Python. Лекция 7. Работа с данными в различных форматах.

Формат CSV

Файл в формате CSV (comma-separated values - значения, разделенные запятыми) - универсальное средство для переноса табличной информации между приложениями (электронными таблицами, СУБД, адресными книгами и т.п.). К сожалению, формат файла не имеет строго определенного стандарта, поэтому между файлами, порождаемыми различными приложениями, существуют некоторые тонкие различия. Внутри файл выглядит примерно так (файл `pr.csv`):

```
name,number,text
a,1,something here
b,2,"one, two, three"
c,3,"no commas here"
```

Для работы с CSV-файлами имеются две основные функции:

```
reader(csvfile[, dialect='excel'[, fmtparam]])
```

Возвращает читающий объект, который является итератором по всем строкам заданного файла. В качестве `csvfile` может выступать любой объект, который поддерживает протокол итератора и возвращает строку при обращении к его методу `next()`. Необязательный аргумент `dialect`, по умолчанию равный `'excel'`, указывает на необходимость использования того или иного набора свойств. Узнать доступные варианты можно с помощью `csv.list_dialects()`. Аргумент может быть одной из строк, возвращаемых указанной функцией, либо экземпляром подкласса класса `csv.Dialect`. Необязательный аргумент `fmtparam` служит для переназначения отдельных свойств по сравнению с заданным параметром `dialect` набором. Все получаемые данные являются строками.

```
writer(csvfile[, dialect='excel'[, fmtparam]])
```

Возвращает пишущий объект для записи пользовательских данных с использованием разделителя в заданный файлоподобный объект. Параметры `dialect` и `fmtparam` имеют тот же смысл, что и выше. Все данные, кроме строк, обрабатывают функцией `str()` перед помещением в файл.

В следующем примере читается CSV-файл и записывается другой, где числа второго столбца увеличены на единицу:

```
import csv
input_file = open("pr.csv", "rb")
rdr = csv.reader(input_file)
output_file = open("pr1.csv", "wb")
wrtr = csv.writer(output_file)
for rec in rdr:
  try:
    rec[1] = int(rec[1]) + 1
  except:
    pass
  wrtr.writerow(rec)
input_file.close()
output_file.close()
```

В результате получится файл pr1.csv следующего содержания:

```
name,number,text
a,2,something here
b,3,"one, two, three"
c,4,no commas here
```

Модуль также определяет два класса для более удобного чтения и записи значений с использованием словаря. Вызовы конструкторов следующие:

```
class DictReader(csvfile, fieldnames[, restkey=None[, restval=None[, dialect='excel']]]])
```

Создает читающий объект, подобный тому, что рассматривался выше, но помещающий считываемые значения в словарь. Параметры `csvfile` и `dialect` те же, что и раньше. Параметр `fieldnames` задает имена полей списком. Параметр `restkey` задает значение ключа для помещения списка значений, для которых не хватило имен полей. Параметр `restval` используется как значение в том случае, если в записи не хватает значений для всех полей. Если параметр `fieldnames` не задан, имена полей будут прочитаны из первой записи CSV-файла. Начиная с Python 2.4, параметр `fieldnames` необязателен. Если он отсутствует, ключи берутся из первой строки CSV-файла.

```
class DictWriter(csvfile, fieldnames[, restval=""[, extrasaction='raise'[, dialect='excel']]])
```

Создает пишущий объект, который записывает в CSV-файл строки, получая данные из словаря. Параметры аналогичны `DictReader`, но `fieldnames` обязателен, так как он задает порядок следования полей. Параметр `extrasaction` указывает на то, какое действие нужно произвести в случае, когда требуемого значения нет в словаре: `'raise'` - возбудить исключение `ValueError`, `'ignore'` - игнорировать.

