Язык программирования Python

Сузи Роман Авриевич

Изучается язык программирования Python, его основные библиотеки и некоторые приложения.

Курс посвящен одному из бурно развивающихся и популярных в настоящее время сценарных языков программирования - Python. Язык Python позволяет быстро создавать как прототипы программных систем, так и сами программные системы, помогает в интеграции программного обеспечения для решения производственных задач. Python имеет богатую стандартную библиотеку и большое количество модулей расширения практически для всех нужд отрасли информационных технологий. Благодаря ясному синтаксису изучение языка не составляет большой проблемы. Написанные на нем программы получаются структурированными по форме, и в них легко проследить логику работы. На примере языка Python рассматриваются такие важные понятия как: объектно-ориентированное программирование, функциональное программирование, событийно-управляемые программы (GUI-приложения), форматы представления данных (Unicode, XML и т.п.). Возможность диалогового режима работы интерпретатора Python позволяет существенно сократить время изучения самого языка и перейти к решению задач в соответствующих предметных областях. Python свободно доступен для многих платформ, а написанные на нем программы обычно переносимы между платформами без изменений. Это обстоятельство позволяет применять для изучения языка любую имеющуюся аппаратную платформу.

Содержание

7.	**Работа с данными в различных форматах**	Работа с современными форматами данных - одно из сильных мест стандартной библиотеки **Python.** В этой лекции будут рассмотрены типичные для **Python** подходы к чтению, преобразованию и записи информации в требуемых форматах. В настоящее время разработано и доступно в Интернете большое количество модулей для всевозможных форматов данных.
8.	**Разработка Web-приложений**	Одна из главных сфер применения языка **Python - web-**приложения - представляется в этой лекции на конкретных примерах. Кроме того, делается акцент на типичных слабых местах безопасности **web-**приложений.
9.	**Сетевые приложения на Python**	В этой лекции рассматривается реализация на **Python** простейшего клиент-серверного приложения, дается представление о типичном для сети **Internet** приложении. Стандартная библиотека **Python** имеет несколько модулей для работы с различными протоколами. Этими модулями охватываются как низкоуровневые протоколы **(TCP/IP, UDP/IP),** так и высокоуровневые **(HTTP, FTP, SMTP, POP3, IMAP, NNTP, ...).** Здесь будет рассмотрена работа с сокетами **(**модуль **socket)** и три модуля высокоуровневых протоколов **(urllib2, poplib, smtplib).** При этом предполагается, что имеется понимание принципов работы **IP-**сети и некоторых ее сервисов, а также представление о системе **WWW.**
10.	**Работа с базой данных**	В этой лекции рассматривается спецификация **DB-API 2.0** и модуль для работы с конкретной базой данных, дается начальное представление о языке запросов **SQL.**
11.	**Многопоточные вычисления**	В этой лекции рассматриваются вопросы взаимодействия потоков **(**нитей**)** в рамках одной программы. Вводятся основные понятия **(**семафоры, очереди, блокировки**).** Делается попытка объяснить особенности параллельного программирования на основе модели многопоточности.
12.	**Создание приложений с графическим интерфейсом пользователя**	В этой лекции рассматривается создание простейшего приложения с графическим интерфейсом пользователя. Для построения интерфейса не применяются визуальные средства **(**"построители интерфейса"**),** а используются возможности графической библиотеки виджетов **(Tk).**
13.	**Интеграция Python с другими языками программирования**	В этой лекции рассматривается встраивание **(embedding)** интерпретатора **Python** в программу на **C,** и, наоборот, написание модулей для **Python** на языке **C (extending).** Кратко описывается инструмент для связывания **C-**библиотек с программами на **Python (SWIG).** Дается обзор связок языка **Python** с другими языками программирования**: C++, Java, OCaml, Prolog.** Коротко говорится о специальном языке для написания модулей расширения **Python - Pyrex.**
14.	**Устройство интерпретатора языка Python**	В этой лекции сделана попытка пролить свет на внутреннее устройство интерпретатора **Python.** Для иллюстрации работы интерпретатора рассматриваются отладчик, профайлер и "дизассемблер"**.**

Лекция #1: Введение в программирование на языке Python

Что такое Python?

О **Python** (лучше произносить "питон", хотя некоторые говорят "пайтон") - предмете данного изучения, лучше всего говорит создатель этого языка программирования, голландец Гвидо ван Россум:

"**Python** - интерпретируемый, объектно-ориентированный высокоуровневый язык программирования с динамической семантикой. Встроенные высокоуровневые структуры данных в сочетании с динамическими типизацией и связыванием делают язык привлекательным для быстрой разработки приложений (**RAD**, **Rapid Application Development**). Кроме того, его можно использовать в качестве сценарного языка для связи программных компонентов. Синтаксис **Python** прост в изучении, в нем придается особое значение читаемости кода, а это сокращает затраты на сопровождение программных продуктов. **Python** поддерживает модули и пакеты, поощряя модульность и повторное использование кода. Интерпретатор **Python** и большая стандартная библиотека доступны бесплатно в виде исходных и исполняемых кодов для всех основных платформ и могут свободно распространяться."

В процессе изучения будет раскрыт смысл этого определения, а сейчас достаточно знать, что **Python** - это универсальный язык программирования. Он имеет свои преимущества и недостатки, а также сферы применения. В поставку **Python** входит обширная стандартная библиотека для решения широкого круга задач. В Интернете доступны качественные библиотеки для **Python** по различным предметным областям: средства обработки текстов и технологии Интернет, обработка изображений, инструменты для создания приложений, механизмы доступа к базам данных, пакеты для научных вычислений, библиотеки построения графического интерфейса и т.п. Кроме того, **Python** имеет достаточно простые средства для интеграции с языками **C**, **C++** (и **Java**) как путем встраивания (**embedding**) интерпретатора в программы на этих языках, так и наоборот, посредством использования библиотек, написанных на этих языках, в **Python**-программах. Язык **Python** поддерживает несколько **парадигм** программирования: императивное (процедурный, структурный, модульный подходы), объектно-ориентированное и функциональное программирование.

Можно считать, что **Python** - это целая технология для создания программных продуктов (и их прототипов). Она доступна почти на всех современных платформах (как **32**-битных, так и на **64**-битных) с компилятором **C** и на платформе **Java**.

Может показаться, что, в программной индустрии нет места для чего-то другого кроме **C/C++**, **Java**, **Visual Basic**, **C#**. Однако это не так. Возможно, благодаря данному курсу лекций и практических занятий у **Python** появятся новые приверженцы, для которых он станет незаменимым инструментом.

Как описать язык?

В этой лекции не ставится цели систематически описать **Python**: для этого существует оригинальное справочное руководство. Здесь предлагается рассмотреть язык одновременно в нескольких аспектах, что достигается набором примеров, которые позволят быстрее приобщиться к реальному программированию, чем в случае строгого академического подхода.

Однако стоит обратить внимание на правильный подход к описанию языка. Создание программы - это всегда коммуникация, в которой программист передает компьютеру информацию, необходимую для выполнения последним действий. То, как эти действия понимает программист (то есть "смысл"), можно назвать **семантикой**. Средством передачи этого смысла является **синтаксис** языка программирования. Ну а то, что делает

интерпретатор на основании переданного, обычно называют **прагматикой**. При написании программы очень важно, чтобы в этой цепочке не возникало сбоев.

Синтаксис - полностью формализованная часть: его можно описать на формальном языке синтаксических диаграмм (что и делается в справочных руководствах). Выражением прагматики является сам интерпретатор языка. Именно он читает записанное в соответствии с синтаксисом "послание" и превращает его в действия по заложенному в нем алгоритму. Неформальным компонентом остается только семантика. Именно в переводе смысла в формальное описание и кроется самая большая сложность программирования. Синтаксис языка **Python** обладает мощными средствами, которые помогают приблизить понимание проблемы программистом к ее "пониманию" интерпретатором. О внутреннем устройстве **Python** будет говориться в одной из завершающих лекций.

История языка Python

Создание **Python** было начато Гвидо ван Россумом **(Guido van Rossum)** в **1991** году, когда он работал над распределенной ОС Амеба. Ему требовался расширяемый язык, который бы обеспечил поддержку системных вызовов. За основу были взяты **ABC** и Модула-**3**. В качестве названия он выбрал **Python** в честь комедийных серий **BBC** "Летающий цирк Монти-Питона", а вовсе не по названию змеи. С тех пор **Python** развивался при поддержке тех организаций, в которых Гвидо работал. Особенно активно язык совершенствуется в настоящее время, когда над ним работает не только команда создателей, но и целое сообщество программистов со всего мира. И все-таки последнее слово о направлении развития языка остается за Гвидо ван Россумом.

Программа на Python

Программа на языке **Python** может состоять из одного или нескольких **модулей**. Каждый модуль представляет собой текстовый файл в кодировке, совместимой с **7**-битной кодировкой **ASCII**. Для кодировок, использующих старший бит, необходимо явно указывать название кодировки. Например, модуль, комментарии или строковые литералы которого записаны в кодировке **KOI8-R**, должен иметь в первой или второй строке следующую спецификацию:

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

Благодаря этой спецификации интерпретатор **Python** будет знать, как корректно переводить символы литералов **Unicode**-строк в **Unicode**. Без этой строки новые версии **Python** будут выдавать предупреждение на каждый модуль, в котором встречаются коды с установленным восьмым битом.

О том, как делать программу модульной, станет известно в следующих лекций. В примерах ниже используются как фрагменты модулей, записанных в файл, так и фрагменты диалога с интерпретатором **Python**. Последние отличаются характерным приглашением >>>. Символ решетка (#) отмечает комментарий до конца строки.

Программа на **Python**, с точки зрения интерпретатора, состоит из **логических строк**. Одна логическая строка, как правило, располагается в одной физической, но длинные логические строки можно явно (с помощью обратной косой черты) или неявно (внутри скобок) разбить на несколько физических:

```
print a, " - очень длинная строка, которая не помещается в", \
   80, "знакоместах"
```

Примечание:

Во всех примерах в основном используется "официальный" стиль оформления кода на **Python** в соответствии с документом **"Python Style Guide"**, который можно найти на сайте **http://python.org**

Основные алгоритмические конструкции

Предполагается, что слушатели уже умеют программировать хотя бы на уровне школьной программы, и потому вполне достаточно провести параллели между алгоритмическими конструкциями и синтаксисом **Python**. Кроме того, **Python** как правило не подводит интуицию программиста (по крайней мере, науке хорошо известны типичные ловушки начинающих программистов на **Python**), поэтому изучать синтаксис **Python** предпочтительнее на примерах, а не с помощью синтаксических диаграмм или форм Бэкуса-Наура.

Последовательность операторов

Последовательные действия описываются последовательными строками программы. Стоит, правда, добавить, что в программах важны отступы, поэтому все операторы, входящие в последовательность действий, должны иметь один и тот же отступ:

```
a = 1
b = 2
a = a + b
b = a - b
a = a - b
print a, b
```

Что делает этот пример? Проверить свою догадку можно с помощью интерактивного режима интерпретатора **Python**.

При работе с **Python** в **интерактивном режиме** как бы вводится одна большая программа, состоящая из последовательных действий. В примере выше использованы операторы присваивания и оператор `print`.

Оператор условия и выбора

Разумеется, одними только последовательными действиями в программировании не обойтись, поэтому при написании алгоритмов используется еще и **ветвление**:

```
if a > b:
  c = a
else:
  c = b
```

Этот кусок кода на **Python** интуитивно понятен каждому, кто помнит, что `if` по-английски значит "если", а `else` - "иначе". Оператор ветвления имеет в данном случае две части, операторы каждой из которых записываются с отступом вправо относительно оператора ветвления. Более общий случай - **оператор выбора** - можно записать с помощью следующего синтаксиса (пример вычисления знака числа):

```
if a < 0:
  s = -1
elif a == 0:
  s = 0
else:
  s = 1
```

Стоит заметить, что `elif` - это сокращенный `else if`. Без сокращения пришлось бы применять **вложенный** оператор ветвления:

```
if a < 0:
  s = -1
else:
  if a == 0:
```

```
    s = 0
  else:
    s = 1
```

В отличие от оператора `print`, оператор `if-else` - составной оператор.

Циклы

Третьей необходимой алгоритмической конструкцией является **цикл**. С помощью цикла можно описать повторяющиеся действия. В **Python** имеются два вида циклов: **цикл ПОКА** (выполняется некоторое действие) и **цикл ДЛЯ** (всех значений последовательности). Следующий пример иллюстрирует **цикл ПОКА** на **Python**:

```
s = "abcdefghijklmnop"
while s != "":
  print s
  s = s[1:-1]
```

Оператор `while` говорит интерпретатору **Python**: "пока верно **условие цикла**, выполнять **тело цикла**". В языке **Python** тело цикла выделяется отступом. Каждое исполнение **тела цикла** будет называться **итерацией**. В приведенном примере убирается первый и последний символ строки до тех пор, пока не останется пустая строка.

Для большей гибкости при организации циклов применяются операторы `break` (прервать) и `continue` (продолжить). Первый позволяет прервать цикл, а второй - продолжить цикл, перейдя к следующей итерации (если, конечно, выполняется условие цикла).

Следующий пример читает строки из файла и выводит те, у которых длина больше **5**:

```
f = open("file.txt", "r")
while 1:
  l = f.readline()
  if not l:
    break
  if len(l) > 5:
    print l,
f.close()
```

В этом примере организован **бесконечный цикл**, который прерывается только при получении из файла пустой строки (`l`), что обозначает конец файла.

В языке **Python логическое значение** несет каждый объект: нули, пустые строки и последовательности, специальный объект `None` и **логический литерал** `False` имеют значение "ложь", а прочие объекты значение "истина". Для обозначения истины обычно используется `1` или `True`.

Примечание:

Литералы `True` и `False` для обозначения логических значений появились в **Python 2.3**.

Цикл ДЛЯ выполняет тело цикла для каждого элемента последовательности. В следующем примере выводится таблица умножения:

```
for i in range(1, 10):
  for j in range(1, 10):
    print "%2i" % (i*j),
  print
```

Здесь циклы for являются **вложенными.** Функция range() порождает список целых чисел из полуоткрытого диапазона [1, 10)**.** Перед каждой итерацией **счетчик цикла** получает очередное значение из этого списка**.** Полуоткрытые диапазоны общеприняты в **Python.** Считается**,** что их использование более удобно и вызывает меньше программистских ошибок**.** Например**,** range(len(s)) порождает список индексов для списка s **(**в **Python-**последовательности первый элемент имеет индекс 0**).** Для красивого вывода таблицы умножения применена **операция форматирования** % **(**для целых чисел тот же символ используется для обозначения операции взятия остатка от деления**).** Строка форматирования **(**задается слева**)** строится почти как **строка форматирования** для printf из **C.**

Функции

Программист может определять собственные функции двумя способами**:** с помощью оператора def или прямо в выражении**,** посредством lambda**.** Второй способ **(**да и вообще работа с функциями**)** будет рассмотрен подробнее в лекции по функциональному программированию на **Python,** а здесь следует привести пример определения и вызова функции**:**

```
def cena(rub, kop=0):
  return "%i руб. %i коп." % (rub, kop)

print cena(8, 50)
print cena(7)
print cena(rub=23, kop=70)
```

В этом примере определена функция двух аргументов **(**из которых второй имеет **значение по умолчанию -** 0**).** Вариантов вызова этой функции с конкретными параметрами также несколько**.** Стоит только заметить**,** что при вызове функции сначала должны идти позиционные параметры**,** а затем**,** именованные**.** Аргументы со значениями по умолчанию должны следовать после обычных аргументов**.** Оператор return возвращает значение функции**.** Из функции можно возвратить только один объект **,** но он может быть кортежем из нескольких объектов**.**

После оператора def имя cena оказывается связанным с функциональным объектом**.**

Исключения

В современных программах передача управления происходит не всегда так гладко**,** как в описанных выше конструкциях**.** Для обработки особых ситуаций **(**таких как деление на ноль или попытка чтения из несуществующего файла**)** применяется механизм **исключений.** Лучше всего пояснить синтаксис оператора try-except следующим примером**:**

```
try:
  res = int(open('a.txt').read()) / int(open('c.txt').read())
  print res
except IOError:
  print "Ошибка ввода-вывода"
except ZeroDivisionError:
  print "Деление на 0"
except KeyboardInterrupt:
  print "Прерывание с клавиатуры"
except:
  print "Ошибка"
```

В этом примере берутся числа из двух файлов и делятся одно на другое**.** В результате этих нехитрых действий может возникнуть несколько исключительных ситуаций**,** некоторые из них отмечены в частях except **(**здесь использованы стандартные встроенные исключения **Python).** Последняя часть except в этом примере улавливает все другие исключения**,**

которые не были пойманы выше. Например, если хотя бы в одном из файлов находится нечисловое значение, функция `int()` возбудит исключение `ValueError`. Его-то и сможет отловить последняя часть `except`. Разумеется, выполнение части `try` в случае возникновения ошибки уже не продолжается после выполнения одной из частей `except`.

В отличие от других языков программирования, в **Python** исключения нередко служат для упрощения алгоритмов. Записывая оператор `try-except`, программист может думать так: "попробую, а если сорвется - выполнится код в `except`". Особенно часто это используется для выражений, в которых значение получается по ключу из отображения:

```
try:
  value = dict[key]
except:
  value = default_value
```

Вместо

```
if dict.has_key(key):
  value = dict[key]
else:
  value = default_value
```

Примечание:

Пример уже несколько устаревшей идиомы языка **Python** иллюстрирует только дух этого подхода: в современном **Python** лучше записать так `value = dict.get(key, default_value)`.

Исключения можно возбуждать и из программы. Для этого служит оператор `raise`. Заодно следующий пример показывает канонический способ определения собственного исключения:

```
class MyError(Exception):
  pass

try:
  ...
  raise MyError, "my error 1"
  ...
except MyError, x:
  print "Ошибка:", x
```

Кстати, все исключения выстроены в иерархию классов, поэтому `ZeroDivisionError` может быть поймана как `ArithmeticError`, если соответствующая часть `except` будет идти раньше.

Для **утверждений** применяется специальный оператор `assert`. Он возбуждает `AssertionError`, если заданное в нем условие неверно. Этот оператор используют для самопроверки программы. В оптимизированном коде он не выполняется, поэтому строить на нем логику алгоритма нельзя. Пример:

```
c = a + b
assert c == a + b
```

Кроме описанной формы оператора, есть еще форма `try-finally` для гарантированного выполнения некоторых действий при передаче управления изнутри оператора `try-finally` вовне. Он может применяться для освобождения занятых ресурсов, что требует обязательного выполнения, независимо от произошедших внутри катаклизмов:

```
try:
  ...
```

```
finally:
  print "Обработка гарантированно завершена"
```

Смешивать вместе формы `try-except` и `try-finally` нельзя.

Встроенные типы данных

Как уже говорилось, все данные в **Python** представлены объектами. Имена являются лишь ссылками на эти объекты и не несут нагрузки по декларации типа. Значения встроенных типов имеют специальную поддержку в синтаксисе языка: можно записать **литерал** строки, числа, списка, кортежа, словаря (и их разновидностей). Синтаксическую же поддержку операций над встроенными типами можно легко сделать доступной и для объектов определяемых пользователей классов.

Следует также отметить, что объекты могут быть **неизменчивыми** и **изменчивыми**. Например, строки в **Python** являются неизменчивыми, поэтому операции над строками создают новые строки.

Карта встроенных типов (с именами функций для приведения к нужному типу и именами классов для наследования от этих типов):

- специальные типы: `None`, `NotImplemented` и `Ellipsis`;
- числа;
  - целые
    - обычное целое `int`
    - целое произвольной точности `long`
    - логический `bool`
  - число с плавающей точкой `float`
  - комплексное число `complex`
- последовательности;
  - неизменчивые:
    - строка `str`;
    - **Unicode-**строка `unicode`;
    - кортеж `tuple`;
  - изменчивые:
    - список `list`;
- отображения:
  - словарь `dict`
- объекты, которые можно вызвать:
  - функции (пользовательские и встроенные);
  - функции-генераторы;
  - методы (пользовательские и встроенные);
  - классы (новые и "классические");
  - экземпляры классов (если имеют метод `__call__`);
- модули;
- классы (см. выше);
- экземпляры классов (см. выше);
- файлы `file`;
- вспомогательные типы `buffer`, `slice`.

Узнать тип любого объекта можно с помощью встроенной функции `type()`.

Тип int и long

Два типа: `int` (целые числа) и `long` (целые произвольной точности) служат моделью для представления целых чисел. Первый соответствует типу `long` в компиляторе **C** для используемой архитектуры. Числовые литералы можно записать в системах счисления с основанием **8**, **10** или **16**:

```
# В этих литералах записано число 10
print 10, 012, 0xA, 10L
```

Набор операций над числами - достаточно стандартный как по семантике, так и по обозначениям:

```
>>> print 1 + 1, 3 - 2, 2*2, 7/4, 5%3
2 1 4 1 2
>>> print 2L ** 1000
10715086071862673209484250490600018105614048117055336074437503883703510511249361224931983788156958581275946729175531468251871452856923140435984577574698574803934567774824230985421074605062371141877954182153046474983581941267398767559165543946077062914571196477686542167660429831652624386837205668069376
>>> print 3 < 4 < 6,      3 >= 5,    4 == 4,    4 != 4  # сравнения
True False True False
>>> print 1 << 8,   4 >> 2,    ~4    # побитовые сдвиги и инверсия
256 1 -5
>>> for i, j in (0, 0), (0, 1), (1, 0), (1, 1):
...   print i, j, ":", i & j, i | j, i ^ j  # побитовые операции
...
0 0 : 0 0 0
0 1 : 0 1 1
1 0 : 0 1 1
1 1 : 1 1 0
```

Значения типа `int` должны покрывать диапазон от **-2147483648** до **2147483647**, а точность целых произвольной точности зависит от объема доступной памяти.

Стоит заметить, что если в результате операции получается значение, выходящее за рамки допустимого, тип `int` может быть неявно преобразован в `long`:

```
>>> type(-2147483648)
<type 'int'>
>>> type(-2147483649)
<type 'long'>
```

Также нужно быть осторожным при записи констант. Ноли в начале числа - признак восьмеричной системы счисления, в которой нет цифры **8**:

```
>>> 008
  File "<stdin>", line 1
    008
      ^
SyntaxError: invalid token
```

Тип float

Соответствует **C**-типу `double` для используемой архитектуры. Записывается вполне традиционным способом либо через точку, либо в нотации с экспонентой:

```
>>> pi = 3.1415926535897931
>>> pi ** 40
7.6912142205156999e+19
```

Кроме арифметических операций, можно использовать операции из модуля `math`.

Примечание:

Для финансовых расчетов лучше применять более подходящий тип.

Из полезных встроенных функций можно вспомнить round(), abs().

Тип complex

Литерал мнимой части задается добавлением j в качестве суффикса (перемножаются мнимые единицы):

```
>>> -1j * -1j
(-1-0j)
```

Тип реализован на базе вещественного. Кроме арифметических операций, можно использовать операции из модуля cmath.

Тип bool

Подтип целочисленного типа для "канонического" обозначения логических величин. Два значения: True (истина) и False (ложь) - вот и все, что принадлежит этому типу. Как уже говорилось, любой объект **Python** имеет истинностное значение, логические операции можно проиллюстрировать с помощью логического типа:

```
>>> for i in (False, True):
...   for j in (False, True):
...     print i, j, ":", i and j, i or j, not i
...
...
False False : False False True
False True : False True True
True False : False True False
True True : True True False
```

Следует отметить, что **Python** даже не вычисляет второй операнд операции and или or, если ее исход ясен по первому операнду. Таким образом, если первый операнд истиннен, он и возвращается как результат or, в противном случае возвращается второй операнд. Для операции and все аналогично.

Тип string и тип unicode

В **Python** строки бывают двух типов: обычные и **Unicode**-строки. Фактически строка - это последовательность символов (в случае обычных строк можно сказать "последовательность байтов"). Строки-константы можно задать в программе с помощью строковых литералов. Для литералов наравне используются как апострофы ('), так и обычные двойные кавычки ("). Для многострочных литералов можно использовать утроенные апострофы или утроенные кавычки. Управляющие последовательности внутри строковых литералов задаются обратной косой чертой (\). Примеры написания строковых литералов:

```
s1 = "строка1"
s2 = 'строка2\nс переводом строки внутри'
s3 = """строка3
с переводом строки внутри"""
u1 = u'\u043f\u0440\u0438\u0432\u0435\u0442'  # привет
u2 = u'Еще пример'    # не забудьте про coding!
```

Для строк имеется еще одна разновидность: **необработанные** строковые литералы. В этих литералах обратная косая черта и следующие за ней символы не интерпретируются как спецсимволы, а вставляются в строку "как есть":

```
my_re = r"(\d)=\1"
```

Обычно такие строки требуются для записи регулярных выражений (о них пойдет речь в лекции, посвященной обработке текстовой информации).

Набор операций над строками включает **конкатенацию "+", повтор "*", форматирование "%".** Также строки имеют большое количество методов, некоторые из которых приведены ниже. Полный набор методов (и их необязательных аргументов) можно получить в документации по **Python**.

```
>>> "A" + "B"
'AB'
>>> "A"*10
'AAAAAAAAAA'
>>> "%s %i" % ("abc", 12)
'abc 12'
```

Некоторые методы строковых объектов будут рассмотрены в лекции, посвященной обработке текстов.

Тип tuple

Для представления константной последовательности (разнородных) объектов используется тип кортеж. Литерал кортежа обычно записывается в круглых скобках, но можно, если не возникают неоднозначности, писать и без них. Примеры записи кортежей:

```
p = (1.2, 3.4, 0.9)   # точка в трехмерном пространстве
for s in "one", "two", "three":  # цикл по значениям кортежа
  print s
one_item = (1,)
empty = ()
p1 = 1, 3, 9   # без скобок
p2 = 3, 8, 5,  # запятая в конце игнорируется
```

Использовать синтаксис кортежей можно и в левой части оператора присваивания. В этом случае на основе вычисленных справа значений формируется кортеж и связывается один в один с именами в левой части. Поэтому обмен значениями записывается очень изящно:

```
a, b = b, a
```

Тип list

В "чистом" **Python** нет массивов с произвольным типом элемента. Вместо них используются списки. Их можно задать с помощью литералов, записываемых в квадратных скобках, или посредством списковых включений. Варианты задания списка приведены ниже:

```
lst1 = [1, 2, 3,]
lst2 = [x**2 for x in range(10) if x % 2 == 1]
lst3 = list("abcde")
```

Для работы со списками существует несколько методов, дополнительных к тем, что имеют неизменчивые последовательности. Все они связаны с изменением списка.

Последовательности

Ниже обобщены основные методы последовательностей. Следует напомнить, что последовательности бывают неизменчивыми и изменчивыми. У последних методов чуть больше.

Синтаксис	Семантика
`len(s)`	Длина последовательности `s`

`x in s`	Проверка принадлежности элемента последовательности. В новых версиях **Python** можно проверять принадлежность подстроки строке. Возвращает `True` или `False`
`x not in s`	**=** `not x in s`
`s + s1`	Конкатенация последовательностей
`s*n` или `n*s`	Последовательность из `n` раз повторенной `s`. Если `n < 0`, возвращается пустая последовательность.
`s[i]`	Возвращает `i`-й элемент `s` или `len(s)+i`-й, если `i < 0`
`s[i:j:d]`	Срез из последовательности `s` от `i` до `j` с шагом `d` будет рассматриваться ниже
`min(s)`	Наименьший элемент `s`
`max(s)`	Наибольший элемент `s`

Дополнительные конструкции для изменчивых последовательностей:

`s[i] = x`	`i`-й элемент списка `s` заменяется на `x`
`s[i:j:d] = t`	Срез от `i` до `j` (с шагом `d`) заменяется на (список) `t`
`del s[i:j:d]`	Удаление элементов среза из последовательности

Некоторые методы для работы с последовательностями

В таблице приведен ряд методов изменчивых последовательностей (например, списков).

Метод	Описание
`append(x)`	Добавляет элемент в конец последовательности
`count(x)`	Считает количество элементов, равных `x`
`extend(s)`	Добавляет к концу последовательности последовательность `s`
`index(x)`	Возвращает наименьшее `i`, такое, что `s[i] == x`. Возбуждает исключение `ValueError`, если `x` не найден в `s`
`insert(i, x)`	Вставляет элемент `x` в `i`-й промежуток
`pop([i])`	Возвращает `i`-й элемент, удаляя его из последовательности
`reverse()`	Меняет порядок элементов `s` на обратный
`sort([cmpfunc])`	Сортирует элементы `s`. Может быть указана своя функция сравнения `cmpfunc`

Взятие элемента по индексу и срезы

Здесь же следует сказать несколько слов об индексировании последовательностей и выделении подстрок (и вообще - подпоследовательностей) по индексам. Для получения отдельного элемента последовательности используются квадратные скобки, в которых стоит выражение, дающее индекс. Индексы последовательностей в **Python** начинаются с нуля. Отрицательные индексы служат для отсчета элементов с конца последовательности (`-1` - последний элемент). Пример проясняет дело:

```
>>> s = [0, 1, 2, 3, 4]
>>> print s[0], s[-1], s[3]
0 4 3
>>> s[2] = -2
>>> print s
[0, 1, -2, 3, 4]
>>> del s[2]
>>> print s
```

```
[0, 1, 3, 4]
```

Примечание:

Удалять элементы можно только из изменчивых последовательностей и желательно не делать этого внутри цикла по последовательности.

Несколько интереснее обстоят дела со **срезами**. Дело в том, что в **Python** при взятии среза последовательности принято нумеровать не элементы, а промежутки между ними. Поначалу это кажется необычным, тем не менее, очень удобно для указания произвольных срезов. Перед нулевым (по индексу) элементом последовательности промежуток имеет номер **0**, после него - **1** и т.д.. Отрицательные значения отсчитывают промежутки с конца строки. Для записи срезов используется следующий синтаксис:

```
последовательность[нач:кон:шаг]
```

где `нач` - промежуток начала среза, `кон` - конца среза, `шаг` - шаг. По умолчанию `нач=0`, `кон=len(последовательность)`, `шаг=1`, если `шаг` не указан, второе двоеточие можно опустить.

А теперь пример работы со срезами:

```
>>> s = range(10)
>>> s
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> s[0:3]
[0, 1, 2]
>>> s[-1:]
[9]
>>> s[::3]
[0, 3, 6, 9]
>>> s[0:0] = [-1, -1, -1]
>>> s
[-1, -1, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> del s[:3]
>>> s
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Как видно из этого примера, с помощью срезов удобно задавать любую подстроку, даже если она нулевой длины, как для удаления элементов, так и для вставки в строго определенное место.

Тип dict

Словарь (хэш, ассоциативный массив) - это изменчивая структура данных для хранения пар ключ-значение, где значение однозначно определяется ключом. В качестве ключа может выступать неизменчивый тип данных (число, строка, кортеж и т.п.). Порядок пар ключ-значение произволен. Ниже приведен литерал для словаря и пример работы со словарем:

```
d = {1: 'one', 2: 'two', 3: 'three', 4: 'four'}
d0 = {0: 'zero'}
print d[1]        # берется значение по ключу
d[0] = 0          # присваивается значение по ключу
del d[0]          # удаляется пара ключ-значение с данным ключом
print d
for key, val in d.items():   # цикл по всему словарю
  print key, val
for key in d.keys():   # цикл по ключам словаря
  print key, d[key]
for val in d.values():   # цикл по значениям словаря
  print val
```

```
d.update(d0)  # пополняется словарь из другого
print len(d)  # количество пар в словаре
```

Тип file

Объекты этого типа предназначены для работы с внешними данными. В простом случае - это файл на диске. Файловые объекты должны поддерживать основные методы: `read()`, `write()`, `readline()`, `readlines()`, `seek()`, `tell()`, `close()` и т.п.

Следующий пример показывает копирование файла:

```
f1 = open("file1.txt", "r")
f2 = open("file2.txt", "w")
for line in f1.readlines():
  f2.write(line)
f2.close()
f1.close()
```

Стоит заметить, что кроме собственно файлов в **Python** используются и файлоподобные объекты. В очень многих функциях просто неважно, передан ли ей объект типа `file` или другого типа, если он имеет все те же методы (и в том же смысле). Например, копирование содержимого по ссылке **(URL)** в файл `file2.txt` можно достигнуть, если заменить первую строку на

```
import urllib
f1 = urllib.urlopen("http://python.onego.ru")
```

О модулях, классах, объектах и функциях будет говориться на других лекциях.

Выражения

В современных языках программирования принято производить большую часть обработки данных в **выражениях**. Синтаксис выражений у многих языков программирования примерно одинаков. Синтаксис выражений **Python** не удивит программиста чем-то новым. (Разве что цепочечные сравнения могут приятно порадовать.)

Приоритет операций показан в нижеследующей таблице (в порядке уменьшения). Для унарных операций `x` обозначает операнд. **Ассоциативность операций** в **Python** - слева-направо, за исключением операции возведения в степень (`**`), которая ассоциативна справа налево.

Операция	Название
`lambda`	лямбда-выражение
`or`	логическое ИЛИ
`and`	логическое И
`not x`	логическое НЕ
`in, not in`	проверка принадлежности
`is, is not`	проверка идентичности
`<,<=,>,>=,!=,==`	сравнения
`|`	побитовое ИЛИ
`^`	побитовое исключающее ИЛИ
`&`	побитовое И
`<<, >>`	побитовые сдвиги
`+, -`	сложение и вычитание

`*, /, %`	умножение, деление, остаток
`+x, -x`	унарный плюс и смена знака
`~x`	побитовое НЕ
`**`	возведение в степень
`x.атрибут`	ссылка на атрибут
`x[индекс]`	взятие элемента по индексу
`x[от:до]`	выделение среза (от и до)
`f(аргумент,...)`	вызов функции
`( ... )`	скобки или кортеж
`[ ... ]`	список или списковое включение
`{кл:зн, ...}`	словарь пар ключ-значение
`` `выражения` ``	преобразование к строке **(repr)**

Таким образом, порядок вычислений операндов определяется такими правилами:

1. Операнд слева вычисляется раньше операнда справа во всех бинарных операциях, кроме возведения в степень.
2. Цепочка сравнений вида `a < b < c ... y < z` фактически равносильна: `(a < b) and (b < c) and ... and (y < z)`.
3. Перед фактическим выполнением операции вычисляются нужные для нее операнды. В большинстве бинарных операций предварительно вычисляются оба операнда (сначала левый), но операции `or` и `and` , а также цепочки сравнений вычисляют такое количество операндов, которое достаточно для получения результата. В невычисленной части выражения в таком случае могут даже быть неопределенные имена. Это важно учитывать, если используются функции с побочными эффектами.
4. Аргументы функций, выражения для списков, кортежей, словарей и т.п. вычисляются слева-направо, в порядке следования в выражении.

В случае неясности приоритетов желательно применять скобки. Несмотря на то, что одни и те же символы могут использоваться для разных операций, приоритеты операций не меняются. Так, `%` имеет тот же приоритет, что и `*`, а потому в следующем примере скобки просто необходимы, чтобы операция умножения произошла перед операцией форматирования:

```
print "%i" % (i*j)
```

Выражения могут фигурировать во многих операторах **Python** и даже как самостоятельный оператор. У выражения всегда есть результат, хотя в некоторых случаях (когда выражение вычисляется ради побочных эффектов) этот результат может быть "ничем" - `None`.

Очень часто выражения стоят в правой части оператора присваивания или расширенного присваивания. В **Python** (в отличие, скажем, от **C**) нет операции присваивания, поэтому синтаксически перед знаком = могут стоять только идентификатор, индекс, срез, доступ к атрибуту или кортеж (список) из перечисленного. (Подробности в документации).

Имена

Об именах (идентификаторах) говорилось уже не раз, тем не менее, необходимо сказать несколько слов об их применении в языке **Python**.

Имя может начинаться с латинской буквы (любого регистра) или подчеркивания, а дальше допустимо использование цифр. В качестве идентификаторов нельзя применять ключевые слова языка и нежелательно переопределять встроенные имена. Список ключевых слов можно узнать так:

```
>>> import keyword
>>> keyword.kwlist
['and', 'assert', 'break', 'class', 'continue', 'def', 'del',
'elif', 'else', 'except', 'exec', 'finally', 'for', 'from',
'global', 'if', 'import', 'in', 'is', 'lambda', 'not', 'or',
'pass', 'print', 'raise', 'return', 'try', 'while', 'yield']
```

Имена, начинающиеся с подчеркивания или двух подчеркиваний, имеют особый смысл. Одиночное подчеркивание говорит программисту о том, что имя имеет местное применение, и не должно использоваться за пределами модуля. Двойным подчеркиванием в начале и в конце обычно наделяются специальные имена атрибутов - об этом будет говориться в лекции по объектно-ориентированному программированию.

В каждой точке программы интерпретатор "видит" три **пространства имен**: локальное, глобальное и встроенное. Пространство имен - отображение из имен в объекты.

Для понимания того, как **Python** находит значение некоторой переменной, необходимо ввести понятие **блока кода**. В **Python** блоком кода является то, что исполняется как единое целое, например, тело определения функции, класса или модуля.

Локальные имена - имена, которым присвоено значение в данном блоке кода. Глобальные имена - имена, определяемые на уровне блока кода определения модуля или те, которые явно заданы в операторе `global`. Встроенные имена - имена из специального словаря `__builtins__`.

Области видимости имен могут быть вложенными друг в друга, например, внутри вызванной функции видны имена, определенные в вызывающем коде. Переменные, которые используются в блоке кода, но связаны со значением вне кода, называются **свободными переменными**.

Так как переменную можно связать с объектом в любом месте блока, важно, чтобы это произошло до ее использования, иначе будет возбуждено исключение `NameError`. Связывание имен со значениями происходит в операторах присваивания, `for`, `import`, в формальных аргументах функций, при определении функции или класса, во втором параметре части `except` оператора `try-except`.

С областями видимости и связыванием имен есть много нюансов, которые хорошо описаны в документации. Желательно, чтобы программы не зависели от таких нюансов, а для этого достаточно придерживаться следующих правил:

1. Всегда следует связывать переменную со значением (текстуально) до ее использования.
2. Необходимо избегать глобальных переменных и передавать все в качестве параметров. Глобальными на уровне модуля должны остаться только имена-константы, имена классов и функций.
3. Никогда не следует использовать `from` модуль `import *` - это может привести к затенению имен из других модулей, а внутри определения функции просто запрещено.

Предпочтительнее переделать код, нежели использовать глобальную переменную Конечно, для программ, состоящих из одного модуля, это не так важно: ведь все определенные на уровне модуля переменные глобальны.

Убрать связь имени с объектом можно с помощью оператора `del`. В этом случае, если объект не имеет других ссылок на него, он будет удален. Для управления памятью в **Python** используется **подсчет ссылок (reference counting)**, для удаления наборов объектов с зацикленными ссылками - **сборка мусора (garbage collection)**.

Стиль программирования

Стиль программирования - дополнительные ограничения, накладываемые на структуру и вид программного кода группой совместно работающих программистов с целью получения удобных для применения, легко читаемых и эффективных программ. Основные ограничения на вид программы дает синтаксис языка программирования, и его нарушения вызывают синтаксические ошибки. Нарушение стиля не приводит к синтаксическим ошибкам, однако как отдельные программисты, так и целые коллективы сознательно ограничивают себя в средствах выражения ради упрощения совместной разработки, отладки и сопровождения программного продукта.

Стиль программирования затрагивает практически все аспекты написания кода:

- именование объектов в зависимости от типа, назначения, области видимости;
- оформление функций, методов, классов, модулей и их документирование в коде программы;
- декомпозиция программы на модули с определенными характеристиками;
- способ включения отладочной информации;
- применение тех или иных функций (методов) в зависимости от предполагаемого уровня совместимости разрабатываемой программы с различными компьютерными платформами;
- ограничение используемых функций из соображений безопасности.

Для языка **Python** Гвидо ван Россум разработал официальный стиль. С оригинальным текстом **"Python Style Guide"** можно ознакомиться по адресу **http://www.python.org/doc/essays/styleguide.html**.

Наиболее существенные положения этого стиля перечислены ниже. В случае сомнений хорошим образцом стиля являются модули стандартной библиотеки.

- Рекомендуется использовать отступы в **4** пробела.
- Длина физической строки не должна превышать **79** символов.
- Длинные логические строки лучше разбивать неявно (внутри скобок), но и явные методы вполне уместны. Отступы строк продолжения рекомендуется выравнивать по скобкам или по первому операнду в предыдущей строке. Текстовый редактор **Emacs** в режиме **python-mode** и некоторые интегрированные оболочки **(IDE)** автоматически делают необходимые отступы в **Python-**программах:

```
def draw(figure, color="White", border_color="Black",
         size=5):
    if color == border_color or \
       size == 0:
        raise "Bad figure"
    else:
        _draw(size, size, (color,
                           border_color))
```

- Не рекомендуется ставить пробелы сразу после открывающей скобки или перед закрывающей, перед запятой, точкой с запятой, перед открывающей скобкой при записи вызова функции или индексного выражения. Также не рекомендуется ставить более одного пробела вокруг знака равенства в присваиваниях. Пробелы вокруг знака равенства не ставятся в случае, когда он применяется для указания значения по умолчанию в определении параметров функции или при задании именованных аргументов.
- Также рекомендуется применение одиночных пробелов вокруг низкоприоритетных операций сравнения и оператора присваивания. Пробелы вокруг более приоритетных операций ставятся в равном количестве слева и справа от знака операции.

Несколько рекомендаций касаются написания комментариев.

- Комментарии должны точно отражать актуальное состояние кода. (Поддержание актуальных комментариев должно быть приоритетной задачей!) После коротких

комментариев можно не ставить точку, тогда как длинные лучше писать по правилам написания текста. Автор **Python** обращается к неанглоязычным программистам с просьбой писать комментарии на английском, если есть хотя бы небольшая вероятность того, что код будут читать специалисты, говорящие на других языках.
- Комментарии к фрагменту кода следует писать с тем же отступом, что и комментируемый код. После "`#`" должен идти одиночный пробел. Абзацы можно отделять строкой с "`#`" на том же уровне. Блочный комментарий можно отделить пустыми строками от окружающего кода.
- Комментарии, относящиеся к конкретной строке, не следует использовать часто. Символ "`#`" должен отстоять от комментируемого оператора как минимум на два пробела.
- Хороший комментарий не перефразирует программу, а содержит дополнительную информацию о действии программы в терминах предметной области.

Все модули, классы, функции и методы, предназначенные для использования за пределами модуля, должны иметь строки документации, описывающие способ их применения, входные и выходные параметры.

- Строка документации для отдельной программы должна объяснять используемые ею ключи, назначение аргументов и переменных среды и другую подобную информацию.
- Для строк документации рекомендуется везде использовать утроенные кавычки (`"""`).
- Однострочная документация пишется в императиве, как команда: "делай это", "возвращай то".
- Многострочная документация содержит расширенное описание модуля, функции, класса. Она будет смотреться лучше, если текст будет написан с тем же отступом, что и начало строки документации.
- Документация для модуля должна перечислять экспортируемые функции, классы, исключения и другие объекты, по одной строке на объект.
- Строка документации для функции или метода должна кратко описывать действия функции, ее входные параметры и возвращаемое значение, побочные эффекты и возможные исключения (если таковые есть). Должны быть обозначены необязательные аргументы и аргументы, не являющиеся частью интерфейса.
- Документация для класса должна перечислять общедоступные методы и атрибуты, содержать рекомендации по применению класса в качестве базового для других классов. Если класс является подклассом, необходимо указать, какие методы полностью заменяют, перегружают, а какие используют, но расширяют соответствующие методы надкласса. Необходимо указать и другие изменения по сравнению с надклассом.
- Контроль версий повышает качество процесса создания программного обеспечения. Для этих целей часто используются **RCS** или **CVS**. **"Python Style Guide"** рекомендует записывать `$Revision: 1.31 $` в переменную с именем `__version__`, а другие данные заключать в комментарии "`#`".

Сегодня сосуществуют несколько более или менее широко распространенных правил именования объектов. Программисты вольны выбрать тот, который принят в их организации или конкретном проекте. Автор **Python** рекомендует придерживаться нижеследующих правил для именования различных объектов, с тем чтобы это было понятно любому программисту, использующему **Python**.

- Имена модулей лучше давать строчными буквами, например, `shelve`, `string`, либо делать первые буквы слов прописными, `StringIO`, `UserDict`. Имена написанных на **C** модулей расширения обычно начинаются с подчеркивания "`_`", а соответствующие им высокоуровневые обертки - с прописных букв: `_tkinter` и `Tkinter`.
- Ключевые слова нельзя использовать в качестве имен, однако, если все-таки необходимо воспользоваться этим именем, стоит добавить одиночное подчеркивание в конце имени. Например: `class_`.

- Классы обычно называют, выделяя первые буквы слов прописными, как в `Tag` или `HTTPServer`.
- Имена исключений обычно содержат в своем составе слово **"error"** (или **"warning"**). Встроенные модули пишут это слово со строчной буквы (как `os.error`) (но могут писать и с прописной): `distutils.DistutilsModuleError`.
- Функции, экспортируемые модулем, могут именоваться по-разному. Можно давать с прописных букв имена наиболее важных функций, а вспомогательные писать строчными.
- Имена глобальных переменных (если таковые используются) лучше начинать с подчеркивания, чтобы они не импортировались из модуля оператором `from-import` со звездочкой.
- Имена методов записываются по тем же правилам, что и имена функций.
- Имена констант (имен, которые не должны переопределяться) лучше записывать прописными буквами, например: `RED`, `GREEN`, `BLUE`.
- При работе с языком **Python** необходимо учитывать, что интерпретатор считает некоторые классы имен специальными (обычно такие имена начинаются с подчеркивания).

Заключение

В этой лекции синтаксис языка показан на примерах, что в случае с **Python** оправдано, так как эта часть языка достаточна проста. Были рассмотрены основные операторы языка, выражения и многие из встроенных типов данных, кратко объяснены принципы работы **Python** с именами, приведены правила официального стиля программирования на **Python**.

Лекция #2: Основные стандартные модули Python

Одним из важных преимуществ языка **Python** является наличие большой библиотеки модулей и пакетов, входящих в стандартную поставку. Как говорят, к **Python** "приложены батарейки".

Понятие модуля

Перед тем как приступить к изучению моделей стандартной библиотеки, необходимо определить то, что в **Python** называется **модулем**.

В соответствии с модульным подходом к программированию большая задача разбивается на несколько более мелких, каждую из которых (в идеале) решает отдельный модуль. В разных методологиях даются различные ограничения на размер модулей, однако при построении модульной структуры программы важнее составить такую композицию модулей, которая позволила бы свести к минимуму связи между ними. Набор классов и функций, имеющий множество связей между своими элементами, было бы логично расположить в одном модуле. Есть и еще одно полезное замечание: модули должно быть легче использовать, чем написать заново. Это значит, что модуль должен иметь удобный **интерфейс**: набор функций, классов и констант, который он предлагает своим пользователям.

В языке **Python** набор модулей, посвященных одной проблеме, можно поместить в **пакет**. Хорошим примером такого пакета является пакет `xml`, в котором собраны модули для различных аспектов обработки **XML**.

В программе на **Python** модуль представлен объектом-модулем, атрибутами которого являются имена, определенные в модуле:

```
>>> import datetime
>>> d1 = datetime.date(2004, 11, 20)
```

В данном примере импортируется модуль `datetime`. В результате работы оператора `import` в текущем пространстве имен появляется объект с именем `datetime`.

Модули для использования в программах на языке **Python** по своему происхождению делятся на обычные (написанные на **Python**) и модули расширения, написанные на другом языке программирования (как правило, на **C**). С точки зрения пользователя они могут отличаться разве что быстродействием. Бывает, что в стандартной библиотеке есть два варианта модуля: на **Python** и на **C**. Таковы, например, модули `pickle` и `cPickle`. Обычно модули на **Python** в чем-то гибче, чем модули расширения.

Модули в Python

Модуль оформляется в виде отдельного файла с исходным кодом. Стандартные модули находятся в каталоге, где их может найти соответствующий интерпретатор языка. Пути к каталогам, в которых **Python** ищет модули, можно увидеть в значении переменной `sys.path`:

```
>>> sys.path
['', '/usr/local/lib/python23.zip', '/usr/local/lib/python2.3',
'/usr/local/lib/python2.3/plat-linux2', '/usr/local/lib/python2.3/lib-tk',
'/usr/local/lib/python2.3/lib-dynload',
'/usr/local/lib/python2.3/site-packages']
```

В последних версиях **Python** модули можно помещать и в **zip**-архивы для более компактного хранения (по аналогии с **jar**-архивами в **Java**).

При запуске программы поиск модулей также идет в текущем каталоге. (Нужно внимательно называть собственные модули, чтобы не было конфликта имен со стандартными или дополнительно установленными модулями.)

Подключение модуля к программе на **Python** осуществляется с помощью оператора `import`. У него есть две формы: `import` и `from-import`:

```
import os
import pre as re
from sys import argv, environ
from string import *
```

С помощью первой формы с текущей областью видимости связывается только имя, ссылающееся на объект модуля, а при использовании второй - указанные имена (или все имена, если применена `*`) объектов модуля связываются с текущей областью видимости. При импорте можно изменить имя, с которым объект будет связан, с помощью `as`. В первом случае пространство имен модуля остается в отдельном имени и для доступа к конкретному имени из модуля нужно применять точку. Во втором случае имена используются так, как если бы они были определены в текущем модуле:

```
os.system("dir")
digits = re.compile("\d+")
print argv[0], environ
```

Повторный импорт модуля происходит гораздо быстрее, так как модули кэшируются интерпретатором. Загруженный модуль можно загрузить еще раз (например, если модуль изменился на диске) с помощью функции `reload()`:

```
import mymodule
. . .
reload(mymodule)
```

Однако в этом случае все объекты, являющиеся экземплярами классов из старого варианта модуля, не изменят своего поведения.

При работе с модулями есть и другие тонкости. Например, сам процесс импорта модуля можно переопределить. Подробнее об этом можно узнать в оригинальной документации.

Встроенные функции

В среде **Python** без дополнительных операций импорта доступно более сотни встроенных объектов, в основном, функций и исключений. Для удобства функции условно разделены по категориям:

1. Функции преобразования типов и классы: `coerce`, `str`, `repr`, `int`, `list`, `tuple`, `long`, `float`, `complex`, `dict`, `super`, `file`, `bool`, `object`
2. Числовые и строковые функции: `abs`, `divmod`, `ord`, `pow`, `len`, `chr`, `unichr`, `hex`, `oct`, `cmp`, `round`, `unicode`
3. Функции обработки данных: `apply`, `map`, `filter`, `reduce`, `zip`, `range`, `xrange`, `max`, `min`, `iter`, `enumerate`, `sum`
4. Функции определения свойств: `hash`, `id`, `callable`, `issubclass`, `isinstance`, `type`
5. Функции для доступа к внутренним структурам: `locals`, `globals`, `vars`, `intern`, `dir`
6. Функции компиляции и исполнения: `eval`, `execfile`, `reload`, `__import__`, `compile`
7. Функции ввода-вывода: `input`, `raw_input`, `open`
8. Функции для работы с атрибутами: `getattr`, `setattr`, `delattr`, `hasattr`
9. Функции-"украшатели" методов классов: `staticmethod`, `classmethod`, `property`
10. Прочие функции: `buffer`, `slice`

Совет:

Уточнить назначение функции, ее аргументов и результата можно в интерактивной сессии интерпретатора **Python**:

```
>>> help(len)
Help on built-in function len:
len(...)
  len(object) -> integer
  Return the number of items of a sequence or mapping.
```

Или так:

```
>>> print len.__doc__
len(object) -> integer
Return the number of items of a sequence or mapping.
```

Функции преобразования типов и классы

Функции и классы из этой категории служат для преобразования типов данных. В старых версиях **Python** для преобразования к нужному типу использовалась одноименная функция. В новых версиях **Python** роль таких функций играют имена встроенных классов (однако семантика не изменилась). Для понимания сути достаточно небольшого примера:

