Лабораторная работа №3. Python-классы.

Краткое описание

В этой ЛР Вы знакомитесь с модулями и ООП в Python, а также осваиваете работу с сетью.

В лабораторной работе необходимо создать набор классов для реализации работы с VK API

Теория

Простейшая форма определения класса выглядит так:

```
class ИмяКласса:
    <оператор-1>
    .
    .
    .
    <оператор-N>
```

Определения классов, как и определения функций (операторы `def`), должны быть исполнены для того, чтобы определить действие. (Вы можете, предположим, поместить определение класса в ветку оператора `if` или внутрь функции.)

На практике, внутри определения класса обычно помещаются определения функций, но позволено использовать и другие операторы — и иногда с пользой — как мы увидим позже. Определения функций внутри класса имеют особенную форму списка аргументов, в связи с соглашениями по вызову методов — опять же, это будет рассмотрено ниже.

При вводе определения класса создаётся новое пространство имён, которое и используется в качестве локальной области видимости. Таким образом, все присваивания локальным переменным происходят в этом новом пространстве имён. В частности, определения функций связываются здесь с именами новых функций.

При успешном окончании парсинга определения класса (по достижении конца определения), создаётся *объект-класс* (`class object`). По существу, это обёртка вокруг содержимого пространства имён, созданного во время определения класса; подробнее

объекты классов мы изучим в следующем разделе. Оригинальная локальная область видимости (та, которая действовала в последний момент перед вводом определения класса) восстанавливается, а объект-класс тут же связывается в ней с именем класса, указанном в заголовке определения класса (в примере — *ИмяКласса*).

Объекты-классы

Объекты-классы поддерживают два вида операций: ссылки на атрибуты и создание экземпляра.

Ссылки на атрибуты (Attribute references) используют стандартный синтаксис, использующийся для всех ссылок на атрибуты в Python: *объект.имя*. Корректными именами атрибутов являются все имена, которые находились в пространстве имён класса при создании объекта-класса. Таким образом, если определение класса выглядело так:

```
class MyClass:
    """Простой пример класса"""
    i = 12345
    def f(self):
        return 'привет мир'
```

то MyClass.i и MyClass.f являются корректными ссылками на атрибуты, возвращающими целое и *объект-функцию* (function object) соответственно. Атрибутам класса можно присваивать значение, так что вы можете изменить значение MyClass.i через присваивание. __doc__ также является корректным атрибутом, возвращающим строку документации, принадлежащей классу: "Простой пример класса".

Создание экземпляра класса использует синтаксис вызова функции. Просто представьте, что объект-класс — это непараметризированная функция, которая возвращает новый экземпляр класса. Например (предполагая класс, приведённый выше):

```
x = MyClass()
```

создаёт новый экземпляр класса и присваивает этот объект локальной переменной x.

Операция *создания экземпляра* (instantiation) создаёт объект данного класса. Большая часть классов предпочитает создавать экземпляры, имеющие определённое начальное состояние. Для этого класс может определять специальный метод под именем __init__(), например так:

```
def __init__(self):
    self.data = []
```

Когда в классе определён метод __init__(), при создании экземпляра автоматически вызывается __init__() нового, только что созданного объекта. Так, в этом примере, новый инициализированный экземпляр может быть получен за счёт выполнения кода:

```
x = MyClass()
```

Конечно же, для большей гибкости, метод __init__() может иметь параметры. В этом случае аргументы, переданные оператору создания экземпляра класса, передаются методу __init__(). Например,

```
>>> class Complex:
...     def __init__(self, real_part, imag_part):
...         self.r = real_part
...         self.i = imag_part
...
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

Объекты-экземпляры

Теперь, что же мы можем делать с объектами-экземплярами? Единственные операции, доступные объектам-экземплярам — это ссылки на атрибуты. Есть два типа корректных имён атрибутов — это атрибуты-данные и методы.

Атрибуты-данные (data attributes) аналогичны «переменным экземпляров» в Smalltalk и «членам-данным» в C++. Атрибуты-данные не нужно описывать: как и переменные, они начинают существование в момент первого присваивания. Например, если x — экземпляр созданного выше MyClass, следующий отрывок кода выведет значение 16, не вызвав ошибок:

```
x.counter = 1
while x.counter < 10:
    x.counter = x.counter * 2
print(x.counter)
del x.counter
```

Другой тип ссылок на атрибуты экземпляра — это *метод* (method). Метод — это функция, «принадлежащая» объекту. (В Python термин не уникален для экземпляров класса: другие объекты также могут иметь методы. Например, объекты-списки имеют методы `append`, `insert`, `remove`, `sort` и т. п. Тем не менее, ниже под термином «метод» мы будем понимать только методы объектов-экземпляров классов, пока отдельно не будет указано иное.)

Корректные имена методов объектов-экземпляров зависят от их класса. По определению, все атрибуты класса, являющиеся объектами-функциями, описывают соответствующие методы его экземпляров. Так, в нашем примере, `x.f` является корректной ссылкой на метод, а `x.i` ей не является, поскольку не является и `MyClass.i`. Но при этом `x.f` — это не то же самое, что `MyClass.f`: это объект-метод, а не объект-функция.

Объекты-методы

Обычно, метод вызывают сразу после его связывания [с функцией]:

```
x.f()
```

На примере `MyClass` такой код возвратит строку `'привет мир'`. Однако, не обязательно вызывать метод так уж сразу: `x.f` — это объект-метод, он может быть отложен и вызван когда-либо позже. Например:

```
xf = x.f
while True:
    print(xf())
```

будет печатать `'привет мир'` до конца времён.

Что конкретно происходит при вызове метода? Вы, возможно, заметили, что `x.f()` выше был вызван без аргументов, хотя в описании функции `f`аргумент был указан. Что же случилось с аргументом? Несомненно, Python порождает исключение когда функция, требующая присутствия аргумента, вызвана без единого — даже, если он на самом деле не используется…

Теперь вы, возможно, догадались: отличительная особенность методов состоит в том, что в качестве первого аргумента функции передаётся объект. В нашем примере вызов `x.f()` полностью эквивалентен вызову `MyClass.f(x)`. В общем случае, вызов метода со списком из *n* аргументов эквивалентен вызову соответствующей функции со списком аргументов, созданным за счёт вставки объекта, вызвавшего метод, перед первым аргументом.

Если вы всё ещё не поняли, как работают методы, взгляд на реализацию возможно прояснит происходящее. Когда атрибут экземпляра ссылается на что-либо, не являющееся атрибутом-данными, производится поиск по классу. Если имя указывает корректный атрибут класса, являющийся объектом-функцией, создаётся метод: через упаковку (указателя на) объекта-экземпляра и найденного объекта-функции в абстрактный объект, получается объект-метод. Когда объект-метод вызывается со списком аргументов, он снова распаковывается и новый список аргументов конструируется из объекта-экземпляра и оригинального списка аргументов, и затем уже с новым списком аргументов вызывается объект-функция.

Наследование

Конечно же, не поддерживай «класс» наследование, не стоило бы называть его «классом». Синтаксис производного класса выглядит так:

```
class ИмяПроизводногоКласса(ИмяБазовогоКласса):
    <оператор-1>
    .
    .
    .
    <оператор-N>
```

Имя *ИмяБазовогоКласса* должно быть определено в области видимости, содержащей определение производного класса. Вместо имени базового класса также позволяется использовать другие выражения. Это может быть полезно, например, когда базовый класс определён в другом модуле:

```
class ИмяПроизводногоКласса(имямодуля.ИмяБазовогоКласса):
```

Использование определения производного класса проходит таким же образом, как и базового. Базовый класс полностью сохраняется по завершению конструирования объекта-класса. Такой метод используется для разрешения ссылок на атрибуты[1]: если запрошенный атрибут не был найден в самом классе, поиск продолжается в базовом классе. Правило применяется рекурсивно, если базовый класс сам является производным от некоторого другого класса.

В создании экземпляров производных классов нет ничего особенного: `ИмяПроизводногоКласса()` создаёт новый экземпляр класса. Ссылки на методы

разрешаются следующим образом: производится поиск соответствующего атрибута класса (спускаясь вниз по цепочке базовых классов, если необходимо) и ссылка на метод считается корректной, если она порождает объект-функцию.

Производные классы могут перегружать методы своих базовых классов. Поскольку у методов нет особых привилегий при вызове других методов того же объекта, метод базового класса, вызывающий другой метод, определённый в этом же классе, может вызвать перегруженный метод производного класса. (Для программистов на C++: все методы в Python фактически виртуальны.)

При перегрузке метода в производном классе возможна не только замена действия метода базового класса с тем же именем, но и его расширение. Существует простой способ вызвать метод базового класса прямым образом: просто вызовите «`ИмяБазовогоКласса.имяметода(self, аргументы)`». Такой способ будет неожиданно полезным и для клиентов. (Обратите внимание, что он работает только если базовый класс определён и импортирован прямо в глобальную область видимости.)

В языке Python есть функции, которые работают с наследованием:

- Используйте `isinstance()` чтобы проверить тип объекта: `isinstance(obj, int)` возвратит `True` только если `obj.__class__` является `int`или некоторым классом, наследованным от `int`.
- Используйте `issubclass()` чтобы проверить наследственность класса: `issubclass(bool, int)` возвратит `True`, поскольку класс `bool` является наследником (subclass) `int`. Однако, `issubclass(float, int)` возвратит `False`, поскольку класс `float` не является наследником `int`.

Множественное наследование

Python также поддерживает форму *множественного наследования* (multiple inheritance). Определение класса с несколькими базовыми классами будет выглядеть так:

```
class ИмяПроизводногоКласса(Базовый1, Базовый2, Базовый3):
   <оператор-1>
   .
   .
   .
   <оператор-N>
```

В простейших случаях и для большинства задач, вы можете представлять себе поиск атрибутов, наследованных от родительского класса в виде «сперва вглубь», затем

«слева-направо». Таким образом, если атрибут не найден в *ИмяПроизводногоКласса*, его поиск выполняется в *Базовом1*, затем (рекурсивно) в базовых классах *Базового1* и только если он там не найден, поиск перейдёт в *Базовый2* и так далее.

На самом деле всё немного сложнее. Порядок разрешения методов[56] (method resolution order) меняется динамически, чтобы обеспечить возможность сотрудничающих вызовов super(). Этот способ известен в некоторых других языках с поддержкой множественного наследования как «вызов-следующего-метода» («call-next-method») и имеет больше возможностей, чем вызов родительского метода в языках с единичным наследованием.

Динамическое упорядочивание (dynamic ordering) имеет важность, поскольку все вариации множественного наследования проявляют в себе эффект ромбовых отношений (когда как минимум один родительский класс может быть доступен различными путями из низшего в иерархии класса). Например, все классы наследуются от object, так что множественное наследование в любом виде предоставляет более одного пути для того, чтобы достичь object. Чтобы защитить базовые классы от двойных и более запросов, динамический алгоритм «выпрямляет» (linearizes) порядок поиска таким образом, что тот сохраняет указанный слева-направо порядок для каждого класса, который вызывает каждый родительский класс только единожды и является монотонным (значит, класс можно сделать наследником, не взаимодействуя с порядком предшествования его родителей). Обобщённые вместе, эти свойства позволяют разрабатывать надёжные и расширяемые классы, используя множественное наследование.

Приватные переменные

В Python имеется ограниченная поддержка идентификаторов, приватных для класса. Любой идентификатор в форме __spam (как минимум два предшествующих символа подчёркивания, как максимум один завершающий) заменяется дословно на _classname__spam, где classname — текущее имя класса, лишённое предшествующих символов подчёркивания. Это *искажение* (mangling) производится без оглядки на синтаксическую позицию идентификатора, поэтому может использоваться для определения переменных, приватных для класса экземпляров, переменных класса, методов, переменных в глобальной области видимости (globals), и даже переменных, использующихся в экземплярах, приватных для этого класса на основе экземпляров *других* классов. Имя может быть обрезано, если его длина превышает 255 символов. Вне

классов, или когда имя состоит из одних символов подчёркивания, искажения не происходит.

Предназначение искажения имён состоит в том, чтобы дать классам лёгкую возможность определить «приватные» переменные экземпляров и методы, не беспокоясь о переменных экземпляров, определённых в производных классах, и о забивании кодом вне класса переменных экземпляров. Обратите внимание, что правила искажения имён разработаны, в основном, чтобы исключить неприятные случайности — решительная душа всё ещё может получить доступ или изменить переменные, предполагавшиеся приватными. В некотором особом окружении, таком как отладчик, это может оказаться полезным — и это единственная причина, по которой лазейка не закрыта. (Внимание: наследование класса с таким же именем как и у базового делает возможным использование приватных переменных базового класса.)

Заметьте, что код, переданный в `exec()` или `eval()`, не предполагает в качестве текущего имени класса имя класса, порождающего вызов — так же, как и в случае эффекта с оператором `global` — эффекта, который также ограничен для всего побайтно-компилирующегося кода. И, такое же ограничение применимо для функций `getattr()`, `setattr()` и `delattr()`, и также для прямой ссылки на `__dict__`.

Исключения — тоже классы

Исключения, определённые пользователем, могут быть также отождествлены с классами. При использовании этого механизма становится возможным создавать расширяемые иерархии исключений.

Оператор `raise` имеет следующие (синтаксически) правильные формы:

```
raise Класс
```

```
raise Экземпляр
```

В первой форме, *`Класс`* должен быть экземпляром типа или класса, производного от него. Первая форма является краткой записью следующего кода:

```
raise Class()
```

Класс в блоке `except` является сопоставимым с исключением, если является этим же классом или самим по себе базовым классом (никаких других способов обхода —

описанный в блоке except производный класс не сопоставим с базовым). Например, следующий код выведет B, C, D в этом порядке:

```
class B(Exception):
    pass
class C(B):
    pass
class D(C):
    pass

for c in [B, C, D]:
    try:
        raise c()
    except D:
        print("D")
    except C:
        print("C")
    except B:
        print("B")
```

Обратите внимание, что если бы блоки except шли в обратном порядке (начиная с «except B»), код вывел бы B, B, B — сработал бы первый совпадающий блок except.

При выводе сообщения об ошибке о необработанном исключении, выводится класс исключения, затем двоеточие и пробел, и наконец экземпляр, приведённый к строке за счёт встроенной функции str().

Задание

Вход:
username или vk_id пользователя

Выход:
Гистограмма распределения возрастов друзей пользователя, поступившего на вход

Пример:
Вход:
reigning

Выход:
19 #
20 ##
21 ##
22 ####################
23 ################
24 ####
25 #
28 #
29 #
30 #
37 #
38 ##
45 #

Указания

За основу возьмите базовый класс:
https://gist.github.com/Abashinos/024c1dcaf92f1ff733c63a07e447ab51

Для реализации методов ВК наследуйтесь от этого базового класса. Создайте один класс для получения id пользователя из username и один для получения и обработки списка друзей. В классах-наследниках необходимо реализовать методы:

- get_params - если есть get параметры (необязательно).
- get_json - если нужно передать post данные (необязательно).
- get_headers - если нужно передать дополнительные заголовки (необязательно).
- response_handler - обработчик ответа. В случае успешного ответа необходим, чтобы преобразовать результат запроса. В случае ошибочного ответа необходим, чтобы сформировать исключение.
- _get_data - внутренний метод для отправки http запросов к VK API.

Для решения задачи нужно обратиться к двум методам VK API

1) users.get - для получения vk id по username

2) friends.get - для получения друзей пользователя. В этом методе нужно передать в get параметрах fields=bdate для получения возраста. Нужно принять во внимание, что не у всех указана дата рождения

Описание методов можно найти тут:
https://vk.com/dev/methods

Разнесите базовый класс, классы наследники и основную программу в разные модули.
Про модули можно прочитать тут:
https://docs.python.org/3/tutorial/modules.html
https://habrahabr.ru/post/166463/

Для выполнения запросов нужно использовать библиотеку *requests*
http://docs.python-requests.org/en/master/

Для обработки дат (дней рождения) используйте встроенную библиотеку *datetime*
https://docs.python.org/3/library/datetime.html

Чтобы установить библиотеку используйте пакетным менеджером *pip*
https://pip.pypa.io/en/stable/quickstart/

Подсказки:

1. Метод get библиотеки requests принимает вторым аргументом словарь get-параметров.
2. Не забывайте, что в классах-наследниках можно перегружать статические поля наследуемого класса.

Дополнительное задание

Постройте гистограмму с использованием *matplotlib*
http://matplotlib.org/examples/statistics/histogram_demo_features.html