Продвинутый курс Python

Урок 3

ООП (продолжение)

Множественное наследование, порядок разрешения методов, объект super. Атрибут __slots__.

Дескрипторы атрибутов, доступ к атрибутам.

Метаклассы. Декораторы классов.

Абстрактные классы, модуль abc.

Сетевое программирование, модуль socketserver.

Введение

Данный урок посвящен продолжению работы с ООП в Python. Будут рассмотрены такие вопросы как множественное наследование, дескрипторы атрибутов, метаклассы. Подразумевается, что слушатель уже знаком с базовыми возможностями работы с классами в Python; такие понятия как «класс», «атрибут», «метод», «наследование», «инкапсуляция», «интерфейс класса», «родительский/дочерний класс» должны быть хотя бы просто «на слуху» и не должны вызывать страха, т.к. новые темы будут затрагивать более сложные и в то же время более интересные возможности классов в Python.

Также будет продолжено изучение сетевого взаимодействия с использованием языка Python.

Краткий экскурс в ООП-1

Перед тем как приступать к уроку, уделим пару мгновений на то, чтобы вспомнить, что изучалось по теме ООП в курсе «Python-1»:

- класс - «абстракция», экземпляр - конкретный «представитель» абстракции;
- атрибут - (как правило) данные класса/экземпляра, метод - функция класса;
- интерфейс - способ «общения» с данным классом (например, «интерфейс итератора»);
- классы бывают классические и «нового стиля». В Python 3 все классы - «нового стиля»;
- тело класса выполняется при первом чтении файла интерпретатором;
- для методов класса существуют специальные декораторы: `@property`, `@classmethod`, `@staticmethod`;
- свойство - вычисляемый атрибут (метод, обёрнутый декоратором `@property`);
- функции `getattr`, `setattr`, `hasattr`;
- в ООП-подходе нередко используют устоявшиеся подходы к решению задач проектирования, такие подходы получили название **шаблонов** (паттернов) проектирования (в курсе Python-1 слушатели знакомились с шаблонами **Строитель**, **Делегирование**, **Фабрика**).

Напомним также, что у свойств могут быть свои атрибуты - `setter`, `getter`, `deleter`:

```
class C:
    def __init__(self):
        self._x = None

    @property
    def x(self):
        """I'm the 'x' property."""
        return self._x

    @x.setter
    def x(self, value):
        self._x = value

    @x.deleter
    def x(self):
        del self._x
```

То же самое может быть записано по-другому:

```
class C:
    def _get_x(self):
        """I'm the 'x' property."""
        return self._x

    def _set_x(self, value):
        self._x = value

    def _del_x(self):
        del self._x

    x = property(_get_x, _set_x, _del_x)
```

Множественное наследование

Напомним, **наследование** – это механизм создания новых классов, призванный настроить или изменить поведение существующего класса. Оригинальный класс называют **базовым классом** или суперклассом. Новый класс называют **производным классом** или подклассом. Когда новый класс создается с использованием механизма наследования, он «наследует» атрибуты базовых классов. Однако производный класс может переопределить любой из этих атрибутов и добавить новые атрибуты.

Наследование определяется перечислением в инструкции class имен базовых классов через запятые. В случае отсутствия подходящего базового класса определяется наследование класса `object`. **`object`** – это класс, который является родоначальником всех объектов в языке Python и предоставляет реализацию по умолчанию некоторых общих методов, таких как `__str__()`.

В языке Python поддерживается **множественное наследование**. Это достигается за счет указания нескольких базовых классов при описании класса. Рассмотрим следующую коллекцию классов:

```
import random

class MusicGenre:
    ''' Базовый класс жанра для музыкальной библиотеки.
        Определяет название жанра и его рейтинг в библиотеке.
    '''
    genre_count = 0
    def __init__(self, name, rating):
        self._name = name
        self._rating = rating
        MusicGenre.genre_count += 1
    def __del__(self):
        MusicGenre.genre_count -= 1
    def rate_up(self, point):
        self._rating += point
    def rate_down(self, point):
        self._rating -= point
    def rating(self):
        return self._rating
```

```
class DrumAndBass(MusicGenre):
    def rating(self):
        # "Случайная" накрутка рейтинга
        if random.randint(0, 4) == 1:
            return self._rating + 1
        else:
            return self._rating

group = DrumAndBass("Pendulum", 65.7)
group.rate_up(5.5)
print("Рейтинг исполнителя {}: {}".format(group._name, group.rating()))

class PromoConcert:
    ''' Промо-концерт - улучшает рейтинг исполнителя
    '''
    points = 5.00
    def go_up(self):
        self.rate_up(self.points)

class Gossip:
    ''' Сплетни, слухи - снижают рейтинг
    '''
    points = 2.70
    def go_down(self):
        self.rate_down(self.points)

# Класс, использующий механизм множественного наследования
class SuperDrums(DrumAndBass, PromoConcert, Gossip):
    def rate_up(self, points):
        super().rate_up(points)

    def rate_down(self, points):
        super().rate_down(points)
```

При использовании множественного наследования порядок поиска атрибутов становится более сложным, потому что появляется несколько возможных путей поиска. Следующие инструкции иллюстрируют эту сложность:

```
artist = SuperDrums("RonWellsJS", 90.00)
artist.go_up()              # <<-- Чему будет равно points ?
artist.go_down()            # <<-- Чему будет равно points ?
```

В этом примере методы go_up() и go_down() имеют уникальные имена и обнаруживаются в соответствующих базовых классах. Однако создается ощущение, что метод go_down() работает с ошибкой, потому что не используют значение атрибута points, инициализированного в его классе. Это обусловлено тем, что атрибут points – это переменная класса, объявленная в двух различных базовых классах. В работе используется одно из этих значений, но какое? (Подсказка: это значение атрибута PromoConcert.points).

Чтобы обеспечить поиск атрибутов при множественном наследовании, все базовые классы включаются в список, в порядке от «более специализированных» к «менее специализированным». Затем, когда

производится поиск, интерпретатор просматривает этот список, пока не найдет первое определение атрибута.

В примере выше класс DrumAndBass является более специализированным, чем класс MusicGenre, потому что он наследует класс MusicGenre. То же относится и к классу SuperDrums, PromoConcert считается более специализированным, чем Gossip, потому что он стоит первым в списке базовых классов. Порядок поиска в базовых классах можно увидеть, если вывести содержимое атрибута __mro__ класса. Например:

```
>>> SuperDrums.__mro__
(<class '__main__.SuperDrums'>,
 <class '__main__.DrumAndBass'>,
 <class '__main__.MusicGenre'>,
 <class '__main__.PromoConcert'>,
 <class '__main__.Gossip'>,
 <class 'object'>)
>>>
```

Проблемы составления линеаризации

В большинстве случаев правила составления линеаризации «интуитивно понятны». То есть производный класс всегда проверяется раньше его базовых классов, а если класс имеет несколько родителей, они всегда будут проверяться в том порядке, в каком были перечислены в объявлении класса. Однако точный порядок просмотра базовых классов в действительности намного сложнее и его нельзя описать с помощью какого-либо «простого» алгоритма, такого как поиск «снизу-вверх» или «слева-направо». Для упорядочения используется алгоритм **C3-линеаризации**, описанный в документе **«A Monotonic Superclass Linearization for Dylan»** (Монотонная линеаризация суперкласса для языка Dylan) (авторы К. Баррет (K. Barrett) и другие, был представлен на конференции OOPSLA'96). Одна из малозаметных особенностей этого алгоритма состоит в том, что его реализация в языке Python препятствует созданию определенных иерархий классов с возбуждением исключения TypeError. Например:

```
class X(object): pass
class Y(X): pass
class Z(X, Y): pass     # TypeError.
                        # Невозможно определить непротиворечивый порядок
                        # разрешения имен методов
```

В данном случае алгоритм разрешения имен методов препятствует созданию Z, потому что он не может определить осмысленный порядок поиска в базовых классах. Например, класс X находится в списке родительских классов перед классом Y, поэтому он должен просматриваться первым. Однако класс Y является более специализированным, потому что наследует класс X. Поэтому если первым будет проверяться класс X, это не позволит отыскать специализированные реализации методов в классе Y, унаследованных от класса X. На практике такие ситуации должны возникать очень редко, и если возникают, это обычно свидетельствует о более серьезных ошибках, допущенных при проектировании программы.

Методы родительских классов

Множественное наследование порождает ещё одну специфическую проблему - обращение к родительским методам в методах дочерних классов. Это можно выполнять разными способами. Рассмотрим пример:

```
class C(B, A):
    def __init__(self):
        # something
        B.__init__(self)
        A.__init__(self)
```

Во-первых, происходит явное обращение к родительским классам - это нехорошо с точки зрения поддержки и развития кода - при необходимости заменить кого-то из классов-предков на другой класс или вообще убрать его, придётся изменять все функции, которые к нему обращались. Во-вторых, в коде выше ничего неизвестно о классах A и B. Возможно, у них есть общие предки, к которым они обращаются аналогичным образом:

```
class A(P1, P2):
    def __init__(self):
        # something
        P1.__init__(self)
        P2.__init__(self)

class B(P1, P2):
    def __init__(self):
        # something
        P1.__init__(self)
        P2.__init__(self)
```

Если это так, то получится, что инициализация общих предков отработает два раза. Это неправильно. Чтобы этого избежать в Python есть класс **`super`**. В версии Python 3 к нему можно обращаться следующим образом:

```
class C(B, A):
    def __init__(self):
        # something
        super().__init__()    # Прототип: super([type[, object-or-type]])
                              # для версий младше 3.0 нужно использовать super(C,
self)
```

Объект класса super запоминает аргументы, переданные ему в момент инициализации (по умолчанию - текущий класс и текущий объект), и при вызове любого метода (super().__init__() в примере выше) проходит по списку линеаризации класса второго аргумента (self.__class__.__mro__), пытаясь вызвать этот метод по очереди для всех классов, следующих за классом в первом аргументе (класс C), передавая в качестве параметра первый аргумент (self). Т.е. для рассматриваемого случая:

```
self.__class__.__mro__ = [C, B, A, P1, P2, ...]
super(C, self).__init__() => B.__init__(self)
super(B, self).__init__() => A.__init__(self)
super(A, self).__init__() => P1.__init__(self)
```

Как видите, из метода `B.__init__` при использовании super будет вызван метод `A.__init__`, хотя класс A с ним никак не связан и не является его предком. В этом случае при необходимости по цепочке однократно отработают методы всех предков.