```
>>> int(23.5)
23
>>> float('12.345')
12.345000000000001
>>> dict([('a', 2), ('b', 3)])
{'a': 2, 'b': 3}
>>> object
<type 'object'>
>>> class MyObject(object):
...   pass
...
```

Числовые и строковые функции

Функции работают с числовыми или строковыми аргументами. В следующей таблице даны описания этих функций.

`abs(x)`	Модуль числа `x`. Результат: `|x|`.
`divmod(x, y)`	Частное и остаток от деления. Результат: (частное, остаток).
`pow(x, y[, m])`	Возведение `x` в степень `y` по модулю `m`. Результат: `x**y % m`.
`round(n[, z])`	Округление чисел до заданного знака после (или до) точки.
`ord(s)`	Функция возвращает код (или **Unicode**) заданного ей символа в односимвольной строке.
`chr(n)`	Возвращает строку с символом с заданным кодом.
`len(s)`	Возвращает число элементов последовательности или отображения.
`oct(n)`, `hex(n)`	Функции возвращают строку с восьмеричным или шестнадцатеричным представлением целого числа `n`.
`cmp(x, y)`	Сравнение двух значений. Результат: отрицательный, ноль или положительный, в зависимости от результата сравнения.
`unichr(n)`	Возвращает односимвольную **Unicode**-строку с символом с кодом `n`.
`unicode(s, [,`	Создает **Unicode**-объект, соответствующий строке `s` в заданной кодировке

encoding[, errors]])	**encoding.** Ошибки кодирования обрабатываются в соответствии с **errors,** который может принимать значения: 'strict' **(**строгое преобразование**),** 'replace' **(**с заменой несуществующих символов**)** или 'ignore' **(**игнорировать несуществующие символы**).** По умолчанию: encoding='utf-8', errors='strict'.

Следующий пример строит таблицу кодировки кириллических букв в **Unicode:**

```
print "Таблица Unicode (русские буквы)".center(18*4)
i = 0
for c in "АБВГДЕЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯ"\
         "абвгдежзийклмнопрстуфхцчшщъыьэюя":
  u = unicode(c, 'koi8-r')
  print "%3i: %1s %s" % (ord(u), c, `u`),
  i += 1
  if i % 4 == 0:
    print
```

Функции обработки данных

Эти функции подробнее будут рассмотрены в лекции по функциональному программированию. Пример с функциями range() и enumerate():

```
>>> for i, c in enumerate("ABC"):
...    print i, c
...
0 A
1 B
2 C
>>> print range(4, 20, 2)
[4, 6, 8, 10, 12, 14, 16, 18]
```

Функции определения свойств

Эти функции обеспечивают доступ к некоторым встроенным атрибутам объектов и другим свойствам. Следующий пример показывает некоторые из этих функций:

```
>>> s = "abcde"
>>> s1 = "abcde"
>>> s2 = "ab" + "cde"
>>> print "hash:", hash(s), hash(s1), hash(s2)
hash: -1332677140 -1332677140 -1332677140
>>> print "id:", id(s), id(s1), id(s2)
id: 1076618592 1076618592 1076618656
```

Здесь, можно увидеть, что для одного и того же строкового литерала "abcde" получается один и тот же объект, тогда как для одинаковых по значению объектов вполне можно получить разные объекты.

Функции для доступа к внутренним структурам

В современной реализации языка **Python** глобальные и локальные переменные доступны в виде словаря благодаря функциям globals() и locals(). Правда, записывать что-либо в эти словари не рекомендуется.

Функция vars() возвращает таблицу локальных имен некоторого объекта **(**если параметр не задан, она возвращает то же, что и locals()**).** Обычно используется в качестве словаря для операции форматирования:

```
a = 1
```

```
b = 2
c = 3
print "%(a)s + %(b)s = %(c)s" % vars()
```

Функции компиляции и исполнения

Функция `reload()` уже рассматривалась, а из остальных функций этой категории особого внимания заслуживает `eval()`. Как следует из названия, эта функция вычисляет переданное ей выражение. В примере ниже вычисляется выражение, которое строится динамически:

```
a = 2
b = 3
for op in "+-*/%":
  e = "a " + op + " b"
  print e, "->", eval(e)
```

У функции `eval()` кроме подлежащего вычислению выражения есть еще два параметра - с их помощью можно задать глобальное и локальное пространства имен, из которых будут разрешаться имена выражения. Пример выше, переписанный для использования с собственным словарем имен в качестве глобального пространства имен:

```
for op in "+-*/%":
  e = "a " + op + " b"
  print e, "->", eval(e, {'a': 2, 'b': 3})
```

Функцией `eval()` легко злоупотребить. Нужно стараться использовать ее только тогда, когда без нее не обойтись. Из соображений безопасности не следует применять `eval()` для аргумента, в котором присутствует непроверенный ввод от пользователя.

Функции ввода-вывода

Функции `input()` и `raw_input()` используются для ввода со стандартного ввода. В серьезных программах их лучше не применять. Функция `open()` служит для открытия файла по имени для чтения, записи или изменения. В следующем примере файл открывается для чтения:

```
f = open("file.txt", "r", 1)
for line in f:
  . . .
f.close()
```

Функция принимает три аргумента: имя файла (путь к файлу), режим открытия ("`r`" - чтение, "`w`" - запись, "`a`" - добавление или "`w+`", "`a+`", "`r+`" - изменение. Также может прибавляться "`t`", что обозначает текстовый файл. Это имеет значение только на платформе **Windows**). Третий аргумент указывает режим буферизации: `0` - без буферизации, `1` - построчная буферизация, больше `1` - буфер указанного размера в байтах.

В новых версиях **Python** функция `open()` является синонимом для `file()`.

Функции для работы с атрибутами

У объектов в языке **Python** могут быть атрибуты (в терминологии языка **C++** - члены-данные и члены-функции). Следующие две программы эквивалентны:

```
# первая программа:
class A:
  pass
a = A()
```

```
a.attr = 1
try:
  print a.attr
except:
  print None
del a.attr

# вторая программа:
class A:
  pass
a = A()
setattr(a, 'attr', 1)
if hasattr(a, 'attr'):
  print getattr(a, 'attr')
else:
  print None
delattr(a, 'attr')
```

Функции-"украшатели" методов классов

Эти функции будут рассмотрены в лекции, посвященной ООП.

Обзор стандартной библиотеки

Модули стандартной библиотеки можно условно разбить на группы по тематике.

1. Сервисы периода выполнения. Модули: `sys`, `atexit`, `copy`, `traceback`, `math`, `cmath`, `random`, `time`, `calendar`, `datetime`, `sets`, `array`, `struct`, `itertools`, `locale`, `gettext`.
2. Поддержка цикла разработки. Модули: `pdb`, `hotshot`, `profile`, `unittest`, `pydoc`. Пакеты `docutils`, `distutils`.
3. Взаимодействие с ОС **(**файлы, процессы**)**. Модули: `os`, `os.path`, `getopt`, `glob`, `popen2`, `shutil`, `select`, `signal`, `stat`, `tempfile`.
4. Обработка текстов. Модули: `string`, `re`, `StringIO`, `codecs`, `difflib`, `mmap`, `sgmllib`, `htmllib`, `htmlentitydefs`. Пакет `xml`.
5. Многопоточные вычисления. Модули: `threading`, `thread`, `Queue`.
6. Хранение данных. Архивация. Модули: `pickle`, `shelve`, `anydbm`, `gdbm`, `gzip`, `zlib`, `zipfile`, `bz2`, `csv`, `tarfile`.
7. Платформо-зависимые модули. Для **UNIX**: `commands`, `pwd`, `grp`, `fcntl`, `resource`, `termios`, `readline`, `rlcompleter`. Для **Windows**: `msvcrt`, `_winreg`, `winsound`.
8. Поддержка сети. Протоколы Интернет. Модули: `cgi`, `Cookie`, `urllib`, `urlparse`, `httplib`, `smtplib`, `poplib`, `telnetlib`, `socket`, `asyncore`. Примеры серверов: `SocketServer`, `BaseHTTPServer`, `xmlrpclib`, `asynchat`.
9. Поддержка **Internet**. Форматы данных. Модули: `quopri`, `uu`, `base64`, `binhex`, `binascii`, `rfc822`, `mimetools`, `MimeWriter`, `multifile`, `mailbox`. Пакет `email`.
10. **Python** о себе. Модули: `parser`, `symbol`, `token`, `keyword`, `inspect`, `tokenize`, `pyclbr`, `py_compile`, `compileall`, `dis`, `compiler`.
11. Графический интерфейс. Модуль `Tkinter`.

Примечание:

Очень часто модули содержат один или несколько классов, с помощью которых создается объект нужного типа, а затем речь идет уже не об именах из модуля, а об атрибутах этого объекта. И наоборот, некоторые модули содержат лишь функции, слишком общие для того, чтобы работать над произвольными объектами **(**либо достаточно большой категорией объектов**)**.

Сервисы периода выполнения

Модуль sys

Модуль **sys** содержит информацию о среде выполнения программы, об интерпретаторе **Python**. Далее будут представлены наиболее популярные объекты из этого модуля: остальное можно изучить по документации.

`exit([c])`	Выход из программы. Можно передать числовой код завершения: `0` в случае успешного завершения, другие числа при аварийном завершении программы.
`argv`	Список аргументов командной строки. Обычно `sys.argv[0]` содержит имя запущенной программы, а остальные параметры передаются из командной строки.
`platform`	Платформа, на которой работает интерпретатор.
`stdin, stdout, stderr`	Стандартный ввод, вывод, вывод ошибок. Открытые файловые объекты.
`version`	Версия интерпретатора.
`setrecursionlimit(limit)`	Установка уровня максимальной вложенности рекурсивных вызовов.
`exc_info()`	Информация об обрабатываемом исключении.

Модуль copy

Этот модуль содержит функции для копирования объектов. Вначале предлагается к рассмотрению "парадокс", который вводит в замешательство новичков в **Python**:

```
lst1 = [0, 0, 0]
lst = [lst1] * 3
print lst
lst[0][1] = 1
print lst
```

В результате получается, возможно, не то, что ожидалось:

```
[[0, 0, 0], [0, 0, 0], [0, 0, 0]]
[[0, 1, 0], [0, 1, 0], [0, 1, 0]]
```

Дело в том, что список `lst` содержит ссылки на один и тот же список! Для того чтобы действительно размножить список, необходимо применить функцию `copy()` из модуля `copy`:

```
from copy import copy
lst1 = [0, 0, 0]
lst = [copy(lst1) for i in range(3)]
print lst
lst[0][1] = 1
print lst
```

Теперь результат тот, который ожидался:

```
[[0, 0, 0], [0, 0, 0], [0, 0, 0]]
[[0, 1, 0], [0, 0, 0], [0, 0, 0]]
```

В модуле **copy** есть еще и функция `deepcopy()` для глубокого копирования, при которой объекты копируются на всю возможную глубину, рекурсивно.

Модули math и cmath

В этих модулях собраны математические функции для действительных и комплексных аргументов. Это те же функции, что используются в языке **C**. В таблице ниже даны функции модуля **math**. Там, где аргумент обозначен буквой **z**, аналогичная функция определена и в модуле **cmath**.

Функция или константа	Описание
`acos(z)`	арккосинус `z`
`asin(z)`	арксинус `z`
`atan(z)`	арктангенс `z`
`atan2(y,x)`	`atan(y/x)`
`ceil(x)`	наименьшее целое, большее или равное `x`
`cos(z)`	косинус `z`
`cosh(x)`	гиперболический косинус `x`
`e`	константа `e`
`exp(z)`	экспонента (то есть, `e**z`)
`fabs(x)`	абсолютное значение `x`
`floor(x)`	наибольшее целое, меньшее или равное `x`
`fmod(x,y)`	остаток от деления `x` на `y`
`frexp(x)`	возвращает мантиссу и порядок `x` как пару `(m, i)`, где `m` - число с плавающей точкой, а `i` - целое, такое, что `x = m * 2.**i`. Если `0`, возвращает `(0,0)`, иначе `0.5 <= abs(m) < 1.0`
`hypot(x,y)`	`sqrt(x*x + y*y)`
`ldexp(m,i)`	`m * (2**i)`
`log(z)`	натуральный логарифм `z`
`log10(z)`	десятичный логарифм `z`
`modf(x)`	возвращает пару `(y,q)` - целую и дробную часть `x`. Обе части имеют знак исходного числа
`pi`	константа пи
`pow(x,y)`	`x**y`
`sin(z)`	синус `z`
`sinh(z)`	гиперболический синус `z`
`sqrt(z)`	корень квадратный от `z`
`tan(z)`	тангенс `z`
`tanh(z)`	гиперболический тангенс `z`

Модуль random

Этот модуль генерирует псевдослучайные числа для нескольких различных распределений. Наиболее используемые функции:

`random()`	Генерирует псевдослучайное число из полуоткрытого диапазона `[0.0, 1.0)`.
`choice(s)`	Выбирает случайный элемент из последовательности `s`.
`shuffle(s)`	Размешивает элементы изменчивой последовательности `s` на месте.

`randrange([start,] stop[, step])`	Выдает случайное целое число из диапазона `range(start, stop, step)`. Аналогично `choice(range(start, stop, step))`.
`normalvariate(mu, sigma)`	Выдает число из последовательности нормально распределенных псевдослучайных чисел. Здесь `mu` - среднее, `sigma` - среднеквадратическое отклонение `(sigma > 0)`

Остальные функции и их параметры можно уточнить по документации. Следует отметить, что в модуле есть функция `seed(n)`, которая позволяет установить генератор случайных чисел в некоторое состояние. Например, если возникнет необходимость многократного использования одной и той же последовательности псевдослучайных чисел.

Модуль time

Этот модуль дает функции для получения текущего времени и преобразования форматов времени.

Модуль sets

Модуль реализует тип данных для множеств. Следующий пример показывает, как использовать этот модуль. Следует заметить, что в **Python 2.4** и старше тип `set` стал встроенным, и вместо `sets.Set` можно использовать `set`:

```
import sets
A = sets.Set([1, 2, 3])
B = sets.Set([2, 3, 4])
print A | B, A & B, A - B, A ^ B
for i in A:
  if i in B:
    print i,
```

В результате будет выведено:

```
Set([1, 2, 3, 4]) Set([2, 3]) Set([1]) Set([1, 4])
2 3
```

Модули array и struct

Эти модули реализуют низкоуровневый массив и структуру данных. Основное их назначение - разбор двоичных форматов данных.

Модуль itertools

Этот модуль содержит набор функций для работы с **итераторами**. Итераторы позволяют работать с данными последовательно, как если бы они получались в цикле. Альтернативный подход - использование списков для хранения промежуточных результатов - требует подчас большого количества памяти, тогда как использование итераторов позволяет получать значения на момент, когда они действительно требуются для дальнейших вычислений. Итераторы будут рассмотрены более подробно в лекции по функциональному программированию.

Модуль locale

Модуль `locale` применяется для работы с культурной средой. В конкретной культурной среде могут использоваться свои правила для написания чисел, валют, времени и даты и т.п. Следующий пример выводит дату сначала в культурной среде **"C"**, а затем на русском языке:

```
import time, locale
```

```
locale.setlocale(locale.LC_ALL, None)
print time.strftime("%d %B %Y", time.localtime (time.time()))
locale.setlocale(locale.LC_ALL, "ru_RU.KOI8-R")
print time.strftime("%d %B %Y", time.localtime (time.time()))
```

В результате:

```
18 November 2004
18 Ноября 2004
```

Модуль gettext

При интернационализации программы важно не только предусмотреть возможность использования нескольких культурных сред, но и перевод сообщений и меню программы на соответствующий язык. Модуль gettext позволяет упростить этот процесс достаточно стандартным способом. Основные сообщения программы пишутся на английском языке. А переводы строк, отмеченных в программе специальным образом, даются в виде отдельных файлов, по одному на каждый язык (или культурную среду). Уточнить нюансы использования gettext можно по документации к **Python**.

Поддержка цикла разработки

Модули этого раздела помогают поддерживать документацию, производить регрессионное тестирование, отлаживать и профилировать программы на **Python**, а также обслуживают распространение готовых программ, создавая среду для конфигурирования и установки пакетов.

В качестве иллюстрации можно предположить, что создается модуль для вычисления простых чисел по алгоритму "решето Эратосфена". Модуль будет находиться в файле Sieve.py и состоять из одной функции primes(N), которая в результате своей работы дает все простые (не имеющие натуральных делителей кроме себя и единицы) числа от 2 до N:

```
import sets
import math
"""Модуль для вычисления простых чисел от 2 до N """
def primes(N):
  """Возвращает все простые от 2 до N"""
  sieve = sets.Set(range(2, N))
  for i in range(2, math.sqrt(N)):
    if i in sieve:
      sieve -= sets.Set(range(2*i, N, i))
  return sieve
```

Модуль pdb

Модуль pdb предоставляет функции отладчика с интерфейсом - командной строкой. Сессия отладки вышеприведенного модуля могла бы быть такой:

```
>>> import pdb
>>> pdb.runcall(Sieve.primes, 100)
> /home/rnd/workup/intuit-python/examples/Sieve.py(15)primes()
-> sieve = sets.Set(range(2, N))
(Pdb) l
 10     import sets
 11     import math
 12     """Модуль для вычисления простых чисел от 2 до N """
 13     def primes(N):
 14       """Возвращает все простые от 2 до N"""
 15  ->   sieve = sets.Set(range(2, N))
 16       for i in range(2, int(math.sqrt(N))):
 17         if i in sieve:
```

```
 18           sieve -= sets.Set(range(2*i, N, i))
 19       return sieve
 20
(Pdb) n
> /home/rnd/workup/intuit-python/examples/Sieve.py(16)primes()
-> for i in range(2, int(math.sqrt(N))):
(Pdb) n
> /home/rnd/workup/intuit-python/examples/Sieve.py(17)primes()
-> if i in sieve:
(Pdb) n
> /home/rnd/workup/intuit-python/examples/Sieve.py(18)primes()
-> sieve -= sets.Set(range(2*i, N, i))
(Pdb) n
> /home/rnd/workup/intuit-python/examples/Sieve.py(16)primes()
-> for i in range(2, int(math.sqrt(N))):
(Pdb) p sieve
Set([2, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39,
41, 43, 45, 47, 49, 51, 53, 55, 57, 59, 61, 63, 65, 67, 69, 71, 73, 75, 77, 79,
81, 83, 85, 87, 89, 91, 93, 95, 97, 99])
(Pdb) n
> /home/rnd/workup/intuit-python/examples/Sieve.py(17)primes()
-> if i in sieve:
(Pdb) n
> /home/rnd/workup/intuit-python/examples/Sieve.py(18)primes()
-> sieve -= sets.Set(range(2*i, N, i))
(Pdb) n
> /home/rnd/workup/intuit-python/examples/Sieve.py(16)primes()
-> for i in range(2, int(math.sqrt(N))):
(Pdb) p sieve
Set([2, 3, 5, 7, 11, 13, 17, 19, 23, 25, 29, 31, 35, 37, 41, 43, 47, 49,
53, 55, 59, 61, 65, 67, 71, 73, 77, 79, 83, 85, 89, 91, 95, 97])
```

Модуль profile

С помощью **профайлера** разработчики программного обеспечения могут узнать, сколько времени занимает исполнение различных функций и методов.

Продолжая пример с решетом Эратосфена, стоит посмотреть, как тратится процессорное время при вызове функции `primes()`:

```
>>> profile.run("Sieve.primes(100000)")
        709 function calls in 1.320 CPU seconds

  Ordered by: standard name

  ncalls  tottime  percall  cumtime  percall filename:lineno(function)
       1    0.010    0.010    1.320    1.320 <string>:1(?)
       1    0.140    0.140    1.310    1.310 Sieve.py:13(primes)
       1    0.000    0.000    1.320    1.320 profile:0(Sieve.primes(100000))
       0    0.000             0.000          profile:0(profiler)
      65    0.000    0.000    0.000    0.000 sets.py:119(__iter__)
     314    0.000    0.000    0.000    0.000 sets.py:292(__contains__)
      65    0.000    0.000    0.000    0.000 sets.py:339(_binary_sanity_check)
      66    0.630    0.010    0.630    0.010 sets.py:356(_update)
      66    0.000    0.000    0.630    0.010 sets.py:425(__init__)
      65    0.010    0.000    0.540    0.008 sets.py:489(__isub__)
      65    0.530    0.008    0.530    0.008 sets.py:495(difference_update)
```

Здесь `ncalls` - количество вызовов функции или метода, `tottime` - полное время выполнения кода функции (без времени нахождения в вызываемых функциях), `percall` - тоже, в пересчете на один вызов, `cumtime` - аккумулированное время нахождения в функции, вместе со всеми вызываемыми функциями. В последнем столбце приведено имя файла, номер строки с функцией или методов и его имя.

Примечание:

"Странные" имена, например, `__iter__`, `__contains__` и `__isub__` - имена методов, реализующих итерацию по элементам, проверку принадлежности элемента (`in`) и операцию `-=`. Метод `__init__` - конструктор объекта (в данном случае - множества).

Модуль unittest

При разработке программного обеспечения рекомендуется применять так называемые **регрессионные испытания**. Для каждого модуля составляется набор тестов, по возможности таким образом, чтобы проверялись не только типичные вычисления, но и "крайние", вырожденные случаи, чтобы испытания затронули каждую ветку алгоритма хотя бы один раз. Тест для данного модуля (написанный сразу после того, как определен интерфейс модуля) находится в файле `test_Sieve.py`:

```
# file: test_Sieve.py
import Sieve, sets
import unittest

class TestSieve(unittest.TestCase):

    def setUp(self):
        pass

    def testone(self):
        primes = Sieve.primes(1)
        self.assertEqual(primes, sets.Set())

    def test100(self):
        primes = Sieve.primes(100)
        self.assert_(primes == sets.Set([2, 3, 5, 7, 11, 13, 17, 19, 23, 29,
31, 37, 41, 43, 47,
                53, 59, 61, 67, 71, 73, 79, 83, 89, 97]))

if __name__ == '__main__':
    unittest.main()
```

Тестовый модуль состоит из определения класса, унаследованного от класса `unittest.TestCase`, в котором описывается подготовка к испытаниям (метод `setUp`) и сами испытания -- методы, начинающиеся на `test`. В данном случае таких испытаний всего два: в первом испытывается случай `N=1`, а во втором -- `N=100`.

Запуск тестов производится выполнением функции `unittest.main()`. Вот как выглядят успешные испытания:

```
$ python test_Sieve.py
..
----------------------------------------------------------------------
Ran 2 tests in 0.002s

OK
```

В процессе разработки перед каждым выпуском все модули прогоняются через регрессионные испытания, чтобы обнаружить, не были ли внесены ошибки. Однако никакие тесты в общем случае не могут гарантировать безошибочности сложной программы. При дополнении модулей тесты также могут быть дополнены, чтобы отразить изменения в проекте.

Кстати, сам **Python** и его стандартная библиотека имеют тесты для каждого модуля - они находятся в каталоге `test` в месте, где развернуты файлы поставки **Python**, и являются частью пакета `test`.

Модуль pydoc

Успех проекта зависит не только от обеспечения эффективного и качественного кода, но и от качества документации. Утилита `pydoc` аналогична команде `man` в **Unix:**

```
$ pydoc Sieve
Help on module Sieve:

NAME
    Sieve - Модуль для вычисления простых чисел от 2 до N

FILE
    Sieve.py

FUNCTIONS
    primes(N)
        Возвращает все простые от 2 до N
```

Эта страница помощи появилась благодаря тому, что были написаны строки документации - как ко всему модулю, так и к функции `primes(N)`.

Стоит попробовать запустить `pydoc` следующей командой:

```
pydoc -p 8088
```

И направить браузер на **URL** **http://127.0.0.1:8088/** - можно получить документацию по модулям **Python** в виде красивого **web**-сайта.

Узнать другие возможности `pydoc` можно, подав команду `pydoc pydoc`.

Пакет docutils

Этот пакет и набор утилит пока что не входит в стандартную поставку **Python**, однако о нем нужно знать тем, кто хочет быстро готовить документацию (руководства пользователя и т.п.) для своих модулей. Этот пакет использует специальный язык разметки **(ReStructuredText)**, из которого потом легко получается документация в виде **HTML**, **LaTeX** и в других форматах. Текст в формате **RST** легко читать и в исходном виде. С этим инструментом можно познакомиться на **http://docutils.sourceforge.net**

Пакет distutils

Данный пакет предоставляет стандартный путь для распространения собственных **Python**-пакетов. Достаточно написать небольшой конфигурационный файл `setup.py`, использующий `distutils`, и файл с перечислением файлов проекта `MANIFEST.in`, чтобы пользователи пакета смогли его установить командой

```
python setup.py install
```

Тонкости работы с `distutils` можно изучить по документации.

Взаимодействие с операционной системой

Различные операционные системы имеют свои особенности. Здесь рассматривается основной модуль этой категории, функции которого работают на многих операционных системах.

Модуль os

Разделители каталогов и другие связанные с этим обозначения доступны в виде констант.

Константа	Что обозначает
os.curdir	Текущий каталог
os.pardir	Родительский каталог
os.sep	Разделитель элементов пути
os.altsep	Другой разделитель элементов пути
os.pathsep	Разделитель путей в списке путей
os.defpath	Список путей по умолчанию
os.linesep	Признак окончания строки

Программа на **Python** работает в операционной системе в виде отдельного процесса. Функции модуля os дают доступ к различным значениям, относящимся к процессу и к среде, в которой он исполняется. Одним из важных объектов, доступных из модуля os, является словарь переменных окружения environ. Например, с помощью переменных окружения **web-**сервер передает некоторые параметры в **CGI-**сценарий. В следующем примере можно получить переменную окружения PATH:

```
import os
PATH = os.environ['PATH']
```

Большая группа функций посвящена работе с файлами и каталогами. Ниже приводятся только те, которые доступны как в **Unix**, так и в **Windows**.

access(path, flags)	Проверка доступности файла или каталога с именем path. Режим запрашиваемого доступа указывается значением flags, составленных комбинацией (побитовым ИЛИ) флагов os.F_OK (файл существует), os.R_OK (из файла можно читать), os.W_OK (в файл можно писать) и os.X_OK (файл можно исполнять, каталог можно просматривать).
chdir(path)	Делает path текущим рабочим каталогом.
getcwd()	Текущий рабочий каталог.
chmod(path, mode)	Устанавливает режим доступа к path в значение mode. Режим доступа можно получить, скомбинировав флаги (см. ниже). Следует заметить, что chmod() не дополняет действующий режим, а устанавливает его заново.
listdir(dir)	Возвращает список файлов в каталоге dir. В список не входят специальные значения "." и "..".
mkdir(path[, mode])	Создает каталог path. По умолчанию режим mode равен 0777, то есть: S_IRWXU|S_IRWXG|S_IRWXO, если пользоваться константами модуля stat.
makedirs(path[, mode])	Аналог mkdir(), создающий все необходимые каталоги, если они не существуют. Возбуждает исключение, когда последний каталог уже существует.
remove(path), unlink(path)	Удаляет файл path. Для удаления каталогов используются rmdir() и removedirs().
rmdir(path)	Удаляет пустой каталог path.
removedirs(path)	Удаляет path до первого непустого каталога. В случае если самый последний вложенный подкаталог в указанном пути - не пустой, возбуждается исключение OSError.
rename(src, dst)	Переименовывает файл или каталог src в dst.
renames(src, dst)	Аналог rename(), создающий все необходимые каталоги для пути dst и удаляющий пустые каталоги пути src.

`stat(path)`	Возвращает информацию о `path` в виде не менее чем десятиэлементного кортежа. Для доступа к элементам кортежа можно использовать константы из модуля `stat`, например `stat.ST_MTIME` (время последней модификации файла).
`utime(path, times)`	Устанавливает значения времен последней модификации (`mtime`) и доступа к файлу (`atime`). Если `times` равен `None`, в качестве времен берется текущее время. В других случаях `times` рассматривается как двухэлементный кортеж (`atime`, `mtime`). Для получения `atime` и `mtime` некоторого файла можно использовать `stat()` совместно с константами модуля `stat`.

Для работы с процессами модуль `os` предлагает следующие функции (здесь упомянуты только некоторые, доступные как в **Unix**, так и в **Windows**):

`abort()`	Вызывает для текущего процесса сигнал `SIGABRT`.
`system(cmd)`	Выполняет командную строку `cmd` в отдельной оболочке, аналогично вызову `system` библиотеки языка **C**. Возвращаемое значение зависит от используемой платформы.
`times()`	Возвращает кортеж из пяти элементов, содержащий время в секундах работы процесса, ОС (по обслуживанию процесса), дочерних процессов, ОС для дочерних процессов, а также время от фиксированного момента в прошлом (например, от момента запуска системы).
`getloadavg()`	Возвращает кортеж из трех значений, соответствующих занятости процессора за последние **1**, **5** и **15** минут.

Модуль stat

В этом модуле описаны константы, которые можно использовать как индексы к кортежам, применяемым функциями `os.stat()` и `os.chmod()` (а также некоторыми другими). Их можно уточнить в документации по **Python**.

Модуль tempfile

Программе иногда требуется создать временный файл, который после выполнения некоторых действий больше не нужен. Для этих целей можно использовать функцию `TemporaryFile`, которая возвращает файловый объект, готовый к записи и чтению.

В следующем примере создается временный файл, куда записываются данные и затем читаются:

```
import tempfile
f = tempfile.TemporaryFile()
f.write("0"*100)  # записывается сто символов 0
f.seek(0)         # уст. указатель на начало файла
print len(f.read())  # читается до конца файла и вычисляется длина
```

Как и следовало ожидать, в результате будет выведено **100**. Временный файл будет удален, как только будут удалены все ссылки на его объект.

Обработка текстов

Модули этой категории будут подробно рассмотрены в отдельной лекции.

Многопоточные вычисления

Модули этой категории станут предметом рассмотрения отдельной лекции.

Хранение данных. Архивация

К этой категории отнесены модули, которые работают с внешними хранилищами данных.

Модуль pickle

Процесс записи объекта в виде последовательности байтов называется **сериализацией**. Для того чтобы сохранить объект во внешней памяти или передать его по каналам связи, его нужно вначале сериализовать.

Модуль `pickle` позволяет сериализовывать объекты и сохранять их в строке или файле. Следующие объекты могут быть сериализованы:

- встроенные типы: `None`, числа, строки (обычные и **Unicode**).
- списки, кортежи и словари, содержащие только сериализуемые объекты.
- функции, определенные на уровне модуля (сохраняется имя, но не реализация!).
- встроенные функции.
- классы, определенные на уровне модуля.
- объекты классов, `__dict__` или `__setstate__()` которых являются сериализуемыми.

Типичный вариант использования модуля приведен ниже.

Сохранение:

```
import pickle, time
mydata = ("abc", 12, [1, 2, 3])
output_file = open("mydata.dat", "w")
p = pickle.Pickler(output_file)
p.dump(mydata)
output_file.close()
```

Восстановление:

```
import pickle
input_file = open("mydata.dat", "r")
mydata = pickle.load(input_file)
print mydata
input_file.close()
```

Модуль shelve

Для хранения объектов в родном для **Python** формате можно применять полку **(shelve)**. По своему интерфейсу полка ничем не отличается от словаря. Следующий пример показывает, как использовать полку:

```
import shelve
data = ("abc", 12)      # - данные (объект)
key = "key"             # - ключ (строка)
filename = "polka.dat"  # - имя файла для хранения полки
d = shelve.open(filename)       # открытие полки
d[key] = data           # сохранить данные под ключом key
                        # (удаляет старое значение, если оно было)
data = d[key]           # загрузить значение по ключу
len(d)                  # получить количество объектов на полке
d.sync()                # запись изменений в БД на диске
del d[key]              # удалить ключ и значение
flag = d.has_key(key)   # проверка наличия ключа
lst = d.keys()          # список ключей
d.close()               # закрытие полки
```

Модули anydbm и gdbm

Для внешнего хранения данных можно использовать примитивные базы данных, содержащие пары ключ-значение. В **Python** имеется несколько модулей для работы с такими базами: `bsddb`, `gdbm`, `dbhash` и т.п. Модуль `anydbm` выбирает один из имеющихся хэшей, поэтому его можно применять для чтения ряда форматов (`any` - любой).

Доступ к хэшу из **Python** мало отличается от доступа к словарю. Разница лишь в том, что хэш еще нужно открыть для создания, чтения или записи, а затем закрыть. Кроме того, при записи хэш блокируется, чтобы не испортить данные.

Модуль csv

Формат **CSV (comma separated values -** значения, разделенные запятыми**)** достаточно популярен для обмена данными между электронными таблицами и базами данных. Следующий ниже пример посвящен записи в **CSV-**файл и чтению из него:

```
mydata = [(1, 2, 3), (1, 3, 4)]
import csv

# Запись в файл:
f = file("my.csv", "w")
writer = csv.writer(f)
for row in mydata:
    writer.writerow(row)
f.close()

# Чтение из файла:
reader = csv.reader(file("my.csv"))
for row in reader:
    print row
```

Платформо-зависимые модули

Эта категория модулей имеет применение только для конкретных операционных систем и семейств операционных систем. Довольно большое число модулей в стандартной поставке **Python** посвящено трем платформам: **Unix**, **Windows** и **Macintosh**.

При создании переносимых приложений использовать платформо-зависимые модули можно только при условии реализации альтернативных веток алгоритма, либо с отказом от свойств, которые доступны не на всех платформах. Так, под **Windows** не работает достаточно обычная для **Unix** функция `os.fork()`, поэтому при создании переносимых приложений нужно использовать другие средства для распараллеленных вычислений, например, многопоточность.

В документации по языку обычно отмечено, для каких платформ доступен тот или иной модуль или даже отдельная функция.

Поддержка сети. Протоколы Интернет

Почти все модули из этой категории, обслуживающие клиентскую часть протокола, построены по одному и тому же принципу: из модуля необходим только класс, объект которого содержит информацию о соединении с сервером, а методы реализуют взаимодействие с сервером по соответствующему протоколу. Таким образом, чем сложнее протокол, тем больше методов и других деталей требуется для реализации клиента.

Примеры серверов используются по другому принципу. В модуле с реализацией сервера описан базовый класс, из которого пользователь модуля должен наследовать свой класс, реализующий требуемую функциональность. Правда, иногда замещать нужно всего один или два метода.

Этому вопросу будет посвящена отдельная лекция.

Поддержка Internet. Форматы данных

В стандартной библиотеке **Python** имеются разноуровневые модули для работы с различными форматами, применяющимися для кодирования данных в сети Интернет и тому подобных приложениях.

Сегодня наиболее мощным инструментом для обработки сообщений в формате **RFC 2822** является пакет **email**. С его помощью можно как разбирать сообщения в удобном для программной обработки виде, так и формировать сообщение на основе данных о полях и основном содержимом (включая вложения).

Python о себе

Язык **Python** является рефлективным языком, в котором можно "заглянуть" глубоко в собственные внутренние структуры кода и данных. Модули этой категории дают возможность прикоснуться к внутреннему устройству **Python**. Более подробно об этом рассказывается в отдельной лекции.

Графический интерфейс

Почти все современные приложения имеют графический интерфейс пользователя. Такие приложения можно создавать и на языке **Python**. В стандартной поставке имеется модуль **Tkinter**, который есть не что иное, как интерфейс к языку **Tcl/Tk**, на котором можно описывать графический интерфейс.

Следует отметить, что существуют и другие пакеты для программирования графического интерфейса: **wxPython** (основан на **wxWindows**), **PyGTK** и т.д. Среди этих пакетов в основном такие, которые работают на одной платформе (реже - на двух).

Помимо возможностей программного описания графического интерфейса, для **Python** есть несколько коммерческих и некоммерческих **построителей графического интерфейса (GUI builders)**, однако в данном курсе они не рассматриваются.

Заключение

В этой лекции говорилось о встроенных функциях языка **Python** и модулях его стандартной библиотеки. Некоторые направления будут рассмотрены более подробно в следующих лекциях. **Python** имеет настолько обширную стандартную библиотеку, что в рамках одной лекции можно только сделать ее краткий обзор, подкрепив небольшими примерами наиболее типичные идиомы при использовании модулей.

Лекция #3: Элементы функционального программирования

Что такое функциональное программирование?

Функциональное программирование - это стиль программирования, использующий только композиции функций. Другими словами, это программирование в выражениях, а не в императивных командах.

Как отмечает Дэвид Мертц **(David Mertz)** в своей статье о функциональном программировании на **Python**, "функциональное программирование - программирование на функциональных языках **(LISP, ML, OCAML, Haskell, ...)"**, основными атрибутами которых являются:

- "Наличие функций первого класса **(**функции наравне с другими объектами можно передавать внутрь функций**).**
- Рекурсия является основной управляющей структурой в программе.
- Обработка списков **(**последовательностей**).**
- Запрещение побочных эффектов у функций, что в первую очередь означает отсутствие присваивания **(**в "чистых" функциональных языках**)**
- Запрещение операторов, основной упор делается на выражения. Вместо операторов вся программа в идеале - одно выражение с сопутствующими определениями.
- Ключевой вопрос: **что** нужно вычислить, а не **как.**
- Использование функций более высоких порядков **(**функции над функциями над функциями**)".**

Функциональная программа

В математике функция отображает объекты из одного множества **(множества определения функции)** в другое **(множество значений функции)**. Математические функции **(**их называют **чистыми)** "механически", однозначно вычисляют результат по заданным аргументам. Чистые функции не должны хранить в себе какие-либо данные между двумя вызовами. Их можно представлять себе черными ящиками, о которых известно только то, что они делают, но совсем не важно, как.

Программы в функциональном стиле конструируются как **композиция** функций. При этом функции понимаются почти так же, как и в математике: они отображают одни объекты в другие. В программировании "чистые" функции - идеал, не всегда достижимый на практике. Практически полезные функции обычно имеют **побочный эффект**: сохраняют состояние между вызовами или меняют состояние других объектов. Например, без побочных эффектов невозможно представить себе функции ввода-вывода. Собственно, такие функции ради этих "эффектов" и используются. Кроме того, математические функции легко работают с объектами, требующими бесконечного объема информации **(**например, вещественные числа**).** В общем случае компьютерная программа может выполнить лишь приближенные вычисления.

Кстати, бинарные операции **"+", "-", "*", "/"**, которые записываются в выражениях, являются "математическими" функциями над двумя аргументами **-- операндами.** Их используют настолько часто, что синтаксис языка программирования имеет для них более короткую запись. Модуль `operator` позволяет представлять эти операции в функциональном стиле:

```
>>> from operator import add, mul
>>> print add(2, mul(3, 4))
14
```

Функция: определение и вызов

Как уже говорилось, определить функцию в **Python** можно двумя способами: с помощью оператора `def` и `lambda`-выражения. Первый способ позволяет использовать операторы. При втором - определение функции может быть только выражением.

Забегая вперед, можно заметить, что методы классов определяются так же, как и функции. Отличие состоит в специальном смысле первого аргумента `self` (в нем передается экземпляр объекта).

Лучше всего рассмотреть синтаксис определения функции на нескольких примерах. После определения соответствующей функции показан один или несколько вариантов ее вызова (некоторые примеры взяты из стандартной библиотеки).

Определение функции должно содержать список **формальных параметров** и **тело определения функции**. В случае с оператором `def` функции также задается некоторое имя. Формальные параметры являются локальными именами внутри тела определения функции, а при вызове функции они оказываются связанными с объектами, переданными как фактические параметры. Значения по умолчанию вычисляются в момент выполнения оператора `def`, и потому в них можно использовать видимые на момент определения имена.

Вызов функции синтаксически выглядит как `объект-функция(фактические параметры)`. Обычно объект-функция - это просто имя функции, хотя это может быть и любое выражение, которое в результате вычисления дает исполняемый объект.

Функция одного аргумента:

```
def swapcase(s):
    return s.swapcase()

print swapcase("ABC")
```

Функция двух аргументов, один из которых необязателен и имеет значение по умолчанию:

```
def inc(n, delta=1):
    return n+delta

print inc(12)
print inc(12, 2)
```

Функция с одним обязательным аргументом, с одним, имеющим значение по умолчанию и неопределенным числом именованных аргументов:

```
def wrap(text, width=70, **kwargs):
   from textwrap import TextWrapper
   # kwargs  - словарь с именами и значениями аргументов
   w = TextWrapper(width=width, **kwargs)
   return w.wrap(text)

print wrap("my long text ...", width=4)
```

Функция произвольного числа аргументов:

```
def max_min(*args):
  # args - список аргументов в порядке их указания при вызове
  return max(args), min(args)

print max_min(1, 2, -1, 5, 3)
```

Функция с обычными (позиционными) и именованными аргументами:

```
def swiss_knife(arg1, *args, **kwargs):
  print arg1
```

```
    print args
    print kwargs
    return None

print swiss_knife(1)
print swiss_knife(1, 2, 3, 4, 5)
print swiss_knife(1, 2, 3, a='abc', b='sdf')
# print swiss_knife(1, a='abc', 3, 4)  # !!! ошибка

lst = [2, 3, 4, 5]
dct = {'a': 'abc', 'b': 'sdf'}
print swiss_knife(1, *lst, **dct)
```

Пример определения функции с помощью `lambda`-выражения дан ниже:

```
func = lambda x, y: x + y
```

В результате `lambda`-выражения получается безымянный объект-функция, которая затем используется, например, для того, чтобы связать с ней некоторое имя. Однако, как правило, определяемые `lambda`-выражением функции, применяются в качестве параметров функций.

В языке **Python** функция может возвратить только одно значение, которое может быть кортежем. В следующем примере видно, как стандартная функция `divmod()` возвращает частное и остаток от деления двух чисел:

```
def bin(n):
    """Цифры двоичного представления натурального числа """
    digits = []
    while n > 0:
        n, d = divmod(n, 2)
        digits = [d] + digits
    return digits

print bin(69)
```

Примечание:

Важно понять, что за именем функции стоит объект. Этот объект можно связать с другим именем:

```
def add(x, y):
  return x + y
addition = add  # теперь addition и add - разные имена одного и того же объекта
```

Пример, в котором в качестве значения по умолчанию аргумента функции используется изменчивый объект (список). Этот объект - один и тот же для всех вызовов функций, что может привести к казусам:

```
def mylist(val, lst=[]):
  lst.append(val)
  return lst

print mylist(1),
print mylist(2)
```

Вместо ожидаемого `[1] [2]` получается `[1] [1, 2]`, так как добавляются элементы к "значению по умолчанию".

Правильный вариант решения будет, например, таким:

```
def mylist(val, lst=None):
```

```
   lst = lst or []
   lst.append(val)
   return lst
```

Конечно, приведенная выше форма может использоваться для хранения в функции некоторого состояния между ее вызовами, однако, практически всегда вместо функции с таким побочным эффектом лучше написать класс и использовать его экземпляр.

Рекурсия

В некоторых случаях описание функции элегантнее всего выглядит с применением вызова этой же функции. Такой прием, когда функция вызывает саму себя, называется **рекурсией**. В функциональных языках рекурсия обычно используется много чаще, чем итерация (циклы).

В следующем примере переписывается функция `bin()` в рекурсивном варианте:

```
def bin(n):
    """Цифры двоичного представления натурального числа """
    if n == 0:
      return []
    n, d = divmod(n, 2)
    return bin(n) + [d]

print bin(69)
```

Здесь видно, что цикл `while` больше не используется, а вместо него появилось условие окончания рекурсии: условие, при выполнении которого функция не вызывает себя.

Конечно, в погоне за красивым рекурсивным решением не следует упускать из виду эффективность реализации. В частности, пример реализации функции для вычисления `n`-го числа Фибоначчи это демонстрирует:

```
def Fib(n):
   if n < 2:
       return n
   else:
       return Fib(n-1) + Fib(n-2)
```

В данном случае количество рекурсивных вызовов растет экспоненциально от числа `n`, что совсем не соответствует временной сложности решаемой задачи.

В качестве упражнения предлагается написать итеративный и рекурсивный варианты этой функции, которые бы требовали линейного времени для вычисления результата.

Предупреждение:

При работе с рекурсивными функциями можно легко превысить глубину допустимой в **Python** рекурсии. Для настройки глубины рекурсии следует использовать функцию `setrecursionlimit(N)` из модуля `sys`, установив требуемое значение `N`.

Функции как параметры и результат

Как уже не раз говорилось, функции являются такими же объектами **Python** как числа, строки или списки. Это означает, что их можно передавать в качестве параметров функций или возвращать из функций.

Функции, принимающие в качестве аргументов или возвращающие другие функции в результате, называют **функциями высшего порядка**. В **Python** функции высшего

порядка применяются программистами достаточно часто. В большинстве случаев таким образом строится механизм обратных вызовов **(callbacks)**, но встречаются и другие варианты. Например, алгоритм поиска может вызывать переданную ему функцию для каждого найденного объекта.

Функция apply()

Функция `apply()` применяет функцию, переданную в качестве первого аргумента, к параметрам, которые переданы вторым и третьим аргументом. Эта функция в **Python** устарела, так как вызвать функцию можно с помощью обычного синтаксиса вызова функции. Позиционные и именованные параметры можно передать с использованием звездочек:

```
>>> lst = [1, 2, 3]
>>> dct = {'a': 4, 'b': 5}
>>> apply(max, lst)
3
>>> max(*lst)
3
>>> apply(dict, [], dct)
{'a': 4, 'b': 5}
>>> dict(**dct)
{'a': 4, 'b': 5}
```

Обработка последовательностей

Многие алгоритмы сводятся к обработке массивов данных и получению новых массивов данных в результате. Среди встроенных функций **Python** есть несколько для работы с последовательностями.

Под **последовательностью** в **Python** понимается любой тип данных, который поддерживает интерфейс последовательности **(**это несколько специальных методов, реализующих операции над последовательностями, которые в данном курсе обсуждаться не будут**).**

Следует заметить, что тип, основной задачей которого является хранение, манипулирование и обеспечение доступа к самостоятельным данным называется **контейнерным типом** или просто **контейнером**. Примеры контейнеров в **Python -** списки, кортежи, словари.

Функции range() и xrange()

Функция `range()` уже упоминалась при рассмотрении цикла `for`. Эта функция принимает от одного до трех аргументов. Если аргумент всего один, она генерирует список чисел от **0 (**включительно**)** до заданного числа **(**исключительно**).** Если аргументов два, то список начинается с числа, указанного первым аргументом. Если аргументов три **-** третий аргумент задает шаг

```
>>> print range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> print range(1, 10)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> print range(1, 10, 3)
[1, 4, 7]
```

Функция `xrange()` **-** аналог `range()`, более предпочтительный для использования при последовательном доступе, например, в цикле `for` или с итераторами. Она возвращает специальный `xrange`-объект, который ведет себя почти как список, порождаемый `range()`, но не хранит в памяти все выдаваемые элементы.

Функция map()

Для применения некоторой функции ко всем элементам последовательности применяется функция `map(f, *args)`. Первый параметр этой функции - функция, которая будет применяться ко всем элементам последовательности. Каждый следующий `n+1`-й параметр должен быть последовательностью, так как каждый его элемент будет использован в качестве `n`-го параметра при вызове функции `f()`. Результатом будет список, составленный из результатов выполнения этой функции.

В следующем примере складываются значения из двух списков:

```
>>> l1 = [2, 7, 5, 3]
>>> l2 = [-2, 1, 0, 4]
>>> print map(lambda x, y: x+y, l1, l2)
[0, 8, 5, 7]
```

В этом примере применена безымянная функция для получения суммы двух операндов ко всем элементам `l1` и `l2`. В случае если одна из последовательностей короче другой, вместо соответствующего операнда будет `None`, что, конечно, собьет операцию сложения. В зависимости от решаемой задачи, можно либо видоизменить функцию, либо считать разные по длине последовательности ошибкой, которую нужно обрабатывать как отдельную ветвь алгоритма.

Частный случай применения `map()` - использование `None` в качестве первого аргумента. В этом случае просто формируется список кортежей из элементов исходных последовательностей:

```
>>> l1 = [2, 7, 5, 3]
>>> l2 = [-2, 1, 0, 4]
>>> print map(None, l1, l2)
[(2, -2), (7, 1), (5, 0), (3, 4)]
```

Функция filter()

Другой часто встречающейся операцией является фильтрование исходной последовательности в соответствии с некоторым предикатом (условием). Функция `filter(f, seq)` принимает два аргумента: функцию с условием и последовательность, из которой берутся значения. В результирующую последовательность попадут только те значения из исходной, для которой `f()` возвратит истину. Если в качестве `f` задано значение `None`, результирующая последовательность будет состоять из тех значений исходной, которые имеют истинностное значение `True`.

Например, в следующем фрагменте кода можно избавится от символов, которые не являются буквами:

```
>>> filter(lambda x: x.isalpha(), 'Hi, there! I am eating an apple.')
'HithereIameatinganapple'
```

Списковые включения

Для более естественной записи обработки списков в **Python 2** была внесена новинка: списковые включения. Фактически это специальный сокращенный синтаксис для вложенных циклов `for` и условий `if`, на самом низком уровне которых определенное выражение добавляется к списку, например:

```
all_pairs = []
for i in range(5):
  for j in range(5):
    if i <= j:
```

```
        all_pairs.append((i, j))
```

Все это можно записать в виде спискового включения так:

```
all_pairs = [(i, j) for i in range(5) for j in range(5) if i <= j]
```

Как легко заметить, списковые включения позволяют заменить `map()` и `filter()` на более удобные для прочтения конструкции.

В следующей таблице приведены эквивалентные выражения в разных формах:

В форме функции	В форме спискового включения
`filter(f, lst)`	`[x for x in lst if f(x)]`
`filter(None, lst)`	`[x for x in lst if x]`
`map(f, lst)`	`[f(x) for x in lst]`

Функция sum()

Получить сумму элементов можно с помощью функции `sum()`:

```
>>> sum(range(10))
45
```

Эта функция работает только для числовых типов, она не может конкатенировать строки. Для конкатенации списка строк следует использовать метод `join()`.

Функция reduce()

Для организации цепочечных вычислений (вычислений с накоплением результата) можно применять функцию `reduce()`, которая принимает три аргумента: функцию двух аргументов, последовательность и начальное значение. С помощью этой функции можно, в частности, реализовать функцию `sum()`:

```
def sum(lst, start):
  return reduce(lambda x, y: x + y, lst, start)
```

Совет:

Следует помнить, что в качестве передаваемого объекта может оказаться список, который позволит накапливать промежуточные результаты. Тем самым, `reduce()` может использоваться для генерации последовательностей.

В следующем примере накапливаются промежуточные результаты суммирования:

```
lst = range(10)
f = lambda x, y: (x[0] + y, x[1]+[x[0] + y])
print reduce(f, lst, (0, []))
```

В итоге получается:

```
(45, [0, 1, 3, 6, 10, 15, 21, 28, 36, 45])
```

Функция zip()

Эта функция возвращает список кортежей, в котором `i`-й кортеж содержит `i`-е элементы аргументов-последовательностей. Длина результирующей последовательности равна длине самой короткой из последовательностей-аргументов:

```
>>> print zip(range(5), "abcde")
[(0, 'a'), (1, 'b'), (2, 'c'), (3, 'd'), (4, 'e')]
```

Итераторы

Применять для обработки данных явные последовательности не всегда эффективно, так как на хранение временных данных может тратиться много оперативной памяти. Более эффективным решением представляется использование **итераторов** - специальных объектов, обеспечивающих последовательный доступ к данным контейнера. Если в выражении есть операции с итераторами вместо контейнеров, промежуточные данные не будут требовать много места для хранения - ведь они запрашиваются по мере необходимости для вычислений. При обработке данных с использованием итераторов память будет требоваться только для исходных данных и результата, да и то необязательно вся сразу - ведь данные могут читаться и записываться в файл на диске.

Итераторы можно применять вместо последовательности в операторе `for`. Более того, внутренне оператор `for` запрашивает от последовательности ее итератор. Объект файлового типа тоже (построчный) итератор, что позволяет обрабатывать большие файлы, не считывая их целиком в память.

Там, где требуется итератор, можно использовать последовательность.

Работа с итераторами рассматривается в разделе, посвященном функциональному программированию, так как итераторами удобно манипулировать именно в функциональном стиле.

Использовать итератор можно и "вручную". Любой объект, поддерживающий интерфейс итератора, имеет метод `next()`, который при каждом вызове выдает очередное значение итератора. Если больше значений нет, возбуждается исключение `StopIteration`. Для получения итератора по некоторому объекту необходимо прежде применить к этому объекту функцию `iter()` (цикл `for` делает этоавтоматически).

В **Python** имеется модуль `itertools`, который содержит набор функций, комбинируя которые, можно составлять достаточно сложные схемы обработки данных с помощью итераторов. Далее рассматриваются некоторые функции этого модуля.

Функция iter()

Эта функция имеет два варианта использования. В первом она принимает всего один аргумент, который должен "уметь" предоставлять свой итератор. Во втором один из аргументов - функция без аргументов, другой - стоповое значение. Итератор вызывает указанную функцию до тех пор, пока та не возвратит стоповое значение. Второй вариант встречается много реже первого и обычно внутри метода класса, так как сложно порождать значения "на пустом месте":

```
it1 = iter([1, 2, 3, 4, 5])

def forit(mystate=[]):
  if len(mystate) < 3:
    mystate.append(" ")
    return " "

it2 = iter(forit, None)

print [x for x in it1]
print [x for x in it2]
```

Примечание:

Если функция не возвращает значения явно, она возвращает `None`, что и использовано в

примере выше.

Функция enumerate()

Эта функция создает итератор, нумерующий элементы другого итератора. Результирующий итератор выдает кортежи, в которых первый элемент - номер (начиная с нуля), а второй - элемент исходной последовательности:

```
>>> print [x for x in enumerate("abcd")]
[(0, 'a'), (1, 'b'), (2, 'c'), (3, 'd')]
```

Функция sorted()

Эта функция, появившаяся в **Python 2.4**, позволяет создавать итератор, выполняющий сортировку:

```
>>> sorted('avdsdf')
['a', 'd', 'd', 'f', 's', 'v']
```

Далее рассматриваются функции модуля `itertools`.

Функция itertools.chain()

Функция `chain()` позволяет сделать итератор, состоящий из нескольких соединенных последовательно итераторов. Итераторы задаются в виде отдельных аргументов. Пример:

```
from itertools import chain
it1 = iter([1,2,3])
it2 = iter([8,9,0])
for i in chain(it1, it2):
    print i,
```

даст в результате

```
1 2 3 8 9 0
```

Функция itertools.repeat()

Функция `repeat()` строит итератор, повторяющий некоторый объект заданное количество раз:

```
for i in itertools.repeat(1, 4):
    print i,
```

```
1 1 1 1
```

Функция itertools.count()

Бесконечный итератор, дающий целые числа, начиная с заданного:

```
for i in itertools.count(1):
    print i,
    if i > 100:
        break
```

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49
50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72
73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95
```

```
96 97 98 99 100 101
```

Функция itertools.cycle()

Можно бесконечно повторять и некоторую последовательность (или значения другого итератора) с помощью функции `cycle()`:

```
tango = [1, 2, 3]
for i in itertools.cycle(tango):
    print i,

1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2
3 1 2 3 1
2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3
1 2 3 1 2
3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 . . .
```

Функции itertools.imap(), itertools.starmap() и itertools.ifilter()

Аналогами `map()` и `filter()` в модуле `itertools` являются `imap()` и `ifilter()`. Отличие `imap()` от `map()` в том, что вместо значения от преждевременно завершившихся итераторов объект `None` не подставляется. Пример:

```
for i in map(lambda x, y: (x,y), [1,2], [1,2,3]):
    print i,

(1, 1) (2, 2) (None, 3)

from itertools import imap
for i in imap(lambda x, y: (x,y), [1,2], [1,2,3]):
  print i,

(1, 1) (2, 2)
```

Здесь следует заметить, что обычная функция `map()` нормально воспринимает итераторы в любом сочетании с итерабельными (поддающимися итерациям) объектами:

```
for i in map(lambda x, y: (x,y), iter([1,2]), [1,2,3]):
    print i,

(1, 1) (2, 2) (None, 3)
```

Функция `itertools.starmap()` подобна `itertools.imap()`, но имеет всего два аргумента. Второй аргумент - последовательность кортежей, каждый кортеж которой задает набор параметров для функции (первого аргумента):

```
>>> from itertools import starmap
>>> for i in starmap(lambda x, y: str(x) + y, [(1,'a'), (2,'b')]):
...   print i,
...
1a 2b
```

Функция `ifilter()` работает как `filter()`. Кроме того, в модуле `itertools` есть функция `ifilterfalse()`, которая как бы добавляет отрицание к значению функции:

```
for i in ifilterfalse(lambda x: x > 0, [1, -2, 3, -3]):
    print i,

-2 -3
```

Функции itertools.takewhile() и itertools.dropwhile()

Некоторую новизну вносит другой вид фильтра: takewhile() и его "отрицательный" аналог dropwhile(). Следующий пример поясняет их принцип действия:

```
for i in takewhile(lambda x: x > 0, [1, -2, 3, -3]):
    print i,

print
for i in dropwhile(lambda x: x > 0, [1, -2, 3, -3]):
  print i,

1
-2 3 -3
```

Таким образом, takewhile() дает значения, пока условие истинно, а остальные значения даже не берет из итератора (именно не берет, а не высасывает все до конца!). И, наоборот, dropwhile() ничего не выдает, пока выполняется условие, зато потом выдает все без остатка.

