Р.А. Сузи

Язык

программирования

Python

Щ**ИНТУИТ**

**НАЦИОНАЛЬНЫЙ ОТКРЫТЫЙ УНИВЕРСИТЕТ**

Язык программирования Python  
2-е издание, исправленное

Сузи Р.А.

Язык программирования Python  
2-е издание, исправленное

Сузи Р.А.

**Сузи RA. Язык программирования Python**

**УДК004.438РуіЬоп(07)**

**ББК24**

**С89**

**Язык программирования Python / Сузи Р.А. - М.: Национальный Открытый  
Университет "ИНТУИТ", 2016 (Основы информационных технологий)**

**ISBN 5-9556-0058-2**

**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.**

**(с) ООО "ИНТУИТ.РУ", 2006-2016  
(с) Сузи Р.А., 2006-2016**

Введение в программирование на языке Python

в этой лекции пойдет речь о синтаксисе языка Python для основных  
алгоритмических юнструкций, литералов, выражений. Буцет приведено  
описание встроенных типов данных, а также сделана попытка  
рассмотреть неюторые вопросы общепринятого в Python стиля  
программирования.

Что такое Python?

о РзДЬоп (лз^іше произносить "питон", хотя неюторые говорят "пайтон")  
- предмете данного изучения, лз^іше всего говорит создатель этого  
языка программирования, голландец Гвидо ван Россум:

'Python - интерпретируемый, объектно-ориентированный  
высоюуровневый язык программирования с динамичесюй семантиюй.  
Встроенные высошуровневые структуры данных в сочетании с  
динамичесюй 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 юмпонентов. Синтаксис  
Python прост в изздіении, в нем придается особое значение читаемости  
юда, а это сокращает затраты на сопровождение программных  
продуктов. Python поддерживает модули и пакеты, поощряя модульность  
и повторное использование юда. Интерпретатор Python и большая  
стандартная библиотека достзпны бесплатно в виде исждных и  
исполняемых юдов для всех основных платформ и могут свободно  
распространяться."

В процессе изучения буцет 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 поставіу РзДйоп входит обширная стандартная  
библиотека для решения широюго круга задач. В Интернете доступны  
качественные библиотеки для Python по различным предметным  
областям: средства обработки текстов и технологии Интернет,  
обработка изображений, инстрзлиенты для создания приложений,  
механизмы доступа к базам данных, пакеты для научных вычислений.

**Сузи RA. Язык программирования Python**

библиотеки построения графического интерфейса и т.п. Кроме того,  
Python имеет достаточно простые средства для интеграции с языками С,  
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 несюлью 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 С и на платформе Java.

Может показаться, что, в программной индустрии нет места для чего-то  
другого кроме C/C++, Java, Visual Basic, С#. Однаю это не так.  
Возможно, благодаря данному і^фсу лекций и практических занятий у  
Python появятся новые приверженцы, для юторых он станет  
незаменимым инстрзлиентом.

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  
академичесюго подхода.

Однаю стоит обратить внимание на правильный подход к описанию  
языка. Создание программы - это всегда юммуникация, в юторой  
программист передает юмпьютеру информацию, необходимзло для  
выполнения последним действий. То, как эти действия понимает  
программист (то есть "смысл"), можно назвать семантиюй. Средством  
передачи этого смысла является синтаксис языка программирования. Ну  
а то, что делает интерпретатор на основании переданного, обычно  
называют прагматиюй. При написании программы очень важно, чтобы  
в этой цепочке не возникало сбоев.

**Сузи RA. Язык программирования Python**

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 юмпонентом остается толью семантика.  
Именно в переводе смысла в формальное описание и кроется самая  
большая сложность программирования. Синтаксис языка Python  
обладает мощными средствами, юторые 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 Россз^ом (Guido van Rossum) в  
1991 году, югда он работал над распределенной ОС Амеба. Ему  
требовался расширяемый язык, юторый бы обеспечил поддержку  
системных вызовов. За основу были взяты АВС и Модута-З. В качестве  
названия он выбрал Python в честь юмедийных серий ВВС "Летающий  
цирк Монти-Питона", а вовсе не по названию змеи. С тех пор Python  
развивался при поддержке тех организаций, в юторых Гвидо работал.  
Особенно активно язык совершенствуется в настоящее время, югда над  
ним работает не толью юманда 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 несюльких  
модзлей. Каждый модуль представляет собой текстовый файл в  
юдировке, совместимой с 7-битной юдировюй ASCII. Для юдировок,  
использующих старший бит, необходимо явно зжазывать название  
юдировки. Например, мод}ль, юмментарии или строювые литералы  
юторого записаны в юдировке KOI8-R, должен иметь в первой или  
второй строке следующую спецификацию:

# -\*- coding: коі8-г -\*-

**Сузи RA. Язык программирования Python**

Благодаря этой специ4мкации 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  
лекциях. В примерах ниже используются как фрагменты модулей,  
записанных в файл, так и фрагменты диалога с интерпретатором РзДЬоп.  
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 (}мзической, но длинные логические строки можно явно (с  
помощью обратной косой черты) или неявно (внутри скобок) разбить на  
несколью физических:

print а," - очень длинная строка, которая не помещается в", \

80, "знакоместах"

Примечание:

Во всех примерах в основном используется "официальный" стиль  
оформления шда на Python в соответствии с доід^ентом 'Python Style  
Guide", юторый можно найти на сайте ссылка: http У/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  
ошибки начинающих программистов на Рздйоп), поэтому изучать  
синтаксис Python предпочтительнее на примерах, а не с помощью  
синтаксических диаграмм или форм Бэкуса-Наура.

**Сузи Р.А.**

**Язык программирования Python**

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:

а = 1  
**Ь = 2**а = а + Ь  
Ь = а - Ь  
а = а - Ь  
print (а, Ь)

Что делает этот пример? Проверить свою догаді^^ 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 а > Ь:  
с = а  
else:  
с = Ь

Этот IQ^COK юда 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.

**Сузи RA. Язык программирования Python**

Более общий слзліай - оператор выбора - можно записать с помощью  
следующего синтаксиса (пример вычисления знака числа):

if а < 0:

**S** = -1  
elif а == 0:

**5 = 0**

else:

**S** = 1

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

if а < 0:

**S =** -1  
else:

if а == 0:  
s = 0  
else:  
s = 1

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

Циклы

Третьей необждимой алгоритмичесюй констрзжцией является цикл. С  
помощью цикла можно описать повторяющиеся действия. В Python  
имеются два вида циклов: цикл ПОКА (выполняется некоторое  
условие) и цикл ДЛЯ (всех значений последовательности). Следующий  
пример иллюстрирует цикл ПОКА на РздЬоп:

**S** = "abcdefghijklmnop"  
whUe s !=  
print s  
s = s[l:-l]

Оператор **while** говорит интерпретатору Python: "пока верно условие

**Сузи RA. Язык программирования Python**

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 гибюсти при организации циклов применяются  
операторы **break** (прервать) и **continue** (продолжить). Первый  
позволяет прервать цикл, а второй - продолжить цикл, перейдя к  
следующей итерации (если, юнечно, выполняется условие цикла).

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

f = open("file.txt", 'Г)  
while ft

1 = f.readline()  
if noth  
break  
iflen(l)>5:  
print 1  
f.close()

В этом примере организован бесюнечный цикл, юторый прерывается  
только при получении из файла пустой строки ( **1** ), что обозначает  
конец файла.

В языке Python логичесюе значение несет каждый объект: **пупи,** пустые  
строки и последовательности, специальный объект **None** и логический  
литерал **False** имеют значение "ложъ", 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 for выполняет тело цикла для каждого элемента  
последовательности. В следующем примере выводится таблица

умножения:

for iinrange[l, 10):  
for j inrangefl, 10):  
print '%2i" % (i\*j),  
print

Здесь циклы for являются вложенными. Функция **range** **()**порождает список целых чисел из полуоткрытого интервала [ 1, **10).**

Перед каждой итерацией счетчик цикла ползліает очередное значение из  
этого списка. Полуоткрытые диапазоны общеприняты в РзДЬоп.  
Считается, что их использование более удобно и вызывает меньше  
программистских ошибок. Например, **range (Іеп** **(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 симвог.  
используется для обозначения операции взятия остатка от деления).  
Строка форматирования (задается слева) строится почти как строка  
форматирования для **print** f **из** С.

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 РзДЬоп, а здесь следует привести пример  
определения и вызова функции:

def сепа(шЬ, кор=0):  
return '%і руб. %і коп." % (шЬ, кор)

print сепа(8, 50)  
print сепа(7)

print сепа(шЬ=23, кор=70)

В этом примере определена функция двух аргументов (из которых  
второй имеет значение по умолчанию - 0 ). Вариантов вызова этой

П

**Сузи RA. Язык программирования Python**

функции с конкретными параметрами также несколью. 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 фзтікции можно возвратить  
только один объект , но он может быть кортежем из нескольких  
объектов.

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

Исключения  
**в современных программах передача управления происходит не всегда  
так гладю, как в описанных выше юнструкциях. 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:**

tiy:

res = int(open('a.txt').read()) / int(open('c.txt').read())  
print res  
except lOError:  
print "Ошибка ввода-вывода"  
except ZeroDivisionError:  
print "Деление на О"  
except Keyboardinterrupt:  
print "Прерывание c клавиатзфы"  
except:

print "Ошибка"

В этом примере берутся числа из дв}лс файлов и делятся одно на другое.  
В результате этих нехитрых действий может возникнуть несюлько  
исключительных ситуаций, неюторые 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

І2

**Сузи RA. Язык программирования Python**

в одном из файлов наждится 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 дрзтих языков программирования, в Python исключения  
нередко служат для зпрощения алгоритмов. Записывая оператор **try-  
except,** программист может думать так: "попробзло, а если сорвется -  
выполнится код в **except** ". Особенно часто это используется для  
выражений, в которых значение полз^ается по ключу из отображения:

tiy:

value = dict[key]  
except:

value = default\_value  
Вместо

if dict.has\_key(key):  
value = dict[key]  
else:

value = default value

Примечание:

Пример уже несколью устаревшей идиомы языка Python иллюстрирует  
толью дух этого подхода: в современном Python лз^ше записать так

**value = diet.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

**Язык программирования Python**

**Сузи Р.А.**

**tiy:**

raise МуЕітог, "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 юде он не выполняется, поэтому строить на нем  
логику алгоритма нельзя. Пример:

с = а + Ь  
assert с == а + Ь

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"

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 специальнзло  
поддержку в синтаксисе языка: можно записать литерал строки, числа,

**І4**

**Сузи RA. Язык программирования Python**

списка, кортежа, словаря (и их разновидностей). Синтаксичесідло же  
поддержку операций над встроенными типами можно легко сделать  
доступной и для объектов определяемых пользователями классов.

Следует также отметить, что объекты могут бытъ 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;  
о целые

* обычное целое int
* целое произвольной точности **long**
* логический **bool**

о число с плавающей точюй **float**о комплексное число complex  
последовательности;  
о неизменчивые:

* строка S t г ;
* Unicode-строка **Unicode** ;
* юртеж **tuple;**о изменчивые:
* список **list;**отображения:

**о** словарь **diet**

объекты, юторые можно вызватъ:

**О**

**О**

**О**

**о**

о 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 ;**

**Сузи RA. Язык программирования Python**

• 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 С 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, ОхА, 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 **1412**

»> print 2L \*\* 1000

1071508607186267320948425049060001810561404811705533607443750

8370351051124936122493198378815695858127594672917553146825187

5285692314043598457757469857480393456777482423098542107460506

7114187795418215304647498358194126739876755916554394607706291

71196477686542167660429831652624386837205668069376

»> 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

**»>fori,jin(0, 0), (0, 1),(1,** 0), (1, 1):

... print i, j,i & j, i I j, i Л j # побитовые операции

**00:000**0 **1:011  
10:011  
11:110**

**Сузи RA. Язык программирования Python**

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 ползліается 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. Нзии в начале  
числа - признак восьмеричной системы счисления, в шторой нет  
цифры 8:

»> 008

File "<stdin>", line 1  
008

**л**

SyntaxError: invalid token

Тип float

Соответствует С-типу **double** для используемой архитектуры.  
Записывается вполне традиционным способом либо через точі^^, либо в  
нотации с экспонентой:

»> рі= 3.1415926535897931  
»> pi \*\* 40

7.6912142205156999e+19

Кроме арифметических операций, можно использовать операции из  
модзия **math.**

Примечание:

Из полезных встроенных фзтікций можно вспомнить **round** **(),  
abs** **().**

Тип complex

Литерал мнимой части задается добавлением j в качестве суф4мкса  
(перемножаются мнимые единицы):

»> -Ij \* -Ij  
(-1-Oj)

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

Тип bool

Подтип целочисленного типа для "каноничесюго" 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 логичесюго типа:

»> for і 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 РзДЬоп даже не вычисляет второй операнд  
операции and или or, если ее исход ясен по первому операнду Таким

**Сузи RA. Язык программирования Python**

образом, если первый операнд истинен, 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 строювых литералов.  
Для литералов наравне используются как апострофы ( ' ), так и  
обычные двойные кавычки ( " ). Для многострочных литералов можно  
использовать утроенные апостро4ы или утроенные кавычки.  
Управляющие последовательности внутри строковых литералов  
задаются обратной юсой чертой ( \ ). Примеры написания строювых  
литералов:

si = "строкаі"

s2 = 'строка2\пс переводом строки внутри'  
s3 = """строкаЗ

с переводом строки внутри"""

и1 = u'\u043hu0440\u0438\u0432\u0435\u0442' # привет

u2 = и'Еще пример' # не забудьте про coding!

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

**ту\_ге = r"(\d)=\l"**

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

Набор операций над строками включает юнкатенацию " + ", повтор " \*  
", форматирование " % ". Также строки имеют большое юличество  
методов, неюторые из юторых приведены ниже. Полный набор

**І9**

**Сузи RA. Язык программирования Python**

методов (и их необязательных аргументов) можно полздіить в  
документации по Python.

»> "А" + 'В"

'АВ'

»> "А"\* 10  
'АААААААААА'

»> "%s %і" % ("abc", 12)

'abc 12'

Неюторые 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:

р = (1.2, 3.4, 0.9) # точка в трехмерном пространстве  
for **S** in "one", "two", "three": # цикл по значениям юртежа  
print s

one\_item = (1,)  
empty = 0

pi = 1, 3, 9 # без скобок

p2 = 3, 8, 5, # запятая в юнце игнорируется

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

а, Ь = Ь, а

Тип list

**Сузи RA. Язык программирования Python**

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:

Istl = [1, 2, 3,]

lst2 = [х\*\*2 for X in range(lO) if x % 2 == 1]

Ist3 = list("abcde")

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

По следовательно сти

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

Синтаксис Семантика

Іеп (S) Длина последовательности s

Проверка принадлежности элемента последовательности.  
X in S В новых версиях Python можно проверять принадлежность  
подстроки строке. Возвращает **True или False**

**X not in**

**=not** **X in** s  
s

**s + si**

**s \* n или  
n\* s  
S [i]**

**s[i:j:d]**

Конкатенация последовательностей

Последовательность из n раз повторенной s. Если п < О,  
возвращается пустая последовательность.

Возвращает і-й элемент s или Іеп (s) - **і-й,** если і < О

Срез из последовательности s от і до j с шагом d бздет  
рассматриваться ниже

min (S) Наименьший элемент s  
max (s) Наибольший элемент s

S [ і ] = X і -й элемент списка s заменяется на х  
S [ і : 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

Неюторые методы для работы с последовательностями

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

**Метод**

**, append(х)**

**, count(х)**

**, extend(s)**

**, index(х)**

**, insert (i, x)  
pop(i)**

**, reverse(s)**

**, sort([cmpfunc])**

Описание

Добавляет элемент в юнец  
последовательности

Считает юличество элементов, равных х

Добавляет к юнцу последовательности  
последовательность s

Возвращает наименьшее і, таюе, что s [ і ]

**== X.** Возбуждает исключение **ValueError,**если X не найден в s

Вставляет элемент х в і -й промежуток  
Возвращает і -й элемент, удаляя его из  
последовательности

Меняет порядок элементов s на обратный  
Сортирует элементы s. Может быть зжазана  
своя функция сравнения **cmpfunc**

Взятие элемента по индексу и срезы

Здесь же следует сказать несюлью слов об индексировании  
последовательностей и выделении подстрок (и вообще  
подпоследовательностей) по индексам. Для получения отдельного  
элемента последовательности используются квадратные сюбки, в  
юторых стоит выражение, дающее индекс. Индексы

**Сузи RA. Язык программирования Python**

последовательностей в Python начинаются с нуля. Отрицательные  
индексы служат для отсчета элементов с юнца последовательности (  
-1 - последний элемент). Пример проясняет дело:

»> **S** = [О, 1, 2, 3, 4]

»> print s[0], s[-l], s[3]

043

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

»> del s[2]

»> print s  
[0, 1, 3, 4]

Примечание:

і^алять элементы можно толью из изменчивых последовательностей  
и желательно не делать этого внутри цикла по последовательности.

Несюлью 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  
**о,** после него - **1** и т.д.. Отрицательные значения отсчитывают  
промежутки с юнца строки. Для записи срезов используется следующий  
синтаксис:

последовательность[нач:юніінг]

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

А теперь пример работы со срезами:  
**»> S** = range(lO)

**»> S**

**Сузи RA. Язык программирования Python**

[О, 1, 2, 3, 4, 5, 6, 7, 8, 9]

»> s[0:3]

[О, 1, 2]

»> s[-l:]

[9]

»> s[::3]

[О, 3, 6, 9]

»> s[0:0] = **[-l,-l,-l]**

**»> s**

1. **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 любзто  
подстроку, даже если она нулевой длины, как для удаления элементов,  
так и для вставки в строго определенное место.

Тип diet

Словарь (хэщ 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'}  
**do** = {0: 'zero'}

print **d[l]** # берется значение по ключу

d0[0] = 0 # присваивается значение по ключу

del d0[0] # удаляется пара ключ-значение с данным ключом

print d

for key, val in d.items(); # цикл no всему словарю  
print key, val

for key in d.keys(): # цикл no ключам словаря  
print key, d[key]

for val in d.values(): # цикл no значениям словаря

**Сузи Р.А.**

print val

d.update(dO) # пополняется словарь из другого  
print len(d) # юличество пар в словаре

Тип file

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

Следующий пример показывает копирование файла:

fl = openC'filel.txt", 'Г)  
**й =** open("file2.txt", Ѵ")  
for line in fl.readlines():

1. write(line)

G.cfoseQ

fl.ctoseQ

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, юпирование содержимого  
по ссылке (URL) в файл f **І1е2** . **txt** можно достигнутъ, если заменитъ  
первую строід^ на

import urllib

fl = urllib.urlopen("httpy/python.onego.ru")

О модулях, классах, объектах и фзлікциях будет говоритъся на других  
лекциях.

Выражения  
**в современных языках программирования принято производитъ  
большую часть обработки данных в выражениях. Синтаксис выражений  
у многих языков программирования примерно одинаков. Синтаксис**

**Сузи RA. Язык программирования Python**

выражений Python не уцивит программиста чем-то новым. (Разве что  
цепочечные сравнения могут приятно порадовать.)

Приоритет операций показан в нижеследующей таблице (в порядке  
увеличения). Для унарных операций х обозначает операнд.  
Ассоциативность операций в Python - слева-направо, за исключением  
операции возведения в степень ( \*\* ), юторая ассоциативна справа  
налево.

**<<, >>**

**+ r -  
\*, /, %**

**+ X, -X**

**~x**

**\***

**X.атрибут  
X[индекс]**

**X[от:до]  
f (аргумент, .  
(...)**

[ **. . .** ]

**{кл:зн, . . . }**

**Операция**

**lambda  
or  
and  
not X**

**in, not in  
is, is not  
<,<=,>,>=,!=,**

Название

лямбда-выражение  
логическое ИЛИ  
логическое И  
логическое НЕ  
проверка принадлежности  
проверка идентичности  
сравнения  
побитовое ИЛИ  
побитовое исключающее ИЛИ  
побитовое И  
побитовые сдвиги  
сложение и вычитание  
З^ножение, деление, остаток  
Зтзарный плюс и смена знака  
побитовое НЕ  
возведение в степень  
ссылка на атрибут  
взятие элемента по индексу  
выделение среза (от и до)  
вызов функции  
сюбки или кортеж  
список или списковое включение  
словарь пар ключ-значение  
**26**

**Язык программирования Python**

**Сузи Р.А.**

**'выражения**

преобразование к строке (герг)

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 **а<Ь<с** **...** **у < z** фактически  
   равносильна: (а < Ь) and (Ь < с) and . . . and (у <  
   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  
   таюе количество операндов, юторое достаточно для получения  
   результата. В невычисленной части выражения в таком случае  
   могут даже быть неопределенные имена. Это важно учитывать,  
   если используются фзтзкции 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 сюбки.  
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 следзлощем примере скобки просто  
необходимы, чтобы операция умножения произошла перед операцией  
форматирования:

print "%і" % (i\*j)

Выражения могут (}мгурировать во многих операторах Python и даже как  
самостоятельный оператор. У выражения всегда есть реззиьтат, хотя в  
некоторых случаях (югда 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 С)  
нет операции присваивания, поэтому синтаксически перед знаком =

**Сузи RA. Язык программирования Python**

могут стоять толью иденти**4**»**1**катор, индекс, срез, доступ к атрибуту или  
шртеж (список) из перечисленного. (Подробности в доіументации).

Имена

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

Имя может начинаться с латинсюй 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', 'finalfy', 'for', 'from',

'global', 'if, 'import', 'in', 'is', 'lambda', 'not', 'or',

'pass', 'print', 'raise', 'return', 'tiy, '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 юнце 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 РздЬоп находит значение неюторой  
переменной, необходимо ввести понятие блока юда. В Python блоюм  
юда является то, что исполняется как единое целое, например, тело  
определения фзлікции, класса или мод}ия.

Локальные имена - имена, юторым присвоено значение в данном блоке

**Сузи RA. Язык программирования Python**

юда. Глобальные имена - имена, определяемые на уровне блока юда  
определения модзия 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 фзтзкции видны имена, определенные в  
вызывающем юде. 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  
проиоюдит в операторах присваивания, **from, import,** в формальных  
аргументах функций, при определении фзтзкции или класса, во втором  
параметре части **except** оператора **try-except.**

С областями видимости и связыванием имен есть много нюансов,  
юторые 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 модзия должны  
   остаться только имена-константы, имена классов и функций.
3. Ниюгда не следует использовать **from** модуль **import** \* - это  
   может привести к затенению имен из других модзлей, а внутри  
   определения функции просто запрещено.

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

Убрать связь имени с объектом можно с помощью оператора **del.** В  
этом слзліае, если объект не имеет других ссылок на него, он будет  
удален. Для зшравления памятью в Python используется подсчет ссылок  
(reference counting), для удаления наборов объектов с зацикленными

**Сузи Р.А.**

ссылками - сборка мусора (garbage соДесііоп).

Стиль программирования

Стиль программирования - дополнительные ограничения,  
накладываемые на структзфу и вид программного юда грзппой  
совместно работающих программистов с целью получения уцобных 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 юда:

* именование объектов в зависимости от типа, назначения, области  
  видимости;
* оформление функций, методов, классов, модулей и их  
  документирование в юде программы;
* деюмпозиция 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 шмпьютерными платформами;
* ограничение используемых функций из соображений  
  безопасности.

Для языка Python Гвидо ван Россзли разработал официальный стиль. С  
оригинальным текстом 'Python Style Guide" можно ознаюмиться по  
адресу ссылка: httpy/www.p**3**^hon.org^doc/essays/styleguide.

Наиболее существенные положения этого стиля перечислены ниже. В  
случае сомнений хорошим образцом стиля являются мод}ии

**Сузи Р.А.**

стандартной библиотеки.

* Рекомендуется использовать отступы в 4 пробела.
* Длина физичесюй строки не должна превышать 79 символов.
* Длинные логические строки л}ліше разбивать неявно (внутри  
  сюбок), 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 неюторые 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))

* He реюмендуется ставить пробелы сразу после открывающей  
  сюбки или перед закрывающей, перед запятой, точкой с запятой,  
  перед открывающей скобкой при записи вызова фзтікции или  
  индексного выражения. Также не реюмендуется ставить более  
  одного пробела вокруг знака равенства в присваиваниях. Пробелы  
  вокруг знака равенства не ставятся в слздіае, югда он применяется  
  для указания значения по умолчанию в определении параметров  
  функции или при задании именованных аргументов.
* Также реюмендуется применение одиночных пробелов вокруг  
  низюприоритетных операций сравнения и оператора  
  присваивания. Пробелы вокруг более приоритетных операций  
  ставятся в равном юличестве слева и справа от знака операции.

Несюлью реюмендаций касаются написания юмментариев.

• Комментарии должны точно отражать актуальное состояние юда.  
(Поддержание актуальных юмментариев должно быть  
приоритетной задачей!) После юротких юмментариев можно не

**Сузи RA. Язык программирования Python**

ставить точку, тогда как длинные лучше писать по правилам  
написания текста. Автор Python обращается к неанглоязычным  
программистам с просьбой писать комментарии на английсюм,  
если есть жтя бы небольшая вероятность того, что юд будут  
читать специалисты, говорящие на других языках.

* Комментарии к фрагменту юда следует писать с тем же отступом,  
  что и комментируемый юд. После " # " должен идти одиночный  
  пробел. Абзацы можно отделять строюй с " # " на том же зфовне.  
  Блочный юмментарий можно отделить пустыми строками от  
  окружающего юда.
* Комментарии, относящиеся к юнкретной строке, не следует

использовать часто. Символ " # " должен отстоять от

юмментируемого оператора как минимум на два пробела.

* Хороший юмментарий не перефразирует программу, а содержит  
  дополнительную информацию о действии программы в терминах  
  предметной области.

Все модули, классы, фзлікции и методы, предназначенные для  
использования за пределами модзия, 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 реюмендуется везде использовать  
  утроенные кавычки ( " " " ).
* Однострочная доіументация пишется в императиве, как юманда:  
  "делай это", "возвращай то".
* Многострочная документация содержит расширенное описание  
  модуля, функции, класса. Она будет смотреться лздіше, 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 кратю  
  описывать действия функции, ее входные параметры и

**Сузи RA. Язык программирования Python**

возвращаемое значение, побочные эффекты и возможные  
исключения (если таковые есть). Должны быть обозначены  
необязательные аргзлиенты и аргументы, не являющиеся частью  
интерфейса.

* Документация для класса должна перечислять общедоступные  
  методы и атрибуты, содержать рекомендации по применению  
  класса в качестве базового для дрзтих классов. Если класс является  
  подклассом, необждимо 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 несколью более или менее широю  
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 бзлжами, например,  
**shelve, string,** либо делать первые буквы слов заглавными,  
**StringlO, UserDict.** Имена написанных на С модзлей  
расширения обычно начинаются с подчеркивания " \_ ", а  
соответствующие им высокоуровневые обертки - с прописных  
букв:**\_tkinter** и **Tkinter.**

Ключевые слова нельзя использовать в качестве имен, однаю,  
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 "еп'ог"  
(или "warning"). Встроенные модули пищут это слово со строчной

**Сузи RA. Язык программирования Python**

буквы (как **OS. error** ) (но могут писать и с прописной):  
**distutils.DistutilsModuleError.**

* Функции, экспортируемые модзием, 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 **f rom-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 языюм 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 достаточно проста. Были  
рассмотрены основные операторы языка, выражения и многие из  
встроенных типов данных, кратю объяснены принципы работы Python  
с именами, приведены правила о**4**мциального стиля программирования  
на Python.

Основные стандартные модули Python

Лекция знаюмит с наиболее важными модулями и пакетами  
стандартных библиотек Python в мере, достаточной для свободного  
ориентирования в них.

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

Понятие модуля

Перед тем как приступить к иззліению модзией стандартной  
библиотеки, необходимо определить то, что в Python называется  
модзием.

В соответствии с модзльным подходом к программированию большая  
задача разбивается на несюлью более мелких, каждую из юторых (в  
идеале) решает отдельный модуль. В разных методологиях даются  
различные ограничения на размер модзлей, однаю при построении  
модзльной структуры программы важнее составить такую композицию  
модзлей, 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 фзлікций, классов и констант, юторый он предлагает  
своим пользователям.

В языке Python набор модзлей, посвященных одной проблеме, можно  
поместить в пакет. Хорошим примером таюго пакета является пакет  
**xml,** в ютором собраны модули для различных аспектов обработки  
XML.

В программе на Python модзль представлен объектом-модулем,  
атрибутами юторого являются имена, определенные в модзле:

»> import datetime

»> dl = datetime.date(2004,11, 20)

**Сузи RA. Язык программирования Python**

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 мод}ли  
расширения, написанные на другом языке программирования (как  
правило, на С). С точки зрения пользователя они могут отличаться  
разве что быстродействием. Бывает, что в стандартной библиотеке есть  
два варианта модзля: на РзДйоп и на С. Таковы, например, мод}ли  
**pickle** и **сРіскІе.** Обычно модули на Python в чем-то гибче, чем  
модзли расширения.

Модутш в Python

Модуль оформляется в виде отдельного файла с исходным юдом.  
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/tocal/lib/p3^hon23.zip', Vusr/local/lib/python2.3',  
7usr/local/lib/python2.3/plat-linux2', Vusr/loca]/lib/python2.3/lib-tk',  
7usr/local/lib/python2.3/lib-dynload',

7usr/local/lib/python2.3/site-packages']

В последних версиях Python модзли 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 теіущем каталоге.  
(Нужно внимательно называть собственные модзли, чтобы не было  
конфликта имен со стандартными или дополнительно установленными  
модзлями.)

Подключение модзля к программе на Python осуществляется с помощью  
оператора **import.** У него есть две формы: **import и f rom-import:**

import os

**Сузи RA. Язык программирования Python**

import рге as re

from sys import argv, environ

from string import \*

C 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  
модзля связываются с текущей областью видимости. При импорте  
можно изменить имя, с юторым объект будет связан, с помощью as. В  
первом слзліае 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 происждит гораздо быстрее, так как мод}ли  
кэшируются интерпретатором. Загруженный мод}ль можно загрузить  
еще раз (например, если модуль изменился на диске) с помощью  
функции **reload** ():

import mynaodule

reload(mynaodule)

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

При работе с модулями естъ и другие тонкости. Например, сам процесс  
импорта модзля можно переопределитъ. Подробнее об этом можно  
узнать в оригинальной документации.

Встроенные функции

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

**Сузи RA. Язык программирования Python**

1. Функции преобразования типов и классы: **coerce, str, repr,  
   int, list, tuple, long, float, complex, diet, super,  
   file, bool, object**
2. Числовые и строковые функции: abs, **divmod, ord,** pow, **len,  
   chr, unichr,** hex, **oct,** emp, **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**

1. Функции для доступа к внутренним структурам: **locals,  
   globals, vars, intern,** **dir**
2. Функции юмпиляции и исполнения: **еѵаі, exeefile,**

**reload, import , compile**

1. Функции ввода-вывода: **input, raw\_input, open**
2. Функции для работы с атрибутами: **getattr, setattr,  
   delattr, hasattr**
3. Функции-"украшатели" методов классов: **statiemethod,**

**classmethod, property**

1. 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 фзлікций 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 строювыми аргументами. В  
следующей таблице даны описания этих функций.

abs (х) Модуль числа х. Результат: | х |.

di vmod (X, Частное и остаток от деления. Результат: (частное,  
у) остаток).

pow (X, у [, Возведение х в степень у по модзию т. Результат: х \* \* у

т] ) % т.

**Z] )**

**ord(s)  
chr(n)**

**round** **(п** **[,** Окрзлление чисел до заданного знака после (или до)  
точки.

Функция возвращает код (или Unicode) заданного ей  
символа в односимвольной строке.

Возвращает строіу^ с символом с заданным юдом.

**Язык программирования Python**

**Unicode(  
[,**

**encoding  
errors]]**

**Сузи Р.А.**

**len (s)**

**oct(n),  
hex(n)**

**cmp(x, y)  
unichr(n)**

Возвращает число элементов последовательности или  
отображения.

Функции возвращают строі^^ с восьмеричным или  
шестнадцатеричным представлением целого числа п.

Сравнение двух значений. Реззльтат: отрицательный,  
ноль или положительный, в зависимости от реззльтата  
сравнения.

**Возвращает односимвольную Unicode-CTpoiQ^ с  
символом с кодом п.**

Создает Unicode-объект, соответствующий строке s в  
заданной юдировке encoding. Ошибки кодирования  
обрабатываются в соответствии с en'ors, 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 юдировки кириллических букв в  
Unicode:

print 'Таблица Unicode (русские буквы)".сетег(18\*4)  
і**=0**

for с in "АБВГДЕЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯ'\  
"абвгдежзийклмнопрстуфхцчшщьыьэюя":

**U = unicode(c, 'коі8-г')**

print "%3i: %ls %s" % (ord(u), c, 'u'),

**i+= 1**

**ifi%4==0:**

print

функции обработки данных

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

**Сузи RA. Язык программирования Python**

**и enumerate** **():**

»> for i, с in enumerate("ABC"):

... print i, c

1. A
2. В  
   2C

»> print range(4, 20, 2)

[4, **6**, **8**, 10, 12, 14, 16, 18]

Функции определения свойств

Эти функции обеспечивают доступ к неюторым встроенным атрибутам  
объектов и дрзтим свойствам. Следзаощий пример показывает  
некоторые из этих функций:

**»> S = "abcde"**

»> si = "abcde"

»> s**2** = "ab" + "cde"

»> print "hash:", hash(s), hash(sl), hash(s**2**)  
hash: -1332677140-1332677140-1332677140  
»> print "id:", id(s), id(sl), id(s**2**)  
id: 1076618592 1076618592 1076618656

Здесь, можно увидеть, что для одного и того же строювого литерала  
" abode " получается один и тот же объект, тогда как для одинаковых по  
значению объектов вполне можно получитъ 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 О. Правда, записывать что-либо в эти словари не  
рекомендуется.**

Функция **varsO** возвращает таблицу локальных имен неюторого

**Сузи RA. Язык программирования Python**

объекта (если параметр не задан, она возвращает то же, что и  
**locals** О ). Обычно используется в качестве словаря для операции  
форматирования:

а = **1  
Ь = 2**с = 3

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

Функции компиляции и исполнения

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

а = **2**Ь = 3

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

Функцией **eval** **()** легю злоупотребить. Нужно стараться использовать  
ее толью тогда, югда без нее не обойтись. Из соображений  
безопасности не следует применять **eval** **()** для аргумента, в ютором  
присутствует непроверенный ввод от пользователя.

**Сузи Р.А.**

Функции ввода-вывода

Функции **input о и raw\_input()** использзтотся для ввода со  
стандартного ввода. В серьезных программах их лз^іше не применять.  
Функция **open** О служит для открытия файла по имени для чтения,  
записи или изменения. В следующем примере файл открывается для  
чтения:

f = open("file.txt", **Y\** 1)  
for line in f:

f.close()

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

**является синонимом для**

В новых версиях Python функция **ореп(;  
file О .**

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 А:  
pass  
а = А()

1. attr = **1**tiy:

print a.attr  
except:

**Сузи Р.А.**

print None  
dela.attr

# вторая программа:  
class А:  
pass  
а = А()

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 c OC (файлы, процессы). Модзли: os, os . **path,  
   getopt, glob, popen2, shutil, select, signal, stat,  
   tempi ile.**
4. Обработка текстов. Модули: **string, re, StringlO, codecs,  
   difflib,** mmap, **sgmllib, htmllib, htmlentitydefs.**Пакет **xml.**
5. Многопоточные вычисления. Модули: **threading, thread.**

**Queue.**

1. Хранение данных. Архивация. Модзии: pickle, **shelve,  
   anydbm, gdbm, gzip, zlib, zipf lie, bz2,** csv, **tarf lie.**
2. Платформо-зависимые модзии. Для UNIX: commands, pwd, **grp,  
   fcntl, resource, termios, readline, rlcompleter.**Для Windows: **msvcrt, \_winreg, winsound.**
3. Поддержка сети. Протоюлы Интернет. Модзии: **cgi. Cookie,**

**urllib, urlparse, httplib, smtplib, poplib,  
telnetlib, socket, asyncore.** Примеры серверов:  
**SocketServer, BaseHTTPServer, xmlrpclib,**

**asynchat.**

1. Поддержка Internet. Форматы данных. Модзяи: **quopri, uu,  
   base64, binhex, binascii, rfc822, mimetools,  
   MimeWriter, multifile,** **mailbox.** Пакет **email.**
2. Python **0** себе. Модзии: parser, **symbol, token, keyword,**inspect, **tokenize, pyclbr, py\_compile, compileall,**dis, **compiler.**
3. Графический интерфейс. Модзяь **Tkinter.**

Примечание:

Очень часто модзли содержат один или несколью классов, с помощью  
юторых создается объект нужного типа, а затем речь идет уже не об  
именах из модзля, а об атрибутах этого объекта. И наоборот, некоторые  
модули содержат лишъ функции, достаточно общие для того, чтобы  
работатъ над произволъными 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 попзлярные

Выход из программы. Можно  
передать числовой юд завершения:  
О в случае успешного завершения,  
другие числа при аварийном  
завершении программы.

**exit([с])**

**argv**

Список аргументов юмандной  
строки. Обычно sys . **argv** [ О ]  
содержит имя запущенной  
программы, а остальные параметры  
передаются из командной строки.

**platform**

Платформа, на юторой работает  
интерпретатор.

Стандартный ввод, вывод, вывод  
**stdin, stdout, stderr** ошибок. Открытые файловые

объекты.

**version** Версия интерпретатора.

Установка уровня максимальной  
set **recur si on limit (limit)** 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:

Istl = [О, О, 0]

1st = [Istl] \* 3  
print **1**st  
lst[**0**][l] = **1**print **1**st

**[[О, О, 0], [0, о, 0], [0, о, 0]]**

**[[О, 1, 0], [О, 1, 0], [О, 1, 0]]**

Дело в том, что список 1st содержит ссылки на один и тот же список!  
Для того чтобы действительно размножить список, необждимо  
применить функцию **сору** **()** **из** модуля **сору:**

from сору import сору  
Istl = **[0, о, 0]**

1st = [copy(lstl) for i in range(3)]  
print **1**st  
lst[**0**][l] = **1**print **1**st

Теперь результат тот, юторый ожидался:

**[[**0**, о,** 0**], [**0**, о,** 0**], [**0**, о,** 0**]]**

**[[**0**,** 1**,** 0**], [**0**, о,** 0**], [**0**, о,** 0**]]**

В модуле сору есть еще и фзлікция **deepcopy** **()** для глз^окого  
копирования, при юторой объекты копируются на всю возможнзло  
глубину, рекурсивно.

Модули math и cmath

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

**Описание**

Функция или  
константа

**acos (z)** аркюсинус z

as in (z) арксинус z

**atan** **(z)** арктангенс z

**atan2(y,x) atan(y/x)**

**Язык программирования Python**

**Сузи Р.А.**

**ceil(х)  
cos (z)  
cosh(x)  
e**

**exp(z)  
fabs(x)  
floor(x)  
fmod(x,у)**

**frexp(x)**

**hypot(x,y)  
Idexp(m,i)  
log (z)  
loglO(z)**

**modf(x)**

**pi**

**pow(x,y)  
sin (z)  
sinh (z)  
sqrt(z)  
tan (z)  
tanh (z)**

наименьшее целое, большее или равное х  
юсинус **Z**

гаперболический юсинус х  
юнстанта е

экспонента (то есть, е \* \* z )  
абсолютное значение х  
наибольшее целое, меньшее или равное х  
остаток от деления х на **у**

возвращает мантиссу и порядок х как пару (т, і), где  
m - число с плавающей точюй, а і - целое, таюе, что х  
= m \* 2 . \* \* і. Если О, возвращает **(0,0),** иначе 0.5  
<= **abs(m) < 1.0  
sqrt(х\*х + у\*у)**m \* (**2\*\*1**)

натуральный логарифм z  
десятичный логарифм z

возвращает пару (у, q) - целую и дробную часть х.

Обе части имеют знак исждного числа

юнстанта пи  
**X\* \*у**синус **Z**

гиперболический синус z  
юрень квадратный от z  
тангенс z

гиперболический тангенс z

Модуль random

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

**random()**

Генерирует псевдослзліайное число из  
полуоткрытого диапазона [**0**.**0**, **1**.**0**]

**Язык программирования Python**

**Сузи Р.А.**

**choice(s)**

Выбирает слз^айный элемент из  
последовательности s.

**shuffle (s)**

Размешивает элементы изменчивой  
последовательности s на месте.

Выдает случайное целое число из диапазона  
**randrange([start,] range(start, stop, step).**

**normalvariate(mu,  
sigma)**

**stop[, step]**

Аналогачно **choice (range (start,  
stop, step)** ).