Соответствующий пример дан ниже. В файле `pr.csv` имена полей заданы в первой строке файла, поэтому можно не задавать `fieldnames`:

```
import csv
input_file = open("pr.csv", "rb")
rdr = csv.DictReader(input_file,
                     fieldnames=['name', 'number', 'text'])
output_file = open("pr1.csv", "wb")
wrtr = csv.DictWriter(output_file,
                     fieldnames=['name', 'number', 'text'])
for rec in rdr:
  try:
    rec['number'] = int(rec['number']) + 1
  except:
    pass
  wrtr.writerow(rec)
input_file.close()
output_file.close()
```

Модуль имеет также другие классы и функции, которые можно изучить по документации. На примере этого модуля можно увидеть общий подход к работе с файлом в некотором формате. Следует обратить внимание на следующие моменты:

- Модули для работы с форматами данных обычно содержат функции или конструкторы классов, в частности `Reader` и `Writer`.
- Эти функции и конструкторы возвращают объекты-итераторы для чтения данных из файла и объекты со специальными методами для записи в файл.

- Для разных нужд обычно требуется иметь несколько вариантов классов читающих и пишущих объектов. Новые классы могут получаться наследованием от базовых классов либо обертыванием функций, предоставляемых модулем расширения (написанным на C). В приведенном примере `DictReader` и `DictWriter` являются обертками для функций `reader()` и `writer()` и объектов, которые они порождают.

Пакет email

Модули пакета `email` помогут разобрать, изменить и сгенерировать сообщение в формате RFC 2822. Наиболее часто RFC 2822 применяется в сообщениях электронной почты в Интернете.

В пакете есть несколько модулей, назначение которых (кратко) указано ниже:

`Message`
Модуль определяет класс `Message` - основной класс для представления сообщения в пакете `email`.

`Parser`
Модуль для разбора представленного в виде текста сообщения с получением объектной структуры сообщения.

`Header`
Модуль для работы с полями, в которых используется кодировка, отличная от ASCII.

`Generator`
Порождает текст сообщения RFC 2822 на основании объектной модели.

`Utils`
Различные утилиты, которые решают разнообразные небольшие задачи, связанные с сообщениями.

В пакете есть и другие модули, которые здесь рассматриваться не будут.

Разбор сообщения. Класс Message

Класс `Message` - центральный во всем пакете `email`. Он определяет методы для работы с сообщением, которое состоит из заголовка (header) и тела (payload). Поле заголовка имеет название и значение, разделенное двоеточием (двоеточие не входит ни в название, ни в значение). Названия полей нечувствительны к регистру букв при поиске значения, хотя хранятся с учетом регистра. В классе также определены методы для доступа к некоторым часто используемым сведениям (кодировке сообщения, типу содержимого и т.п.).

Следует заметить, что сообщение может иметь одну или несколько частей, в том числе вложенных друг в друга. Например, сообщение об ошибке доставки письма может содержать исходное письмо в качестве вложения.

Пример наиболее употребительных методов экземпляров класса `Message` с пояснениями:

```
>>> import email
>>> input_file = open("pr1.eml")
>>> msg = email.message_from_file(input_file)
```

Здесь используется функция `email.message_from_file()` для чтения сообщения из файла pr1.eml. Сообщение можно получить и из строки с помощью функции `email.message_from_string()`. А теперь следует произвести некоторые операции над этим сообщением (не стоит обращать внимания на странные имена - сообщение было взято из папки СПАМ). Доступ к полям по имени осуществляется так:

```
>>> print msg['from']
"felton olive" <zinakinch@thecanadianteacher.com>
>>> msg.get_all('received')
['from mail.onego.ru\n\tby localhost with POP3 (fetchmail-6.2.5
polling mail.onego.ru account spam)\n\tfor spam@localhost
(single-drop); Wed, 01 Sep 2004 15:46:33 +0400 (MSD)',
'from thecanadianteacher.com ([222.65.104.100])\n\tby mail.onego.ru
(8.12.11/8.12.11) with SMTP id i817UtUN026093;\n\tWed, 1 Sep 2004
11:30:58 +0400']
```

Стоит заметить, что в электронном письме может быть несколько полей с именем `received` (в этом примере их два).

