Python. Лекция 1.
Введение в программирование на Python.

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

Что такое Python?

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

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

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

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

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

Как описать язык?

В этой лекции не ставится цели систематически описать Python: для этого существует оригинальное справочное руководство. Здесь предлагается рассмотреть язык одновременно в нескольких аспектах, что достигается набором примеров, которые

позволят быстрее приобщиться к реальному программированию, чем в случае строгого академического подхода.

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

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

История языка Python

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

Программа на Python

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

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

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

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

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

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

Примечание:

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

Основные алгоритмические конструкции

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

Последовательность операторов

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

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

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

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

Оператор условия и выбора

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

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

Этот кусок кода на Python интуитивно понятен каждому, кто помнит, что `if` по-английски значит "если", а `else` - "иначе". Оператор ветвления имеет в данном случае

две части, операторы каждой из которых записываются с отступом вправо относительно оператора ветвления. Более общий случай - **оператор выбора** - можно записать с помощью следующего синтаксиса (пример вычисления знака числа):

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

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

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

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

Циклы

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

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

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

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

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

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

```
    print l,
f.close()
```

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

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

Примечание:

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

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

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

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

Функции

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

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

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

В этом примере определена функция двух аргументов (из которых второй имеет **значение по умолчанию** - `0`). Вариантов вызова этой функции с конкретными параметрами также несколько. Стоит только заметить, что при вызове функции сначала должны идти позиционные параметры, а затем, именованные. Аргументы со

значениями по умолчанию должны следовать после обычных аргументов. Оператор `return` возвращает значение функции. Из функции можно возвратить только один объект , но он может быть кортежем из нескольких объектов.

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

Исключения

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

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

В этом примере берутся числа из двух файлов и делятся одно на другое. В результате этих нехитрых действий может возникнуть несколько исключительных ситуаций, некоторые из них отмечены в частях `except` (здесь использованы стандартные встроенные исключения Python). Последняя часть `except` в этом примере улавливает все другие исключения, которые не были пойманы выше. Например, если хотя бы в одном из файлов находится нечисловое значение, функция `int()` возбудит исключение `ValueError`. Его-то и сможет отловить последняя часть `except`. Разумеется, выполнение части `try` в случае возникновения ошибки уже не продолжается после выполнения одной из частей `except`.

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

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

Вместо

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

Примечание:

Пример уже несколько устаревшей идиомы языка Python иллюстрирует только дух

этого подхода: в современном Python лучше записать так `value = dict.get(key, default_value)`.

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

```
class MyError(Exception):
  pass

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

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

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

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

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

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

Встроенные типы данных

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

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

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

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

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

Тип int и long

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

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

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

```
>>> print 1 + 1, 3 - 2, 2*2, 7/4, 5%3
2 1 4 1 2
>>> print 2L ** 1000
10715086071862673209484250490600018105614048117055336074437503883703510511249361224931983788156958581275946729175531468251871452856923140435984577574698574803934567774824230985421074605062371141877954182153046474983581941267398767559165543946077062914571964776865421676604298316526243868372056680693769
>>> print 3 < 4 < 6,      3 >= 5,    4 == 4,    4 != 4  # сравнения
True False True False
>>> print 1 << 8,   4 >> 2,    ~4    # побитовые сдвиги и инверсия
256 1 -5
```

```
>>> for i, j in (0, 0), (0, 1), (1, 0), (1, 1):
...   print i, j, ":", i & j, i | j, i ^ j  # побитовые операции
...
0 0 : 0 0 0
0 1 : 0 1 1
1 0 : 0 1 1
1 1 : 1 1 0
```

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

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

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

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

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

Тип float

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

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

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

> **Примечание**:
>
> Для финансовых расчетов лучше применять более подходящий тип.

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

Тип complex

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

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

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

Тип bool

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

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

Следует отметить, что Python даже не вычисляет второй операнд операции `and` или `or`, если ее исход ясен по первому операнду. Таким образом, если первый операнд 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 строковых литералов. Для литералов наравне используются как апострофы ('), так и обычные двойные кавычки ("). Для многострочных литералов можно использовать утроенные апострофы или утроенные кавычки. Управляющие последовательности внутри строковых литералов задаются обратной косой чертой (\ ). Примеры написания строковых литералов:

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

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

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

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

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

