ЛАБОРАТОРНАЯ РАБОТА

Тема: ***Организация ветвлений. Предикаты***

Цель работы: Изучение использования разных форм условного оператора и применения предикатов и других методов для работы со строками.

Содержание

Два смысла термина "оператор" 1
Функции и методы. Предикаты 1
ЗАДАНИЕ 1 2
ЗАДАНИЕ 2 3
ЗАДАНИЕ 3 4
Вопросы для самоконтроля 5
Справочная информация 5
1. Некоторые методы строковых объектов 5
2. Простое ветвление (конструкция `if-else`) 5
3. Ветвление по многим направлениям 6
4. Запись условий 7

Два смысла термина "оператор"

В программировании понятие "оператор" применяется к двумя разными вещам.

Во-первых, операторами (*англ.* `operator`) называют символы, которыми обозначаются действия, выполняемые с операндами при вычислении выражений (+, -, >= и т.д.).

Во-вторых, операторами (*англ.* `operator`, `statement`) называют законченные предложения программ. Эти конструкции могут быть простыми, как оператор присваивания, или сложными, как условный оператор.

Особенность выражений — вычисление завершается получением некоторого результата, который далее подставляется вместо выражения. Например, при выполнении инструкции `print(1+2)` сначала вычисляется выражение `1+2`, а затем выражение заменяется результатом: `print(3)`.

Выполнение (вычисление) операторов не подразумевает возвращение результатов. Например, попытка выполнить инструкцию `print(x=3)` приведет к ошибке.

Функции и методы. Предикаты

Предикат в программировании — это функция, принимающая один или более аргументов и возвращающая значения булева типа (`True` или `False`).

К предикатам относятся функции, которые проверяют наличие у объектов некоторого свойства или их принадлежность к определенному типу. Такие функции должны возвращать ответ типа `да` (принадлежит, истина) или `нет` (не принадлежит, ложь), которые соответствуют булевым значениям.

Встроенные функции-предикаты есть практически во всех системах программирования. Обычно они имеют имена, начинающиеся с приставки `is` или `is_`. (т.е. `является ...`).

В Питоне есть предикаты для работы с символьными строками. Но эти предикаты не обычные функции, а *методы* принадлежат объектам данных (т.е. символьным строкам). Различие между методами и обычными функциями проявляется в обращении к ним.

Обычные функции существуют "сами по себе". Поэтому к ним можно обращаться непосредственно. Например, встроенная функция `len()` вызывается так:

```
>>> len("12345")
```

Методы — собственность объектов данных, они "подневольны". Их непосредственно вызвать нельзя, а нужно обращаться через объект хозяина.

Пример.

Метод строковых объектов `is`**`alpha`**`()` проверяет, построена ли строка только из символов *алфа*вита. Он возвращает `True`, если его "хозяином" является *непустая* строка, состоящая только букв, и `False`, когда в строке присутствует хотя бы один посторонний символ, например, пробел, или строка пустая.

```
>>> "Питон".isalpha()
True
>>> "".isalpha()
False
```

Список некоторых предикатов и методов строковых объектов Питона приведен в справочной информации.

ЗАДАНИЕ 1 (*модернизация приложения для поиска палиндромов*)

Скопировать программу, полученную в задании 3 лабораторной работы №6 именем `palyndrome_7-1.py`.

1. Усиление контроля за вводимой информацией.

В лабораторной работе №6 требовалось разработать функцию `get_word()`, которая реализует диалог с пользователем и выполняет простую проверку, не ввел ли пользователь пустую строку (задание 3).

Предположим в ответ на подсказку пользователь ввел, слово "**Т**опо**т**". Из-за различия в регистрах буквы "т" приложение ответит, что это слово *палиндромом не является*.

А, если пользователь введет "12-21", то ответом будет "палиндром". Конечно, в редких случаях может возникнуть задача, искать строки-перевёртыши любого содержания. но мы далее ограничимся строками, состоящими только из букв (словами).