Некоторые важные данные можно получить в готовом виде, например, тип содержимого, кодировку:

```
>>> msg.get_content_type()
'text/plain'
>>> print msg.get_main_type(), msg.get_subtype()
text plain
>>> print msg.get_charset()
None
>>> print msg.get_params()
[('text/plain', ''), ('charset', 'us-ascii')]
>>> msg.is_multipart()
False
```

или список полей:

```
>>> print msg.keys()
['Received', 'Received', 'Message-ID', 'Date', 'From', 'User-Agent',
'MIME-Version', 'To', 'Subject', 'Content-Type',
'Content-Transfer-Encoding', 'Spam', 'X-Spam']
```

Так как сообщение состоит из одной части, можно получить его тело в виде строки:

```
>>> print msg.get_payload()
sorgeloosheid  hullw ifesh nozama decompresssequenceframes

Believe it or not, I have tried several sites to b"_"uy presription
medication. I should say that currently you are still be the best amony
...
```

Теперь будет рассмотрен другой пример, в котором сообщение состоит из нескольких частей. Это сообщение порождено вирусом. Оно состоит из двух частей: HTML-текста и вложенного файла с расширением cpl. Для доступа к частям сообщения используется метод `walk()`, который обходит все его части. Попутно следует собрать типы содержимого (в списке `parts`), поля `Content-Type` (в `ct_fields`) и имена файлов (в `filenames`):

```
import email
parts = []
ct_fields = []
filenames = []
f = open("virus.eml")
msg = email.message_from_file(f)
for submsg in msg.walk():
```

```
      parts.append(submsg.get_content_type())
      ct_fields.append(submsg.get('Content-Type', ''))
      filenames.append(submsg.get_filename())
      if submsg.get_filename():
          print "Длина файла:", len(submsg.get_payload())
    f.close()
    print parts
    print ct_fields
    print filenames
```

В результате получилось:

```
Длина файла: 31173
['multipart/mixed', 'text/html', 'application/octet-stream']
['multipart/mixed;\n        boundary="--------hidejpxkblmvuwfplzue"',
'text/html; charset="us-ascii"',
'application/octet-stream; name="price.cpl"']
[None, None, 'price.cpl']
```

Из списка `parts` можно увидеть, что само сообщение имеет тип `multipart/mixed`, тогда как две его части - `text/html` и `application/octet-stream` соответственно. Только с последней частью связано имя файла (price.cpl). Файл читается методом `get_payload()` и вычисляется его длина.

Кстати, в случае, когда сообщение является контейнером для других частей, `get_payload()` выдает список объектов-сообщений (то есть экземпляров класса `Message`).

Формирование сообщения

Часто возникает ситуация, когда нужно сформировать сообщение с вложенным файлом. В следующем примере строится сообщение с текстом и вложением. В качестве класса для порождения сообщения можно использовать не только `Message` из модуля `email.Message`, но и `MIMEMultipart` из `email.MIMEMultipart` (для сообщений из нескольких частей), `MIMEImage` (для сообщения с графическим изображением), `MIMEAudio` (для аудиофайлов), `MIMEText` (для текстовых частей):

```
    # Загружаются необходимые модули и функции из модулей
    from email.Header import make_header as mkh
    from email.MIMEMultipart import MIMEMultipart
    from email.MIMEText import MIMEText
    from email.MIMEBase import MIMEBase
    from email.Encoders import encode_base64

    # Создается главное сообщение и задаются некоторые поля
    msg = MIMEMultipart()
    msg["Subject"] =  mkh([("Привет", "koi8-r")])
    msg["From"] = mkh([("Друг", "koi8-r"), ("<friend@mail.ru>", "us-ascii")])
    msg["To"]   =   mkh([("Друг2",   "koi8-r"),   ("<friend2@yandex.ru>",   "us-
ascii")])

    # То, чего будет не видно, если почтовая программа поддерживает MIME
    msg.preamble = "Multipart message"
    msg.epilogue = ""

    # Текстовая часть сообщения
    text = u"""К письму приложен файл с архивом.""".encode("koi8-r")
    to_attach = MIMEText(text, _charset="koi8-r")
    msg.attach(to_attach)

    # Прикладывается файл
```

```
fp = open("archive_file.zip", "rb")
to_attach = MIMEBase("application", "octet-stream")
to_attach.set_payload(fp.read())
encode_base64(to_attach)
to_attach.add_header("Content-Disposition", "attachment",
                     filename="archive_file.zip")
fp.close()
msg.attach(to_attach)

print msg.as_string()
```

