**Изучаем Python**

**Марк Лутц 2011, издание O’ Reilly**

**Том 4**

***Часть 6. Классы и ООП.*** *стр. 693*

**Глава 25. ООП: общая картина.** стр. 695

Весь написанный нами код до этого, был основан на объектах (object-based) - в сценариях мы передавали объекты, использовали в выражениях, вызывали их методы и т.д. Но чтобы наш код стал по-настоящему объектно-ориентированным (ОО), наши объекты должны участвовать в иерархии наследования.

Мы присутпаем к изучению классов - программных компонентов, используемых для реализации новых типов объектов, поддерживающих наследование. Классы - основные инструменты ООП в Питон. ООП представляет подход к программированию, при котором мы разлагаем код программ на составляющие, чтобы уменьшить его избыточность, и пишем новые программы, адаптируя имеющийся код, а не изменяя его.

Классы создаются с пом. инструкции class. Увидим, чт ообъекты, определяемые с пом. классов, очень напоминают встр. типы знакомые нам. На самом деле классы лишь применяют и дополняют понятия, которые мы уже рассмотрели. Проще - они представляют собой пакеты ф-ций, которые в основном использ. и обрабатывают объекты встр. типов. Основное назнач. классов в том, чтобы создавать и манипулировать новыми объектами, кроме того они поддерживают механизм наследования - иной способ адапатции кода и его повторного использ., чем мы узнали до этого.

ООП в Питно явл-ся необязательным и на первых порах нам можно не использ. классы. Они больше нужны для стратегического программирования (долгосрочная разработка). Но тем неменее классы полезный и мощный инструмент.

**Зачем нужны классы?**

Помните - "программы выполняют некоторые действия над чем-то"? Так вот классы - это лишь способ определить новое "что-то", они явл-ся отражением реальных объектов в мире программ. Напр. хотим сделать машину для готовки пиццы. Если реализовать её на основе классов, то мы могли бы смоделировать более близкое к реальности строение машины взаимосвязь между её механизмами. Здесь нам полезны два аспекта ООП:

- ***Наследование.*** Машина по изготовлению пиццы - это машина и должна иметь характеристики машин как таковых. В терминах ООП это наз-ся "наследованием" свойств более общей категории машин. Эти свойства нужно реализовать всего один раз и их можно будет использ. для создания любых типов машин нужных нам.

- ***Композиция.*** Машина для пиццы состоит из множества компонентов, которые работают как единое целое. Напр. у нас будут манипуляторы (раскатывать тесто напр.), двигатели (надо же чем-то двигать машину) и т.д. На языке ООП наша машина - это пример композиции; она содержит др. объекты, которые активизируются для выполнения определенных действий. Каждый компонент можно оформить как класс.

Общие идеи ООП, как наследование и композиция, применимы к любым приложениям, которые можно разложить на объекты. Напр. интерфейс - набор визуальных элементов управления, кнопок, меток и т.д., которые рисуются на экране в момент, когда выполняется рисование их контейнеров (композиция). Также мы можем создать свои элементы - кнопки с уникальными шрифтами, новые цвета у элементов, вообщем свои специализированные версии общих эелементов (наследование).

Классы - это программ. компоненты Питона, как ф-ции и модули: они представляют собой еще один способ упаковки кода и данных. У классов есть свое пространство имен. Но есть и три важных отличия:

- ***Множество экземпляров.*** Классы это как-бы фабрики по созданию объектов. Каждый раз когда вызывается класс, создается объект со своим пространством имен. Объект, созданный из класса, имеет доступ к атрибутам класса и получает своё пространство имен для своих собственных данных, отличных от др. объектов.

- ***Адаптация через наследование.*** Наследование - мы можем расширять возможности класса, переопределяя его атрибуты за пределами самого класса. Классы могут создавать иерархии пространства имен, которые определяют имена для использ-ия объектами, созданными из классов в иерархии.

- ***Перегрузка операторов.*** Со спец. протоколом оформления методов, классы могут определять объекты, над которыми можно делать операции, которые мы знаем по встр. типам (напр. извлечение среза, конкатенация, индексация и т.д.). Питон предоставляет спец. обработчики для классов, для перехвата и реализации любой встр. операции.

**ООП с высоты 30000 футов.** стр. 697

**Поиск унаследованных атрибутов.**

ООП в Питон проще в изучении и использ-ии чем в С++ или Java. Фактически ООП в Питон сводится к выражению:

object.attribute

Оно нам знакомо и мы его много использовали. Однако когда это выражение применяется к объекту, полученному с пом. class, интерпретатор начинает поиск - поиск в дереве связанных объектов, который заканчивается, как только будет встречено первое появление атрибута attribute. Когда в деле классы это выражение можно перевести так:

- Найти первое вхождение атрибута attribute, просмотрев объект object, а потом все классы в дереве наследования выше него, снизу вверх и слева направо.

Эта процедура поиска также еще наз-ся поиском в дереве наследования. Объекты ниже наследуют атрибуты у объектов выше по дереву.

Т.е. с пом. программ. кода мы действительно создаем деервья связанных объектов и интерпретатор поднимается по этому дереву вверх в поисках атрибутов.

**Рисунок 25.1 Дерево классов.** стр. 698

На рисунке дерево из 5 объектов, помеченных именами переменных. Два эземпляра внизу (I1, I2), класс выше их (С1) и два суперкласса в самом верху (С2 и С3), т.е. два объекта экземпляров и три объекта классов. Эти объекты явл-ся пространствами имен (пакетами переменных), а наследование означает просто поиск снизу вверх, с целью найти самое нижнее вхождение атрибута с нужным именем. Программ. код задает структуру таких деревьев.

Итак, каждый из этих объектов обладает набором атрибутов, а именно - I1.name; I2.name; C1.x, C1.y; C2.x, C2.z; C3.w, C3.z. Заметим, что классы и экземпляры порождаются от двух разных типов объектов:

- ***Классы.*** Играют роль фабрик экземпляров. Их атрибуты обеспечивают поведение - данные и функции - т.е. наследуются всеми экземплярами, созданными от них (напр. ф-ция, считающая зарплату, исходя из часового тарифа).

- ***Экземпляры***. Представляют конкретные элементы программы. Их атрибуты хранят данные, которые могут отличаться в конкретных объектах (напр. номер карточки соц. страхования).

Т.о., по дереву, экземпляры наследуют атрибуты своих классов, а классы наследуют атрибуты классов выше по дереву.

Классы, выше по дереву (С2 и С3), обычно наз. суперклассами (или родительские, базовые), а классы ниже по дереву (С1), обычно наз. подклассами (или дочерние). Суперклассы реализуют общее поведения для подклассов, а подклассы могут переопределять поведение наследуемое от суперкласса, переопределяя имена суперклассов ниже в дереве.

Допустим мы создали дерево по рис. 25.1 и пишем:

I2.w

Этот код демонстрирует наследование. Т.к. это выражение вида object.attribute, оно приводит к запуску поиска в дереве - интерпретатор ищет атрибут w начиная с I2 и идет вверх. Порядок такой - I2, C1, C2, C3.

Остановится когда найдет атрибут или возбудит исключение если w не найдет вообще. Т.о. автоматич. поиск найдет имя I2.w как C2.w. В терминологии ООП это наз-ся I2 "наследует" атрибут w от C3.

В итоге видим, что два экземпляра наследуют от своих классов 4 атрибута: w, x, y, z. Другие атрибуты будут анйдены в различ. местах в дереве. Например:

- Для I1.x и I2.x атрибут х будет найден в С1, где поиск остановится, т.к. С1 нах-ся ниже в дереве чем С2.

- Для I1.y и I2.y атрибут y будет найден в С1, где поиск остановится, т.к. это единственное место где он появл-ся.

- Для I1.z и I2.z атрибут z будет найден в С2, т.к. С2 нах-ся в дереве левее чем С3.

- Для I2.name атрибут name будет найден в I2, т.к. в этом случае поиск вообще не будет идти по дереву.

Обратите внимание на атрибут х в С1, т.к. мы видим, что С1 переопределяет этот атрибут ниже в дереве, тем самым замещая версию атрибута, расположенную выше в С2. Ниже мы увидим, что такие переопределения составляют основу адаптации программ. обеспечения в ООП.

**Классы и экземпляры.** стр. 700

Классы и экземпляры (отдельные типы объектов) вып-ют почти одну роль: каждый из этих типов служит неким представлением пространства имен - пакета переменных и места, где определяются атрибуты. Т.о. можно сказать, что классы и экземпляры похожи на модули, при этом объекты в деревьях классов содержат автоматически определяемые ссылки на др. объекты пространств имен, и классы соотв. инструкциям, а не файлам.

Различие между ними такое, что классы представляют собой своего рода фабрики по производству экземпляров. Напр. есть класс Employee, определяющих характеристики служащего - из этого класса мы можем создавать фактические экземпляры класса Employee.

Это еще одно отличие классов от модулей: у нас всегда есть всего один экземпляр модуля в памяти (поэтмоу мы его и перезагружаем при необходимости), но экземпляров класса мы можем сделать сколько хотим.

У классов обычно есть ф-ции, присоединенные к ним (пр. computeSalary), а у экземпляров - элементы данных, используемые ф-циями класса (пр. hoursWorked). Фактически модель ООП не так сильно отличается от классич. модели обработки данных, основанной на программах и записях, - в ООП экземпляры подобны записям с "данными", а классы - "программам", обрабатывающим эти данные. Однако в ООП есть еще иерархия наследования.

**Вызовы методов классов.**

Выше мы видели как атрибут I2.w в нашем дереве классов транслируется в C3.w при поиске. Нуэно понимать и знать, что точно так же наследуются и методы (т.е. ф-ции, присоединенные к классам в виде атрибутов).

Если ссылка I2.w - это вызов ф-ции, то оно означает: "вызвать ф-цию C3.w для обработки I2". Т.е. интерпретатор автоматич. отобразит вызов I2.w() на вызов C3.w(), передав унаследованной ф-ции экземпляр в виде первого аргумента.

Фактич. всякий раз когда вызывается ф-ция, присоединенная к классу, подразумевается не класс целиком, а экземпляр класса. Этот экземпляр, или контекст, явл-ся одной из причин, почему данная модель названа объектно-ориентированной, - всегда существует объект, над которым выполняются действия. Напр. мы вызываем метод giveRaise, присоедиенный как атрибут к классу Employee, - вызов этого метода не имел бы смысла без указания служащего, которому дается надбавка к зарплате.

Как мы узнаем далее, Питон передает методам подразумеваемый экземпляр в виде спец. первого аргумента, в соответствии с соглашением именуемого self. Также узнаем, что методы могут вызываться как через экземпляры (bob.giveRaise()), так и через классы (Employee.giveRaise(bob)).

**Создание деревьев классов.**

За всеми идеями ООП стоит реальный код. Мы создаем деревья и объекты в них с пом. инструкций class и вызовов классов, которые рассмотрим позднее. В двух словах:

- Каждая инструкция class создает новый объект класса.

- Каждый раз, когда вызывается класс, он создает новый объект экземпляра.

- Экземпляры автомат. свзываются с классами, из которых они были созданы.

- Классы связаны со своими суперклассами, перечисленными в круглых скобках в заголовке инструкции class, при этом порядок следования в списке определяет порядок расположения в дереве.

Например чтобы создать деревео на рис. 25.1 мы можем написать такой код (без реализации классов конечно, она пока не нужна):

class C2: ... # Создать объекты классов (на рис. это овалы)

class C3: ...

class C1(C2, C3): ... # Создаем объект класса связанного с суперклассами

I1 = C1() # Создаем объекты экземпляров (на рис. прямоугольники)

I2 = C1() # связанные со своими классами

- как видим экземпляры конструируем вызвав класс С1 дважды, как если бы это была обычная ф-ция. Экземпляры помнят класс, из которого они созданы, а класс С1 помнит свои суперклассы.

В этом примере использ-ся т.н. множественное наследование, которое означает, что некий класс имеет более одного суперкласса над собой в дереве классов. А порядок суперклассов с кобках определяет порядок поиска атрибутов.

Большое значени имеет то, к какому из объектов присоединяется тот или иной атрибут - что определяет его область видимости. Атрибуты, присоединямые к экземплярам, принадлежат только этим конкретным экземплярам, но атрибуты, присоединенные к классам, использ-ся и подклассами и экземплярами.

Далее увидим, что:

- Атрибуты обычно присоед-ся к классам с пом. инструкций присваивания внутри class, а не во вложенных инструкциях def, определяющих ф-ции.

- Атрибуты обычно присоед-ся к экземплярам с пом. присваивания значений спец. аргументу с именем self, передаваемому ф-циям внутри классов.

Напр. классы поределяют поведение своих экземпляров с пом. ф-ций, создаваемых def внутри инструкций class. А т.к. такие влож. инструкции def выполняют присваивание именам внутри класса, они присоед-ся к объектам классов в виде атрибутов и будут унаследованы всеми экземплярами и подклассами:

class C1(C2, C3): # Создать и связать класс С1

def setname(self, who): # Присвоить: С1.setname

self.name = who # self - либо I1, либо I2

I1 = C1() # Создать два экземпляра

I2 = C1()

I1.setname('bob') # Записать 'bob' в I1.name

I2.setname('mel') # Записать 'mel' в I2.name

print(I1.name) # Выведет 'bob'

Синтаксис инструкции def здесь самый обычный. С функциональной точки зрения, когда def появл-ся внутри инструкции class, как в нашем примере, она обычно наз-ся методом и автомат. принимает спец. первый аргумент с именем self, который содержит ссылку на обрабатываемый экземпляр.

Примечание. Если бы мы ранее изучали С++ или Java, то заметили бы, что в Питон имя self - это то же, что и указатель this, но в языке Питон аргумент self всегда использ-ся явно, чтобы сделать обращения к атрибутам более очевидным.

Т.к. классы способны делать множество экземпляров их методы использ-т этот, получаемый автоматически, аргумент self для получения или изменения атрибутов конкретного экземпляра, который обрабатывается методом. В нашем прим. имя self использ-ся для сохранения имени служащего в конкретном экземпляре.

Как и простые переменные, атрибуты классов и экземпляров не объявляются заранее, а появл-ся когда им присв-ся значение. Когда метод присв-ет значение атрибуту с пом. имени self, он т.о. создает атрибут экземпляра, находящегося в нижнем уровне дерева классов, потмоу что имя self автоматич. ссылается на обрабатываемый экземпляр.

Благодаря тому, что все объекты дерева классов - всего лишь объекты пространств имен, можно получать или устанавливать любой из их атрибутов, используя соотв. имена. Например выражение C1.setname явл-ся таким же допустимым, как и I1.setname, с условием, что имена C1 и I1 находятся в области видимости программ. кода.

Сейчас класс С1 не присоед-ет атрибут name к экземплярам, пока не будет вызван метод setname. Если нужно, чтобы атрибут name всегда присутствовал в экземплярах, то он должен создаваться на этапе создания калсса:

class C1(C2, C3):

def \_\_init\_\_(self, who): # Создать имя при создании класса

self.name = who # self - либо I1, либо I2

I1 = C1('bob') # Записать 'bob' в I1.name

I2 = C1('mel') # Записать 'mel' в I2.name

print(I1.name) # Выведет 'bob'

При таком решении интерпретатор автоматич. будет вызывать метод с именем \_\_init\_\_ каждый раз при создании экземпляра класса. Новый экземпляр будут передаваться методу \_\_init\_\_ в виде первого аргумента self, а любые значения, перечисленные в круглых скобках при вызове класса, будут передаваться во втором и последующих аргументах. Т.о. инициализация экземпляров будет выполняться в момент их создания.

Метод \_\_init\_\_ известен как конструктор, т.к. он запускается на этапе конструирования экземпляра. Этот метод явл-ся представителем класса методов, которые наз-ся методами перегрузки операторов. Такие методы наслед-ся в дереве как обычно, а их миена нач-ся и закан-ся двумя подчеркиваниями. Интерпретатор вызывает их автоматич. когда экземпляры, поддерживающие их, участвуют в соответствующих операциях, и они, глав. образом, явл-ся альтернативой вызовам простых методов. Так же они не обязательны, если их нет то соотв. операции просто не поддерживаются экземплярами.

**ООП - это многократное использование программного кода.**

Конечно в ООП есть не только наследование. Напр. перегрузка операторов может применяться в др. случаях - классы могут предоставлять свои реализации таких операций как доступ по индексу, получение значений атрибутов, выво ди мн. др. Но вообще ООП реализует поиск атрибутов в деревьях.

Используя классы мы адаптируем программы, вместо того, чтобы изменять код или писать новый в каждом проекте - ООП дает нам способы многократного использования программ. кода.

Классы это действительно лишь пакеты ф-ций и др. имен, которые напоминают во многом модули. Но также они обеспечивают удобную структуру размещения кода и переменных. И, в отличие от ф-ций и модулей, благодаря дереву наследования и поиску в нем, дают широкие возможности по адаптации программ. кода.

Напр. методы - это обычные ф-ции со спец. первым аргументом, поэтому мы можем подражать некоторым их чертам поведения, вручную передавая объекты для обработки обычным ф-циям. Методы в наследовании классов дают возможность адаптировать существующее программное обеспечение, создавая новые подклассы, определяющие новые методы, вместо того, чтобы изменять сущзествующий код. Такое невозможно в случае с модулями или ф-циями.

Пример. Пусть нас привлекли к реализации БД, где хранится информация о служащих. На базе ООП мы могли бы начать с реализации суперкласса с общими категориями для служащих:

class Employee:

def computeSalary(self): ...

def giveRaise(self): ...

def promote(self): ...

def retire(self): ...

Реализовав общее можно специализировать его для каждой категории служащих (у них же есть отличия). Т.е. можно создать подклассы, в которых будут изменения на основе отличий от типового представления служащего - а общие черты останутся от суперкласса. Напр. з\п инженера начисляется по особым правилам (не по часовому тарифу например), т.о. нам нужно переопределить всего один метод:

class Engineer(Employee): # Специализированный подкласс

def computeSalary(self): ... # Особенная реализация метода

- Новая версия computeSalary будет нах-ся в дереве классов ниже и будет замещать (переопределять) общую версию метода в классе Employee. Далее можно будет создавтаь экземпляры разновидностей классов служащих соответственно с принадлежностью:

bob = Employee() # Поведение по умолчанию

mel = Engineer() # Особые правила начисления з\п

Обратите внимание, что можно создавать экземпляры любых классов в дереве, - класс, экземпляр которого создается, определяет уровень, откуда будет начинаться поиск атрибутов. Напр. в дальнейшем эти два объекта экземпляров (bob и mel) могли бы быть встроены в большой контейнерный объект (напр. список или экземпляр др. класса), котоый представляет отдел или компанию, реализуя идею композиции (см. начало главы).

Если нам понадобится узнать з\п этих служащих можно вычислить её в соотв. с правилами классов этих объектов, благодаря поиску в дереве наследования:

company = [bob, mel] # Составной объект

for emp in company:

print(emp.computeSalary()) # Вызвать версию метода данного объекта

Отметим, что список company мог бы храниться в файле (удобно для БД), например с пом. модуля pickle. Кроме того в Питон входит модуль shelve, который мог бы сохранять экземпляры классов в файлах с доступом по ключу, то же самое обеспечивает сторонняя разработка ZDOB и даже более качественно.

Последний пример - это еще одна разновидность полиморфизма (см. гл. 4 и 16). Т.е. смысл операции зависит от объекта, над которым она выполняется. Здесь метод computeSalary определяется для каждого объекта в дереве наследования.

В др. приложениях полиморфизм может применяться для сокрытия (т.е. для инкапсуляции) различий интерфейсов. Напр. программа, которая обрабатывает потоки данных, может работать с объектами, имеющими методы ввода и вывода, не заботясь о том, что эти методы делают в действительности:

def processor(reader, converter, writer):

while 1:

data = reader.read()

if not data: break

data = converter(data)

writer.rwite(data)

Передавая экземпляры классов с необходимыми интерфейсными методами read и write, специализированными под различные источники данных, мы можем использовать одну и ту же ф-цию processor для работы с любыми источниками данных, как уже существующими, так и с тем, что появятся в будущем:

class Reader:

def read(self): ... # Поведение и инструменты оп умолчанию

def other(self): ...

class FileReader(Reader):

def read(self): ... # Чтение из локального файла

class SocketReader(Reader): ...

def read(self): ... # Чтение из сокета

...

processor(FileReader(...), Converter, FileWriter(...))

processor(SocketReaderReader(...), Converter, TapeWriter(...))

processor(FtpReader(...), Converter, XmlWriter(...))

Благодаря тому, что внутр. реализация методов read и write была разделена по типам источников данных, их можно изменять, не трогая программный код, подобный приведенному, который использует их.

Фактически ф-ция processor сама может быть классом, реализующим логику работы ф-ции преобразования converter, которую могут унаследовать подклассы, и позволяющим встраивать экземпляры, выполняющие чтение и запись, в соотв. с принципом композиции.

Когда мы привыкнем программировать в ООП, то обнаружим, что наша задача сводится к тому, чтобы в новом проекте подобрать уже имеющиеся суперклассы для нашей задачи.

На практике мы можем найти или купить уже готовые библиотеки суперклассов, известные как "фреймворки", в которых задачи программирования реализвоаны в классах. Напр. интерфейсы к БД, протоколы тестирования, создание графич. интерфейса и мн. др.

Конечно для понимания ООП и адаптации кода нужно время, нужно много работать и учиться. Кстати, ряд программистов изучают и составляют списки наиболее часто встречающихся решений в ООП, сейчас они известны как - шаблоны проектирования.

**Глава 26. Основы программирования классов.** стр. 709

Теперь мы приступим к изучению синтаксиса модели классов в Питон. В этой главе рассмотрим неск-ко простых классов в действии.

Классы обладают тремя основными отличительными особенностями. На самом простом уровне они представляют собой лишь пространства имен, во многом похожие на модули, которые мы изучали в пятой части книги. Но, в отличие от модулей, классы также поддерживают создание множества объектов, реализуют наследуемое пространство имен и перегрузку операторов.

**Классы генерируют множество экземпляров объектов.**

Чтобы понять эту возможность, для начала нужно понимать, что в ОО модели Питон существует две разновидности объектов: объекты классов и объекты экземпляров.

Объекты классов реализуют поведение по умолчанию и играют роль фабрик по производству объектов экземпляров.

Объекты экземпляров - это настоящие объекты, обрабатываемые программой, - каждый представляет собой самостоятельное пространство имен, но наследует имена (т.е. автоматич. имеет доступ к ним) класса, из которого был создан.

Объекты классов считаются инструкциями, а объекты экземпляров - вызовами. Каждый раз когда мы вызываем класс, мы получаем новый экземпляр этого класса. Классы - это фабрики, которые могут воспроизводит ьмножество экземпляров. В противоположность этому каждый модуль может присутствовать в программе в единственном экземпляре.

Как мы увидим ниже классы в Питон сочетают в себе черты, напоминающие ф-ции и модули, но они совершенно не похожи на классы в др. ЯП.

**Объекты классов реализуют поведение по умолчанию.**

Когда вып-ся инструкция class, она создает объект класса. Приведем неск-ко отличительных характеристик классов в Питон.

- **Инструкция class создает объект класса и присваивает ему имя.** Как и def, class явл-ся выполняемой инструкцией. Когда она вып-ся она создает новый объект класса и присваивает его имени, указанному в заголовке class. Как и def инструкция class обычно вып-ся при первом импорте содержащих их файлов.

- **Операции присваивания внутри инструкции class создают атрибуты класса.** Как и в модулях, операции присваивания на верх. уровне (не влож. в инструкции def создают атрибуты объекта класса. Инструкция class преобразует свою область видимости в пространство имен атрибутов объекта класса, так же, как глоабльная область видимости модуля преобразуется в его пространство имен. При выполнении инструкции class атрибуты класса становятся доступны по их составным (полным) именам: object.name.

- **Атрибуты класса описывают состояние объекта и его поведение.** Т.е. атрибуты хранят информацию о состоянии и описывают поведение, которым обладают все экземпляры класса, - инструкции def, вложенные в class, создают методы, которые обрабатывают экзмепляры.

**Объекты экземпляров - это конкретные элементы.**

Когда выз-ся объект класса, возвращается объект экземпляра. Приведем неск-ко характеристик экземпляров.

- **Вызов объекта класса как функции создает новый объект экземпляра.** Когда вызываем класс создается и возвращается новый объект экземпляра. Экземпляры - это конкретные элементы данных в нашей проге.

- **Каждый объект экземпляра наследует атрибуты класса и приобретает свое собственное пространство имен.** Экземпляры созд-ся из классов и представляют собой новые пространства имен, они первоначальн опустые, но наследуют атрибуты класса из которого созданы.

- **Операции присваивания значений атрибутам, через ссылку self в методах, создают атрибуты в каждом отдельном экземпляре.** Методы класса, в первом аргументе (с именем self), получают ссылку на обрабатываемый объект экземпляра - присв-ие атрибутам через ссылку self создает или изменяет даннеы экземпляра, а не класса.

**Первый пример.**

Определим класс FirstClass в интерактив. оболочке:

>>> class FirstClass: # Определяет объект класса

def setdata(self, value): # Определяет метод класса

self.data = value # self - это экземпляр

def display(self):

print(self.data) # self.data - данные экземпляра

Здесь мы работаем в оболочке, но обычно такие инструкции вып-ся во время импорта вмещающего файл модуля. Как и ф-ции, классы не существуют пока интерпретатор не достигнет инструкций class и не выполнит их.

class нач-ся с заголовка (имя класса), затем идет тело класса (как обычно с отступами если нужно). У нас влож. инструкции это инструкции def - они определяют ф-ции, реализующие поведение класса.

Как мы знаем def на самом деле явл-ся операциями присв-ия - в нашем случае они присваивают объекты ф-ций именам setdata и display в области видимости инструкции class и тем самым создают атрибуты, присоединяемые к классу: FirstClass.setdata и FirstClass.display. Т.е. любое имя, которому присв-ся знач. на верх уровне во вложенном блоке класса становится атрибутом этого класса.

Ф-ции внутри классов обычно наз-ся методами. Но в первом аргументы они автоматич. получают подразумеваемый объект экземпляра - конекст вызова метода. Создадим пару экземпляров:

>>> x = FirstClass()

>>> y = FirstClass()

- Создаются два экземпляра с отдельными пространствами имен.

Вызывая класс таким образом мы создаем объекты экземпляров (обратите внимание на круглые скобки), которые явл-ся лишь пространством имен и имеют доступ к атрибутам класса.

Теперь у нас есть три объекта - два экземпляра и класс. Они связаны пространствами имен, по ООП объект х "наследует" класс FirstClass как и y.

Здесь у нас, в дереве наследования, атрибут "data" обнаруживается в экземплярах, а "setdata" и "display" - в классе, расположенном выше их.

Изначально оба экземпляра пустые, но они свзяаны с классом, из которого были созданы. Если через имя экземпляра обратиться к атрибуту объекта класса, то поиск вернет нам значение атрибута класса (при условии, что в экземпляре отсутствует одноименный атрибут):

>>> x.setdata("King Arthur") # Вызов метода: self - это х

>>> y.setdata(3.14159) # Эквивалентно: FirstClass.setdata(y, 3.14159)

Ни x, ни y не имеют атрибута setdata, поэтмоу интерпретатор идет по ссылке от экземпляра к классу. В этом и проявл-ся суть наследования в Питон - поиск имен в связанных объектах.

В ф-ции setdata внутри класса FirstClass значение аргумента записывается в self.data. Имя self внутри метода - имя самого первого аргумента, в соответствии с принятым соглашением, - автоматич. ссылается на обрабатываемый экземпляр (x или y), поэтмоу присваивание сохраняет значения в пространстве имен экземпляра, а не класса (так созд-ся имена-атрибуты "data" у экземпляров).

По причине того, что классы могут делать много экземпляров, методы должны использ-ть аргумент self, чтобы получит ьдоступ к обрабатываемому экземпляру. Пр ивызове метода display, чтобы вывести значение атрибутов self.data, мы увидим, что для каждого экземпляра они разные; с др. стороны, имя display само по себе одинаковое в x и y, т.к. оно пришло (унаследовано) из класса:

>>> x.display() # В каждом экземпляре свои значения self.data

King Arthur

>>> y.display()

3.14159

Видите - в атрибутах data экземпляров мы сохранили объекты различных типов (строку и число). Как и везде в Питон, атрибуты экземпляров (иногда наз-ся членами) никак не объявл-ся - они появл-ся, как только будет выполнена первая опреация присваивания (как и переменные). Т.е. если вызвать метод display до вызова метода setdata, то получим сообщение об ошибке.

Еще один способ позволяет изменять атрибуты экземпляров в самом классе, выполняя присваивание как с пом. аргумента self внутри методов, так и за пределами класса, когда экземпляр явно участвует в операции присваивания:

>>> x.data = "New value"

>>> x.display()

New value

Хоть это и редко применяется, но сущ-ет возможность создания новых атрибутов в пространстве имен экземпляра, присваивая значение за пределами методов класса:

>>> x.anothername = "spam"

Эта операция присоединит новый атрибут anothername, который затем сможет использ-ся любыми методами класса в объекте экземпляра х. Обычно классы создают все атрибуты экземпляров за счет присваивания значений аргументу self, но это не обязательно - программы могут получать, изменять или создавать атрибуты в любых объектах, к которым имеют доступ.

**Классы адаптируются посредством наследования.** Стр. 713

Помимо роли фабрик классы позволяют нам вводить новые компоненты (за счет подклассов), а не изменением существующего кода. Объекты экземпляров, созданнеы из класса, наследуют атрибуты класса. Классы могут наследовать др. классы, что открывает нам путь к иерархии классов, поведение которых специализируется за счет переопределения обощенных атрибутов, находящихся выше в дереве классов, подклассами, находящимися ниже. Т.е. чем ниже подклассы тем они спецрализированнее.

В Питон экземпляры наследуют классы, а классы наследуют суперклассы. Опишем основные идеи, лежащие в основе механизма наследования атрибутов:

- **Суперклассы перечисляются в круглых скобках в заголовке инструкции class.** Т.е., чтобы унаследовать атрибуты класса нужно указать его в кругл. скобках в заголовке инструкции class. Наследующий класс наз-ся подклассом, а наследуемый - суперклассом.

- **Классы наследуют атрибуты своих суперклассов.** Как экземпляры наследуют атрибуты своих классов, так и классы наследуют все атрибуты, определяемые в суперклассах.

- **Экземпляры** **наследуют атрибуты всех доступных классов.** Экземпляр наследует атрибуты своего класса, а также всех суперклассов. Поиск имен сначала проверяет сам экземпляр, потом его класс и потом все суперклассы.

- **Каждое обращение object.attribute вызывает новый независимый поиск.** Интерпретатор проводит отдельный поиск в дереве классов для каждого атрибута, который ему встречается в выражении запроса. Сюда входят ссылки на экземпляры и классы из инструкции class (напр. X.attr), а также ссылки на атрибуты аргумента экземпляра self в методах класса. Каждое выражение self.attr в методе вызывает поиск attr в self и выше.

- **Изменения в подклассах не затрагивают суперклассы.** Замещение имен суперкласса в подклассах изменяет подклассы и тем самым изменяет унаследованное поведение.

Классы обеспечивают разложение на отдельные операции и адаптацию кода лучше, чем это могут сделать др. компоненты языка. Классы позволяют минимизировать избыточность программ. кода, а с др. - позволяют адаптировать уже существующий код, вместо его изменения.

**Второй пример.**

Итак, второй пример будет основан на предыдущем. Определим новый класс SecondClass, который наследует класс FirstClass и добавляет кое-что своё:

>>> class SecondClass(FirstClass): # Наследует метод setdata

def display(self): # Изменяет метод display print('Current value = "%s"' % self.data)

- т.о. SecondClass замещает атрибут display своего суперкласса.

В этом случае мы говорим, что класс SecondClass переопределяет метод display класса FirstClass. Иногда такая замена атрибутов, за счет переопределения, называется перегрузкой.

Сейчас класс SecondClass специализирует класс FirstClass, изменяя поведение метода display. С др. стороны он наследует метод setdata (как и все экземпляры подкласса). Создадим экземпляр:

>>> z = SecondClass()

>>> z.setdata(42) # Найдет setdata в FirstClass

>>> z.display() # Найдет переопределенный метод в SecondClass

Current value = "42"

- думаю нам понятны наследование и поиск и схема пространства имен в этом примере.

Заметим. Специализация, выполненная в классе SecondClass, находится полностью за пределами класса FirstClass. Т.е. она не оказывает влияние на уже созданнеы или создаваемые впоследствии объекты класса FirstClass, напр. х:

>>> x.display() # Старое значение х (см. выше)

New value

Т.о. мы не изменяли класс FirstClass, мы адаптировали его. Этот пример показывает нам как можно вносить изменения с пом. внеш. компонентов (подклассов).

**Классы - это атрибуты в модулях.**

Напомним, что в именах классов нет ничего необычного, это всего лишь переменные, которым присваиваются объекты во время выполнения инструкции class, а ссылки на объекты можно получить с пом. обычных выражений.

Напр. если бы определение класса FirstClass находилось в файле модуля, а не было введено в оболочке, мы могли бы импортировать этот модуль и использовать имя в строке заголовка инструкции class:

from modulename import FirstClass # Скопировать имя в мою область видимости

class SecondClass(FirstClass): # Использовать имя класса непосредственно

def display(self): ...

Или эквивалентный вариант:

import modulename # Доступ ко всему модулю целиком

class SecondClass(modulename.FirstClass): # Указать полное имя

def display(self): ...

Имена классов, как и всё остальное, всегда нах-ся в модулях, и поэтому при их употреблении необходимо следовать правилам, которые мы рассматривали в пятой части книги. Напр. в одном модуле могут нах-ся определения сразу нескольких классов - подобно другим инструкциям в модулях, инструкции class выполняются в ходе операции импорта и определяют имена, которые в свою очередь становятся атрибутами модуля.

Вообще любой модуль может содержать самые произвольные сочетания из любого числа переменных, ф-ций и классов, и все эти имена будут вести себя в модуле одинаково. Рассмотрим файл food.py:

# food.py

var = 1 # food.var

def func(): # food.func

...

class spam: # food.spam

...

class ham: # food.ham

...

class eggs: # food.eggs

...

Это правило остается справедливым даже если модуль и класс имеют одинаковые имена. Например имеется файл person.py:

class person:

...

Чтобы получить доступ к классу нам нужно обратиться к модулю как обычно:

import person # Импортируем модуль

x = person.person() # Класс внутри модуля

Имя person.person ссылается на класс person внутри модуля person, если мы просто будем использ-ть имя person, то обратимся к модулю, а не классу, но только если мы не использовали from:

from person import person

x = person()

Как и любые др. переменные, мы не увидим класс в файле модуля, пока не импортируем его или как-то не извлечем класс из вмещающего файла. Если для вас это сложно, то не используйте одинаковые имена для модулей и классов.

Согласно соглашениям, имена классов в Питон должны начинаться с заглавной буквы, для визуального отличия:

import person

x = person.Person()

И еще, несмотря на то, что классы и модули явл-ся пространствами имен для подключения атрибутов, они представляют собой совершенно разные структуры: модуль это отражение целого файла, а класс - лишь инструкция внутри файла.

**Классы могут переопределять операторы языка Питон.** стр. 717

Перегрузка операторов позволяет объектам, созданным из классов, перехватывать и участвовать в операциях, которые применяются к встроенным типам: сложение, получение среза, вывод, квалификация имен и т.д.

Можно полностью реализовать класс в виде методов, однако перегрузка операторов позволяет объектам теснее интегрироваться в объектную модель Питона. Кроме того перегрузка позволяет нашим объектам действовать так же, как действуют встр. объекты, потому что она позволяет создавать менее противоречивые и более простые в изучении интерфейсы объектов и обеспечивает возможность обрабатывать объекты, созданные из классов, программным кодом, который предполагает взаимодействие со встр. типами.

Опишем основные идеи, лежащие в основе мехханизма перегрузки операторов:

- **Имена методов, начинающиеся и заканчивающиеся двумя символами подчеркивания (\_\_Х\_\_), имеют специальное назначение.** Перегрузка реализуется за счет создания методов со спец. именами для перехватывания операций. Питон определяет фиксированные и неизменяемые имена методов для каждой из операций.

- **Такие методы вызываются автоматически, когда экземпляр участвует во встроенных операциях.** Напр. если объект экземпляра наследует метод \_\_add\_\_, этот метод будет вызываться всякий раз, когда объект будет появл-ся в операции сложения (+). Возвращаемое значение метода становится результатом соотв. операции.

- **Классы могут переопределять большинство встроенных операторов.** Есть много спец. имен методов для реализации перегрузки почти всех операторов встр. типов. Это операторы выражений, базовые операции, такие как вывод и создание объекта.

- **В методах перегрузки операторов не существует аргументов со значениями по умолчанию, и ни один из таких методов не явл-ся обязательным для реализации.** Если класс не определяе ти не наследует методы перегрузки операторов, то это значит, что экземпляры класса не поддерживают эти операции. Напр. если нет метода \_\_add\_\_, попытка выполнить операцию + приведет к ошибке.

- **Операторы позволяют интегрировать классы в объектную модель Питона.** Благодаря перегрузке операторов, объекты, реализованные на базе классов, действуют подобно встр. типам и тем самым обеспечивают непротиворечивые и совместимые интерфейсы.

Вообще перегрузка не явл-ся обязательной функциональной особенностью, если она не нужна в вашем классе, то и хорошо, используйте стандартные методы. Напр. приложению работающему с БД не нужны такие операторы как \* и + (нам нечего умножать и складывать в БД, со всеми задачами справятся стандартные методы работы с БД).

Мы не будем изучать перегрузку каждого оператора, доступного в Питон. Но есть один метод перегрузки оператора, который можно встретить практически в любом классе: метод \_\_init\_\_, который известен как конструктор и использ-ся для инициализации состояния объектов. Наряду с self он явл-ся одним из ключей понимания ООП в Питон.

**Третий пример.**

Определим подкласс, производный от SecondClass и реализующий три спец. метода, которые будут вызываться интерпретатором автоматически.

И не путайте имя метода \_\_init\_\_ с файлом \_\_init\_\_.py в пакетах модулей!!!

- Метод \_\_init\_\_ вызывается, когда создается новый объект экземпляра (аргумент self представляет новый объект ThirdClass).

- Метод \_\_add\_\_ вызывается, когда экземпляр ThirdClass участвует в операции +.

- Метод \_\_str\_\_ вызывается при выводе объекта (точнее когда он преобразуется в строку для вывода вызовом встр. ф-ции str или её эквивалентом внутри интерпретатора).

Новый подкласс также определяет метод mul, котоырй изменяет сам объект в памяти. Приступим к испытаниям:

class ThirdClass(SecondClass): # Наследует SecondClass

def \_\_init\_\_(self, value): # Вызывается из ThirdClass(value)

self.data = value

def \_\_add\_\_(self, other): # Для выражения "self + other"

return ThirdClass(self.data + other)

def \_\_str\_\_(self): # Вызывается из print(self), str()

return '[ThirdClass: %s]' % self.data

def mul(self, other): # Изменяет сам объект: обычный метод

self.data \*= other

>>> a = ThirdClass("abc") # Вызывается новый метод \_\_init\_\_

>>> a.display() # Унаследованный метод

Current value = "abc"

>>> print(a) # \_\_str\_\_ : возвращает строку

[ThirdClass: abc]

>>> b = a + 'xyz' # Новый \_\_add\_\_ : создается новый экземпляр

>>> b.display()

Current value = "abcxyz"

>>> print(b) # \_\_str\_\_ : возвращает строку

[ThirdClass: abcxyz]

>>> a.mul(3) # mul: изменяет сам экземпляр

>>> print(a)

[ThirdClass: abcabcabc]

Класс ThirdClass "наследует" класс SecondClass, поэтмоу ег оэкземпляры наследуют метод display от SecondClass. Но теперь, при создании экземпляра ThirdClass ему передается доп. аргумент (напр. "abc"). Это значение передается конструктору \_\_init\_\_ в аргументе value, где присв-ся атрибуту self.data. В итоге при создании экземпляра класса ThirdClass значение атрибута data устанавливается автоматически, благодаря чему отпадает необходимость вызывать метод setdata после создания экземпляра.

Далее, объекты ThirdClass могут участвовать в операциях + и в вызовах ф-ции print. При операции сложения (+) объект экземпляра слева от оператора передается методу \_\_add\_\_ в виде аргумента self, а значение справа - в виде аргумента other.

Т.е. механизм перегрузки операторов отображает операторы выражений и другие встр. операции, выполняемые над экземплярами классов, на спец. имена методов в классе. Здесь у нас использ-е оператора "+" приводит к вызову метода \_\_add\_\_.

Как видим в метод \_\_add\_\_ передается два аргумента и поэтому, из выражения вида a + "xyz", объект экземпляра (self) а и аргумент (other) "xyz" складываются. И получается строка. Т.о. независимо от того что вернет метод \_\_add\_\_, это значение будет интерпретироваться как результат операции сложения.

Когда объект будет участвовать в вызове print(), интерпретатор вызывает метод \_\_str\_\_ объекта и передает ему сам объект - любая строка, которую вернет метод, будет расцениваться, как строковое представление объекта для вывода. Переопределив метод \_\_str\_\_, мы получим возможность использ. ф-цию print для отображения объектиов этого класса, вместо того, чтобы вызывать метод display.

Методы со псец. именами наследуются подклассами и экземплярами, как любые др. имена, которые присв-ся в инструкции class. Если методы отсутствуют в классе идет их поиск в суперклассах. Имена методов пепегрузки не явл-ся встроенными именами или зарезервированными словами - это обычные атрибуты, которые отыскиваются интерпретатором в случае появления объектов в различ. контекстах. Обычно интерпретатор вызывает их автоматич., но они могут вызываться и программным кодом - метод \_\_init\_\_, часто взывается вручную, чтобы запустить конструктор суперкласса.

Внимание! Посмотрите, что метод \_\_add\_\_ создает и возвращает НОВЫЙ объект экземпляра этого класса (вызывая ThirdClass, которому передается значение результата), а метод mul ИЗМЕНЯЕТ текущий объект экземпляра (выполняя присваивание атрибуту аргумента self). Т.к. перегрузка - это всего лишь механизм отображения выражений на методы, мы можем интерпретировать операторы в своих объектах классов, как нам угодно.

