КОД #СТАТЬИ 8 фев 2023

Краткий курс ООП на Python: как избежать путаницы в коде

Самая популярная парадигма современной разработки: обучаем питонистов на кошечках, напитках и вечеринках.

Поделиться

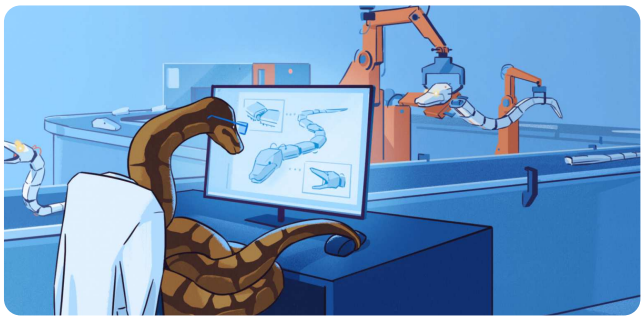


Иллюстрация: Катя Павловская для Skillbox Media

Иван Стуков

Журналист, изучает Python. Любит разбираться в мелочах, общаться с людьми и понимать их.

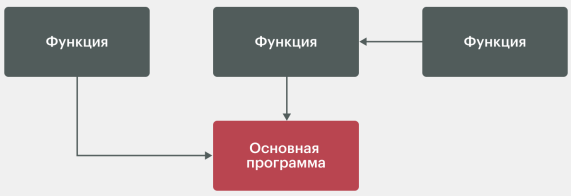
Объектно-ориентированное программирование применяют практически все крупные компании, потому что эта методика упрощает разработку. Но в то же время её боятся многие начинающие разработчики. Поэтому в этой статье мы покажем, что это на самом деле не так уж и сложно.

Зачем придумали ООП

Краеугольное понятие в ООП — объект. Это такой своеобразный контейнер, в котором сложены данные и прописаны действия, которые можно с этими данными совершать.

Чтобы понять, чем объекты так полезны и для чего их изобрели, сравним ООП с другой методикой разработки — процедурной. В ней весь код можно поделить на два вида: основную программу и вспомогательные функции, которые могут вызываться как программой, так и другими функциями:

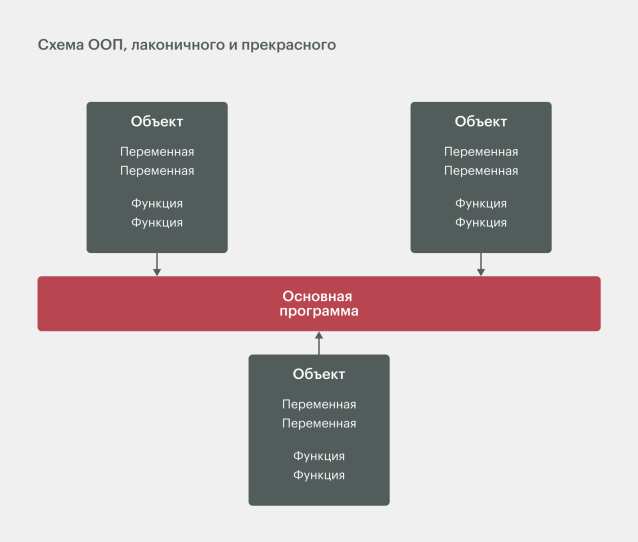
Схема взаимодействия функций с основной программой и между собой



Инфографика: Skillbox Media

У такого программирования есть существенный недостаток — части кода сильно зависят друг от друга. Например, основная программа вызывает функцию, та вызывает вторую, та, в свою очередь, — третью. При этом, допустим, вторую функцию могут параллельно вызывать ещё несколько других, а также основная программа. Схематически вся эта процедурная путаница представлена на рисунке:

а объекты, внутри которых уже лежат собственные переменные и функции. Так выстраивается более иерархичная структура. Переменные внутри объектов называются полями, или **атрибутами**, а функции — **методами**.



Инфографика: Skillbox Media

Объекты независимы друг от друга и самодостаточны, так что, если мы сломаем что-то в одном объекте, это никак не отразится на других. Более того: даже если мы полностью изменим содержание объекта, но сохраним его поведение, весь код продолжит работать.

Как работают классы

Каждый объект в ООП строится по определённому классу — абстрактной модели, описывающей, из чего состоит объект и что с ним можно делать.

