Python. Лекция 6. Обработка текстов. Регулярные выражения. Unicode.

Под **обработкой текстов** понимается анализ, преобразование, поиск, порождение текстовой информации. По большей части работа с естественными текстами не будет глубже, чем это возможно без систем искусственного интеллекта. Кроме того, здесь предполагается опустить рассмотрение обработки текстов посредством текстовых процессоров и редакторов, хотя некоторые из них (например, Cooledit) предоставляют возможность писать макрокоманды на Python.

Следует отметить, что для Python созданы также модули для работы с естественными языками, а также для лингвистических исследований. Хорошим учебным примером может служить `nltk` (the Natural Language Toolkit).

Стоит отметить проект PyParsing (http://pyparsing.sourceforge.net), с помощью которого можно организовать обработку текста по заданной грамматике.

Строки

Строки в языке Python являются типом данных, специально предназначенным для обработки текстовой информации. Строка может содержать произвольно длинный текст (ограниченный имеющейся памятью).

В новых версиях Python имеются два типа строк: обычные строки (последовательность байтов) и Unicode-строки (последовательность символов). В Unicode-строке каждый символ может занимать в памяти 2 или 4 байта, в зависимости от настроек периода компиляции. Четырехбайтовые знаки используются в основном для восточных языков.

Примечание:

В языке и стандартной библиотеке за некоторыми исключениями строки и Unicode-строки взаимозаменяемы, в собственных приложениях для совместимости с обоими видами строк следует избегать проверок на тип. Если это необходимо, можно проверять принадлежность базовому (для строк и Unicode-строк) типу с помощью `isinstance(s, basestring)`.

При использовании Unicode-строк, следует мысленно принять точку зрения, относительно которой именно Unicode-представление является главным, а все остальные кодировки - лишь частные случаи представления текста, которые не могут передать всех символов. Без такой установки будет непонятно, почему преобразование из восьмибитной кодировки называется `decode` (декодирование). Для внешнего представления можно с успехом использовать кодировку UTF-8, хотя, конечно, это зависит от решаемых задач.

Кодировка Python-программы

Для того чтобы Unicode-литералы в Python-программе воспринимались интерпретатором правильно, необходимо указать кодировку в начале программы, записав в первой или второй строке примерно следующее (для Unix/Linux):

```
# -*- coding: koi8-r -*-
```

или (под Windows):

```
# -*- coding: cp1251 -*-
```

Могут быть и другие варианты:

```
# -*- coding: latin-1 -*-
# -*- coding: utf-8 -*-
# -*- coding: mac-cyrillic -*-
# -*- coding: iso8859-5 -*-
```

Полный перечень кодировок (и их псевдонимов):

```
>>> import encodings.aliases
>>> print encodings.aliases.aliases
{'iso_ir_6': 'ascii', 'maccyrillic': 'mac_cyrillic',
'iso_celtic': 'iso8859_14', 'ebcdic_cp_wt': 'cp037',
'ibm500': 'cp500', ...
```

Если кодировка не указана, то считается, что используется `us-ascii`. При этом интерпретатор Python будет выдавать предупреждения при запуске модуля:

```
sys:1: DeprecationWarning: Non-ASCII character '\xf0' in file example.py
on line 2, but no encoding declared;
see http://www.python.org/peps/pep-0263.html for details
```

Строковые литералы

Строки можно задать в программе с помощью **строковых литералов**. Литералы записываются с использованием апострофов `'`, кавычек `"` или этих же символов, взятых трижды. Внутри литералов обратная косая черта имеет специальное значение. Она служит для ввода специальных символов и для указания символов через коды. Если перед строковым литералом поставлено `r`, обратная косая черта не имеет специального значения (`r` от английского слова raw, строка задается "как есть"). Unicode-литералы задаются с префиксом `u`. Вот несколько примеров:

```
s1 = "строка 1"
s2 = r'\1\2'
s3 = """apple\ntree"""  # \n - символ перевода строки
s4 = """apple
tree"""  # строка в утроенных кавычках может иметь внутри переводы строк
s5 = '\x73\65'
u1 = u"Unicode literal"
u2 = u'\u0410\u0434\u0440\u0435\u0441'
```

Примечание:

Обратная косая черта не должна быть последним символом в литерале, то есть, `"str\"` вызовет синтаксическую ошибку.

Указание кодировки позволяет применять в Unicode-литералах указанную в начале программы кодировку. Если кодировка не указана, можно пользоваться только кодами символов, заданными через обратную косую черту.

