

Оглавление

2	Тестирование и отладка программ	2
2.1	Введение в ООП	2
2.1.1	Чем хорошо объектно-ориентированное программирование?	2
2.1.2	Отличие класса от объекта	3
2.1.3	Отличие интерфейса класса от реализации	4
2.2	Парадигмы ООП	6
2.2.1	Инкапсуляция и полиморфизм в Python	6
2.2.2	SOLID принципы ООП	7
2.3	Разработка системы классов	8
2.3.1	Парадигма наследования в Python	8
2.3.2	Абстрактные классы и библиотека abc	9

Неделя 2

Тестирование и отладка программ

2.1 Введение в ООП

2.1.1 Чем хорошо объектно-ориентированное программирование?

Объектно-ориентированный подход:

- Объектно-ориентированный образ мышления;
- Переход от переменных и функций к объектам;
- Взаимодействие между объектами.

Когда вы описываете, например, движение автомобиля по дороге, то оперируете не огромным количеством переменных, которые изменяются по каким-то законам, а вы оперируете целым объектом — автомобилем, который выполняет какие-то действия.

Класс состоит из:

- переменных, которые хранятся и изменяются вместе с классом;
- методов — функций, которые выполняют какие-то действия с классом.

ООП применяется повсеместно прежде всего, благодаря простоте и естественности использования: мы один раз пишем (или импортируем из библиотеки) какой-то класс, который обладает каким-то состоянием и может его изменять. А потом из различных классов и объектов мы строим большое приложение как из кубиков Lego. При использовании объектов класса, мы концентрируемся на том, как их связать друг с другом, а не на том, как они использованы и реализованы внутри.

Другим достоинством ООП является то, что важные компоненты собраны в одном месте и код становится структурированным. Кроме того, инкапсуляция позволяет защищать некоторые данные от несанкционированного доступа, что может уберечь систему от весьма существенных поломок.

Объектно-ориентированный подход позволяет быстро и легко расширять и обновлять существующую систему, не внося в неё существенных изменений.

Отдельным важным достоинством является возможность повторного использования ООП систем: объект, решающий некоторую распространенную задачу, может быть встроен как компонент в большое количество различных систем, которым требуется решений этой задачи. Это сильно упрощает и ускоряет разработку приложений.

Наряду с этим, у ООП есть главный недостаток — слишком сложная система классов. Для полноценного использования средств ООП требуется хорошо понимать его возможности и парадигмы. Очень сложно корректно спроектировать систему классов таким образом, чтобы она работала действительно эффективно. В различных библиотеках содержатся десятки классов, имеющие сотни методов и способные взаимодействовать друг с другом.

И поскольку всё, что есть в Python, является объектами, ООП играет очень важную роль. Целые числа, строки, списки, кортежи и словари — это всё объекты. Даже функции и классы сами по себе являются целыми объектами. Когда вы складываете два числа, вы вызываете метод `add` от целого числа. Когда вы добавляете элемент в список, вы вызываете метод списка `append`.

2.1.2 Отличие класса от объекта

Жизненный путь **объекта**:

1. Создаётся в памяти;
2. Происходит инициализация вызовом метода `init`;
3. Ссылка на объект сохраняется в некоторую переменную;
4. Мы свободно пользуемся объектом, обращаясь к нему по ссылке: вызываем его методы и просматриваем его состояние;
5. Когда на объект нет внешних ссылок, он удаляется.

Удаление происходит в одной из двух ситуаций. Поскольку у каждого объекта есть счётчик внешних ссылок на него, при отсутствии обращений к объекту, он будет удален с использованием механизма `DECREF`.

Если же счётчик не достиг нуля, то `Garbage Collector` освобождает память от объектов, на которые нет внешних ссылок из программы (но могут быть ссылки из других объектов, не связанных с программой).

Определения понятий:

- Объект — это структура, которая хранится в памяти, имеет некоторое состояние, умеет его изменять и взаимодействовать с другими объектами при помощи методов;
- Класс — это описание структуры объекта. В нём описаны переменные и методы объекта, как он будет создаваться в памяти и что необходимо сделать при его удалении;

- Объект — класс, позволяющий хранить информацию о других объектах: как будет создаваться и что сможет сделать некоторый объект. Кроме того, он может самостоятельно создавать объекты других типов.

2.1.3 Отличие интерфейса класса от реализации

Интерфейс (абстракция) — это совокупность всех способов доступа и взаимодействия с некоторым объектом. Интерфейс объекта состоит из публичных методов и переменных, доставшихся классу от его родительских классов и из его собственных публичных полей. Интерфейс будет видеть пользователь объекта, и именно он будет описан в пользовательской документации.