- ***Уточнение формальной постановки задачи***.

В постановке задачи в лабораторной работе №6 по поводу содержания строк и регистров используемых в них букв никаких ограничений не было указано. Поэтому постановку необходимо уточнить:

Приложение предназначено для анализа на принадлежность к палиндромам одного слова — символьной строки, состоящей только из букв. Буквы рассматриваются без учета их регистров.

Замечание

Ответственность за осмысленность вводимых слов и возможное смешивание букв латинского алфавита и кириллицы несет пользователь.

- ***Внесение изменений в функцию get_word()***.

Чтобы функция `get_word()` обеспечивала проверку содержания введённой строки новым требованиям, нужно внести в функцию следующие изменения.

Применить к введенной пользователем строке предикат `isalpha()`. результат становится возвращаемым значением функции.

Иначе, если введённая строка пустая или содержит посторонние символы, выводится сообщение "`Ошибка ввода!!!`" и выполнение программы завершается.

Требуется реализовать описанные действия с помощью условного оператора вида `if-else`.

2. Превращение функции `is_palindrom()` в предикат.

В файле программы создать копию функции `is_palindrom()` под именем `is_palindrom_old(word)`.

В задании 1 лабораторной работы №6 требовалось создать функцию `is_palindrom(word)`, которая выполняет главную для приложения проверку: является ли переданное ей слово `word` палиндромом или нет.

Функция с помощью оператора `return` возвращала результат вычисления условного *выражения*, а именно, строку "`палиндром`" или "`не палиндром`".

- ***Изменение возвращаемого значения функции is_palindrom().***

Очевидно, функция `is_palindrom` по своему смыслу выполняет роль предиката — она проверяет, обладает ли заданное слово свойством палиндрома или нет. Поэтому будет логично, если `is_palindrome` будет возвращать не текстовые строки "`палиндром`" или "`не палиндром`", а, как любой предикат, универсальные значения `True` или `False`.

В этом случае функцию `is_palindrome()` можно будет использовать не только для обслуживания потребностей функции `create_message()` (для чего она возвращала конкретный текст), но и для других задач.

В этом случае инструкция с возвращением значения из функции упростится, потому что достаточно будет возвратить результат проверки условия.

Когда нужно обеспечить независимость результата проверки от регистров букв в словах, то применяется простой прием: все буквы слова преобразуются к одному регистру.

Например, в Питоне можно с помощью *метода* `lower()` выполнить преобразование букв в заданной строке к нижнему регистру. Т.е. получится слово только из строчных букв.

Использовать этот метод для преобразования содержания переменной `word` и запомнить результат в переменной `word_lower`.

Если теперь проверку на палиндром выполнить для `word_lower`, то регистры букв в исходном слове влияния оказывать не будут роли не будут.

Внести необходимые изменения в код функции `is_palindrome()`.

- ***Построение строки с результатом вычислений.***

Так как смысл возвращаемого значения функции `is_palindrome` изменился, то нужно внести изменения в функцию `create_message()`, формирующую текст итогового сообщения.

В лаб. раб. №6 в строку для вывода добавлялся готовый текст полученный при выполнении `is_palindrome` из переданный в переменную `what_is`. Теперь этот текст должен формироваться в функции `create_message()`, что по смыслу действия также более логично для этой функции.

Воспользуемся тем, что форматная строка при подстановке аргументов позволяет выполнять действия, аналогичные конкатенации строк. Например,

```
>>> s1="Привет, "
>>> s2="Питон"
>>> "%s%s"%(s1,s2)
'Привет, Питон'
```

Текст "`не палиндром`" можно рассматривать как результат конкатенации строк "`не `" и "`палиндром`". Чтобы получить просто "`палиндром`", при конкатенации частицу "`не `" нужно заменить пустой строкой.

Внесем в функцию следующие изменения

Определим переменную `prefix`, которой с помощью условного выражения, основанного на значении `what_is`, присвоим значение "" или "`не `".