Выдает число из последовательности  
нормально распределенных  
псевдослучайных чисел. Здесь mu - среднее,  
**sigma** - средиеквадратичесюе отклонение

**(sigma > 0)**

Остальные фзтзкции и их параметры можно уточнить по документации.  
Следует отметить, что в модзле есть функция **seed(n),** юторая  
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([l, 2, 3])  
В = sets.Set([2, 3, 4])

**Сузи RA. Язык программирования Python**

print А**I в,** А & **в,** А - **в,** А **л в**for і in А:  
if i in В:  
print i,

В результате будет выведено:

Set([l, 2, 3, 4]) Set([2, 3]) Set([l]) Set([l, 4])

23

Модули array и struct

Эти модули реализзлот низюуровневый массив и структзфу 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 ползліались в цикле. Альтернативный подход - использование  
списков для хранения промежуточных реззльтатов - 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 кзльтзфной среде могут использоваться свои правила для  
написания чисел, валют, времени и даты и т.п. Следующий пример  
выводит дату сначала в кзльтурной среде "С", а затем на русском языке:

import time, locale

locale.setlocale(locale.LC\_ALL, None)

print tirne.strftirne('%d %B %Y", time.localtime (time.time()))

locale.setlocale(locale.LC\_ALL, "ru\_RU.KOI**8**-R")

print time.strftime('%d %В %Y", time.localtime (time.time()))

В результате:

18 November 2004  
18 Ноября 2004

Модуль gettext

При интернационализации программы важно не толью предусмотреть  
возможность использования несюльких культурных сред, но и перевод  
сообщений и меню программы на соответствующий язык. Модуль  
**gettext** позволяет упростить этот процесс достаточно стандартным  
способом. Основные сообщения программы пищутся на английсюм  
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  
шнфігурирования и установки пакетов.

В качестве иллюстрации можно предположить, что создается модзиь  
для вычисления простых чисел по алгоритму "решето Эратосфена".  
Модуль будет находиться в файле **Sieve.ру** и состоять из одной  
функции **primes (N),** юторая 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 """

**5І**

**Сузи Р.А.**

def primes(N):

"""Возвращает все простые от 2 до N"""  
sieve = sets.Set(range(2, N))  
for i in range(2, int(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/S ieve. py( 15)primes()

-> sieve = sets.Set(range(2, N))

**(Pdb)1**

**11  
12**

**13**

**14**

**15**

**16**

**17**

**18**

**19**

**20**

**(Pdb)n**

* **/home/rnd/workup/intuit- python/examples/S ieve. py( 16)primes()  
  -> for i in range(2, int(math.sqrt(N))):**

**(Pdb)n**

* **/home/rnd/workup/intuit- python/examples/S ieve. py( 17)primes()  
  -> if i in sieve:**

**(Pdb)n**

* **/home/rnd/workup/intuit- python/examples/S ieve. py( 18)primes()**

**10** 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, int(math.sqrt(N))):  
if i in sieve:

sieve -= sets.Set(range(2\*i, N, i))  
return sieve

-> sieve -= sets.Set(range(2\*i, N, i))

**(Pdb)n**

* /home/rnd/workup/intuit- python/examples/S ieve. 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/S ieve. py( 17)primes()

-> if i in sieve:

**(Pdb)n**

* /home/rnd/workup/intuit- python/examples/S ieve. py( 18)primes()

-> sieve -= sets.Set(range(2\*i, N, i))

**(Pdb)n**

* /home/rnd/workup/intuit- python/examples/S ieve. 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

C 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 ftmction calls in 1.320 CPU seconds

Ordered by: standard name

ncalls tottime percall cumtime percall filenamedineno(ftmction)

1. 0.010 0.010 1.320 1.320 <string>:l(?)

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** | | pro file :**0**(pro filer) |
| 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(\_binaiy\_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 1 |
| 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** - юличество вызовов фз^кции или метода, **tottime** -  
полное время выполнения юда функции (без времени нахождения в  
вызываемых функциях), ре г **call** - тоже, в пересчете на один вызов,  
**cumtime** - аккумулированное время нахождения в фзлзкции, вместе со  
всеми вызываемыми функциями. В последнем столбце приведено имя  
файла, номер строки с функцией или методом 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 слз^іае - множества).

Модуль unittest

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

# file: test\_Sieve.py

**Сузи Р.А.**

import Sieve, sets  
import unittest

class Testsieve(unittest.TestCase):

def setUp(selQ:  
pass

def testone(self):

primes = Sieve.primes(l)  
self.assertEqual(primes, sets.SetQ)

def testlOO(selQ:

primes = Sieve.primes(lOO)

self.assert\_(primes == sets.Set([2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, ^  
53, 59, 61, 67, 71, 73, 79, 83, 89, 97]))

if name == ' main

unittest.mainO

Тестовый модуль состоит из определения класса, унаследованного от  
класса **unittest. TestCase,** в ютором описывается подготовка к  
испытаниям (метод **setup** ) и сами испытания — методы,  
начинающиеся на **test.** В данном слзліае 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

Run 2 tests in 0.002s

OK

В процессе разработки перед каждым выпуском все мод}ли  
прогоняются через регрессионные испытания, чтобы обнаружить, не  
были ли внесены ошибки. Однако никакие тесты в общем случае не

**Сузи RA. Язык программирования Python**

могут гарантировать безошибочности сложной программы. При  
дополнении модзэтей тесты также могут быть дополнены, чтобы  
отразить изменения в проекте.

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

Модуль pydoc

Успех проекта зависит не толью от обеспечения эффективного и  
качественного юда, но и от качества документации. Утилита **pydoc**аналогична шманде man в Unix:

$ pydoc Sieve  
Help on пюdule Sieve:

NAME

Sieve - Модуль для вычисления простых чисел от 2 до N

FILE  
Sieve. РУ

FUNCTIONS

primes(N)

Возвращает все простые от 2 до N

Эта страница помощи появилась благодаря тому, что были написаны  
строки документации - как ю всему модулю, так и к функции

**primes(N) .**

Стоит попробовать запустить **pydoc** следующей юмандой:  
pydoc -р 8088

И направить браузер на URL ссылка: httpy/127.0.0.1:8088/ - можно  
полздіить документацию по модулям Python в виде красивого web-сайта.

Узнать другие возможности **pydoc** можно, подав юманду **pydoc**

**Сузи Р.А.**

**pydoc.**

Пакет docutils

Этот пакет и набор утилит пока что не вждит в стандартнзто поставку  
Python, однако о нем нужно знать тем, кто жчет быстро готовить  
документацию (руководства пользователя и т.п.) для своих модулей.  
Этот пакет использует специальный язык разметки (ReStructuredText), из  
юторого потом легю получается документация в виде HTML, LaTeX и в  
других форматах. Текст в формате RST легю читать и в исходном виде.  
С этим инструментом можно познаюмиться на ссылка:  
httpy/docutils.sourceforge.net

Пакет distutils

Данный пакет предоставляет стандартный путь для распространения  
собственных Python-пакетов. Достаточно написать небольшой  
шн**4**мгурационный файл **setup.**ру, использзлощий **distutils,** и  
файл с перечислением файлов проекта **MANIFEST, in,** чтобы  
пользователи пакета смогли его установить юмандой

рЗДЬоп setup.РУ install

Тонюсти работы с **distutils** можно иззліить по документации.

Взаимодействие с операционной системой

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

Модуль OS

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

**Сузи RA. Язык программирования Python**

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.de fpath** 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 юторой 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

РАШ = 05.епѵігоп['РАШ']

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

**chdir(path)  
getcwd()**

**access(path,  
flags)**

Проверка доступности файла или каталога с  
именем **path.** Режим запрашиваемого доступа  
указывается значением **flags,** составленных  
юмбинацией (побитовым ИЛИ) флагов  
OS . F\_OK (файл существует), os . R\_OK (из  
файла можно читать), os . W\_OK (в файл  
можно писать) и os . Х\_ОК (файл можно  
исполнять, каталог можно просматривать).  
Делает **path** текущим рабочим каталогом.  
Текущий рабочий каталог.

Устанавливает режим доступа к **path в**

**Язык программирования Python**

**stat(path)**

**remove(path),  
unlink(path)**

**rmdir(path)  
removedirs(path)**

**rename(src, dst)**

**renames(src,  
dst)**

**makedirs(path[,  
mode])**

**mkdir(path[,  
mode])**

**listdir(dir)**

**chmod(path,  
mode)**

значение **mode.** Режим доступа можно  
получить, сюмбинировав флаги (см. ниже).  
Следует заметить, что chmod () не дополняет  
действующий режим, а устанавливает его  
заново.

Возвращает список файлов в каталоге **dir.** В  
список не вждят специальные значения " . " и

Создает каталог **path.** По умолчанию режим  
mode равен О 7 7 7, то есть:

**S\_IRWXU I S\_IRWXG I S\_IRWXO,** если  
пользоваться юнстантами модуля **stat.**Аналог **mkdir** **(),** создающий все  
необждимые каталоги, если они не  
существуют. Возбуждает исключение, когда  
последний каталог уже существует.  
i^^aляeт файл **path.** Для удаления каталогов  
используются **rmdir () и removedirs** **().**

^аляет пустой каталог **path,**і^аляет **path** до первого непустого каталога. В  
случае если самый последний вложенный  
подкаталог в зжазанном пути - не пустой,  
возбуждается исключение **OSError.**

Переименовывает файл или каталог src в  
**dst.**

Аналог **rename** **(),** создающий все  
необждимые каталоги для пути ds t и  
удаляющий пустые каталоги пути **src.**Возвращает информацию о **path** в виде не  
менее чем десятиэлементного юртежа. Для  
доступа к элементам кортежа можно  
использовать юнстанты из модуля **stat,**например **stat. ST\_MTIME** (время  
последней модификации файла).

Устанавливает значения времен последней

**Язык программирования Python**

**utime(path,  
times)**

модис}мкации (mt ime ) и доступа к файлу (  
**atime** ). Если **times** равен **None,** в качестве  
времен берется текущее время. В дрзтих  
слзліаях **times** рассматривается как  
двухэлементный кортеж ( **atime, mtime** ).

Для получения **atime и mtime** неюторого  
файла можно использовать **stat** **()** совместно  
с юнстантами модуля **stat.**

Для работы с процессами модуль os предлагает следзлощие функции  
(здесь упомянуты толью неюторые, доступные как в Unix, так и в  
Windows):

**getloadavg(;**

**times()**

**system(cmd)**

**abort(:**

Вызывает для текущего процесса сигнал **SIGABRT.**

Выполняет юманднзло строку cmd в отдельной  
оболочке, аналогично вызову **system** библиотеки  
языка С. Возвращаемое значение зависит от  
используемой платформы.

Возвращает юртеж 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).

Возвращает юртеж из трех значений,  
соответствзлощих занятости процессора за последние  
1, 5 и 15 минут.

Модуль stat

В этом модуле описаны юнстанты, юторые можно использовать как  
индексы к юртежам, применяемым функциями **os.statO** и  
**OS** . chmod () (а также неюторыми другими). Их можно уточнить в  
документации по Python.

**Сузи Р.А.**

Модуль tempfile

Программе иногда требуется создать временный файл, юторый после  
выполнения некоторых действий больше не нужен. Для этих целей  
можно использовать функцию **Temporary File,** юторая возвращает  
файловый объект, готовый к записи и чтению.

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

import tempfile

f = tempfiIe.TemporaiyFile()

f.write("0"\*100) # записывается сто символов О

f.seek(**0**) # уст. указатель на начало файла

print len(f.read()) # читается до юнца файла и вычисляется длина

Как и следовало ожидать, в результате будет выведено 100. Временный  
файл будет удален, как толью 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 модзии, юторые работают с внешними  
хранилищами данных.

Модуль pickle

**Сузи RA. Язык программирования Python**

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 толью сериализуемые  
  объекты.
* функции, определенные на уровне модуля (сохраняется имя, но не  
  реализация!).
* встроенные фзлікции.
* классы, определенные на уровне модуля.
* объекты классов, **diet или setstate** **()** юторые

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\_fiIe)

p.dump(mydata)

output\_file.close()

Восстановление:

import pickle

input\_file = open("mydata.dat", T")  
mydata = pickle.load(input\_file)  
print mydata  
input\_file.close()

Модуль shelve

**Сузи RA. Язык программирования Python**

Для хранения объектов в родном для Python формате можно применять  
полі^^ (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.syncO # запись изменений в БД на диске

del d[key] # удалить ключ и значение

flag = d.has\_key(key) # проверка наличия ключа  
1st = d.keysO # список ключей  
d.closeO # закрытие полки

Модули anydbm и gdbm

Для внешнего хранения данных можно использовать примитивные  
базы данных, содержащие пары ключ-значение. В Р**3**ДІЮП имеется  
несколью модулей для работы с такими базами: **bsddb,** gdbm, **dbhash**и т.п. Модуль **anydbm** выбирает один из имеющижя хэшей, поэтому его  
можно применять для чтения ряда форматов **(any** - любой).

Доступ к хэщу 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

**Сузи RA. Язык программирования Python**

записи в CSV-файл и чтению из него:

mydata = [(1, 2, 3), (1, 3, 4)]  
**import CSV**

* Запись в файл:

f = fileC'my.csv", V")  
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 толью при условии реализации  
альтернативных веток алгоритма, либо с отказом от свойств, юторые  
доступны не на всех платформах. Так, под Windows не работает  
достаточно обычная для Unix функция **os.forkO,** 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

**Сузи RA. Язык программирования Python**

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

Примеры серверов использзлотся по другому принципу В модуле с  
реализацией сервера описан базовый класс, из юторого пользователь  
модзля должен наследовать свой класс, реализзлощий требуемзло  
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 модзли для  
работы с различными форматами, применяющимися для юдирования  
данных в сети Интернет и тому подобных приложениях.

Сегодня наиболее мощным инструментом для обработки сообщений в  
формате 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 языюм, в ютором можно  
"заглянуть" глубою в собственные внутренние структзфы юда и данных.  
Модули этой категории дают возможность приюснуться к внутреннему  
устройству Python. Более подробно об этом рассказывается в отдельной  
лекции.

Графический интерфейс

**Сузи RA. Язык программирования Python**

Почти все современные приложения имеют графический интерфейс  
пользователя. Такие приложения можно создавать и на языке Python. В  
стандартной поставке имеется модзль Tkinter, юторый есть не что иное,  
как интерфейс к языку ТсІ/Тк, на ютором можно описывать графический  
интерфейс.

Следует отметить, что сзтдествуют и ДРЗ^ие пакеты для

программирования гра**4**мчесюго интерфейса: \ѵхРздЬоп (основан на  
wxWidgets), PyGTK и т.д. Среди этих пакетов в основном такие, которые  
работают на одной платформе (реже - на двух).

Помимо возможностей программного описания графичесюго

интерфейса, для Python есть несюлько коммерческих и некоммерческих  
построителей гра(}»іческого интерфейса (GUI builders), однако в данном  
курсе они не рассматриваются.

Заключение

в этой лекции говорилось о встроенных фзтікциях языка Python и  
модзлях его стандартной библиотеки. Неюторые направления будут  
рассмотрены более подробно в следующих лекциях. Python имеет  
настолько обширную стандартную библиотеід^, что в рамках одной  
лекции можно только сделать ее краткий обзор, подкрепив небольшими  
примерами наиболее типичные идиомы при использовании модулей.

**Сузи RA. Язык программирования Python**

Элементы функционального программирования

Эта лекция может показаться необычной для того, кто использует  
императивные языки программирования (вроде Pascal, C++ или Java).  
Тем не менее, функциональный поджд дает программисту мощные  
средства, позволяя создавать не только более компактный, но и более  
устойчивый к ошибкам программный юд. Совсем не обязательно  
писать с помощью Р**3**ДІЮП чисто фзтікциональные программы, но  
необходимо научиться видеть, где элементы функционального  
программирования принесут максимальный эффект.

Функции являются абстракциями, в которых детали реализации  
некоторого действия скрываются за отдельным именем. Хорошо  
написанный набор функций позволяет использовать их много раз.  
Стандартная библиотека Python содержит множество готовых и  
отлаженных функций, многие из которых достаточно зтзиверсальны,  
чтобы работать с широким спектром входных данных. Даже если  
некоторый зліасток юда не используется несюлько раз, но по входным и  
выходным данным он достаточно автономен, его смело можно  
выделить в отдельную функцию.

Эта лекция более ориентирована на практические соображения, а не на  
теорию функционального программирования. Однако там, где нужно,  
будут употребляться и поясняться соответствзлощие термины.

Далее будут подробно рассмотрены описание и использование функций  
в РзДЬоп, рекурсия, передача и возврат фзлікций в качестве параметров,  
обработка последовательностей и итераторы, а также такое понятие как  
генератор. Будет продемонстрировано, что в Python функции являются  
объектами (и, значит, могут быть переданы в качестве параметров и  
возвращены в результате выполнения функций). Кроме того, речь  
пойдет о том, как можно реализовать некоторые механизмы  
функционального программирования, не имеющие в Python прямой  
синтаксичесюй поддержки, но широю распространенные в языках  
функционального программирования.

Что такое функциональное программирование?

Функциональное программирование - это стиль программирования.

**Сузи RA. Язык программирования Python**

использующий только юмпозиции функций. Другими словами, это  
программирование в выражениях, а не в императивных юмандах.

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 стрзжтурой в  
  программе.
* Обработка списюв (последовательностей).
* Запрещение побочных эффектов у функций, что в первую очередь  
  означает отсутствие присваивания (в "чистых" функциональных  
  языках)
* Запрещение операторов, основной зтюр делается на выражения.  
  Вместо операторов вся программа в идеале - одно выражение с  
  сопутствующими определениями.
* Ключевой вопрос: что нужно вычислить, а не как.
* Использование фзтзкций более высоких порядков (фзтзкции над  
  функциями над фзтзкциями).

Функциональная программа  
**в математике фзтзкция 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 двзлия вызовами. Их можно представлять себе черными  
ящиками, о юторых известно толью то, что они делают, но совсем не  
важно, как.**

Программы в фзлзкциональном стиле юнстрзлзруются как юмпозиция  
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

**Сузи RA. Язык программирования Python**

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 использзлотся. Кроме того, математические  
функции легко работают с объектами, требующими бесюнечного объема  
информации (например, вещественные числа). В общем случае  
компьютерная программа может выполнить лишь приближенные  
вычисления.

Кстати, бинарные операции " + ", " - ", " \* ", " / ", которые  
записываются в выражениях, являются "математическими" фзтзкциями  
над двумя аргументами — операндами. Их использзлот настолько часто,  
что синтаксис языка программирования имеет для них более короткую  
запись. Модзль **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 толью 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 класса).

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

Определение фзткции должно содержать список формальных  
параметров и тело определения функции. В **случае** с оператором def

**Сузи RA. Язык программирования Python**

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 з^олчанию  
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,  
юторое в реззльтате вычисления дает исполняемый объект.

Функция одного аргумента:

def swapcase(s):  
return s.swapcaseO

print swapcase("ABC")

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

definc(n, deha=l):  
return n+delta

print inc(**12**)  
print inc(**12**, **2**)

Функция c 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 c именами и значениями аргументов  
w = TextWrapper(width=width, \*\*kwargs)  
return w.wrap(text)

print wrapC'my long text...", width=4)

**Сузи Р.А.**

Функция произвольного числа аргументов:

def imx\_min(\*args):

# args - список аргументов в порядке их указания при вызове  
return max(args), min(args)

print max\_min(l, 2, -1, 5, 3)

Функция с обычными (позиционными) и именованными аргументами:

def swiss\_knife(argl, \*args, \*\*kwargs):  
print argl  
print args  
print kwargs  
return None

print swiss\_knife(l)

print swiss\_knife(l, 2, 3, 4, 5)

print swiss\_knife(l, 2, 3, a='abc', b='sdf)

# print swiss\_knife(l, a='abc', 3, 4) # !!! ошибка

1st = [2, 3, 4, 5]

dct = {'a': 'abc', 'b': **'sdf}**

print swiss\_knife(l, \*lst, \*\*dct)

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

func = lambda х, у: х + у

В реззльтате **lambda** -выражения полздгается безымянный объект-  
функция, юторая затем используется, например, для того, чтобы связать  
с ней некоторое имя. Однако, как правило, определяемые **lambda** -  
выражением фзтікции, применяются в качестве параметров фзтікций.

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

**Сузи Р.А.**

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 понеть, что за именем фзпкции стоит объект. Этот объект можно  
связать с другим именем:

def add(x, у):  
return **X** + у

addition = add # теперь addition и add - разные имена одного и того же о

Пример, в ютором в качестве значения по з^олчанию аргумента  
функции используется изменчивый объект (список). Этот объект - один  
и тот же для всех вызовов фзтікций, что может привести к казусам:

def mylist(va], lst=[]):  
lst.append(vaf)  
return **1**st

print mylist(l),  
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):

**1**st = **1**st or []  
lst.append(vaf)

**Сузи Р.А.**

return **1**st

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

Рекурсия  
**в неюторых слзліаях описание функции элегантнее всего выглядит с  
применением вызова этой же фзтікции. Такой прием, югда функция  
вызывает саму себя, называется реі^рсией. В фзтікциональных 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 п== **0**:  
return []

п, 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  
юторого 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 п -го числа Фибоначчи это  
демонстрирует:

def Fib(n):  
**ifn<2**:  
return n  
else:

return Fib(n-l) + Fib(n-2)

В данном слз^ае юличество рекурсивных вызовов растет  
экспоненциально от числа п, что совсем не соответствует временной  
сложности решаемой задачи.

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

Предупреждение:

При работе с рекурсивными функциями можно легю 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 фзлікции в реззльтате, 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 переданнзло  
ему функцию для каждого найденного объекта.

Функция арр1у()

Функция **apply** о применяет функцию, переданную в качестве  
первого аргумента, к параметрам, которые переданы вторым и третъим

**Сузи RA. Язык программирования Python**

аргументом. Эта функция в Python устарела, так как вызвать фзтікцию  
можно с помощью обычного синтаксиса вызова функции. Позиционные  
и именованные параметры можно передать с использованием  
звездочек:

»> 1st = [1, 2, 3]

»> **dct=** {'а':4, **Ъ':5}**

»> арр**1**у(тах, **1**st)

3

»> max(\*lst)

3

»> apply(dict, [], dct)

{'а': 4, **Ъ':5}**

»> dict(\*\*dct)

{'а': 4, **Ъ':5}**

Обработка последовательностей

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

Под последовательностью в Python понимается любой тип данных,  
юторый поддерживает интерфейс последовательности (это несюлько  
специальных методов, реализующих операции над  
последовательностями, юторые в данном курсе обсуждаться не будут).

Следует заметить, что тип, основной задачей юторого является  
хранение, манипзлирование и обеспечение доступа к самостоятельным  
данным называется юнтейнерным типом или просто юнтейнером.  
Примеры юнтейнеров в Python - списки, юртежи, словари.

Функции range() и xrange()

Функция **range** о уже упоминалась при рассмотрении цикла **for.**Эта функция принимает от одного до трех аргументов. Если аргумент  
всего один, она генерирует список чисел от О (включительно) до

**Сузи RA. Язык программирования Python**

заданного числа (исключительно). Если аргументов два, то список  
начинается с числа, указанного первым аргументом. Если аргументов  
три - третий аргумент задает шаг

»> print range(lO)

[О, 1, 2, 3, 4, 5, 6, 7, 8, 9]

»> print range(l, 10)

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

»> print range(l, 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,  
юторый ведет себя почти как список, порождаемый **range** **(),** но не  
хранит в памяти все выдаваемые элементы.

Функция тарО

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

В следующем примере складываются значения из двух списков:

**»> 11 = [2, 7, 5, 3]**

**»> 12 = [-2, 1, о, 4]**

»> print map(lambda х, у: х+у, **11**,**12**)

**[0, 8, 5, 7]**

В этом примере применена безымянная функция для ползліения суммы  
двух операндов ко всем элементам 11 и 12. В **случае** если одна из  
последовательностей юроче другой, вместо соответствующего операнда

**Сузи RA. Язык программирования Python**

будет **None,** что, конечно, собьет операцию сложения. В зависимости  
от решаемой задачи, можно либо видоизменить функцию, либо считать  
разные по длине последовательности ошибюй, юторую нужно  
обрабатывать как отдельную ветвь алгоритма.

Частный сл}ліай применения тар () - использование **None** в качестве  
первого аргумента. В этом слз^ае просто формируется список кортежей  
из элементов исходных последовательностей:

»> 11 = [2, 7, 5, 3]

»> 12 = [-2, 1, О, 4]

»> print map(None, **11**,**12**)

[(2, -2), (7, 1), (5, 0), (3, 4)]

Функция filterO

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

Например, в следующем фрагменте юда можно избавится от символов,  
юторые не являются буквами:

»> filter(larnbda х: x.isalpha(), 'Щ there! I am eating an apple.')  
'Hitherelameatinganapple'

Списковые включения

Для более естественной записи обработки списюв в Python 2 была  
внесена новинка: списювые включения. Фактически это специальный  
сокращенный синтаксис для вложенных циклов for и условий if, на

**Сузи RA. Язык программирования Python**

самом низюм з^овне юторых определенное выражение добавляется к  
списку, например:

alLpairs = []  
for i in range(5):  
for j in range(5):  
**ifi<=j:**

all\_pairs.append((i, j))

Bee это можно записать в виде спискового включения так:

alLpairs = [(і, j) for i in range(5) for j in range(5) if i <= j]

Как легко заметить, списювые включения позволяют заменить тар ()  
и **filter** О на более здобные для прочтения юнструкции.

В следующей таблице приведены эквивалентные выражения в разных  
формах:

В форме фзтнщии В форме списювого включения

**filter(f, 1st) [х for X in 1st if f(x) ]  
filter(None, 1st) [x for x in 1st if x]  
map(f, 1st) [f(x) for x in 1st]**

Функция sum()

Получить сумму элементов можно с помощью фзтікции sum ():

»> sum(range(10))

45

Эта фзтікция работает толью для числовых типов, она не может  
шнкатенировать строки. Для юнкатенации списка строк следует  
использовать метод j оіп ().

Функция reduceO

**Сузи RA. Язык программирования Python**

Для организации цепочечных вычислений (вычислений с накоплением  
результата) можно применять функцию **reduce** **(),** юторая 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, Jst, start)

Совет:

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

В следзлощем примере накапливаются промежуточные результаты  
суммирования:

1st = range(lO)

f = lambda х, у: (х[**0**] + у, х[**1**]+[х[**0**] + у])  
print reduce(f, 1st, (О, []))

В итоге пол}ліается:

**(45, [О, 1, 3, 6, 10, 15, 21, 28, 36, 45])**

Функция zipO

Эта функция возвращает список юртежей, в ютором і -й кортеж  
содержит і -е элементы аргументов-последовательностей. Длина  
результирующей последовательности равна длине самой коротюй из  
поел едовательностей- аргументов:

»> print zip(range(5), "abede")

**[(0, 'а'), (1,** 'b'), **(2, 'с'), (3, 'd'), (4, 'е')1**

**Сузи Р.А.**

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 юнтейнера. Если  
в выражении есть операции с итераторами вместо юнтейнеров,  
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 реззльтата, 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 целиюм 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 **Stop Iteration.** Для  
полздіения итератора по некоторому объекту необходимо прежде  
применить к этому объекту фзлзкцию **iter** **()** (цикл for делает это  
автоматически).

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

**Сузи Р.А.**

Функция iter()

Эта фзтікция имеет два варианта использования. В первом она  
принимает всего один аргумент, юторый 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":

itl = iter([l, 2, 3, 4, 5])

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

Й**2** = iter(forit. None)

print **[x** for **X** in itl]

print **[x** for **X** in it2]

Примечание:

Если функция не возвращает значения явно, она возвращает **None,** что  
и использовано в примере выше.

Функция enumerateO

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

»> print [х for X in enumerate("abcd")]

**Сузи Р.А.**

[(О, 'а'), (1, 'Ь'), (2, 'с'), (3, 'd')]

Функция sortedO

Эта функция, появившаяся в Python 2.4, позволяет создавать итератор,  
выполняющий сортировід^:

»> sorted('avdsdf)

['а', 'd', 'd', 'f, 's', **W]**

Далее рассматриваются фзтікции модуля **itertools.**

Функция itertools.chainO

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

from itertooJs import chain  
itl = iter([ 1,2,3])  
k2 = iter([8,9,0j)  
for i in chainfitl, it**2**):  
print i,

даст в реззльтате  
1 2 **3890**

Функция itertools.repeatO

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

for і in itertools.repeat(l, 4):  
print i.

**Сузи Р.А.**1111

Функция itertools.countO

Бесконечный итератор, дающий целые числа, начиная с заданного:

for і in itertools.count(l):  
print i,  
**ifi> 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.cycleO

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

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

**12312312312312312312312312312312312  
3 123 1**

**23123123123123123123123123123123123**

**12312**

**31231231231231231231231231231231...**

**и**

Функции iteitools.imapO, itertools.starmap()  
itertools.ifilterO

**Сузи RA. Язык программирования Python**

Аналогами **map() и filter** **()** в модуле **itertools** являются  
**ітар () и if ilter** **().** Отличие imap () **оттарО** в том, что вместо  
значения от преждевременно завершившижя итераторов объект **None**не подставляется. Пример:

for і in пир(1атЬс1а х, у: (х,у), [1,2], [1,2,3]):  
print і,

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

from itertooJs 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 тар () нормально  
воспринимает итераторы в любом сочетании с итерабельными  
(поддающимися итерациям) объектами:

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

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

Функция **itertools . starmap** **()** подобна **itertools** . imap (),  
**HO** 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.

la **2**b

Функция if **ilter** 0 работает как f **ilter** **().** Кроме того, в модзие  
**itertools** есть функция if **ilterf alse** **(),** юторая как бы  
добавляет отрицание к значению фзлікции:

**Сузи Р.А.**

for і in ifiJterfalsefJambda x: x > 0, [1, -2, 3, -3]):  
print i,

**-2-3**

Функции itertools.takewhileO и itertools.dropwhile()

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

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

print

for i in dropwhiIe(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.izipO

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

Функция itertools.groupbyO

Эта функция дебютировала в Python 2.4. Функция принимает два

**Сузи RA. Язык программирования Python**

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

1st = 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.teeO

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

Собственный итератор

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

class Fibonacci:

'""Птератор 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(selQ:  
if self.n < self, max:

a, self.n, self.a, selfb = 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 ютором 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 следзлощего значения. При вызове такая функция ничего не  
вычисляет, а создает объект с интерфейсом итератора для получения  
значений. Другими словами, если функция должна возвращатъ  
последователъностъ, из нее доволъно просто сделать генератор,  
юторый будет фзлікционально эквивалентной "ленивой" реализацией.  
Ленивыми называются вычисления, которые откладываются до самого  
последнего момента, югда пол}ліаемое в результате значение сразу  
используется в другом вычислении.

Для примера с последовательностью Фибоначчи можно построить  
такой вот генератор:

defFib(N):  
а, Ь = О, 1  
for і in xrange(N):

**Сузи Р.А.**

yield а

а, Ь = Ь, а + Ь

Использовать его не сложнее, чем любой другой итератор:

for і in Fib(lOO):  
print i,

Однаю 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 списювому,  
но вместо квадратных скобок используются круглые. Списювое  
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 неюторые замены:

for line in (Lreplace("-"," -") for **1** in open("input.dat")):  
print line

Ничто не мешает использовать итераторы и для записи в файл:

open("output.dat", "w").writelines(

Lreplace("-"," -") for 1 in open("input.dat"))

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

Карринг

**Сузи RA. Язык программирования Python**

Библиотека Xoltar toolkit (автор Bryn Keller) включает модзль  
**functional,** юторый позволяет упростить использование  
возможностей функционального программирования. Модзль  
**functional** применяет "чистый" Python. Библиотеку можно найти по  
адресу: ссылка: httpy/sourceforge.net/projects/xoltar-toolkit.

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

from functional import curry  
def subtract(x, y):  
return **X - у**

print subtract(3, 2)  
subtract\_from\_3 = cuny(subtract, 3)  
print subtract\_from\_3(2)  
print cuny(subtract, 3)(2)

Bo всех трех случаях будет выведено 1. В следзаощем примере  
полздіается новая функция, подставляя второй аргумент. Вместо дрзтого  
аргумента вставляется специальное значение **Blank:**

from functional import curry. Blank  
def subtract(x, y):  
return **X - у**

print subtract(3, 2)

subtract\_2 = curry(subtract. Blank, 2)

print subtract\_2(3)

print cuny(subtract. Blank, 2)(3)

Заключение  
**в этой лекции рассматривался принцип построения функциональных  
программ. Кроме того, было показано, что в РзДЬоп и его стандартных  
модзлях имеются достаточно мощные выразительные средства для  
создания функциональных программ. В слзліае, югда требуются  
дополнительные возможности, например, карринг, их можно легю**

**Сузи Р.А.**

реализовать или взять готовзло 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 толью шгда они непосредственно  
требуются.

Ссылки по теме

Статья Д. Мертца ссылка: httpy/www-106.ibm.conydeveloperworks/libraiy/l-  
prog.html

**comp.Jang.functional ссылка:**

Часто задаваемые вопросы в  
http y/www. cs. nott.ac. uk/~ gmh/faq .html

Объектно-ориентированное программирование

Язык Python имеет достаточно мощную, но, вместе с тем, самобытнзто  
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.

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).

Примечание:

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

Примечание:

ОО программирование - это методология написания шда. Здесь не

**Сузи RA. Язык программирования Python**

будет подробно рассматриваться объектно-ориентированный анализ и  
объектно-ориентированное проектирование, юторые не менее важны  
как стадии создания программного обеспечения.

Основные понятия

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

**программа**

При объектно-ориентированном программировании  
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 неюторые 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 бытъ  
разными объектами. Говорят, что они имеют разную индивидуальность.

**Сузи RA. Язык программирования Python**

в языке Python для определения класса используется оператор **class:**

class имя\_класса(класс**1**, класс**2,...):**

# 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 деюмпозиции  
на части "дозированной" сложности. Отдельный объект, даже вместе с  
совокупностью его связей с другими объектами, человеюм  
воспринимается легче (именно так он привык оперировать в реальном  
мире), чем что-то неструктурированное и монотонное.

Перед тем как начать написание даже самой простеньюй 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 сюнцентрироваться на свойствах,  
значимых для задачи..

Выделяемые объекты необязательно должны походить на (}мзические  
объекты - ведь это абстракции, за которыми скрываются процессы,  
взаимодействия, отношения.

>^ачная декомпозиция стоит многого. От нее зависят не только  
юличественные характеристики юда (быстродействие, занимаемая  
память), но и трздоемкость дальнейшего развития и сопровождения.  
При отсутствии соответствующего опыта лучше не загадывать буг\ущих  
путей развития программы, а делать ее как можно проще, под  
конкретнзао задачу.

Даже если просто перечислить все сзтцествительные, встретившиеся в

**Сузи RA. Язык программирования Python**

описании задачи (явно или неявно), ползліится непложй 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 встречалисъ много раз: ведь каждое  
число, строка, функция, модуль и т.п. - это объекты. Некоторые  
встроенные объекты имеют в Python синтаксическую поддержід^ (для  
задания литералов). Таковы числа, строки, списки, кортежи и некоторые  
другие типы.

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

а = 3  
Ь = 4.0  
с = а + Ь

Здесь происхэдит следующее. Сначала имя " а " связывается в  
локальном пространстве имен с объектом-числом 3 (целое число).  
Затем " Ь " связывается с объектом-числом 4.0 (число с плавающей  
точюй). После этого над объектами 3 и 4.0 выполняется операция  
сложения, и имя " с " связывается с ползліившимся объектом. Кстати,  
операциями, в основном, **6**yztyr называтъся методы, юторые имеют в  
Python синтаксическую поддержку, в данном случае - ин(}мкснзло записъ.  
То же самое можно записатъ как:

с = а. add (Ь)

Здесь add () - метод объекта а, юторый реализует операцию +

между этим объектом и другим объектом.

Узнатъ набор методов неюторого объекта можно с помощью  
встроенной фзлікции dir ():

**Сузи RA. Язык программирования Python**

»> dir(a)

[' abs ' add ' and ' class ' cmp ' coerce

' delattr ' div ' divmod ' doc ' float

' floordiv ' getattribute ' getnewargs ' hash

' hex ' init ' int ' invert ' long

' Ishift ' 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 заказать на еще одну особенность Р**3**ДІЮП. Не только  
ин(}мксные 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:

с. abs О

Объекты появляются в результате вызова фзаікций-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 ссылі^^ на объект) из пространства  
имен:

а = **1**

**#...**

del а

# имени а больше нет

Типы и классы

Тип определяет область допустимых значений объекта и набор  
операций над ним. В ООП тип тесно связан с поведением - действиями

**Сузи RA. Язык программирования Python**

объекта, состоящими в изменении внутреннего состояния и вызовами  
методов дрзтих 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 юторым они принадлежат. Таким  
образом, тип и класс в Python становятся синонимами.

Интерпретатор языка Python всегда может сказать, к каюму 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 неюторыми  
параметрами. Объект-класс, как и объект-фзлікция, может быть вызван.  
Это и будет вызовом юнструктора:

»> import sets

»> s = sets.Set([l, 2, 3])

В этом примере мод}ль **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 юнструктора, определенные классы  
имеют и деструктор - метод, юторый вызывается при уничтожении  
объекта. В языке Python объект уничтожается в случае удаления

**Сузи RA. Язык программирования Python**

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 редю.  
Обычно в том слзліае, когда объект управляет ресзд)СОм, который нужно  
корректно вернуть в определенное состояние.

Еще один способ ползліить объект неюторого типа - использование  
функций-фабрик. По синтаксису вызов фзлзкции-фабрики не отличается  
от вызова констрзлсгора класса.

Определение класса

Пусть в жде анализа данной предметной области необждимо  
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 РздЬоп 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, (vl, v**2**)):  
self.vertices.add(vl)  
self, vertices. add(v**2**)  
self.edges.add((vl, v**2**))

def has\_edge(self, (vl, v**2**)):  
return (vl, v**2**) in self.edges

def str (self):

**Сузи RA. Язык программирования Python**

return "%s; %s" % (self.vertices, self edges)

Использовать юіасс можно следуюіцим образом:  
**g=G([l,** 2, 3, 4], [(1, 2), (2, 3), (2, 4)])  
print g

g.add\_vertex(5)  
g.add\_edge((5,6))  
print g.has\_edge((l,**6**))  
print g

что даст в результате

Set([l, 2, 3, 4]); Set([(2, 4), (1, 2), (2, 3)])

False

Set([l, 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 in it .

(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  
исполъзуется специалъный метод str ().

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

print isinstance(g, G)

Инкапсутіяция

Обычно считается, что без инкапсзляции невозможно представить себе  
ООП, что это ключевое понятие. История развития методологий  
программирования движима борьбой со сложностью разработки  
программного обеспечения. Оложность больших программных систем,  
в создании которых злгаствует сразу большое юличество разработчиюв,

**Сузи RA. Язык программирования Python**

уменьшается, если на верхнем уровне не видно деталей реализации  
нижних зфовней. Собственно, процедзфный подход был первым шагом  
на этом пути. Под инкапсуляцией (encapsulation, 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 РзДЬоп 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 неюторому  
атрибуту (не методу) напрямую, если, юнечно, 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 общедостзпным 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(sel^: return self. x

def setx(self, value): self. x = value

def dek(sel^: del self. x

X = property(getx, setx, dek, 'Tm the 'x' property.")

Синтаксически доступ к свойству х будет обычной ссылкой на атрибут:  
»> с = С()

**Сузи RA. Язык программирования Python**

»> с.х = **1**»> print с.х  
**1**

»> del с.х

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\_**

**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  
**(self,** должен вычислить значение атрибута либо  
возбудить исключение **AttributeError.**Для ползліения полного контроля над  
атрибутами в "новых" классах (то есть  
потомках **object)** использз^те метод  
 **getattribute** **().**

Этот метод вызывается при присваивании  
значения неюторому атрибуту В отличие от

**getattr** **(),** метод всегда вызывается,

**setattr (self,**

**name, value)**

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 реіддзсию. Для присваивания  
значений атрибутов предпочтительнее

присваивать словарю **diet :**

**self. diet [name] = value** или

(для "новых" классов) - обращение к

**setattr** О базового класса:

**objeet. setattr (self, name,**

**value).**

**Язык программирования Python**

**Сузи Р.А.**

**delattr\_**

**name)**

**(self.** 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 юторого будут ключи словаря, а значениями - значения из  
словаря по заданным ключам:

class AttDict(object):

def init (self. dict=None):

object. setattr (self, '\_selfdict', diet 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 общедостзпный интерфейс. Обычно применяется  
одиночное подчеркивание, юторое в языке не играет особой роли, но

**Сузи RA. Язык программирования Python**

как бы говорит программисту: "этот метод толью 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:

... х**=0**... \_х = **0**х**=0**

»> dir(X)

**\_module\_', '\_х', 'х']**

['\_Х ^х',' doc\_

Полиморфизм  
**в переводе с гречесюго полимор4мзм означает "многоформие". Так в  
информатике называют возможность использования одного имени для  
выполнения различных действий.**

Можно встретить множество определений полимор(}мзма (также есть  
несюлью видов полиморфизма) в зависимости от языка  
программирования. Как правило, в качестве примера проявления  
полиморфизма приводят переопределение методов в подклассах. При  
этом можно создать фзтзкцию, требующую формального аргз^ента -  
экземпляра базового класса, а в качестве фактичесюго 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 полимор4мзм 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 полздіить и с помощью наследования, но в Python это не  
толью не обязательно, но иногда и противоречит здравому смыслу

При написании функции в Python обычно не проверяется, к каюму типу  
(классу) относится тот или иной аргумент: неюторые методы просто

**Сузи RA. Язык программирования Python**

применяются к переданному объекту Тем самым функции полз^іаются  
максимально обобщенными: они не требуют от объектов-параметров  
большего, чем наличие методов с определенным именем, набором  
аргументов и семантиюй.