Благодаря такому подходу можно, например, определить взаимосвязь классов для музыкальной библиотеки, которая будет последовательно проходя по цепочке предков составлять список всех музыкантов, связанных с выбранными музыкальными жанрами. Будет удобно вместо того, чтобы переписывать каждый раз список музыкантов для каждого жанра просто унаследовать его от жанров-предшественников. Тогда список будет формироваться автоматически:

```
class A:
    def genre(self):
        return ['Blues']

    def artist(self):
        return []

class B(A):
    def genre(self):
        return ['Electric blues'] + super().genre()

class C(A):
    def artist(self):
        return ['B.B. King'] + super().artist()

class D(A): pass

class E(B):
    def genre(self):
        return ['Soul blues'] + super().genre()

class F(B):
    def genre(self):
        return ['Blues rock'] + super().genre()

    def artist(self):
        return ['Eric Clapton'] + super().artist()

class G(C, D): pass

class H(E, F, G):
    def genre(self):
        return ['Boogie rock'] + super().genre()


if __name__ == "__main__":
    H = H()

    print('List of artists: ')
    for artist in H.artist():
        print (' - ' + artist)

    print ('List of linked genres: ')
    for genre in H.genre():
        print (' - ' + genre)
```

В результате получится следующее:

```
List of artists:
 - Eric Clapton
 - B.B. King
List of linked genres:
 - Boogie rock
 - Soul blues
 - Blues rock
 - Electric blues
 - Blues
```

Исходя из этого списка, можно понять, каким образом обходился список линеаризации.

Видно, что ни один из родительских методов не был вызван дважды, иначе в списке жанров или музыкантов обнаружились бы повторения. Кроме того обратите внимание, что в самом старшем классе A определены оба метода — genre и artist.

Вызов super() не совершает никаких проверок, то есть при попытке обратиться к несуществующему методу возникнет исключение AttributeError: “super“ object has no attribute “artist“.

Резюме

- Если возможно, то проще обойтись без множественного наследования.
- Множественное наследование удобно для объявления так называемых классов-примесей (Mixins). Класс-примесь, обычно определяющий набор методов, объявляется, чтобы потом «подмешивать» его в другие классы, с целью расширения их функциональных возможностей. Обычно предполагается, что существуют другие методы, и методы в классах-примесях встраиваются поверх них. Эту возможность иллюстрируют классы PromoConcert и Gossip, объявленные в примере выше.

__slots__

Как Вы помните, Python позволяет добавлять атрибуты уже созданному экземпляру пользовательского класса, т.е. код ниже будет вполне легитимным:

```
class A:
    pass

a = A()
a.x = 13

print(a.x)
```

Существует возможность ограничить класс определенным набором имен атрибутов, определив специальную переменную __slots__ (только для классов «нового типа»). Например:

```
class Account(object):
    __slots__ = ('name', 'balance')
    ...

acc = Account()
acc.x = 13

# Traceback (most recent call last):
# File "<pyshell#12>", line 1, in <module>
# a.x = 13
# AttributeError: 'Account' object has no attribute 'x'
```

Если в классе определена переменная __slots__, экземпляры такого класса смогут иметь атрибуты только с указанными именами. В противном случае будет возбуждаться исключение AttributeError. Это ограничение исключает возможность добавления новых атрибутов к существующим экземплярам и решает проблему присваивания значений атрибутам, в именах которых допущена опечатка.

В действительности переменная __slots__ задумывалась совсем не в качестве меры предосторожности. Фактически, это инструмент оптимизации по объему занимаемой памяти и скорости выполнения. Экземпляры классов, где определена переменная __slots__, уже не используют словарь для хранения данных экземпляра. Вместо него используется более компактная структура данных, в основе которой лежит массив. Применение переменной __slots__ в программах, создающих огромное число объектов, может существенно уменьшить объем потребляемой памяти и увеличить скорость выполнения.

Следует заметить, что переменная __slots__ по-особенному воздействует на наследование. Если в базовом классе используется переменная __slots__, производный класс также должен объявлять атрибут __slots__ со списком имен своих атрибутов (даже если он не добавляет новых атрибутов), чтобы иметь возможность использовать преимущества, предоставляемые этой переменной. Если этого не сделать, производный класс будет работать медленнее и занимать памяти даже больше, чем в случае, когда переменная __slots__ не используется ни в одном из классов!

Кроме того, использование переменной __slots__ может нарушить работоспособность программного кода, который ожидает, что экземпляры будут иметь атрибут __dict__. Хотя такая ситуация не характерна для прикладного программного кода, тем не менее вспомогательные библиотеки и другие инструменты поддержки объектов могут использовать словарь __dict__ для отладки, сериализации объектов и выполнения других операций.

Наконец, наличие объявления переменной __slots__ не требует переопределения в классе таких методов, как __getattribute__(), __getattr__() и __setattr__(). По умолчанию эти методы учитывают возможность наличия переменной __slots__. Кроме того, следует подчеркнуть, что нет никакой необходимости добавлять имена методов и свойств в переменную __slots__, так как они хранятся не на уровне экземпляров, а на уровне класса.

Дескрипторы атрибутов

Введение

При использовании свойств (@property) доступ к атрибутам управляется серией пользовательских функций get, set и delete. Такой способ управления атрибутами не вполне универсален, т.к. для каждого однотипного атрибута должен быть свой набор get/set/delete-методов. Более универсальным подходом может быть использование объекта дескриптора. Дескриптор – это обычный объект, представляющий

значение атрибута. За счет реализации одного или более специальных методов __get__(), __set__() и __delete__() он может подменять механизмы доступа к атрибутам и влиять на выполнение этих операций.

Рассмотрим пример дескриптора, контролирующего тип значения для атрибута и препятствующего удалению атрибута из экземпляра объекта:

```
class TypedProperty:
    def __init__(self, name, type_name, default=None):
        self.name = "_" + name
        self.type = type_name
        self.default = default if default else type_name()

    def __get__(self, instance, cls):
        return getattr(instance, self.name, self.default)

    def __set__(self, instance, value):
        if not isinstance(value, self.type):
            raise TypeError("Значение должно быть типа %s" % self.type)
        setattr(instance, self.name, value)

    def __delete__(self, instance):
        raise AttributeError("Невозможно удалить атрибут")

class Foo:
    name = TypedProperty("name", str)
    num = TypedProperty("num", int, 42)
```

В этом примере класс TypedProperty определяет дескриптор, выполняющий проверку типа при присваивании значения атрибуту и возбуждающий исключение при попытке удалить атрибут. Например:

```
f = Foo()
a = f.name           # Неявно вызовет Foo.name.__get__(f, Foo)
f.name = "Гвидо"     # Вызовет Foo.name.__set__(f, "Guido")
del f.name           # Вызовет Foo.name.__delete__(f)
```

Рассмотрим поподробнее некоторые детали.

Протокол дескриптора

Для создания дескриптора нужно реализовать класс с одним или более методом:

- __get__(self, obj, type=None) - должен вернуть значение value;
- __set__(self, obj, value) - возвращает None;
- __delete__(self, obj) - возвращает None.

Типы дескрипторов

Дескрипторы делят на 2 типа:

1. Дескриптор данных (data-descriptor) - реализует метод __set__ (может также иметь методы __get__ и __delete__). Всегда перегружает словарь экземпляра.

2. Простой дескриптор (non-data-descriptor) - не имеет метода __set__ (реализует методы __get__ и/или __delete__). Может быть перегружен через словарь экземпляра.

Хранение значений атрибутов

При работе с дескрипторами атрибутов возникает вопрос о способе хранения значений атрибутов. Возможные пути решения этого вопроса:

1. **Хранить данные в атрибуте объекта дескриптора** - при этом данные будут общими для всех экземпляров классов, использующих этот дескриптор:

```
# Первый способ сохранить данные - просто в атрибуте объекта дескриптора.
class Grade:
    def __init__(self):
        self._value = 0

    def __get__(self, instance, instance_type):
        return self._value

    def __set__(self, instance, value):
        if not (1 <= value <= 5):
            raise ValueError("Оценка должна быть от 1 до 5")
        self._value = value



class Exam():
    ''' Класс Экзамен.
        Для простоты хранит только оценку за экзамен.
    '''
    grade = Grade()

# Но не стоит забывать, что при таком подходе
# данные будут сохранены на уровне атрибута класса Экзамен!!!
# Т.е. будут общими для всех экземпляров класса Экзамен.

# Для демонстрации создадим два Экзамена:
math_exam = Exam()
math_exam.grade = 3
language_exam = Exam()
language_exam.grade = 5

print("  Проверим результаты: ")
print("Первый экзамен ", math_exam.grade, " - верно?")
print("Второй экзамен ", language_exam.grade, " - верно?")
print('Потому что... ')
print('math_exam.grade is language_exam.grade =', math_exam.grade is
language_exam.grade)
```

2. **Хранить данные в отдельном словаре объекта дескриптора** - ключом будет служить сам объект внешнего класса:

```
class Grade:
    def __init__(self):
        self._values = {}

    def __get__(self, instance, instance_type):
        if instance is None: return self
        return self._values.get(instance, 0)

    def __set__(self, instance, value):
        if not (1 <= value <= 5):
            raise ValueError("Оценка должна быть от 1 до 5")
        self._values[instance] = value
```

Хотя данное решение достаточно простое и полноценно работает, оно будет приводить к утечкам памяти! Словарь _values будет хранить ссылку на каждый внешний экземпляр класса, который когда-либо передавался в метод __set__. Это приведет к тому, что счётчик ссылок у внешних экземпляров никогда не будет равен нулю, и сборщик мусора никогда не выполнит свою работу.

