1.3 Основы работы со строками

Строка - неизменяемая последовательность символов. Строка представляет последовательность символов в кодировке Unicode, заключенных в кавычки. Причем для определения строк Python позволяет использовать как одинарные, так и двойные кавычки:

```
message = "Hello World!"
print(message)  # Hello World!

name = 'Tom'
print(name)  # Tom
```

Если строка длинная, ее можно разбить на части и разместить их на разных строках кода. В этом случае вся строка заключается в круглые скобки, а ее отдельные части - в кавычки:

```
text = ("Laudate omnes gentes laudate "
        "Magnificat in secula ")
print(text)
```

Если же мы хотим определить многострочный текст, то такой текст заключается в тройные двойные или одинарные кавычки:

```
'''
Это комментарий
'''
text = '''Laudate omnes gentes laudate
Magnificat in secula
Et anima mea laudate
Magnificat in secula
'''
print(text)
```

При использовании тройных одинарных кавычек не стоит путать их с комментариями: если текст в тройных одинарных кавычках присваивается переменной, то это строка, а не комментарий.

1.3.1 Управляющие последовательности в строке

Строка может содержать ряд специальных символов - управляющих последовательностей или escape-последовательности. Некоторые из них:

- \: позволяет добавить внутрь строки слеш
- \': позволяет добавить внутрь строки одинарную кавычку
- \": позволяет добавить внутрь строки двойную кавычку
- \n: осуществляет переход на новую строку
- \t: добавляет табуляцию (4 отступа)

Используем некоторые последовательностей:

```
text = "Message:\n\"Hello World\""
print(text)
```

Консольный вывод программы:

```
Message:
"Hello World"
```

Хотя подобные последовательности могут нам помочь в некоторых делах, например, поместить в строку кавычку, сделать табуляцию, перенос на другую строку. Но они также могут и мешать. Например:

```
path = "C:\python\name.txt"
print(path)
```

Здесь переменная path содержит некоторый путь к файлу. Однако внутри строки встречаются символы "\n", которые будут интерпретированы как

управляющая последовательность. Так, мы получим следующий консольный вывод:

```
C:\python
ame.txt
```

Чтобы избежать подобной ситуации, перед строкой ставится символ r:

```
path = r"C:\python\name.txt"
print(path)
```

1.3.2 Вставка значений в строку

Python позволяет встраивать в строку значения других переменных. Для этого внутри строки переменные размещаются в фигурных скобках {}, а перед всей строкой ставится символ f:

```
userName = "Tom"
userAge = 37
user = f"name: {userName}  age: {userAge}"
print(user)   # name: Tom  age: 37
```

В данном случае на место {userName} будет вставляться значение переменной userName. Аналогично на вместо {userAge} будет вставляться значение переменной userAge.

1.3.3 Обращение к символам строки

И мы можем обратиться к отдельным символам строки по индексу в квадратных скобках:

```
string = "hello world"
c0 = string[0]  # h
```

```
    print(c0)
    c6 = string[6]  # w
    print(c6)
    c11 = string[11]  # ошибка IndexError: string index
out of range
    print(c11)
```

Индексация начинается с нуля, поэтому первый символ строки будет иметь индекс 0. А если мы попытаемся обратиться к индексу, которого нет в строке, то мы получим исключение IndexError. Например, в случае выше длина строки 11 символов, поэтому ее символы будут иметь индексы от 0 до 10.

Чтобы получить доступ к символам, начиная с конца строки, можно использовать отрицательные индексы. Так, индекс -1 будет представлять последний символ, а -2 - предпоследний символ и так далее:

```
    string = "hello world"
    c1 = string[-1]  # d
    print(c1)
    c5 = string[-5]  # w
    print(c5)
```

При работе с символами следует учитывать, что строка - это неизменяемый (immutable) тип, поэтому если мы попробуем изменить какой-то отдельный символ строки, то мы получим ошибку, как в следующем случае:

```
    string = "hello world"
    string[1] = "R"
```

Мы можем только полностью переустановить значение строки, присвоив ей другое значение.