Операции над строками

К операциям над строками, которые имеют специальную синтаксическую поддержку в языке, относятся, в частности конкатенация (склеивание) строк, повторение строки, форматирование:

```
>>> print "A" + "B",  "A"*5,  "%s" % "A"
AB AAAAA A
```

В операции форматирования левый операнд является **строкой формата**, а правый может быть либо кортежем, либо словарем, либо некоторым значением другого типа:

```
>>> print "%i" % 234
234
>>> print "%i %s %3.2f" % (5, "ABC", 23.45678)
5 ABC 23.46
>>> a = 123
>>> b = [1, 2, 3]
>>> print "%(a)i: %(b)s" % vars()
123: [1, 2, 3]
```

Операция форматирования

В строке формата кроме текста могут употребляться спецификации, регламентирующие формат выводимого значения. Спецификация имеет синтаксис

```
"%" [ключ][флаг*][шир][.точность][длина_типа]спецификатор
ключ: "(" символ за исключением круглых скобок* ")"
флаг: "+" | "-" | пробел | "#" | "0"
шир: ("1" ... "9")("0" ... "9")* | "*"
точность: ("1" ... "9")* | "*"
длина_типа: "a" ... "z" | "A" ... "Z"
спецификатор: "a" ... "z" | "A" ... "Z" | "%"
```

Где символы обозначают следующее:

`ключ`
Ключ из словаря.

`флаги`
Дополнительные свойства преобразования.

`шир`
Минимальная ширина поля.

`точность`
Точность (для чисел с плавающей запятой).

`длина_типа`
Модификатор типа.

`спецификатор`
Тип представления выводимого объекта.

В следующей таблице приведены некоторые наиболее употребительные значения для спецификации форматирования.

Символ	Где применяется	Что указывает
0	`флаг`	Заполнение нулями слева
-	`флаг`	Выравнивание по левому краю
+	`флаг`	Обязательный вывод знака числа

пробел	флаг	Использовать пробел на месте знака числа
d, i	спецификатор	Знаковое целое
u	спецификатор	Беззнаковое целое
o	спецификатор	Восьмеричное беззнаковое целое
x, X	спецификатор	Шестнадцатеричное беззнаковое целое (со строчными или прописными латинскими буквами)
e, E	спецификатор	Число с плавающей запятой в формате с экспонентой
f, F	спецификатор	Число с плавающей запятой
g, G	спецификатор	Число с плавающей точкой в более коротком написании (автоматически выбирается e или f)
c	спецификатор	Одиночный символ (целое число или односимвольная строка)
r	спецификатор	Любой объект, приведенный к строке функцией `repr()`
s	спецификатор	Любой объект, приведенный к строке функцией `str()`
%	спецификатор	Знак процента. Для задания одиночного процента необходимо записать %%

Индексы и срезы

Следует напомнить, что строки являются неизменчивыми последовательностями, поэтому к ним можно применять операции взятия элемента по индексу и срезы:

```
>>> s = "транспорт"
>>> print s[0], s[-1]
т т
>>> print s[-4:]
порт
>>> print s[:5]
транс
>>> print s[4:8]
спор
```

Примечание:

При выделении среза нумеруются не символы строки, а промежутки между ними.

Модуль string

До того как у строк появились методы, для операций над строками применялся модуль `string`. Приведенный пример демонстрирует, как вместо функции из `string` использовать метод (кстати, последнее более эффективно):

```
>>> import string
>>> s = "one,two,three"
>>> print string.split(s, ",")
['one', 'two', 'three']
>>> print s.split(",")
['one', 'two', 'three']
```

В версии Python 3.0 функции, которые доступны через методы, более не будут дублироваться в модуле `string`.

В Python 2.4 появилась альтернатива использованию операции форматирования: класс `Template`. Пример:

```
>>> import string
```

```
>>> tpl = string.Template("$a + $b = ${c}")
>>> a = 2
>>> b = 3
>>> c = a + b
>>> print tpl.substitute(vars())
2 + 3 = 5
>>> del c  # удаляется имя c
>>> print tpl.safe_substitute(vars())
2 + 3 = $c
>>> print tpl.substitute(vars(), c=a+b)
2 + 3 = 5
>>> print tpl.substitute(vars())
Traceback (most recent call last):
  File "/home/rnd/tmp/Python-2.4b2/Lib/string.py", line 172, in substitute
    return self.pattern.sub(convert, self.template)
  File "/home/rnd/tmp/Python-2.4b2/Lib/string.py", line 162, in convert
    val = mapping[named]
KeyError: 'c'
```

Объект-шаблон имеет два основных метода: `substitute()` и `safe_substitute()`. Значения для подстановки в шаблон берутся из словаря (`vars()` содержит словарь со значениями переменных) или из именованных фактических параметров. Если есть неоднозначность в задании ключа, можно использовать фигурные скобки при написании ключа в шаблоне.