**Когда следует использовать перегрузку операторов?**

Конечно использовать или нет перегрузку мы определяем сами. Выбор зависит от того, насколько близко наш класс должен имитировать поведение встр. типов. Как мы знаем, если метод перегрузки оператора отсутствует в определении класса и не наследуется из суперкласса, соотв. операция не будет поддерживаться экземплярами этого класса - при попытке вып-ть такую операцию мы получим исключение (или действие по умолч.).

Вообще перегрузка операторов чаще использ. при реализации объектов с матем. природой - класс вектора или матрицы (напр. перегрузка сложения). Для упрощения классов можно не использ. перегрузку, а использ. явные методы.

С др. стороны мы можем использовать перегрузку операторов, чтобы иметь возможность передавать объекты, определяемые пользователем, в ф-цию, которая вып-ет операции, поддерживаемые встр. типами (списки или словари напр.). Наличие того же набора операторов в нашем классе обеспечит поддержку нашими объектами тех же самых интерфейсов и т.о. совместимость с используемой ф-цией. В гл. 29 мы ознакомимся с длоп. приемами перегрузки операторов в примерах.

Метод-конструктор \_\_init\_\_ присутствует почти во всех реалистичных классах. Он позволяет классам немедленно заполнять атрибуты вновь созданных экземпляров, поэтому конструктор полезно использ. практически во всех разновидностях наших классов. Фактически, даже при том, что в Питон атрибуты не оьновляются, мы сможем без труда определить какие атрибуты принадлежат экземплярам, просмотрев программ. код метода \_\_init\_\_.

**Самый простой в мире класс на языке Python.**

Приступим к детальному изучению синтаксиса инструкции class, и еще раз вспомним, что модель наследования, на которой основаны классы, очень проста - суть её заключается в организации поиска атрибутов в деревьях взаимосвязанных объектов. Фактически можно создавать вообще пустые классы. Создадим класс без атрибутов (объект пустого пространства имен):

>>> class rec: pass

- вот нам и пригодилась пустая операция pass.

Теперь можем начинать присоединять атрибуты выполняя присваивание за пределами класса:

>>> rec.name = 'Bob'

>>> rec.age = 40

Когда атрибуты созданы можно обращаться к ним с пом. обычного синтаксиса. Классы - это полноценные объекты, даже если нет ни одного экземпляра. Фактич. они лишь самостоятельные пространства имен, поэтмоу пока есть ссылка на класс, мы можем в любое время добавлять или изменять его атрибуты по своему усмотрению. Посмотрим, что будет если создать два экземпляра класса:

>>> y = rec() # Экземпляры наследуют имена из класса

>>> x = rec()

Они начинают свое существование как объекты абсолютно пустых пространств имен. Однако они наследуют из класса атрибуты:

>>> x.name, y.name

('Bob', 'Bob')

- Эти экземпляры получили атрибут name из класса, своих атрибутов у них пока нет.

Но если сейчас атрибуту экземпляра присвоить др. значение, то будет создан (изменен) атрибут именно этого объекта. Т.е. операция присваивания значеняи атрибуту действует только на тот объект, к которому она прмиеняется. Экземпляр х получает собственный атрибут name, а экземпляр y по-прежнему наследует name от класса:

>>> x.name = 'Sue'

>>> rec.name, x.name, y.name

('Bob', 'Sue', 'Bob')

Фактич. атрибуты объекта пространства имен обычно реализованы в виде словарей, и деревья наследования классов тоже всего лишь словари со ссылками на др. словари.

Напр. в большинстве объектов, созданных на базе классов, есть атрибут \_\_dict\_\_, который явл-ся словарем пространства имен. Пример - порядок следования имен и перечень внутренних имен вида \_\_X\_\_ может изменяться от версии к версии интерпретатора, но имена, которые использовали, присутствует везде:

>>> list(rec.\_\_dict\_\_.keys())

['age', '\_\_dict\_\_', 'name', '\_\_weakref\_\_', '\_\_module\_\_', '\_\_doc\_\_']

>>> list(x.\_\_dict\_\_.keys())

['name']

>>> list(y.\_\_dict\_\_.keys())

[]

Видим, что в классе есть атрибуты age и name, объект х имеет только свой атрибут name, а объект y по-прежнему пуст.

Каждый экземпляр имеет ссылку на свой наследумый класс, она наз-ся \_\_class\_\_ :

>>> x.\_\_class\_\_

<class '\_\_main\_\_.rec'>

Классы также имеют атрибут \_\_bases\_\_ , который представляет собой кортеж его суперклассов:

>>> rec.\_\_bases\_\_

(<class 'object'>,)

>>> ThirdClass.\_\_bases\_\_

(<class '\_\_main\_\_.SecondClass'>,)

Главное, что нужно понимать - модель классов в Питон очень динамична. Классы и экземпляры - это всего лишь объекты пространств имен с атрибутами, создаваемыми на лету присваиванием.

Даже методы, которые созда-ся def, влож. в инструкцию class, могут создаваться независимо от объекта класса. Пример поределения простой ф-ции вне класса:

>>> def upperName(self):

return self.name.upper() # Аргумент self по-прежнему необходим

Это простая ф-ция и она может вызываться как обычная ф-ция, при услови, что объект, получаемый ею, имеет атрибут name (в данном случае имя аргумента self не имеет никакого особого смысла).

>>> upperName(x) # Вызов как обычной ф-ции

'SUE'

Однако, если эту ф-цию присвоить атрибуту нашего класса, она станет методом, вызываемым из любого экземпляра (и через имя самого класса, при условии, что ф-ции будет вручную передан экземпляр):

>>> rec.method = upperName

>>> x.method() # Вызват ьметод для обработки х

'SUE'

>>> y.method() # То же самое, но в self передается y

'BOB'

>>> rec.method(x) # Можно вызвать через имя экземпляра или класса

'SUE'

Обычно заполнение классов произв-ся внутри инструкции class, а атрибуты экземпляров создаются присваиванием значений атрибутам аргумента self в методах. Но это не явл-ся обязательным.

**Классы и словари.** стр. 723

Мы познакомились с основными особенностями модели классов. В гл. 8 мы увидели как можно использ. словари для хранения записей свойств сущностей в программах. Оказывается классы тоже могут играть эту роль - хранить информацию, как словари, но при этом могут включать логику обработки этой информации в методах.

Пример со словарем:

>>> rec = {}

>>> rec['name'] = 'mel'

>>> rec['age'] = 40

>>> rec['job'] = 'trainer/writer'

>>>

>>> print(rec['name'])

mel

А теперь пример записи на основе класса, с заменой ключей атрибутами:

>>> class rec: pass

>>> rec.name = 'mel'

>>> rec.age = 40

>>> rec.job = 'trainer/writer'

>>>

>>> print(rec.age)

40

Создав пустой класс мы заполняем его, присваивая значения его атрибутам.

Этот прием работает, но для каждой отдельной записи придется писать новую инструкцию class. Все-таки более удобно будет создавать экземпляры класса для новых записей:

>>> class rec: pass

>>> pers1 = rec()

>>> pers1.name = 'mel'

>>> pers1.age = 40

>>> pers1.job = 'trainer'

>>>

>>> pers2 = rec()

>>> pers2.name = 'dave'

>>> pers2.job = 'developer'

>>>

>>> pers1.name, pers2.name

('mel', 'dave')

Два отдельных объекта экземпляра и, соответственно, два разных атрибута name. И даже кол-во атрибутов у экземпляров может быть разным (напр. age). Т.о. каждый экземпляр имеет своё пространство имен, т.е. свой словарь атрибутов.

Мы могли бы создать более полноценный класс:

>>> class Person:

def \_\_init\_\_(self, name, job):

self.name = name

self.job = job

def info(self):

return (self.name, self.job)

>>> rec1 = Person('mel', 'trainer')

>>> rec2 = Person('vls', 'developer')

>>>

>>> rec1.job, rec2.info()

('trainer', ('vls', 'developer'))

В этот класс мы добавили логику (методы) инициализации экземпляров на этапе создания и сбора атрибутов в кортеж. Конструктор налагает некоторые ограничения целостности, требуя значения для двух атрибутов - name и job. Методы класса и атрибуты экземпляра вместе образуют пакет, объединяющий данные и логику.

Классы - это гибкость и универсальность, недоступные встр. типам и функциям.

**Глава 27. Более реалистичный пример.** стр. 728

В этой главе мы создадим множество классов, решающих конкретные задачи - сохранение и обработку инфомрации о людях. Те компоненты, которые мы называем экземпляры и классы, часто могут играть роль записей и программ в традиционном понимании.

Мы создадим два класса:

- Person - класс, который предоставляет и обрабатывает информацию о людях;

- Manager - адаптированная версия (помним об ООП - адаптация кода) класса Person, модифицирующая унаследованное поведение.

Создадим экземпляры обоих классов и протестируем их. Увидим пример - мы сохраним наши экземпляры в хранилище, в объектно-ориентированной базе данных, обеспечивающей долговременное хранение. Этот шаблон даже можно будет применять для создания своей БД на Питон.

И самое главное - обучение. Мы будем продолжать осваивать основы, постепенно будем создавать классы, чтобы увидеть как из отдельных особенностей составляются законченные программы.

Всё это позволить нам изучить и рассмотреть все основные идеи модели ООП в Питон. И в итоге поймем, что система классов в Питоне, сводится к поиску атрибутов в дереве объектов и наличию спец. первого аргумента (self) в функциях классов.

**Шаг 1: создание экземпляров.** стр. 729

Начнем создание главного класса Person. Создадим новый файл где будем писать наш код. В Питон есть соглашение, согласно которому имена модулей начинаются со строчной буквы, а имена классов - с прописной. Так, в соотв. с соглаш. первому аргументу методов класса присв-ся имя self. Эти соглаш. не обьязательны, но общеприняты. Итак:

# Файл person.py (начало)

class Person:

Мы можем написать в одном файле много ф-ций и классов, поэтому название person.py может потерять смысл, если позже мы добавим в него доп. компоненты, никак не связанные с его начальным предназначением.

Но пока будет предполагать, что всё, что нах-ся в этом файле так или иначе связано с классом Person. В идеале так и должно быть - ведь мы знаем, что модуль только выигрывает, когда он создан ради единственной, логически связанной цели.

**Конструкторы.**

Первое, что нам нужно сделать в классе Person - записать основные сведения о человеке, т.е. заполнить поля записи. В Питон они наз-ся атрибутами объекта и обычно создаются с пом. операций присваивания значений атрибутам аргумента self в методах класса. Обычно первые знач. атрибутам присв-ся в методе конструтора \_\_init\_\_, который вызывается автоматически каждый раз, когда созд-ся новый экземпляр. Добавим конструкцию к классу:

# Добавим инициализацию полей записи

class Person:

def \_\_init\_\_(self, name, job, pay): # Конструктор принимает 3 аргумента

self.name = name # Заполняет поля при создании

self.job = job # self - новый экземпляр класса

self.pay = pay

Итак, мы передаем конструктору (метод \_\_init\_\_) аргументы с данными, которые будут храниться экземпляром, и присваиваем их атрибутам аргумента self.

В терминах ООП аргумент self представляет вновь созданный экземпляр, а аргументы name, job и pay превращаются в информацию о состоянии - данные, сохраняемые в объекте для последующего использования.

Видим, что имена аргументов использ-ся дважды в операциях присваивания. Это может показаться избыточным, но это не так. Аргумент job, например, - это локальная переменная в области видимости ф-ции \_\_init\_\_, а self.job - это атрибут экземпляра, который явл-ся подразумеваемым контекстом вызова метода. Т.е. это две разные переменные, которые по совпадению имеют одно и то же имя.

Присваивая знач. локал. переменной job атрибуту self.job (с пом. операции self.job = job) мы сохраняем его в экземпляре для последующего использования.

Метод \_\_init\_\_ обычен, кроме того, что он вызывается автоматич. при создании экземпляра и его первый аргумент имеет спец. значение. А так это самая обычная ф-ция, т.е. мы можем определять знач. аргументов по умолчанию, например.

Пример, сделаем job необязательным аргументом - знач. по умолч. будет None (т.е. безработный). И если job = None, то аргумент pay (зарплата) должен быть равен 0 (безработные не получают з\п). На самом деле мы вынуждены указать знач. по умолч. для pay, т.к. это требует синтаксис Питона - любые аргументы в заголовке ф-ции, следующие за первым аргументом, имеющим знач. по умолч., также должны иметь знач. по умолч.:

# Добавим значения по умолчанию для аргументов конструктора

class Person:

def \_\_init\_\_(self, name, job=None, pay=0):

self.name = name

self.job = job

self.pay = pay

Теперь, при создании экземпляров, нам достаточно передать только аргумент name, остальные будут по умолч., если их не указать явно.

**Тестирование в процессе разработки.**

Как видим, программирование на Питон сводится к постепенному наращиванию возможностей - т.е. мы пишем код, тестируем его, потом добавляем еще, снова тестируем и так далее. Т.е. лучше тестировать понемногу, а не всё сразу.

Протестируем наш класс, создав пару экземпляров и просмотрев их атрибуты. Это можно сделать в интерактив. сеансе, но это утомительно, набирать кучу инструкций. Для полного тестирования лучше добавлять программ. код в конец файла, содержащего тестируемые объекты:

# Добавляем код для самопроверки

class Person:

def \_\_init\_\_(self, name, job=None, pay=0):

self.name = name

self.job = job

self.pay = pay

bob = Person('Bob Smith')

sue = Person('Sue Jones', job='dev', pay=100000)

print(bob.name, bob.pay)

print(sue.name, sue.pay)

- Тестирование класса. Запустит \_\_init\_\_ автоматически, извлечет атрибуты, увидим, что атрибуты в объектах sue и bob отличаются.

Итак, объект bob получает знач. атрибутов по умолч., а для объекта sue знач. всех атрибутов указываются явно. Причем мы используем для sue именованные аргументы, хотя могли бы и позиционные, однако именованные информативнее и их можно указывать в любом порядке.

Еще раз видим, что \_\_init\_\_ это обычная ф-ция, поддерживающая все особенности ф-ций. Если запустить файл как сценарий, то увидим:

user@home:~$ python3.4 person.py

Bob Smith 0

Sue Jones 100000

Примечание!!! В gvim я поставил табуляцию = 4 (я так понимаю 1 таб = 4 пробела), ну это для удобства вида просто, а в терминале получил ошибку:

user@home:~$ python3.4 person.py

File "person.py", line 5

self.name = name

^

TabError: inconsistent use of tabs and spaces in indentation

- переводится как "Смешение пробелов и табуляции в отступах.", пришлось переоткрывать файл заново и ставить обычные табы (они длинные и некрасиво выглядит, но зато работает).

Посмотрел непечатаемые знаки и вижу, что двойные Табы отображаются как Табы, а одинарный (где def) не отображается как Таб, видимо просто как 4 пробела, почему не знаю, нужно все-таки заняться изучением vim, а не только Питона. Нужно изучить этот вопрос, иначе будут некрасивые лестницы отступов в коде. Ну или пока так попробовать..

Каждый из объектов bob и sue явл-ся независимыми записями с собств. информацией. Технич. они явл-ся пространствами имен - как и все экземпляры классов, каждый из них обладает собств. копией информации о состоянии. Т.о. как и встр. типы, классы играют роль своеобразной фабрики объектов.

**Двоякое использование программного кода.**

Тесты работают без проблем, но есть одно но - инструкции print будут выполняться и при запуске файла как сценария и при импорте как модуля. Это не очень хорошо, т.к. клиентской проге не зачем знать как выглядят наши тесты и вообще тесты не нужны.

Конечно можно тесты убрать в отдельный файл, однако удобнее когда тесты нах-ся в том же файле, что и тестируемый код (может это соглашения Питона, в Руби мы писали тесты отдельным файлом, напр., надеюсь дальше будет понятно, или потом на практике). Но лучше оформим тесты, чтобы они выполнялись когда файл запускается как сценарий, а не при импорте. А поможет нам проверка атрибута \_\_name\_\_ модуля:

# Предусмотрим возможность импорта файла и запуска его,

# как самостоятельный сценарий

class Person:

def \_\_init\_\_(self, name, job=None, pay=0):

self.name = name

self.job = job

self.pay = pay

if \_\_name\_\_ == '\_\_main\_\_': # Только когда файл запускается как сценарий

bob = Person('Bob Smith')

sue = Person('Sue Jones', job='dev', pay=100000)

print(bob.name, bob.pay)

print(sue.name, sue.pay)

Теперь проверим работает ли тестирование тогда когда нам нужно:

C:\Python34>person.py

Bob Smith 0

Sue Jones 100000

>>> import person # Всё работает как надо, при импорте тестов нет

>>>

**Замечание.** Я снова наступаю на те же грабли, что и всегда.

После импорта модуля с классом Person, я попытался создать экземпляр класса и потерпел неудачу:

>>> import person

>>> bob = Person('Bob')

...

NameError: name 'Person' is not defined

Интерпретатор выдавал мне сообщение о том, что имя Person не существует, хотя в атрибутах модуля он был:

>>> list(person.\_\_dict\_\_.keys())

['\_\_spec\_\_', 'Person', '\_\_cached\_\_', '\_\_loader\_\_', '\_\_package\_\_', '\_\_builtins\_\_', '\_\_doc\_\_', '\_\_file\_\_', '\_\_name\_\_']

В итоге, в результате долгих поисков проблема была определена - оказывается я напрасно вызывал таким образом класс, потому что его НУЖНО было вызывать вот так:

>>> import person

>>> bob = person.Person('Bob Dilan')

>>> bob.name

'Bob Dilan'

А если мы хотим вызывать так как делали в прошлых примерах, когда классы мы создавали ПРЯМО В оболочке, то нужно импортировать класс Person вот ТАКИМ способом, через from:

>>> from person import Person

>>> bob = Person('Bob')

>>> bob.name

'Bob'

Очень ужасная забывчивость и невнимательность у меня, нужно с ней обязательно бороться, с такими ошибками в программисты не берут.

**Шаг 2: добавление методов, определяющих поведение.** Стр. 733

Итак, сейчас наш класс играет роль фабрики записей - создает записи и заполняет их поля (атрибуты экземпляров). Теперь мы уже можем применять различ. операции к этим записям.

Большую часть свой работы классы выполняют за счет внедрения и обработки данных базовых типов, таких как списки и строки. Т.е. если мы знаем как работать в базовыми типами, то и с классами разберемся - классы явл-ся лишь небольшой структурной надстрокой.

Напр. поле name в наших объектах явл-ся обычной строкой, поэтому мы можем извлекать фамилии разбивая значение по пробелам и индексированием, т.е. операциями над базовыми типами:

>>> name = 'Bob Smith' # Простая строка

>>> name.split() # Извлечение фамилии

['Bob', 'Smith']

>>> name.split()[-1] # Или и [1] если имя всегда состоит из двух компонентов

'Smith'

Также можем работать и с зарплатой, изменяя её состояние с пом. присваивания:

>>> pay = 100000 # Простая переменная за пределами класса

>>> pay \*= 1.10 # Поднимем з\п на 10% или pay = pay \* 1.10

>>> print(pay)

110000.00000000001

Чтобы применить эти операции к объектам класса Person, просто подставьте имена bob.name и sue.name на место name и pay, т.е. операции будут проводиться над атрибутами класса:

# Обработка встроенных типов: строки, изменяемость

class Person:

def \_\_init\_\_(self, name, job=None, pay=0):

self.name = name

self.job = job

self.pay = pay

if \_\_name\_\_ == '\_\_main\_\_':

bob = Person('Bob Smith')

sue = Person('Sue Jones', job='dev', pay=100000)

print(bob.name, bob.pay)

print(sue.name, sue.pay)

print(bob.name.split()[-1]) # Извлечь фамилию

sue.pay \*= 1.10 # Повысить з\п

print(sue.pay)

- в предпоследней строке мы изменяем значение атрибута pay у объекта sue. В некотором смысле объект sue явл-ся изменяемым - он допускает непосредственное изменение свой информации о состоянии, подобно списку при вызове метода append:

C:\Python34>person.py

Bob Smith 0

Sue Jones 100000

Smith

110000.00000000001

Опытный программист подскажет нам, что такой подход не желателен на практике. Выполнение операций за пределами класса, как у нас, может привести к проблемам при сопровождении.

Представим, что в разных местах программыесть одинаковые фрагменты, извлекающие фамилию. Если нам нужно будет изменить их, то придется искать и изменять ВСЕ такие фрагменты. То же самое относится и к операцией с зарплатой. Один только поиск всех фрагментов может быть проблематичным.

**Методы реализации.**

Приступаем к реализации концепции под названием - инкапсуляция. Её идея в том, чтобы спрятать логику операций за интерфейсами, и т.о. каждая операция будет иметь единственную реализацию в программе. Если в будущем нам нужно будет внести изменения в логику, нам нужно будет модифицировать код всего лишь в одном месте, а не по всей программе.

Еще плюс - можно будет изменять реализацию операции как угодно (рефакторинг, оптимизация), не рискуя нарушить работу кода, и опять же - всего лишь в одном месте, а не по всей программе.

В терминах Питона это означает, что мы должны реализовать операции над объектами в виде методов класса, а не разбрасывать их по всей программе.

Простыми словами опишем - напр. нам нужно переключить скорость передачи в авто, т.е. провести операцию "переключение скорости", для этого у нас есть метод "перемещение рычага КПП" и нас совсем не интересует, что будет делать рычаг КПП внутри самой коробки передач, движения шестерен внутри, валдов и прочее и прочее. У фирмы БМВ масса автомобилей различных моделей и размеров, но концепция операция-метод-реализация остается, для всего многообразия автомобилей, одинакова. Вот это простое описание инкапсуляции.

Вернемся к программированию. Сосредоточив код реализации в одном месте мы устраняем избыточность и упрощаем его сопровождение, что явл-ся одной из сильных сторон классов. Оформление операций в виде методов позволяет применять их к любым экземплярам класса, а не только к тем, которые явно задействованы в обработке.

К практике. Мы переместим реализацию двух операций из программы в методы класса, добившись инкапсуляции и изменим код самопроверки:

# Добавлены методы, инкапсулирующие операции

class Person:

def \_\_init\_\_(self, name, job=None, pay=0):

self.name = name

self.job = job

self.pay = pay

def lastName(self): # Два метода, реализующие поведение экземпляров

return self.name.split()[-1] # self - подразумеваемый экземпляр

def giveRaise(self, percent):

self.pay = int(self.pay \* (1 + percent)) # если "прибавка к з\п" изменится

# то изменять нам только один метод

if \_\_name\_\_ == '\_\_main\_\_':

bob = Person('Bob Smith')

sue = Person('Sue Jones', job='dev', pay=100000)

print(bob.name, bob.pay)

print(sue.name, sue.pay)

print(bob.lastName(), sue.lastName()) # Вместо жестко определенных операций

sue.giveRaise(.10) # используем методы

print(sue.pay)

- give - давать; Raise - повышение, подъем; percent - процент. Т.е. метод giveRaise дает нам з\п служащего с надбавкой, если она есть (у Sue мы её сделали 10%).

И еще была синтаксическая ошибка, в тестах написал метод lastName с большой буквы "sue.LastName" - внимательнее будьте!

Как мы знаем методы - это обычные ф-ции, которые присоед-ся к классам и предназначены для обработки экземпляров этих классов. Экземпляр - это подразумеваемый контекст вызова метода, который автоматич. передается в виде аргумента self.

С методом giveRaise мы уже разобрались, он выполняет операцию над объектом self (по увеличению з\п, если конечно передавать положительное число, не отрицательное и не 0).

Метод lastName возвращает результат, это фактич. вызов ф-ции - вычисляет значение, которое затем может использ-ся вызывающей программой (у нас рез-т просто выводится для обзора).

Рез-ты сценария будут те же, что и прежде - т.к. мы всего лишь реструктурировали программный код, не изменяя его поведения:

C:\Python34>person.py

Bob Smith 0

Sue Jones 100000

Smith Jones

110000

Обсудим несколько моментов нашего нового кода:

- Первое. Обратим внимание, что поле pay (зарплата) в объекте sue получает целочисленное значение после его увеличения - внутри мы преобразовываем рез-т в целое число с пом. ф-ции int. Конечно нужно самим определять как лучше - int или float, нужно ли округлять или нет.

К тому же мы могли округлить до центов с пом. ф-ции round(N, 2), использовать тип decimal или хранить в виде веществ. чисел и отображать с пом. стркои формата %.2f или {0:.2f}. Еще один способ - можно импортировать ф-цию money и отображать сумму з\п с запятыми, центами и знаком доллара (см. модуль formats.py в гл. 24).

- Второе. Мы добавили вывод фамилии из объекта sue - т.к. теперь логика получения фамилии инкапсулирована в методе и можем применить её к любому экземпляру класса. Интерпретатор сам сообщает методу, какой экземпляр должен обрабатываться, автоматич. передавая его в первом аргументе (self).

Одно слабое место у нас есть, - это нулевая з\п в объекте bob, к нему мы не можем применть метод giveRaise, т.к. в итоге получим тот же 0, а не увеличение з\п.

Ну и как говорилось мной выше - у нас просто подразумевается, что в percent передаются нормальные цифры процентов, но в этом месте все-таки желательна бы проверка значения (мало ли, отрицательное число будет или вообще 1000%). В одной из глав мы разберем способы решения этой проблемы, когда будем изучать декораторы функций и исследовать инструкцию assert - альтернативы, позволяющие реализовать автоматич. проверку в ходе разработки.

**Шаг 3: перегрузка операторов.** стр. 737

Итак, у нас есть класс, который может инициализировать экземпляры и обладает методами.

Но есть моменты по тестам - нам приходится вручную извлекать и выводить значения отдельных атрибутов (bob.ame, sue.name). Хотелось бы , чтобы вывод экземпляра целиком показывал нам нужную информацию.

Но формат вывода объектов экземпляров по умолчанию, выглядит неудобно - выводит имя класса и его адрес памяти (зачем он нам?). Чтобы это увидеть измените последнюю строку на вызов print(sue):

C:\Python34>person.py

Bob Smith 0

Sue Jones 100000

Smith Jones

<\_\_main\_\_.Person object at 0x0075F7B0>

**Реализация отображения.**

В помощь нам придет перегрузка операторов - добавит ьв класс метод, который перехватывает и выполняет встр. операцию, когда она применяется к экземплярам класса.

Нам поможет популярный метод (вторйо после \_\_init\_\_) - метод \_\_str\_\_ (см. пред. главу). Метод \_\_str\_\_ вызывается всякий раз, когда экземпляр преобраз-ся в строку для вывода. Т.о. фактич. всё, что мы получаем при выводе объекта, явл-ся возвращаемым значением метода \_\_str\_\_ этого объекта, который может быть определен в классе объекта или унаследован от суперкласса (методы с подчеркиваниями наследуются так же как любые другие).

Метод \_\_str\_\_ перехватывает определнную операцию и вып-ет специфическую реализацию поведения объекта, участвующего в операции. Добавим его в наш класс:

# Добавлен метод \_\_str\_\_, реализующий вывод объектов

class Person:

def \_\_init\_\_(self, name, job=None, pay=0):

self.name = name

self.job = job

self.pay = pay

def lastName(self):

return self.name.split()[-1]

def giveRaise(self, percent):

self.pay = int(self.pay \* (1 + percent))

def \_\_str\_\_(self): # Новый спец. метод

return '[Person: %s, %s]' % (self.name, self.pay) # Строка для вывода

if \_\_name\_\_ == '\_\_main\_\_':

bob = Person('Bob Smith')

sue = Person('Sue Jones', job='dev', pay=100000)

print(bob)

print(sue)

print(bob.lastName(), sue.lastName())

sue.giveRaise(.10)

print(sue)

В методе \_\_str\_\_ в строке вывода мы применили оператор форматирования % - для реализации нужных действий классы могут использ-ть встр. типы объектов и операции. Т.е. запомните - всё, что мы знаем о встр. типах данных и ф-циях, может прим-ся при создании классов. По сути классы лишь добавляют структурный уровень, организующий ф-ции и данные в виде единого объекта и имеющий возможность расширения.

Мы также изменили тесты - теперь выводятся объекты целиком, а не отдельные объекты, смотрим:

C:\Python34>person.py

[Person: Bob Smith, 0]

[Person: Sue Jones, 100000]

Smith Jones

[Person: Sue Jones, 110000]

Далее мы узнаем, что родственный метод перегрузки \_\_repr\_\_ возвращает представление объекта в виде программ. кода. И иногда классы переопределяют оба метода: \_\_str\_\_ - для удобочитаемого вывода, а \_\_repr\_\_ - для вывода доп. сведений об объектах (для разработчика напр.). Т.к. операция вывода автоматич. вызывает метод \_\_str\_\_, а интерактив. оболочка выводит рез-ты с пом. \_\_repr\_\_, подходящие варианты вывода могут предоставляться обеим категориям клиентов.

**Шаг 4: адаптация поведения с помощью подклассов.** стр. 739

Сейчас у нас есть большинство механизмов ООП Питона: класс создает экземпляры, есть методы обработки и даже операторы перегрузки. Наш класс фактич. объединяет логику и данные в единый самостоятельный программ. компонент, упрощая поиск и модификацию программ. кода. Также имеется инкапсуляция.

Единственно чего мы еще не делали, это - адаптация программного кода за счет наследования. Конечно наши экземпляры наследуют методы своего класса. Но для полноты обучения принципам ООП нам нужно определить отношения типа суперкласс/подкласс. Ведь основная идея ООП - это возможность адаптации уже имеющегося и действующего программ. кода.

**Создание подклассов.**

Адаптируем наш класс Person расширив его подклассом Manager, в котором мы заместим унаследованный метод giveRaise. Сначала определим класс:

class Manager(Person): # Определили подкласс класса Person

Класс Manager наследует и может адаптировать суперкласс Person. В частности подкласс Manager буде треализовать свой спосо бувеличения з\п.

Допустим наш менеджер получает не только прибавку (percent), но и доп. премию, которая по умолч. будет 10%. Напр. если прибавка будет 10%, то увеличение у менеджера будет уже 20%. Наш новый метод замещает и адаптирует оригинальную реализацию в классе Person:

class Manager(Person):

def giveRaise(self, percent, bonus=.10):

**Расширение методов: неправильный способ.**

Неправильный способ заключается в простом копировании реализации метода giveRaise из класса Person и его изменении в классе Manager:

class Manager(Person):

def giveRaise(self, percent, bonus=.10):

self.pay = int(self.pay \* (1 + percent + bonus))

Конечно этот вариант будет работать. Проблема в том, что всякий раз когда вы копируете программ код, вы фактич. усложняете его сопровождение в будущем. Если в будущем у нас появятся изменения в суперклассе в этом методе, то нам придется делать их в двух местах. Это маленький пример, но наглядно демонстрирует эту проблему.

**Расширение методов: правильный способ.**

На самом деле нам нужно лишь дополнить оригинальный метод giveRaise, а не заменить его полностью. Правильный способ состоит в том, чтобы вызвать оригинальную версию с измененными аргументами:

class Manager(Person):

def giveRaise(self, percent, bonus=.10):

Person.giveRaise(self, percent + bonus)

Данная реализация учитывает то обстоятельство, что методы класса могут вызываться либо обращением к экземпляру (обычный способ, когда интерпретатор автоматич. передает экземпляр в аргументе self), либо обращением к классу (менее распространенный метод, когда экземпляр передается вручную). Вспомним, что вызов метода instance.method(args...) автоматич. транслируется интерпретатором в эквивалентную форму:

class.method(instance, args...)

где класс, содержащий вызываемый метод, определяется в соотв. с правилами поиска в дереве наследования, которые действуют и для методов.

В своих сценариях вы можете использовать любую форму вызова, но не забывайте об их различиях - при обращении непосредственно к классу вы должны передавать объект экземпляра вручную. Метод всегда должен получать объект экземпляра тем или иным способом, однако интерпретатор вып-ет автоматич. передачу только при вызове метода через обращение к экземпляру.

При вызове метода через обращение к классу вы сами должны передавать экземпляр в аргументе self - внутри метода, такого как giveRaise, аргумент self уже содержит подразумеваемый объект вызова, т.е. сам экземпляр.

Вызов через обращение к классу начинает поиск от класса и выше по дереву классов. Именно поэтому в классе Manager мы должны вызывать оригинальный метод через класс Person.giveRaise(), т.к. если бы мы вызывали его через self.giveRaise(), то у нас появилась бы рекурсия.

На первый взгляд правильная и неправильная версии мало отличаются, но это отличие может иметь огромное значения для сопровождения в будущем - в нашем случае у нас основная логика работы метода по-прежнему остается в одном месте (представьте сотни таких "двойных" методов в проекте, замучаетесь изменять в случае чего).

Приводим полное содержимое файла модуля после последнего шага:

# Добавлен подкласс, адаптирующий поведение суперкласса

class Person:

def \_\_init\_\_(self, name, job=None, pay=0):

self.name = name

self.job = job

self.pay = pay

def lastName(self):

return self.name.split()[-1]

def giveRaise(self, percent):

self.pay = int(self.pay \* (1 + percent))

def \_\_str\_\_(self):

return '[Person: %s, %s]' % (self.name, self.pay)

class Manager(Person):

def giveRaise(self, percent, bonus=.10):

Person.giveRaise(self, percent + bonus)

if \_\_name\_\_ == '\_\_main\_\_':

bob = Person('Bob Smith')

sue = Person('Sue Jones', job='dev', pay=100000)

print(bob)

print(sue)

print(bob.lastName(), sue.lastName())

sue.giveRaise(.10)

print(sue)

tom = Manager('Tom Jones', 'mgr', 50000) # Экземпляр Manager: \_\_init\_\_

tom.giveRaise(.10) # Вызов адаптированной версии

print(tom.lastName()) # Вызов унаследованного метода

print(tom) # Вызов унаследованного \_\_str\_\_

- мы добавили новый код проверки (последние 4 строки) для наших новых изменений.

[Person: Bob Smith, 0]

[Person: Sue Jones, 100000]

Smith Jones

[Person: Sue Jones, 110000]

Jones

[Person: Tom Jones, 60000] # Видим, что з\п увеличилась не на 10%, как у sue, а на 20%

**Полиморфизм в действии.**

Добавим еще код для еще более полного действования наследования:

if \_\_name\_\_ == '\_\_main\_\_':

...

print('--All three--')

for object in (bob, sue, tom): # Обработка объектов обобщенным способом

object.giveRaise(.10) # Вызовет метод giveRaise этого объекта

print(object) # Вызовет общий метод \_\_str\_\_

Результаты работы:

[Person: Bob Smith, 0]

[Person: Sue Jones, 100000]

Smith Jones

[Person: Sue Jones, 110000]

Jones

[Person: Tom Jones, 60000]

--All three--

[Person: Bob Smith, 0]

[Person: Sue Jones, 121000]

[Person: Tom Jones, 72000]

В добавленном коде переменная object может ссылаться либо на экземпляр класса Person, либо класса Manager, а интерпретатор автоматич. вызовет соотв-щий метод giveRaise - для bob, sue оригинальнй метод из класса Person, а для объекта tom - адаптированная версия из класса Manager.

Этот пример демонстрирует нам понятие полиморфизма в Питон, - действие операции giveRaise зависит от того, к какому объекту она применяется. Именно благодаря полиморфизму, в этом примере sue получает прибавку 10%, а tom - 20%.

С др. стороны, операция вывода вызывает одну и ту же версию метода \_\_str\_\_ для всех трех объектов, потому что в программ. коде присутствует только одна его версия - в классе Person.

**Наследование, адаптация и расширение.**

Классы могут обладать еще большей гибкостью - могут наследовать, адаптировать, расширять реализацию суперклассов. Напр. в нашем подклассе Manager могли бы быть уникальные методы, отсутствующие в классе Person. Пример, появился новый метод someThingElse:

class Person:

def lastName(self): ...

def giveRaise(self, ...): ...

def \_\_str\_\_(self): ...

class Manager(Person): # Наследование

def giveRaise(self, ...): ... # Адаптация

def someThingElse(self, ...) # Расширение

tom = Manager()

tom.lastName() # Унаследованный метод

tom.giveRaise() # Адаптированная версия

tom.someThingElse() # Дополнительный метод

print(tom) # Унаследованный метод перегрузки

Доп. методы, такие как метод someThingElse? расширяют возможности существующего ПО и доступны только для объектов класса Manager. В наших примерах мы такие методы не создавали, но назначение и использование их в общем-то понятно.

**ООП: основная идея.**

Наш небольшой код иллюстрирует основное преимущество ООП: используя объектно-ориентированный стиль, мы адаптируем имеющийся программный код, а не копируем и не изменяем его. В целом применение ООП способно существенно сократить время разработки, по сравнению с др. подходами.

Так в наших примерах мы могли бы реализовать отдельную операцию giveRaise, не прибегая к созданию подкласса, но ни один др. способ не позволил бы нам получить такой же оптимальный код:

- Мы могли бы создать новый, независимый класс Manager, но при этом пришлось бы реализовывать все методы, имеющиеся в классе Person и действующие одинаково в классе Manager.

- Мы могли бы просто изменить класс Person, чтобы удовлетворить требованиям, предъявляемым к методу giveRaise класса Manager, но при этом нарушилась бы корректная работа, там где нужен оригинальный метод класса Person.

- Мы могли бы скопировать класс Person целиком, присвоить копии имя Manager и изменить метод giveRaise, но от этого наш код стал бы избыточным, и это усложнило бы нам задачу - вероятные изменения в будущем в классе Person, не будут отражаться в классе Manager и придется вручную переносить изменения.

Адаптируемые иерархии, на основе классов, обеспечивают оптимальное решение для ПО, которое предполагается развивать в дальнейшем. Адаптация и расширение имеющегося кода с применением ООП - сильный союзник программиста.

**Шаг 5: адаптация конструкторов.** стр. 745

Обратим внимание - кажется бессмысленно указывать значение 'mgr' (менеджер) в аргументе job при создании объекта класса Manager: эта должность уже подразумевается названием класса. Нужно заполнять этот атрибут автоматически.

Для этого адаптируем логику работы конструктора в классе Manager так, чтобы он автоматич. подставлял название должности. Т.е. нам нужно переопределить метод \_\_init\_\_ в классе Manаger, чтобы 'mgr' ставилось автоматич. Как и в случае с giveRaise, нам нужно вызывать оригинальный метод \_\_init\_\_ из класса Person.

Приводим новый вид сценария:

# Добавлен адаптированный конструктор в подкласс

class Person:

def \_\_init\_\_(self, name, job=None, pay=0):

self.name = name

self.job = job

self.pay = pay

def lastName(self):

return self.name.split()[-1]

def giveRaise(self, percent):

self.pay = int(self.pay \* (1 + percent))

def \_\_str\_\_(self):

return '[Person: %s, %s]' % (self.name, self.pay)

class Manager(Person):

def \_\_init\_\_(self, name, pay): # Переопределенный конструктор

Person.\_\_init\_\_(self, name, 'mgr', pay) # Вызов оригинального конструктора

# со значением 'mgr' в аргументе job

def giveRaise(self, percent, bonus=.10):

Person.giveRaise(self, percent + bonus)

if \_\_name\_\_ == '\_\_main\_\_':

bob = Person('Bob Smith')

sue = Person('Sue Jones', job='dev', pay=100000)

print(bob)

print(sue)

print(bob.lastName(), sue.lastName())

sue.giveRaise(.10)

print(sue)

tom = Manager('Tom Jones', 50000) # Указывать должность не требуется:

tom.giveRaise(.10) # Подразумевается/устанавливается

print(tom.lastName()) # классом

print(tom)

Здесь мы используем тот же прием расширения конструктора \_\_init\_\_, который использовали для метода giveRaise. Вызываем метод \_\_init\_\_ класса Person, чтобы задействовать его логику и экземпляры класса Manager получили атрибуты.

Такая форма вызова конструктора суперкласса из конструктора подкласса широко использ-ся в программировании на Питон. Положительная сторона такого ступенчатого вызова методов в том, что мы можем явно передать конструктору суперкласса только необходимые аргументы или вообще не вызывать его: возможность отказа от вызова конструктора суперкласса позволяет полностью заместить логику его работы, а не дополнять её.

Результаты работы сценария не изменились - т.к. мы не изменили логику его работы, лишь реструктурировали код, чтобы избавиться от некоторой избыточности:

[Person: Bob Smith, 0]

[Person: Sue Jones, 100000]

Smith Jones

[Person: Sue Jones, 110000]

Jones

[Person: Tom Jones, 60000]

**ООП проще чем может показаться.**

Наши небольшие классы задействуют почти все наиболее важные концепции механизма ООП в Питон:

- Создание экземпляров - заполнение атрибутов экземпляров.

- Методы, реализующие поведение - инкапсуляция логики в методах класса.

- Перегрузка операторов - реализация поддержки встроенных операций, таких как вывод.

- Адаптация поведения - переопределение специализированных версий методов в подклассах.

- Адаптация конструкторов - добавление логики инициализации, в дополнение к логике суперкласса.

Большинство этих механизмов основаны на: поиске атрибутов в дереве наследования, спец. аргументе self методов и автоматич. выборе нужного метода перегрузки операторов.

Также мы уменьшили избыточность кода и т.о. повысили простоту изменений в будущем - оформили логику работы классов в виде методов и предусмотрели вызовы методов суперкласса.

Конечно есть ряд доп. концепций классов, такие как декораторы и метаклассы. Но что касается основ, то наши кдассы используют все базовые механизмы ООП.

**Другие способы комбинирования классов.**

Отчасти, объединение классов в крупных программах это искусство. Поэтому помимо наследования программисты использ. др. способы комбинирования классов. Напр. часто использ. прием вложения объектов друг в друга для создания составных объектов - к примеру мы могли бы использ. это для создания класса Manager, вложив в него объект класса Person, а не наследуя этот класс.

След. альтернатив. реализация использует метод \_\_getattr\_\_ перегрузки операторов, чтобы перехватывать обращения к несуществующим атрибутам и делегировать эти обращения вложенноу объекту, вызовом встр. ф-ции getattr. Здесь также имеется адаптированная версия метода giveRaise, которая изменяет значение аргумента, передаваемого методу вложенного объекта.