Например, у нас есть класс «Кошка», обладающий атрибутами «порода», «окрас», «возраст» и методами «мяукать», «мурчать», «умываться», «спать». Присваивая атрибутам определённые значения, можно создавать вполне конкретные объекты.

Допустим:

- Порода = абиссинская.
- Окрас = рыжий.
- Возраст = 4.

Таким образом мы можем создать сколь угодно много разных кошек:

Инфографика: Skillbox Media

При этом любой объект класса «Кошка» (неважно, рыжая она, серая или чёрная) ...мяукать, мурчать, умываться и спать — если мы пропишем соответствующие ...

Принципы ООП на Python

Всё объектно-ориентированное программирование строится на четырёх понятиях: инкапсуляции, наследовании, полиморфизме и абстракциях. Поэтому давайте объявим наш класс «Кошка» и будем объяснять ООП на нём:

```
class Cat():
  def __init__(self, breed, color, age):
     self.breed = breed
     self.color = color
     self.age = age

  def meow(self):
     print('Мяу!')

  def purr(self):
     print('Мрррр')
```

Метод `__init__` — инициализатор класса. Он вызывается сразу после создания объекта, чтобы присваивать значения динамическим атрибутам. `self` — ссылка на текущий объект, она даёт доступ к атрибутам и методам, с которыми вы работаете. Её аналог в других языках программирования — `this`.

Примечание 1. Слово `self` общепринятое, но не обязательное, вместо него можно использовать любое другое. Однако это может запутать тех, кто будет читать ваш код.

Примечание 2. Названия классов принято писать с прописной буквы, а объектов — со строчной.

Итак, мы создали класс `Cat`, в котором объявили три атрибута: порода — `breed`, цвет — `color` и возраст — `age`. А ещё добавили два метода, чтобы наша кошка умела мяукать — `meow()` и мурчать — `purr()`.

Давайте создадим пару объектов нашего класса:

```
cat1 = Cat('Абиссинская', 'Рыжая', 4)
cat2 = Cat('Британская', 'Серая', 2)
```

Отлично, теперь, когда у нас есть основа, приступим к изучению принципов ООП.

Инкапсуляция

Доступ к данным объекта должен контролироваться, чтобы пользователь не мог изменить их в произвольном порядке и что-то поломать. Поэтому для работы с данными программисты пишут методы, которые можно будет использовать вне класса и которые ничего не сломают внутри.

Вернёмся к нашим кошечкам. Мы можем разрешить изменять атрибут «возраст», но только в большую сторону, а атрибуты «порода» и «цвет» лучше открыть только для чтения — ведь порода кошки не меняется, а цвет если и меняется, то не по её инициативе.

В нашем классе «Кошка» мы сделали все атрибуты открытыми, поэтому давайте это исправим:

```
class Cat():
  def __init__(self, breed, color, age):
     self._breed = breed
     self._color = color
     self._age = age

  @property
  def breed(self):
    return self._breed

  @property
  def color(self):
    return self._color

  @property
  def age(self):
    return self._age

  @age.setter
  def age(self, new_age):
    if new_age > self._age:
      self._age = new_age
    return self._age
```

Код стал выглядеть немного сложнее, но мы сейчас всё объясним. Сначала мы сделали все атрибуты закрытыми с помощью символа `_`. Он говорит интерпретатору, что эта переменная будет доступна только внутри методов класса.

Нам всё ещё нужно получать доступ к атрибутам, поэтому мы предоставляем его через `@property` и объявляем для каждого атрибута свой метод — `breed`, `color`, `age`. В каждом из этих методов мы возвращаем значение нашего закрытого атрибута. Это делает его доступным только для чтения.

И последнее — мы должны позволить пользователям увеличивать возраст кота. Для этого воспользуемся `@age.setter` и ещё раз объявим метод `age`, а внутри него проверим простое условие и вернём значение атрибута.

Теперь создадим экземпляр класса:

```
cat = Cat('Абиссинская', 'Рыжая', 4)
```

Выведем значения атрибутов:

```
print(cat.breed) # Абиссинская
print(cat.color) # Рыжая
print(cat.age) # 4
```

И попробуем изменить атрибут `age`:

```
cat.age = 5
print(cat.age) # 5
```

Всё успешно. А теперь сделаем это с другим атрибутом:

```
cat.breed = 'Сиамская'
print(cat.breed) # AttributeError: can't set attribute on line 34 in main.p
```

Мы получили ошибку, потому что запретили изменять этот атрибут.

