Классы

Составитель: В.С. Воронов

СОДЕРЖАНИЕ

1 Классы

Классы представляют собой средства объединения данных и поведения. Каждый новый класс задает новый тип данных и позволяет создавать объекты или экземпляры этого типа. Экземпляр может содержать атрибуты, необходимые для хранения данных и характеризующие состояние объекта, а также методы для изменения состояния, которые определяются классом. Класс является одним из ключевых понятий в объектно-ориентированном программировании (ООП).

По сравнению с другими языками программирования, механизм классов Python характерен минимумом нового синтаксиса и семантики. Классы Python предоставляют все стандартные возможности объектно-ориентированного программирования: механизм наследования классов позволяет использовать несколько базовых классов, способствуя повторному использованию кода, производный класс может переопределять любые методы своего базового класса или классов, а метод может вызывать метод базового класса с тем же именем. Объекты могут содержать произвольные количества и виды данных. Как и в случае с модулями, классы обладают динамической природой Python: они создаются во время выполнения и могут быть изменены в дальнейшем после создания.

Синтаксис определения класса довольно прост. Для этого используется ключевое слово `class`, за которым следует имя класса. Имена классов принято писать, используя CamelCase или «верблюжью» нотацию.

```
class FooBar:
    pass
```

Определения классов, также как и определения функций должны располагаться до их использования.

При объявлении класса создается новое пространство имен и используется как локальная область видимости (по аналогии с функциями), внутри которой могут объявляться переменные (атрибуты) и функции,

которые называются методами и имеют немного отличный от обычных функций вид объявления. В Python классы это тоже объекты (всё – объекты!). При определении класса создается объект класса, который связывается с переменной в глобальной области видимости, имеющей тоже имя, что и имя класса (переменная будет иметь имя `FooBar` из примера выше).

Объекты классов поддерживают несколько операций: создание экземпляров и ссылки на атрибуты.

Для обращения к атрибутам используется стандартный синтаксис – точечная нотация `obj.attribute`. Атрибутами выступают все имена, определенные в пространстве имен класса. Рассмотрим пример простейшего класса.

```
class FooBar:
    """Class example"""
    attribute = 42
    def boo(self):
        return f'This class is called "{self.__class__.__name__}"
.'
```

Классы могут содержать строку документации, которая располагается сразу после объявления класса и использоваться для вывода справки с помощью функции `help(FooBar)`. Строка документации храниться в специальном атрибуте `__doc__`, получить к нему доступ можно через точечную нотацию `FooBar.__doc__`. В этом примере `FooBar.attribute` и `FooBar.boo` будут действительными ссылками на атрибуты класса `FooBar`, возвращающими целое число и объект функции. Стоит отметить, что все методы должны принимать хотя бы один аргумент. Этим аргументом выступает `self`. О его назначении будет рассказано позже. Его имя не фиксировано, но по соглашению его именуют именно так. Атрибут `FooBar.attribute` можно изменить, используя обычное присваивание.

```
FooBar.attribute = 3.14159
```

Экземпляр класса создается с использованием круглых скобок («вызов» класса).

```
x = FooBar()
```

Такая запись создаст новый объект экземпляра класса `FooBar` и свяжет его с именем переменной `x`. Экземпляры также могут получать доступ к атрибутам, используя точечную нотацию `instance.name`.

```
x.attribute
x.boo()
```

В этом случае будет создан «пустой» экземпляр класса. Для того чтобы управлять начальным состоянием экземпляра нужно использовать дандер-метод `__init__`, который неявно вызывается при создании экземпляра и инициализирует его начальное состояние. Важно отметить, что метод `__init__` не должен ничего возвращать, т.е. не должен содержать инструкции `return`.

```
class Point2D:
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

Тогда создание экземпляра класса будет следующим:

```
p = Point2D(1, 0)
print(f'x = {p.x}; y = {p.y}')
```

Для экземпляров допустимыми операциями являются только ссылки на атрибуты. О них пойдет речь далее.

1.1 Атрибуты класса и экземпляра

Как упоминалось выше, атрибуты могут быть разделены на два вида: атрибуты данных (data attributes) и методы. Рассмотрим простой пример.

```
class Counter:
    global_counter = 0
    def __init__(self, initial=0):
        self.counter = initial

    def increment(self):
        self.counter += 1
        self.__class__.global_counter += 1

    def get_local_counter(self):
```

```
        return self.counter

    def get_global_counter(self):
        return self.__class__.global_counter
```

Эта бессмысленная и беспощадная реализация счетчика с двумя состояниями – локальным и глобальным.