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

def get\_last(x):  
return х[-**1**]

print get\_last([l, 2, 3])  
print get\_last("abcd")

Описанной фзтікции будет подходитъ в качестве аргумента все, от чего  
можно взятъ индекс -1 (последний элемент). Однако семантика "взятие  
последнего элемента" выполняется толью для последовательностей.  
Функция будет работать и для словарей, но смысл при этом будет  
немного дрзтой.

Имитация типов

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

class CountArgs(object):

def call (self, \*args, \*\*kwargs):

return len(args) + len(kwargs)

cc = CountArgsO  
print cc(l, 3, 4)

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

**Сузи RA. Язык программирования Python**

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

class Point:

def init (self, x, y):

self-coord = (x, y)  
def nonzero (selQ:

return self.coord[**0**] != **0** or self.coord[l] != **0**

def cmp (self, p):

return cmp(self.coord, p.coord)

for **X** in range(-3, 4):  
for у in range(-3, 4):  
if Point(x, y) < Point(y, x):  
print

elif Point(x, y):  
print  
else:  
print "o",  
print

Программа выведет:

**о**

**\* \* \* \*  
\* \* \***

\* **\*  
\***

В данной программе класс **Point** (Точка) имеет метод

**nonzero** О, который определяет истинностное значение

объекта класса. Истину будут давать только точки, отличные от (О,

О). Другой метод - стр () - вызывается при необходимости

сравнить точку и другой объект (имеющий как и точка атрибут coord,  
юторый содержит кортеж как минимум из дв}^ элементов). Нужно

заметитъ, что вместо стр можно определитъ отделъные методы

для операций сравнения: **It, 1е, пе, eq.**

**Сузи RA. Язык программирования Python**

ge , gt (для <, <=, ! =, ==, >=, > соответственно).

Достаточно легю имитировать и числовые типы. Класс, который  
пользуется здобством 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, югда 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 юторого нижележащие классы  
иерархии наследзлот свои атрибуты, уточняя и расширяя поведение  
вышележащего класса. Обычно принципом построения классификации  
является отношение "IS-A" ("ЕСТЬ"). Например, класс Окружность в  
программе - графичесюм 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 сзперклассом) для класса  
Окружность.

В языке РзДЬоп во главе иерархии ("новых') классов стоит класс

**object.** Для ориентации в иерархии существуют некоторые  
встроенные функции, юторые будут рассмотрены ниже. Функция  
**issubclass** **(х,** **у)** может сказать, является ли класс х подклассом  
класса у:

»> 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 класси4»ікации 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: юлесо является  
частью машины).

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

В Python примером абстрактного класса является встроенный тип  
**basestring,** у юторого есть шнкретные подклассы **str** и **Unicode.**

Множественное наследование

**Сузи RA. Язык программирования Python**

в отличие, например, от Java, в языке Python можно наследовать класс  
от несюльких 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 сконстрзпрованный  
  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 А:

def a(selQ: return 'а'  
class В:

def b(self): return 'b'  
class C:

def c(selQ: return 'c'

class AB(A, B):

pass

class BC(B, C):

pass

**Сузи Р.А.**

class АВС(А, В, С):  
pass

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

def ma(self): return 'а'  
def mb(seJ^; return 'b'  
def mc(self): return 'c'

class AB:  
a = ma  
b = mb

class BC:  
b = mb  
c = me

class ABC:  
a = ma  
b = mb  
c = me

Порядок разрешения методов

В случае, когда надклассы имеют одинаковые методы, использование  
того или иного метода определяется порядюм 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 оЬj **ect.**

Для "классических" классов порядок несколью отличается от порядка  
разрешения методов в "новых" классах. Нужно стараться избегать  
множественного наследования или применять его очень аюд^атно.

**Сузи Р.А.**

Агрегация

Контейнеры

Под контейнером обычно понимают объект, основным назначением  
юторого является хранение и обеспечение доступа к другам объектам.  
Контейнеры реализуют отношение "HAS-A" ("ИМЕЕТ") между  
объектами. Встроенные типы, список и словарь — яркие примеры  
контейнеров. Можно построить собственные типы юнтейнеров,  
юторые будут иметь свою логиід^ 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[-l]  
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:

**Сузи RA. Язык программирования Python**

»> s = StackQ  
»> s.push(l)

»> s.push(**2**)

»> s.push("abc")

»> print s.popO  
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 догадатъся, 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  
неюторых действий могут возбуждаться исключения. Например, при  
попытке снять элемент с вершины пустого стека.

Итераторы

Итераторы - это объекты, юторые 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. **1**st, n):

**Сузи Р.А.**

self.n = n  
self. 1st = 1st  
self, current = 0

def iter (selQ:

return self  
def next(selQ:  
if self. 1st:

self.current = (self.current + self.n - 1) % len(self.lst)  
return self.lst.pop(self.current)  
else:

raise Stopiteration

print rangefl, **11**)  
for i in Zahlreim(range(l, 11), 5):  
print i,

Программа выдаст

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

5 10 **62981473**

В этой программе делегировано управление доступом к элементам  
списка (или любого другого контейнера, имеющего метод pop (п) для  
взятия и удаления п -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 слзліае 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:

**Сузи Р.А.**

**lst=[l, 2, 3]**if **1** in **1**st:

**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 несюлько  
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, юторые 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 возможностъю образования циклических ссылок, в Р**3**ДІЮП 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 следзлощего  
примера:

»> import weakref

**Сузи Р.А.**

»>

»> class MyClass(object):

... def str (self):

... return "MyClass"  
»>

**# создается экземпляр класса**

»> s = MyClassO  
»> print s  
MyClass

»> si = weakref.proxy(s) # создается прокси-объект  
»> print si # прокси-объект работает как исждный

MyClass

»> ss = weakref.ref(s) # создается слабая ссылка на него  
»> print ss() # вызовом ссылки получается исждный объект

* **удаляется единственная сильная ссылка на объект**
* **теперь исждного объекта не сзтцествует**

MyClass  
»> del s  
»> print ss()

None

»> print si

Traceback (most recent call last):

File "<stdin>", line 1, in ?

ReferenceError: weakly-referenced object no longer exists

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

Статический метод

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

class A(object):  
def name():

return A. name

name = staticmethod(name)

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

Статичесюму 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 заданнзто в нем фзтікцию. Теперь писать name три  
раза не потребовалось. Декораторов может быть несюлько, 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 РзДЬоп 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 **els.**

Пример использования метода класса можно найти в модзие **tree**пакета nit к (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 процедзру преобразования дерева одного типа  
в дерево другого типа. Эта процедура абстрагируется от деталей  
реализации конкретных типов, описывая обобщенный алгоритм  
преобразования:

m

**Сузи RA. Язык программирования Python**

class Tree:

**#...**

def convert(cls, val):

if isinstance(val, Tree):  
children = [cls.convert(child) for child in val]  
return cls(vaLnode, 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,  
юторые 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  
написатъ класс, назначением юторого буцет создание других классов  
динамически, во время выполнения программы.

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

**Сузи Р.А.**

def ds\_factoiy\_fi(func):  
class X(object):  
pass

setattx(X, fimc. name , ftmc)

return X

Использование будет выглядеть так:

def my\_method(self):  
print "self:", self

My\_Class = cls\_£actoiy\_f(my\_method)  
my\_object = My\_Class()  
my\_obj ect. my\_method()

В этом примере функция **cls\_f actory\_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 юторого  
будут классы. Такой класс, от которого порождаются классы, и  
называется метаклассом.

В Python имеется класс **type,** юторый на деле является метаклассом.  
Вот как с помощью его юнструктора 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 пол}діится класс, эквивалентный следующему:

class My\_Class(object):  
def my\_method(self):  
print "self:", self

**Сузи RA. Язык программирования Python**

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 (ds, name, bases, diet):

print "Выделение памяти под класс", name

return type. new (cIs, name, bases, diet)

def init\_(cls, name, bases, diet):

print "Инициализация класса", name  
return super(My\_T**3**pe, els). init fcls. name, bases, diet)

»> my=My\_Type("X",(), {})

Выделение памяти под класс X  
Инициализация класса X

В этом примере не проиоюдит 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 cis,** чтобы напомнить, что экземпляр,  
над юторым работает программист, является не просто объектом, а  
классом.

Мультиметоды

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

**Сузи RA. Язык программирования Python**

обслуживает:

»> import operator  
»> operator.add(l, **2**)

3

»> operator.add(**1**.**0**, **2**)

3.0

»> operator.add(l, **2**.**0**)

3.0

»> operator.add(l, l+**2**j)

(**2**+**2**j)

»> operator.add(l+**2**j, **1**)

(**2**+**2**j)

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

Для организации собственных мультиметодов можно воспользоваться  
модзлем **Multimethod** (автор Neel Krishnaswami), который легю найти  
в Интернете. Следзлощий пример, адаптированный из документации  
модзля, показывает построение собственного мультиметода:

from Multimethod import Method, Generic, AmbiguousMethodError

* юіассы, для юторых будет определен мзльтиметод  
  class А: pass

class В(А): pass

* функции мзльтиметода  
  def ml (а, Ь): return 'АА'  
  def ш2(а, Ь): return 'АВ'  
  def тЗ(а, Ь): return 'ВА'
* определение мзльтиметода (без одной функции)  
  g = GenericO

g.add\_method(Method((A, А), ml))  
g.add\_method(Method((A, В), m2))  
g.add\_method(Method((B, A), m3))

**Сузи Р.А.**

# применение муаьтиметода  
tiy:

print 'Типы аргзщентов:', 'Результат'  
print'A,A:',g(A(), АО)  
print 'А, В:', g(A(), ВО)  
print 'В, А:', g(B(), АО)  
print 'В, В:', g(B(), ВО)  
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,**юторый позволяет сериализовать объекты.

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

**getinitargs ()**

**getstate ()**

**setstate (state)**

**Должен возвращать кортеж из аргументов,  
который будет передаваться на вход метода**

**in it () при создании объекта.**

**Должен возвращать словарь, в котором  
выражено состояние объекта. Если этот  
метод в классе не определен, то**

**используется атрибут diet , который**

**есть у каждого объекта.**

**Должен восстанавливать объекту ранее  
сохраненное состояние state.**

В следующем примере класс СС управляет своим копированием (точно  
так же экземпляры этого класса смогут консервироваться и

**Сузи Р.А.**

расюнсервироваться 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, diet):

print id(self), "setstate", diet  
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, О, 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.**

**Сузи RA. Язык программирования Python**

**created\_gmtime, copied.** Первый из этих атрибутов может быть  
восстановлен передачей параметра юнструктору. 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 толью то, что нужно для воссоздания объекта, тогда как

атрибут **diet** может содержать много лишнего. Более того,

**diet** может содержать объекты, юторые 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 толью их имена и принадлежность  
модулям.

Для хранения объектов используются не толью простейшие механизмы  
хранения вроде **pickle. dump** / **pickle,** **load** или полки **shelve.**Сериализованные объекты РздЬоп 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  
юнсервировать на передающей стороне и расюнсервировать на  
принимающей.

**Сузи Р.А.**

Критика ООП

Объектно-ориентированный поджд сегодня считается "самым  
передовым". Однаю не следует слепо верить в его всемогущество.  
Отдача (в виде сюрости разработки) от объектного проектирования  
чувствуется толью в больших проектах и в проектах, юторые  
родственны объектному подходу: построение гра**4**»ічесюго интерфейса,  
моделирование систем и т.п.

Также спорна большая гибюсть 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 юторой в каждой ветке  
алгоритма обрабатывается свой тип данных (то есть таюе изменение  
требует редактирования одного места в юде). При ООП изменять  
придется каждый класс, внося в него новый метод (то есть изменения в  
несюльких местах). Зато ООП выигрывает при внесении нового типа  
данных: ведь изменения происходят толью в одном месте, где  
описываются все методы для данного типа. При процедзрном подходе  
приходится изменять несюлью процедзр. Сказанное иллюстрируется  
ниже. Пусть имеются классы **А, В, С** и методы а, Ь, с:

**#ООП**class А:  
def а()  
**defbO  
def с()**

class В:  
def а()  
**defbO**def с()

class С:  
def а()  
**defbO  
def c()**

# процедурный подход

def а(х):  
if type(x) is A  
if type(x) is В  
if type(x) is C

def b(x):  
if type(x) is A  
if type(x) is В  
if type(x) is C

def c(x):  
if type(x) is A  
if type(x) is В  
if type(x) is C

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

**#ООП**

class D:  
def а()  
**defbO  
def с()**

# процедурный подход

def а(х):  
if type(x) is **А:...**if type(x) is **В:...**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 с(х):  
if type(x) is A  
if type(x) is В  
if type(x) is C  
if type(x) is D

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

# процедурный подход

def d(x):  
if type(x) is A  
if type(x) is В  
if type(x) is C

#oon

class A:  
def a()  
**defbO  
def c()  
defdO**

class B:  
def a()  
**defbO  
def c()  
defdO**

class C:  
def a()  
**defbO  
def cQ  
defdO**

Язык программирования Python изначально был ориентирован на

**Сузи RA. Язык программирования Python**

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 вждных  
данных), и классы (югда операции достаточно специ**4**мчны).  
Обобщенная природа фзтікций Python и полимор4мзм, не завязанный  
целиком на наследовании - вот свойства языка Python, позволяющие  
иметь болыцую гибкость в комбинации процедурного и объектно-  
ориентированного подходов.

Заключение

Даже достаточно неформальное введение в ООП потребовало  
определения большого юличества 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-А, USE-А).  
Слушатели полздіили представление о том, что таюе 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 практичесюй работы.

Ссылки

Дэвид Мертц ссылка: httpy/www-106.ibm.com/developerworks/linux/libraiy/l-  
pymeta.html

**Г.Г\. /1.9ЫІЧ lipUI|JCUVir**

Численные алгоритмы. Матричные вычисления

в данной лекции рассматривается пакет Numeric для осуществления  
численных расчетов и выполнения матричных вычислений,  
приводится обзор дрзтих пакетов для наз^ных вычислений.

Numeric Python

Этот раздел в достаточной степени устарел. Сейчас мод}ль называется  
пшпру. В целом, всё, что было в Numeric, доступно и в numpy, но имена  
могут не совпадать. Numeric доступен как numpy/oldnumeric. Рекомендуем  
обратиться к документации.

Numeric Python - это несколью модзлей 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 здобным. Стоит  
заметить, что неюторые 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 МА ;**
* статистичесюго 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 модзля  
**Numeric** могут быть многомерными, то есть иметь более одной  
размерности.

**Сузи Р.А.**

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, югда  
ее аргумент уже является массивом:

»> from Numeric import \*

»> print array([[l, 2], [3, 4], [5, **6**]])

[[**12**]

[3 4]

[5 **6**]]

»> print array([[l, 2, 3], [4, 5, **6**]], float)

[[ 1. 2. 3.]

[ 4. 5. **6**.]]

»> print array([78, 85, 77, 69, 82, 73, 67], 'c')

**[NUMERIC]**

В качестве элементов массива можно использовать следующие типы:  
**Int8-Int32, Unsignedlnt8-Unsignedlnt32, 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)

**Сузи Р.А.**

»> 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 О . Эта фзтікция сразу создает объект-массив  
нужной формы из последовательности.

»> import Numeric

»> print Numeric.reshape("aбpaкaдaбp", (5, -1))

[[а б]

[р а]

[к а]

[Д а]

[бр]]

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

Через атрибут **flat** можно получить одномерное представление  
массива:

»> а = аггау([[1, 2], [3, 4]])  
»> Ь = a.flat

»> Ь

аггау([1, 2, 3, 4])

»> Ь[0] = 9  
»> Ь

аггау([9, 2, 3, 4])

»> а

аггау([[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('TNUMERlC", (3, 2))

[[N U]

[М E]

[RI]]

»> print Numeric.resize('TNUMERlC", (3, 4))

[[N U M E]

[RIC N]

[U M E R]]

Функция **Numeric. zeros** () порождает массив из одних нулей, а  
**Numeric** . **ones (**) - из одних единиц. Единичную матрицу можно  
полздіить 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**]

[111**]]**

»> print Numeric.identity(4)

**[[1** **0** **0** **0**]

**[0** **10** **0**]

**[0** **0** **10**]

[0001**]]**

Для копирования массивов можно использовать метод **сору** ():

»> import Numeric

»> а = Numeric.arrayrange(9)

»> a.shape = (3, 3)

»> print a

**Сузи Р.А.**

[[О 1 2]

[3 4 5]

[6 7 8]]

**# операция над юпией**

»> а1 = а.соруО  
»> а1[0, 1] = -1  
»> print al  
[[0-12]

[3 4 5]

[6 7 8]]

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

»> a.tolistO

[[0, 1, 2], [3, **4,** 5], [6, 7, 8]]

Срезы

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

»> import Numeric

»> а = 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[l,2]**

**9**

**»> print a[l,:]**

**[ 7 8 9 10 11 12]  
»> print a[l]**

**[ 7 8 9 10 11 12]  
»> print a[:,l]**

**# элемент 1,2**

**# строка 1**

**# тоже строка 1**

**# столбец 1**

**Сузи Р.А.**

[ 2 8 14 20]

»> print а[-2,:] # предпоследняя строка

[13 14 15 16 17 18]

»> print а[0:2Д:3] # окно 2x2  
[[2 3]

[8 9]]

* **каждый третий элемент строки 1**
* **элементы строк в обратном порядке**

»> print а[1,::3]

**[7 10]**

»> print а[:,::-1]

[[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 юпирует массив (как это имеет место со списками), а дает  
доступ к неюторой части массива. Далее в примере меняется на 0  
каждый третий элемент строки 1:

»> а[1,::3] = Numeric.аггау([0,0])

»> print а  
[[ 1 2 3 4 5 6]

[ о 8 9 о 11 12]

[13 14 15 16 17 18]

[19 20 21 22 23 24]]

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

»> import Numeric

»> а = 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]

**Сузи Р.А.**

**[16 17 18]]**

**[[19 20 21]**

**[22 23 24]]]]  
»> print а[0,...]  
[[[ 1 2 3]**

**[4 5 6]]**

**[[7 8 9]**

**[10** **11** **12**]]]

**»> print а[0,:,:,0]  
[[1 4]**

**[ 7 10]]**

**»> print а[0,...,0]  
[[1 4]**

**[ 7 10]]**

**# О-й блок**

**# срез по первой и последней размерностям**

**# то же, но с использованием многоточия**

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**

**add(х, у),**subtract **(х, у)**

**multiply(х, у),**divide {х, **у)**

remainder(x, **у),  
fmod(x, у)**

**power(х, у)**

**sqrt(х)**

**negative(х),  
absolute(х) ,**fabs**(х)**

ceil**(х), floor(х)**

Описание

Сложение и вычитание

Умножение и деление

Получение остатка от деления (для целых  
чисел и чисел с плавающей запятой)

Возведение в степень  
Извлечение юрня квадратного

Смена знака и абсолютное значение

Наименьшее (наибольшее) целое, большее  
(меньшее) или равное аргументу

**Язык программирования Python**

**hypot(x, у)**

**sin (х), cos(х),  
tan(х)  
arcsin(х),  
arccos(х),  
arctan(х)**

**arctan2(х, у)**

**sinh(х), cosh(х) ,  
tanh(х)**

**arcsinh(х),  
arccosh(х) ,  
arctanh(х)**

**ехр (х)**

**log(х), loglO(х)**

**maximum(х, у),  
minimum(х, у)**

**conjugate(х)**

**equal(х, у),  
not\_equal(х, у)**

**greater (х, у),  
greater\_equal(х, у)  
less (х, у),  
less\_equal(х, у)**

**logical\_and(х, у),  
logical\_or(х, у)**

**logical\_xor(х, у)**

**logical\_not(х)**

**bitwise\_and(х, у),  
bitwise\_or(х, у)**

**bitwise\_xor(х, у)  
invert(х)**

**left\_shift(х, п) ,  
right shift(х, п)**

Длина гипотенузы (даны длины двух  
катетов)

Тригонометрические функции

Обратные тригонометрические функции

Арктангенс от частного аргумента  
Гиперболические функции

Обратные гиперболические функции  
Экспонента (е^ )

Натуральный и десятичный логарифмы  
Максимум и минимум  
Сопряжение (для юмплексных чисел)  
Равно, не равно

Больше, больше или равно

Меньше, меньше или равно

Логические И, ИЛИ

Логическое исключающее ИЛИ  
Логические НЕ

Побитовые И, ИЛИ

Побитовое исключающее ИЛИ  
Побитовая инверсия  
Побитовые сдвиги влево и вправо на п  
битов

**Сузи RA. Язык программирования Python**

Перечисленные фз^кции являются объектами типа и fun с и  
применяются к массивам поэлементно. Эти функции имеют  
специальные методы:

**accumulate** () Аккумулирование результата.

**outer О** Внешнее "произведение",

**reduce** О Сокращение,

**reduceat** **()** Сокращение в заданных точках.

**# T.e. 1+2+3+4**

* **T.e. [1+3 2+4]**
* **T.e. [1+2 3+4]**
* **T.e. [1 1+2 1+2+3 1+2+3+4]**

**# T.e. [0+1+2 3+4+5 6+7+8+Э]  
#T.e. [[1+3 1+4] [2+3 2+4]]**

**»> from Numeric import add  
»>add([[l,2U3,4]], [[1, QUO, 1]])  
array([[2, 2],**

**[3, 5]])**

**»>add([[l,2U3,4]], [1,0])  
array([[2, 2],**

**[4, 4]])**

**»>add([[l,2],[3,4]], 1)  
array([[2, 3],**

**[4, 5]])**

**»> add.reduce([l, 2, 3, 4])**

**10**

**»> add.reduce([[l, 2], [3, 4]], 0)  
array([4, 6])**

**»> add.reduce([[l, 2], [3, 4]], 1)  
array([3, 7])**

**»> add.accumuiate([l, 2, 3, 4])  
array([ 1, 3, 6, 10])**

**»> add.reduceat(range(10), [0, 3, 6])  
arra}^[ 3, 12, 30])**

**»> add.outer([l,2], [3,4])  
array([[4, 5],**

**[5, 6]])**

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

Методы **accumulate (),** **reduce () и reduceat** **()** принимают  
необязательный аргумент - номер размерности, используемой для

**Сузи Р.А.**

**Язык программирования Python**

**По умолчанию применяется нулевая**

соответствующего действия,  
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)

**, 1.41421356])**

»> sqrt(a, r)  
**аггаз^[ 0. ,1.**

»> print r

[ 0. 1. 1.41421356]

Предупреждение:

He следует использовать в качестве приемника результата массив,  
юторый 4мгурирует в предыдущих аргументах функции, так как при  
этом результат может быть испорчен. Следзлощий пример показывает  
именно такой вариант:

»> import Numeric

»> m = Numeric.array([0, 0, 0, 1, 0, 0, 0, 0])

»> add(m[:-l], m[l:], m[l:])  
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:

**Язык программирования Python**

**Функция**

**sum(a, axis)  
cumsum(a, axis)  
product(a, axis)**

Аналог функции

**add.reduce(a, axis)  
add.accumulate(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** за<азывает 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 з^олчанию номер  
измерения (третий аргумент) равен нулю.

»> import Numeric

»> а = 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 78 9]]

**Сузи Р.А.**

»> print Numeric.take(a, [1], 1)

**[[1]**

**[6]**

**[И]**

[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. Реззльтат **Numeric. take (а, [[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.traceQ

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

а Исходный массив.

**axisl**

**offset** Смещение вправо от "главной" диагонали (по умолчанию 0).  
Первое из измерений, на юторых берется диагональ (по  
умолчанию 0).

**axis2**

Второе измерение, образующее вместе с первым плоскость.

на юторой и берется диагональ. По умолчанию **axis2 = l.**

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

»> а = Numeric.reshape(Numeric.arrayrange(16), (4, 4))

**Сузи Р.А.**

»> print а  
[[0 12 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.chooseQ

Эта функция использует один массив с целыми числами от О до п для  
выбора значения из одного из заданных массивов:

»> а = Numeric.identity(4)

»> bO = Numeric.reshape(Numeric.arrayrange(16), (4, 4))

»> Ы = -Numeric.reshape(Numeric.arrayrange(16), (4, 4))

»> print Numeric.choose(a, (bO, **bl))**

**[[0123]**

**[ 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

Сравнение **а** и Ь с заданными

**allclose (а, Ь[,** относительными **ер s** и абсолютными **А**

**Язык программирования Python**

**clip (a, a\_min,  
a\_max)**

**compress(cond, a[,  
axis] )**

***concatenate*(a[,  
axis] )**

**convolve (a, b[,  
mode])**

**argsort(a[,** axis]

**Сузи Р.А.**

**eps[, А]])**

**alltrue(a[, axis]  
argmax(a[,** axis])

**argmin(a[**, axis]**)**

погрешностями. По умолчанию **eps** равен

1. **Ое-1,** а **А = 1. Ое-8.**

Логичесюе **И** по всей оси **axis** массива **а**

Индекс максимального значения в массиве  
по заданному измерению **axis**Индекс минимального значения в массиве  
по заданному измерению **axis**Индексы отсортированного массива, такие,  
что **take (а, argsort (а,  
axis) ,axis)** дает отсортированный  
массив **а,** как если бы было выполнено  
**sort(а, axis)**

**array(a[, type])**

**arrayrange (start [,  
stop[, step[,  
type]]])**

**asarray(a[, type[,  
savespace]])**

**choose(a,**

**(bO,...,bn))**

Создание массива на основе  
последовательности **а** данного типа **type**

Аналог **range** () для массивов

То же, что и **array О** , но не создает новый  
массив, если **а** уже является массивом.  
Создает массив на основе элементов,  
взятых по индексам из **а** (индексы от **О** до **п**включительно). Формы массивов **а, Ь1,**

**. .., bn** должны совпадать  
Обрубает значения массива **а** так, чтобы  
они наждились между значениями из  
**a\_min** и **а\_тах** поэлементно  
Возвращает массив толью из тех элементов  
массива **а,** для юторыхусловие **cond**истинно (не нуль)

Соединение двух массивов (юнкатенация)  
по заданному измерению **axis** (по  
умолчанию - по нулевой)

Свертка двух массивов. Аргумент **mode**может принимать значения О, 1 или 2

Взаимная юрреляция двух массивов.

**Язык программирования Python**

**indices(dims[,  
type])**

**fromstring(s[,  
count[, type]]**

**identity (n)**

**fromfunction(f,  
dims)**

***dumps*(obj**

**dump(obj, file!**

**dot(a, b)**

**Сузи Р.А.**

**cross\_correlate(а,  
Ь[, mode])**

**cumproduct(а[,  
axis] )**

**cumsum(a[, axis])**

**diagonal(a[, к[,  
axisl[, axis2]]])**

Параметр mode может принимать значения  
О, 1 или 2

Произведение по измерению **axis** массива  
а с промежуточными результатами

Суммирование с промежуточными  
результатами

Взятие к -й диагонали массива а в  
плоскости измерений **axisl и axis2**Внутреннее (матричное) произведение  
массивов. По определению:  
**innerproduct(а, swapaxes(Ь,**

**-1,** **-2)** **),** т.е. с переставленными  
последними измерениями, как и должно  
быть при перемножении матриц

Запись массива а (в двоичном виде) в  
открытый файловый объект **file.** Файл  
должен быть открыт в бинарном режиме. В  
файл можно записать несюлью объектов  
подряд

Строка с двоичным представлением объекта  
оЬ j

Строит массив, получая информацию от  
функции f (), в качестве аргументов  
которой выступают значения юртежа  
индексов. Фактически является  
сокращением для

f(\* **tuple**(**indices(dims)))**

Создание массива на основе бинарных  
данных, хранящихся в строке

Возвращает двумерный массив формы (п,  
п)

Возвращает массив индексов заданной  
длины по каждому измерению с  
изменением поочередно по каждому  
изменению. Например, **indices** ( [2 ,

**Язык программирования Python**

1. ] ) [ 1 ] дает двумерный массив [ [ О,

**1] , [О, 1] ].**

**load(file)  
loads(s)**

***nonzero*(a)**

**ones(shape[,  
type])**

**outerproduct(a, b)  
product(a[, axis])**

**put (a, indices, b)**

**putmask(a, mask,  
b)**

**ravel(a)**

**repeat (a, n[,  
axis] )**

**reshape (a, shape)**

**innerproduct(а, Ь)**

Внутреннее произведение двух массивов  
(по общему измерению). Для успешной  
операции а . **shape** [ **-1** ] должен быть  
равен Ь . **shape** [ **-1** **].** Форма результата  
будет а . **shape [:-1 ] +**

**Ь . shape** [ : **-1** **].** Элементы пропадающего  
измерения попарно умножаются и  
ползліающиеся произведения суммируются  
Чтение массива из файла **file.** Файл  
должен быть открыт в бинарном режиме  
Возвращает объект, соответствующий  
бинарному представлению, заданному в  
строке

Возвращает индексы ненулевых элементов  
одномерного массива

Массив из единиц заданной формы **shape**и обозначения типа **type**Внешнее произведение а и Ь

Произведение по измерению **axis** массива  
а

Присваивание частям массива, а [ п ] =  
**Ь[п]** для всех индексов **indices**Присваивание а элементов из Ь, для  
которых маска **mask** имеет значение истина

Превращение массива в одномерный.  
Аналогично **reshape (а, (-1,))**

Повторяет элементы массива а п раз по  
измерению **axis**

Возвращает массив нужной формы (нового  
массива не создает). Количество элементов  
в исходном и новом массивах должно  
совпадать

***resize{а,* shape)**

**searchsorted(а, i;**

**shape(a)**

**sometrue(a[,  
axis] )**

**sort (a [, axis])**

**sum(a[, axis])**

**swapaxes (a, axisl,  
axisl)**

**take(a, indices[,  
axis] )**

**trace (a[, к[,  
axisl [, axis2]]])**

**transpose(a[,  
axes])**

**where(cond, al,  
a2)**

**zeros(shape[,  
type])**

**Язык программирования Python**

**Возвращает массив с произвольной новой  
формой shape. Размер исждного массива  
не важен**

**Для каждого элемента из і найти место в  
массиве а. Массив а должен быть  
одномерным и отсортированным. Результат  
имеет форму массива і**

**Возвращает форму массива а  
Логичесюе ИЛИ по всему измерению axis  
массива а**

**Сортировка элементов массива по  
заданному измерению**

**Суммирование по измерению axis  
массива а**

**Смена измерений (частный случай  
транспонирования)**

**Выбор частей массива а на основе индексов  
indices по измерению axis**

**Сумма элементов вдоль диагонали, то есть  
add.reduce(diagonal(а, к,  
axisl, axis2))**

**Перестановка измерений в соответствии с  
axes, либо, если axes не заданы -  
расположение их в обратном порядке**

**Выбор элементов на основании условия  
cond из а 1 (если не нуль) и а2 (при нуле)  
поэлементно. Равносилен  
choose(not\_equal(cond, 0), (у,**

**х) ). Формы массивов-аргументов а1 и а2  
должны совпадать**

**Массив из нзлей заданной формы shape и  
обозначения типа type**

**В этой таблице в качестве обозначения типа type можно зл^азывать  
рассмотренные выше юнстанты: Int, Float и т.п.**

**Сузи RA. Язык программирования Python**

Модуль **Numeric** также определяет константы **е** (число е) и рі (число  
пи).

Модуль LinearAlgebra

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

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

»> import Numeric, LinearAlgebra  
»> print LinearAlgebra.determinant(

Numeric.array([[l, -2],

[1, 5]]))

7

Функция **LinearAlgebra.solve\_linear\_equations()**

решает линейные уравнения вида **ax=b** по заданным аргументам **а** и Ь:

»> import Numeric, LinearAlgebra

»> а = Numeric.array([[1.0, 2.0], [0.0, 1.0]])

»> b = Numeric.array([1.2, 1.5])

»> **X** = LinearA^ebra.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 нзлевой определитель, система имеет не  
единственное решение и возбуждается исключение

**LinearAlgebraError:**

»> а = Numeric.аітау([[1.о, 2.0], [0.5, 1.0]])

»> **X** = LinearA^ebra.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.LinA^rror: Singular matrix

Функция **LinearAlgebra . inverse** **()** находит обратнзто матрицу.  
Однаю не следует решать линейные уравнения с помощью  
**LinearAlgebra** . **in** verse () з^ножением 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 . eigen vectors** **()** - пару:  
собственные значения, собственные вектора:

»> from Numeric import array, dot

»> from LinearAlgebra import eigenvalues, eigenvectors

»> a = array([[-5, 2], [2, -7]])

»> Imd = e^envalues(a)

»> print "Собственные значения:", Imd  
Собственные значения: [-3.76393202 -8.23606798]

»> (Imd, v) = eigenvectors(a)

»> print "Собственные вектора:"

Собственные вектора:

**»> print V**

[[ 0.85065081 0.52573111]

[-0.52573111 0.85065081]]

»> print "Проверка:", dot(a, v[0j) - v[0] \* lmd[0]

Проверка: [ -4.440892lOe-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

**Сузи RA. Язык программирования Python**

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

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

»> import RandomArray

»> print RandomArray.random(10) # массив из 10 псевдослучайных чис(  
[ 0.28374212 0.19260929 0.07045474 0.30547682 0.10842083 0.14049(  
0.01347435 0.37043894 0.47362471 0.37673479]

»> print RandomArray.random([3,3]) # массив 3x3 из псевдослучайных ч  
[[0.53493741 0.44636754 0.20466961]

[ 0.8911635 0.03570878 0.00965272]

[ 0.78490953 0.20674807 0.23657821]]

Функция **RandomArray. randint** **()** для полз^гения массива  
равномерно распределенных чисел из заданного интервала и заданной  
формы:

»> print RandomArray.randint(l, 10, [10])

[8 **19975253** 2]

»> print RandomArray.randint(l, 10, [10])

[2 **25577343 7]**

**помощью**

Можно получать и случайные перестановки

**RandomArray.permutation():**

»> print RandomArray.permutation(6)

**[40 1325]**

»> print RandomArray.permutation(6)

[12 **0354]**

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

»> print RandomArray.normal(0, 1, 30)

[-1.0944078 1.24862444 0.20415567-0.74283403 0.72461408-0.57834!

**Сузи RA. Язык программирования Python**

0.30957144 0.8682853 1.10942173 -0.39661118 1.33383882 1.5481861  
0.18814971 0.89728773 -0.86146659 0.0184834 -1.46222591 -0.784274  
1.09295738 -1.09731364 1.34913492 -0.75001568 -0.11239344 2.73692]  
-0.19881676-0.49245331 1.54091263-1.81212211 0.46522358-0.08338

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.** | Многомерное нормальное |
| **cov, shape=[])** | распределение |
| **negative binomial(trials, p, shape=[])** | Негативное биномиальное |
| **noncentral F(dfn, dfd,** | Нецентральное F- |
| **nconc, shape=[] )** | распределение |
| **noncentral chi square(df,** | Нецентральное хи-квадрат |
| **nconc, shape=[])** | распределение |
| **normal(mean, std, shape=[])** | Нормальное распределение |
| **permutation**(n) | Слзліайная 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=l, shape=[])** | Случайное целое |
| **standard normal(shape=[])** | Стандартное нормальное распределение |

**Сузи RA. Язык программирования Python**

**uniform (min, max, shape=** [ ] ) Равномерное распределение

Заключение

в этой лекции рассматривался набор модзлей 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 РзДЬоп: ссылка: httpУ/[www.scipy.org/](http://www.scipy.org/)

Обработка текстов. Регулярные выражения. Unicode

в этой лекции дается кратюе представление о возможностях языка  
Python по обработке текстовой информации. Рассмотрены синтаксис и  
семантика регулярных выражений, а также неюторые вопросы  
использования 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 зліебным примером может служить nit к (the Natural Language  
Toolkit).

Стоит отметить проект PyParsing (сайтгсылка:

httpy/pyparsing.sourceforge.net), с помощью которого можно организовать  
обработід^ 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  
языюв.

в языке и стандартной библиотеке за неюторыми 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, юторые не могут передать всех символов. Без  
такой установки будет непонятно, почему преобразование из  
восьмибитной юдировки называется **decode** (деюдирование). Для  
внешнего представления можно с успехом использовать юдировку UTF-  
8, хотя, юнечно, это зависит от решаемых задач.

Кодировка Python-программы

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

* coding: коі8-г  
  или (под Windows):
* coding: ср1251

Могут быть и другие варианты:

**#-\*-** coding: latin-1

* coding: utf-8
* coding: mac-cyrillic -\*-
* -\*- coding: iso8859-5 -\*-

**Сузи RA. Язык программирования Python**

Полный перечень юдировок (и их псевдонимов):

»> import encodings.aliases  
»> print encodings.aJiases.aliases  
{'iso\_ir\_6': 'ascii', 'maccyrillic': 'mac\_cyrillic',

'iso\_celtic': 'iso8859\_14', 'ebcdic\_cp\_wt': 'cp037',

'ibmSOO':'cpSOO',...

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

sys:l: DeprecationWaming: Non-ASCII character '\xfO' in file example.py

on line 2, but no encoding declared;

see httpy/[www.python.org/peps/pep-0263.html](http://www.python.org/peps/pep-0263.html) for details

Строковые литералы

Строки можно задать в программе с помощью строювых литералов.  
Литералы записываются с использованием апострофов ', кавычек "  
или этих же символов, взятых трижды. Внутри литералов обратная  
юсая черта имеет специальное значение. Она служит для ввода  
специальных символов и для указания символов через коды. Если перед  
строковым литералом поставлено г, обратная юсая черта не имеет  
специального значения ( г от английсюго слова raw, строка задается  
"как есть"). Unicode-литералы задаются с пре(}»іксом и. Вот несюлью  
примеров:

si = "строка 1"  
s2 = г'\1\2'

s3 = """apple\ntree""" # \n - символ перевода строки  
s4 = """apple

tree""" # строка в утроенных кавычках может иметь внутри переводы сі

s5 = '\x73\65'

ul = u'Unicode literal"

u2 = u'\u0410\u0434\u0440\u0435\u0441'

Обратная юсая черта не должна быть последним символом в литерале,  
то есть, **"str\** " вызовет синтаксическую ошибі^^.

М<азание юдировки позволяет применять в Unicode-литералах  
указаннзто в начале программы юдирові^^. Если юдировка не указана,  
можно пользоваться только юдами символов, заданными через  
обратную юсую черту.

