Python. Лекция 4.
Элементы ООП.

Python проектировался как объектно-ориентированный язык программирования. Это означает[1], 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 (Объектно-ориентированное программирование) изобилует значительным числом терминов, зачастую затемняющих суть вопроса. В данном изложении будут употребляться только те термины, которые необходимы на практике для взаимопонимания разработчиков или для расширения кругозора. Так как в разных языках программирования ООП имеет свои нюансы, в скобках иногда будут даваться синонимы или аналоги того или иного термина.

Примечание:

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

Основные понятия

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

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

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

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

[1] по Алану Кэю, автору объектно-ориентированного языка Smalltalk

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

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

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

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

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

Абстракция и декомпозиция

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

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

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

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

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

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

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

Объекты

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

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

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

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

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

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

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

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

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

```
abs(c)
```

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

```
c.__abs__()
```

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

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

Типы и классы

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

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

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

Примечание:

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

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

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

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

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

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

Определение класса

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

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

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

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

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

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

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

Использовать класс можно следующим образом:

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

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

что даст в результате

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

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

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

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

Инкапсуляция

Обычно считается, что без инкапсуляции невозможно представить себе ООП, что это ключевое понятие. История развития методологий программирования движима борьбой со сложностью разработки программного обеспечения. Сложность больших программных систем, в создании которых участвует сразу большое количество разработчиков, уменьшается, если на верхнем уровне не видно деталей реализации нижних уровней. Собственно, процедурный подход был первым шагом на этом пути. Под **инкапсуляцией** (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 Python инкапсуляции не придается принципиального значения: ее соблюдение зависит от дисциплинированности программиста. В других языках программирования имеются определенные градации доступности методов объекта.

Доступ к свойствам

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

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

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

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

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

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

`__getattr__(self, name)`	Этот метод объекта вызывается в том случае, если атрибут не найден другим способом (его нет в данном экземпляре или в дереве классов). Здесь name - имя атрибута. Метод должен вычислить значение атрибута либо возбудить исключение `AttributeError`. Для получения полного контроля над атрибутами в "новых" классах (то есть потомках `object`)

	используйте метод `__getattribute__()`.
`__setattr__(self, name, value)`	Этот метод вызывается при присваивании значения некоторому атрибуту. В отличие от `__getattr__()`, метод всегда вызывается, а не только тогда, когда атрибут может быть найден в экземпляре класса, поэтому нужно с осторожностью присваивать значения атрибутам внутри этого метода: это может вызвать рекурсию. Для присваивания значений атрибутов предпочтительнее присваивать словарю `__dict__`: `self.__dict__[name] = value` или (для "новых" классов) - обращение к `__setattr__()` базового класса: `object.__setattr__(self, name, value)`.
`__delattr__(self, name)`	Как можно догадаться из названия, этот метод служит для удаления атрибута.

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

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

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

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

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

ad = AttDict({'a': 1, 'b': 10, 'c': '123'})
print ad.a, ad.b, ad.c
ad.d = 512
print ad.d
```

Сокрытие данных

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

```
>>> class X:
...   x = 0
...   _x = 0
...   __x = 0
...
```

```
>>> dir(X)
['_X__x', '__doc__', '__module__', '_x', 'x']
```

Полиморфизм

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

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

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

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

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

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

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

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

Имитация типов

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

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

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

Как видно из этого примера, экземпляры класса `CountArgs` можно вызывать подобно функциям (в результате будет возвращено количество переданных

параметров). При попытке вызова экземпляра на самом деле будет вызван метод `__call__()` со всеми аргументами.

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

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

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

Программа выведет:

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

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

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

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

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

Отношения между классами

Наследование

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

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

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

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

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

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

Множественное наследование

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

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

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

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

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

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

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

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

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

class AB:
  a = ma
  b = mb

class BC:
  b = mb
  c = mc

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

Порядок разрешения методов

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

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

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

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

Агрегация

Контейнеры

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

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

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

Использование:

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

```
>>> print s
1 : 2
```

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

Примечание:

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

Итераторы

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

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

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

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

Программа выдаст

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

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

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

Ассоциация

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

```
lst = [1, 2, 3]
if 1 in lst:
   ...
```

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

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

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

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

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

Слабые ссылки

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

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

```
>>> import weakref
>>>
>>> class MyClass(object):
...  def __str__(self):
...   return "MyClass"
...
>>>
>>> s = MyClass()          # создается экземпляр класса
>>> print s
MyClass
>>> s1 = weakref.proxy(s)  # создается прокси-объект
>>> print s1               # прокси-объект работает как исходный
MyClass
>>> ss = weakref.ref(s)    # создается слабая ссылка на него
>>> print ss()             # вызовом ссылки получается исходный объект
MyClass
>>> del s                    # удаляется единственная сильная ссылка на
объект
>>> print ss()             # теперь исходного объекта не существует
None
```

```
>>> print s1
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ReferenceError: weakly-referenced object no longer exists
```

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

Статический метод

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

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

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

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

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

```
class A(object):

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

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

Метод класса

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

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

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

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

```
            return val
    convert = classmethod(convert)
```

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

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

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

Метаклассы

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

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

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

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

Использование будет выглядеть так:

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

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

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

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

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

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

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

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

В результате получится класс, эквивалентный следующему:

```
class My_Class(object):
```

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

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

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

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

Примечание:

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

Мультиметоды

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

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

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

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

```
from Multimethod import Method, Generic, AmbiguousMethodError

# классы, для которых будет определен мультиметод
class A: pass
```

```
class B(A): pass

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

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

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

Устойчивые объекты

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

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

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

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

```
from time import time, gmtime
import copy
class CC:
    def __init__(self, created=time()):
        self.created = created
        self.created_gmtime = gmtime(created)
        self._copied = 1
        print id(self), "init", created
    def __getinitargs__(self):
        print id(self), "getinitargs", self.created
```

```
            return (self.created,)
        def __getstate__(self):
            print id(self), "getstate", self.created
            return {'_copied': self._copied}
        def __setstate__(self, dict):
            print id(self), "setstate", dict
            self._copied = dict['_copied'] + 1
        def __repr__(self):
            return "%s obj: %s %s %s" % (id(self), self._copied,
                                         self.created, self.created_gmtime)

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

В результате будет получено

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

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

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

Примечание:

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

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

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

Критика ООП

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

Также спорна большая гибкость объектных программ к изменениям. Она зависит от того, вносится ли новый метод (для серии объектов) или новый тип объекта. При процедурном подходе при появлении нового метода пишется отдельная процедура, в которой в каждой ветке алгоритма обрабатывается свой тип данных (то есть такое изменение требует редактирования одного места в коде). При ООП изменять придется каждый класс, внося в него новый метод (то есть изменения в нескольких местах). Зато ООП выигрывает при внесении нового типа данных: ведь изменения происходят только в одном месте, где описываются все методы для данного типа. При процедурном подходе приходится изменять несколько процедур. Сказанное иллюстрируется ниже. Пусть имеются классы `A`, `B`, `C` и методы `a`, `b`, `c`:

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

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

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

# процедурный подход

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

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

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

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

```
# ООП

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

```
# процедурный подход

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

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

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

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

```
# процедурный подход

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

# ООП

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

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

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

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

Заключение

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

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