```
x = Counter(5)
x.increment()
print(x.get_local_counter())  # 6
print(x.get_global_counter())  # 1
```

Здесь в комментариях указано, какой результат будет выведен на экран.

Начнем рассмотрение с атрибутов данных, их также можно разделить на две группы: атрибуты класса и атрибуты экземпляра. Атрибуты класса объявляются непосредственно внутри пространства имен класса, по аналогии с локальными переменными. В классе Counter атрибутом класса будет `global_counter`. Атрибуты класса доступны непосредственно из объекта класса, т.е. к ним можно обратиться с помощью синтаксиса `class_obj.attribute` или в случае примера выше – `Counter.global_counter`. Есть еще один, менее правильный, способ объявить атрибут класса – сделать это после определения класса «вручную», используя присваивание. Например:

```
Counter.attribute = 1
print(Counter.attribute)
```

Их также можно удалять с помощью оператора `del`.

```
del Counter.attribute
```

Атрибуты экземпляра создаются методом `__init__`, с помощью конструкции `self.name = value`. В примере класса `Counter` атрибутом экземпляра будет атрибут `counter`. При этом значения этих атрибутов будут разные у каждого экземпляра. Ссылку на атрибут экземпляра можно получить из объекта экземпляра, обратившись по имени атрибута.

```
print(x.counter)  # 6
```

Атрибуты экземпляра можно создавать и вне метода `__init__`, используя присваивание.

```
x.attribute = 42
print(x.attribute)  # 42
del x.attribute
```

Из экземпляра класса можно получить ссылку на атрибут класса, также обратившись по его имени.

```
print(x.global_counter)  # 1
```

При этом атрибуты класса будут одинаковы у каждого экземпляра.

```
x = Counter(5)
x.increment()
y = Counter(42)
y.increment()
y.increment()
print(x.get_local_counter())  # 6
print(x.global_counter)  # 3
print(y.get_local_counter())  # 44
print(y.global_counter)  # 3
```

1.2 Методы

Методы классов это обычные функции, определяемые в теле класса.

```
class Simple:
    def foo(self):
        print(f'Class name: {self.__class__.__name__}')
```

Обращение к методам происходит с помощью точечной нотации, а вызов аналогично функциям, с помощью круглых скобок.

```
x = Simple()
x.foo()  # Class name: Simple
```

Методы являются объектами типа `method`, поэтому их не обязательно вызывать. С ними можно выполнять те же операции, что и с функциями, т.е. хранить в структурах данных, удалять, передавать в качестве аргументов, возвращать из других функций и методов.

```
m = x.foo  # <bound method Simple.foo of <__main__.Simple object
at 0x015981D8>>
```

Отличительной чертой методов является явная передача первого аргумента. По соглашению его принято именовать `self` и никак иначе. Этот аргумент отвечает за передачу экземпляра класса в метод. Поэтому при вызове метода `obj.method()` не нужно передавать этот аргумент вручную, это делается неявно. Эту конструкцию можно переписать в эквивалентную `Class.method(obj)`, вызвав метод у класса и, передав в качестве аргумента `self` экземпляр класса `obj`. Аналогом аргумента `self` является `this` в других языках программирования.

```
x.foo()
Simple.foo(x)
```

Строит отметить, что тип верхнего выражения – `method`, а второго – `function`.

Работа методов происходит следующим образом. Когда происходит обращение к атрибуту экземпляра, т.е. `obj.method()`, происходит поиск в классе экземпляра. Если имя обозначает атрибут, тип которого является функцией, то происходит создание объекта метода. Это происходит с помощью упаковки указателя объекта (экземпляра) и функции в новый объект – объект метода. В момент вызова метода с набором аргументов происходит создание нового набора аргументов из объекта экземпляра и списка остальных аргументов, в конце вызывается функция с этим набором аргументов.

1.2.1 Связанные и несвязанные методы

Стоит сразу упомянуть, что концепция несвязанных методов была удалена в версии языка 3.0. Изначально эта концепция рассматривалась в качестве способа обеспечения «равноправия» между всеми объектами и в том числе методами. Рассмотрим эту концепцию подробнее на следующем простом примере класса с парой методов.

```
class A:
```

```
    def __init__(self, x):
        self.x = x
    def foo(self, y):
        print(f'{self.x = }, {y = }')  # only for 3.8
```

Если методы рассматривать как объекты первого класса, то они должны поддерживать связывание с именами переменных и соответственно вызов как обычных функций. Рассмотрим следующий вариант вызова `b = A.spam`. В этом случае переменная `b` относится непосредственно к методу класса `A`, который на самом деле является функцией. Но методы немного отличаются от обычных функций наличием первого аргумента, в качестве которого передается экземпляр класса, в котором определен метод.

В результате было введено понятие как несвязанный метод. В версиях языка < 3.0 это был отдельный тип, который налагал ограничения на то, что первым аргументом должен быть экземпляр класса, в котором был объявлен метод. Таким образом, если нужно было вызвать `b` в качестве функции, потребовалось бы создать экземпляр класса `A` и передать его первым аргументом.