В итоге наш класс Manager превращается в контроллер, который вызывае тметоды вложенного объекта, а не методы суперкласса:

# Альтернативная версия класса Manager

class Person:

... то же самое ...

class Manager(Person):

def \_\_init\_\_(self, name, pay):

self.person = Person(name, 'mgr', pay) # Влож. объект Person

def giveRaise(self, percent, bonus=.10): # Перехватывает и делегирует

self.person.giveRaise(percent + bonus)

def \_\_getattr\_\_(self, attr): # Делегирует обращения

return getattr(self.person, attr) # ко всем остальным атрибутам

def \_\_str\_\_(self):

return str(self.person) # Требуется перегрузка (в 3.0)

if \_\_name\_\_ == '\_\_main\_\_':

... то же самое ...

Этот вариант класса Manager представляет известный шаблон проектирования под названием - делегирование, - составная структура служит оберткой вокруг вложенного объекта, управляет им и перенаправляет ему вызовы методов.

В нашем случае, для конкретно нашего класса, этот прием конечно же не так хорош как механизм наследования. Реализация сложнее и запутаннее, мы привели этот пример лишь для изученяи альтернативы. Конечно делегирование может быть хорошо для других случаев, но не для наших классов.

Использование вложенных объектов будет успешно когда круг взаимодействий контейнера с влож. объектами уже, чем предполагает прием адаптации. Напр. уровень контроллера альтернативного класса Manager может пригодиться для отслеживания и проверки вызовов методов других объектов.

Для объединения других объектов в виде множества можно было бы использовать гипотетический агрегатный класс Department, как будет показано ниже. Для исследования этого класса можно добавить его в конец файла person.py:

if \_\_name\_\_ == '\_\_main\_\_':

bob = Person('Bob Smith')

sue = Person('Sue Jones', job='dev', pay=100000)

print(bob)

print(sue)

tom = Manager('Tom Jones', 50000)

print(tom)

print('--Department--')

class Department:

def \_\_init\_\_(self, \*args):

self.members = list(args)

def addMember(self, person):

self.members.append(person)

def giveRaises(self, percent):

for person in self.members:

person.giveRaise(percent)

def showAll(self):

for person in self.members:

print(person)

development = Department(bob, sue)

development.addMember(tom)

development.giveRaises(.10)

development.showAll()

Результаты вывода:

[Person: Bob Smith, 0]

[Person: Sue Jones, 100000]

[Person: Tom Jones, 50000]

--Department--

[Person: Bob Smith, 0]

[Person: Sue Jones, 110000]

[Person: Tom Jones, 60000]

***Замечание.*** Здесь у нас в выводе везде класс Person, хотя у объекта tom должно быть Manager, сначала хотел реализовать свой метод для класса Manager, но не смог переделать его так же элегантно как метод giveRaise. Пробовал передавать атрибут имени класса объекта, но он выводится в неудобной форме, на этом я оставил попытки реализации своего метода \_\_str\_\_ для класса Manager. Далее в книге будет раскрыт ответ на этот важный вопрос, лично замеченный мной.

В этом примере использ-ся оба приема, наследование и встраивание - объекты класса Department явл-ся составными объектами, которые управляют другими встр. объектами, но сами встр. объекты классов person и Manager используют механизм наследования для адаптации своего поведения.

Еще один пример - графич. интерфейс пользователя, в его реализации для адаптации поведения и внеш. вида кнопок и меток может использ-ся механизм наследования, а для создания пакетов встр. виджетов, таких как формы ввода, калькуляторы, текстовые редакторы, - прием встраивания.

**Перехват обращений к встроенным атрибутам в версии 3.0**

В Питон 3.0 альтернатив. реализация класса Manager, основанная не делегировании, не может перехватывать и делегировать вызовы методов перегрузки операторов, таких как \_\_str\_\_, без их определения. Это общая проблема классов с делегированием.

Напоним, что встр. операции, напр вывод и обращение по индексу, неявно вызывают методы перегрузки операторов, такие как \_\_str\_\_ и \_\_getitem\_\_. В 3.0 подобные операции не использ. менеджеры атрибутов для неявного получения ссылок на атрибуты: они не использ. ни метод \_\_getattr\_\_ (вызывается при обращении к неопределенным атрибутам), ни \_\_getattribute\_\_ (вызывается при обращении к любым атрибутам). Именно поэтому нам пришлось переопределить метод \_\_str\_\_ в альтерантив. реализации класса Manager, чтобы обеспечить вызов метода встр. объекта Person при запуске сценария.

Технически это обсуловлено тем, что при работе с обычными классами интерпретатор ищет методы перегрузки операторов в экземплярах, а при работе с классами нового стиля - нет. Он пропускает экземпляр и пытается искать требуемый метод в классе.

Классы, с делегированием, в 3.0 в общем случае могут переопределять методы перегрузки операторов, чтобы делегировать вызовы вложенным объектам, либо вручную, либо с помощью др. инструментов или суперклассов. Мы вернемся позднее к проблеме с управлением атрибутами и снова обсудим эту тему.

**Шаг 6: использование инструментов интроспекции.** стр. 750

Добавим последний штрих перед сохранением наших объектов в БД. Нам нужно сгладить две проблемы:

- Если обратить внимание как наши объекты выводятся на экран, то можно заметить, что объект tom, класса Manager, помечается как объект класса Person. Технич. это не ошибка, но было бы более правильным, чтобы отображалось точное имя класса объекта (т.е. самого нижнего класса в иерархии). Кстати я сам об этом задумывался уже (см. выше).

- Еще более важный момент, в текущей версии отображается информация только о тех атрибутах, которые мы явно указали в методе \_\_str\_\_, чего может оказаться недостаточным в будущем. Напр. сейчас мы не можем убедиться, что атрибут job в объекте tom получает значение 'mgr' в конструкторе класса Manager, потому что метод \_\_str\_\_, который реализован в классе Person, не выводит его. Более того, если мы как-то изменим или расширим набор атрибутов в методе \_\_init\_\_, мы должны будем добавить вывод новых атрибутов в методе \_\_str\_\_.

Как видим у нас есть достаточно вопросов с заделкой "на будущее". И вообще, попытки избежать выполнения лишней работы в будущем вообще достойный поощрения.

**Специальные атрибуты классов.**

Обе проблемы можно решить с пом. инструментов интроспекции - спец. атрибутов и ф-ций, обеспечивающих доступ к внутр. реализации объектов. Это узкоспециализированные инструменты, и обычно использ-ся программистами, создающими инстурменты для др. программистов, и гораздо реже обычными программами. Эти инструменты позволяют писать программ. код, оперирующий классами обощенными способами. Мы могли бы использовать две особенности:

- Встроенные атрибут instance.\_\_class\_\_ в экземпляре ссылается на класс этого экземпляра. Классы имеют атрибут \_\_name\_\_, подобно модулям, и последовательность \_\_bases\_\_, обеспечивающую доступ к суперклассам. Мы можем использ. эти атрибуты при выводе имени класса, к котормоу принадлежит экземпляр, вместо того, чтобы вывод жестко заданное имя.

- Встр. атрибут object.\_\_dict\_\_ содержит словарь с парами ключ/значение, каждая из которых соответствует определенному атрибуту в пространстве имен объекта (включая модули, классы и экземпляры). Т.к. значением этого атрибута явл-ся словарь, мы можем получать из него список ключей, знач. атрибутов по ключам, выполнять итерации по ключам и т.д., и тем самым обеспечить обобщенный способ обработки всех атрибутов. На основе этого словаря мы можем реализовать вывод всех атрибутов, имеющихся в любом экземпляре, а не тольк ов тех, которые явно будут указаны в методе вывода.

Приведем пример использ-ия этих инструментов в IDLE. Внимание - мы импортируем класс Person с пом. from - имена классов располагаются в модуле и могут импортироваться как обычные ф-ции и переменные:

>>> from person import Person

>>> bob = Person('Bob Smith')

>>> print(bob) # Вызов метода \_\_str\_\_ объекта bob

[Person: Bob Smith, 0]

>>> bob.\_\_class\_\_ # Выведет класс объекта bob

<class 'person.Person'>

>>> bob.\_\_class\_\_.\_\_name\_\_ # Выведет удобочитаемое имя класса объекта bob

'Person'

>>> list(bob.\_\_dict\_\_.keys()) # Атрибуты - это действительно ключи словаря

['name', 'job', 'pay']

>>> for key in bob.\_\_dict\_\_:

print(key, '=>', bob.\_\_dict\_\_[key]) # Обращение по индексам

...

name => Bob Smith

job => None

pay => 0

>>> for key in bob.\_\_dict\_\_:

print(key, '=>', getattr(bob, key)) # Аналогично выражению obj.attr,

... # где attr - переменная

name => Bob Smith

job => None

pay => 0

Как уже отмечалось, некоторые атрибуты экземпляров могут отсутствовать в словаре \_\_dict\_\_, если класс экземпляра определяет атрибут \_\_slots\_\_, который явл-ся дополнительной и малопонятной особенностью классов нового стиля (и всех классов в Питон 3.0), которая использ-ся для организации хранения атрибутов в виде массива и которую мы будем обсуждать в главах 30 и 31. Т.к. на самом деле слоты принадлежат классам, а не экземплярам, и они очень редко использ-ся на практике, мы можем просто игнорировать их в нашем примере и сосредоточиться на обычном атрибуте \_\_dict\_\_.

**Обобщенный инструмент отображения.**

Мы можем использ-ть эти инструменты в суперклассе для точного отображения имен классов и вывода значений всех атрибутов экземпляров любых классов.

Создадим новый файл - независимый модуль с именем classtools.py, реализующий единственный класс. В методе \_\_str\_\_ этого класса использ-ся инструменты интроспекции, поэтмоу он может работать с любыми экземплярами, независимо от того, как атрибутами они обладают.

А т.к. это - класс, а он автоматич. превращается в обощенный инструмент отображения: благодаря наследованию, он может добавляться в любые классы, где требуется обеспечить вывод данной информации.

А если нам нужно будет когда-нибудь в будущем изменить формат вывода информации, достаточно будет изменить только этот класс:

# Файл classtools.py (новый)

# Различные утилиты и инструменты для работы с классами

class AttrDisplay:

"""

реализует наследуемый метод перегрузки операции вывода, отображающий имена классов экземпляров

и все атрибуты в виде пар имя=значение, имеющиеся в экземплярах (исключая атрибуты,

наследуемые от классов). Может добавляться в любые классы и работать с любыми экземплярами.

"""

def gatherAttrs(self):

attrs = []

for key in sorted(self.\_\_dict\_\_):

attrs.append('%s=%s' % (key, getattr(self, key)))

return ', '.join(attrs)

def \_\_str\_\_(self):

return '[%s: %s]' % (self.\_\_class\_\_.\_\_name\_\_, self.gatherAttrs())

if \_\_name\_\_ == '\_\_main\_\_':

class TopTest(AttrDisplay):

count = 0

def \_\_init\_\_(self):

self.attr1 = TopTest.count

self.attr2 = TopTest.count+1

TopTest.count += 2

class SubTest(TopTest):

pass

X, Y = TopTest(), SubTest()

print(X) # Выведет все атрибуты экземпляра

print(Y) # Выведет имя класса, самого близкого в дереве классов

Обратите внимание на стркои документирования - мы поместили их даже в класс и можем даже в метод, и эти строки доступны ф-цией help и инструментом PyDoc.

Запускаем сценарий и код самопроверки создаст два экземпляра и выведет их - метод \_\_str\_\_ выведет имена классов экземпляров и все их атрибуты в виде пар имя=значение, по именам атрибутов по алфавиту:

C:\Python34>classtools.py

[TopTest: attr1=0, attr2=1]

[SubTest: attr1=2, attr2=3]

**Атрибуты экземпляров и атрибуты классов.**

Если изучить код самопроверки модуля classtools, то можно заметить, что класс отображает только атрибуты экземпляров, присоединенные непосредственно к объекту в самом низу дерева - то есть те, что содержатся в атрибуте \_\_dict\_\_ объекта self.

Т.е. мы не получаем инфу об атрибутах, унаследованных экземплярами от классов выше по дереву (атрибут count, напр.). Унаследованные атрибуты класса присоед-ся только к объекту класса и не повторяются в экземплярах.

Если нам нужно узнать унаследованные атрибуты, мы можем с пом. ссылки \_\_class\_\_ получить доступ к классу экземпляра и извлечь из его словаря \_\_dict\_\_ атрибуты класса, а затем выполнит ьитерации через содержимое атрибута \_\_bases\_\_ класса, чтобы подняться до уровня суперклассов (так высоко как будет нужно). Если не усложнять код, вместо \_\_dict\_\_ можно вызывать встр. ф-цию dir, передав ей экземпляр и получить тот же результат:

>>> from person import Person

>>> bob = Person('Bob Smith')

>>> list(bob.\_\_dict\_\_.keys())

['name', 'job', 'pay']

>>>

>>> dir(bob)

['\_\_class\_\_', '\_\_delattr\_\_', '\_\_dict\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_le\_\_', '\_\_lt\_\_', '\_\_module\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', '\_\_weakref\_\_', 'giveRaise', 'job', 'lastName', 'name', 'pay']

**Выбор имен в инструментальных классах.**

Еще одна тонкость: т.к. наш класс AttrDisplay в модуле classtools явл-ся обощенным инструментом, чтобы служить суперклассом для любых др. классов, мы должны помнить о возможности случайных конфликтов имен с клиентскими классами.

Напр. если в клиентских подклассах будет переопределено имя gatherAttrs это может нарушить работу нашего класса, т.к. произойдет замещение методов.

Чтобы убедиться в этом добавим реализацию метода gatherAttrs в класс TotTest в коде самопроверки:

class TopTest(AttrDisplay):

...

def gatherAttrs(self): # Replaces method in AttrDisplay!

return 'Spam'

- а вот результат:

C:\Python34>classtools.py

[TopTest: Spam]

[SubTest: Spam]

Конечно это не всегда плохо, иногда нужно создавать др. методы в подклассах, для адаптации или по иным причинам.

Чтобы минимизировать вероятность конфликта имен, программисты Питон часто добавляют символ подчеркивания в начало имени метода, НЕ предназначенного для использования за пределами класса: \_gatherAttrs в нашем случае - конечно это не гарантия (кто мешает нам в подклассе тоже добавить подчерк), но - это общепринятое соглашение об именовании внутренних методов классов в языке Питон.

Лучшее, но реже используемое решение- это два подчеркивания в начало имени метода - \_\_gatherAttrs. Интерпретатор автоматич. дополняет такие имена, включая в них имя вмещающего класса, что обеспечивает им истинную уникальность. Эта особенность обычно называют псевдочастные атрибуты класса.

**Окончательные версии наших классов.**

Теперь мы можем импортировать наш обощенный инструмент из модуля classtools, добавить в список наследования нашего базового класса и удалить из него реализацию нашего раннего метода \_\_str\_\_. Окончательная реализация модуля person.py:

# Файл person.py (окончательная версия)

from classtools import AttrDisplay # Импортируем обощенный инструмент

class Person(AttrDisplay):

"""

Создает и обрабатывает записи с информацией о людях

"""

def \_\_init\_\_(self, name, job=None, pay=0):

self.name = name

self.job = job

self.pay = pay

def lastName(self): # Предполагается, что фамилия

return self.name.split()[-1] # указана последней

def giveRaise(self, percent):

self.pay = int(self.pay \* (1 + percent)) # Процент - величина в диапазоне 0..1

class Manager(Person):

"""

Версия класса Person, адаптированная в соответствии со специальными требованиями

"""

def \_\_init\_\_(self, name, pay):

Person.\_\_init\_\_(self, name, 'mgr', pay)

def giveRaise(self, percent, bonus=.10):

Person.giveRaise(self, percent + bonus)

if \_\_name\_\_ == '\_\_main\_\_':

bob = Person('Bob Smith')

sue = Person('Sue Jones', job='dev', pay=100000)

print(bob)

print(sue)

print(bob.lastName(), sue.lastName())

sue.giveRaise(.10)

print(sue)

tom = Manager('Tom Jones', 50000)

tom.giveRaise(.10)

print(tom.lastName())

print(tom)

В окончательную версию мы добавили несколько комментариев, описывающих наши классы. И теперь наш сценарий выводит все атрибуты, а не только те, что были явно указаны в оригинальной версии метода \_\_str\_\_ (становится немного понятнее частое упоминание явно и неявно указанных атрибутов и прочих элементов кода, надеюсь дальше станет только понятнее что же такое явно и неявно указанные элементы).

Также корректно указывается имя ближайшего класса, и убедимся, что атрибут job был корректно инициализирован в конструкторе класса Manager:

C:\Python34>person.py

[Person: job=None, name=Bob Smith, pay=0]

[Person: job=dev, name=Sue Jones, pay=100000]

Smith Jones

[Person: job=dev, name=Sue Jones, pay=110000]

Jones

[Manager: job=mgr, name=Tom Jones, pay=60000]

С точки зрения перспективы, наш класс, реализующий вывод атрибутов, можно рассматривать как обощенный инструмент, который можно использовать как суперкласс для любых других классов, обеспечивающий вывод атрибутов.

**Шаг 7 (последний): сохранение объектов в базе данных.** стр. 757

У нас имеется система из двух модулей, причем один из них можно использ. и в др. программах. Единственный минус - объекты, создаваемые классами не явл-ся настоящими записями БД, они удаляются по завершении программы. И сейчас мы займемся созданием хранилища объектов.

**Модули pickle, shelve и dbm.**

Эти модули обеспечивают сохранение объектов во всех версиях Питона:

***pickle*** - Преобразует произвольные объекты языка Питон в строку байтов и обратно.

***dbm*** (в Питон 2.6 anydbm) - Реализует сохранение строк в файлах, обеспечивающих возможность обращения по ключу.

***shelve*** - Использует первые два модуля, позволяя сохранять объекты в файлах-хранилищах, обеспчивающих возможность обращения по ключу.

Модуль pickle обеспечивает самые общие средства преобразования объектов: он способен превратить практич. любой объект, находящийся в памяти, в строку байтов, а затем восстанвоить. Почти все объекты, знакомые нам - списки, словари, вложенные комбинации объектов, даже экземпляры классов. Сохраняя объекты в файле в виде строк с пом. модуля pickle, мы обеспечиваем долговременное хранение этих объектов, позднее можно загрузить эти стркои и восстановить данные.

С пом. pickle достаточно просто сделать сохранение объектов в простых файлах и загрузку их из файлов, однако модуль shelve обеспечивает доп. удобсва, позволяя сохранять объекты, обработанные модулем pickle, по ключу. Модуль shelve преобразует объект в строку с пом. модуля pickle и сохраняет её под указанным ключом в файле dbm. Позднее shelve извлекает строку по ключу и воссоздает оригинальный объект в памяти, с пом. модуля pickle. На вид сложно, однако в программе обращение к объектам в хранилище выглядит как обращение к элементам словаря - мы обращаемся к объекту по ключу, сохраняем его, выполняя присваивание по ключу, и можем использ. инструменты словарей, такие как len, in, dict.keys, и т.п. Модуль shelve автоматич. отображает операции со словарем на объекты, хранящиеся в файле.

Отличие хранилища от словаря в том, что хранилища необходимо предварительно открывать, а затем закрывать после внесения изменений. Т.о. хранилища можно рассматривать как простые БД. Хранилища не поддерживают запросы (напр. SQL), не поддерживают транзакции и т.п., но в целом способны извлекат ьпользу из языка Питон.

**Сохранение объектов в хранилище.**

Напишем новый сценарий, который сохраняет экземпляры наших классов в хранилище. Назовем его makedb.py. и в него нужно импортировать наши классы.

На самом деле есть два способа импорта, как мы знаем:

import person # Загружает класс с пом. инструкции import

bob = person.Person(...) # Обращение к классу через имя модуля

from person import Person # Загружает класс с пом. инструкции from

bob = Person(...) # Обращение по непосредственному имени класса

Мы будем использовать инструкцию from (меньше кода нужно писать). Итак создадим нам модуль, с экземплярами, сохранение объектов в хранилище и кодом самопроверки:

# Файл makedb.py: сохраняет объекты Person в хранилище

from person import Person, Manager

bob = Person('Bob Smith')

sue = Person('Sue Jones', job='dev', pay=100000)

tom = Manager('Tom Jones', 50000)

import shelve

db = shelve.open('persondb') # Имя файла хранилища

for object in (bob, sue, tom): # В качестве ключа использовать атрибут name

db[object.name] = object # и сохранить объект в хранилище

db.close() # Закрыть после внесения изменений

Заметим, что при присваивании объектов, в качестве ключей используются значения атрибутов name. Это просто для удобства мы сделали - ключами могут быть любые строки, которые можно было бы создать с применением уникальных характеристик, таких как идентификатор процесса и отметка времени (можно получит ьс пом. модулей os и time). Одно ограничение - ключи должны быть только строками и должны быть уникальными, т.к. под каждым ключем можно сохранять только один объект. А вот значения по ключу могут быть объектами любого типа.

Т.к. мы не создали никаких опознавательных выводов инфы, то просто запускаем сценарий и если ошибок нет, то объекты сохранились:

C:\Python34>makedb.py

C:\Python34>

- всё нормально, я конечно ошибся при импорте - забыл после Person указать класс Manager, но интерпретатор всё видит и указал на мою оплошность.

У меня все ошибки от невнимательности, я так понимаю они практически у всех такие же, все-таки логику работы и команды мы продумываем, а не пишем наугад, а вот рассеяность внимания сказывается на простых ошибках.

**Исследование хранилища в интерактивном сеансе.**

Сейчас у нас в каталоге имеется три файла имя которых начинается с "persondb" (у кого-то может и меньше или больше, у меня 3 файла). Как и ф-ция open, ф-ция shelve.open() создает файлы в текущем рабочем каталоге, если указанное имя не содержит полный путь. Но независимо от этого они обеспечивают доступ по ключу к представлениям объектов, созданных, с пом. модуля pickle.

Эти файлы и явл-ся нашей базой данных, их и нужно копировать если хотите создать копию или перенести БД. Файлы имеют двоичный формат и не имеют большого смысла вне контекста модуля shelve. Кстати у нас Питон 3.0 поэтому 3 файла. Рассмотрим их:

# Модуль, позволяющий получит ьсписок файлов в каталоге:

# проверка наличия файлов

>>> import glob

>>> glob.glob('person\*')

['person.py', 'persondb.bak', 'persondb.dat', 'persondb.dir']

# Тип файла: текстовый - для строк, двоичный - для байтов

>>> print(open('persondb.dir').read())

'Tom Jones', (1024, 91)

'Sue Jones', (512, 92)

'Bob Smith', (0, 80)

>>> print(open('persondb.dat', 'rb').read())

b'\x80\x03cperson\nPerson\nq\x00)\x81q\x01}q\x02(X\x04\x00\x00\x00nameq\x03X\t\x00\x00\x00Bob Smithq\x04X\x03\x00\x00\x00jobq\x05NX\x03\x00\x00\x00payq\x06K\x00ub.\x00\x00\x00\x00...

... часть строк опущена...

Такая шифровка может изменяться от платформы к платформе, и точно не может называться дружественным интерфейсом к БД. Мы можем написать сценарий или поработать с хранилищем прямо в интерактивной оболочке.

Т.к. хранилища явл-ся объектами Питон, содержащими другие объекты Питон, мы можем работать с ними, используя обычный синтаксис Питона. Пример листинга интерактивного сеанса, который играет роль клиента базы данных:

>>> import shelve

>>> db = shelve.open('persondb') # Открыть хранилище

>>> len(db) # В хранилище содержится три "записи"

3

>>> list(db.keys()) # keys - это оглавление

['Tom Jones', 'Sue Jones', 'Bob Smith']

>>> bob = db['Bob Smith'] # Извлечь объект bob по ключу

>>> print(bob) # Вызовет \_\_str\_\_ из AttrDisplay

[Person: job=None, name=Bob Smith, pay=0]

>>> bob.lastName() # Вызовет lastName из Person

'Smith'

>>> for key in db: # Итерации, извлечение, вывод

print(key, '=>', db[key])

Tom Jones => [Manager: job=mgr, name=Tom Jones, pay=50000]

Sue Jones => [Person: job=dev, name=Sue Jones, pay=100000]

Bob Smith => [Person: job=None, name=Bob Smith, pay=0]

>>> for key in sorted(db): # Итерации через отсортированный

print(key, '=>', db[key]) # список ключей

Bob Smith => [Person: job=None, name=Bob Smith, pay=0]

Sue Jones => [Person: job=dev, name=Sue Jones, pay=100000]

Tom Jones => [Manager: job=mgr, name=Tom Jones, pay=50000]

Обратите внимание, нам не нужно импортировать классы Person и Manager. Мы можем вызвать метод lastName объекта bob и вывести его содержимое, даже при том, что класс Person этого объекта находится вне области видимости.

Это обусловлено тем, что, когда модуль pickle преобразует экземпляр класса, он записывает атрибуты экземпляра self вместе с именем класса, из которого был создан, и именем модуля, где находится определение этого класса. Т.о когда объект bob извлекается из хранилища, интерпретатор автоматич. импортирует класс и связывает с ним объект bob.

Благодаря такому поведению после загрузки экземпляры классов автоматич. обретают поведение своего класса. Мы должны импортировать наши классы, только если необходимо создавать новые экземпляры, но не для работы с существующими. Это влечет за собой некторые моменты:

- *Недостаток* заключ. в том, что позднее, когда вып-ся загрузка экземпляров, классы и их модули должы быть доступны для импорта. Т.е. классы сохраняемых объектов должны быть определены на верхнем уровне модуля, который нах-ся в одном из каталогов в пути поиска sys.path (и не должны нах-ся в модуле \_\_main\_\_ сценария, если только они не использ-ся только в пределах этого модуля). Из-за этих требований, в некоторых приложениях для сохранения использ-ся более простые объекты, такие как словари и списки, особенно если они передаются через Интернет.

- *Преимущество* заключ-ся в том, что изменения в реализации класса автоматич. будут восприняты экземплярами после их загрузки - часто нет никакой необходимости обновлять сохраненные объекты, потому что обычно изменения касаются только реализации методов класса.

Конечно у модуля shelve есть некоторые ограничения, но в целом shelve и pickle явл-ся отличными инструментами, когда речь идет о реализации простого хранилища объектов.

**Обновление объектов в хранилище.**

Создадим еще один сценарий, который обновляет экземпляры (записи) при каждом запуске, чтобы мы могли убедиться, что наши объекты действительно сохраняются между запусками программы (т.е. при каждом запуске доступны их текущие значения). Сценарий updatedb.py выводит содержимое БД и увеличивает з\п в одном из объектов при каждом запуске. Благодаря особенностям ООП и механизму наследования updatedb автоматич. вызывает наш метод \_\_str\_\_ и пользуется методом giveRaise:

# Файл updatedb.py: обновляет объекты класса Person в БД

import shelve

db = shelve.open('persondb') # Открыть хранилище в файле с указанным именем

for key in sorted(db): # Обойти и отобразить объекты в базе данных

print(key, '\t=>', db[key]) # Вывод в требуемом формате

sue = db['Sue Jones'] # Извлечь объект по ключу

sue.giveRaise(.10) # Изменить объект в памяти вызовом метода

db['Sue Jones'] = sue # Присвоить по ключу, чтобы обновить объект в хранилище

db.close() # Закрыть после внесения изменений

Запустим несколько раз сценарий и увидим как повышается з\п у sue:

C:\Python34>updatedb.py

Bob Smith => [Person: job=None, name=Bob Smith, pay=0]

Sue Jones => [Person: job=dev, name=Sue Jones, pay=100000]

Tom Jones => [Manager: job=mgr, name=Tom Jones, pay=50000]

C:\Python34>updatedb.py

Bob Smith => [Person: job=None, name=Bob Smith, pay=0]

Sue Jones => [Person: job=dev, name=Sue Jones, pay=110000]

Tom Jones => [Manager: job=mgr, name=Tom Jones, pay=50000]

C:\Python34>updatedb.py

Bob Smith => [Person: job=None, name=Bob Smith, pay=0]

Sue Jones => [Person: job=dev, name=Sue Jones, pay=121000]

Tom Jones => [Manager: job=mgr, name=Tom Jones, pay=50000]

C:\Python34>updatedb.py

Bob Smith => [Person: job=None, name=Bob Smith, pay=0]

Sue Jones => [Person: job=dev, name=Sue Jones, pay=133100]

Tom Jones => [Manager: job=mgr, name=Tom Jones, pay=50000]

Всё это - результат работы модулей shelve и pickle, и поведения реализованного в наших классах. К тому же мы можем проверить результаты запуска сценария с помощью интерактив. оболочки:

>>> import shelve

>>> db = shelve.open('persondb')

>>> rec = db['Sue Jones']

>>> print(rec)

[Person: job=dev, name=Sue Jones, pay=146410]

>>> rec.lastName()

'Jones'

>>> rec.pay

146410

**Рекомендации на будущее.**

Мы увидели и применили основные механизмы ООП языка Питон на практике. Создали полноценные классы. Создали настоящую БД. Конечно мы могли бы существенно увеличить ньюансы и тонкие настройки, добавить мелкие детали, но и основы это уже хорошо.

Кроме того, мы могли бы расширить круг наших интересов и использовать инструменты, поставляемые в составе Питона или доступные в мире ПО:

- *Графический интерфейс пользователя.* Мы могли бы графич. интерфейс для работы с БД (удобнее и дружелюбнее). Это можно с пакетом tkinter или сторонними инструментами WxPython или PyQt.

- *Веб-сайты.* Веб-интерфейсы опережают графич. интерфейсы в плане доступности, с Интернетом (Сетью) доступ к БД растет. Веб-сайты можно строить на основе CGI-сценариев, или использовать полноценные веб-фреймворки сторонних производителей, такие как Django, TurboGears, Pylons, Zope, Google App Engine.

- *Веб-службы.* Мы можем предоставлять более прямой способ (отличный от веб-клиентов) извлечения записей из БД на стороне веб-сервера - посредством интерфейсов веб-служб, таких как SOAP, XML-RPC и т.д.

- *Базы данных.* Для больших и мощных БД мы могли бы использовать механизм хранения данных такой как ZODB, или традиционные реляционные БД, такие как MySQL, Oracle, PostgreSQL или SQLite.

- *Механизмы объектно-реляционных отображений (ORM).* При переходе на использование реляционных БД нам не придется отказываться от инструментов ООП, имеющихся в Питон. Механизмы объектно-реляционного отображения (object-relational mapping, ORM), такие как SQLObject и SQLAlchemy, могут автоматич. отображать реляционные таблицы и записи в классы и экземпляры на зяыке Питон и обратно, благодаря чему мы можем обрабатывать хранимые данные, используя привычный синтаксис классов языка Питон. Эти механизмы позволяют объединить сильные стороны реляционных БД и модели классов в Питон.

**Глава 28. Подробнее о программировании классов.** стр. 769

**Инструкция class.**

Инструкция class в Питон существенно отлич-ся от других ООП языков. Напр. как и в С++ инструкция class явл-ся основным инструментом ООП, но в отличие от С++, в Питон она не явл-ся объявлением.

Подобно инструкции def инструкция class создает объект и явл-ся неявной инструкцией присваивания (еще одна отсылка к неявному/явному) - когда она выполняется, создается объект класса, ссылка на который сохраняется в имени, использованном в заголовке инструкции.

Как и def, инструкция class явл-ся настоящим выполняемым программ. кодом - класс не существует, пока поток выполнения не достигнет инструкции class, которая определяет его (напр. при импорте модуля).

**Общая форма.** стр. 770

Инструкция class - это составная инструкция, с блоком операторов, обычно под строкой заголовка. В заголовке, после имени, в круглых скобках, через запятую, перечисляются суперклассы (если они есть). Пример общей формы:

class <name>(superclass, ...): # Присваивание имени

data = value # Совместно используемые данные класса

def method(self, ...): # Методы

self.member = value # Данные экземпляров

Внутри инструкции class любая операция присваивания создает атрибут класса, а методы со спец. именами перегружают операторы (помним \_\_init\_\_, который вызывается во время создания объекта экземпляра).

**Пример.**

Как мы знаем, классы - это всего лишь пространства имен, т.е. инструменты, определяющие имена (атрибуты), с пом. которых клиентам экспортируются данные и логика. Как же class порождает пространство имен?

А вот как. Как и в модулях, инструкции, вложенные в тело инструкции class, создают атрибуты класса. Когда интерпретатор доходит до инструкции class (далее просто - class, запомним, что class - это инструкция), а не когда происходит вызов класса, он выполняет все инструкции в её теле. Все присваивания, в итоге, создают имена в локал. области видимости класса, которые становятся атрибутами объекта класса. Поэтмоу классы напоминают модули и ф-ции:

- Подобно ф-циям, class явл-ся локал. областями видимости, где располагаются имена, созданные вложенными операциями присваивания.

- Подобно именам в модуле, имена, созданные внутри class, становятся атрибутами объекта класса.

Основное отличие классов в том, что их пространства имен составляют основу механизма наследования в Питон.

Т.к. class - это составная инструкция, в её тело могут быть вложены любые инструкции - print, =, if, def и т.д. Все инструкции внутри class выполняются, когда выполняется сама инструкция class (а не когда класс вызывается для создания экземпляра). Операции присваивания именам внутри class создают атрибуты класса, а влож. инструкции def создают методы класса, кроме того атрибуты класса создаются и др. инструкциями, выполняющими присваивание.

Напр. присваивание объекта, не являющегося ф-цией, атрибутам, создает атрибуты данных, совместн оиспользуемых всеми экземплярами:

>>> class ShareData:

spam = 42 # Присваивание создает атрибут данных класса

...

>>> x = ShareData()

>>> y = ShareData()

>>> x.spam, y.spam # Экземпляры наследуют и совместно используют атрибут spam

(42, 42)

spam созд-ся на верхнем уровне в class, имя присоед-ся к классу и поэтому совместно использ-ся всеми экземплярами.

Можно изменять значение атрибута, выполняя присваивание через имя класса, и обращаться к нему через имена экземпляров или класса:

>>> ShareData.spam = 99

>>> x.spam, y.spam, ShareData.spam

(99, 99, 99)

Такие атрибуты могут хранить инфу, доступную всем экземплярам (напр. счетчик кол-ва созданных экземпляров, увидим в гл. 31).

А теперь попробуем присвоить знач. атрибуту spam не через имя класса, а через имя экземпляра:

>>> x.spam = 88

>>> x.spam, y.spam, ShareData.spam

(88, 99, 99)

Присваивание атрибуту экземпляра изменяет (или создает) имя в экземпляра, а не в классе (и это логично). Вообще, поиск в дереве наследования производится только при попытке чтения атрибута, но не присваивании: присваивание атрибуту объекта всегда изменяет сам объект, а не что-то другое (при условии, что в классе не переопредлелен метод перегрузки \_\_getattr\_\_, который отвечает за операцию присвавиания).

Приведем более понятный пример:

class MixedNames: # Определили класс

data = 'spam' # Присваивание атрибуту класса

def \_\_init\_\_(self, value): # Присваивание имени метода

self.data = value # Присваивание атрибуту экземпляра

def display(self):

print(self.data, MixedNames.data) # Атрибут экземпляра, атрибут класса

У нас две инстр. def связывают атрибуты класса с методами. Также есть присваивание = . Первое присваивание data идет внутри class, оно создается в локал. области видимости класса и становится атрибутов объекта класса. Как и все атрибуты класса, data наследуется и использ-ся всеми экземплярами класса, которые не имеют собственного атрибута data:

>>> x = MixedNames(1) # Создаются два объекта экземпляра

>>> y = MixedNames(2) # каждый из которых имеет свой атрибут data

>>> x.display(); y.display() # self.data - это другие атрибуты,

1 spam # а MixedNames.data - тот же самый

2 spam

Суть примера в том, что атрибут data находится в двух местах: в объектах экземпляров (атрибут self.data внутри \_\_init\_\_) и в классе (data = 'spam').

Сохраняя атрибуты в разных объектах мы определяем области их видимости. Атрибуты классов использ-ся всеми экземплярами, а атрибуты экземпляров уникальны для каждого экземпляра.

В прошлом примере выражения x.data и y.data возвращают атрибут экземпляра, которые переопределяют то же самое имя в классе. А выражение MixedNames.data явно обращается к атрибуту класса.

**Методы.** стр. 772

Методы - это обычные объекты функций, которые созд-ся инструкциями def в теле class (как я понимаю это тема о методах классов, а не методах вообще). Методы реализуют поведение, наследуемое объектами экземпляров. Технич. методы работают так же как ф-ции, с одним исключением: в первом аргументе методам всегда передается подразумеваемый объект экземпляра.

Т.е. интерпретатор автоматич. отображает вызов метода экземпляра на метод класса следующим образом.

Вывод метода экземпляра:

instance.method(args...)

автоматически преобразуется в вызов метода класса:

class.method(instance, args...)

- где класс определяется в результате поиска имени метода по дереву наследования. Фактич. в Питон оба метода допустимы для использования.

Первый аргумент (отличит. особенность) обычно наз-ся self, в соотв. с общепринятыми соглашениями (технич. можете писать хоть qwerty, т.к. значение имеет позиция аргумента). Этот аргумент обеспечивает доступ к экземпляру, т.е. субъекту вызова - т.к. экземпляров может быть много аргумент self обеспечивает доступ к конкретному экземпляру.

Аргумент self всегда должен использоваться явно, в отличие от др. ЯП, напр. в С++ аргумент self напоминает указатель this, но использ-ся всегда явно.

**Пример метода.**

Пусть есть следующий класс:

class NextClass: # Определение класса

def printer(self, text): # Определение метода

self.message = text # Изменение экземпляра

print(self.message) # Обращение к экземпляру

Имя printer ссылается на объект ф-ции, а т.к. оно создается в области (т.е. области видимости) class, оно становится атрибутом объекта класса и буде тунаследовано всеми экземплярами. Обычно такие методы как printer предназначены для обработки экземпляров:

>>> x = NextClass() # Создаем экземпляр

>>> x.printer('instance call') # Вызываем его метод

instance call

>>> x.message # Экземпляр изменился

'instance call'

Когда метод вызывается через имя экзмпляра, то сначала находится метод printer, а затем его аругменту self автоматич. присв-ся объект экземпляра (х). В аргумент text записывается строка, переданная в вызов метода ('instance call').

Обратите внимание - интерпретатор автоматич. передает в первом аргументе self ссылку на сам экземпляр, поэтому нам достаточно передать методу только один аргумент (text).

Методы могут вызываться как через экземпляр так и через сам класс:

>>> NextClass.printer(x, 'class call') # Прямой вызов метода класса

class call

>>> x.message # Экземпляр снова изменился

'class call'

Если попытаться вызвать метод без указания конкретного экземпляра, то получим ошибку:

>>> NextClass.printer('bad call')

...

TypeError: printer() missing 1 required positional argument: 'text'

**Вызов конструкторов суперклассов.**

Методы обычно вызываются через экземпляры, но и вызов через класс может играть особую роль. Напр. вызов конструктора.

Метод \_\_init\_\_ наслед-ся так же, как и другие атрибуты. Это значит, что во время создания экземпляра интерпретатор отыскивает только один метод \_\_init\_\_. Если в подклассе нужно выполнение действий, предусматриваемых конструктором суперкласса, то нужно вызывать \_\_init\_\_ через имя класса:

class Super:

def \_\_init\_\_(self, x):

...программный код по умолчанию...

class Sub(Super):

def \_\_init\_\_(self, x, y):

Super.\_\_init\_\_(self, x)

...адаптированный код...

I = Sub(1, 2)

Такое использование конструктора суперкласса нужн овыполнять только при действителньой необходимости. Реальный пример см. в предыдущей главе в классе Manager.

**Другие возможности методов.**

В главе 31 мы познакомимся с статическими методами, которые не предполагают наличие объекта экземпляра в первом аргументе. Родственные им методы класса принимают в первом аргументе сам класс и могу тиспольз-ся для манипулирования данными, принадлежащими конкретному классу.

**Наследование.** стр. 775

Основное назначение class - поддержка наследования имен. Действите наследования заключ-ся в операции поиска в дереве определений атрибутов. Когда использ-ся выражение вида object.attr (где object - это объект экземпляра или класса), интерпретатор ищем первое вхождение атрибута attr в дереве пространств имен снизу вверх, начиная с object. Сюда относятся и ссылки на атрибуты аргумента self внутри методов.

**Создание дерева атрибутов.**

- Атрибуты экземпляров создаются посредством присваивания атрибутам аргумента self в методах.

- Атрибуты классов создаются инструкциями (присваивания), расположенными внутри инструкции class.

- Ссылки на суперклассы создаются путем перечисления классов в круглых скобках в заголовке class.

В рез-те появл-ся дерево пространств имен с атрибутами, которое иде тот экземпляров к классам, из которых они были созданы, и ко всем суперклассам, перечисленным в class. Поиск идет так же, снизу вверх, от экземпляров к суперклассам.

**Рисунок 28.1 Способы создания и заполнения имен в дереве пространств имен.** стр. 776

**Специализация унаследованных методов.**

Наследование - мощный инструмент адаптации и специализации программ, за счет добавления подклассов мы создаем целые системы, без изменения уже существующего кода.

Переопределение унаследованных имен дает множество приемов специализации. Подклассы могут полностью замещать унаследованные атрибуты, предоставлять атрибуты, которые ожидается отыскать в суперклассах, и расширять методы суперклассов за счет их вызова из методов подкласса. Мы уже знаем прием с замещением. Пример как выполняется расширение:

>>> class Super:

def method(self):

print('in Super.method')

>>> class Sub(Super):

def method(self): # Переопределить метод

print('starting Sub.method') # Дополнительное действие

Super.method(self) # Выполнить действие по умолчанию

print('ending Sub.method')

Главное здесь - это прямые вызовы методов суперкласса. Класс Sub замещает метод method класса Super, своей специализированной версией. Но внутри этого спец. метода происходит вызов оригинального метода из класса Super. Т.о. метод Sub.method не замещает полностью метод Super.method, а просто расширяет его:

>>> x = Super()

>>> x.method() # Вызывается метод Super.method

in Super.method

>>> x = Sub()

>>> x.method() # Вызывается Sub.method, который вызывает Super.method

starting Sub.method

in Super.method

ending Sub.method

Этот прием расширения часто использ-ся в конструкторах.