Операции над строками

к операциям над строками, юторые имеют специальнзто  
синтаксическую поддержку в языке, относятся, в частности  
конкатенация (склеивание) строк, повторение строки, форматирование:

»> print "А" + "В", "А"\*5, "%s"%'A"

АВАААААА

В операции форматирования левый операнд является строкой формата,  
а правый может быть либо юртежем, либо словарем, либо неюторым  
значением другого типа:

»> print "%і" % 234  
234

»> print "%i %s %3.2f' % (5, "ABC", 23.45678)

5 ABC 23.46  
»> a = 123  
»> b = [1, 2, 3]

»> print "%(a)r %(b)s" % vars()

123: [1, 2, 3]

Операция форматирования

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

**Сузи Р.А.**

синтаксис

[ключ][флаг\*][шир][.точность][длина\_типа]спецификатор  
ключ:"(" символ за исключением круглых скобок\* ")"  
флаг:"+" I I пробел | "#" | "О"  
шир:("1"... "9")("0"... "9")\* I "\*"  
точность: ("1"... "9")\* | "\*"  
длина\_типа: "а"... "z" | "А"... "Z"  
спецификатор: "а"... "z" | "А"... "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

О **флаг** Заполнение нулями слева

**Язык программирования Python**

**Выравнивание по левому краю  
Обязательный вывод знака числа  
Использовать пробел на месте знака числа  
Знаювое целое  
Беззнаковое целое  
Восьмеричное беззнаювое целое  
Шестнадцатеричное беззнаювое целое (со  
строчными или прописными латинскими  
буквами)**

**Число с плавающей запятой в формате с  
экспонентой**

**Число с плавающей запятой**

**Число с плавающей точюй в более юротюм  
g, G спецификатор написании (автоматически выбирается е**

**или f)**

**Одиночный символ (целое число или  
односимвольная строка)**

**Любой объект, приведенный к строке  
функцией г ер г ()**

**Любой объект, приведенный к строке  
функцией str ()**

**Знак процента. Для задания одиночного  
процента необходимо записатъ %%**

**флаг  
+ флаг**

**пробел флаг  
d, і спецификатор**

**U спецификатор**

**о спецификатор**

**X, X спецификатор**

1. **Е спецификатор**
2. **F спецификатор**

**спецификатор**

**спецификатор**

**спецификатор**

**% спецификатор**

**Индексъі и срезы**

**Следует напомнитъ, что строки являются неизменчивыми  
последователъностями, поэтому к ним можно применятъ операции  
взятия элемента по индексу и срезы:**

**»> S = '‘транспорт"**

**»> print s[0], s[-l]  
т т**

**»> print s[-4:]**

**Сузи Р.А.**

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.Temp]ate('$a + $b = ${c}")  
»> a = 2  
»> b = 3  
»> c = a + b

**Сузи RA. Язык программирования Python**

»> print tpLsubstitute(varsO)

1. + 3 = 5

»> del c # удаляется имя с  
»> print tpLsafe\_substitute(vars())

2 + 3 = $c

»> print tpLsubstitute(vars(), c=a+b)

2 + 3 = 5

»> print tpLsubstitute(varsO)

Traceback (most recent call last):

File "/home/rnd/tmp/P3^hon-2.4b2/Lib/string.py", line 172, in substitute  
return self.pattern.sub(convert, self.template)

File "/home/rnd/tmp/P3^hon-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 с}мгурные  
скобки при написании ключа в шаблоне.

Методы строк

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

**Метод  
center(w)  
count(sub)**

**encode([enc[,  
errors]])**

**endswith*(suffix)***

**Описание**

**Центрирует строі^^ в поле длины w  
Число вхождений строки sub в строке  
Возвращает строід^в юдировке епс.  
Параметр errors может принимать  
значения "strict" (по умолчанию),  
"ignore", "replace" или  
"xmlcharrefreplace"**

**Оканчивается ли строка на suffix**

**Сузи Р.А.**

**expandtabs**

**find(sub [,start  
[,end]])**

**index(sub[  
end]])**

**start[,**

**isalnum()  
isalpha ()**

**isdecimal()**

**isdigit ()  
islower()  
isnumeric()**

**isspace ()**

**Язык программирования Python**

**( [ tab size** ] ) Заменяет символы табзляции на

пробелы. По умолчанию **tabsize=8**

Возвращает наименьший индекс, с  
юторого начинается вхождение  
подстроки sub в строку. Параметры  
**start Hend** ограничивают поиск  
окном **start: end,** но возвращаемый  
индекс соответствует исходной строке.  
Если подстрока не найдена,  
возвращается -1

Аналогично f ind (), но возбуждает  
исключение **ValueError** в случае  
неудачи

Возвращает **True,** если строка  
содержит толью буквы и цифры и  
имеет ненуаевзло длину. Иначе —

**False**

Возвращает **True,** если строка  
содержит толью буквы и длина  
ненулевая

Возвращает **True,** если строка  
содержит толью десятичные знаки  
(толью для строк Unicode) и длина  
ненулевая

Возвращает **True,** если содержит  
толью ци4ры и длина ненулевая  
Возвращает **True,** если все буквы  
строчные (и их более одной), иначе —

**False**

Возвращает **True,** если в строке толью  
числовые знаки (толью для Unicode)

Возвращает **True,** если строка состоит  
толью из пробельных символов.  
Внимание! Для пустой строки  
возвращается **False**

**upper()**

**translate(table[,  
do] )**

**j oin(seq)**

**lower()**

**Istrip ()**

**replace(old, new[,  
n] )**

**rstrip ( )**

**split([sep[,** n]**])**

**startswith*(prefix)  
strip*()**

**translate(table)**

**Язык программирования Python**

Соединение строк из  
последовательности seq через  
разделитель, заданный строюй  
Приводит CTpoiq^ к нижнему регистру  
букв

Удаляет пробельные символы слева

Возвращает юпию строки, в юторой  
подстроки old заменены **new.** Если  
задан параметр п, то заменяются только  
первые п вхождений

Удаляет пробельные символы справа  
Возвращает список подстрок,  
получающихся разбиением строки а  
разделителем **sep.** Параметр п  
определяет максимальное юличество  
разбиений (слева)

Начинается ли строка с подстроки

prefix

Удаляет пробельные символы в начале  
и в конце строки

Производит преобразование с помощью  
таблицы перекодировки **table,**содержащей словарь для перевода юдов  
в коды (или в **None,** чтобы удалить  
символ). Для Unicode-строк  
То же, но для обычных строк. Вместо  
словаря - строка переюдировки на 256  
символов, юторзло можно  
сформировать с помощью функции  
**string.такеtrans()** .  
Необязательный параметр dc задает  
строку с символами, юторые  
необходимо удалить  
Переводит буквы строки в верхний  
регистр

в следзтощем примере применяются методы **splitO и join()** для  
разбиения строки в список (по разделителям) и обратное объединение  
списка строк в строку

»> **S** = 'This is an example."

»> 1st = s.spM("")

»> print 1st

['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 fit in filenames:

... if fi3.1ower().endswith(".txt"):  
print fit

file.txt

str.txt

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

import string

text = open(string. file [:-l]).read()

start = 0  
while 1:

found = text.find("def", start)  
if found == -1:  
break

print text[found:found + 60].split("(')[0]  
start = found + 1

**Сузи RA. Язык программирования Python**

Важным для преобразования текстовой информации является метод  
**replace** **(),** юторый рассматривается ниже:

»> а = "Это текст, в ютором встречаются запятые , поставленные не'  
»> Ь = a.repJaceC'

»> 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 юличества строк в одну. Лучше  
накапливать строки в списке, а затем с помощью **join** **()** собирать в  
одну строку:

»> for і in xrange(lOOO):

... a += str(i) # неэффективно!

»> a = "".join([str(i) for i in xrange(lOOO)]) # более эффективно

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

Модуль StringlO  
**в некоторых случаях желательно работать со строкой как с файлом.  
Модуль StringlO как раз дает такую возможность.**

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

**Сузи RA. Язык программирования Python**

import StringlO  
my\_string = "1234567890"  
fl = StringlO.StringlOO  
2 = StringlO.StringIO(my\_string)

Далее c файлами f 1 и f2 можно работать как с обычными файловыми  
объектами.

Для пол}ліения содержимого такого файла в виде строки применяется  
метод **getvalue** **():**

fl.getvalueO

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

Модуль difflib

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

Функция dif f lib . **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.

**Сузи Р.А.**

Требуемое количество ближайших строк,  
cutoff

Коэффициент (из диапазона [О, 1]) необходимого зфовня совпадения  
строк. Строки, которые при сравнении с **word** дают меньшее значение,  
игнорируются.

Следующий пример показывает функцию

dif f lib . **get\_close\_matches** **()** в действии:

»> import unicodedata

»> names = [unicodedata.narne(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 далею не всегда. Например, в методах **find() и  
replace!)** задается всего одна строка. В реальных задачах такая  
однозначность встречается довольно редю, чаще требуется найти или  
заменить строки, отвечающие неюторому 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 модзль  
ге. В следующем примере регулярное выражение помогает выделить из  
текста все числа:

»> import re  
»> pattern = r"[0-9]+"

»> number\_re = re.compile(pattem)

**Сузи Р.А.**

»> nun±)er\_re.findall("122 234 65435")  
['122', '234', '65435']

в этом примере шаблон **pattern** описывает множество строк,  
юторые состоят из одного или более символов из набора " О " , " 1" ,

..., "9" . Фзтзкция ге. **compile** О юмпилирует шаблон в

специальный Regex-объект, юторый имеет несколько методов, в том  
числе метод f **indall** **()** для пол}діения списка всех непересекающихся  
вхождений строк, удовлетворяющих шаблону, в заданную строку.

То же самое можно было сделать и так:

»> import ге

»> 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 слздгае лздгше записывать строковые литералы именно так,  
чтобы исключить влияние специальных последовательностей,  
записываемых через обратную юсую черту.

Синтаксис регулярного выражения

Синтаксис регулярных выражений в Python почти таюй 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 использзлот

**Сузи RA. Язык программирования Python**

шаблон по-разному. Так, **search** **()** ищет первое вжждение строки,  
удовлетворяющей шаблону, в заданной строке, а **match** **()** требует,  
чтобы строка удовлетворяла шаблону с самого начала.

Символы, имеющие специальное значение в записи регулярных  
выражений:

**[**

**II іг "**

**II \_j\_ II**II **9** II

**"{m,n}**

**Символ**

Что обозначает в регулярном выражении  
Любой символ  
Начало строки  
Конец строки

Повторение фрагмента нзль или более раз (жадное)  
Повторение фрагмента один или более раз (жадное)  
Предыдущий фэагмент либо присутствует, либо отсутствует  
Повторение предыдзлцего фрагмента от m до п раз  
включительно (жадное)

Любой символ из набора в сюбках. Можно задавать  
диапазоны символов с идущими подряд юдами, например:  
a-z

Любой символ не из набора в скобках

**II II**

Обратная косая черта отменяет специальное значение  
следующего за ней символа

**II \* 9 II**

**II ? II**

Фрагмент справа или фрагмент слева  
Повторение фрагмента нзль или более раз (не жадное)  
Повторение фрагмента один или более раз (не жадное)  
" Повторение предыдзлцего фрагмента от m до п раз

{т, п} ? " включительно (не жадное)

Если А и В - регулярные выражения, то их конкатенация АВ является  
новым регулярным выражением, причем конкатенация строк а и Ь будет  
удовлетворять АВ, если а удовлетворяет А и Ь удовлетворяет В. Можно  
считать, что конкатенация - основной способ составления регулярных  
выражений.

**Сузи RA. Язык программирования Python**

Сюбки, 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 "а" соответствует регулярному  
выражению **"[a-z]",** строка **"fruit"** соответствует  
**"fruit I vegetable",** а вот строка **"apple"** не соответствует  
шаблону **"pineapple".**

В таблице ниже вместо регвыр может быть записано регзлярное  
выражение, вместо имя - иденти4мкатор, а флаги будут рассмотрены  
ниже.

**"(?<!регвыр)'  
" (?**

**Р<имя>регвыр]**

**Обозначение  
"(регвыр)"**

**"(?:регвыр)"**

**"(?=регвыр)"  
"(?!регвыр)"**

**"(?<=регвыр)**

Описание

Обособляет регулярное выражение в скобках и  
выделяет группу

Обособляет регулярное выражение в скобках без  
выделения грзппы

Взгляд вперед: строка должна соответствовать  
заданному регулярному выражению, но  
дальнейшее сопоставление с шаблоном начнется с  
того же места

То же, но с отрицанием соответствия

Взгляд назад: строка должна соответствовать, если  
до этого момента соответствует регулярному  
выражению. Не занимает места в строке, к  
юторой применяется шаблон. Параметр регвыр  
должен быть фіксированной длины (то есть, без

**ч П и \* п ^**

То же, но с отрицанием соответствия  
„ Выделяет именованную группу с именем имя  
Точно соответствует выделенной ранее

**Язык программирования Python**

**"(?#регвыр)"**

**"(?(имя)рвіI  
рв2) "**

**"(?флаг)"**

именованной группе с именем имя  
Комментарий (игнорируется)

Если группа с номером или именем имя  
оказалась определена, результатом будет  
сопоставление с **рві,** иначе - с **рв2.** Часть | рв2  
может отсутствовать

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 |
| **"\А"** | Промежуток перед началом всей строки (почти аналогично **" ^" )** |
| **"** \ Z **"** | Промежуток перед юнцом всей строки (почти аналогично **"$" )** |
| **"\Ь"** | Промежуток между символами перед словом или после него |
| **"\В"** | Наоборот, не соответствует промежутку между символами на границе слова |
| **"\d"** | Цифра. Аналогично **" [** 0 - **9 ] "** |
| **"\s"** | Любой пробельный символ. Аналогично **"**  **[\t\n\r\f**\ѵ]**"** |
| **"\S"** | Любой непробельный символ. Аналогично **"**  **[ ''\t\n\r\f**\ѵ] **"** |
| **"\w"** | Любая цифра или буква (зависит от флага **LOCALE )** |
| **"\W"** | Любой символ, не являющийся цис{рой или буквой (зависит от флага **LOCALE )** |

Флаги, используемые с регулярными выражениями:

**Сузи RA. Язык программирования Python**

Флаги, используемые с регулярными выражениями:

"(?і)", ге.І, re.IGNORECASE  
Сопоставление проводится без зліета регистра букв.

"(?L)", re.L, re.LOCALE

Влияет на определение буквы в **"\w", "\W", "\b", "\В"** в

зависимости от текущей культурной среды (locale).

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 " \ п".  
"(?х)", ге.Х, re.VERBOSE

Если задан, пробельные символы, не экранированные в шаблоне  
обратной юсой чертой, являются незначащими, а все, что расположено  
после символа **"#",** — юмментарии. Позволяет записывать регзлярное  
выражение в несколью строк для улучшения его читаемости и записи  
комментариев.

"(?и)", ге.и, re.UNlCODE

В шаблоне и в строке использован Unicode.

Методы объекта-шаблона  
**в результате успешной компиляции шаблона функцией  
ге. compile О ползліается шаблон-объект (он именуется  
SRE\_Pattern ), который имеет несюлько методов, неюторые из них  
будут рассмотрены. Как обычно, подробности и информация о  
дополнительных аргументах - в документации по Python.**

**Сузи RA. Язык программирования Python**

Сопоставляет строку s с шаблоном, возвращая в слз^ае 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^O])

Разбивает строід^ 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 реззльтатами сравнения для всех  
неперекрывающижя подстрок, удовлетворяющих шаблону.

sub(repl, s)

Заменяет в строке s все (или толью **count,** если он задан) вхождения  
неперекрывающижя подстрок, удовлетворяющих шаблону, на строку,  
заданную с помощью **гері.** В качестве **гері** может выступать строка  
или функция. Возвращает строку с выполненными заменами. В первом  
случае строка **гері** подставляется не просто так, а интерпретируется с  
заменой вхождений **"\номер"** на группу с соответствующим номером  
и вхождений **"\д<имя>"** на грзппу с номером или именем имя. В  
случае, югда **гері** - функция, ей передается объект с результатом  
каждого успешного сопоставления, а из нее возвращается строка для  
замены.

замены.

subn(repl, s)

Аналогачен **sub ()**, но возвращает юртеж из строки с выполненными  
заменами и числа замен.

В следующем примере строка разбивается на подстроки по заданному  
шаблону:

»> import ге

»> delim\_re = re.compile(r"[

»> text = ’This,is;example"

»> print delim\_re.split(text)

['This', 'is', 'example']

A теперь можно узнать, чем именно были разбиты строки:

»> delim\_re = re.compile(r"([:,;])")

»> print delim\_re.split(text)

['This',',', **'is','exanple']**

Примеры шаблонов

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

**г'^b\w+\b"**

Соответствует слову из букв и знаков подчеркивания.  
r"[+-]?\d+"

Соответствует целому числу. Возможно, со знаком.  
r'X[+-]?\d+\)"

Число, стоящее в скобках. Скобки использзлотся в самих регулярных

**Сузи Р.А.**

г"[а-сА-С]{2}"

Соответствует строке из двух букв **"а", "Ь" или "с".** Например,  
**"Ас", "СС", "Ьс".**

г"аа|ЬЬ|сс|АА|ВВ|СС"

Строка из двух одинаковых букв.

г"([а-сА-С])\1"

Строка из двух одинаковых **63**^, но шаблон задан с использованием  
групп

г"аа|ЬЬ".

Соответствует **"аа" или "ЬЬ"**г"а(а|Ь)Ь"

Соответствует **"ааЬ" или "abb"**r"A(?:\d{8}|\d{4}):\s\*(.\*)$"

Соответствует строке, юторая 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 грзппу не выделен.

r"(\w+)=.\*\b\l\b"

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

r"(?P<var>\w+)=.\*\b(?P=var)\b"

То же самое, но теперь используется именованная группа **ѵаг.**r^bregular(?=\s+exp cession)".

Соответствует слову **"regular”** только в том случае, если за ним

**Сузи RA. Язык программирования Python**

Соответствует слову **"regular”** толью в том случае, если за ним  
после пробелов следует **"expression"**

r"(?<=regular )expression"

Соответствует слову **"expression",** перед юторым 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 регзлярное  
выражение, при условии, что есть пример строки, юторой шаблон  
должен здовлетворять. Взят кусочек лога iptables, его необходимо  
разобрать для получения полей. Интересны строки, в юторых после  
**kernel:** стоит PAY:, а в этих строках нужно получить дату, значения  
DST, LEN и DPT:

import re

def debug\_regex(regex, example):

"""Отладка per. выражения. Перед отладюй лучше убрать лишние сюбі  
last\_good = ""  
for i in range(l, len(regex)):  
try:

if re.compiIe(regex[ :i]).match(example):  
last\_good = regexfi]  
except:  
continue  
return last\_good

example = '""TNov 27 15:57:59 lap kernel: PAY: lN=ethO OUT=  
MAC=00:50:da:d9:df:a2:00:00:lc:b0:c9:db:08i)0SRC=192.168.1.200 DST=1  
LEN=1500 TOS=OxOO PREC=0x00 TIL=64 ID=31324 DF PROTO=TCP S  
WINDOW=17520 RES=0x00 ACK PSH URGP=0

**Сузи RA. Язык программирования Python**

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+ kem  
PAY:.+ DST=(?P<dst>\S+) \* LEN=(?P<len>\d+).\* DPT=(?P<dpt>\d+)""")

for line in openCmessage.log"):  
m = log\_re.match(line)  
if nr

print "%(date)s %(dst)s:%(dpt)s size=%(len)s" % m.groupdict()

В результате пол}ліается

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

**Сузи Р.А.**

Nov 27 15:57:59 192.168.1.115:1039 size=40

Анализ записи числа

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

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

* Следующие части числового литерала выделяются с помощью грзлш:
* \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 'е' или ’Е)

Например:  
import re

decoder = re.compne(r'A([-+]?)0\*(\d\*)((?:\.\d\*)?)((?:[eE][-+]?\d+)?)$')

print decoder.match("12.234").groups()  
print decoder.match("-0.23e-7").groups()  
print decoder.match("lelO").groups()

Получим

(", '12', '.234', ")

('-', ", '.23', 'e-7')

(", T, ", 'elO')

Множественная замена  
**в некоторых приложениях требуется производить в тексте сразу  
несколью замен. Для решения этой задачи можно использовать метод  
sub О вместе со специальной функцией, которая и будет управлять**

def multisub(subs\_dict, text):  
def\_multisub(match\_obj):

return str(subs\_dict[match\_obj .groupO ])

multisub\_re = re.compile("|".join(subs\_dict.keys()))  
return multisub\_re.sub(\_multisub, text)

repLdict = {'one': 1, 'two': 2, 'three': 3}  
print multisub(repl\_dict, "One, two, three")

Будет выведено  
One, 2, 3

В качестве упражнения предлагается сделать версию, юторая 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 несюлькими файлами можно использовать  
модзиь **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

**Сузи Р.А.**

Работа с Unicode

До появления Unicode символы в юмпьютере юдировались одним  
байтом (а то и толью семью битами). Один байт охватывает диапазон  
юдов от О до 255 включительно, а это значит, что больше двух  
алфавитов, цифр, знаюв пунктуации и неюторого набора специальных  
символов в одном байте не помещается. Каждый производитель  
использовал свою юдирові^^ для одного и того же алфавита. Например,  
до настоящего времени дожили целых пять юдировок букв кириллицы,  
и каждый пользователь не раз видел в своем браузере или электронном  
письме пример несоответствия юдировок.

Стандарт Unicode - единая юдировка для символов всех языюв мира.  
Это большое облегчение и неюторое неудобство одновременно. Плюс  
состоит в том, что в одной Unicode-строке помещаются символы  
совершенно различных языюв. Минус же в том, что пользователи  
привыкли применять однобайтовые юдировки, большинство  
приложений ориентировано на них, во многих системах поддержка  
Unicode осуществляется лишь частично, так как требует огромной  
работы по разработке шрифтов. Правда, символы одной юдировки  
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 юдировок  
Unicode (utf-7, utf-8, utf-16,...). В неюторых из этих юдировок имеет  
значение принятый на данной платформе порядок байтов (big-endian,  
старшие разряды в юнце или little-endian, младшие в юнце). Узнать  
порядок байтов можно, прочитав атрибут из модуля **sys.** На платформе  
Intel это выглядит так:

»> sys.byteorder

'litde'

Для исключения неоднозначности документ в Unicode может быть в  
самом начале снабжен ВОМ (Ьзде-order mark - метка порядка байтов) -  
Unicode-символом с юдом Oxfeff. Для данной платформы строка байтов  
для ВОМ будет таюй:

**Сузи RA. Язык программирования Python**

Для преобразования строки в Unicode необходимо знать, в какой  
юдировке заюдирован текст. Предположим, что это **ср1251.** Тогда  
преобразовать текст в Unicode можно следующим способом:

»> **S** = "Строка в ср1251"

»> s.decode("cpl251")

u'\u0421\u0442\u0440\u043e\u043a\u0430 \u0432 ср1251'

То же самое с помощью встроенной функции **Unicode** ():

»> unicode(s, 'ср1251')

u'\u0421\u0442\u0440\u043e\u043a\u0430 \u0432 ср1251'

Одной из полезных функций этого модзия является функция  
**codecs.** **open** **(),** позволяющая открыть файл в другой юдировке:

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

Здесь:

filename

Имя файла.

mode

Режим открытия файла  
епс

Кодировка.

errors

Режим реагирования на ошибки юдировки ('strict' - возбуждать  
исключение, 'replace' - заменять отсутствзтощие символы, 'ignore' -  
игнорировать ошибки).

buffer

Режим буферизации (О - без буферизации, 1 - построчно, п - байт

**Сузи Р.А.**

buffer

**- без буферизации, 1 - построчно, п - байт**

Режим буферизации (О  
буфера).

Заключение

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

Ссылки

NLTK

ссылка: httpy/nltk.sourceforge.net

Работа с данными в различных форматах

Работа с современными форматами данных - одно из сильных мест  
стандартной библиотеки Pythoa 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 юличество модулей для всевозможных  
форматов данных.

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 рг . cs ѵ):

name,number,text

1. 1, something here
2. 2,"one, two, three"
3. 3,"no commas here"

Для работы с CSV-файлами имеются две основные функции:

reader(csvfile[, dialect='excel'[, frntparam]])

Возвращает читающий объект, юторый является итератором по всем  
строкам заданного файла. В качестве **csvf** **ііе** может выступать любой  
объект, который поддерживает протокол итератора и возвранрет 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 помогцью **csv. list\_dialects** **().**Аргумент может быть одной из строк, возврагцаемых зжазанной  
функцией, либо экземпляром подкласса класса **csv. Dialect.**Необязательный аргумент **frntparam** служит для переназначения

**Сузи RA. Язык программирования Python**

отдельных свойств по сравнению с заданным параметром **dialect**набором. Все получаемые данные являются строками.

writer(csvfile[, dialect='excel'[, frntparam]])

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

В следзлощем 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("prl.csv", 'Vb")  
wrtr = csv.writer(output\_file)  
for rec in rdr:  
try:

rec[l] = int(rec[l]) + 1  
except:  
pass

wrtr.writerow(rec)

input\_file.close()

output\_file.close()

В результате пол}діится файл prl.csv следующего содержания:

name,number,text

1. 2,something here
2. 3,"one, two, three"
3. 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='e?

**Сузи RA. Язык программирования Python**

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 **rest** **key** 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.

class DictWriter(csvfile, fieldnames[, restval=""[, extrasaction='raise'[, dialect='ex

Создает пинущий объект, юторый записывает в CSV-файл строки,  
полздіая данные из словаря. Параметры аналогачны **DictReader,** но  
**fieldnames** обязателен, так как он задает порядок следования полей.  
Параметр **extrasaction** указывает на то, каюе действие нужно  
произвести в случае, когда требуемого значения нет в словаре:  
**'raise'** - возбудить исключение **ValueError, 'ignore' -**игнорировать.

Соответствзлощий пример дан ниже. В файле рг. 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("prl.csv", 'Vb")  
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()

**Сузи Р.А.**

output\_file.close()

Модуль имеет также другие классы и функции, юторые можно изучить  
по документации. На примере этого модуля можно з^идеть общий  
подход к работе с файлом в нештором 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 несколью вариантов  
  классов читающих и пиіщлцих объектов. Новые классы могут  
  получаться наследованием от базовых классов либо  
  обертыванием фзлікций, предоставляемых модулем расширения  
  (написанным на С). 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 несюлько модулей, назначение юторых (кратю) зл^азано  
ниже:

Message

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

Parser

Модуль для разбора представленного в виде текста сообщения с  
полздіением объектной структуры сообщения.

**Сузи Р.А.**

Header

Модуль для работы с полями, в юторых используется юдировка,  
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, юторые здесь рассматриваться не будут.

Разбор сообщения. Класс Message

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

Следует заметить, что сообщение может иметь одну или несюлько  
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("prl.eml')

»> msg = emaimessage\_from\_file(input\_file)

**Сузи RA. Язык программирования Python**

Здесь используется функция **email. message\_from\_f** **ііе** **()** для  
чтения сообщения из файла prl.emL Сообщение можно получить и из  
строки с помощью функции **email. message\_f rom\_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']

"fehon olive" <[zinakinch@thecanadianteacher.com](mailto:zinakinch@thecanadianteacher.com)>

»> msg.get\_all('received')

['from maiLonego.m\n\tby localhost with POPS (fetchmail-6.2.5  
polling maiLonego.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 maiLonego.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 несколью полей  
с именем **received** (в этом примере их два).

Неюторые важные данные можно получить в готовом виде, например,  
тип содержимого, юдировку:

»> 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'j

**Сузи RA. Язык программирования Python**

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 currendy you are still be the best amony

Теперь будет рассмотрен другой пример, в ютором сообщение состоит  
из нескольких частей. Это сообщение порождено вирусом. Оно состоит  
из двз^ частей: HTML-текста и вложенного файла с расширением cpL  
Для достзпа 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\_**6**elds = []  
filenames = []  
f = open("virus.eml")  
msg = emaiLmessage\_fi-om\_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 пол}ліилось:

Длина файла: 31173

['multipart/mixed', 'text/html', 'application/octet-stream']

**Сузи RA. Язык программирования Python**

['multipart/mixed;\n boundaiy=" hidejpxkblmvuw^lzue'",

'text/html; charset="us-ascu'”,

'appUcation/octet-stream; name="price.cpl'"]

[None, None, 'price.срГ]

Из списка **parts** можно увидеть, что само сообщение имеет тип  
**multipart/mixed,** тогда как две его части - **text/html** и  
**application/octet-St ream** соответственно. Только с последней  
частью связано имя файла (price.срі). Файл читается методом  
**get\_payload** () и вычисляется его длина.

Кстати, в случае, югда 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** модзля **email .Message,  
но и MIMEMultipart из email.MIMEMultipart** (для сообщений  
из несюльких частей), **MIME Image** (для сообщения с грасЩческим  
изображением), **MIMEAudio** (для аудиофайлов), **MIMEText** (для  
текстовых частей):

* Загружаются необходимые модули и фзлікции из модулей  
  from email Header import make\_header as mkh

from emailMIMEMultipart import MIMEMultipart  
from emailMIMEText import MIMEText  
from emailMIMEBase import MlMEBase  
from emailEncoders import encode\_base64

* Создается главное сообщение и задаются некоторые поля  
  msg = MIMEMultipartO

msg["Subject"] = ткЬ([("Привет", "коі**8**-г")])

msg['Erom"] = ткЬ([("Друг", "коі**8**-г'), ("<friend(2)maiLru>", "us-ascn')])  
msgf'To"] = ткЬ([("Друг2", "коі**8**-г'), ("<[friend2@yandex.ru](mailto:friend2@yandex.ru)>", "us-ascn')])

* То, чего буцет не видно, если почтовая программа поддерживает MI1V  
  msg.preamble = "Multipart message"

msg.epilogue =""

* Текстовая часть сообщения

text = u"""K письму приложен файл с архивом.""".епсос1е("коі8-г")  
to\_attach = MIMEText(text, \_charset="koi**8**-r")  
msg.attach(to\_attach)

* Прикладывается файл

fp = open("archive\_file.zip", "rb")

to\_attach = MIMEBase("application", "octet-stream")

to\_attach.set\_payload(f]**3**.read())

encode\_base64(to\_attach)

to\_attach.add\_header("Content-Disposition", "attachment",  
filename="archive\_file.zip")

fp.cfoseQ

msg.attach(to\_attach)  
print msg.as\_string()

В этом примере видно сразу несюлько модзлей пакета **email.** Функция  
**make\_header** () из **email. Header** позволяет заюдировать  
содержимое для заголовка:

»> from emaiLHeader import make\_header

»> print make\_header([('^pyr", "koi**8**-r"), ("<friend@maiLm>", "us-ascii")])  
=?koi8-r?b?5NLVxw==?= <friend@maiLru>

»> print make\_header([(u'^p**5**T", ""), ("<friend(**2**)mairu>", "us-ascn")])  
=?utf-8?b?w6TDksOVw4c=?= <friend@maiLru>

Функция **email. Encoders . encode\_base64** **()** воздействует на  
переданное ей сообщение и юдирует тело с помощью base64. Другие  
варианты: **encode\_quopri** **()** - юдировать quoted printable,

**encode\_7or8bit** **()** - оставить семь или восемь бит. Эти функции  
добавляют необходимые поля.

Аргументы констрзжторов классов из МІМЕ-модулей пакета **email:**

**Сузи Р.А.**

class MIMEBase(\_maintype, \_subtype, \*\*\_params)

Базовый класс для всех использующих MIME сообщений (подклассов  
**Message** ). Тип содержимого задается через **\_maintype** и  
**\_subtype.**

class MIMENonMukipartO

Подкласс для **MIMEBase,** в ютором запрещен метод **attach** (), отчего  
он гарантированно состоит из одной части.

class MIMEMultipart([\_subtype[, boundary[, \_subparts[, \_params]]]])

Подкласс для **MIMEBase,** юторый является базовым для МІМЕ-  
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 МІМЕ-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 ' rf **с822** **'.**

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

**Сузи RA. Язык программирования Python**

Подкласс **MIMENonMultipart.** Используется для создания МІМЕ-  
сообщений текстового типа. Главный тип - **text,** подтип указывается с  
помощью **\_subtype.** Данные задаются параметром **\_text.**Посредством **\_charset** можно указать юдирові^^ (по з^олчанию  
**' 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 юдировка, в юторой 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=C

=?koi8-r?Q?=CE=CE=CF=C7=CF\_=28164\_bytes=29\_=D0=CF=CC=Dl\_=

=?koi8-r?Q?=C5=CD=CF=CA\_=D3=CF=CF=C2=DD=C5=CE=C9=Dl=2

=?koi8-r?Q?=D2=Cl=DA=C2=C9=CC=CF=D3=D8\_=CE=Cl\_=CB=D5=]

=?koi8-r?Q?\_=D3=CF=CF=C2=DD=C5=CE=C9=C9=2C\_=CE=CF\_=CC:

=?koi8-r?Q?\_=D3=CF=C2=C9=D2=Cl=C5=D4=D3=Dl\_=D7\_=D4=C5=(

=?koi8-r?Q?=D3\_=D0=CF=CD=CF=DD=D8=C0\_emaU=2EHeader=2Edec(

=?koi8-r?Q?=28=29?="'"'

import emailHeader

for text, enc in emaiLHeader.decode\_header(subj):  
print enc, text

В результате будет выведено:

коі**8**-г Это пример очень длинного (164 bytes) поля с темой сообщения.  
Оно разбилось на куски в сообщении, но легю собирается в текст  
с помощью emaiLHeader.decode\_header()

Следует заметить, что юдировку можно не зжазывать:

»> emaiHeader.decode\_header("simple text")

[('simple text'. None)]

**»> emaiHeader.decode\_header("npHMep")**

**Сузи RA. Язык программирования Python**

[('\xd0\xd2\xc9\xcd\xc5\xd2', None)]

>» emaiHeader.decode\_header("=?KOI8-R?Q?=DO=D2=CF\_?=Linux")  
[('\xd0\xd2\xcf'коі**8**-г'), ('Linux', None)]

Если в первом случае можно подразумевать us-ascii, то во втором случае  
о юдировке придется догадываться: вот почему в электронных письмах  
нельзя просто так использовать восьмибитные юдировки. 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 (Extensble 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 дрзтое.

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

**Сузи RA. Язык программирования Python**

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 Туре  
Definition). Объявление типа документа задает грамматику, юторой 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. Однаю это обстоятельство необходимо помнить и не  
употреблять в XML восьмибитную юдировід^ (во всяком случае, без  
явного указания).

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

<?xml version="1.0" encoding="iso-8859-l'?>

<expression>

<operation type="+">

<operand>**2**</operand>

<operand>

<operation type="\*">

<operand>3</operand>

<operand>4</operand>

</operation>

</operand>

</operation>

</expression>

XML-документ всегда имеет структзфу дерева, в корне юторого 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. Полз^іается, что документ имеет древеснзто структзфу.

Примечание:

**Сузи RA. Язык программирования Python**

Следует заметить, что в отличие от HTML, в XML одиночные  
(непарные) теги записываются с юсой чертой: **<BR/>,** а атрибуты - в  
кавычках. В XML имеет значение регистр **63**^ 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 источниюв 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  
конкретнзло 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 xmLsax.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")

**Сузи RA. Язык программирования Python**

g.startElement("operand", {})  
g.startElement("operation", {"Із^е": "\*"})  
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 xmLdom import minidom

dom = minidomDocumentO

el = domcreateElement("expression")

domappendChild(el)

pi = domcreateElement("operation")

pl.setAttribute('type','+')

xl = domcreateElement("operand")

xl .appendChild(domcreateTextNode("2"))

pl.appendCMd(xl)

el.appendChild(pl)

p2 = domcreateElement("operation")

p2.setAttribute('type', '\*')

x2 = domcreateElement("operand")

x2.appendChild(domcreateTextNode("3"))

p2.appendCMd(x2)

x3 = domcreateElement("operand")

x3.appendChild(domcreateTextNode("4"))

p2.appendCMd(x3)

x4 = domcreateElement("operand")

x4.appendChild(p2)

pl.appendCMd(x4)

print doratoprettyxml()

**Сузи RA. Язык программирования Python**

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

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 О . 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 xmldomminidom

dom = xmLdom.minklom.parse("expressk)n.xml")

dom.normalize()

def output\_tree(node, level=0):  
if node.nodeType == node.TEXT\_NODE:  
if node.nodeValue.strip():  
print". "\*level, node.nodeValue.stripO  
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 cliild in node.childNodes:  
output\_tree(cMd, level+1)

output\_tree(dom)

**Сузи RA. Язык программирования Python**

в этом примере дерево выводится с помощью определенной функции  
**output\_tree** (), юторая принимает на вжде узел и вызывается  
рекурсивно для всех вложенных узлов.

В результате пол}ліается примерно следующее:

#docLunent  
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 несюлько 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. node Name** дает имя узла (соответствует названию тега),  
**node. attributes** дает доступ к атрибутам узла,  
**node . chi IdNodes** применяется для доступа к дочерним узлам. Этих  
свойств достаточно, чтобы рекурсивно обойти дерево.

Все звлы являются экземплярами подклассов класса **Node.** Они могут  
быть следующих типов:

**Метод для со:  
createElement(tagnan  
createAttribute(name**

**createTextN ode(data**

**Описание**

**Элемент**

**Атрибут**

**Текстовый**

**узел**

Название

ELEMENT\_NODE

ArnUBUTE\_NODE

TEXT NODE

**Раздел**

**CDAIA**

**Сузи Р.А.**

CDAIA\_SECTION\_NODE

ENTITY\_REFERENCE\_NODE

ENTITY\_NODE

PROCESSING\_INSTRUCTION\_NODE

COMMENT\_NODE

DOCUMENT\_NODE

DOCUMENT\_TYPE\_NODE

DOCUMENT\_FRAGMENT\_NODE  
NOWION NODE

Ссылка на  
сущность

**Сзтідіость**

Инструкция createProcessingInstr  
по обработке data)

Комментарий createComment(com!

Документ

Тип

документа

Фрагмент

документа

Нотация

В DOM документ является деревом, в узлах юторого стоят объекты  
нескольких возможных типов. Узлы могут иметь атрибуты или данные.  
Доступ к узлам можно осуществлять через атрибуты вроде  
**childNodes** (дочерние узлы), **Tirs t Chi id** (первый дочерний узел),  
**lastChild** (последний дочерний }вел), **parentNode** (родитель),  
**nextSibling** (следзаощий брат), **previousSibling** (предыдущий  
брат).

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

Узел типа **ELEMENT\_NODE,** ПОМИМО перечисленных методов "просто"  
узла, имеет много дрзтих методов. Вот основные из них:

tagName

**Сузи RA. Язык программирования Python**

Имя типа элемента.  
getElementsByTagN ame(tagname)

Получает элементы с указанным именем tagname среди всех потомков  
данного элемента.

getAttribute(attname)

Получить значение атрибута с именем **attname.**getAttributeN ode(attrname)

Возвращает атрибут с именем **attrname** в виде объекта-}вла.

removeAttribute(attname)

>^^алить атрибут с именем **attname.**removeAttributeN ode(oklAttr)

>^^алить атрибут **oldAttr** (задан в виде объекта-узла).  
setAttribute(attname, value)

Устанавливает значение атрибута **attname** равным строке **value.**setAttributeN ode(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, юторая будет  
вычислять значение выражения, заданного в XML-представлении.

Пространства имен

**Сузи RA. Язык программирования Python**

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

Следующий пример XML-шда показывает синтаксис пространств имен  
(файл foaf.rdf);

<?хпі1 version="1.0" encoding='XJTF-8'?>

<rdf:RDF

xmlns :dc=' "http У/http У/purL org/dc/elements/1.1/"  
xmlnsTd£s="httpy/www.w3.org^2000/01/rdf-schema#"  
xmlns :foaf= "http y/xmlns. com/foaFO. 1/"  
xmlnsTdf="httpy/[www.w3.org/1999/02/22-rdf-S3Titax-ns#](http://www.w3.org/1999/02/22-rdf-S3Titax-ns%23)"

**>**

<rdf:Descriptionrdf:nodeID="\_:jCBxPzi01">

<foafTiick>donna</foafTiick>

<foafTiame>Donna Fales</foaf:name>

<rdfttype rdf:resource="httpy/xmlns.com/foaFO.l/Person"/>

</rdf:Description>

</rdf:RDF>

Примечание:

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

**Сузи RA. Язык программирования Python**

Названия пространств имен следуют в виде префиксов к названиям  
элементов. Эти названия - не просто имена. Они соответствзтот  
идентификаторам, юторые должны быть заданы в виде URI (Universal  
Resource Identifier, универсальный указатель ресурса). В примере выше  
упоминаются пять пространств имен (xmlns, dc, rdfe, foaf и rdf), из  
юторых толью первое не требует объявления, так как является  
встроенным. Из них реально использованы толью три: (xmlns, foaf и rdf).

Пространства имен позволяют выделять из XML-документа части,  
относящиеся к различным схемам, что важно для тех инстрз^ентов,  
юторые интерпретируют XML.

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

Получить UR1, юторый соответствует пространству имен данного  
элемента, можно с помощью атрибута **namespaceURI.**

В следующем примере печатается URI элементов:

import xmLdomminidom

dom = xmLdomminidom.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:

rdfiRDF httpy/www.w3.org^l999/02/22-rdf-syntax-ns#  
rdftDescription http У/www. w3. org^l999/02/22- rdf- syntax- ns#  
foafmick httpy/xmlns.com/foafiO.l/  
foafTiame httpy/xmlns.com/foafiO.l/  
rdfttype httpy/www.w3.org^l999/02/22-rdf-syntax-ns#

Следует заметить, что указание пространства имен может быть сделано

**Сузи Р.А.**

ДЛЯ имен не толью элементов, но и атрибутов.

Подробнее узнать о работе с пространствами имен в хті-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, однаш то, что было представлено, поможет быстрее  
разобраться с любым модулем для обработки формата или построить  
свой модуль так, чтобы другие могли понять ваши намерения.

**Сузи Р.А.**

Разработка Web-приложений

Одна из главных сфер применения языка РзДЬоп - web-приложения -  
представляется в этой лекции на юнкретных примерах. Кроме того,  
делается акцент на типичных слабых местах безопасности web-  
приложений.