Читайте также:

Инкапсуляция, модификаторы доступа: 3 часть гайда по ООП

Наследование

Классы могут передавать свои атрибуты и методы классам-потомкам. Например, мы хотим создать новый класс «Домашняя кошка». Он практически идентичен классу «Кошка», но у него появляются новые атрибуты «хозяин» и «кличка», а также метод «клянчить вкусняшку».

Достаточно объявить «Домашнюю кошку» наследником «Кошки» и прописать новые атрибуты и методы — вся остальная функциональность перейдёт от родителя к потомку.

Давайте объявим новый класс:

```
class HomeCat(Cat):
  def __init__(self, breed, color, age, owner, name):
    super().__init__(breed, color, age)
    self._owner = owner
    self._name = name

  @property
  def owner(self):
    return self._owner

  @property
  def name(self):
    return self._name

  def getTreat(self):
    print('Мяу-мяу')
```

В первой строке мы как раз наследуем все методы и атрибуты класса `Cat`. А чтобы всё создалось корректно, мы должны вызвать метод `super()` в методе `__init__()` и через него заполнить атрибуты класса-родителя. Поэтому мы и передаём в этот метод «породу», «окрас» и «возраст».

Кроме атрибутов для класса-родителя у класса-потомка есть и собственные атрибуты: «хозяин» — `owner` и «кличка» — `name`. Их мы будем использовать только в этом классе, поэтому они будут недоступны для класса-родителя.

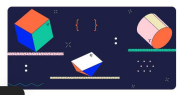
Мы сразу сделали атрибуты класса-потомка закрытыми и объявили для них собственные методы. А также добавили метод `getTreat()`, которого нет в классе-родителе.

Давайте создадим объект класса:

```
my_cat = HomeCat('Сиамская', 'Белая', 3, 'Иван', 'Роза')

print(my_cat.owner)
print(my_cat.breed)
my_cat.getTreat() # Мяу-мяу
my_cat.purr() # Мрррр
```

Как видим, у нас работают и новые методы, и старые. Наследование прошло успешно.



Читайте также:

Наследование и ещё немного полиморфизма: 5 часть гайда по ООП

Полиморфизм

Этот принцип позволяет применять одни и те же команды к объектам разных классов, даже если они выполняются по-разному. Например, помимо класса «Кошка», у нас есть никак не связанный с ним класс «Попугай» — и у обоих есть метод «спать». Несмотря на то что кошки и попугаи спят по-разному (кошка сворачивается клубком, а попугай сидит на жёрдочке), для этих действий можно использовать одну команду.

Допустим у нас есть два класса — «Кошка» и «Попугай»:

```
class Cat:
  def sleep(self):
    print('Свернулся в клубок и сладко спит.')

class Parrot:
  def sleep(self):
    print('Сел на жёрдочку и уснул.')
```

А теперь пусть у нас есть метод, который ожидает, что ему на вход придёт объект, у которого будет метод `sleep`:

```
def homeSleep(animal):
  animal.sleep()
```

Посмотрим, как это будет работать:

```
cat = Cat()
parrot = Parrot()
homeSleep(cat) # Свернулся в клубок и сладко спит.
homeSleep(parrot) # Сел на жёрдочку и уснул.
```

Хотя классы разные, их одноимённые методы работают похожим образом. Это и есть полиморфизм.

Читайте также:

Полиморфизм в ООП, перегрузка методов и операторов

Абстракция

При создании класса мы упрощаем его до тех атрибутов и методов, которые нужны именно в этом коде, не пытаясь описать его целиком и отбрасывая всё второстепенное. Например, у всех хищников есть метод «охотиться», поэтому все животные, которые являются хищниками, автоматически будут уметь охотиться.