**Приемы организации взаимодействия классов.**

В файле specialize.py определяется неск-ко классов, которые показывают приемы использ-ия классов:

- ***Super*** - Определяет метод method и метод delegate, который предполагает наличие метода action в подклассе.

- ***Inheritor*** - Не предоставляет никаких новых имен, поэтому он получает всё, что определено тольк ов классе Super.

- ***Replacer*** - переопределяет метод method класса Super.

- ***Extender*** - Адаптирует метод method класса Super, переопределяя и вызывая его, чтобы выполнить действия, предусмотренные по умолчанию.

- ***Provider*** - Реализует метод action, котоырй ожидается методом delegate класса Super.

Рассмотрим как эти классы адаптируют свой общий суперкласс. Содержимое файла:

class Super:

def method(self):

print('in Super.method') # Поведение по умолчанию

def delegate(self):

self.action() # Ожидаемый метод

class Inheritor(Super): # Наследует методы как они есть

pass

class Replacer(Super): # Полностью замещает метод method

def method(self):

print('in Replacer.method')

class Extender(Super): # Расширяет поведение метода method

def method(self):

print('starting Extender.method')

Super.method(self)

print('ending Extender.method')

class Provider(Super): # Определяет необходимый метод

def action(self):

print('in Provider.action')

if \_\_name\_\_ == '\_\_main\_\_':

for klass in (Inheritor, Replacer, Extender):

print('\n' + klass.\_\_name\_\_ + '...')

klass().method()

print('\nProvider...')

x = Provider()

x.delegate()

Отметим моменты. Код тестирования создает экземпляры трех разных классов в цикле for. Т.к. классы - это объекты, можно поместить их в кортеж и создавать экземпляры единообразным способом. Кроме того классы, как и модули, имеют атрибут \_\_name\_\_ - он содержит строку с именем класса, указанным в заголовке инструкции class. Запустим файл specialize.py:

C:\Python34>specialize.py

Inheritor...

in Super.method

Replacer...

in Replacer.method

Extender...

starting Extender.method

in Super.method

ending Extender.method

Provider...

in Provider.action

**Абстрактные суперклассы.** стр. 778

Обратим внимание как работает класс Provider в прошлом примере. Когда через экземпляр класса Provider вызывается метод delegate, инициируются две независимые процедуры поиска:

1. При вызове x.delegate интерпретатор отыскивает метод delegate в классе Super, начиная поиск от экземпляра класса Provider и двигаясь вверх по дереву наследования. Экземпляр х передается методу в виде аргумента self.

2. Внутри метода Super.delegate выражение self.action приводит к запуску нового, независимого поиска в дереве наследования, начиная от экземпляра self и дальше вверх по дереву. Т.к. аргумент self ссылается на экземпляр класса Provider, метод action будет найден в подклассе Provider.

Такой способ "восполнения пробелов" в реализации - обычное дело для платформ ООП. В терминах метода delegate такие суперклассы, как в этом примере, иногда называют абстрактными суперклассами- классы, которые предполагают, что часть их функциональности будет реализована их подклассами. Если ожидаемый метод не определен в подклассе - получим исключение.

Разработчики иногда делают такие требования к подклассам более очевидныыми, с пом. инструкций assert или возмуждая исключение NotImplementedError с пом. инструкции raise. Пример с assert:

>>> class Super:

def delegate(self):

self.action()

def action(self):

assert False, 'action must be defined!'

>>> X = Super()

>>> X.delegate()

...

assert False, 'action must be defined!'

AssertionError: action must be defined!

С assert мы познакомимся позднее, а пока заметим, что если выражение возвращает ложь, она возбуждает исключение с указанным сообщением об ошибке. В нашем случае выражение всегда возвращает ложь, чтобы появилась ошибка, если метод action нигде не будет переопределен и поиск остановится на оригинальной версии.

В некоторых классах напротив, в таких методах-заглушках исключение NotImplementedError возбуждается напрямую:

>>> class Super:

def delegate(self):

self.action()

def action(self):

raise NotImplementedError('action must be defined!')

>>> X = Super()

>>> X.delegate()

...

raise NotImplementedError('action must be defined!')

NotImplementedError: action must be defined!

При работе с экземплярами мы также будем получать исключения, если подклассы не обеспечут свою реализацию ожидаемого метода, замещающего метод суперкласса:

>>> class Sub(Super): pass

>>> X = Sub()

>>> X.delegate()

...

NotImplementedError: action must be defined!

>>> class Sub(Super):

def action(self): print('spam')

>>> X = Sub()

>>> X.delegate()

spam

Более реалистичный пример использования концепций, представленных в этом разделе, мы получим в упражнениях позднее.

**Абстрактные суперклассы в Питон 2.6 и 3.0**

В обоих версиях Питон, абстрактные суперклассы могут быть реализованы спец. синтаксической конструкцией определения класса.

В 3.0 для этого использ-ся именованный аргумент в заголовке инструкции class и спец. декоратор @abstract методов:

from abc import ABCMeta, abstractmethod

class Super(metaclass=ABCMeta):

@absctractmethod

def method(self, ...):

pass

В 2.6 вместо именованного аргумента в заголовке class, используется атрибут класса:

class Super:

\_\_metaclass\_\_ = ABCMeta

@absctractmethod

def method(self, ...):

pass

В любом случае получаем одно и то же - мы лишены возможности создавать экземпляры, если метод не будет определен ниже в дереве классов. Пример:

>>> from abc import ABCMeta, abstractmethod

>>>

>>> class Super(metaclass=ABCMeta):

def delegate(self):

self.action()

@abstractmethod

def action(self):

pass

>>> X = Super()

TypeError: Can't instantiate abstract class Super with abstract methods action

>>> class Sub(Super): pass

...

>>> X = Sub()

TypeError: Can't instantiate abstract class Sub with abstract methods action

>>> class Sub(Super):

def action(self): print('spam')

...

>>> X = Sub()

>>> X.delegate()

spam

Такой класс, с абстрактным методом, не может использ-ся для создания экземпляров (т.е. нам не удастся создать экземпляр вызовом этого класса), если все абстрактные методы не будут реализвоаны в подклассах.

Такой ход имеет свои преимущества - ошибку мы получим уже при попытке создания экземпляра, а не позднее, при попытке вызова отсутствующего метода.

Этот прием основан на декораторах функций и объявлениях метаклассов, с которыми мы познакомимся позднее.

**Пространства имен: окончание истории.** стр. 781

Всмоним все правила, используемые при разрешении имен. Первое, что нам нужно запомнить: квалифицированные и неквалифицированные имена интерпретируются по-разному, и некоторые области видимости служат для инициализации пространств имен объектов:

- Неквалифицированные имена (напр. Х) располагаются в областях видимости.

- Квалифицированные имена атрибутов (напр. object.X) принадлежат пространствам имен объектов.

- Некоторые области видимости инициализируют пространства имен объектов (в модулях и классах).

**Простые имена: глобальные, пока не выполняется присваивание.**

Поиск неквалифицированных простых имен выполняется в соответствии с правилом лексической видимости LEGB, выведенном для функций в гл. 17:

- *Присваивание (X = value).* Операция присваивания делает имена локальными: создает или изменяет имя Х в текущей локальной области видимости, если имя не объявлено глоабльным.

- *Ссылка (Х).* Пытается отыскать имя Х в текущей локал. области видимости, затем в области видимости каждой из вмещающих функций, затем в текущей глоабл. области видимости и, наконец, во встроенной области видимости.

**Имена атрибутов: пространства имен объектов.**

Квалифицированные имена атрибутов ссылаются на атрибуты конкретных объектов и к ним прмиеняются правила, предназначенные для модулей и классов. Для объектов классов и экземпляров правила дополняются процедурой поиска в дереве наследования:

- *Присваивание (object.X = value).* Создает или изменяет атрибут Х в пространстве имен объекта object, и ничего больше. Поиск по дереву наследования идет только при поптыке получить ссылку на атрибут, но не при операции присваивания.

- *Ссылка (object.X).* Для объектов, созданных на основе классов, поиск атрибута Х производится сначала в объекте object, затем во всех классах выше в дереве наследования. Если объекты не из класса, напр. модули, атрибут Х извлекается непосредственно из объекта object.

**"Дзен" пространств имен в Питон: классификация имен происходит при присваивании.**

Из-за различий в процедурах поиска простых и составных имен и нескольких уровней поиска в обеих процедурах, иногда бывает трудно сказать где будет найдено имя.

В Питон, место, где выполняется присваивание, имеет крайне важное значение - оно полностью определяет область видимости или объект, где будет размещаться имя.

Файл manynames.py показывает как эти принципы переводятся в программ. код и обощает идеи, касающиеся пространств имен, которые мы встречали в книге:

# manynames.py

X = 11 # Глобальное (в модуле) имя/атрибут (X, или manynames.X)

def f():

print(X) # Обращение к глобальному имени Х (11)

def g():

X = 22 # Локальная (в ф-ции) переменная (Х, скрывает имя Х в модуле)

print(X)

class C:

X = 33 # Атрибут класса (С.Х)

def m(self):

X = 44 # Локальная переменная в методе (Х)

self.X = 55 # Атрибут экземпляра (instance.X)

В этом файле пять раз выполняется присваивание имени Х. Но, т.к. присв-ие вып-ся в пяти разных метсах, то все пять Х явл-ся совершенно разными переменными.

Ниже описывается продолжение файла, где создается экземпляр и выводятся значения всех имеющихся переменных Х:

# manynames.py, продолжение

if \_\_name\_\_ == '\_\_main\_\_':

print(X) # 11: модуль (за пределами файла manynames.py)

f() # 11: глобальная

g() # 22: локальная

print(X) # 11: переменная модуля не изменилась

obj = C() # Создать экземпляр

print(obj.X) # 33: переменная класса, унаследованная экземпляром

obj.m() # Присоединить атрибут Х к экземпляру

print(obj.X) # 55: экземпляр

print(C.X) # 33: класс (она же obj.X, если в экземпляре нет Х)

#print(C.m.X) # ОШИБКА: видима только в методе

#print(g.X) # ОШИБКА: видима только в ф-ции

Обратите внимание, что мы можем добарться до атрибута класса (С.Х), но никогда не сможем получить доступ к локальным переменным в ф-циях или методах, находясь за пределами соотв. инструкций def. Локал. переменные видны только программ. коду внутри инструкции def и существуют в памяти только во время выполнения ф-ции или метода.

Некоторые имена из этого файла видны и в других модулях, но сначала мы должны импортировать модуль manynames.py:

# otherfile.py

import manynames

X = 66

print(X) # 66: здешняя глобальная переменная

print(manynames.X) # 11: глобальная, ставшая атрибутом в результате импорта

manynames.f() # 11: X в manynames, не здешняя глобальная!

manynames.g() # 22: локальная в ф-ции, в другом файле

print(manynames.C.X) # 33: атрибут класса в другом модуле

I = manynames.C()

print(I.X) # 33: всё еще атрибут класса

I.m()

print(I.X) # 55: а теперь атрибут экземпляра!

Отметим, что manynames.f() выводи тзначение Х из модуля manynames, а не переменной из текущего модуля - обасть видимости всегда определяется местоположением инструкции присваивания в программ. коде (т.е. лексически) и не зависит от того, что импортируется и куда.

Крмое того, собственный атрибут Х в экземпляре отсутствовал, пока не был вызван метод I.m(), - атрибуты, как и любые переменные, появл-ся на свет во время операции присваивания, а не до неё.

Как мы знаем с помощью инструкций global и nonlocal функции могут изменять переменные, находящиеся за их пределами - эти инструкции не только обеспечивают доступ к переменным для записи, но и изменяют правила приявязки инструкций присваивания к пространствам имен:

X = 11 # Глобальная в модуле

def g1():

print(X) # Ссылка на глоабльную переменную в модуле

def g2():

global X

X = 22 # Изменит глобальную переменную в модуле

def h1():

X = 33 # Локальная в функции

def nested():

print(X) # Ссылка на локальную переменную в объемлющей функции

def h2():

X = 33 # Локальная в функции

def nested():

nonlocal X # Инструкция из Питон 3.0

X = 44 # Изменит локальную переменную в объемлющей функции

Конечно пространства имен в Питон позволяют предотвратить случайный конфликт имен, но это совсем не значит, что мы должны использовать переменные с одинаковыми именами, желательно так совсем не делать.

**Словари пространств имен.**

Мы знаем, что пространства имен модулей фактически реализованы как словари и доступны в виде встроенного атрибута \_\_dict\_\_. То же относится к объектам классов и экземпляров: обращение к квалифицированному имени атрибута фактич. явл-ся операцией доступа к эклементу словаря, а механизм наследования атрибута работает лишь как поиск в связанных словарях.

Фактич. объекты экземпляра и класса - это просто словари со ссылками ведущими вглубь интерпретатора.

Проследим, с пом. интерактив. оболочки, как растут словари пространств имен, когда в игру вступают классы (суперкласс и подкласс со своими данными):

>>> class Super:

def hello(self):

self.data1 = 'spam'

>>> class Sub(Super):

def hola(self):

self.data2 = 'eggs'

Когда мы создаем экземпляр подкласса, он начинает своё существование с пустым словарем пространства имен, но имеет ссылку на суперкласс. Фактич. дерево наследования доступно в виде спец. атрибутов, которые мы можем проверить. Экземпляры обладают атрибутом \_\_class\_\_, который ссылается на класс, а классы имеют атрибут \_\_bases\_\_, который явл-ся кортежем, содержащим ссылки на суперклассы выше в дереве наследования:

>>> X = Sub()

>>> X.\_\_dict\_\_ # Словарь пространства имен экземпляра

{}

>>> X.\_\_class\_\_ # Класс экземпляра

<class '\_\_main\_\_.Sub'>

>>> Sub.\_\_bases\_\_ # Суперклассы данного класса

(<class '\_\_main\_\_.Super'>,)

>>> Super.\_\_bases\_\_ # В Питон 2.6 возвращает пустой кортеж ()

(<class 'object'>,)

Т.к. в классах вып-ся присваивание атрибутам аргумента self, тем самым они заполняют объекты экземпляров, т.е. атрибуты включаются в словари пространств имен экземпляров, а не классов.

В пространство имен объекта экземпляра записываются данные, которые могут отличаться для разных экземпляров, и аргумент self явл-ся точкой входа в это пространство имен:

>>> Y = Sub()

>>> X.hello()

>>> X.\_\_dict\_\_

{'data1': 'spam'}

>>> X.hola()

>>> X.\_\_dict\_\_

{'data1': 'spam', 'data2': 'eggs'}

>>> list(Sub.\_\_dict\_\_.keys())

['\_\_module\_\_', 'hola', '\_\_doc\_\_']

>>> list(Super.\_\_dict\_\_.keys())

['\_\_dict\_\_', '\_\_weakref\_\_', '\_\_doc\_\_', '\_\_module\_\_', 'hello']

>>> Y.\_\_dict\_\_

{}

Обратите внимание на имена в словарях классов, содержащие символы подчеркивания, - эти имена определяются интерпретатором автоматич. Большинство из них обычно не использ-ся в программах, но сущ-ют такие инструменты, которые используют некоторые из этих имен (напр. \_\_doc\_\_ - хранит строки документирования).

А на примере второго экземпляра Y мы видим, что экземпляры имеют каждый свой независимый словарь, который изначально пуст, что мы и видим - несмотря на полный словарь экземпляра Х, словарь экземпляра Y по-прежнему пуст.

Т.к. атрибуты фактич. явл-ся ключами словаря, существует два способа получать и изменять их значения - по квалифицированным именам и индексированием по ключу:

>>> X.data1, X.\_\_dict\_\_['data1']

('spam', 'spam')

>>> X.data3 = 'toast'

>>> X.\_\_dict\_\_

{'data1': 'spam', 'data3': 'toast', 'data2': 'eggs'}

>>> X.\_\_dict\_\_['data3'] = 'ham'

>>> X.data3

'ham'

Такая эквивалентность применяется только к атрибутам, фактич. присоединенным к экземпляру. Т.к. обращение по квалифиц. имени запускает поиск по дереву наследования, такой сопосб может обеспечивать доступ к атрибутам, которые нельзя получить индексирвоанием словаря. Напр. унаследованный атрибут X.hello недоступен через выражение X.\_\_dict\_\_['hello'].

Ниже увидим работу ф-ции dir с объектами классов и экземпляров. Эта ф-ция прим-ся к объектам, имеющим атрибуты: dir(object) напоминает вызов object.\_\_dict\_\_.keys(). Ф-ция dir сортирует свой список и включает в него некоторые системные атрибуты, а в 3.0 она добавляет имена, унаследованные от класса object, котоырй явл-ся суперклассом всех классов:

>>> X.\_\_dict\_\_, Y.\_\_dict\_\_

({'data1': 'spam', 'data3': 'ham', 'data2': 'eggs'}, {})

>>> list(X.\_\_dict\_\_.keys())

['data1', 'data3', 'data2']

>>> dir(X)

['\_\_class\_\_', '\_\_delattr\_\_', '\_\_dict\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_le\_\_', '\_\_lt\_\_', '\_\_module\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', '\_\_weakref\_\_', 'data1', 'data2', 'data3', 'hello', 'hola']

>>>

>>> dir(Sub)

['\_\_class\_\_', '\_\_delattr\_\_', '\_\_dict\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_le\_\_', '\_\_lt\_\_', '\_\_module\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', '\_\_weakref\_\_', 'hello', 'hola']

>>>

>>> dir(Super)

['\_\_class\_\_', '\_\_delattr\_\_', '\_\_dict\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_le\_\_', '\_\_lt\_\_', '\_\_module\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', '\_\_weakref\_\_', 'hello']

**Ссылки на пространства имен.**

Атрибуты \_\_class\_\_ и \_\_bases\_\_ позволяют осматриват ьиерархии наследования в нашем коде. Напр. их можно использ. для отображения дерева классов на экране, пример:

# classtree.py

"""

Выполянет обход дерева наследования снизу вверх, используя ссылки на пространства имен, и отображает суперклассы с отступами

"""

def classtree(cls, indent):

print('.' \* indent + cls.\_\_name\_\_) # Вывести имя класса

for supercls in cls.\_\_bases\_\_: # Рекурсивный обход всех суперклассов

classtree(supercls, indent+3) # Каждый суперкласс может быть посещен

# более одного раза

def instancetree(inst):

print('Tree of', inst) # Показать экземпляр

classtree(inst.\_\_class\_\_, 3) # Взойти к его классу

def selftest():

class A: pass

class B(A): pass

class C(A): pass

class D(B, C): pass

class E: pass

class F(D, E): pass

instancetree(B())

instancetree(F())

if \_\_name\_\_ == '\_\_main\_\_': selftest()

Функция classtree в этом сценарии явл-ся рекурсивной - она выводит имя класса, используя атрибут \_\_name\_\_, и затем начинает подъем к суперклассам, вызывая саму себя. Это позволяет ф-ции выполнять обход деревьев классов произвольной формы - в процессе рекурсии выполняется подъем по дереву и заканчивается по достижении корневых суперклассов, у которых атрибут \_\_bases\_\_ пуст.

Запустим наш сценарий, он построи тпустое дерево классов, создаст два экземпляра и выведет структуры классов, соотв. им:

C:\Python34>classtree.py

Tree of <\_\_main\_\_.selftest.<locals>.B object at 0x017CF7D0>

...B

......A

.........object

Tree of <\_\_main\_\_.selftest.<locals>.F object at 0x017CF7D0>

...F

......D

.........B

............A

...............object

.........C

............A

...............object

......E

.........object

- здесь видим, что в 3.0 у нас появляется суперкласс всех объектов object (в 2.6 его нет). Отступы, отмеченные точками, обозначают высоту в дереве классов.

Мы можем импортировать эту ф-цию везде где нам нужно будет отобразить дерево классов:

>>> class Emp: pass

>>> class Person(Emp): pass

>>> bob = Person()

>>> import classtree

>>>

>>> classtree.instancetree(bob)

Tree of <\_\_main\_\_.Person object at 0x026B64F0>

...Person

......Emp

.........object

**Еще раз о строках документирования.**

Строки, описывающие модули так же могут использ-ся для описания компонентов класса, причем они могут быть везде, во всех инструкциях программы.

Пример, docstr.py - показаны места в коде, где могут быть строки документирования:

"I am: docstr.\_\_doc\_\_"

def func(args):

"I am: docstr.func.\_\_doc\_\_"

pass

class spam:

"I am: spam.\_\_doc\_\_ or docstr.spam.\_\_doc\_\_"

def method(self, arg):

"I am: spam.method.\_\_doc\_\_ or self.method.\_\_doc\_\_"

pass

Преимущество строк документирования в том, что их содержимое доступно во время выполнения. Т.е. если текст оформлен в виде строк документирования, можно будет обратиться к атрибуту \_\_doc\_\_ объекта, чтобы получить его описание:

>>> import docstr

>>> docstr.\_\_doc\_\_

'I am: docstr.\_\_doc\_\_'

>>> docstr.func.\_\_doc\_\_

'I am: docstr.func.\_\_doc\_\_'

>>> docstr.spam.\_\_doc\_\_

'I am: spam.\_\_doc\_\_ or docstr.spam.\_\_doc\_\_'

>>> docstr.spam.method.\_\_doc\_\_

'I am: spam.method.\_\_doc\_\_ or self.method.\_\_doc\_\_'

Есть еще PyDoc - инструмент, который позволяет формировать отчеты из всех этих строк. Пример:

>>> help(docstr)

Help on module docstr:

NAME

docstr - I am: docstr.\_\_doc\_\_

CLASSES

builtins.object

spam

class spam(builtins.object)

| I am: spam.\_\_doc\_\_ or docstr.spam.\_\_doc\_\_

|

| Methods defined here:

|

| method(self, arg)

| I am: spam.method.\_\_doc\_\_ or self.method.\_\_doc\_\_

|

| ----------------------------------------------------------------------

| Data descriptors defined here:

|

| \_\_dict\_\_

| dictionary for instance variables (if defined)

|

| \_\_weakref\_\_

| list of weak references to the object (if defined)

FUNCTIONS

func(args)

I am: docstr.func.\_\_doc\_\_

FILE

/home/user/docstr.py

Строки документирования лучше использ-ть для функционального описания (что делают объекты), а комментарии "#" - для небольших пояснений (как действуют выражения, напр.).

**Классы и модули.** стр. 791

Различия заметить трудно. Опишем объекты в двух словах.

Модули:

- Это пакеты данных и исполняемого кода;

- Создаются как файлы с программным кодом на языке Питон или как расширения на языке С;

- Задействуются операцией импортирования.

Классы:

- Реализуют новые объекты;

- Создаются с помощью инструкции class;

- Задействуются операцией вызова;

- Всегда располагаются внутри модуля.

Классы так же поддерживают перегрузку операторов, создание множества экземпляров и наследование.

Нужно понимать, что несмотря на то, что классы и модуля явл-ся пространствами имен, между ними есть существенные различия.

**Глава 29. Перегрузка операторов.** стр. 794

**Основы.**

Термин "перегрузка операторов" означает всего лишь перехватывание встроенных операций с пом. методов классов - интерпретатор автоматич. вызывает эти методы при выполнении встр. операций над экземплярами классов, а методы должны возвращать значения, которые будут интерпретироваться как результаты соответствующих операций.

Обзор ключевых идей, в основе механизма перегрузки:

- Перегрузка операторов в Питон позволяет классам участвовать в обычных операциях.

- Классы в Питон могут перегружать такие операции, как вывод, вызов ф-ций, обращение к атрибутам и т.д.

- Перегрузка делает экземпляры классов более похожими на встр. типы.

- Перегрузка заключается в реализации в классах методов со спец. именами.

Т.е. если в классе есть метод со спец. именем, он будет автоматич. вызываться при выполнении соответствующей методу операции над экземплярами этого класса. Как мы знаем методы перегрузки не явл-ся обязательными. Если же они использ-ся, то дают классам имитировать интерфейсы встр. типов.

**Конструкторы и выражения: \_\_init\_\_ and \_\_sub\_\_**

Пример, класс Number в файле number.py, реализует метод перегрузки операции создания экземпляра (\_\_init\_\_), а также метод реализации операции вычитания (\_\_sub\_\_). Смотрим:

class Number:

def \_\_init\_\_(self, start): # Вызов Number(start)

self.data = start

def \_\_sub\_\_(self, other): # Выражение: экземпляр - other

return Number(self.data - other) # результат - новый экземпляр

>>> from number import Number # Извлекаем класс из модуля

>>> X = Number(5) # Number.\_\_init\_\_(X, 5)

>>> Y = X - 2 # Number.\_\_sub\_\_(X, 2)

>>> Y.data # Y - новый экземпляр класса Number

3

**Общие методы перегрузки операторов.**

Почти всё, что можно сделать с объектами встр. типов, такими как целые числа и списки, можно реализовать и в классах.

**Таблица 29.1 Общие методы перегрузки операторов.** стр. 795

Отображение операторов выражений или операций на методы, со спец именами (спец. имена - два подчерка в начале и конце имени) предопределяется языком Питон (и описывается стандарт. руководством).

Методы перегрузки могут наследоваться от суперклассов, как любые др. методы. Кроме того они явл-ся необязательными, если метод не релизован, значит соответствующая ему операция не поддерживается классом. Хотя некоторые встр. операции, напр. вывод, имеют реализацию по умолчанию (т.к. они наследуются от класса object, суперкласса всех классов, в 3.0).

**Доступ к элементам по индексу и извлечение срезов: \_\_getitem\_\_ и \_\_setitem\_\_**

Если метод \_\_getitem\_\_ есть в определении класса (или наследуется им), он автоматич. будет вызываться интерпретатором в случае применения операций индексирования к экземплярам.

Когда экземпляр Х появл-ся в выражении извлечения элемента по индексу, таком как X[i], интерпретатор Питона вызывает метод \_\_getitem\_\_, наследуемый этим экземпляром, передавая методу объект Х в первом аргументе и индекс, указанный в квадрат. скобках. Пример класса, возвращающего квадрат значения индекса:

>>> class Indexer:

def \_\_getitem\_\_(self, index):

return index \*\* 2

...

>>> X = Indexer()

>>> X[2] # Выражение X[i] вызывает X.\_\_getitem\_\_(i)

4

>>> for i in range(5):

print(X[i], end=' ') # Вызывает \_\_getitem\_\_(X, i) в каждой итерации

...

0 1 4 9 16

**Извлечение срезов.**

Метод \_\_getitem\_\_ вызывается не только при выполнении обращения по индексу, но и при извлечении срезов. Напомним извлечение среза на примере списка, где сипольз-ся верх. и ниж. границы и шаг:

>>> L = [5, 6, 7, 8, 9]

>>> L[2:4]

[7, 8]

>>> L[1:]

[6, 7, 8, 9]

>>> L[:-1]

[5, 6, 7, 8]

>>> L[::2]

[5, 7, 9]

На самом деле параметры среза определяются с пом. объекта среза, который и передается реализации операции индексирования списка. Фактич. мы всегда можем передать объект среза вручную:

>>> L[slice(2, 4)]

[7, 8]

>>> L[slice(1, None)]

[6, 7, 8, 9]

>>> L[slice(None, -1)]

[5, 6, 7, 8]

>>> L[slice(None, None, 2)]

[5, 7, 9]

Эта особенность имеет знач. для классов, реализующих \_\_getitem\_\_, - этот метод будет вызываться и для выполнения операций обращения к элементам по индексам (с целочисленным индексом), и для выполнения операций извлечения срезов (с объектом среза).

Наш класс в предыдущем примере не способен обрабатывать операцию среза, потому что его логика принимает лишь целочисленные индексы, реализуем другой класс, где в аргументе передается целое число, как и прежде:

>>> class Indexer:

data = [5, 6, 7, 8, 9]

def \_\_getitem\_\_(self, index): # Вызывается при индексировании или

print('getitem:', index) # извлечении среза

return self.data[index] # Выполнение индексирования

# или извлечения среза

>>> X = Indexer()

>>> X[0] # При индексировании \_\_getitem\_\_ получает целое число

getitem: 0

5

>>> X[1]

getitem: 1

6

>>> X[-1]

getitem: -1

9

А когда метод вызывается для извлечения среза, он получает объект среза, который просто передается списку, встроенному в класс Indexer, в виде выражения обращения по индексу:

>>> X[2:4]

getitem: slice(2, 4, None)

[7, 8]

>>> X[1:]

getitem: slice(1, None, None)

[6, 7, 8, 9]

>>> X[:-1]

getitem: slice(None, -1, None)

[5, 6, 7, 8]

>>> X[::2]

getitem: slice(None, None, 2)

[5, 7, 9]

Метод \_\_setitem\_\_ присваивания элементу по индексу, точно так же обслуживает обе операции - присваивание элементу по индексу и присваивание срезу. Он получает объект среза, который может передаваться другим операциям присваивания по индексу:

def \_\_setitem\_\_(self, index, value): # Реализует присваивание

... # по индексу или по срезу

self.data[index] = value # Присваивание по индексу или по срезу

До Питон 3.0 методами работы с индексами и срезами были \_\_getslice\_\_ и \_\_setslice\_\_. В Питон 3.0 этих методов сейчас нет.

Кроме того, метод \_\_index\_\_ не имеет отношение к индексированию - этот метод возвращает целое число, представляющее экземпляр, и используется встроенными типами, которые выполняют преобразование целых чисел в строку цифр:

>>> class C:

def \_\_index\_\_(self):

return 255

>>> X = C()

>>> hex(X)

'0xff'

>>> bin(X)

'0b11111111'

>>> oct(X)

'0o377'

Этот метод может использ-ся в операциях, требующих целое число, включая и индексирование:

>>> ('C' \* 256)[255]

'C'

>>> ('C' \* 256)[X] # Х используется как индекс (не Х[i])

'C'

>>> ('C' \* 256)[X:] # X используется как индекс (не X[i:])

'C'

**Итерации по индексам: \_\_getitem\_\_**

Опишем прием, не всегда понятный новчикам, но полезный на практике. Инструкция for многократно применяет операцию индексирования к последовательности, используя индексы от нуля и выше, пока не появится исключение выхода за границы.

Если метод \_\_getitem\_\_ реализован, инструкции циклов for буду твызывать его на каждом шаге цикла, с постоянным увеличением смещения. Т.о. знаем, что любой встроенный или определяемый пользователем объект, к котормоу применима операция индексирования, также может участвовать в итерациях:

>>> class Stepper:

def \_\_getitem\_\_(self, i):

return self.data[i]

>>> X = Stepper() # X - это экземпляр класса Stepper

>>> X.data = "Spam"

>>>

>>> X[1] # Индексирование, вызывается \_\_getitem\_\_

'p'

>>> for item in X: # Циклы for вызывают \_\_getitem\_\_

print(item, end=' ') # Инструкция for индексирует элементы 0..N

S p a m

Любой класс, поддерживающий циклы for, автоматич. поддерживает все итерационные контексты Питона. Например, оператор проверки на принадлженость in, генераторы списков, встр. ф-ция map, присваивание списков и кортежей и конструкторы типов также автоматич. вызывают метод \_\_getitem\_\_, если он определен, например:

>>> 'p' in X

True

>>> [c for c in X] # Генератор списков

['S', 'p', 'a', 'm']

>>> list(map(str.upper, X))

['S', 'P', 'A', 'M']

>>> (a, b, c, d) = X # Присваивание последовательностей

>>> a, c, d

('S', 'a', 'm')

>>> list(X), tuple(X), ''.join(X)

(['S', 'p', 'a', 'm'], ('S', 'p', 'a', 'm'), 'Spam')

>>> X

<\_\_main\_\_.stepper object at 0x0171FAD0>

**Итераторы: \_\_iter\_\_ и \_\_next\_\_** стр. 802

В настоящее время все итерационные контексты в Питон пытаются сначала использовать метод \_\_iter\_\_, и только потом - метод \_\_getitem\_\_, т.к. \_\_iter\_\_ обеспечивает более оптимальную поддержку итерационных конетекстов чем \_\_getitem\_\_.

Итерационные контексты вызывают встр. ф-цию iter, чтобы определить наличие метода \_\_iter\_\_, который должен возвращать объект итератора. Если он есть, то интерпретатор будет вызывать метод \_\_next\_\_ объекта итератора для получения элементов до тех пор, пока не будет возбуждено исключение StopIteration. Если метод \_\_iter\_\_ отсутствует, интерпретатор переходит к схеме с \_\_getitem\_\_ и начинает извлекать элементы по индексам, пока не будет исключение IndexError. Есть ф-ция next, вызов next(I) - это то же самое, что вызов I.\_\_next\_\_().

**Итераторы определяемые пользователем.**

Схема с методом \_\_iter\_\_ реализует в классах итераторы простой реализацией итерационного протокола. Файл iters.py определяет класс итератора, который возвращает квадраты числа:

class Squares:

def \_\_init\_\_(self, start, stop): # Сохранить состояние при создании

self.value = start - 1

self.stop = stop

def \_\_iter\_\_(self): # Возвращает итератор в iter()

return self

def \_\_next\_\_(self): # Возвращает квадрат в каждой итерации

if self.value == self.stop: # Также вызывается функцией next

raise StopIteration

self.value += 1

return self.value \*\* 2

>>> grom iters import Squares

SyntaxError: invalid syntax

>>> from iters import Squares

>>> for i in Squares(1, 5): # for вызывает iter(), который вызывает \_\_iter\_\_()

print(i, end=' ') # на каждой итерации вызывается \_\_next\_\_

...

1 4 9 16 25

Здесь объект итератора - это просто экземпляр self, поэтому метод \_\_next\_\_ явл-ся частью этого класса. В сложных ситуациях объект итератора может быть определен как отдельный класс и объект со своей собственной информацией о состоянии, с целью поддержки нескольких активных итераций на одних и тех же данных. Об окончании итераций интерпретатору сообщается с пом. инструкции raise.

Итерации по встр. типам можно также выполнять вручную:

>>> X = Squares(1, 5) # Итерации вручную выполняет инструкция цикла

>>> I = iter(X) # iter вызовет \_\_iter\_\_

>>> next(I) # next вызовет \_\_next\_\_

1

>>> next(I)

4

...

>>> next(I)

25

>>> next(I)

StopIteration

Такая реализация явл-ся более универсальной чем использование \_\_getitem\_\_. Но с другой стороны итераторы, реализованные на методе \_\_iter\_\_, иногда могут оказаться более сложными и менее удобными, чем метод \_\_getitem\_\_. Но они действительно предназначены для итераций, а не для случайного индексирования, фактич. они вообще не перегружают операцию индексирования:

>>> X = Squares(1, 5)

>>> X[1]

...

TypeError: 'Squares' object does not support indexing

Схема на основе \_\_iter\_\_ реализована во всех итерационных контекстах, к которым применим метод \_\_getitem\_\_ (проверка на вхождение, конструкторы, присваивание последовательностей и т.д.). В отличие от \_\_getitem\_\_, схема с \_\_iter\_\_ предназначена для выполнения обхода элементов всего один раз, а не несколько:

>>> X = Squares(1, 5)

>>> [n for n in X] # Получить все элементы

[1, 4, 9, 16, 25]

>>> [n for n in X] # Теперь объект пуст

[]

>>> [n for n in Squares(1, 5)] # Создать новый объект итератора

[1, 4, 9, 16, 25]

>>> list(Squares(1, 3))

[1, 4, 9]

Этот пример можно реализовать даже проще, применив функции-генраторы. Ф-ция, в отличие от класса, автоматич. сохраняет информацию о своем состоянии между итерациями. Конечно можно было бы использ-ть просто цикл for, ф-цию map или генератор списков, но классы могут оказаться лучше при моделировании более сложных итераций, пример рассмотрим далее.

**Несколько итераторов в одном объекте.**

Мы знаем, что объект итератора может быть определен как отдельный класс. Сейчас посмотрим, что происходит при выполнении обхода элементов встроенных типов, напр. строки:

>>> S = 'ace'

>>> for x in S:

for y in S:

print(x + y, end=' ')

aa ac ae ca cc ce ea ec ee

- для каждого элемента строки мы делали обход каждого из элементов строки, это понятно по выводу.

Здесь, внеш. цикл получает итератор строки вызовом ф-ции iter и каждый вложенный цикл делает то же самое, чтобы получить независимый итератор. Т.к. каждый итератор хранит собственную инфу о состоянии, каждый цикл управляет своим собственным положением в строке, независимо от любых других активных циклов.

Ранее мы уже видлели такие примеры, напр. функции-генераторы и выражения-генераторы, встр. функции map и zip.

При создании собств. итераторов мы сами выбираем делать единств. итератор или множество независимых. Чтобы сделать множество итераторов, метод \_\_iter\_\_ должен не просто возвращать аргумент self, а создавать новый объект итератора со своей информацией о состоянии.

Пример, класс итератора, который пропускает каждый второй элемент:

class SkipIterator:

def \_\_init\_\_(self, wrapped):

self.wrapped = wrapped # Информация о состоянии

self.offset = 0

def \_\_next\_\_(self):

if self.offset >= len(self.wrapped): # Завершить итерацию

raise StopIteration

else:

item = self.wrapped[self.offset] # Иначе перешагнуть и вернуть

self.offset += 2

return item

class SkipObject:

def \_\_init\_\_(self, wrapped): # Сохранить используемый элемент

self.wrapped = wrapped

def \_\_iter\_\_(self):

return SkipIterator(self.wrapped) # Каждый раз новый итератор

if \_\_name\_\_ == '\_\_main\_\_':

alpha = 'abcdef'

skipper = SkipObject(alpha) # Создать объект-контейнер

I = iter(skipper) # Создать итератор для него

print(next(I), next(I), next(I)) # Обойти элементы 0, 2, 4

for x in skipper: # for вызывает \_\_iter\_\_ автоматически

for y in skipper: # Вложенные циклы for также вызывают \_\_iter\_\_

print(x + y, end=' ') # Каждый итератор помнит своё состояние, смещение

Замечание! В книге была ошибка (намеренно или нет, не знаю) в классе SkipIterator имя метода \_\_next\_\_ было написано без подчеркиваний, просто next, поэтому мы получали ошибку:

C:\Python34>skipper.py

I = iter(skipper)

TypeError: iter() returned non-iterator of type 'SkipIterator'

Когда я написал имя метода правильно (раскинул мозгами чуток), то всё заработало как надо:

C:\Python34>skipper.py

a c e

aa ac ae ca cc ce ea ec ee

Ранее, наш класс Squares поддерживал всего одну активную итерацию, во влож. циклах нужно было вызывать Squares снова, чтобы получить новый объект.

Здесь у нас имеется единственный объект SkipObject, который создает множество объектов итераторов.

Как обычно, то же самое можно сделать с пом. встр. инструментов, напр. получением среза с третьим граничным значением, для пропуска элементов:

>>> S = 'abcdef'

>>> for x in S[::2]:

for y in S[::2]:

print(x + y, end=' ')

aa ac ae ca cc ce ea ec ee

Но есть отличия. Первое - каждое выражение извлечения среза физически сохраняет весь список с результатами в памяти, тогда как итераторы воспроизводят по одному значению за раз, что очень экономит память, если объем результатов большой. Второе - операции извлечения среза создают новые объекты, поэтому, на самом деле, итерации не протекают одновременно в одном и том же объекте.

Чтобы сделать реализацию похожую на класс нам нужно создать единственный объект для обхода, заранее выполнив извлечение среза:

>>> S = 'abcdef'

>>> S = S[::2]

>>> S

'ace'

>>> for x in S:

for y in S:

print(x + y, end=' ')

aa ac ae ca cc ce ea ec ee

- Но здесь по-прежнему список с результатами целиком хранится в памяти.

Итераторы, определяемые пользователем - это мощный инструмент, с их помощью можно создавать произвольные объекты, которые выглядят и ведут себя подобно другим последовательностям и итерируемым объектам.

Мы даже могли бы использ-ть этот механизм, напр. для создания объекта базы данных, чтобы одновременно выполнять несколько итераций в одном и том же наборе данных.

**Проверка на вхождение: \_\_contains\_\_, \_\_iter\_\_ и \_\_getitem\_\_**

Перегрузка операторов нередко образует многослойную архитектуру: классы могут предоставлять реализацию специфических методов или обощенные альтернативы, используемые в крайнем случае. Например:

- Операции сравнения в 2.6 используют спец. методы, такие ка к\_\_lt\_\_, если они присутствуют, или более обощенный метод \_\_cmp\_\_. В 3.0 используются только специализированные методы, а \_\_cmp\_\_ не использ.

- Операция проверки логич. значения сначала пробует вызвать специализ. метод \_\_bool\_\_ (возвращает явное True или False), а если его нет, то вызывает более обощенный метод \_\_len\_\_ (возвращаемое ненулевое значение трактуется как True). В 2.6 действия такие же, но вместо \_\_bool\_\_ используется метод \_\_nonzero\_\_.

К итерациям, обычно классы реализуют оператор in проверки на вхождение - либ ов виде метода \_\_iter\_\_, либо \_\_getitem\_\_. Но можно и более специализ. версию, реализовав метод \_\_contains\_\_, - он имеет преимущество перед \_\_iter\_\_, который, в свою очередь, имеет преимущество перед \_\_getitem\_\_. В случае отображений \_\_contains\_\_ должен определять членство, прмиеняя ключи (быстрый поиск), а в случае последовательностей - производить поиск.