Методы строк

В таблице ниже приведены некоторые наиболее употребительные методы объектов-строк и unicode-объектов.

Метод	Описание
`center(w)`	Центрирует строку в поле длины `w`
`count(sub)`	Число вхождений строки `sub` в строке
`encode([enc[, errors]])`	Возвращает строку в кодировке `enc`. Параметр `errors` может принимать значения `"strict"` (по умолчанию), `"ignore"`, `"replace"` или `"xmlcharrefreplace"`
`endswith(suffix)`	Оканчивается ли строка на `suffix`
`expandtabs([tabsize])`	Заменяет символы табуляции на пробелы. По умолчанию `tabsize=8`
`find(sub [,start [,end]])`	Возвращает наименьший индекс, с которого начинается вхождение подстроки `sub` в строку. Параметры `start` и `end` ограничивают поиск окном `start:end`, но возвращаемый индекс соответствует исходной строке. Если подстрока не найдена, возвращается `-1`
`index(sub[, start[, end]])`	Аналогично `find()`, но возбуждает исключение `ValueError` в случае неудачи
`isalnum()`	Возвращает `True`, если строка содержит только буквы и цифры и имеет ненулевую длину. Иначе -- `False`
`isalpha()`	Возвращает `True`, если строка содержит только буквы и длина ненулевая
`isdecimal()`	Возвращает `True`, если строка содержит только десятичные знаки (только для строк Unicode) и длина ненулевая
`isdigit()`	Возвращает `True`, если содержит только цифры и длина ненулевая
`islower()`	Возвращает `True`, если все буквы строчные (и их более одной), иначе -- `False`

isnumeric()	Возвращает True, если в строке только числовые знаки (только для Unicode)
isspace()	Возвращает True, если строка состоит только из пробельных символов. Внимание! Для пустой строки возвращается False
join(seq)	Соединение строк из последовательности seq через разделитель, заданный строкой
lower()	Приводит строку к нижнему регистру букв
lstrip()	Удаляет пробельные символы слева
replace(old, new[, n])	Возвращает копию строки, в которой подстроки old заменены new. Если задан параметр n, то заменяются только первые n вхождений
rstrip()	Удаляет пробельные символы справа
split([sep[, n]])	Возвращает список подстрок, получающихся разбиением строки a разделителем sep. Параметр n определяет максимальное количество разбиений (слева)
startswith(prefix)	Начинается ли строка с подстроки prefix
strip()	Удаляет пробельные символы в начале и в конце строки
translate(table)	Производит преобразование с помощью таблицы перекодировки table, содержащей словарь для перевода кодов в коды (или в None, чтобы удалить символ). Для Unicode-строк
translate(table[, dc])	То же, но для обычных строк. Вместо словаря - строка перекодировки на 256 символов, которую можно сформировать с помощью функции string.maketrans(). Необязательный параметр dc задает строку с символами, которые необходимо удалить
upper()	Переводит буквы строки в верхний регистр

В следующем примере применяются методы split() и join() для разбиения строки в список (по разделителям) и обратное объединение списка строк в строку

```
>>> s = "This is an example."
>>> lst = s.split(" ")
>>> print lst
['This', 'is', 'an', 'example.']
>>> s2 = "\n".join(lst)
>>> print s2
This
is
an
example.
```

Для проверки того, оканчивается ли строка на определенное сочетание букв, можно применить метод endswith():

```
>>> filenames = ["file.txt", "image.jpg", "str.txt"]
>>> for fn in filenames:
...   if fn.lower().endswith(".txt"):
...     print fn
...
file.txt
str.txt
```

Поиск в строке можно осуществить с помощью метода find(). Следующая программа выводит все функции, определенные в модуле оператором def:

```
import string
text = open(string.__file__[:-1]).read()
```

```
start = 0
while 1:
  found = text.find("def ", start)
  if found == -1:
    break
  print text[found:found + 60].split("(")[0]
  start = found + 1
```

Важным для преобразования текстовой информации является метод `replace()`, который рассматривается ниже:

```
>>> a = "Это текст , в котором встречаются запятые , поставленные не так."
>>> b = a.replace(" ,", ",")
>>> print b
Это текст, в котором встречаются запятые, поставленные не так.
```

Рекомендации по эффективности

При работе с очень длинными строками или большим количеством строк, применяемые операции могут по-разному влиять на быстродействие программы.