Разработка интерфейса:

- Интерфейс должен быть продуманным;
- Прежде всего надо думать о пользователе;
- Все необходимые методы взаимодействия с объектом должны входить в интерфейс;
- Все служебные методы и переменные в интерфейс не входят.

Не забывайте две важные концепции из дзена Python "Красивое лучше, чем уродливое." и "простое лучше, чем сложное." Таким образом интерфейс это оболочка, а внутренности это реализация.

Разберём отличия интерфейса и релацизации на примере списка (list):

Метод	Что делает
<code>list.append(x)</code>	Добавляет элемент в конец списка
<code>list.extend(L)</code>	Расширяет список <code>list</code> , добавляя в конец все элементы списка <code>L</code>
<code>list.insert(i, x)</code>	Вставляет на <code>i</code> -ый элемент значение <code>x</code>
<code>list.remove(x)</code>	Удаляет первый элемент в списке, имеющий значение <code>x</code> . <code>ValueError</code> , если такого элемента не существует
<code>list.pop([i])</code>	Удаляет <code>i</code> -ый элемент и возвращает его. Если индекс не указан, удаляется последний элемент
<code>list.index(x, [start [, end]])</code>	Возвращает положение первого элемента со значением <code>x</code> (при этом поиск ведётся от <code>start</code> до <code>end</code>)
<code>list.count(x)</code>	Возвращает количество элементов со значением <code>x</code>
<code>list.sort([key=функция])</code>	Сортирует список на основе функции
<code>list.reverse()</code>	Разворачивает список
<code>list.copy()</code>	Поверхностная копия списка
<code>list.clear()</code>	Очищает список

Это всё интерфейс. Но реализация списка в Python это **3000** строк на языке C. Там написано, сколько памяти и какого типа следует выделять при создании списка, каким образом в нём хранятся переменные и как получить к ним доступ.

Например, при вызове метода **append()**, в конец списка вставляется новый элемент. Причём, если место заканчивается, под список выделяется больше памяти, и чем больше список, тем больше выделяется. При вставке элемента в середину списка, кроме выделения памяти придётся сдвинуть последующие элементы на 1.

При выталкивании элемента (метод **pop()**), возвращается элемент, лежащий в последней ячейке списка, и его длина уменьшается на 1. Но реально-го удаления объекта из списка не происходит до тех пор, пока его длина не станет меньше половины размера выделенной памяти. Только тогда данные будут очищены, а память освобождена.

Интерфейс — это совокупность всех публичных переменных и методов класса. Реализация — это внутреннее устройство объекта, которое не видит пользователь.

2.2 Парадигмы ООП

2.2.1 Инкапсуляция и полиморфизм в Python

Парадигмы ООП:

- наследование;
- инкапсуляция;
- полиморфизм.

Наследование позволяет создавать сложные системы классов. Основные понятия в наследовании это класс-родитель и класс-потомок, наследующий все публичные методы и поля класса-родителя. Это позволяет строить иерархии классов.



Рис. 2.1: Наследование

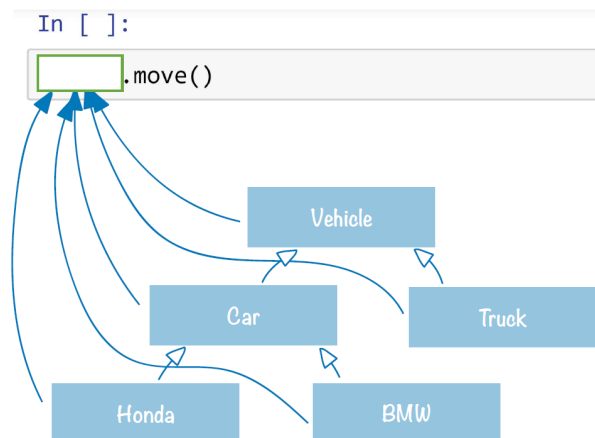


Рис. 2.2: Полиморфизм

Полиморфизм тесно связан с наследованием, интерфейсом и реализацией. Этот принцип позволяет использовать один и тот же код для работы с различными объектами, имеющими одинаковый интерфейс, но обладающими различной его реализацией. Такая ситуация как раз часто возникает при наследовании: мы можем работать с классами-потомками так же, как работали бы с родительским классом.

Пусть у нас есть объект, который мы хотим описать. В процедурном программировании это бы делалось с помощью переменных и функций. В случае ООП мы упаковываем их в единый компонент. Это и есть **инкапсуляция**.