В форматную строку добавим дополнительную спецификацию, предназначенную для префиксной строки, а в список аргументов — переменную `prefix`. Общую для двух сообщений строку "`палиндром`", можно включить прямо в список аргументов или предварительно сохранить в какой-то вспомогательной переменной и добавить в список вывода эту переменную.

ЗАДАНИЕ 2 (*Ветвление по нескольким направлениям*)

В справочной информации приведены синтаксические конструкции, обеспечивающие возможность реализации в программах ветвления вычислительного процесса по многих направлениям.

1. В вузах официально принята четырехбальная текстовая система оценивания знаний: отлично, хорошо, удовлетворительно, неудовлетворительно. Однако на бытовом уровне часто используется школьная числовая система оценок (`5`,`4`,`3` и `2`).

а) Создать новый файл `grade1.py`.

б) Определить в программе функцию `grade1(mark)`, которая получает через аргумент оценку по школьной системе и возвращает её вузовский эквивалент.

в) Добавить в файл программы инструкции, с помощью которых у пользователя запрашивается оценка по школьной системе, эта оценка передается функции `grade1(mark)` и на экран выводится текстовое значение оценки.

2. В ЮФУ принята рейтинговая система оценивания знаний. Итоговая оценка выставляется по следующим критериям:

Баллы	*Оценка*
`85-100`	`Отлично`
`71-84`	`Хорошо`

60-70	Удовлетворительно
< 60	Неудовлетворительно

а) Создать новый файл `grade2.py`.

б) Определить в программе функцию `grade2(mark)`, которая получает через аргумент рейтинговую оценку в баллах и возвращает соответствующий текстовый эквивалент.

При выполнении задания применить хотя бы один раз каждую из синтаксических форм.

в) Добавить в файл программы инструкции, с помощью которых у пользователя запрашивается оценка в баллах, эта оценка передается функции `grade2(mark)` и на экран выводится текстовое значение оценки.

3. Изучается поведение условного оператора в случае, когда налагаемые условия не являются взаимоисключающими.

а) Создать новый файл `solutions.py`, для программы сообщающей о количестве решений у квадратного уравнения.

б) Определить в программе функцию `number_of_solutions(a,b,c)`, которая получает через аргумент значения коэффициентов уравнения, вычисляет дискриминант `d` и возвращает определенное по значению `d` сообщение о количестве решений квадратного уравнений, используя следующие условия:

`d < 0` — "нет действительных решений",

`d >= 0` — "есть корни",

`d == 0` — "два равных корня",

`d > 0` — "два различных корня".

в) Добавить в файл программы инструкции, необходимые пользователю для ввода коэффициентов `a`, `b` и `c` квадратного уравнения и вызова функции

г) Для тестирования программы использовать квадратные уравнения с разными значениями дискриминанта:

$x^2 + x + 1 = 0$,
$x^2 + 2x + 1 = 0$,
$x^2 - 3x + 2 = 0$.

Что выдает программа в примерах, где уравнение имеет два равных или два различных корня и почему?

Усовершенствовать проверки, чтобы при наличии корней выдавалось не одно, а два сообщения: "`есть корни`" и "`два равных корня`"/"`два корня`". Использовать вложенные операторы.

ЗАДАНИЕ 3 (*Поиск палиндромов — уточнение сообщений об ошибках*)

При вводе слова пользователь может совершить два вида ошибок: ввести недопустимые символы и ввести пустую строку.

Приложение пока в обоих случаях выдает одно и то же сообщение, хотя желательно, чтобы оно классифицировало ошибку.

Для этого в функцию **`get_word()`** требуется добавить проверку типа ошибки.

а) *Общая схема проверки.*

Для введенной строки вычисляется значение предиката `isalpha()`, которое используется как логическое выражение в конструкции `if`.

Если получено `True`, то функция `get_word()` *возвращает введенную строку*.