В этом примере видно сразу несколько модулей пакета `email`. Функция `make_header()` из `email.Header` позволяет закодировать содержимое для заголовка:

```
>>> from email.Header import make_header
>>> print make_header([("Друг", "koi8-r"), ("<friend@mail.ru>", "us-ascii")])
=?koi8-r?b?5NLVxw==?= <friend@mail.ru>
>>> print make_header([(u"Друг", ""), ("<friend@mail.ru>", "us-ascii")])
=?utf-8?b?w6TDksOVw4c=?= friend@mail.ru
```

Функция `email.Encoders.encode_base64()` воздействует на переданное ей сообщение и кодирует тело с помощью base64. Другие варианты: `encode_quopri()` - кодировать quoted printable, `encode_7or8bit()` - оставить семь или восемь бит. Эти функции добавляют необходимые поля.

Аргументы конструкторов классов из MIME-модулей пакета `email`:

`class MIMEBase(_maintype, _subtype, **_params)`
Базовый класс для всех использующих MIME сообщений (подклассов `Message`). Тип содержимого задается через `_maintype` и `_subtype`.

`class MIMENonMultipart()`
Подкласс для `MIMEBase`, в котором запрещен метод `attach()`, отчего он гарантированно состоит из одной части.

`class MIMEMultipart([_subtype[, boundary[, _subparts[, _params]]]])`
Подкласс для `MIMEBase`, который является базовым для MIME-сообщений из нескольких частей. Главный тип `multipart`, подтип указывается с помощью `_subtype`.

`class MIMEAudio(_audiodata[, _subtype[, _encoder[, **_params]]])`
Подкласс `MIMENonMultipart`. Используется для создания MIME-сообщений, содержащих аудио данные. Главный тип - `audio`, подтип указывается с помощью `_subtype`. Данные задаются параметром `_audiodata`.

`class MIMEImage(_imagedata[, _subtype[, _encoder[, **_params]]])`
Подкласс `MIMENonMultipart`. Используется для создания MIME-сообщений с графическим изображением. Главный тип - `image`, подтип указывается с помощью `_subtype`. Данные задаются параметром `_imagedata`.

`class MIMEMessage(_msg[, _subtype])`
Подкласс `MIMENonMultipart` для класса `MIMENonMultipart` используется для создания MIME-объектов с главным типом `message`. Параметр `_msg` применяется в качестве тела и должен являться экземпляром класса `Message` или его потомков. Подтип задается с помощью `_subtype`, по умолчанию `'rfc822'`.

`class MIMEText(_text[, _subtype[, _charset]])`

Подкласс `MIMENonMultipart`. Используется для создания MIME-сообщений текстового типа. Главный тип - `text`, подтип указывается с помощью `_subtype`. Данные задаются параметром `_text`. Посредством `_charset` можно указать кодировку (по умолчанию `'us-ascii'`).