Рассмотрим класс Predator:

```
class Predator:
  def hunt(self):
    print('Охотится...')
```

Этот класс будет общим для всех животных, которые являются хищниками, — например, кошек:

```
class Cat(Predator):
  def __init__(self, name, color):
    super().__init__()
    self._name = name
    self._color = color

  @property
  def name(self):
    return self._name

  @property
  def color(self):
    return self._color
```

У кошки есть свои атрибуты: «имя» — `name` и «окрас» — `color`. Но при этом она потомок хищников, а значит, умеет охотиться:

```
cat = Cat('Даниэла', 'Чёрный')
cat.hunt() # Охотится…
```

Читайте также:

Абстрактные классы и интерфейсы: 6 часть гайда по ООП

Примеры реализации ООП на Python

Давайте ещё пофантазируем и посоздаём классы.

Представьте ситуацию: вашего друга пригласили на пафосную вечеринку в закрытый клуб. Там довольно странный этикет: в разное время все должны пить строго определённые напитки. Причём любой из них, в зависимости от ситуации

строго определённые напитки. Причём любой из них, в зависимости от ситуации, все пьют определённым способом: или обычными глотками по 20 мл, или маленькими по 10, или залпом всё, что осталось. Более того: размер глотка для одного и того же напитка может внезапно поменяться по ходу вечеринки.

Вы выучиваете все эти дурацкие правила и вызываетесь помочь другу, но общаться с ним можете только через микронаушник. Таким образом, друг становится интерфейсом вашего взаимодействия с напитками.

Для начала создадим класс `Drink`:

```
class Drink:
    pass # Пока ставим заглушку вместо будущего кода.
```

У любого напитка есть атрибуты: название, стоимость в рублях и объём в миллилитрах. Предположим для простоты, что на нашей вечеринке принято всегда пить из посуды одинакового объёма (200 мл), а остальные атрибуты могут меняться от напитка к напитку.

Соответственно, объём — это статический атрибут, неизменный во всех объектах класса. Название и стоимость, напротив, — динамические: они принадлежат не всему классу в целом, а конкретному объекту, и их значение определяется уже после его создания.

```
class Drink:
    # Присваиваем значение статическому атрибуту.
    volume = 200

    # Создаём метод для инициализации объекта.
    def __init__ (self, name, price):
        # Присваиваем значения динамическим атрибутам.
        self.name = name
        self.price = price
```

Создадим объект `coffee` — экземпляр класса `Drink`. В нашем примере создание нового объекта обозначает заказ нового напитка:

```
coffee = Drink ('Кофе', 300)
```

Теперь у нас есть объект `coffee`, который содержит статический атрибут `volume`, полученный от класса `Drink`, и динамические атрибуты `name` и `price`, которые мы указали при создании объекта. Давайте попробуем к ним обратиться:

```
print (coffee.name)
>>> Кофе

print (coffee.price)
>>> 300

print (coffee.volume)
>>> 200
```

Так как статические атрибуты определяются на уровне класса, то и обращаться к ним можно не только через объект, но и через сам класс:

```
Drink.volume
>>> 200
```

К динамическим атрибутам мы так обратиться не сможем.

Итак, напиток заказан, и с ним нужно что-то делать. Так как вы общаетесь через микронаушник, то не видите, в каком состоянии напиток друга. Что ж, попросим друга сообщить вам об этом. Для этого добавим ещё один метод внутри класса `Drink`:

```
class Drink:

    volume = 200

    def __init__ (self, name, price):
        self.name = name
        self.price = price

    # Метод, чтобы запросить у друга информацию о напитке.
    def drink_info (self):
        print (f'Название: {self.name}. Стоимость: {self.price}. Объём: {se



# Создаём объект coffee.
coffee = Drink ('Кофе', 300)
# Обращаемся к методу drink_info через объект coffee.
coffee.drink_info()

>>> Название: Кофе. Стоимость: 300. Объём: 200
```

Тусовка делает первый глоток. Скомандуем другу, чтобы он присоединился. Для этого нужен ещё один динамический атрибут `remains`, информирующий нас, сколько миллилитров напитка осталось. Изначально остаток будет равен объёму . После этого прописываем метод, указывающий товарищу, сколько тно глотать в соответствии с этикетом:

```
class Drink:

    volume = 200

    def __init__ (self, name, price):
        self.name = name
        self.price = price
        # Устанавливаем начальное значение атрибута remains.
        self.remains = self.volume

    def drink_info (self):
        # Добавляем информацию об остатке напитка в метод drink_info.
        print (f'Название: {self.name}. Стоимость: {self.price}. Начальный

    # Метод, чтобы сказать другу сделать глоток.
    def sip (self):
        # Проверяем, достаточно ли напитка осталось.
        if self.remains >= 20:
            self.remains -= 20
            print ('Друг сделал глоток')
        # Если напитка не хватает, сообщаем об этом.
        else:
            print ('Не хватает напитка для полноценного глотка')

coffee = Drink ('Кофе', 300)
coffee.sip() # Говорим другу сделать глоток.
coffee.drink_info() # Узнаём информацию о напитке.


>>> Друг сделал глоток
>>> Название: Кофе. Стоимость: 300. Начальный объём: 200. Осталось: 180
```