Например, не рекомендуется многократно использовать операцию конкатенации для склеивания большого количества строк в одну. Лучше накапливать строки в списке, а затем с помощью `join()` собирать в одну строку:

```
>>> a = ""
>>> for i in xrange(1000):
...   a += str(i)        # неэффективно!
...
>>> a = "".join([str(i) for i in xrange(1000)])  # более эффективно
```

Конечно, если строка затем обрабатывается, можно применять итераторы, которые позволят свести использование памяти к минимуму.

Модуль StringIO

В некоторых случаях желательно работать со строкой как с файлом. Модуль `StringIO` как раз дает такую возможность.

Открытие "файла" производится вызовом `StringIO()`. При вызове без аргумента - создается новый "файл", при задании строки в качестве аргумента - "файл" открывается для чтения:

```
import StringIO
my_string = "1234567890"
f1 = StringIO.StringIO()
f2 = StringIO.StringIO(my_string)
```

Далее с файлами `f1` и `f2` можно работать как с обычными файловыми объектами.

Для получения содержимого такого файла в виде строки применяется метод `getvalue()`:

```
f1.getvalue()
```

Противоположный вариант (представление файла на диске в виде строки) можно реализовать на платформах Unix и Windows с использованием модуля `mmap`. Здесь этот модуль рассматриваться не будет.

Модуль difflib

Для приблизительного сравнения двух строк в стандартной библиотеке предусмотрен модуль `difflib`.

Функция `difflib.get_close_matches()` позволяет выделить n близких строк к заданной строке:

```
get_close_matches(word, possibilities, n=3, cutoff=0.6)
```
где

`word`
Строка, к которой ищутся близкие строки.

`possibilities`
Список возможных вариантов.

`n`
Требуемое количество ближайших строк.

`cutoff`
Коэффициент (из диапазона [0, 1]) необходимого уровня совпадения строк. Строки, которые при сравнении с `word` дают меньшее значение, игнорируются.

Следующий пример показывает функцию `difflib.get_close_matches()` в действии:

```
>>> import unicodedata
>>> names = [unicodedata.name(unicode(chr(i))) for i in range(40, 127)]
>>> print difflib.get_close_matches("LEFT BRACKET", names)
['LEFT CURLY BRACKET', 'LEFT SQUARE BRACKET']
```
В списке `names` - названия Unicode-символов с ASCII-кодами от 40 до 127.

Регулярные выражения

Рассмотренных стандартных возможностей для работы с текстом достаточно далеко не всегда. Например, в методах `find()` и `replace()` задается всего одна строка. В реальных задачах такая однозначность встречается довольно редко, чаще требуется найти или заменить строки, отвечающие некоторому шаблону.

Регулярные выражения (regular expressions) описывают множество строк, используя специальный язык, который сейчас и будет рассмотрен. (Строка, в которой задано регулярное выражение, будет называться шаблоном.)

Для работы с регулярными выражениями в Python используется модуль `re`. В следующем примере регулярное выражение помогает выделить из текста все числа:

```
>>> import re
>>> pattern = r"[0-9]+"
>>> number_re = re.compile(pattern)
>>> number_re.findall("122 234 65435")
['122', '234', '65435']
```
В этом примере шаблон `pattern` описывает множество строк, которые состоят из одного или более символов из набора `"0"`, `"1"` , ..., `"9"` . Функция `re.compile()` компилирует шаблон в специальный Regex-объект, который имеет несколько методов, в том числе метод `findall()` для получения списка всех непересекающихся вхождений строк, удовлетворяющих шаблону, в заданную строку.

То же самое можно было сделать и так:

```
>>> import re
>>> re.findall(r"[0-9]+", "122 234 65435")
['122', '234', '65435']
```

Предварительная компиляция шаблона предпочтительнее при его частом использовании, особенно внутри цикла.

Примечание:

Следует заметить, что для задания шаблона использована необработанная строка. В данном примере она не требовалась, но в общем случае лучше записывать строковые литералы именно так, чтобы исключить влияние специальных последовательностей, записываемых через обратную косую черту.

Синтаксис регулярного выражения

Синтаксис регулярных выражений в Python почти такой же, как в Perl, grep и некоторых других инструментах. Часть символов (в основном буквы и цифры) обозначают сами себя. Строка **удовлетворяет (соответствует)** шаблону, если она входит во множество строк, которые этот шаблон описывает.