Под web-приложением буцет пониматься программа, основной  
интерфейс пользователя которой работает в стандартном WWW-  
браузере под зправлением HTML и XML-документов. Для улучшения  
качества интерфейса пользователя часто применяют JavaScript, однако  
это несколько снижает универсальность интерфейса. Следует заметить,  
что интерфейс можно построить на Java- или Flash-апплетах, однаю,  
такие приложения сложно назвать web-приложениями, так как Java или  
Flash могут использовать собственные протоюлы 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 близю 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 слз^іае с WWW,  
web-сервер может направить запрос на генерацию страницы по  
определенному сценарию. Этот сценарий, полз^ив на вход данные от  
web-сервера (тот, в свою очередь, мог полз^ить их от пользователя),  
генерирует готовый объект (изображение, аудиоданные, таблицу стилей  
и **Т.П.).**

При вызове сценария Web-сервер передает ему информацию через

**Сузи RA. Язык программирования Python**

стандартный ввод, переменные окружения и, для ISINDEX, через  
аргументы юмандной строки (они доступны через sys . **argv** ).

Два основных метода передачи данных из заполненной в бразвере  
формы Web-серверу (и CGI-сценарию) - **GET** и **POST.** В зависимости от  
метода данные передаются по-разному. В первом случае они  
юдируются и помещаются прямо в URL, например:

**http: //example . сот/cgi-bin/a. cgi?a=l&b=3.** Сценарий  
полздіает их в переменной окружения с именем **QUERY\_STRING.** В  
случае метода **POST** они передаются на стандартный ввод.

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

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

#!/usr/bin/python

import os

print """Content-Тзфе: text/plain  
%s % os.environ

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

Основные переменные окружения, достаточные для создания  
сценариев:

QUERY\_STRING

Строка запроса.

**Сузи RA. Язык программирования Python**

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 браузера клиента.

REQUEST\_URI  
Строка запроса (URI).

HTTP\_ACCEPT\_LANGUAGE  
Желательный язык доі^ента.

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

**{**

'DOCUMENT\_ROOT: Vvar/www/html',

'SERVER\_ADDR': '127.0.0.1',

'SERVER\_PORT:'80',

'GATEWAYJNTERFACE': 'CGI/1.1',

'HTTP\_ACCEPT\_LANGUAGE': 'en-us, en;q=0.50',

'REMOTE\_ADDR': '127.0.0.1',

**Сузи RA. Язык программирования Python**

'SERVER\_NAME': 'rnd.onego.ru',

'HTTP\_CONNECTION': 'close',

'HTTP\_USER\_AGENT: 'Mozilla/5.0 **(Xll;** U; Linux Б86; en-US;  
rv:1.0.1) Gecko/20021003',

'HTTP\_ACCEPT\_CHARSET: 'ISO-8859-1, utf-8;q=0.66, \*;q=0.66',

**'H11P\_** 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=l',

'PATH': '/sbinyusr/sbinybinyusr/binyusr/XllR6/bin',

'QUERY\_STRING': 'a=l&b=2',

' SC RIPT\_FILEN AME': '/var/www/cgi-bin/test.py',

'HTTP\_KEEP\_ALIVE': '300',

'HTTP\_HOST: 'localhost',

'REQUEST\_METHOD': 'GET,

'SERVER\_SIGNArURE': 'Apache/1.3.23 Server at rnd.onego.ru Port 80',  
'SCRIPT\_NAME': '/cgi-bin/testpy,

'SERVER\_ADMIN': 'root@localhost',

'SERVER\_SOFTWARE': 'Apache/1.3.23 (Unix) (Red-Hat/Linux)  
niod\_python/2.7.8 Python/1.5.2 PHP/4.1.2',

'SERVER\_PROTOCOL': 'НТГР/1.0',

'REMOTE\_PORT: '39251'

**}**

Следующий ССІ-сценарий выдает черный квадрат (в нем используется  
модзиь **Image** для обработки изображений):

#!/usr/bin/python

import sys

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

i = Image.new('RGB", (10,10))  
Lmdraw\_rectangle((0,0,10,10), 1)  
Lsave(sys.stdout, "jpeg")

**Сузи Р.А.**

Модуль cgi  
**в Python имеется поддержка CGI в виде модзия cgi. Следующий  
пример показывает неюторые из его возможностей:**

#!/usr/bin/python

* coding: ср1251  
  import cgi, os
* анализ запроса

f = cgLFieldStorageQ  
if f.has\_key("a"):  
a = fl"a"].value  
else:  
a = "0"

* обработка запроса  
  **b** = str(int(a)+l)

mytext = open(os.environ["SCRIPT\_FILENAME"]).read()  
mytext\_html = cgLescape(mytext)

* формирование ответа  
  print """Content-Тзфе: text/html

**<html><head><tide>PeiueHHe примера: %(b)s = %(a)s + l</tide></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" ѵа1ие="Обработать">

</form> </td> </tr> </table>

<pre>

%(mytext\_html)s

</pre>

**</bodyx/html> % vars()**

В этом примере к заданному в форме числу прибавляется 1. Кроме того,

**Сузи RA. Язык программирования Python**

выводится исходный юд самого сценария. Следует заметить, что для  
экранирования символов >, <, & использована функция

**сді. escape** **О.** 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  
полздіения 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 толью то, что ожидает сценарий.

Например, не следует подставлять полученные от пользователя данные  
в путь к файлу, в качестве аргументов к фзлікции **еѵаі** **()** и ей  
подобных; параметров юмандной строки; частей в SQL-запросах к базе  
данных. Также не стоит вставлять ползліенные данные напрямую в  
формируемые страницы, если эти страницы будет видеть не толью  
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-юдом, постзпившим  
напрямую от пользователя, рискуют обработать в своем браузере  
зловредный юд, использующий брешь в его защите.

Даже если CGI-сценарий используется исключительно другими  
сценариями через запрос на URL, нужно проверять вхэдные 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 О** : 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, наюнец, фаза вывода готового объекта (текста, HTML-доід^ента,  
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. Неюторые формы порождают несколью значений для одного  
имени. Получить все значения можно с помощью метода **get list ():**

1st = form.getlist("fld")

Список 1st будет содержать столью значений, сколью полей с именем  
**fid** полздіено 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 = cgLFieldStorageO  
file\_contents =""  
if form.has\_key("filename'):  
fileitem= form[ "filename"]  
if fileitemfile:

**Сузи RA. Язык программирования Python**

fiIe\_contents = """<Р> С одержи мое переданного файла:  
<PRE>%s</PRE>' % fileitemfile.readO

print """Content-Тзфе: text/html

<HTML><HEAD><TITLE>3arpy3Ka файла</ПІЪЕ></НЕАВ>  
<ВООѴ><Н1>Загрузка файла</Н1>

<P><FOElM ENCTYPE="multipart/form-data"

ACTION="getfiIe.cgi" METHOD=TOST'>

<Ьг>Файл: <1NPUT TYPE="file" NAME="filenanie">

<br><INPUT TYPE="submit" NAME='button" VALUE='Передать файл">  
</FORM>

%s

</BODY></HTML>' % file\_contents

В начале следует рассмотреть web-форму, юторая приведена в конце  
сценария: именно она будет выводиться пользователю при обращении  
по CGI-сценарию. Форма имеет поле типа **file,** юторое в web-  
браузере 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(l)

Или, если не нужно показывать ошибки в браузере:  
import cgitb

cgitb.enable(0, logdir="/tmp")

Только необходимо помнить, что следует убрать эти строки, югда  
сценарий будет отлажен, так как он выдает ід^сочки юда сценария. Это  
может быть использовано злоумышленниками, с тем чтобы найти  
уязвимости в CGI-сценарии или подсмотреть пароли (если таювые

**Сузи Р.А.**

присугствзтот 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 РздЬоп 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 существзаот 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 специальный модуль для web-сервера Apache,  
называемый **mod\_python.**

Пусть модзль установлен на web-сервере в соответствии с  
инструкциями, данными в его документации.

Модуль **mod\_python** позволяет сценарию-обработчику вклиниваться  
в процесс обработки Ні іР-запроса сервером Apache на любом этапе,  
для чего сценарий должен иметь определенным образом названные  
функции.

Сначала нужно выделить каталог, в ютором 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 кон4»ігурации Apache  
следующие строки:

**Сузи RA. Язык программирования Python**

<Directoiy "/var/www/html/m**3**webdir">

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.ру.** Этот сценарий будет реагировать на любой запрос  
вида httpy/locaJhost/\*.py.

Следующий сценарий **mprocess** **.ру** 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.writeC <HTML><HEAD><TITLE>Heno, world!<тТЕЕ></НЕАВ>

<BODY>HeUo, world!</BODYx/H™L> **')**

return apache.OK

Отличия сценария-обработчика от CGI-сценария:

1. Сценарий-обработчик не запускается при каждом НТТР-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 юпию сценария и интерпретатора  
   Python.
2. Как следствие п.1 различные НТТР-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 ру, тогда как CGI-сценарий обычно запускается при  
   обрагцении по конкретному имени.

**Сузи RA. Язык программирования Python**

1. В сценарии-обработчике нельзя рассчитывать на то, что он  
   увидит модули, расположенные в том же каталоге. Возможно,  
   придется добавить некоторые каталоги в sys . **path.**
2. Текущий рабочий каталог (его можно узнать с помощью функции  
   **os.getcwdO** ) также не наждится в одном каталоге с  
   обработчиюм.
3. #!-строка в первой строке сценария не определяет версию  
   интерпретатора Python. Работает версия, для шторой был  
   сшмпилирован **mod\_python.**
4. Все необждимые параметры передаются в обработчик в виде  
   Request-объекта. Возвращаемые значения также передаются через  
   этот объект.
5. Web-сервер замечает, что сценарий-обработчик изменился, но не  
   заметит изменений в импортируемых в него модулях. Команда  
   **touch mprocess** **.ру** обновит дату изменения файла сценария.
6. 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 пол}діить другим  
путем: **req. add\_common\_vars () ; params =**

**req.subprocess\_env.**

1. Так как сценарий-обработчик не является "одноразовым", как CGI-  
   сценарий, из-за ошибок программирования (как самого сценария,  
   так и других шмпонентов) могут возникать утечки памяти  
   (программа не освобождает ставную ненужной память). Следует  
   установить значение параметра **MaxRequestsPerChild**(максимальное число запросов, обрабатываемое одним  
   процессом-потомюм) больше нуля.

Другой возможный обработчик - сценарий идентификации:

def authenhandler(req):  
password = req.get\_basic\_auth\_pw()  
user = req. connection, user  
if user == "userl" and password == "secret":  
return apache.OK  
else:

return apache. H ГI P\_UN AUTHO RIZED

**Сузи RA. Язык программирования Python**

Эту функцию следует добавить в модуль **mprocess.py,** юторый был  
рассмотрен ранее. Кроме того, нужно дополнить кон4мгурацию,  
назначив обработчик для запросов идентификации (  
**PythonAuthenHandler** ), а также обычные для Apache директивы  
**AuthType, AuthName, require,** определяющие способ  
авторизации:

<Directoiy "/var/www/html/m**3**webdir">

AddHandler python-program .py  
PythonHandler mprocess  
P**3**^honAuthenHandler mprocess  
AuthType Basic  
AuthName "My page"  
require valid-user  
</Directory>

Разумеется, это - всего лишь пример. В реальности иденти4мкация  
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 полз^іения.  
PythonTransHandler

Позволяет изменить URI запроса (в том числе имя виртуального  
сервера).

PythonHeaderParserHandler  
Обработка полей запроса.

PythonAccessHandler

Обработка ограничений доступа (например, по ІР-адресу).  
PythonAuthenHandler

**Сузи Р.А.**

Идентификация пользователя.

PythonTypeHandler

Определение и/или настройка типа документа, языка и т.д.  
PythonFixupHandler

Изменение полей непосредственно перед вызовом обработчиюв  
содержимого.

PythonHandler

Основной обработчик запроса.

PythonInitHandler

**PythonPostReadRequestHandler** или

**PythonHeaderParserHandler** в зависимости от нахождения в  
конс}мгурации ѵѵеЬ-сервера.

PythonLogHandler

Управление записью в логи.

PythonCleanupHandler

Обработчик, вызываемый непосредственно перед зтзичтожением  
Request-объекта.

Неюторые из этих обработчиков работают толью глобально, так как при  
вызове даже каталог их приложения может быть неизвестен (таюв,  
например, **PythonPostReadRequestHandler ).**

С помощью **mod\_python** можно строить web-сайты с динамическим  
содержимым и юнтролировать неюторые аспекты работы web-сервера  
Apache через РздЬоп-сценарии.

Среды разработки

**Сузи RA. Язык программирования Python**

Для создания ѴѴеЬ-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 юторых CMS-система является подсистемой. Поэтому,  
выбирая систему для юнкретных нужд, стоит уточнить, какие функции  
она должна выполнять.

Язык Python, хотя и уступает РНР по юличеству созданных на нем web-  
систем, имеет несюлько достаточно популярных приложений. Самым  
ярким примером средства для создания сервера web-приложений  
является Zope (произносится "зоп") (см. ссылка: httpy/zope.org) (Z Object  
Publishing Environment, среда публикации объектов ). Zope имеет  
встроенный web-сервер, но может работать и с дрзтими 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 Логні^^ до таюй степени, что Содержанием могут  
заниматься одни люди (менеджеры по содержимому). Формой - другие  
(web-дизайнеры), а Логиюй - третьи (программисты). В сл}ліае с Zope  
Логиід^ 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 толью в качестве среды разработки web-приложений, но  
и с точки зрения подходов. Например, уникальная объектно-  
ориентированная модель Zope позволяет довольно гибю описывать  
требуемое приложение.

**Сузи Р.А.**

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 несюлько языков  
  сценариев: Python, Perl и собственный DTML (D ocument Т emplate  
  М arkup L anguage, 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 юторой загрузку документа по URI можно сравнитъ с  
отправюй сообщения объекту. Объекты Zope разложены по папкам  
(folders), к которым привязаны политики достзліа для пользователей,  
имеющих определенные роли. В качестве объектов могут выступатъ  
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

<!- #ѵаг document tide ->

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

**Сузи RA. Язык программирования Python**

других источников данных (например, из базы данных посредством  
выполнения выборки функцией SELECT).

Сейчас для описания документа Zope все чаще применяется ZPT ( Z оре  
Р age Т emplates, шаблоны страниц Zope), которые в свою очередь  
используют TAL ( Т emplate А ttribute L anguage, 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 идентис}мкатором:

xmlnstal="httpy/xmLzope.org^namespaces/tal"

Оператор TAL имеет имя и значение (что выражается именем и  
значением атрибута). Внутри значения обычно записано TAL-  
выражение, синтаксис юторого описывается другим языком - TALES  
(Template Attribute Language Expression S**3**mtax, синтаксис выражений lAL).

Таким образом, ZPT наполняет содержимым шаблоны, интерпретируя  
атрибуты TAL. Например, чтобы Zope подставил название документа  
(тег **TITLE** ), шаблон может иметь следующий юд:

<title tal:content="here/title">Doc Title</title>

Стоит заметить, что приведенный юд сойдет за юд на HTML, то есть,  
Web-дизайнер может на любом этапе работы над проектом  
редактировать шаблон в HTML-редакторе (при условии, что тот  
сохраняет незнаюмые 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>

<Д tal;define="s modules/string"  
tal:repeat="el python:s.digits">

<a hre^'DUMMY"  
tal;attributes="href stringydigit/$el"  
tal;content="er'>SELECTION</a>

</li>

</ul>

Этот шаблон породит следзлощий реззльтат:

**<ul>**

**<li><a**

**<li><a**

**<li><a**

**<li><a**

**<li><a**

**<li><a**

**<li><a**

**<li><a**

**<li><a**

**<li><a**

**</ul>**

hre^"/digit/0">0</a></li>  
hre^ "/digit/1 "> 1 </a></li>  
hre^ "/digit/2 ">2 </a></li>  
hre^"/digit/3">3</a></li>  
hre^"/digit/4">4</a></li>  
hre^ "/digit/5 "> 5</a></li>  
hre^"/digit/6">6</a></li>  
hre^ "/digit/7"> 7</a></li>  
hre^"/digit/8">8</a></li>  
hre^"/digit/9">9</a></li>

Здесь нужно обратить внимание на два основных момента:

* в шаблоне можно использовать выражения Python (в данном  
  примере переменная s определена как модзль Python) и  
  переменную-счетчик цикла е1, которая проюдит итерации по  
  строке **string, digits.**
* с помощью TAL можно задавать не только содержимое элемента,  
  но и атрибута тега (в данном примере использовался атрибут

**href ).**

**Сузи RA. Язык программирования Python**

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 уцалось "расставить сюбки", выведя "общий множитель за  
скобки". Другими словами, хорошие программисты должны быть  
достаточно "ленивы", чтобы найти оптимальную декомпозицию  
решаемой задачи. При проектировании динамичесюго web-сайта Zope  
позволяет разместить "множители" и "сюбки" так, чтобы достигнуть  
максимального повторного использования юда (как разметки, так и  
сценариев). Помогает этому уникальный подход к построению  
взаимоотношений между объектами: заимствование (acquisition).

Пусть неюторый объект (доід^ент, 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 глубою вложенными  
папками от папок, юторые ближе к юрню. Заимствование является  
очень важной юнцепцией Zope, без понимания юторой Zope сложно  
грамотно применять, и наоборот, ее понимание позволяет эюномить  
силы и время, повторно используя объекты в разработке.

Самое интересное, что заимствовать объекты можно также из  
параллельных папок. Пусть, например, рядом с папюй Example  
находится папка **Zigzag,** в юторой лежит нужный объект (его  
наименование note). При этом в папке Example программиста  
интересует объект **index\_html,** внутри юторого вызывается note.  
Обычный путь к объекту **index\_html** будет происюдить по URI вроде  
**http: //zopeserver/Example/.** А вот если нужно использоватъ  
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 юнкретную страницу, юмбинируя один или  
несюлью независимых аспектов.

**Сузи Р.А.**

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 мод}лей для web-сервера, таких как  
mod\_python. Наконец, существзлот технологии вроде Zope, которые  
предоставляют специализированные сервисы, позволяющие создавать  
web-приложения.**

**Сузи Р.А.**

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  
несколью модзией для работы с различными протоюлами. Этими  
модзиями охватываются как низкозфовневые протоюлы (TCP/IP,  
UDP/IP), так и высокоуровневые (HTTP, FTP, SMTP, POP3, IMAP, NNTP,  
...). Здесь будет рассмотрена работа с сокетами (модуль socket) и три  
модзия высоюуровневых протоюлов (urllib2, poplib, smtplb). При этом  
предполагается, что имеется понимание принципов работы ІР-сети и  
некоторых ее сервисов, а также представление о системе WWW.**

Работа с сокетами

Применяемая в ІР-сетях архитектура клиент-сервер использует ІР-  
пакеты для коммуникации между клиентом и сервером. Клиент  
отправляет запрос серверу, на который тот отвечает. В слздіае с TCP/IP  
между клиентом и сервером устанавливается соединение (обычно с  
двусторонней передачей данных), а в слзліае с UDP/IP - клиент и сервер  
обмениваются пакетами (дейтаграммамми) с негарантированной  
доставюй.

Каждый сетевой интерфейс ІР-сети имеет уникальный в этой сети  
адрес **(ІР-адрес** ). Упрощенно можно считать, что каждый компьютер в  
сети Интернет имеет собственный ІР-адрес. При этом в рамках одного  
сетевого интерфейса может быть несюлью сетевых портов. Для  
установления сетевого соединения приложение клиента должно  
выбрать свободный порт и установить соединение с серверным  
приложением, юторое слушает (listen) порт с определенным номером на  
удаленном сетевом интерфейсе. Пара ІР-адрес и порт характериззлот  
сокет (гнездо) - начальную (конечную) точку сетевой коммуникации.  
Для создания соединения TCP/IP необходимо два сокета: один на  
локальной машине, а дрзтой - на удаленной. Таким образом, каждое  
сетевое соединение имеет ІР-адрес и порт на локальной машине, а  
также ІР-адрес и порт на удаленной машине.

Модуль **socket** обеспечивает возможность работать с сокетами из

**Сузи RA. Язык программирования Python**

Python. Сокеты используют транспортный уровень согласно

семиуровневой модели OSI (Open Systems Interconnection,  
взаимодействие открытых систем), то есть относятся к более низюму  
уровню, чем большинство описываемых в этом разделе протоколов.

Уровни модели OSI;

Физический

Поток битов, передаваемых по 4мзичесюй линии. Определяет  
параметры с}мзичесюй линии.

Канальный (Ethernet, РРР, AIM и т.п.)

Кодирует и деюдирует данные в виде потока битов, справляясь с  
ошибками, возникающими на физичесюм уровне в пределах 4»ізически  
единой сети.

Сетевой (IP)

Марпрутизирует 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, POPS, IMAP и т.д.)

**Сузи RA. Язык программирования Python**

Поддерживает юнкретные 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 протоюлов и адресацию. Данное  
изложение будет затрагивать толью домен Internet, а именно протоюлы  
TCP/IP и UDP/IP, поэтому для указания юммуникационного домена при  
создании сокета будет указываться юнстанта **socket. AF\_INET.**

В качестве примера следует рассмотреть простейшую клиент-сервернзло  
пару. Сервер будет принимать строку и отвечать клиенту. Сетевое  
устройство иногда называют хостом (host), поэтому будет употребляться  
этот термин по отношению к юмпьютеру на ютором работает сетевое  
приложение.

Сервер:

import socket, string

def do\_something(x):

1st = map(None, x);

lst.reverse();

return string.join(Jst, "")

**HOST=""** #Iocalhost

PORT = 33333

srv = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  
srv.bind((HOST, PORIl)  
while 1:

print "Слушаю порт 33333"  
srv.listen(l)

sock, addr = srv.accept()  
while 1:

pal = sock.recv(1024)  
if not pah  
break

print "Получено от %s:%s:" % addr, pal  
lap = do\_something(pal)

**Сузи Р.А.**

print "Отправлено %s%s:" % addr, lap  
sock.send(lap)  
sock.cfoseQ

Клиент:

import socket

HOST = "" # удаленный юмпьютер (localhost)

PORT = 33333 # порт на удаленном компьютере

sock = socket.socket(socket.AF\_lNET, socket.SOCK\_STREAM)

sock.connect((HOST, PORT))

**sock.send('TlAЛИHДPOM')**

result = sock.recv(1024)

sock.closeQ

print "Получено:", result

Примечание:

В примере использованы русские буквы: необждимо указывать  
юдировку.

Прежде всего, нужно запустить сервер. Сервер открывает сокет на  
локальной маижне на порту 33333, и адресе 127.0.0.1. После этого он  
слушает ( **listen** **()** ) порт. Когда на порту появляются данные,  
принимается ( **accept** **()** ) вждящее соединение. Метод **accept** **()**возвращает пару - Socket-объект и адрес удаленного компьютера,  
устанавливающего соединение (пара - ІР-адрес, порт на здаленной  
машине). После этого можно применять методы **гесѵ() и send()**для общения с клиентом. В **гесѵ** **()** 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 **гесѵ** **()** - аналогично тому,  
что происходит на сервере.

**Сузи RA. Язык программирования Python**

Модуль **socket** имеет несколью вспомогательных функций. В  
частности, функции для работы с системой доменных имен **(DNS** ):

»> import socket

»> socket.gethostbyname('www.onego.m')

('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** О . Она позволяет преобразовывать  
наименования Интернет-сервисов в обіцепринятые номера портов:

»> for srv in 'http', 'ftp', 'imap', 'pop3', 'smtp':

... print socketgetservbyname(srv, 'tcp'), srv

80 http  
21 ftp  
143 imap  
110 pop3  
25 smtp

Модуль также содержит большое количество юнстант для указания  
протоколов, типов сокетов, коммуникационных доменов и т.п. Другие  
функции модуля **socket** можно при необходимости изучить по  
документации.

Модуль smtplib

Сообіцения электронной почты в Интернете передаются от клиента к  
серверу и между серверами в основном по протоюлу SMTP ( S imple М  
ail Т ransfer Р rotocol, простой протоюл передачи почты). Протоюл SMTP  
и ESMTP (расширенный вариант SMTP) описаны в ссылка: RFC 821 -  
httpy/www.£aqs.org/rfcs/rfc821.html и ссылка: RFC 1869

httpy/www.£aqs.org/rfcs/rfc821.htmL Для работы с SMTP в стандартной  
библиотеке модулей имеется мод}ль **smtplib.** Для того чтобы начать  
SMTP-соединение с сервером электронной почты, необходимо в начале

**Сузи RA. Язык программирования Python**

создать объект для управления SMTP-сессией с помощью констрзш'ора  
класса **SMTP:**

smtplib.SMTP([host[, port]])

Параметры **host и port** задают адрес и порт SMTP-сервера, через  
юторый будет отправляться почта. По умолчанию, **port=2** **5.** Если  
**host** задан, конструктор сам установит соединение, иначе придется  
отдельно вызывать метод **connect** **().** Экземпляры класса **SMTP**имеют методы для всех распространенных юманд SMTP-протокола, но  
для отправки почты достаточно вызова конструктора и методов  
**sendmail () и quit** **():**

# -\*- coding: ср1251 -\*-

from smtplib import SMTP

fromaddr = "student@mairu" # От шго

toaddr = "[rnd@onego.ru](mailto:rnd@onego.ru)" # Кому

message = "'"Тгош: 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 изздіаю курс по языку Python и  
отправляю письмо его автору

**ІП11І**

connect = SMTP('maiLonego.m')  
connect set\_debuglevel( 1)

connectsendmail(fromaddr, toaddr, message % varsQ)  
connect quit()

Следует заметить, что **toaddr** в сообщении (в поле То ) и при  
отправке могут не совпадать. Дело в том, что получатель и отправитель  
в ходе SMTP-сессии передается командами SMTP-протоюла. При  
запуске указанного выше примера на экране появится отладочная  
информация (ведь уровень отладки задан равным 1):

send: 'еЫо rnd.onego.ru\r\n'

**Сузи RA. Язык программирования Python**

reply: '250-rraiLonego.ru Hello as3-042.dialup.onego.ru [195.161.147.4], pleasi

send: 'mail FROM:<student@maiLru> size=270\r\n'

reply: '250 2.1.0 <student@maiLru>... Sender ok\r\n'

send: 'rcpt TO:<[md@onego.ru](mailto:md@onego.ru)>\r\n'

reply: '250 2.1.5 <md(2)onego.ru>... Recipient ok\r\n'

send: 'data\r\n'

reply: '354 Enter mail, end with"." on a line by itselhr\n'

send: 'From: Student <student@maiLm>\r\n...'

reply: '250 2.0.0 iBPFgQ7q028433 Message accepted for deliveiy\r\n'

send: 'quit\r\n'

reply: '221 2.0.0 maiLonego.ru closing connection\r\n'

Из этой (несколью сокращенной) отладочной информации можно  
увидеть, что клиент отправляет (send) команды SMTP-серверу (EHLO,  
MAIL FROM, RCPT ТО, DAIA, QUIT), a тот выполняет юманды и  
отвечает (reply), возвранря юд возврата.

В ходе одной SMTP-сессии можно отправить сразу несколью писем  
подряд, если не вызывать **quit** **().**

В принципе, юманды SMTP можно подавать и отдельно: для этого у  
объекта-соединения есть методы ( **helo (), ehlo** **(),** expn (),  
**help О , mail (), rcpt (), vrfу (), send** **(),** noop (), **data** **()** ),  
соответствующие одноименным юмандам SMTP-протоюла.

Можно задать и произвольную юманду SMTP-серверу с помощью  
метода **docmd().** В следующем примере показан простейший  
сценарий, юторый могут использовать те, кто время от времени  
принимает почту на свой сервер по протоюлу SMTP от почтового  
сервера, на ютором хранится очередь сообщений для неюторого  
домена:

from smtplib import SMTP  
connect = SMTP('mx.abcde.ru')  
connect set\_debuglevel( 1)  
connectdocmd('ETRN md.abcde.ru")  
connect quit()

Этот простенький сценарий предлагает серверу mx. **abode. ru**

**Сузи RA. Язык программирования Python**

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 неюторых из них приведено ниже:

smtplib.SMTP Exception

Базовый класс для всех исключений модуля,  
smtplib. S МТР S erverDisconnec ted

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. S МТР S enderRe fused  
Отправитель отвергнут  
smtplib.SMTPRecipientsReftjsed  
Все получатели отвергнуты сервером,  
smtplib.SMTPDataError

Сервер ответил неизвестным юдом на данные сообщения.

smtplib.SMTPConnectError  
Ошибка установления соединения,  
smtplib.SMTPHelo Error

Сервер не ответил правильно на команду HELO или отверг ее.

Модуль рорИЬ

**Сузи RA. Язык программирования Python**

Еще один протоюл - POP3 ( Р ost О ffice Р rotocol, почтовый протоюл) -  
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-юманд полздіить требуемые сообщения. Объект-соединение  
POP3 можно установить посредством конструктора класса **POP3** из  
**модзия pop lib:**

poplib.POP3(host[, port])

Где **host** - адрес РОРЗ-сервера, **port** - порт на сервере (по  
умолчанию 110), **рор\_оЬ** j - объект для управления сеансом работы с  
РОРЗ-сервером.

Следующий пример демонстрирует основные методы для работы с  
РОРЗ-соединением;

import poplib, email

* Учетные данные пользователя:

SERVER = "pop.server.com"

USERNAME = "user"

USERPAS SWORD = "secretword"

p = poplib.POP3(SERVER)  
print p.getwelcomeO

* этап идентификации  
  print p.user(USERNAME)

print p.pass\_(USERPASSWORD)

* этап транзакций  
  response, 1st, octets = p.list()  
  print response

for msgnum, msgsize in [LsplitO for i in 1st]:  
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:

**Сузи RA. Язык программирования Python**

(resp, lines, octets) = p.retr(msgnum)  
msgtxt = "\n".join(lines)+”\n\n"  
msg = email niessage\_from\_string(nKgtxt)

print "\* От: %(from)s\n\* Кому: %(to)s\n\* Тема: %(subject)s\n" % msg  
# msg содержит заголовки сообщения или все сообщение (если оно неі

* этап обновления  
  print p.quitO

Примечание:

Разз^еется, чтобы пример сработал юрректно, необждимо внести  
реальные з^іетные данные.

При выполнении сценарий выведет на экран примерно следующее.

+ОК POP3 pop.server.com server ready  
+ОК 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](mailto:online@kaspersky.com)
* Кому: [user@server.com](mailto:user@server.com)
* Тема: KL Online Activation  
  +OK Sayonara

**Метод**

**Команда**

**POP3**

**Описание**

Эти и другие методы экземпляров класса **POP3** описаны ниже:

**getwelcome(!**

Получает строку s с приветствием РОРЗ-  
сервера

Посылает юманду **USER** с указанием имени

**Язык программирования Python**

**QUIT**

**retr(num)**

**dele(num)  
rset ()  
noop ()**

**quit ()  
top(num.**

**RETR**

**num**

**DELE**

**num**

**RSET**

**NOOP**

**list([num])**

**LIST**

**[num]**

**stat ()**

**rpop(user)**

**apop(user,  
secret)**

**pass (pwd)**

**name**

**PASS**

**pwd**

**APOP**

**user**

**secret**

**RPOP**

**user**

**STAT**

**Сузи Р.А.**

**user(name)**

пользователя name. Возвращает строку с  
ответом сервера

Отправляет пароль пользователя в юманде  
PASS. После этой юманды и до  
выполнения команды QUIT почтовый ящик  
блокируется

Идентификация на сервере по АРОР

Идентификация по методу RPOP

Возвращает юртеж с информацией о  
почтовом ящике. В нем m - количество  
сообщений, 1 - размер почтового ящика в  
байтах

Возвращает список сообщений в формате  
**(resp, ['num octets', ...]),**если не указан num, и " +ОК num  
**octets",** если указан. Список 1st  
состоит из строк в формате **"num  
octets".**

Загружает с сервера сообщение с номером  
num и возвращает юртеж с ответом сервера  
**(resp, 1st,** **octets)**

З^аляет сообщение с номером num

Отменяет пометки здаления сообщений

Ничего не делает (поддерживает  
соединение)

Отключение от сервера. Сервер выполняет  
все необходимые изменения (удаляет  
сообщения) и снимает блокировід^  
почтового ящика

**TOP num**

Команда аналогична RETR, но загружает  
толью заголовок и **lines** строк тела

**Язык программирования Python**

**uidl([num])**

**Сузи Р.А.**

**top(num,  
lines)**

TOP num толью заголовок и **lines** строк тела  
lines сообщения. Возвращает юртеж **(resp,  
1st,** **octets)**

Сокращение от "unique-id listing" (список  
уникальных идентис}мкаторов сообщений).

**UIDL**

**[num]**

Формат результата: **(resp, 1st,  
octets),** если num не указан, и **"+0К  
num uniqid",** если указан. Список 1st  
состоит из строк вида " +ОК num  
**uniqid"**

В этой таблице num обозначает номер сообщения (он не меняется на  
протяжении всей сессии), **resp** — ответ сервера, возвращается для  
любой юманды, начинается с **"+ОК** " для успешных операций (при  
неудаче возбуждается исключение **рорІіЬ. proto\_error** ).  
Параметр **octets** обозначает юличество байт в принятых данных,  
**uniqid** - иденти(]»ікатор сообщения, генерируемый сервером.

Работа с РОРЗ-сервером состоит из трех фаз: идентификации,  
транзакций и обновления. На этапе идентис]мкации сразу после  
создания РОРЗ-объекта разрешены толью юманды USER, PASS (иногда  
АРОР и RPOP). После идентис]мкации сервер получает информацию о  
пользователе и настзпает этап транзакций. Здесь злиестны остальные  
юманды. Этап обновления вызывается юмандой QUIT, после юторой  
РОРЗ-сервер обновляет почтовый ящик пользователя в соответствии с  
поданными юмандами, 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 слзліаях, так и при сложных  
обстоятельствах, в частности при необходимости передавать данные  
формы, идентис]мкации, достзпа через прокси и т.п.

Стоит отметить, что при работе с WWW используется в основном  
протоюл HTTP, однаю WWW охватывает не толью HTTP, но и многие  
другие схемы (FTP, gopher, HTTPS и т.п.). Используемая схема обычно

**Сузи Р.А.**

указана в самом начале URL.

Функции для загрузки сетевых объектов

Простой слз^ай пол}Аіения WWW-объекта по известному URL показан в  
следующем примере:

import urllib

doc = urllib.urlopen("httpy/python.onego.ru").read()  
print doc[:40]

Функция **urllib. urlopen** **()** создает файлоподобный объект,  
юторый читает методом **read().** Дрзтие методы этого объекта:  
**readline (), readlines (), f ileno (), close** **()** работают как и  
у обычного файла, а также есть метод **info(),** который возвращает  
соответствующий полз^енному с сервера Message-объект. Этот объект  
можно использовать для получения дополнительной информации:

»> import urllib

»> f = urllib.urlopen("httpy/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-T**3**^e']  
text/html; charset=windows-1251

C 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 СЕТ или метода POST. Метод СЕТ  
связан с юдированием всех передаваемых параметров после знака " ? "  
в URL, а при методе POST данные передаются в теле НТГР-запроса.  
Оба варианта передачи представлены ниже:

import urllib

data = {"search": "Python"}

**Сузи Р.А.**

enc\_data = urllib.urlencode(data)

* метод GET

f = urllib.urlopen("httpy/searchengine.com/search" + '?" + enc\_data)  
print f.read()

* метод POST

f = urllib.urlopen("httpy/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"), ('ЪиДоп", 'Привет''),]

»> enc\_data = urllib.urlencode(data)

»> print enc\_data

n= l&n=3&n=4&button=%F0%D2%C9%D7%C 5%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  
Мае OS они берутся из конфигурации Internet. Задать прокси-сервер  
можно и как именованный параметр **proxies** к  
**urllib.urlopen():**

* Использовать указанный прокси  
  proxies = {'http': 'httpy/www.proxy.com;3128'}  
  f = urllib.urlopen(some\_url, proxies=proxies)
* He 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). Фзтзкция возвращает кортеж  
**(filepath, headers) , где filepath** - имя локального файла, в  
юторый 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:  
юличество принятых блоюв, размер блока и общий размер  
принимаемого объекта в байтах (если он неизвестен, этот параметр  
равен -1).

В следзлощем 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-y/download.fedora.redhat.com/pub/fedora/linux/core/3/SRPMS/' + I

def download(url, file):  
import urllib, time  
start\_t = time.timeO

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

**Сузи RA. Язык программирования Python**

print "%6.2f %% %6.0f s %6.0f s %6i / %-6i bytes" % (  
p\*100, elapsed, est\_t, dkJsize, size)  
else:

print "%6i/ %-6i bytes" % (dldsize, size)  
urllib.urJretrieve(URL, FILE, progress)  
download(URL, FILE)

Эта программа выведет примерно следующее (процент от полного  
объема закачки, прошедшие секунды, предполагаемое оставшееся время,  
закачанные байты, полное юличество байтов):

|  |  |  |  |
| --- | --- | --- | --- |
| 0.00 % | 1 s | Os | 0 / 6952309 bytes |
| 0.12 % | 5s | 3941 s | 8192 / 6952309 bytes |
| 0.24 % | 7s | 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 | 2441s | 57344 / 6952309 bytes |

Функции для анализа URL

Согласно доі^енту ссылка: RFC 2396

httpy/www.£aqs.org/rfcs/rfc2396.html URL должен строиться по следующему  
шаблону:

scherney/nedoc/path;parameters?queiy#fragment

где

scheme

Адресная схема. Например: http, ftp, gopher,  
nedoc

**Сузи RA. Язык программирования Python**

Местонажждение в сети,  
path

Путь к ресурсу  
params  
Параметры,  
query

Строка запроса,  
fragment

Идентификатор фрагмента.

Одна из фзтікций 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 ѵѵеЬ-сервер. Она предназначена для экранирования пути к ресурсу,  
поэтому оставляет ' / ' как есть. Например:

»> urllib.quote("[rnd@onego.ru](mailto:rnd@onego.ru)")

'rnd%40onego.ru'

»> urllib.quote("a = b + c")

'a%20%3D%20b%20%2B%20c'

»> urllib.quote("0/l/l")

**'**0**/**1**/**1**'**

»> urllib.quote("0/l/l", 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.

**Сузи RA. Язык программирования Python**

unquote(s)

Преобразование, обратное **quote** **().** Пример:

»> urllib.unquote('a%20%3D%20b%20%2B%20c')

'а = b + с'

Преобразование, обратное **quote\_plus** **().** Пример:

»> urllib.unquote\_plus('a+=+b+%2B+c')

'а = Ь + с'

Для анализа URL можно использовать функции из модуля **urlparse:**urlparse(url, scheme=", allow\_fragments=l)

Разбирает URL в 6 компонентов (сохраняя экранирование символов):

**scheme://netloc/path;params?query#frag**

urJsplit(url, scheme=", allow\_fragments=l)

Разбирает URL в 5 компонентов (сохраняя экранирование символов):

**scheme://netloc/path?query#frag**

urlunparse((scheme, netloc, url, params, query, fragment))

Собирает URL из 6 юмпонентов.

urlunsplit((scheme, netloc, url, query, fragment))

Собирает URL из 5 юмпонентов.

Пример:

»> from urlparse import urlsplit, urlunsplit  
»> URL = "httpy/google.conVsearch?q=Python"

»> print urlsplit(URL)

('http', 'google.com', '/search', 'q=Python',")

»> print urlunsplit(

('http', 'google.com', '/search', 'q=Python',"))  
http y/google. **CO** tiVsearch?q=Python

**Сузи RA. Язык программирования Python**

Еще одна фз^кция того же модзэтя **urlparse** позволяет корректно  
соединить две части URL - базовзло и относительную:

»> import urlparse

»> urlparse.urljoin('httpy/python.onego.ru', 'itertooJs.html')

' http y/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 слздгай можно  
использовать модзль для работы с протоюлом HTTP - **httplib** - и  
создать собственный класс для НТТР-запросов (в лекциях модуль  
**httplib** не рассматривается). Однако вполне вероятно, что нужная  
функциональность уже имеется в модзле **urllib2.**

Одна из полезных возможностей этих модулей - доступ к web-объектам,  
требующий авторизации. Ниже будет рассмотрен пример, юторый не  
только обеспечит доступ с авторизацией, но и обозначит основнзло  
идею модзля **urllib2:** использование обработчиков (handlers), каждый  
из юторых 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. Н i ’ 1 PBasic AuthHandler()  
authinfo.add\_password('My page', 'localhost', 'userl', 'secret')

* Доступ через прокси

proj^\_support = urllib2.ProxyHandler({'http': 'httpy/localhost:8080'})

* Создание нового открывателя с указанными обработчиками  
  opener = urllib2.build\_opener(proxy\_support,

authinfo,

urllib2. С acheFTP Handler)

# 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('httpy/localhost/mywebdir/')  
print f.read()[:100]

В этом примере получен доступ к странице, которую охраняет  
mod\_p**3**Thon (см. предыдущую лекцию). Первый аргумент при вызове  
метода **add\_password** () задает область действия (realm)  
идентификационных данных (он задан директивой **AuthName** "Му  
**раде"** в юнфигурации web-сервера). Остальные параметры  
достаточно понятны: имя хоста, на юторый нужно получить доступ,  
имя пользователя и его пароль. Разз^еется, для юрректной 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 РзДЬоп.  
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 достзп к каталогу с авторизацией, используя добавление  
заголовка в НТТР-запрос:

import urllib2, base64

req = urllib2.Request('<http://localhost/mywebdir'>)  
b64 = base64.encodestring('userl:secret').strip()

**Сузи RA. Язык программирования Python**

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 (заюдированные base64) иденти(}мкационные  
данные в поле **Authorization** HTTP-запроса.

