Глава 6. Строки

6.1. Строка - это последовательность

Строка - это последовательность символов. Вы можете получить доступ к одному символу с помощью оператора скобок:

```
>>> fruit = 'banana'
>>> letter = fruit[1]
```

Вторая инструкция извлекает символ из позиции 1, хранящейся в переменной *fruit* и помещает его в переменную *letter*.

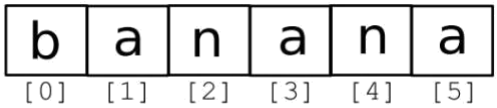
Выражение в скобках называется индексом (index). Индекс обозначает, какой символ в последовательности вы хотите получить:

```
>>> print letter
a
```

Для большинства людей, первая буква в *'banana'* это *b*, а не *a*. Но в Python индекс - это смещение от начала строки, смещение для первого символа – нуль:

```
>>> letter = fruit[0]
>>> print letter
b
```

Таким образом, *b* - это нулевая буква *'banana'*, *a* - это первая буква и т.д.



Вы можете использовать любое выражение, включая переменные или операторы, в качестве индекса, но значение индекса должно быть целочисленным (integer):

```
>>> letter = fruit[1.5]
TypeError: string indices must be integers
```

6.2. Получение длины строки с использованием len *len* - встроенная функция, которая возвращает число символов в строке:

```
>>> fruit = 'banana'
>>> len(fruit)
6
```

Чтобы получить последний символ в строке можно попробовать следующее:

```
>>> length = len(fruit)
>>> last = fruit[length]
IndexError: string index out of range
```

Причина *IndexError* в том, что не существует буквы с индексом 6 в строке *'banana'*, поэтому исправленный вариант:

```
>>> last = fruit[length-1]
>>> print last
a
```

Взамен можно использовать отрицательные индексы, которые считаются с конца. Выражение *fruit[-1]* выдаст последний символ, *fruit[-2]* - второй с конца строки и т.д.

6.3. Обход через строку с помощью цикла

Множество вычислений включают посимвольную обработку строк. Часто они начинаются с начала строки, выбирают каждый символ по очереди, делают что-то с ним и продолжают до окончания строки. Этот шаблон обработки называется *обход* (traversal). Один из вариантов написания обхода с помощью цикла *while*:

```
>>> index = 0
>>> while index < len(fruit):
     letter = fruit[index]
     print letter
     index = index + 1


b
a
n
a
n
a
```

Этот цикл обходит строку и отображает каждый символ строки отдельно. Условием цикла является *index < len(fruit)*, т.е. когда *index* становится равным длине строки, условие становится *ложным* (false) и тело цикла прекращает выполняться.

Другой способ написания обхода с помощью цикла *for*:

```
>>> for char in
     fruit: print char
```

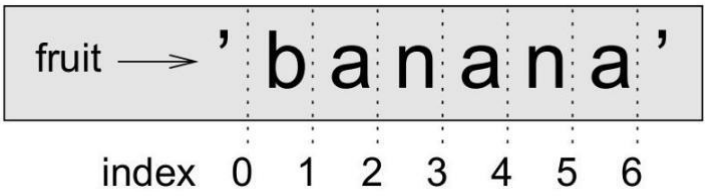
Каждый раз, в цикле следующий символ в строке присваивается переменной *char*. Цикл работает, пока не закончатся символы.

6.4. Срез строки

Часть строки называется *срезом* (slice). Выбор среза аналогичен выбору символа:

```
>>> s = 'Monty Python'
>>> print
s[0:5] Monty
>>> print s[6:13]
Python
>>>
```

Оператор *[n:m]* возвращает часть строки от *n*-ого символа до *m*-ого символа, включая первый, но исключая последний. Это противоречивое поведение, но оно позволяет представить индексные указатели между символами, как на следующей диаграмме:



Если опустить первый индекс (перед двоеточием), то срез будет начинаться с начала строки. Если вы опустите второй индекс - срез завершится в конце строки:

```
>>> fruit = 'banana'
>>> fruit[:3]
'ban'
```

```
>>> fruit[3:]
'ana'
```

Если первый индекс больше или равен второму, то результатом будет *пустая строка* (empty string), представленная двумя кавычками:

```
>>> fruit = 'banana'
>>> fruit[3:3]
''
```

6.5. Строки являются неизменяемыми

Заманчиво использовать оператор [ ] с левой стороны при присвоении с намерением изменить символ в строке. Например:

```
>>> greeting = 'Hello, world!'
>>> greeting[0] = 'J'
TypeError: object does not support item assignment
```

Объект (object) в этом случае является строкой, *элемент* (item) - символ, который пытаемся изменить. На данный момент будем считать, что объект - некоторая вещь как переменная, мы уточним это определение позже. Элемент - одно из значений последовательности.