```
>>> "A" + "B"
```

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

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

Тип tuple

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

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

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

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

Тип list

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

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

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

Последовательности

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

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

Дополнительные конструкции для изменчивых последовательностей:

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

Некоторые методы для работы с последовательностями

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

Метод	Описание
`append(x)`	Добавляет элемент в конец последовательности
`count(x)`	Считает количество элементов, равных `x`
`extend(s)`	Добавляет к концу последовательности последовательность `s`

`index(x)`	Возвращает наименьшее `i`, такое, что `s[i] == x`. Возбуждает исключение `ValueError`, если `x` не найден в `s`
`insert(i, x)`	Вставляет элемент `x` в `i` -й промежуток
`pop(i)`	Возвращает `i` -й элемент, удаляя его из последовательности
`reverse(s)`	Меняет порядок элементов `s` на обратный
`sort([cmpfunc])`	Сортирует элементы `s`. Может быть указана своя функция сравнения `cmpfunc`

Взятие элемента по индексу и срезы

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

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

Примечание:

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

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

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

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

А теперь пример работы со срезами:

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

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

Тип dict

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

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

Тип file

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

Следующий пример показывает копирование файла:

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

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

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

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

Выражения

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

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

Операция	Название
`lambda`	лямбда-выражение
`or`	логическое ИЛИ
`and`	логическое И
`not x`	логическое НЕ
`in, not in`	проверка принадлежности
`is, is not`	проверка идентичности
`<,<=,>,>=,!=,==`	Сравнения
`|`	побитовое ИЛИ
`^`	побитовое исключающее ИЛИ
`&`	побитовое И
`<<, >>`	побитовые сдвиги
`+, -`	сложение и вычитание
`*, /, %`	умножение, деление, остаток
`+x, -x`	унарный плюс и смена знака

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

Таким образом, порядок вычислений операндов определяется такими правилами:

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

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

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

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

Очень часто выражения стоят в правой части оператора присваивания или расширенного присваивания. В Python (в отличие, скажем, от C) нет операции

присваивания, поэтому синтаксически перед знаком = могут стоять только идентификатор, индекс, срез, доступ к атрибуту или кортеж (список) из перечисленного. (Подробности в документации).

Имена

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

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

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

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

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

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

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

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

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

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

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

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

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

Стиль программирования

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

Стиль программирования затрагивает практически все аспекты написания кода:

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

Для языка Python Гвидо ван Россум разработал официальный стиль. С оригинальным текстом «Python Style Guide», с недавних пор включенным в состав Python Enhancement Proposals можно ознакомиться здесь.

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

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

оболочки (IDE) автоматически делают необходимые отступы в Python-программах:

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

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

Несколько рекомендаций касаются написания комментариев.

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

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

- Строка документации для отдельной программы должна объяснять используемые ею ключи, назначение аргументов и переменных среды и другую подобную информацию.
- Для строк документации рекомендуется везде использовать утроенные кавычки (`"""`).
- Однострочная документация пишется в императиве, как команда: "делай это", "возвращай то".
- Многострочная документация содержит расширенное описание модуля, функции, класса. Она будет смотреться лучше, если текст будет написан с тем же отступом, что и начало строки документации.

- Документация для модуля должна перечислять экспортируемые функции, классы, исключения и другие объекты, по одной строке на объект.
- Строка документации для функции или метода должна кратко описывать действия функции, ее входные параметры и возвращаемое значение, побочные эффекты и возможные исключения (если таковые есть). Должны быть обозначены необязательные аргументы и аргументы, не являющиеся частью интерфейса.
- Документация для класса должна перечислять общедоступные методы и атрибуты, содержать рекомендации по применению класса в качестве базового для других классов. Если класс является подклассом, необходимо указать, какие методы полностью заменяют, перегружают, а какие используют, но расширяют соответствующие методы надкласса. Необходимо указать и другие изменения по сравнению с надклассом.
- Контроль версий повышает качество процесса создания программного обеспечения. Для этих целей часто используются RCS или CVS. "Python Style Guide" рекомендует записывать `$Revision: 1.31 $` в переменную с именем `__version__`, а другие данные заключать в комментарии " `#` ".

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

- Имена модулей лучше давать строчными буквами, например, `shelve`, `string`, либо делать первые буквы слов прописными, `StringIO`, `UserDict`. Имена написанных на C модулей расширения обычно начинаются с подчеркивания " `_` ", а соответствующие им высокоуровневые обертки - с прописных букв: `_tkinter` и `Tkinter`.
- Ключевые слова нельзя использовать в качестве имен, однако, если все-таки необходимо воспользоваться этим именем, стоит добавить одиночное подчеркивание в конце имени. Например: `class_`.
- Классы обычно называют, выделяя первые буквы слов прописными, как в `Tag` или `HTTPServer`.
- Имена исключений обычно содержат в своем составе слово "error" (или "warning"). Встроенные модули пишут это слово со строчной буквы (как `os.error`) (но могут писать и с прописной): `distutils.DistutilsModuleError`.
- Функции, экспортируемые модулем, могут именоваться по-разному. Можно давать с прописных букв имена наиболее важных функций, а вспомогательные писать строчными.
- Имена глобальных переменных (если таковые используются) лучше начинать с подчеркивания, чтобы они не импортировались из модуля оператором `from-import` со звездочкой.
- Имена методов записываются по тем же правилам, что и имена функций.
- Имена констант (имен, которые не должны переопределяться) лучше записывать прописными буквами, например: `RED`, `GREEN`, `BLUE`.
- При работе с языком Python необходимо учитывать, что интерпретатор считает некоторые классы имен специальными (обычно такие имена начинаются с подчеркивания).

Заключение

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