Функция itertools.izip()

Функция izip() аналогична встроенной zip(), но не тратит много памяти на построение списка кортежей, так как итератор выдает их строго по требованию.

Функция itertools.groupby()

Эта функция дебютировала в **Python 2.4**. Функция принимает два аргумента: итератор (обязательный) и необязательный аргумент - функцию, дающую значение ключа: groupby(iterable[, func]). Результатом является итератор, который возвращает двухэлементный кортеж: ключ и итератор по идущим подряд элементам с этим ключом. Если второй аргумент опущен, элемент итератора сам является ключом. В следующем примере группируются идущие подряд положительные и отрицательные элементы:

```
import itertools, math
lst = map(lambda x: math.sin(x*.4), range(30))
for k, i in itertools.groupby(lst, lambda x: x > 0):
  print k, list(i)
```

Функция itertools.tee()

Эта функция тоже появилась в **Python 2.4**. Она позволяет клонировать итераторы. Первый аргумент - итератор, подлежащий клонированию. Второй (N) -- количество необходимых копий. Функция возвращает кортеж из N итераторов. По умолчанию N=2. Функция имеет смысл, только если итераторы задействованы более или менее параллельно. В противном случае выгоднее превратить исходный итератор в список.

Собственный итератор

Для полноты описания здесь представлен пример итератора, определенного пользователем. Если пример не очень понятен, можно вернуться к нему после изучения объектно-ориентированного программирования:

```
class Fibonacci:
  """Итератор последовательности Фибоначчи до N"""

  def __init__(self, N):
    self.n, self.a, self.b, self.max = 0, 0, 1, N

  def __iter__(self):
    # сами себе итератор: в классе есть метод next()
    return self
```

```
  def next(self):
    if self.n < self.max:
      a, self.n, self.a, self.b = self.a, self.n+1, self.b, self.a+self.b
      return a
    else:
      raise StopIteration

# Использование:
for i in Fibonacci(100):
  print i,
```

Простые генераторы

Разработчики языка не остановились на итераторах. Как оказалось, в интерпретаторе **Python** достаточно просто реализовать **простые генераторы**. Под этим термином фактически понимается специальный объект, вычисления в котором продолжаются до выработки очередного значения, а затем приостанавливаются до возникновения необходимости в выдаче следующего значения. Простой генератор формируется функцией-генератором, которая синтаксически похожа на обычную функцию, но использует специальный оператор `yield` для выдачи следующего значения. При вызове такая функция ничего не вычисляет, а создает объект с интерфейсом итератора для получения значений. Другими словами, если функция должна возвращать последовательность, из нее довольно просто сделать генератор, который будет функционально эквивалентной "ленивой" реализацией. **Ленивыми** называются вычисления, которые откладываются до самого последнего момента, когда получаемое в результате значение сразу используется в другом вычислении.

Для примера с последовательностью Фибоначчи можно построить такой вот генератор:

```
def Fib(N):
  a, b = 0, 1
  for i in xrange(N):
    yield a
    a, b = b, a + b
```

Использовать его не сложнее, чем любой другой итератор:

```
for i in Fib(100):
  print i,
```

Однако следует заметить, что программа в значительной степени упростилась.

Генераторное выражение

В **Python 2.4** по аналогии со списковым включением появилось **генераторное выражение**. По синтаксису оно аналогично списковому, но вместо квадратных скобок используются круглые. Списковое включение порождает список, а, значит, можно ненароком занять очень много памяти. Генератор же, получающийся в результате применения генераторного выражения, списка не создает, он вычисляет каждое следующее значение строго по требованию (при вызове метода `next()`).

В следующем примере можно прочесть из файла строки, в которых производятся некоторые замены:

```
for line in (l.replace("- ", " - ") for l in open("input.dat")):
  print line
```

Ничто не мешает использовать итераторы и для записи в файл:

```
open("output.dat", "w").writelines(
        l.replace(" - ", " - ") for l in open("input.dat"))
```

Здесь для генераторного выражения не потребовалось дополнительных скобок, так как оно расположено внутри скобок вызова функции.

Карринг

Библиотека **Xoltar toolkit (**автор **Bryn Keller)** включает модуль `functional`, который позволяет упростить использование возможностей функционального программирования. Модуль `functional` применяет **"**чистый**" Python.** Библиотеку можно найти по адресу: **http://sourceforge.net/projects/xoltar-toolkit**.

При **карринге (**частичном применении**)** функции создается новая функция, задавая некоторые аргументы исходной. Следующий пример иллюстрирует частичное применение вычитания:

```
from functional import curry
def subtract(x, y):
    return x + y

print subtract(3, 2)
subtract_from_3 = curry(subtract, 3)
print subtract_from_3(2)
print curry(subtract, 3)(2)
```

Во всех трех случаях будет выведено `1`. В следующем примере получается новая функция, подставляя второй аргумент. Вместо другого аргумента вставляется специальное значение `Blank`:

```
from functional import curry, Blank
def subtract(x, y):
    return x + y

print subtract(3, 2)
subtract_2 = curry(subtract, Blank, 2)
print subtract_2(3)
print curry(subtract, Blank, 2)(3)
```

Заключение

В этой лекции рассматривался принцип построения функциональных программ. Кроме того, было показано, что в **Python** и его стандартных модулях имеются достаточно мощные выразительные средства для создания функциональных программ. В случае, когда требуются дополнительные возможности, например, карринг, их можно легко реализовать или взять готовую реализацию.

Следует отметить, что итераторы **-** это практичное продолжение функционального начала в языке **Python.** Итераторы по сути позволяют организовать так называемые **ленивые вычисления (lazy computations),** при которых значения вычисляются только когда они непосредственно требуются.

Ссылки по теме

Статья Д. Мертца **http://www-106.ibm.com/developerworks/library/l-prog.html**

Часто задаваемые вопросы в **comp.lang.functional** **http://www.cs.nott.ac.uk/~gmh/faq.html**

Лекция #4: Объектно-ориентированное программирование

Python проектировался как объектно-ориентированный язык программирования. Это означает (по Алану Кэю, автору объектно-ориентированного языка **Smalltalk**), что он построен с учетом следующих принципов:

1. Все данные в нем представляются объектами.
2. Программу можно составить как набор взаимодействующих объектов, посылающих друг другу сообщения.
3. Каждый объект имеет собственную часть памяти и может состоять из других объектов.
4. Каждый объект имеет тип.
5. Все объекты одного типа могут принимать одни и те же сообщения (и выполнять одни и те же действия).

Язык **Python** имеет достаточно мощную, но, вместе с тем, самобытную поддержку объектно-ориентированного программирования. В этой лекции ООП представляется без лишних формальностей. Работа с **Python** убеждает, что писать программы в объектно-ориентированном стиле не только просто, но и приятно.

Примечание:

К сожалению, большинство введений в ООП (даже именитых авторов) изобилует значительным числом терминов, зачастую затемняющих суть вопроса. В данном изложении будут употребляться только те термины, которые необходимы на практике для взаимопонимания разработчиков или для расширения кругозора. Так как в разных языках программирования ООП имеет свои нюансы, в скобках иногда будут даваться синонимы или аналоги того или иного термина.

Примечание:

ОО программирование - это методология написания кода. Здесь не будет подробно рассматриваться объектно-ориентированный анализ и объектно-ориентированное проектирование, которые не менее важны как стадии создания программного обеспечения.

Основные понятия

При процедурном программировании программа разбивается на части в соответствии с алгоритмом: каждая часть (**подпрограмма**, **функция**, **процедура**) является составной частью алгоритма.

При объектно-ориентированном программировании программа строится как совокупность взаимодействующих объектов.

С точки зрения объектно-ориентированного подхода, **объект** - это нечто, обладающее **значением (состоянием)**, **типом (поведением)** и **индивидуальностью**. Когда программист выделяет объекты в предметной области, он обычно абстрагируется (отвлекается) от большинства их свойств, концентрируясь на существенных для задачи свойствах. Над объектами можно производить **операции** (посылая им сообщения). В языке **Python** все данные представлены в виде объектов.

Взаимодействие объектов заключается в вызове **методов** одних объектов другими. Иногда говорят, что объекты посылают друг другу **сообщения**. Сообщения - это запросы к объекту выполнить некоторые действия. (**Сообщения**, **методы**, **операции**, **функции-члены** являются синонимами).

Каждый объект хранит свое **состояние (**для этого у него есть **атрибуты)** и имеет определенный набор **методов. (**Синонимы: атрибут**, поле**, **слот**, **объект-член**, переменная экземпляра)**.** Методы определяют **поведение** объекта**.** Объекты класса имеют общее поведение**.**

Объекты описываются не индивидуально**,** а с помощью **классов. Класс -** объект**,** являющийся шаблоном объекта**.** Объект**,** созданный на основе некоторого класса**,** называется **экземпляром класса.** Все объекты определенных пользователем классов являются экземплярами класса**.** Тем не менее**,** объекты даже с одним и тем же состоянием могут быть разными объектами**.** Говорят**,** что они имеют разную **индивидуальность.**

В языке **Python** для определения класса используется оператор `class`**:**

```
class имя_класса(класс1, класс2, ...):
  # определения методов
```

Класс определяет **тип** объекта**,** то есть его возможные состояния и набор операций**.**

Абстракция и декомпозиция

Абстракция в ООП позволяет составить из данных и алгоритмов обработки этих данных объекты**,** отвлекаясь от несущественных **(**на некотором уровне**)** с точки зрения составленной информационной модели деталей**.** Таким образом**,** программа подвергается **декомпозиции** на части "дозированной" сложности**.** Отдельный объект**,** даже вместе с совокупностью его связей с другими объектами**,** человеком воспринимается легче **(**именно так он привык оперировать в реальном мире**),** чем что**-**то неструктурированное и монотонное**.**

Перед тем как начать написание даже самой простенькой объектно**-**ориентированной программы**,** необходимо провести анализ предметной области**,** для того чтобы выявить в ней классы объектов**.**

При выделении объектов необходимо абстрагироваться **(**отвлечься**)** от большинства присущих им свойств и сконцентрироваться на свойствах**,** значимых для задачи**..**

Выделяемые объекты необязательно должны походить на физические объекты **-** ведь это абстракции**,** за которыми скрываются процессы**,** взаимодействия**,** отношения**.**

Удачная декомпозиция стоит многого**.** От нее зависят не только количественные характеристики кода **(**быстродействие**,** занимаемая память**),** но и трудоемкость дальнейшего развития и сопровождения**.** При отсутствии соответствующего опыта лучше не загадывать будущих путей развития программы**,** а делать ее как можно проще**,** под конкретную задачу**.**

Даже если просто перечислить все существительные**,** встретившиеся в описании задачи **(**явно или неявно**),** получится неплохой список кандидатов в классы**.**

При процедурном подходе тоже используется декомпозиция**,** но при объектно**-**ориентированном подходе производится декомпозиция не самого алгоритма на более мелкие части**,** а предметной области на классы объектов**.**

Объекты

До этой лекции объекты **Python** встречались много раз**:** ведь каждое число**,** строка**,** функция**,** модуль и т**.**п**. -** это объекты**.** Некоторые встроенные объекты имеют в **Python** синтаксическую поддержку **(**для задания литералов**).** Таковы числа**,** строки**,** списки**,** кортежи и некоторые другие типы**.**

Теперь следует посмотреть на них в свете только что приведенных определений**.** Пример**:**

```
a = 3
b = 4.0
c = a + b
```

Здесь происходит следующее. Сначала имя "a" связывается в локальном пространстве имен с объектом-числом **3** (целое число). Затем "b" связывается с объектом-числом **4.0** (число с плавающей точкой). После этого над объектами **3** и **4.0** выполняется операция сложения, и имя "c" связывается с получившимся объектом. Кстати, операциями, в основном, будут называться методы, которые имеют в **Python** синтаксическую поддержку, в данном случае - инфиксную запись. То же самое можно записать как:

```
c = a.__add__(b)
```

Здесь `__add__()` - метод объекта `a`, который реализует операцию `+` между этим объектом и другим объектом.

Узнать набор методов некоторого объекта можно с помощью встроенной функции `dir()`:

```
>>> dir(a)
['__abs__', '__add__', '__and__', '__class__', '__cmp__', '__coerce__',
'__delattr__', '__div__', '__divmod__', '__doc__', '__float__',
'__floordiv__', '__getattribute__', '__getnewargs__', '__hash__',
'__hex__', '__init__', '__int__', '__invert__', '__long__',
'__lshift__', '__mod__', '__mul__', '__neg__', '__new__',
'__nonzero__', '__oct__', '__or__', '__pos__', '__pow__',
'__radd__', '__rand__', '__rdiv__', '__rdivmod__', '__reduce__',
'__reduce_ex__', '__repr__', '__rfloordiv__', '__rlshift__',
'__rmod__', '__rmul__', '__ror__', '__rpow__', '__rrshift__',
'__rshift__', '__rsub__', '__rtruediv__', '__rxor__',
'__setattr__', '__str__', '__sub__', '__truediv__', '__xor__']
```

Здесь стоит указать на еще одну особенность **Python**. Не только инфиксные операции, но и встроенные функции ожидают наличия некоторых методов у объекта. Например, можно записать:

```
abs(c)
```

А функция `abs()` на самом деле использует метод переданного ей объекта:

```
c.__abs__()
```

Объекты появляются в результате вызова функций-фабрик или конструкторов классов (об этом ниже), а заканчивают свое существование при удалении последней ссылки на объект. Оператор `del` удаляет имя (а значит, и одну ссылку на объект) из пространства имен:

```
a = 1
# ...
del a
# имени a больше нет
```

Типы и классы

Тип определяет область допустимых значений объекта и набор операций над ним. В ООП тип тесно связан с **поведением** - действиями объекта, состоящими в изменении внутреннего состояния и вызовами методов других объектов.

Ранее в языке **Python** встроенные типы данных не являлись **экземплярами класса**, поэтому считалось, что это были просто объекты определенного типа. Теперь ситуация изменилась, и объекты встроенных типов имеют классы, к которым они принадлежат. Таким образом, тип и класс в **Python** становятся синонимами.

Интерпретатор языка **Python** всегда может сказать, к какому типу относится объект. Однако с точки зрения применимости объекта в операции его принадлежность к классу не играет решающей роли: гораздо важнее, какие методы поддерживает объект.

Примечание:

Пока что в **Python** есть "классические" и "новые" классы. Первые классы определяются сами по себе, а вторые обязательно ведут свою родословную от класса `object`. Для целей данного изложения разница между этими видами классов не имеет значения.

Экземпляры классов могут появляться в программе не только из литералов или в результате операций. Обычно для получения объекта класса достаточно вызвать **конструктор** этого класса с некоторыми параметрами. Объект-класс, как и объект-функция, может быть вызван. Это и будет вызовом конструктора:

```
>>> import sets
>>> s = sets.Set([1, 2, 3])
```

В этом примере модуль `sets` содержит определение класса `Set`. Вызывается конструктор этого класса с параметром `[1, 2, 3]`. В результате с именем `s` будет связан объект-множество из трех элементов `1, 2, 3`.

Следует заметить, что, кроме конструктора, определенные классы имеют и **деструктор** - метод, который вызывается при уничтожении объекта. В языке **Python** объект уничтожается в случае удаления последней ссылки на него либо в результате сборки мусора, если объект оказался в неиспользуемом цикле ссылок. Так как **Python** сам управляет распределением памяти, деструкторы в нем нужны очень редко. Обычно в том случае, когда объект управляет ресурсом, который нужно корректно вернуть в определенное состояние.

Еще один способ получить объект некоторого типа - использование **функций-фабрик**. По синтаксису вызов функции-фабрики не отличается от вызова конструктора класса.

Определение класса

Пусть в ходе анализа данной предметной области необходимо определить класс Граф. Граф - это множество вершин и набор ребер, попарно соединяющий эти вершины. Над графом можно проделывать операции, такие как добавление вершины, ребра, проверка наличия ребра в графе и т.п. На языке **Python** определение класса может выглядеть так:

```
from sets import Set as set  # тип для множества

class G:
  def __init__(self, V, E):
    self.vertices = set(V)
    self.edges = set(E)

  def add_vertex(self, v):
    self.vertices.add(v)

  def add_edge(self, (v1, v2)):
    self.vertices.add(v1)
    self.vertices.add(v2)
    self.edges.add((v1, v2))

  def has_edge(self, (v1, v2)):
    return (v1, v2) in self.edges

  def __str__(self):
    return "%s; %s" % (self.vertices, self.edges)
```

Использовать класс можно следующим образом:

```
g = G([1, 2, 3, 4], [(1, 2), (2, 3), (2, 4)])

print g
g.add_vertex(5)
g.add_edge((5,6))
print g.has_edge((1,6))
print g
```

что даст в результате

```
Set([1, 2, 3, 4]); Set([(2, 4), (1, 2), (2, 3)])
False
Set([1, 2, 3, 4, 5, 6]); Set([(2, 4), (1, 2), (5, 6), (2, 3)])
```

Как видно из предыдущего примера, определить класс не так уж сложно. Конструктор класса имеет специальное имя `__init__`. (Деструктор здесь не нужен, но он бы имел имя `__del__`.) Методы класса определяются в пространстве имен класса. В качестве первого формального аргумента метода принято использовать `self`. Кроме методов в объекте класса имеются два атрибута: `vertices` (вершины) и `edges` (ребра). Для представления объекта `G` в виде строки используется специальный метод `__str__()`.

Принадлежность классу можно выяснить с помощью встроенной функции `isinstance()`:

```
print isinstance(g, G)
```

Инкапсуляция

Обычно считается, что без инкапсуляции невозможно представить себе ООП, что это ключевое понятие. История развития методологий программирования движима борьбой со сложностью разработки программного обеспечения. Сложность больших программных систем, в создании которых участвует сразу большое количество разработчиков, уменьшается, если на верхнем уровне не видно деталей реализации нижних уровней. Собственно, процедурный подход был первым шагом на этом пути. Под **инкапсуляцией (incapsulation**, что можно перевести по-разному, но на нужные ассоциации хорошо наводит слово "обволакивание") понимается сокрытие информации о внутреннем устройстве объекта, при котором работа с объектом может вестись только через его общедоступный **(public)** интерфейс. Таким образом, другие объекты не должны вмешиваться в "дела" объекта, кроме как используя вызовы методов.

В языке **Python** инкапсуляции не придается принципиального значения: ее соблюдение зависит от дисциплинированности программиста. В других языках программирования имеются определенные градации доступности методов объекта.

Доступ к свойствам

В языке **Python** не считается зазорным получить доступ к некоторому атрибуту (не методу) напрямую, если, конечно, этот атрибут описан в документации как часть интерфейса класса. Такие атрибуты называются **свойствами (properties)**. В других языках программирования принято для доступа к свойствам создавать специальные методы (вместо того чтобы напрямую обращаться к общедоступным членам-данным). В **Python** достаточно использовать ссылку на атрибут, если свойство ни на что в объекте не влияет (то есть другие объекты могут его произвольно менять). Если же свойство менее тривиально и требует каких-то действий в самом объекте, его можно описать как свойство (пример взят из документации к **Python**):

```
class C(object):
  def getx(self): return self.__x
  def setx(self, value): self.__x = value
```

```
    def delx(self): del self.__x
    x = property(getx, setx, delx, "I'm the 'x' property.")
```

Синтаксически доступ к свойству `x` будет обычной ссылкой на атрибут:

```
>>> c = C()
>>> c.x = 1
>>> print c.x
1
>>> del c.x
```

А на самом деле будут вызываться соответствующие методы: `setx()`, `getx()`, `delx()`.

Следует отметить, что в экземпляре класса в **Python** можно организовать доступ к любым (даже несуществующим) атрибутам, обрабатывая запрос на доступ к атрибуту группой специальных методов:

`__getattr__(self, name)`	Этот метод объекта вызывается в том случае, если атрибут не найден другим способом (его нет в данном экземпляре или в дереве классов). Здесь **name** - имя атрибута. Метод должен вычислить значение атрибута либо возбудить исключение `AttributeError`. Для получения полного контроля над атрибутами в "новых" классах (то есть потомках `object`) используйте метод `__getattribute__()`.
`__setattr__(self, name, value)`	Этот метод вызывается при присваивании значения некоторому атрибуту. В отличие от `__getattr__()`, метод всегда вызывается, а не только тогда, когда атрибут может быть найден в экземпляре класса, поэтому нужно с осторожностью присваивать значения атрибутам внутри этого метода: это может вызвать рекурсию. Для присваивания значений атрибутов предпочтительнее присваивать словарю `__dict__`: `self.__dict__[name] = value` или (для "новых" классов) - обращение к `__setattr__()` базового класса: `object.__setattr__(self, name, value)`.
`__delattr__(self, name)`	Как можно догадаться из названия, этот метод служит для удаления атрибута.

Следующий небольшой пример демонстрирует все перечисленные моменты. В этом примере из словаря создается объект, именами атрибутов которого будут ключи словаря, а значениями - значения из словаря по заданным ключам:

```
class AttDict(object):
  def __init__(self, dict=None):
    object.__setattr__(self, '_selfdict', dict or {})

  def __getattr__(self, name):
    if self._selfdict.has_key(name):
      return self._selfdict[name]
    else:
      raise AttributeError

  def __setattr__(self, name, value):
    if name[0] != '_':
      self._selfdict[name] = value
    else:
      raise AttributeError

  def __delattr__(self, name):
    if name[0] != '_' and self._selfdict.has_key(name):
      del self._selfdict[name]

ad = AttDict({'a': 1, 'b': 10, 'c': '123'})
```

```
print ad.a, ad.b, ad.c
ad.d = 512
print ad.d
```

Сокрытие данных

Подчеркивание **("_")** в начале имени атрибута указывает на то, что он не входит в общедоступный интерфейс. Обычно применяется одиночное подчеркивание, которое в языке не играет особой роли, но как бы говорит программисту: "этот метод только для внутреннего использования". Двойное подчеркивание работает как указание на то, что атрибут - приватный. При этом атрибут все же доступен, но уже под другим именем, что и иллюстрируется ниже:

```
>>> class X:
...   x = 0
...   _x = 0
...   __x = 0
...
>>> dir(X)
['_X__x', '__doc__', '__module__', '_x', 'x']
```

Полиморфизм

В переводе с греческого полиморфизм означает "многоформие". Так в информатике называют возможность использования одного имени для выполнения различных действий.

Можно встретить множество определений полиморфизма (также есть несколько видов полиморфизма) в зависимости от языка программирования. Как правило, в качестве примера проявления полиморфизма приводят переопределение методов в подклассах. При этом можно создать функцию, требующую формального аргумента - экземпляра базового класса, а в качестве фактического аргумента давать экземпляр подкласса. Функция будет вызывать метод объекта с именем, а за именем будут скрываться различные действия. В связи с этим полиморфизм обычно связывают с иерархией наследования.

В **Python** полиморфизм связан не с наследованием, а с набором и смыслом доступных методов в экземпляре класса. Ниже будет показано, что, имея определенные методы, можно воссоздать класс для строки или любого другого встроенного типа. Для этого необходимо определить свойственный типу набор методов. Конечно, нужный набор методов можно получить и с помощью наследования, но в **Python** это не только не обязательно, но иногда и противоречит здравому смыслу.

При написании функции в **Python** обычно не проверяется, к какому типу (классу) относится тот или иной аргумент: некоторые методы просто применяются к переданному объекту. Тем самым функции получаются максимально обобщенными: они не требуют от объектов-параметров большего, чем наличие методов с определенным именем, набором аргументов и семантикой.

Следующий пример показывает полиморфизм в том виде, в котором он свойственен **Python**:

```
def get_last(x):
  return x[-1]

print get_last([1, 2, 3])
print get_last("abcd")
```

Описанной функции будет подходить в качестве аргумента все, от чего можно взять индекс `-1` (последний элемент). Однако семантика "взятие последнего элемента" выполняется только для последовательностей. Функция будет работать и для словарей, но смысл при этом будет немного другой.

Имитация типов

Для иллюстрации понятия полиморфизма можно построить собственный тип, похожий на встроенный тип "функция". Построить класс, объекты которого вызываются подобно методам или функциям, можно так:

```
class CountArgs(object):
  def __call__(self, *args, **kwargs):
    return len(args) + len(kwargs)

cc = CountArgs()
print cc(1, 3, 4)
```

Как видно из этого примера, экземпляры класса `CountArgs` можно вызывать подобно функциям (в результате будет возвращено количество переданных параметров). При попытке вызова экземпляра на самом деле будет вызван метод `__call__()` со всеми аргументами.

Следующий пример показывает, что сравнением экземпляров класса тоже можно управлять:

```
class Point:
  def __init__(self, x, y):
    self.coord = (x, y)
  def __nonzero__(self):
    return self.coord[0] != 0 or self.coord[1] != 0
  def __cmp__(self, p):
    return cmp(self.coord, p.coord)

for x in range(-3, 4):
  for y in range(-3, 4):
    if Point(x, y) < Point(y, x):
      print "*",
    elif Point(x, y):
      print ".",
    else:
      print "o",
  print
```

Программа выведет:

```
. * * * * * *
. . * * * * *
. . . * * * *
. . . o * * *
. . . . . * *
. . . . . . *
. . . . . . .
```

В данной программе класс `Point` (Точка) имеет метод `__nonzero__()`, который определяет истинностное значение объекта класса. Истину будут давать только точки, отличные от `(0, 0)`. Другой метод - `__cmp__()` - вызывается при необходимости сравнить точку и другой объект (имеющий как и точка атрибут **coord**, который содержит кортеж как минимум из двух элементов). Нужно заметить, что вместо `__cmp__` можно определить отдельные методы для операций сравнения: `__lt__`, `__le__`, `__ne__`, `__eq__`, `__ge__`, `__gt__` (для `<`, `<=`, `!=`, `<>`, `=>`, `>` соответственно).

Достаточно легко имитировать и числовые типы. Класс, который пользуется удобством синтаксиса инфиксного `+`, можно определить так:

```
class Plussable:
```

```
  def __add__(self, x):
    ...
  def __radd__(self, x):
    ...
  def __iadd__(self, x):
    ...
```

Здесь метод `__add__()` вызывается, когда экземпляр класса `Plussable` стоит слева от сложения, `__radd__()` - если справа от сложения и метод слева от него не имеет метода `__add__()`. Метод `__iadd__()` нужен для реализации `+=`.

Отношения между классами

Наследование

На практике часто возникает ситуация, когда в предметной области выделены очень близкие, но вместе с тем неодинаковые классы. Одним из способов сокращения описания классов за счет использования их сходства является выстраивание классов в **иерархию**. В корне этой иерархии стоит базовый класс, от которого нижележащие классы иерархии **наследуют** свои атрибуты, уточняя и расширяя поведение вышележащего класса. Обычно принципом построения классификации является отношение **"IS-A" ("**ЕСТЬ**")**. Например, класс Окружность в программе - графическом редакторе может быть унаследован от класса Геометрическая Фигура. При этом Окружность будет являться **подклассом (**или субклассом**)** для класса Геометрическая Фигура, а Геометрическая Фигура **- надклассом (**или суперклассом**)** для класса Окружность.

В языке **Python** во главе иерархии **("**новых**")** классов стоит класс `object`. Для ориентации в иерархии существуют некоторые встроенные функции, которые будут рассмотрены ниже. Функция `issubclass(x, y)` может сказать, является ли класс `x` подклассом класса `y`:

```
>>> class A(object): pass
...
>>> class B(A): pass
...
>>> issubclass(A, object)
True
>>> issubclass(B, A)
True
>>> issubclass(B, object)
True
>>> issubclass(A, str)
False
>>> issubclass(A, A)  # класс является подклассом самого себя
True
```

В основе построения классификации всегда стоит принцип, играющий наиболее важную роль в анализируемой и моделируемой системе. Следует заметить, что одним из "перегибов" при использовании ОО методологии является искусственное выстраивание иерархии классов. Например, не стоит наследовать класс Машина от класса Колесо **(**внимательные заметят, что здесь отношение другое: колесо является частью машины**)**.

Класс называется **абстрактным**, если он предназначен только для наследования. Экземпляры абстрактного класса обычно не имеют большого смысла. Классы с рабочими экземплярами называются **конкретными**.

В **Python** примером абстрактного класса является встроенный тип `basestring`, у которого есть конкретные подклассы `str` и `unicode`.

Множественное наследование

В отличие, например, от **Java**, в языке **Python** можно наследовать класс от нескольких классов. Такая ситуация называется **множественным наследованием (multiple inheritance)**.

Класс, получаемый при множественном наследовании, объединяет поведение своих надклассов, комбинируя стоящие за ними абстракции.

Использовать множественное наследование следует очень осторожно, а необходимость в нем возникает реже одиночного.

- Множественное наследование можно применить для получения класса с заданными общедоступными методами, причем методы задает один родительский класс, а реализуются они на основе методов второго класса. Первый класс может быть полностью абстрактным.
- Множественное наследование применяется для добавления **примесей (mixins)**. Примесь - специально сконструированный класс, добавляющий в некоторый класс какую-либо черту поведения (привнесением атрибутов). Примеси обычно являются абстрактными классами.
- Изредка множественное наследование применяется в своем основном смысле, когда объекты класса, получающегося в результате множественного наследования, предназначаются для использования в качестве объектов всех родительских классов.

В случае с **Python** наследование можно считать одним из способов собрать нужные комбинации методов в серии классов:

```
class A:
  def a(self): return 'a'
class B:
  def b(self): return 'b'
class C:
  def c(self): return 'c'

class AB(A, B):
  pass
class BC(B, C):
  pass
class ABC(A, B, C):
  pass
```

Впрочем, собрать нужные методы можно и по-другому, без использования наследования:

```
def ma(self): return 'a'
def mb(self): return 'b'
def mc(self): return 'c'

class AB:
  a = ma
  b = mb

class BC:
  b = mb
  c = mc

class ABC:
  a = ma
  b = mb
  c = mc
```

Порядок разрешения методов

В случае, когда надклассы имеют одинаковые методы, использование того или иного метода определяется **порядком разрешения методов (method resolution order).** Для "новых" классов узнать этот порядок очень просто с помощью атрибута `__mro__`:

```
>>> str.__mro__
(<type 'str'>, <type 'basestring'>, <type 'object'>)
```

Это означает, что сначала методы ищутся в классе `str`, затем в `basestring`, а уже потом - в `object`.

Для "классических" классов порядок несколько отличается от порядка разрешения методов в "новых" классах. Нужно стараться избегать множественного наследования или применять его очень аккуратно.

Агрегация

Контейнеры

Под **контейнером** обычно понимают объект, основным назначением которого является хранение и обеспечение доступа к другим объектам. Контейнеры реализуют отношение **"HAS-A" ("**ИМЕЕТ**")** между объектами. Встроенные типы, список и словарь **--** яркие примеры контейнеров. Можно построить собственные типы контейнеров, которые будут иметь свою логику доступа к хранимым объектам. В контейнере хранятся не сами объекты, а ссылки на них.

Для практических нужд в **Python** обычно хватает встроенных контейнеров **(**словаря и списка**),** но если это необходимо, можно создать и другие. Ниже приведен класс Стек, реализованный на базе списка:

```
class Stack:
  def __init__(self):
    """Инициализация стека"""
    self._stack = []
  def top(self):
    """Возвратить вершину стека (не снимая)"""
    return self._stack[-1]
  def pop(self):
    """Снять со стека элемент"""
    return self._stack.pop()
  def push(self, x):
    """Поместить элемент на стек"""
    self._stack.append(x)
  def __len__(self):
    """Количество элементов в стеке"""
    return len(self._stack)
  def __str__(self):
    """Представление в виде строки"""
    return " : ".join(["%s" % e for e in self._stack])
```

Использование:

```
>>> s = Stack()
>>> s.push(1)
>>> s.push(2)
>>> s.push("abc")
>>> print s.pop()
abc
>>> print len(s)
2
>>> print s
1 : 2
```

Таким образом, контейнеры позволяют управлять набором (любых) других объектов в соответствии со структурой их хранения, не вмешиваясь во внутренние дела объектов. Узнав интерфейс класса `Stack`, можно и не догадаться, что он реализован на основе списка, и каким именно образом он реализован с помощью него. Но для использования стека это не важно.

Примечание:

В данном примере для краткости изложения не учтено, что в результате некоторых действий могут возбуждаться исключения. Например, при попытке снять элемент с вершины пустого стека.

Итераторы

Итераторы - это объекты, которые предоставляют последовательный доступ к элементам контейнера (или генерируемым "на лету" объектам). Итератор позволяет перебирать элементы, абстрагируясь от реализации того контейнера, откуда он их берет (если этот контейнер вообще есть).

В следующем примере приведен итератор, выдающий значения из списка по принципу "считалочки" по `N`:

```
class Zahlreim:
  def __init__(self, lst, n):
    self.n = n
    self.lst = lst
    self.current = 0
  def __iter__(self):
    return self
  def next(self):
    if self.lst:
      self.current = (self.current + self.n - 1) % len(self.lst)
      return self.lst.pop(self.current)
    else:
      raise StopIteration

print range(1, 11)
for i in Zahlreim(range(1, 11), 5):
  print i,
```

Программа выдаст

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
5 10 6 2 9 8 1 4 7 3
```

В этой программе делегировано управление доступом к элементам списка (или любого другого контейнера, имеющего метод `pop(n)` для взятия и удаления `n`-го элемента) классу-итератору. Итератор должен иметь метод `next()` и возбуждать исключение `StopIteration` по завершении итераций. Кроме того, метод `__iter__()` должен выдавать итератор по экземпляру класса (в данном случае итератор - он сам (`self`)).

В настоящее время итераторы приобретают все большее значение, и о них много говорилось в лекции по функциональному программированию.

Ассоциация

Если в случае агрегации имеется довольно четкое отношение "ИМЕЕТ" **(HAS-A)** или "СОДЕРЖИТСЯ-В", которое даже отражено в синтаксисе **Python**:

```
lst = [1, 2, 3]
```

```
if 1 in lst:
   ...
```

то в случае ассоциации ссылка на экземпляр другого класса используется без отношения включения одного в другой или принадлежности. О таком отношении между классами говорят как об отношении **USE-A (**"ИСПОЛЬЗУЕТ"**).** Это достаточно общее отношение зависимости между классами.

В языке **Python** границы между агрегацией и ассоциацией несколько размыты, так как объекты при агрегации обычно не хранятся в области памяти, выделенной под контейнер **(**хранятся только ссылки**).**

Объекты могут также ссылаться друг на друга. В этом случае возникают **циклические ссылки**, которые при неаккуратном использовании могут привести **(**в старых версиях **Python)** к утечкам памяти. В новых версиях **Python** для циклических ссылок работает сборщик мусора.

Разумеется, при "чистой" агрегации циклических ссылок не возникает.

Например, при представлении дерева ссылки могут идти от родителей к детям и обратно от каждого дочернего узла к родительскому.

Слабые ссылки

Для обеспечения ассоциаций объектов без свойственных ссылкам проблем с возможностью образования циклических ссылок, в **Python** для сложных структур данных и других видов использования, при которых ссылки не должны мешать удалению объекта, предлагается механизм слабых ссылок. Такая ссылка не учитывается при подсчете ссылок на объект, а значит, объект удаляется с исчезновением последней "сильной" ссылки.

Для работы со слабыми ссылками применяется модуль `weakref`. Основные принципы его работы станут понятны из следующего примера:

```
>>> import weakref
>>>
>>> class MyClass(object):
...  def __str__(self):
...   return "MyClass"
...
>>>
>>> s = MyClass()          # создается экземпляр класса
>>> print s
MyClass
>>> s1 = weakref.proxy(s)  # создается прокси-объект
>>> print s1               # прокси-объект работает как исходный
MyClass
>>> ss = weakref.ref(s)    # создается слабая ссылка на него
>>> print ss()             # вызовом ссылки получается исходный объект
MyClass
>>> del s                  # удаляется единственная сильная ссылка на объект
>>> print ss()             # теперь исходного объекта не существует
None
>>> print s1
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ReferenceError: weakly-referenced object no longer exists
```

К сожалению, поведение прокси-объекта не совсем такое, как у исходного: он не может быть ключом словаря, так как является нехэшируемым.

Статический метод

Иногда необходимо использовать метод, принадлежащий классу, а не его экземпляру. В этом случае можно описать **статический метод**. До появления декораторов (до **Python 2.4**) определять статический метод приходилось следующим образом:

```
class A(object):
  def name():
    return A.__name__
  name = staticmethod(name)

print A.name()
a = A()
print a.name()
```

Статическому методу не передается параметр с экземпляром класса. Он ему попросту не нужен.

В **Python 2.4** для применения описателей **(descriptors)** был придуман новый синтаксис - декораторы:

```
class A(object):

  @staticmethod
  def name():
    return A.__name__
```

Смысл декоратора в том, что он "пропускает" определяемую функцию (или метод) через заданную в нем функцию. Теперь писать `name` три раза не потребовалось. Декораторов может быть несколько, и применяются они в обратном порядке.

Метод класса

Если статический метод имеет свои аналоги в **C++** и **Java**, то метод класса основан на том, что в **Python** классы являются объектами. В отличие от статического метода, в метод класса первым параметром передается объект-класс. Вместо `self` для подчеркивания принадлежности метода к методам класса принято использовать `cls`.

Пример использования метода класса можно найти в модуле `tree` пакета `nltk` **(Natural Language ToolKit**, набор инструментов для естественного языка**)**. Ниже приведен лишь фрагмент определения класса `Tree` (базового класса для других подклассов**)**. Метод `convert` класса `Tree` определяет процедуру преобразования дерева одного типа в дерево другого типа. Эта процедура абстрагируется от деталей реализации конкретных типов, описывая обобщенный алгоритм преобразования:

```
class Tree:
  # ...
  def convert(cls, val):

      if isinstance(val, Tree):
          children = [cls.convert(child) for child in val]
          return cls(val.node, children)
      else:
          return val
  convert = classmethod(convert)
```

Пример использования (взят из строки документации метода `convert()`):

```
>>> # Преобразовать tree в экземпляр класса Tree
>>> tree = Tree.convert(tree)
>>> # " " " " " ParentedTree
>>> tree = ParentedTree.convert(tree)
>>> # " " " " " MultiParentedTree
```

```
>>> tree = MultiParentedTree.convert(tree)
```

Метод класса позволяет более естественно описывать действия, которые связаны в основном с классами, а не с методами экземпляра класса.

Метаклассы

Еще одним отношением между классами является отношение класс-метакласс. **Метакласс** можно считать "высшим пилотажем" объектно-ориентированного программирования, но, к счастью, в **Python** можно создавать собственные метаклассы.

В **Python** класс тоже является объектом, поэтому ничего не мешает написать класс, назначением которого будет создание других классов динамически, во время выполнения программы.

Пример, в котором класс порождается динамически в функции-фабрике классов:

```
def cls_factory_f(func):
  class X(object):
    pass
  setattr(X, func.__name__, func)
  return X
```

Использование будет выглядеть так:

```
def my_method(self):
  print "self:", self

My_Class = cls_factory_f(my_method)
my_object = My_Class()
my_object.my_method()
```

В этом примере функция `cls_factory_f()` возвращает класс с единственным методом, в качестве которого используется функция, переданная ей как аргумент. От этого класса можно получить экземпляры, а затем у экземпляров - вызвать метод `my_method`.

Теперь можно задаться целью построить класс, экземплярами которого будут классы. Такой класс, от которого порождаются классы, и называется **метаклассом**.

В **Python** имеется класс `type`, который на деле является метаклассом. Вот как с помощью его конструктора можно создать класс:

```
def my_method(self):
  print "self:", self

My_Class = type('My_Class', (object,), {'my_method': my_method})
```

В качестве первого параметра `type` передается имя класса, второй параметр - базовые классы для данного класса, третий - атрибуты.

В результате получится класс, эквивалентный следующему:

```
class My_Class(object):
  def my_method(self):
    print "self:", self
```

Но самое интересное начинается при попытке составить собственный метакласс. Проще всего наследовать метакласс от метакласса **type** (пример взят из статьи Дэвида Мертца):

```
>>> class My_Type(type):
```

```
...  def __new__(cls, name, bases, dict):
...   print "Выделение памяти под класс", name
...   return type.__new__(cls, name, bases, dict)
...  def __init__(cls, name, bases, dict):
...   print "Инициализация класса", name
...   return super(My_Type, cls).__init__(cls, name, bases, dict)
...
>>> my = My_Type("X", (), {})
Выделение памяти под класс X
Инициализация класса X
```

В этом примере не происходит вмешательство в создание класса. Но в `__new__()` и `__init__()` имеется **полный** программный контроль над создаваемым классом в период выполнения.

Примечание:

Следует заметить, что в метаклассах принято называть первый аргумент методов не `self`, а `cls`, чтобы напомнить, что экземпляр, над которым работает программист, является не просто объектом, а классом.

Мультиметоды

Некоторые объектно-ориентированные "штучки" не входят в стандартный **Python** или стандартную библиотеку. Ниже будут рассмотрены **мультиметоды** - методы, сочетающие объекты сразу нескольких различных классов. Например, сложение двух чисел различных типов фактически требует использования мультиметода. Если "одиночный" метод достаточно задать для каждого класса, то мультиметод требует задания для каждого сочетания классов, которые он обслуживает:

```
>>> import operator
>>> operator.add(1, 2)
3
>>> operator.add(1.0, 2)
3.0
>>> operator.add(1, 2.0)
3.0
>>> operator.add(1, 1+2j)
(2+2j)
>>> operator.add(1+2j, 1)
(2+2j)
```

В этом примере `operator.add` ведет себя как мультиметод, выполняя разные действия для различных комбинаций параметров.

Для организации собственных мультиметодов можно воспользоваться модулем `Multimethod` (автор **Neel Krishnaswami)**, который легко найти в Интернете. Следующий пример, адаптированный из документации модуля, показывает построение собственного мультиметода:

```
from Multimethod import Method, Generic, AmbiguousMethodError

# классы, для которых будет определен мультиметод
class A: pass
class B(A): pass

# функции мультиметода
def m1(a, b): return 'AA'
def m2(a, b): return 'AB'
def m3(a, b): return 'BA'
```

```
# определение мультиметода (без одной функции)
g = Generic()
g.add_method(Method((A, A), m1))
g.add_method(Method((A, B), m2))
g.add_method(Method((B, A), m3))

# применение мультиметода
try:
    print 'Типы аргументов:', 'Результат'
    print 'A, A:', g(A(), A())
    print 'A, B:', g(A(), B())
    print 'B, A:', g(B(), A())
    print 'B, B:', g(B(), B())
except AmbiguousMethodError:
    print 'Неоднозначный выбор метода'
```

Устойчивые объекты

Для того чтобы объекты жили дольше, чем создавшая их программа, необходим механизм их представления в виде последовательности байтов. Во второй лекции уже рассматривался модуль `pickle`, который позволяет сериализовать объекты.

Здесь же будет показано, как класс может способствовать более качественному консервированию объекта. Следующие методы, если их определить в классе, позволяют управлять работой модуля `pickle` и рассмотренной ранее функции глубокого копирования. Другими словами, правильно составленные методы дают возможность воссоздать объект, передав самую суть - состояние объекта.

`__getinitargs__()`	Должен возвращать кортеж из аргументов, который будет передаваться на вход метода `__init__()` при создании объекта.
`__getstate__()`	Должен возвращать словарь, в котором выражено состояние объекта. Если этот метод в классе определен, то используется атрибут `__dict__`, который есть у каждого объекта.
`__setstate__(state)`	Должен восстанавливать объекту ранее сохраненное состояние `state`.

В следующем примере класс **CC** управляет своим копированием (точно так же экземпляры этого класса смогут консервироваться и расконсервироваться при помощи модуля `pickle`):

```
from time import time, gmtime
import copy
class CC:
    def __init__(self, created=time()):
        self.created = created
        self.created_gmtime = gmtime(created)
        self._copied = 1
        print id(self), "init", created
    def __getinitargs__(self):
        print id(self), "getinitargs", self.created
        return (self.created,)
    def __getstate__(self):
        print id(self), "getstate", self.created
        return {'_copied': self._copied}
    def __setstate__(self, dict):
        print id(self), "setstate", dict
        self._copied = dict['_copied'] + 1
    def __repr__(self):
        return "%s obj: %s %s %s" % (id(self), self._copied,
                                     self.created, self.created_gmtime)

a = CC()
```

```
print a
b = copy.deepcopy(a)
print b
```

В результате будет получено

```
1075715052 init 1102751640.91
1075715052 obj: 1 1102751640.91 (2004, 12, 11, 7, 54, 0, 5, 346, 0)
1075715052 getinitargs 1102751640.91
1075729452 init 1102751640.91
1075715052 getstate 1102751640.91
1075729452 setstate {'copied': 1}
1075729452 obj: 2 1102751640.91 (2004, 12, 11, 7, 54, 0, 5, 346, 0)
```

Состояние объекта состоит из трех атрибутов: `created`, `created_gmtime`, `copied`. Первый из этих атрибутов может быть восстановлен передачей параметра конструктору. Второй - вычислен в конструкторе на основе первого. А вот третий не входит в интерфейс класса и может быть передан только через механизм `getstate`/`setstate`. Причем, по смыслу этого атрибута при каждом копировании он должен увеличиваться на единицу (хотя в разных случаях атрибут может требовать других действий или не требовать их вообще). Следует включить отладочные операторы вывода, чтобы отследить последовательность вызовов методов при копировании.

Механизм `getstate`/`setstate` позволяет передавать при копировании только то, что нужно для воссоздания объекта, тогда как атрибут `__dict__` может содержать много лишнего. Более того, `__dict__` может содержать объекты, которые просто так сериализации не поддаются, и поэтому `getstate`/`setstate` - единственная возможность обойти подобные ограничения.

Примечание:

Следует заметить, что сериализация функций и классов - лишь кажущаяся: на принимающей стороне должны быть определения функций и классов, передаются же только их имена и принадлежность модулям.

Для хранения объектов используются не только простейшие механизмы хранения вроде `pickle.dump`/`pickle.load` или полки `shelve`. Сериализованные объекты **Python** можно хранить в специализированных хранилищах объектов (например, **ZODB**) или реляционных базах данных.

Это также касается передачи объектов по сетям передачи данных. Если простейшие объекты (вроде строк или чисел) можно передавать напрямую через **HTTP**, **XML-RPC**, **SOAP** и т.д., где они имеют собственный тип, то произвольные объекты необходимо консервировать на передающей стороне и расконсервировать на принимающей.

Критика ООП

Объектно-ориентированный подход сегодня считается "самым передовым". Однако не следует слепо верить в его всемогущество. Отдача (в виде скорости разработки) от объектного проектирования чувствуется только в больших проектах и в проектах, которые родственны объектному подходу: построение графического интерфейса, моделирование систем и т.п.

Также спорна большая гибкость объектных программ к изменениям. Она зависит от того, вносится ли новый метод (для серии объектов) или новый тип объекта. При процедурном подходе при появлении нового метода пишется отдельная процедура, в которой в каждой ветке алгоритма обрабатывается свой тип данных (то есть такое изменение требует редактирования одного места в коде). При ООП изменять придется каждый класс, внося в него новый метод (то есть изменения в нескольких местах). Зато ООП выигрывает при

внесении нового типа данных: ведь изменения происходят только в одном месте, где описываются все методы для данного типа. При процедурном подходе приходится изменять несколько процедур. Сказанное иллюстрируется ниже. Пусть имеются классы `A`, `B`, `C` и методы `a`, `b`, `c`:

```
# ООП
class A:
  def a(): ...
  def b(): ...
  def c(): ...

class B:
  def a(): ...
  def b(): ...
  def c(): ...

class C:
  def a(): ...
  def b(): ...
  def c(): ...