Для данного решения вместо обычного dict нужно использовать класс weakref.WeakKeyDictionary:

```
from weakref import WeakKeyDictionary
class Grade:
    def __init__(self):
        self._values = WeakKeyDictionary()
        ...
```

Модуль weakref обеспечивает поддержку слабых ссылок. В обычном случае сохранение ссылки на объект приводит к увеличению счетчика ссылок, что препятствует уничтожению объекта, пока значение счетчика не достигнет нуля. Слабая ссылка позволяет обращаться к объекту, не увеличивая его счетчик ссылок.

Класс WeakKeyDictionary([dict]) создает словарь, в котором ключи представлены слабыми ссылками. Когда количество обычных ссылок на объект ключа становится равным нулю, соответствующий элемент словаря автоматически удаляется. В необязательном аргументе dict передается словарь, элементы которого добавляются в возвращаемый объект типа WeakKeyDictionary. Слабые ссылки могут создаваться только для объектов определенных типов, поэтому существует большое число ограничений на допустимые типы объектов ключей. В частности, встроенные строки НЕ МОГУТ использоваться в качестве ключей со слабыми ссылками. Однако экземпляры пользовательских классов, объявляющих метод __hash__(), могут играть роль ключей. Экземпляры класса WeakKeyDictionary имеют два дополнительных метода, iterkeyrefs() и keyrefs(), которые возвращают слабые ссылки на ключи.

Данное решение всё же содержит незначительное ограничение - в одном внешнем классе нельзя сохранять данные дескрипторов одного типа (например, Экзамен с несколькими Оценками).

3. **Хранить данные в отдельном атрибуте внешнего класса** - требуется только определить способ именования атрибута. Такой подход, помимо прочего, позволяет в одном внешнем классе создавать несколько объектов-дескрипторов одного класса:

```
class Grade:
    def __init__(self, name):
        # Для данного подхода необходимо сформировать отдельное имя атрибута
        self.name = '_' + name

    def __get__(self, instance, instance_type):
        if instance is None:
            return self
        return "*{}*".format(getattr(instance, self.name))

    def __set__(self, instance, value):
        if not (1 <= value <= 100):
            raise ValueError("Балл ЕГЭ должен быть от 1 до 100")
        setattr(instance, self.name, value)

class ExamEGE():
    ''' Комплексный экзамен, на котором оцениваются разные критерии. '''
    # Для обновлённого Grade нужно добавить строковые имена.
    math_grade = Grade('math_grade')
    writing_grade = Grade('writing_grade')
    science_grade = Grade('science')

# Проверим обновлённый дескриптор Оценку и объекты Экзамены.
first_exam = ExamEGE()
first_exam.writing_grade = 3
first_exam.math_grade = 4
print("Содержимое first_exam.__dict__:")
print(' ', first_exam.__dict__)
second_exam = ExamEGE()
second_exam.writing_grade = 2
second_exam.science_grade = 5
print("  Проверим результаты: ")
print("Первый экзамен ", first_exam.writing_grade, first_exam.math_grade,
      " - верно?")
print("Второй экзамен ", second_exam.writing_grade, second_exam.science_grade, "
- верно?")
```

Поиск атрибутов

Действия чтения, записи и удаления атрибута Python будет выполнять по-разному.

Чтение атрибута

При попытке получить значение атрибута (print(obj.attr)) выполняются следующие действия:

1. Если attr - это специальный атрибут (на уровне Python’а), вернуть его.
2. Существует ли attr в obj.__class__.__dict__ (т.е. obj.__class__.__dict__[“attr”])?
    - Если да и это дескриптор данных, вернуть результат работы дескриптора (результат метода __get__ дескриптора).

- Выполнить аналогичную проверку во всех базовых классах obj.__class__.

3. Существует ли attr в obj.__dict__?
    - Если да, вернуть это значение.
    - Если obj - это класс, то выполнить проверку во всех его базовых классах:
        - если в этом классе или его базовых классах существует дескриптор, вернуть его результат дескриптора.
4. Существует ли attr в obj.__class__.__dict__?
    - Если существует и это не дескриптор данных (non-data), вернуть результат дескриптора;
    - Если существует и это не дескриптор, просто вернуть его;
    - Выполнить аналогичную проверку для всех базовых классов obj.__class__.
5. Создать исключение AttributeError.

Запись атрибута

Установка значения для атрибутов (obj.attr = data) выполняется проще чтения:

1. Существует ли attr в obj.__class__.__dict__?
    - Если да и это дескриптор данных, использовать дескриптор для установки значения (метод __set__ дескриптора).
    - Выполнить такую же проверку во всех базовых классах obj.__class__.
2. Добавить значение data для ключа attr в словарь obj.__dict__ (т.е. obj.__dict__[“attr”] = data).

Удаление атрибута

Удаление атрибута (del obj.attr) выполняется аналогично записи:

1. Существует ли attr в obj.__class__.__dict__?
    - Если да и это дескриптор данных, использовать дескриптор для удаления атрибута (метод __delete__ дескриптора);
    - Выполнить такую же проверку во всех базовых классах obj.__class__.
2. Добавить значение data для ключа attr в словарь obj.__dict__ (т.е. obj.__dict__[“attr”] = data).

Доступ к атрибутам

В дополнение к работе с дескрипторами рассмотрим методы, обеспечивающие доступ к атрибутам:

- __getattr__ - вызывается, когда атрибут не найден в словаре __dict__ экземпляра класса;
- __getattribute__ - вызывается всякий раз, когда выполняется доступ к атрибуту объекта, даже если атрибут не существует в словаре __dict__ экземпляра класса; вызывается при обращении к функциям getattr, hasattr;

- __setattr__ - вызывается всякий раз, когда атрибут назначается экземпляру класса (в т.ч. при обращении к функции setattr).

Приведём примеры, демонстрирующие работу данных методов:

```
# -----  __getattr__ + __getattribute__
class ValidatingDB:
    def __init__(self):
        self.exists = 5

    def __getattr__(self, name):
        print(' ValidatingDB.__getattr__(%s)' % name)
        value = 'Super %s' % name
        setattr(self, name, value)
        return value

    def __getattribute__(self, name):
        print(' ValidatingDB.__getattribute__(%s)' % name)
        return super().__getattribute__(name)

data = ValidatingDB()
print('Атрибут exists:', data.exists)
print('Атрибут foo: ', data.foo)
print('Снова атрибут foo: ', data.foo)
print('Есть ли атрибут zoom в объекте:', hasattr(data, 'zoom'))
print('Атрибут face в объекте, доступ через getattr:', getattr(data, 'face'))

# Использование метода __setattr__
class SavingDB:
    def __setattr__(self, name, value):
        print(' SavingDB.__setattr__(%s, %r)' % (name, value))
        # Сохранение данных в БД
        # ...
        super().__setattr__(name, value)

data = SavingDB()
print('data.__dict__ до установки атрибута: ', data.__dict__)
data.foo = 5
print('data.__dict__ после установки атрибута: ', data.__dict__)
data.foo = 7
print('data.__dict__ в итоге:', data.__dict__)
```

При реализации методов __getattribute__ и __setattr__ можно столкнуться с ситуацией рекурсии, когда методы вызываются при каждом обращении к атрибуту объекта, и в итоге Python исчерпывает стек вызовов и прерывает свою работу:

```
class BrokenDictionaryDB(object):
    def __init__(self, data):
        self._data = {}

    def __getattribute__(self, name):
        print('Called __getattribute__(%s)' % name)
        return self._data[name]

data = BrokenDictionaryDB({'foo': 3})
print(data.foo)
```

Загвоздка в том, что метод __getattribute__ обращается к self._data, что снова приводит к вызову __getattribute__, который снова обращается к self._data и так далее, пока не остановится работа интерпретатора.

Для решения этой проблемы внутри методов __getattribute__ и __setattr__ необходимо обращаться к атрибуту объекта через объект super, т.е. super().__getattribute__ или super().__setattr__:

```
class DictionaryDB(object):
    def __init__(self, data):
        self._data = data

    def __getattribute__(self, name):
        data_dict = super().__getattribute__('_data')
        return data_dict[name]

data = BrokenDictionaryDB({'foo': 'This is the right way!'})
print(data.foo)
```

Резюме

- Экземпляры дескрипторов создаются только на уровне класса (ни для экземпляра в отдельности, внутри метода __init__() или в других методах).
- Имя атрибута-дескриптора в классе, имеет более высокий приоритет перед другими атрибутами на уровне экземпляров.
- Следует понимать разницу между дескриптором данных и простым дескриптором.
- Сначала вызываются методы __getattribute__()/ __setattr__(), потом уже методы __get__()/__set__ дескриптора.
- Следует избегать бесконечной рекурсии при реализации методов __getattribute__ и __setattr__.

Метаклассы

[Metaclasses] are deeper magic than 99% of users should ever worry about. If you wonder whether you need them, you don't (the people who actually need them know with certainty that they need them, and don't need an explanation about why).
— Tim Peters
inventor of the timsort algorithm and prolific Python contributor

Многим известно это высказывание Тима Петерса и под влиянием этой фразы некоторые программисты Python считают, что на изучение метаклассов не стоит тратить времени. Несмотря на это, рекомендуем Вам уделить внимание и приложить усилия к изучению темы метаклассов - как минимум, Вы будете лучше разбираться в технологиях, которыми пользуетесь, как максимум, Вы сможете применить полученные сведения для создания собственного фреймворка.

Знакомство с метаклассами

Когда программа на языке Python объявляет класс, само определение этого класса становится объектом. Например:

```
class Foo(object):
    pass

isinstance(Foo, object)          # Вернет True
```

Если задуматься над этим примером, можно понять, что что-то должно было создать объект Foo. Созданием объектов такого рода управляет специальный объект, который называется метаклассом. Метакласс – это объект, который знает, как создавать классы и управлять ими.