```
b = A.foo
a = A(42)
b(a, 4)  # self.x = 42, y = 4
```

Сейчас это просто функция, т.е. `type(b)` вернет `<class 'function'>`. И от ограничения на тип первого аргумента не накладываются ограничения, т.е. не обязательно передавать именно экземпляр класса, в котором был объявлен этот метод.

```
class B:
    pass
mock = B()
mock.x = 42
b(mock, 5)  # self.x = 42, y = 5
```

Такое поведение называется утиной типизацией, которая будет рассмотрена отдельно.

Связанные методы возникают, если создается имя, которое ссылается на метод конкретного экземпляра.

```
a = A(42)
b = a.foo
print(type(b))  # <class 'method'>
print(b)  # <bound method A.foo of <__main__.A object ...>>
```

Здесь переменная b ссылается на метод класса A, но ссылка получена через экземпляр a. В этом случае тип таких объектов уже method, а не function. Этот объект называется связанным методом. Сам объект представляет собой оболочку для объекта функции-метода. Эта оболочка неявно хранит ссылку на исходный экземпляр a, который был использован для получения метода. Таким образом, становиться доступным вызов b без передачи первого аргумента, он будет передан неявно самой оболочкой (как в декораторах).

```
b(5)  # self.x = 42, y = 5
```

Подробнее об этой концепции можно прочитать в блоге Гвидо ван Россума [1].

1.2.2 Методы класса, статические методы

В Python также есть методы класса и статические методы. Отличие методов класса от обычных методов заключается в том, что в качестве первого аргумента передается не ссылка на экземпляр класса, а ссылка на сам класс. Статические методы могут вообще не содержать никаких аргументов.

Рассмотрим пример класса для работы с датами. Пусть требуется хранить день, месяц и год. При этом требуется создавать объект даты из строкового представления, например, элементов, разделенных точкой. Кроме этого нужно иметь возможность проверить, является ли строка с датой корректной. Для простоты рассмотрим обычные правила для даты. Перейдем к реализации.

```
class Date:
    def __init__(self, day, month, year):
        self.day = day
        self.month = month
        self.year = year
```

```
    @classmethod
    def from_str(cls, date_as_str):
        day, month, year = map(int, date_as_str.split('.'))
        return cls(day, month, year)

    @staticmethod
    def is_date_valid(date_as_str):
        day, month, year = map(int, date_as_str.split('.'))
        return day <= 31 and month <= 12 and year <= 2038

print(Date.is_date_valid('19.01.2038'))  # True
date = Date.from_str('19.01.2038')
print(date.day, date.month, date.year)  # 19 1 2038
```

Разберем подробнее класс `Date`. Первым методом он содержит обычный метод инициализации экземпляра `__init__`. Далее идет метод с декоратором `@classmethod`, который указывает на то, что метод является методом класса. Этот метод принимает первым аргументом ссылку на класс. В этом случае аргумент принято именовать как `cls`. А вызывается метод непосредственно у класса, т.е. как `Date.from_str('19.01.2038')`. Здесь по аналогии со связанными методами, ссылка на класс передается неявно. Реализация метода `from_str` довольно проста. Сначала идет преобразования строки вида `'dd.mm.yyyy'` в три целых числа, затем вызывается конструктор класса, куда передаются эти значения, т.е. создается экземпляр класса `Date`.

Такой подход имеет несколько существенных плюсов. Во-первых, этот код теперь можно использовать повторно. Во-вторых, это хорошее применение объектно-ориентированной парадигмы программирования, а в частности инкапсуляции. Теперь при наследовании от класса `Date`, все потомки будут содержать метод `from_str`.

Далее в классе едет определение метода `is_date_valid`, который отвечает за проверку строки с датой на корректность. Здесь также строка разбивается на три числа, и после они проверяются по простым правилам. Перед объявлением этого метода указан декоратор `@staticmethod`, который указывает на то, что метод является статическим. Это означает, что метод, по

сути, является функцией и не принимает никаких обязательных аргументов, как метод экземпляра или метод класса. Из этого также следует, что методу не будет доступно состояние класса и экземпляра, т.е. он не может получить доступ к их атрибутам.

Статические методы удобно использовать, когда какая-либо функциональность связана с какими-либо классами, но не требует их реализации. В этом случае имеет место сгруппировать его с этими объектами.

1.3 Создание экземпляра

Создание экземпляра происходит в момент выполнения инструкции `obj = Class()`. В Python создание экземпляра происходит в два шага. Первым происходит создание непосредственно объекта экземпляра заданного класса, после чего этот экземпляр инициализируется. Рассмотрим эти шаги.

За создание нового объекта отвечает метод `__new__`. Этот метод статический, это говорит о том, что первым аргументом он принимает объект класса, однако применять соответствующий декоратор. Он первым вызывается при исполнении инструкции `obj = Class()`. Результатом выполнения `__new__` должен быть объект, обычно типа `Class`. Следом за методом создания объекта экземпляра, но до возвращения нового экземпляра пользователю, вызывается метод `__init__`, который отвечает за инициализацию атрибутов экземпляра. Он является методом экземпляра и первым аргументом принимает объект экземпляра. Метод `__init__` возвращает `None`, любой отличный результат выполнения будет порождать исключение `TypeError`. Рассмотрим на примере.