# процедурный подход

def a(x):
  if type(x) is A: ...
  if type(x) is B: ...
  if type(x) is C: ...

def b(x):
  if type(x) is A: ...
  if type(x) is B: ...
  if type(x) is C: ...

def c(x):
  if type(x) is A: ...
  if type(x) is B: ...
  if type(x) is C: ...
```

При внесении нового типа объекта изменения в ОО-программе затрагивают только один модуль, а в процедурной - все процедуры:

```
# ООП

class D:
  def a(): ...
  def b(): ...
  def c(): ...

# процедурный подход

def a(x):
  if type(x) is A: ...
  if type(x) is B: ...
  if type(x) is C: ...
  if type(x) is D: ...

def b(x):
  if type(x) is A: ...
  if type(x) is B: ...
  if type(x) is C: ...
  if type(x) is D: ...

def c(x):
  if type(x) is A: ...
  if type(x) is B: ...
```

```
  if type(x) is C: ...
  if type(x) is D: ...
```

И наоборот, теперь нужно добавить новый метод обработки. При процедурном подходе просто пишется новая процедура, а вот для объектного приходится изменять все классы:

```
# процедурный подход

def d(x):
  if type(x) is A: ...
  if type(x) is B: ...
  if type(x) is C: ...

# ООП

class A:
  def a(): ...
  def b(): ...
  def c(): ...
  def d(): ...

class B:
  def a(): ...
  def b(): ...
  def c(): ...
  def d(): ...

class C:
  def a(): ...
  def b(): ...
  def c(): ...
  def d(): ...
```

Язык программирования **Python** изначально был ориентирован на практические нужды. Приведенное выше выражается в стандартной библиотеке **Python**, то есть в том, что там применяются и функции **(**обычно сильно обобщенные на довольно широкий круг входных данных**),** и классы **(**когда операции достаточно специфичны**).** Обобщенная природа функций **Python** и полиморфизм, не завязанный целиком на наследовании **-** вот свойства языка **Python**, позволяющие иметь большую гибкость в комбинации процедурного и объектно**-**ориентированного подходов**.**

Заключение

Даже достаточно неформальное введение в ООП потребовало определения большого количества терминов. В лекции была сделана попытка с помощью примеров передать не столько букву, сколько дух терминологии ООП. Были рассмотрены все базовые понятия: объект, тип, класс и виды отношений между объектами **(IS-A**, **HAS-A**, **USE-A).** Слушатели получили представление о том, что такое инкапсуляция и полиморфизм в стиле ООП, а также наследование **-** продление времени жизни объекта за рамками исполняющейся программы, известное как устойчивость объекта **(object persistence).** Были указаны недостатки ООП, но при этом весь предыдущий материал объективно свидетельствовал о достоинствах этого подхода**.**

Возможно, что именно эта лекция приведет слушателей к пониманию ООП, пригодному и удобному для практической работы**.**

Ссылки

Дэвид Мертц **http://www-106.ibm.com/developerworks/linux/library/l-pymeta.html**

Лекция #5: Численные алгоритмы. Матричные вычисления

Numeric Python - это несколько модулей для вычислений с многомерными массивами, необходимых для многих численных приложений. Модуль `Numeric` вносит в **Python** возможности таких пакетов и систем как **MatLab**, **Octave** (аналог **MatLab**), **APL**, **J**, **S+**, **IDL**. Пользователи найдут **Numeric** достаточно простым и удобным. Стоит заметить, что некоторые синтаксические возможности **Python** (связанные с использованием срезов) были специально разработаны для **Numeric**.

Numeric Python имеет средства для:

- матричных вычислений `LinearAlgebra`;
- быстрого преобразования Фурье `FFT`;
- работы с недостающими экспериментальными данными `MA`;
- статистического моделирования `RNG`;
- эмуляции базовых функций программы **MatLab**.

Модуль Numeric

Модуль `Numeric` определяет полноценный тип-массив и содержит большое число функций для операций с массивами. **Массив -** это набор однородных элементов, доступных по индексам. Массивы модуля `Numeric` могут быть многомерными, то есть иметь более одной **размерности**.

Создание массива

Для создания массива можно использовать функцию `array()` с указанием содержимого массива (в виде вложенных списков) и типа. Функция `array()` делает копию, если ее аргумент - массив. Функция `asarray()` работает аналогично, но не создает нового массива, когда ее аргумент уже является массивом:

```
>>> from Numeric import *
>>> print array([[1, 2], [3, 4], [5, 6]])
[[1 2]
 [3 4]
 [5 6]]
>>> print array([[1, 2, 3], [4, 5, 6]], Float)
[[ 1.  2.  3.]
 [ 4.  5.  6.]]
>>> print array([78, 85, 77, 69, 82, 73, 67], 'c')
[N U M E R I C]
```

В качестве элементов массива можно использовать следующие типы: `Int8-Int32`, `UnsignedInt8-UnsignedInt32`, `Float8-Float64`, `Complex8-Complex64` и `PyObject`. Числа **8**, **16**, **32** и **64** показывают количество битов для хранения величины. Типы `Int`, `UnsignedInteger`, `Float` и `Complex` соответствуют наибольшим принятым на данной платформе значениям. В массиве можно также хранить ссылки на произвольные объекты.

Количество размерностей и длина массива по каждой оси называются формой массива **(shape)**. Доступ к форме массива реализуется через атрибут `shape`:

```
>>> from Numeric import *
>>> a = array(range(15), Int)
>>> print a.shape
(15,)
>>> print a
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14]
```

```
>>> a.shape = (3, 5)
>>> print a.shape
(3, 5)
>>> print a
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]]
```

Методы массивов

Придать нужную форму массиву можно функцией `Numeric.reshape()`. Эта функция сразу создает объект-массив нужной формы из последовательности.

```
>>> import Numeric
>>> print Numeric.reshape("абракадабр", (5, -1))
[[а б]
 [р а]
 [к а]
 [д а]
 [б р]]
```

В этом примере `-1` в указании формы говорит о том, что соответствующее значение можно вычислить. Общее количество элементов массива известно **(10)**, поэтому длину вдоль одной из размерностей задавать не обязательно.

Через атрибут `flat` можно получить одномерное представление массива:

```
>>> a = array([[1, 2], [3, 4]])
>>> b = a.flat
>>> b
array([1, 2, 3, 4])
>>> b[0] = 9
>>> b
array([9, 2, 3, 4])
>>> a
array([[9, 2],
       [3, 4]])
```

Следует заметить, что это новый вид того же массива, поэтому присваивание значений его элементам приводит к изменениям в исходном массиве.

Функция `Numeric.resize()`похожа на `Numeric.reshape`, но может подстраивать число элементов:

```
>>> print Numeric.resize("NUMERIC", (3, 2))
[[N U]
 [M E]
 [R I]]
>>> print Numeric.resize("NUMERIC", (3, 4))
[[N U M E]
 [R I C N]
 [U M E R]]
```

Функция `Numeric.zeros()` порождает массив из одних нулей, а `Numeric.ones()` - из одних единиц. Единичную матрицу можно получить с помощью функции `Numeric.identity(n)`:

```
>>> print Numeric.zeros((2,3))
[[0 0 0]
 [0 0 0]]
>>> print Numeric.ones((2,3))
[[1 1 1]
 [1 1 1]]
```

```
>>> print Numeric.identity(4)
[[1 0 0 0]
 [0 1 0 0]
 [0 0 1 0]
 [0 0 0 1]]
```

Для копирования массивов можно использовать метод `copy()`:

```
>>> import Numeric
>>> a = Numeric.arrayrange(9)
>>> a.shape = (3, 3)
>>> print a
[[0 1 2]
 [3 4 5]
 [6 7 8]]
>>> a1 = a.copy()
>>> a1[0, 1] = -1   # операция над копией
>>> print a
[[0 1 2]
 [3 4 5]
 [6 7 8]]
```

Массив можно превратить обратно в список с помощью метода `tolist()`:

```
>>> a.tolist()
[[0, 1, 2], [3, 4, 5], [6, 7, 8]]
```

Срезы

Объекты-массивы `Numeric` используют расширенный синтаксис выделения среза. Следующие примеры иллюстрируют различные варианты записи срезов. Функция `Numeric.arrayrange()` является аналогом `range()` для массивов.

```
>>> import Numeric
>>> a = Numeric.arrayrange(24) + 1
>>> a.shape = (4, 6)
>>> print a             # исходный массив
[[ 1  2  3  4  5  6]
 [ 7  8  9 10 11 12]
 [13 14 15 16 17 18]
 [19 20 21 22 23 24]]
>>> print a[1,2]        # элемент 1,2
9
>>> print a[1,:]        # строка 1
[ 7  8  9 10 11 12]
>>> print a[1]          # тоже строка 1
[ 7  8  9 10 11 12]
>>> print a[:,1]        # столбец 1
[ 2  8 14 20]
>>> print a[-2,:]       # предпоследняя строка
[13 14 15 16 17 18]
>>> print a[0:2,1:3]    # окно 2x2
[[2 3]
 [8 9]]
>>> print a[1,::3]      # каждый третий элемент строки 1
[ 7 10]
>>> print a[:,::-1]     # элементы строк в обратном порядке
[[ 6  5  4  3  2  1]
 [12 11 10  9  8  7]
 [18 17 16 15 14 13]
 [24 23 22 21 20 19]]
```

Срез не копирует массив (как это имеет место со списками), а дает доступ к некоторой части массива. Далее в примере меняется на **0** каждый третий элемент строки **1:**

```
>>> a[1,::3] = Numeric.array([0,0])
>>> print a
[[ 1  2  3  4  5  6]
 [ 0  8  9  0 11 12]
 [13 14 15 16 17 18]
 [19 20 21 22 23 24]]
```

В следующих примерах находит применение достаточно редкая синтаксическая конструкция: срез с многоточием (`Ellipsis`). Многоточие ставится для указания произвольного числа пропущенных размерностей (`:,:,...,:`):

```
>>> import Numeric
>>> a = Numeric.arrayrange(24) + 1
>>> a.shape = (2,2,2,3)
>>> print a
[[[[ 1  2  3]
   [ 4  5  6]]
  [[ 7  8  9]
   [10 11 12]]]
 [[[13 14 15]
   [16 17 18]]
  [[19 20 21]
   [22 23 24]]]]
>>> print a[0,...]          # 0-й блок
[[[ 1  2  3]
  [ 4  5  6]]
 [[ 7  8  9]
  [10 11 12]]]
>>> print a[0,:,:,0]        # срез по первой и последней размерностям
[[ 1  4]
 [ 7 10]]
>>> print a[0,...,0]        # то же, но с использованием многоточия
[[ 1  4]
 [ 7 10]]
```

Универсальные функции

Модуль **Numeric** определяет набор функций для применения к элементам массива. Функции применимы не только к массивам, но и к последовательностям (к сожалению, итераторы пока не поддерживаются). В результате получаются массивы.

Функция	Описание
`add(x, y), subtract(x, y)`	Сложение и вычитание
`multiply(x, y), divide(x, y)`	Умножение и деление
`remainder(x, y), fmod(x, y)`	Получение остатка от деления (для целых чисел и чисел с плавающей запятой)
`power(x)`	Возведение в степень
`sqrt(x)`	Извлечение корня квадратного
`negative(x), absolute(x), fabs(x)`	Смена знака и абсолютное значение
`ceil(x), floor(x)`	Наименьшее (наибольшее) целое, большее (меньшее) или равное аргументу
`hypot(x, y)`	Длина гипотенузы (даны длины двух катетов)
`sin(x), cos(x), tan(x)`	Тригонометрические функции
`arcsin(x), arccos(x),`	Обратные тригонометрические функции

`arctan(x)`	
`arctan2(x, y)`	Арктангенс от частного аргумента
`sinh(x), cosh(x), tanh(x)`	Гиперболические функции
`arcsinh(x), arccosh(x), arctanh(x)`	Обратные гиперболические функции
`exp(x)`	Экспонента **(**e^x**)**
`log(x), log10(x)`	Натуральный и десятичный логарифмы
`maximum(x, y), minimum(x, y)`	Максимум и минимум
`conjugate(x)`	Сопряжение **(**для комплексных чисел**)**
`equal(x, y), not_equal(x, y)`	Равно**,** не равно
`greater(x, y), greater_equal(x, y)`	Больше**,** больше или равно
`less(x, y), less_equal(x, y)`	Меньше**,** меньше или равно
`logical_and(x, y), logical_or(x, y)`	Логические И**,** ИЛИ
`logical_xor(x, y)`	Логическое исключающее ИЛИ
`logical_not(x)`	Логические НЕ
`bitwise_and(x, y), bitwise_or(x, y)`	Побитовые И**,** ИЛИ
`bitwise_xor(x, y)`	Побитовое исключающее ИЛИ
`invert(x)`	Побитовая инверсия
`left_shift(x, n), right_shift(x, n)`	Побитовые сдвиги влево и вправо на **n** битов

Перечисленные функции являются объектами типа `ufunc` и применяются к массивам поэлементно**.** Эти функции имеют специальные методы**:**

`accumulate()`	Аккумулирование результата**.**
`outer()`	Внешнее "произведение"**.**
`reduce()`	Сокращение**.**
`reduceat()`	Сокращение в заданных точках**.**

Пример с функцией `add()` позволяет понять смысл универсальной функции и ее методов**:**

```
>>> from Numeric import add
>>> add([[1, 2], [3, 4]],  [[1, 0], [0, 1]])
array([[2, 2],
       [3, 5]])
>>> add([[1, 2], [3, 4]],  [1, 0])
array([[2, 2],
       [4, 4]])
>>> add([[1, 2], [3, 4]],  1)
array([[2, 3],
       [4, 5]])
>>> add.reduce([1, 2, 3, 4])                    # т.е. 1+2+3+4
10
>>> add.reduce([[1, 2], [3, 4]], 0)             # т.е. [1+3 2+4]
array([4, 6])
>>> add.reduce([[1, 2], [3, 4]], 1)             # т.е. [1+2 3+4]
array([3, 7])
>>> add.accumulate([1, 2, 3, 4])                # т.е. [1 1+2 1+2+3 1+2+3+4]
array([ 1,  3,  6, 10])
>>> add.reduceat(range(10), [0, 3, 6])          # т.е. [0+1+2 3+4+5 6+7+8+9]
```

```
array([ 3, 12, 30])
>>> add.outer([1,2], [3,4])                    # т.е. [[1+3 1+4] [2+3 2+4]]
array([[4, 5],
       [5, 6]])
```

Методы `accumulate()`, `reduce()` и `reduceat()` принимают необязательный аргумент - номер размерности, используемой для соответствующего действия. По умолчанию применяется нулевая размерность.

Универсальные функции, помимо одного или двух необходимых параметров, позволяют задавать и еще один аргумент, для приема результата функции. Тип третьего аргумента должен строго соответствовать типу результата. Например, функция `sqrt()` даже от целых чисел имеет тип `Float`.

```
>>> from Numeric import array, sqrt, Float
>>> a = array([0, 1, 2])
>>> r = array([0, 0, 0], Float)
>>> sqrt(a, r)
array([ 0.        ,  1.        ,  1.41421356])
>>> print r
[ 0.          1.          1.41421356]
```

Предупреждение:

Не следует использовать в качестве приемника результата массив, который фигурирует в предыдущих аргументах функции, так как при этом результат может быть испорчен. Следующий пример показывает именно такой вариант:

```
>>> import Numeric
>>> m = Numeric.array([0, 0, 0, 1, 0, 0, 0, 0])
>>> add(m[:-1], m[1:], m[1:])
array([0, 0, 1, 1, 1, 1, 1])
```

В таких неоднозначных случаях необходимо использовать промежуточный массив.

Функции модуля Numeric

Следующие функции модуля `Numeric` являются краткой записью некоторых наиболее употребительных сочетаний функций и методов:

Функция	Аналог функции
`sum(a, axis)`	`add.reduce(a, axis)`
`cumsum(a, axis)`	`add.accumulate(a, axis)`
`product(a, axis)`	`multiply.reduce(a, axis)`
`cumproduct(a, axis)`	`multiply.accumulate(a, axis)`
`alltrue(a, axis)`	`logical_and.reduce(a, axis)`
`sometrue(a, axis)`	`logical_or.reduce(a, axis)`

Примечание:

Параметр `axis` указывает размерность.

Функции для работы с массивами

Функций достаточно много, поэтому подробно будут рассмотрены только две из них, а остальные сведены в таблицу.

Функция Numeric.take()

Функция `Numeric.take()` позволяет взять часть массива по заданным на определенном измерении индексам. По умолчанию номер измерения (третий аргумент) равен нулю.

```
>>> import Numeric
>>> a = Numeric.reshape(Numeric.arrayrange(25), (5, 5))
>>> print a
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]
 [15 16 17 18 19]
 [20 21 22 23 24]]
>>> print Numeric.take(a, [1], 0)
[ [5 6 7 8 9]]
>>> print Numeric.take(a, [1], 1)
[[ 1]
 [ 6]
 [11]
 [16]
 [21]]
>>> print Numeric.take(a, [[1,2],[3,4]])
[[[ 5  6  7  8  9]
  [10 11 12 13 14]]
 [[15 16 17 18 19]
  [20 21 22 23 24]]]
```

В отличие от среза, функция `Numeric.take()` сохраняет размерность массива, если конечно, структура заданных индексов одномерна. Результат `Numeric.take(a, [[1,2],[3,4]])` показывает, что взятые по индексам части помещаются в массив со структурой самих индексов, как если бы вместо `1` было написано `[5 6 7 8 9]`, а вместо `2` - `[10 11 12 13 14]` и т.д.

Функции Numeric.diagonal() и Numeric.trace()

Функция `Numeric.diagonal()` возвращает диагональ матрицы. Она имеет следующие аргументы:

`a`	Исходный массив.
`offset`	Смещение вправо от "главной" диагонали (по умолчанию **0**).
`axis1`	Первое из измерений, на которых берется диагональ (по умолчанию **0**).
`axis2`	Второе измерение, образующее вместе с первым плоскость, на которой и берется диагональ. По умолчанию `axis2=1`.

Функция `Numeric.trace()` (для вычисления `следа матрицы`) имеет те же аргументы, но суммирует элементы на диагонали. В примере ниже рассмотрены обе эти функции:

```
>>> import Numeric
>>> a = Numeric.reshape(Numeric.arrayrange(16), (4, 4))
>>> print a
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
>>> for i in range(-3, 4):
...   print "Sum", Numeric.diagonal(a, i), "=", Numeric.trace(a, i)
...
Sum [12] = 12
Sum [ 8 13] = 21
Sum [ 4  9 14] = 27
Sum [ 0  5 10 15] = 30
Sum [ 1  6 11] = 18
```

```
Sum [2 7] = 9
Sum [3] = 3
```

Функция Numeric.choose()

Эта функция использует один массив с целыми числами от `0` до `n` для выбора значения из одного из заданных массивов:

```
>>> a = Numeric.identity(4)
>>> b0 = Numeric.reshape(Numeric.arrayrange(16), (4, 4))
>>> b1 = -Numeric.reshape(Numeric.arrayrange(16), (4, 4))
>>> print Numeric.choose(a, (b0, b1))
[[  0   1   2   3]
 [  4  -5   6   7]
 [  8   9 -10  11]
 [ 12  13  14 -15]]
```

Свод функций модуля Numeric

Следующая таблица приводит описания функций модуля **Numeric.**

Функция и ее аргументы	Назначение функции
`allclose(a, b[, eps[, A]])`	Сравнение `a` и `b` с заданными относительными `eps` и абсолютными `A` погрешностями. По умолчанию `eps` равен `1.0e-1`, а `A = 1.0e-8`.
`alltrue(a[, axis])`	Логическое И по всей оси `axis` массива `a`
`argmax(a[, axis])`	Индекс максимального значения в массиве по заданному измерению `axis`
`argmin(a[, axis])`	Индекс минимального значения в массиве по заданному измерению `axis`
`argsort(a[, axis])`	Индексы отсортированного массива, такие, что `take(a,argsort(a, axis),axis)` дает отсортированный массив `a`, как если бы было выполнено `sort(a, axis)`
`array(a[, type])`	Создание массива на основе последовательности `a` данного типа `type`
`arrayrange(start[, stop[, step[, type]]])`	Аналог `range()` для массивов
`asarray(a[, type[, savespace]])`	То же, что и `array()`, но не создает новый массив, если `a` уже является массивом.
`choose(a, (b0,...,bn))`	Создает массив на основе элементов, взятых по индексам из `a` (индексы от `0` до `n` включительно). Формы массивов `a`, `b1`, `...`, `bn` должны совпадать
`clip(a, a_min, a_max)`	Обрубает значения массива `a` так, чтобы они находились между значениями из `a_min` и `a_max` поэлементно
`compress(cond, a[, axis])`	Возвращает массив только из тех элементов массива `a`, для которых условие `cond` истинно (не нуль)
`concatenate(a[, axis])`	Соединение двух массивов (конкатенация) по заданному измерению `axis` (по умолчанию - по нулевой)
`convolve(a, b[, mode])`	Свертка двух массивов. Аргумент `mode` может принимать значения `0`, `1` или `2`
`cross_correlate(a, b[, mode])`	Взаимная корреляция двух массивов. Параметр `mode` может принимать значения `0`, `1` или `2`

`cumproduct(a[, axis])`	Произведение по измерению `axis` массива `a` с промежуточными результатами
`cumsum(a[, axis])`	Суммирование с промежуточными результатами
`diagonal(a[, k[, axis1[, axis2]]])`	Взятие `k`-й диагонали массива `a` в плоскости измерений `axis1` и `axis2`
`dot(a, b)`	Внутреннее (матричное) произведение массивов. По определению: `innerproduct(a, swapaxes(b, -1, -2))`, т.е. с переставленными последними измерениями, как и должно быть при перемножении матриц
`dump(obj, file)`	Запись массива `a` (в двоичном виде) в открытый файловый объект `file`. Файл должен быть открыт в бинарном режиме. В файл можно записать несколько объектов подряд
`dumps(obj)`	Строка с двоичным представлением объекта `obj`
`fromfunction(f, dims)`	Строит массив, получая информацию от функции `f()`, в качестве аргументов которой выступают значения кортежа индексов. Фактически является сокращением для `f(*tuple(indices(dims)))`
`fromstring(s[, count[, type]])`	Создание массива на основе бинарных данных, хранящихся в строке
`identity(n)`	Возвращает двумерный массив формы `(n, n)`
`indices(dims[, type])`	Возвращает массив индексов заданной длины по каждому измерению с изменением поочередно по каждому изменению. Например, `indices([2, 2])[1]` дает двумерный массив `[[0, 1], [0, 1]]`.
`innerproduct(a, b)`	Внутреннее произведение двух массивов (по общему измерению). Для успешной операции `a.shape[-1]` должен быть равен `b.shape[-1]`. Форма результата будет `a.shape[:-1] + b.shape[:-1]`. Элементы пропадающего измерения попарно умножаются и получающиеся произведения суммируются
`load(file)`	Чтение массива из файла `file`. Файл должен быть открыт в бинарном режиме
`loads(s)`	Возвращает объект, соответствующий бинарному представлению, заданному в строке
`nonzero(a)`	Возвращает индексы ненулевых элементов одномерного массива
`ones(shape[, type])`	Массив из единиц заданной формы `shape` и обозначения типа `type`
`outerproduct(a, b)`	Внешнее произведение `a` и `b`
`product(a[, axis])`	Произведение по измерению `axis` массива `a`
`put(a, indices, b)`	Присваивание частям массива, `a[n] = b[n]` для всех индексов `indices`
`putmask(a, mask, b)`	Присваивание `a` элементов из `b`, для которых маска `mask` имеет значение истина
`ravel(a)`	Превращение массива в одномерный. Аналогично `reshape(a, (-1,))`
`repeat(a, n[, axis])`	Повторяет элементы массива `a` `n` раз по измерению `axis`
`reshape(a, shape)`	Возвращает массив нужной формы (нового массива не создает). Количество элементов в исходном и новом массивах должно совпадать
`resize(a, shape)`	Возвращает массив с произвольной новой формой `shape`. Размер исходного массива не важен
`searchsorted(a, i)`	Для каждого элемента из `i` найти место в массиве `a`. Массив `a`

	должен быть одномерным и отсортированным. Результат имеет форму массива `i`
`shape(a)`	Возвращает форму массива `a`
`sometrue(a[, axis])`	Логическое ИЛИ по всему измерению `axis` массива `a`
`sort(a[, axis])`	Сортировка элементов массива по заданному измерению
`sum(a[, axis])`	Суммирование по измерению `axis` массива `a`
`swapaxes(a, axis1, axis1)`	Смена измерений (частный случай транспонирования)
`take(a, indices[, axis])`	Выбор частей массива `a` на основе индексов `indices` по измерению `axis`
`trace(a[, k[, axis1[, axis2]]])`	Сумма элементов вдоль диагонали, то есть `add.reduce(diagonal(a, k, axis1, axis2))`
`transpose(a[, axes])`	Перестановка измерений в соответствии с `axes`, либо, если `axes` не заданы - расположение их в обратном порядке
`where(cond, a1, a2)`	Выбор элементов на основании условия `cond` из `a1` (если не нуль) и `a2` (при нуле) поэлементно. Равносилен `choose(not_equal(cond, 0), (y, x))`. Формы массивов-аргументов `a1` и `a2` должны совпадать
`zeros(shape[, type])`	Массив из нулей заданной формы `shape` и обозначения типа `type`

В этой таблице в качестве обозначения типа `type` можно указывать рассмотренные выше константы: `Int`, `Float` и т.п.

Модуль `Numeric` также определяет константы `e` (число **e**) и `pi` (число пи).

Модуль LinearAlgebra

Модуль `LinearAlgebra` содержит алгоритмы линейной алгебры, в частности нахождение определителя матрицы, решений системы линейных уравнений, обращение матрицы, нахождение собственных чисел и собственных векторов матрицы, разложение матрицы на множители: Холецкого, сингулярное, метод наименьших квадратов.

Функция `LinearAlgebra.determinant()` находит определитель матрицы:

```
>>> import Numeric, LinearAlgebra
>>> print LinearAlgebra.determinant(
...     Numeric.array([[1, -2],
...                    [1, 5]]))
7
```

Функция `LinearAlgebra.solve_linear_equations()` решает линейные уравнения вида `ax=b` по заданным аргументам `a` и `b`:

```
>>> import Numeric, LinearAlgebra
>>> a = Numeric.array([[1.0, 2.0],  [0.0, 1.0]])
>>> b = Numeric.array([1.2, 1.5])
>>> x = LinearAlgebra.solve_linear_equations(a, b)
>>> print "x =", x
x = [-1.8  1.5]
>>> print "Проверка:", Numeric.dot(a, x) - b
Проверка: [ 0.  0.]
```

Когда матрица `a` имеет нулевой определитель, система имеет не единственное решение и возбуждается исключение `LinearAlgebraError`:

```
>>> a = Numeric.array([[1.0, 2.0],  [0.5, 1.0]])
>>> x = LinearAlgebra.solve_linear_equations(a, b)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "/usr/local/lib/python2.3/site-packages/Numeric/LinearAlgebra.py", line
98,
    in solve_linear_equations raise LinAlgError, 'Singular matrix'
LinearAlgebra.LinAlgError: Singular matrix
```

Функция `LinearAlgebra.inverse()` находит обратную матрицу. Однако не следует решать линейные уравнения с помощью `LinearAlgebra.inverse()` умножением на обратную матрицу, так как она определена через `LinearAlgebra.solve_linear_equations()`:

```
def inverse(a):
    return solve_linear_equations(a, Numeric.identity(a.shape[0]))
```

Функция `LinearAlgebra.eigenvalues()` находит собственные значения матрицы, а `LinearAlgebra.eigenvectors()` - пару: собственные значения, собственные вектора:

```
>>> from Numeric import array, dot
>>> from LinearAlgebra import eigenvalues, eigenvectors
>>> a = array([[-5, 2],  [2, -7]])
>>> lmd = eigenvalues(a)
>>> print "Собственные значения:", lmd
Собственные значения: [-3.76393202 -8.23606798]
>>> (lmd, v) = eigenvectors(a)
>>> print "Собственные вектора:"
Собственные вектора:
>>> print v
[[ 0.85065081  0.52573111]
 [-0.52573111  0.85065081]]
>>> print "Проверка:", dot(a, v[0]) - v[0] * lmd[0]
Проверка: [ -4.44089210e-16   2.22044605e-16]
```

Проверка показывает, что тождество выполняется с достаточно большой точностью (числа совсем маленькие, практически нули): собственные числа и векторы найдены верно.

Модуль RandomArray

В этом модуле собраны функции для генерации массивов случайных чисел различных распределений и свойств. Их можно применять для математического моделирования.

Функция `RandomArray.random()` создает массивы из псевдослучайных чисел, равномерно распределенных в интервале `(0, 1)`:

```
>>> import RandomArray
>>> print RandomArray.random(10)  # массив из 10 псевдослучайных чисел
[ 0.28374212  0.19260929  0.07045474  0.30547682  0.10842083  0.14049676
       0.01347435  0.37043894  0.47362471  0.37673479]
>>> print RandomArray.random([3,3])  # массив 3x3 из псевдослучайных чисел
[[ 0.53493741  0.44636754  0.20466961]
 [ 0.8911635   0.03570878  0.00965272]
 [ 0.78490953  0.20674807  0.23657821]]
```

Функция `RandomArray.randint()` для получения массива равномерно распределенных чисел из заданного интервала и заданной формы:

```
>>> print RandomArray.randint(1, 10, [10])
[8 1 9 9 7 5 2 5 3 2]
>>> print RandomArray.randint(1, 10, [10])
[2 2 5 5 7 7 3 4 3 7]
```

Можно получать и случайные перестановки с помощью `RandomArray.permutation()`:

```
>>> print RandomArray.permutation(6)
[4 0 1 3 2 5]
>>> print RandomArray.permutation(6)
[1 2 0 3 5 4]
```

Доступны и другие распределения для получения массива нормально распределенных величин с заданным средним и стандартным отклонением:

```
>>> print RandomArray.normal(0, 1, 30)
[-1.0944078   1.24862444  0.20415567 -0.74283403  0.72461408 -0.57834256
 0.30957144  0.8682853   1.10942173 -0.39661118  1.33383882  1.54818618
 0.18814971  0.89728773 -0.86146659  0.0184834  -1.46222591 -0.78427434
 1.09295738 -1.09731364  1.34913492 -0.75001568 -0.11239344  2.73692131
 -0.19881676 -0.49245331  1.54091263 -1.81212211  0.46522358 -0.08338884]
```

Следующая таблица приводит функции для других распределений:

Функция и ее аргументы	Описание
`F(dfn, dfd, shape=[])`	**F-**распределение
`beta(a, b, shape=[])`	Бета-распределение
`binomial(trials, p, shape=[])`	Биномиальное распределение
`chi_square(df, shape=[])`	Распределение хи-квадрат
`exponential(mean, shape=[])`	Экспоненциальное распределение
`gamma(a, r, shape=[])`	Гамма-распределение
`multivariate_normal(mean, cov, shape=[])`	Многомерное нормальное распределение
`negative_binomial(trials, p, shape=[])`	Негативное биномиальное
`noncentral_F(dfn, dfd, nconc, shape=[])`	Нецентральное **F-**распределение
`noncentral_chi_square(df, nconc, shape=[])`	Нецентральное хи-квадрат распределение
`normal(mean, std, shape=[])`	Нормальное распределение
`permutation(n)`	Случайная перестановка
`poisson(mean, shape=[])`	Пуассоновское распределение
`randint(min, max=None, shape=[])`	Случайное целое
`random(shape=[])`	Равномерное распределение на интервале **(0, 1)**
`random_integers(max, min=1, shape=[])`	Случайное целое
`standard_normal(shape=[])`	Стандартное нормальное распределение
`uniform(min, max, shape=[])`	Равномерное распределение

Заключение

В этой лекции рассматривался набор модулей для численных вычислений. Модуль `Numeric` определяет тип многомерный массив и множество функций для работы с массивами. Также были представлены модули для линейной алгебры и моделирования последовательностей случайных чисел различных распределений.

Ссылки

Сайт, посвященный **Numeric Python**: http://www.pfdubois.com/numpy/

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** (сайт:**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
`alnum()`	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 их 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:

```
vcodecs.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.**

Ссылки

NLTK

http://nltk.sourceforge.net

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), (родитель), `nextSibling` (следующий брат), `previousSibling` (предыдущий брат).`parentNode`

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 Locator**, 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.

Лекция #8: Разработка Web-приложений

Под **web-приложением** будет пониматься программа, основной интерфейс пользователя которой работает в стандартном **WWW**-браузере под управлением **HTML** и **XML**-документов. Для улучшение качества интерфейса пользователя часто применяют **JavaScript**, однако это несколько снижает универсальность интерфейса. Следует заметить, что интерфейс можно построить на **Java**- или **Flash**-апплетах, однако, такие приложения сложно назвать **web**-приложениями, так как **Java** или **Flash** могут использовать собственные протоколы для общения с сервером, а не стандартный для **WWW** протокол **HTTP**.

При создании **web**-приложений стараются отделить Форму (внешний вид, стиль), Содержание и Логику обработки данных. Современные технологии построения **web**-сайтов дают возможность подойти достаточно близко к этому идеалу. Тем не менее, даже без применения многоуровневых приложений можно придерживаться стиля, позволяющего изменять любой из этих аспектов, не затрагивая (или почти не затрагивая) двух других. Рассуждения на эту тему будут продолжены в разделе, посвященном средам разработки.

CGI-сценарии

Классический путь создания приложений для **WWW** - написание **CGI**-сценариев (иногда говорят - скриптов). **CGI** (**Common Gateway Interface**, общий шлюзовой интерфейс) - это стандарт, регламентирующий взаимодействие сервера с внешними приложениями. В случае с **WWW**, **web**-сервер может направить запрос на генерацию страницы по определенному сценарию. Этот сценарий, получив на вход данные от **web**-сервера (тот, в свою очередь, мог получить их от пользователя), генерирует готовый объект (изображение, аудиоданные, таблицу стилей и т.п.).

При вызове сценария **Web**-сервер передает ему информацию через стандартный ввод, переменные окружения и, для **ISINDEX**, через аргументы командной строки (они доступны через `sys.argv`).

Два основных метода передачи данных из заполненной в браузере формы **Web**-серверу (и **CGI**-сценарию) - `GET` и `POST`. В зависимости от метода данные передаются по-разному. В первом случае они кодируются и помещаются прямо в **URL**, например: `http://host/cgi-bin/a.cgi?a=1&b=3`. Сценарий получает их в переменной окружения с именем `QUERY_STRING`. В случае метода `POST` они передаются на стандартный ввод.

Для корректной работы сценарии помещаются в предназначенный для этого каталог на **web**-сервере (обычно он называется `cgi-bin`) или, если это разрешено конфигурацией сервера, в любом месте среди документов **HTML**. Сценарий должен иметь признак исполняемости. В системе **Unix** его можно установить с помощью команды `chmod a+x`.

Следующий простейший сценарий выводит значения из словаря `os.environ` и позволяет увидеть, что же было ему передано:

```
#!/usr/bin/python

import os
print """Content-Type: text/plain

%s""" % os.environ
```

С помощью него можно увидеть установленные **Web**-сервером переменные окружения. Выдаваемый **CGI**-сценарием **web**-серверу файл содержит заголовочную часть, в которой указаны поля с мета-информацией (тип содержимого, время последнего обновления документа, кодировка и т.п.).

Основные переменные окружения, достаточные для создания сценариев:

`QUERY_STRING`

Строка запроса.

`REMOTE_ADDR`

IP-адрес клиента.

`REMOTE_USER`

Имя клиента **(**если он был идентифицирован**).**

`SCRIPT_NAME`

Имя сценария.

`SCRIPT_FILENAME`

Имя файла со сценарием.

`SERVER_NAME`

Имя сервера.

`HTTP_USER_AGENT`

Название броузера клиента.

`REQUEST_URI`

Строка запроса **(URI).**

`HTTP_USER_AGENT`

Имя сервера.

`HTTP_ACCEPT_LANGUAGE`

Желательный язык документа.

Вот что может содержать словарь `os.environ` в **CGI-**сценарии**:**

```
{
'DOCUMENT_ROOT': '/var/www/html',
'SERVER_ADDR': '127.0.0.1',
'SERVER_PORT': '80',
'GATEWAY_INTERFACE': 'CGI/1.1',
'HTTP_ACCEPT_LANGUAGE': 'en-us, en;q=0.50',
'REMOTE_ADDR': '127.0.0.1',
'SERVER_NAME': 'rnd.onego.ru',
'HTTP_CONNECTION': 'close',
'HTTP_USER_AGENT': 'Mozilla/5.0 (X11; U; Linux i586; en-US;
rv:1.0.1) Gecko/20021003',
'HTTP_ACCEPT_CHARSET': 'ISO-8859-1, utf-8;q=0.66, *;q=0.66',
'HTTP_ACCEPT': 'text/xml,application/xml,application/xhtml+xml,
text/html;q=0.9,text/plain;q=0.8,video/x-mng,image/png,image/jpeg,
image/gif;q=0.2,text/css,*/*;q=0.1',
'REQUEST_URI': '/cgi-bin/test.py?a=1',
'PATH': '/sbin:/usr/sbin:/bin:/usr/bin:/usr/X11R6/bin',
'QUERY_STRING': 'a=1&b=2',
```

```
'SCRIPT_FILENAME': '/var/www/cgi-bin/test.py',
'HTTP_KEEP_ALIVE': '300',
'HTTP_HOST': 'localhost',
'REQUEST_METHOD': 'GET',
'SERVER_SIGNATURE': 'Apache/1.3.23 Server at rnd.onego.ru Port 80',
'SCRIPT_NAME': '/cgi-bin/test.py',
'SERVER_ADMIN': 'root@localhost',
'SERVER_SOFTWARE': 'Apache/1.3.23 (Unix)  (Red-Hat/Linux)
mod_python/2.7.8 Python/1.5.2 PHP/4.1.2',
'SERVER_PROTOCOL': 'HTTP/1.0',
'REMOTE_PORT': '39251'
}
```

Следующий **CGI-**сценарий выдает черный квадрат **(**в нем используется модуль `Image` для обработки изображений**):**

```
#!/usr/bin/python

import sys
print """Content-Type: image/jpeg
"""

import Image
i = Image.new("RGB", (10,10))
i.im.draw_rectangle((0,0,10,10), 1)
i.save(sys.stdout, "jpeg")
```

Модуль cgi

В **Python** имеется поддержка **CGI** в виде модуля `cgi`**.** Следующий пример показывает некоторые из его возможностей**:**

```
#!/usr/bin/python
# -*- coding: cp1251 -*-
import cgi, os

# анализ запроса
f = cgi.FieldStorage()
if f.has_key("a"):
  a = f["a"].value
else:
  a = "0"

# обработка запроса
b = str(int(a)+1)
mytext = open(os.environ["SCRIPT_FILENAME"]).read()
mytext_html = cgi.escape(mytext)

# формирование ответа
print """Content-Type: text/html

<html><head><title>Решение примера: %(b)s = %(a)s + 1</title></head>
<body>
%(b)s
<table width="80%%"><tr><td>
<form action="me.cgi" method="GET">
<input type="text" name="a" value="0" size="6">
<input type="submit" name="b" value="Обработать">
</form></td></tr></table>
<pre>
%(mytext_html)s
</pre>
</body></html>""" % vars()
```

В этом примере к заданному в форме числу прибавляется **1.** Кроме того, выводится исходный код самого сценария. Следует заметить, что для экранирования символов `>`, `<`, `&` использована функция `cgi.escape()`. Для формирования **Web-**страницы применена операция форматирования. В качестве словаря для выполнения подстановок использован словарь `vars()` со всеми локальными переменными. Знаки процента пришлось удвоить, чтобы они не интерпретировались командой форматирования. Стоит обратить внимание на то, как получено значение от пользователя. Объект `FieldStorage` **"**почти**"** словарь, с тем исключением, что для получения обычного значения нужно дополнительно посмотреть атрибут `value`. Дело в том, что в сценарий могут передаваться не только текстовые значения, но и файлы, а также множественные значения с одним и тем же именем.

Осторожно**!**

При обработке входных значений **CGI-**сценариев нужно внимательно и скрупулезно проверять допустимые значения. Лучше считать, что клиент может передать на вход все, что угодно. Из этого всего необходимо выбрать и проверить только то, что ожидает сценарий.

Например, не следует подставлять полученные от пользователя данные в путь к файлу, в качестве аргументов к функции `eval()` и ей подобных; параметров командной строки; частей в **SQL-**запросах к базе данных. Также не стоит вставлять полученные данные напрямую в формируемые страницы, если эти страницы будет видеть не только клиент, заказавший **URL (**например, такая ситуация обычна в **web-**чатах, форумах, гостевых книгах**),** и даже в том случае, если единственный читатель этой информации **-** администратор сайта. Тот, кто смотрит страницы с непроверенным **HTML-**кодом, поступившим напрямую от пользователя, рискуют обработать в своем браузере зловредный код, использующий брешь в его защите.

Даже если **CGI-**сценарий используется исключительно другими сценариями через запрос на **URL**, нужно проверять входные значения столь же тщательно, как если бы данные вводил пользователь. **(**Так как недоброжелатель может подать на **web-**сервер любые значения**).**

В примере выше проверка на допустимость произведена при вызове функции `int()`: если было бы задано нечисловое значение, сценарий аварийно завершился, а пользователь увидел `Internal Server Error`.

После анализа входных данных можно выделить фазу их обработки. В этой части **CGI-**сценария вычисляются переменные для дальнейшего вывода. Здесь необходимо учитывать не только значения переданных переменных, но и факт их присутствия или отсутствия, так как это тоже может влиять на логику сценария.

И, наконец, фаза вывода готового объекта **(**текста, **HTML-**документа, изображения, мультимедиа-объекта и т.п.**).** Проще всего заранее подготовить шаблон страницы **(**или ее крупных частей**),** а потом просто заполнить содержимым из переменных.

В приведенных примерах имена появлялись в строке запроса только один раз. Некоторые формы порождают несколько значений для одного имени. Получить все значения можно с помощью метода `getlist()`:

```
lst = form.getlist("fld")
```

Список `lst` будет содержать столько значений, сколько полей с именем `fld` получено из **web-**формы **(**он может быть и пустым, если ни одно поле с заданным именем не было заполнено**).**

В некоторых случаях необходимо передать на сервер файлы **(**сделать **upload).** Следующий пример и комментарий к нему помогут разобраться с этой задачей**:**

```
#!/usr/bin/env python
```

```
import cgi

form = cgi.FieldStorage()
file_contents = ""
if form.has_key("filename"):
  fileitem = form["filename"]
  if fileitem.file:
    file_contents = """<P>Содержимое переданного файла:
    <PRE>%s</PRE>""" % fileitem.file.read()

print """Content-Type: text/html

<HTML><HEAD><TITLE>Загрузка файла</TITLE></HEAD>
<BODY><H1>Загрузка файла</H1>
<P><FORM ENCTYPE="multipart/form-data"
ACTION="getfile.cgi" METHOD="POST">
<br>Файл: <INPUT TYPE="file" NAME="filename">
<br><INPUT TYPE="submit" NAME="button" VALUE="Передать файл">
</FORM>
%s
</BODY></HTML>""" % file_contents
```

В начале следует рассмотреть **web-**форму, которая приведена в конце сценария: именно она будет выводиться пользователю при обращении по **CGI-**сценарию. Форма имеет поле типа `file`, которое в **web-**броузере представляется полоской ввода и кнопкой **"Browse".** Нажимая на кнопку **"Browse",** пользователь выбирает файл, доступный в ОС на его компьютере. После этого он может нажать кнопку "Передать файл" для передачи файла на сервер.

Для отладки **CGI-**сценария можно использовать модуль `cgitb`. При возникновении ошибки этот модуль выдаст красочную **HTML-**страницу с указанием места возбуждения исключения. В начале отлаживаемого сценария нужно поставить

```
import cgitb
cgitb.enable(1)
```

Или, если не нужно показывать ошибки в браузере:

```
  import cgitb
  cgitb.enable(0, logdir="/tmp")
```

Только необходимо помнить, что следует убрать эти строки, когда сценарий будет отлажен, так как он выдает кусочки кода сценария. Это может быть использовано злоумышленниками, с тем чтобы найти уязвимости в **CGI-**сценарии или подсмотреть пароли **(**если таковые присутствуют в сценарии**).**

Что после CGI?

К сожалению, строительство интерактивного и посещаемого сайта на основе **CGI** имеет свои ограничения, главным образом, связанные с производительностью. Ведь для каждого запроса нужно вызвать как минимум один сценарий **(**а значит **-** запустить интерпретатор **Python),** из него, возможно, сделать соединение с базой данных и т.д. Время запуска интерпретатора **Python** достаточно невелико, тем не менее, на занятом сервере оно может оказывать сильное влияние на загрузку процессора.

Желательно, чтобы интерпретатор уже находился в оперативной памяти, и были доступны соединения с базой данных.

Такие технологии существуют и обычно опираются на модули, встраиваемые в **web-**сервер.

Для ускорения работы **CGI** используются различные схемы, например, **FastCGI** или **PCGI (Persistent CGI)**. В данной лекции предлагается к рассмотрению специальным модуль для **web**-сервера **Apache**, называемый `mod_python`.

Пусть модуль установлен на **web**-сервере в соответствии с инструкциями, данными в его документации.

Модуль `mod_python` позволяет сценарию-обработчику вклиниваться в процесс обработки **HTTP**-запроса сервером **Apache** на любом этапе, для чего сценарий должен иметь определенным образом названные функции.

Сначала нужно выделить каталог, в котором будет работать сценарий-обработчик. Пусть это каталог `/var/www/html/mywebdir`. Для того чтобы **web**-сервер знал, что в этом каталоге необходимо применять `mod_python`, следует добавить в файл конфигурации **Apache** следующие строки:

```
<Directory "/var/www/html/mywebdir>
  AddHandler python-program .py
  PythonHandler mprocess
</Directory>
```

После этого необходимо перезапустить **web**-сервер и, если все прошло без ошибок, можно приступать к написанию обработчика `mprocess.py`. Этот сценарий будет реагировать на любой запрос вида **http://localhost/*.py**.

Следующий сценарий `mprocess.py` выведет в браузере страницу со словами `Hello, world!`:

```
from mod_python import apache

def handler(req):
  req.content_type = "text/html"
  req.send_http_header()
  req.write("""<HTML><HEAD><TITLE>Hello, world!</TITLE></HEAD>
  <BODY>Hello, world!</BODY></HTML>""")
  return apache.OK
```

Отличия сценария-обработчика от **CGI**-сценария:

1. Сценарий-обработчик не запускается при каждом **HTTP**-запросе: он уже находится в памяти, и из него вызываются необходимые функции-обработчики (в приведенном примере такая функция всего одна - `handler()`). Каждый процесс-потомок **web**-сервера может иметь свою копию сценария и интерпретатора **Python**.
2. Как следствие п.**1** различные **HTTP**-запросы делят одни и те же глобальные переменные. Например, таким образом можно инициализировать соединение с базой данных и применять его во всех запросах (хотя в некоторых случаях потребуются блокировки, исключающие одновременное использование соединения разными потоками (нитями) управления).
3. Обработчик задействуется при обращении к любому "файлу" с расширением `py`, тогда как **CGI**-сценарий обычно запускается при обращении по конкретному имени.
4. В сценарии-обработчике нельзя рассчитывать на то, что он увидит модули, расположенные в том же каталоге. Возможно, придется добавить некоторые каталоги в `sys.path`.
5. Текущий рабочий каталог (его можно узнать с помощью функции `os.getcwd()`) также не находится в одном каталоге с обработчиком.
6. **#!**-строка в первой строке сценария не определяет версию интерпретатора **Python**. Работает версия, для которой был скомпилирован `mod_python`.
7. Все необходимые параметры передаются в обработчик в виде **Request**-объекта. Возвращаемые значения также передаются через этот объект.

8. **Web**-сервер замечает, что сценарий-обработчик изменился, но не заметит изменений в импортируемых в него модулях. Команда `touch mprocess.py` обновит дату изменения файла сценария.
9. Отображение `os.environ` в обработчике может быть обрезанным. Кроме того, вызываемые из сценария-обработчика другие программы его не наследуют, как это происходит при работе с **CGI**-сценариями. Переменные можно получить другим путем: `req.add_common_vars(); params = req.subprocess_env.`
10. Так как сценарий-обработчик не является "одноразовым", как **CGI**-сценарий, из-за ошибок программирования (как самого сценария, так и других компонентов) могут возникать утечки памяти (программа не освобождает ставшую ненужной память). Следует установить значение параметра `MaxRequestsPerChild` (максимальное число запросов, обрабатываемое одним процессом-потомком) больше нуля.

Другой возможный обработчик - сценарий идентификации:

```
def authenhandler(req):
  password = req.get_basic_auth_pw()
  user = req.connection.user
  if user == "user1" and password == "secret":
      return apache.OK
  else:
      return apache.HTTP_UNAUTHORIZED
```

Эту функцию следует добавить в модуль `mprocess.py`, который был рассмотрен ранее. Кроме того, нужно дополнить конфигурацию, назначив обработчик для запросов идентификации (`PythonAuthenHandler`), а также обычные для **Apache** директивы `AuthType`, `AuthName`, `require`, определяющие способ авторизации:

```
<Directory "/var/www/html/mywebdir>
  AddHandler python-program .py
  PythonHandler mprocess
  PythonAuthenHandler mprocess
  AuthType Basic
  AuthName "My page"
  require valid-user
</Directory>
```

Разумеется, это - всего лишь пример. В реальности идентификация может быть устроена намного сложнее.

Другие возможные обработчики (по документации к `mod_python` можно уточнить, в какие моменты обработки запроса они вызываются):

```
PythonPostReadRequestHandler
```

Обработка полученного запроса сразу после его получения.

```
PythonTransHandler
```

Позволяет изменить **URI** запроса (в том числе имя виртуального сервера).

```
PythonHeaderParserHandler
```

Обработка полей запроса.

```
PythonAccessHandler
```

Обработка ограничений доступа (например, по **IP**-адресу).

```
PythonAuthenHandler
```

Идентификация пользователя.

```
PythonTypeHandler
```

Определение и/или настройка типа документа, языка и т.д.

```
PythonFixupHandler
```

Изменение полей непосредственно перед вызовом обработчиков содержимого.

```
PythonHandler
```

Основной обработчик запроса.

```
PythonInitHandler
```

`PythonPostReadRequestHandler` или `PythonHeaderParserHandler` в зависимости от нахождения в конфигурации **web**-сервера.

```
PythonLogHandler
```

Управление записью в логи.

```
PythonCleanupHandler
```

Обработчик, вызываемый непосредственно перед уничтожением **Request**-объекта.