Уровни доступа в Python

Чтобы нам не приходилось каждый раз проверять, хватает ли напитка для нужного глотка, напишем служебный метод `_is_enough`. Затем перепишем метод `sip` и добавим методы `small_sip` и `drink_all`:

```
# Продолжаем дописывать методы класса Drink.

    # Служебный метод, чтобы узнать, достаточно ли напитка.
    def _is_enough (self, need):
        if self.remains >= need and self.remains > 0:
            return True
        print ('Осталось недостаточно напитка')
        return False

    # Говорим другу сделать глоток.
    def sip (self):
        if self._is_enough(20) == True:
            self.remains -= 20
            print ('Друг сделал глоток')

    # Говорим другу сделать маленький глоток.
    def small_sip (self):
        if self._is_enough(10) == True:
            self.remains -= 10
            print ('Друг сделал маленький глоток')

    # Говорим другу выпить напиток залпом.
    def drink_all (self):
        if self._is_enough(0) == True:
            self.remains = 0
            print ('Друг выпил напиток залпом')

coffee = Drink ('Кофе', 300) # Заказываем кофе.
coffee.remains = 10 #Приравниваем остаток кофе к 10 мл.
coffee.sip() # Пытаемся сделать обычный глоток.
coffee.learn_info() # Узнаём информацию о напитке.

>>> Осталось недостаточно напитка
>>> Название: Кофе. Стоимость: 300. Начальный объём: 200. Осталось: 10
```

Обратите внимание ещё на такой нюанс: в строке `coffee.remains = 10` мы извне вмешались в объект и приравняли его атрибут `remains` к 10. Это удалось потому,

что все атрибуты и методы в Python по умолчанию являются публичными, то есть доступными извне.

Чтобы регулировать вмешательство во внутреннюю работу объекта, в ООП есть несколько уровней доступа: **публичный** (public), **защищённый** (protected) и **приватный** (private). Защищённые атрибуты и методы можно вызывать только внутри класса и его классов-наследников. Приватные — только внутри класса: даже наследники не имеют доступа к ним.

n это реализовано следующим образом: перед защищёнными атрибутами дами пишут одинарное нижнее подчёркивание (`_example`), перед приватными — двойное (`__example`). Именно это мы сделали в методе `_is_enough`. Одинарным нижним подчёркиванием мы объявили его защищённым.

Одинарным нижним подчёркиванием мы объявили его защищённым.

При этом в Python само по себе объявление атрибутов и методов защищёнными и приватными не ограничивает доступ к ним извне. Мы всё ещё можем вызвать метод `_is_enough` из любого места программы:

```
# Вызываем метод _is_enough и спрашиваем его, осталось ли хотя бы 10 мл нап
coffee._is_enough(10)
>>> True
```

Атрибуты и методы, объявленные приватными, вызвать напрямую уже нельзя, но есть обходной путь:

```
# Создаём класс Drink с приватным атрибутом __volume.
class Drink:
    __volume = 200
# Создаём экземпляр класса Drink.
coffee = Drink()
# Используем обходной путь, чтобы обратиться к приватному атрибуту.
coffee._Drink__volume
>>> 200
```

> **Примечание.** Возможность игнорировать уровни доступа — нарушение важного для ООП принципа инкапсуляции. Поэтому, несмотря на наличие технической возможности, программисты, пишущие на Python, договорились не обращаться к защищённым и приватным методам откуда-то извне.