```
class A:
    def __new__(cls):
        print('NEW')
```

```
        return super().__new__(cls)
    def __init__(self):
        print('INIT')
a = A()  # NEW; INIT
```

Здесь при создании экземпляра `a = A()` происходит сначала вызов сначала `__new__`, затем `__init__`, что можно увидеть при выводе сообщений. Назначение функции super будет рассмотрено позднее.

По своей сути конструкция `a = A()` будет эквивалентна:

```
a = A.__new__(A)  # NEW
a.__init__()  # INIT
```

2 Наследование

Наследование это одна из концепций объектно-ориентированной парадигмы программирования, согласно которой тип данных может наследовать данные и функциональность некоторого существующего типа, способствуя повторному использованию компонентов программного обеспечения.

Существует два основных вида наследования: одиночное и множественное. Python поддерживает оба этих вида. Рассмотрим некоторые основные понятия в наследовании.

Подклассом (дочерним классом, наследником) принято называть класс, определенный через наследование от другого класса.

Класс, стоящий на вершине иерархии наследования, называют базовым классом. Иерархию наследования можно представить в виде дерева, у которого обычно один корневой узел, которым и является базовый класс.

В Python классы предки указываются в скобках сразу после имени класса в его определении. Рассмотрим простой пример наследования с классами, которые ничего не делают.

```
class A:  # базовый класс
    pass
```

```
class B(A):  # подкласс 1
    pass
class C(A):  # подкласс 2
    pass
class D(B):  # подкласс 3
    pass
```

Эти четыре класса можно представить в виде графа, изображенного на рисунке 1.

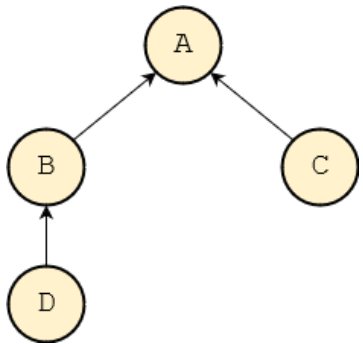


Рисунок 1 – Дерево классов

Здесь `A` является базовым классом, у которого два потомка – `B` и `C`. У класса `B`, в свою очередь, также есть потомок – класс `D`. Кроме этого `B` и `C` являются подтипами `A`, а `D` подтипом `B`. При этом если `B` подтип `A`, а `D` подтип `B`, то `D` также является подтипом `A`. Для проверки этого в Python есть встроенная функция `isinstance`, которая может проверять типы с учетом наследования. Также у классов есть специальный атрибут `__bases__`, который содержит кортеж классов-предков.

```
print(f'bases A: {A.__bases__}')  # (<class 'object'>,)
print(f'bases B: {B.__bases__}')  # (<class '__main__.A'>,)
print(f'bases C: {C.__bases__}')  # (<class '__main__.A'>,)
print(f'bases D: {D.__bases__}')  # (<class '__main__.B'>,)
```

Обратите внимание, что класс `A` неявно наследуется от базового класса `object`.

```
print(isinstance(a, object))  # True
print(isinstance(b, (A, object)))  # True
print(isinstance(c, (A, object)))  # True
```

```
print(isinstance(d, (B, A, object)))  # True
```

Функция `isinstance` на самом деле проверяет атрибут `obj.__class__` у объекта и сравнивает его со значением второго аргумента.

Аналог функции `isinstance`, но для классов называется `issubclass`, она первым аргументов принимает класс, а вторым один или несколько предков и возвращает `True`, если класс является подклассом указанных классов.