Пример класса, реализует все три метода, выполняет проверку на вхождение, и попробуем использовать экземпляр класса в итерационных контекстах:

class Iters:

def \_\_init\_\_(self, value):

self.data = value

def \_\_getitem\_\_(self, i): # Крайний случай для итераций

print('get[%s]:' % i, end='') # А также для индексирования и срезов

return self.data[i]

def \_\_iter\_\_(self): # Предпочтительный для итераций

print('iter=> ', end='') # Возможен только 1 активный итератор

self.ix = 0

return self

def \_\_next\_\_(self):

print('next:', end='')

if self.ix == len(self.data): raise StopIteration

item = self.data[self.ix]

self.ix += 1

return item

def \_\_contains\_\_(self, x): # Предпочтительный для оператора "in"

print('contains: ', end='')

return x in self.data

X = Iters([1, 2, 3, 4, 5]) # Создать экземпляр

print(3 in X) # Проверка на вхождение

for i in X: # Циклы

print(i, end=' | ')

print()

print([i \*\* 2 for i in X]) # Другие итерационные контексты

print(list(map(bin, X)))

I = iter(X) # Обход вручную (именно так действуют

while True: # другие итерационные конексты)

try:

print(next(I), end=' @ ')

except StopIteration:

break

При запуске специализир. метод \_\_contains\_\_ будет вызван операцией проверки на вхождение, \_\_iter\_\_ будет вызываться в контексте итераций, в ходе которых многократно вызывается метод \_\_next\_\_, а метод \_\_getitem\_\_ не будет вызван ни разу:

contains: True

iter=> next:1 | next:2 | next:3 | next:4 | next:5 | next:

iter=> next:next:next:next:next:next:[1, 4, 9, 16, 25]

iter=> next:next:next:next:next:next:['0b1', '0b10', '0b11', '0b100', '0b101']

iter=> next:1 @ next:2 @ next:3 @ next:4 @ next:5 @ next:

Посмотрим, что будет если закомментировать метод \_\_contains\_\_, теперь операция проверки на вхождение будет использовать обобщенный метод \_\_iter\_\_:

iter=> next:next:next:True

iter=> next:1 | next:2 | next:3 | next:4 | next:5 | next:

iter=> next:next:next:next:next:next:[1, 4, 9, 16, 25]

iter=> next:next:next:next:next:next:['0b1', '0b10', '0b11', '0b100', '0b101']

iter=> next:1 @ next:2 @ next:3 @ next:4 @ next:5 @ next:

А теперь закомментим оба метода \_\_contains\_\_ и \_\_iter\_\_ - при проверке на вхождение и в других итерациях использ-ся \_\_getitem\_\_, которому индексы передаются в порядке возрастания:

get[0]:get[1]:get[2]:True

get[0]:1 | get[1]:2 | get[2]:3 | get[3]:4 | get[4]:5 | get[5]:

get[0]:get[1]:get[2]:get[3]:get[4]:get[5]:[1, 4, 9, 16, 25]

get[0]:get[1]:get[2]:get[3]:get[4]:get[5]:['0b1', '0b10', '0b11', '0b100', '0b101']

get[0]:1 @ get[1]:2 @ get[2]:3 @ get[3]:4 @ get[4]:5 @ get[5]:

Как видим, метод \_\_getitem\_\_ явл-ся еще более обощенным чем \_\_iter\_\_: помимо итераций он используется в индексировании и извлечение срезов.

При извлечении среза методу \_\_getitem\_\_ передается объект, содержащий параметры среза, благодаря этому наш класс автоматич. поддерживает операцию среза:

>>> import iters

contains: True

iter=> next:1 | next:2 | next:3 | next:4 | next:5 | next:

iter=> next:next:next:next:next:next:[1, 4, 9, 16, 25]

iter=> next:next:next:next:next:next:['0b1', '0b10', '0b11', '0b100', '0b101']

iter=> next:1 @ next:2 @ next:3 @ next:4 @ next:5 @ next:

>>> X = iters.Iters('spam') # Индексирввоание

>>> X[0] # \_\_getitem\_\_(0)

get[0]:'s'

>>> 'spam'[1:] # Извлечение среза

'pam'

>>> 'spam'[slice(1, None)] # Объект среза

'pam'

>>> X[1:] # \_\_getitem\_\_(slice(..))

get[slice(1, None, None)]:'pam'

>>> X[:-1]

get[slice(None, -1, None)]:'spa'

В реальных случаях, когда класс не явл-ся последовательностью, реализация метода \_\_iter\_\_ может оказаться совсем простой, потому что в это случае нет необходимости управлять целочисленными индексами, а метод \_\_contains\_\_ позволяет реализовать более оптимальный способ проверки на вхождение.

**Обращения к атрибутам: \_\_getattr\_\_ и \_\_setattr\_\_**

Метод \_\_getattr\_\_ выполняет операцию получения ссылки на атрибут. Т.е. он вызывается с именем атрибута в виде строки всякийц раз, когда обнаруживается попытка получить ссылку на неопределенный (несуществующий) атрибут. Этот метод не вызывается, если интерпретатор может обнаружить атрибут посредством выполнения процедуры поиска в дереве наследования. Вследствие этого метод \_\_getattr\_\_ удобно использовать для обобщенной обработки запросов к атрибутам. Например:

>>> class empty:

def \_\_getattr\_\_(self, attrname):

if attrname == "age":

return 40

else:

raise AttributeError(attrname)

>>> X = empty()

>>> X.age

40

>>> X.name

...

AttributeError: name

- кстати в книге была снова ошибка, намеренно или нет не знаю. В последней строке был такой синтаксис "raise AttributeError, attrname" - в итоге я исправил на рабочую версию - "raise AttributeError(attrname)".

В нашем примере класс empty и его экземпляр X не имеют своих атрибутов, поэтому, при обращение к атрибуту X.age вызывается метод \_\_getattr\_\_ - в аргументе self передается экземпляр (Х), а в аргументе attrname - строка с именем неопределенного атрибута ("age"). В результате мы получаем атрибут, вычисляемый динамически.

Для атрибутов, которые не предусмотрены классом, \_\_getattr\_\_ возбуждает исключение AttributeError, сообщая интерпретатору, что это неопределенные имена, - ошибка когда пытаемся обратиться к X.name.

Родственный метод перегрузки \_\_setattr\_\_ перехватывает ВСЕ попытки присваивания значений атрибутам. Если \_\_setattr\_\_ определен, выражение self.attr = value будет преобразовано в вызов метода self.\_\_setattr\_\_('attr', value).

Нужно быть внимательным, любая попытка присваивания любому атрибуту аргумента self приводит к повторному вызову метода \_\_setattr\_\_, вызывая бесконечный цикл рекурсивных вызовов!

Если нужно его использовать, все присваивания придется выполнять посредством словаря атрибутов. Используйте self.\_\_dict\_\_['name'] = x, а не self.name = x:

>>> class accesscontrol:

def \_\_setattr\_\_(self, attr, value):

if attr == 'age':

self.\_\_dict\_\_[attr] = value

else:

raise AttributeErros(attr + ' not allowed')

>>> X = accesscontrol()

>>> X.age = 40

>>> X.age

40

>>> X.name = 'mel'

...

NameError: name 'AttributeErros' is not defined

**Другие способы управления атрибутами.**

В будущем нам могут понадобиться и др. способы управления атрибутами:

- Метод \_\_getattribute\_\_ вызывается при обращении к любым атрибутам, не только к неизвестным, но при реализации его следует быть осторожнее, чем при реализации \_\_getattr\_\_, чтобы избежать зацикливания.

- Встроенная ф-ция property позволяет ассоциировать специальные методы с операциями чтения и записи над определенными атрибутами класса.

- Дескрипторы предоставляют возможность ассоциировать методы \_\_get\_\_ и \_\_set\_\_ класса с операциями доступа к определенным атрибутам класса.

Эти доп. средства мы обсудим позднее.

**Имитация частных атрибутов экземпляра: часть 1**

Фрагмент явл-ся обощением предыдущего примера и позволяет каждому подклассу иметь свой перечень частных имен атрибутов, которым нельзя присваивать значения в экземплярах:

class PrivateExc(Exception): pass # Об исключениях позднее

class Privacy:

def \_\_setattr\_\_(self, attrname, value): # Вызывается self.attrname = value

if attrname in self.privates:

raise PrivateExc(attranme, self)

else:

self.\_\_dict\_\_[attrname] = value # self.attrname = value

# вызовет зацикливание, не забываем!

class Test1(Privacy):

privates = ['age']

class Test2(Privacy):

privates = ['name', 'pay']

def \_\_init\_\_(self):

self.\_\_dict\_\_['name'] = 'Tom'

В результате получим:

>>> from privac import Test1, Test2

>>> x = Test1()

>>> y = Test2()

>>> x.name = 'Bob'

>>> y.name = 'Sue'

NameError: name 'attranme' is not defined

>>> y.age = 30

>>> x.age = 40

NameError: name 'attranme' is not defined

Это лишь первая прикидочная реализация частных атрибутов в Питон (т.е. запрет на изменение атрибутов вне класса - разъяснили хоть, а то я или забыл или не объяснялось, что такое частные атрибуты).

Питон не может объявлять частные атрибуты, но такие приемы могут их имитировать. Но это решение нужно дополнить, необходима возможность изменять значения частных атрибутов из подклассов и использовать метод \_\_getattr\_\_ и класс-обертку (иногда наз-ся прокси-класс), чтобы контролировать получение значений частных атрибутов.

Позднее для полной реализации мы будем использовать декораторы классов. Тем не менее этот способ сокрытия атрибутов редко используется на практике. Программисты на Питоне способны писать крупные системы без частных объявлений.

Перехват операций обращения к атрибутам и присваивание им значений - вообще очень полезный прием. Он обеспечивает возможность делегирования - способ, позволяющий обертывать встр. объекты объектами-контроллерами, добавлять новое поведение и делегировать выполнение операций обернутым объектам.

**Строковое представление объектов: \_\_repr\_\_ и \_\_str\_\_**

Пример, реализуем \_\_init\_\_, \_\_add\_\_ и \_\_repr\_\_, котоырй возвращает строковое представление экземпляров. В строку будем преобразовывать self.data. Если метод \_\_repr\_\_ (или родственный ему \_\_str\_\_) определен, он атоматически будет вызываться при попытках вывести экземпляр класса или преобразовать его в строку. Т.е. вывод наших объекто в удобочитаемом формате.

Строковое представление по умолчанию неудобочитаемое:

class adder:

def \_\_init\_\_(self, value=0):

self.data = value

def \_\_add\_\_(self, other):

self.data += other

>>> x = adder() # Формат отображения по умолчанию

>>> print(x)

<\_\_main\_\_.adder object at 0x02407ED0>

>>> x

<\_\_main\_\_.adder object at 0x02407ED0>

Используя методы преобразования в строковое представление, можем выводить информацию и форматировать её:

class addrepr(adder): # Наследуте \_\_init\_\_, \_\_add\_\_ из класса adder

def \_\_repr\_\_(self): # Добавляет строковое представление

return 'addrepr(%s)' % self.data # Преобразует в строку программного кода

>>> x = addrepr(2) # Вызовет \_\_init\_\_

>>> x + 1 # Вызовет \_\_add\_\_

>>> x ` # Вызовет \_\_repr\_\_

addrepr(3)

>>> print(x) # Вызовет \_\_repr\_\_

addrepr(3)

>>> str(x), repr(x) # Вызовет \_\_repr\_\_

('addrepr(3)', 'addrepr(3)')

Почему имеется два метода? Дело вот в чем:

- Встр. ф-ции print и str (а также её внутр. эквивалент, котоырй использ-ся ф-цией print) сначала пытаются использовать метод \_\_str\_\_. Вообще этоот метод должен возвращать строковое представление объекта в удобном виде.

- Во всех сотальных случаях использ. метод \_\_repr\_\_: ф-цией автоматич. вывода в интерактив. оболочке, ф-цией repr, при выводе влож. объектов, а также ф-циями print и str, когда в классе отсутствует метод \_\_str\_\_. Этот метод должен возвращать строку, которая могла бы использ-ся как программ. код для воссоздания объекта или содержать информацию, полезную для разработчиков.

Др. словами, метод \_\_repr\_\_ использ-ся везде, за исключениме ф-ций print и str, если определен метод \_\_str\_\_. Но есди \_\_str\_\_ отсутствует, операции вывода будут использ. \_\_repr\_\_, но не наоборот - в остальных случаях, напр. ф-цией автоматич. вывода в интерактив. оболочке всегда использ-ся\_\_repr\_\_; даже не бывает попыток использовать \_\_str\_\_:

class addstr(adder):

def \_\_str\_\_(self): # есть \_\_str\_\_, но нет \_\_repr\_\_

return '[Value: %s]' % self.data # Преобразовать в красивую строку

>>> x = addstr(3)

>>> x + 1

>>> x # По умолч. выз-ся \_\_repr\_\_, которого нет

<\_\_main\_\_.addstr object at 0x024D6910>

>>> print(x) # Вызовет \_\_str\_\_

[Value: 4]

>>> str(x), repr(x)

('[Value: 4]', '<\_\_main\_\_.addstr object at 0x024D6910>')

Поэтому, если нам нужно единое ототбражение во всех контекстах, лучше использовать метод \_\_repr\_\_. Но можна определить сразу оба метода. Для конечного пользователя - \_\_str\_\_, а для программиста спец. инфа будет выводиться \_\_repr\_\_:

class addboth(adder):

def \_\_str\_\_(self):

return '[Value: %s]' % self.data # Удобочитаемая строка

def \_\_repr\_\_(self):

return 'addboth(%s)' % self.data # Строка программного кода

>>> x = addboth(4)

>>> x + 1

>>> x # Вызовет \_\_repr\_\_

addboth(5)

>>> print(x) # Вызовет \_\_str\_\_

[Value: 5]

>>> str(x), repr(x)

('[Value: 5]', 'addboth(5)')

Замечания по использованию. Первое, - оба метода, \_\_str\_\_ и \_\_repr\_\_ должны возвращать строки, поэтому не забывайте выполнять преобразование в случае необходимости. Второе, - в зависимости от логики преобразования, реализованной в контейнерном объекте, операция вывода может вызывать метод \_\_str\_\_ только для объектов верхнего уровня - влож. объекты по-прежнему могут выводиться с применением их методов \_\_repr\_\_ или метода по умолчанию. Пример:

class Printer:

def \_\_init\_\_(self, val):

self.val = val

def \_\_str\_\_(self): # Используется для вывода самого экземпляра

return str(self.val) # Преобразует результат в строку

>>> obj = [Printer(2), Printer(3)]

>>> for x in obj: print(x) # При выводе экземпляра будет вызван \_\_str\_\_,

... # но не тогда, когда экземпляр 2 находится в списке!

2

3

>>> print(obj)

[<\_\_main\_\_.Printer object at 0x02417690>, <\_\_main\_\_.Printer object at 0x024D6B70>]

>>> obj

[<\_\_main\_\_.Printer object at 0x02417690>, <\_\_main\_\_.Printer object at 0x024D6B70>]

Чтобы реализовать адаптированную версию метода, независимую от реализации контейнера, реализуйте метод \_\_repr\_\_, а не \_\_str\_\_:

class Printer:

def \_\_init\_\_(self, val):

self.val = val

def \_\_repr\_\_(self): # \_\_repr\_\_ использ-ся print, если нет \_\_str\_\_

return str(self.val) # \_\_repr\_\_ используетс интерактивной оболочкой

# и для вывода влож. объектов

>>> obj = [Printer(2), Printer(3)]

>>> for x in obj: print(x) # Нет \_\_str\_\_: вызовет \_\_repr\_\_

...

2

3

>>> print(obj) # Вызовет \_\_repr\_\_, а не \_\_str\_\_

[2, 3]

>>> obj

[2, 3]

На практике, метод \_\_str\_\_ (и его родственник \_\_repr\_\_) является вторым по частоте использования после \_\_init\_\_.

**Правостороннее сложение и операция приращения: \_\_radd\_\_ и \_\_iadd\_\_**

С технич. точки зрения, метод \_\_add\_\_ не поддерживает использ. объектов экземпляров справа от оператора +. Чтобы реализовать такую поддержку и обеспечить перестановку операндов, необходимо реализовать метод \_\_radd\_\_. Интерпретатор вызывает метод \_\_radd\_\_, только когда экземпляр нашего класса появл-ся справа от оператора +. Во всех др. случаях, когда объект появл-ся слева, вызывается метод \_\_add\_\_:

class Commuter:

def \_\_init\_\_(self, val):

self.val = val

def \_\_add\_\_(self, other):

print('add', self.val, other)

return self.val + other

def \_\_radd\_\_(self, other):

print('radd', self.val, other)

return other + self.val

>>> x = Commuter(88)

>>> y = Commuter(99)

>>> x + 1 # \_\_add\_\_: экземпляр + не\_экземпляр

add 88 1

89

>>> 1 + y # \_\_radd\_\_: не\_экземпляр + экземпляр

radd 99 1

100

>>> x + y # \_\_add\_\_: экземпляр + экземпляр

add 88 <\_\_main\_\_.Commuter object at 0x021F9C30>

radd 99 88

187

Обычное выполнение метода \_\_radd\_\_ понятно, достаточно обратить внимание на порядок следования операндов. А вот когда в выражении участвуют экземпляры разных классов (у нас ведь x и y экземпляры одного класса), интерпретатор предпочитает вызывать метод экземпляра, расположенного слева. Вообще, когда складываются экземпляры, интерпретатор вызывает метод \_\_add\_\_, который, в свою очередь, вызывает метод \_\_radd\_\_, упрощая левый операнд.

На практике, когда нужно рапространить тип класса на результат, может оказаться нужным убедиться в безопасности операции преобразования и избежать вложенности, проверкой типа. Если в следующем примере не проверить тип ф-цией isinstance, мы получим экземпляр класса Commuter, значением атрибута val которого, явл-ся другой экземпляр класса Commuter, при сложении двух экземпляров:

class Commuter: # Тип класса распространяется на результат

def \_\_init\_\_(self, val):

self.val = val

def \_\_add\_\_(self, other):

if isinstance(other, Commuter): other = other.val

return Commuter(self.val + other)

def \_\_radd\_\_(self, other):

return Commuter(other + self.val)

def \_\_str\_\_(self):

return '<Commuter: %s>' % self.val

>>> x = Commuter(88)

>>> y = Commuter(99)

>>> print(x + 10) # Результат - лругой экземпляр класса Commuter

<Commuter: 98>

>>> print(10 + y)

<Commuter: 109>

>>> z = x + y # Нет вложения: не происходит рекурсивный вызов \_\_radd\_\_

>>> print(z)

<Commuter: 187>

>>> print(z + 10)

<Commuter: 197>

>>> print(z + z)

<Commuter: 374>

**Комбинированная операция сложения.**

Чтобы сделать поддержку комбинированной операции сложения += нужно реализовать метод \_\_iadd\_\_ или \_\_add\_\_. Последний использ-ся при отсутствии первого. Пример:

class Number:

def \_\_init\_\_(self, val):

self.val = val

def \_\_iadd\_\_(self, other): # \_\_iadd\_\_ явно реализует операцию x += y

self.val += other # Обычно возвращает self

return self

>>> x = Number(5)

>>> x +=1

>>> x +=1

>>> x.val

7

class Number:

def \_\_init\_\_(self, val):

self.val = val

def \_\_add\_\_(self, other): # \_\_add\_\_ - как крайнее средство: x = (x + y)

return Number(self.val + other) # Распространяет тип класса

>>> x = Number(5)

>>>

>>> x +=1

>>> x +=1

>>> x.val

7

Любой двухместный оператор имеет похожий правосторонний метод перегрузки и метод, реализующий комбинированную операцию присваивания (\_\_mul\_\_, \_\_rmul\_\_ и \_\_imul\_\_). Правосторонние методы - это достаточно сложная тема и на практике использ-ся очень редко.

**Операция вызова: \_\_call\_\_**

Метод \_\_call\_\_ вызывается при обращении к экземпляру как к функции. Это не повторяющееся определение - если метод \_\_call\_\_ присутствует, интерпретатор будет вызывать его, когда экземпляр вызывается как фнукция, передавая ему любые позиционные и именованные аргументы:

class Callee:

def \_\_call\_\_(self, \*pargs, \*\*kargs): # Реализует вызов экземпляра

print('Called:', pargs, kargs) # Принимает любые аргументы

>>> C = Callee()

>>> C(1, 2, 3) # С - вызываемый объект

Called: (1, 2, 3) {}

>>> C(1, 2, 3, x=4, y=5)

Called: (1, 2, 3) {'x': 4, 'y': 5}

Метод \_\_call\_\_ поддерживает все схемы передачи аргументов, обсуждавшиеся в главе 18, - все, что передается экземпляру, передается этому методу наряду с обычным аргументом self, в котором передается сам экземпляр. Напр. следующие определения метода:

class C:

def \_\_call\_\_(self, a, b, c=5, d=6): ... # Обычные и со значениями

# по умолчанию

class C:

def \_\_call\_\_(self, \*pargs, \*\*kargs): ... # Произвольные аргументы

class C:

def \_\_call\_\_(self, \*pargs, d=6, \*\*kargs): ... # Аргументы, которые могут

# передаваться только по

# имени в версии 3.0

- соответствуют следующим вызовам экземпляра:

X = C()

X(1, 2) # Аргументы со значениями по умолчанию опущены

X(1, 2, 3, 4) # Позиционные

X(a=1, b=2, d=4) # Именованные

X(\*[1, 2], \*\*dict(c=3, d=4)) # Распаковывание произвольных аргументов

X(1, \*(2,), c=3, \*\*dict(d=4)) # Смешанные режимы

Суть в том, что классы и экземпляры, имеющие метод \_\_call\_\_, поддерживают тот же синтаксис и семантику передачи аргументов, что и обычные ф-ции и методы.

Реализация операции вызова, позволяет экземплярам имитировать поведение ф-ций, а также сохранять информацию о состоянии:

>>> class Prod:

def \_\_init\_\_(self, value): # Принимает единственный аргумент

self.value = value

def \_\_call\_\_(self, other):

return self.value \* other

>>> x = Prod(2) # "Запоминает" 2 в своей области видимости

>>> x(3) # 3 (передано) \* 2 (сохраненное значение)

6

>>> x(4)

8

В этом примере реализация метода \_\_call\_\_ может показаться ненужной. То же саоме можно было сделать с пом. простого метода:

>>> class Prod:

def \_\_init\_\_(self, value):

self.value = value

def comp(self, other):

return self.value \* other

>>> x = Prod(3)

>>> x.comp(3)

9

>>> x.comp(4)

12

Однако метод \_\_call\_\_ может оказаться удобнее при взаимодействии с прикладными интерфейсами, где ожидается ф-ция, - это позволяет создавать объекты, совместимые с интерфейсами, ожидающими получит ьф-цию, которые к тому же способны сохранять информацию о своем состоянии между вызовами. Ыактич. этот метод занимает третье место после - \_\_init\_\_, \_\_str\_\_ и \_\_repr\_\_.

**Функциональные интерфейсы и программный код обратного вызова.** стр. 818

Инструментальный набор для создания графич. интерфейса tkinter позволяет регистрироват ьф-ции как обработчики событий (они же - ф-ции обратного вызова); когда возникают какие-либо события, tkinter вызывает зарегистрированные объекты.

Если нам нужен обработчик событий, способный сохранять своё состояние между вызовами, мы можем зарегистрировать связанный метод класса, либо экземпляр класса, который с пом. метода \_\_call\_\_ обеспечивает совместимость с функциональным интерфейсом.

Позднее мы подробно изучим связанные методы, а пока разберем пример использ-ия метода \_\_call\_\_ для построения графич. интерфейса. Определим класс, который определяет объект, поддерживающий функциональный интерфейс, и имеет инфу о состоянии, сохраняя цвет, в который должна окрашиваться нажатая кнопка:

class Callback:

def \_\_init\_\_(self, color): # Функция + информация о состоянии

self.color = color

def \_\_call\_\_(self): # Поддерживает вызовы без аргументов

print('turn', self.color)

Теперь можем зарегистрировать экземпляры этого класса в контексте графич. интерфейса, как обработчики событий для кнопок, несмотря на то, что реализация графич. интерфейса предполагает вызывать обработчики событий как обычные ф-ции без аргументов:

cb1 = Callback('blue') # "Запомнить" голубой цвет

cb2 = Callback('green')

B1 = Button(command=cb1) # Зарегистрировать обработчик

B2 = Button(command=cb2) # Зарегистрировать обработчик

Когда кнопка будет нажата, объект экземпляра будет вызван как простая ф-ция, точно как в вызовах ниже. А т.к. он сохраняет инфу о состоянии в атрибутах экземпляра, он помнит, что нужно сделать:

cb1() # По событию: выведет 'blue'

cb2() # Выведет 'green'

Фактич. это один из лучших способов сохранения инфы о состоянии в Питон - он лучше способов знакомых нам, связанных с ф-циями (глобал. переменные, ссылки в области видимости объемлющей ф-ции и изменяемые аргументы со знач. по умолчанию). Благодаря ООП состояние можно сохранять явно, посредством присваивания значений атрибутам.

Рассмотрим еще два способа, используемые для сохранения инфы о состоянии в ф-циях обратного вызова. В первом варианте использ-ся lambda-функция с аргументами, имеющими занч. по умолчанию:

cb3 = (lambda color='red': 'turn' + color) # Или: по умолчанию

print(cb3())

Во-втором - используются связанные методы класса. Объект связанного метода - это объект, который запоминает экземпляр self и ссылку на ф-цию, так, что потом можно вызывать простую ф-цию без использования экземпляра:

class Callback:

def \_\_init\_\_(self, color): # Класс с инфой о состоянии

self.color = color

def changeColor(self): # Обычный метод

print('turn', self.color)

cb1 = Callback('blue')

cb2 = Callback('yellow')

B1 = Button(command=cb1.changeColor) # Ссылка, не вызов

B2 = Button(command=cb2.changeColor) # Запоминаются функция+self

Когда кнопка будет нажата, имитируется поведение графич. интерфейса и вызывается метод changeColor, котоырй обработает инфц о состоянии объекта:

object = Callback('blue')

cb = object.changeColor # Регистрация обработчика событий

cb() # По событию выведет 'blue'

Этот прием более простой, но менее универсальный чем перегрузка операции вызова с пом. метода \_\_call\_\_.

**Сравнение: \_\_lt\_\_, \_\_gt\_\_ и другие.**

Согласно табл. 29.1 классы могут определять методы, реализующие все шесть операций сравнения: <, >, <=, >=, ==, и !=. Имейте в виду некоторые моменты:

- В отличие от методоы \_\_add\_\_/\_\_radd\_\_, методы сравнения не имеют правосторонних версий. Вместо этого, когда операцию сравнения поддерживает только один операнд, использ-ся зеркальные методы сравнивания (методы \_\_lt\_\_ и \_\_gt\_\_ явл-ся зеркальными по отношению друг к другу).

- Среди операторов сравнения нет неявных взаимоотношений. Суть в том, что истинность операции == не предполагает ложность операции !=, т.е. чтобы гаранитровать корректное поведение обоих операторов, нужно реализовать оба метода, \_\_eq\_\_ и \_\_ne\_\_.

- В Питон 2.6 все орперации сравнения можно было реализовать в виде одного метода \_\_cmp\_\_ - он должен выполнить сравнивание и вернуть число меньшее, равное или больше нуля, чтобы показать, что аргумент self меньше, равен или больше второго аргумента соответственно. На практике использовалась ф-ция cmp(x, y). В Питон 3.0 и этот метод и ф-ция были удалены.

Приведем краткий пример:

class C:

data = 'spam'

def \_\_gt\_\_(self, other):

return self.data > other

def \_\_lt\_\_(self, other):

return self.data < other

>>> X = C()

>>> print(X > 'ham') # Вызовет \_\_gt\_\_

True

>>> print(X < 'ham') # Вызовет \_\_lt\_\_

False

**Метод \_\_cmp\_\_ в Питон 2.6 (удален в 3.0).**

Целочисленное возвращаемое значение этого метода используется для вычисления результата, возвращаемого оператором. Пример (версия 2.6 конечно):

class C:

data = 'spam'

def \_\_cmp\_\_(self, other):

return cmp(self.data, other)

X = C()

print(X > 'ham') # Выведет True

print(X < 'ham') # Выведет False

Хотя метода \_\_cmp\_\_ нет в 3.0 он по-прежнему может встречаться нам в программах, т.к. его удаление было неожиданным для большинства программистов и он широко использ-ся. Нам нужно знать большинство тонких моментов версии 2.6, для поддержки программ написанных ранее, а не только под редакцией версии Питона 3.0.

**Проверка логического значения: \_\_bool\_\_ и \_\_len\_\_.**

В логическом контексте - интерпретатор сначала пытается напрямую получить логическое значение с пом. метода \_\_bool\_\_ и только потом, если этот метод не реализован, пытается вызвать метод \_\_len\_\_, чтобы выяснить истинность объекта, исходя из его длины. Обычно первый метод возвращает логич. значение исходя из знач. атрибутов или др. инфы:

class Truth:

def \_\_bool\_\_(self): return True

>>> X = Truth()

>>> if X: print('yes!')

...

yes!

class Truth:

def \_\_bool\_\_(self): return False

>>> X = Truth()

>>> bool(X)

False

Если \_\_bool\_\_ отсутствует, то определсяется длина объекта, т.к. непустой объект интерпретируется как истинный (т.е. если длина не равно 0, то объект True, в противном случае - False):

class Truth:

def \_\_len\_\_(self): return 0

>>> X = Truth()

>>> if not X: print('no!')

...

no!

Если есть оба метода, то предпочтение отдается \_\_bool\_\_:

class Truth:

def \_\_bool\_\_(self): return True # в 3.0 первым опробуется \_\_bool\_\_

def \_\_len\_\_(self): return 0 # в 2.6 первым опробуется \_\_len\_\_

>>> X = Truth()

>>> if X: print('yes!')

...

yes!

Если ни один из методов не определен, объект просто считается истинным (что-то такое из метафизики):

>>> class Truth: pass

...

>>> X = Truth()

>>> bool(X)

True

**Логические значения в Питон 2.6**

В 2.6 вместо \_\_bool\_\_ нужно использ. \_\_nonzero\_\_. А вот метод \_\_len\_\_ в 2.6 также использ-ся в крайнем случае (можно провреить дома на своей убунте, у меня там обе версии установлены).

В 3.0 метод \_\_nonzero\_\_ в свою очередь работать не будет, как и наоборот \_\_bool\_\_ в 2.6.

**Уничтожение объектов: \_\_del\_\_**

Метод \_\_del\_\_ вызывается автоматич. когда освобождается память, занятая объектом (т.е. "сборка мусора"):

class Life:

def \_\_init\_\_(self, name='unknown'):

print('Hello', name)

self.name = name

def \_\_del\_\_(self):

print('Goodbye', self.name)

>>> brian = Life('Brian')

Hello Brian

>>> brian = 'Loretta'

Goodbye Brian

Когда переменной присваивается строка, теряется последняя ссылка на экземпляр класса Life, что приводит к вызову деструктора. Этот метод удобно использовать для реализации заключительных действий (как напр. завершение соединения с сервером). Однако в Питон деструкторы использ-ся не так часто как в др. ЯП.

С одной стороны интерпретатор автоматич. освобождает память, занятую экземпляром и деструктор не нужен как-бы. С другой стороны, не всегда можно предсказать, когда произойдет уничтожение экземпляра, поэтому часто лучше выполнять завершающие действия в явно вызываемом методе (или в инструкции try/finally) - в некоторых случаях в системных таблицах могут сохраняться ссылки на наши объекты, что может мешать вызову деструктора.

Еще недостаток - исключения внутри \_\_del\_\_ выводят сообщения в поток sys.stderr, а не вызывают события исключений. Кроме того, перекрестные (циклические) ссылки между объектами могут мешать сборке мусора.

**Глава 30. Шаблоны проектирования с классами.** стр. 828

**Python и ООП.**

Напомним о трех кратких идеях ООП:

- *Наследование.* Наследование основано на механизме поиска атрибутов в языке Питон (в выражение X.name).

- *Полиморфизм.* Назначение метода method в выражении X.method зависит от типа (класса) X.

- *Инкапсуляция.* Методы и операторы реализуют поведение; сокрытие данных - это соглашение по умолчанию.

Мы уже знаем, что такое наследование, достаточно хорошо изучили его. Знакомы с полиморфизмом - он произрастает из отсутствия объявлений типов в языке Питон. Т.к. разрешение имен атрибутов производится на этапе выполнения, объекты, реализующие одинаковые интерфейсы, явл-ся взаимозаменяемыми - клиентам не требуется знать тип объекта, реализующего вызываемый метод.

Инкапсуляция в Питон означает упаковывание - т.е. сокрытие подробностей реализации за интерфесом объекта. Это не означает принудительное сокрытие, но оно может быть реализовано (см. далее гл. 38). Инкапсуляция позволяет изменять реализацию интерфесов объекта, не оказывая влияния на пользователей этого объекта.

**Перегрузка посредством сигнатур вызова (точнее её невозможность).**

В некоторых ООП ЯП под полиморфизмом также понимается возможность перегрузки ф-ций, основанной на сигнатурах типов их аргументов. Но т.к. в Питон нет объявлений типов, эта концепция неприменима - полиморфизм Питона основан на интерфейсах объектов, а не типах.

Можем попробовать перегрузку методов, изменяя списки их аргументов:

class C:

def meth(self, x):

...

def meth(self, x, y, z):

...

Это рабочий код, но т.к. инструкция def просто присваивает объект некоторому имени в области видимости класса, сохранено будет только последнее определение метода (это все равно, что записать подряд: X = 1, а затем Х = 2, в рез-те Х будет равно 2).

Выбор на основе типа можно реализовать с помощью идеи проверки типа (гл. 4 и 9), или с пом. возможности передачи списка аргументов (гл. 18):

class C:

def meth(self, \*args):

if len(args) == 1:

...

elif type(arg[0]) == int:

...

Однако это на крайний случай, т.к. все-таки следует писать код, который опирается на интерфейс объекта, а не на конкретный тип данных. Такой подход полезнее и универсальнее:

class C:

def meth(self, x):

x.operation() # Предполагается, что х работает правильно

Так же рекомендуется выбирать разные имена методов, выполняющих разные операции, и не полагаться на сигнатуры вызова (это относится ко всем ЯП).

Основное мастерство владения ООП в Питон заключается в умении комбинировать классы в программе для достижения поставленных целей.

**ООП и наследование: взаимосвязи типа "является".**

С точки зрения программиста, наследование вступает в игру с момента появления квалифицированного имени атрибута, при разрешение которого запускается поиск имен в экземплярах, в их классах и затем в суперклассах. С точки зрения проектироващика, наследование - это способ указать принадлежность к некоторому набору: класс определяет набор свойств, которые могут быть унаследованы и адаптированы более специализированными наборами (т.е. подклассами).

Вернемся к машины по деланию пиццы. Итак, мы решили сделать робота, но у нас он будет полноправным служащим с заработной платой, плюс у нас буду теще живые работники конечно (официанты, повара и т.д.).

Наш коллектив можно определить четырьмя классами из файла примера employes.py. Самый общий класс Employe, реализует поведение, общее для всех работников, такое как повышение з\п (giveRaise) и вывод на экран (\_\_repr\_\_).

Есть две категории служащих, т.е. два подкласса, наследующих класс Employe: Chef (повар) и Server (официант). Оба подкласса переопределяют унаследованный метод work, чтобы выводит более специализир. сообщения.

Наконец наш робот моделируется еще более специализир. классом PizzaRobot, наследующим класс Chef, который наследует Employe. В терминах ООП мы называем такие взаимоотношения "является": робот является поваром, а повар явл-ся служащим. Приводим содержимое файла employes.py:

class Employe:

def \_\_init\_\_(self, name, salary=0):

self.name = name

self.salary = salary

def giveRaise(self, percent):

self.salary = self.salary + (self.salary \* percent)

def work(self):

print(self.name, "does stuff")

def \_\_repr\_\_(self):

return "<Employe: name=%s, salary=%s>" % (self.name, self.salary)

class Chef(Employe):

def \_\_init\_\_(self, name):

Employe.\_\_init\_\_(self, name, 50000)

def work(self):

print(self.name, "makes food")

class Server(Employe):

def \_\_init\_\_(self, name):

Employe.\_\_init\_\_(self, name, 40000)

def work(self):

print(self.name, "interfaces with customer")

class PizzaRobot(Chef):

def \_\_init\_\_(self, name):

Chef.\_\_init\_\_(self, name)

def work(self):

print(self.name, "makes pizza")

if \_\_name\_\_ == "\_\_main\_\_":

bob = PizzaRobot('bob') # Создать робота с именем bob

print(bob) # Вызвать унаследованный метод \_\_repr\_\_

bob.work() # Выполнить действие, зависящее от типа

bob.giveRaise(0.20) # Увеличить роботу зарплату на 20%

print(bob); print()

for klass in Employe, Chef, Server, PizzaRobot:

obj = klass(klass.\_\_name\_\_)

obj.work()

Когда вып-ся код самопроверки, включенный в состав модуля, создается робот по приготовлению пиццы с именм bob, он наследует атрибуты классов: PizzaRobot, Chef и Employe:

C:\Python34>employes.py

<Employe: name=bob, salary=50000>

bob makes pizza

<Employe: name=bob, salary=60000.0>

Employe does stuff

Chef makes food

Server interfaces with customer

PizzaRobot makes pizza

**ООП и композиция: взаимосвязи типа "имеет".**

С точки зрения программиста, композиция - это прием встраивания других объектов в объект-контейнер и использование их для реализации методов контейнера. Для проектировщика композиция - это один из способов представить взаимоотношения в прикладной области. Но вместо того, чтобы определять принадлежность к множеству, при композиционном подходе все части объединяются в единое целое.

Композиция отражает взаимоотношения между частями, которые обычно называются отношениями "имеет". В некоторых книгах по ООП композиция называется агрегированием (различие в терминах в том, что "агрегирование" использ-ся для описания более слабой зависисмости между контейнером и его содержимым); в этой книге термин "композиция" использ-ся лишь для обозначения коллекции встраиваемых объектов. Вообще составные классы реализуют все свои интерфейсы, управляя работой встраиваемых объектов.

Теперь, когда у нас имеются реализации классов работников, объединим их в коллектив пиццерии и позволим присутпить к работе. Наша пиццерия - это составной объект: в нем есть печь, работники - официанты и повара. Когда поступает заказ все компоненты пиццерии начинают действовать - т.е. официант принимает заказ, повар готовит пиццу и т.д. Пример, файл pizzashop.py имитирует все объекты и отношения между ними:

from employes import PizzaRobot, Server

class Customer:

def \_\_init\_\_(self, name):

self.name = name

def order(self, server):

print(self.name, "orders from", server)

def pay(self, server):

print(self.name, "pays for item to", server)

class Oven:

def bake(self):

print("oven bakes")

class PizzaShop:

def \_\_init\_\_(self):

self.server = Server('Pat') # Встроить другие объекты

self.chef = PizzaRobot('Bob') # Робот по имени Bob

self.oven = Oven()

def order(self, name):

customer = Customer(name) # Активизировать другие объекты

customer.order(self.server) # Клиент делает заказ официанту

self.chef.work()

self.oven.bake()

customer.pay(self.server)

if \_\_name\_\_ == "\_\_main\_\_":

scene = PizzaShop() # Создать составной объект

scene.order('Homer') # Имитировать заказ клиента Homer

print('...')

scene.order('Shaggy') # Имитировать заказ клиента Shaggy

Класс PizzaShop - это контейнер и контроллер - это конструктор, который создает и встраивает экземпляры классов работников, написанные нами в предыдущем разделе, а также экземпляры класса Oven, который определен здесь.

Когда код самопроверки этого модуля вызывает метод order класса PizzaShop, встроенным объектам предлагается приступить к выполнению своих обязанностей. Обратите внимание, что для каждого клиента мы создаем новый экземпляр класса Customer и передаем встроенный объект Server (официант) методам класса Customer (клиент) - клиенты приходят и уходят, а официант остается частью коллектива. Кроме того, обратите внимание, работники по-прежнему вовлечены во взаимосвязи наследования - композиция и наследование - это взаимодополняющие инструменты.

Запускаем наш модуль и пиццерия обсулжит два заказа - от Гомера (Homer) и Шагии (Shaggy):

C:\Python34>pizzashop.py

Homer orders from <Employe: name=Pat, salary=40000>

Bob makes pizza

oven bakes

Homer pays for item to <Employe: name=Pat, salary=40000>

...

Shaggy orders from <Employe: name=Pat, salary=40000>

Bob makes pizza

oven bakes

Shaggy pays for item to <Employe: name=Pat, salary=40000>

Эта имитация наглядно демонстрирует объекты и взаимодействия между ними, а также составные объекты в действии. Классы могут представлять практически любые объекты и взаимоотношения между ними, которые можно выразить словами; для этого просто замените имена существительные - классами, глаголы - методами, и вы получите первый черновой набросок проекта.

**Еще раз об обработке потоков.**

Рассмотрим более реалистичный пример композиции. Вспомним универсальную ф-цию обработки потоков данных, гл. 25:

def processor(reader, converter, writer):

while 1:

data = reader.read()

if not data: break

data = converter(data)

writer.write(data)

Вместо простой ф-ции мы могли бы реализовать обработку в виде класса, который использует прием композиции, чтобы обеспечить наследование и более удобную конструкцию кода. Файл streams.py:

class Processor:

def \_\_init\_\_(self, reader, writer):

self.reader = reader

self.writer = writer

def process(self):

while 1:

data = self.reader.readline()

if not data: break

data = self.converter(data)

self.writer.write(data)

def converter(self, data):

assert False, 'converter must be defined'

Этот класс определяет метод converter, который, как ожидается ,будет переопределен в подклассах. Это пример использования абстрактных суперклассов, с которыми мы познакомились в гл. 28. При таком подходе объекты чтения (reader) и записи (writer) встраиваются в экземпляр класса (композиция), а логика преобразования поставляется в виде подкласса (наследование), а не в виде отдельной ф-ции. Ниже приводится содержимое файла converter.py:

from streams import Processor

class Uppercase(Processor):

def converter(self, data):

return data.upper()

if \_\_name\_\_ == '\_\_main\_\_':

import sys

obj = Uppercase(open('spam.txt'), sys.stdout)

obj.process()

Здесь класс Uppercase наследует логику цикла обработки потока данных (и всё остальное из суперкласса). В нем необходимо определить лишь то, что будет уникальным для него - логику преобразования данных. Если запустить это файл, он создаст и запустит экземпляр класса Uppercase, который прочитает содержимое файла spam.txt, преобразует все символы в верх. регистр и выведет их в поток stdout:

C:\Python34>type spam.txt

spam

Spam

SPAM!