Разбор поля заголовка

В примере выше поле `Subject` формировалось с помощью `email.Header.make_header()`. Разбор поля поможет провести другая функция: `email.Header.decode_header()`. Эта функция возвращает список кортежей, в каждом из них указан кусочек текста поля и кодировка, в которой этот текст был задан. Следующий пример поможет понять суть дела:

```
    subj                                    =                                    """=?koi8-
r?Q?=FC=D4=CF_=D0=D2=C9=CD=C5=D2_=CF=DE=C5=CE=D8_=C4=CC=C9?=
     =?koi8-r?Q?=CE=CE=CF=C7=CF_=28164_bytes=29_=D0=CF=CC=D1_=D3_=D4?=
     =?koi8-r?Q?=C5=CD=CF=CA_=D3=CF=CF=C2=DD=C5=CE=C9=D1=2E_=EF=CE=CF_?=
     =?koi8-r?Q?=D2=C1=DA=C2=C9=CC=CF=D3=D8_=CE=C1_=CB=D5=D3=CB=C9_=D7?=
     =?koi8-r?Q?_=D3=CF=CF=C2=DD=C5=CE=C9=C9=2C_=CE=CF_=CC=C5=C7=CB=CF?=
     =?koi8-r?Q?_=D3=CF=C2=C9=D2=C1=C5=D4=D3=D1_=D7_=D4=C5=CB=D3=D4_?=
     =?koi8-r?Q?=D3_=D0=CF=CD=CF=DD=D8=C0_email=2EHeader=2Edecode=5Fheader?=
     =?koi8-r?Q?=28=29?="""
    import email.Header
    for text, enc in email.Header.decode_header(subj):
      print enc, text
```

В результате будет выведено:

```
koi8-r Это пример очень длинного (164 bytes) поля с темой сообщения.
Оно разбилось на куски в сообщении, но легко собирается в текст
с помощью email.Header.decode_header()
```

Следует заметить, что кодировку можно не указывать:

```
>>> email.Header.decode_header("simple text")
[('simple text', None)]
>>> email.Header.decode_header("пример")
[('\xd0\xd2\xc9\xcd\xc5\xd2', None)]
>>> email.Header.decode_header("=?KOI8-R?Q?=D0=D2=CF_?=Linux")
[('\xd0\xd2\xcf ', 'koi8-r'), ('Linux', None)]
```

Если в первом случае можно подразумевать us-ascii, то во втором случае о кодировке придется догадываться: вот почему в электронных письмах нельзя просто так использовать восьмибитные кодировки. В третьем примере русские буквы закодированы, а латинские - нет, поэтому в результате `email.Header.decode_header()` список из двух пар.

В общем случае представить поле сообщения можно только в Unicode. Создание функции для такого преобразования предлагается в качестве упражнения.

Язык XML

В рамках одной лекции довольно сложно объяснить, что такое XML, и то, как с ним работать. В примерах используется входящий в стандартную поставку пакет `xml`.

XML (Extensible Markup Language, расширяемый язык разметки) позволяет налаживать взаимодействие между приложениями различных производителей, хранить и подвергать обработке сложно структурированные данные.

Язык XML (как и HTML) является подмножеством SGML, но его применения не ограничены системой WWW. В XML можно создавать собственные наборы тегов для конкретной предметной области. В XML можно хранить и подвергать обработке базы данных и знаний, протоколы взаимодействия между объектами, описания ресурсов и многое другое.

Новичкам не всегда понятно, зачем нужно использовать такой достаточно многословный формат, когда можно создать свой, компактный формат для хранения тех же самых данных. Преимущество XML состоит в том, что вместе с данными он хранит и контекстную информацию: теги и их атрибуты имеют имена. Немаловажно также, что XML сегодня - единый общепринятый стандарт, для которого создано немало инструментальных средств.

Говоря об XML, надо иметь в виду, что XML-документы бывают **формально-правильными** (well-formed) и **состоятельными** (valid). Состоятельный XML-документ - это формально-правильный XML-документ, имеющий **объявление типа документа** (DTD, Document Type Definition). Объявление типа документа задает грамматику, которой текст документа на XML должен удовлетворять. Для простоты изложения здесь не будет рассматриваться DTD, предпочтительнее ограничиться формально-правильными документами.

Для представления букв и других символов XML использует Unicode, что сокращает проблемы с представлением символов различных алфавитов. Однако это обстоятельство необходимо помнить и не употреблять в XML восьмибитную кодировку (во всяком случае, без явного указания).