Некоторые из этих обработчиков работают только глобально, так как при вызове даже каталог их приложения может быть неизвестен (таков, например, `PythonPostReadRequestHandler`).

С помощью `mod_python` можно строить **web**-сайты с динамическим содержимым и контролировать некоторые аспекты работы **web**-сервера **Apache** через **Python**-сценарии.

Среды разработки

Для создания **Web**-приложений применяются и более сложные средства, чем **web**-сервер с расположенными на нем статическими документами и **CGI**-сценариями. В зависимости от назначения такие программные системы называются серверами **web**-приложений, системами управления содержимым (**CMS**, **Content Management System**), системы **web**-публикации и средствами для создания **WWW**-порталов. Причем **CMS**-система может быть выполнена как **web**-приложение, а средства для создания порталов могут базироваться на системах **web**-публикации, для которых **CMS**-система является подсистемой. Поэтому, выбирая систему для конкретных нужд, стоит уточнить, какие функции она должна выполнять.

Язык **Python**, хотя и уступает **PHP** по количеству созданных на нем **web**-систем, имеет несколько достаточно популярных приложений. Самым ярким примером средства для создания сервера **web**-приложений является **Zope** (произносится "зоп") (см. **http://zope.org**) (**Z Object Publishing Environment**, **среда публикации объектов**). **Zope** имеет встроенный **web**-сервер, но может работать и с другими **Web**-серверами, например, **Apache**. На основе **Zope** можно строить **web**-порталы, например, с помощью **Plone**/**Zope**, но можно разрабатывать и собственные **web**-приложения. При этом **Zope** позволяет разделить Форму, Содержание и Логику до такой степени, что Содержанием могут заниматься одни люди (менеджеры по содержимому), Формой - другие (**web**-дизайнеры), а Логикой - третьи (программисты). В случае с **Zope** Логику можно задать с помощью языка **Python** (или, как вариант, **Perl**), Форма может быть создана в графических или специализированных **web**-редакторах, а работа с содержимым организована через **Web**-формы самого **Zope**.

Zope и его объектная модель

В рамках этой лекции невозможно детально рассмотреть такой инструмент как **Zope**, поэтому стоит лишь заметить, что он достаточно интересен не только в качестве среды разработки **web-**приложений, но и с точки зрения подходов. Например, уникальная объектно-ориентированная модель **Zope** позволяет довольно гибко описывать требуемое приложение.

Zope включает в себя следующие возможности:

- **Web-**сервер. **Zope** может работать с **Web-**серверами через **CGI** или использовать свой встроенный **Web-**сервер **(ZServer).**
- Среда разработчика выполнена как **Web-**приложение. **Zope** позволяет создавать **Web-**приложения через **Web-**интерфейс.
- Поддержка сценариев. **Zope** поддерживает несколько языков сценариев: **Python, Perl** и собственный **DTML (Document Template Markup Language,** язык разметки шаблона документа**).**
- База данных объектов. **Zope** использует в своей работе устойчивые объекты, хранимые в специальной базе данных **(ZODB).** Имеется достаточно простой интерфейс для управления этой базой данных.
- Интеграция с реляционными базами данных. **Zope** может хранить свои объекты и другие данные в реляционных СУБД: **Oracle, PostgreSQL, MySQL, Sybase** и т.п.

В ряду других подобных систем **Zope** на первый взгляд кажется странным и неприступным, однако тем, кто с ним "на ты", он открывает большие возможности.

Разработчики **Zope** исходили из лежащей в основе **WWW** объектной модели, в которой загрузку документа по **URI** можно сравнить с отправкой сообщения объекту. Объекты **Zope** разложены по папкам **(folders),** к которым привязаны **политики доступа** для пользователей, имеющих определенные **роли.** В качестве объектов могут выступать документы, изображения, мультимедиа-файлы, адаптеры к базам данных и т.п.

Документы **Zope** можно писать на языке **DTML -** дополнении **HTML** с синтаксисом для включения значений подобно **SSI (Server-Side Include).** Например, для вставки переменной с названием документа можно использовать

```
<!- #var document_title ->
```

Следует заметить, что объекты **Zope** могут иметь свои атрибуты, а также методы, в частности, написанные на языке **Python.** Переменные же могут появляться как из заданных пользователем значений, так и из других источников данных **(**например, из базы данных посредством выполнения выборки функцией **SELECT).**

Сейчас для описания документа **Zope** все чаще применяется **ZPT (Zope Page Templates,** шаблоны страниц **Zope),** которые в свою очередь используют **TAL (Template Attribute Language,** язык шаблонных атрибутов**).** Он позволяет заменять, повторять или пропускать элементы документа описываемого шаблоном документа. "Операторами" языка **TAL** являются **XML-**атрибуты из пространства имен **TAL.** Пространство имен сегодня описывается следующим идентификатором:

```
xmlns:tal="http://xml.zope.org/namespaces/tal"
```

Оператор **TAL** имеет имя и значение **(**что выражается именем и значением атрибута**).** Внутри значения обычно записано **TAL-**выражение, синтаксис которого описывается другим языком **- TALES (Template Attribute Language Expression Syntax,** синтаксис выражений **TAL).**

Таким образом, **ZPT** наполняет содержимым шаблоны, интерпретируя атрибуты **TAL.** Например, чтобы **Zope** подставил название документа **(**тег `TITLE`**),** шаблон может иметь следующий код:

```
<title tal:content="here/title">Doc Title</title>
```

Стоит заметить, что приведенный код сойдет за код на **HTML**, то есть, **Web-**дизайнер может на любом этапе работы над проектом редактировать шаблон в **HTML-**редакторе **(**при условии, что тот сохраняет незнакомые атрибуты из пространства имен **tal).** В этом примере `here/title`является выражением **TALES.** Текст `Doc Title` служит ориентиром для **web-**дизайнера и заменяется значением выражения `here/title`, то есть, будет взято свойство `title` документа **Zope.**

Примечание:

В **Zope** объекты имеют свойства. Набор свойств зависит от типа объекта, но может быть расширен в индивидуальном порядке. Свойство `id` присутствует всегда, свойство `title` обычно тоже указывается.

В качестве более сложного примера можно рассмотреть организацию повтора внутри шаблона **(**для опробования этого примера в **Zope** нужно добавить объект **Page Template):**

```
<ul>
   <li tal:define="s modules/string"
       tal:repeat="el python:s.digits">
   <a href="DUMMY"
      tal:attributes="href string:/digit/$el"
      tal:content="el">SELECTION</a>
  </li>
</ul>
```

Этот шаблон породит следующий результат:

```
<ul>
   <li><a href="/digit/0">0</a></li>
   <li><a href="/digit/1">1</a></li>
   <li><a href="/digit/2">2</a></li>
   <li><a href="/digit/3">3</a></li>
   <li><a href="/digit/4">4</a></li>
   <li><a href="/digit/5">5</a></li>
   <li><a href="/digit/6">6</a></li>
   <li><a href="/digit/7">7</a></li>
   <li><a href="/digit/8">8</a></li>
   <li><a href="/digit/9">9</a></li>
</ul>
```

Здесь нужно обратить внимание на два основных момента:

- в шаблоне можно использовать выражения **Python (**в данном примере переменная `s` определена как модуль **Python)** и переменную**-**счетчик цикла `el`, которая проходит итерации по строке `string.digits`.
- с помощью **TAL** можно задавать не только содержимое элемента, но и атрибута тега **(**в данном примере использовался атрибут `href`**).**

Детали можно узнать по документации. Стоит лишь заметить, что итерация может происходить по самым разным источникам данных: содержимому текущей папки, выборке из базы данных или, как в приведенном примере, по объекту **Python.**

Любой программист знает, что программирование тем эффективнее, чем лучше удалось "расставить скобки", выведя "общий множитель за скобки". Другими словами, хорошие программисты должны быть достаточно "ленивы", чтобы найти оптимальную декомпозицию решаемой задачи. При проектировании динамического **web-**сайта **Zope** позволяет разместить "множители" и "скобки" так, чтобы достигнуть максимального повторного

использования кода (как разметки, так и сценариев). Помогает этому уникальный подход к построению взаимоотношений между объектами: заимствование **(acquisition)**.

Пусть некоторый объект (документ, изображение, сценарий, подключение к базе данных и т.п.) расположен в папке **Example**. Теперь объекты этой папки доступны по имени из любых нижележащих папок. Даже политики безопасности заимствуются более глубоко вложенными папками от папок, которые ближе к корню. Заимствование является очень важной концепцией **Zope**, без понимания которой **Zope** сложно грамотно применять, и наоборот, ее понимание позволяет экономить силы и время, повторно используя объекты в разработке.

Самое интересное, что заимствовать объекты можно также из параллельных папок. Пусть, например, рядом с папкой **Example** находится папка **Zigzag**, в которой лежит нужный объект (его наименование **note**). При этом в папке **Example** программиста интересует объект `index_html`, внутри которого вызывается **note**. Обычный путь к объекту `index_html` будет происходить по **URI** вроде `http://zopeserver/Example/`. А вот если нужно использовать **note** из **Zigzag** (и в папке **Example** его нет), то **URI** будет: `http://zopeserver/Zigzag/Example/`. Таким образом, указание пути в **Zope** отличается от традиционного пути, скажем, в **Unix**: в пути могут присутствовать "зигзаги" через параллельные папки, дающие возможность заимствовать объекты из этих папок. Таким образом, можно сделать конкретную страницу, комбинируя один или несколько независимых аспектов.

Заключение

В этой лекции были рассмотрены различные подходы к использованию **Python** в **web**-приложениях. Самый простой способ реализации **web**-приложения - использование **CGI**-сценариев. Более сложным является использование специальных модулей для **web**-сервера, таких как `mod_python`. Наконец, существуют технологии вроде **Zope**, которые предоставляют специализированные сервисы, позволяющие создавать **web**-приложения.

Лекция #9: Сетевые приложения на Python

Работа с сокетами

Применяемая в IP-сетях архитектура клиент-сервер использует IP-пакеты для коммуникации между клиентом и сервером. Клиент отправляет запрос серверу, на который тот отвечает. В случае с TCP/IP между клиентом и сервером устанавливается соединение (обычно с двусторонней передачей данных), а в случае с UDP/IP - клиент и сервер обмениваются пакетами (дейтаграммамми) с негарантированной доставкой.

Каждый сетевой интерфейс IP-сети имеет уникальный в этой сети адрес (IP-адрес). Упрощенно можно считать, что каждый компьютер в сети Интернет имеет собственный IP-адрес. При этом в рамках одного сетевого интерфейса может быть несколько сетевых портов. Для установления сетевого соединения приложение клиента должно выбрать свободный порт и установить соединение с серверным приложением, которое слушает (listen) порт с определенным номером на удаленном сетевом интерфейсе. Пара IP-адрес и порт характеризуют сокет (гнездо) - начальную (конечную) точку сетевой коммуникации. Для создания соединения TCP/IP необходимо два сокета: один на локальной машине, а другой - на удаленной. Таким образом, каждое сетевое соединение имеет IP-адрес и порт на локальной машине, а также IP-адрес и порт на удаленной машине.

Модуль `socket` обеспечивает возможность работать с сокетами из Python. Сокеты используют транспортный уровень согласно семиуровневой модели OSI (Open Systems Interconnection, взаимодействие открытых систем), то есть относятся к более низкому уровню, чем большинство описываемых в этом разделе протоколов.

Уровни модели OSI:

Физический

Поток битов, передаваемых по физической линии. Определяет параметры физической линии.

Канальный (Ethernet, PPP, ATM и т.п.)

Кодирует и декодирует данные в виде потока битов, справляясь с ошибками, возникающими на физическом уровне в пределах физически единой сети.

Сетевой (IP)

Маршрутизирует информационные пакеты от узла к узлу.

Транспортный (TCP, UDP и т.п.)

Обеспечивает прозрачную передачу данных между двумя точками соединения.

Сеансовый

Управляет сеансом соединения между участниками сети. Начинает, координирует и завершает соединения.

Представления

Обеспечивает независимость данных от формы их представления путем преобразования форматов. На этом уровне может выполняться прозрачное (с точки зрения вышележащего уровня) шифрование и дешифрование данных.

Приложений (HTTP, FTP, SMTP, NNTP, POP3, IMAP и т.д.)

Поддерживает конкретные сетевые приложения. Протокол зависит от типа сервиса.

Каждый сокет относится к одному из коммуникационных доменов. Модуль `socket` поддерживает домены **UNIX** и **Internet**. Каждый домен подразумевает свое семейство протоколов и адресацию. Данное изложение будет затрагивать только домен **Internet**, а именно протоколы **TCP/IP** и **UDP/IP**, поэтому для указания коммуникационного домена при создании сокета будет указываться константа `socket.AF_INET`.

В качестве примера следует рассмотреть простейшую клиент-серверную пару. Сервер будет принимать строку и отвечать клиенту. Сетевое устройство иногда называют хостом **(host)**, поэтому будет употребляться этот термин по отношению к компьютеру, на котором работает сетевое приложение.

Сервер:

```
import socket, string

def do_something(x):
  lst = map(None, x);
  lst.reverse();
  return string.join(lst, "")

HOST = ""                 # localhost
PORT = 33333
srv = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
srv.bind((HOST, PORT))
while 1:
  print "Слушаю порт 33333"
  srv.listen(1)
  sock, addr = srv.accept()
  while 1:
    pal = sock.recv(1024)
    if not pal:
      break
    print "Получено от %s:%s:" % addr, pal
    lap = do_something(pal)
    print "Отправлено %s:%s:" % addr, lap
    sock.send(lap)
  sock.close()
```

Клиент:

```
import socket

HOST = ""                 # удаленный компьютер (localhost)
PORT = 33333              # порт на удаленном компьютере
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect((HOST, PORT))
sock.send("ПАЛИНДРОМ")
result = sock.recv(1024)
sock.close()
print "Получено:", result
```

Примечание:

В примере использованы русские буквы: необходимо указывать кодировку.

Прежде всего, нужно запустить сервер. Сервер открывает сокет на локальной машине на порту **33333**, и адресе **127.0.0.1**. После этого он слушает (`listen()`) порт. Когда на порту появляются данные, принимается (`accept()`) входящее соединение. Метод `accept()` возвращает пару - **Socket**-объект и адрес удаленного компьютера, устанавливающего соединение (пара - **IP**-адрес, порт на удаленной машине). После этого можно применять

методы recv() и send() для общения с клиентом. В recv() задается число байтов в очередной порции. От клиента может прийти и меньшее количество данных.

Код программы-клиента достаточно очевиден. Метод connect() устанавливает соединение с удаленным хостом (в приведенном примере он расположен на той же машине). Данные передаются методом send() и принимаются методом recv()- аналогично тому, что происходит на сервере.

Модуль socket имеет несколько вспомогательных функций. В частности, функции для работы с системой доменных имен (DNS):

```
>>> import socket
>>> socket.gethostbyaddr('www.onego.ru')
('www.onego.ru', [], ['195.161.136.4'])
>>> socket.gethostbyaddr('195.161.136.4')
('www.onego.ru', [], ['195.161.136.4'])
>>> socket.gethostname()
'rnd.onego.ru'
```

В новых версиях **Python** появилась такая функция как socket.getservbyname(). Она позволяет преобразовывать наименования Интернет-сервисов в общепринятые номера портов:

```
>>> for srv in 'http', 'ftp', 'imap', 'pop3', 'smtp':
...   print socket.getservbyname(srv, 'tcp'), srv
...
80 http
21 ftp
143 imap
110 pop3
25 smtp
```

Модуль также содержит большое количество констант для указания протоколов, типов сокетов, коммуникационных доменов и т.п. Другие функции модуля socket можно при необходимости изучить по документации.

Модуль smtplib

Сообщения электронной почты в Интернете передаются от клиента к серверу и между серверами в основном по протоколу **SMTP (Simple Mail Transfer Protocol**, простой протокол передачи почты). Протокол **SMTP** и **ESMTP** (расширенный вариант **SMTP)** описаны в **RFC 821** и **RFC 1869**. Для работы с **SMTP** в стандартной библиотеке модулей имеется модуль smtplib. Для того чтобы начать **SMTP**-соединение с сервером электронной почты, необходимо в начале создать объект для управления **SMTP**-сессией с помощью конструктора класса SMTP:

```
smtplib.SMTP([host[, port]])
```

Параметры host и port задают адрес и порт **SMTP**-сервера, через который будет отправляться почта. По умолчанию, port=25. Если host задан, конструктор сам установит соединение, иначе придется отдельно вызывать метод connect(). Экземпляры класса SMTP имеют методы для всех распространенных команд **SMTP**-протокола, но для отправки почты достаточно вызова конструктора и методов sendmail() и quit():

```
# -*- coding: cp1251 -*-
from smtplib import SMTP
fromaddr = "student@mail.ru"     # От кого
toaddr = "rnd@onego.ru"          # Кому
message = """From: Student <%(fromaddr)s>
To: Lecturer <%(toaddr)s>
```

```
Subject: From Python course student
MIME-Version: 1.0
Content-Type: text/plain; charset=Windows-1251
Content-Transfer-Encoding: 8bit

Здравствуйте! Я изучаю курс по языку Python и
отправляю письмо его автору.
"""
connect = SMTP('mail.onego.ru')
connect.set_debuglevel(1)
connect.sendmail(fromaddr, toaddr, message % vars())
connect.quit()
```

Следует заметить, что `toaddr` в сообщении (в поле `To`) и при отправке могут не совпадать. Дело в том, что получатель и отправитель в ходе **SMTP-**сессии передается командами **SMTP-**протокола. При запуске указанного выше примера на экране появится отладочная информация (ведь уровень отладки задан равным **1**):

```
send: 'ehlo rnd.onego.ru\r\n'
reply: '250-mail.onego.ru Hello as3-042.dialup.onego.ru [195.161.147.4],
pleased to meet you\r\n'
send: 'mail FROM:<student@mail.ru> size=270\r\n'
reply: '250 2.1.0 <student@mail.ru>... Sender ok\r\n'
send: 'rcpt TO:<rnd@onego.ru>\r\n'
reply: '250 2.1.5 <rnd@onego.ru>... Recipient ok\r\n'
send: 'data\r\n'
reply: '354 Enter mail, end with "." on a line by itself\r\n'
send: 'From: Student <student@mail.ru>\r\n . . . '
reply: '250 2.0.0 iBPFgQ7q028433 Message accepted for delivery\r\n'
send: 'quit\r\n'
reply: '221 2.0.0 mail.onego.ru closing connection\r\n'
```

Из этой (несколько сокращенной) отладочной информации можно увидеть, что клиент отправляет **(send)** команды **SMTP-**серверу **(EHLO, MAIL FROM, RCPT TO, DATA, QUIT)**, а тот выполняет команды и отвечает **(reply)**, возвращая код возврата.

В ходе одной **SMTP-**сессии можно отправить сразу несколько писем подряд, если не вызывать `quit()`.

В принципе, команды **SMTP** можно подавать и отдельно: для этого у объекта-соединения есть методы (`helo()`, `ehlo()`, `expn()`, `help()`, `mail()`, `rcpt()`, `vrfy()`, `send()`, `noop()`, `data()`), соответствующие одноименным командам **SMTP-**протокола.

Можно задать и произвольную команду **SMTP-**серверу с помощью метода `docmd()`. В следующем примере показан простейший сценарий, который могут использовать те, кто время от времени принимает почту на свой сервер по протоколу **SMTP** от почтового сервера, на котором хранится очередь сообщений для некоторого домена:

```
from smtplib import SMTP
connect = SMTP('mx.abcde.ru')
connect.set_debuglevel(1)
connect.docmd("ETRN rnd.abcde.ru")
connect.quit()
```

Этот простенький сценарий предлагает серверу `mx.abcde.ru` попытаться связаться с основным почтовым сервером домена `rnd.abcde.ru` и переслать всю накопившуюся для него почту.

При работе с классом `smtplib.SMTP` могут возбуждаться различные исключения. Назначение некоторых из них приведено ниже:

`smtplib.SMTPException`

Базовый класс для всех исключений модуля.

`smtplib.SMTPServerDisconnected`

Сервер неожиданно прервал связь (или связь с сервером не была установлена).

`smtplib.SMTPResponseException`

Базовый класс для всех исключений, которые имеют код ответа **SMTP-**сервера.

`smtplib.SMTPSenderRefused`

Отправитель отвергнут

`smtplib.SMTPRecipientsRefused`

Все получатели отвергнуты сервером.

`smtplib.SMTPDataError`

Сервер ответил неизвестным кодом на данные сообщения.

`smtplib.SMTPConnectError`

Ошибка установления соединения.

`smtplib.SMTPHeloError`

Сервер не ответил правильно на команду `HELO` или отверг ее.

Модуль poplib

Еще один протокол - **POP3 (Post Office Protocol,** почтовый протокол**) -** служит для приема почты из почтового ящика на сервере (протокол определен в **RFC 1725).**

Для работы с почтовым сервером требуется установить с ним соединение и, подобно рассмотренному выше примеру, с помощью **SMTP-**команд получить требуемые сообщения. Объект-соединение **POP3** можно установить посредством конструктора класса `POP3` из модуля `poplib`:

```
poplib.POP3(host[, port])
```

Где `host` - адрес **POP3-**сервера, `port` - порт на сервере (по умолчанию **110),** `pop_obj` - объект для управления сеансом работы с **POP3-**сервером.

Следующий пример демонстрирует основные методы для работы с **POP3-**соединением:

```
import poplib, email
# Учетные данные пользователя:
SERVER = "pop.server.com"
USERNAME = "user"
USERPASSWORD = "secretword"

p = poplib.POP3(SERVER)
print p.getwelcome()
# этап идентификации
print p.user(USERNAME)
```

```
print p.pass_(USERPASSWORD)
# этап транзакций
response, lst, octets = p.list()
print response
for msgnum, msgsize in [i.split() for i in lst]:
  print "Сообщение %(msgnum)s имеет длину %(msgsize)s" % vars()
  print "UIDL =", p.uidl(int(msgnum)).split()[2]
  if int(msgsize) > 32000:
    (resp, lines, octets) = p.top(msgnum, 0)
  else:
    (resp, lines, octets) = p.retr(msgnum)
  msgtxt = "\n".join(lines)+"\n\n"
  msg = email.message_from_string(msgtxt)
  print "* От: %(from)s\n* Кому: %(to)s\n* Тема: %(subject)s\n" % msg
  # msg содержит заголовки сообщения или все сообщение (если оно небольшое)

# этап обновления
print p.quit()
```

Примечание:

Разумеется, чтобы пример сработал корректно, необходимо внести реальные учетные данные.

При выполнении сценарий выведет на экран примерно следующее.

```
+OK POP3 pop.server.com server ready
+OK User name accepted, password please
+OK Mailbox open, 68 messages
+OK Mailbox scan listing follows
Сообщение 1 имеет длину 4202
UIDL = 4152a47e00000004
* От: online@kaspersky.com
* Кому: user@server.com
* Тема: KL Online Activation

...

+OK Sayonara
```

Эти и другие методы экземпляров класса `POP3` описаны ниже:

Метод	Команда POP3	Описание
`getwelcome()`		Получает строку `s` с приветствием **POP3-**сервера
`user(name)`	**USER name**	Посылает команду `USER` с указанием имени пользователя `name`. Возвращает строку с ответом сервера
`pass_(pwd)`	**PASS pwd**	Отправляет пароль пользователя в команде **PASS.** После этой команды и до выполнения команды **QUIT** почтовый ящик блокируется
`apop(user, secret)`	**APOP user secret**	Идентификация на сервере по **APOP**
`rpop(user)`	**RPOP user**	Идентификация по методу **RPOP**
`stat()`	**STAT**	Возвращает кортеж с информацией о почтовом ящике. В нем `m` - количество сообщений, `l` - размер почтового ящика в байтах
`list([num])`	**LIST [num]**	Возвращает список сообщений в формате `(resp, ['num octets', ...])`, если не указан `num`, и `"+OK num octets"`, если указан. Список `lst` состоит из строк в формате `"num octets"`.
`retr(num)`	**RETR num**	Загружает с сервера сообщение с номером `num` и возвращает

		кортеж с ответом сервера `(resp, lst, octets)`
`dele(num)`	**DELE num**	Удаляет сообщение с номером `num`
`rset()`	**RSET**	Отменяет пометки удаления сообщений
`noop()`	**NOOP**	Ничего не делает **(**поддерживает соединение**)**
`quit()`	**QUIT**	Отключение от сервера**.** Сервер выполняет все необходимые изменения **(**удаляет сообщения**)** и снимает блокировку почтового ящика
`top(num, lines)`	**TOP num lines**	Команда аналогична **RETR,** но загружает только заголовок и `lines` строк тела сообщения**.** Возвращает кортеж `(resp, lst, octets)`
`uidl([num])`	**UIDL [num]**	Сокращение от **"unique-id listing" (**список уникальных идентификаторов сообщений**).** Формат результата**:** `(resp, lst, octets)`**,** если `num` не указан**,** и `"+OK num uniqid"`**,** если указан**.** Список `lst` состоит из строк вида `"+OK num uniqid"`

В этой таблице `num` обозначает номер сообщения **(**он не меняется на протяжении всей сессии**),** `resp` **--** ответ сервера**,** возвращается для любой команды**,** начинается с `"+OK "` для успешных операций **(**при неудаче возбуждается исключение `poplib.proto_error`**).** Параметр `octets` обозначает количество байт в принятых данных**.** `uniqid` **-** идентификатор сообщения**,** генерируемый сервером**.**

Работа с **POP3-**сервером состоит из трех фаз**:** идентификации**,** транзакций и обновления**.** На этапе идентификации сразу после создания **POP3-**объекта разрешены только команды **USER**, **PASS (**иногда **APOP** и **RPOP)**. После идентификации сервер получает информацию о пользователе и наступает этап транзакций**.** Здесь уместны остальные команды**.** Этап обновления вызывается командой **QUIT,** после которой **POP3-**сервер обновляет почтовый ящик пользователя в соответствии с поданными командами**,** а именно **-** удаляет помеченные для удаления сообщения**.**

Модули для клиента WWW

Стандартные средства языка **Python** позволяют получать из программы доступ к объектам **WWW** как в простых случаях**,** так и при сложных обстоятельствах**,** в частности при необходимости передавать данные формы**,** идентификации**,** доступа через прокси и т**.**п**.**

Стоит отметить**,** что при работе с **WWW** используется в основном протокол **HTTP,** однако **WWW** охватывает не только **HTTP,** но и многие другие схемы **(FTP**, **gopher**, **HTTPS** и т**.**п**.).** Используемая схема обычно указана в самом начале **URL.**

Функции для загрузки сетевых объектов

Простой случай получения **WWW-**объекта по известному **URL** показан в следующем примере**:**

```
import urllib
doc = urllib.urlopen("http://python.onego.ru").read()
print doc[:40]
```

Функция `urllib.urlopen()` создает файлоподобный объект**,** который читает методом `read()`**.** Другие методы этого объекта**:** `readline()`**,** `readlines()`**,** `fileno()`**,** `close()` работают как и у обычного файла**,** а также есть метод `info()`**,** который возвращает соответствующий полученному с сервера **Message-**объект**.** Этот объект можно использовать для получения дополнительной информации**:**

```
>>> import urllib
>>> f = urllib.urlopen("http://python.onego.ru")
```

```
>>> print f.info()
Date: Sat, 25 Dec 2004 19:46:11 GMT
Server: Apache/1.3.29 (Unix) PHP/4.3.10
Content-Type: text/html; charset=windows-1251
Content-Length: 4291
>>> print f.info()['Content-Type']
text/html; charset=windows-1251
```

С помощью функции `urllib.urlopen()` можно делать и более сложные вещи, например, передавать **web**-серверу данные формы. Как известно, данные заполненной **web**-формы могут быть переданы на **web**-сервер с использованием метода **GET** или метода **POST**. Метод **GET** связан с кодированием всех передаваемых параметров после знака `"?"` в **URL**, а при методе **POST** данные передаются в теле **HTTP**-запроса. Оба варианта передачи представлены ниже:

```
import urllib
data = {"search": "Python"}
enc_data = urllib.urlencode(data)

# метод GET
f = urllib.urlopen("http://searchengine.com/search" + "?" + enc_data)
print f.read()

# метод POST
f = urllib.urlopen("http://searchengine.com/search", enc_data)
print f.read()
```

В некоторых случаях данные имеют повторяющиеся имена. В этом случае в качестве параметра `urllib.urlencode()` можно использовать вместо словаря последовательность пар имя-значение:

```
>>> import urllib
>>> data = [("n", "1"), ("n", "3"), ("n", "4"), ("button", "Привет"),]
>>> enc_data = urllib.urlencode(data)
>>> print enc_data
n=1&n=3&n=4&button=%F0%D2%C9%D7%C5%D4
```

Модуль `urllib` позволяет загружать **web**-объекты через прокси-сервер. Если ничего не указывать, будет использоваться прокси-сервер, который был задан принятым в конкретной ОС способом. В **Unix** прокси-серверы задаются в переменных окружения `http_proxy`, `ftp_proxy` и т.п., в **Windows** прокси-серверы записаны в реестре, а в **Mac OS** они берутся из конфигурации **Internet**. Задать прокси-сервер можно и как именованный параметр `proxies` к `urllib.urlopen()`:

```
# Использовать указанный прокси
proxies = proxies={'http': 'http://www.proxy.com:3128'}
f = urllib.urlopen(some_url, proxies=proxies)
# Не использовать прокси
f = urllib.urlopen(some_url, proxies={})
# Использовать прокси по умолчанию
f = urllib.urlopen(some_url, proxies=None)
f = urllib.urlopen(some_url)
```

Функция `urlretrieve()` позволяет записать заданный **URL** сетевой объект в файл. Она имеет следующие параметры:

```
urllib.urlretrieve(url[, filename[, reporthook[, data]]])
```

Здесь `url` - **URL** сетевого объекта, `filename` - имя локального файла для помещения объекта, `reporthook` - функция, которая будет вызываться для сообщения о состоянии загрузки, `data` - данные для метода **POST** (если он используется). Функция возвращает

кортеж `(filepath, headers)`, где `filepath` - имя локального файла, в который закачан объект, `headers` - результат метода `info()` для объекта, возвращенного `urlopen()`.

Для обеспечения интерактивности функция `urllib.urlretrieve()` вызывает время от времени функцию, заданную в `reporthook()`. Этой функции передаются три аргумента: количество принятых блоков, размер блока и общий размер принимаемого объекта в байтах (если он неизвестен, этот параметр равен -1).

В следующем примере программа принимает большой файл и, чтобы пользователь не скучал, пишет процент от выполненной загрузки и предполагаемое оставшееся время:

```
FILE = 'boost-1.31.0-9.src.rpm'
URL = 'http://download.fedora.redhat.com/pub/fedora/linux/core/3/SRPMS/' + FILE

def download(url, file):
  import urllib, time
  start_t = time.time()

  def progress(bl, blsize, size):
    dldsize = min(bl*blsize, size)
    if size != -1:
      p = float(dldsize) / size
      try:
        elapsed = time.time() - start_t
        est_t = elapsed / p - elapsed
      except:
        est_t = 0
      print "%6.2f %% %6.0f s %6.0f s %6i / %-6i bytes" % (
          p*100, elapsed, est_t, dldsize, size)
    else:
      print "%6i / %-6i bytes" % (dldsize, size)

  urllib.urlretrieve(URL, FILE, progress)

download(URL, FILE)
```

Эта программа выведет примерно следующее (процент от полного объема закачки, прошедшие секунды, предполагаемое оставшееся время, закачанные байты, полное количество байтов):

```
 0.00 %      1 s      0 s      0 / 6952309 bytes
 0.12 %      5 s   3941 s   8192 / 6952309 bytes
 0.24 %      7 s   3132 s  16384 / 6952309 bytes
 0.35 %     10 s   2864 s  24576 / 6952309 bytes
 0.47 %     12 s   2631 s  32768 / 6952309 bytes
 0.59 %     15 s   2570 s  40960 / 6952309 bytes
 0.71 %     18 s   2526 s  49152 / 6952309 bytes
 0.82 %     20 s   2441 s  57344 / 6952309 bytes
 ...
```

Функции для анализа URL

Согласно документу **RFC 2396** URL должен строиться по следующему шаблону:

```
scheme://netloc/path;parameters?query#fragment
```

где

```
scheme
```

Адресная схема. Например: **http**, **ftp**, **gopher**.

```
netloc
```

Местонахождение в сети.

```
path
```

Путь к ресурсу.

```
params
```

Параметры.

```
query
```

Строка запроса.

```
frag
```

Идентификатор фрагмента.

Одна из функций уже использовалась для формирования **URL -** `urllib.urlencode()`**.** Кроме нее в модуле `urllib` имеются и другие функции:

```
quote(s, safe='/')
```

Функция экранирует символы в **URL,** чтобы их можно было отправлять на **web-**сервер**.** Она предназначена для экранирования пути к ресурсу**,** поэтому оставляет `'/'` как есть**.** Например:

```
>>> urllib.quote("rnd@onego.ru")
'rnd%40onego.ru'
>>> urllib.quote("a = b + c")
'a%20%3D%20b%20%2B%20c'
>>> urllib.quote("0/1/1")
'0/1/1'
>>> urllib.quote("0/1/1", safe="")
'0%2F1%2F1'
quote_plus(s, safe='')
```

Функция экранирует некоторые символы в **URL (**в строке запроса**),** чтобы их можно было отправлять на **web-**сервер**.** Аналогична `quote()`**,** но заменяет пробелы на плюсы**.**

```
unquote(s)
```

Преобразование**,** обратное `quote_plus()`**.** Пример:

```
>>> urllib.unquote('a%20%3D%20b%20%2B%20c')
'a = b + c'
unquote_plus(s)
```

Преобразование**,** обратное `quote_plus()`**.** Пример:

```
>>> urllib.unquote_plus('a+=+b+%2B+c')
'a = b + c'
```

Для анализа **URL** можно использовать функции из модуля `urlparse`**:**

```
urlparse(url, scheme='', allow_fragments=1)
```

Разбирает **URL** в **6** компонентов **(**сохраняя экранирование символов**):**
`scheme://netloc/path;params?query#frag`

```
urlsplit(url, scheme='', allow_fragments=1)
```

Разбирает **URL** в **6** компонентов **(**сохраняя экранирование символов**):**
`scheme://netloc/path?query#frag`

```
urlunparse((scheme, netloc, url, params, query, fragment))
```

Собирает **URL** из **6** компонентов**.**

```
urlunsplit((scheme, netloc, url, query, fragment))
```

Собирает **URL** из **5** компонентов**.**

Пример**:**

```
>>> from urlparse import urlsplit, urlunsplit
>>> URL = "http://google.com/search?q=Python"
>>> print urlsplit(URL)
('http', 'google.com', '/search', 'q=Python', '')
>>> print urlunsplit(
...     ('http', 'google.com', '/search', 'q=Python', ''))
http://google.com/search?q=Python
```

Еще одна функция того же модуля `urlparse` позволяет корректно соединить две части **URL - ** базовую и относительную**:**

```
>>> import urlparse
>>> urlparse.urljoin('http://python.onego.ru', 'itertools.html')
'http://python.onego.ru/itertools.html'
```

Возможности urllib2

Функциональности модулей `urllib` и `urlparse` хватает для большинства задач**,** которые решают сценарии на **Python** как **web-**клиенты**.** Тем не менее**,** иногда требуется больше**.** На этот случай можно использовать модуль для работы с протоколом **HTTP - `httplib` -** и создать собственный класс для **HTTP-**запросов **(**в лекциях модуль `httplib` не рассматривается**).** Однако вполне вероятно**,** что нужная функциональность уже имеется в модуле `urllib2`**.**

Одна из полезных возможностей этих модулей **-** доступ к **web-**объектам**,** требующий авторизации**.** Ниже будет рассмотрен пример**,** который не только обеспечит доступ с авторизацией**,** но и обозначит основную идею модуля `urllib2:` использование обработчиков **(handlers),** каждый из которых решает узкую специфическую задачу**.**

Следующий пример показывает**,** как создать собственный открыватель **URL** с помощью модуля `urllib2` **(**этот пример взят из документации по **Python):**

```
import urllib2

# Подготовка идентификационных данных
authinfo = urllib2.HTTPBasicAuthHandler()
authinfo.add_password('My page', 'localhost', 'user1', 'secret')

# Доступ через прокси
proxy_support = urllib2.ProxyHandler({'http' : 'http://localhost:8080'})

# Создание нового открывателя с указанными обработчиками
```

```
opener = urllib2.build_opener(proxy_support,
                              authinfo,
                              urllib2.CacheFTPHandler)
# Установка поля с названием клиента
opener.addheaders = [('User-agent', 'Mozilla/5.0')]

# Установка нового открывателя по умолчанию
urllib2.install_opener(opener)

# Использование открывателя
f = urllib2.urlopen('http://localhost/mywebdir/')
print f.read()[:100]
```

В этом примере получен доступ к странице, которую охраняет **mod_python** (см. предыдущую лекцию). Первый аргумент при вызове метода `add_password()` задает область действия **(realm)** идентификационных данных (он задан директивой `AuthName "My page"` в конфигурации **web**-сервера). Остальные параметры достаточно понятны: имя хоста, на который нужно получить доступ, имя пользователя и его пароль. Разумеется, для корректной работы примера нужно, чтобы на локальном **web**-сервере был каталог, требующий авторизации.

В данном примере явным образом затронуты всего три обработчика: `HTTPBasicAuthHandler`, `ProxyHandler` и `CacheFTPHandler`. В модуле `urllib2` их более десятка, назначение каждого можно узнать из документации к используемой версии **Python**. Есть и специальный класс для управления открывателями: `OpenerDirector`. Именно его экземпляр создала функция `urllib2.build_opener()`.

Модуль `urllib2` имеет и специальный класс для воплощения запроса на открытие **URL**. Называется этот класс `urllib2.Request`. Его экземпляр содержит состояние запроса. Следующий пример показывает, как получить доступ к каталогу с авторизацией, используя добавление заголовка в **HTTP**-запрос:

```
import urllib2, base64
req = urllib2.Request('http://localhost/mywebdir')
b64 = base64.encodestring('user1:secret').strip()
req.add_header('Authorization', 'Basic %s' % b64)
req.add_header('User-agent', 'Mozilla/5.0')
f = urllib2.urlopen(req)
print f.read()[:100]
```

Как видно из этого примера, ничего загадочного в авторизации нет: **web**-клиент вносит (закодированные **base64**) идентификационные данные в поле `Authorization` **HTTP**-запроса.

Примечание:

Приведенные два примера почти эквивалентны, только во втором примере прокси-сервер не назначен явно.

XML-RPC сервер

До сих пор высокоуровневые протоколы рассматривались с точки зрения клиента. Не менее просто создавать на **Python** и их серверные части. Для иллюстрации того, как разработать программу на **Python**, реализующую сервер, был выбран протокол **XML-RPC**. Несмотря на свое название, конечному пользователю необязательно знать **XML** (об этом языке разметки говорилось на одной из предыдущих лекций), так как он скрыт от него. Сокращение **RPC** (**Remote Procedure Call**, вызов удаленной процедуры) объясняет суть дела: с помощью **XML-RPC** можно вызывать процедуры на удаленном хосте. Причем при помощи **XML-RPC** можно абстрагироваться от конкретного языка программирования за счет использования общепринятых типов данных (строки, числа, логические значения и т.п.). В

языке **Python** вызов удаленной функции по синтаксису ничем не отличается от вызова обычной функции:

```
import xmlrpclib

# Установить соединение
req = xmlrpclib.ServerProxy("http://localhost:8000")

try:
    # Вызвать удаленную функцию
    print req.add(1, 3)
except xmlrpclib.Error, v:
    print "ERROR",
```

А вот как выглядит **XML-RPC**-сервер (для того чтобы попробовать пример выше, необходимо сначала запустить сервер):

```
from SimpleXMLRPCServer import SimpleXMLRPCServer
srv = SimpleXMLRPCServer(("localhost", 8000))  # Запустить сервер
srv.register_function(pow)  # Зарегистрировать функцию
srv.register_function(lambda x,y: x+y, 'add')  # И еще одну
srv.serve_forever()  # Обслуживать запросы
```

С помощью **XML-RPC** (а этот протокол достаточно "легковесный" среди других подобных протоколов) приложения могут общаться друг с другом на понятном им языке вызова функций с параметрами основных общепринятых типов и такими же возвращаемыми значениями. Преимуществом же **Python** является удобный синтаксис вызова удаленных функций.

Внимание!

Разумеется, это только пример. При реальном использовании необходимо позаботиться, чтобы **XML-RPC** сервер отвечал требованиям безопасности. Кроме того, сервер лучше делать многопоточным, чтобы он мог обрабатывать несколько потоков одновременно. Для многопоточности (она будет обсуждаться в отдельной лекции) не нужно многое переделывать: достаточно определить свой класс, скажем, `ThreadingXMLRPCServer`, в котором вместо `SocketServer.TCPServer` использовать `SocketServer.ThreadingTCPServer`. Это предлагается в качестве упражнения. Наводящий вопрос: где находится определение класса `SimpleXMLRPCServer`?

Заключение

В этой лекции на практических примерах и сведениях из документации были показаны возможности, которые дает стандартный **Python** для работы в Интернете. Из сценария на **Python** можно управлять соединением на уровне сокетов, а также использовать модули для конкретного сетевого протокола или набора протоколов. Для работы с сокетами служит модуль `socket`, а модули для высокоуровневых протоколов имеют такие названия как `smtplib`, `poplib`, `httplib` и т.п. Для работы с системой **WWW** можно использовать модули `urllib`, `urllib2`, `urlparse`. Указанные модули рассмотрены с точки зрения типичного применения. Для решения нестандартных задач лучше обратиться к другим источникам: документации, исходному коду модулей, поиску в Интернете. В этой лекции говорилось и о серверной составляющей высокоуровневых сетевых протоколов. В качестве примера приведена клиент-серверная пара для протокола **XML-RPC**. Этот протокол создан на основе **HTTP**, но служит специальной цели.

Лекция #10: Работа с базой данных

Основные понятия реляционной СУБД

Реляционная база данных - это набор таблиц с данными.

Таблица - это прямоугольная матрица, состоящая из строк и столбцов. Таблица задает отношение **(relation).**

Строка - запись, состоящая из полей **-** столбцов. В каждом поле может содержаться некоторое значение, либо специальное значение **NULL (**пусто**).** В таблице может быть произвольное количество строк. Для реляционной модели порядок расположения строк не определен и не важен.

Каждый **столбец** в таблице имеет собственное имя и тип.

Что такое DB-API 2

Вынесенная в заголовок аббревиатура объединяет два понятия: **DB (Database,** база данных**)** и **API (Application Program Interface,** интерфейс прикладной программы**).**

Таким образом, **DB-API** определяет интерфейс прикладной программы с базой данных. Этот интерфейс, описываемый ниже, должен реализовывать все модули расширения, которые служат для связи **Python-**программ с базами данных. Единый **API (**в настоящий момент его вторая версия**)** позволяет абстрагироваться от марки используемой базы данных, при необходимости довольно легко менять одну СУБД на другую, изучив всего один набор функций и методов.

DB-API 2.0 описан в **PEP 249 (**сайт **http://www.python.org/peps/pep-0249.html/),** и данное ниже описание основано именно на нем.

Описание DB API 2.0

DB API 2.0 регламентирует интерфейсы модуля расширения для работы с базой данных, методы объекта-соединения с базой, объекта-курсора текущей обрабатываемой записи, объектов различных для типов данных и их конструкторов, а также содержит рекомендации для разработчиков по реализации модулей. На сегодня **Python** поддерживает через модули расширения многие известные базы данных **(**уточнить можно на **web-**странице по адресу **http://www.python.org/topics/database/).** Ниже рассматриваются почти все положения **DB-API** за исключением рекомендаций для разработчиков новых модулей.

Интерфейс модуля

Здесь необходимо сказать о том, что должен предоставлять модуль для удовлетворения требований **DB-API 2.0.**

Доступ к базе данных осуществляется с помощью **объекта-соединения (connection object). DB-API-**совместимый модуль должен предоставлять функцию-конструктор `connect()` для класса объектов-соединений. Конструктор должен иметь следующие именованные параметры:

- `dsn` Название источника данных в виде строки
- `user` Имя пользователя
- `password` Пароль
- `host` Адрес хоста, на котором работает СУБД
- `database` Имя базы данных.

Методы объекта-соединения будут рассмотрены чуть позже.

Модуль определяет константы, содержащие его основные характеристики:

- `apilevel` Версия **DB-API ("1.0"** или **"2.0").**
- `threadsafety` Целочисленная константа, описывающая возможности модуля при использовании потоков управления:
- `0` Модуль не поддерживает потоки.
- `1` Потоки могут совместно использовать модуль, но не соединения.
- `2` Потоки могут совместно использовать модуль и соединения.
- `3` Потоки могут совместно использовать модуль, соединения и курсоры. **(**Под совместным использованием здесь понимается возможность использования упомянутых ресурсов без применения семафоров**).**
- `paramstyle` Тип используемых пометок при подстановке параметров. Возможны следующие значения этой константы:
- `"format"` Форматирование в стиле языка **ANSI C (**например, `"%s"`, `"%i"`**).**
- `"pyformat"` Использование именованных спецификаторов формата в стиле **Python (**`"%(item)s"`**)**
- `"qmark"` Использование знаков `"?"` для пометки мест подстановки параметров.
- `"numeric"` Использование номеров позиций **(**`":1"`**).**
- `"named"` Использование имен подставляемых параметров **(**`":name"`**).**

Модуль должен определять ряд исключений для обозначения типичных исключительных ситуаций: `Warning` **(**предупреждение**),** `Error` **(**ошибка**),** `InterfaceError` **(**ошибка интерфейса**),** `DatabaseError` **(**ошибка, относящаяся к базе данных**).** А также подклассы этого последнего исключения: `DataError` **(**ошибка обработки данных**),** `OperationalError` **(**ошибка в работе или сбой соединения с базой данных**),** `IntegrityError` **(**ошибка целостности базы данных**),** `InternalError` **(**внутренняя ошибка базы данных**),** `ProgrammingError` **(**программная ошибка, например, ошибка в синтаксисе **SQL-**запроса**),** `NotSupportedError` **(**при отсутствии поддержки запрошенного свойства**).**

Объект-соединение

Объект-соединение, получаемый в результате успешного вызова функции `connect()`, должен иметь следующие методы:

- `close()` Закрывает соединение с базой данных.
- `commit()` Завершает транзакцию.
- `rollback()` Откатывает начатую транзакцию **(**восстанавливает исходное состояние**).** Закрытие соединения при незавершенной транзакции автоматически производит откат транзакции.
- `cursor()` Возвращает объект-курсор, использующий данное соединение. Если база данных не поддерживает курсоры, модуль сопряжения должен их имитировать.

Под **транзакцией** понимается группа из одной или нескольких операций, которые изменяют базу данных. Транзакция соответствует логически неделимой операции над базой данных, а частичное выполнение транзакции приводит к нарушению целостности БД. Например, при переводе денег с одного счета на другой операции по уменьшению первого счета и увеличению второго являются транзакцией. Методы `commit()` и `rollback()` обозначают начало и конец транзакции в явном виде. Кстати, не все базы данных поддерживают механизм транзакций.

Следует отметить, что в зависимости от реализации **DB-API 2.0** модуля, необходимо сохранять ссылку на объект-соединение в продолжение работы курсоров этого соединения. В частности, это означает, что нельзя сразу же получать объект-курсор, не привязывая объект-соединение к некоторому имени. Также нельзя оставлять объект-соединение в локальной переменной, возвращая из функции или метода объект-курсор.

Объект-курсор

Курсор (от англ. **cursor - CURrrent Set Of Records**, текущий набор записей) служит для работы с результатом запроса. Результатом запроса обычно является одна или несколько прямоугольных таблиц со столбцами-полями и строками-записями. Приложение может читать и обрабатывать полученные таблицы и записи в таблице по одной, поэтому в курсоре хранится информация о текущей таблице и записи. Конкретный курсор в любой момент времени связан с выполнением одной **SQL**-инструкции.

Атрибуты объекта-курсора тоже определены **DB-API**:

- `arraysize` Атрибут, равный количеству записей, возвращаемых методом `fetchmany()`. По умолчанию равен **1**.
- `callproc(procname[, params])` Вызывает хранимую процедуру `procname` с параметрами из изменчивой последовательности `params`. Хранимая процедура может изменить значения некоторых параметров последовательности. Метод может возвратить результат, доступ к которому осуществляется через `fetch`-методы.
- `close()` Закрывает объект-курсор.
- `description` Этот доступный только для чтения атрибут является последовательностью из семиэлементных последовательностей. Каждая из этих последовательностей содержит информацию, описывающую один столбец результата:
- `(name, type_code, display_size, internal_size, precision, scale, null_ok)` Первые два элемента (имя и тип) обязательны, а вместо остальных (размер для вывода, внутренний размер, точность, масштаб, возможность задания пустого значения) может быть значение `None`. Этот атрибут может быть равным `None` для операций, не возвращающих значения.
- `execute(operation[, parameters])` Исполняет запрос к базе данных или команду СУБД. Параметры (`parameters`) могут быть представлены в принятой в базе данных нотации в соответствии с атрибутом `paramstyle`, описанным выше.
- `executemany(operation, seq_of_parameters)` Выполняет серию запросов или команд, подставляя параметры в заданный шаблон. Параметр `seq_of_parameters` задает последовательность наборов параметров.
- `fetchall()` Возвращает все (или все оставшиеся) записи результата запроса.
- `fetchmany([size])` Возвращает следующие несколько записей из результатов запроса в виде последовательности последовательностей. Пустая последовательность означает отсутствие данных. Необязательный параметр `size` указывает количество возвращаемых записей (реально возвращаемых записей может быть меньше). По умолчанию `size` равен атрибуту `arraysize` объекта-курсора.
- `fetchone()` Возвращает следующую запись (в виде последовательности) из результата запроса или `None` при отсутствии данных.
- `nextset()` Переводит курсор к началу следующего набора данных, полученного в результате запроса (при этом часть записей в предыдущем наборе может остаться непрочитанной). Если наборов больше нет, возвращает `None`. Не все базы данных поддерживают возврат нескольких наборов результатов за одну операцию.
- `rowcount` Количество записей, полученных или затронутых в результате выполнения последнего запроса. В случае отсутствия **execute**-запросов или невозможности указать количество записей равен **-1**.
- `setinputsizes(sizes)` Предопределяет области памяти для параметров, используемых в операциях. Аргумент `sizes` задает последовательность, где каждый элемент соответствует одному входному параметру. Элемент может быть объектом-типом соответствующего параметра или целым числом, задающим длину строки. Он также может иметь значение `None`, если о размере входного параметра ничего нельзя сказать заранее или он предполагается очень большим. Метод должен быть вызван до **execute**-методов.
- `setoutputsize(size[, column])` Устанавливает размер буфера для выходного параметра из столбца с номером `column`. Если `column` не задан, метод устанавливает

размер для всех больших выходных параметров. Может использоваться, например, для получения **больших бинарных объектов (Binary Large Object, BLOB)**.

Объекты-типы

DB-API 2.0 предусматривает названия для объектов-типов, используемых для описания полей базы данных:

Объект	Тип
`STRING`	Строка и символ
`BINARY`	Бинарный объект
`NUMBER`	Число
`DATETIME`	Дата и время
`ROWID`	Идентификатор записи
`None`	**NULL**-значение (отсутствующее значение)

С каждым типом данных (в реальности это - классы) связан конструктор. Совместимый с **DB-API** модуль должен определять следующие конструкторы:

- `Date(год, месяц, день)` Дата.
- `Time(час, минута, секунда)` Время.
- `Timestamp(год, месяц, день, час, минута, секунда)` Дата-время.
- `DateFromTicks(secs)` Дата в виде числа секунд `secs` от начала эпохи (**1** января **1970** года).
- `TimeFromTicks(secs)` Время, то же.
- `TimestampFromTicks(secs)` Дата-время, то же.
- `Binary(string)` Большой бинарный объект на основании строки `string`.