C:\Python34>converter.py

SPAM

SPAM

SPAM!

Для обработки потоков различных видов достаточно передать конструктору класса объекты требуемых типов. Приведем пример реализации вывода в файл вместо стандартного потока вывода.

Замечание! Когда я пытался выполнить указанный ниже код, то ошибки не выводились, файл spamup.txt создавался исправно, НО был пустым!!! Пришлось добавить одну строку в классе Processor в метод process:

def process(self):

while 1:

data = self.reader.readline()

if not data: break

data = self.converter(data)

self.writer.write(data)

self.writer.close()

- self.writer.close() - данные записываются в файл после закрытия! Поэтому наш spamup.txt был пустым! Причем эту строку нельзя добавлять в цикл while, т.к. цикл закроется после первого чтения строки из атрибута reader!

- Нелегко это было понять и сделать, нужно читать, чтобы на лету понимать код...

- Прежний вывод в stdout по-прежнему работает, я проверил.

Итак, вывод в файл:

>>> import converters

>>> prog = converters.Uppercase(open('spam.txt'), open('spamup.txt', 'w'))

>>> prog.process()

C:\Python34>type spamup.txt

SPAM

SPAM

SPAM!

Мы могли бы также реализовать объекты, обернутые в классы, которые определяют необходимые интерфейсные методы ввода и вывода. Пример, где вывод осуществляется через класс, который обертывает выводимый текст в теги HTML:

>>> from converters import Uppercase

>>>

>>> class HTMLize:

def write(self, line):

print('<PRE>%s</PRE>' % line.rstrip())

>>> Uppercase(open('spam.txt'), HTMLize()).process()

<PRE>SPAM</PRE>

<PRE>SPAM</PRE>

<PRE>SPAM!</PRE>

...

AttributeError: 'HTMLize' object has no attribute 'close'

- последняя строка возникла из-за того, что я добавил закрытие файла вывода, а первые три строки вывода правильные, как в учебнике. Можно закомментировать строку с close(), тогда вывод будет без ошибок, но это не суть важно.

Заметим, что было получено два вида преобразований - приведение символов к верхнему регистру (наследованием) и преобразование в формат HTML (композицией), хотя основная логика обработки в оригинальном суперклассе Processor ничег оне знает ни об одном из них. Коду, выполняющему обработку, нужны только метод write - в классах, выполняющих запись, и метод convert. Его совершенно не интересует, что делают эти методы. Такой полиморфизм и инкапсуляция логики составляют основу такой мощи классов.

В этом примере, суперкласс Processor реализует только цикл сканирования файла. Его можно было бы расширить доп. инструментами и превратить в фреймворк. И потом использщовать фреймворк во многих своих программах.

Композиция и наследование - часто взаимодополняющие (а иногда и альтернативные) приемы.

**Классы и их хранение.**

Помимо имитации реальных взаимодействий, классы для пиццерии могли бы также использ-ся как основа БД пиццерии. Экземпляры классов можно сохранять с пом. модулей pickle и shelve. Модуль pickle ничуть не сложнее shelve:

import pickle

object = someClass()

file = open(filename, 'wb') # Создать внешний файл

pickle.dump(object, file) # Сохранить объект в файле

import pickle

file = open(filename, 'rb')

object = pickle.loac(file) # Позднее извлечь обратно

Модуль pickle преобразует объект в памяти в последовательность байтов (на самом деле - в строки), которые можно сохранять в файлы, передавать по сети и т.д. При извлечении идет обратное: из байтов преобразуется в объекты в памяти.

Модуль shelve реализует похожую возможность, но он автоматич. сохраняет объекты в БД с доступом по ключу, которая предоставляет интерфейс, похожий на интерфейс словаря:

import shelve

object = someClass()

dbase = shelve.open('filename')

dbase['key'] = object # Сохранить под ключом key

import shelve

dbase = shelve.open('filename')

object = dbase['key'] # Позднее извлечь обратно

Мы можем легко создать БД сотрудников, записывая экземпляры объектов в файл:

>>> from pizzashop import PizzaShop

>>> shop = PizzaShop()

>>> shop.server, shop.chef

(<Employe: name=Pat, salary=40000>, <Employe: name=Bob, salary=50000>)

>>>

>>> import pickle

>>> pickle.dump(shop, open('shopfile.dat', 'wb'))

Мы можем сохранить в файле весь составной объект, всю пиццерию. Восстановить его можно будет всего одной инструкцией:

>>> import pickle

>>> obj = pickle.load(open('shopfile.dat', 'rb'))

>>> obj.server, obj.chef

(<Employe: name=Pat, salary=40000>, <Employe: name=Bob, salary=50000>)

>>> obj.order('Sue')

Sue orders from <Employe: name=Pat, salary=40000>

Bob makes pizza

oven bakes

Sue pays for item to <Employe: name=Pat, salary=40000>

**ООП и делегирование: объекты-обертки.** Стр. 837

По делегированием обычно подразумевается наличие объекта-контроллера, куда встраиваются другие объекты, получающие запросы на выполнение операций. В Питон делегирование часто реализ-ся с пом. метода \_\_getattr\_\_, потмоу что он перехватывает попытки доступа к несуществующим атрибутам. Класс-обертка (иногда наз-ся прокси-классом) может использовать метод \_\_getattr\_\_ для перенаправления обращений к обернутому объекту. Класс-обертка повторяет интерфейс обернутого объекта и может добавлять доп. опции.

Пример, файл trace.py:

class wrapper:

def \_\_init\_\_(self, object):

self.wrapped = object # Сохранить объект

def \_\_getattr\_\_(self, attrname):

print('Trace:', attrname) # Отметить факт извлечения

return getattr(self.wrapped, attrname) # Делегировать извлечение

Здесь, для извлечение из обернутого объекта атрибута использ. ф-ция getattr - вызов getattr(X, N) аналогичен выражению X.N, за исключением того, что N - это выражение, которое во время выполнения представлено строкой, а не именем переменной.

Фактич. вызов getattr(X, N) по его действию можно сравнить с выражением X.\_\_dict\_\_[N], только в первом случае выполняется поиск по дереву наследования, как и в X.N, а во втором - нет.

Этот прием в нашем классе-обертке можно использовать для управления доступом к любому объекту с атрибутами - спискам, словарям и даже классам и экземплярам.

Прмиер, класс wrapper, который выводи тсообщение при каждом обращении к атрибуту и делегирует запрос обернутому объекту wrapped:

>>> from trace import wrapper

>>> x = wrapper([1, 2, 3]) # Обернуть список

>>> x.append(4) # Делегировать операцию методу списка

Trace: append

>>> x.wrapped # Вывести обернутый объект

[1, 2, 3, 4]

>>> x = wrapper({"a": 1, "b": 2}) # Обернуть словарь

>>> list(x.keys()) # Делегировать операцию методу словаря

Trace: keys

['a', 'b']

В итоге интерфейс обернутого объекта расширяется за счет методов класса-обертки. Этот прием может использ-ся для регистрации вызовов методов, перенаправления вызовов методов дополнительному или адаптированному программному коду и т.д.

***Примечание.***

В Питон 2.6 методы перегрузки операторов, которые выз-ся встр. операциями, выбираются с пом. методов, перехватывающих обращения к атрибутам, таких как \_\_getattr\_\_. Операция вывода обернутого объекта, напр., выызвает этот метод, чтобы выбрать \_\_repr\_\_ или \_\_str\_\_, и затем производит вызов соответствующего метода обернутого объекта.

В Питон 3.0 схема другая - операция вывода не обращается к методу \_\_getattr\_\_ и использует процедуру вывода по умолчанию. В 3.0 интерпретатор пытается отыскать метод перегрузки оператора в классах, пропуская этап поиска в экземпляре. Имейте в виду, вам может понадобиться переопределить методы перегрузки операторов в классах-обертках (вручную, с пом. инструментов или с пом. суперклассов), если хотите, чтобы оин действовали в Питон 3.0.

**Псевдочастные атрибуты класса.** стр. 839

Помимо структурирования кода, в классах часто приходится решать проблемы использования имен. Мы знаем, что все имена, которым присваивается знач. на верх. уровне модуля, становятся глобал. для этого модуля. То же относится и к классам - сокрытие данных регулируется соглашениями, и клиенты могут получать и изменять любые атрибуты класса или экземпляра по своему усмотрению.

Фактич. все атрибуты явл-ся "общедоступными" (public) и "виртуальными" (virtual), если говорить в терминах С++ - они доступны отовсюду и динамически отыскиваются во время выполнения.

Питон поддерживает и такое понятие как "искажение" (mangling) имен (т.е. расширение) с целью придать им черты локальных имен для класса. Искаженные имена иногда оишбочно называют "частными атрибутами", но на самом деле это всего лишь способ ограничить доступ к именам в классе - искажение имен не предотвращает доступ из программного кода, находящегося за пределами класса. Эта особенность предназначена в основном для избежания конфликта имен в экземплярах, а не для ограничения доступа к именам, - поэтому искаженные имена лучше называть "псевдочастными", чем "частными".

Псевдочастные имена - это дополнительная и совершенно не обязательная возможность, и вы, можете не считать её полезной, пока не столкнетесь с необходимостью создания инструментов общего назначения и многоуровневыъ иерархий классов в проектах, создаваемых командами программистов. Вместо псевдочастных имен, программисты часто дают внутренним атрибутам имена, начинающиеся с одного символа подчеркивания (напр. \_Х), - согласно неофициальным соглашениям, атрибуты с такими именами не должны изменяться за пределами класса (для интерпретатора такие имена не явл-ся специальными).

**Об искажении имен в общих чертах.**

О действии механизма искажения имен: имена внутри инструкции class, которые начинаются с двух символов подчеркивания, но не заканчиваются двумя символами подчеркивания, автоматич. расширяются за счет включения имени вмещающего класса.

Напр. имя \_\_Х, в классе с именем Spam автоматич. изменится на \_Spam\_\_X: к оригинальному имени будет добавлен префикс из символа подчеркивания и имени вмещающего класса, в рез-те буде тполучено уникальное имя.

Искажение имен происходит только внутри инструкций class и тольк одля имен, кодотрые начинаются двумя подчерками. Однако это происходит со всеми именами, которые начинаются с двух подчерков, включая имена методов и имена атрибутов экземпляров (напр. с классм Spam, ссылка на атрибут экземпляра self.\_\_X буде тпреобразована в self.\_Spam\_\_X).

**Для чего нужны псевдочастные атрибуты?**

Их задача состоит в том, чтобы обеспечить способ сохранности атрибутов экземпляра.

В Питон всякий раз, когда в пределах метода класса выполняется присваивание атрибуту аргумента self (напр. self.attr = value), создается или изменяется атрибут экземпляра (поиск в дереве идет только пр иполучении ссылки, а не при присваивании знач.). Это верно даже когда неск-ко классов в иерархии выполняют присваивание одному и тому же атрибуту, поэтому конфликты реальны.

Напр., когда программист писал класс, он думал, что экземпляры будут владеть атрибутом Х. В методах класса выполняется присваивание этмоу атрибуту и позже извлекается знач.:

class C1:

def meth1(self): self.X = 88 # Предполагается, что Х - это мой атрибут

def meth2(self): print(self.X)

Далее пусть др. программист исходил из того же предположения, когда писал свой класс:

class C2:

def metha(self): self.X = 99 # И мой тоже

def methb(self): print(self.X)

Каждый класс по отдельности работает нормально. Проблема возникает, когда оба класса оказываются в одном дереве наследования:

class C3(C1, C2): ...

I = C3() # У меня только один атрибут Х!

Теперь значение, которое получит каждый класс из выражения self.X, будет зависеть от того, кто из них последним присвоил значение. Все операции присваивания атрибуту self.X будут воздействовать на один и тот же экземпляр, у которого может быть тольк оодин атрибут X - I.X, - независимо от того, сколько классов используют это имя.

Чтобы гаранитрвоат ьпринадлженость атрибута тому классу, который его использует, достаточно в имени атрибута поставит ьдва символа подчеркивания везде, где оно используется классом, пример в файле private.py:

class C1:

def meth1(self): self.\_\_X = 88 # Теперь Х - мой атрибут

def meth2(self): print(self.\_\_X) # Превратится в \_С1\_\_Х

class C2:

def metha(self): self.\_\_X = 99 # И мой тоже

def methb(self): print(self.\_\_X) # Превратится в \_С2\_\_Х

class C3(C1, C2): pass

I = C3() # В I два имени Х

I.meth1(); I.metha()

print(I.\_\_dict\_\_)

I.meth2(); I.methb()

При этом видим, что имена атрибтов будут дополнены именами классов. Посмотрим это на практике:

>>> import private

{'\_C2\_\_X': 99, '\_C1\_\_X': 88}

88

99

Этот прием помогает избежать конфликта имен в экземплярах, но заметьте, что он не обеспечивает настоящего сокрытия данных.

Псевдочастные атрибуты удобно использовать в крупных проектах, т.к. там большое кол-во имен. Если метод использ-ся только внутри класса, и этот класс может наследовать и наследоваться др. классами, два подчеркивания гарантируют, что имя метода не будет конфликтовать с др. именами в дереве, особенно, когда использ-ся прием множественного наследования:

class Super:

def method(self): ... # Фактический прикладной метод

class Tool:

def \_\_method(self): ... # Получит имя \_Tool\_\_method

def other(self): self.\_\_method() # Используется внутренний метод

class Sub1(Tool, Super): ...

def actions(self): self.method() # Вызове тметод Super.method

class Sub2(Tool):

def \_\_init\_\_(self): self.method = 99 # Не уничтожит метод Tool.\_\_method

Напомним, что поиск в классе Sub1, если пойдет по суперклассам, буде тискать сначала в Tool, а потом уже в Super.

Не используйте эти приемы без нужды, не загромождайте код лишними символами - используйте только когда действительн онужно обеспечить принадлежност ьатрибута единственному классу.

**Методы - это объекты: связанные и несвязанные методы.**

Методы вообще и связанные в т.ч., упрощают решение многих задач в Питон. Связанные методы мы видели в гл. 29, когда изучали метод \_\_call\_\_. Связанные методы обладают большой гибкостью.

Мы знаем, что ф-ции могут обрабатываться как обычные объекты. Методы - это разновидность объектов, напомниающая ф-ции, - они могут присваиваться переменным, передаваться ф-циям, сохраняться в структурах данных и т.д. Доступ к методам класса осущ-ся через экземпляр класса или через сам класс и, фактич., в языке Питон имеется две разновидности методов:

- ***Несвязанные методы класса: без аргумента self.*** Попытка обращения к функциональному атрибуту класса через имя класса возвращает объект несвязанного метода. Чтобы вызвать этот метод, нужн оявно передать ему объект экземпляра в виде первого аргумента. В 3.0 несвязанные методы напоминают простые ф-ции и могут вызываться через имя класса. В 2.6 несвязанные методы - это иной тип данных, они не могут вызываться без передачи им ссылки на экземпляр.

- ***Связанные методы экземпляра: пара self + функция.*** Попытка обращения к функциональному атрибуту класса через имя экземпляра возвращает объект связанного метода. Интерпретатор автоматич. упаковывает экземпляр с ф-цией в объект связанного метода, поэтому нам не требуется передавать экземпляр в вызов такого метода.

Оба вида методов - полноценные объекты. Могут передаваться в программе как строки или числа даже. При запуске, оба требуют наличия экземпляра в первом аргументе (т.е. знач. для self). Поэтому в прошлой главе мы явно передавали экземпляр при вызове методов суперкласса из методов подкласса - с технич. точки зрения такие вызовы порождаюи объекты связанных методов.

Вызывая объект связ. метода, интер-р (интерпретатор) автоматич. подставляет экземпляр, который использ-ся при создании объекта связанного метода. Это значит, что объекты связ. методов обычно взаимозаменяемы с объектами простых ф-ций и создание их особенно полезно в случае интерфейсов, изначально ориентированных на использ-е ф-ций.

Пусть есть класс:

class Spam:

def doit(self, message):

print(message)

Обычно мы создаем экземпляр и вызываем его метод для вывода содержимого аргумента:

object1 = Spam()

object1.doit('hello world')

На самом деле попутно созд-ся объект связ. метода - как раз перед круглыми скобками в вызове метода. Т.е. мы можем получить связ. метод и без его вызова. Квалифиц. имя object.name - это выражение, которое возвращает объект.

В след. примере это выражение возвращает объект связ. метода, в котором упакованы вместе экземпляр (object1) и метод (Spam.doit). Можем присвоить этот связ. метод другому имени и использ. это имя для вызова ка кобычную ф-цию:

object1 = Spam()

x = object1.doit # Объект связ. метода: экземпляр + функция

x('hello world') # То же, что и object1.doit('...')

С др. стороны, если для получения метода doit использовать имя класса, мы получим объект несвязанного метода, который просто ссылается на объект ф-ции. Чтобы вызвать метод этого типа, нужно явно передавать экземпляр класса в первом аргументе:

object1 = Spam()

t = Spam.doit # Объект несвязанного метода

t(object1, 'howdy') # Передать экземпляр

Те же правила действуют внутри методов класса, когда использ. атрибуты аргумента self, которы ессылаются на ф-ции в классе. Выражение self.method возвращает объект связанного метода, потому что self - это объект экземпляра:

class Eggs:

def m1(self, n):

print(n)

def m2(self):

x = self.m1 # Еще один объект связанного метода

x(42) # Выглядит как обычная ф-ция

Eggs().m2() # Выведет 42

**В Питон 3.0 несвязанные методы являются функциями.**

В 3.0 были убрано понятие несвязанных методов. Несвяз. методы, которые мы описали, в 3.0 обрабатываются как обычные ф-ции. Чаще всего это никак не влияет на код - в любом случае при вызове метода относительно экземпляра в первом аргументе ему будет передан сам экземпляр.

Но для программ где выполняется явная проверка типа, это изменение может быть существенным, - если вывести тип метода, не получающего ссылку на экземпляр, в 2.6 будет выведено "unbound method" (несвяз. метод), а в 3.0 - "function" (функция).

В 3.0 метод может вызываться без передачи ему ссылки на экземпляр при условии, что сам метод не ожидает её получить и вызывается исключительно через обращение к имени класса. Т.е. в 3.0 ссылка на экземпляр передается методу, только когда он вызывается относительно экземпляра. При вызове метода через имя класса передавать ему экземпляр требуется, только если он ожидает получить его:

class Selfless:

def \_\_init\_\_(self, data):

self.data = data

def selfless(arg1, arg2): # Простая ф-ция в 3.0

return arg1 + arg2

def normal(self, arg1, arg2): # Ожидает получить экземпляр при вызове

return self.data + arg1 + arg2

>>> X = Selfless(2)

>>> X.normal(3, 4) # Экземпляр передается автоматич.

9

>>> Selfless.normal(X, 3, 4) # Метод ожидает получить self: передается вручную

9

>>> Selfless.selfless(3, 4) # Вызов экземпляра: работает в 3.0, но ошибка в 2.6!

7

Ошибка в 2.6 будет потому что по умолч. несвязанные методы требуют, чтобы им передавалась ссылка на экземпляр, а в 3.0 ошибки не будет, т.к. в этой версии такие методы интерпретируются, как простые ф-ции, не требующие передачи экземпляра. Конечно могут быть ошибки (вдруг забудем передать экземпляр), но с др. стороны это позволяет использ. методы как простые ф-ции, с условием, что им не передается, и они не ожидают получит ьаргумент self со ссылкой на экземпляр.

Два примера с ошибкой в обоих версиях, - в первом случае методу автоматич. будет передан экземпляр, которого он не ожидает, а во втором метод не получит ожидаемый экземпляр:

>>> X.selfless(3, 4)

TypeError: selfless() takes 2 positional arguments but 3 were given

>>> Selfless.normal(3, 4)

TypeError: normal() missing 1 required positional argument: 'arg2'

В 3.0 для методов, которые не принимают аргумент self, выз-ся только через имя класса и никогда не выз-ся относительн оэкземпляра больше не применяется декоратор staticmethod - такие методы действуют как обычные функции, не получая аргумент с экземпляром.

В 2.6 вызов таких методов будет выдавать ошибку, если экземпляр не передавать вручную.

С практич. точки зрения связанные методы важнее. Они представляют собой объекты, объединяющие в себе экземпляры и ф-ции, поэтому их можно интерпретировать, как обычные вызываемые объекты, т.е. как обычные функции.

**Связанные методы и другие вызываемые объекты.**

Напомним, связ. методы можно передавать внутри программы как объекты, или ф-ции. Также они могут использ-ся как любые вызываемые объекты, с обычным синтаксисом. Прмиер, сохранение четырех объектов связ. методов в списке и их вызов как обычных ф-ций:

>>> class Number:

def \_\_init\_\_(self, base):

self.base = base

def double(self):

return self.base \* 2

def triple(self):

return self.base \* 3

>>> x = Number(2) # Объекты экземпляров класса

>>> y = Number(3) # Атрибуты + методы

>>> z = Number(4)

>>> x.double() # Обычный непосредственный вызов

4

>>> acts = [x.double, y.double, y.triple, z.double] # Список связ-х методов

>>> for act in acts: # Вызовы откладываются

print(act()) # Вызов как функции

...

4

6

9

8

Замечание! А вот, что я получил когда попытался по-другому выполнить цикл for, нужно подумать:

>>> for act in acts:

print(act) # Вызов не как функции

...

<bound method Number.double of <\_\_main\_\_.Number object at 0x02A769D0>>

<bound method Number.double of <\_\_main\_\_.Number object at 0x02A76590>>

<bound method Number.triple of <\_\_main\_\_.Number object at 0x02A76590>>

<bound method Number.double of <\_\_main\_\_.Number object at 0x02A76B10>>

Внимание!!! Нашел причину почему так, потому что выше, когда мы создает список acts, мы вызываем методы класса Number БЕЗ круглых скобок, поэтому в моём эксперименте мы получаем такие странные выводы объектов, а не числа.

Как и простые ф-ции, объекты связ. методов обладают инфой, позволяющей провести интроспекцию, включая атрибуты, обеспечивающие доступ к объекту экземпляра и к методу. Вызов связ. метода просто задействует эту пару:

>>> bound = x.double

>>> bound.\_\_self\_\_, bound.\_\_func\_\_

(<\_\_main\_\_.Number object at 0x02A769D0>, <function Number.double at 0x02A7B078>)

>>> bound.\_\_self\_\_.base

2

>>> bound() # Вызовет bound.\_\_func\_\_(bound.\_\_self\_\_, ...)

4

Фактич. связ. методы - это лишь одна из разновидностей вызываемых объектов в Питон. Пример, простые ф-ции, определенные с пом. инструкции def или lambda, экземпляры, наследующие метод \_\_call\_\_, и связанные методы экземпляров могут обрабатываться и вызываться одинаковыми способами:

>>> def square(arg):

return arg \*\* 2 # Простые ф-ции (def или lambda)

>>> class Sum:

def \_\_init\_\_(self, val): # Вызываемые экземпляры

self.val = val

def \_\_call\_\_(self, arg):

return self.val + arg

>>> class Product:

def \_\_init\_\_(self, val): # Связанные методы

self.val = val

def method(self, arg):

return self.val \* arg

>>> sobj = Sum(2)

>>> pobj = Product(3)

>>> actions = [square, sobj, pobj.method] # Функцияб экземплярб метод

>>> for act in actions: # Все три вызываются одинаково

print(act(5)) # Вызов любого вызываемого

... # объекта с одним аргументом

25

7

15

>>> actions[-1](5) # Индексы, генераторы, отображения

15

>>> [act(5) for act in actions]

[25, 7, 15]

>>> list(map(lambda act: act(5), actions))

[25, 7, 15]

>>>

Технич. классы тоже пронадлежат к категории вызываемых объектов, но обычно они выз-ся для создания экземпляров, а не выполнения какой-то работы, как в примере ниже:

>>> class Negate:

def \_\_init\_\_(self, val): # Классы - тоже вызываемые объекты

self.val = -val # Но вызываются для создания объектов

def \_\_repr\_\_(self): # Реализует вывод экземпляра

return str(self.val)

>>> actions = [square, sobj, pobj.method, Negate] # Вызвать класс тоже

>>> for act in actions:

print(act(5))

25

7

15

-5

>>> [act(5) for act in actions] # Вызовет \_\_repr\_\_, а не \_\_str\_\_!

[25, 7, 15, -5]

>>> table = {act(5): act for act in actions} # Генератор словарей

>>> for (key, value) in table.items():

print('{0:2} => {1}'.format(key, value)) # Метод str.format

…

25 => <function square at 0x02A7B198>

15 => <bound method Product.method of <\_\_main\_\_.Product object at 0x02A76CD0>>

Traceback (most recent call last):

File "<pyshell#100>", line 2, in <module>

print('{0:2} => {1}'.format(key, value))

TypeError: non-empty format string passed to object.\_\_format\_\_

- У меня в IDLE возникает вот такая ошибка, пока не буду заострять внимание, нужно будет разобраться позднее, вернуться к генераторам словарей и почитать, может нужно что-то изменить или добавить. Автор книги (или редакторы) любит такие загадки иногда вставлять...

**Связанные методы и функции обратного вызова.**

Одно из мест где можн оувидет ьобъекты связ. методов, в которых автоматич. сохр-ся экземпляр класса вместе с ф-цией, - это программ. код, регистрирующий методы как обработчики событий в интерфейсе tkinter GUI. Простой пример:

def handler():

... сохраняет инфу о состоянии в глоабл. переменных ...

...

widget = Button(text='spam', command=handler)

Чтобы зарегистрировать обработку щелчка на кнопке, обычно передаем в аргументе с именем command вызываемый объект, который не имеет входных аргументов. Здесь часто использ-ся имена простых ф-ций (и lambda-выражений), но можно передать и методы класса - при условии, что они будут связ. методами:

class MyWidget:

def handler(self):

... сохраняет инфу о состоянии в self.attr ...

def makewidgets(self):

b = Button(text='spam', command=self.handler)

Здесь self.handler - это объект связ. метода, в котором сохр-ся self и MyWidget.handler. Т.к. self ссылается на оригинальный экземпляр, позднее, когда метод handler будет вызван для обработки события, он получит доступ к атрибутам экземпляра, где може тхраниться инфа о состоянии.

**Множественное наследование: примесные классы.** стр. 849

Многие шаблоны ООП основаны на объединенииразличных наборов методов. Когда в заголовке class в скобках перечислено более одного суперкласса мы знаем - в игру вступает механизм множественного наследования - класс и его экземпляры наследуют имена из всех перечисленных суперклассов.

При поиске в дереве классов, суперклассы, из заголовка, обходятся слева направо. Т.о. поиск может быть достаточно крупным в больших деревьях:

- В случае классических классов (до Питон 3.0) поиск идет сначала до вершины дерева, а потом уже слева направо.

- В новых классах (с Питон 3.0) поиск идет сначала по уровням дерева, а потом уже вверх и т.д.

Однако, независимо от модели, если у класса неск-ко суперклассов, то они просматриваются слева направо, в том порядке в каком они следуют в заголовке class.

Множ. наследование хорошо для моделирования объектов, принадлежащих более чем одной группе. Напр. человек может быть инженером, писателем, музыкантом и т.д.

Самый распространенный случай использ. множ. наследования - это "смешивание" методов общего назначения из неск-х суперклассов. Обычно такие суперклассы наз-ся примесными классами - они представляют методы, которые добавл-ся в прикладные классы наследованием.

**Создание примесных классов, реализующих вывод.**

Мы знаем, что вывод экземпляра класса, по умолч., не очень информативен:

>>> class Spam:

def \_\_init\_\_(self): # Нет метода \_\_repr\_\_ или \_\_str\_\_

self.data = "food"

>>> X = Spam()

>>> print(X)

<\_\_main\_\_.Spam object at 0x02A7E250>

Конечно для вывода можно использ. \_\_repr\_\_ или \_\_str\_\_. Но зачем писать эти методы в каждом классе где есть вывод? Удобнее было бы написать вывод всего один раз в классе инструментов и наследовать его во всех нужных классах.

Для этого и использ. примесные классы. Т.е. определив метод вывода один раз в суперклассе, можно использ. его везде где пожелаем. Мы уже видели похожие инструменты:

- Класс AttrDisplay в гл. 27 - вывод атрибутов экземпляра с пом. \_\_str\_\_, но он не предусматривал подъем по дереву клссов.

- Модуль classtree.py в гл. 28 - содержит ф-ции, позволяющие обход деревьев классов и выводит ьих схематическое представление, но они не выводят атрибуты объектов и не объединены в класс, который можно наследовать.

Мы напишем три примесных класса, которые смогут отображать списки атрибутов экземпляра, унаследованных атрибутов и атрибутов всех объектов в дереве классов. Попробуем применить их в режиме множ. наследования и используем приемы программирования, чтобы сделать наши классы более универсальными.

**Получение списка атрибутов экземпляра с помощью \_\_dict\_\_.**

Пример, примесный класс ListInstance, в файле lister.py, реализующий метод \_\_str\_\_ для всех классов, наследующих его. Также его логика может использ-ся экземплярами его подклассов:

# lister.py

class ListInstance:

"""

Примесный класс, реализация форматирования строки при вызове print() и str() с экземпляром в виде аргумента, через наследование метода \_\_str\_\_; отображает только атрибуты экземпляра; self - экземпляр самого нижнего класса; во избежании конфликта имен использ. имена вида \_\_Х

"""

def \_\_str\_\_(self):

return '<Instance of %s, address %s:\n%s>' % (

self.\_\_class\_\_.\_\_name\_\_, # Имя клиентского класса

id(self), # Адрес экземпляра

self.\_\_attrnames()) # Список пар name=value

def \_\_attrnames(self):

result = ''

for attr in sorted(self.\_\_dict\_\_): # Словарь атрибутов

result += '\tname %s=%s\n' % (attr, self.\_\_dict\_\_[attr])

return result

Для получения имени класса экземпляра и списка атрибутов в классе ListInstance использ-ся известные приемы:

- Каждый экземпляр имеет встр. атрибут \_\_class\_\_ (из которого создан), а каждый класс имеет атрибут \_\_name\_\_, ссылка на имя класса. Т.о. self.\_\_class\_\_.\_\_name\_\_ - извлекает имя класса.

- Основная работа класса - просмотреть атрибуты экземпляра (атрибут \_\_dict\_\_) и сконструировать строку, с именами и знач. всех атриубтов экземпляра (с сортировкой).

Т.о. класс ListInstance напоминает класс AttrDisplay, но использ. два новых приема:

- Отображает адрес экземпляра в памяти, вызывая встр. ф-цию id, которая возвращает адрес объекта.

- Использует псевдочастные имена для своих методов: \_\_attrnames. Правило, с дополнением именем вмещающего класса, распространяется на атрибуты класса и атрибуты экземпляра, присоединяемы ек объекту self.

Класс ListInstance определяет метод \_\_str\_\_ перегрузки операторов, поэтому при выводе экземпляров, наследующих этот класс, они автоматич. будут выводить свои атрибуты, давая больше инфы чем просто свой адрес.

Используем класс:

>>> from lister import ListInstance

>>> class Spam(ListInstance): # Наследует метод \_\_str\_\_

def \_\_init\_\_(self):

self.data1 = 'food'

>>> x = Spam()

>>> print(x) # print() и str() вызывают \_\_str\_\_

<Instance of Spam, address 31193872:

name data1=food

> # У меня в IDLE был перенос >

# оказывается в книге он тоже есть, не знаю почему

Можем получить рез-ты в виде строки, без вывода на экран, с пом ф-ции str. А ф-ция автоматич. вывода по-прежнему будет использ. формат по умолч.:

>>> str(x)

'<Instance of Spam, address 31193872:\n\tname data1=food\n>'

>>> x

<\_\_main\_\_.Spam object at 0x01DBFB10>

Класс ListInstance полезен в любых классах, - даже уже имеющих один или более суперклассов. Здесь в игру вступает механизм множественного наследования: добавляя ListInstance в список суперклассов в заголовке class, вы получаете реализацию \_\_str\_\_ "в подарок", что не мешает наследовать существующие суперклассы:

# testmix.py

from lister import \* # Импортировать инструментальные классы

class Super:

def \_\_init\_\_(self): # Метод \_\_init\_\_ суперкласса

self.data1 = 'spam' # Создать атрибуты экземпляра

def ham(self):

pass

class Sub(Super, ListInstance): # Подмешать методы ham и \_\_str\_\_

def \_\_init\_\_(self): # Инструментальные классы имеют доступ к self

Super.\_\_init\_\_(self)

self.data2 = 'eggs' # Добавить атрибуты экземпляра

self.data3 = 42

def spam(self): # Определить еще один метод

pass

if \_\_name\_\_ == "\_\_main\_\_":

X = Sub()

print(X) # Вызовет подмешанный метод \_\_str\_\_

Здесь класс Sub наследует имена из двух классов, Super и ListInstance, и ест ьеще собственные имена. Если создать и вывести экземпляр класса Sub, автоматич. будет получено адаптированное его представление:

C:\Python34>testmix.py

<Instance of Sub, address 24116848:

name data1=spam

name data2=eggs

name data3=42

>

Реализация ListInstance может работать с любыми классами, потому что аргумент self ссылается на экземпляр подкласса, который наследует ListInstance. В некоем смысле примесные классы - это классы, эквивалентные модулям. Пример, работа ListInstance в режиме простого наследования с экземплярами разных классов, к которым присоед-ся атрибуты за пределами класса:

>>> import lister

>>> class C(lister.ListInstance): pass

>>> x = C()

>>> x.a = 1; x.b = 2; x.c = 3

>>> print(x)

<Instance of C, address 44557360:

name a=1

name b=2

name c=3

>

Примесные классы также позволяют оптимизировать код как и любые другие классы. Напр. мы захотим добавить вывод все атрибуты класса, которые были унаследованы экземпляром - изменения автоматич. будет действовать во всех подклассах.

**Получение списка атрибутов экземпляра с помощью функции dir.**

Улучшим ListInstance так, чтобы он отображал все атрибуты экземпляра, - и собственные и унаследованные. Вместо сканирования словаря \_\_dict\_\_ будем использовать ф-цию dir - словарь хранит только атрибуты экземпляра, а dir возвращает все унаследованные атрибуты.

Приводим пример новой реализации, класс переименуем для удобства понимания:

# lister.py, продолжение

class ListInhert:

"""

Используем ф-цию dir(), в 3.0 выводится больше имен атрибутов, т.к. есть суперкласс object; метод getattr() позволяет получить знач. унаследованных атрибутов, отсутствующих в self.\_\_dict\_\_; реализует \_\_str\_\_, а не \_\_repr\_\_, дабы не получить бесконечный цикл при выводе связ. методов!

"""

def \_\_str\_\_(self):

return '<Instance of %s, address %s:\n%s>' % (

self.\_\_class\_\_.\_\_name\_\_, # Имя класса экземпляра

id(self), # Адрес экземпляра

self.\_\_attrnames()) # Список пар name=value

def \_\_attrnames(self):

result = ''

for attr in dir(self): # Передать экземпляр ф-ции dir()

if attr[:2] == '\_\_' and attr[-2:] == '\_\_': # Пропустить

result += '\tname %s=<>\n' % attr # внутренние имена

else:

result += '\tname %s=%s\n' % (attr, getattr(self, attr))

return result

Новая реализация пропускает имена вида \_\_X\_\_, т.к. они чаще всего для внутр. использ. и нам не нужны. Также использ. ф-цию getattr - она поддерживает поиск имен в дереве наследования.

Чтобы проверить изменим файл testmix.py:

class Sub(Super, ListInhert): # Подмешать \_\_str\_\_

Обратите внимание на искажение имени метода \_\_attrnames в классе ListInhert:

C:\Python34>testmix.py

<Instance of Sub, address 25165424:

name \_ListInhert\_\_attrnames=<bound method Sub.\_\_attrnames of <\_\_main\_\_.S

ub object at 0x017FFE70>>

name \_\_class\_\_=<>

name \_\_delattr\_\_=<>

name \_\_dict\_\_=<>

name \_\_dir\_\_=<>

name \_\_doc\_\_=<>

name \_\_eq\_\_=<>

name \_\_format\_\_=<>

name \_\_ge\_\_=<>

name \_\_getattribute\_\_=<>

name \_\_gt\_\_=<>

name \_\_hash\_\_=<>

name \_\_init\_\_=<>

name \_\_le\_\_=<>

name \_\_lt\_\_=<>

name \_\_module\_\_=<>

name \_\_ne\_\_=<>

name \_\_new\_\_=<>

name \_\_reduce\_\_=<>

name \_\_reduce\_ex\_\_=<>

name \_\_repr\_\_=<>

name \_\_setattr\_\_=<>

name \_\_sizeof\_\_=<>

name \_\_str\_\_=<>

name \_\_subclasshook\_\_=<>

name \_\_weakref\_\_=<>

name data1=spam

name data2=eggs

name data3=42

name ham=<bound method Sub.ham of <\_\_main\_\_.Sub object at 0x017FFE70>>

name spam=<bound method Sub.spam of <\_\_main\_\_.Sub object at 0x017FFE70>>

>

Заметим, перегрузка вывода с \_\_repr\_\_ будет попадать в бесконечный цикл - при попытке отобразить значение метода вызывается метод \_\_repr\_\_ класса, которому принадлжеит отображаемый метод, чтобы вывести инфу о классе. Т.е. если \_\_repr\_\_ класса ListInhert попытается отобразить метод, то при выводе инфы о классе, которому принадлжеит отображаемый метод, снова буде твызван \_\_repr\_\_ класса ListInhert. Эту проблему сложно заметить, но она есть!

**Получение списка атрибутов с привязкой к объектам в дереве классов.**

Сейчас класс ListInhert не сообщает нам из каких классов были унаследованы те или иные имена. Наш след. примесный класс выводит схему дерева классов, отображая атрибуты, присоединенные к каждому объекту. Для этого идет обход дерева от атрибута \_\_class\_\_ экземпляра к его класу и затем от тарибута \_\_bases\_\_ класса ко всем суперклассам, сканируя словари \_\_dict\_\_:

# lister.py, продолжение

class ListTree:

"""

Прмес. класс, в нем метод \_\_str\_\_ смотрит всё дерево классов и создает список атрибутов всех объектов, нах-ся выше self; вызывается ф-циями print(), str() и возвр. строку со списком; для избегания конфликта имен использ. имена вида \_\_Х; для рекурсив. обхода суперклассов использ. выражение-генератор; также использ. метод str.format()

"""

def \_\_str\_\_(self):

self.\_\_visited = {}

return '<Instance of {0}, address {1}:\n{2}{3}>'.format(

self.\_\_class\_\_.\_\_name\_\_,

id(self),

self.\_\_attrnames(self, 0),

self.\_\_listclass(self.\_\_class\_\_, 4))

def \_\_listclass(self, aClass, indent):

dots = '.' \* indent

if aClass in self.\_\_visited:

return '\n{0}<Class {1}:, address {2}: (see above)>\n'.format(

dots,

aClass.\_\_name\_\_,

id(aClass))

else:

self.\_\_visited[aClass] = True

genabove = (self.\_\_listclass(c, indent+4)

for c in aClass.\_\_bases\_\_)

return '\n{0}<Class {1}, address {2}:\n{3}{4}{5}>\n'.format(

dots,

aClass.\_\_name\_\_,

id(aClass),

self.\_\_attrnames(aClass, indent),

''.join(genabove),

dots)

def \_\_attrnames(self, obj, indent):

spaces = ' ' \* (indent + 4)

result = ''

for attr in sorted(obj.\_\_dict\_\_):

if attr.startswith('\_\_')and attr.endswith('\_\_'):

result += spaces + '{0}=<>\n'.format(attr)

else:

result += spaces + '{0}={1}\n'.format(attr, getattr(obj, attr))

return result

Обратим внимание, как использ. выражение-генератор для реализации рекурсивного обхода суперклассов - оно активируется вложенным строковым методом join. Также вместо оператора форматирвоания % мы используем метод format, т.к. явная нумерация аргументов может упростить чтение кода. Т.е. мы заменили первую строку на вторую:

return '<Instance of %s, address %s:\n%s%s' % (...) # Выражение

return '<Instance of {0}, address {1}:\n{2}{3}'.format(...) # Метод

Теперь изменим файл testmix.py:

class Sub(Super, ListTree):

И смотрим вывод сценария:

C:\Python34>testmix.py

<Instance of Sub, address 31129200:

\_ListTree\_\_visited={}

data1=spam

data2=eggs

data3=42

....<Class Sub, address 31297584:

\_\_doc\_\_=<>

\_\_init\_\_=<>

\_\_module\_\_=<>

spam=<function Sub.spam at 0x01DB26F0>

........<Class Super, address 31264304:

\_\_dict\_\_=<>

\_\_doc\_\_=<>

\_\_init\_\_=<>

\_\_module\_\_=<>

\_\_weakref\_\_=<>

ham=<function Super.ham at 0x01DB2660>

............<Class object, address 1702851304:

\_\_class\_\_=<>

\_\_delattr\_\_=<>

\_\_dir\_\_=<>

\_\_doc\_\_=<>

\_\_eq\_\_=<>

\_\_format\_\_=<>

\_\_ge\_\_=<>

\_\_getattribute\_\_=<>

\_\_gt\_\_=<>

\_\_hash\_\_=<>

\_\_init\_\_=<>

\_\_le\_\_=<>

\_\_lt\_\_=<>

\_\_ne\_\_=<>

\_\_new\_\_=<>

\_\_reduce\_\_=<>

\_\_reduce\_ex\_\_=<>

\_\_repr\_\_=<>

\_\_setattr\_\_=<>

\_\_sizeof\_\_=<>

\_\_str\_\_=<>

\_\_subclasshook\_\_=<>

............>

........>

........<Class ListTree, address 31263856:

\_ListTree\_\_attrnames=<function ListTree.\_\_attrnames at 0x01DB27C8>

\_ListTree\_\_listclass=<function ListTree.\_\_listclass at 0x01DB2780>

\_\_dict\_\_=<>

\_\_doc\_\_=<>

\_\_module\_\_=<>

\_\_str\_\_=<>

\_\_weakref\_\_=<>

............<Class object:, address 1702851304: (see above)>

........>

....>

>

В 2.6 методы определяются как несвяз. (unbound). Посмотрите также на искажение имени таблицы \_\_visited, получилось красивое длинное имя, которое плохо совпадает с др. именами (юмор).