Причина ошибки в том, что строки являются *неизменяемыми* (immutable), поэтому вы не можете изменить существующую строку. Лучшее, что можно сделать - создать новую строку, которая является вариантом оригинальной:

```
>>> greeting = 'Hello, world!'
>>> new_greeting = 'J' + greeting[1:]
>>> print new_greeting
Jello, world!
```

Этот пример соединяет новую первую букву со срезом *greeting*, это не влияет на оригинальную строку.

6.6. Циклы и счет

Следующая программа подсчитывает количество вхождений буквы *a* в строку:

```
>>> word = 'banana'
>>> count = 0
>>> for letter in word:
     if letter == 'a':
          count = count + 1


>>> print count
3
```

Эта программа демонстрирует другой шаблон вычислений под названием - *счетчик* (counter). Переменной *count* присваивается нулевое начальное значение, затем это значение *инкрементируется* (увеличивается на 1) всякий раз, когда встречается символ *a*. Когда цикл завершается, в *count* содержится результат подсчета.

6.7. Оператор in

Слово *in* - это логический оператор, который принимает две строки и возвращает *True*, если первая строка является подстрой во второй строке:

```
>>> 'a' in 'banana'
True
>>> 'seed' in 'banana'
False
```

6.8. Сравнение строк

Оператор сравнения работает для строк. Рассмотрим вариант, если две строки *одинаковые* (equal):

```
>>> word = 'banana'
>>> if word == 'banana':
     print 'All right, bananas.'
```

Другие операции сравнения полезны для ввода слова в алфавитном порядке:

```
word = 'banan'

if word < 'banana':
    print 'Your word, ' + word + ', comes before banana.'
elif word > 'banana':
    print 'Your word, ' + word + ', comes after
banana.' else:
    print 'All right, bananas.'
```

Python не обрабатывает буквы верхнего и нижнего регистра так же как это делают люди. Все буквы верхнего регистра идут перед буквами нижнего регистра, так:

```
Your word, Pineapple, comes before banana.
```

Общий подход к решению этой проблемы - преобразовать строку в стандартный формат, такой как нижний регистр, перед выполнением сравнения. Имейте это в виду, чтобы защитить себя от вооруженного Ананасом человека (Pineapple).

6.9. Строковые методы

Строка - пример объекта в Python. Объекты содержат данные (фактически саму строку) и методы, которые являются функциями, встроенными в объект и доступными для любого *экземпляра* (instance) объекта.

В Python есть функция *dir*, которая выводит список доступных методов для объекта. Функция *type* показывает тип объекта:

```
>>> stuff = 'Hello world'
>>> type(stuff)
<type 'str'>
>>> dir(stuff)
['capitalize', 'center', 'count', 'decode', 'encode',
'endswith', 'expandtabs', 'find', 'format', 'index',
'isalnum', 'isalpha', 'isdigit', 'islower', 'isspace',
'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip',
'partition', 'replace', 'rfind', 'rindex', 'rjust',
'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines',
'startswith', 'strip', 'swapcase', 'title', 'translate',
'upper', 'zfill']
>>> help(str.capitalize)
Help on method_descriptor:
capitalize(...)
S.capitalize() -> string
Return a copy of the string S with only its first character
capitalized.
>>>
```

Вы можете воспользоваться функцией *help*, чтобы получить дополнительную информацию о методе. Лучший источник документации по строковым методам находится тут: docs.python.org/library/string.html

Метод вызывается подобно функции - он принимает на вход аргументы и возвращает значение, но различается синтаксис. Мы вызываем метод путем добавления имени метода к имени переменной, используя точку в качестве разделителя.

Например, метод *upper* получает на вход строку и возвращает новую строку со всеми буквами в верхнем регистре:

```
>>> word = 'banana'
>>> new_word = word.upper()
>>> print new_word
BANANA
```

Пустые круглые скобки означают, что метод не имеет аргументов. Вызов метода называется *вызовом* (invocation), в этом случае мы можем сказать, что вызвали метод *upper* объекта *word*.

Рассмотрим строковый метод *find*:

```
>>> word = 'banana'
>>> index = word.find('a')
>>> print index
1
>>>
```

В этом примере мы вызвали метод *find* объекта *word* и передали в качестве входного параметра букву, которую ищем. Метод *find* может искать подстроки, а не только отдельные символы:

```
>>> word.find('na')
2
```

В качестве второго аргумента метод *find* принимает индекс, с которого начинается поиск вхождения:

```
>>> word.find('na', 3)
4
```

Одной из распространенных задач является устранение пробелов (пробелов, табуляции или символов перевода строки) с самого начала и до конца строки с помощью метода *strip*:

```
>>> line = ' Here we go '
>>> line.strip()
'Here we go'
```

Некоторые методы, такие как *startswith* возвращают логические значения:

```
>>> line = 'Please have a nice day'
>>> line.startswith('Please')
True
>>> line.startswith('p')
False
```

Предварительно применим метод *lower*, переводящий строку в нижний регистр:

```
>>> line = 'Please have a nice day'
>>> line.startswith('p')
False
>>> line.lower() 'please
have a nice day'
>>> line.lower().startswith('p')
True
```

В последнем примере мы вызвали в одном выражении подряд два метода.