Так что и мы объявим защищёнными атрибуты `volume` и `remains`, чтобы помнить: ими стоит пользоваться только внутри класса `Drink` и его наследников. Теперь всё выглядит так:

```
class Drink:

    # Определяем статический атрибут.
    _volume = 200

    # Вызываем инициализатор класса и определяем динамические атрибуты.
    def __init__ (self, name, price):
        self.name = name
        self.price = price
        self._remains = self._volume

    # Просим друга сообщить информацию о напитке.
    def drink_info (self):
        print (f'Название: {self.name}. Стоимость: {self.price}. Начальный

    # Служебный метод, позволяющий узнать, достаточно ли напитка.
    def _is_enough (self, need):
        if self._remains >= need and self._remains > 0:
            return True
        print ('Осталось недостаточно напитка')
        return False

    # Говорим другу сделать глоток.
    def sip (self):
        if self._is_enough(20) == True:
            self._remains -= 20
            print ('Друг сделал глоток')

    # Говорим другу сделать маленький глоток.
    def small_sip (self):
        if self._is_enough(10) == True:
            self._remains -= 10
            print ('Друг сделал маленький глоток')

    # Говорим другу выпить напиток залпом.
    def drink_all (self):
        if self._is_enough(0) == True:
            self._remains = 0
            print ('Друг выпил напиток залпом')
```

Наследование в Python

Вечеринка идёт полным ходом. Но тут случается непредвиденное: ваш друг, который уже слегка приспособился к местным нравам и даже начал получать удовольствие, внезапно кричит вам в наушник, что вечер вновь перестаёт быть томным. «Они объявили время соков! — паникует он. — А у каждого сока свой вкус, тут чёрт ногу сломит!»

Действительно. Хьюстон, у нас проблемы. Сок, на первый взгляд, — напиток как напиток: его тоже можно пить глотками и залпом, у него есть цена и объём. Но, как пел гражданин Шнуров, есть один момент: в отличие от любого напитка, у сока появляется новый, специфический атрибут, который не поддерживается классом `Drink`, — вкус фрукта или ягоды, из которых он выжат.

Мы [illegible]икуем: даже из самой сложной ситуации всегда есть как минимум два [illegible]. Можно, конечно, полностью скопировать класс `Drink` и изменить в этой [illegible]всё, что нам нужно. Но мы поступим изящнее — создадим класс `Juice` и сделаем его наследником класса `Drink`:

```
# Создаём класс-потомок и указываем в скобках родительский класс, от которо
class Juice (Drink):

    # Вызываем инициализатор класса и указываем в нём новый динамический ат
    def __init__ (self, name, price, taste):
        # Вызываем конструктор класса-родителя и просим его определить знач
        super().__init__ (name, price)
        # Определяем значение нового динамического атрибута taste.
        self.taste = taste
```

> **Примечание.** Обратите внимание, что из класса-потомка мы не можем напрямую обратиться к приватным атрибутам и методам класса-родителя.

Создаём объект класса `Juice` и вызываем в нём методы, унаследованные от родительского класса `Drink`:

```
# Создаём экземпляр класса Juice.
apple_juice = Juice ('сок', 250, 'яблочный')

# Пробуем вызвать методы, прописанные в родительском классе Drink.
apple_juice.small_sip() # Говорим другу сделать маленький глоток.
apple_juice.sip() # Говорим другу сделать обычный глоток.
apple_juice.drink_info() # Просим друга сообщить информацию о напитке.

>>> Друг сделал маленький глоток

>>> Друг сделал глоток
>>> Название: сок. Стоимость: 250. Начальный объём: 200. Осталось: 170
```

Родительский класс `Drink` поделился с потомком своими атрибутами и методами, так что нам не пришлось писать их заново.

Теперь посмотрим на атрибут `name`. В классе `Drink`, когда мы могли заказать что угодно, от кофе и чая до кваса и коктейля, имело смысл каждый раз указывать название напитка. Но в классе `Juice` название всегда будет одинаковым: «сок». Тогда зачем всё время при заказе сока спрашивать атрибут `name`?

Переопределим в классе `Juice` метод `__init__`: пусть значением атрибута `name` всегда будет «сок». И затем снова закажем яблочный сок:

```
class Juice (Drink):

    # Создаём статический атрибут, который будет содержать название нашего
    _juice_name = 'сок'

    # Вызываем инициализатор класса и указываем в нём только те аргументы,
    def __init__ (self, price, taste):
        # Передаём конструктору родительского класса значение атрибута __ju
        super().__init__ (self._juice_name, price)
        self.taste = taste

apple_juice = Juice (250, 'яблочный') # Создаём объект класса Juice.
```

Что же именно происходит при создании объекта `apple_juice`?