В 3.0 несвяз. методы интерпретируются как простые функции, ну и конечно же список атрибутов и суперклассов побольше чем в 2.6 (помним про суперкласс всех классов - object).

В 3.0 также исключается возможность многократного перечисления одного и того же объекта класса, за счет использ. таблицы посещенных классов.

Для интереса можно смешать этот класс с каким-нибудь другим, более существенным классом, напр. с классом Button из модуля tkinter. Класс ListTree желательно указать первым в списке суперклассов в заголовке class, чтобы его метод \_\_str\_\_ имел преимущество, - т.к. класс Button имеет свой метод \_\_str\_\_:

>>> from lister import ListTree

>>> from tkinter import Button

>>> class MyButton(ListTree, Button): pass

>>> B = MyButton(text='spam')

>>> open('savetree.txt', 'w').write(str(B))

19850

Я не стал печатать print(B) в IDLE, а просто посмотрел файл savetree.txt в редакторе, строк там действительно много, - аж 324! Начинаются они вот так:

<Instance of MyButton, address 38041232:

\_ListTree\_\_visited={}

\_name=38041232

\_tclCommands=[]

... очень много строк опущено ...

Можно было бы выводить дерево в графическом интерфейсе, но это уже на самостоятельное изучение.

Примесные классы явл-ся мощным инструментом повторного использ. кода, что тесно связано с ООП. Множ. наследование может быть благом при грамотном применении, но и может осложнить жизнь при неаккуратном и чрезмерном употреблении.

***Примечание*. Поддержка слотов.**

Т.к. классы ListInstance и ListTree, выполняют сканирование словарей экземпляров, они не поддерживают атрибуты, хранящиеся в слотах - новой и редко используемой особенности, которую изучим далее, где увидим как атрибуты экземпляров объявл-ся в атрибуте \_\_slots\_\_ класса.

Напр., если в Super, добавить инструкцию \_\_slots\_\_=['data1'], а в Sub инструкцию \_\_slots\_\_=['data3'], классы ListInstance и ListTree обнаружат в экземпляре тольк оатрибут data2 - класс ListTree выведет атрибуты data1 и data3, но только как атрибуты объектов классов Super и Sub, и со знач. в спец. формате.

Чтобы сделать поддержку слотов, нужно изменить цикл, реализующий сканирование словарей \_\_dict\_\_, так чтобы он дополнительно выполнял итерации через списки \_\_slots\_\_, а для извлечения знач. вместо индексирования словаря \_\_dict\_\_ используйте ф-цию getattr (это уже сделано в ListTree).

Т.к. экземпляры наследуют атрибут \_\_slots\_\_ только от ближайшего класса, нам може тпотребоваться прибегнуть к хитрости для обработки атрибутов \_\_slots\_\_, присутствующих в некоторых суперклассах (ListTree уже отображает их как атрибуты классов). Класс ListInhert лишен этих недостатков, потому что ф-ция dir уже объединяет имена из словаря \_\_dict\_\_ и имена из атрибутов \_\_slots\_\_ всех классов. Знайте, что слоты и обычные атрибуты экземпляра - это разные типы имен.

**Классы - это объекты: универсальные фабрики объектов.** стр. 861

Иногда нужно, чтобы объекты создавались в ответ на какие-то условия, которые невозможно предсказать на этапе разработки программы. Фабричный шаблон проектирования позволяет реализовать такой подход.

Классы - это объекты, поэтому их легко передавать между компонентами проги, сохранять в структурах данных и т.д. Классы можно передавать ф-циям, которые создают объекты произвольных типов - в кругах ООП такие ф-ции наз-ют фабриками.

Синтаксис из гл. 18 может вызывать любые классы (вернее даже любой вызываемый объект, в т.ч. ф-ции, классы, методы) с любым числом аргументов конструкторов за один присест:

def factory(aClass, \*args): # Кортеж с переменным числом аргументов

return aClass(\*args) # Вызов aClass (или apply, только в 2.6)

class Spam:

def doit(self, message):

print(message)

class Person:

def \_\_init\_\_(self, name, job):

self.name = name

self.job = job

object1 = factory(Spam) # Создать объект Spam

object2 = factory(Person, "Guido", "guru") # Создать объект Person

Как видим здесь у нас определна ф-ция-генератор объектов factory. Она ждет получить объект класса (любого) вместе с одним или более аргументами конструктора класса. Ф-ция использ. спец. синтаксис вызова с переменным числом аргументов, чтобы создать и вернуть экземпляр.

Это единственная фабричная ф-ция, которую нам придется написать на Питон - т.к. она работает с любыми классами и с любыми аргументами конструктора.

Можно даже улучшить её, чтобы была поддержка именованных аргументов конструктора; ф-ция может собрать их в аргумент \*\*args и передать в класс третьим аргументом:

def factory(aClass, \*args, \*\*kwargs):

return aClass(\*args, \*\*kwargs)

**Зачем нужны фабрики?**

Итак, чем же хороша ф-ция factory? На практике мы не можем продемонстрировать, т.к. для книги это слишком большой пример. Тем не менее такая фабрика могла бы помочь изолировать программный код от динамически настраиваемой конструкции объекта.

Фспомним ф-цию processor из гл. 25, и пример композиции в этой главе. В обоих случаях принимаются объекты, выполняющие чтение и запись обрабатываемого потока данных. В оригинальном примере мы вручную передавали экземпляры спец. классов, такие как FileWrite и SocketReader, позднее мы передавали жестко заданные объекты файла, потока и преобразования. Внеш. источники могу топределяться и настройками и в элементах управления графич. интерфейса.

В нашем динамическом мире не представляется возможным жестко задавать в сценарии объекты, реализующие интерфейс к потоку данных, но можно задавать в процессе выполнения.

Напр. в файле с настройками може топред-ся имя класса потока, который должен быть импортирован из модуля, и доп. аргументы конструктора. Тут нам и может помочь фабричная ф-ция, или похожий кусок кода. Напр. представим, что требуемые классы даже не существовали в тот момент, когда мы писали свой программный код:

classname = ... определяется конфигурацинным файлом ...

classarg = ... определяется конфигурацинным файлом ...

import streamtypes # Специализированный программный код

aclass = getattr(streamtypes, classname) # Извлечь из модуля

reader = factory(aclass, classarg) # Получить экземпляр aclass(classarg)

processor(reader, ...)

Здесь встр. ф-ция getattr снова использ-ся для извлечения атрибута модуля, имя которого задано в виде строки (это тоже, что и записать выражение obj.attr, где attr - это строка). Т.к. этот фрагмент предполагает наличие у конструктора единственного аргумента, то, строго говоря, здесь не требуется ни ф-ция factory, ни ф-ция apply - мы могли бы просто создать экземпляр класса обращением aclass(classarg). Эти ф-ции более полезны в случаях, когда кол-во аргументов заранее не известно, и соотв. повышают гибкость реализации.

**Прочие темы, связанные с проектированием.**

В этой главе мы ближе познакомились с наследованием, композицией, делегированием, множественным наследованием, связанным иметодами и фабриками - типичными шаблонами проектирования, которые сипольз-ся в комбинации с классами при создании программ на Питон. В книге обсуждаются и другие темы, связаннеы с проектированием:

- Абстрактные суперклассы (гл. 28);

- Декораторы (гл. 31 и 38);

- Подклассы встроенных типов (гл. 31);

- Статические методы и методы классов (гл. 31);

- Управляемые атрибуты (гл. 37);

- Метаклассы (гл. 31 и 39).

**Глава 31. Дополнительные возможности классов.** стр. 865

**Расширение встроенных типов.** стр. 866

Помимо реализации объектов новых типов, классы иногда расширяют возможности встр. типов Питона. Напр. добавить в списки доп. методы вставки и удаления, или можно создать класс, который обертывает (встраивает) объект списка и экспортирует методы вставки и удаления, которые особым образом обрабатывают список, напр. как наш прием делегирования в гл. 30.

**Расширение типов встраиванием.**

Пример на основе ф-ций для работы с множествами из гл. 16 и 18. Файл setwrapper.py реализуте новый тип объектов за счет перемещения нескольких ф-ций в методы и добавления перегрузки нескольких основных операторов. Др. словами этот класс просто обертывает список, добавляя доп. операции:

class Set:

def \_\_init\_\_(self, value=[]): # Конструктор

self.data = [] # Управляет списком

self.concat(value)

def interset(self, other): # other - любая последовательность

res = [] # self - подразумеваемый объект

for x in self.data:

if x in other: # Выбрать общие элементы

res.append(x)

return Set(res) # Вернуть новый экземпляр Set

def union(self, other): # other - любая последовательность

res = self.data[:] # Копировать список

for x in other: # Добавить элементы из other

if not x in res:

res.append(x)

return Set(res)

def concat(self, value): # Аргумент value: список, Set...

for x in value: # Удалить дубликаты

if not x in self.data:

self.data.append(x)

def \_\_len\_\_(self): return len(self.data) # len(self)

def \_\_getitem\_\_(self, key): return self.data[key] # self[i]

def \_\_and\_\_(self, other): return self.intersect(other) # self & other

def \_\_or\_\_(self, other): return self.union(other) # self | other

def \_\_repr\_\_(self): return 'Set:' + repr(self.data) # Вывод

Выполним несколько операций с нашим классом:

>>> from setwrapper import Set

>>> x = Set([1, 3, 5, 7])

>>> print(x.union(Set([1, 4, 7])))

Set:[1, 3, 5, 7, 4]

>>> print(x | Set([1, 4, 6]))

Set:[1, 3, 5, 7, 4, 6]

Перегрузка операции доступа по индексам позволяет экземплярам класса Set выглядеть как настоящие списки.

**Расширение типов наследованием.**

Ф-ции преобразования типов, такие как list, str, dict, tuple превратились в имена встр. типов в Питон позднее 2.2, и теперь вызов ф-ции преобразования типа в действительности явл-ся вызовом конструктора типа.

Это дает адаптировать или расширять поведение встр. типов с пом class: нужн опросто создать подклассы с новыми именами типов, где реализовать изменения.

Напр, мы хотим, чтобы списки начинали отчет элементов не с 0, а с 1. Для этого, создаем подкласс, который изменит эту особенность списков, пример - файл typesubclass.py:

# Подкласс встр типа\класса list.

# Отображает дипапазон 1..N на 0..N-1

class MyList(list):

def \_\_getitem\_\_(self, offset):

print('(indexing %s at %s)' % (self, offset))

return list.\_\_getitem\_\_(self, offset-1)

if \_\_name\_\_ == "\_\_main\_\_":

print(list('abc'))

x = MyList('abc') # \_\_init\_\_ наследуется из списка

print(x) # \_\_repr\_\_ наследуется из списка

print(x[1]) # MyList.\_\_getitem\_\_

print(x[3]) # Изменяет поведение метода суперкласса

x.append('spam'); print(x) # Атрибуты, унаследованные от суперкласса list

x.reverse(); print(x)

Здесь у нас подкласс MyList расширяет метод \_\_getitem\_\_ простым отображением диапазона значений от 1 до N на необходимый спискам диапазон от 1 до N-1.

C:\Python34>typesubclass.py

['a', 'b', 'c']

['a', 'b', 'c']

(indexing ['a', 'b', 'c'] at 1)

a

(indexing ['a', 'b', 'c'] at 3)

c

['a', 'b', 'c', 'spam']

['spam', 'c', 'b', 'a']

Полезность такого изменения индексация не очень понятна, но для примера подходит.

Такой шаблон дает нам алтернативу реализации множества - в виде подкласса встр. списка, а не самостоятельного класса.

Пример, файл setsubclass.py, адаптирует списки, добавляя методы и операторы, используемые для работы с множествами. Всё остальное поведение наслед-ся от встр. суперкласса list:

class Set(list):

def \_\_init\_\_(self, value=[]):

list.\_\_init\_\_([])

self.concat(value)

def intersect(self, other):

res = []

for x in self:

if x in other:

res.append(x)

return Set(res)

def union(self, other):

res = Set(self)

res.concat(other)

return res

def concat(self, value):

for x in value:

if not x in self:

self.append(x)

def \_\_and\_\_(self, other): return self.intersect(other)

def \_\_or\_\_(self, other): return self.union(other)

def \_\_repr\_\_(self): return 'Set:' + list.\_\_repr\_\_(self)

if \_\_name\_\_ == "\_\_main\_\_":

x = Set([1, 3, 5, 7])

y = Set([2, 1, 4, 5, 6])

print(x, y, len(x))

print(x.intersect(y), y.union(x))

print(x & y, x | y)

x.reverse()$ print(x)

C:\Python34>setsubclass.py

Set:[1, 3, 5, 7] Set:[2, 1, 4, 5, 6] 4

Set:[1, 5] Set:[2, 1, 4, 5, 6, 3, 7]

Set:[1, 5] Set:[1, 3, 5, 7, 2, 4, 6]

Set:[7, 5, 3, 1]

Есть более эффективный способ реализации множеств - с пом. словарей, который позволяют заменить последовательное сканирование, используемое в данной реализации, на операцию обращения по ключу (хеширование) и т.о. повысить скорость работы.

Еще пример наследования - тип bool, т.к. bool - это подкласс типа int с двумя экземплярами (True и False), которые ведут себя как целые числа 1 и 0, но наследуют измененные версии методов вывода, особым образом отображающие их имена.

**Классы "нового стиля".** стр. 869

С версии 2.2 в Питон появились классы "нового стиля". Классы, следующие оригинальной модели называют "классическими классами". В 3.0 остались только "новые классы", но в версиях 2.Х классы делятся на две категории:

- В 3.0 все классы автоматич. относятся к "новому стилю", и это не зависит наследуют они явно класс object или нет. Т.е. все классы явно или неявно наследуют object, и все объекты явл-ся экземплярами класса object.

- В 2.6 и ранних версиях классы должны явно наследовать класс object (или др. встр. тип), чтобы считаться классами "нового стиля" и получать их особенности конечно.

В этом разделе мы будем описывать особенности классов "нового стиля" по отношению к Питон 2.Х, а для пользователей 3.0 мы должны считать эти особенности просто описанием существующих особенностей класса, т.к. для нас "новый стиль" - это наши обычные классы (не забываем, у нас Питон 3.4).

Отличие новых классов от классических, в версиях 2.6 и ранее в том, что новые классы наследуют либо встр. тип либо класс object:

class Newstyle(object):

... обычный программный код ...

Т.о. любые классы, наследующие object или встр. тип, автоматич. интерпретируются ка кклассы нового стиля. Если где-то в дереве наследования есть встр. тип, этот класс будет считаться классом нового стиля. Классы, не наследующие встр. класс, такой как object, считаются классическими.

Классы нового стиля незначительно отличаются от классических, более того, классич. модель в 2.6 используемая сейчас, по-прежнему работает по всем описанным принципам.

Классы нового стиля явл-ся обратно совместимыми с классическими. Но т.к. они некоторым образом изменили поведение классов, их нужно представлять как отдельный инструмент. Напр. малозаметные отличия, такие как ромобоидальная схема поиска в дереве и особенности выполнения встр. операций над атрибутами, доступ к которым контролируется методами, такими как \_\_getattr\_\_, могут привести к нарушению работы устаревшего кода, если не внести соотв. изменения.

**Изменения в классах нового стиля.**

Наиболее заметные отличия:

- *Классы и типы были объединены.* Классы теперь явл-ся типами, а типы - классами. Фактич. эти два термина стали синонимами. Вызов встр. ф-ции type(I) теперь возвращает класс, из которого был получен экземпляр; обычно тот, что указан в атрибуте I.\_\_class\_\_, а не обобщенный тип "instance". Кроме того, сами классы явл-ся экземплярами класса type, который может наследовать в подклассах для изменения процедуры создания классов, и все классы (а следовательно и типы) наследуют класс object.

- *Порядок поиска в дереве наследования.* В новых классах поиск идет сначала в ширину, а затем в высоту.

- *Извлечение атрибутов встроенными операциями.* Встр. операции больше не использ. методы \_\_getattr\_\_ и \_\_getattribute\_\_ для неявного извлечения атрибутов. Т.е. эти методы не вызываются для получения ссылок на методы перегрузки операторов с именами вида \_\_X\_\_ - поиск таких имен начинается с класса ,а не с экземпляра.

- *Новые особенности.* Новые классы имеют ряд особенностей, включая слоты, свойства, дескрипторы и новый метод \_\_getattribute\_\_. Чаще всего они использ-ся разработчиками, создающими инструментальные средства.

**Изменения в модели типов.**

Теперь классы явл-ся типами, сами классы экземплярами класса type, а типами экземпляров классов явл-ся их классы. Фактич. между встр. типами и пользовательскими нет никаких отличий. Поэтмоу и есть возможность наследовать встр. типы в своих классах, - поскольку подкласс, наследующий встроенный тип, такой как list, квалифицируется, как класс нового стиля, он становится пользовательским типом данных.

Эти изменения оказываются полезны при попытке явно выполнить проверку типа. В классической модели экземпляры классов автоматич. относятся к обобщенному типу "instance", а встр. объекты имеют более определнные типы:

# Python 2.6

class C: pass # Классические классы в 2.6

...

I = C()

type(I) # Экземпляры классов

<type 'instance'>

I.\_\_class\_\_

<class \_\_main\_\_.C at 0x025085A0>

type(C) # Но классы не явл-ся типами

<type 'classobj'>

C.\_\_class\_\_

AttributeError: class C has no attribute '\_\_class\_\_'

type([1, 2, 3])

<type 'list'>

type(list)

<type 'type'>

list.\_\_class\_\_

<type 'type'>

С появлением классов нового стиля типом экземпляра класса явл-ся сам класс, из которого он был создан, т.е. классы явл-ся обычными пользовательскими типами данных, - типом экземпляра явл-ся его класс, а пользовательские классы имеют тот же тип, что и встроенные объекты типов. Теперь классы тоже имеют атрибут \_\_class\_\_, потому что они явл-ся экземплярами класса type:

# Python 2.6

class C(object): pass # Классы нового стиля в 2.6

I = C()

type(I) # Типом экземпляра явл-ся его класс

<class '\_\_main\_\_.C'>

I.\_\_class\_\_

<class '\_\_main\_\_.C'>

type(C) # Классы явл-ся пользовательскими типами данных

<type 'type'>

C.\_\_class\_\_

<type 'type'>

type([1, 2, 3]) # Для встр. типов ничег оне изменилось

<type 'list'>

type(list)

<type 'type'>

list.\_\_class\_\_

<type 'type'>

То же самое справедливо для всех классов в 3.0, потому что в этой версии все классы автоматич. считаются классами нового стиля, даже если они явно не наследуют никаких суперклассов. Т.о. фактич. различия между пользовательскими классами и встроенными типами в 3.0 полностью исчезло:

# Python 3.4, выполняем команды в IDLE

>>> class C: pass # В 3.0 все классы явл-ся классами нового стиля

>>> I = C()

>>> type(I) # Типом экземпляра явл-ся его класс

<class '\_\_main\_\_.C'>

>>> I.\_\_class\_\_

<class '\_\_main\_\_.C'>

>>> type(C) # Классы - это типы, а типы - это классы

<class 'type'>

>>> C.\_\_class\_\_

<class 'type'>

>>> type([1, 2, 3]) # Классы и встр. типы ничем не отличаются

<class 'list'>

>>> type(list)

<class 'type'>

>>> list.\_\_class\_\_

<class 'type'>

Как видим, в 3.0 действительно классы явл-ся типами, а типы - классами. С технич. точки зрения каждый класс созд-ся из метакласса - класса с именм type или его подкласса, адаптирующего или управляющего процедурой создания классов.

**Значимость для операций проверки типа.**

В 3.0 типы экземпляров классов можно сравнивать непосредственно, точно так же как сравниваются встроенные объекты типов. Это обусловленно тем, что теперь классы явл-ся типами, а типом экземпляра явл-ся его класс:

# Python 3.4, выполняем команды в IDLE

>>> class C: pass

...

>>> class D: pass

...

>>> c = C()

>>> d = D()

>>> type(c) == type(d) # 3.0: сравниваются классы экземпляров

False

>>> type(c), type(d)

(<class '\_\_main\_\_.C'>, <class '\_\_main\_\_.D'>)

>>> c.\_\_class\_\_, d.\_\_class\_\_

(<class '\_\_main\_\_.C'>, <class '\_\_main\_\_.D'>)

>>> c1, c2 = C(), C()

>>> type(c1) == type(c2)

True

В случае классических классов в 2.6 и ранее, сравнивать типы экземпляров практич. бессмысленно, т.к. все экземпляры в этой модели имеют один тип - "instance". Чтобы действительн осравнить типы экземпляров, нужно сравнит ьзначения их атрибутов \_\_class\_\_ (в 3.0 это тприем действует так же, но надобности в нем нет):

# Python 2.6

>>> class C: pass

...

>>> class D: pass

...

>>> c = C()

>>> d = D()

>>> type(c) == type(d) # 2.6: все экземпляры имеют один и тот же тип

True

>>> c.\_\_class\_\_ == d.\_\_class\_\_ # Следует явно сравнивать классы

False

>>> type(c), type(d)

(<type 'instance'>, <type 'instance'>)

>>> c.\_\_class\_\_, d.\_\_class\_\_

(<class \_\_main\_\_.C at 0x0254687A0>, <class \_\_main\_\_.D at 0x0344687D0>)

Классы нового стиля в 2.6 действуют точно так же как и все классы в 3.0 - при сравнении типов экземпляров автоматич. сравниваются их классы:

# Python 2.6

>>> class C(object): pass

...

>>> class D(object): pass

...

>>> c = C()

>>> d = D()

>>> type(c) == type(d)

False

>>> type(c), type(d)

(<class '\_\_main\_\_.C'>, <class '\_\_main\_\_.D'>)

>>> c.\_\_class\_\_, d.\_\_class\_\_

(<class '\_\_main\_\_.C'>, <class '\_\_main\_\_.D'>)

Конечно в Питон проверка типов не явл-ся правильным решением, но когда нужно проверить тип экземпляра класса, лучше использовать встр. ф-цию instance.

**Все объекты наследуют класс "object".**

Т.к. все типы теперь явл-ся классами, каждый объект наследует встр. класс object, прямо или косвенно, через свой суперкласс. Рассмотрим пример:

>>> class C: pass

...

>>> X = C()

>>> type(X) # Теперь типом явл-ся сам класс экземпляра

<class '\_\_main\_\_.C'>

>>> type(C)

<class 'type'>

Типом экземпляра явл-ся сам класс, а типом класса явл-ся класс type. Также верно, что экземпляры и классы наследуют встр. класс object, т.к. он явно или неявно явл-ся суперклассом любого класса:

>>> isinstance(X, object)

True

>>> isinstance(C, object)

True

То же относится и к встр. типам, т.к. в модели нового стиля типы явл-ся классами и также наследуют object:

>>> type('spam')

<class 'str'>

>>> type(str)

<class 'type'>

>>> isinstance('spam', object)

True

>>> isinstance(str, object)

True

Фактич. сам класс type наследует класс object, а класс object наследует класс type, даже при том, что оба явл-ся совершенно различными объектами, - циклическая связь, венчающая объектную модель и вытекающая из того факта, что типы явл-ся классами, которые генерируют другие классы:

>>> type(type) # Все классы - это типы, и наоборот

<class 'type'>

>>> type(object)

<class 'type'>

>>> isinstance(type, object) # Все классы наследуют object, даже класс type

True

>>> isinstance(object, type) # Типы создают классы, и type явл-ся классом

True

>>> type is object

False

**Ромбоидальное наследование.** стр. 876

Классическая модель отличается наследованием - ромобоидальная схема, когда более одного суперкласса наследуют один и тот же суперкласс более высого уровня. Эта схема сложная и редко используемая.

В двух словах, в классич. модели поиск в дереве сначала движется строго вверх по дереву, а потом слева направо - т.е. сначала интер-р идет вверх всеми возможными путями по левой стороне дерева, потом идет назад и начинает поиск с первого суперкласса, расположенного правее предыдущего.

В новой модели поиск идет сначала в иширину - интер-р сначала смотрит все суперклассы, стоящие правее того, где поиск уже произведен, и только потом начинает подъем всеми возможными путями к суперклассу. Т.е. поиск идет по уровням дерева.

Т.о. в новой модели суперклассы, расположенные ниже, получают возможность переопределять атрибуты суперклассов, стоящих выше, независимо от вида деревьев множ. наследования. Кроме того каждый суперкласс просматривается не более одного раза, даже если он наслед-ся несколькими подклассами.

**Пример ромбоидального наследования.**

Два суперкласса В и С, которые наследуются классом D, имеют общего предка - класс А:

>>> class A: # Классическая модель (Питон 2.6)

attr = 1

>>> class B(A): # В и С имеют общего предка - А

pass

>>> class C(A):

attr = 2

>>> class D(B,C): # Сначала поиск дойдет до А, потом до С

pass

>>> x = D()

>>> x.attr # Порядок поиска: x, D, B, A

1

Здесь атрибут attr будет найден в классе А, потому что в классич. модели поимск идет сначала в высоту дерева, и только потмо вправо - поиск будет идти так: D, B, A, C.

В классах нового стиля поиск будет идти иначе, в след. порядке: D, B, C, A, и остановится на классе С:

>>> class A:

attr = 1

>>> class B(A):

pass

>>> class C(A):

attr = 2

>>> class D(B,C):

pass

>>> x = D()

>>> x.attr

2

**Явное разрешение конфликтов имен.**

Если ромбоидальный поиск для вас сложен или нужно полное управление поиском, можно сделать выбор нужного аргумента из любого места в дереве, выполнив присваивание или обозначив его в нужном месте:

>>> class A: # Классическая модель

attr = 1

>>> class B(A): # В и С имеют общего предка - А

pass

>>> class C(A):

attr = 2

>>> class D(B,C): # Выбрать С, справа

attr = C.attr

>>> x = D()

>>> x.attr # Работает как класс нового стиля

2

Здесь у нас идет имитация нового стиля: присваивание атрибуту attr в классе D явно выбирает версию атрибута из класса C, благодаря этому нарушается поиск классич. стиля.

Так же классы нового стиля могут имитировать порядок поиска в классич. модели:

>>> class A:

attr = 1

>>> class B(A):

pass

>>> class C(A):

attr = 2

>>> class D(B,C): # Выбрать A.attr, выше

attr = B.attr

>>> x = D() # Работает как классич. класс

>>> x.attr

1

Такими способами вы можете просто игнорировать порядок поиска и не полагаться на предположения куда вас может вывести поиск в той или иной модели.

Такой способ выбора атрибутов может применяться и к методам (методы - это обычные объекты):

>>> class A: # Примеры с новым стилем

def meth(s): print('A.meth')

>>> class C(A):

def meth(s): print('C.meth')

>>> class B(A):

pass

>>> class D(B,C): pass

>>> x = D() # Зависит от типа класса

>>> x.meth()

C.meth

>>> class D(B,C): meth = C.meth

>>> x = D()

>>> x.meth()

C.meth

>>> class D(B,C): meth = B.meth

>>> x = D()

>>> x.meth()

A.meth

Здесь мы явно выбираем методы, выполняя присваивание именам, находящимся ниже в дереве. Мы могли бы просто вызвать метод желаемого класса явно - на практике этот подход явл-ся более общепринятым, в особенности при работе с конструкторами:

class D(B,C):

def meth(self): # Переопределяется ниже

....

C.meth(self) # Вызовом выбрать метод класса С

Явное разрешение конфликтов имен гаранитрует, что правильная работа вашег окода не будет зависеть от версии Питона в будущем. Кроме того явное указание имен атрибутов может помочь при множественном наследовании, делая код ясным и уменьшая влияние оишбки расчетов поиска в дереве наследования.

**Пределы влияния изменений в порядке поиска.**

Помните, что изменение поиска в дереве наследования затрагивает только ромбоидальную схему, - во всех других схемах поиск не изменился при появлении классов нового стиля.

Но заметим, что суперкласс object всегда находится выше любого суперкласса, т.о. любой случай множ. наследования буде тсоотв. ромбоидальной схеме (пока непонятно как, но будем читать, изучать и поймем). Т.е. в новой модели класс object играет ту же роль, что и класс А в наших недавних примерах.

Не менее важно то, что суперкласс object в новой модели предоставляет методы по умолч., реализующие различ. встр. операции, включая методы вывода \_\_str\_\_ и \_\_repr\_\_. Можете вызвать ф-цию dir(object) и увидеть сами.

**Другие расширения в классах нового стиля.** стр. 880

**Слоты экземпляров.**

Присваивая список имен атрибутов в виде строк атрибуту \_\_slots\_\_ класса, в классах нового стиля можно ограничить множество разрешенный атрибутов для экземпляров класса и оптимизироват ьиспольз. памяти и производительность.

Обычно этот атрибут устанавливается присваиванием последовательности имен стрко переменной \_\_slots\_\_ на верз. уровне в инструкции class: только имена, перечисленные в списке \_\_slots\_\_, могут использ. как атрибуты экземпляра. Но, как и со всеми именами, прежде чем получить доступ к атрибутам экземпляра, им должно быть присвоено значение. Пример:

>>> class Limiter(object):

\_\_slots\_\_ = ['age', 'name', 'job']

...

>>> x = Limiter()

>>> x.age # Присваивание должно быть выполнено раньше использования

AttributeError: age

>>> x.age = 40

>>> x.age

40

>>> x.ape = 1000 # Недопустимое имя: отсутствует в списке \_\_slots\_\_

AttributeError: 'Limiter' object has no attribute 'ape'

Слоты - это своего рода нарушение динамической природы языка Питон, которая диктует, что операция присваивания может создавать любые имена. Но тем не менее слоты помогают нам убрать ошибки от простых опечаток, например. Выделение памяти для словаря с атрибутами, в каждом экземпляре, может оказаться дорогим удовольствием. Поэтому, для экономии пространства в памяти, атрибуты, перечисленные в слотах, сохраняются не в словаре, а в виде последовательности, что дает большую скорость их поиска.

**Слоты и обобщенные инструменты.**

Фактич. некоторые экземпляры со слотами вообще могут не иметь атрибут словаря \_\_dict\_\_, что может сделать некоторые метапрограммы намного более сложными.

Инструменты, которые получают списки атрибутов или обращаются к атрибутам, используя имена в виде строк, например, должны использовать более универсальные механизмы, чем атрибут \_\_dict\_\_. К таким механизмам можно отнести встр. ф-ции getattr, setattr и dir, способные отыскивать атрибуты и в \_\_dict\_\_ и в \_\_slots\_\_.

Напр. экземпляры классов, где использ. слоты, обычно не имеют атрибут словаря \_\_dict\_\_ - вместо него пространство для атрибутов в экземпляре выделяется с применением дескрипторов класса, которые будут рассматриваться в гл. 37. Только имена, перечисленные в \_\_clots\_\_, смогут использ-ся как атрибуты экземпляра, однако значения этих атрибутов могут извлекаться и изменяться обычными способами:

>>> class C:

\_\_slots\_\_ = ['a', 'b'] # По умолчанию наличе \_\_slots\_\_ означает

# отсутствие \_\_dict\_\_

>>> x = C()

>>> x.a = 1

>>> x.a

1

>>> x.\_\_dict\_\_

AttributeError: 'C' object has no attribute '\_\_dict\_\_'

>>> getattr(x, 'a')

1

>>> setattr(x, 'b', 2) # Однако ф-ции getattr() и setattr()

>>> x.b # по-прежнему работают

2

>>> 'a' in dir(x) # И dir () также отыскивает атрибуты в слотах

True

>>> 'b' in dir(x)

True

В отсутствие слвоаря с пространством имен невозможно присвоить значения атрибутам экземпляра, имена которых отсутствуют в списке слотов:

>>> class D:

\_\_slots\_\_ = ['a', 'b']

def \_\_init\_\_(self): self.d = 4 # Невозможно добавить новый атрибут,

# когда отсутствует атрибут \_\_dict\_\_

>>> x = D()

AttributeError: 'D' object has no attribute 'd'

Но все-таки добавлять атрибуты можно - для этого нужно включить имя \_\_dict\_\_ в список \_\_slots\_\_, разрешив тем самым создать словарь с пространством имен. В этом случае будут действовать оба метханизма хранения имен:

>>> class D:

\_\_slots\_\_ = ['a', 'b', '\_\_dict\_\_'] # Добавляем \_\_dict\_\_ в слоты

c = 3 # Атрибуты класса действуют как обычно

def \_\_init\_\_(self): self.d = 4 # Имя d будет добавлено в \_\_dict\_\_,

# а не в \_\_slots\_\_

>>> x = D()

>>> x.d

4

>>> x.\_\_dict\_\_ # Некоторые объекты имеют оба атрибута, \_\_dict\_\_ и \_\_slots\_\_

{'d': 4} # getattr() может извлекать атрибуты любого типа

>>> x.\_\_slots\_\_

['a', 'b', '\_\_dict\_\_']

>>> x.c

3

>>> x.a # Все атрибуты экземпляра не определены,

AttributeError: a # пока им не будет присвоено значение

>>> x.a = 1

>>> getattr(x, 'a'), getattr(x, 'c'), getattr(x, 'd')

(1, 3, 4)

Если будет нужно реализовать универсальный способ получения знач. всех атрибутов экземпляра, нужно учесть наличие двух форм хранения атрибутов или использ. ф-цию dir, которая дополнительно возвращает все унаследованные атрибуты (в примере для полученяи ключей использ. итератор):

>>> for attr in list(x.\_\_dict\_\_) + x.\_\_slots\_\_:

print(attr, '=>', getattr(x, attr))

d => 4

a => 1

b => 2

\_\_dict\_\_ => {'d': 4}

- в книге не было ничего написано, но мне пришлось вручную присвоить атрибуту x.b значение 2, т.к. в противном случае после строки a => 1 интерпретатор выдавал мне ошибку.

Т.к. любой из этих атрибутов может отсутствовать, более правильный способ выглядит, как показано ниже:

>>> for attr in list(getattr(x, '\_\_dict\_\_', [])) + getattr(x, '\_\_slots\_\_', []):

print(attr, '=>', getattr(x, attr))

Примечание! У меня все равно ошибка выходит, как и в предыдущем примере. Не знаю почему, но все равно выдает ошибку, хотя должен выдать знач. по умолчанию. Разберемся потом, когда подучим Питон.

...

d => 4

a => 1

Traceback (most recent call last):

File "<pyshell#78>", line 2, in <module>

print(attr, '=>', getattr(x, attr))

AttributeError: b

**Несколько суперклассов со списками \_\_slots\_\_.**

Обратите внимание, что в этой реализации просматривается содержимое атрибута \_\_slots\_\_ только самого нижнего в дереве класса, наследуемого экземпляром. Если в дереве есть неск-ко классов, имеющих атрибуты \_\_slots\_\_, универсальные инструменты должны иначе подходить к получению списка атрибутов (напр. рассматривать имена слотов как атрибуты классов, а не экземпляров).

Объявления слотов могут присутствовать сразу в неск-ких классах в дереве, они они имеют доп. ограничения, которые будет сложно объяснить, пока вы еще не знаете, что слоты реализованы в виде дескрипторов на уровне класса (рассмотрим это позднее):

- Если подкласс наследует суперкласс, который не имеет атрибута \_\_slots\_\_, атрибут \_\_dict\_\_ суперкласса будет доступен всегда, что делает бессмысленным использование атрибута \_\_slots\_\_ в подклассе.

- Если класс определяет слот с тем же именем, что и суперкласс, версия имени, объявленная в суперклассе, будет доступна только при непосредственном обращении к дескриптору в суперклассе.

- Поскольку объявление \_\_slots\_\_ имеет значение только для класса, в котором оно присутствует, подклассы автоматич. получат атрибут \_\_dict\_\_, если не определяет свой атрибут \_\_slots\_\_.

В общем для получения списка атрибутов экземпляра при использ. слотов в нескольких классах может потребоваться: подъем по дереву классов вручную, использование ф-ции dir или подход, при котором имена слотов рассматриваются как совершенно отдельная категория имен:

>>> class E:

\_\_slots\_\_ = ['c', 'd'] # Суперкласс имеет слоты

>>> class D(E):

\_\_slots\_\_ = ['a', '\_\_dict\_\_'] # Его подкласс тоже имеет слоты

>>> X = D()

>>> X.a=1; X.b=2; X.c=3 # Экземпляр объединяет слоты в себе

>>> X.a, X.c

(1, 3)

>>> E.\_\_slots\_\_ # Но в классах слоты не объединяются

['c', 'd']

>>> D.\_\_slots\_\_

['a', '\_\_dict\_\_']

>>> X.\_\_slots\_\_ # Экземпляр наследует \_\_slots\_\_

['a', '\_\_dict\_\_'] # \*ближайшего\* класса

>>> X.\_\_dict\_\_ # И имеет собственный атрибут \_\_dict\_\_

{'b': 2}

>>> for attr in list(getattr(X, '\_\_dict\_\_', [])) + getattr(X, '\_\_slots\_\_', []):

print(attr, '=>', getattr(X, attr))

b => 2 # Слоты суперкласса отсутствуют!

a => 1

\_\_dict\_\_ => {'b': 2}

>>> dir(X) # dir() включает имена всех слотов

['\_\_class\_\_', '\_\_delattr\_\_', '\_\_dict\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_le\_\_', '\_\_lt\_\_', '\_\_module\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_slots\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', 'a', 'b', 'c', 'd']

Когда нужно универсальное решение, слоты лучше рассматривать как атрибуты класса, а не пытаться представлять их акк обычные атрибуты экземпляра.

**Свойства класса.**

Механизм, известный как свойства, обеспечивает в классах новго тсиля еще один способ определния методов, вызываемых автоматич. при обращении или присваивании атрибутам экземпляра. Это т.н. альтерантива методам \_\_getattr\_\_ и \_\_setattr\_\_. Свойства обладают тем же эффектом, что и эти два метода, тольк ов этом случае вып-ся вызов метода даже при простом обращении к атрибуту, что полезно если эти атрибуты вычисляются динамически.

Свойства (и слоты) оснвоаны на новом понятии дескрипторов атрибутов - темы слишком сложной, чтобы обсуждать её здесь (возможно позднее, не указал автор, или самостоятельно).

Свойства - это тип объектов, котоырй присваивается именам атрибутов класса. Они создаются вызовом ф-ции property, которой передаются три метода (обработчик операций чтения, присваивания и удаления), и строкой документирования - если в каком-то аргументе передается знач. None, то эта операция не поддерживается.

Определение свойств, обычно, идет на верх. уровне инструкции class (напр. name = property(...)). При попытке доступа к атрибуту класса (т.е. obj.name) автоматич. будет вызываться один из методов доступа. Например метод \_\_getattr\_\_ позволяет классам перехватывать попытки доступа к неопределнным атрибутам класса:

>>> class Classic:

def \_\_getattr\_\_(self, name):

if name == 'age':

return 40

else:

raise AttributeError

>>> x = Classic()

>>> x.age # Вызовет метод \_\_getattr\_\_

40

>>> x.name # Вызовет метод \_\_getattr\_\_

AttributeError

Ниже этот же пример, но с использ. свойств (для корректной работы в 2.6 нужно, чтобы классы прямо или косвенно наследовали класс object, т.к. у нас 3.4 то можно не беспокоиться):

>>> class Newprops:

def getage(self):

return 40

age = property(getage, None, None, None) # get, set, del, docs

>>> x = Newprops()

>>> x.age # Вызовет метод getage

40

>>> x.name # Нормальная опреация извлечения

AttributeError: 'Newprops' object has no attribute 'name'

Иногда свойства могут быть даже быстрее чем использ-ие стандартных подходов. Напр. когда добавляется поддержка операции присваивания атрибуту, свойства становятся более привлекательными - код выглядит компактнее и в операцию присваивания не вовлекаются доп. вызовы методов, если не требуется производить доп. вычислений:

>>> class Newprops:

def getage(self):

return 40

def setage(self, value):

print('set age:', value)

self.\_age = value

age = property(getage, setage, None, None)

>>> x = Newprops()

>>> x.age # Вызовет метод getage

40

>>> x.age = 42 # Вызовет метод setage

set age: 42

>>> x.\_age # Нормальная операция извлечения; нет вызова getage

42

>>> x.job = 'trainer' # Нормальная операция присваивания; нет вызова setage

>>> x.job # Нормальная операция извлечения; нет вызова getage

'trainer'

При классическом решении проблемы класс мог бы производить лишние вызовы метода и, возможно, выполнять присваивание значения атрибуту с использ. словаря (или с пом. нового метода \_\_setattr\_\_, наследуемого классами нового стиля от суперкласса object), чтобы избежать зацикливания:

>>> class Classic:

def \_\_getattr\_\_(self, name): # При ссылке на неопределенный атрибут

if name == 'age':

return 40

else:

raise AttributeError

def \_\_setattr\_\_(self, name, value): # Для всех операций присваивания

print('set:', name, value)

if name == 'age':

self.\_\_dict\_\_['\_age'] = value

else:

self.\_\_dict\_\_[name] = value

>>> x = Classic()

>>> x.age # Вызовет метод \_\_getattr\_\_

40

>>> x.age = 41 # Вызовет метод \_\_setattr\_\_

set: age 41

>>> x.\_age # Определен: нет вызова \_\_getattr\_\_

41

>>> x.age

40

>>> x.job = 'trainer' # Запустит метод \_\_setattr\_\_ опять

set: job trainer # Определен: нет вызова \_\_getattr\_\_

>>> x.job

'trainer'

Для этого примера свойства обладают неоспоримым преимуществом. Но в некоторых приложениях \_\_getattr\_\_ и \_\_setattr\_\_ по-прежнему могут быть востребованы для более динамичных или универсальных интерфейсов. Например, во многих случаях нельзя заранее определить набор атрибутов, которые могут даже не существовать на момент написания класса. В таких случая использ. \_\_getattr\_\_ и \_\_setattr\_\_ предпочтительнее. Также простые ситуации могут обсулживаться этими обработчиками, поэтому свойства нужно рассматривать как дополнительное и необязательное к использованию расширение.