Здесь стоит также отметить, что различные операции используют шаблон по-разному. Так, `search()` ищет первое вхождение строки, удовлетворяющей шаблону, в заданной строке, а `match()` требует, чтобы строка удовлетворяла шаблону с самого начала.Символы, имеющие специальное значение в записи регулярных выражений:

Символ	Что обозначает в регулярном выражении
`"."`	Любой символ
`"^"`	Начало строки
`"$"`	Конец строки
`"*"`	Повторение фрагмента нуль или более раз (жадное)
`"+"`	Повторение фрагмента один или более раз (жадное)
`"?"`	Предыдущий фрагмент либо присутствует, либо отсутствует
`"{m,n}"`	Повторение предыдущего фрагмента от `m` до `n` раз включительно (жадное)
`"[...]"`	Любой символ из набора в скобках. Можно задавать диапазоны символов с идущими подряд кодами, например: `a-z`
`"[^...]"`	Любой символ не из набора в скобках
`"\"`	Обратная косая черта отменяет специальное значение следующего за ней символа
`"|"`	Фрагмент справа или фрагмент слева
`"*?"`	Повторение фрагмента нуль или более раз (не жадное)
`"+?"`	Повторение фрагмента один или более раз (не жадное)
`"{m,n}?"`	Повторение предыдущего фрагмента от `m` до `n` раз включительно (не жадное)

Если A и B - регулярные выражения, то их конкатенация AB является новым регулярным выражением, причем конкатенация строк a и b будет удовлетворять AB, если a удовлетворяет A и b удовлетворяет B. Можно считать, что конкатенация - основной способ составления регулярных выражений.

Скобки, описанные ниже, применяются для задания приоритетов и выделения групп (фрагментов текста, которые потом можно получить по номеру или из словаря, и даже сослаться в том же регулярном выражении).

Алгоритм, который сопоставляет строки с регулярным выражением, проверяет соответствие того или иного фрагмента строки регулярному выражению. Например, строка "a" соответствует регулярному выражению "[a-z]", строка "fruit" соответствует "fruit|vegetable", а вот строка "apple" не соответствует шаблону "pineapple".

В таблице ниже вместо регвыр может быть записано регулярное выражение, вместо имя - идентификатор, а флаги будут рассмотрены ниже.

Обозначение	Описание
"(регвыр)"	Обособляет регулярное выражение в скобках и выделяет группу
"(?:регвыр)"	Обособляет регулярное выражение в скобках без выделения группы
"(?=регвыр)"	Взгляд вперед: строка должна соответствовать заданному регулярному выражению, но дальнейшее сопоставление с шаблоном начнется с того же места
"(?!регвыр)"	То же, но с отрицанием соответствия
"(?<=регвыр)"	Взгляд назад: строка должна соответствовать, если до этого момента соответствует регулярному выражению. Не занимает места в строке, к которой применяется шаблон. Параметр регвыр должен быть фиксированной длины (то есть, без "+" и "*")
"(?<!регвыр)"	То же, но с отрицанием соответствия
"(?P<имя>регвыр)"	Выделяет именованную группу с именем имя
"(?P=имя)"	Точно соответствует выделенной ранее именованной группе с именем имя
"(?#регвыр)"	Комментарий (игнорируется)
"(?(имя)рв1|рв2)"	Если группа с номером или именем имя оказалась определена, результатом будет сопоставление с рв1, иначе - с рв2. Часть |рв2 может отсутствовать
"(?флаг)"	Задает флаг для всего данного регулярного выражения. Флаги необходимо задавать в начале шаблона

В таблице ниже описаны специальные последовательности, использующие обратную косую черту:

Последовательность	Чему соответствует
"\1" - "\9"	Группа с указанным номером. Группы нумеруются, начиная с 1
"\A"	Промежуток перед началом всей строки (почти аналогично "^")
"\Z"	Промежуток перед концом всей строки (почти аналогично "$")
"\b"	Промежуток между символами перед словом или после него
"\B"	Наоборот, не соответствует промежутку между символами на границе слова
"\d"	Цифра. Аналогично "[0-9]"
"\s"	Любой пробельный символ. Аналогично "[\t\n\r\f\v]"
"\S"	Любой непробельный символ. Аналогично "[^\t\n\r\f\v]"
"\w"	Любая цифра или буква (зависит от флага LOCALE)
"\W"	Любой символ, не являющийся цифрой или буквой (зависит от флага LOCALE)

Флаги, используемые с регулярными выражениями:

"(?i)", re.I, re.IGNORECASE
Сопоставление проводится без учета регистра букв.