```
print(issubclass(D, (B, A, object)))  # True
```

Здесь есть интересный момент. Класс является подклассом самого себя.

```
print(issubclass(D, D))  # True
```

Сохранение базовых классов необходимо для процесса поиска атрибутов. Поиск осуществляется рекурсивно снизу вверх по иерархии наследования, начиная с класса у которого был указан этот атрибут, затем у его базовых классов и т.д.

2.1 Перегрузка методов

Производные классы могут переопределять методы своих базовых классов. Методы равноправны атрибутом, поэтому их поиск происходит аналогично. Переопределить метод можно двумя путями: полностью заменить метод базового класса с тем же именем и расширить его.

Полностью заменить метод можно путем его объявления и реализацией нового тела метода.

```
class A:
    def __init__(self):
        self._a = 42
    def get_a(self):
        return self._a
class B(A):
    def get_a(self):
        print(f'a = {self._a}')
```

```
a = A()
print(a.get_a())  # 42
b = B()
b.get_a()  # 'a = 42'
```

В этом примере класс A реализует метод get_a, который просто возвращает атрибут _a. Это так называемый *getter*. Класс B, являющийся наследником A, переопределяет этот метод. При этом метод get_a не изменяется в классе A. В этом примере, также виден эффект от применения наследования. Он заключается в том, что в классе B появился атрибут _a, который в нем не был определен, а унаследовался от A. Когда вызывается метода get_a у экземпляра b, происходит поиск этого метода в словаре (__dict__) экземпляра. После того, как он не будет там найден, поиск переходит в словарь класса.

```
print(b.__dict__)  # {'_a': 42}
print(B.__dict__)  # {..., 'get_a': <function B.get_a ...>, ...}
```

Бывают случаи, когда нужно дополнить логику работы метода и нет необходимости полностью реализовывать метод базового класса. В этом случае используется магическая функция super.

```
class A:
    def __init__(self, a=42):
        self._a = a
class B(A):
    def __init__(self, a=0):
        super().__init__(a=a)
        self._new_a = a
b = B()
print(b._a, b._new_a)  # 0 0
```

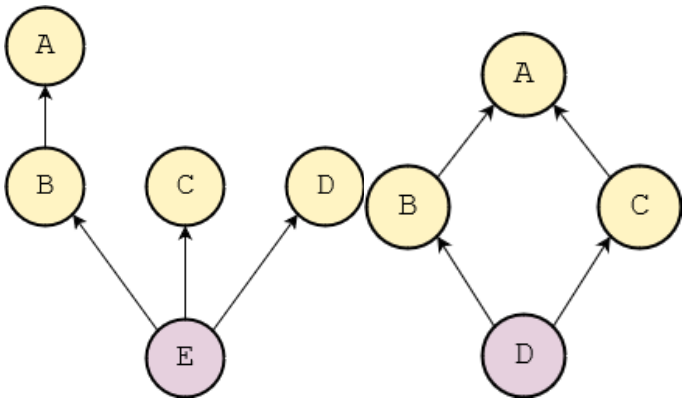
Здесь класс A определяет метод __init__, который устанавливает единственный атрибут. В свою очередь класс B переопределяет этот метод, но не полностью, а используя __init__ из класса A, При этом используется функция super.

Функция `super` предназначена для поиска родительских классов. И на самом деле принимает два аргумента, первым является класс, а вторым экземпляр класса. Непосредственно в теле класса можно использовать эту функцию без явной передачи этих аргументов, интерпретатор сделает это сам. Но при использовании ее вне класса нужно вручную передавать аргументы. Функция super возвращает специальный объект, который выполняет делегирования вызова методов родительскому классу. В данном примере вызов `__init__(a=a)` будет перенаправлен в класс `A`, но ее также можно использовать в других методах.

Классы переопределять довольно большой набор специальных методов, например методы сравнения (`__eq__`, `__le__` и др.).

2.2 Множественное наследование

Python также поддерживает множественное наследование. Оно заключается в наличии у класса потомка нескольких базовых классов. Его схема приведена на рисунке 2А. Зачастую наличие множественного наследования ведет к серьезным проблемам. В частности множественное наследование может приводить к ромбовидному наследованию, схема которого изображена на рисунке 2Б.



А Б

Рисунок 2 – Множественное наследование

В Python множественное наследование имеет синтаксическую поддержку и определяется указанием нескольких базовых классов, разделенных запятыми:

```
class MyClass(Base1, Base2, ..., BaseN)
```

алгоритм C3

__mro__, порядок разрешения методов

2.2.1 Классы-примеси (Mix-in)

Классы-примеси или *mixin* это базовые классы, которые реализуют конкретную функциональность.