6.10. Разбор (parsing) строк

Часто мы хотим заглянуть в строку и найти подстроку. Для примера, если мы имеем несколько строк, отформатированных следующим образом:

*From stephen.marquard@**uct.ac.za** Sat Jan 5 09:14:16 200*

И мы хотим вытащить только вторую часть адреса (например, uct.ac.za) из каждой строки. Это можно сделать с использованием метода *find* и строкового среза.

Во-первых, мы находим положение at-символа (@) в строке. Затем находим позицию первого пробела после at-символа. После этого с использованием строкового среза извлекаем часть строки, которую ищем:

```
>>> data = 'From stephen.marquard@uct.ac.za Sat Jan 5

09:14:16 2008'
>>> atpos = data.find('@')
>>> print atpos
21
>>> sppos = data.find(' ',atpos)
>>> print sppos
31
>>> host = data[atpos+1:sppos]
>>> print host
uct.ac.za
```

Мы используем версию метода *find*, которая позволяет нам задать позицию в строке, начиная с которой хотим выполнять поиск.

6.11. Оператор форматирования

Оператор форматирования *%* позволяет создавать строки, заменяя часть строки с данными, хранящимися в переменных. Когда он применяется к целым числам - это операция взятия по модулю.

Первым операндом является *строка форматирования* (format string), которая содержит одну или более *последовательностей форматирования* (format sequences), которые определяют форматирование второго оператора. Результатом является строка.

Например, последовательность форматирования '*%d*' означает, что второй оператор должен быть отформатирован как целочисленное значение (*d* от decimal):

```
>>> camels = 42
>>> '%d' % camels
'42'
```

Результатом является строка '42', которую не следует путать с целочисленным значением 42.

Последовательность форматирования может появиться в любой части строки, поэтому возможно встраивать последовательность в предложение:

```
>>> camels = 42
>>> 'I have spotted %d camels.' %
camels 'I have spotted 42 camels.'
```

Если есть более чем одна форматируемая последовательность в строке, второй аргумент должен быть *кортежем* (tuple). Каждая форматируемая последовательность сочетается по порядку с элементом кортежа.

В следующем примере используется '*%d*' форматирование для целочисленных значений, '*%g*' - для чисел с плавающей точкой (не спрашивайте, почему), и '*%s*' - для форматирования строки:

```
>>> 'In %d years I have spotted %g %s.' % (3, 0.1,
'camels') 'In 3 years I have spotted 0.1 camels.'
```

Количество элементов в кортеже должно совпадать с количеством форматируемых последовательностей в строке. Кроме того, типы элементов должны соответствовать форматируемым последовательностям:

```
>>> '%d %d %d' % (1, 2)
TypeError: not enough arguments for format string
>>> '%d' % 'dollars'
TypeError: illegal argument type for built-in operation
```

Более подробно об операторе форматирования:
docs.python.org/lib/typesseq-strings.html

6.13. Словарь

Счетчик (counter): A variable used to count something, usually initialized to zero and then incremented.

Пустая строка (empty string): A string with no characters and length 0, represented by two quotation marks.

Оператор форматирования (format operator): An operator, %, that takes a format string and a tuple and generates a string that includes the elements of the tuple formatted as specified by the format string.

Последовательность форматирования (format sequence): A sequence of characters in a format string, like %d, that specifies how a value should be formatted.

Строка форматирования (format string): A string, used with the format operator, that contains format sequences.

Флаг (flag): A boolean variable used to indicate whether a condition is true.

Вызов (invocation): A statement that calls a method.

Неизменяемый (immutable): The property of a sequence whose items cannot be assigned.

Индекс (index): An integer value used to select an item in a sequence, such as a character in a string.

Элемент (item): One of the values in a sequence.

method: A function that is associated with an object and called using dot notation.

Объект (object): Something a variable can refer to. For now, you can use “object” and “value” interchangeably.

Поиск (search): A pattern of traversal that stops when it finds what it is looking for.

Последовательность (sequence): An ordered set; that is, a set of values where each value is identified by an integer index.

Срез (slice): A part of a string specified by a range of indices.

Обход (traverse): To iterate through the items in a sequence, performing a similar operation on each.

6.14. Упражнение

1. Перечислите встроенные функции для работы со строками в Python.