"(?L)", re.L, re.LOCALE
Влияет на определение буквы в "\w", "\W", "\b", "\B" в зависимости от текущей культурной среды (locale).

"(?m)", re.M, re.MULTILINE
Если этот флаг задан, "^" и "$" соответствуют началу и концу любой строки.

"(?s)", re.S, re.DOTALL
Если задан, "." соответствует также и символу конца строки "\n".

"(?x)", re.X, re.VERBOSE
Если задан, пробельные символы, не экранированные в шаблоне обратной косой чертой, являются незначащими, а все, что расположено после символа "#", -- комментарии. Позволяет записывать регулярное выражение в несколько строк для улучшения его читаемости и записи комментариев.

"(?u)", re.U, re.UNICODE
В шаблоне и в строке использован Unicode.

Методы объекта-шаблона

В результате успешной компиляции шаблона функцией re.compile() получается шаблон-объект (он именуется SRE_Pattern), который имеет несколько методов, некоторые из них будут рассмотрены. Как обычно, подробности и информация о дополнительных аргументах - в документации по Python.

match(s)
Сопоставляет строку s с шаблоном, возвращая в случае удачного сопоставления объект с результатом сравнения (объект SRE_Match). В случае неудачи возвращает None. Сопоставление начинается от начала строки.

search(s)
Аналогичен match(s), но ищет подходящую подстроку по всей строке s.

split(s[, maxsplit=0])
Разбивает строку на подстроки, разделенные подстроками, заданными шаблоном. Если в шаблоне выделены группы, они попадут в результирующий список, перемежаясь с подстроками между разделителями. Если указан maxsplit, будет произведено не более maxsplit разбиений.

findall(s)
Ищет все неперекрывающиеся подстроки s, удовлетворяющие шаблону.

finditer(s)
Возвращает итератор по объектам с результатами сравнения для всех неперекрывающихся подстрок, удовлетворяющих шаблону.

sub(repl, s)
Заменяет в строке s все (или только count, если он задан) вхождения неперекрывающихся подстрок, удовлетворяющих шаблону, на строку, заданную с помощью repl. В качестве repl может выступать строка или функция. Возвращает строку с выполненными заменами. В первом случае строка repl подставляется не просто так, а интерпретируется с заменой вхождений "\номер" на группу с соответствующим номером и вхождений "\g<имя>" на группу с номером или именем

имя. В случае, когда repl - функция, ей передается объект с результатом каждого успешного сопоставления, а из нее возвращается строка для замены.

```
subn(repl, s)
```

Аналогичен sub(), но возвращает кортеж из строки с выполненными заменами и числа замен.

В следующем примере строка разбивается на подстроки по заданному шаблону:

```
>>> import re
>>> delim_re = re.compile(r"[:,;]")
>>> text = "This,is;example"
>>> print delim_re.split(text)
['This', 'is', 'example']
```

А теперь можно узнать, чем именно были разбиты строки:

```
>>> delim_re = re.compile(r"([:,;])")
>>> print delim_re.split(text)
['This', ',', 'is', ';', 'example']
```

Примеры шаблонов

Владение регулярными выражениями может существенно ускорить построение алгоритмов для обработки данных. Лучше всего познакомиться с шаблонами на конкретных примерах:

```
r"\b\w+\b"
```

Соответствует слову из букв и знаков подчеркивания.

```
r"[+-]?\d+"
```

Соответствует целому числу. Возможно, со знаком.

```
r"\([+-]?\d+\)"
```

Число, стоящее в скобках. Скобки используются в самих регулярных выражениях, поэтому они экранируются "\".

```
r"[a-cA-C]{2}"
```

Соответствует строке из двух букв "a", "b" или "c". Например, "Ac", "CC", "bc".

```
r"aa|bb|cc|AA|BB|CC"
```

Строка из двух одинаковых букв.

```
r"([a-cA-C])\1"
```

Строка из двух одинаковых букв, но шаблон задан с использованием групп

```
r"aa|bb".
```

Соответствует "aa" или "bb"

```
r"a(a|b)b"
```

Соответствует "aab" или "abb"

```
r"^(?:\d{8}|\d{4}):\s*(.*)$"
```

Соответствует строке, которая начинается с набора из восьми или четырех цифр и двоеточия. Все, что идет после двоеточия и после следующих за ним пробелов, выделяется в группу с номером 1, тогда как набор цифр в группу не выделен.

```
r"(\w+)=.*\b\1\b"
```

Слова слева и справа от знака равенства присутствуют. Операнд "\1" соответствует группе с номером 1, выделенной с помощью скобок.

```
r"(?P<var>\w+)=.*\b(?P=var)\b"
```
То же самое, но теперь используется именованная группа var.

```
r"\bregular(?=\s+expression)".
```
Соответствует слову "regular" только в том случае, если за ним после пробелов следует "expression"

```
r"(?<=regular )expression"
```
Соответствует слову "expression", перед которым стоит "regular" и один пробел.