Примечание:

Приведенные два примера почти эквивалентны, толью во втором  
примере прокси-сервер не назначен явно.

XML-RPC сервер

До сих пор высоюуровневые протоюлы рассматривались с точки  
зрения клиента. Не менее просто создавать на Python и их серверные  
части. Для иллюстрации того, как разработать программу на Р}ДЬоп,  
реализующую сервер, был выбран протоюл XML-RPC. Несмотря на  
свое название, юнечному пользователю необязательно знать XML (об  
этом языке разметки говорилось на одной из предыдущих лекций), так  
как он скрыт от него. Сокращение RPC ( R emote Р rocedure С all, вызов  
удаленной процедуры) объясняет суть дела: с помощью XML-RPC  
можно вызывать процедуры на удаленном хосте. Причем при помощи  
XML-RPC можно абстрагироваться от юнкретного языка  
программирования за счет использования общепринятых типов данных  
(строки, числа, логические значения и т.п.). В языке РзДйоп вызов  
удаленной функции по синтаксису ничем не отличается от вызова  
обычной функции:

import xmlrpclib

# Установить соединение

req = xmlrpclib.ServerProj^"httpy/localhost:8000")  
tiy:

**Сузи RA. Язык программирования Python**

# Вызвать удаленную функцию  
print req.add(l, 3)  
except xmlrpcJib.Error, v:  
print "ERROR",

A 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 фз^кцию  
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 "легювесный" среди  
других подобных протоюлов) приложения могут общаться друг с другом  
на понятном им языке вызова функций с параметрами основных  
общепринятых типов и такими же возвраиремыми значениями.  
Преимуществом же Python является удобный синтаксис вызова  
удаленных функций.

Внимание!

Разз^еется, это толью пример. При реальном использовании  
необходимо позаботиться, чтобы XML-RPC сервер отвечал  
требованиям безопасности. Кроме того, сервер лзліше делать  
многопоточным, чтобы он мог обрабатывать несюлью потоюв  
одновременно. Для многопоточности (она будет обсуждаться в  
отдельной лекции) не нужно многое переделывать: достаточно  
определить свой класс, скажем, **ThreadingXMLRPCServer,** в  
ютором вместо **SocketServer . TCPServer** использовать  
**SocketServer . ThreadingTCPServer.** Это предлагается в  
качестве упражнения. Наводящий вопрос: где нахэдится определение  
класса **SimpleXMLRPCServer?**

Заключение

**Сузи RA. Язык программирования Python**

в этой лекции на практических примерах и сведениях из доід^ентации  
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 протоюла или набора протоюлов. 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 мод}ли  
**urllib, urllib2, urlparse.** Указанные модули рассмотрены с  
точки зрения типичного применения. Для решения нестандартных  
задач лучше обратиться к другим источникам: документации, исходному  
юду модулей, поисіо^ в Интернете. В этой лекции говорилось и о  
серверной составляющей высоюуровневых сетевых протоколов. В  
качестве примера приведена клиент-серверная пара для протокола  
XML-RPC. Этот протоюл создан на основе HTTP, но служит  
специальной цели.

**Сузи Р.А.**

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.**

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-API2

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, должны  
реализовывать все мод}ли расширения, юторые 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 легю менять одну СУБД на  
другую, ИЗЗД**1**ИВ всего один набор функций и методов.

DB-API 2.0 описан в РЕР 249 (сайт ссылка:  
httpy/[www.python.org/dev/peps/pep-0249/](http://www.python.org/dev/peps/pep-0249/)), и данное ниже описание  
основано именно на нем.

**Сузи Р.А.**

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 (уточнитъ  
можно на web-странице по адресу ссылка:  
httpy/[www.python.org/topics/database/](http://www.python.org/topics/database/)). Ниже рассматриваются почти все  
положения DB-API за исключением рекомендаций для разработчиюв  
новых модулей.

Интерфейс модутія

Здесь необходимо сказать о том, что должен предоставлять мод}ль 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-юнструктор **connect** О 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 юнстанты, содержащие его основные  
характеристики:

**арііеѵеі** Версия DB-API ("1.0"или "2.0").

**threadsafety** Целочисленная юнстанта, описывающая  
возможности модуля при использовании потоюв управления:

**Сузи Р.А. Язык программирования Python**

**о**

**о**

1. Модзиь не поддерживает потоки.
2. Потоки могут совместно использовать модзль, но не  
   соединения.

о 2 Потоки могут совместно использовать модуль и  
соединения.

о 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:  
о **"format"** Форматирование в стиле языка ANSI С  
(например, **"%s", "%і" ).**

**о " РУ format"** Использование именованных

специ4мкаторов формата в стиле РзДЬоп( "% **(item) s" )  
о "qmark"** Использование знаков "?" для пометки мест  
подстановки параметров.

**о**

**о**

**"numeric"** Использование номеров позиций ( " : 1" ).  
**"named"** Использование имен подставляемых параметров  
**( " : name" ).**

Модуль должен определять ряд исключений для обозначения типичных  
исключительных ситуаций: **Warning** (предзпреждение). **Error**(ошибка), **InterfасеЕггог** (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

**Сузи RA. Язык программирования Python**

функции **connect** **(),** должен иметь следующие методы:

* **close** О Закрывает соединение с базой данных.
* **commit** О Завершает транзакцию.
* **rollbackO** Откатывает начатую транзакцию (восстанавливает  
  исходное состояние). Закрытие соединения при незавершенной  
  транзакции автоматически производит откат транзакции.
* **cursor** О Возвращает объект-курсор, использующий данное  
  соединение. Если база данных не поддерживает курсоры, модзль  
  сопряжения должен их имитировать.

Под транзакцией понимается группа из одной или нескольких  
операций, юторые 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, теідлций набор  
записей) служит для работы с результатом запроса. Реззльтатом запроса  
обычно является одна или несюлько 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.

**Сузи RA. Язык программирования Python**

Конкретный курсор в любой момент времени связан с выполнением  
одной SQL-инструкции.

Атрибуты объекта-курсора тоже определены DB-API:

**arraysize** Атрибут, равный юличеству записей, возвращаемых  
методом **fetchmany ()**. По умолчанию равен 1.  
**callproc (procname [, params])** Вызывает хранимзло  
процедуру **procname** с параметрами из изменчивой  
последовательности **params.** Хранимая процедура может  
изменить значения некоторых параметров последовательности.  
Метод может возвратить результат, доступ к юторому  
осуществляется через **fetch** -методы,  
**close О** Закрывает объект-курсор.

**description** Этот доступный толью для чтения атрибут  
является последовательностью из семиэлементных  
последовательностей. Каждая из этих последовательностей  
содержит информацию, описывающую один столбец результата:

(папзе, type\_code, display\_size, internal\_size, precision, scale, пиД\_ок)

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 юманду 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 юманд, подставляя параметры в  
заданный шаблон. Параметр **seq\_of\_parameters** задает  
последовательность наборов параметров.

**fetchallO** Возвращает все (или все оставшиеся) записи  
результата запроса.

**fetchmany** ( **[size]** ) Возвращает следующие несюлью  
записей из реззльтатов запроса в виде последовательности

**Сузи RA. Язык программирования Python**

последовательностей. Пустая последовательность означает  
отсутствие данных. Необязательный параметр **size** заказывает  
количество возвращаемых записей (реально возвращаемых  
записей может быть меньше). По умолчанию **size** равен  
атрибуту **arraysize** объекта-кзрсора.

* **fetchoneO** Возвращает следзаощую запись (в виде  
  последовательности) из результата запроса или **None** при  
  отсутствии данных.
* **nextsetO** Переводит курсор к началу следующего набора  
  данных, полученного в реззльтате запроса (при этом часть  
  записей в предыдущем наборе может остаться непрочитанной).  
  Если наборов больше нет, возвращает **None.** Не все базы данных  
  поддерживают возврат несюльких наборов результатов за одну  
  операцию.
* **rowcount** Количество записей, ползліенных или затронутых в  
  результате выполнения последнего запроса. В случае отсутствия  
  execute-запросов или невозможности указать юличество записей  
  равен -1.
* set **inputs izes (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 ( В іпагу L arge О bject,  
  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  
**ROW ID** Идентификатор записи

**None** NULL-значение (отсутствующее значение)

С каждым типом данных (в реальности это - классы) связан  
конструктор. Совместимый с DB-API модзль должен определять  
следующие констрзжторы:

* **Date (год, месяц, день)** Дата.
* **Time (час, минута, секунда)** Время.
* **Times tamp(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.
* **Times tampFromTicks** (**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 юнкретных примерах будет показано, как работатъ с  
базой данных из программы на языке РзіЬоп. 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 дрзтим источникам данных (файлам, сетевым объектам).

Именно поэтому для демонстрации выбрана СУБД SQLite, работающая  
как под Unix, так и под Windows. Кроме установки собственно SQLite

**Сузи RA. Язык программирования Python**

(сайт ссылка: httpy/sqlite.org) и модуля сопряжения с Python (ссылка:  
httpy/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 сзлцественную 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  
любзло другую СУБД потребует минимальных изменений в вызове  
функции **connect** О 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 несюльких курсоров (вызов метода объекта-  
  соединения **cursor** **()** с ползліением 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** О ).
* Когда все необходимые транзакции произведены, подключение  
  закрывается вызовом метода **close** **()** объекта-соединения.

Знакомство с СУБД

**Сузи RA. Язык программирования Python**

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 РзДЬоп,  
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 юторой будет храниться  
телепрограмма. В этой базе будет таблица со следующими полями:

**tvdate,**

**tvweekday,**

**tvchannel,**

**tvtimel,**

**tvtime2,**

**prname,**

**prgenre.**

Здесь **tvdate** - дата, **tvchannel** - канал, **tvtimel и tvtime2** -  
время начала и юнца передачи, **prname** - название, **prgenre** - жанр.  
Конечно, в этой таблице есть фзлікциональная зависимость (

**Сузи RA. Язык программирования Python**

**tvweekday** вычисляется на основе **tvdate и tvtimel** ), но **из**практических соображений БД к нормальным формам приводиться не  
будет. Кроме того, буцет создана таблица с названиями дней недели  
(устанавливает соответствие между номером дня и днем недели):

* **weekday,**
* wdname.

Следующий сценарий создаст таблицу в базе данных (в слзліае 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.cursorO

tiy:

cu.execute('""'

CREATE TABLE tv (  
tvdate DATE,  
tvweekday INTEGER,  
tvchannel VARCHAR(30),  
tvtimel TIME,  
tvtime2 TIME,  
prname VARCHAR(150),  
prgenre VARCHAR(40)

**):**

**in II**

except db.DatabaseEiTor, x:  
print "Ошибка:", x

c.commitO  
tiy:

cu.execute('""'

CREATE TABLE wd (  
weekday INTEGER,  
wdname VARCHAR(ll)

**Сузи RA. Язык программирования Python**

**):**

**III tii^**

except db.DatabaseEiTor, x:  
print "Ошибка:", x  
c.commitQ  
c.closeO

Здесь просто исполняются SQL-инструкции, и обрабатывается ошибка  
базы данных, если таковая слз^ится (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

с = db.connect(database="tvprogram")  
cu = c.cursorO

tiy:

cu.execute('"'"DROP TABLE tv;'""')  
except db.DatabaseError, x:  
print "Ошибка:", x  
c.commitO

tiy:

cu.execute('""'DROP TABLE wd;""')  
except db.DatabaseError, x:  
print "Ошибка:", x  
c.commitO  
c.closeO

Наполнение базы данных

Теперь можно наполнить таблицы значениями. Следует начать с  
расшифровки числовых значений для дней недели:

weekdays = ["Воскресенье", "Понедельник", "Вторник", "Среда",

**Сузи RA. Язык программирования Python**

'Четверг", "Пятница", "Суббота", ’Воскресенье"]  
import sqlite as db

с = db.connect(database="tvprogram")  
cu = c.cursorO

caexecute( DELETE FROM wd;""")

caexecutemany( 'INSERT INTO wd VALUES (%s, %s); ,

enumerate(weekdays))

c.commitQ

c.closeQ

Стоит напомнить, что встроенная функция **enumerate ()** создает  
список пар номер-значение, например:

»> print [і for i in enumerate(['a', 'b', 'c'])j  
[(0, 'a'), (1, 'b'), (2, 'c')j

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):

1. 9.00|ОРТ|Новости|Новости|9.15
2. 9.15|ОРТ|'НЕЖНЫЙ ЯД"|Сериал110.15
3. 10.151ОРТ|'Маски-шоу"|Юмористическая программаі 10.45
4. 10.451ОРТГЧеловеки закон"||11.30
5. 11.30|ОРТ|'НОВЫЕ ПРИКЛЮЧЕНИЯ СИНДБАДА"|Сериа.

Следующая программа разбирает CSV-файл и записывает данные в  
таблицу tv:

import calendar, csv  
import sqlite as db

from sqlite.main import Time, Date ## Толью для  
db.Date, db.Time = Date, Time ## sqlite

c = db.connect(database="tvprogram")

**Сузи Р.А.**

CU = c.cursorO

input\_file = open("tv.csv", "rb")  
rdr = csv.DictReader(input\_fiIe,

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)

recl'begd'] = db.Date(bdy, bdm, bdd)

recl'begt'] = db.Time(bth, btm, 0)

rec['endt'] = db.Time(eth, еіід 0)

caexecute( INSERT INTO tv

(tvdate, tvweekday, tvchannel, tvtimel, tvtime2, prname, prgenre)

VALUES(

%(begd)s, %(wd)s, %(channel)s, %(begt)s, %(endt)s,

%(prname)s, %(prgenre)s);""", rec)  
input\_file.close()  
c.commitQ

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.** та **in** 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

**Сузи RA. Язык программирования Python**

некорректной при попытке выполнить подстановку самостоетельно,  
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

с = db.connect(database="tvprogram")  
cu = c.cursorO

caexecute("SELECT weekday, wdname FROM wd ORDER BY weekday;")  
for i, n in cafetchall():  
print i, n

Если все было сделано правильно, получится:

О 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

с = db.connect(database="tvprogram")

cu = c.cursorO

caexecuteC"'"

SELECT tvdate, tvtimel, wd.wdname, tvchannel, prname, prgenre  
FROM tv, wd

WHERE wd.weekday = tvweekday

**Сузи RA. Язык программирования Python**

ORDER BY tvdate, tvtimel; **J**

for rec in cu.fetchall():  
dt = rec[0] + rec[l]  
weekday = rec[2]  
channel = rec[3]  
name = rec[4]  
genre = rec[5]

print "%s, %02L%02L%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 полз^ить 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 6yztyr проюдить более глубокую  
обработід^, их следует переводить во внутреннее представление сразу  
после получения по запросу. Тем самым тип даты из модуля DB-API не  
будет влиять на дрзтие 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 модзль **sqlite,** с изменением его имени  
на db. Это было сделано не случайно. Дело в том, что подобные  
модзли, поддерживающие DB-API 2.0, есть и для других СУБД, и даже  
не в единственном числе. Согласно информации на сайте  
[www.pythoaorg](http://www.pythoaorg) 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 4»ірмой еСепіх (ссылка:  
http y/www. egenix. com).

**DC0racle2, cx\_Oracle** Для СУБД Oracle.

**PyGresQL, psycopg, pyPgSQL** Для СУБД PostgreSQL.  
**Sybase** Для Sybase,  
**sapdbapi** Для СУБД SAP.

**KInterbasDB** Для СУБД Firebird (это потомокInterbase).

**PyADO** Адаптер к Microsoft ActiveX Data Objects (толью под  
Windows).

Примечание:

Для СУБД PostgreSQL нужно взять не P**3**<SreSQL, 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 сюрости 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, юнечно, соответствующий модуль был установлен.

Однаю в общем случае при переходе с одной СУБД на дрзлуло могут  
возникать нестыювки, даже, несмотря на поддержку одной версии DB-  
АРІ. 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 толью 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 юторых позволят работать с XML  
так же легю и стабильно, как с реляционными СУБД. Язык  
программирования Python - один из полигонов этих исследований.

**Сузи RA. Язык программирования Python**

Решая конкретнзао задачу, разработчик программного обеспечения  
должен сделать выбор средств, наиболее подждящих 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 достзпа и хранения. Так, XML можно хранить в реляционной  
БД, а табличные данные - в XML, однаю это неестественно. Из-за этого  
сложность и подверженность ошибкам программного продзлсга  
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), юторого  
должны придерживаться все разработчики модулей сопряжения с  
реляционными базами данных. Благодаря этому API юд 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,  
юторые должен содержать модзль 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 происхэдит коммуникация с СУБД на  
прикладном уровне.

Здесь была использована СУБД SQLite и соответствующий модзль  
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  
полздіенных данных.

В конце лекции дан список других пакетов и модулей, которые  
позволяют Python-программе работать со многими современными  
СУБД.

**Сузи Р.А.**

Ссылки

Модули mxDateTime и др. ссылка: httpy/www.len±)urg.com/files/python/

СУБД SQLite ссылка: httpy/sqlite.org

Модуль сопряжения с SQLite ссылка: httpy/pysqlite.org

**Сузи Р.А.**

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.**

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 модзле os языка Python. Здесь же речь пойдет о потоках.

Потоки управления (threads) образуются и работают в рамках одного  
процесса. В однопоточном приложении (программе, юторая не  
использует дополнительных потоюв) 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 юманду 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 потоюв может в дальнейшем еще несюлью раз  
раздваиваться. (При этом один из потоюв всегда остается главным, и  
его завершение означает завершение всей программы.) В каждый  
момент времени интерпретатор знает, какую юманду каюй поток  
должен выполнить, и уделяет кванты времени каждому потоку. Таюе,  
казалось бы, незначительное усложнение механизма выполнения  
программы на самом деле требует качественных изменений в

**Сузи RA. Язык программирования Python**

программе - ведь деятельность потоков должна быть согласована.  
Нельзя допускать, чтобы потоки одновременно изменяли один и тот же  
объект, результат такого изменения, сюрее всего, нарушит целостность  
объекта.

Одним из классических средств согласования потоков являются  
объекты, называемые семафорами. Семафоры не допускают  
выполнения некоторого участка юда несколькими потоками  
одновременно. Самый простой семафор - замок (lock) или mutex (от  
английсюго mutuaky exclusive, взаимоисключающий). Для того чтобы  
поток мог продолжить выполнение кода, он должен сначала захватить  
замок. После захвата замка поток выполняет определенный }діасток юда  
и потом освобождает замок, чтобы другой поток мог его получить и  
пройти дальше к выполнению охраняемого замюм здіастка программы.  
Поток, столкнувшись с занятым другим потоюм замюм, 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: несюлью видов семафоров (классы замюв **Lock,  
RLock** и класс **Semaphore** ) и другие механизмы взаимодействия  
между потоками (классы **Event** и **Condition** ), класс **Timer** для  
запуска функции по прошествии неюторого времени. Модзль **Queue**реализует очередь, юторой могут пользоваться сразу несюлью потоюв.  
Для создания и (низюуровневого) управления потоками в стандартном  
модзле **thread** определен класс **Thread.**

Пример многопоточной программы

в следующем примере создается два дополнительных потока, юторые  
выводят на стандартный вывод каждый свое:

import threading

def proc(n):  
print "Процесс", n

**Сузи RA. Язык программирования Python**

pi = threading.Thread(target=proc, name="tl", args=["l"])  
p2 = threading.Thread(target=proc, name="t2", args=["2"])  
pl.startQ  
p2.start()

Сначала полздіается два объекта класса **Thread,** которые затем и  
запускаются с различными аргументами. В данном случае в потоках  
работает одна и та же функция **ргос (),** юторой передается один  
аргумент, заданный в именованном параметре **args** констрзжтора  
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  
**"tl" 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  
  Іеп **(threading.enumerate() ).**
* **currentThread** **()** Возвращает текущий объект-поток, то есть  
  соответствующий потоку управления, юторый вызвал эту  
  функцию. Если поток не был создан через мод}иь **threading,**будет возвращен объект-поток с сокращенной функциональностью  
  (dummy thread object).
* **enumerate** О Возвращает список активных потоюв.  
  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 двзлия способами: передать конструктору класса  
исполняемый объект и аргументы к нему или путем наследования

**Сузи RA. Язык программирования Python**

полздіить 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 потоюв (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

pi = threading.Thread(target=proc, name="tl", kwargs={"n": "1"})  
p2 = threading.Thread(target=proc, name="t2", kwargs={"n": "2"})  
pl.startO  
p2.start()

To же самое можно проделать через наследование от класса  
**threading. Thread** с определением собственного констрзжтора и  
метода run ():

import threading

class T(threading. Thread):  
def init\_(self, n):

threading.Thread. init fself. name="t" + n)

self.n = n  
def run(self):  
print 'Процесс", self.n

pi = T("l")  
p2 = T("2")

pl.startQ

p2.stait()

Самое первое, что необждимо 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 потоюв можно управлять вызовом методов:

**start** О ДаетnoTOiq^жизнь.

run () Этот метод представляет действия, юторые должны быть  
выполнены в потоке.

**join** ([ **timeout]** ) Поток, юторый вызывает этот метод,  
приостанавливается, ожидая завершения потока, чей метод  
вызван. Параметр **timeout** (число с плавающей точюй)  
позволяет зжазать время ожидания (в сеі^ыдах), по истечении  
которого приостановленный поток продолжает свою работу  
независимо от завершения потока, чей метод **join** был вызван.  
Вызывать **j оіп** () неюторого 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, поток, вызвавший  
**join О,** желает, чтобы поток по завершении присоединился к  
вызывающему метод потоі^^.

**getNameO** Возвращает имя потока. Для главного потока это  
**"MainThread".**

**setName (name)** Присваивает потоід^имя **name.  
isAliveO** Возвращает истину, если поток работает (метод  
run () уже вызван, но еще не завершился).

**isDaemonO** 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 толью для потоюв, юторые еще не запущены.

**Сузи RA. Язык программирования Python**

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, буцут 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** О . Следующий простой пример, печатающий на стандартном  
выводе **Hello, world!** поясняет сказанное:

defheUoQ:

print "НеДо, world!"

t = Timer(30.0, ЬеИо)  
t.start()

Замки

Простейший замок может быть реализован на основе класса **Lock**модзия **threading.** Замок имеет два состояния: он может быть или  
открыт, или заперт. В последнем слздіае 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 замюм). 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.

**Сузи RA. Язык программирования Python**

**• locked** () Возвращает текущее состояние замка ( **True** - заперт,  
**False** - открыт). Следует иметь в виду, что даже если состояние  
замка толью что проверено, это не означает, что он сохранит это  
состояние до следующей юманды.

Имеется еще один вариант замка - **threading. RLock,** юторый  
отличается от **threading. Lock** тем, что неюторый поток может  
запрашивать его запирание много раз. Отпирание таюго замка должно  
происходить столью же раз, сюлью было запираний. Это может быть  
полезно, например, внутри реі^рсивных функций.

Когда нужны замки?

Замки позволяют ограничивать вход в неюторую область программы  
одним потоюм. Замки могут потребоваться для обеспечения  
целостности структуры данных. Например, если для юрректной работы  
программы требуется добавление определенного элемента сразу в  
несюлью списюв или словарей, такие операции в многопоточном  
приложении следует обставить замками. Вокруг атомарных операций  
над встроенными типами (операций, юторые не вызывают исполнение  
каюго-то другого юда на Python) замки ставить необязательно.  
Например, метод **append** () (встроенного) списка является атомарной  
операцией, а тот же метод, реализованный пользовательским классом,  
может требовать блокировок. В слз^ае сомнений, юнечно, лучше  
перестраховаться и поставить замки, однаю следует минимизировать  
общее время действия замка, так как замок останавливает другие  
потоки, пытающиеся попасть в ту же область программы. Отсутствие  
замка в критичесюй части программы, работающей над общими для  
двух и более потоюв ресзрсами, может привести к случайным,  
трудноуловимым ошибкам.

Тупиювая ситуация (deadlock)

Замки применяются для управления доступом к ресзрсу юторый нельзя  
использовать совместно. В программе таких ресурсов может быть  
несюлью. При работе с замками важно хорошо продумать, не зайдет ли  
выполнение программы в тзпик (deadlock) из-за того, что двум потокам

**Сузи RA. Язык программирования Python**

потребуются одни и те же ресз^сы, но ни тот, ни другой не смогут их  
полздіить, 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(l)

print 'Процесс %s выполняется" % n  
for r in rs:

resource[r].release()

print 'Процесс %s закончил выполнение" % n

pi = threading.Thread(target=proc, name="tl", args=["l", "AB"])

p2 = threading.Thread(target=proc, name="t2", args=["2", 'BA"])

pl.startQ

p2.start()

pl.joinQ

p2.join()

В этом примере два потока (tl и t2) запрашивают замки к одним и тем  
же ресурсам (А и В), но в разном порядке, отчего полздіается, 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 А

Процесс 1 получил ресзфс А  
Процесс 2 запрашивает ресурс В  
Процесс 2 получил ресзфс В  
Процесс 1 запрашивает ресурс В  
Процесс 2 запрашивает ресурс А

**Сузи RA. Язык программирования Python**

Существуют методики, позволяющие избежать подобных тзпиюв,  
однако их рассмотрение не вждит в рамки данной лекции. Можно  
посоветовать следзаощие приемы:

* построить логику приложения так, чтобы ниюгда не запрашивать  
  замки к двум ресзрсам сразу Возможно, придется определить  
  составной ресурс. В частности, к данному примеру можно было  
  бы определить замок "АВ" для указания эксклюзивного достзпа к  
  ресурсам А и В.
* строго зтюрядочить 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 несюлью потоюв. Семафор  
имеет счетчик запросов, уменьшающийся с каждым вызовом метода  
**acquire О** и увеличивающийся при каждом вызове **release** **().**Счетчик не может стать меньше нуля, поэтому в таком состоянии  
потокам приходится ждать, как и в слзліае с замками, пока значение  
счетчика не з^еличится.

Конструктор класса **threading. Semaphore** принимает в качестве  
(необязательного) аргумента начальное состояние счетчика (по  
умолчанию оно равно 1, что соответствует замі^ класса **Lock** ). Методы  
**acquire О** и **release** () действуют аналогично описанным выше  
одноименным методам у замюв.

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:

**Сузи Р.А.**

from threading import BoundedSemaphore  
maxconnections = 5  
# Подготовка семафора

pool\_sema = BoundedSemaphore(value=maxconnections)

* Внутри потока:

pool\_sema.acquire()  
conn = connectdbQ

* ... 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 неюторого  
события от одного потока одному или несюльким другим потокам.  
Объекты-события имеют внутренний флаг, который может находитъся в  
установленном или сброшенном состоянии. При своем создании флаг  
события находится в сброшенном состоянии. Если флаг в  
установленном состоянии, ожидания не происждит: поток, вызвавший  
метод **wait** **()** для ожидания события, просто продолжает свою работу.  
Ниже приведены методы экземпляров класса **threading . Event:**

* **set** О Устанавливает внутренний флаг, сигнализирующий о  
  наступлении события. Все ждущие данного события потоки  
  выходят из состояния ожидания.
* **clear** О Сбрасывает флаг. Все события, которые вызывают  
  метод **wait** О этого объекта-события, будут находитъся в

**Сузи RA. Язык программирования Python**

состоянии ожидания до тех пор, пока флаг сброшен, или по  
истечении заданного таймаута.

* **is Set О** 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 неюторого 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 О** Снимает замок.

**wait (** [ **timeout] )** Переводит поток в режим ожидания. Этот  
метод может быть вызван толью в том случае, если вызывающий  
его поток получил замок. Метод снимает замок и блокирует поток  
до появления объявлений, то есть вызовов методов **notif** у () и  
**notifyAllO** другими потоками. Необязательный аргумент  
**timeout** задает таймаут ожидания в секундах. При выходе из  
ожидания поток снова запрашивает замок и возвращается из  
метода **wait ().**

**notify О** Выводит из режима ожидания один из потоюв,

**Сузи RA. Язык программирования Python**

ожидающих данные условия. Метод можно вызвать, только  
овладев замком, ассоциированным с условием. Документация  
предупреждает, что в будущих реализациях модуая из целей  
оптимизации этот метод будет прерывать ожидание сразу  
несюльких потоков. Сам по себе метод **notify (**) не приводит к  
продолжению выполнения ожидавших условия потоков, так как  
этому препятствует занятый замок. Потоки получают управление  
толью после снятия замка потоюм, вызвавшим метод **notif** у ().  
**• notifyAllO** Этот метод аналогичен методу **notify** (), но  
прерывает ожидание всех ждущих выполнения условия потоюв.

В следзаощем примере условия используются для оповещения потоюв о  
прибытии новой порции данных (организуется связь производитель -  
потребитель, producer - consumer):

import threading

cv = threading.Condition()  
class Item:

"""Класс-юнтейнер для элементов, юторые будут потребляться  
в потоках'"'"

def init\_(self):

self.\_items = []  
def is\_available(self):  
return len(self.\_items) > 0  
def get(self):  
return seIf.\_items.pop()  
def make(self i):  
self.\_items.append(i)

item = Item()  
def consume():

""'Потребление очередного элемента (с ожиданием его появлег  
cv.acquireO

while not iterais\_available():  
cv.waitO  
it = iteragetO

cv.releaseO  
return it

def consumer():  
while True:  
print consumeQ

def produce(i):

'""Занесение нового элемента в контейнер и оповещение поток

cv.acquireO

iteramake(i)

cv.notifyO

cv.releaseO

pi = threading.Thread(target=consumer, name="tl")  
p 1 .setDaemon(True)

p2 = threading.Thread(target=consumer, name="t2")

p2 .setDaemon(True)

pl.startQ

p2.start0

produceC'ITEMl")

produce("ITEM2")

produce("ITEM3")

produce("ITEM4")

pl.joinQ

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  
заюнчится, так как имеет бесконечный цикл в потоках, а в главном  
потоке - ожидание завершения этих потоков. Еще одна особенность -  
признак демона, установленный с помощью метода **setDaemonO**объекта-потока до его старта.

Очередь

**Сузи RA. Язык программирования Python**

Процесс, показанный в предыдущем примере, имеет значение,  
достойное отдельного модуля. Такой модзль 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,**заведзлощий 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 появлег  
return item.get()

def consumer():  
while True:  
print consumeO

def produce(i):

'""Занесение нового элемента в контейнер и оповещение поток  
iteraput(i)

р1 = threading.Thread(target=consumer, name="tl")  
р 1 .setDaemon(True)

р2 = threading.Thread(target=consumer, name="t2")

p2 .setDaemon(True)

pl.startO

p2.start()

produceC'ITEMl")

produce("ITEM2")

produce("ITEM3")

produce("ITEM4")

pl.joinQ

p2.join()

**Сузи RA. Язык программирования Python**

Следует отметить, что все блокировки спрятаны в реализации очереди,  
поэтому в коде они явным образом не присутствзлот.

Модуль thread

По сравнению с модзием **threading,** модзоть **thread** предоставляет  
низюуровневый доступ к потокам. Многие функции модуля  
**threading,** юторый рассматривался до этого, реализованы на базе  
модзия **thread.** Здесь стоит сделать некоторые замечания по  
применению потоков вообще. Документация по Python предупреждает,  
что использование потоков имеет особенности:

* Исключение **Keyboardinterrupt** (прерывание от  
  клавиатзфы) может быть получено любым из потоков, если в  
  поставке Python нет модзля **signal** (для обработки сигналов).
* Не все встроенные функции, блокированные ожиданием ввода,  
  позволяют другим потокам работать. Правда, основные функции  
  вроде **time . sleep О , select. select** **(),** метод **read()**файловых объектов не блокируют другие потоки.
* Невозможно прервать метод **acquire {),** так как исключение  
  **Keyboardinterrupt** возбуждается толью после возврата из  
  этого метода.
* Нежелательно, чтобы главный поток завершался раньше других  
  потоюв, так как не буцут выполнены необходимые деструкторы и  
  даже части **finally** в операторах **try-finally.** Это связано  
  с тем, что почти все операционные системы завершают  
  приложение, у юторого завершился главный поток.

Визуализация работы потоков

Следующий пример иллюстрирует параллельность выполнения  
потоюв, используя возможности библиотеки графических примитивов  
Т kin ter (она вждит в стандартнзло поставі^^ Python). Несюлью  
потоюв наперегонки увеличивают размеры прямоугольника неюторого  
цвета. Цветом победившего потока окрашивается кнопка Go:

import threading, time, sys

**Сузи RA. Язык программирования Python**

fromTkinter import Тк, Canvas, Button, LEFT, RIGHT, NORMAL, DI:  
global champion

# Задается дистанция, цвет полосок и другие параметры  
distance = 300

colors = ['Red", "Orange", "Yekow","Green","Blue","DarkBlue",'Violet"]  
nrunners = len(colors) # юличество дополнительных потоюв  
positions = [0] \* nrunners # список текущих позиций  
h, h2 = 20, 10 # параметры высоты полосок  
def run(n):

"""Программа бега п-го зліастника (потока)"  
global champion  
while 1:

**# интенсивные вычисления**

for i in range(lOOOO):  
pass

graph\_lock.acquire()

positionsfn] += 1 # передвижение на шаг

if positionsfn] == distance: # если уже финиш

if champion is None: # и чемпион еще не определен,

champion = colorsfn] # 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):  
positionsfi] = 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, positionsfn], n\*h+h2)

**Сузи RA. Язык программирования Python**

**tk.update\_klletasks() # прорисовка изменений  
graph\_lock.release()**

def quit():

'""Вышд из программы"""

tk.quitO

sys.exit(O)

* Прорисовка окна, основы для прямоугольниюв и самих прямо
* кнопок для пуска и выхода  
  tk = Тк()

ік.ііЙе("Соревнование потоков")

с = Canvas(tk, width=distance, height=nrunners\*h, bg="White")  
c.packO

rects = [c.create\_rectangle(0, i\*h, 0, i\*h+h2, fi]l=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, регзлирующий доступ к функции пакета Тк  
  graph\_lock = threading.Lock()
* Цикл проведения соревнований  
  while 1:

go\_b-Config(state=NORMAL), quit\_b-Config(state=NORMAL)  
tk.mainloopO # 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(posillons) < distance\*nrunners:  
update\_positions()  
update\_positions()

**go\_b-Config(bg=champion)** # Кнопка окрашивается в цвет побе/  
**tk. update\_idletasks()**

Примечание:

**Сузи RA. Язык программирования Python**

Эта программа использует неюторые возможности языка Python 2.3  
(встроеннзло функцию sum () и списювые включения), поэтому для ее  
выполнения нужен Python версии не меньше 2.3.

Заключение

Навыки параллельного программирования необждимы 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 несюлько  
потоюв.

Параллельное программирование требует тщательной отработки  
взаимодействия между потоками управления. Неюторые зліастки юда  
необходимо ограждать от одновременного использования двумя  
различными потоками, дабы не нарушить целостность изменяемых  
структур данных или логиід^ работы с внешними ресзрсами. Для  
ограждения з^астюв юда использзлотся замки и семафоры.

Стандартная библиотека Python предоставляет довольно неплохой  
набор возможностей для многопоточного программирования в модзлях  
**threading и thread,** а также неюторые полезные вспомогательные  
модзли (например. **Queue ).**

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 гра4»іческой библиотеки виджетов (Тк).

Обзор графических библиотек

Строить гра4»іческий интерфейс пользователя (GUI, G raphical U ser I  
nterface) для программ на языке Python можно при помощи  
соответствующих библиотек компонентов графичесюго интерфейса  
или, используя кальку с английсюго, библиотек виджетов.

Следующий список далеко не полон, но отражает многообразие  
существующих решений:

**Tkinter** Многоплатформенный пакет имеет хорошее управление  
расположением компонентов. Интерфейс выглядит одинаюво на  
различных платформах (Unix, Windows, Macintosh). Входит в  
стандартнзло поставку РздЬоп. В качестве доід^ентации можно  
использовать руководство "Ап Introduction to Tkinter" ('Введение в  
Tkinter"), написанное Фредриюм Лзтздом: ссылка:

httpy/[www.pythonware.com/Iibraiy/tkinter/introduction/](http://www.pythonware.com/Iibraiy/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: ссылка:  
httpy/www. wxpython.org/

**PyGTK** Набор визуальных юмпонентов 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 вокрзт MFC, поставляется вместе с

оболочкой в пакете win32all; только для Windows.

**pyFLTK** Аналог Xforms, поддержка OpenGL. Имеется для

**Сузи RA. Язык программирования Python**

платформ Windows и Unix. Сайт: ссылка: httpy/pyfltk.sourceforge.net/

* **AWT, JFC, Swing** Поставляется вместе с Jython, а для Jython  
  доступны средства, юторые использует Java. Поддерживает  
  платформу Java.
* **anygui** Независимый от нижележащей платформы пакет для  
  построения графического интерфейса для программ на РзДЬоп.  
  Сайт: ссылка: httpy/anyguLsourceforge.net/
* **PythonCard** Построитель графического интерфейса, сходный по  
  идеологии с HypeiCard/MetaCard. Разработан на базе **wxPython.**Сайт: ссылка: httpy/pythoncard.sourceforge.net/

Список актуальных ссылок на различные гра(}мческие библиотеки,  
доступные из РздЬоп, можно найти по следующему адресу: ссылка:  
http y/phaseit. net/claird/comp. Jang. 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 иж+ или на IVlotif, а те - тоже используют для  
вывода X Window. Кстати, для iVIotif в Python имеются свои привязки.

В лекции будет рассматриваться пакет Tl<inter, который по сути является  
обертюй для Тс]/Т1< - известного графичесюго пакета для сценарного  
языка TcL На примере этого пакета легко из}діить основные принципы  
построения графического интерфейса пользователя.

О графическом интерфейсе

Почти все современные гра4»іческие интерфейсы общего назначения  
строятся по модели WIMP - Window, Icon, Menu, Pointer (окно, иконка,  
меню, указатель). Внутри оюн рисуются элементы графичесюго  
интерфейса, юторые для кратюсти будут называться виджетами (widget  
- ип'учка). Меню могут располагаться в различных частях окна, но их  
поведение достаточно однотипно: они служат для выбора действия из  
набора предопределенных действий. Пользователь графичесюго  
интерфейса "объясняет" юмпьютерной программе требуемые действия с  
помощью указателя. Обычно указателем служит курсор мыши или  
джойстика, однаю есть и другие "указательные" устройства. С помощью

**Сузи RA. Язык программирования Python**

июнок грас}мческий интерфейс приобретает независимость от языка и в  
некоторых случаях позволяет быстрее ориентироваться в интерфейсе.

Основной задачей графичесюго интерфейса является упрощение  
коммзпикации между пользователем и компьютером. Об этом следует  
постоянно помнить при проектировании интерфейса. Применение  
имеющихся в наличии у программиста (или дизайнера) средств при  
создании гра4»ічесюго интерфейса нужно свести до минимз^а,  
выбирая наиболее здобные пользователю виджеты в каждом  
конкретном случае. Кроме того, полезно следовать принципу  
наименьшего удивления: из формы интерфейса должно быть понятно  
его поведение. Плохо продз^анный 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). Однаю если приложение - скорее констрзжтор, из  
юторого пользователь может строить нужные ему решения, типичным  
действием является именно построение решения. Определить  
типичные действия не всегда легю, поэтому юмпромиссом может быть  
гибрид, в ютором есть "мастера" и хорошие возможности для  
собственных построений. Тем не менее, гра4мческий интерфейс не  
является самым эффективным интерфейсом во всех слзліаях. Для многих  
предметных областей решение проще выразить с помощью деклараций  
на неютором формальном языке или алгоритма на сценарном языке.

Основы Тк

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, гра(}»іческое приложение  
выполняет бесюнечный 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 гра(}»іческого интерфейса  
(виджетах) и обрабатываются прикрепленными к этим виджетам

**Сузи RA. Язык программирования Python**

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 дрзтих виджетов) ведают так  
называемые менеджеры расположения. Виджет устанавливается на  
место по правилам менеджера расположения. Эти правила могут  
определять не толью юординаты виджета, но и его размеры. В Тк  
имеются три типа менеджеров расположения: простой упаювщик  
(pack), сетка (grid) и произвольное расположение (place).

