументов, допуская использование значений по умолчанию. Или изменить некоторые или все значения по умолчанию, например: `print_setup(paper="A4", color=True)`. Но если мы попытаемся использовать позиционные аргументы, например: `print_setup("A4")`, будет возбуждено исключение `TypeError`.

Так же, как мы распаковываем последовательности для заполнения позиционных параметров, можно распаковывать и отображения – с помощью оператора распаковывания отображений (**). Мы можем использовать оператор **, чтобы передать содержимое словаря в функцию `print_setup()`. Например:

```
options = dict(paper="A4", color=True)
print_setup(**options)
```

В данном случае пары «ключ-значение» словаря `options` будут распакованы, и каждое значение будет ассоциировано с параметром, чье имя соответствует ключу этого значения. Если в словаре обнаружится ключ, не совпадающий ни с одним именем параметра, будет возбуждено исключение `TypeError`. Любые аргументы, для которых в словаре не найдется соответствующего элемента, получат значение по умолчанию, но если такие аргументы не имеют значения по умолчанию, будет возбуждено исключение `TypeError`.

Кроме того, имеется возможность использовать оператор распаковывания вместе с параметрами в объявлении функции. Это позволяет создавать функции, способные принимать любое число именованных аргументов. Ниже приводится функция `add_person_details()`, которая принимает номер карточки социального страхования и фамилию в виде позиционных аргументов, а также произвольное число именованных аргументов:

```
def add_person_details(ssn, surname, **kwargs):
    print("SSN =", ssn)
    print(" surname =", surname)
    for key in sorted(kwargs):
        print(" {0} = {1}".format(key, kwargs[key]))
```

Эта функция может вызываться как только с двумя позиционными аргументами, так и с дополнительной информацией, например: `add_person_details(83272171, "Luther", forename="Lexis", a`
Такая возможность обеспечивает огромную гибкость. Конечно, мы можем также одновременно принимать переменное число позиционных аргументов и переменное число именованных аргументов:

```
def print_args(*args, **kwargs):
    for i, arg in enumerate(args):
    print("positional argument {0} = {1}".format(i, arg))
    for key in kwargs:
        print("keyword argument {0} = {1}".format(key, kwargs[key]))
```

Эта функция просто выводит полученные аргументы. Она может вызываться вообще без аргументов или с произвольным числом позиционных и именованных аргументов.

4.2. Доступ к переменным в глобальной области видимости

Иногда бывает удобно иметь несколько глобальных переменных, доступных из разных функций программы. В этом нет ничего плохого, если речь идет о «константах», но в случае переменных – это не самый лучший выход, хотя для коротких одноразовых программ это в некоторых случаях можно считать допустимым.

Программа digit_names.py принимает необязательный код языка (`en` или `ru`) и число в виде аргументов командной строки и выводит названия всех цифр заданного числа. То есть если в командной строке программе было передано число `123`, она выведет `one two three`. В программе имеется три глобальные переменные:

```
Language = "en"

ENGLISH = {0: "zero", 1: "one", 2: "two", 3: "three", 4: "four",
           5: "five", 6: "six", 7: "seven", 8: "eight", 9: "nine"}
RUSSIAN = {0: "ноль", 1: "один", 2: "два", 3: "три", 4: "четыре",
           5: "пять", 6: "шесть", 7: "семь", 8: "восемь", 9: "девять"}
```

Мы следуем соглашению, в соответствии с которым имена переменных, играющих роль констант, записываются только символами верхнего регистра, и установили английский язык по умолчанию.

В некотором другом месте программы выполняется обращение к переменной `Language`, и ее значение используется при выборе соответствующего словаря:

```
def print_digits(digits):
    dictionary = ENGLISH if Language == "en" else RUSSIAN
    for digit in digits:
        print(dictionary[int(digit)], end=" ")
    print()
```

Когда интерпретатор Python встречает имя переменной `Language` внутри функции, он пытается отыскать его в локальной области видимости (в области видимости функции) и не находит. Поэтому он продолжает поиск в глобальной области видимости (в области видимости файла `.py`), где и обнаруживает его.

Ниже приводится содержимое функции `main()` программы. Она изменяет значение переменной `Language` в случае необходимости и вызывает функцию `print_digits()` для вывода результата.

```
def main():
    if len(sys.argv) == 1 or sys.argv[1] in {"-h", "--help"}:
        print("usage: {0} [en|ru] number".format(sys.argv[0]))
        sys.exit()

    args = sys.argv[1:]
    if args[0] in {"en", "ru"}:
        global Language
        Language = args.pop(0)
    print_digits(args.pop(0))
```

Обратите внимание на использование инструкции `global` в этой функции. Эта инструкция используется для того, чтобы сообщить интерпретатору, что данная переменная существует в глобальной области видимости (в области видимости файла `.py`) и что операция присваивания должна применяться к глобальной переменной; без этой инструкции операция присваивания создаст локальную переменную с тем же именем.