Работа с базой данных из Python-приложения

Далее в лекции на конкретных примерах будет показано, как работать с базой данных из программы на языке **Python**. Нужно отметить, что здесь не ставится цели постичь премудрости языка запросов (это тема отдельного курса). Простые примеры позволят понять, что при программировании на **Python** доступ к базе данных не сложнее доступа к другим источникам данных (файлам, сетевым объектам).

Именно поэтому для демонстрации выбрана СУБД **SQLite**, работающая как под **Unix**, так и под **Windows**. Кроме установки собственно **SQLite** (сайт **http://sqlite.org**) и модуля сопряжения с **Python** (**http://pysqlite.org**), каких-либо дополнительных настроек проводить не требуется, так как **SQLite** хранит данные базы в отдельном файле: сразу приступать к созданию таблиц, занесению в них данных и произведению запросов нельзя. Выбранная СУБД (в силу своей "легкости") имеет одну существенную особенность: за одним небольшим исключением, СУБД **SQLite** не обращает внимания на типы данных (она хранит все данные в виде строк), поэтому модуль расширения `sqlite` для **Python** проделывает дополнительную работу по преобразованию типов. Кроме того, СУБД **SQLite** поддерживает достаточно большое подмножество свойств стандарта **SQL92**, оставаясь при этом небольшой и быстрой, что немаловажно, например, для **web**-приложений. Достаточно сказать, что **SQLite** поддерживает даже транзакции.

Еще раз стоит повторить, что выбор учебной базы данных не влияет на синтаксис использованных средств, так как модуль `sqlite`, который будет использоваться, поддерживает **DB-API 2.0**, а значит, переход на любую другую СУБД потребует минимальных изменений в вызове функции `connect()` и, возможно, использования более удачных типов данных, свойственных целевой СУБД.

Схематично работа с базой данных может выглядеть примерно так:

- Подключение к базе данных (вызов `connect()` с получением объекта-соединения).
- Создание одного или нескольких курсоров (вызов метода объекта-соединения `cursor()` с получением объекта-курсора).
- Исполнение команды или запроса (вызов метода `execute()` или его вариантов).
- Получение результатов запроса (вызов метода `fetchone()` или его вариантов).
- Завершение транзакции или ее откат (вызов метода объекта-соединения `commit()` или `rollback()`).
- Когда все необходимые транзакции произведены, подключение закрывается вызовом метода `close()` объекта-соединения.

Знакомство с СУБД

Допустим, программное обеспечение установлено правильно, и можно работать с модулем `sqlite`. Стоит посмотреть, чему будут равны константы:

```
>>> import sqlite
>>> sqlite.apilevel
'2.0'
>>> sqlite.paramstyle
'pyformat'
>>> sqlite.threadsafety
1
```

Отсюда следует, что `sqlite` поддерживает **DB-API 2.0**, подстановка параметров выполняется в стиле строки форматирования языка **Python**, а соединения нельзя совместно использовать из различных потоков управления (без блокировок).

Создание базы данных

Для создания базы данных нужно установить, какие таблицы (и другие объекты, например индексы) в ней будут храниться, а также определить структуры таблиц (имена и типы полей).

Задача - создание базы данных, в которой будет храниться телепрограмма. В этой базе будет таблица со следующими полями:

- `tvdate`,
- `tvweekday`,
- `tvchannel`,
- `tvtime1`,
- `tvtime2`,
- `prname`,
- `prgenre`.

Здесь `tvdate` - дата, `tvchannel` - канал, `tvtime1` и `tvtime2` - время начала и конца передачи, `prname` - название, `prgenre` - жанр. Конечно, в этой таблице есть функциональная зависимость (`tvweekday` вычисляется на основе `tvdate` и `tvtime1`), но из практических соображений БД к нормальным формам приводиться не будет. Кроме того, таблица будет создана с названиями дней недели (устанавливает соответствие между номером дня и днем недели):

- `weekday`,
- `wdname`.

Следующий сценарий создаст таблицу в базе данных (в случае с **SQLite** заботиться о создании базы данных не нужно: файл создастся автоматически. Для других баз данных

необходимо перед этим создать базу данных, например, **SQL-**инструкцией CREATE DATABASE**):**

```
import sqlite as db

c = db.connect(database="tvprogram")
cu = c.cursor()

try:
  cu.execute("""
    CREATE TABLE tv (
      tvdate DATE,
      tvweekday INTEGER,
      tvchannel VARCHAR(30),
      tvtime1 TIME,
      tvtime2 TIME,
      prname VARCHAR(150),
      prgenre VARCHAR(40)
    );
""")
except db.DatabaseError, x:
  print "Ошибка: ", x
c.commit()

try:
  cu.execute("""
    CREATE TABLE wd (
      weekday INTEGER,
      wdname VARCHAR(11)
    );
""")
except db.DatabaseError, x:
  print "Ошибка: ", x
c.commit()
c.close()
```

Здесь просто исполняются **SQL-**инструкции, и обрабатывается ошибка базы данных, если таковая случится **(**например, при попытке создать таблицу с уже существующим именем**).** Для того чтобы таблицы создавались независимо, используется commit()**.**

Кстати, удалить таблицы из базы данных можно следующим образом:

```
import sqlite as db

c = db.connect(database="tvprogram")
cu = c.cursor()

try:
  cu.execute("""DROP TABLE tv;""")
except db.DatabaseError, x:
  print "Ошибка: ", x
c.commit()

try:
  cu.execute("""DROP TABLE wd;""")
except db.DatabaseError, x:
  print "Ошибка: ", x
c.commit()
c.close()
```

Наполнение базы данных

Теперь можно наполнить таблицы значениями. Следует начать с расшифровки числовых значений для дней недели:

```
weekdays = ["Воскресенье", "Понедельник", "Вторник", "Среда",
            "Четверг", "Пятница", "Суббота", "Воскресенье"]

import sqlite as db

c = db.connect(database="tvprogram")
cu = c.cursor()
cu.execute("""DELETE FROM wd;""")
cu.executemany("""INSERT INTO wd VALUES (%s, %s);""",
           enumerate(weekdays))
c.commit()
c.close()
```

Стоит напомнить, что встроенная функция `enumerate()` создает список пар номер-значение, например:

```
>>> print [i for i in enumerate(['a', 'b', 'c'])]
[(0, 'a'), (1, 'b'), (2, 'c')]
```

Из приведенного примера ясно, что метод `executemany()` объекта-курсора использует второй параметр - последовательность - для массового ввода данных с помощью **SQL**-инструкции `INSERT`.

Предположим, что телепрограмма задана в файле **tv.csv** в формате **CSV** (он уже обсуждался):

```
10.02.2003 9.00|ОРТ|Новости|Новости|9.15
10.02.2003 9.15|ОРТ|"НЕЖНЫЙ ЯД"|Сериал|10.15
10.02.2003 10.15|ОРТ|"Маски-шоу"|Юмористическая программа|10.45
10.02.2003 10.45|ОРТ|"Человек и закон"||11.30
10.02.2003 11.30|ОРТ|"НОВЫЕ ПРИКЛЮЧЕНИЯ СИНДБАДА"|Сериал|12.00
```

Следующая программа разбирает **CSV**-файл и записывает данные в таблицу `tv`:

```
import calendar, csv
import sqlite as db
from sqlite.main import Time, Date  ## Только для
db.Date, db.Time = Date, Time       ## sqlite

c = db.connect(database="tvprogram")
cu = c.cursor()

input_file = open("tv.csv", "rb")
rdr = csv.DictReader(input_file,
          fieldnames=['begt', 'channel', 'prname', 'prgenre',
 'endt'])
for rec in rdr:
    bd, bt = rec['begt'].split()
    bdd, bdm, bdy = map(int, bd.split('.'))
    bth, btm = map(int, bt.split('.'))
    eth, etm = map(int, rec['endt'].split('.'))
    rec['wd'] = calendar.weekday(bdy, bdm, bdd)
```

```
                    rec['begd'] = db.Date(bdy, bdm, bdd)
                    rec['begt'] = db.Time(bth, btm, 0)
                    rec['endt'] = db.Time(eth, etm, 0)

                    cu.execute("""INSERT INTO tv
                    (tvdate, tvweekday, tvchannel, tvtime1, tvtime2, prname,
prgenre)
                    VALUES (
                    %(begd)s, %(wd)s, %(channel)s, %(begt)s, %(endt)s,
                    %(prname)s, %(prgenre)s);""", rec)
                input_file.close()
                c.commit()
```

Большая часть преобразований связана с получением дат и времен **(**приходится разбивать строки на части в соответствии с форматом даты и времени**).** День недели получен с помощью функции из модуля `calendar`**.**

Примечание:

Из-за небольшой ошибки в пакете `sqlite` конструкторы `Date`, `Time` и т.д. не попадают из модуля `sqlite.main` при импорте из `sqlite`, поэтому пришлось добавить две строки, специфичные для `sqlite`, в универсальный "модуль" с именем `db`.

В этом же примере было продемонстрировано использование словаря для вставки значений в таблицу базы данных. Следует заметить, что подстановка выполняется внутри вызова `execute()` в соответствии с типами переданных значений. **SQL-**инструкция **INSERT** была бы некорректной при попытке выполнить подстановку самостоятельно, например, операцией форматирования `%`.

Выборки из базы данных

Базы данных создаются для удобства хранения и извлечения больших объемов. Следующий нехитрый пример позволяет проверить, правильно ли были введены в таблицу дни недели:

```
                import sqlite as db

                c = db.connect(database="tvprogram")
                cu = c.cursor()
                cu.execute("SELECT weekday, wdname FROM wd ORDER BY weekday;")
                for i, n in cu.fetchall():
                  print i, n
```

Если все было сделано правильно, получится:

```
                0 Воскресенье
                1 Понедельник
                2 Вторник
                3 Среда
                4 Четверг
                5 Пятница
                6 Суббота
                7 Воскресенье
```

Несложно догадаться, как сделать выборку телепрограммы:

```
import sqlite as db

c = db.connect(database="tvprogram")
cu = c.cursor()
cu.execute("""
SELECT tvdate, tvtime1, wd.wdname, tvchannel, prname, prgenre
FROM tv, wd
WHERE wd.weekday = tvweekday
ORDER BY tvdate, tvtime1;""")
for rec in cu.fetchall():
  dt = rec[0] + rec[1]
  weekday = rec[2]
  channel = rec[3]
  name = rec[4]
  genre = rec[5]
  print "%s, %02i.%02i.%04i %s %02i:%02i %s (%s)" % (
         weekday, dt.day, dt.month, dt.year, channel,
         dt.hour, dt.minute, name, genre)
```

В этом примере в качестве типа для даты и времени используется тип из `mx.DateTime`. Именно поэтому стало возможным получить год, месяц, день, час и минуту обращением к атрибуту. Кстати, `datetime`-объект стандартного модуля `datetime` имеет те же атрибуты. В общем случае для даты и времени может использоваться другой тип, поэтому если получаемые из базы даты будут проходить более глубокую обработку, их следует переводить во внутреннее представление сразу после получения по запросу. Тем самым тип даты из модуля **DB-API** не будет влиять на другие части программы.

Другие СУБД и Python

Модуль `sqlite` дает прекрасные возможности для построения небольших и быстрых баз данных, однако для полноты изложения предлагается обзор модулей расширения **Python** для других СУБД.

Выше везде импортировался модуль `sqlite`, с изменением его имени на `db`. Это было сделано не случайно. Дело в том, что подобные модули, поддерживающие **DB-API 2.0**, есть и для других СУБД, и даже не в единственном числе. Согласно информации на сайте **www.python.org DB-API 2.0**-совместимые модули для **Python** имеют следующие СУБД или протоколы доступа к БД:

- `zxJDBC` Доступ по **JDBC.**
- `MySQL` Для СУБД **MySQL.**
- `mxODBC` Доступ по **ODBC,** продается фирмой **eGenix (http://www.egenix.com).**
- `DCOracle2, cx_Oracle` Для СУБД **Oracle.**
- `PyGresQL, psycopg, pyPgSQL` Для СУБД **PostgreSQL.**
- `Sybase` Для **Sybase.**
- `sapdbapi` Для СУБД **SAP.**
- `KInterbasDB` Для СУБД **Firebird (**это потомок **Interbase).**
- `PyADO` Адаптер к **Microsoft ActiveX Data Objects (**только под **Windows).**

Примечание:

Для СУБД **PostgreSQL** нужно взять не **PyGreSQL,** а **psycopg,** так как в первом есть небольшие проблемы с типом для даты и времени при вставке параметров в методе `execute()`. Кроме того, `psycopg` оптимизирован для скорости и многопоточности **(**`psycopg.threadsafety=2`**).**

Таким образом, в примерах, используемых в этой лекции, вместо `sqlite` можно применять, например, `psycopg`: результат должен быть тем же, если, конечно, соответствующий модуль был установлен.

Однако в общем случае при переходе с одной СУБД на другую могут возникать нестыковки, даже, несмотря на поддержку одной версии **DB-API**. Например, у модулей могут различаться `paramstyle`. В этом случае придется немного переделать параметры к вызову `execute()`. Могут быть и другие причины, поэтому переход на другую СУБД следует тщательно тестировать.

Иметь интерфейс **DB-API** могут не только базы данных. Например, разработчики проекта `fssdb` стремятся построить **DB-API 2.0** интерфейс к... файловой системе.

Несмотря на достаточно хорошие теоретические основы и стабильные реализации, реляционная модель - не единственная из успешно используемых сегодня. К примеру, уже рассматривался язык **XML** и интерфейсы для работы с ним в **Python**. Древовидная модель данных **XML** для многих задач является более естественной, и в настоящее время идут исследования, результаты которых позволят работать с **XML** так же легко и стабильно, как с реляционными СУБД. Язык программирования **Python** - один из полигонов этих исследований.

Решая конкретную задачу, разработчик программного обеспечения должен сделать выбор средств, наиболее подходящих для решения задачи. Очень многие подходят к этому выбору с предвзятостью, выбирая неоптимальную (для данной задачи или подзадачи) модель данных. В результате данные, которые по своей природе легче представить другой моделью, приходится хранить и обрабатывать в выбранной модели, зачастую невольно моделируя более естественные структуры доступа и хранения. Так, **XML** можно хранить в реляционной БД, а табличные данные - в **XML**, однако это неестественно. Из-за этого сложность и подверженность ошибкам программного продукта возрастают, даже если использованные инструменты высокого качества.

Заключение

В рамках данной лекции были рассмотрены возможности связи **Python** с системами управления реляционными базами данных. Для **Python** разработан стандарт, называемый **DB-API** (версия **2.0**), которого должны придерживаться все разработчики модулей сопряжения с реляционными базами данных. Благодаря этому **API** код прикладной программы становится менее зависимым от марки используемой базы данных, его могут понять разработчики, использующие другие базы данных. Фактически **DB-API 2.0** описывает имена функций и классов, которые должен содержать модуль сопряжения с базой данных, и их семантику. Модуль сопряжения должен содержать класс объектов-соединений с базой данных и класс для курсоров - специальных объектов, через которые происходит коммуникация с СУБД на прикладном уровне.

Здесь была использована СУБД **SQLite** и соответствующий модуль расширения **Python** для сопряжения с этой СУБД - `sqlite`, так как он поддерживает **DB-API 2.0** и достаточно прост в установке. С его помощью были продемонстрированы основные приемы работы с базой данных: создание и наполнение таблиц, выполнение выборок и анализ полученных данных.

В конце лекции дан список других пакетов и модулей, которые позволяют **Python**-программе работать со многими современными СУБД.

Ссылки

Модули **mxDateTime** и др. **http://www.lemburg.com/files/python/**

СУБД **SQLite** **http://sqlite.org**

Модуль сопряжения с **SQLite** **http://pysqlite.org**

Лекция #11: Многопоточные вычисления

О потоках управления

В современной операционной системе, даже не выполняющей ничего особенного, могут одновременно работать несколько **процессов (processes)**. Например, при запуске программы запускается новый процесс. Функции для управления процессами можно найти в стандартном модуле `os` языка **Python**. Здесь же речь пойдет о потоках.

Потоки управления (threads) образуются и работают в рамках одного процесса. В однопоточном приложении (программе, которая не использует дополнительных потоков) имеется только один поток управления. Говоря упрощенно, при запуске программы этот поток последовательно исполняет встречаемые в программе операторы, направляясь по одной из альтернативных ветвей оператора выбора, проходит через тело цикла нужное число раз, выбирается к месту обработки исключения при возбуждении исключения. В любой момент времени интерпретатор **Python** знает, какую команду исполнить следующей. После исполнения команды становится известно, какой команде передать управление. Эта ниточка непрерывна в ходе выполнения программы и обрывается только по ее завершении.

Теперь можно представить себе, что в некоторой точке программы ниточка раздваивается, и каждый поток идет своим путем. Каждый из образовавшихся потоков может в дальнейшем еще несколько раз раздваиваться. (При этом один из потоков всегда остается главным, и его завершение означает завершение всей программы.) В каждый момент времени интерпретатор знает, какую команду какой поток должен выполнить, и уделяет кванты времени каждому потоку. Такое, казалось бы, незначительное усложнение механизма выполнения программы на самом деле требует качественных изменений в программе - ведь деятельность потоков должна быть согласована. Нельзя допускать, чтобы потоки одновременно изменяли один и тот же объект, результат такого изменения, скорее всего, нарушит целостность объекта.

Одним из классических средств согласования потоков являются объекты, называемые **семафорами**. Семафоры не допускают выполнения некоторого участка кода несколькими потоками одновременно. Самый простой семафор - **замок (lock)** или **mutex** (от английского **mutually exclusive**, взаимоисключающий). Для того чтобы поток мог продолжить выполнение кода, он должен сначала захватить замок. После захвата замка поток выполняет определенный участок кода и потом освобождает замок, чтобы другой поток мог его получить и пройти дальше к выполнению охраняемого замком участку программы. Поток, столкнувшись с занятым другим потоком замком, обычно ждет его освобождения.

Поддержка многопоточности в языке **Python** доступна через использование ряда модулей. В стандартном модуле `threading` определены нужные для разработки многопоточной **(multithreading)** программы классы: несколько видов семафоров (классы замков `Lock`, `RLock` и класс `Semaphore`) и другие механизмы взаимодействия между потоками (классы `Event` и `Condition`), класс `Timer` для запуска функции по прошествии некоторого времени. Модуль `Queue` реализует очередь, которой могут пользоваться сразу несколько потоков. Для создания и (низкоуровневого) управления потоками в стандартном модуле `thread` определен класс `Thread`.

Пример многопоточной программы

В следующем примере создается два дополнительных потока, которые выводят на стандартный вывод каждый свое:

```
import threading

def proc(n):
  print "Процесс", n
```

```
p1 = threading.Thread(target=proc, name="t1", args=["1"])
p2 = threading.Thread(target=proc, name="t2", args=["2"])
p1.start()
p2.start()
```

Сначала получается два объекта класса `Thread`, которые затем и запускаются с различными аргументами. В данном случае в потоках работает одна и та же функция `proc()`, которой передается один аргумент, заданный в именованном параметре `args` конструктора класса `Thread`. Нетрудно догадаться, что метод `start()` служит для запуска нового потока. Таким образом, в приведенном примере работают три потока: основной и два дополнительных (с именами `"t1"` и `"t2"`).

Функции модуля threading

В модуле `threading`, который здесь используется, есть функции, позволяющие получить информацию о потоках:

- `activeCount()` Возвращает количество активных в настоящий момент экземпляров класса `Thread`. Фактически, это `len(threading.enumerate())`.
- `currentThread()` Возвращает текущий объект-поток, то есть соответствующий потоку управления, который вызвал эту функцию. Если поток не был создан через модуль `threading`, будет возвращен объект-поток с сокращенной функциональностью **(dummy thread object)**.
- `enumerate()` Возвращает список активных потоков. Завершившиеся и еще не начатые потоки не входят в список.

Класс Thread

Экземпляры класса `threading.Thread` представляют потоки **Python**-программы. Задать действия, которые будут выполняться в потоке, можно двумя способами: передать конструктору класса исполняемый объект и аргументы к нему или путем наследования получить новый класс с переопределенным методом `run()`. Первый способ был рассмотрен в примере выше. Конструктор класса `threading.Thread` имеет следующие аргументы:

```
Thread(group, target, name, args, kwargs)
```

Здесь `group` - группа потоков (пока что не используется, должен быть равен `None`), `target` - объект, который будет вызван в методе `run()`, `name` - имя потока, `args` и `kwargs` - последовательность и словарь позиционных и именованных параметров (соответственно) для вызова заданного в параметре `target` объекта. В примере выше были использованы только позиционные параметры, но то же самое можно было выполнить и с применением именованных параметров:

```
import threading

def proc(n):
  print "Процесс", n

p1 = threading.Thread(target=proc, name="t1", kwargs={"n": "1"})
p2 = threading.Thread(target=proc, name="t2", kwargs={"n": "2"})
p1.start()
p2.start()
```

То же самое можно проделать через наследование от класса threading.Thread с определением собственного конструктора и метода run():

```
import threading

class T(threading.Thread):
  def __init__(self, n):
    threading.Thread.__init__(self, name="t" + n)
    self.n = n
  def run(self):
    print "Процесс", self.n

p1 = T("1")
p2 = T("2")
p1.start()
p2.start()
```

Самое первое, что необходимо сделать в конструкторе - вызвать конструктор базового класса. Как и раньше, для запуска потока нужно выполнить метод start() объекта-потока, что приведет к выполнению действий в методе run().

Жизнью потоков можно управлять вызовом методов:

- start()Дает потоку жизнь.
- run()Этот метод представляет действия, которые должны быть выполнены в потоке.
- join([timeout])Поток, который вызывает этот метод, приостанавливается, ожидая завершения потока, чей метод вызван. Параметр timeout (число с плавающей точкой) позволяет указать время ожидания (в секундах), по истечении которого приостановленный поток продолжает свою работу независимо от завершения потока, чей метод join был вызван. Вызывать join() некоторого потока можно много раз. Поток не может вызвать метод join() самого себя. Также нельзя ожидать завершения еще не запущенного потока. Слово **"join"** в переводе с английского означает "присоединить", то есть, метод, вызвавший join(), желает, чтобы поток по завершении присоединился к вызывающему метод потоку.
- getName()Возвращает имя потока. Для главного потока это "MainThread".
- setName(name)Присваивает потоку имя name.
- isAlive()Возвращает истину, если поток работает (метод run() уже вызван, но еще не завершился).
- isDaemon()Возвращает истину, если поток имеет признак демона. Программа на **Python** завершается по завершении всех потоков, не являющихся демонами. Главный поток демоном не является.
- setDaemon(daemonic)Устанавливает признак daemonic того, что поток является демоном. Начальное значение этого признака заимствуется у потока, запустившего данный. Признак можно изменять только для потоков, которые еще не запущены.

В модуле Thread пока что не реализованы возможности, присущие потокам в **Java** (определение групп потоков, приостановка и прерывание потоков извне, приоритеты и некоторые другие вещи), однако они, скорее всего, будут созданы в недалеком будущем.

Таймер

Класс threading.Timer представляет действие, которое должно быть выполнено через заданное время. Этот класс является подклассом класса threading.Thread, поэтому запускается также методом start(). Следующий простой пример, печатающий на стандартном выводе Hello, world! поясняет сказанное:

```
def hello():
  print "Hello, world!"
```

```
t = Timer(30.0, hello)
t.start()
```

Замки

Простейший замок может быть реализован на основе класса `Lock` модуля `threading`. Замок имеет два состояния: он может быть или открыт, или заперт. В последнем случае им владеет некоторый поток. Объект класса `Lock` имеет следующие методы:

- `acquire([blocking=True])`Делает запрос на запирание замка. Если параметр `blocking` не указан или является истиной, то поток будет ожидать освобождения замка. Если параметр не был задан, метод не возвратит значения. Если `blocking` был задан и истинен, метод возвратит `True` (после успешного овладения замком). Если блокировка не требуется (то есть задан `blocking=False`), метод вернет `True`, если замок не был заперт и им успешно овладел данный поток. В противном случае будет возвращено `False`.
- `release()`Запрос на отпирание замка.
- `locked()`Возвращает текущее состояние замка (`True` - заперт, `False` - открыт). Следует иметь в виду, что даже если состояние замка только что проверено, это не означает, что он сохранит это состояние до следующей команды.

Имеется еще один вариант замка - `threading.RLock`, который отличается от `threading.Lock` тем, что некоторый поток может запрашивать его запирание много раз. Отпирание такого замка должно происходить столько же раз, сколько было запираний. Это может быть полезно, например, внутри рекурсивных функций.

Когда нужны замки?

Замки позволяют ограничивать вход в некоторую область программы одним потоком. Замки могут потребоваться для обеспечения целостности структуры данных. Например, если для корректной работы программы требуется добавление определенного элемента сразу в несколько списков или словарей, такие операции в многопоточном приложении следует обставить замками. Вокруг атомарных операций над встроенными типами (операций, которые не вызывают исполнение какого-то другого кода на **Python**) замки ставить необязательно. Например, метод `append()` (встроенного) списка является атомарной операцией, а тот же метод, реализованный пользовательским классом, может требовать блокировок. В случае сомнений, конечно, лучше перестраховаться и поставить замки, однако следует минимизировать общее время действия замка, так как замок останавливает другие потоки, пытающиеся попасть в ту же область программы. Отсутствие замка в критической части программы, работающей над общими для двух и более потоков ресурсами, может привести к случайным, трудноуловимым ошибкам.

Тупиковая ситуация (deadlock)

Замки применяются для управления доступом к ресурсу, который нельзя использовать совместно. В программе таких ресурсов может быть несколько. При работе с замками важно хорошо продумать, не зайдет ли выполнение программы в **тупик (deadlock)** из-за того, что двум потокам потребуются одни и те же ресурсы, но ни тот, ни другой не смогут их получить, так как они уже получили замки. Такая ситуация проиллюстрирована в следующем примере:

```
import threading, time

resource = {'A': threading.Lock(), 'B': threading.Lock()}

def proc(n, rs):
 for r in rs:
   print "Процесс %s запрашивает ресурс %s" % (n, r)
```

```
    resource[r].acquire()
    print "Процесс %s получил ресурс %s" % (n, r)
    time.sleep(1)
  print "Процесс %s выполняется" % n
  for r in rs:
    resource[r].release()
  print "Процесс %s закончил выполнение" % n

p1 = threading.Thread(target=proc, name="t1", args=["1", "AB"])
p2 = threading.Thread(target=proc, name="t2", args=["2", "BA"])
p1.start()
p2.start()
p1.join()
p2.join()
```

В этом примере два потока **(t1** и **t2)** запрашивают замки к одним и тем же ресурсам **(A** и **B)**, но в разном порядке, отчего получается, что ни у того, ни у другого не хватает ресурсов для дальнейшей работы, и они оба безнадежно повисают, ожидая освобождения нужного ресурса. Благодаря операторам `print` можно увидеть последовательность событий:

```
Процесс 1 запрашивает ресурс A
Процесс 1 получил ресурс A
Процесс 2 запрашивает ресурс B
Процесс 2 получил ресурс B
Процесс 1 запрашивает ресурс B
Процесс 2 запрашивает ресурс A
```

Существуют методики, позволяющие избежать подобных тупиков, однако их рассмотрение не входит в рамки данной лекции. Можно посоветовать следующие приемы:

- построить логику приложения так, чтобы никогда не запрашивать замки к двум ресурсам сразу. Возможно, придется определить составной ресурс. В частности, к данному примеру можно было бы определить замок **"AB"** для указания эксклюзивного доступа к ресурсам **A** и **B**.
- строго упорядочить все ресурсы **(**например, по цене**)** и всегда запрашивать их в определенном порядке **(**скажем, начиная с более дорогих ресурсов**)**. При этом перед заказом некоторого ресурса поток должен отказаться от заблокированных им более дешевых ресурсов.

Семафоры

Семафоры (их иногда называют семафорами Дийкстры **(Dijkstra)** по имени их изобретателя**)** являются более общим механизмом синхронизации потоков, нежели замки. Семафоры могут допустить в критическую область программы сразу несколько потоков. Семафор имеет счетчик запросов, уменьшающийся с каждым вызовом метода `acquire()` и увеличивающийся при каждом вызове `release()`. Счетчик не может стать меньше нуля, поэтому в таком состоянии потокам приходится ждать, как и в случае с замками, пока значение счетчика не увеличится.

Конструктор класса `threading.Semaphore` принимает в качестве **(**необязательного**)** аргумента начальное состояние счетчика **(**по умолчанию оно равно **1**, что соответствует замку класса `Lock`**)**. Методы `acquire()` и `release()` действуют аналогично описанным выше одноименным методам у замков.

Семафор может применяться для охраны ограниченного ресурса. Например, с его помощью можно вести **пул** соединений с базой данных. Пример такого использования семафора **(**заимствован из документации к **Python)** дан ниже:

```
from threading import BoundedSemaphore
maxconnections = 5
# Подготовка семафора
pool_sema = BoundedSemaphore(value=maxconnections)

# Внутри потока:

pool_sema.acquire()
conn = connectdb()
# ... использование соединения ...
conn.close()
pool_sema.release()
```

Таким образом, применяется не более пяти соединений с базой данных. В примере использован класс `threading.BoundedSemaphore`. Экземпляры этого класса отличаются от экземпляров класса `threading.Semaphore` тем, что не дают сделать `release()` больше, чем сделан `acquire()`.

События

Еще одним способом коммуникации между объектами являются **события**. Экземпляры класса `threading.Event` могут быть использованы для передачи информации о наступлении некоторого события от одного потока одному или нескольким другим потокам. Объекты-события имеют внутренний флаг, который может находиться в установленном или сброшенном состоянии. При своем создании флаг события находится в сброшенном состоянии. Если флаг в установленном состоянии, ожидания не происходит: поток, вызвавший метод `wait()` для ожидания события, просто продолжает свою работу. Ниже приведены методы экземпляров класса `threading.Event`:

- `set()`Устанавливает внутренний флаг, сигнализирующий о наступлении события. Все ждущие данного события потоки выходят из состояния ожидания.
- `clear()`Сбрасывает флаг. Все события, которые вызывают метод `wait()` этого объекта-события, будут находиться в состоянии ожидания до тех пор, пока флаг сброшен, или по истечении заданного таймаута.
- `isSet()`Возвращает состояние флага.
- `wait([timeout])`Переводит поток в состояние ожидания, если флаг сброшен, и сразу возвращается, если флаг установлен. Аргумент `timeout` задает таймаут в секундах, по истечении которого ожидание прекращается, даже если событие не наступило.

Составить пример работы с событиями предлагается в качестве упражнения.

Условия

Более сложным механизмом коммуникации между потоками является механизм условий. Условия представляются в виде экземпляров класса `threading.Condition` и, подобно только что рассмотренным событиям, оповещают потоки об изменении некоторого состояния. Конструктор класса `threading.Condition` принимает необязательный параметр, задающий замок класса `threading.Lock` или `threading.RLock`. По умолчанию создается новый экземпляр замка класса `threading.RLock`. Методы объекта-условия описаны ниже:

- `acquire(...)`Запрашивает замок. Фактически вызывается одноименный метод принадлежащего объекту-условию объекта-замка.
- `release()`Снимает замок.
- `wait([timeout])`Переводит поток в режим ожидания. Этот метод может быть вызван только в том случае, если вызывающий его поток получил замок. Метод снимает замок и блокирует поток до появления объявлений, то есть вызовов методов

notify() и notifyAll() другими потоками. Необязательный аргумент timeout задает таймаут ожидания в секундах. При выходе из ожидания поток снова запрашивает замок и возвращается из метода wait().

- notify()Выводит из режима ожидания один из потоков, ожидающих данные условия. Метод можно вызвать, только овладев замком, ассоциированным с условием. Документация предупреждает, что в будущих реализациях модуля из целей оптимизации этот метод будет прерывать ожидание сразу нескольких потоков. Сам по себе метод notify() не приводит к продолжению выполнения ожидавших условия потоков, так как этому препятствует занятый замок. Потоки получают управление только после снятия замка потоком, вызвавшим метод notify().
- notifyAll()Этот метод аналогичен методу notify(), но прерывает ожидание всех ждущих выполнения условия потоков.

В следующем примере условия используются для оповещения потоков о прибытии новой порции данных (организуется связь производитель - потребитель, **producer - consumer):**

```
import threading

cv = threading.Condition()

class Item:
  """Класс-контейнер для элементов, которые будут потребляться
  в потоках"""
  def __init__(self):
    self._items = []
  def is_available(self):
    return len(self._items) > 0
  def get(self):
    return self._items.pop()
  def make(self, i):
    self._items.append(i)

item = Item()

def consume():
  """Потребление очередного элемента (с ожиданием его появления)"""
  cv.acquire()
  while not item.is_available():
    cv.wait()
  it = item.get()
  cv.release()
  return it

def consumer():
  while True:
    print consume()

def produce(i):
  """Занесение нового элемента в контейнер и оповещение потоков"""
  cv.acquire()
  item.make(i)
  cv.notify()
  cv.release()

p1 = threading.Thread(target=consumer, name="t1")
p1.setDaemon(True)
p2 = threading.Thread(target=consumer, name="t2")
p2.setDaemon(True)
p1.start()
p2.start()
produce("ITEM1")
produce("ITEM2")
produce("ITEM3")
produce("ITEM4")
```

```
p1.join()
p2.join()
```

В этом примере условие cv отражает наличие необработанных элементов в контейнере item. Функция produce() "производит" элементы, а consume(), работающая внутри потоков, "потребляет". Стоит отметить, что в приведенном виде программа никогда не закончится, так как имеет бесконечный цикл в потоках, а в главном потоке - ожидание завершения этих потоков. Еще одна особенность - признак демона, установленный с помощью метода setDaemon() объекта-потока до его старта.

Очередь

Процесс, показанный в предыдущем примере, имеет значение, достойное отдельного модуля. Такой модуль в стандартной библиотеке языка **Python** есть, и он называется Queue.

Помимо исключений - Queue.Full (очередь переполнена) и Queue.Empty (очередь пуста) - модуль определяет класс Queue, заведующий собственно очередью.

Собственно, здесь можно привести аналог примера выше, но уже с использованием класса Queue.Queue:

```
import threading, Queue

item = Queue.Queue()

def consume():
  """Потребление очередного элемента (с ожиданием его появления)"""
  return item.get()

def consumer():
  while True:
    print consume()

def produce(i):
  """Занесение нового элемента в контейнер и оповещение потоков"""
  item.put(i)

p1 = threading.Thread(target=consumer, name="t1")
p1.setDaemon(True)
p2 = threading.Thread(target=consumer, name="t2")
p2.setDaemon(True)
p1.start()
p2.start()
produce("ITEM1")
produce("ITEM2")
produce("ITEM3")
produce("ITEM4")
p1.join()
p2.join()
```

Следует отметить, что все блокировки спрятаны в реализации очереди, поэтому в коде они явным образом не присутствуют.

Модуль thread

По сравнению с модулем threading, модуль thread предоставляет низкоуровневый доступ к потокам. Многие функции модуля threading, который рассматривался до этого,

реализованы на базе модуля thread. Здесь стоит сделать некоторые замечания по применению потоков вообще. Документация по **Python** предупреждает, что использование потоков имеет особенности:

- Исключение KeyboardInterrupt (прерывание от клавиатуры) может быть получено любым из потоков, если в поставке **Python** нет модуля signal (для обработки сигналов).
- Не все встроенные функции, блокированные ожиданием ввода, позволяют другим потокам работать. Правда, основные функции вроде time.sleep(), select.select(), метод read() файловых объектов не блокируют другие потоки.
- Невозможно прервать метод acquire(), так как исключение KeyboardInterrupt возбуждается только после возврата из этого метода.
- Нежелательно, чтобы главный поток завершался раньше других потоков, так как не будут выполнены необходимые деструкторы и даже части finally в операторах try-finally. Это связано с тем, что почти все операционные системы завершают приложение, у которого завершился главный поток.

Визуализация работы потоков

Следующий пример иллюстрирует параллельность выполнения потоков, используя возможности библиотеки графических примитивов Tkinter (она входит в стандартную поставку **Python**). Несколько потоков наперегонки увеличивают размеры прямоугольника некоторого цвета. Цветом победившего потока окрашивается кнопка **Go**:

```
            import threading, time, sys
            from Tkinter import Tk, Canvas, Button, LEFT, RIGHT, NORMAL,
DISABLED

            global champion

            # Задается дистанция, цвет полосок и другие параметры
            distance = 300
            colors =
["Red","Orange","Yellow","Green","Blue","DarkBlue","Violet"]
            nrunners = len(colors)      # количество дополнительных потоков
            positions = [0] * nrunners  # список текущих позиций
            h, h2 = 20, 10              # параметры высоты полосок

            def run(n):
              """Программа бега n-го участника (потока)"""
              global champion
              while 1:
                for i in range(10000):           # интенсивные вычисления
                  pass
                graph_lock.acquire()
                positions[n] += 1                # передвижение на шаг
                if positions[n] == distance:     # если уже финиш
                  if champion is None:           # и чемпион еще не определен,
                    champion = colors[n]         # назначается чемпион
                  graph_lock.release()
                  break
                graph_lock.release()

            def ready_steady_go():
              """Инициализация начальных позиций и запуск потоков"""
              graph_lock.acquire()
              for i in range(nrunners):
                positions[i] = 0
                threading.Thread(target=run, args=[i,]).start()
              graph_lock.release()

            def update_positions():
```

```
            """Обновление позиций"""
            graph_lock.acquire()
            for n in range(nrunners):
              c.coords(rects[n], 0, n*h, positions[n], n*h+h2)
            tk.update_idletasks()  # прорисовка изменений
            graph_lock.release()

          def quit():
            """Выход из программы"""
            tk.quit()
            sys.exit(0)

          # Прорисовка окна, основы для прямоугольников и самих
прямоугольников,
          # кнопок для пуска и выхода
          tk = Tk()
          tk.title("Соревнование потоков")
          c = Canvas(tk, width=distance, height=nrunners*h, bg="White")
          c.pack()
          rects = [c.create_rectangle(0, i*h, 0, i*h+h2, fill=colors[i])
                   for i in range(nrunners)]
          go_b = Button(text="Go", command=tk.quit)
          go_b.pack(side=LEFT)
          quit_b = Button(text="Quit", command=quit)
          quit_b.pack(side=RIGHT)

          # Замок, регулирующий доступ к функции пакета Tk
          graph_lock = threading.Lock()

          # Цикл проведения соревнований
          while 1:
            go_b.config(state=NORMAL), quit_b.config(state=NORMAL)
            tk.mainloop()             # Ожидание нажатия клавиш
            champion = None
            ready_steady_go()
            go_b.config(state=DISABLED), quit_b.config(state=DISABLED)
            # Главный поток ждет финиша всех участников
            while sum(positions) < distance*nrunners:
              update_positions()
            update_positions()
            go_b.config(bg=champion)     # Кнопка окрашивается в цвет
победителя
            tk.update_idletasks()
```

Примечание:

Эта программа использует некоторые возможности языка **Python 2.3 (**встроенную функцию `sum()` и списковые включения**),** поэтому для ее выполнения нужен **Python** версии не меньше **2.3.**

Заключение

Навыки параллельного программирования необходимы любому профессиональному программисту**.** Одним из вариантов организации **(**псевдо**)** параллельного программирования является многопоточное программирование **(**другой вариант**,** более свойственный **Unix-**системам **-** многопроцессное программирование **-** здесь не рассматривается**).** В обычной **(**однопоточной**)** программе действует всего один поток управления**,** а в многопоточной одновременно могут работать несколько потоков**.**

Параллельное программирование требует тщательной отработки взаимодействия между потоками управления**.** Некоторые участки кода необходимо ограждать от одновременного

использования двумя различными потоками, дабы не нарушить целостность изменяемых структур данных или логику работы с внешними ресурсами. Для ограждения участков кода используются замки и семафоры.

Стандартная библиотека **Python** предоставляет довольно неплохой набор возможностей для многопоточного программирования в модулях `threading` и `thread`, а также некоторые полезные вспомогательные модули (например, `Queue`).

Лекция #12: Создание приложений с графическим интерфейсом пользователя

Обзор графических библиотек

Строить **графический интерфейс пользователя (GUI, Graphical User Interface)** для программ на языке **Python** можно при помощи соответствующих **библиотек компонентов графического интерфейса** или, используя кальку с английского, **библиотек виджетов.**

Следующий список далеко не полон, но отражает многообразие существующих решений:

- `Tkinter` Многоплатформенный пакет имеет хорошее управление расположением компонентов. Интерфейс выглядит одинаково на различных платформах **(Unix, Windows, Macintosh).** Входит в стандартную поставку **Python.** В качестве документации можно использовать руководство **"An Introduction to Tkinter" ("**Введение в **Tkinter"),** написанное Фредриком Лундом**:** **http://www.pythonware.com/library/tkinter/introduction/**
- `wxPython` Построен на многоплатформной библиотеке **wxWidgets (**раньше называлась **wxWindows).** Выглядит родным для всех платформ**,** активно совершенствуется**,** осуществлена поддержка **GL.** Имеется для всех основных платформ. Возможно**,** займет место **Tkinter** в будущих версиях **Python.** Сайт**:** **http://www.wxpython.org/**
- `PyGTK` Набор визуальных компонентов для **GTK+** и **Gnome.** Только для платформы **GTK.**
- `PyQT/PyKDE` Хорошие пакеты для тех**,** кто использует **Qt (**под **UNIX** или **Windows)** или **KDE.**
- `Pythonwin` Построен вокруг **MFC,** поставляется вместе с оболочкой в пакете **win32all;** только для **Windows.**
- `pyFLTK` Аналог **Xforms,** поддержка **OpenGL.** Имеется для платформ **Windows** и **Unix.** Сайт**:** **http://pyfltk.sourceforge.net/**
- `AWT, JFC, Swing` Поставляется вместе с **Jython,** а для **Jython** доступны средства**,** которые использует **Java.** Поддерживает платформу **Java.**
- `anygui` Независимый от нижележащей платформы пакет для построения графического интерфейса для программ на **Python.** Сайт**:** **http://anygui.sourceforge.net/**
- `PythonCard` Построитель графического интерфейса**,** сходный по идеологии с **HyperCard/MetaCard.** Разработан на базе `wxPython`**.** Сайт**:** **http://pythoncard.sourceforge.net/**

Список актуальных ссылок на различные графические библиотеки**,** доступные из **Python,** можно найти по следующему адресу**:** **http://phaseit.net/claird/comp.lang.python/python_GUI.html**

Библиотеки могут быть многоуровневыми. Например**, PythonCard** использует **wxPython,** который**,** скажем**,** на платформе **Linux** базируется на многоплатформной **GUI-**библиотеке **wxWindows,** которая**,** в свою очередь**,** базируется на **GTK+** или на **Motif,** а те **-** тоже используют для вывода **X Window.** Кстати**,** для **Motif** в **Python** имеются свои привязки.

В лекции будет рассматриваться пакет **Tkinter,** который по сути является оберткой для **Tcl/Tk -** известного графического пакета для сценарного языка **Tcl.** На примере этого пакета легко изучить основные принципы построения графического интерфейса пользователя.

О графическом интерфейсе

Почти все современные графические интерфейсы общего назначения строятся по модели **WIMP - Window, Icon, Menu, Pointer (**окно**,** иконка**,** меню**,** указатель**).** Внутри окон рисуются

элементы графического интерфейса, которые для краткости будут называться **виджетами (widget -** штучка**)**. Меню могут располагаться в различных частях окна, но их поведение достаточно однотипно: они служат для выбора действия из набора предопределенных действий. Пользователь графического интерфейса "объясняет" компьютерной программе требуемые действия с помощью указателя. Обычно указателем служит курсор мыши или джойстика, однако есть и другие "указательные" устройства. С помощью иконок графический интерфейс приобретает независимость от языка и в некоторых случаях позволяет быстрее ориентироваться в интерфейсе.

Основной задачей графического интерфейса является упрощение коммуникации между пользователем и компьютером. Об этом следует постоянно помнить при проектировании интерфейса. Применение имеющихся в наличии у программиста **(**или дизайнера**)** средств при создании графического интерфейса нужно свести до минимума, выбирая наиболее удобные пользователю виджеты в каждом конкретном случае. Кроме того, полезно следовать принципу наименьшего удивления: из формы интерфейса должно быть понятно его поведение. Плохо продуманный интерфейс портит ощущения пользователя от программы, даже если за фасадом интерфейса скрывается эффективный алгоритм. Интерфейс должен быть удобен для типичных действий пользователя. Для многих приложений такие действия выделены в отдельные серии экранов, называемые "мастерами" **(wizards)**. Однако если приложение **-** скорее конструктор, из которого пользователь может строить нужные ему решения, типичным действием является именно построение решения. Определить типичные действия не всегда легко, поэтому компромиссом может быть гибрид, в котором есть "мастера" и хорошие возможности для собственных построений. Тем не менее, графический интерфейс не является самым эффективным интерфейсом во всех случаях. Для многих предметных областей решение проще выразить с помощью деклараций на некотором формальном языке или алгоритма на сценарном языке.

Основы Tk

Основная черта любой программы с графическим интерфейсом **- интерактивность**. Программа не просто что-то считает **(**в пакетном режиме**)** от начала своего запуска до конца: ее действия зависят от вмешательства пользователя. Фактически, графическое приложение выполняет бесконечный цикл обработки событий. Программа, реализующая графический интерфейс, **событийно-ориентирована**. Она ждет от интерфейса событий, которые и обрабатывает сообразно своему внутреннему состоянию.

Эти события возникают в элементах графического интерфейса **(**виджетах**)** и обрабатываются прикрепленными к этим виджетам обработчиками. Сами виджеты имеют многочисленные свойства **(**цвет, размер, расположение**)**, выстраиваются в иерархию принадлежности **(**один виджет может быть хозяином другого**)**, имеют методы для доступа к своему состоянию.

Расположением виджетов **(**внутри других виджетов**)** ведают так называемые **менеджеры расположения**. Виджет устанавливается на место по правилам менеджера расположения. Эти правила могут определять не только координаты виджета, но и его размеры. В **Tk** имеются три типа менеджеров расположения: простой упаковщик **(pack)**, сетка **(grid)** и произвольное расположение **(place)**.

Но этого для работы графической программы недостаточно. Дело в том, что некоторые виджеты в графической программе должны быть взаимосвязаны определенным образом. Например, полоска прокрутки может быть взаимосвязана с текстовым виджетом: при использовании полоски текст в виджете должен двигаться, и наоборот, при перемещении по тексту полоска должна показывать текущее положение. Для связи между виджетами в **Tk** используются переменные, через которые виджеты и передают друг другу параметры.

Классы виджетов

Для построения графического интерфейса в библиотеке **Tk** отобраны следующие классы виджетов **(**в алфавитном порядке**)**:

- `Button` **(**Кнопка**)** Простая кнопка для вызова некоторых действий **(**выполнения определенной команды**).**
- `Canvas` **(**Рисунок**)** Основа для вывода графических примитивов**.**
- `Checkbutton` **(**Флажок**)** Кнопка**,** которая умеет переключаться между двумя состояниями при нажатии на нее**.**
- `Entry` **(**Поле ввода**)** Горизонтальное поле**,** в которое можно ввести строку текста**.**
- `Frame` **(**Рамка**)** Виджет**,** который содержит в себе другие визуальные компоненты**.**
- `Label` **(**Надпись**)** Виджет может показывать текст или графическое изображение**.**
- `Listbox` **(**Список**)** Прямоугольная рамка со списком**,** из которого пользователь может выделить один или несколько элементов**.**
- `Menu` **(**Меню**)** Элемент**,** с помощью которого можно создавать всплывающие **(popup)** и ниспадающие **(pulldown)** меню**.**
- `Menubutton` **(**Кнопка**-**меню**)** Кнопка с ниспадающим меню**.**
- `Message` **(**Сообщение**)** Аналогично надписи**,** но позволяет заворачивать длинные строки и менять размер по требованию менеджера расположения**.**
- `Radiobutton` **(**Селекторная кнопка**)** Кнопка для представления одного из альтернативных значений**.** Такие кнопки**,** как правило**,** действует в группе**.** При нажатии на одну из них кнопка группы**,** выбранная ранее**,** "отскакивает"**.**
- `Scale` **(**Шкала**)** Служит для задания числового значения путем перемещения движка в определенном диапазоне**.**
- `Scrollbar` **(**Полоса прокрутки**)** Полоса прокрутки служит для отображения величины прокрутки в других виджетах**.** Может быть как вертикальной**,** так и горизонтальной**.**
- `Text` **(**Форматированный текст**)** Этот прямоугольный виджет позволяет редактировать и форматировать текст с использованием различных стилей**,** внедрять в текст рисунки и даже окна**.**
- `Toplevel` **(**Окно верхнего уровня**)** Показывается как отдельное окно и содержит внутри другие виджеты**.**

Все эти классы не имеют отношений наследования друг с другом **-** они равноправны**.** Этот набор достаточен для построения интерфейса в большинстве случаев**.**

События

В системе современного графического интерфейса имеется возможность отслеживать различные события**,** связанные с клавиатурой и мышью**,** и происходящие на "территории" того или иного виджета**.** В **Tk** события описываются в виде текстовой строки **-** шаблона события**,** состоящего из трех элементов **(**модификаторы**,** тип события и детализация события**).**

Тип события	Содержание события
`Activate`	Активизация окна
`ButtonPress`	Нажатие кнопки мыши
`ButtonRelease`	Отжатие кнопки мыши
`Deactivate`	Деактивация окна
`Destroy`	Закрытие окна
`Enter`	Вхождение курсора в пределы виджета
`FocusIn`	Получение фокуса окном
`FocusOut`	Потеря фокуса окном
`KeyPress`	Нажатие клавиши на клавиатуре
`KeyRelease`	Отжатие клавиши на клавиатуре
`Leave`	Выход курсора за пределы виджета
`Motion`	Движение мыши в пределах виджета
`MouseWheel`	Прокрутка колесика мыши

Reparent	Изменение родителя окна
Visibility	Изменение видимости окна

Примеры описаний событий строками и некоторые названия клавиш приведены ниже:

"<ButtonPress-3>" или просто "<3>" - щелчок правой кнопки мыши (то есть, третьей, если считать на трехкнопочной мыши слева-направо). "<Shift-Double-Button-1>" - двойной щелчок мышью (левой кнопкой) с нажатой кнопкой **Shift**. В качестве модификаторов могут быть использованы следующие (список неполный):

```
Control, Shift, Lock,
Button1-Button5 или B1-B5,
Meta, Alt, Double, Triple.
```

Просто символ обозначает событие - нажатие клавиши. Например, "k" - тоже, что "<KeyPress-k>". Для неалфавитно-цифровых клавиш есть специальные названия:

```
Cancel, BackSpace, Tab, Return, Shift_L, Control_L, Alt_L,
Pause, Caps_Lock, Escape, Prior, Next, End, Home, Left,
Up, Right, Down, Print, Insert, Delete, F1, F2, F3, F4, F5, F6,
F7,
F8, F9, F10, F11, F12, Num_Lock, Scroll_Lock, space, less
```

Здесь <space> обозначает пробел, а <less> - знак меньше. <Left>, <Right>, <Up>, <Down> - стрелки. <Prior>, <Next> - это **PageUp** и **PageDown**. Остальные клавиши более или менее соответствуют надписям на стандартной клавиатуре.

Примечание:

Следует заметить, что Shift_L, в отличие от Shift, нельзя использовать как модификатор.

В конкретной среде комбинации, означающие что-то особенное в системе, могут не дойти до графического приложения. Например, известный всем **Ctrl-Alt-Del**.