В предыдущем примере метаклассом, под управлением которого создается объект Foo, является класс с именем type(). Если попытаться вывести тип объекта Foo, можно увидеть, что он имеет тип type:

```
>>> type(Foo)
<class 'type'>
```

Когда с помощью инструкции class определяется новый класс, выполняется определенный набор действий. Во-первых, тело класса выполняется интерпретатором, как последовательность инструкций, с использованием отдельного частного словаря. Инструкции выполняются точно так же, как и в обычном программном коде, кроме того, что дополнительно производится изменение имен частных членов класса (начинающихся с префикса __). В заключение имя класса, список базовых классов и словарь передаются конструктору метакласса, который создает соответствующий объект класса. Следующий пример демонстрирует, как это делается:

```
class_name = "Foo"          # Имя класса
class_parents = (object, ) # Базовые классы
class_body = """            # Тело класса
    def __init__(self, x):
        self.x = x
    def blah(self):
        print("Hello World")
"""
class_dict = { }
# Выполнить тело класса с использованием локального словаря class_dict
exec(class_body, globals(), class_dict)

# Создать объект класса Foo
Foo = type(class_name,class_parents,class_dict)
```

Заключительный этап создания класса, когда вызывается метакласс type(), можно изменить. Повлиять на события, происходящие на заключительном этапе определения класса, можно несколькими способами. Например, в классе можно явно указать его метакласс, добавив именованный аргумент metaclass в кортеж с именами базовых классов (в Python 3) (установив переменную класса __metaclass__ в Python 2):

```
class Foo(metaclass=type):
    pass
    ...
```

Если метакласс явно не указан, инструкция class проверит первый элемент в кортеже базовых классов (если таковой имеется). В этом случае метаклассом будет тип первого базового класса. То есть инструкция class Foo(object): pass создаст объект Foo того же типа, которому принадлежит класс object.

Если базовые классы не указаны, инструкция class проверит наличие аргумента с именем metaclass. Если такой аргумент присутствует, он будет использоваться при создании классов. С помощью этого аргумента можно управлять созданием классов.

Если аргумент metaclass не будет найден, интерпретатор будет использовать метакласс по умолчанию. В Python 3 метаклассом по умолчанию является type.

Методы __new__, __init__, __call__

В основном метаклассы используются во фреймворках, когда требуется более полный контроль над определениями пользовательских объектов. Когда определяется нестандартный метакласс, он обычно наследует класс type и переопределяет такие методы, как __init__() или __new__(). Ниже приводится пример метакласса, который требует, чтобы все методы снабжались строками документирования:

```
class DocMeta(type):
    def __init__(self, clsname, bases, clsdict):
        for key, value in clsdict.items():
            # Пропустить специальные и частные методы
            if key.startswith("__"): continue

            # Пропустить любые невызываемые объекты
            if not hasattr(value, "__call__"): continue

            # Проверить наличие строки документирования
            if not getattr(value, "__doc__"):
                raise TypeError("%s must have a docstring" % key)

        type.__init__(self, clsname, bases, clsdict)
```

В этом метаклассе реализован метод __init__(), который проверяет содержимое словаря класса. Он отыскивает в словаре методы и проверяет, имеют ли они строки документирования. Если в каком-либо методе строка документирования отсутствует, возбуждается исключение TypeError. В противном случае для инициализации класса вызывается реализация метода type.__init__().

Чтобы воспользоваться этим метаклассом, класс должен явно выбрать его. Обычно для этой цели сначала определяется базовый класс, такой, как показано ниже:

```
class Documented(metaclass=DocMeta):
    pass
```

А затем этот базовый класс используется, как родоначальник всех объектов, которые должны включать в себя описание. Например:

```
class Foo(Documented):
    def spam(self, a, b):
        ''' Метод spam делает кое-что '''
        pass
    def boo(self):
        print('A little problem')
```

Этот пример иллюстрирует одно из основных применений метаклассов, состоящий в проверке и сборе информации об определениях классов. Метакласс ничего не изменяет в создаваемом классе, он просто выполняет некоторые дополнительные проверки.

В более сложных случаях перед тем, как создать класс, метакласс может не только проверять, но и изменять содержимое его определения. Если предполагается вносить какие-либо изменения, необходимо переопределить метод __new__(), который выполняется перед созданием класса. Этот приём часто объединяется с приёмом обертывания атрибутов дескрипторами или свойствами, потому что это единственный способ получить имена, использованные в классе. В качестве примера ниже приводится модифицированная версия дескриптора TypedProperty_v2, который был реализован в теме «Дескрипторы»:

```
class TypedProperty_v2:
    ''' Дескриптор атрибутов, контролирующий принадлежность указанному типу '''
    def __init__(self, type_name, default=None):
        self.name = None
        self.type = type_name
        if default: self.default = default
        else: self.default = type_name()

    def __get__(self, instance, cls):
        return getattr(instance, self.name, self.default)
    def __set__(self, instance, value):
        if not isinstance(value, self.type):
            raise TypeError("Значение должно быть типа %s" % self.type)
        setattr(instance, self.name, value)
    def __delete__(self, instance):
        raise AttributeError("Невозможно удалить атрибут")
```

В данном примере атрибуту name дескриптора просто присваивается значение None. Заполнение этого атрибута будет поручено метаклассу. Например:

```
class TypedMeta(type):
    def __new__(cls, clsname, bases, clsdict):
        slots = []
        for key, value in clsdict.items():
            if isinstance(value, TypedProperty_v2):
                value.name = "_" + key
            slots.append(value.name)
        clsdict['__slots__'] = slots
        return type.__new__(cls, clsname, bases, clsdict)

class Typed(metaclass=TypedMeta):
    ''' Базовый класс для объектов, определяемых пользователем '''
    pass
```

В этом примере метакласс просматривает словарь класса с целью отыскать экземпляры класса TypedProperty_v2. В случае обнаружения такого экземпляра он устанавливает значение атрибута name и добавляет его в список имен slots. После этого в словарь класса добавляется атрибут __slots__ и вызывается метод __new__() метакласса type, который создает объект класса. Ниже приводится пример использования нового метакласса:

```
class Foo(Typed):
    name = TypedProperty_v2(str)
    num = TypedProperty_v2(int, 42)
```

Реализация метода __call__() в метаклассе позволяет управлять классом в момент создания экземпляра класса (по аналогии обращения к имени класса как к функции). Результатом работы метода __call__ метакласса должен быть экземпляр пользовательского класса.

Используя возможности метаклассов с использованием метода __call__, можно интересно решить задачу по реализации шаблона «Одиночка» (Singleton - для класса может быть создан только один экземпляр, все новые вызовы конструктора объекта будут возвращать созданный ранее экземпляр):

```
class Singleton(type):
    def __init__(self, *args, **kwargs):
        # У каждого подконтрольного класса будет атрибут __instance,
        # который будет хранить ссылку на созданный экземпляр класса
        self.__instance = None
        super().__init__(*args, **kwargs)

    def __call__(self, *args, **kwargs):
        if self.__instance is None:
            # Если ещё не создан ни один экземпляр класса, то создаём его
            self.__instance = super().__call__(*args, **kwargs)
            return self.__instance
        else:
            # Если уже есть экземпляр класса, то возвращаем его
            return self.__instance

class A(metaclass=Singleton):
    def __init__(self):
        print('Class A')

class B(metaclass=Singleton):
    def __init__(self):
        print('Class B')

# Создадим несколько экземпляров каждого класса и проверим их на идентичность
a_1 = A()
a_2 = A()
b_1 = B()
b_2 = B()

print('a_1 is a_2 - ', a_1 is a_2)
print('b_1 is b_2 - ', b_1 is b_2)
print('a_1 is b_1 - ', a_1 is b_1)
print('a_2 is b_2 - ', a_2 is b_2)
```

Метод __prepare__

В дополнение к методам __new__, __init__ и __call__ в Python 3 для метаклассов был добавлен метод __prepare__. Этот специальный метод относится только к метаклассам и обязан быть методом класса (т. е. должен быть снабжен декоратором @classmethod). Интерпретатор вызывает метод __prepare__ до метода __new__, чтобы тот создал отображение (словарь), которое будет заполнено атрибутами из тела класса. Первым аргументом __prepare__ получает сам метакласс, а за ним имя конструируемого класса и кортеж его базовых классов, а вернуть он должен отображение, которое будет передано в последнем аргументе методу __new__ и далее методу __init__, когда метакласс примется за построение нового класса.

Используя метод __prepare__ можно, например, использовать упорядоченный словарь (collections.OrderedDict) вместо обычного словаря для отображения атрибутов класса (полный код примера содержится в файле meta_prepare.py в примерах к уроку):

```
import collections

class EntityMeta(type):
    """ Метакласс для прикладных классов с контролируемыми полями """
    @classmethod
    def __prepare__(cls, name, bases):
        # Атрибуты класса будут теперь храниться в экземпляре OrderedDict
        return collections.OrderedDict()

    def __init__(cls, name, bases, attr_dict):
        super().__init__(name, bases, attr_dict)
        cls._field_names = []    # Атрибут _field_names создаётся в
конструируемом классе
        for key, attr in attr_dict.items():
            if isinstance(attr, TypedProperty_v2):
                type_name = type(attr).__name__
                attr.name = '_{}_{}'.format(type_name, key)
                cls._field_names.append((key, attr.name))

class Entity(metaclass=EntityMeta):
    """ Прикладной класс с контролируемыми полями """
    @classmethod
    def field_names(cls):
        ''' Просто возвращает поля в порядке добавления '''
        for name in cls._field_names:
            yield name
```

После простых модификаций, показанных в примере, можно обойти поля типа TypedProperty_v2 любого подкласса Entity, воспользовавшись методом класса field_names:

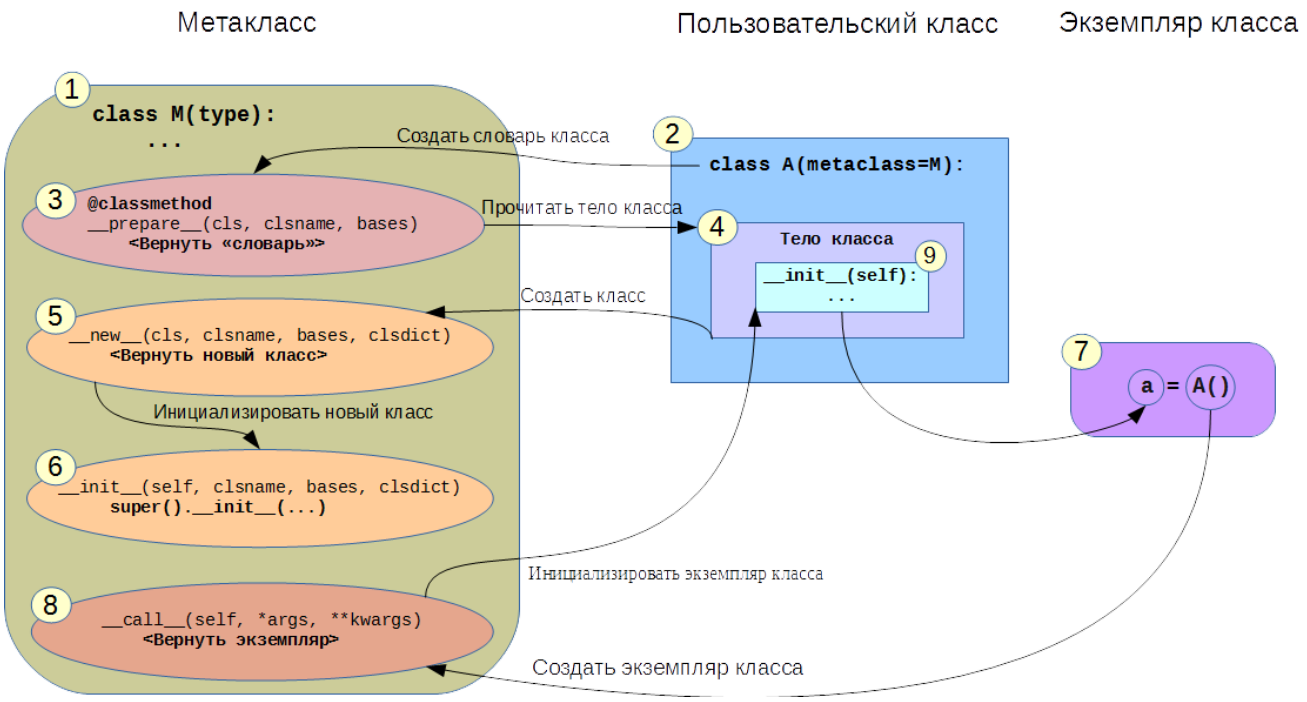
```
class LineItem(Entity):
    description = TypedProperty(str, 'Simple Line')
    weight = TypedProperty(int, 13)
    price = TypedProperty(float, 19.99)

for name in LineItem.field_names():
    print(name)
```

Начиная с Python 3.6 по умолчанию метод __prepare__ возвращает OrderedDict.

Резюме

Метаклассы способны коренным образом изменять поведение и семантику пользовательских классов. Однако не следует злоупотреблять этой возможностью и изменять поведение классов так, чтобы оно существенно отличалось от поведения, описанного в стандартной документации. Пользователи будут обескуражены, если создаваемые ими классы не будут придерживаться обычных правил программирования для классов.

Подводя итоги знакомства с метаклассами, можно привести диаграмму взаимодействия метакласса, класса и экземпляра класса.



* Примеры использования метаклассов

Приведём некоторые примеры использования метаклассов из реальных проектов.

Django

Django - многим уже знакомый свободный фреймворк для веб-приложений на языке Python, использующий шаблон проектирования MVC. Предлагаем часть метода `__new__` метакласса ModelBase из Django 1.8 (данный метакласс достаточно объёмен, заинтересованный слушатель может самостоятельно ознакомиться с кодом данного класса):

```
class ModelBase(type):
    """ Metaclass for all models. """
    def __new__(cls, name, bases, attrs):
        super_new = super(ModelBase, cls).__new__

        # Also ensure initialization is only performed for subclasses of Model
        # (excluding Model class itself).
        parents = [b for b in bases if isinstance(b, ModelBase)]
        if not parents:
            return super_new(cls, name, bases, attrs)

        # Create the class.
        module = attrs.pop('__module__')
        new_class = super_new(cls, name, bases, {'__module__': module})
        attr_meta = attrs.pop('Meta', None)
        abstract = getattr(attr_meta, 'abstract', False)
        if not attr_meta:
            meta = getattr(new_class, 'Meta', None)
        else:
            meta = attr_meta
        base_meta = getattr(new_class, '_meta', None)

        # Look for an application configuration to attach the model to.
        app_config = apps.get_containing_app_config(module)
        ...
    ...
```

SQLAlchemy

SQLAlchemy - это программная библиотека на языке Python для работы с реляционными СУБД с применением технологии ORM. Служит для синхронизации объектов Python и записей реляционной базы данных. SQLAlchemy позволяет описывать структуры баз данных и способы взаимодействия с ними на языке Python без использования SQL (SQLAlchemy будет рассмотрена в теме «Взаимодействие с БД» в данном курсе).

1. Метакласс DeclarativeMeta (создаёт классы относящиеся к sqlalchemy.schema.Table):

```
class DeclarativeMeta(type):
    def __init__(cls, classname, bases, dict_):
        if '_decl_class_registry' not in cls.__dict__:
            _as_declarative(cls, classname, cls.__dict__)
        type.__init__(cls, classname, bases, dict_)

    def __setattr__(cls, key, value):
        _add_attribute(cls, key, value)
```

2. Функция declarative_base() создаёт метакласс (фабрика метаклассов), который в дальнейшем будет создать классы, связанные с таблицами БД:

```
def declarative_base(bind=None, metadata=None, mapper=None, cls=object,
                     name='Base', constructor=_declarative_constructor,
                     class_registry=None,
                     metaclass=DeclarativeMeta):
    lcl_metadata = metadata or MetaData()
    if bind:
        lcl_metadata.bind = bind

    if class_registry is None:
        class_registry = weakref.WeakValueDictionary()

    bases = not isinstance(cls, tuple) and (cls,) or cls
    class_dict = dict(_decl_class_registry=class_registry,
                      metadata=lcl_metadata)
    if isinstance(cls, type):
        class_dict['__doc__'] = cls.__doc__

    if constructor:
        class_dict['__init__'] = constructor
    if mapper:
        class_dict['__mapper_cls__'] = mapper

    return metaclass(name, bases, class_dict)
```

Scapy

Scapy – интерактивная оболочка и программная библиотека для манипулирования сетевыми пакетами на языке программирования Python (Python 2). Класс Packet_metaclass служит для создания классов различных сетевых пакетов:

```
class Packet_metaclass(type):
    def __new__(cls, name, bases, dct):
        if "fields_desc" in dct: # perform resolution of references to other
packets
            current_fld = dct["fields_desc"]
            resolved_fld = []
            for f in current_fld:
                if isinstance(f, Packet_metaclass): # reference to another
fields_desc
                    for f2 in f.fields_desc:
                        resolved_fld.append(f2)
                else:
                    resolved_fld.append(f)
        else: # look for a field_desc in parent classes
            resolved_fld = None
            for b in bases:
                if hasattr(b,"fields_desc"):
                    resolved_fld = b.fields_desc
                    break
       ...

    def __getattr__(self, attr):
        for k in self.fields_desc:
            if k.name == attr:
                return k
        raise AttributeError(attr)

    def __call__(cls, *args, **kargs):
        if "dispatch_hook" in cls.__dict__:
            cls =  cls.dispatch_hook(*args, **kargs)
        i = cls.__new__(cls, cls.__name__, cls.__bases__, cls.__dict__)
        i.__init__(*args, **kargs)
        return i
```

Kivy

Kivy - кроссплатформенный графический фреймворк на Python, направленный на создание новейших пользовательских интерфейсов даже для приложений, работающих с сенсорными экранами. Приложения, написанные на Kivy, могут работать не только на таких традиционных платформах как Linux, OS X и Windows, но также на Android, iOS и Rapberry Pi.

Метакласс WidgetMetaclass служит для регистрации новых виджетов:

```
class WidgetMetaclass(type):
    '''Metaclass to automatically register new widgets for the
    :class:`~kivy.factory.Factory`.
    .. warning::
        This metaclass is used by the Widget. Do not use it directly!
    '''
    def __init__(mcs, name, bases, attrs):
        super(WidgetMetaclass, mcs).__init__(name, bases, attrs)
        Factory.register(name, cls=mcs)

#: Base class used for Widget, that inherits from :class:`EventDispatcher`
WidgetBase = WidgetMetaclass('WidgetBase', (EventDispatcher, ), {})
```

Декораторы классов

Ранее мы рассмотрели, как с помощью метаклассов можно управлять процессом создания классов. Однако иногда бывает достаточно выполнить некоторые действия уже после того, как класс будет определен, например, добавить класс в реестр или в базу данных. Альтернативный подход к решению подобных задач заключается в использовании декоратора класса. Декоратор класса – это функция, которая принимает и возвращает класс. Например:

```
registry = { }
def register(cls):
    registry[cls.__clsid__] = cls
    return cls
```

В этом примере функция register отыскивает в классе атрибут __clsid__. Если этот атрибут определен, он используется для добавления класса в словарь, который служит для отображения идентификаторов классов в объекты классов. Эту функцию можно использовать как декоратор, поместив его непосредственно перед определением класса. Например:

```
@register
class Foo(object):
    __clsid__ = "123-456"
    def bar(self):
        pass
```

Здесь синтаксис декоратора был применён, главным образом, ради удобства. Того же самого результата можно добиться другим способом:

```
class Foo(object):
    __clsid__ = "123-456"
    def bar(self):
        pass

register(Foo)        # Зарегистрировать класс
```

Можно до бесконечности придумывать разные ухищрения, которые можно было бы реализовать в функции-декораторе класса, однако, лучше все-таки избегать чрезмерных превращений, таких как создание обертки вокруг класса или переопределение его содержимого.

Этап импорта и этап выполнения

Для успешного метапрограммирования необходимо знать, когда интерпретатор Python обрабатывает каждый блок кода. Существуют термины «этап импорта» и «этап выполнения», но они определены не строго. На этапе импорта интерпретатор производит синтаксический анализ исходного кода модуля сверху вниз за один проход и генерирует исполняемый байт-код. На этом этапе обнаруживаются синтаксические ошибки. Если в локальном кэше __pycache__ существует актуальный рус-файл, то этот этап пропускается, поскольку уже имеется готовый к выполнению байт-код.