1. Мы вызываем инициализатор класса `Juice` и в скобках передаём ему аргументы `price` и `taste`.

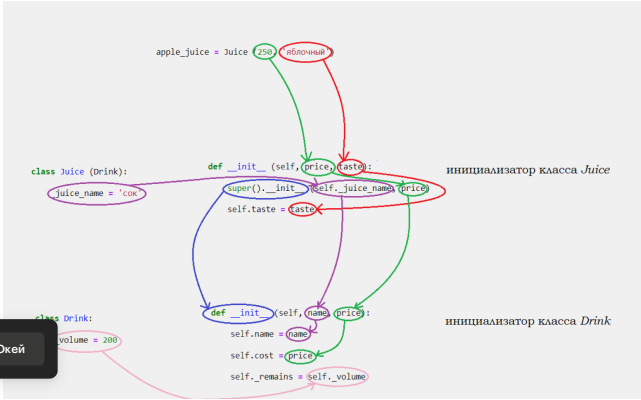
2. Инициализатор класса `Juice` с помощью функции `super()` вызывает другой инициализатор — родительского класса `Drink`.

3. Инициализатор класса `Drink` просит передать ему аргументы `name` и `price`. В качестве аргумента `name` он получает статический атрибут `_juice_name`, который мы прописали в классе `Juice`. А аргумент `price` подтягивается из инициализатора класса `Juice`.

4. В инициализаторе класса `Drink` присваиваются значения атрибутам `name`, `price` и `_remains`.

5. В инициализаторе класса `Juice` присваивается значение атрибуту `taste`.

Если вам всё ещё сложно сориентироваться, что откуда берётся и куда передаётся, посмотрите на эту схему. Разными цветами здесь обозначены пути, по которым атрибутам присваиваются их значения:



Изображение: Skillbox Media

Итак, вы объяснили другу, что больше не надо каждый раз объявлять, что он хочет именно сок, — а то все подумают, что он деревенщина. Всем и так понятно, что он не компот заказывает. Но посмотрите, что происходит, когда мы просим его сообщить информацию об экземпляре класса `Juice`:

```
apple_juice = Juice (250, 'яблочный')
apple_juice.drink_info()
>>> Название: сок. Стоимость: 250. Начальный объём: 200. Осталось: 200
```

Он сообщает нам, что пьёт сок, но не говорит, какой именно. Чтобы получить от друга дополнительную информацию, переопределим метод `drink_info` родительского класса:

```
class Juice (Drink):

    # Определяем статический атрибут.
    _juice_name = 'сок'

    # Вызываем инициализатор класса и определяем динамические атрибуты.
    def __init__ (self, price, taste):
        super().__init__ (self._juice_name, price)
        self.taste = taste

    # Переопределяем родительский метод drink_info, чтобы он сообщал нам ин
    def drink_info (self):
        print (f'Вкус сока: {self.taste}. Стоимость: {self.price}. Начальны



# Создаём экземпляр класса Juice и вызываем метод learn_info.
apple_juice = Juice (250, 'яблочный')
apple_juice.drink_info()

>>>  Вкус сока: яблочный. Стоимость: 250. Начальный объём: 200. Осталось: 2
```

Так мы реализовали принцип полиморфизма. Неважно, что пьёт наш друг — кофе или сок, мы можем запросить у него информацию о напитке одной и той же командой `drink_info`. И приятель уже сам будет ориентироваться по ситуации: если он пьёт сок, то сообщит нам его вкус, а если любой другой напиток — его название.

> **Примечание.** Все классы в Python по умолчанию являются наследниками суперкласса `object` и наследуют его атрибуты и методы. Такими унаследованными методами, например, являются встроенные `__new__`, `__init__`, `__del__` и многие другие.

Вечеринка потихоньку подходит к концу, и вашего товарища пока не спалили. Время соков прошло, и каждый теперь волен пить то, что пожелает. Вроде бы можно расслабиться. Но, как вы знаете, у нас и тамада хороший, и конкурсы интересные: посетителей внезапно огорошивают новой затеей. Рассаживайтесь, говорят, за столики в соответствии со стоимостью только что заказанного напитка. Все начинают выкрикивать, почём бокалы в их руках, а официанты отводят их на новые места. Друг снова в ступоре, но мы его спасём.