Когда получено `False`, то в конструкции `elif` проверяется условие "`введена пустая строка`" (как это сделать?).

Если строка пустая, то функция `get_word()` *возвращает* текст "`Пустая строка`".

Иначе (`else`) функция `get_word()` *возвращает* текст "`Недопустимый символ`".

Написать новый вариант функции `get_word()`, реализующий описанную проверку.

б) Проверить работу приложения во всех случаях: вводится допустимое слово, вводится недопустимое слово (например, с пробелом), вводится пустая строка.

Вопросы для самоконтроля

1. В каких двух смыслах используется термин "оператор"?

2. Что в программировании понимается под предикатом?

3. В чем различие между функциями и методами объектов?

4. Привести примеры методов-предикатов строковых объектов и их объяснить смысл их возвращаемых значений?

5. Какая конструкция языка Питон обеспечивает ветвление вычислительного процесса по двум направлениям, на основе взаимоисключающих условий?

6. Как выполняет свою работу условный оператор типа `if-elif-else`?

7. Можно ли реализовать ветвление и многим направлениям с помощью операторов `if-else`?

8. В чем различие между двумя способами организации ветвления по многим направлениям?

Справочная информация

1. Некоторые методы строковых объектов

Предикаты

а) **`isalnum`**`()` — `True` для *непустой* строки, состоящей только из букв и цифр, иначе `False`.

б) **`isalpha`**`()` — `True` для *непустой* строки, состоящей только из букв, иначе `False`.

в) **`islower()`** — `True` для *непустой* строки, состоящей только из букв нижнего регистра (строчных), иначе `False`.

г) **`isupper()`** — `True` для *непустой* строки, состоящей только из букв верхнего регистра (прописных, заглавных), иначе `False`.

д) **`isspace()`** — `True` для *непустой* строки, состоящей только из пробельных символов, т.е. из пробелов, табуляций `\t` и переводов строк `\n`, иначе `False`.

е) **`isdecimal(), isdigit(), isnumeric()`**— возвращает `True` для непустой строки, состоящей только из цифр, иначе `False`.

Различие между методами проявляется в тонких деталях, связанных с `Unicode`. В этой таблице, помимо кодов для обычных цифр (сохранена первая половина таблицы ASCII), есть еще коды для представления часто встречающихся в математике цифровых обозначений.

Примеры

Код `u\2155` представляет символ обыкновенной дроби ⅕. Для этого кода `True` вернет только `isnumeric()`, т.е. это вообще число, но не цифра и не обычное десятичное число из кодов таблицы ASCII.

Код `\u00B2` представляет цифру `2`, но в роли верхнего индекса. Для этого кода `True` вернут `isdigit()` и `isnumeric()`, т.е. это цифра и вообще число, но не обычное десятичное число.

Методы преобразования строк

а) **`lower`**() — преобразует все буквы в строке к нижнему регистру.

б) **`upper`**() — преобразует все буквы в строке к верхнему регистру.

2. Простое ветвление (конструкция `if-else`)

В простейшем случае условный оператор состоит из *строки с условием* и *тела*. (лаб. раб. №6). Условие управляет "барьером", преграждающим путь к инструкциям тела оператора.

Если условие выполнено (`True`), то барьер "поднимается". Если оно не выполнено (`False`), то барьер "опущен" и тело оператора нужно "обойти" стороной, как если бы условного оператора не было вообще.

а) Выбор одного из двух путей вычислений

Более интересным является управляющая конструкция, в которой по условию выбирается один из двух возможных путей продолжения вычислений. В этом случае для описания альтернативных вариантов действий конструкция условного оператора должна включать два

блока инструкций. Один блок, как и ранее, записывается после ключевого слова `if` с условием, что означает "`если условие выполнено`". Второй блок записывается после ключевого слова `else` , что означает "иначе, когда условие не выполнено".

```
if логическое_выражение:
        блок инструкций
        (условие выполнено)
else:
        блок инструкций
        (условие не выполнено)
инструкция после условного оператора
```

Говорят, что оператор `if-else` определяет две *ветви* вычислений, поэтому соответствующий прием программирования называют реализацией *ветвления*.