Хотя компиляция, в целом, выполняется на этапе импорта, на этой стадии могут происходить и другие вещи, потому что почти каждое предложение в Python является исполняемым в том смысле, что в нем может выполняться пользовательский код, изменяющий состояние программы. В частности, предложение import - не просто объявление, оно еще и выполняет весь код, находящийся на верхнем уровне импортируемого модуля, при первом его импорте в память процесса - при последующих операциях импорта того же модуля используется кэшированный код, так что происходит только связывание имен. Этот верхнеуровневый код может делать все, что угодно, включая такие типичные для «этапа выполнения» действия, как подключение к базе данных. Поэтому граница между «этапом импорта» и «этапом выполнения» размыта: предложение import может активировать любые действия, которые принято считать частью «этапа выполнения».

Понятие «код, находящийся на верхнем уровне» требует уточнения. Интерпретатор выполняет предложение def па верхнем уровне модуль, когда этот модуль импортируется, но что получается в результате? Интерпретатор компилирует тело функции (если данный модуль импортируется впервые) и связывает объект функции с глобальным именем, но он отнюдь не выполняет тело функции. Проще говоря, это означает, что интерпретатор определяет верхнеуровневую функцию на этапе импорта, но выполняет ее тело, только если она будет вызвана на этапе выполнения.

Для классов все выглядит по-другому: на этапе импорта интерпретатор выполняет тело каждого класса, даже классов, вложенных в другие классы. Это означает, что определяются атрибуты и методы класса, а затем строится сам объект класса. В этом смысле тело класса является «верхнеуровневым кодом»: оно выполняется на этапе импорта.

Рассмотрим пример, позволяющий лучше понять каждый из этапов (пример позаимствован из книги Рамальо Лучано «Python. К вершинам мастерства») (см. директорию evaltime в примерах к уроку). Пример состоит из двух скриптов: основного - evaltime.py и вспомогательного - evalsupport.py.

```
# evaltime.py:
# ----------------------- Демонстрация работы интерпретатора ------------------
#                              Основной модуль демонстрации
from evalsupport import deco_alpha
print('<[1]> evaltime module start')

class ClassOne():
    print('<[2]> ClassOne body')
    def __init__(self):
        print('<[3]> ClassOne.__init__')

    def __del__(self):
        print('<[4]> ClassOne.__del__')

    def method_x(self):
        print('<[5]> ClassOne.method_x')
class ClassTwo(object) :
    print('<[6]> ClassTwo body')

@deco_alpha
class ClassThree():
    print('<[7]> ClassThree body')
    def method_y (self):
        print('<[8]> ClassThree.method_y')

class ClassFour(ClassThree):
    print('<[9]> ClassFour body')
    def method_y (self ):
        print( '< [10] > ClassFour .method_y' )

if __name__ == '__main__':
    print ('<[11]> ClassOne tests', 30 * '.')
    one = ClassOne()
    one.method_x()
    print ('<[12] > ClassThree tests', 30 * '.')
    three = ClassThree()
    three.method_y()
    print('<[13]> ClassFour tests', 30 * '.')
    four = ClassFour()
    four.method_y()
```

```
# evalsupport.py:
# ---------------------- Демонстрация работы интерпретатора ------------------
#                            Вспомогательный модуль
print('<[100]> evalsupport module start')
def deco_alpha (cls):
    print ('<[200]> deco_alpha')
    def inner_1 (self ) :
        print ('<[300]> deco_alpha;inner_1 ')
    cls.method_y = inner_1
    return cls

# BEGIN META_ALEPH
class MetaAleph(type):
    print('<[400]> MetaAleph body')
    def __init__(cls, name, bases, dic):
        print( ' <[500]> MetaAleph.__init__')

        def inner_2( self ) :
            print ('<[600]> MetaAleph.__init__.inner_2')

        cls.method_z = inner_2
# END META_ALEPH

print('<[700]> evalsupport module end')
```

Необходимо выполнить 2 разных запуска скрипта evaltime.py:

1. Импортировать модуль, например, в интерактивном режиме >>> import evaltime.
2. Выполнить модуль командой python3 evaltime.py

Несколько замечаний по импортированию модуля:

1. Вся последовательность действий запускается одним лишь предложением import evaltime.
2. Интерпретатор выполняет тело каждого класса в импортированном модуле и в модуле evalsupport, от которого он зависит.
3. Интерпретатор обрабатывает тело декорированного класса еще до вызова присоединенной к нему декораторной функции: декоратор должен получить объект класса, а, значит, этот объект нужно предварительно построить.
4. В этом случае выполняется только одна пользовательская функция: декоратор deco_alpha.

Основная цель второго варианта запуска - показать, что действие декоратора класса может не распространяться на подклассы. ClassFour определен как подкласс ClassThree. Декоратор @deco_alpha применяется к ClassThree и заменяет в нем метод method_y, но это никак не отражается на ClassFour. Разумеется, если бы метод ClassFour.method_y вызывал ClassThree.method_y через функцию super(), то наблюдался бы эффект декоратора, поскольку выполнялась бы функция inner_1 .

Абстрактные базовые классы

Абстрактные базовые классы реализуют механизм организации объектов в иерархии, позволяющий утверждать о наличии требуемых методов и делать другие выводы.

Для определения абстрактного базового класса используется модуль abc. Этот модуль определяет метакласс (ABCMeta) и группу декораторов (@abstractmethod и @abstractproperty), использование которых демонстрируется ниже:

```
from abc import ABCMeta, abstractmethod, abstractproperty

class Foo(metaclass=ABCMeta):
    @abstractmethod
    def spam(self, a, b):
        pass
    @abstractproperty
    def name(self):
        pass
```

В определении абстрактного класса должна быть объявлена ссылка на метакласс ABCMeta, как показано выше. Это совершенно необходимо, потому что реализация абстрактных классов опирается на метаклассы. Внутри абстрактного класса Foo определения методов и свойств, которые должны быть реализованы в подклассах, дополняются с помощью декораторов @abstractmethod и @abstractproperty.

Абстрактный класс не может использоваться непосредственно для создания экземпляров. Если попытаться создать экземпляр предыдущего класса Foo, будет возбуждено исключение:

```
>>> f = Foo()
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class Foo with abstract methods spam
```

Это ограничение переносится и на производные классы. Например, если представить, что имеется класс Bar, наследующий класс Foo, но не имеющий реализации одного или более абстрактных методов, то попытка создать экземпляр класса Bar завершится неудачей с аналогичной ошибкой. Благодаря этой дополнительной проверке абстрактные классы являются удобным инструментом, когда необходимо гарантировать реализацию свойств и методов в подклассах.

Абстрактный класс определяет, какие свойства и методы должны быть реализованы в подклассах, но он не предъявляет никаких требований к аргументам или возвращаемым значениям. То есть абстрактный класс не может потребовать, чтобы метод в подклассе принимал те же самые аргументы, что и абстрактный метод. То же относится и к свойствам - абстрактный класс требует определить свойство, но не требует, чтобы в подклассе была реализована поддержка тех же самых операций над свойством (get, set и delete), что и в базовом классе.

Абстрактный класс не может использоваться для создания экземпляров, но он может определять свойства и методы для использования в подклассах. Кроме того, из подкласса допускается вызывать методы, которые были объявлены абстрактными в базовом классе. Например, вызов Foo.spam(a, b) в подклассе считается допустимым.

Абстрактные базовые классы позволяют регистрировать существующие классы как наследующие этот базовый класс. Делается это с помощью метода register():

```
class Grok:
    def spam(self, a, b):
        print("Grok.spam")

Foo.register(Grok) # Зарегистрирует Grok, как наследующий абстрактный базовый
класс Foo
```

Когда производится регистрация класса в некотором абстрактном базовом классе, операции проверки типа (такие как isinstance() и issubclass()) с привлечением абстрактного базового класса будут возвращать True для экземпляров зарегистрированных классов. В ходе регистрации класса в абстрактном базовом классе не проверяется, действительно ли регистрируемый класс реализует абстрактные свойства и методы. Процедура регистрации оказывает влияние только на операции проверки типа и не производит дополнительных проверок на наличие ошибок в регистрируемом классе.

В отличие от многих других объектно-ориентированных языков программирования, встроенные типы в языке Python организованы в виде достаточно простой иерархии. Например, если взглянуть на такие встроенные типы, как int или float, можно заметить, что они являются прямыми наследниками класса object, родоначальника всех объектов, а не какого-то промежуточного базового класса, представляющего числа. Это усложняет разработку программ, которые проверяют и манипулируют объектами, опираясь на их принадлежность к некоторой категории, такой как числа.

Механизм абстрактных классов позволяет решить эту проблему за счет включения существующих объектов в иерархии типов, определяемых пользователем. Более того, некоторые библиотечные модули организуют встроенные типы по категориям, в соответствии с различными особенностями, которыми они обладают. Модуль collections содержит абстрактные базовые классы, реализующие различные операции над последовательностями, множествами и словарями. Модуль numbers содержит абстрактные базовые классы, которые могут использоваться для организации иерархии числовых типов.

Сетевое программирование. Модуль socketserver

Продолжим тему сетевого программирования рассмотрением модуля socketserver.

Модуль socketserver объявляет классы, упрощающие реализацию серверов на основе сокетов TCP, UDP и домена UNIX.

Обработчики

Для использования классов модуля socketserver необходимо сначала объявить класс обработчика, производный от базового класса BaseRequestHandler. Экземпляр h класса BaseRequestHandler реализует один или более методов из тех, что перечислены ниже:

- h.finish() - вызывается для выполнения завершающих операций после того, как метод handle() закончит работу. По умолчанию этот метод ничего не делает. Он не вызывается, если метод setup() или handle() возбуждает исключение.

- h.handle() - этот метод выполняет фактическую работу в соответствии с запросом. Он вызывается без аргументов, но может использовать некоторые атрибуты экземпляра для получения необходимой информации. Атрибут h.request содержит запрос, h.client_address – адрес клиента и h.server – экземпляр сервера, вызвавшего обработчик. Для потоковых протоколов, таких как TCP, атрибут h.request будет содержать объект сокета. Для дейтаграмм (UDP) он будет содержать строку байтов с принятыми данными.