Следующий пример достаточно простого XML-документа дает представление об этом формате (файл expression.xml):

```
<?xml version="1.0" encoding="iso-8859-1"?>
<expression>
  <operation type="+">
    <operand>2</operand>
    <operand>
      <operation type="*">
        <operand>3</operand>
        <operand>4</operand>
      </operation>
    </operand>
  </operation>
</expression>
```

XML-документ всегда имеет структуру дерева, в корне которого сам документ. Его части, описываемые вложенными парами тегов, образуют узлы. Таким образом, ребра дерева обозначают "непосредственное вложение". Атрибуты тега можно считать листьями, как и наиболее вложенные части, не имеющие в своем составе других частей. Получается, что документ имеет древесную структуру.

Примечание:

Следует заметить, что в отличие от HTML, в XML одиночные (непарные) теги записываются с косой чертой: `<BR/>`, а атрибуты - в кавычках. В XML имеет значение регистр букв в названиях тегов и атрибутов.

Формирование XML-документа

Концептуально существуют два пути обработки XML-документа: последовательная обработка и работа с объектной моделью документа.

В первом случае обычно используется **SAX** (Simple API for XML, простой программный интерфейс для XML). Работа SAX заключается в чтении источников данных (input source) XML-анализаторами (XML-reader) и генерации последовательности событий (events), которые обрабатываются объектами-обработчиками (handlers). SAX дает последовательный доступ к XML-документу.

Во втором случае анализатор XML строит **DOM** (Document Object Model, объектная модель документа), предлагая для XML-документа конкретную объектную модель. В рамках этой модели узлы DOM-дерева доступны для произвольного доступа,а для переходов между узлами предусмотрен ряд методов.

Можно применить оба этих подхода для формирования приведенного выше XML-документа.

С помощью SAX документ сформируется так:

```
import sys
from xml.sax.saxutils import XMLGenerator
g = XMLGenerator(sys.stdout)
g.startDocument()
g.startElement("expression", {})
g.startElement("operation", {"type": "+"})
g.startElement("operand", {})
g.characters("2")
g.endElement("operand")
g.startElement("operand", {})
g.startElement("operation", {"type": "*"})
g.startElement("operand", {})
g.characters("3")
g.endElement("operand")
g.startElement("operand", {})
g.characters("4")
g.endElement("operand")
g.endElement("operation")
g.endElement("operand")
g.endElement("operation")
g.endElement("expression")
g.endDocument()
```

Построение дерева объектной модели документа может выглядеть, например, так:

```
from xml.dom import minidom
dom = minidom.Document()
e1 = dom.createElement("expression")
dom.appendChild(e1)
p1 = dom.createElement("operation")
p1.setAttribute('type', '+')
x1 = dom.createElement("operand")
x1.appendChild(dom.createTextNode("2"))
p1.appendChild(x1)
e1.appendChild(p1)
p2 = dom.createElement("operation")
p2.setAttribute('type', '*')
x2 = dom.createElement("operand")
x2.appendChild(dom.createTextNode("3"))
p2.appendChild(x2)
x3 = dom.createElement("operand")
x3.appendChild(dom.createTextNode("4"))
p2.appendChild(x3)
x4 = dom.createElement("operand")
x4.appendChild(p2)
p1.appendChild(x4)
print dom.toprettyxml()
```

Легко заметить, что при использовании SAX команды на генерацию тегов и других частей выдаются последовательно, а вот построение одной и той же DOM можно выполнять различными последовательностями команд формирования узла и его соединения с другими узлами.

Конечно, указанные примеры носят довольно теоретический характер, так как на практике строить XML-документы таким образом обычно не приходится.

Анализ XML-документа

Для работы с готовым XML-документом нужно воспользоваться XML-анализаторами. Анализ XML-документа с порождением объекта класса `Document` происходит всего в одной строчке, с помощью функции `parse()`. Здесь стоит заметить, что кроме стандартного пакета `xml` можно поставить пакет `PyXML` или альтернативные коммерческие пакеты. Тем не менее, разработчики стараются придерживаться единого API, который продиктован стандартом DOM Level 2:

```
import xml.dom.minidom
dom = xml.dom.minidom.parse("expression.xml")

dom.normalize()

def output_tree(node, level=0):
  if node.nodeType == node.TEXT_NODE:
    if node.nodeValue.strip():
      print ". "*level, node.nodeValue.strip()
  else:   # ELEMENT_NODE или DOCUMENT_NODE
    atts = node.attributes or {}
    att_string = ", ".join(
        ["%s=%s " % (k, v)  for k, v in atts.items()])
    print ". "*level, node.nodeName, att_string
    for child in node.childNodes:
      output_tree(child, level+1)

output_tree(dom)
```

В этом примере дерево выводится с помощью определенной функции `output_tree()`, которая принимает на входе узел и вызывается рекурсивно для всех вложенных узлов.