Но этого для работы гра(}»ічесюй программы недостаточно. Дело в том,  
что неюторые виджеты в гра4мчесюй 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 Тк используются переменные, через  
юторые виджеты и передают друг дрзту параметры.

Классы виджетов

Для построения графичесюго интерфейса в библиотеке Тк отобраны  
следующие классы виджетов (в алфавитном порядке):

* **Button** (Кнопка) Простая кнопка для вызова неюторых  
  действий (выполнения определенной шманды).
* **Canvas** (Рисзтзок) Основа для вывода графических примитивов.
* **Checkbutton** (Флажок) Кнопка, юторая умеет переключаться  
  между двз^я состояниями при нажатии на нее.
* **Entry** (Поле ввода) Горизонтальное поле, в юторое можно  
  ввести строку текста.
* **Frame** (Рамка) Виджет, юторый содержит в себе другие  
  визуальные шмпоненты.
* **Label** (Надпись) Виджет может показывать текст или  
  графичесюе изображение.

**Язык программирования Python**

**Listbox** (Список) Прямозтольная рамка со списком, из юторого  
пользователь может выделить один или несколью 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.

**Тор level** (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 клавиатзрой и мышью, и  
происходящие на 'VeppHTopHn" того или иного виджета. В Тк события  
описываются в виде текстовой строки - шаблона события, состоящего  
из трех элементов (модификаторы, тип события и детализация  
события).**

Содержание события  
Активизация окна  
Нажатие кнопки мыши  
Отжатие кнопки мыши  
Деактивация окна  
Закрытие окна

**Сузи Р.А.**

Тип события  
**Activate  
ButtonPress  
ButtonRelease  
Deactivate**Destroy  
**Enter  
Focusin  
FocusOut**KeyPress  
**KeyRelease  
Leave  
Motion  
MouseWheel  
Reparent  
Visibility**

Вхождение курсора в пределы виджета  
Полздіение фоід^са окном  
Потеря фокуса окном  
Нажатие клавиши на клавиатуре  
Отжатие клавиши на клавиатз^е  
Выход курсора за пределы виджета  
Движение мыши в пределах виджета  
Прокрутка юлесика мыши  
Изменение родителя окна  
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-l>"** - двойной щелчок  
мышью (левой кнопкой) с нажатой кнопюй Shift. В качестве  
моди(}»ікаторов могут быть использованы следующие (список  
неполный):

Control, Shift, Lock,

Buttonl-Buttons или B1-B5,

Meta, Alt, Double, Triple.

Просто символ обозначает событие - нажатие клавиши. Например, "к"  
- тоже, что **"<ffeyPress-к>".** Для неалфавитно-цифровых клавиш  
есть специальные названия:

**Сузи RA. Язык программирования Python**

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, FI, F2, F3, F4, F5, F6, F7,  
F8, F9, FIO, **Fll,** F12, Num\_Lock, Scroll\_Lock, space, less

Здесь **<space>** обозначает пробел, a **<less>** - знак меньше. **<Lef t>,  
<Right>, <Up>, <Down>** - стрелки. **<Prior>, <Next>** - это PageUp  
и PageDown. Остальные клавиши более или менее соответствзлот  
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 графичесюго приложения. Например,  
известный всем Ctrl-Alt-DeL

Следующая программа позволяет печатать направляемые виджету  
события, в частности - кеуззлд а также анализировать, как различные  
клавиши можно представить в шаблоне события:

from Tkinter import \*

tk = Тк() # 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 юнца текста  
for к in dir(event): # цикл по атрибутам события

if к[0] != # берутся толью неслужебные атрибуты

# готовится описание атрибута события  
еѵ = "%15s: %s\n" % (к, repr(getattr(event, к)))  
txt.insert(END, еѵ) # добавляется в юнец текста

* привязывается виджету txt фзлікция event\_info для обработки
* соответствзлощих шаблону <KeyPress>  
  txt.bind("<KeyPress>", event\_info)

**Сузи RA. Язык программирования Python**

tk.mainloopO # главный цикл обработки событий

При нажатии клавиши Esc в окне можно увидеть примерно следующее:

char; '\х1Ь'

delta: 9  
height: О  
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-юд клавиатуры).
* **keysym** Символичесюе имя клавиши.
* **serial** Серийный номер события. Увеличивается по мере  
  возникновения событий.
* **time** Время возникновения события. Все время увеличивается.
* **wi dge t** Виджет, в ютором возникло событие.
* X, у Координаты зжазателя 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

**Сузи RA. Язык программирования Python**

виджет, юторый 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 Тк существуют и дрзтие возможности  
управления событиями, юторые можно изучить по документации.

Создание и конфигурирование виджета

Создание виджета происждит вызовом конструктора соответствзтощего  
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  
несколью).

Каждый виджет имеет свойства, которые можно устанавливать  
(конфигурировать) с помоіцью методов **configO** (или  
**configure** О ) и читать с помощью методов, подобных методам  
работы со словарями. Ниже приведен возможный синтаксис для  
работы со свойствами:

widget.con^(option=value,...)

widget[ "option"] = value  
value = widget[ "option"]  
widget. keys()

В случае, югда 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 юнфигурацию 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 напрямзто

**Сузи RA. Язык программирования Python**

связана с полем ввода. Этот пример нарочно использует очень много  
свойств, чтобы продемонстрировать возможности по  
кон4мгурированию:

from Tkinter import \*  
tk = Тк()  
tv = StringVarO  
Label(tk,

textvariable=tv,

relief="groove",

bordeiAvidth=3,

font=("Courier", 20, ’bold"),

jus tify= LEFT,

width=50,

padx=10,

pady=20,

takefocus=False,

).pack()

Entiy(tk,

textvariable^tv,

takefocus=Tme,

).pack()

tv.set("123")

tk.mainloopO

В результате на экране можно увидеть:

123

**123**

Виджеты юнфигурируются 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), **take focus** (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 фоіуса:

from Tkinter import \*  
tk = Tk()  
tv = StringVarO  
Entry(tk,

textvariable=tv,

takefocus=Tme,

borderwidth=10,

).pack()

mycolorl = "#%02X%02X%02X" % (200, 200, 20)

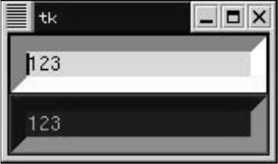
Entry(tk,

* **fg, текст виджета  
  # bg, фон виджета**

**# подсветка при фокусе**

* **подсветка без фокуса**

textvariable^tv,  
takefocus=Tme,  
borderwidth=10,  
foreground=mycolorl,  
background=''#0000FF",  
highlightcolor='green',  
highlightbackground='red',



).pack()  
tv.set("123")  
tk.mainloopO

**Сузи RA. Язык программирования Python**

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 **е,** получить  
значение можно так: **е** . **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 = ScroUedText(tk) # виджет текста с прокрутюй

txt.pack() # виджет размещается

for **X** in range(l, 1024): **#** виджет наполняется текстовым содер  
txtinsert(END, str(2L\*\*x)+"\n")

tk.mainloopO

**11579208923731619S4235709850086879078S326998466S640S640394S7S84007913129639936**

**231584178474632390847141970017375815706539969331281128078915168015826259279872**

**463168356949264781694283940034751631413079938662S622S61S7830336031652518559744**

**926336713898529563388567880069503262826159877325124512315660672063305037119488**

**1852673427797059126777135760139006525652319754650249024631321344126610074238976**

**3705346855594118253554271520278013051304639509300498049262642688253220148477952**

**7410693711188236507108543040556026102609279018600996098525285376506440296955904**

**14821387422376473014217086081112052205218558037201992197050570753012880593911808**

**29642774844752946028434172162224104410437116074403984394101141506025761187823616**

**59285549689505892056868344324448208820874232148807968788202283012051522375647232**

**11857109937901178411373668864889641764174846429761593757640456602410304475129446**

**4**

**23714219875802356822747337729779283528349692859523187515280913204820608950258892**

**8**

**147428439751604713645494675459558567056699385719046375030561826409641217900517785**

**6**

**94856879503209427290989350919117134113398771438092750061123652819282435801035571**

**2**

**18971375900641885458197870183823426822679754287618550012224730563856487160207114**

**24**

**37942751801283770916395740367646853645359508575237100024449461127712974320414228**

**[48**

**75885503602567541832791480735293707290719017150474200048898922255425948640828456**

**96**

**Л**

Теперь следует рассмотреть методы и свойства виджета с  
форматированным текстом более подробно.

Для навигации в тексте в Тк предусмотрены специальные индексы.  
Индексы вроде **1**.**0** и **END** уже встречались - это начало текста (первая  
строка, нулевой символ) и его конец. (В Тк строки нзлиерзлотся с  
единицы, а символы строки - с нуля). Более полный список индексов:

1. **С** Здесь **L** - номер строки, а **С** - 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  
М участка текста.

**SEL\_FIRST, SEL\_LAST** Индексы начала **и** конца выделенного  
текста.

**М** Пользователь может определять свои именованные позиции в  
тексте (аналогично **END, INSERT** или **CURRENT** ). При  
редактировании текста маркеры бyz^yт сдвигаться с заданными для  
них правилами.

Ѳх, у Символ текста, ближайший к точке с координатами х, у.

**Сузи RA. Язык программирования Python**

Следующий пример показывает, как снабдить форматированный текст  
гапертекстовыми возможностями:

from Tkinter import \*  
import urUib  
tk = Tk()

txt = Text(tk, width=64) # поле c текстом  
txt.grid(row=0, column=0, rowspan=2)

addr=Text(tk, background="White", width=64, height=l) # поле аді:  
addr.grid(row=0, column=l)

page=Text(tk, background="White", width=64) # поле c Ьппккодоіѵ  
page.grid(row=l, column=l)

def fetch\_url(event):

click\_point = "@%s,%s" % (event.x, event.y)

trs = txt.tag\_ranges("href ■) # список областей текста, отмеченні

**url = ""**

* определяется, на какой участок пришелся щелчок мыши, и (
* соответствующий ему URL  
  for i in range(0, len(trs), 2):

if txt.compare(trs[i], "<=", click\_point) and \  
txt.compare(ckck\_point, "<=", trs[i+l]):  
url = txt.get(trs[i], trs[i+l])  
html\_doc = urllib.urlopen(url).read()  
addr.deleteC’l.O", END)

addr.insert("1.0", url) # URL помещается в поле адреса  
page.deleteC'l.O", END)

page.insert("1.0", html\_doc) # показывается HTML-документ

textfrags = ["Python main site:", "httpУ/[www.python.org](http://www.python.org)",

"\nJython site:", "httpy/wMw.jython.org",

"VnThat is all!"]  
for frag in textfrags:  
if frag.startswith("http:"):

txt.insert(END, frag, "href) # URL помеідается в текст с метюі  
else:

txt.insert(END, frag) # фрагмент помещается в текст

# ссылки отмечаются подчеркиванием и синим цветом

**Сузи RA. Язык программирования Python**

txt.tag\_config("href, foreground='Blue", underline=l)

* при щелчке мыши на тексте, отмеченном как "href,
* следует вызывать fetch\_url()  
  txt.tag\_bind("href, "<1>", fetch\_url)

tk.mainloopO # запускается цикл событий

В результате (после нажатия на гаперссылку) можно увидеть примерно  
следующее:

**^http //vw python ocg**

**Python жат site, python orq**

**Jython aite http //wv python ory  
That 1» alll**

<!DOCTYPE ht»l PUBLIC •-//•3C//IITD HIML 4 01 TransitionalZ/EH"  
•http //vwv w3. org/ra/html4/Ioooe .dtd" >  
<?;a»l'8tyle9heet href»"./C8«/ht2htJil cee\* type»"text/cee"?>  
<htJil>

<!— THIS PACE IS AUTOMATICALLY GENERATED DO NOT EDIT. — >

<!— Thu Aug 12 10:20 00 20CM —>

<!— USING HT2KIML 2.0 —>

<1— SEE htt**4**> //ht2htnl 8f net —>

<1— User-epecifled header»

Title Python Ргодтамжіпд Language

<head>

<titie>Pythen Prograiuiing Language</title>

<meta http-equiv»“Content-Type" content»\*text/htnl; chareet^ieo-  
8859-1" **/>**

<neta na«ie«"generator" content»'HT2ffIKL/2.0" />

<meta naMe>"keyvorda"

content»"Python ргодгажжіпд language object oriented veb free  
eource" />

<meta name»"de8cription" content»"Home page for Python> an  
interpreted, interactive, obgect-oriented. extensible  
prograjuiing language It provides an extraordinary coabination

Для придания неюторым 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 фзлзкции **fetch\_url** **()** нужно в начале определить, на какой  
именно }ліасток 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 О получается текстовое значение найденного интервала. Найдя  
URL, его в поле записываются адреса, и получается HTML-юд,

**Сузи Р.А.**

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, имеющижя в Тк. В трех рамках  
можно применить различные менеджеры: pack, grid и place:

from Tkinter import \*  
tk = Tk()

* Создаем три рамки  
  frames = {}

**b={}**

for **&1** in 1, 2, 3:

**f=** Frame(tk, width=100, height=200, bg="White")  
f.pack(side=LEFT, ffll=BOTH)  
frames[fri] = f

for bn in 1, 2, 3, 4: # Создаются кнопки для каждой из рамок  
b[fii, bn] = Button(frames[fn], text='%s.%s" % (fri, bn))

* Первая рамка:
* Сначала две кнопки прикрепляются к левому краю  
  Ь[1, l].pack(side=LEFT, Ш1=ВОТН, expand=l)

Ь[1, 2].pack(side=LEFT, Ш1=ВОТН, expand=l)

* Еще две - к нижнему

Ь[1, 3].pack(side=BOTTOM, 611=Y)

Ь[1, 4].pack(side=BOTTOM, 611=ВОТН)

* Вторая рамка:
* Две кнопки свер^^

Ь[2, l].grid(row=0, column=0, sticky=NW+SE)  
b[2, 2].grid(row=0, column=l, sticky=NW+SE)

**Сузи RA. Язык программирования Python**

* и одна на две юлонки в низу

Ь[2, 3].grid(row=l, column=0, columnspan=2, sticky=NW+SE)

* Третья рамка:
* Кнопки высотой и шириной в 40% рамки, якорь в левом Bef
* Координаты яюря 1/10 от ширины и высоты рамки

Ь[3, 1].р1асе(ге1х=0.1, ге1у=0.1, relwidth=0.4, relheight=0.4, anchor^

* Кнопка строго по центру. Якорь в центре кнопки

Ь[3, 2].р1асе(ге1х=0.5, ге1у=0.5, relwidth=0.4, relheight=0.4, anchor^

* Яюрь по центру кнопки. Координаты яюря 9/10 от ширины  
  Ь[3, 3].р1асе(ге1х=0.9, ге1у=0.9, relwidth=0.4, relheight=0.4, anchor=

**2.1**

**2.2**

**1.1**

**1.2**

**3.^**

**— 3.2**

tk.mainloopO  
Результат следующий:

**2.3**

**1.4**

**3.3**

**1.3**

Менеджер pack просто заполняет внутреннее пространство на  
основании предпочтения того или иного края, необходимости  
заполнить все измерение. В неюторых слздіаях ему приходится менять  
размеры подчиненных виджетов. Этот менеджер стоит использовать  
толью 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 юординаты в одной из ячеек сетки ( row - строка, **column** -  
столбец), а также, если нужно, столью последзлощих ячеек (в строках

**Сузи RA. Язык программирования Python**

ниже или в столбцах правее) сюлько он может занять (свойства  
**rowspan или columnspan** ). Это самый габкий 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 Ткіптет можно выводить не только текст, примитивные  
формы (с помощью виджета Canvas), но и растровые изображения.  
Следующий пример демонстрирует вывод июнки с растровым  
изображением (для этого примера нужно предварительно установить  
пакет Python Imaging Library PIL):

import Tkinter, Image, ImageTk

FILENAME = "lena.jpg" # файл c 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.packO

Tkinter.LabeI(tk, text=FILENAME).pack()  
tk.mainloopO

В результате пол}лгается:



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 юординатам (О, 0) в рисунке. В данном  
примере это северо-западный угол (NW - North-West). Другие  
возможности: п (север), w (запад), s (юг), е (восток), пе, sw, se и с (центр).

В следующем примере показаны графические примитивы, которые  
можно использовать на рисзтзке (приведенные комментарии объясняют  
свойства графических объектов внутри виджета-рисунка):

from Tkinter import \*  
tk = Тк()

# Рисунок 300x300 пикселей, фон - белый  
с = Canvas(tk, width=300, height=300, bg="white")

c.create\_arc((5, 5, 50, 50), style=PIESLICE) **#** Сектор **("iq^cok пи[**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 стрелюй на юнце  
  c.create\_line([(5, 55), (55, 55), (30, 95)], arrow=LAST)
* Кривая (сглаженная ломаная)  
  c.create\_line([(105, 55), (155, 55), (130, 95)], smooth=l)
* Многоуголъник зеленого цвета

c.create\_poiygon([(205, 55), (255, 55), (230, 95)], M="green")

* Овал

c.create\_ovai((5, 105, 50, **120),)**

**Сузи RA. Язык программирования Python**

* Прямоугольник красного цвета с большой серой границей  
  c.create\_rectangle((105, 105, 150, 130), fill="red",

out]ine="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!",  
justi^CENTER, anchor="se")  
c.create\_oval((205, 205, 206, 206), outline="red")

c.packO

tk.mainloopO



В результате работы этой программы на экране появится окно:

**Hello,**

**Hello**

**Hello, my friend!\_  
my friend!**

**Сузи RA. Язык программирования Python**

Следует заметить, что методы **create\_\*** создают объекты, свойства  
юторых 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 Yekow Green LightBlue Blue Violet".split()

**R= 10**

tk = Tk()

c = Canvas(tk, bg="White", width="4i", height=300, reke^SUNKEN  
c.pack(expand=l, 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), йП="В1аск")

c.tag\_bind(oval, "<1>", change\_bak)

tk.mainloopO

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 слз^гайным образом методом  
**itemconf igure** (). Тег **CURRENT** в Tkinter использован для указания  
объекта, юторый 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 ютором и будут  
демонстрироваться изображения (опять потребуется пакет PIL):

**Сузи RA. Язык программирования Python**

from Tkinter import \*

import Image, ImageTk, tkFileDialog  
global img, imgobj

def showQ:  
global img, imgobj  
# Запрос на имя файла  
filename = tkFileDialog.askopenfilename()

if filename != (): # Если имя файла было задано noabSOBaxeaeN

* рисуется изображение из файла  
  src\_img = Image.open(filename)

img = ImageTk. Photo lmage(src\_img)

* юнфигурируется изображение на рисунке  
  c.itemconfigure(irrgobJ, 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)

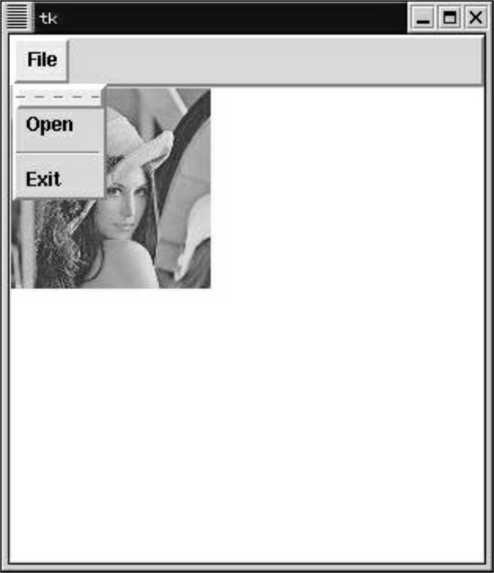
с = Canvas(tk, width=300, height=300, bg="white")

* готовим объект-изображение на рисунке  
  imgobj = c.create\_image(0, 0)

с.раскО

tk.mainloopO

Приложение (с загруженной картинкой) будет выглядеть так:



Стоит отметить, что здесь пришлось применить две глобальные  
переменные. Это не очень жрошо. Сзлцествует другой поджд, югда  
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\_(selQ

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=seif.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(seJQ:

filename = tkFileDialog.askopenfilename()  
if filename != ():

src\_img = Image.open(filename)  
self.img = ImageTk. Photo lmage(src\_img)  
seIf.c.itemconfigure(self.imgobj, image=self.img, anchor="nw")

app = App()  
app.mainloopO

В объекте заключена информация, юторая до этого была глобальной со  
всеми следующими из этого ограничениями. Можно пойти дальше и  
выделить в отдельный метод настройід^ 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 юдировками, в частности, с кодировками для  
кириллицы. Это связано с переходом на Unicode Тс1/Тк. Проблем можно  
избежать, если использовать юдировку UTF-8 в строках, которые  
должны выводиться в виджетах.

Заключение  
**в этой лекции было дано представление о (невизуальном)  
программировании графичесюго интерфейса для РздЬоп на примере  
пакета Tkinter. Программа с гра(}»іческим интерфейсом - событийно-  
управляемая программа, проводящая время в цикле обработки событий.  
События могут быть вызваны функционированием графичесюго  
интерфейса или другими причинами (например, по таймеру). Обычно  
события возникают в виджетах и неюторые из них должны  
обрабатываться приложением. В Tkinter событие представлено**

**Сузи RA. Язык программирования Python**

отдельным объектом, из атрибутов которого можно установить, каково  
было положение указателя (курсора мыши), в каюм виджете произошло  
событие и т.п.

Здесь были рассмотрены классы элементов интерфейса (виджеты), их  
свойства и методы. Виджеты имеют большое количество свойств и  
методов. Нешторые 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 гра4»іческого изображения.

Виджеты располагаются внутри дрзтого виджета (например, рамки) в  
соответствии с набором правил. Этот набор правил реализзлот  
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  
инкапс}лировать информацию, которую в противном слздіае пришлось  
бы делать глобальной.

Нужно отметить, что для построения интерфейса можно использовать  
не только чистый Tkinter. Например, в Python доступны мод}ли  
**ScrolledText и Тіх,** пополняющие набор виджетов. Кроме того,  
можно найти пакеты для специальных виджетов (например, для  
отображения дерева).

Построение графичесюго интерфейса невизуальными способами - не  
такая сложная задача, если использовать Tkinter. Этот пакет входит в  
стандартную поставід^ Python и потому может использоваться почти  
везде, где установлен Pythoa

Ссылки

Список актуальных ссылок на различные графические библиотеки  
можно найти по следующему адресу:

ссылка: httpy/phaseit.net/c]aird/comp.lang.python/p**3**Thon\_GUI.html

**Интеграция Python  
программирования**

**другими**

**языками**

в этой лекции рассматривается встраивание (embedding)  
интерпретатора Python в программу на С, и, наоборот, написание  
модзией для Python на языке С (extending). Кратко описывается  
инструмент для связывания С-библиотек с программами на Python  
(SWIG). Дается обзор связок языка Python с другими языками  
программирования: C++, Java, OCaml, Prolog. Коротю говорится о  
специальном языке для написания модулей расширения РздЬоп - Ругех.

С API

Доступные из языка Python модули расширяются за счет модзией  
расширения (extension modules). Модзии расширения можно писать на  
языке С или C++ и вызывать из программ на Python. В этой лекции речь  
пойдет о реализации Python, называемой CP**3**Thon(Jython, реализация  
Python на платформе Java не будет рассматриваться).

Сама необходимость использования языка С может возникнуть, если  
реализуемый алгоритм, будучи запрограммирован на Python, работает  
медленно. Например, высокопроизводительные операции с массивами  
модзия Numeric (о котором говорилось в одной из предыдущих лекций)  
написаны на языке С. Модзоти расширения позволяют объединить  
эффективность порождаемого компилятором C/C++ юда с удобством и  
гибкостью интерпретатора Python. Необходимые сведения для создания  
модзией расширения для Python даны в исчерпывающем объеме в  
стандартной документации, а именно в докзлиенте "РздЬоп/С API  
Reference Manual" (справочное руюводство по "Python/C АРГ). Здесь  
будут рассмотрены лишь основные принципы построения модзия  
расширения, без детальных подробностей об API. Стоит заметить, что  
возможности Python равно доступны и в C++, просто они выражены в  
С-декларациях, юторые можно использовать в **C++.**

Все необходимые для модуля расширения определения находятся в  
заголовочном файле Pythoah, который должен нахэдится где-то на пути  
заголовочных файлов компилятора C/C++. Следует пользоваться теми  
же версиями библиотек, с которыми был отюмпилирован РзДйоп.

**Сузи Р.А.**

Желательно, и той же марюй компилятора C/C++.

Связь с интерпретатором Python из кода на С осзтцествляется путем  
вызова фзтікций, определенных в интерпретаторе Python. Все функции  
начинаются на Ру или **\_Ру,** потому во избежание конфликтов в модзиях  
расширения не следует определять функций с подобными именами.

Через С API доступны все встроенные возможности языка Python (при  
необходимости, детальнее изучить этот вопрос можно по  
документации):

**высокоуровневый интерфейс интерпретатора (функции и макросы  
Ру\_Маіп(), 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 здалении Python-  
объектов в С/С++-юде.

**обработка исключений ( РуЕгг\* -фзтікции и РуЕхс\_\* -  
константы, например, PyErr\_NoMemory () и**

**РуЕхс\_ІОЕггог)**

**управление процессом и сервисы операционной системы (  
Py\_FatalError(), Py\_Exit(), Py\_AtExit(),**

**(**

**PyOS\_CheckStack (), и другие фзтікции/макросы PyOS\* ),  
импорт модзяей (PyImport\_Import() и другие),  
поддержка сериализации объектов**

**PyMarshal\_WriteObj ectToFile(),**

**и т.п.)**

**аргументов**

**РуАгд VaParseO,**

**(**

**PyMarshal\_ReadObjectFromFile(  
поддержка анализа строки**

**PyArg\_ParseTuple(),**

**PyArg\_ParseTupleAndKeywords(),  
PyArg\_VaParseTupleAndKeywords(),  
PyArg\_UnpackTuple () и Py\_BuildValue () ). C помощью  
этих функций облегчается задача полздіения в коде на С**

**Сузи RA. Язык программирования Python**

**параметров, заданных при вызове функции из Python. Функции  
PyArg\_Parse\* принимают в качестве аргумента строку  
формата полз^енных аргументов,**

* **поддержка протоюлов абстрактных объектов: + Протоюл объекта**

**( PyObject\_Print (), PyObject\_HasAttrString (),  
PyObj ect\_GetAttrString (), PyObj ect\_HasAttr(),  
PyObj ect\_GetAttr 0 , PyOb j ect\_RichCompare (), ...,**

**PyObj ect\_IsInstance(), PyCallable\_Check(),**

**PyOb j ect\_Call (), PyOb j ect\_Dir () и другие). To, что  
должен уметь делать любой объект Python + Протокол числа (  
PyNumber\_Check (), PyNumber\_Add (), ...,**

**PyNumber\_And (), ..., PyNumber\_InPlaceAdd (), ...,**

**PyNumber\_Coerce (), PyNumber\_Int (),...). To, 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 (), ...) +**

**Протоюл отображения (например, словарь является  
отображением) (функции: PyMapping\_Check (),**

**PyMapping\_Length(), PyMapping\_HasKey(),**

**PyMapping\_Keys (), ..., PyMapping\_SetI temString (),  
PyMapping\_GetItemstring () и др.) + Протоюл итератора (  
РуІ ter\_Check (), РуІ ter\_Next () ) + Протоюл буфера (  
PyObj ect\_AsCharBuffer(),**

**PyObj ect\_AsReadBuffer(),**

**PyObj ect\_AsWriteBuf fer(),**

**PyObject\_CheckReadBuffer() )**

* **поддержка встроенных типов данных. Аналогично описанному в  
  предыдущем пункте, но уже для юнкретных встроенных типов  
  данных. Например: + Булевский объект ( PyBool\_Check () -  
  проверка принадлежности типу РуВоо1\_Туре, Py\_False -  
  объект False, Py\_True - объект True,**
* **управление памятью (то есть кучей интерпретатора Python)**

**(функции РуМет МаІІосО, РуМет Realloc (),**

**Сузи RA. Язык программирования Python**

**PyMem\_Free(), PyMem\_New(), PyMem\_Resize(),  
PyMem\_Del()** ). Разумеется, можно применять и средства  
выделения памяти C/C++, однаю, в этом слз^іае 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 С не имеет  
возможности распознать.

• структуры для определения объектов встроенных типов (  
**РуОЬ j ect, PyVarObject** и много других)

Примечание

Под протоюлом здесь понимается набор методов, юторые должен  
поддерживать тот или иной класс для организации операций со своими  
экземплярами. Эти методы доступны не толью из Python (например,  
**1еп(а)** дает длину последовательности), но и из юда на С (

**PySequence Length () ).**

Написание модуля расширения

Если необходимость встроить РзДЬоп 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  
С/С++-библиотеки имеют привязки к Python.

Привязка к Python, хотя и может быть несюлью 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  
архитектзфные изменения, юторые позволят упростить использование  
библиотеки.

**Сузи RA. Язык программирования Python**

в качестве примера можно привести выдержку из исходного юда модзия  
**md5,** юторый реализует фзтікцию для получения тс15-дайджеста.  
Модуль приводится в целях иллюстрации (то есть, с сокращениями).  
Модуль вводит собственный тип данных, **MDSType,** поэтому можно  
увидеть не толью реализацию функций, но и способ описания  
встроенного типа. В рамках этого і^са не изучить все тонюсти  
программирования модулей расширения, главное понять дух этого  
занятия. На юмментарии автора курса лекций заказывает двойной слэш  
//:

// заголовочные файлы  
#include Tythoah"

#include "mdS.h"

// В частности, в заголовочном файле mdS.h есть следзлощие опрі  
// typedef unsigned char \*POINTER;

// typedef unsigned int UINT4;

// typedef struct {

// UINT4 state[4]; **/\*** state (ABCD) **\*/**

**И** UINT4 count[2]; /\* number of bits, modulo 2^64 (Isb first) \*/

// unsigned char buffer!64]; /\* input buffer \*/

//} MD5\_CTX;

// Структура объекта MDStype  
typedef struct {

**MD5\_CTX  
} mdSobject;**

**mdS;**

**/\* the context holder \*/**

PyObject\_HEAD

// Определение типа объекта MDStype  
static PyTypeObject MDStype;

// Макрос проверки типа MDStype

#define is\_mdSobject(v) ((v)->ob\_type == &MDStype)

// Порождение объекта типа MDStype  
static mdSobject \*  
newmdSobject(void)

mdSobject \*md5p;

md5p = PyObject\_New(md5object, &MD5type);  
if (md5p == NULL)

return NULL; // не хватило памяти  
MD5Init(8£md5p->md5); // инициализация  
return md5p;

// Определения методов

// Освобождение памяти из-под объекта  
static void

md5\_dealIoc(md5object \*md5p) { PyObject\_Del(md5p); }  
static PyObject \*

md5\_update(md5object \*self, PyObject \*args)

{

unsigned char \*cp;  
int len;

// разбор строки аргументов. Формат указывает следзлощее:  
// **S#** - один параметр, строка (заданная зжазателем и длиной  
//: - разделитель  
// update - название метода  
if (!PyArg\_ParseTuple(args, "s#:update", &ср, &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\

**Сузи RA. Язык программирования Python**

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];

**I\*** make a temporary copy, and perform the final \*/  
mdContext = self->md5;

MD5Final(aDigest, &mdContext);

// реззльтат возвраіщется в виде строки

return PyString\_FromStringAndSize((char \*)aDigest, 16);

}

// и строка документации

PyDoc\_STRVAR(digest\_doc, "digest() -> string\n\...");

static PyObject \*  
md5\_hexd^est(md5object \*self)

{

// Реализация метода на С

}

PyDoc\_STRVAR(hexdigest\_doc, "hexd^est() -> stringVn...");

// Здесь было определение метода сорз^)  
// Методы объекта в сборе.

// Для каждого метода указывается название, имя метода на С  
// (с приведением к типу PyCFunction), способ передачи аргзщені  
// METH\_VARARGS (переменное кол-во) или METH\_NOARGS (і  
// В юнце массива - метка ошнчания списка аргументов.

**Сузи RA. Язык программирования Python**

static PyMethodDef md5\_methods[] = {

{"update", (PyCFunction)md5\_update, METH\_VARARGS, update,  
{"digest", (PyCFunction)md5\_digest, METH\_NOARGS, digest\_do  
{"hexdigest", (PyCFunction)md5\_hexdigest, METH\_NOARGS, hexdigi  
{"copy", (PyGFunction)md5\_copy, METH\_NOARGS, copy\_do  
{NULL, NULL} /\* sentinel \*/

};

// Атрибуты **md5-o6beKTa** обслуживает эта функция, реализуя мет

// getattr.

static PyObject \*

md5\_getattr(md5object \*self, char \*name)

{

// атрибут-данное d^est\_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 модуаю mdS  
PyDoc\_STRVAR(module\_doc, 'This module implements

// Строка документации к классу mdS

PyDoc\_STRVAR(md5type\_doc, "An mdS represents the object...");

// Структура для объекта MDStype с описаниями для интерпрета'  
static PyTypeObject MDStype = {

PyObject\_HEAD\_lNlT(NULL)  
**о,** /\*ob\_size\*/

"mdS.mdS", /\*tp\_name\*/  
sizeof(md5object), /\*tp\_size\*/

0, /\*tp\_itemsize\*/

**I\*** methods \*/

(destructor)md5\_deaДoc, /\*tp\_dealloc\*/

0, /\*tp\_print\*/

(getattrfunc)md5\_getattr, /\*tp\_getattr\*/

0, /\*tp\_setattr\*/

/\*tp\_compare\*/

**};**

**о,**

**о,**

**о,**

**о,**

**о,**

**о,**

**о,**

**о,**

**о,**

**о,**

**о,**

**о,**

**md5type\_doc,**

/\*tp\_repr\*/

/\*tp\_as\_number\*/

/\* tp\_as\_seq uence\*/  
/\*tp\_as\_mapping\*/  
/\*tp\_hash\*/

/\*tp\_call\*/

/\*tp\_str\*/

/\*tp\_getattro\*/

/\*tp\_setattro\*/

/\*tp\_as\_buffer\*/

/\*tp\_xxx4\*/

/\*tp\_doc\*/  
// Функции модуля md5:

// Функция new() для пол}діения нового объекта типа md5t3^e  
static PyObject \*

MD5\_new(PyObject \*self, PyObject \*args)

{

mdSobject \*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, 'ls#:new", &ср, &len))  
return NULL;

if((md5p = newmdSobjectO) == NULL)  
return NULL;

// Если был задан параметр ср:  
if(cp)

MD5Update(&md5p->md5, ср, len);

return (PyObject \*)md5p;

}  
// Строка документации для new()

PyDoc\_STRVAR(new\_doc, "new([arg]) -> md5 object

// Список фзтікций, которые данный модуль экспортирует  
static PyMethodDef md5\_ftmctions[] = {

{"new", (PyCFunction)MD5\_new, METH\_VARARGS, ne

{"md5", (PyCFunction)MD5\_new, METH\_VARARGS, ne

{NULL, NULL} /\* Sentinel\*/

};

// Следует заметить, что mdS - то же самое, что new. Эта функция  
// обратной совместимости со старым модзлем mdS

// Инициализация модуля

PyMODINIT\_FUNC

initmd5(void)

{

PyObject \*m, \*d;

MD5type.ob\_type = &PyType\_Type;

// Инициализируется модуль

m = Py\_InitModuIe3("md5", md5\_iinctions, module\_doc);

// Получается словарь с именами модуля  
d = PyModule\_GetDict(m);

//Добавляется атрибут MDSType (тип тб5-объекта) ксловар  
PyDict\_SetItemString(d, "MDSType", (PyObject \*)&MD5type);

// Добавляется целая константа digest\_size к модулю  
PyModule\_AddIntConstant(m, "digest\_size", 16);

}

На основе этого примера можно строить собственные модули  
расширения, ознаюмившись с документацией по С/АРІ и документом  
'Extending and Embedding" ('Расширение и встраивание") из стандартной  
поставки РзДЬоп. Перед тем, как приступать к созданию своего модуля,  
следует з^едиться, что это целесообразно: подходящего модзля еще не  
создано и реализация в виде чистого РзДЬоп неэффективна. Если создан

**Сузи RA. Язык программирования Python**

действительно полезный модуль, его можно предложить для включения  
в поставку РзтЬоп. Для этого нужно просто связаться с кем-нибудь из  
разработчиков по электронной почте или предложить модуль в виде  
"патча" через ссылка: httpy/sourceforge.net.

Пример встраивания интерпретатора в программу на С

Интерпретатор Python может быть встроен в программу на С с  
использованием С API. Это л}ліше всего демонстрирует уже  
работающий пример:

/\* File : demo.с \*/

/\* Пример встраивания интерпретатора Python в другую програ\  
#include Tythoah"

main(int argc, char \*\*argv)

{

/\* Передает argv[0] интерпретатору Python \*/  
Py\_SetProgramName(argv[0 ]);

/\* Инициализация интерпретатора \*/

Py\_Initialize();

**/>к**

**I\*** Выполнение операторов Python (как бы модуль main ) \*/

PyRm\_SirnpleString("import time\n");

PyRun\_SimpleString("print tinie.localtirne(tirne.time())\n'');

**/>K**

**I\*** Завершение работы интерпретатора \*/  
Py\_Finalize();

}

Компиляция этого примера с помощью компилятора gcc может быть  
выполнена, например, так:

ѵег="2.3"

gcc -fpic demo.с -DHAVE\_CONFIG\_H -Jm-lpython${ver} \

-Ipthread -lutil-ldl\

-I/usr/local/include/python${ver} \

-L/usr/local/lib/python${ver}/config \

**-Wl,-E\**

**-0** demo

Здесь следует отметить следующие моменты:

* программу необждимо юмпилировать вместе с библиотекой  
  libpython соответствующей версии (для этого используется опция -  
  **1,** за которой следует имя библиотеки) и еще с библиотеками,  
  которые требуются для Python: **libpthread, libm, libutil** и  
  т.п.)
* опция **pic** порождает юд, не зависящий от позиции, что  
  позволяет в дальнейшем динамически юмпоновать юд
* обычно требуется явно указать каталог, в ютором лежит  
  заголовочный файл Pythoah (в gcc это делается опцией -1 )
* чтобы получившийся исполняемый файл мог юрректно  
  предоставлять имена для динамически загружаемых модулей,  
  требуется передать юмпоновщику опцию -Е: это можно сделать  
  из **дсс с** помощью опции **-W1,** **-Е.** (В противном случае, модзль  
  **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 упаювщик  
и генератор интерфейсов) - это программное средство, сильно  
упрощающее (во многих сл}ліаях - автоматизирующее) использование  
библиотек, написанных на С и C++, а также на других языках

**Сузи RA. Язык программирования Python**

программирования, в том числе (не в последнюю очередь!) на РзДЬоп.  
Нужно отметить, что SWIG обеспечивает достаточно полнзло  
поддержку практически всех возможностей C++, включая  
предобработід/, классы, указатели, наследование и даже шаблоны **C++.**Последнее очень важно, если необходимо создать интерфейс к  
библиотеке шаблонов.

Пользоваться SWIG достаточно просто, если з^еть применять  
компилятор и компоновщик (что в любом слз^ае требуется при  
программировании на C/C++).

Простой пример использования SWIG

Предположим, что есть программа на С, реализующая неюторзло  
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\*)(caIloc(256, sizeofi(int)));  
**if(freq !=NULL)**for (ptr = s; \*ptr; ptr++)  
freq[\*ptr] += 1;  
return freq;

}

Для того чтобы можно было воспользоваться этой функцией из РзДЬоп,  
нужно написать интерфейсный файл (расширение .1) примерно  
следующего содержания:

/\* File : freq.i \*/

%rrKDdule freq

%typemap(out) int \* {

int i;

$result = PyTuple\_New(256);  
for(i=0; i<256; i++)

PyTuple\_SetItem($result, i, PyLong\_FromLong($l[i]));  
free($l);

}

extern int \* frequency(char s[]);

Интерфейсные файлы содержат инструкции самого SWIG и фрагменты  
С/С++-юда, 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 (подразз^евая, что используется компилятор gcc), создание  
модзия расширения может быть выполнено примерно так:

swig -python freq.i

gcc -с -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.** РУ, которые вместе и дают доступ к требуемой функции:

»> import freq

»> freq.frequency("ABCDEF")[60:75]

**(OL, OL, OL, OL, OL, IL, IL, IL, IL, IL, IL, OL, OL, OL, OL)**

Помимо этого, можно посмотреть на содержимое файла freq\_wrap.c,  
юторый 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 **\*argl;**

int ^result;

if(!PyArg\_ParseTuple(args,(char \*)"s:frequency",&argl)) goto fail;  
result = (int \*)frequency(argl);

{

int i;

resukobj = 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 зпражнения, предлагается сопоставить это определение с  
файлом freq.i и понять, что происходит внутри функции  
**\_wrap\_f requency** **().** Подсказка: можно посмотреть еще раз  
комментарии к С-юду 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 **Creg. frequency** **()** имеет утечки памяти,  
выполняемый процесс очень быстро займет всю имеющуюся память.