Есть мнение, что в Python нет инкапсуляции, но это не верно. Часто под инкапсуляцией понимают сокрытие данных от пользователя - скрытые методы будут доступны только самому экземпляру класса. Сокрытие данных позволяет, во-первых, построить простой и удобный интерфейс класса и, во-вторых, упростить его реализацию.

Разберём 1 случай: в вашем классе реализован важный для работы системы служебный метод, который бесполезен для пользователя. Тогда этот

метод можно скрыть, сделав приватным.

Во 2 случае рассмотрим класс «котик», у которого есть публичные поля возраст и вес. Без проверок входных данных в эти поля можно ввести отрицательные значения, что бессмысленно и может вызвать ошибки. Однако, можно сделать атрибут приватным, а доступ к нему осуществлять с использованием специальных методов — **getter()**(который возвращает значение атрибута) и **setter()**(который устанавливает значение атрибута, при прохождении проверки входных данных). Кроме того, приватные поля не наследуются.

Псевдоскрытие данных в Python устроено так: перед приватными полями следует поставить два нижних подчеркивания (`__private`). Если попытаться прочитать эти поля у экземпляра класса Python выдаст ошибку:

```
class SampleClass:
    def __init__(self):
        self.public = "public"
        self.__private = "private"

    def public_method(self):
        print(f"private data: {self.__private}")

c = SampleClass()
c.public
'public'
c.__private
AttributeError: 'SampleClass' object has no attribute '__private'
c._SampleClass__private
'private'
```

Просто Python автоматически переименовывает приватные поля по правилам: нижнее подчеркивание, название класса, название приватного поля с двумя нижними подчеркиваниями в начале. Посмотреть список всех полей можно командой `dir`.

2.2.2 SOLID принципы ООП

Частые проблемы ООП:

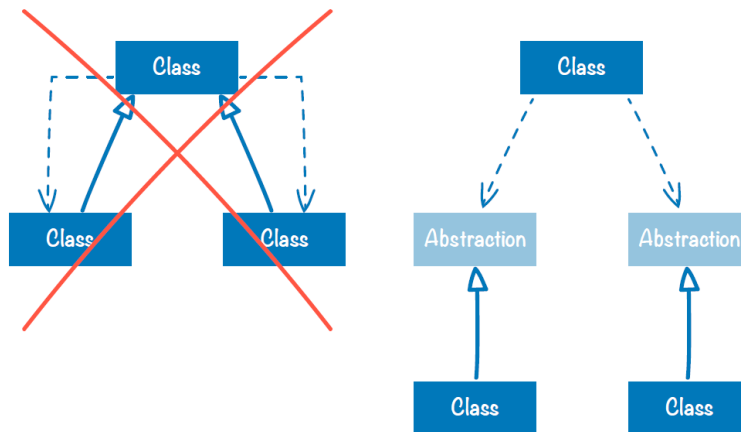
- негибкие системы;
- избыточные зависимости между компонентами;
- ненужная функциональность;
- невозможность повторного использования.

Пять принципов создания качественных систем — **SOLID**:

- **Single responsibility** — у каждого объекта должна быть только одна ответственность и всё его поведение должно быть направлено на обеспечение только этой ответственности;
- **Open/closed** — классы должны быть открыты для расширения (новыми сущностями), но закрыты для изменения;
- **Liskov substitution** (принцип Барбары Лисков) — функции, которые используют базовый тип должны иметь возможность использовать его подтипы не зная об этом;
- **Interface segregation** (принцип разделения интерфейсов) — клиенты не должны зависеть от методов, которые они не используют;
- **Dependency inversion** (принцип инверсии зависимостей) — модули верхних уровней не должны зависеть от модулей нижних уровней. Оба типа модулей должны зависеть от абстракции. Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.

Разберём подробнее последний принцип. Допустим, что модули верхних уровней напрямую зависят от модулей нижних уровней. Внесём изменение в модуль нижнего уровня. Так как от него зависят модули верхних уровней, их так же придётся менять. Таким образом, придётся переписать часть программы. Если же добавить абстракции, от которых зависят модули обоих уровней, то прямой зависимости нет, и изменения в модулях одного уровня не затронут

другой.



2.3 Разработка системы классов

2.3.1 Парадигма наследования в Python

Пусть у нас есть простая иерархия классов: родитель и потомок, который наследует все методы родительского класса (за некоторым исключением).