Замечание
Если не использовать инструкцию `global`, программа сохранит свою работоспособность, но когда интерпретатор встретит переменную `Language` в условной инструкции `if`, он попытается отыскать ее в локальной области видимости (в области видимости функции) и, не обнаружив ее, создаст новую локальную переменную с именем `Language`, оставив глобальную переменную `Language` без изменений. Эта малозаметная ошибка будет проявляться только в случае запуска программы с аргументом `ru`, потому что в этом случае будет создана новая локальная переменная `Language`, в которую будет записано значение `ru`, а глобальная переменная `Language`, которая используется функцией `print_digits()`, по-прежнему будет иметь значение `en`.

В сложных программах лучше вообще не использовать глобальные переменные, за исключением констант, которые не требуют употребления инструкции `global`.

4.3. Лямбда-функции

Лямбда-функции – это функции, для создания которых используется следующий синтаксис:

```
lambda parameters: expression
```

Часть `parameters` является необязательной, а если она присутствует, то обычно представляет собой простой список имен переменных, разделенных запятыми, то есть позиционных аргументов, хотя при необходимости допускается использовать полный синтаксис определе- ния аргументов, используемый в инструкции `def`. Выражение `expression` не может содержать условных инструкций или циклов (хотя условные выражения являются допустимыми), а также не может содержать инструкцию `return` (или `yield`). Результатом лямбда-выражения является анонимная функция. Когда вызывается лямбда-функция, она возвращает результат вычисления выражения `expression`. Если выражение expression представляет собой кортеж, оно должно быть заключено в круглые скобки.

Ниже приводится пример простой лямбда-функции, которая добавляет (или не добавляет) суффикс `s` в зависимости от того, имеет ли аргумент значение 1:

```
s = lambda x: "" if x == 1 else "s"
```

Лямбда-выражение возвращает анонимную функцию, которая присваивается переменной `s`. Любая (вызываемая) переменная может вызываться как функция при помощи круглых скобок, поэтому после выполнения некоторой операции можно при помощи функции `s()` вывести сообщение с числом обработанных файлов, например:

```
print("{0} file{1} processed".format(count, s(count)))
```

Лямбда-функции часто используются в виде аргумента `key` встроенной функции `sorted()` или метода `list.sort()`. Предположим, что имеется список, элементами которого являются трехэлементные кортежи (номер группы, порядковый номер, название), и нам необходимо отсортировать этот список различными способами. Ниже приводится пример такого списка:

```
elements = [(2, 12, "Mg"), (1, 11, "Na"), (1, 3, "Li"), (2, 4, "Be")]
```

Отсортировав список, мы получим следующий результат:

```
[(1, 3, 'Li'), (1, 11, 'Na'), (2, 4, 'Be'), (2, 12, 'Mg')]
```

Ранее, когда мы рассматривали функцию `sorted()`, то видели, что имеется возможность изменить порядок сортировки, если в аргументе `key` передать требуемую функцию. Например, если необходимо отсортировать список не по естественному порядку: номер группы, порядковый номер и название, а по порядковому номеру и названию, то мы могли бы написать маленькую функцию `def ignore0(e): return e[1], e[2]` и передавать ее в аргументе `key`. Но создавать в программе массу крошечных функций, подобных этой, очень неудобно, поэтому часто используется альтернативный подход, основанный на применении *лямбда-функций*:

```
elements.sort(key=lambda e: (e[1], e[2]))
```

Здесь в качестве значения аргумента key используется выражение `lambda e: (e[1], e[2])`, которому в виде аргумента e последовательно передаются все трехэлементные кортежи из списка. Круглые скобки, окружающие *лямбда-выражение*, обязательны, когда выражение является кортежем и *лямбда-функция* создается как аргумент другой функции. *Для* достижения того же эффекта можно было бы использовать операцию извлечения среза:

```
elements.sort(key=lambda e: e[1:3])
```

Немного более сложная версия обеспечивает возможность сортировки по названию, без учета регистра символов, и порядковому номеру:

```
elements.sort(key=lambda e: (e[2].lower(), e[1]))
```

5. Утверждения

Что произойдет, если функция получит аргументы, имеющие ошибочные значения? Что случится, если в реализации алгоритма будет допущена ошибка и вычисления будут выполнены неправильно? Самое неприятное, что может произойти, – это то, что программа будет

выполняться без каких-либо видимых проблем, но будет давать неверные результаты. Один из способов избежать таких коварных проблем состоит в том, чтобы писать тесты. Другой способ состоит в том, чтобы определить предварительные условия и ожидаемый конечный результат, и сообщать об ошибке, если они не соответствуют друг другу. В идеале следует использовать как тестирование, так и метод на основе сравнения предварительных условий и ожидаемых результатов.

Предварительные условия и ожидаемый результат можно задать с помощью инструкции `assert`, которая имеет следующий синтаксис:

```
assert boolean_expression, optional_expression
```

Если выражение `boolean_expression` возвращает значение `False`, возбуждается исключение `AssertionError`. Если задано необязательное выражение `optional_expression`, оно будет использовано в качестве аргумента исключения `AssertionError`, что удобно для передачи сообщений об ошибках. Однако следует отметить, что утверждения предназначены для использования разработчиками, а не конечными пользователями. Проблемы, возникающие в процессе нормальной эксплуатации программы, такие как отсутствующие файлы или ошибочные аргументы командной строки, должны обрабатываться другими средствами, например, посредством вывода сообщений об ошибках или записи сообщений в файл журнала.