Так как объявить стоимость можно для любого напитка, пропишем метод `tell_price` в классе `Drink` — и дочерний класс `Juice` автоматически унаследует его:

```
    def tell_price (self):
        print (f'Друг объявляет стоимость своего напитка')
        return self.price
```

Теперь проверим, действительно ли он работает с объектами как класса `Drink`, так и класса `Juice`:

```
tea = Drink ('чай', 500)
print (tea.tell_price()) # Сначала друг объявит стоимость чая.
beetlejuice = Juice (1988, 'жучиный')
print (beetlejuice.tell_price()) # Потом друг объявит стоимость жучиного со

>>> Друг объявляет стоимость своего напитка
>>> 500
>>> Друг объявляет стоимость своего напитка
>>> 1988
```

Профит, коллеги: ваш друг уходит с вечеринки с новой подружкой и приглашением на следующее мероприятие. А всё благодаря вам и объектно-ориентированному программированию.

Что запомнить

Подведём краткие итоги:

- Объектно-ориентированное программирование — распространённая и эффективная парадигма, которая подходит для выполнения многих задач. Здесь основной строительной единицей программы является е функция, а объект, представляющий собой экземпляр некоторого асса.
- ООП строится вокруг четырёх основных принципов: абстракция,

инкапсуляция, наследование и полиморфизм.

- Язык Python отлично поддерживает ООП. В нём всё является объектом, даже числа и сами классы. Тем не менее в Python есть баг с уровнями доступа, нарушающий принцип инкапсуляции. Но при ответственном подходе к работе с кодом это не должно стать проблемой.

Больше интересного про код в нашем телеграм-канале. Подписывайтесь!

ЧИТАЙТЕ ТАКЖЕ:

- Как начать программировать на Python: руководство для новичков
- Тест: Какой язык создадите вы — Java или Python?
- Учимся верстать: что такое CSS

Поделиться

Новости

Инженеры Google из Японии представили клавиатуру-кепку Gboard Caps

06 окт 2023

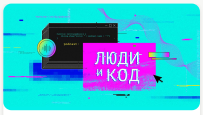
Итоги презентации Made by Google 2023: анонс Pixel 8, Pixel Watch 2 и релиз Android 14

06 окт 2023

Apple выпустила обновление iOS 17.0.3 с патчем от перегрева

05 окт 2023

Это интересно



Stable Diffusion, Midjourney и все-все-все: что под капотом у графических нейросетей

Тест: насколько хорошо вы разбираетесь в КПК

Что такое Singleton и как его использовать в разработке приложений

Аккредитация IT-компании: что нужно знать и как получить

Стремительный дракон: проекты и перспективы генеративного

КОД #ПОДКАСТ 6 окт 2023

Stable Diffusion, Midjourney и все-все-все: что под капотом у графических нейросетей

Подкаст «Люди и код», выпуск № 89: Дмитрий Савостьянов.

Поделиться

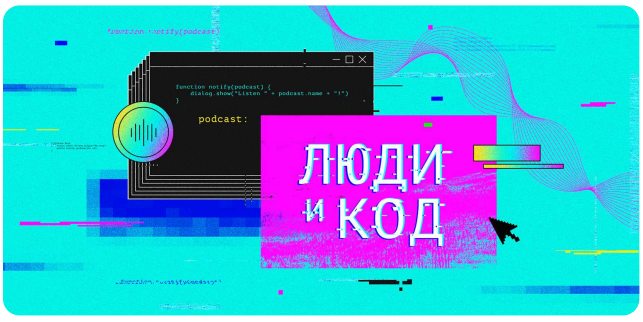


Иллюстрация: Polina Vari / Skillbox Media

Антон Сёмин

Пишет об истории IT, разработке и советской кибернетике. Знает Python, JavaScript и немного C++, но предпочитает писать на русском.

У нашего подкаста появился отдельный телеграм-канал. Подписывайтесь, чтобы не пропустить новые выпуски, голосовать за новые темы, предлагать гостей и присылать свои вопросы для следующих выпусков.

Дмитрий Савостьянов