Следует заметить, что примеры со взглядом назад могут сильно влиять на производительность, поэтому их не стоит использовать без особой необходимости.

Отладка регулярных выражений

Следующий небольшой сценарий позволяет отлаживать регулярное выражение, при условии, что есть пример строки, которой шаблон должен удовлетворять. Взят кусочек лога iptables, его необходимо разобрать для получения полей. Интересны строки, в которых после kernel: стоит PAY:, а в этих строках нужно получить дату, значения DST, LEN и DPT:

```
import re

def debug_regex(regex, example):
  """Отладка рег. выражения. Перед отладкой лучше убрать лишние скобки """
  last_good = ""
  for i in range(1, len(regex)):
    try:
       if re.compile(regex[:i]).match(example):
          last_good = regex[:i]
    except:
      continue
  return last_good

example = """Nov 27 15:57:59 lap kernel: PAY: IN=eth0 OUT=
 MAC=00:50:da:d9:df:a2:00:00:1c:b0:c9:db:08:00           SRC=192.168.1.200
DST=192.168.1.115
 LEN=1500  TOS=0x00  PREC=0x00  TTL=64  ID=31324  DF  PROTO=TCP  SPT=8080
DPT=1039
 WINDOW=17520 RES=0x00 ACK PSH URGP=0"""

log_re = r"""[A-Za-z]{3}\s+\d+\s+\d\d\d\d:\d\d) \S+ kernel: PAY: .+
 DST=(?P<dst>\S+).* LEN=(?P<len>\d+).* DPT=(?P<dpt>\d+) """

print debug_regex(log_re, example)
```
Функция debug_regex() пробует сопоставлять пример с увеличивающимися порциями регулярного выражения и возвращает последнее удавшееся сопоставление:

```
[A-Za-z]{3}\s+\d+\s+\d\d
```
Сразу видно, что не поставлен символ :.

Примеры применения регулярного выражения

Обработка лога

Предыдущий пример регулярного выражения позволит выделить из лога записи с определенной меткой и подать их в сокращенном виде:

```
import re
log_re  =  re.compile(r"""(?P<date>[A-Za-z]{3}\s+\d+\s+\d\d:\d\d:\d\d)  \S+
kernel:
 PAY: .+ DST=(?P<dst>\S+).* LEN=(?P<len>\d+).* DPT=(?P<dpt>\d+) """)

for line in open("message.log"):
  m = log_re.match(line)
  if m:
     print "%(date)s %(dst)s:%(dpt)s size=%(len)s" % m.groupdict()
```

В результате получается

```
Nov 27 15:57:59 192.168.1.115:1039 size=1500
Nov 27 15:57:59 192.168.1.200:8080 size=40
Nov 27 15:57:59 192.168.1.115:1039 size=515
Nov 27 15:57:59 192.168.1.200:8080 size=40
Nov 27 15:57:59 192.168.1.115:1039 size=40
Nov 27 15:57:59 192.168.1.200:8080 size=40
Nov 27 15:57:59 192.168.1.115:1039 size=40
```

Анализ записи числа

Хороший пример регулярного выражения можно найти в модуле `fpformat`. Это регулярное выражение позволяет разобрать запись числа (в том виде, в каком числовой литерал принято записывать в Python):

```
decoder = re.compile(r'^([-+]?)0*(\d*)((?:\.\d*)?)(([eE][-+]?\d+)?)$')
# Следующие части числового литерала выделяются с помощью групп:
# \0 - весь литерал
# \1 - начальный знак или пусто
# \2 - цифры слева от точки
# \3 - дробная часть (пустая или начинается с точки)
# \4 - показатель (пустой или начинается с 'e' или 'E')
```

Например:

```
import re
decoder = re.compile(r'^([-+]?)0*(\d*)((?:\.\d*)?)((?:[eE][-+]?\d+)?)$')

print decoder.match("12.234").groups()
print decoder.match("-0.23e-7").groups()
print decoder.match("1e10").groups()
```

Получим

```
('', '12', '.234', '')
('-', '', '.23', 'e-7')
('', '1', '', 'e10')
```