Следующая программа позволяет печатать направляемые виджету события, в частности - **keysym**, а также анализировать, как различные клавиши можно представить в шаблоне события:

```
from Tkinter import *
tk = Tk()        # основное окно приложения
txt = Text(tk)   # текстовый виджет, принадлежащий окну tk
txt.pack()       # располагается менеджером pack

# функция обработки события
def event_info(event):
  txt.delete("1.0", END)   # удаляется с начала до конца текста
  for k in dir(event):     # цикл по атрибутам события
    if k[0] != "_":        # берутся только неслужебные
атрибуты
      # готовится описание атрибута события
      ev = "%15s: %s\n" % (k, repr(getattr(event, k)))
      txt.insert(END, ev)  # добавляется в конец текста

# привязывается виджету txt функция event_info для обработки
событий,
# соответствующих шаблону <KeyPress>
txt.bind("<KeyPress>", event_info)
tk.mainloop()    # главный цикл обработки событий
```

При нажатии клавиши **Esc** в окне можно увидеть примерно следующее:

```
           char: '\x1b'
          delta: 9
         height: 0
        keycode: 9
         keysym: 'Escape'
     keysym_num: 65307
            num: 9
     send_event: False
         serial: 159
          state: 0
           time: -1072960858
           type: '2'
         widget: <Tkinter.Text instance at 0x401e268c>
          width: 0
              x: 83
         x_root: 448
              y: 44
         y_root: 306
```

Следует объяснить некоторые из этих атрибутов:

- `char` Нажатый символ **(**для некоторых событий **- ??)**
- `height, width` Высота и ширина**.**
- `focus` Был ли в момент события фокус у окна**?**
- `keycode` Код символа **(**скан**-**код клавиатуры**).**
- `keysym` Символическое имя клавиши**.**
- `serial` Серийный номер события**.** Увеличивается по мере возникновения событий**.**
- `time` Время возникновения события**.** Все время увеличивается**.**
- `widget` Виджет**,** в котором возникло событие**.**
- `x, y` Координаты указателя в виджете во время события**.**
- `x_root, y_root` Координаты указателя на экране во время события**.**

В принципе**,** совсем необязательно**,** чтобы события обрабатывал тот же виджет**,** который их первично принял**.** Например**,** можно перенаправить все события внутри подчиненных виджетов на данный виджет с помощью метода `grab_set()` **(**`grab_release()` освобождает виджет от этой обязанности**).** В **Tk** существуют и другие возможности управления событиями**,** которые можно изучить по документации**.**

Создание и конфигурирование виджета

Создание виджета происходит вызовом конструктора соответствующего класса**.** Вызов конструктора имеет следующий синтаксис:

```
Widget([master[, option=value, ...]])
```

Здесь `Widget` **-** класс виджета**,** `master` **-** виджет**-**хозяин**,** `option` и `value` **-** конфигурационная опция и ее значение **(**таких пар может быть несколько**).**

Каждый виджет имеет свойства**,** которые можно устанавливать **(**конфигурировать**)** с помощью методов `config()` **(**или `configure()`**)** и читать с помощью методов**,** подобных методам работы со словарями**.** Ниже приведен возможный синтаксис для работы со свойствами**:**

```
widget.config(option=value, ...)
widget["option"] = value
value = widget["option"]
widget.keys()
```

В случае, когда имя свойства совпадает с ключевым словом языка **Python**, принято использовать после имени одиночное подчеркивание. Так, свойство `class` нужно задавать как `class_`, а `to` как `to_`.

Изменять конфигурацию виджета можно в любой момент. Это изменение прорисуется на экране по возвращении в цикл обработки событий или при явном вызове `update_idletasks()`.

Следующий пример показывает окно с двумя виджетами внутри - полем ввода и надписью. С помощью переменной надпись напрямую связана с полем ввода. Этот пример нарочно использует очень много свойств, чтобы продемонстрировать возможности по конфигурированию:

```
from Tkinter import *
tk = Tk()
tv = StringVar()
Label(tk,
      textvariable=tv,
      relief="groove",
      borderwidth=3,
      font=("Courier", 20, "bold"),
      justify=LEFT,
      width=50,
      padx=10,
      pady=20,
      takefocus=False,
      ).pack()
Entry(tk,
      textvariable=tv,
      takefocus=True,
      ).pack()
tv.set("123")
tk.mainloop()
```

В результате на экране можно увидеть:

Виджеты конфигурируются прямо при создании. Более того, виджеты не связываются с именами, их только располагают внутри виджета-окна. В данном примере использованы свойства `textvariable` (текстовая переменная), `relief` (рельеф), `borderwidth` (ширина границы), `justify` (выравнивание), `width` (ширина, в знакоместах), `padx` и `pady` (прослойка в пикселях между содержимым и границами виджета), `takefocus` (возможность принять фокус при нажатии клавиши **Tab**), `font` (шрифт, один из способов его задания). Эти свойства достаточно типичны для многих виджетов, хотя иногда единицы измерения могут отличаться, например, для виджета `Canvas` ширина задается в пикселях, а не в знакоместах.

В следующем примере демонстрируются возможности по назначению цветов фону, переднему плану (тексту), выделению виджета (подсветка границы) в активном состоянии и при отсутствии фокуса:

```
from Tkinter import *
tk = Tk()
tv = StringVar()
Entry(tk,
      textvariable=tv,
      takefocus=True,
      borderwidth=10,
      ).pack()
mycolor1 = "#%02X%02X%02X" % (200, 200, 20)
Entry(tk,
      textvariable=tv,
      takefocus=True,
      borderwidth=10,
      foreground=mycolor1,              # fg, текст виджета
      background="#0000FF",             # bg, фон виджета
      highlightcolor='green',           # подсветка при фокусе
      highlightbackground='red',        # подсветка без фокуса
      ).pack()
tv.set("123")
tk.mainloop()
```

При желании можно задать стилевые опции для всех виджетов сразу: с помощью метода `tk_setPalette()`. Помимо использованных выше свойств в этом методе можно использовать `selectForeground` и `selectBackground` (передний план и фон выделения), `selectColor` (цвет в выбранном состоянии, например, у `Checkbutton`), `insertBackground` (цвет точки вставки) и некоторые другие.

Примечание:

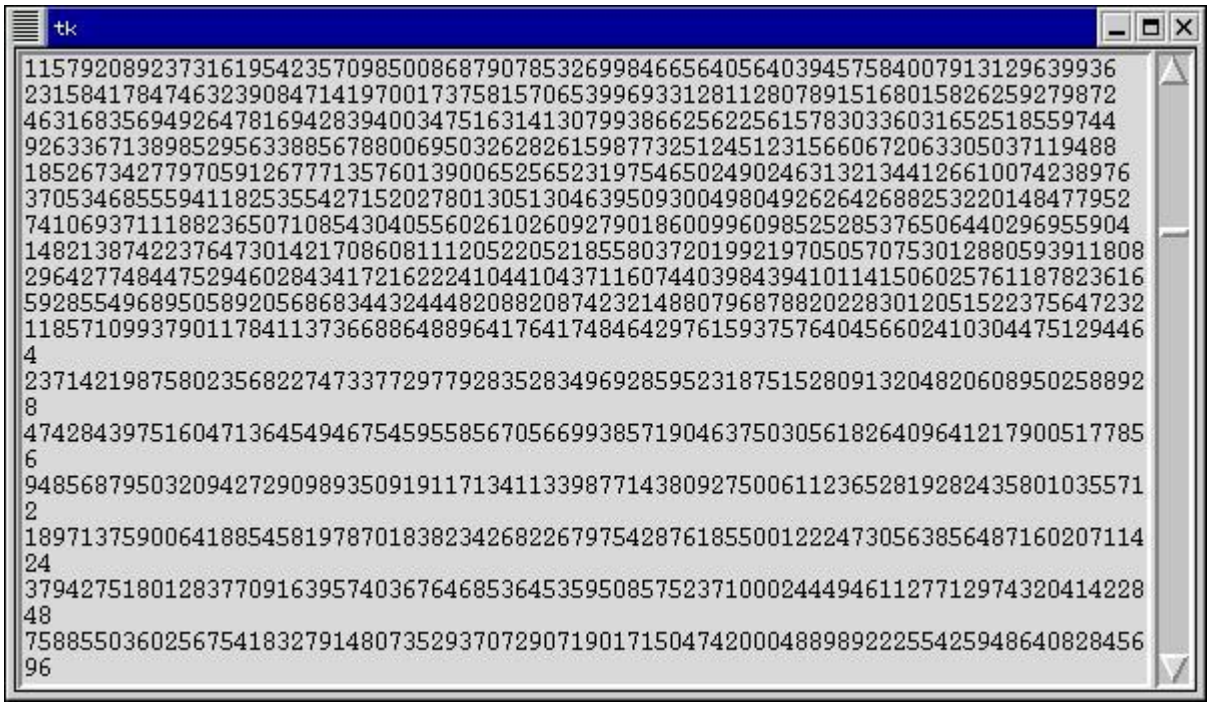
Получить значение из поля ввода можно и при помощи метода `get()`. Например, если назвать объект класса `Entry` именем `e`, получить значение можно так: `e.get()`. Правда, этот метод не обладает той же гибкостью, что метод `get()` экземпляров класса для форматированного текста `Text`: можно взять только все значение целиком.

Виджет форматированного текста

Для того чтобы показать работу с нетривиальным виджетом, можно взять виджет `ScrolledText` из одноименного модуля **Python**. Этот виджет аналогичен рамке с форматированным текстом и вертикальной полосой прокрутки:

```
from Tkinter import *
from ScrolledText import ScrolledText

tk = Tk()                   # окно верхнего уровня
txt = ScrolledText(tk)      # виджет текста с прокруткой
txt.pack()                  # виджет размещается
```

```
for x in range(1, 1024):    # виджет наполняется текстовым
содержимым
  txt.insert(END, str(2L**x)+"\n")

tk.mainloop()
```



Теперь следует рассмотреть методы и свойства виджета с форматированным текстом более подробно.

Для навигации в тексте в **Tk** предусмотрены специальные индексы. Индексы вроде `1.0` и `END` уже встречались - это начало текста (первая строка, нулевой символ) и его конец. (В **Tk** строки нумеруются с единицы, а символы строки - с нуля). Более полный список индексов:

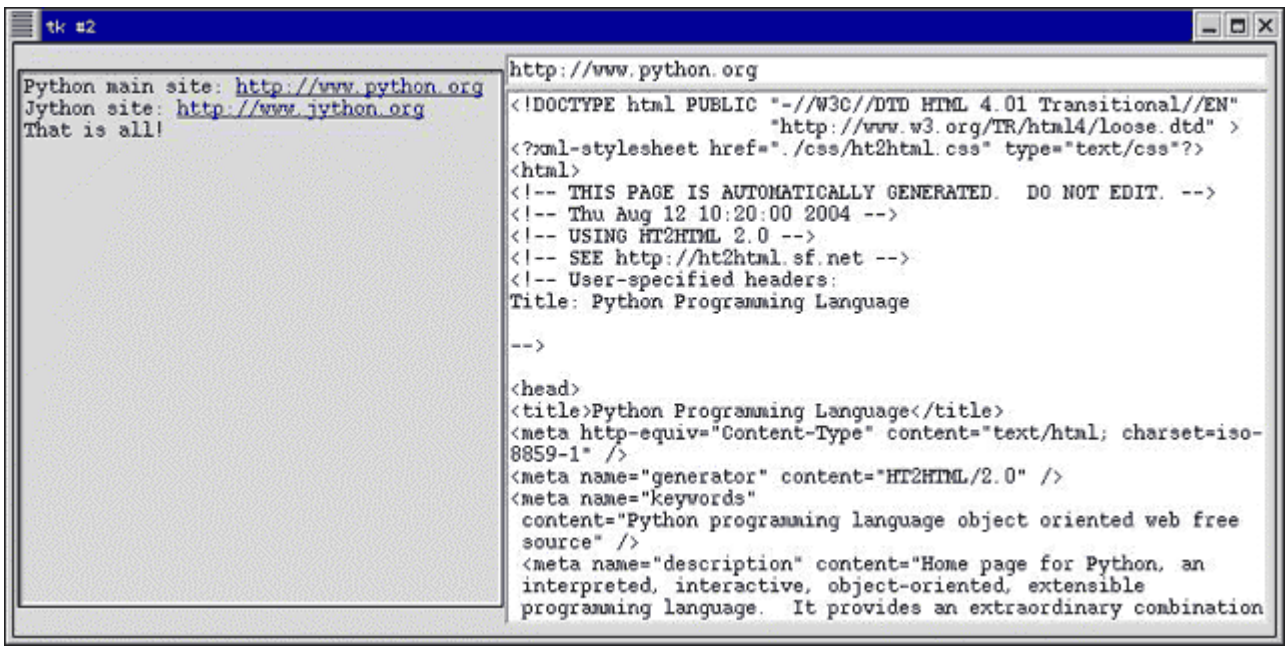
- `L.C` Здесь `L` - номер строки, а `C` - номер символа в строке.
- `INSERT` Точка вставки.
- `CURRENT` Символ, ближайший к курсору мыши.
- `END` Позиция сразу за последним символом в тексте
- `M.first, M.last` Индексы начала и конца помеченного тегом **M** участка текста.
- `SEL_FIRST, SEL_LAST` Индексы начала и конца выделенного текста.
- `M` Пользователь может определять свои именованные позиции в тексте (аналогично `END`, `INSERT` или `CURRENT`). При редактировании текста маркеры будут сдвигаться с заданными для них правилами.
- `@x,y` Символ текста, ближайший к точке с координатами `x`, `y`.

Следующий пример показывает, как снабдить форматированный текст гипертекстовыми возможностями:

```
from Tkinter import *
import urllib
tk = Tk()
txt = Text(tk, width=64)  # поле с текстом
txt.grid(row=0, column=0, rowspan=2)
addr=Text(tk, background="White", width=64, height=1)  # поле
адреса
addr.grid(row=0, column=1)
```

```
                page=Text(tk, background="White", width=64)  # поле с html-
кодом
                page.grid(row=1, column=1)

                def fetch_url(event):
                  click_point = "@%s,%s" % (event.x, event.y)
                  trs = txt.tag_ranges("href")  # список областей текста,
отмеченных как href
                  url = ""
                  # определяется, на какой участок пришелся щелчок мыши, и
берется
                  # соответствующий ему URL
                  for i in range(0, len(trs), 2):
                    if txt.compare(trs[i], "<=", click_point) and \
                       txt.compare(click_point, "<=", trs[i+1]):
                      url = txt.get(trs[i], trs[i+1])
                  html_doc = urllib.urlopen(url).read()
                  addr.delete("1.0", END)
                  addr.insert("1.0", url)  # URL помещается в поле адреса
                  page.delete("1.0", END)
                  page.insert("1.0", html_doc)  # показывается HTML-документ

                textfrags = ["Python main site: ", "http://www.python.org",
                       "\nJython site: ", "http://www.jython.org",
                       "\nThat is all!"]
                for frag in textfrags:
                  if frag.startswith("http:"):
                    txt.insert(END, frag, "href")  # URL помещается в текст с
меткой href
                  else:
                    txt.insert(END, frag)         # фрагмент помещается в текст

                # ссылки отмечаются подчеркиванием и синим цветом
                txt.tag_config("href", foreground="Blue", underline=1)
                # при щелчке мыши на тексте, отмеченном как "href",
                # следует вызывать fetch_url()
                txt.tag_bind("href", "<1>", fetch_url)

                tk.mainloop()         # запускается цикл событий
```

В результате **(**после нажатия на гиперссылку**)** можно увидеть примерно следующее**:**



Для придания некоторым участкам текста особых свойств необходимо их отметить тегом. В данном случае **URL** отмечается тегом `href`. Позднее с помощью метода `tag_config()` задаются свойства отображения текста, отмеченного таким тегом. Методом `tag_bind()` привязывается некоторое событие (щелчок мыши) с вызовом заданной функции (`fetch_url()`).

В самой функции `fetch_url()`нужно в начале определить, на какой именно участок текста пришелся щелчок мыши. Для этого с помощью метода `tag_ranges()` получаются все интервалы, которые отмечены как `href`. Для определения конкретного **URL** проводятся сравнения (методом `compare()`) точки щелчка мышью с каждым из интервалов. Так находится интервал, на который попал щелчок, и с помощью метода `get()`получается текстовое значение найденного интервала. Найдя **URL**, его в поле записываются адреса, и получается **HTML**-код, соответствующий **URL**.

Этот пример показывает основные принципы работы с форматированным текстом. Примененными методами арсенал виджета не исчерпывается. О других методах и свойствах можно узнать из документации.

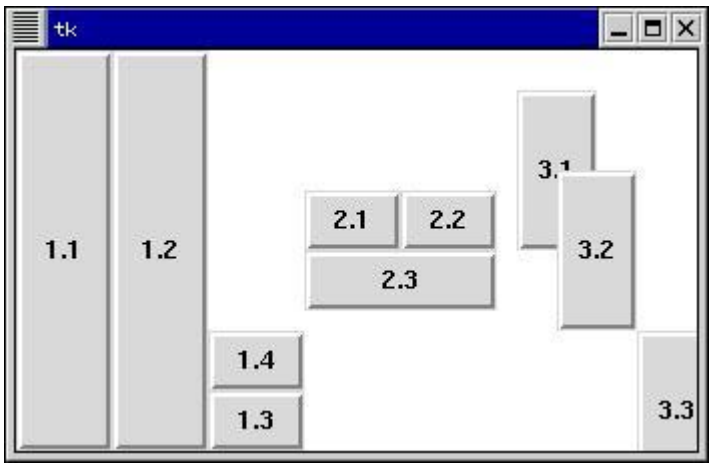
Менеджеры расположения

Следующий пример достаточно нагляден, чтобы понять принципы работы менеджеров расположения, имеющихся в **Tk**. В трех рамках можно применить различные менеджеры: **pack**, **grid** и **place**:

```
from Tkinter import *
tk = Tk()

# Создаем три рамки
frames = {}
b = {}
for fn in 1, 2, 3:
  f = Frame(tk, width=100, height=200, bg="White")
  f.pack(side=LEFT, fill=BOTH)
  frames[fn] = f
  for bn in 1, 2, 3, 4:  # Создаются кнопки для каждой из рамок
    b[fn, bn] = Button(frames[fn], text="%s.%s" % (fn, bn))

# Первая рамка:
# Сначала две кнопки прикрепляются к левому краю
b[1, 1].pack(side=LEFT, fill=BOTH, expand=1)
b[1, 2].pack(side=LEFT, fill=BOTH, expand=1)
# Еще две - к нижнему
b[1, 3].pack(side=BOTTOM, fill=Y)
b[1, 4].pack(side=BOTTOM, fill=BOTH)

# Вторая рамка:
# Две кнопки сверху
b[2, 1].grid(row=0, column=0, sticky=NW+SE)
b[2, 2].grid(row=0, column=1, sticky=NW+SE)
# и одна на две колонки в низу
b[2, 3].grid(row=1, column=0, columnspan=2, sticky=NW+SE)

# Третья рамка:
# Кнопки высотой и шириной в 40% рамки, якорь в левом верхнем
углу.
# Координаты якоря 1/10 от ширины и высоты рамки
b[3, 1].place(relx=0.1, rely=0.1, relwidth=0.4, relheight=0.4,
anchor=NW)
# Кнопка строго по центру. Якорь в центре кнопки
b[3, 2].place(relx=0.5, rely=0.5, relwidth=0.4, relheight=0.4,
anchor=CENTER)
# Якорь по центру кнопки. Координаты якоря 9/10 от ширины и
высоты рамки
```

```
                b[3, 3].place(relx=0.9, rely=0.9, relwidth=0.4, relheight=0.4,
anchor=CENTER)

                tk.mainloop()
```

Результат следующий:



Менеджер **pack** просто заполняет внутреннее пространство на основании предпочтения того или иного края**,** необходимости заполнить все измерение**.** В некоторых случаях ему приходится менять размеры подчиненных виджетов**.** Этот менеджер стоит использовать только для достаточно простых схем расположения виджетов**.**

Менеджер **grid** помещает виджеты в клетки сетки **(**это очень похоже на способ верстки таблиц в **HTML).** Каждому располагаемому виджету даются координаты в одной из ячеек сетки **(**`row` **-** строка**,** `column` **-** столбец**),** а также**,** если нужно**,** столько последующих ячеек **(**в строках ниже или в столбцах правее**)** сколько он может занять **(**свойства `rowspan` или `columnspan`**).** Это самый гибкий из всех менеджеров**.**

Менеджер **place** позволяет располагать виджеты по произвольным координатам и с произвольными размерами подчиненных виджетов**.** Размеры и координаты могут быть заданы в долях от размера виджета**-**хозяина**.**

Непосредственно внутри одного виджета нельзя использовать более одного менеджера расположения**:** менеджеры могут наложить противоречащие ограничения на вложенные виджеты и внутренние виджеты просто не смогут быть расположены**.**

Изображения в Tkinter

Средствами **Tkinter** можно выводить не только текст**,** примитивные формы **(**с помощью виджета **Canvas),** но и растровые изображения**.** Следующий пример демонстрирует вывод иконки с растровым изображением **(**для этого примера нужно предварительно установить пакет **Python Imaging Library, PIL):**

```
import Tkinter, Image, ImageTk

FILENAME = "lena.jpg"  # файл с графическим изображением

tk = Tkinter.Tk()
c = Tkinter.Canvas(tk, width=128, height=128)
src_img = Image.open(FILENAME)

img = ImageTk.PhotoImage(src_img)
c.create_image(0, 0, image=img, anchor="nw")
c.pack()
```

```
Tkinter.Label(tk, text=FILENAME).pack()

tk.mainloop()
```

В результате получается:

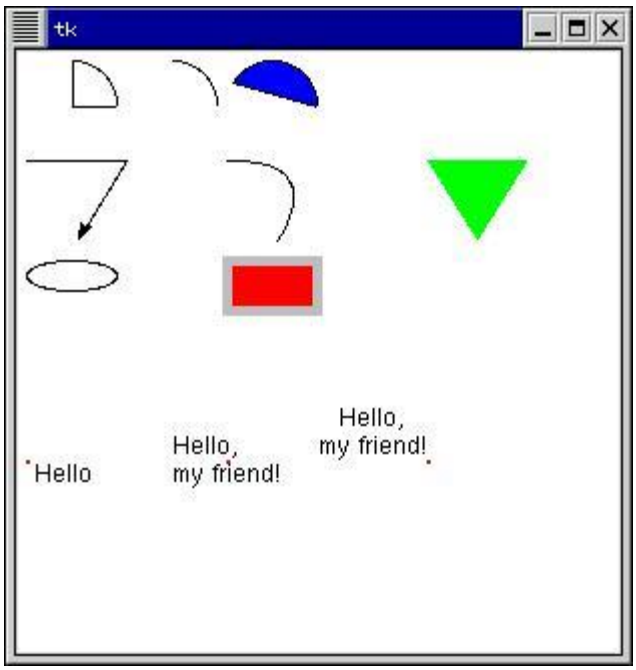
Здесь использован виджет-рисунок (Canvas). С помощью функций из пакетов Image и ImageTk из PIL получается объект-изображение, подходящее для включения в рисунок Tkinter. Свойство anchor задает угол, который привязывается к координатам (0, 0) в рисунке. В данном примере это северо-западный угол (NW - North-West). Другие возможности: n (север), w (запад), s (юг), e (восток), ne, sw, se и с (центр).

В следующем примере показаны графические примитивы, которые можно использовать на рисунке (приведенные комментарии объясняют свойства графических объектов внутри виджета-рисунка):

```
from Tkinter import *

tk = Tk()
# Рисунок 300x300 пикселей, фон - белый
c = Canvas(tk, width=300, height=300, bg="white")

c.create_arc((5, 5, 50, 50), style=PIESLICE)  # Сектор ("кусок
пирога")
c.create_arc((55, 5, 100, 50), style=ARC)     # Дуга
c.create_arc((105, 5, 150, 50), style=CHORD,  # Сегмент
             start=0, extent=150, fill="blue")  # от 0 до 150
градусов
# Ломаная со стрелкой на конце
c.create_line([(5, 55), (55, 55), (30, 95)], arrow=LAST)
# Кривая (сглаженная ломаная)
c.create_line([(105, 55), (155, 55), (130, 95)], smooth=1)
# Многоугольник зеленого цвета
c.create_polygon([(205, 55), (255, 55), (230, 95)],
fill="green")
# Овал
c.create_oval((5, 105, 50, 120), )
# Прямоугольник красного цвета с большой серой границей
c.create_rectangle((105, 105, 150, 130), fill="red",
                   outline="grey", width="5")
# Текст
c.create_text((5, 205), text=" Hello", anchor="nw")
# Эта точка визуально обозначает угол привязки
c.create_oval((5, 205, 6, 206), outline="red")
# Текст с заданным выравниванием
c.create_text((105, 205), text="Hello,\nmy friend!",
              justify=LEFT, anchor="c")
c.create_oval((105, 205, 106, 206), outline="red")
# Еще один вариант
```

```
c.create_text((205, 205), text="Hello,\nmy friend!",
              justify=CENTER, anchor="se")
c.create_oval((205, 205, 206, 206), outline="red")

c.pack()
tk.mainloop()
```

В результате работы этой программы на экране появится окно:



Следует заметить, что методы `create_*` создают объекты, свойства которых можно менять в дальнейшем: переместить в другое место, перекрасить, удалить, изменить порядок и т.д. В следующем примере можно нарисовать кружок, меняющий цвет по щелчку мыши:

```
from Tkinter import *
from random import choice

colors = "Red Orange Yellow Green LightBlue Blue
Violet".split()
R = 10

tk = Tk()
c = Canvas(tk, bg="White", width="4i", height=300,
relief=SUNKEN)
c.pack(expand=1, fill=BOTH)

def change_ball(event):
  c.coords(CURRENT, (event.x-R, event.y-R, event.x+R,
event.y+R))
  c.itemconfigure(CURRENT, fill=choice(colors))

oval = c.create_oval((100-R, 100-R, 100+R, 100+R),
fill="Black")
c.tag_bind(oval, "<1>", change_ball)
tk.mainloop()
```

Здесь нарисован кружок радиуса `R` , с ним связана функция `change_ball()` по нажатию кнопки мыши. В указанной функции заданы новые координаты кружка (его центр

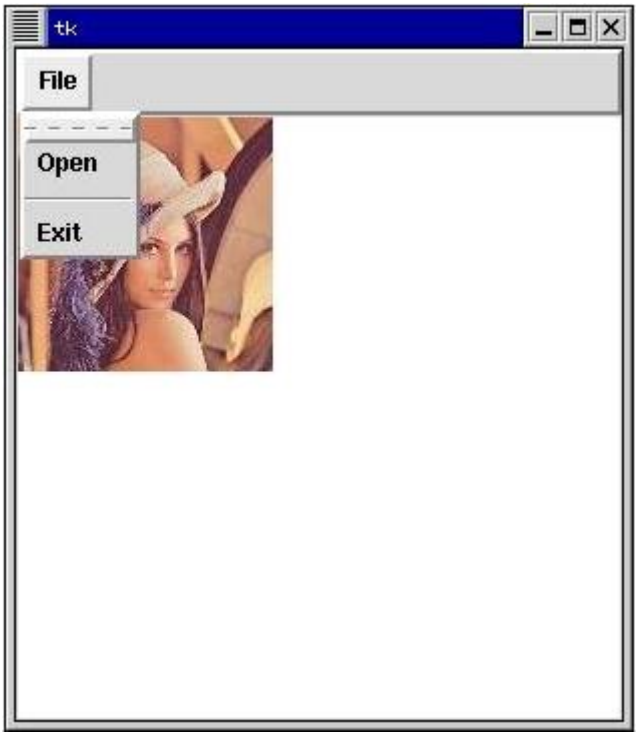
расположен в месте щелчка мыши) и затем изменен цвет случайным образом методом itemconfigure(). Тег CURRENT в **Tkinter** использован для указания объекта, который принял событие.

Графическое приложение на Tkinter

Теперь следует рассмотреть небольшое приложение, написанное с использованием Tkinter. В этом приложении будет загружен файл с графическим изображением. Приложение будет иметь простейшее меню **File** с пунктами **Open** и **Exit**, а также виджет **Canvas**, на котором и будут демонстрироваться изображения (опять потребуется пакет **PIL**):

```
from Tkinter import *
import Image, ImageTk, tkFileDialog
global img, imgobj

def show():
   global img, imgobj
   # Запрос на имя файла
   filename = tkFileDialog.askopenfilename()
   if filename != ():  # Если имя файла было задано
пользователем
     # рисуется изображение из файла
     src_img = Image.open(filename)
     img = ImageTk.PhotoImage(src_img)
     # конфигурируется изображение на рисунке
     c.itemconfigure(imgobj, image=img, anchor="nw")

tk = Tk()
main_menu = Menu(tk)       # формируется меню
tk.config(menu=main_menu)  # меню добавляется к окну
file_menu = Menu(main_menu)  # создается подменю
main_menu.add_cascade(label="File", menu=file_menu)
# Заполняется меню File
file_menu.add_command(label="Open", command=show)
file_menu.add_separator()  # черта для отделения пунктов меню
file_menu.add_command(label="Exit", command=tk.destroy)

c = Canvas(tk, width=300, height=300, bg="white")
# готовим объект-изображение на рисунке
imgobj = c.create_image(0, 0)
c.pack()

tk.mainloop()
```

Приложение (с загруженной картинкой) будет выглядеть так:



Стоит отметить, что здесь пришлось применить две глобальные переменные. Это не очень хорошо. Существует другой подход, когда приложение создается на основе окна верхнего уровня. Таким образом, само приложение становится особым виджетом. Переделанная программа представлена ниже:

```
                from Tkinter import *
                import Image, ImageTk, tkFileDialog

                class App(Tk):
                  def __init__(self):
                    Tk.__init__(self)
                    main_menu = Menu(self)
                    self.config(menu=main_menu)
                    file_menu = Menu(main_menu)
                    main_menu.add_cascade(label="File", menu=file_menu)
                    file_menu.add_command(label="Open", command=self.show_img)
                    file_menu.add_separator()
                    file_menu.add_command(label="Exit", command=self.destroy)

                    self.c = Canvas(self, width=300, height=300, bg="white")
                    self.imgobj = self.c.create_image(0, 0)
                    self.c.pack()

                  def show_img(self):
                     filename = tkFileDialog.askopenfilename()
                     if filename != ():
                       src_img = Image.open(filename)
                       self.img = ImageTk.PhotoImage(src_img)
                       self.c.itemconfigure(self.imgobj, image=self.img,
anchor="nw")

                app = App()
                app.mainloop()
```

В объекте заключена информация, которая до этого была глобальной со всеми следующими из этого ограничениями. Можно пойти дальше и выделить в отдельный метод настройку меню (если приложение будет динамически изменять меню, объекты-меню тоже могут быть сохранены в приложении).

Примечание:

На некоторых системах новые версии **Python** плохо работают с национальными кодировками, в частности, с кодировками для кириллицы. Это связано с переходом на **Unicode Tcl/Tk**. Проблем можно избежать, если использовать кодировку **UTF-8** в строках, которые должны выводиться в виджетах.

Заключение

В этой лекции было дано представление о (невизуальном) программировании графического интерфейса для **Python** на примере пакета **Tkinter**. Программа с графическим интерфейсом - событийно-управляемая программа, проводящая время в цикле обработки событий. События могут быть вызваны функционированием графического интерфейса или другими причинами (например, по таймеру). Обычно события возникают в виджетах и некоторые из них должны обрабатываться приложением. В **Tkinter** событие представлено отдельным объектом, из атрибутов которого можно установить, каково было положение указателя (курсора мыши), в каком виджете произошло событие и т.п.

Здесь были рассмотрены классы элементов интерфейса (виджеты), их свойства и методы. Виджеты имеют большое количество свойств и методов. Некоторые свойства и методы достаточно универсальны (их имеют все или почти все виджеты), другие же специфичны для конкретного класса виджетов. Графический пакет **Python Imaging Library (PIL)** предоставляет класс объекта для расположения в виджете-рисунке растрового графического изображения.

Виджеты располагаются внутри другого виджета (например, рамки) в соответствии с набором правил. Этот набор правил реализуют менеджеры расположения, которых в **Tkinter** три: **pack**, **grid** и **place**.

Приложение с графическим интерфейсом можно построить на базе окна верхнего уровня, простым наследованием. Этот подход позволяет инкапсулировать информацию, которую в противном случае пришлось бы делать глобальной.

Нужно отметить, что для построения интерфейса можно использовать не только чистый **Tkinter**. Например, в **Python** доступны модули `ScrolledText` и `Tix`, пополняющие набор виджетов. Кроме того, можно найти пакеты для специальных виджетов (например, для отображения дерева).

Построение графического интерфейса невизуальными способами - не такая сложная задача, если использовать **Tkinter**. Этот пакет входит в стандартную поставку **Python** и потому может использоваться почти везде, где установлен **Python**.

Ссылки

Список актуальных ссылок на различные графические библиотеки можно найти по следующему адресу:

http://phaseit.net/claird/comp.lang.python/python_GUI.html

Лекция #13: Интеграция Python с другими языками программирования

C API

Доступные из языка **Python** модули расширяются за счет **модулей расширения (extension modules).** Модули расширения можно писать на языке **C** или **C++** и вызывать из программ на **Python.** В этой лекции речь пойдет о реализации **Python,** называемой **CPython(Jython,** реализация **Python** на платформе **Java** не будет рассматриваться**).**

Сама необходимость использования языка **C** может возникнуть, если реализуемый алгоритм, будучи запрограммирован на **Python,** работает медленно. Например, высокопроизводительные операции с массивами модуля **Numeric (**о котором говорилось в одной из предыдущих лекций**)** написаны на языке **C.** Модули расширения позволяют объединить эффективность порождаемого компилятором **C/C++** кода **с** удобством и гибкостью интерпретатора **Python.** Необходимые сведения для создания модулей расширения для **Python** даны в исчерпывающем объеме в стандартной документации, а именно в документе **"Python/C API Reference Manual" (**справочное руководство по **"Python/C API").** Здесь будут рассмотрены лишь основные принципы построения модуля расширения, без детальных подробностей об **API.** Стоит заметить, что возможности **Python** равно доступны и в **C++,** просто они выражены в **C-**декларациях, которые можно использовать в **C++.**

Все необходимые для модуля расширения определения находятся в заголовочном файле **Python.h,** который должен находится где-то на пути заголовочных файлов компилятора **C/C++.** Следует пользоваться теми же версиями библиотек, с которыми был откомпилирован **Python.** Желательно, и той же маркой компилятора **C/C++.**

Связь с интерпретатором **Python** из кода на **C** осуществляется путем вызова функций, определенных в интерпретаторе **Python.** Все функции начинаются на `Py` или `_Py`, потому во избежание конфликтов в модулях расширения не следует определять функций с подобными именами.

Через **C API** доступны все встроенные возможности языка **Python (**при необходимости, детальнее изучить этот вопрос можно по документации**):**

- высокоуровневый интерфейс интерпретатора **(**функции и макросы `Py_Main()`, `PyRun_String()`, `PyRun_File()`, `Py_CompileString()`, `PyCompilerFlags()` и т.п.**),**
- функции для работы со встроенным интерпретатором и потоками **(**`Py_Initialize()`, `Py_Finalize()`, `Py_NewInterpreter()`, `Py_EndInterpreter()`, `Py_SetProgramName()` и другие**),**
- управление подсчетом ссылок **(**макросы `Py_INCREF()`, `Py_DECREF()`, `Py_XINCREF()`, `Py_XDECREF()`, `Py_CLEAR()`**).** Требуется при создании или удалении **Python-**объектов в **C/C++-**коде.
- обработка исключений **(**`PyErr*`**-**функции и `PyExc_*`**-**константы, например, `PyErr_NoMemory()` и `PyExc_IOError`**)**
- управление процессом и сервисы операционной системы **(**`Py_FatalError()`, `Py_Exit()`, `Py_AtExit()`, `PyOS_CheckStack()`, и другие функции**/**макросы `PyOS*`**),**
- импорт модулей **(**`PyImport_Import()` и другие**),**
- поддержка сериализации объектов **(**`PyMarshal_WriteObjectToFile()`, `PyMarshal_ReadObjectFromFile()` и т.п.**)**
- поддержка анализа строки аргументов **(**`PyArg_ParseTuple()`, `PyArg_VaParse()`, `PyArg_ParseTupleAndKeywords()`, `PyArg_VaParseTupleAndKeywords()`, `PyArg_UnpackTuple()` и `Py_BuildValue()`**).** С помощью этих функций облегчается задача получения в коде на **C** параметров, заданных при вызове функции из **Python.** Функции `PyArg_Parse*` принимают в качестве аргумента строку формата полученных аргументов,

- поддержка протоколов абстрактных объектов: + Протокол объекта (PyObject_Print(), PyObject_HasAttrString(), PyObject_GetAttrString(), PyObject_HasAttr(), PyObject_GetAttr(), PyObject_RichCompare(), ..., PyObject_IsInstance(), PyCallable_Check(), PyObject_Call(), PyObject_Dir() и другие). То, что должен уметь делать любой объект Python + Протокол числа (PyNumber_Check(), PyNumber_Add(), ..., PyNumber_And(), ..., PyNumber_InPlaceAdd(), ..., PyNumber_Coerce(), PyNumber_Int(), ...). То, что должен делать любой объект, представляющий число + Протокол последовательности (PySequence_Check(), PySequence_Size(), PySequence_Concat(), PySequence_Repeat(), PySequence_InPlaceConcat(), ..., PySequence_GetItem(), ..., PySequence_GetSlice(), PySequence_Tuple(), PySequence_Count(), ...) + Протокол отображения (например, словарь является отображением) (функции: PyMapping_Check(), PyMapping_Length(), PyMapping_HasKey(), PyMapping_Keys(), ..., PyMapping_SetItemString(), PyMapping_GetItemString() и др.) + Протокол итератора (PyIter_Check(), PyIter_Next()) + Протокол буфера (PyObject_AsCharBuffer(), PyObject_AsReadBuffer(), PyObject_AsWriteBuffer(), PyObject_CheckReadBuffer())
- поддержка встроенных типов данных. Аналогично описанному в предыдущем пункте, но уже для конкретных встроенных типов данных. Например: + Булевский объект (PyBool_Check() - проверка принадлежности типу PyBool_Type, Py_False - объект False, Py_True - объект True,
- управление памятью (то есть кучей интерпретатора Python) (функции PyMem_Malloc(), PyMem_Realloc(), PyMem_Free(), PyMem_New(), PyMem_Resize(), PyMem_Del()). Разумеется, можно применять и средства выделения памяти C/C++, однако, в этом случае не будут использоваться преимущества управления памятью интерпретатора Python (сборка мусора и т.п.). Кроме того, освобождение памяти нужно производить тем же способом, что и ее выделение. Еще раз стоит напомнить, что повторное освобождение одной и той же области памяти (а равно использование области памяти после ее освобождения) чревато серьезными ошибками, которые компилятор C не имеет возможности распознать.
- структуры для определения объектов встроенных типов (PyObject, PyVarObject и много других)

Примечание

Под **протоколом** здесь понимается набор методов, которые должен поддерживать тот или иной класс для организации операций со своими экземплярами. Эти методы доступны не только из Python (например, len(a) дает длину последовательности), но и из кода на C (PySequence_Length()).

Написание модуля расширения

Если необходимость встроить Python в программу возникает нечасто, то его расширение путем написания модулей на C/C++ - довольно распространенная практика. Изначально Python был нацелен на возможность расширения, поэтому в настоящий момент очень многие C/C++-библиотеки имеют привязки к Python.

Привязка к Python, хотя и может быть несколько автоматизирована, все же это процесс творческий. Дело в том, что если предполагается интенсивно использовать библиотеку в Python, ее привязку желательно сделать как можно более тщательно. Возможно, в ходе привязки будет сделана объектно-ориентированная надстройка или другие архитектурные изменения, которые позволят упростить использование библиотеки.

В качестве примера можно привести выдержку из исходного кода модуля md5, который реализует функцию для получения **md5**-дайджеста. Модуль приводится в целях иллюстрации (то есть, с сокращениями). Модуль вводит собственный тип данных, MD5Type, поэтому можно увидеть не только реализацию функций, но и способ описания встроенного

типа. В рамках этого курса не изучить все тонкости программирования модулей расширения, главное понять дух этого занятия. На комментарии автора курса лекций указывает двойной слэш //:

```
            // заголовочные файлы
            #include "Python.h"
            #include "md5.h"

            // В частности, в заголовочном файле md5.h есть следующие
определения:
            // typedef unsigned char *POINTER;
            // typedef unsigned int UINT4;

            // typedef struct {
            //  UINT4 state[4];        /* state (ABCD) */
            //  UINT4 count[2];        /* number of bits, modulo 2^64 (lsb
first) */
            //  unsigned char buffer[64];  /* input buffer */
            // } MD5_CTX;

            // Структура объекта MD5type
            typedef struct {
                    PyObject_HEAD
                    MD5_CTX     md5;            /* the context holder */
            } md5object;

            // Определение типа объекта MD5type
            static PyTypeObject MD5type;

            // Макрос проверки типа MD5type
            #define is_md5object(v)             ((v)->ob_type == &MD5type)

            // Порождение объекта типа MD5type
            static md5object *
            newmd5object(void)
            {
                    md5object *md5p;
                    md5p = PyObject_New(md5object, &MD5type);
                    if (md5p == NULL)
                            return NULL;        // не хватило памяти
                    MD5Init(&md5p->md5);        // инициализация
                    return md5p;
            }

            // Определения методов

            // Освобождение памяти из-под объекта
            static void
            md5_dealloc(md5object *md5p) { PyObject_Del(md5p); }

            static PyObject *
            md5_update(md5object *self, PyObject *args)
            {
                    unsigned char *cp;
                    int len;

                    // разбор строки аргументов. Формат указывает следующее:
                    // s# - один параметр, строка (заданная указателем и
длиной)
                    // : - разделитель
                    // update - название метода
                    if (!PyArg_ParseTuple(args, "s#:update", &cp, &len))
                            return NULL;

                    MD5Update(&self->md5, cp, len);
```

```
                // Даже возврат None требует увеличения счетчика ссылок
                Py_INCREF(Py_None);
                return Py_None;
        }

        // Строка документации метода update
        PyDoc_STRVAR(update_doc,
        "update (arg)\n\
        \n\
        Update the md5 object with the string arg. Repeated calls are\n\
        equivalent to a single call with the concatenation of all the\n\
        arguments.");

        // Метод digest
        static PyObject *
        md5_digest(md5object *self)
        {
                MD5_CTX mdContext;
                unsigned char aDigest[16];

                /* make a temporary copy, and perform the final */
                mdContext = self->md5;
                MD5Final(aDigest, &mdContext);

                // результат возвращается в виде строки
                return PyString_FromStringAndSize((char *)aDigest, 16);
        }

        // и строка документации
        PyDoc_STRVAR(digest_doc,  "digest() -> string\n\ ...");


        static PyObject *
        md5_hexdigest(md5object *self)
        {
            // Реализация метода на C
        }

        PyDoc_STRVAR(hexdigest_doc,  "hexdigest() -> string\n...");


        // Здесь было определение метода copy()

        // Методы объекта в сборе.
        // Для каждого метода указывается название, имя метода на C
        // (с приведением к типу PyCFunction), способ передачи аргументов:
        // METH_VARARGS (переменное кол-во) или METH_NOARGS (нет
аргументов)
        // В конце массива - метка окончания спиСка аргументов.
        static PyMethodDef md5_methods[] = {
        {"update",    (PyCFunction)md5_update,    METH_VARARGS,
update_doc},
        {"digest",    (PyCFunction)md5_digest,    METH_NOARGS,
digest_doc},
        {"hexdigest", (PyCFunction)md5_hexdigest, METH_NOARGS,
hexdigest_doc},
        {"copy",      (PyCFunction)md5_copy,      METH_NOARGS,  copy_doc},
        {NULL, NULL}                             /* sentinel */
        };

        // Атрибуты md5-объекта обслуживает эта функция, реализуя метод
        // getattr.
        static PyObject *
        md5_getattr(md5object *self, char *name)
        {
```

```
        // атрибут-данное digest_size
        if (strcmp(name, "digest_size") == 0) {
                return PyInt_FromLong(16);
        }
        // поиск атрибута-метода ведется в списке
        return Py_FindMethod(md5_methods, (PyObject *)self, name);
}

// Строка документации к модулю md5
PyDoc_STRVAR(module_doc, "This module implements ...");

// Строка документации к классу md5
PyDoc_STRVAR(md5type_doc, "An md5 represents the object...");

// Структура для объекта MD5type с описаниями для интерпретатора
static PyTypeObject MD5type = {
        PyObject_HEAD_INIT(NULL)
        0,                      /*ob_size*/
        "md5.md5",              /*tp_name*/
        sizeof(md5object),      /*tp_size*/
        0,                      /*tp_itemsize*/
        /* methods */
        (destructor)md5_dealloc,  /*tp_dealloc*/
        0,                      /*tp_print*/
        (getattrfunc)md5_getattr, /*tp_getattr*/
        0,                      /*tp_setattr*/
        0,                      /*tp_compare*/
        0,                      /*tp_repr*/
        0,                      /*tp_as_number*/
        0,                          /*tp_as_sequence*/
        0,                      /*tp_as_mapping*/
        0,                      /*tp_hash*/
        0,                      /*tp_call*/
        0,                      /*tp_str*/
        0,                      /*tp_getattro*/
        0,                      /*tp_setattro*/
        0,                      /*tp_as_buffer*/
        0,                      /*tp_xxx4*/
        md5type_doc,                    /*tp_doc*/
};


// Функции модуля md5:

// Функция new() для получения нового объекта типа md5type
static PyObject *
MD5_new(PyObject *self, PyObject *args)
{
        md5object *md5p;
        unsigned char *cp = NULL;
        int len = 0;

        // Разбор параметров. Здесь вертикальная черта
        // в строке формата означает окончание
        // списка обязательных параметров.
        // Остальное - как и выше: s# - строка, после : - имя
        if (!PyArg_ParseTuple(args, "|s#:new", &cp, &len))
                return NULL;

        if ((md5p = newmd5object()) == NULL)
                return NULL;

        // Если был задан параметр cp:
        if (cp)
                MD5Update(&md5p->md5, cp, len);
```

```
        return (PyObject *)md5p;
}

// Строка документации для new()
PyDoc_STRVAR(new_doc, "new([arg]) -> md5 object ...");

// Список функций, которые данный модуль экспортирует
static PyMethodDef md5_functions[] = {
        {"new",             (PyCFunction)MD5_new, METH_VARARGS,
new_doc},
        {"md5",             (PyCFunction)MD5_new, METH_VARARGS,
new_doc},
        {NULL,              NULL}   /* Sentinel */
};
// Следует заметить, что md5 - то же самое, что new. Эта функция
оставлена для
// обратной совместимости со старым модулем md5

// Инициализация модуля
PyMODINIT_FUNC
initmd5(void)
{
        PyObject *m, *d;

        MD5type.ob_type = &PyType_Type;
        // Инициализируется модуль
        m = Py_InitModule3("md5", md5_functions, module_doc);
        // Получается словарь с именами модуля
        d = PyModule_GetDict(m);
        // Добавляется атрибут MD5Type (тип md5-объекта) к словарю
        PyDict_SetItemString(d, "MD5Type", (PyObject *)&MD5type);
        // Добавляется целая константа digest_size к модулю
        PyModule_AddIntConstant(m, "digest_size", 16);
}
```

На основе этого примера можно строить собственные модули расширения, ознакомившись с документацией по **C/API** и документом **"Extending and Embedding" ("**Расширение и встраивание**")** из стандартной поставки **Python.** Перед тем, как приступать к созданию своего модуля, следует убедиться, что это целесообразно: подходящего модуля еще не создано и реализация в виде чистого **Python** неэффективна. Если создан действительно полезный модуль, его можно предложить для включения в поставку **Python.** Для этого нужно просто связаться с кем-нибудь из разработчиков по электронной почте или предложить модуль в виде "патча" через **http://sourceforge.net.**

Пример встраивания интерпретатора в программу на C

Интерпретатор **Python** может быть встроен в программу на **C** с использованием **C API.** Это лучше всего демонстрирует уже работающий пример:

```
/* File : demo.c */
/* Пример встраивания интерпретатора Python в другую программу */
#include "Python.h"

main(int argc, char **argv)
{
    /* Передает argv[0] интерпретатору Python */
    Py_SetProgramName(argv[0]);

    /* Инициализация интерпретатора */
    Py_Initialize();

    /* ... */
```

```
    /* Выполнение операторов Python (как бы модуль __main__) */
    PyRun_SimpleString("import time\n");
    PyRun_SimpleString("print time.localtime(time.time())\n");

    /* ... */

    /* Завершение работы интерпретатора */
    Py_Finalize();
}
```

Компиляция этого примера с помощью компилятора **gcc** может быть выполнена, например, так:

```
ver="2.3"
gcc -fpic demo.c -DHAVE_CONFIG_H -lm -lpython${ver} \
  -lpthread -lutil -ldl \
  -I/usr/local/include/python${ver} \
  -L/usr/local/lib/python${ver}/config \
  -Wl,-E \
  -o demo
```

Здесь следует отметить следующие моменты:

- программу необходимо компилировать вместе с библиотекой **libpython** соответствующей версии **(**для этого используется опция `-l`, за которой следует имя библиотеки**)** и еще с библиотеками, которые требуются для **Python**: `libpthread`, `libm`, `libutil` и т.п.**)**
- опция `pic` порождает код, не зависящий от позиции, что позволяет в дальнейшем динамически компоновать код
- обычно требуется явно указать каталог, в котором лежит заголовочный файл **Python.h (**в **gcc** это делается опцией `-I`**)**
- чтобы получившийся исполняемый файл мог корректно предоставлять имена для динамически загружаемых модулей, требуется передать компоновщику опцию `-E`: это можно сделать из `gcc` с помощью опции `-Wl,-E`. **(**В противном случае, модуль `time`, а это модуль расширения в виде динамически загружаемого модуля, не будет работать из-за того, что не увидит имен, определенных в **libpython)**

Здесь же следует сделать еще одно замечание: программа, встраивающая **Python**, не должна много раз выполнять `Py_Initialize()` и `Py_Finalize()`, так как это может приводить к утечке памяти. Сам же интерпретатор **Python** очень стабилен и в большинстве случаев не дает утечек памяти.

Использование SWIG

SWIG (Simplified Wrapper and Interface Generator, упрощенный упаковщик и генератор интерфейсов**) -** это программное средства, сильно упрощающее **(**во многих случаях **-** автоматизирующее**)** использование библиотек, написанных на **C** и **C++**, а также на других языках программирования, в том числе **(**не в последнюю очередь**!)** на **Python**. Нужно отметить, что **SWIG** обеспечивает достаточно полную поддержку практически всех возможностей **C++**, включая предобработку, классы, указатели, наследование и даже шаблоны **C++**. Последнее очень важно, если необходимо создать интерфейс к библиотеке шаблонов.

Пользоваться **SWIG** достаточно просто, если уметь применять компилятор и компоновщик **(**что в любом случае требуется при программировании на **C/C++)**.

Простой пример использования SWIG

Предположим, что есть программа на **C**, реализующая некоторую функцию **(**пусть это будет вычисление частоты появления различных символов в строке**):**

```
/* File : freq.c */
#include <stdlib.h>

int * frequency(char s[]) {
  int *freq;
  char *ptr;
  freq = (int*)(calloc(256, sizeof(int)));
  if (freq != NULL)
    for (ptr = s; *ptr; ptr++)
      freq[*ptr] += 1;
  return freq;
}
```

Для того чтобы можно было воспользоваться этой функцией из **Python**, нужно написать интерфейсный файл **(**расширение **.i)** примерно следующего содержания**:**

```
/* File : freq.i */
%module freq

%typemap(out) int * {
  int i;
  $result = PyTuple_New(256);
  for(i=0; i<256; i++)
    PyTuple_SetItem($result, i, PyLong_FromLong($1[i]));
  free($1);
}

extern int * frequency(char s[]);
```

Интерфейсные файлы содержат инструкции самого **SWIG** и фрагменты **C/C++-**кода, возможно, с макровключениями **(**в примере выше**:** `$result`, `$1`**).** Следует заметить, что для преобразования массива целых чисел в кортеж элементов типа `long`, необходимо освободить память из**-**под исходного массива, в котором подсчитывались частоты.