При этом у него могут быть свои методы и атрибуты, отличные от родительского класса. Некоторые из методов родительского класса у потомка могут быть переопределены и выполнять другую задачу.

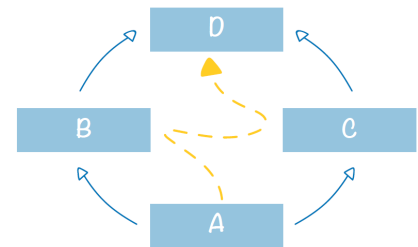
Мы бы могли реализовать это и без наследования, однако в таком случае пришлось бы переписывать в "потомка" интерфейс "родителя" и полностью его реализовать. Хуже, когда часть интерфейса, которая должна быть общей, изменяется: переписывать нужно одновременно все места, где должен быть общий код.

В этом случае и спасает наследование. Потомку передаются публичные методы и переменные родителя, но приватные поля не наследуются.

Инициализатор класса (метод **init**) также наследуется. Однако если его переопределить у класса потомка, при создании объекта метод **init** родительского класса вызван не будет. Чтобы всё-таки вызвать этот метод, нужно явно прописать его вызов в соответствующем методе класса потомка.

Но не во всех случаях использование сложной иерархии классов оправдано. Чем структура сложнее, тем тяжелее поддерживать программу и читать её код. Например, линейная структура, в которой каждый следующий класс наследуется от предыдущего (при условии что система классов не будет расширяться), является избыточной.

Python поддерживает множественное наследование, благодаря которому класс потомок может обладать функциональностью нескольких классов родителей. Но и здесь есть подводные камни, например: "Ромб смерти". А наследуется от В и С. А они в свою очередь от общего предка D, в котором есть некоторый метод `my_method`, переопределённый в В и С по-разному. Тогда непонятно, из какого родительского класса потомку А брать этот метод. С версии 2.3 установлен порядок, по принципу С3-линеаризации: в нашем случае, в А будет реализация метода в первом из классов, где он будет явно определён.



2.3.2 Абстрактные классы и библиотека abc

Абстрактные классы не имеют экземпляров, они описывают некоторый интерфейс (например животное - абстрактный класс, от которого наследуются ежик, котик и утка). Причём мы можем общаться со всеми потомками класса используя этот интерфейс. Когда мы объявляем класс абстрактным, мы говорим системе следить, чтобы экземпляров класса не было.

Кроме того, абстрактные методы реализуются не в родительском классе, а в классе-потомке. Но надо следить, чтобы они действительно везде были реализованы. **Абстрактные методы** — это методы, содержащие только определение без реализации.

Виртуальный класс — класс, который при множественном наследовании не

включается в классы-потомки, а заменяется ссылкой в них. Но в Python нет явного управления памятью, и о них сложно говорить. Во многих ООП языках класс-наследник является просто копией родительского класса с некоторыми дополнительными методами. При множественном наследовании это может приводить к неочевидным проблемам, например, "ромб смерти". В классах B и C будут содержаться копии метода, определённого в A, и они будут считаться отдельными методами. А в Python такой проблемы нет.

Реализация **виртуальных функций** определяется уже на этапе выполнения. Т.е. в ООП языках мы говорим, что на данном месте будет вызван некоторый метод, но какой именно метод, мы определим хотим определить только когда дойдём до места. В Python все методы считаются виртуальными и такая концепция в нём не рассматривается.

```
from abc import ABC, abstractmethods

class A:
    @abstractmethod
    def do_something(self):
        print("Hi!")

a = A()
a.do_something()
```

Hi!

Но мы хотели, чтобы нельзя было создать экземпляр абстрактного класса. Абстрактный метод должен быть реализован не в самом методе, а в его потомках. Попробуем это сделать:

```
class A(ABC):
    @abstractmethod
    def do_something(self):
        print("Hi!")

a = A()
```

TypeError: Can't instantiate abstract class A with abstract method... Вылетает ошибка: мы не можем создать экземпляр класса, в котором содержится нереализованный абстрактный метод. Создадим класс B, который будет наследоваться от A. Реализуем в B метод do_something_else(), который будет что-то печатать. При этом мы не реализовали метод do_something.

```
class B(A):
    def do_something_else(self):
        print("Hello")
```

```
b = B()
b.do_something()
```

TypeError: Can't instantiate abstract class B with abstract methods do_something

У нас вылетает та же ошибка: абстрактный метод не реализован.

```
class B(A):
    def do_something(self):
        print("Hi2!")

    def do_something_else(self):
        print("Hello")

b = B()
b.do_something()
b.do_something_else()
```