В результате получается примерно следующее:

```
#document
.  expression
. .  operation type=+
. . .  operand
. . . .  2
. . .  operand
. . . .  operation type=*
. . . . .  operand
. . . . . .  3
. . . . .  operand
. . . . . .  4
```

Здесь же применяется метод `normalize()` для того, чтобы все текстовые фрагменты были слиты воедино (в противном случае может следовать подряд несколько узлов с текстом).

Можно заметить, что даже в небольшом примере использовались атрибуты узлов: `node.nodeType` указывает тип узла, `node.nodeValue` применяется для доступа к данным, `node.nodeName` дает имя узла (соответствует названию тега), `node.attributes` дает

доступ к атрибутам узла. `node.childNodes` применяется для доступа к дочерним узлам. Этих свойств достаточно, чтобы рекурсивно обойти дерево.

Все узлы являются экземплярами подклассов класса `Node`. Они могут быть следующих типов:

Название	**Описание**	**Метод для создания**
ELEMENT_NODE	Элемент	createElement(tagname)
ATTRIBUTE_NODE	Атрибут	createAttribute(name)
TEXT_NODE	Текстовый узел	createTextNode(data)
CDATA_SECTION_NODE	Раздел CDATA	
ENTITY_REFERENCE_NODE	Ссылка на сущность	
ENTITY_NODE	Сущность	
PROCESSING_INSTRUCTION_NODE	Инструкция по обработке	createProcessingInstruction(target, data)
COMMENT_NODE	Комментарий	createComment(comment)
DOCUMENT_NODE	Документ	
DOCUMENT_TYPE_NODE	Тип документа	
DOCUMENT_FRAGMENT_NODE	Фрагмент документа	
NOTATION_NODE	Нотация	

В DOM документ является деревом, в узлах которого стоят объекты нескольких возможных типов. Узлы могут иметь атрибуты или данные. Доступ к узлам можно осуществлять через атрибуты вроде `childNodes` (дочерние узлы), `firstChild` (первый дочерний узел), `lastChild` (последний дочерний узел), `parentNode` (родитель), `nextSibling` (следующий брат), `previousSibling` (предыдущий брат).

Выше уже говорилось о методе `appendChild()`. К нему можно добавить методы `insertBefore(newChild, refChild)` (вставить `newChild` до `refChild`), `removeChild(oldChild)` (удалить дочерний узел), `replaceChild(newChild, oldChild)` (заметить `oldChild` на `newChild`). Есть еще метод `cloneNode(deep)`, который клонирует узел (вместе с дочерними узлами, если задан `deep=1`).

Узел типа `ELEMENT_NODE`, помимо перечисленных методов "просто" узла, имеет много других методов. Вот основные из них:

`tagName`
Имя типа элемента.

`getElementsByTagName(tagname)`
Получает элементы с указанным именем `tagname` среди всех потомков данного элемента.

`getAttribute(attname)`
Получить значение атрибута с именем `attname`.

`getAttributeNode(attrname)`
Возвращает атрибут с именем `attrname` в виде объекта-узла.

`removeAttribute(attname)`

Удалить атрибут с именем `attname`.

`removeAttributeNode(oldAttr)`
Удалить атрибут `oldAttr` (задан в виде объекта-узла).

`setAttribute(attname, value)`
Устанавливает значение атрибута `attname` равным строке `value`.

`setAttributeNode(newAttr)`
Добавляет новый узел-атрибут к элементу. Старый атрибут заменяется, если имеет то же имя.