Вообще говоря, те же вычисления можно обеспечить без `else` двумя конструкциями `if` . Например, код

```
if x > 0:
  инструкции1
else:
  инструкции2
```

можно заменить на

```
if x > 0:
  инструкции1
if x <= 0:
  инструкции2
```

Но у варианта с использованием двух `if` есть очевидный недостаток: когда первое условие удовлетворено, то проверка второго условия всё равно будет выполняться, хотя очевидно, что она закончится неудачей. Использование `else` позволяет избежать лишних действий.

б) Условные выражения и условные операторы

В предыдущих лабораторных работах рассматривались условные *выражения*, которые строятся с помощью тех же ключевых слов `if-else`.

Основная особенность условного выражения — для вычислений в нём выбирается одно из двух *выражений* и только выражений. А в условном операторе в каждом случае может выполняться любое количество любых *инструкций*, т.е. целый фрагмент программы.

3. Ветвление по многим направлениям

Когда возможных путей продолжения вычислений больше двух, то для выбора каждого пути должно быть задано свое условие.

Дополнительные условия в условном операторе указываются после ключевого слова `elif`:

```
if условие_1:
        инструкции_1
elif условие _2:
        инструкции_2
elif условие _3:
        инструкции_3
инструкция после условного оператора
```

Если в конструкции `if-else` ветви выбираются при взаимоисключающих обстоятельствах, то в `if-elif-else` условия могут "перекрываться", т.е. одновременно может быть удовлетворено несколько условий. Поэтому нужно понимать *принципы работы оператора*:

- условия проверяются в порядке их появления в программе;
- реализуются инструкции, которые соответствуют *первому* найденному выполненному условию и на этом выполнение оператора *завершается*;
- если ни одно условий не соблюдено, то выполнение условного оператора *прекращается* без каких-либо действий;

• для случая, когда нет удовлетворенных условий, можно определить дополнительный набор инструкций, указав его после ключевого слова `else`.

На следующем рисунке показана схема организация ветвление по четырем направлениям.

В Питоне с помощью оператора `if-elif-else` она программируется следующим образом:

```
if условие_1:
        инструкции_1
elif условие_2:
        инструкции_2
elif условие_3:
        инструкции_3
else:
        инструкции_4
инструкция после условного оператора
```

Эта же схема может быть реализована также с помощью *вложенных условных операторов* `if-else`.

```
if условие_1:
    инструкции_1
else:
    if условие_2:
            инструкции_2
    else:
            if условие_3:
                инструкции_3
            else:
                инструкции_4
инструкция после условного оператора
```

Чтобы не возникало логических ошибок, вложенные операторы `if` всегда нужно помещать только после `else`!

Сравните конструкции `if-elif-else` и вложенных `if-else` с точки зрения формирования блоков инструкций.

4. Запись условий

В роли условий выступают выражения, результаты вычисления которых могут быть интерпретированы как `True` или `False`.

Часто условия записываются с помощью операторов сравнения. Операторы сравнения позволяют проверить выполнение конкретного отношения между операндами (больше, равно, меньше или равно и т.п.). Например, `x>=3`.

Если требуется проверить, попадает ли значение в заданный диапазон, то для одного и того же объекта данных необходимо выполнить две проверки, результаты которых учитываются совместно. Для этого можно комбинировать условия с помощью логической связки "**и**". Например, условие попадания величины `x` в диапазон от 1 до 5 строится из двух проверок: "`x >= 1`" **и** "`x <= 5`".

В Питоне это записывается с помощью логического оператора `and`:

```
x >= 1 and x <= 5.
```

Но в Питоне при задании двусторонних ограничениях на одну и ту же переменную разрешено можно также использовать обычную для математики и более компактную запись: `1 <= x <= 5`.