- h.setup() - этот метод вызывается перед методом handle() для выполнения операций по инициализации. По умолчанию этот метод ничего не делает. Если необходимо реализовать в сервере дополнительные настройки соединения, такие как подключение по протоколу SSL, эти операции должны быть реализованы в этом методе.

Ниже приводится пример класса обработчика, реализующего простой сервер времени, который может работать как с потоками, так и с дейтаграммами:

```
from socketserver import BaseRequestHandler
import socket
import time

class BasicTimeHandler(BaseRequestHandler):
    def handle(self):
        ''' Выполняет фактическую работу в соответствии с запросом
        '''
        resp = time.ctime() + "\n"
        if isinstance(self.request, socket.socket):
            # Работа с потоком
            self.request.sendall(resp.encode('latin-1'))
        else:
            # Работа с дейтаграммой
            self.server.socket.sendto(resp.encode('latin-1'),
                                      self.client_address)
```

Если заранее известно, что обработчик будет работать только с потоковыми протоколами, такими как TCP, в качестве родительского можно использовать класс StreamRequestHandler, а не BaseRequestHandler. Этот класс определяет два атрибута: h.wfile – объект, похожий на файл, который отправляет данные клиенту, и h.rfile – объект, похожий на файл, позволяющий принимать данные от клиента. Например:

```
from socketserver import StreamRequestHandler
import time

class TCPTimeHandler(StreamRequestHandler):
    def handle(self):
        resp = time.ctime() + "\n"
        self.wfile.write(resp.encode('latin-1'))
```

Если обработчик должен работать только с отдельными пакетами и всегда возвращать ответ отправителю, в качестве родительского можно использовать класс DatagramRequestHandler вместо BaseRequestHandler. Этот класс реализует тот же интерфейс файлов, что и класс StreamRequestHandler. Например:

```
from socketserver import DatagramRequestHandler
import time

class UDPTimeHandler(DatagramRequestHandler):
    def handle(self):
        resp = time.ctime() + "\n"
        self.wfile.write(resp.encode('latin-1'))
```

В данном случае все данные, записанные в self.wfile, собираются в единый пакет, который отправляется клиенту по завершении работы метода handle().

Серверы

Чтобы задействовать обработчик, его необходимо подключить к объекту сервера. В модуле объявлено четыре основных класса серверов:

- TCPServer(address, handler) - сервер, поддерживающий протокол TCP IPv4. В аргументе address передается кортеж вида (host, port). В аргументе handler – экземпляр подкласса класса BaseRequestHandler, описанного выше.
- UDPServer(address, handler) - сервер, поддерживающий протокол UDP IPv4. Аргументы address и handler имеют тот же смысл, что и в конструкторе TCPServer().
- UnixStreamServer(address, handler) - сервер, реализующий потоковый протокол с использованием сокетов домена UNIX. Наследует класс TCPServer.
- UnixDatagramServer(address, handler) - сервер, поддерживающий протокол дейтаграмм с использованием сокетов домена UNIX. Наследует класс UDPServer.

Экземпляры всех четырех классов серверов обладают следующими основными методами:

- s.fileno() - возвращает целочисленный дескриптор файла серверного сокета. Наличие этого метода обеспечивает возможность использовать экземпляры серверов в операциях опроса, таких как функция select().
- s.serve_forever() - обслуживает неограниченное число запросов.
- s.shutdown() - останавливает цикл serve_forever().

Следующие атрибуты позволяют получать некоторую основную информацию о настройках действующего сервера:

- s.RequestHandlerClass - пользовательский класс обработчика, который был передан конструктору сервера.
- s.server_address - адрес, на котором сервер ожидает получения запросов от клиентов, например такой кортеж: (“127.0.0.1”, 80).
- s.socket - объект сокета, используемый для приема входящих запросов.

Приведём пример использования экземпляра класса TimeHandler в составе сервера TCP:

```
from socketserver import TCPServer

serv = TCPServer(('', 10000), BasicTimeHandler)
serv.serve_forever()
```

Ниже приводится пример использования обработчика в составе сервера UDP:

```
from socketserver import UDPServer

serv = UDPServer(('', 10000), BasicTimeHandler)
serv.serve_forever()
```

Ключевым аспектом модуля socketserver является обособленность обработчиков от серверов. То есть один и тот же обработчик можно подключить к серверам самых разных типов, не меняя его реализацию.

Определение собственных серверов

Зачастую для серверов требуется определять различные параметры настройки, чтобы учесть такие особенности, как различные семейства адресов, предельное время ожидания, многозадачность и другие. Для этого необходимо определить свой класс, производный от одного из четырех базовых классов серверов, описанных в предыдущем разделе. Для настройки параметров сокета, составляющего основу сервера, могут быть определены следующие атрибуты класса:

- Server.address_family - семейство адресов, используемое сокетом сервера. По умолчанию используется значение socket.AF_INET. Если необходимо обеспечить поддержку IPv6, следует использовать значение socket.AF_INET6.
- Server.allow_reuse_address - логический флаг, разрешающий или запрещающий повторное использование адреса сокета. Это бывает удобно, когда необходимо быстро перезапустить сервер на том же порту, после того как программа завершится (в противном случае придется ожидать несколько минут). По умолчанию используется значение False.
- Server.request_queue_size - размер очереди запросов, который передается методу listen() сокета. По умолчанию используется значение 5.
- Server.socket_type - тип сокета, используемого сервером, такой как socket.SOCK_STREAM или socket.SOCK_DGRAM.
- Server.timeout - предельное время ожидания в секундах, в течение которого сервер будет ожидать поступления новых запросов. По истечении этого интервала времени будет вызываться метод handle_timeout() сервера (описывается ниже), после чего сервер опять возвращается к ожиданию. Это значение не используется для установки предельного времени ожидания в сокете. Однако если для сокета было определено предельное время ожидания, вместо этого значения будет использоваться значение из сокета.

Пример создания сервера, который позволяет повторно использовать номер порта:

```
class TcpTimeServer(TCPServer):
      # allow_reuse_address разрешает/запрещает повторное использование адреса
сокета
      allow_reuse_address = True
serv = TcpTimeServer(('', 10000), BasicTimeHandler)
serv.serve_forever()
```

При желании в классах, производных от базовых классов серверов, можно переопределять следующие методы. Если вы будете переопределять какой-либо из этих методов, не забудьте вызвать одноименный метод базового класса.

- Server.activate() - выполняет операцию listen() на стороне сервера. Сокет сервера доступен в виде атрибута self.socket.
- Server.bind() - выполняет операцию bind() на стороне сервера.
- Server.handle_error(request, client_address) - обрабатывает неперехваченные исключения, которые возникают в процессе работы. Для получения информации о последнем исключении следует использовать функцию sys.exc_info() или функции из модуля traceback.
- Server.handle_timeout() - обрабатывает ситуации, когда операции завершаются по истечении предельного времени ожидания. Переопределяя этот метод и изменяя значение предельного времени ожидания, можно в цикл событий сервера интегрировать дополнительные операции.
- Server.verify_request(request, client_address) - этот метод можно переопределить, чтобы реализовать проверку соединения перед обработкой запроса. Это может быть реализация сетевой защиты или каких-то других проверок.

Дополнительные возможности сервера можно получить за счет использования классов-примесей. С их помощью может быть добавлена многозадачность, реализованная на основе потоков управления или дочерних процессов. Для этой цели определены следующие классы:

- ForkingMixIn - класс-примесь, который в UNIX создает дочерние процессы сервера, позволяя одновременно обслуживать множество клиентов. Атрибут класса max_children определяет максимальное количество дочерних процессов, а атрибут класса timeout определяет интервал времени, через который будут выполняться попытки ликвидировать процессы-зомби. Атрибут экземпляра active_children содержит количество активных процессов.
- ThreadingMixIn - класс-примесь, который модифицирует сервер так, что он создает новые потоки управления, позволяя одновременно обслуживать множество клиентов. Этот класс не имеет ограничений на количество создаваемых потоков управления. По умолчанию создаются недемонические потоки, если в атрибут класса daemon_threads не записать значение True.

Чтобы добавить к серверу эти возможности, следует использовать механизм множественного наследования и класс-примесь указывать первым в списке. Например, ниже приводится пример сервера времени, запускающего дочерние потоки:

```
from socketserver import TCPServer, ThreadingMixIn

class ThreadingTimeServer(ThreadingMixIn, TCPServer):
    allow_reuse_address = True
    max_children = 10

serv = ThreadingTimeServer(('', 10000), BasicTimeHandler)
serv.serve_forever()
```

Поскольку ситуация, когда параллельно выполняется несколько серверов, достаточно типична, для этой ситуации в модуле SocketServer предопределены следующие классы серверов:

- ForkingUDPServer(address, handler);
- ForkingTCPServer(address, handler);
- ThreadingUDPServer(address, handler);
- ThreadingTCPServer(address, handler).

В действительности эти классы объявлены, как производные классы от классов-примесей и классов серверов. В качестве примера ниже приводится объявление класса ForkingTCPServer:

```
class ForkingTCPServer(ForkingMixIn, TCPServer): pass
```

Не переживайте, если пока не сильно знакомы с терминами "потоки"/"процессы". Более плотному знакомству с многопоточным программированием будет посвящено занятие "Потоки и многозадачность" в рамках данного курса.