```
class Mixin:
    def complex_method(self):
        return complex_functionality(self)
```

Примеси предназначены для создания дочерних классов, т.е. классов унаследованных от классов-примесей, для обеспечения дополнительной функциональности. Понятие *mixin* предполагает «смешивание» с другим кодом. В результате классы-примеси не предполагают создание экземпляров, так как в большинстве случаев они не имею атрибутов, а реализуют только методы. Например, следующее применение *mixin*'а не имеет смысла, в этом случае можно было просто создать функцию `complex_method` без использования класса.

```
obj = Mixin()
```

2.3 Поиск атрибутов

Как уже упоминалось, при создании объекта класса происходит создание нового пространства имен, которое представляется в виде словаря. Это говорит о том, что все атрибуты не только класса, но и экземпляра доступны в виде словаря.

```
class A:
    foo = 42
    def __init__(self):
        self.attribute = 'spam'

    def bar(self):
        print("It's bar")
```

Убедиться в этом можно, используя специальный атрибут `__dict__`, который есть как у классов, так и у экземпляров. Есть также встроенная функция `vars`, которая возвращает `__dict__`.

```
a = A()
print(A.__dict__)  #  {..., 'foo': 42, '__init__': <function
A.__init__ at 0x015CF3D0>, 'bar': <function A.bar at 0x015CF340>,
'__dict__': <attribute '__dict__' of 'A' objects>, ..., '__doc__':
None}
print(a.__dict__)  # {'attribute': 'spam'}
```

Видно, что у класса есть помимо атрибутов и методов, определенных в его теле, содержится еще ряд специальных методов и атрибутов, таких как строка документации, модуль и прочее.

В связи с тем, что атрибуты хранятся в словаре, то поиск, изменение, добавление и удаление атрибута это операции над словарем.

```
a = A()
a.__dict__['pi'] = 3.14159
print(a.pi)  # 3.14159
del a.__dict__['pi']
print('pi' in a.__dict__)  # False
```

Кроме этого поиск атрибутов происходит динамически в момент выполнения программы.

Представление пространства имен в виде словаря несет некоторые накладные расходы по памяти. Также зачастую динамическое управление атрибутами не нужно или нежелательно, например, нужно запретить удаление и добавление новых атрибутов. Решить эти проблемы можно, жестко зафиксировав структуру класса. Для этого используется механизм `__slots__`.

Переменная `__slots__` задает атрибуты экземпляров. Она обычно определяется сразу после объявления класса и должна содержать список строк, или другую итерируемую коллекцию строк, где строки будут именами атрибутов.

```
class Base:
    __slots__ = ('foo', 'bar')
```

В этом примере у экземпляра класса `Base` будет только два атрибута `foo` и `bar`.

```
b = Base()
b.foo, b.bar = 1, 2
b.boo = 42  # AttributeError
```

Использование `__slots__` позволяет получить более быстрый доступ к атрибутам и экономит место в памяти. Эти достоинства достигаются за счет хранения ссылок на атрибуты в специальных слотах, а не в `__dict__`. Также класс не будет содержать `__dict__` вовсе, если в родительских классах также были объявлены `__slots__`.

```
class A:
    def __init__(self):
        self.a = 0
class B(A):
    __slots__ = 'b', 'c'
b = B()
print(b.__dict__)  # {'a': 0}
```

Еще одним подводным камнем в использовании `__slots__` и наследования заключается в том, что нужно объявлять каждый атрибут только в одном `__slots__`. Если некоторые атрибуты указаны в нескольких переменных, то это не вызовет исключения, однако эти объекты будут занимать больше места в памяти.

```
class Base:
    __slots__ = ('foo', 'bar')
class A(Base):
    __slots__ = ('baz', )
class B(Base):
    __slots__ = ('foo', 'bar', 'baz')


from sys import getsizeof
print(getsizeof(A()), getsizeof(B()))  # 28 36
```

В результате, основным требованием для использования слотов можно выделить отсутствие классов с `__dict__` в иерархии наследования.

При использовании механизма слотов можно разрешить использование `__dict__`, просто указав `'__dict__'` как элемент в переменной `__slots__`.

Использование множественного наследования и `__slots__` может вызвать большие проблемы. Например:

```
class BaseA:
    __slots__ = ('a', )
class BaseB:
    __slots__ = ('b', )
class C(BaseA, BaseB):  # TypeError
    __slots__ = ()
```

В этом случае создание класса `C` вызовет исключение. В случае наличия контроля над базовыми классами решить эту проблему можно, удалив у них `__slots__` или сделав их пустыми. Также можно воспользоваться абстрактными классами, о которых речь пойдет позже.