Множественная замена

В некоторых приложениях требуется производить в тексте сразу несколько замен. Для решения этой задачи можно использовать метод `sub()` вместе со специальной функцией, которая и будет управлять заменами:

```
import re

def multisub(subs_dict, text):
    def _multisub(match_obj):
        return str(subs_dict[match_obj.group()])
```

```
    multisub_re = re.compile("|".join(subs_dict.keys()))
    return multisub_re.sub(_multisub, text)


repl_dict = {'one': 1, 'two': 2, 'three': 3}

print multisub(repl_dict, "One, two, three")
```

Будет выведено

```
One, 2, 3
```

В качестве упражнения предлагается сделать версию, которая бы не учитывала регистр букв.

В приведенной программе вспомогательная функция `_multisub()` по полученному объекту с результатом сравнения возвращает значение из словаря с описаниями замен `subs_dict`.

Работа с несколькими файлами

Для упрощения работы с несколькими файлами можно использовать модуль `fileinput`. Он позволяет обработать в одном цикле строки всех указанных в командной строке файлов:

```
import fileinput
for line in fileinput.input():
    process(line)
```

В случае, когда файлов не задано, обрабатывается стандартный ввод.

Работа с Unicode

До появления Unicode символы в компьютере кодировались одним байтом (а то и только семью битами). Один байт охватывает диапазон кодов от 0 до 255 включительно, а это значит, что больше двух алфавитов, цифр, знаков пунктуации и некоторого набора специальных символов в одном байте не помещается. Каждый производитель использовал свою кодировку для одного и того же алфавита. Например, до настоящего времени дожили целых пять кодировок букв кириллицы, и каждый пользователь не раз видел в своем браузере или электронном письме пример несоответствия кодировок.

Стандарт Unicode - единая кодировка для символов всех языков мира. Это большое облегчение и некоторое неудобство одновременно. Плюс состоит в том, что в одной Unicode-строке помещаются символы совершенно различных языков. Минус же в том, что пользователи привыкли применять однобайтовые кодировки, большинство приложений ориентировано на них, во многих системах поддержка Unicode осуществляется лишь частично, так как требует огромной работы по разработке шрифтов. Правда, символы одной кодировки можно перевести в Unicode и обратно.

Здесь же следует заметить, что файлы по-прежнему принято считать последовательностью байтов, поэтому для хранения текста в файле в Unicode требуется использовать одну из транспортных кодировок Unicode (utf-7, utf-8, utf-16,...). В некоторых из этих кодировок имеет значение принятый на данной платформе порядок байтов (big-endian, старшие разряды в конце или little-endian, младшие в конце). Узнать порядок байтов можно, прочитав атрибут из модуля `sys`. На платформе Intel это выглядит так:

```
>>> sys.byteorder
'little'
```

Для исключения неоднозначности документ в Unicode может быть в самом начале снабжен BOM (byte-order mark - метка порядка байтов) - Unicode-символом с кодом 0xfeff. Для данной платформы строка байтов для BOM будет такой:

```
>>> codecs.BOM_LE
'\xff\xfe'
```

Для преобразования строки в Unicode необходимо знать, в какой кодировке закодирован текст. Предположим, что это `cp1251`. Тогда преобразовать текст в Unicode можно следующим способом:

```
>>> s = "Строка в cp1251"
>>> s.decode("cp1251")
u'\u0421\u0442\u0440\u043e\u043a\u0430 \u0432 cp1251'
```

То же самое с помощью встроенной функции `unicode()`:

```
>>> unicode(s, 'cp1251')
u'\u0421\u0442\u0440\u043e\u043a\u0430 \u0432 cp1251'
```

Одной из полезных функций этого модуля является функция `codecs.open()`, позволяющая открыть файл в другой кодировке:

```
codecs.open(filename, mode[, enc[, errors[, buffer]]])
```

Здесь:

`filename`
Имя файла.

`mode`
Режим открытия файла

`enc`
Кодировка.

`errors`
Режим реагирования на ошибки кодировки ('strict' - возбуждать исключение, 'replace' - заменять отсутствующие символы, 'ignore' - игнорировать ошибки).

`buffer`
Режим буферизации (0 - без буферизации, 1 - построчно, n - байт буфера).

Заключение

В этой лекции были рассмотрены основные типы для манипулирования текстом: строки и Unicode-строки. Достаточно подробно описаны регулярные выражения - один из наиболее эффективных механизмов для анализа текста. В конце приведены некоторые функции для работы с Unicode.