**Метод \_\_getattribute\_\_ и дескрипторы.**

Метод \_\_getattribute\_\_ есть только в классах нового стиля и позволяет перехватывать все попытки обращения к атрибутам, а не только к неопределенным (как \_\_getattr\_\_). Кроме того этот метод более сложен в обращении, чем \_\_getattr\_\_, из-за более высокой вероятности зацикливания, ии чем \_\_setattr\_\_, но по др. причинам.

В дополнение к свойствам и методам перегрузки в Питон есть понятие дескрипторов атрибутов - классов, с методами \_\_get\_\_ и \_\_set\_\_, которые присваиваются атрибутам классов. Они наследуются экземплярами и перехватывают попытки доступа к определенным атрибутам.

Дескрипторы представляют собой более обобщенную форму свойств. Фактич. свойства - это упрощенный вариант определения дескрипторов специфич. типа, основанных на вызовах функций, управляющих доступом к атрибутам. Кроме того, дескрипторы использ-ся для реализации слотов, с которыми мы познакомились выше.

**Метаклассы.**

Наряду с изменениями нового стиля (классы - это типы, а типы - это классы), в Питон был выработан более согласованный протокол метаклассов, которые явл-ся подклассами объекта type и реализуют операции создания классов.

Они обеспечивают отличную возможность управления объектами классов и их расширения. Тема метаклассов широка и для большинства программистов не явл-ся необходимой, поэтому рассматривать её не будем, только коснемся немного далее.

**Статические методы и методы класса.** стр. 887

С Питон 2.2 появилась возможность определять методы класса, которые можно вызвать без участия экземпляра: статические методы работают почти так же, как обычные ф-ции, только расположенные внутри класса, а методы класса получают сам класс вместо экземпляра. Оба вида можно использовать и в классических классах.

Чтобы эти методы работали, внутри класса должны вызываться спец. ф-ции staticmethod и classmethod или использ-ся декораторы. В 3.0 методы, которым не передается ссылка на экземпляр и которые вызываются только через имя класса, не требуют объявления с пом. ф-ции staticmethod, но такое обьъявление обязательно для методов, которые предполагается вызывать через экземпляры.

**Зачем нужны спец. методы?**

Обычно методы получают объект экземпляра в первом аргументе (self), который играет роль подразумеваемого объекта вызова метода. На этот момент (в книге) есть две воможности изменить эту модель. Прежде чем узнать о них, поясним почему это важнодля нас.

Иногда в прогах нужно организовать обработку данных, связанных с классами, а не экземплярами. Напр. следить за числом экземпляров или вести список всех экземпляров класса, находящихся в данный момент в памяти. Такая информация связана с классами и должна обрабатываться на уровне класса, а не экземпляров. Т.е. такая инфа обычно сохр-ся в самом классе и обрабатывается, независимо от наличия экземпляров класса.

Для решения таких задач часто хватает простых ф-ций, определяемых за пределами класса. Такие ф-ции могут обращаться к атрибутам класса через его имя - им требуется доступ к данным класса и никогда - к экземплярам. Но чтобы теснее связать такой код с классом и обеспечит ьвозможную адаптацию, лучше помещать такие ф-ции внутрь самого класса. Для этого нам и нужны методы класса, которы е не ожидают получить аргумент self с экземпляром.

Статические методы - простые ф-ции без аргумента self, вложенные в определение класса и предназначенные для работы с атрибутами класса, а не экземпляра. Статич. методы никогда не получают ссылку self на экземпляр, независимо от того, вызываются они через имя класса или через экземпляр. Такие методы обычно использ. для обработки информации, имеющей отношение ко всем экземплярам, а не для реализации поведения экземпляров.

Крмое того есть понятие методов класса. Методы класса использ. реже, и в первом аргументе им автоматич. передается объект класса, независимо от того, вызываются они через имя класса или через экземпляр. Такие метолы могут получить доступ к данным класса через аргумент self, даже когда они вызываются относительно экземпляра. Обычные методы (т.н методы экземпляра) пр ивызове получают подразумеваемый экземпляр, а статич. методы и методы класса - нет.

**Статические методы в 2.6 и 3.0**

Статич. методы поддерживаются и в 2.6 и в 3.0, н ов 3.0 требования к ним несколько изменились.

Напомним, что в обоих версиях, когда метод вызывается относительно экземпляра, ему всегда передается ссылка на этот экземпляр. Но в 3.0 извлечение методов через имя класса интерпретируется иначе чем в 2.6:

- В 2.6 операция извлеченяи метода по имени класса возвращает несвязанный метод, при вызове которого требуется вручную передавать экземпляр.

- В 3.0 операция извлечения метода по имени класса возвращает простую функцию, которой не требуется передавать ссылку на экземпляр.

Др. словами, в 2.6 методам всегда нужно передавать экземпляр, не зависимо вызываются они через имя класса или через экземпляр. В 3.0 напротив, экземпляр требуется передавать методам, тольк оесли они ожидают получить его - методы без аргумента self иогут вызываться через имя класса без передачи ссылки на экземпляр. Т.е. в 3.0 можно объявлять простые ф-ции внутри класса, при условии, что они не ожидают получить и им не будет передаваться аргумент со ссылкой на экземпляр. В результате:

- В 2.6 всегда нужн ообъявлять метод как статич., чтобы можно было вызывать его без передачи ссылки на экземпляр, не зависимо вызывается от через имя класса или через экземпляр.

- В 3.0 не нужн ообъявлять метод как статич., если он будет вызываться только через имя класса, но мы обязаны объявлять его статич., если он может вызываться через экземпляр.

Для пример - пусть нам нужно использ. атрибуты класса для подсчета числа экземпляров, созданных из класса. В файле spam.py - класс содержит счетчик в виде атрибута класса, конструктор, который наращивает счетчик, и метод, выводящий знач. счетчика. Не забываем, что атрибуты класса совместно использ. всеми экземплярами. Поэтому наличие счетчика непосредственно в объекте класса гарантирует, что он будет хранить число всех экземпляров:

class Spam:

numInstances = 0

def \_\_init\_\_(self):

Spam.numInstances = Spam.numInstances + 1

def printNumInstances():

print("Number of instances created: ", Spam.numInstances)

Метод printNumInstances обрабатывает данные класса, а не экземпляров - эти данные явл-ся общими для всех экземпляров. Поэтому нам нужна возможность вызывать его, не передавая ссылку на экземпляр. Ведь нам не нужно создавтаь новый экземпляр для получения числа экземпляров, ведь это изменит число экземпляров, которое мы хотим получить (замкнутый круг)! Др. словами нам нужен "статический" метод, не имеющий аргумента self.

Будет ли работать такая реализация зависит от версии интерпретатора и способа вызова - через имя класса или через экземпляр. В версии 2.6 и ранее не будет работать - вызов метода printNumInstances без аргумента self, хоть через имя класса, хоть через экземпляр будет неудачным.

Проблема состоит в том, что в 2.6 несвязанные методы экземпляра - это не то же самое, что простые ф-ции. Т.е. вызов метода, такого как printNumInstances, через имя класса и без передачи ему экземпляра будет неудачным в 2.6, но будет работать в 3.0. С др. стороны, попытка вызвать метод относительн оэкземпляра будет неудачна в обеих версиях Питон, т.к. в этом случае интерпретатор автоматич. передаст экземпляр методу, который не имеет соотв. аргумента, чтобы принять его:

Spam.printNumInstances() # Ошибка в 2.6, работает в 3.0

instances.printNumInstances() # Ошибка в обеих версиях

Итак, чтобы в обеих версиях вызывать и так и так статич. метод нужно использ. другую реализацию, или пометить методы как специальные. Рссмотрим оба способа по порядку.

**Альтернативы статическим методам.**

Если для доступа к атрибутам класса нужн овызывать ф-ции, которые не принмиают ссылку на экземпляр, самая простая мысль - нужн осделать метод обычной ф-цией, а не методом класса. Пример:

def printNumInstances():

print("Number of instances created: ", Spam.numInstances)

class Spam:

numInstances = 0

def \_\_init\_\_(self):

Spam.numInstances = Spam.numInstances + 1

>>> import spam

>>> a = spam.Spam()

>>> b = spam.Spam()

>>> c = spam.Spam()

>>> spam.printNumInstances()

Number of instances created: 3

>>> spam.Spam.numInstances

3

- Важный замечание: ф-ция может находиться слишком далеко от определения класса (в сценарии или модуле) и не может адаптироваться в классах.

Т.к. имя класса доступно простой ф-ции в виде глобальной переменой, всё работает прекрасно. Обратите внимание, что имя самой ф-ции также явл-ся глоабльным, но тольк ов этом единственном модуле - оно не будет конфликтовать с именами в др. модулях программы.

Поскольку в Питон уже есть модули, играющие роль инструментов разделения пространств имен, можно было бы утверждать, что обычно нет никакой необходимости упаковывать функции в классы, если они не реализуют функциональность объектов. Простые ф-ции внутри модуля, способны решать большую часть задач, которые возлагаются на методы класса, не имеющие аргумента self, и уже связаны с классом, т.к. расположены в том же модуле.

К сожалению такой подход далек от идеала. С одной стороны в облатсь видимости файла добавл-ся лишнее имя, которое использ. для рабоыт с единственным классом. С другой - ф-ция не имеет тесной связи с классом. Фактич. определение ф-ции может находиться за сотни строк от определения класса. Но самое неприятное, что простые ф-ции не могут адаптироваться в подклассах (т.е. переопределяться или замещаться в подклассах).

Мы можем еще раз изменить наш сценарий, используя обычный метод и вызывая его через (или передавать вручную) экземпляр:

class Spam:

numInstances = 0

def \_\_init\_\_(self):

Spam.numInstances = Spam.numInstances + 1

def printNumInstances(self):

print("Number of instances created: ", Spam.numInstances)

>>> from spam import Spam

>>> a, b, c = Spam(), Spam(), Spam()

>>> a.printNumInstances()

Number of instances created: 3

>>> Spam.printNumInstances(a)

Number of instances created: 3

>>> Spam().printNumInstances() # Эта попытка извлечь счетчик изменяет его!

Number of instances created: 4

К сожалению, такой подход непригоден в случае отсутствия доступного экземпляра, а создание нового экземпляра изменяет данные класса, как видим в последней строке. Лучшее решение заключается в том, чтобы каким-либо способом пометить метод класса, который не требует передавать ему ссылку на экземпляр.

Использование статических методов и методов класса.

Итак, есть еще способ писать простые ф-ции, связаннеы с классом, которые могут вызываться через имя класса или экземпляры. И появилась возможност ьсоздавать классы со статич. методами и методами класса, ни один из которы хне требует передачи экземпляра класса в виде аргумента. Для этого нм нужны ф-ции staticmethod и classmethod. Обе ф-ции помещают объект как специальный, т.е. не требующий передачи экземпляра, в случае с ф-цией staticmethod, и как требующий передачу класса, в случае с ф-цией classmethod. Например:

class Methods:

def imeth(self, x): # Обычный метод экземпляра

print(self, x)

def smeth(x): # Статический метод: экземпляр не передается

print(x)

def cmeth(cls, x): # Метод класса: получает класс, но не экземпляр

print(cls, x)

smeth = staticmethod(smeth) # Сделать smeth статическим методом

cmeth = classmethod(cmeth) # Сделать cmeth методом класса

Обратите внимание, как последние две строки просто переприсваивают имена методов smeth и cmeth, т.е. переопределяются инструкции def, выполненные ранее.

С технич. точки зрения Питон теперь поддерживает три вида методов: методы экземпляра, статические методы и методы класса. Кроме того, в 3.0 эта модель дополнена возможностью без лишних сложносте создавать в классах простые ф-ции, которые играют роль статических методов при обращении к ним через имя класса.

Методы экземпляра - это обычные методы, которые мы видели в этой книге. Для воздействия на объект экземпляра всегда следует вызывать методы экземпляра. Когда методы вызываются через экземпляр, интер-р автоматич. передает экземпляр в первом аргументе - когда метод вызывается через имя класса, экземпляр необходимо передавать методам вручную:

>>> obj = Methods() # Создать экземпляр

>>> obj.imeth(1) # Обычный вызов, через экземпляр

<\_\_main\_\_.Methods object at 0x02706630> 1 # Будет преобразован в вызов imeth(obj, 1)

>>> Methods.imeth(obj, 2) # Обычный вызов, через класс

<\_\_main\_\_.Methods object at 0x02706630> 2 # Экземпляр передается явно

Статич. методы, напротив, вызываются без аргумента с экземпляром. В отличие от простых ф-ций, за пределами класса их имена ограничены областью видимости класса, в котором они определяются и могут отыскиваться механизмом наследования.

Ф-ции, которым не передаются ссылки на экземпляры, в 3.0 могут вызываться обычным образом через имя класса, но в 2.6 по умолч. - никогда. Применение встр. ф-ции staticmethod обеспечивает возможность вызывать такие методы через экземпляр в 3.0, и через имя класса и через экземпляр - в 2.6 (в 3.0 вызов через имя класса возможен и без staticmethod, но вызов через экземпляр - нет):

>>> Methods.smeth(3) # Вызов статич. метода, через имя класса

3 # Экземпляр не передается и не ожидается

>>> obj.smeth(4) # Вызов статич. метода, через экземпляр

4 # Экземпляр не передается

Методы класса похожи на них, он интер-р автоматич. передает методам класса сам класс (а не экземпляр) в первом аргументе, независимо от того, вызываются они через имя класса или через экземпляр:

>>> Methods.cmeth(5) # Вызов метода класса, через имя класса

<class '\_\_main\_\_.Methods'> 5 # Будет преобразован в вызов cmeth(Methods, 5)

>>> obj.cmeth(6) # Вызов метода класса, через экземпляр

<class '\_\_main\_\_.Methods'> 6 # Будет преобразован в вызов cmeth(Methods, 6)

**Подсчет количества экземпляров с помощью статических методов.**

Реализуем статич. метод, он помечен как специальный, поэтому ему никогда автоматич. не будет передаваться ссылка на экземпляр:

class Spam:

numInstances = 0

def \_\_init\_\_(self):

Spam.numInstances += 1

def printNumInstances():

print("Number of instances created: ", Spam.numInstances)

printNumInstances = staticmethod(printNumInstances)

Такая реализация позволяет вызывать метод, не принимающий аргумент self, через имя класса или через любой экземпляр в обеих версиях Питон:

>>> from spam import Spam

>>> a, b, c = Spam(), Spam(), Spam()

>>> Spam.printNumInstances() # Вызывается как простая функция

Number of instances created: 3

>>> a.printNumInstances() # Аргумент с экземпляром не передается

Number of instances created: 3

По сравнению с простым перемещением printNumInstances за пределы класса, эта версия требует дополнительный вызов ф-ции staticmethod. При этом здесь область видимости ф-ции ограничена классом (не будет конфликта с др. именами в модуле), код перемещен туда, где он используется (внутрь class), и подклассы получают возможност ьадаптировать статический метод наследованием - этот подход более удобен, чем импорт ф-ций из файлов, в которых находятся определения суперклассов. Это иллюстрирует следующий подкласс и листинг нового интерактивного сеанса:

>>> from spam import Spam

>>> class Sub(Spam):

def printNumInstances(): # Переопределяет статич. метод

print("Extra stuff ...")

Spam.printNumInstances() # Который вызывает оригинал

printNumInstances = staticmethod(printNumInstances)

>>> a = Sub()

>>> b = Sub()

>>> a.printNumInstances() # Вызов через экземпляр подкласса

Extra stuff ...

Number of instances created: 2

>>> Sub.printNumInstances() # Вызов через имя подкласса

Extra stuff ...

Number of instances created: 2

>>> Spam.printNumInstances()

Number of instances created: 2

Кроме того, классы могут наследовать статич. методы, не переопределяя их, - они будут вызываться без передачи им ссылки на экземпляр, независимо от их местоположения в дереве классов:

>>> class Other(Spam): pass

...

>>> c = Other()

>>> c.printNumInstances()

Number of instances created: 3

**Подсчет экземпляров с помощью методов класса.**

Итак, использ. метод класса, который в первом аргументе принимает класс экземпляра. Методы класса автоматич. получают объект экземпляра:

>>> class Spam:

numInstances = 0 # Вместо статич. метода используется метод класса

def \_\_init\_\_(self):

Spam.numInstances += 1

def printNumInstances(cls):

print("Number of instances: ", cls.numInstances)

printNumInstances = classmethod(printNumInstances)

Использ. этот класс так же как и пердыдущая версия, но его метод printNumInstances принимает объект класса, а не экземпляра, независимо от того, вызывается он через имя класса или через экземпляр:

>>> a, b = Spam(), Spam()

>>> a.printNumInstances() # В первом аргументе передается класс

Number of instances: 2

>>> Spam.printNumInstances() # Также в первом аргументе передается класс

Number of instances: 2

Знайте, что методы класса принимают самый близкий к объекту вызова класс. Напр. определим подкласс, адаптирующий метод Spam.printNumInstances так, чтобы он доп. выводил свой аргумент cls, и проверим:

>>> class Spam:

numInstances = 0

def \_\_init\_\_(self):

Spam.numInstances += 1

def printNumInstances(cls):

print("Number of instances: ", cls.numInstances, cls)

printNumInstances = classmethod(printNumInstances)

>>> class Sub(Spam):

def printNumInstances(cls):

print("Extra stuff ...", cls)

Spam.printNumInstances()

printNumInstances = classmethod(printNumInstances)

>>> class Other(Spam): pass

- всякий раз, когда будет вызываться метод класса, интер-р будет передавать ему ближайший класс, даже для подклассов, не имеющих собственной реализации метода класса:

>>> x, y = Sub(), Spam()

>>> x.printNumInstances() # Вызов через экземпляр подкласса

Extra stuff ... <class '\_\_main\_\_.Sub'>

Number of instances: 2 <class '\_\_main\_\_.Spam'>

>>> Sub.printNumInstances() # Вызов через сам подкласс

Extra stuff ... <class '\_\_main\_\_.Sub'>

Number of instances: 2 <class '\_\_main\_\_.Spam'>

>>> y.printNumInstances()

Number of instances: 2 <class '\_\_main\_\_.Spam'>

В первом случае метод класса выз-ся через экземпляр подкласса Sub, и интер-р передает ближайший к экземпляру класс Sub. Тут все правильно, переопределнный метод вызывает оригинальный метод, который принимает класс Spam в первом аргументе. Но посмотрите что будет при обращении к объекту, который просто наследует метод класса:

>>> z = Other()

>>> z.printNumInstances()

Number of instances: 3 <class '\_\_main\_\_.Other'>

Здесь методу класса Spam передается класс Other. В этом случае метод работает потому, что он всего лишь извлекает значение счетчика, котоырй обнаруживает в классе Spam, благодаря механизму наследования. Однако если бы этот метод попытался присвоить новое знач. атрибуту класса, он изменил бы атрибут класса Other, а не Spam! В нашем случае было бы лучше жестко указать имя класса, в котором производится изменение данных, чем полагаться на передаваемый аргумент класса.

**Подсчет экземпляров для каждого класса с помощью методов класса.**

Фактич. методы класса всегда получают ближайший класс в дереве наследования, поэтому:

- Применение статич. методов, в которых явно указывается имя класса, может оказаться более удачным решением для обработки данных класса.

- Методы классов лучше подходят для обработки данных, которые могут отличаться для каждого конкретного класса иерархии.

Напр. для реализации счетчиков экземпляров каждого класса в отдельности лучше подошли бы методы класса. В след. примере суперкласс определяет метод класса, управляющий инфой о состоянии, которая отличается для разных классов в дереве. - подобно тому как методы экземпляра управляют информацией о состоянии экземпляров:

>>> class Spam:

numInstances = 0

def count(cls): # Счетчик экземпляров для каждого отдельного класса

cls.numInstances += 1 # cls - ближайший к экземпляру класс

def \_\_init\_\_(self):

self.count() # Передаст self.\_\_class\_\_ для подсчета

count = classmethod(count)

>>> class Sub(Spam):

numInstances = 0

def \_\_init\_\_(self): # Переопределяет \_\_init\_\_

Spam.\_\_init\_\_(self)

>>> class Other(Spam): # Наследует \_\_init\_\_

numInstances = 0

>>> x = Spam()

>>> y1, y2 = Sub(), Sub()

>>> z1, z2, z3 = Other(), Other(), Other()

>>>

>>> x.numInstances, y1.numInstances, z1.numInstances

(1, 2, 3)

>>> Spam.numInstances, Sub.numInstances, Other.numInstances

(1, 2, 3)

Статич. методы и методы класса могут использ. и в других ситуациях, которые мы не будем рассматривать здесь - ищите доп. инфу в др. источниках. В последних версиях Питон создание статич. методов и методов класса можно упростить, используя декораторы функций.

**Декораторы и метаклассы: часть 1.** стр. 896

Декораторы функций обеспечивают способ определения специальных режимов работы функций, обертывая ихдополнительным слоем логики, реализованной в виде других функций.

Декораторы явл-ся универсальным инструментом: их удобно использ. для добавления самой разной лоигки не только к статич. методам, но и к любым другим функциям. Напр. их можно использ. для расширения функций программным кодом, выполняющим регистрацию вызовов этих ф-ций, проверяющим типы передаваемых аргументов в процессе отладки и т.д. В некоей степени декораторы ф-ций напоминают шаблон проектирования делегирования, но их главная цель состоит в том, чтобы расширять определенные функции или методы, а не весь интерфейс объекта.

Питон располагает несколькими встр. декораторами, но можно создавать и свои собственные. Хоть декораторы и не привязаны строго к классам, тем не менее пользовательские декораторы часто оформляются как классы, в которых сохр-ся оригинальная ф-ция наряду с др. данными. Кроме того недавно появилось расширение (в 2.6 и 3.0) - декораторы классов, непосредственно связанные с моделью классов, и метаклассы, играющие похожую роль.

**Основы декораторов функций.** стр. 897

Синтаксически, декоратор функции - это разновидность объвления функции времени выполнения. Декоратор ф-ции записывается в строке, непосредственно перед строкой с инструкцией def, которая определяет ф-цию или метод, и состоит из символа @, за которым следует то, что называется мета-функцией, - функция (или др. вызываемый объект), которая управляет другой функцией. В наст. время декораторы могут быть оформлены в виде декораторов:

class C:

@staticmethod # Синтаксис декорирования

def meth(): …

С технич. точки зрения, это объявление имеет тот же эффект, что и фрагмент ниже (передача ф-ции декоратору и присваивание рез-та первоначальному имени ф-ции):

class C:

def meth():

...

meth = staticmethod(meth) # Повторное присваивание имени

Результат, возвращаемый функцией-декоратором, повторно присв-ся имени метода. В рез-те вызов метода по имени ф-ции фактич будет приводить к вызову рез-та, полученному от декоратора staticmethod.

Декоратор может возвращать объекты любого типа, поэтмоу данный прием позволяет декоратору вставлять доп. уровень логики, который будет запускаться при каждом вызове. Декоратор ф-ции может возвращать как оригинальную ф-цию, так и новый объект, в котором хранится оригинальная ф-ция, переданная декоратору, которая будет вызываться косвенно, после того как будет выполнен доп. слой логики.

Благодаря этому расширению мы имеем удачный способ объявить статич. метод в примере из предыдущего раздела в обеич версиях:

class Spam:

numInstances = 0

def \_\_init\_\_(self):

Spam.numInstances = Spam.numInstances + 1

@staticmethod

def printNumInstances():

print("Number of instances created: ", Spam.numInstances)

a = Spam()

b = Spam()

c = Spam()

Spam.printNumInstances() # Теперь вызовы могут производиться как через класс,

# так и через экземпляр!

a.printNumInstances() # В обоих случаях будет выведено:

# "Number of instances created: 3"

Имейте в виду, что staticmethod - это все та же встр. ф-ция. Она может использ. как декоратор просто потому, что принимает др. функцию в виде аргумента и возвращает вызываемый объект. Фактич. любая такая ф-ция может использ. в качестве декоратора, даже пользовательские ф-ции, которые мы пишем сами.

**Первый пример декоратора функций.**

Вспомним, как в гл. 29 говорилось, что метод перегрузки оператора \_\_call\_\_ реализует в экземплярах классов интерфейс вызова функций. В след. примере этот метод использ. в определении класса, который сохраняет декорируемую ф-цию в экземпляре и перехватывает вызовы по оригинальному имени. А т.к. это класс, кроме всего прочего в нем имется возможность хранить инфу о состоянии (счетчик произведенных вызовов):

class tracer:

def \_\_init\_\_(self, func):

self.calls = 0

self.func = func

def \_\_call\_\_(self, \*args):

self.calls += 1

print('call %s to %s' % (self.calls, self.func.\_\_name\_\_))

self.func(\*args)

@tracer # То же, что и spam = tracer(spam)

def spam(a, b, c): # Обертывает spam в объект-декоратор

print(a, b, c)

spam(1, 2, 3) # На самом деле вызывается объект-обертка

spam('a', 'b', 'c') # Т.е. вызывается метод \_\_call\_\_ в классе

spam(4, 5, 6) # Метод \_\_call\_\_ выполняет доп. действия

# и вызывает оригинальную ф-цию

Замечание! Написал название класса с большой буквы, как положено, но при импорте получил ошибку, т.к. имя декоратора @tracer с маленькой буквы. Попробовал написать оба имени с заглавной буквы и всё сработало! Но лучше оставлю с маленькими, все-таки ф-ции принято писать с маленькой, а класс потерпит, хотя в настоящем коде лучше писать с большой буквы, все-таки декоратор это декоратор, а соглашения по классам важнее наверное.

Ф-ция spam передается декоратору tracer, поэтому, когда производится обращение к оригинальному имени spam, в действительности вызывается метод \_\_call\_\_ в классе. Этот метод подсчитывает и регистрирует вызовы, а затем вызывает оригинальную обернутую ф-цию. Обратите внимание, как использ-ся синтаксис аргумента \*name для упаковывания и распаковывания аргументов, передаваемых ф-ции, - благодаря этому данный декоратор может использ-ся для обертывания любой ф-циии, с любым числом позиционных аргументов.

В рез-те к оригинальной ф-ции spam добавляется слой доп. логики. Ниже приводим вывод, первая строка - создана классом tracer, а вторая - ф-цией spam (запускаем импортом модуля сценария import spam):

call 1 to spam

1 2 3

call 2 to spam

a b c

call 3 to spam

4 5 6

Итак, данный декоратор действует как обычная ф-ция, принимающая позиционные аргументы, но он не возвращает результат вызова декорируемой ф-ции, не обрабатывает именованные аргументы и не сможет декорировать методы классов (при декорировании методов, метод \_\_call\_\_ мог бы передавать только экземпляр класса tracer). Как мы узнаем далее, есть много способов декорирования ф-ций, включая влож. инструкции def, - некоторые из них лучше подходят для декорирования методов, чем способ, представленный здесь.

**Декораторы классов и метаклассы.**

В 2.6 и 3.0 теперь можно применять декораторы не только к ф-циям, но и к классам. Декораторы классов похожи на декораторы функций, но они запускаются после инструкции class, чтобы повторно присвоить имя класса вызываемому объекту. Кроме того, они могут использоваться для изменения классов сразу после их создания или добавлять доп. слой логики уже после создания экземпляров.

При применении декоратора к классу программный код вида:

def decorator(aClass): ...

@decorator

class C: ...

отображается в следующем эквиваленте:

def decorator(aClass): ...

class C: ...

C = decorator(C)

Декоратор класса может расширить функциональность самого класса или вернуть объект, который будет перехватывать последующие попытки конструирования экземпляров. Напр. в примере из раздела "Подсчет экземпляров для каждого класса с пом. методов класса" выше, мы могли бы использовать этот прием для автоматич. добавления в классы счетчика экземпляров и любых других необходимых данных:

def count(aClass):

aClass.numInstances = 0

return aClass # Возвращает сам класс, а не обертку

@count

class Spam: ... # То же, что и Spam = count(Spam)

@count

class Sub(Spam): ... # Инструкция numInstances = 0 не нужна здесь

@count

class Other(Spam): ...

Метаклассы представляют собой похожий инструмент на основе классов, область применения которого отчасти перекрывает область применения декораторов классов. Они представляют альтернативную модель управления созданием объектов классов за счет создания подклассов класса type и включения их в инструкцию class:

class Meta(type):

def \_\_new\_\_(meta, classname, supers, classdict): ...

class C(metaclass=Meta): ...

В 2.6 включение метакласса отличается, - вместо именованного аргумента в заголовке инструкции class для этих целей использ-ся атрибут metaclass:

class C:

\_\_metalass\_\_ = Meta

...

Обычно метакласс переопределяет метод \_\_new\_\_ или \_\_init\_\_ класса type, с целью взять на себя управление созданием или инициализацией нового объекта класса. Суть состоит в том, чтобы определить код, который будет вызываться автоматич. на этапе создания класса.

**Типичные проблемы при работе с классами.** стр. 901

Большая часть типичных проблем с классами, сводится к проблемам, связанным с пространствами имен. Некоторые темы, которые мы затронем, явл-ся скорее передовыми приемами использ. классов, чем проблемами.

**Изменение атрибутов класса может приводить к побочным эффектам.**

Теоретически классы (и экземпляры классов) относятся к категории изменяемых объектов. Подобно таким встр. типам, как списки и словари, они могту изменяться непосредственно, путем присваивания значений атрибутам, и, как и со списками и словарями, это означает, что изменение класса или экземпляра может оказывать влияние на множественнеы ссылки на них.

Обычно нам это и нужно (так объекты изменяют своё состояние), но, изменяя атрибуты, об этом необходимо помнить. Все экземпляры класса совместно используют одно и то же пространство имен класса, поэтому любые изменения на уровне класса буду тотражаться на всех экземплярах, если, конечно, они не имеют собственных версий атрибутов класса.

Классы, модули и экземпляры - это всего лишь объекты с пространствами имен атрибутов, поэтому во время выполнения они обычно изменяются с помощью операций присваивания.

Рассмотрим класс, в его теле выполняется присваивание имени а, в рез-те чего создается атрибут Х.а, который во время выполнения располагается в объекте класса и будет унаследован всеми экземплярами класса Х:

>>> class X:

a = 1 # Атрибут класса

>>> I = X()

>>> I.a # Унаследован экземпляром

1

>>> X.a

1

Всё как обычно. Но обратим внимание на динамическое изменение атрибута класса, за пределами инструкции class: это приведет к одновременному изменению атрибутам во всех объектах, наследующих его от класса. Кроме того, новые экземпляры, созданнеы в ходе работы проги, получают динамически установленное значение, а то, что напсиано в исходниках кода:

>>> X.a = 2

>>> I.a

2

>>> J = X()

>>> J.a

2

- но присваивание атрибуту а в экземплярах изменит только сам этот объект экземпляра.

Что это - полезная особенность или оапсная ловушка? Решать вам. Фактич. мы можем вып-ть действия с атрибутами класса без создания экземпляров, - так можно имитировать "записи" и "структуры" данных, имеющиеся в др. ЯП. Пример программы:

>>> class X: pass # Создаем несколько пространств имен атрибутов

>>> class Y: pass

>>> X.a = 1 # Использовать атрибуты класса как переменные

>>> X.b = 2 # В программе нет ни одного экземпляра класса

>>> X.c = 3

>>> Y.a = X.a + X.b + X.c

>>> for X.i in range(Y.a): print(X.i)

0

1

2

3

4

5

- кстати, вывод будет тот же если использовать простую переменную i, а не X.i (это уже мои опыты):

>>> for i in range(Y.a): print(i)

0

1

2

3

4

5

Здесь классы X и Y действуют ка кмодули "без файлов" - пространства имен для хранения переменных, которые не конфликтуют между собой. Это допустимый приём, но не подходщий для применения к классам, написанным другими программистами - т.к. мы точно не знаем какие атрибуты можно изменять безнаказанно. Если вы имитируете структуру на языке С, лучше изменять экземпляры, а не класс, т.к. в этом случае изменения будут касаться единственного объекта:

class Record: pass

X = Record()

X.name = 'bob'

X.job = 'Pizza maker'

**Модификация изменяемых атрибутов класса также может иметь побочные эффекты.**

Эта проблема явл-ся продолжением предыдущей. Атрибуты класса совместно используются всеми его экземплярами, поэтому, если атрибут класса ссылается на изменяемый объект, изменение этого объекта из любого экземпляра отразится сразу на всех экземплярах:

>>> class C:

shared = [] # Атрибут класса

def \_\_init\_\_(self):

self.perobj = [] # Атрибут экземпляра

>>> x = C() # Два экземпляра

>>> y = C() # неявно используют один и тот же атрибут класса

>>> y.shared, y.perobj

([], [])

>>> x.shared.append('spam') # Окажет влияние на объект y также!

>>> x.perobj.append('spam') # Изменит данные, принадлежащие только объекту х

>>> x.shared, x.perobj

(['spam'], ['spam'])

>>> y.shared, y.perobj # В объекте y наблюдаются изменения,

(['spam'], []) # произведенные через объект х

>>> C.shared # Сохраненный в классе и совместно используемый

['spam']

Здесь у нас, разделяемые объекты, которые совместно используются несколькими функциями, объекты уровня модуля, которые совместно используются несколькими импортирующими модулями, и изменяемые аргументы функций, которые совместно используются вызывающим и вызываемым кодом.

Всё это явл-ся разновидностями ситуации наличия ссылок на изменяемый объект - изменения в объекте, выполненные с пом. любой из этих ссылок, можно будет наблюдать с пом. всех остальных. В нашем случае всеми экземплярами совместно используются атрибуты класса, через механизм наследования, но по сути это тот же самый феномен, который может осложняться различными способами присваивания атрибутам экземпляров:

x.shared.append('spam') # Изменит разделяемый объект, присоединенный к классу

x.shared = 'spam' # Изменит или создаст атрибут экземпляра х

Но это все-таки не проблема, это одна из особенностей использования изменяемых объектов в атрибутах классов, о которой следует помнить.

**Множественное наследование: порядок имеет значение.**

Подчеркнем, - порядок, в котором перечислены суперклассы в строке заголовка инструкции class, может иметь критическое значение. В ходе поиска классы просматриваются слева направо, в соответствии следования их в заголовке инструкции.

В примере множ. наследования, в гл. 30, предположим, что класс Super тоже реализует свой метод \_\_str\_\_:

class ListTree:

def \_\_str\_\_(self): ...

class Super:

def \_\_str\_\_(self): ...

class Sub(ListTree, Super): # Будет унаследован метод \_\_str\_\_ класса ListTree,

# так как он стоит в списке первым

x = Sub() # Поиск сначала будет выполняться в классе

# ListTree, а затем в классе Super

От какого класса мы унаследовали бы метод - от класса Lister или Super? Это зависит от того какой класс стоит первым в заголовке класса Sub. Поэтому мы и поставили ListTree первым в списке, т.к. нам нужен именно его метод \_\_str\_\_.

А теперь предположим, что классы Super и ListTree имеют свои версии еще одного одноименного атрибута. Тут уже порядок в заголовке определения подкласса не поможет (иначе у нас поменяются методы \_\_str\_\_), - мы должн вручную переопределить рез-т наследования, явно выполнив присваивание имени атрибута в классе Sub:

class ListTree:

def \_\_str\_\_(self): ...

def other(self): ...

class Super:

def \_\_str\_\_(self): ...

def other(self): ...

class Sub(ListTree, Super): # Унаследует \_\_str\_\_ класса ListTree, т.к. он

# первый в списке

other = Super.other # Явно выбирается версия атрибута из класса Super

def \_\_init\_\_(self):

...

x = Sub() # Поиск сначала вып-ся в Sub и только потом в ListTree/Super

Здесь присваивание атрибуту с именем other в классе Sub создает атрибут Sub.other - ссылку на объект Super.other. А т.к. эта ссылка находится ниже в дереве классов, это не даст механизму наследования выбрат ьверсию атрибута ListTree.other, который был обнаружен первым при обычных обстоятельствах. Так же, если бы класс Super стоял первым в списке, то чтобы атрибут other наследовался обычнным способом, нам могло бы потребоваться явно выбрать методы класса ListTree:

class Sub(Super, ListTree): # Получить Super.other по наследованию

\_\_str\_\_ = ListTree.\_\_str\_\_ # Явно выбрать ListTree.\_\_str\_\_

Множ. наследование достаточно сложная тема, и применять его нужно осторожно и только в крайней необходимости.

Множ. наследование дает лучшие рез-ты, когда суперклассы явл-ся максимально автономными, а псевдочастные атрибуты с именами вида \_\_Х могут помочь в локализации имен, чтобы ограничить конфликты имен в суперклассах.

**Методы, классы и вложенные области видимости.**

Эта проблема была ликвидирована в Питон 2.2 введением областей видимости влож. ф-ций. Эта тема для ознакомления, увидим, что происходит в случае вложения ф-ций, когда один из уровней вложенности явл-ся классом.

Классы, как и ф-ции, обладают своими областями видимости, которые обладают сходными проявлениями в теле class. Еще и методы, по сути, явл-ся влож. ф-циями. Похоже, что путаница часто возникает, когда имеются классы, вложенные друг в друга.

Пример, ф-ция generate возвр. экземпляр влож. класса Spam. Внутри ф-ции имя класса Spam нах-ся в локал. области ф-ции generate. Но в Питоне до 2.2 внутри метода method имя класса Spam недоступно - method имеет доступ только к своей локал. области видимости, к области модуля, вмещающего ф-цию generate и к встр. именам:

def generate(): # Не работает в питон ниже 2.2

class Spam:

count = 1

def method(self): # Имя Spam недоступно:

print(Spam.count) # Не локальное (def), не глобальное (модуль),

# не встроенное

return Spam()

generate().method()

Если запустим в Питон ниже 2.2 получим ошибку:

... текст сообщения об ошибке опущен...

print(Spam.count) # Не локальное (def), не глобальное (модуль)

NameError: Spam # не встроенное

В версиях 2.2 и выше будет работать, т.к. все локал. области вмещающих ф-ций автоматич. видимы для влож. ф-ций (включая и влож. методы).

И даже в версии 2.2 методам недоступна локал. область видимости вмещающего класса - им доступны только области видимости вмещающих ф-ций. Именно поэтому методы должны использ-ть self или имя класса.

Заставить работать наш пример в версиях ниже 2.2 можно. Простой способ - переместить имя Spam в область вмещающего модуля с пом. глобал. объявления, т.к. методу method доступны глоабл. имена в модуле, то ошибки уже не будет:

def generate():

global Spam

class Spam:

count = 1

def method(self):

print(Spam.count)

return Spam()

generate().method()

Еще лучше реструктурировать код так, чтобы вместо global определение класса Spam было наверх. уровне модуля:

def generate():

return Spam()

class Spam:

count = 1

def method(self):

print(Spam.count)

generate().method()

На самом деле, такой подход рекомендуется использ. во всех версиях Питона - код выглядит проще, если в нем отсутствует вложенные классы и ф-ции.

Если нужно что-то сложное и замысловатое, то можно избавиться от ссылки на имя Spam в методе method, используя спец. атрибут \_\_class\_\_, который возвр. класс объекта экземпляра:

def generate():

class Spam:

count = 1

def method(self):

print(self.\_\_class\_\_.count)

return Spam()

generate().method()

**Делегирование в версии 3.0: \_\_getattr\_\_ и встроенные операции.**

Классы, использующие метод \_\_getattr\_\_ для делегирования обернутым объектам операций обращения к атрибутам, будут терпеть неудачу в 3.0, если методы перегрузки не будут переопределены в классе-обертке.

В 2.6 и 3.0 обращения к методам перегрузки производятся встр. операциями неявно, минуя обычную схему выбора методов-обработчиков. Напр. метод \_\_str\_\_, использ. для вывода, никогда не вызывает \_\_getattr\_\_. Вместо это интер-р в 3.0 пытается отыскать требуемые имена в классах, припуская этап поиска в экземпляре. Чтобы решить эту проблему, подобные методы должны быть переопределены в классах-обертках.

**"Многослойное обертывание".**

При грамотном использ. ООП и многократного использ. кода можно серьезно скоратить время разработки. Но неправильное использ. понимания абстракции в ООП может осложнить понимание кода. Если классы наслоены слишком глубоко, код становится малопонятным.

Здесь применимо одно из правил языка Питон - не усложняйте решение задачи, если оно не явл-ся таковым. Обертывание кода несколькими слоями классов на грани разумного - всегда плохая идея. Абстракция - это основа полиморфизма и инкапсуляции, и может быть эффективным инструментом. Также сделайте интерфейсы своих классов интутитивно понятными. Избегайте чрезмерной абстракции и сохраняйте иерархии своих классов короткими и плоскими, если нет причин делать иначе.

На заметку, "ООП глазами специалиста".

- Повторное использование программного кода. Это самая простая (и самая основная) причина использования ООП. Наследование в классах позволяет программисту писать программы, адаптируя существующий код, а не писать каждый новый проект с самого начала.

- Инкапсуляция. Сокрытие деталей реализации за интерфейсом объекта предохраняет пользователей класса от необходимости изменять свой программный код.

- Организация. Классы предоставляют новые локал. области видимости, которые минимизируют вероятность конфликтов имен. Кроме того, они обеспечивают место для естественного размещения программного кода реализации и управления состоянием объекта.

- Поддержка. Классы обеспечивают естественное разделение программного кода, что позволяет уменьшить его избыточность. Благодаря организации и возможности повторного использования программного кода в случае необходимости бывает достаточно изменить всего одну копию программного кода.

- Непротиворечивость. Классы и возможность наследования позволяют реализовать общие интерфейсы и, следовательно, обеспечить единообразие вашего кода - такой код легко отлаживать, выглядит осмысленнее и прост в сопровождении.

- Полиморфизм. Это скорее свойство ООП, чем причина его использвоания, но благодаря поддержке общности кода, полиморфизм делает код более гибким, расширяет область его применения и, следовательно, увеличивает его шансы на повторное использование.

- Другие. И, конечно, причина номер один состоит в том, что упоминание о владении приемами ООП увеличивает шанс быть принятым на работу! (это конечно шутка, но знание ООП в программировании будет не лишним).

И, конечно же, вы не оцените ООП пока не будете использовать его какое-то время. Выберите себе проект, изучите больише примеры, поработайте над упражнениями - будет сложно, но рез-т вам понравится.

**Упражнения к шестой части.** стр. 909