```
from abc import ABC

class AbstractA(ABC):
    __slots__ = ()
class BaseA(AbstractA):
    __slots__ = ('a',)

class AbstractB(ABC):
    __slots__ = ()
class BaseB(AbstractB):
    __slots__ = ('b',)

class Child(AbstractA, AbstractB):
    __slots__ = ('a', 'b')
c = Child() # no problem!
```

Подробнее с механизмом `__slots__` можно ознакомиться здесь [2].

2.4 Управление доступом (сокрытие)

Сокрытие это принцип проектирования, заключающийся в разграничении доступа различных частей программы к внутренним компонентам друг друга. В одних языках (например, C++) термин тесно пересекается (вплоть до отождествления) с инкапсуляцией, в других эти понятия абсолютно независимы. В некоторых языках (например, Smalltalk или Python) сокрытие отсутствует, хотя возможности инкапсуляции развиты хорошо.

В Python нет разделения атрибутов на публичные (public), приватные (private) и защищенные (protected). Однако в сообществе языка принято соглашение об именовании атрибутов классов. Оно гласит, что атрибуты для внутреннего использования начинают с префикса нижнее подчеркивание.

```
class A:
    public = 42
```

```
    _private = 'foo'
```

Такая возможность не обеспечивается интерпретатором, поэтому пользователь может получить к таким атрибутам доступ через точечную нотацию по аналогии с обычными атрибутами.

```
a = A()
print(a.public, a._private)  # 42 foo
```

Обращение к атрибутам, которые начинаются с нижним подчеркиванием, говорит о том, что пользователь знает, что делает.

Существует еще один способ сокрыть атрибут от обращения извне. Для этого необходимо добавить к атрибуту префикс с двумя нижними подчеркиваниями. В этом случае интерпретатор изменит имя атрибута, для избегания конфликтов из-за совпадения имен. Это называется искажением имени (name mangling).

```
class A:
    def __init__(self):
        public = 42
        _private = 'foo'
        __very_private = 'bar'
```

После создания экземпляра атрибута с именем `__very_private` существовать не будет.

```
a = A()
print(a.__vary_private)  # AttributeError
```

Интерпретатор преобразует его специальным образом. Это помогает избежать проблем, связанных с конфликтом имен при переопределении переменных в классах наследниках.

```
print(dir(a))  # ['_A__very_private', ..., '_private', 'public']
print('_A__very_private' in dir(a))  # True
print(a._A__very_private)  # bar
```

Это и есть искажение имен в Python и оно также распространяется на методы. Рассмотрим поведение при расширении базового класса `A`.

```
class B(A):
```

```
    def __init__(self):
        super().__init__()
        self.public = 'spam'
        self._private = 'spam'
        self.__very_private = 'spam'
```

Атрибуты `public` и `_private` ожидаемо будут переопределены. А атрибута `__very_private` по-прежнему существовать не будет.

```
b = B()
print(b.public)  # spam
print(b._private)  # spam
print(b.__very_private)  # AttributeError
```

Обратиться к атрибуту `__very_private` можно через измененное имя.

```
print(b._B__very_private)  # spam
print(b._A__very_private)  # bar
```

Но довольно очевидно, что атрибут `__very_private` класса-предка `A` переопределен не был.

Стоит сказать несколько слов о применении. Разделять атрибуты и методы на приватные и публичные с помощью принятого соглашения, используя префикс с одним нижним подчеркиванием, является хорошей практикой. Применять же имена вида `__var` стоит с осторожность, всегда держа в голове то, как это имя будет преобразовано интерпретатором.

3 Утиная типизация (Duck typing)

Утиной типизацией (duck typing) или неявной типизацией называют определение факта реализации интерфейса объектом без явного указания или наследования этого интерфейса, а по реализации полного набора его методов. Идея заключается в том, что неважно как выглядят данные, важно то, что можно с ними делать. Зачастую утиная типизация является результатом применения «утиного теста» в программировании. Этот тест

гласит: «Если что-то выглядит как утка, плавает и крякает как утра, то это, вероятно, и есть утка».

Одним из простейших примеров является сложение двух чисел. В статически типизированных языках складывать можно лишь числа одного типа. Нельзя просто так взять и прибавить к целому числу число с плавающей точкой.

Python позволяет каждому объекту определить, что значит операция сложения. По своей сути выражение `1 + 3` является синтаксическим сахаром для вызова метода `int.__add__(1, 3)` целочисленного типа. Это значит, что если определить метод `__add__` для произвольного класса, то можно будет делать все что угодно. Большинство операций в Python это синтаксический сахар для специальных методов и функций.

Рассмотрим еще один простой пример.