Здесь стоит заметить, что атрибуты в рамках элемента повторяться не должны. Их порядок также не важен с точки зрения информационной модели XML.

В качестве упражнения предлагается составить функцию, которая будет вычислять значение выражения, заданного в XML-представлении.

Пространства имен

Еще одной интересной особенностью XML, о которой нельзя не упомянуть, являются пространства имен. Они позволяют составлять XML-документы из кусков различных схем. Например, таким образом в XML-документ можно включить кусок HTML, указав во всех элементах HTML принадлежность особому пространству имен.

Следующий пример XML-кода показывает синтаксис пространств имен (файл foaf.rdf):

```
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF
   xmlns:dc="http://http://purl.org/dc/elements/1.1/"
   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
   xmlns:foaf="http://xmlns.com/foaf/0.1/"
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
>
  <rdf:Description rdf:nodeID="_:jCBxPziO1">
    <foaf:nick>donna</foaf:nick>
    <foaf:name>Donna Fales</foaf:name>
    <rdf:type rdf:resource="http://xmlns.com/foaf/0.1/Person"/>
  </rdf:Description>
</rdf:RDF>
```

Примечание:

Пример позаимствован из пакета cwm, созданного командой разработчиков во главе с Тимом Бернерс-Ли, создателем технологии WWW. Кстати, cwm тоже написан на Python. Пакет cwm служит обработчиком данных общего назначения для семантической сети - новой идеи, продвигаемой Тимом Бернерс-Ли. Коротко суть идеи состоит в том, чтобы сделать современный "веб" много полезнее, формализовав знания в виде распределенной базы XML-документов, по аналогии с тем как WWW представляет собой распределенную базу документов. Отличие глобальной семантической сети от WWW в том, что она даст машинам возможность обрабатывать знания, делая логические выводы на основании заложенной в документах информации.

Названия пространств имен следуют в виде префиксов к названиям элементов. Эти названия - не просто имена. Они соответствуют идентификаторам, которые должны быть заданы в виде **URI** (Universal Resource Identifie, универсальный указатель

ресурса). В примере выше упоминаются пять пространств имен (xmlns, dc, rdfs, foaf и rdf), из которых только первое не требует объявления, так как является встроенным. Из них реально использованы только три: (xmlns, foaf и rdf).

Пространства имен позволяют выделять из XML-документа части, относящиеся к различным схемам, что важно для тех инструментов, которые интерпретируют XML.

В пакете `xml` есть методы, понимающие механизм пространств имен. Обычно такие методы и атрибуты имеют в своем имени буквы NS.

Получить URI, который соответствует пространству имен данного элемента, можно с помощью атрибута `namespaceURI`.

В следующем примере печатается URI элементов:

```
import xml.dom.minidom
dom = xml.dom.minidom.parse("ex.xml")

def output_ns(node):
  if node.nodeType == node.ELEMENT_NODE:
    print node.nodeName, node.namespaceURI
  for child in node.childNodes:
    output_ns(child)

output_ns(dom)
```

Пример выведет:

```
rdf:RDF http://www.w3.org/1999/02/22-rdf-syntax-ns#
rdf:Description http://www.w3.org/1999/02/22-rdf-syntax-ns#
foaf:nick http://xmlns.com/foaf/0.1/
foaf:name http://xmlns.com/foaf/0.1/
rdf:type http://www.w3.org/1999/02/22-rdf-syntax-ns#
```

Следует заметить, что указание пространства имен может быть сделано для имен не только элементов, но и атрибутов.

Подробнее узнать о работе с пространствами имен в xml-пакетах для Python можно из документации.

Заключение

В этой лекции были рассмотрены варианты обработки текстовой информации трех достаточно распространенных форматов: CSV, Unix mailbox и XML. Конечно, форматов данных, даже основанных на тексте, гораздо больше, однако то, что было представлено, поможет быстрее разобраться с любым модулем для обработки формата или построить свой модуль так, чтобы другие могли понять ваши намерения.