Теперь **(**подразумевая, что используется компилятор **gcc),** создание модуля расширения может быть выполнено примерно так**:**

```
swig -python freq.i
gcc -c -fpic freq_wrap.c freq.c  -DHAVE_CONFIG_H
   -I/usr/local/include/python2.3 -
I/usr/local/lib/python2.3/config
gcc -shared freq.o freq_wrap.o -o _freq.so
```

После этого в рабочем каталоге появляется файлы `_freq.so` и `freq.py`, которые вместе и дают доступ к требуемой функции**:**

```
>>> import freq
>>> freq.frequency("ABCDEF")[60:75]
(0L, 0L, 0L, 0L, 0L, 1L, 1L, 1L, 1L, 1L, 1L, 0L, 0L, 0L, 0L)
```

Помимо этого, можно посмотреть на содержимое файла **freq_wrap.c**, который был порожден **SWIG**: в нем, среди прочих вспомогательных определений, нужных самому **SWIG**, можно увидеть что-то подобное проиллюстрированному выше примеру модуля `md5`. Вот фрагмент этого файла с определением обертки для функции `frequency()`:

```
                extern int *frequency(char []);
                static PyObject *_wrap_frequency(PyObject *self, PyObject
*args) {
                    PyObject *resultobj;
                    char *arg1 ;
                    int *result;

                    if(!PyArg_ParseTuple(args,(char *)"s:frequency",&arg1))
goto fail;
                    result = (int *)frequency(arg1);

                    {
                        int i;
                        resultobj = PyTuple_New(256);
                        for(i=0; i<256; i++)
                        PyTuple_SetItem(resultobj, i,
PyLong_FromLong(result[i]));
                        free(result);
                    }
                    return resultobj;
                    fail:
                    return NULL;
                }
```

В качестве упражнения, предлагается сопоставить это определение с файлом **freq.i** и понять, что происходит внутри функции `_wrap_frequency()`. Подсказка: можно посмотреть еще раз комментарии к **C**-коду модуля `md5`.

Стоит еще раз напомнить, что в отличие от **Python**, в языке **C/C++** управление памятью должно происходить в явном виде. Именно поэтому добавлена функция `free()` при преобразовании типа. Если этого не сделать, возникнут **утечки памяти**. Эти утечки можно обнаружить, при многократном выполнении функции:

```
                >>> import freq
                >>> for i in xrange(1000000):
                ...   dummy =  freq.frequency("ABCDEF")
                >>>
```

Если функция `freq.frequency()` имеет утечки памяти, выполняемый процесс очень быстро займет всю имеющуюся память.

Интеграция Python и других систем программирования

Язык программирования **Python** является сценарным языком, а значит его основное назначение - интеграция в единую систему разнородных программных компонентов. Выше рассматривалась (низкоуровневая) интеграция с **C/C++**-приложениями. Нужно заметить, что в большинстве случаев достаточно интеграции с использованием протокола. Например, интегрируемые приложения могут общаться через **XML-RPC**, **SOAP**, **CORBA**, **COM**, **.NET** и т.п. В случаях, когда приложения имеют интерфейс командной строки, их можно вызывать из **Python** и управлять стандартным вводом-выводом, переменными окружения. Однако есть и более интересные варианты интеграции.

Современное состояние дел по излагаемому вопросу можно узнать по адресу: http://www.python.org/moin/IntegratingPythonWithOtherLanguages

Java

Документация по **Jython** (это реализация **Python** на **Java**-платформе) отмечает, что **Jython** обладает следующими неоспоримыми преимуществами над другими языками, использующими **Java**-байт-код:

- **Jython**-код динамически компилирует байт-коды **Java**, хотя возможна и статическая компиляция, что позволяет писать апплеты, сервлеты и т.п.;
- Поддерживает объектно-ориентированную модель **Java**, в том числе, возможность наследовать от абстрактных **Java**-классов;
- **Jython** является реализацией **Python** - языка с практичным синтаксисом, обладающего большой выразительностью, что позволяет сократить сроки разработки приложений в разы.

Правда, имеются и некоторые ограничения по сравнению с "обычным" **Python**. Например, **Java** не поддерживает множественного наследования, поэтому в некоторых версиях **Jython** нельзя наследовать классы от нескольких **Java**-классов (в тоже время, множественное наследование поддерживается для **Python**-классов).

Следующий пример (файл **lines.py**) показывает полную интеграцию **Java**-классов с интерпретатором **Python**:

```
                # Импортируются модули из Java
                from java.lang import System
                from java.awt import *
                # А это модуль из Jython
                import random

                # Класс для рисования линий на рисунке
                class Lines(Canvas):
                  # Реализация метода paint()
                  def paint(self, g):
                    X, Y = self.getSize().width, self.getSize().height
                    label.setText("%s x %s" % (X, Y))
                    for i in range(100):
                      x1, y1 = random.randint(1, X), random.randint(1, Y)
                      x2, y2 = random.randint(1, X), random.randint(1, Y)
                      g.drawLine(x1, y1, x2, y2)

                # Метки, кнопки и т.п.
                panel = Panel(layout=BorderLayout())
                label = Label("Size", Label.RIGHT)
                panel.add(label, "North")
                button = Button("QUIT", actionPerformed=lambda e:
System.exit(0))
                panel.add(button, "South")
                lines = Lines()
                panel.add(lines, 'Center')

                # Запуск панели в окне
                import pawt
                pawt.test(panel, size=(240, 240))
```

Программы на **Jython** можно компилировать в **Java** и собирать в **jar**-архивы. Для создания **jar**-архива на основе модуля (или пакета) можно применить команду `jythonc`, которая входит в комплект **Jython**. Из командной строки это можно сделать примерно так:

```
jythonc -d -c -j lns.jar lines.py
```

Для запуска приложения достаточно запустить `lines` из командной строки:

```
java -classpath "$CLASSPATH" lines
```

В переменной `$CLASSPATH` должны быть пути к архивам `lns.jar` и `jython.jar`.

Prolog

Для тех, кто хочет использовать **Prolog** из **Python**, существует несколько возможностей:

- Версия **GNU Prolog** (сайт: **http://gprolog.sourceforge.net**) интегрируется с **Python** посредством пакета **bedevere** (сайт: **http://bedevere.sourceforge.net**)
- Имеется пакет **PyLog** (**http://www.gocept.com/angebot/opensource/Pylog**) для работы с **SWI-Prolog** (**http://www.swi-prolog.org**) из **Python**
- Можно использовать пакет `pylog` (доступен с сайта: **http://christophe.delord.free.fr/en/pylog/**), который добавляет основные возможности **Prolog** в **Python**

Эти три варианта реализуют различные способы интеграции возможностей **Prolog** в **Python**. Первый вариант использует **SWIG**, второй организует общение с **Prolog**-системой через конвейер, а третий является специализированной реализацией **Prolog**.

Следующий пример показывает использование модуля `pylog`:

```
from pylog import *

exec(compile(r"""
man('Socrates').
man('Democritus').
mortal(X) :- man(X).
"""))

WHO = Var()
queries = [mortal('Socrates'),
           man(WHO),
           mortal(WHO)]
for query in queries:
  print "?", query
  for _ in query():
    print "    yes:", query
```

Что выдает результат:

```
? mortal(Socrates)
    yes: mortal(Socrates)
? man(_)
    yes: man(Socrates)
    yes: man(Democritus)
? mortal(_)
    yes: mortal(Socrates)
    yes: mortal(Democritus)
```

Разумеется, это не "настоящий" **Prolog**, но с помощью модуля `pylog` любой, кому требуются логические возможности **Prolog** в **Python**, может написать программу с использованием **Prolog**-синтаксиса.

OCaml

Язык программирования **OCaml** - это язык функционального программирования (семейства **ML**, что означает **Meta Language**), созданный в институте **INRIA**, Франция. Важной особенностью **OCaml** является то, что его компилятор порождает исполняемый код, по быстродействию сравнимый с C, родной для платформ, на которых **OCaml** реализован. В то же время, будучи функциональным по своей природе, он приближается к **Python** по степени выразительности. Именно поэтому для **OCaml** была создана **библиотека Pycaml**, фактически реализующая аналог **C API** для **OCaml**. Таким образом, в программах на **OCaml** могут использоваться модули языка **Python**, в них даже может быть встроен интерпретатор **Python**. Для **Python** имеется большое множество адаптированных **C**-библиотек, это дает возможность пользователям **OCaml** применять в разработке комбинированное преимущество **Python** и **OCaml**. Минусом является только необходимость знать функции **Python/C API**, имена которого использованы для связи **OCaml** и **Python**.

Следующий пример (из **Pycaml**) показывает программу для **OCaml**, которая определяет модуль для **Python** на **OCaml** и вызывает встроенный интерпретатор **Python**:

```
                let foo_bar_print = pywrap_closure
                  (fun x -> pytuple_fromarray (pytuple_toarray x)) ;;
                let sd = pyimport_getmoduledict () ;;
                let mx = pymodule_new "CamlModule" ;;
                let cd = pydict_new () ;;
                let cx = pyclass_new (pynull (), cd, pystring_fromstring
"CamlClass") ;;
                let cmx = pymethod_new (foo_bar_print,(pynull ()),cx) ;;
                let _ = pydict_setitemstring (cd, "CamlMethod", cmx) ;;
                let _ = pydict_setitemstring (pymodule_getdict mx, "CamlClass",
cx) ;;
                let _ = pydict_setitemstring (sd, "CamlModule", mx) ;;
                let _ = pyrun_simplestring
                  ("from CamlModule import CamlClass\n" ^
                   "x = CamlClass()\n" ^
                   "for i in range(100000):\n" ^
                   "  x.CamlMethod(1,2,3,4)\n" ^
                   "print 'Done'\n")
```

Pyrex

Для написания модулей расширения можно использовать специальный язык - **Pyrex** - который совмещает синтаксис **Python** и типы данных **C**. Компилятор **Pyrex** написан на **Python** и превращает исходный файл (например, **primes.pyx**) в файл на **C** - готовый для компиляции модуль расширения. Язык **Pyrex** заботится об управлении памятью, удаляя после себя ставшие ненужными объекты. Пример файла из документации к **Pyrex** (для вычисления простых чисел):

```
                def primes(int kmax):
                  cdef int n, k, i
                  cdef int p[1000]
                  result = []
                  if kmax > 1000:
                    kmax = 1000
                  k = 0
                  n = 2
                  while k < kmax:
                    i = 0
```

```
    while i < k and n % p[i] <> 0:
      i = i + 1
    if i == k:
      p[k] = n
      k = k + 1
      result.append(n)
    n = n + 1
  return result
```

В результате применения компилятора **Pyrex,** нехитрой компиляции и компоновки **(**с помощью **GCC):**

```
pyrexc primes.pyx
gcc primes.c -c -fPIC -I /usr/local/include/python2.3
gcc -shared primes.o -o primes.so
```

Получается модуль расширения с функцией `primes()`**:**

```
>>> import primes
>>> primes.primes(25)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53,
59, 61,
67, 71, 73, 79, 83, 89, 97]
```

Разумеется**,** в **Pyrex** можно использовать **C-**библиотеки**,** именно поэтому он**,** как и **SWIG,** может служить для построения оберток **C-**библиотек для **Python.**

Следует отметить**,** что для простых операций **Pyrex** применяет **C,** а для обращения к объектам **Python -** вызовы **Python/C API.** Таким образом**,** объединяется выразительность **Python** и эффективность **C.** Конечно**,** некоторые вещи в **Pyrex** не доступны**,** например**,** генераторы**,** списковые включения и **Unicode,** однако**,** цель **Pyrex -** создание быстродействующих модулей расширения**,** и для этого он превосходно подходит**.** Ознакомится с **Pyrex** можно по документации **(**которая**,** к сожалению**,** есть пока только на английском языке**).**

Заключение

В этой лекции кратко рассматривались основные возможности интеграции интерпретатора **Python** и других систем программирования**.** Базовая реализация языка **Python** написана на **C,** поэтому **Python** имеет программный интерфейс **Python/C API,** который позволяет программам на **C/C++** обращаться к интерпретатору **Python,** отдельным объектам**,** модулям и типам данных**.** Состав **Python/C API** достаточно обширен**,** поэтому речь шла лишь о некоторых основных его элементах**.**

Был рассмотрен процесс написания модуля расширения на **C** как напрямую**,** так и с использованием генератора интерфейсов **SWIG.** Также кратко говорилось о возможности встраивания интерпретатора **Python** в программу на C или **OCaml.**

Язык **Python (**с помощью специальной его реализации **- Jython)** прозрачно интегрируется с языком **Java:** в **Python-**программе**,** выполняемой под **Jython** в **Java-**апплете или **Java-**приложении**,** можно использовать практически любые **Java-**классы**.**

На примере языка **Prolog** были показаны различные подходы к добавлению возможностей логического вывода в **Python-**программы**:** независимая реализация **Prolog-**машины**,** связь с **Prolog-**интерпретатором через конвейер**,** связь через **Python/C API.**

Интересный гибрид **C** и **Python** представляет из себя язык **Pyrex.** Этот язык создан с целью упростить написание модулей расширения для **Python** на **C,** и использует структуры данных **C** и подобный **Python** синтаксис**.** Несмотря на некоторые смысловые и синтаксические отличия как от **C,** так и от **Python,** язык **Pyrex** помогает существенно сократить время разработки модулей расширения**,** сохранив эффективность компилятора **C** и знакомый синтаксис **Python.**

В данной лекции не были представлены другие возможности интеграции**,** например библиотека шаблонов **C++ Boost Python,** которая позволяет интегрировать **Python** и **C++.** Кроме того**,** из **Python** можно использовать библиотеки**,** написанные на Фортране **(**проект **F2PY).**

Развитые и гибкие интеграционные возможности **Python** являются его основным преимуществом в качестве языка для интеграции приложений**.** Из лекции нетрудно заключить**,** что **Python** легко взаимодействует с другими системами**.**

Ссылки

Библиотека **Boost Python** для **C++** **http://www.boost.org**

Лекция #14: Устройство интерпретатора языка Python

Лексический анализ

Лексический анализатор языка программирования разбивает исходный текст программы (состоящий из одиночных символов) на лексемы - неделимые "слова" языка.

Основные категории лексем **Python:** идентификаторы и ключевые слова **(NAME),** литералы **(STRING, NUMBER** и т.п.**),** операции **(OP),** разделители**,** специальные лексемы для обозначения **(**изменения**)** отступов **(INDENT, DEDENT)** и концов строк **(NEWLINE),** а также комментарии **(COMMENT).** Лексический анализатор доступен через модуль `tokenize`**,** а определения кодов лексем содержатся в модуле `token` стандартной библиотеки **Python.** Следующий пример показывает лексический анализатор в действии**:**

```
        import StringIO, token, tokenize

        prog_example = """
        for i in range(100): # comment
          if i % 1 == 0: \
          print ":", t**2
        """.strip()

        rl = StringIO.StringIO(prog_example).readline

        for t_type, t_str, (br,bc), (er,ec), logl in
tokenize.generate_tokens(rl):
          print "%3i %10s : %20r" % (t_type, token.tok_name[t_type], t_str)
```

А вот что выведет эта программа**,** разбив на лексемы исходный код примера**:**

```
        prog_example:

         1       NAME :                'for'
         1       NAME :                  'i'
         1       NAME :                 'in'
         1       NAME :              'range'
        50         OP :                  '('
         2     NUMBER :                '100'
        50         OP :                  ')'
        50         OP :                  ':'
        52    COMMENT :          '# comment'
         4    NEWLINE :                 '\n'
         5     INDENT :                  '  '
         1       NAME :                 'if'
         1       NAME :                  'i'
        50         OP :                  '%'
         2     NUMBER :                  '1'
        50         OP :                 '=='
         2     NUMBER :                  '0'
        50         OP :                  ':'
         1       NAME :              'print'
         3     STRING :                '":"'
        50         OP :                  ','
         1       NAME :                  't'
        50         OP :                 '**'
         2     NUMBER :                  '2'
         6     DEDENT :                   ''
         0  ENDMARKER :                   ''
```

Фактически получен поток лексем, который может использоваться для различных целей. Например, для синтаксического "окрашивания" кода на языке **Python**. Словарь `token.tok_name` позволяет получить мнемонические имена для типа лексемы по номеру.

Синтаксический анализ

Вторая стадия преобразования исходного текста программы в байт-код интерпретатора состоит в синтаксическом анализе исходного текста. Модуль `parser` содержит функции `suite()` и `expr()` для построения **деревьев синтаксического разбора** соответственно для кода программ и выражений **Python**. Модуль `symbol` содержит номера символов грамматики **Python**, словарь для получения названия символа из грамматики **Python**.

Следующая программа анализирует достаточно простой код **Python** (`prg`) и порождает дерево синтаксического разбора (**AST**-объект), который тут же можно превращать в кортеж и красиво выводить функцией `pprint.pprint()`. Далее определяется функция для превращения номеров символов в их мнемонические обозначения (имена) в грамматике:

```
import pprint, token, parser, symbol

prg = """print 2*2"""

pprint.pprint(parser.suite(prg).totuple())

def pprint_ast(ast, level=0):
  if type(ast) == type(()):
    for a in ast:
      pprint_ast(a, level+1)
  elif type(ast) == type(""):
    print repr(ast)
  else:
    print " "*level,
    try:
      print symbol.sym_name[ast]
    except:
      print "token."+token.tok_name[ast],

print
pprint_ast(parser.suite(prg).totuple())
```

Эта программа выведет следующее (структура дерева отражена отступами):

```
(257,
 (264,
  (265,
   (266,
    (269,
     (1, 'print'),
     (292,
      (293,
       (294,
        (295,
         (297,
          (298,
           (299,
            (300,
             (301,
              (302,
               (303, (304, (305, (2, '2')))),
               (16, '*'),
               (303, (304, (305, (2, '2'))))))))))))))),
   (4, ''))),
```

```
(0, ''))

 file_input
  stmt
   simple_stmt
    small_stmt
     print_stmt
      token.NAME 'print'
      test
       and_test
        not_test
         comparison
          expr
           xor_expr
            and_expr
             shift_expr
              arith_expr
               term
                factor
                 power
                  atom
                   token.NUMBER '2'
                token.STAR '*'
                factor
                 power
                  atom
                   token.NUMBER '2'
    token.NEWLINE ''
   token.ENDMARKER ''
```

Получение байт-кода

После того как получено дерево синтаксического разбора, компилятор должен превратить его в байт-код, подходящий для исполнения интерпретатором. В следующей программе проводятся отдельно синтаксический анализ, компиляция и выполнение (вычисление) кода (и выражения) в языке **Python**:

```
import parser

prg = """print 2*2"""
ast = parser.suite(prg)
code = ast.compile('filename.py')
exec code

prg = """2*2"""
ast = parser.expr(prg)
code = ast.compile('filename1.py')
print eval(code)
```

Функция `parser.suite()` (или `parser.expr()`) возвращает **AST**-объект (дерево синтаксического анализа), которое методом `compile()` компилируется в **Python** байт-код и сохраняется в кодовом объекте `code`. Теперь этот код можно выполнить (или, в случае выражения - вычислить) с помощью оператора `exec` (или функции `eval()`).

Здесь необходимо заметить, что недавно в **Python** появился пакет `compiler`, который объединяет модули для работы анализа исходного кода на **Python** и генерации кода. В данной лекции он не рассматривается, но те, кто хочет глубже изучить эти процессы, может обратиться к документации по **Python**.

Изучение байт-кода

Для изучения байт-кода **Python**-программы можно использовать модуль `dis` **(**сокращение от "дизассемблер"**),** который содержит функции**,** позволяющие увидеть байт-код в мнемоническом виде**.** Следующий пример иллюстрирует эту возможность**:**

```
>>> def f():
...   print 2*2
...
>>> dis.dis(f)
  2           0 LOAD_CONST               1 (2)
              3 LOAD_CONST               1 (2)
              6 BINARY_MULTIPLY
              7 PRINT_ITEM
              8 PRINT_NEWLINE
              9 LOAD_CONST               0 (None)
             12 RETURN_VALUE
```

Определяется функция `f()`**,** которая должна вычислить и напечатать значение выражения `2*2`**.** Функция `dis()` модуля `dis` выводит код функции `f()` в виде некого "ассемблера"**,** в котором байт-код **Python** представлен мнемоническими именами**.** Следует заметить**,** что при интерпретации используется стек**,** поэтому `LOAD_CONST` кладет значение на вершину стека**,** а `BINARY_MULTIPLY` берет со стека два значения и помещает на стек результат их перемножения**.** Функция без оператора `return` возвращает значение `None`**.** Как и в случае с кодами для микропроцессора**,** некоторые байт-коды принимают параметры**.**

Мнемонические имена можно увидеть в списке `dis.opname` **(**ниже печатаются только задействованные имена**):**

```
>>> import dis
>>> [n for n in dis.opname if n[0] != "<"]
['STOP_CODE', 'POP_TOP', 'ROT_TWO', 'ROT_THREE', 'DUP_TOP',
'ROT_FOUR',
 'NOP', 'UNARY_POSITIVE', 'UNARY_NEGATIVE', 'UNARY_NOT',
'UNARY_CONVERT',
 'UNARY_INVERT', 'LIST_APPEND', 'BINARY_POWER', 'BINARY_MULTIPLY',
 'BINARY_DIVIDE', 'BINARY_MODULO', 'BINARY_ADD', 'BINARY_SUBTRACT',
 'BINARY_SUBSCR', 'BINARY_FLOOR_DIVIDE', 'BINARY_TRUE_DIVIDE',
 'INPLACE_FLOOR_DIVIDE', 'INPLACE_TRUE_DIVIDE', 'SLICE+0',
'SLICE+1',
 'SLICE+2', 'SLICE+3', 'STORE_SLICE+0', 'STORE_SLICE+1',
'STORE_SLICE+2',
 'STORE_SLICE+3', 'DELETE_SLICE+0', 'DELETE_SLICE+1',
'DELETE_SLICE+2',
 'DELETE_SLICE+3', 'INPLACE_ADD', 'INPLACE_SUBTRACT',
'INPLACE_MULTIPLY',
 'INPLACE_DIVIDE', 'INPLACE_MODULO', 'STORE_SUBSCR',
'DELETE_SUBSCR',
 'BINARY_LSHIFT', 'BINARY_RSHIFT', 'BINARY_AND', 'BINARY_XOR',
'BINARY_OR',
 'INPLACE_POWER', 'GET_ITER', 'PRINT_EXPR', 'PRINT_ITEM',
'PRINT_NEWLINE',
 'PRINT_ITEM_TO', 'PRINT_NEWLINE_TO', 'INPLACE_LSHIFT',
'INPLACE_RSHIFT',
 'INPLACE_AND', 'INPLACE_XOR', 'INPLACE_OR', 'BREAK_LOOP',
'LOAD_LOCALS',
 'RETURN_VALUE', 'IMPORT_STAR', 'EXEC_STMT', 'YIELD_VALUE',
'POP_BLOCK',
 'END_FINALLY', 'BUILD_CLASS', 'STORE_NAME', 'DELETE_NAME',
 'UNPACK_SEQUENCE', 'FOR_ITER', 'STORE_ATTR', 'DELETE_ATTR',
'STORE_GLOBAL',
```

```
            'DELETE_GLOBAL', 'DUP_TOPX', 'LOAD_CONST', 'LOAD_NAME',
'BUILD_TUPLE',
            'BUILD_LIST', 'BUILD_MAP', 'LOAD_ATTR', 'COMPARE_OP',
'IMPORT_NAME',
            'IMPORT_FROM', 'JUMP_FORWARD', 'JUMP_IF_FALSE', 'JUMP_IF_TRUE',
            'JUMP_ABSOLUTE', 'LOAD_GLOBAL', 'CONTINUE_LOOP', 'SETUP_LOOP',
            'SETUP_EXCEPT', 'SETUP_FINALLY', 'LOAD_FAST', 'STORE_FAST',
'DELETE_FAST',
            'RAISE_VARARGS', 'CALL_FUNCTION', 'MAKE_FUNCTION', 'BUILD_SLICE',
            'MAKE_CLOSURE', 'LOAD_CLOSURE', 'LOAD_DEREF', 'STORE_DEREF',
            'CALL_FUNCTION_VAR', 'CALL_FUNCTION_KW', 'CALL_FUNCTION_VAR_KW',
            'EXTENDED_ARG']
```

Легко догадаться, что **LOAD** означает загрузку значения в стек, **STORE -** выгрузку, **PRINT -** печать, **BINARY -** бинарную операцию и т.п.

Отладка

В интерпретаторе языка **Python** заложены возможности отладки программ, а в стандартной поставке имеется простейший **отладчик -** `pdb`. Следующий пример показывает программу, которая подвергается отладке, и типичную сессию отладки:

```
# File myfun.py
def fun(s):
  lst = []
  for i in s:
    lst.append(ord(i))
  return lst
```

Так может выглядеть типичный процесс отладки:

```
>>> import pdb, myfun
>>> pdb.runcall(myfun.fun, "ABCDE")
> /examples/myfun.py(4)fun()
-> lst = []
(Pdb) n
> /examples/myfun.py(5)fun()
-> for i in s:
(Pdb) n
> /examples/myfun.py(6)fun()
-> lst.append(ord(i))
(Pdb) l
  1     #!/usr/bin/python
  2     # File myfun.py
  3     def fun(s):
  4       lst = []
  5       for i in s:
  6  ->     lst.append(ord(i))
  7       return lst
[EOF]
(Pdb) p lst
[]
(Pdb) p vars()
{'i': 'A', 's': 'ABCDE', 'lst': []}
(Pdb) n
> /examples/myfun.py(5)fun()
-> for i in s:
(Pdb) p vars()
{'i': 'A', 's': 'ABCDE', 'lst': [65]}
(Pdb) n
```

```
> /examples/myfun.py(6)fun()
-> lst.append(ord(i))
(Pdb) n
> /examples/myfun.py(5)fun()
-> for i in s:
(Pdb) p vars()
{'i': 'B', 's': 'ABCDE', 'lst': [65, 66]}
(Pdb) r
- Return -
> /examples/myfun.py(7)fun()->[65, 66, 67, 68, 69]
-> return lst
(Pdb) n
[65, 66, 67, 68, 69]
>>>
```

Интерактивный отладчик вызывается функцией `pdb.runcall()` и на его приглашение `(Pdb)` следует вводить команды. В данном примере сессии отладки были использованы некоторые из следующих команд: `l` (печать фрагмент трассируемого кода), `n` (выполнить все до следующей строки), `s` (сделать следующий шаг, возможно, углубившись в вызов метода или функции), `p` (печать значения), `r` (выполнить все до возврата из текущей функции).

Разумеется, некоторые интерактивные оболочки разработчика для **Python** предоставляют функции отладчика. Кроме того, отладку достаточно легко организовать, поставив в ключевых местах программы, операторы `print` для вывода интересующих параметров. Обычно этого достаточно, чтобы локализовать проблему. В **CGI**-сценариях можно использовать модуль `cgitb`, о котором говорилось в одной из предыдущих лекций.

Профайлер

Для определения мест в программе, на выполнение которых уходит значительная часть времени, обычно применяется **профайлер**.

Модуль profile

Этот модуль позволяет проанализировать работу функции и выдать статистику использования процессорного времени на выполнение той или иной части алгоритма.

В качестве примера можно рассмотреть профилирование функции для поиска строк из списка, наиболее похожих на данную. Для того чтобы качественно профилировать функцию `difflib.get_close_matches()`, нужен большой объем данных. В файле **russian.txt** собрано **160** тысяч слов русского языка. Следующая программа поможет профилировать функцию `difflib.get_close_matches()`:

```
import difflib, profile

def print_close_matches(word):
  print "\n".join(difflib.get_close_matches(word + "\n",
open("russian.txt")))

profile.run(r'print_close_matches("профайлер")')
```

При запуске этой программы будет выдано примерно следующее:

```
провайдер

трайлер
```

```
бройлер

899769 function calls (877642 primitive calls) in 23.620 CPU seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
1    0.000    0.000   23.610   23.610 <string>:1(?)
1    0.000    0.000   23.610   23.610 T.py:6(print_close_matches)
1    0.000    0.000    0.000    0.000 difflib.py:147(__init__)
1    0.000    0.000    0.000    0.000 difflib.py:210(set_seqs)
159443    1.420    0.000    1.420    0.000 difflib.py:222(set_seq1)
2    0.000    0.000    0.000    0.000 difflib.py:248(set_seq2)
2    0.000    0.000    0.000    0.000 difflib.py:293(__chain_b)
324261    2.240    0.000    2.240    0.000 difflib.py:32(_calculate_ratio)
28317    1.590    0.000    1.590    0.000 difflib.py:344(find_longest_match)
6474    0.100    0.000    2.690    0.000 difflib.py:454(get_matching_blocks)
28317/6190    1.000    0.000    2.590    0.000 difflib.py:480(__helper)
6474    0.450    0.000    3.480    0.001 difflib.py:595(ratio)
28686    0.240    0.000    0.240    0.000 difflib.py:617(<lambda>)
158345    8.690    0.000    9.760    0.000 difflib.py:621(quick_ratio)
159442    2.950    0.000    4.020    0.000 difflib.py:650(real_quick_ratio)
1    4.930    4.930   23.610   23.610 difflib.py:662(get_close_matches)
1    0.010    0.010   23.620   23.620
profile:0(print_close_matches("профайлер"))
0    0.000             0.000          profile:0(profiler)
```

Здесь колонки таблицы показывают следующие значения: **ncalls -** количество вызовов **(**функции**),** **tottime -** время выполнения кода функции **(**не включая времени выполнения вызываемых из нее функций**),** **percall -** то же время**,** в пересчете на один вызов**,** **cumtime -** суммарное время выполнения функции **(**и всех вызываемых из нее функций**),** **filename -** имя файла**,** **lineno -** номер строки в файле**,** **function -** имя функции **(**если эти параметры известны**).**

Из приведенной статистики следует**,** что наибольшие усилия по оптимизации кода необходимо приложить в функциях `quick_ratio()` **(**на нее потрачено **8,69** секунд**),** `get_close_matches()` **(4,93** секунд**),** затем можно заняться `real_quick_ratio()` **(2,95** секунд**)** и `_calculate_ratio()` **(**секунд**).**

Это лишь самый простой вариант использования профайлера: модуль `profile` **(**и связанный с ним `pstats`**)** позволяет получать и обрабатывать статистику: их применение описано в документации.

Модуль timeit

Предположим**,** что проводится оптимизация небольшого участка кода. Необходимо определить**,** какой из вариантов кода является наиболее быстрым. Это можно сделать с помощью модуля `timeit`.

В следующей программе используется метод `timeit()` для измерения времени**,** необходимого для вычисления небольшого фрагмента кода. Измерения проводятся для трех вариантов кода**,** делающих одно и то же: конкатенирующих десять тысяч строк в одну строку. В первом случае используется наиболее естественный**,** "лобовой" прием инкрементной конкатенации**,** во втором **-** накопление строк в списке с последующим объединением в одну строку**,** в третьем применяется списковое включение**,** а затем объединение элементов списка в одну строку:

```
from timeit import Timer

t = Timer("""
res = ""
```

```
for k in range(1000000,1010000):
  res += str(k)
""")
print t.timeit(200)

t = Timer("""
res = []
for k in range(1000000,1010000):
  res.append(str(k))
res = ",".join(res)
""")
print t.timeit(200)

t = Timer("""
res = ",".join([str(k) for k in range(1000000,1010000)])
""")
print t.timeit(200)
```

Разные версии **Python** дадут различные результаты прогонов:

```
# Python 2.3
77.6665899754
10.1372740269
9.07727599144

# Python 2.4
9.26631307602
9.8416929245
7.36629199982
```

В старых версиях **Python** рекомендуемым способом конкатенации большого количества строк являлось накопление их в списке с последующим применением функции `join()` **(**кстати**,** инкрементная конкатенация почти в восемь раз медленнее этого приема**).** Начиная с версии **2.4,** инкрементная конкатенация была оптимизирована и теперь имеет даже лучший результат**,** чем версия со списками **(**которая вдобавок требует больше памяти**).** Но чемпионом все**-**таки является работа со списковым включением**,** поэтому свертывание циклов в списковое включение позволяет повысить эффективность кода**.**

Если требуются более точные результаты**,** рекомендуется использовать метод `repeat(n, k)` **-** он позволяет вызывать `timeit(k)` `n` раз**,** возвращая список из `n` значений**.** Необходимо отметить**,** что на результаты может влиять загруженность компьютера**,** на котором проводятся испытания**.**

Оптимизация

Основная реализация языка **Python** пока что не имеет оптимизирующего компилятора**,** поэтому разговор об оптимизации касается только **оптимизации кода** самим программистом**.** В любом языке программирования имеются свои характерные приемы оптимизации кода**.** Оптимизация **(**улучшение**)** кода может происходить в двух **(**зачастую конкурирующих**)** направлениях**:** скорость и занимаемая память**.** В условиях достатка оперативной памяти приложения обычно оптимизируют по скорости**.** При оптимизации по времени программы для одноразового вычисления следует иметь в виду**,** что в общее время решения задачи входит не только выполнение программы**,** но и время ее написания**.** Не стоит тратить усилия на оптимизацию программы**,** если она будет использоваться очень редко**.**

Следует учитывать**,** что программа**,** реализующая некоторый алгоритм**,** не может быть оптимизирована до бесконечно малого времени вычисления**:** используемый алгоритм имеет

определенную **временную сложность** и программу, основанную на слишком сложном алгоритме, существенно оптимизировать не удастся. Можно попытаться сменить алгоритм (хотя многие задачи этого сделать не позволяют) или ослабить требования к решениям. Иногда помогает упрощение алгоритма. К сожалению, оптимизация кода, как и программирование - задача неформальная, поэтому умение оптимизировать код приходит с опытом.

Если скорость работы программы при большой длине данных не устраивает, следует поискать более эффективный алгоритм. Если же более эффективный алгоритм практически нецелесообразен, можно попытаться провести оптимизацию кода.

Собственно, в данном примере для модуля `timeit` уже показан практический способ нахождения оптимального кода. Стоит также отметить, что с помощью профайлера нужно определить места кода, отнимающие наибольшую часть времени. Обычно это действия, выполняемые в самом вложенном цикле. Можно попытаться вынести из цикла все, что можно вычислить в более внешнем цикле или вообще вне цикла.

В языке **Python** вызов функции является относительно дорогостоящей операцией, поэтому на критичных по скорости участках кода следует избегать вызова большого числа функций.

В некоторых случаях работу программы на **Python** можно ускорить в несколько раз с помощью специального оптимизатора (он не входит в стандартную поставку **Python**, но свободно распространяется): `psyco`. Для ускорения программы достаточно добавить следующие строки в начале главного модуля программы:

```
import psyco
psyco.full()
```

Правда, некоторые функции не поддаются "компиляции" с помощью `psyco`. В этих случаях будут выданы предупреждения. Посмотрите документацию по `psyco` с тем, чтобы узнать ограничения в его использовании и способы их преодоления.

Еще одним вариантом ускорения работы приложения является переписывание критических участков алгоритма на языках более низкого уровня (C/C++) и использование модулей расширения из **Python**. Однако эта крайняя мера обычно не требуется или модули для задач, требующих большей эффективности, уже написаны. Например, для работы с растровыми изображениями имеется прекрасная библиотека модулей **PIL (Python Imaging Library)**. Численные расчеты можно выполнять с помощью пакета **Numeric** и т.д.

Pychecker

Одним из наиболее интересных инструментов для анализа исходного кода **Python** программы является **Pychecker**. Как и **lint** для языка **C**, **Pychecker** позволяет выявлять слабости в исходном коде на языке **Python**. Можно рассмотреть следующий пример с использованием **Pychecker**:

```
import re, string
import re
a = "a b c"

def test(x, y):
  from string import split
  a = "x y z"
  print split(a) + x

test(['d'], 'e')
```

Pychecker выдаст следующие предупреждения:

```
badcode.py:1: Imported module (string) not used
badcode.py:2: Imported module (re) not used
badcode.py:2: Module (re) re-imported
badcode.py:5: Parameter (y) not used
badcode.py:6: Using import and from ... import for (string)
badcode.py:7: Local variable (a) shadows global defined on line
3
badcode.py:8: Local variable (a) shadows global defined on line
3
```

В первой строке импортирован модуль, который далее не применяется, то же самое с модулем `re`. Кроме того, модуль `re` импортирован повторно. Другие проблемы с кодом: параметр **y** не использован; модуль `string` применен как в операторе `import`, так и во `from-import`; локальная переменная `a` затеняет глобальную, которая определена в третьей строке.

Можно переписать этот пример так, чтобы **Pychecker** выдавал меньше предупреждений:

```
import string
a = "a b c"

def test(x, y):
  a1 = "x y z"
  print string.split(a1) + x

test(['d'], 'e')
```

Теперь имеется лишь одно предупреждение:

```
goodcode.py:4: Parameter (y) not used
```

Такое тоже бывает. Программист должен лишь убедиться, что он не сделал ошибки.

Исследование объекта

Даже самые примитивные объекты в языке программирования **Python** имеют возможности, общие для всех объектов: можно получить их уникальный идентификатор **(**с помощью функции `id()`**)**, представление в виде строки **-** даже в двух вариантах **(**функции `str()` и `repr()`**)**; можно узнать атрибуты объекта с помощью встроенной функции `dir()` и во многих случаях пользоваться атрибутом `__dict__` для доступа к словарю имен объекта. Также можно узнать, сколько других объектов ссылается на данный с помощью функции `sys.getrefcount()`. Есть еще сборка мусора, которая применяется для освобождения памяти от объектов, которые более не используются, но имеют ссылки друг на друга **(**циклические ссылки**)**. Сборкой мусора **(garbage collection)** можно управлять из модуля `gc`.

Все это подчеркивает тот факт, что объекты в **Python** существуют не сами по себе, а являются частью системы: они и их отношения строго учитываются интерпретатором.

Сразу же следует оговориться, что **Python** имеет две стороны интроспекции: **"**официальную**"**, которую поддерживает описание языка и многие его реализации, и

"неофициальную", которая использует особенности той или иной реализации. С помощью "официальных" средств интроспекции можно получить информацию о принадлежности объекта тому или иному классу (функция `type()`), проверить принадлежность экземпляра классу (`isinstance()`), отношение наследования между классами (`issubclass()`), а также получить информацию, о которой говорилось чуть выше. Это как бы приборная доска машины. С помощью "неофициальной" интроспекции (это то, что под капотом) можно получить доступ к чему угодно: к текущему фрейму исполнения и стеку, к байт-коду функции, к некоторым механизмам интерпретатора (от загрузки модулей до полного контроля над внутренней средой исполнения). Сразу же стоит сказать, что этот механизм следует рассматривать (и тем более вносить изменения) очень деликатно: разработчики языка не гарантируют постоянство этих механизмов от версии к версии, а некоторые полезные модули используют эти механизмы для своих целей. Например, упомянутый ранее ускоритель выполнения **Python**-кода `psyco` очень серьезно вмешивается во фреймы исполнения, заменяя их своими объектами. Кроме того, разные реализации **Python** могут иметь совсем другие внутренние механизмы.

Сказанное стоит подкрепить примерами.

В первом примере исследуется объект с помощью "официальных" средств. В качестве объекта выбрана обычная строка:

```
>>> s = "abcd"
>>> dir(s)
['__add__', '__class__', '__contains__', '__delattr__', '__doc__',
'__eq__', '__ge__', '__getattribute__', '__getitem__',
'__getnewargs__',
'__getslice__', '__gt__', '__hash__', '__init__', '__le__',
'__len__',
'__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__',
'__str__', 'capitalize', 'center', 'count', 'decode',
'encode', 'endswith', 'expandtabs', 'find', 'index', 'isalnum',
'isalpha',
'isdigit', 'islower', 'isspace', 'istitle', 'isupper', 'join',
'ljust',
'lower', 'lstrip', 'replace', 'rfind', 'rindex', 'rjust', 'rstrip',
'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title',
'translate', 'upper', 'zfill']
>>> id(s)
1075718400
>>> print str(s)
abcd
>>> print repr(s)
'abcd'
>>> type(s)
<type 'str'>
>>> isinstance(s, basestring)
True
>>> isinstance(s, int)
False
>>> issubclass(str, basestring)
True
```

"Неофициальные" средства интроспекции в основном работают в области представления объектов в среде интерпретатора. Ниже будет рассмотрено, как главная (на настоящий момент) реализация **Python** может дать информацию об определенной пользователем функции:

```
>>> def f(x, y=0):
...   """Function f(x, y)"""
...   global s
```

```
...   return t + x + y
...
>>> f.secure = 1     # присваивается  дополнительный атрибут
>>> f.func_name      # имя
'f'
>>> f.func_doc       # строка документации
'Function f(x, y)'
>>> f.func_defaults  # значения по умолчанию
(0,)
>>> f.func_dict      # словарь атрибутов функции
{'secure': 1}
>>> co = f.func_code # кодовый объект
>>> co
<code object f at 0x401ec7e0, file "<stdin>", line 1>
```

Кодовые объекты имеют свои атрибуты:

```
>>> co.co_code       # байт-код
't\x00\x00|\x00\x00\x17|\x01\x00\x17Sd\x01\x00S'
>>> co.co_argcount   # число аргументов
2
>>> co.co_varnames   # имена переменных
('x', 'y')
>>> co.co_consts     # константы
(None,)
>>> co.co_names      # локальные имена
('t', 'x', 'y')
>>> co.co_name       # имя блока кода (в нашем случае - имя
функции)
'f'
```

и так далее. Более правильно использовать для получения всех этих сведений модуль `inspect`.

Модуль inspect

Основное назначение модуля `inspect` - давать приложению информацию о модулях, классах, функциях, трассировочных объектах, фреймах исполнения и кодовых объектах. Именно модуль `inspect` позволяет заглянуть "на кухню" интерпретатора **Python**.

Модуль имеет функции для проверки принадлежности объектов различным типам, с которыми он работает:

Функция	Проверяемый тип
inspect.isbuiltin	Встроенная функция
inspect.isclass	Класс
inspect.iscode	Код
inspect.isdatadescriptor	Описатель данных
inspect.isframe	Фрейм
inspect.isfunction	Функция
inspect.ismethod	Метод
inspect.ismethoddescriptor	Описатель метода
inspect.ismodule	Модуль

inspect.isroutine	Функция или метод
inspect.istraceback	Трассировочный объект

Пример:

```
>>> import inspect
>>> inspect.isbuiltin(len)
True
>>> inspect.isroutine(lambda x: x+1)
True
>>> inspect.ismethod(''.split)
False
>>> inspect.isroutine(''.split)
True
>>> inspect.isbuiltin(''.split)
True
```

Объект типа модуль появляется в **Python-**программе благодаря операции импорта. Для получения информации о модуле имеются некоторые функции, а объект-модуль обладает определенными атрибутами, как продемонстрировано ниже:

```
>>> import inspect
>>> inspect.ismodule(inspect)
True
>>>
inspect.getmoduleinfo('/usr/local/lib/python2.3/inspect.pyc')
('inspect', '.pyc', 'rb', 2)
>>>
inspect.getmodulename('/usr/local/lib/python2.3/inspect.pyc')
'inspect'
>>> inspect.__name__
'inspect'
>>> inspect.__dict__
. . .
>>> inspect.__doc__
"Get useful information from live Python objects.\n\nThis
module encapsulates
.. . .
```

Интересны некоторые функции, которые предоставляют информацию об исходном коде объектов:

```
>>> import inspect
>>> inspect.getsourcefile(inspect)   # имя файла исходного кода
'/usr/local/lib/python2.3/inspect.py'
>>> inspect.getabsfile(inspect)      # абсолютный путь к файлу
'/usr/local/lib/python2.3/inspect.py'
>>> print inspect.getfile(inspect)   # файл кода модуля
/usr/local/lib/python2.3/inspect.pyc
>>> print inspect.getsource(inspect)  # исходный текст модуля
(в виде строки)
# -*- coding: iso-8859-1 -*-
"""Get useful information from live Python objects.
. . .
>>> import smtplib
>>> # Комментарий непосредственно перед определением объекта:
>>> inspect.getcomments(smtplib.SMTPException)
'# Exception classes used by this module.\n'
```

```
>>> # Теперь берем строку документирования:
>>> inspect.getdoc(smtplib.SMTPException)
'Base class for all exceptions raised by this module.'
```

С помощью модуля inspect можно узнать состав аргументов некоторой функции с помощью функции inspect.getargspec():

```
>>> import inspect
>>> def f(x, y=1, z=2):
...   return x + y + z
...
>>> def g(x, *v, **z):
...   return x
...
>>> print inspect.getargspec(f)
(['x', 'y', 'z'], None, None, (1, 2))
>>> print inspect.getargspec(g)
(['x'], 'v', 'z', None)
```

Возвращаемый кортеж содержит список аргументов (кроме специальных), затем следуют имена аргументов для списка позиционных аргументов (*) и списка именованных аргументов (**), после чего - список значений по умолчанию для последних позиционных аргументов. Первый аргумент-список может содержать вложенные списки, отражая структуру аргументов:

```
>>> def f((x1,y1), (x2,y2)):
...   return 1
...
>>> print inspect.getargspec(f)
([['x1', 'y1'], ['x2', 'y2']], None, None, None)
```

Классы (как вы помните) - тоже объекты, и о них можно кое-что узнать:

```
>>> import smtplib
>>> s = smtplib.SMTP
>>> s.__module__  # модуль, в котором был определен объект
'smtplib'
>>> inspect.getmodule(s)  # можно догадаться о происхождении
объекта
<module 'smtplib' from '/usr/local/lib/python2.3/smtplib.pyc'>
```

Для визуализации дерева классов может быть полезна функция inspect.getclasstree(). Она возвращает иерархически выстроенный в соответствии с наследованием список вложенных списков классов, указанных в списке-параметре. В следующем примере на основе списка всех встроенных классов-исключений создается дерево их зависимостей по наследованию:

```
import inspect, exceptions

def formattree(tree, level=0):
  """Вывод дерева наследований.
  tree - дерево, подготовленное с помощью
inspect.getclasstree(),
    которое представлено списком вложенных списков и кортежей.
```

```
    В кортеже entry первый элемент - класс, а второй - кортеж с
его
    базовыми классами. Иначе entry - вложенный список.
    level - уровень отступов
    """
    for entry in tree:
      if type(entry) is type(()):
        c, bases = entry
        print level * "  ", c.__name__, \
              "(" + ", ".join([b.__name__ for b in bases]) + ")"
      elif type(entry) is type([]):
        formattree(entry, level+1)

v = exceptions.__dict__.values()
exc_list = [e for e in v
            if inspect.isclass(e) and issubclass(e, Exception)]

formattree(inspect.getclasstree(exc_list))
```

С помощью функции `inspect.currentframe()` можно получить текущий фрейм исполнения. Атрибуты фрейма исполнения дают информацию о блоке кода, исполняющегося в точке вызова метода. При вызове функции (и в некоторых других ситуациях) на стек кладется соответствующий этому фрейму блок кода. При возврате из функции текущим становится фрейм, хранившийся в стеке. Фрейм содержит контекст выполнения кода: пространства имен и некоторые другие данные. Получить эти данные можно через атрибуты фреймового объекта:

```
import inspect

def f():
  fr = inspect.currentframe()
  for a in dir(fr):
    if a[:2] != "__":
      print a, ":", str(getattr(fr, a))[:70]

f()
```

В результате получается

```
f_back : <frame object at 0x812383c>
f_builtins : {'help': Type help() for interactive help, or
help(object) for help ab
f_code : <code object f at 0x401d83a0, file "<stdin>", line 11>
f_exc_traceback : None
f_exc_type : None
f_exc_value : None
f_globals : {'f': <function f at 0x401e0454>, '__builtins__':
<module '__builtin__
f_lasti : 68
f_lineno : 16
f_locals : {'a': 'f_locals', 'fr': <frame object at 0x813c34c>}
f_restricted : 0
f_trace : None
```

Здесь `f_back` - предыдущий фрейм исполнения (вызвавший данный фрейм), `f_builtins` - пространство встроенных имен, как его видно из данного фрейма, `f_globals` - пространство глобальных имен, `f_locals` - пространство локальных имен, `f_code` - кодовый объект (в данном случае - байт-код функции `f()`), `f_lasti` - индекс последней

выполнявшейся инструкции байт-кода, `f_trace` - функция трассировки для данного фрейма (или `None`), `f_lineno` - текущая строка исходного кода, `f_restricted` - признак выполнения в ограничительном режиме.

Получить информацию о стеке интерпретатора можно с помощью функции `inspect.stack()`. Она возвращает список кортежей, в которых есть следующие элементы:

```
(фрейм-объект, имя_файла, строка_в_файле, имя_функции,
список_строк_исходного_кода, номер_строки_в_коде)
```

Трассировочные объекты также играют важную роль в интроспективных возможностях языка **Python**: с их помощью можно отследить место возбуждения исключения и обработать его требуемым образом. Для работы с трассировками предусмотрен даже специальный модуль - `traceback`.

Трассировочный объект представляет содержимое стека исполнения от места возбуждения исключения до места его обработки. В обработчике исключений связанный с исключением трассировочный объект доступен посредством функции `sys.exc_info()` (это третий элемент возвращаемого данной функцией кортежа).

Трассировочный объект имеет следующие атрибуты:

- `tb_frame` Фрейм исполнения текущего уровня.
- `tb_lineno` и `tb_lasti` Номер строки и инструкции, где было возбуждено исключение.
- `tb_next` Следующий уровень стека (другой трассировочный объект).

Одно из наиболее частых применений модуля `traceback` - "мягкая" обработка исключений с выводом отладочной информации в удобном виде (в лог, на стандартный вывод ошибок и т.п.):

```
#!/usr/bin/python

def dbg_except():
  """Функция для отладки операторов try-except"""
  import traceback, sys, string
  print sys.exc_info()
  print " ".join(traceback.format_exception(*sys.exc_info()))

def bad_func2():
  raise StandardError

def bad_func():
  bad_func2()

try:
  bad_func()
except:
  dbg_except()
```

В результате получается примерно следующее:

```
(<class exceptions.StandardError at 0x4019729c>,
<exceptions.StandardError instance at 0x401df2cc>,
<traceback object at 0x401dcb1c>)
Traceback (most recent call last):
   File "pr143.py", line 17, in ?
```

```
    bad_func()
  File "pr143.py", line 14, in bad_func
    bad_func2()
  File "pr143.py", line 11, in bad_func2
    raise StandardError
StandardError
```

Функция `sys.exc_info()` дает кортеж с информацией о возбужденном исключении **(**класс исключения**,** объект исключения и трассировочный объект**).** Элементы этого кортежа передаются как параметры функции `traceback.format_exception()`**,** которая и печатает информацию об исключении в уже знакомой форме**.** Модуль `traceback` содержит и другие функции **(**о них можно узнать из документации**),** которые помогают форматировать те или иные части информации об исключении**.**

Разумеется**,** это еще не все возможности модуля `inspect` и свойств интроспекции в **Python,** а лишь наиболее интересные функции и атрибуты**.** Подробнее можно прочитать в документации или даже в исходном коде модулей стандартной библиотеки **Python.**

Заключение

С помощью возможностей интроспекции удается рассмотреть фазы работы транслятора **Python:** лексический анализ**,** синтаксический разбор и генерации кода для интерпретатора**,** саму работу интерпретатора можно видеть при помощи отладчика**.**

Вместе с тем**,** в этой лекции было дано представление об использовании профайлера для исследования того**,** на что больше всего тратится процессорное время в программе**,** а также затронуты некоторые аспекты оптимизации **Python-**программ и варианты оптимизации кода на **Python** по скорости**.**

Наконец**,** интроспекция позволяет исследовать не только строение программы**,** но и объектов**,** с которыми работает эта программа**.** Были рассмотрены возможности **Python** по получению информации об объектах **-** этом основном строительном материале**,** из которого складываются данные любой **Python-**программы**.**