```
class RpgCharacter:
    def __init__(self, weapon):
        self._weapon = weapon
    def battle(self):
        self._weapon.attack()
```

Здесь класс `RpgCharacter` получает объект `weapon` и в методе `battle` вызывает у него метод `attack`. Класс `RpgCharacter` не зависит от конкретной реализации объекта `weapon`. Это может быть любой объект, начиная от меча, заканчивая посохом Гендальфа. Есть только одно ограничение – наличие метода `attack`.

4 Декораторы классов

Пример. Паттерн «одиночка» или singleton.

```
def singleton(cls):
    instances = {}
    def get_instance():
```

```
        if cls not in instances:
            instances[cls] = cls()
        return instances[cls]
    return get_instance

@singleton
class A:
    def __init__(self):
        self._a = 42

a = A()
b = A()
print(a._a, b._a)
print(a is b)
a._a = "It's a trap"
print(f'_a from a {a._a = }')
print(f'_a from b {b._a = }')
```

5 Некоторые «магические» атрибуты и методы классов

В Python многие объекты имеют так называемые «магические» или атрибуты и методы. Они имеют специальный формат имени, а именно, оно начинается и заканчивается двумя подчеркиваниями. За это их еще называют дандер-атрибутами. Вот некоторые из них:

а) `__doc__` хранит документацию класса, и используется при вызове функции help;

б) `__name__` содержит имя класса;

в) `__modle__` содержит строку с именем модуля, или `'__main__'` если модуль был запущен, а не импортирован;

г) `__bases__` содержит кортеж базовых классов, в котором всегда будет элемент `<class 'object'>`;

д) `__dict__` – словарь со всеми атрибутами класса;

е) `__class__` содержит ссылку на объект класса текущего объекта.

Подробнее рассмотрим специальные методы для управления строковым представлением объекта.

Когда вы пытаетесь распечатать в консоли экземпляр какого-либо класса, то, скорее всего, получите неудовлетворительный результат. По умолчанию выводится только строка, содержащая имя класса и его уникальный идентификатор.

```
class Point:
    def __init__(self, x, y):
        self._x = x
        self._y = y
p = Point(0, 0)
print(p)  # <__main__.Point object at 0x06866718>
```

Для более удобно представления объекта в виде строки существуют дандер-методы `__repr__` и `__str__`. Метод `__str__` используется при распечатке объекта через функцию `print` и `str` и предназначен для восприятия человеком, т.е. его результат должен быть в первую очередь удобочитаемым.

```
class Point:
    def __init__(self, x, y):
        self._x = x
        self._y = y
    def __str__(self):
        return f'({self._x}; {self._y})'
p = Point(0, 0)
print(p)  # (0; 0)
```

В случае с методом `__repr__` идея состоит в том, что его результат должен быть, прежде всего, однозначным. Его результат в основном используется разработчиками и для отладки. При реализации этого метода стоит ориентироваться на то, чтобы его результат можно было скопировать и вставить в консоль и исполнить как фрагмент Python кода, который вернет

объект нужного класса. Также для этого метода есть специальная функция `repr()`.

```
class Point:
    def __init__(self, x, y):
        self._x = x
        self._y = y
    def __repr__(self):
        return f'{self.__class__.__name__}({self._x}, {self._y})'
p = Point(0, 0)
print(p)  # Point(0, 0)
```

Методы `__repr__` и `__str__` имеют еще некоторые различия, с которыми можно ознакомиться в документации. В заключение стоит порекомендовать реализовывать хотя бы метод `__repr__`, даже если он не будет восстанавливать полное состояние объекта. Даже если опустить реализацию `__str__`, то при его отсутствии будет вызван `__repr__`.

6 Дескрипторы

подробнее про __slots__

6.1 Свойства

Свойства (property) это специальные

Снаружи класса свойства выглядят как атрибуты, но их значения вычисляются динамически во время выполнения, а не хранятся в экземпляре. При этом в __dict__ не будет создан ключ для этого атрибута. Это позволяет не хранить то, что можно вычислить.

7 Метаклассы

8 Абстрактные классы

абстракции

9 Полезные ссылки

1. First-class Everything [Электронный ресурс] // The History of Python. – Режим доступа: https://python-history.blogspot.com/2009/02/first-class-everything.html

2. Usage of __slots__? [Электронный ресурс] // stackoverflow.com. – Режим доступа: https://stackoverflow.com/questions/472000/usage-of-slots

3.