Создание собственных серверов приложений

Класс socketserver часто используется другими модулями из стандартной библиотеки для реализации серверов, работающих с прикладными протоколами, такими как HTTP и XML-RPC. Функциональность этих серверов также можно приспосабливать под свои нужды через множественное наследование и переопределение методов, объявленных в базовых классах. Например, ниже приводится сервер XML-RPC, порождающий дочерние процессы, который принимает только соединения, исходящие с петлевого (loopback) интерфейса:

```
from xmlrpc.server import SimpleXMLRPCServer
from socketserver import ThreadingMixIn

class MyXMLRPCServer(ThreadingMixIn, SimpleXMLRPCServer):
    def verify_request(self, request, client_address):
        host, port = client_address
        if host != '127.0.0.1':
            return False
        return SimpleXMLRPCServer.verify_request(self, request, client_address)

# Пример использования
def add(x, y):
    return x+y

server = MyXMLRPCServer(("", 45000))
server.register_function(add)
server.serve_forever()
```

Чтобы опробовать этот пример, необходимо импортировать модуль xmlrpc.client. Запустите сервер, реализация которого представлена выше, а затем запустите отдельный процесс интерпретатора Python:

```
>>> import xmlrpc.client
>>> s = xmlrpc.client.ServerProxy("http://localhost:45000")
>>> s.add(3,4)
7
>>>
```

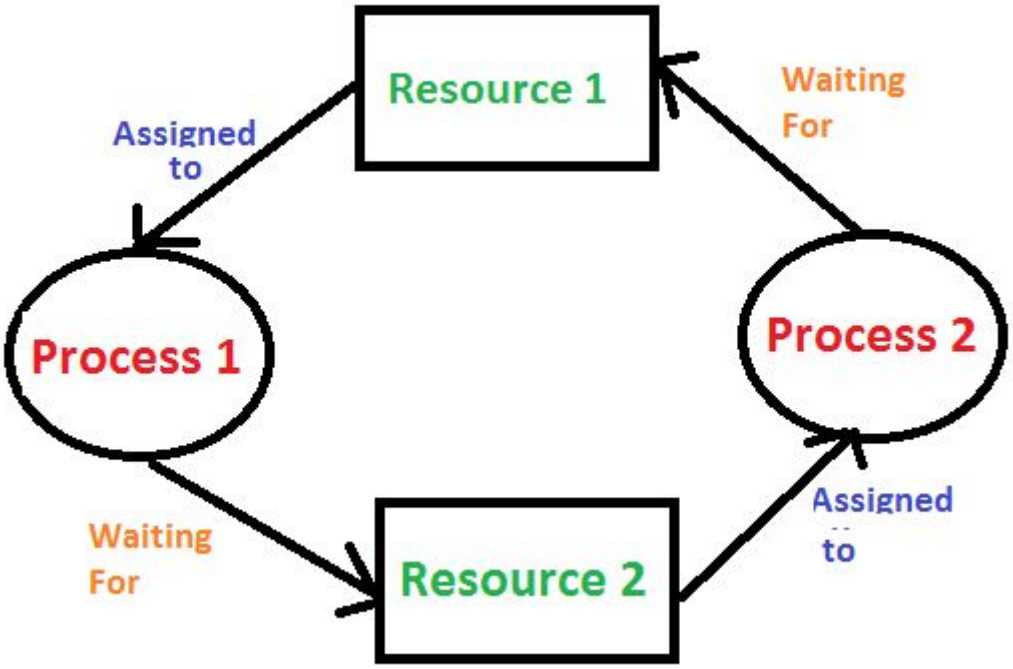
Чтобы убедиться, что сервер отвергает попытки соединения с другого адреса, попробуйте выполнить тот же код на другом компьютере в сети. Для этого замените строку «localhost» сетевым именем компьютера, на котором запущен сервер.

Замечания по сетевому программированию

В ходе работы с сетью вероятно у многих слушателей возникали вопросы по некоторым аспектам взаимодействия с сокетами и сетевыми соединениями. Уделим внимание таким моментам как блокировка, закрытие соединений и сжатие данных.

Взаимная блокировка

Взаимная блокировка (англ. deadlock) — ситуация в многозадачной среде или СУБД, при которой несколько процессов находятся в состоянии бесконечного ожидания ресурсов, занятых самими этими процессами.



Такая ситуация может получиться при использовании протокола TCP.

Для предотвращения входа в блокировку при обмене большими объемами данных между клиентом и сервером существует два возможных решения:

1. Отключение блокировки сокета - при этом методы send и recv будут сразу возвращать управление, если нет возможности отправить или принять данные.

2. Использование подходов для раздельной обработки данных от разных адресатов: создание отдельных процессов/потоков для входных/выходных потоков данных или использование системных вызовов select() или poll().

Закрытие соединений

На прошлых занятиях не был сделан акцент на закрытии соединений. При закрытии соединения одной стороной вторая сторона ведёт себя как будто был достигнут конец файла, тогда метод recv() возвращает пустую строку.

Но что, если размер передаваемых данных заранее неизвестен, а клиенту (который передаёт этот массив данных) при этом нужно получить ответ сервера?

В этом случае имеет смысл использовать метод сокета shutdown() при завершении передачи. Это помогает исключить ситуацию, когда сервер бесконечно вычитывает данные из сокета ожидая окончания данных.

Напомним, метод:

- s.shutdown(how) - отключает одну или обе стороны соединения. Если в аргументе how передается значение 0, дальнейший прием данных будет запрещен. Если в аргументе how передается значение 1, будет запрещена дальнейшая передача данных. Если в аргументе how передается значение 2, будут запрещены и прием, и передача данных.

Таким образом, метод shutdown() позволяет сделать «полузакрытый» сокет (half-closed; закрытый для передачи, открытый для приёма), не обрывая соединение.

Разница между закрытием (s.close()) сокета и выключением в обоих направлениях (s.shutdown(2)) будет видна при совместном использовании одного сокета в разных процессах. Тогда закрытие сокета в одном процессе будет оставлять сокет открытым в другом процессе. Метод shutdown(2) в таком случае сделает сокет недоступным для всех процессов.

Сжатие

Время передачи данных по сети критичнее времени работы процессора, поэтому в некоторых случаях для данных следует выполнять сжатие, и уже потом передавать по сети.

Один из способов сжатия - GNU zlib - является очень распространенным для передачи в Интернет. В Python модуль zlib доступен в стандартной библиотеке. Примечательной особенностью данного формата является работа с кадрами данных - при разжатии может быть использовано не всё сообщение, а только часть для получения соответствующей части исходного сообщения, что может быть удобно при потоковой обработке данных.

Небольшой пример работы с модулем zlib:

```
import zlib

# Сформируем сообщение, содержащее сжатые и несжатые данные
data = zlib.compress(b'Python') + b'.' + zlib.compress(b'zlib') + b'.'
print(data)
_len = len(data)
print(_len)

# Предположим теперь, что сжатые данные приходят блоками по 8 байт

d = zlib.decompressobj()
print(d.decompress(data[0:8]), d.unused_data)

# Поскольку d.unused_data содержит пустую строку, значит не все данные были
разжаты.
print(d.decompress(data[8:16]), d.unused_data)

# Непустая строка в d.unused_data сигнализирует, что первая часть данных была
разжата.
# Теперь нужно разжать оставшиеся данные:

d = zlib.decompressobj()
print(d.decompress(b'x'), d.unused_data)
print(d.decompress(data[16:24]), d.unused_data)
print(d.decompress(data[24:]), d.unused_data)
```

Итоги

Тема метапрограммирования весьма нетривиальна и в большинстве случаем с наскока в неё не погрузиться. Тем не менее, знание о работе метаклассов даёт возможность лаконично управлять созданием и модификацией обычных классов (при этом осознавая, где метакласс необходим, а где без него можно обойтись), а также понимать устройство крупных библиотек.

Для лучшего закрепления темы настоятельно рекомендуем Вам ещё раз самостоятельно разобраться в примерах кода к данному уроку.

Домашнее задание

Основное задание

1. Повторить SQL к следующему занятию.
2. Перейти к объектной модели в реализации проекта “Мессенджер”. В качестве основы слушателям предлагается следующая ООП-модель системы:
    1. Класс JIMСообщение - класс, реализующий сообщение (msg) по протоколу JIM.
    2. Класс JIMОтвет - класс, реализующий ответ (response) по протоколу JIM.
    3. Класс Клиент - класс, реализующий клиентскую часть системы.
    4. Класс Чат - класс, обеспечивающий взаимодействие двух клиентов.
    5. Класс ЧатКонтроллер - класс, обеспечивающий передачу данных из Чата в ГрафическийЧат и обратно; обрабатывает события от пользователя (ввод данных, отправка сообщения).
    6. Класс ГрафическийЧат - базовый класс, реализующий интерфейс пользователя (UI) - вывод сообщений чата, ввод данных от пользователя - служит базой для разных интерфейсов пользователя (консольный, графический, WEB).
        - Дочерний класс КонсольныйЧат - обеспечивает ввод/вывод в простой консоли.
    7. Класс Сервер - базовый класс сервера мессенджера; может иметь разных потомков - работающих с потоками или выполняющих асинхронную обработку.
    8. Класс Хранилище - базовый класс, обеспечивающий сохранение данных (сохранение информации о пользователях на сервере, сохранение сообщений на стороне клиента).
        - Дочерний класс ФайловоеХранилище - обеспечивает сохранение информации в текстовых файлах
3. Для всех методов и функций необходимо написать тесты.

- Уместно воспользоваться дескрипторами для реализации атрибутов классов.

Дополнительное задание

1. Реализовать метакласс ClientVerifier, выполняющий базовую проверку класса Клиент (для некоторых проверок уместно использовать модуль dis):
    - отсутствие вызовов accept и listen для сокетов
    - использование сокетов для работы по TCP
    - отсутствие создания сокетов на уровне классов, т.е. отсутствие конструкций вида:

```
class Client:
    s = socket()
    ...
```

2. Реализовать метакласс ServerVerifier, выполняющий базовую проверку класса Сервер:
    - отсутствие вызовов connect для сокетов;
    - использование сокетов для работы по TCP.

Дополнительные материалы

1. Пользовательские атрибуты в Python
2. Metaprogramming with Metaclass in Python
3. Understanding Python metaclasses
4. PyCon 2013. David Beazley. Python 3 Metaprogramming
5. Порядок разрешения методов в Python

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. David Beazley, Brian K. Jones «Python Cookbook», Third Edition.
2. Brandon Rhodes, John Goerzen «Foundations of Python Network Programming».
3. Бизли Дэвид. «Python. Подробный справочник», 4-е издание.
4. Лучано Ромальо. «Python. К вершинам мастерства».
5. Лутц Марк. «Изучаем Python», 4-е издание.