Интеграция Python и других систем программирования

Язык программирования Python является сценарным языюм, а значит

**Сузи RA. Язык программирования Python**

его основное назначение - интеграция в единую систему разнородных  
программных компонентов. Выше рассматривалась (низкоуровневая)  
интеграция с С/С++-приложениями. Нужно заметить, что в  
большинстве случаев достаточно интеграции с использованием  
протокола. Например, интегрируемые приложения могут общаться  
через XML-RPC, SOAP, CORBA, СОМ, .NET и т.п. В случаях, югда  
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: ссылка:

http y/www. python. org/moin/IntegratingPythonWithO therLanguages

Java

Документация no Jython (это реализация Python на Java-платформе)  
отмечает, что Jython обладает следзтощими неоспоримыми  
преимзтцествами над другими языками, использзтощими Java-байт-юд:

* Jython-юд динамически компилирует байт-коды Java, хотя  
  возможна и статическая юмпиляция, 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 Рздйоп - 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 неюторых версиях Jython нельзя наследовать классы от  
нескольких Java-классов (в то же время, множественное наследование  
поддерживается для Python-классов).

Следующий пример (файл lines.py) показывает полную интеграцию Java-  
классов с интерпретатором Python:

**Сузи Р.А.**

# Импортируются модули из Java

from java.lang import System  
from java.awt import \*

# A 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.getSizeQ.width, self.getSize().height  
labeLsetText("%s x %s" % (X, Y))  
for i in range(lOO):

xl, yl = randomrandint(l, X), randorarandint(l, Y)  
x2, **yl** = randomrandint(l, X), randorarandint(l, Y)  
g.drawLine(xl, yl, x2, **yl)**

* Метки, кнопки и т.п.

panel = Panel(layout=BorderLayout())  
label = Label("Size", LabeLRIGHI)  
paneLadd(label, "North")

button = Button("QUIT', actionPerformed=lambda e: Systemexit(O))  
paneLadd(button, "South")  
lines = Lines()  
paneLadd(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,** юторая входит в комплект Jython. Из  
командной строки это можно сделать примерно так:

jythonc -d -с -j Ins.jar lines.ру

Для запуска приложения достаточно запустить **lines** **из** юмандной  
строки:

**Сузи Р.А.**

java -classpath "SCLASSPATH" lines

в переменной **$CLASSPATH** должны быть пути к архивам Ins . j аг и  
**jython. jar.**

Prolog

Для тех, кто хочет использовать Prolog из Python, существует несюлько  
возможностей:

* Версия GNU Prolog (сайт: ссылка: httpy/gprolog.sourceforge.net)  
  интегрируется с РздЬоп посредством пакета bedevere (сайт: ссылка:  
  httpy/bedevere.sourceforge.net)
* Имеется пакет PyLog (ссылка:

httpy/[www.gocept.com/angebot/opensource/Pytog](http://www.gocept.com/angebot/opensource/Pytog)) для работы с SWI-  
Prolog (ссылка: httpy/[www.swi-prolog.org](http://www.swi-prolog.org)) из Python

* Можно использовать пакет **pylog** (доступен с сайта: ссылка:  
  httpy/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 модзия **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 queiy in queries:  
**printquery**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: mortaJ(Democritus)

Разумеется, это не "настояіций" Prolog, но с помогцью 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 1NR1A, Франция. Важной особенностью OCaml  
является то, что его юмпилятор порождает исполняемый юд, по  
быстродействию сравнимый с С, 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 реализуюгцая аналог С API для OCamL Таким образом, в  
программах на OCaml могут использоваться модули языка Python, в них  
даже может быть встроен интерпретатор Pythoa Для Python имеется  
большое множество адаптированных С-библиотек, это дает  
возможность пользователям OCaml применять в разработке  
комбинированное преимугцество Python и OCamL Минусом является  
только необходимость знать фзлгкции Python/C API, имена юторого

**Сузи Р.А.**

использованы для связи ОСатІи РздЬоп.

Следующий пример (из Pycaml) показывает программу для OCaml,  
юторая 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 (pynuU (), cd, pystring\_fromstring "CamlClass");  
let cmx = pymethod\_new (foo\_bar\_print,(pyndl ()),cx);;  
let \_ = pydict\_setitemstring (cd, "CamlMethod", cmx);;  
let \_ = pydict\_setitemstring (pymodule\_getdict mx, "CamlClass", cx);;  
let\_ = pydict\_setitemstring (sd, "CamlModule", mx);;  
let \_ = p**3**Tun\_simplestring  
("from CamlModule import CamlClass\n" л  
"x = CamlClass()\n" л  
"for i in range(100000):\n" л  
" x.CamlMethod(l,2,3,4)\n"^

"print ’Done'\n")

Pyrex

Для написания модулей расширения можно использовать специальный  
язык - Ругех - юторый совмещает синтаксис Python и типы данных С.  
Компилятор Ругех написан на Python и превращает исждный файл  
(например, primes.рух) в файл на С - готовый для компиляции модуль  
расширения. Язык Рзтех заботится об управлении памятью, удаляя  
после себя ставшие ненужными объекты. Пример файла из  
документации к Ругех (для вычисления простых чисел):

def primes(int kmax):

cdef int n, k, i  
cdef int p[1000]  
result = []  
if kmax > 1000:

kmax = 1000  
к = 0  
n = 2

while к < kmax:  
i=0

while i < к and n % p[i] <> 0:  
**i=i+ 1  
ifi==** k:  
p[k] = n  
k = k+ 1  
result.append(n)  
n = n + 1  
return result

В результате применения юмпилятора Pyrex, нехитрой компиляции и  
компоновки (с помощью GCC):

рзтехс primes, рух

gcc primes.с -с -fPIC -I /usr/local/include/python2.3  
gcc -shared primes.о -о 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 С-библиотеки, именно поэтому  
он, как и SWIG, может служить для построения оберток С-библиотек  
для Python.

Следует отметить, что для простых операций Pyrex применяет С, а для  
обращения к объектам РздЬоп - вызовы Python/C API. Таким образом,  
объединяется выразительностъ Python и эффективность С. Конечно,  
некоторые вещи в Pyrex не доступны, например, генераторы, списювые  
включения и Unicode, однако, цель Pyrex - создание быстродействующих  
модзлей расширения, и для этого он превосходно подждит. Ознакомится  
с Pyrex можно по документации (юторая, к сожалению, есть пока только  
на английском языке).

**Сузи Р.А.**

Заключение  
**в этой лекции кратю рассматривались основные возможности  
интеграции интерпретатора Python и других систем программирования.  
Базовая реализация языка Python написана на С, поэтому Python имеет  
программный интерфейс Python/C API, юторый позволяет программам  
на C/C++ обращаться к интерпретатору Python, отдельным объектам,  
модзлям и типам данных. Состав Python/C API достаточно обширен,  
поэтому речь шла лишь о неюторых основных его элементах.**

Был рассмотрен процесс написания модуля расширения на С как  
напрямую, так и с использованием генератора интерфейсов SWIG.  
Также кратко говорилось о возможности встраивания интерпретатора  
Python в программу на С 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 логичесюго вывода в Python-программы:  
независимая реализация Prolog-машины, связь с Prolog-  
интерпретатором через конвейер, связь через Python/C API.

Интересный гибрид С и Python представляет из себя язык Ругех. Этот  
язык создан с целью упростить написание модзлей расширения для  
Python на С, и использует структзфы данных С и подобный Python  
синтаксис. Несмотря на некоторые смысловые и синтаксические  
отличия как от С, так и от Python, язык Ругех помогает существенно  
сократить время разработки модзлей расширения, сохранив  
эффективность компилятора С и знакомый синтаксис Рз^оп.

В данной лекции не были представлены другие возможности  
интеграции, например библиотека шаблонов C++ Boost Рзтйоп, юторая  
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

**Сузи RA. Язык программирования Python**

**ОСНОВНЫМ** преимуществом в качестве языка **для** интеграции  
приложений. Из лекции нетрудно заключить, что Python легю  
взаимодействует с другими системами.

Ссылки

Библиотека Boost РздЬоп для C++ ссылка: httpy/[www.boost.org](http://www.boost.org)

Устройство интерпретатора языка Python

в этой лекции сделана попытка пролить свет на внутреннее устройство  
интерпретатора Pythoa Для иллюстрации работы интерпретатора  
рассматриваются отладчик, профайлер и "дизассемблер".

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 (ОР),  
разделители, специальные лексемы для обозначения (изменения)  
отступов (INDENT, DEDENT) и концов строк (NEWLINE), а также  
комментарии (COMMENT). Лексический анализатор доступен через  
модзль **tokenize,** а определения юдов лексем содержатся в модзле  
**token** стандартной библиотеки Pythoa Следующий пример  
показывает лексический анализатор в действии:

import StringlO, token, tokenize

prog\_example = """  
for i in range(lOO): # comment  
**ifi%l==0:\  
printt\*\*2** .stripO

rl = StringlO.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)

A вот что выведет эта программа, разбив на лексемы исходный юд  
примера:

prog\_example:

|  |  |  |
| --- | --- | --- |
| 1 | NAME: | 'for' |
| 1 | NAME: |  |
| 1 | NAME: | 'in' |
| 1 | NAME: | 'range' |
| 50 | OP: |  |
| 2 | NUMBER: | '100 |
| 50 | OP: | ■)' |
| 50 | OP: | **i.i** |
| 52 | COMMENT: | '# con |
| 4 | NEWLINE: | '\n' |
| 5 | INDENT: | **1** • |
| 1 | NAME: | 'if |
| 1 | NAME: | 'i' |
| 50 | OP: |  |
| 2 | NUMBER: | '1' |
| 50 | OP: | **1** » |
| 2 | NUMBER: | 'O' |
| 50 | OP: | **i.i** |
| 1 | NAME: | 'print' |
| 3 | STRING: | **»M,»n** |
| 50 | OP: | **• 1** > |
| 1 | NAME: | 't' |
| 50 | OP: |  |
| 2 | NUMBER: | '2' |
| 6 | DEDENT: | **и** |
| 0 | ENDMARKER: |  |

Фактически получен поток лексем, который может использоваться для  
различных целей. Например, для синтаксичесюго "окрашивания" юда  
на языке Pythoa Словарь **token. tok\_name** позволяет получить  
мнемонические имена для типа лексемы по номеру

Синтаксический анализ

Вторая стадия преобразования исходного текста программы в байт-юд  
интерпретатора состоит в синтаксическом анализе исходного текста.  
Модуль **parser** содержит функции **suite () и ехрг()** для  
построения деревьев синтаксического разбора соответственно для юда

**Сузи RA. Язык программирования Python**

программ и выражений Python. Модуль **symbol** содержит номера  
символов грамматики РзДЬоп, словарь для получения названия символа  
из грамматики Python.

Следующая программа анализирует достаточно простой юд Python (  
prg ) и порождает дерево синтаксического разбора (AST-объект),  
юторый тут же можно превращать в юртеж и красиво выводить  
функцией **pprint.pprintO.** Далее определяется функция для  
превращения номеров символов в их мнемонические обозначения  
(имена) в грамматике:

import pprint, token, parser, ззтпЬоІ

prg = print 2\*2

pprint.pprint(parser.suite(prg).totuple())

def pprint\_ast(ast, level=0):  
if type(ast) == type(O):  
for a in ast:  
pprint\_ast(a, level+1)  
elif type(ast) == typeC""):  
print repr(ast)  
else:

print" "\* level,  
tiy:

print symboLs**3**Tn\_name[ast]  
except:

print "token."+tokeatok\_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'

tokeaSTAR

factor

power

atom

token-NUMBER '2'  
tokeaNEWLINE"  
token-ENDMARKER"

Получение байт-кода

После того как полздіено дерево синтаксического разбора, компилятор  
должен превратить его в байт-юд, подходящий для исполнения  
интерпретатором. В следующей программе проводятся отдельно  
синтаксический анализ, компиляция и выполнение (вычисление) юда  
(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('filenamel.py’)

print evaI(code)

Функция **parser. suite** **()** (или **parser. expr** **()** ) возвращает  
AST-объект (дерево синтаксического анализа), которое методом  
**compile** **()** юмпилируется в Python байт-юд и сохраняется в юдовом  
объекте **code.** Теперь этот юд можно выполнить (или, в случае  
выражения - вычислить) с помощью оператора **ехес** (или функции  
**еѵаі** **()** ).

Здесь необходимо заметить, что недавно в Python появился пакет  
**compiler,** юторый объединяет модули для работы анализа исходного  
юда на Рздйоп и генерации юда. В данной лекции он не

**Сузи RA. Язык программирования Python**

рассматривается, но те, кто жчет глубже иззліить эти процессы, может  
обратиться к документации по РзДЬоп.

Изучение байт-кода

Для из}ліения байт-юда Python-программы можно использовать модуль  
**dis** (сокращение от "дизассемблер"), юторый содержит функции,  
позволяющие увидеть байт-юд в мнемоничесюм виде. Следующий  
пример иллюстрирует эту возможность:

**print 2\*2**

**»> dis.dis(f)**

**О LOAD\_CONST  
3 LOAD\_CONST**

1. **BINARY\_MULT1PLY**
2. **PRINTJTEM**
3. **PRINT\_NEWLINE**
4. **LOAD\_CONST  
   12 RETURN VALUE**

**1(2)**

**1(2)**

**0 (None)**

»> deff():

Определяется функция f (), которая должна вычислить и напечатать  
значение выражения **2\*2.** Функция **dis()** модуля **dis** выводит юд  
функции f () в виде некого "ассемблера", в ютором байт-юд 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 слзліае с юдами  
для микропроцессора, неюторые байт-юды принимают параметры.

Мнемонические имена можно увидеть в списке **dis.opname** (ниже  
печатаются толью задействованные имена):

»> import dis

»> [n for n in dis.opname if n[0] != "<"]

['STOP\_CODE', 'POP\_TOP', 'ROT\_TWO', 'ROT\_THREE', 'DUP\_TC

**Сузи RA. Язык программирования Python**

'NOP', 'UNARY\_POSITIVE, 'UNARY\_NEGAT1VE', 'UNARY\_N01  
'UNARYJNVERT, 'LIST\_APPEND', 'B1NARY\_P0WER', 'BINAR'^  
'BINARY\_DIVIDE', 'BlNARY\_MODULO', 'BINARY\_ADD', 'BINA  
'BINARY\_SUBSCR', 'BINARY\_FLOOR\_DIVIDE', 'BINARY\_TRUE  
'INPLACE\_FLOOR\_DIVIDE', 'INPLACE\_TRUE\_DIVIDE', 'SLICE^  
'SLICE+2', 'SLICE+3', 'STORE\_SLICE+0', 'STORE\_SLICE+l', 'STC  
'STORE\_SLICE+3', 'DELETE\_SLICE+0', 'DELETE\_SLICE+1', 'DEL  
'DELETE\_SLICE+3', 'INPLACE\_ADD', 'INPLACE\_SUBTRACT,'!]  
'INPLACE\_D1V1DE', 'lNPLACE\_MODULO', 'STORE\_SUBSCR', 'E  
'BINARY\_LSHIFT, 'B1NARY\_RSHIFT, 'BINARY\_AND', 'BINARY  
'INPLACE\_POWER, 'GETJTER', 'PRINT\_EXPR', 'PR1NT\_1TEM',  
'PR1NT\_1TEM\_T0', 'PRINT\_NEWLlNE\_TO', 'INPLACE\_LSHIFT,  
'INPLAGE\_AND', 'INPLAGE\_XOR', 'INPLACE\_OR', 'BREAK\_LO  
'RETURN\_VALUE', 'IMPORT\_S'IAR', 'EXEC\_STMT, 'YIELD\_VAL  
'END\_FINALLY', 'BUILD\_CLASS', 'STORE\_NAME', 'DELETE\_N/  
'UNPAGK\_SEQUENCE', 'FORJTER', 'STORE\_AITR', 'DELETE\_y^  
'DELETE\_GLOBAL', 'DUP\_TOPX', 'LOAD\_GONST, 'LOAD\_NAA  
'BUILD\_LIST, 'BUILD\_MAP', 'LOAD\_ATTR', 'COMPARE\_OP', 'IN  
'IMPORT\_FROM', 'JUMP\_FORWARD', 'JUMP\_IF\_FALSE', 'JUMP  
'JUMP\_ABSOLUTE', 'LOAD\_GLOBAL', 'GONTlNUE\_LOOP', 'SF  
'SETUP\_EXCEPT, 'SETUP\_FINALLY', 'LOAD\_FAST, 'STORE\_F/  
'RAISE\_VARARGS', 'GALL\_FUNGTION', 'MAKE\_FUNGTION', 'E  
'MAKE\_GLOSURE', 'LOAD\_CLOSURE', 'LOAD\_DEREF', 'STORE  
'GALL\_FUNGTION\_VAR', 'CALL\_FUNGTION\_KW, 'GALL\_FUN  
'EXTENDED\_ARG']

Легю догадаться, что LOAD означает загрузку значения в стек, STORE -  
выгрузі^^, 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. Оледующий пример показывает программу, юторая подвергается  
отладке, и типичную сессию отладки:

# File тз^.ру  
def ftm(s):  
lst = []

for i in s:

lst.append(ord(i))  
return 1st

Так может выглядеть типичный процесс отладки:

»> import pdb, myfun

»> pdb.runcall(myfun.fun, "ABCDE")

* /examples/myfijn.py(4)fun()

-> **lst=[]**

**(Pdb)n**

* /examples/myfijn.py(5)fun()

-> for **iins:**

**(Pdb)n**

* /examples/myfijn.py(6)fun()

-> lst.append(ord(i))

**(Pdb)1**

1. #!/usr/bin/python
2. # File myfun.py
3. def fun(s):
4. 1st = []
5. for i in s:
6. -> lst.append(ord(i))
7. return 1st  
   [EOF]

(Pdb) p 1st

**[]**

(Pdb) p vars()

{'i':'A', 's':'ABCDE', 'lst':[]}

**(Pdb)n**

* /examples/myfijn.py(5)fun()

-> for **iins:**

(Pdb) p vars()

{'i':'A, 's':'ABCDE', '1st': [65]}

**(Pdb)n**

* /examples/myfijn.py(6)fun()

-> lst.append(ord(i))

**(Pdb)n**

* /examples/myfijn.py(5)fun()

**Сузи RA. Язык программирования Python**

-> for i in s:

(Pdb) p vars()

{T: 'B', 's': 'ABCDE, '1st': [65, 66]}

**(Pdb)r**- Return -

> /examples/myfijn.py(7)fun()->[65, 66, 67, 68, 69]

-> return 1st  
**(Pdb)n**

[65, 66, 67, 68, 69]

**»>**

Интерактивный отладчик вызывается фзтікцией **pdb.runcallO** и на  
его приглашение **(Pdb)** следует вводить команды. В данном примере  
сессии отладки были использованы неюторые из следующих команд: 1  
(печать фрагмента трассируемого юда), п (выполнить все до следующей  
строки), S (сделать следующий шаг, возможно, углубившись в вызов  
метода или функции), р (печать значения), г (выполнить все до  
возврата из текущей фзтікции).

Разумеется, неюторые интерактивные оболочки разработчика для  
Python предоставляют функции отладчика. Кроме того, отладку  
достаточно легю 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 ютором говорилось в  
одной из предыдущих лекций.

Профайлер

Для определения мест в программе, на выполнение юторых злюдит  
значительная часть времени, обычно применяется профайлер.

Модуль profile

Этот модуль позволяет проанализировать работу фзлікции и выдать  
статистику использования процессорного времени на выполнение той  
или иной части алгоритма.

**Сузи RA. Язык программирования Python**

в качестве примера можно рассмотреть профилирование функции для  
поиска строк из списка, наиболее пожжих на даннзло. Для того чтобы  
качественно прос{млировать функцию

dif f lib . **get\_close\_matches** **(),** нужен большой объем данных.  
В файле russiaatxt собрано 160 тысяч слов русского языка. Следзлощая  
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.tx  
рго1і1е.шп(г'ргіпСс105е\_таю1іе5("профайлер")')

При запуске этой программы будет выдано примерно следующее:  
провайдер

трайлер

бройлер

899769 fijnction calls (877642 primitive calls) in 23.620 CPU seconds  
Ordered by: standard name

ncalls tottime регсаД cumtime percaU filenameJineno(function)

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 0.000 | 0.000 | 23.610 |
| 1 | 0.000 | 0.000 | 23.610 |
| 1 | 0.000 | 0.000 | 0.000 |
| 1 | 0.000 | 0.000 | 0.000 |
| 159443 | | 1.420 | 0.000 |
| 2 | 0.000 | 0.000 | 0.000 |
| 2 | 0.000 | 0.000 | 0.000 |
| 324261 | | 2.240 | 0.000 : |
| 28317 1.590 0.000 1, | | | |

1. <string>:l(?)
2. T.py:6(print\_ctose\_matches)

0.000 difflib.py:147( init\_)

0.000 difflib.py210(set\_seqs)

1.420 0.000 difflib.py:222(set\_seql)

0.000 difflib.py248(set\_seq2)

0.000 **difflib.**py293( **chain\_b)**

2.240 0.000 difflib.py:32(\_calculate\_ratio)

) 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(<]ambda>)

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 ргоШе:0(ргіпі\_с1О5е\_таІсЬе5("профайле|  
0 0.000 0.000 profile:0(profiler)**

Здесь **ЮЛОНКИ** таблицы показывают следующие значения: ncalls -  
юличество вызовов (фзтікции), tottime - время выполнения юда  
функции (не включая времени выполнения вызываемых из нее  
функций), регсаіі - то же время, в пересчете на один вызов, cumtime -  
суммарное время выполнения функции (и всех вызываемых из нее  
функций), filename - имя файла, Ііпепо - номер строки в файле, function -  
имя функции (если эти параметры известны).

Из приведенной статистики следует, что наибольшие усилия по  
оптимизации юда необходимо приложить в функциях  
**quick\_ratio** **()** (на нее потрачено 8,69 сеі^пд),  
**get\_close\_matches** **()** (4,93 сеі^тд), затем можно заняться

**real\_quick\_ratіо** **()** (2,95 секунд) и **\_calculate\_ratio** **()**(секунд).

Это лишь самый простой вариант использования профайлера: модзль  
**profile** (и связанный с ним **pstats** ) позволяет пол}діать и  
обрабатывать статистиід^: их применение описано в документации.

Модуль timeit

Предположим, что проводится оптимизация небольшого з^астка юда.  
Необходимо определить, каюй из вариантов юда является наиболее  
быстрым. Это можно сделать с помощью модуля **timeit.**

В следующей программе используется метод **timeit** **()** для измерения  
времени, необюдимого для вычисления небольшого фрагмента юда.  
Измерения проводятся для трех вариантов юда, делающих одно и то же:  
юнкатенирзлощих десять тысяч строк в одну строку. В первом случае  
используется наиболее естественный, "лобовой" прием инкрементной

**Сузи RA. Язык программирования Python**

юнкатенации, во втором - накопление строк в списке с последующим  
объединением в одну строід^, в третьем применяется списювое  
включение, а затем объединение элементов списка в одну строі^^:

from timeit import Timer

t = Timer(

res =""

for к in range(1000000,1010000):  
res += str(k)

print t.timeit(200)

t = Timer(

res = []

for к in range(1000000,1010000):  
res.append(str(k))  
res = ",".join(res)

print t.timeit(200)

t = Timer(

res = ",".join([str(k) for к 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 РзлЬоп рекомендуемым способом конкатенации

**Сузи RA. Язык программирования Python**

большого юличества 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 юнкатенация была оптимизирована и теперь  
имеет даже лучший результат, чем версия со списками (юторая вдобавок  
требует больше памяти). Но чемпионом все-таки является работа со  
списковым включением, поэтому свертывание циклов в списювое  
включение позволяет повысить эффективность кода.

Если требуются более точные реззльтаты, рекомендуется использовать  
метод **repeat** **(п,** **к)** - он позволяет вызывать **timeit(k)** п раз,  
возвращая список из п значений. Необходимо отметить, что на  
результаты может влиять загруженность компьютера, на котором  
проводятся испытания.

Оптимизация

Основная реализация языка Python пока что не имеет оптимизирзлощего  
компилятора, поэтому разговор об оптимизации касается только  
оптимизации кода самим программистом. В любом языке  
программирования имеются свои характерные приемы оптимизации  
юда. Оптимизация (улучшение) кода может происходить в двух  
(зачастую юнкзфирующих) направлениях: сюрость и занимаемая  
память. В условиях достатка оперативной памяти приложения обычно  
оптимизирзлот по сюрости. При оптимизации по времени программы  
для одноразового вычисления следует иметь в виду, что в общее время  
решения задачи вхэдит 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 временнзло  
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 юда, как и программирование - задача

**Сузи RA. Язык программирования Python**

неформальная, поэтому зрение оптимизировать код приходит с опытом.

Если сюрость работы программы при большой длине данных не  
устраивает, следует поискать более эффективный алгоритм. Если же  
более эффективный алгоритм практически нецелесообразен, можно  
попытаться провести оптимизацию юда.

Собственно, в данном примере для модуля **time** **it** уже показан  
практический способ нахождения оптимального юда. Стоит также  
отметить, что с помощью профайлера нужно определить места юда,  
отнимающие наиболыщло 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 сюрости з^іастках юда следует  
избегать вызова большого числа функций.

В неюторых слзліаях работу программы на Python можно усюрить в  
несюлью раз с помощью специального оптимизатора (он не вждит в  
стандартную поставку Python, но свободно распространяется): **psyco.**Для усюрения программы достаточно добавить следзлощие строки в  
начале главного модзля программы:

import psyco

psyco. full()

Правда, неюторые функции не поддаются "юмпиляции" с помощью  
**psyco.** В этих слзліаях будут выданы предупреждения. Посмотрите  
документацию по **psyco** с тем, чтобы звнать ограничения в его  
использовании и способы их преодоления.

Еще одним вариантом усюрения работы приложения является  
переписывание критических участюв алгоритма на языках более  
низюго уровня (C/C++) и использование модзлей расширения из РзДйоп.  
Однаю эта крайняя мера обычно не требуется или модули для задач,  
требующих большей эффективности, уже написаны. Например, для  
работы с растровыми изображениями имеется прекрасная библиотека  
модзлей PIL (Python Imaging Library). Численные расчеты можно

**Сузи Р.А.**

**ВЫПОЛНЯТЬ** с помощью пакета Numeric **и** т.д.

Pychecker

Одним из наиболее интересных инструментов для анализа исходного  
юда Python программы является Pychecker. Как и lint для языка С,  
Pychecker позволяет выявлять слабости в исходном юде на языке РзДЬоп.  
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 у z"  
print spkt(a) + **X**

test(['d'], 'e')

Pychecker выдаст следзлощие предзпреждения:

badcode.py:!: 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:?: 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 мод}иь, юторый далее не применяется,  
то же самое с модулем ге. Кроме того, модуль ге импортирован  
повторно. Другие проблемы с юдом: параметр у не использован; модзиь  
**string** применен как в операторе **import, таки во from-import** ;  
локальная переменная а затеняет глобальную, которая определена в  
третьей строке.

Можно переписать этот пример так, чтобы Pychecker выдавал меньше

**Сузи Р.А.**

предупреждений:

import string

а = "а b с"

def test(x, у):  
al = "xyz"  
print string, split(al) + X

test(['d'], 'e')

Теперь имеется лишь одно предупреждение:  
goodcode.py:4: Parameter (у) not used

Такое тоже бывает. Программист должен лишь з^едиться, 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 иденти4»гкатор (с помощью функции **id()** ),  
представление в виде строки - даже в двух вариантах (функции str ()  
и **герг** О ); можно звнать атрибуты объекта с помощью встроенной  
функции dir О и во многих слзліаях пользоваться атрибутом

**diet** для доступа к словарю имен объекта. Также можно узнать,

сколью других объектов ссылается на данный с помощью функции  
sys . **getref count** **().** Есть еще сборка мусора, юторая применяется  
для освобождения памяти от объектов, которые более не использзаотся,  
но имеют ссылки друг на дрзта (циклические ссылки). Сборюй мусора  
(garbage соДеегіоп) можно зправлять из модуля дс.

Все это подчеркивает тот факт, что объекты в Python сзлцествуют 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: "официальнзло", которую поддерживает описание языка

и многие его реализации, и "неофициальную", юторая использует  
особенности той или иной реализации. С помощью "официальных"  
средств интроспекции можно получить информацию о  
принадлежности объекта тому или иному классу (фзтікция **type** **()** ),  
проверить принадлежность экземпляра классу ( **isinstanceO** ),  
отношение наследования между классами **(issub class** **()** ), а также  
полздіить 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-юду функции, к неюторым  
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 гарантирзлот постоянство этих  
механизмов от версии к версии, а неюторые полезные мод}ии  
используют эти механизмы для своих целей. Например, упомянутый  
ранее усюритель выполнения Python-юда **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 = "abed"

»> dir(s)

[' add ',' class ',' contains ',' delattr ',' doc ',

\_eq ',' ge ',' getattribute ',' getitem ',' gemewargs ',

\_getslice ',' gt ',' hash ',' init ' le ',' len ',

\_lt ',' mod ',' mill ' ne ',' new ',' reduce ',

\_reduce\_ex ',' repr ',' rmod ',' rmul ' setattr ',

\_str ', 'capitalize', 'center', 'count', 'decode',

'encode', 'endswith', 'expandtabs', 'find', 'index', 'isalnum', 'isalpha',  
'isdigit', 'islower', 'isspace', 'istide', 'isupper', 'join', 'ljust',

'lower', 'Istrip', 'replace', 'rfind', 'rindex', 'rjust', 'rstrip',

'split', 'splitlines', 'startswith', 'strip', 'swapease', 'tide',

'translate', 'upper', 'zfifi']

»> id(s)

1075718400  
»> print str(s)  
abed

»> print repr(s)

'abed'

»> type(s)

<type 'str'>

>» isinstanee(s, basestring)

True

»> isinstanee(s, int)

False

>» issubelass(str, basestring)

True

"Неофициальные" ередетва интроепекции в оеновном работают в  
облаети предетавления объектов в ереде интерпретатора. Ниже будет  
раеемотрено, как главная (на наетоящий момент) реализация Python  
может дать информацию об определенной пользователем функции:

»> def f(x, у=0):

... 'Funetion f(x, у)

... global s

... return t + **X + у**

»> f.seeure = 1 # приеваиваетея дополнительный атрибут

»> f.hmo\_name # имя  
'f

»> f.hme\_doe # отрока документации  
'Funetion f(x, у)'

»> f.hino\_de£aults # значения по умолчанию

**(0,)**

»> f.hino\_diet # оловарь атрибутов функции  
{'seoure': 1}

»> со = f.hinc\_code # юдовый объект  
»> со

<code object fat 0х401ес7е0, file "<stdin>", line 1>

Кодовые объекты имеют свои атрибуты:

**Сузи RA. Язык программирования Python**

»> co.co\_code # байт-юд

't\x00\x00|\x00\x00\xl7|\x01\х00\х17Sd\xO l\xOOS'

»> co.co\_argcount # число аргументов  
**2**

»> co.co\_varnames # имена переменных  
('х', 'у')

»> co.co\_consts # константы  
(None,)

»> co.co\_names # локальные имена  
Ct', **'X', 'у')**

»> co.co\_name # имя блока юда (в нашем слзліае - имя фзтікі  
'f

и так далее. Более правильно использовать для полздіения всех этих  
сведений модуль **inspect.**

Модуль inspect

Основное назначение модуля **inspect -** давать приложению  
информацию о модулях, классах, фзлікциях, трассировочных объектах,  
фреймах исполнения и кодовых объектах. Именно модуль **inspect**позволяет заглянуть "на і^^ю" интерпретатора Python.

**Проверяемый тип  
Встроенная функция  
Класс  
Код**

**Описатель данных  
Фрейм  
Функция  
Метод**

**Функция  
inspectisbuikin  
inspectisclass  
inspectiscode  
inspectisdatadescriptor  
inspectis frame  
inspectis function  
inspectismethod**

Модуль имеет фзлікции для проверки принадлежности объектов  
различным типам, с юторыми он работает:

inspectismethoddescriptor Описатель метода

Модуль

**Сузи Р.А.**

**inspectismodule**

**inspect-isroutine**

**inspectistraceback**

функция или метод  
Трассировочный объект

Пример:

»> import inspect

»> inspect.isbuiltin(len)

Тше

»> inspect.isroutine(lambda х: х+1)

Тше

»> inspect.ismethod(".split)

False

»> inspect.isroutine(".split)

Тше

»> inspect.isbuiltin(".split)

Тше

Объект типа модуль появляется в Python-программе благодаря операции  
импорта. Для ползлгения информации о модуле имеются нешторые  
функции, а объект-модуль обладает определенными атрибутами, как  
продемонстрировано ниже:

»> import inspect

»> inspect.ismodule(inspect)

Тше

»> inspect.getmoduleinfo('/usr/local/lib/python2.3/inspect.pyc')  
('inspect', '.рус', 'rb', 2)

»> inspect.getmodulename('/usr/local/lib/p3thon2.3/inspect.pyc')  
'inspect'

»> inspect. name

'inspect'

»> inspect. diet

»> inspect. doc

"Get useful information from live Python objects.\n\nThis module encaf

Интересны неюторые функции, которые предоставляют информацию

**Сузи Р.А.**

об исждном коде объектов:

»> import inspect

»> inspect.getsourcefile(inspect) # имя файла исходного кода  
Vusr/local/lib/python2.3/inspect.py'

»> inspect.getabsfile(inspect) # абсолютный путь к файлу  
Vusr/local/lib/python2.3/inspect.py'

»> print inspect.getfile(inspect) # файл юда модуля  
/usr/local/lib/python2.3/inspect.pyc

»> print inspect.getsource(inspect) # исходный текст модуля (в в

**#-\*-** coding: iso-8859-1-\*-

'"'"Get useiil information from live РздЬоп objects.

»> import smtplib

»> # Комментарий непосредственно перед определением об  
»> inspect.getcomments(smtplib.SMTPException)

'# Exception classes used by this noodule.\n'

»> # Теперь берем строід^ документирования:

»> inspect.getdoc(smtplib.SMTPException)

'Base class for all exceptions raised by this module.'

C помощью модуля **inspect** можно звнать состав аргументов  
некоторой функции с помощью функции **inspect.** **getargspec** **():**

**»>** import inspect

»> def f(x, y=l, z=2):

... return **X** + у + z

»> def g(x, \*v, \*\*z):

... return **X**

»> print inspect.getargspec(Q  
(['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 (\*\*),

**Сузи RA. Язык программирования Python**

после чего - список значений по умолчанию для последних  
позиционных аргзщентов. Первый аргумент-список может содержать  
вложенные списки, отражая структуру аргзщентов:

»> def f((xl,yl), (х2,у2)):

... return 1

»> print inspect.getargspec(Q

([['xl', 'yl'], ['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 ютором был определен объект

'smtplib'

»> inspect.getmodule(s) # можно догадаться о происхождении  
<module 'smtplib' from '/usr/local/lib/python2.3/smtplib.pyc'>

Для визуализации дерева классов может быть полезна функция  
**inspect. getclasstree** **О.** 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.getclasstreef),  
которое представлено списком вложенных списков и юртеж  
В кортеже entry первый элемент - класс, а второй - юртеж с **і**базовыми классами. Иначе entry - вложенный список,  
level - зровень отступов  
for entry in tree:  
if type(entry) is type(()):  
c, bases = entry  
print level \* " ", c. name . \

**Сузи Р.А.**

**Язык программирования Python**

**for b in bases]) + ")"**

**"(" + ", ".join([b.\_rame\_  
elif type(entry) is type([]):  
formattree(entry, level+1)**

V = exceptions. diet .values()

exc\_list = [e for e in V

if inspect.isclass(e) and issubclass(e, Exception)]  
forma ttree(inspect.getclasstree(exc\_iist))

C помощью функции **inspect. currentframe** **()** можно получить  
текущий фрейм исполнения. Атрибуты фрейма исполнения дают  
информацию о блоке юда, исполняющегося в точке вызова метода. При  
вызове функции (и в неюторых других ситуациях) на стек кладется  
соответствующий этому фрейму блок юда. При возврате из функции  
текущим становится фрейм, хранившийся в стеке. Фрейм содержит  
шнтекст выполнения юда: пространства имен и неюторые другие  
данные. Получить эти данные можно через атрибуты фреймового  
объекта:

import inspect

**deffQ:**

fr = inspect.currentframeO  
for a in dir(fr):  
ifa[2] !=

print **a,str(getattr(fr,** a))[:70]

**Ю**

в результате пол}ліается

f\_back : <frame object at 0x812383c>

Lbuildns : {'help': T}^e help() for interactive help, or help(object) for h  
f\_code : <code object f at 0x401d83a0, file "<stdin>", line 11>  
f\_exc\_traceback : None  
f\_exc\_type : None  
f\_exc\_value : None

Lglobals : {'f: <function f at 0x401e0454>,' builtins ': <module '\_

f\_]asti: 68  
LJineno : 16

Llocals : {'a': 'f\_locals', 'fr': <frame object at 0x813c34c>}

Lrestricted : 0  
Ltrace : 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** - юдовый объект  
(в данном слз^ае - байт-юд функции f() ), **f\_lasti** - индекс  
последней выполнявшейся инструкции байт-юда, **f\_trace** - функция  
трассировки для данного фрейма (или **None ), f\_lineno** - текущая  
строка исждного юда, **f\_restricted** - признак выполнения в  
ограничительном режиме.

Получить информацию о стеке интерпретатора можно с помощью  
функции **inspect. stack** О . Она возвращает список юртежей, в  
юторых есть следующие элементы:

(фрейм-объект, имя\_файла, строка\_в\_файле, имя\_функции,

список\_строк\_исходного\_юда, 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  
модзль - **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\_inf** о **()** (это третий  
элемент возвращаемого данной функцией юртежа).

Трассировочный объект имеет следующие атрибуты:

* **tb\_f** rame Фрейм исполнения текущего уровня.
* tb **linenoHtb lasti** Номер строки и инструкции, где было

**Сузи RA. Язык программирования Python**

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 tiy-except"""  
import traceback, sys, string  
print sys.exc\_info()

print" ".join(traceback.format\_exception(\*sys.exc\_info()))

def bad\_iinc2():  
raise StandardError

def bad\_iinc():  
bad\_kinc2()

try:

bad\_kinc()

except:

dbg\_except()

В результате пол}ліается примерно следующее:

(<class exceptions.StandardError at 0x40197290,

<exceptions.StandardError instance at 0x401df2cc>,  
<traceback object at 0x401dcblc>)

Traceback (most recent call last):

File "prl43.py", line 17, in ?  
bad\_iinc()

File "prl43.py", line 14, in bad\_func  
bad\_iinc2()

File "prl43.py", line 11, in bad\_ftmc2  
raise StandardError

StandardError

Функция **sys. exc\_inf** о **()** дает кортеж с информацией о  
возбужденном исключении (класс исключения, объект исключения и  
трассировочный объект). Элементы этого кортежа передаются как  
параметры функции **traceback. f ormat\_exception** **(),** юторая и  
печатает информацию об исключении в уже знакомой форме. Модзиь  
**traceback** содержит и другие функции (о них можно звнать 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 юде модулей стандартной библиотеки Pythoa

Заключение

с помоа(ью возможностей интроспекции удается рассмотреть фазы  
работы транслятора Python: лексический анализ, синтаксический разбор  
и генерации юда 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 юда на РздЬоп по сюрости.

Наюнец, интроспекция позволяет исследовать не толью строение  
программы, но и объектов, с юторыми работает эта программа. Были  
рассмотрены возможности Python по ползліению информации об  
объектах - этом основном строительном материале, из юторого  
складываются данные любой Python-программы.

**Сузи Р.А.**

Содержание

**2**

**3**

1. **35  
   67**

**91**

**126**

**148**

**177**

**199**

**218**

**241**

**259**

**277**

**302**

**325**

**Титульная страница  
Выходные данные**

**Лекция 1. Введение в программирование на языке  
Python**

**Лекция 2. Основные стандартные модули Python**

**Лекция 3. Элементы функционального  
программирования**

**Лекция 4. Объектно-ориентированное  
программирование**

**Лекция 5. Численные алгоритмы. Матричные  
вычисления**

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

**Лекция 7. Работа с данными в различных форматах  
Лекция 8. Разработка Web-приложений  
Лекция 9. Сетевые приложения на Python  
Лекция 10. Работа с базой данных  
Лекция 11. Многопоточные вычисления**

**Лекция 12. Создание приложений с графическим  
интерфейсом пользователя**

**Лекция 13. Интеграция Python с другими языками  
программирования**

**Лекция 14. Устройство интерпретатора языка Python**