1.3.4 Получение подстроки

При необходимости мы можем получить из строки не только отдельные символы, но и подстроку. Для этого используется следующий синтаксис:

- string[:end]: извлекается последовательность символов начиная с 0-го индекса по индекс end (не включая)
- string[start:end]: извлекается последовательность символов начиная с индекса start по индекс end (не включая)
- string[start:end:step]: извлекается последовательность символов начиная с индекса start по индекс end (не включая) через шаг step

Используем все варианты получения подстроки:

```
string = "hello world"

# с 0 до 5 индекса
sub_string1 = string[:5]
print(sub_string1)      # hello

# со 2 до 5 индекса
sub_string2 = string[2:5]
print(sub_string2)      # llo

# с 2 по 9 индекса через один символ
sub_string3 = string[2:9:2]
print(sub_string3)      # lowr
```

Получение среза строки от позиции begin до позиции end с шагом step. Любой параметр может быть пропущен:

```
>>> s = "abcdef"
>>> s[1:]
'bcdef'
>>> s[:-1]
'abcde'
>>> s[1:-1]
'bcde'
>>> s[1::2]
'bdf'
>>> s[::2]
'ace'
>>> s[::-2]
'fdb'
```

1.3.5 Операции со строками

Получение длины строки. Для получения длины строки можно использовать функцию len():

```
string = "hello world"
length = len(string)
print(length)   # 11
```

Конкатенация строк. Одной из самых распространенных операций со строками является их объединение или конкатенация. Для объединения строк применяется операция сложения:

```
name = "Tom"
surname = "Smith"
fullname = name + " " + surname
print(fullname)  # Tom Smith
```

С объединением двух строк все просто, но что, если нам надо сложить строку и число? В этом случае необходимо привести число к строке с помощью функции str():

```
name = "Tom"
age = 33
info = "Name: " + name + " Age: " + str(age)
print(info)  # Name: Tom Age: 33
```

Повторение строки n раз. Для повторения строки определенное количество раз применяется операция умножения:

```
print("a" * 3)  # aaa
print("he" * 4)  # hehehehe
```

1.3.6 Сравнение строк

Особо следует сказать о сравнении строк. При сравнении строк принимается во внимание символы и их регистр. Так, цифровой символ условно меньше, чем любой алфавитный символ. Алфавитный символ в верхнем регистре условно меньше, чем алфавитные символы в нижнем регистре. Например:

```
    str1 = "1a"
    str2 = "aa"
    str3 = "Aa"
    print(str1 > str2)   # False, так как первый символ в
str1 - цифра
    print(str2 > str3)   # True, так как первый символ в
str2 - в нижнем регистре
```

Поэтому строка "1a" условно меньше, чем строка "aa". Вначале сравнение идет по первому символу. Если начальные символы обоих строк представляют цифры, то меньшей считается меньшая цифра, например, "1a" меньше, чем "2a".

Если начальные символы представляют алфавитные символы в одном и том же регистре, то смотрят по алфавиту. Так, "aa" меньше, чем "ba", а "ba" меньше, чем "ca".

Если первые символы одинаковые, в расчет берутся вторые символы при их наличии.

Зависимость от регистра не всегда желательна, так как, по сути, мы имеем дело с одинаковыми строками. В этом случае перед сравнением мы можем привести обе строки к одному из регистров.

Функция lower() приводит строку к нижнему регистру, а функция upper() - к верхнему.

```
    str1 = "Tom"
    str2 = "tom"
    print(str1 == str2)  # False - строки не равны
```

```
print(str1.lower() == str2.lower())  # True
```

1.3.7 Поиск в строке

С помощью выражения term in string можно найти подстроку term в строке string. Если подстрока найдена, то выражение вернет значение True, иначе возвращается значение False:

```
string = "hello world"
exist = "hello" in string
print(exist)    # True

exist = "sword" in string
print(exist)    # False
```