**Модуль 4. Объектно-ориентированный дизайн**

[Шаблоны проектирования (паттерны банды четырех) 1](#_Toc146844796)

[1. Паттерн Строитель 3](#_Toc146844797)

[2. Фабричный метод (Factory Method) 7](#_Toc146844798)

[Магический метод \_\_new\_\_. Пример паттерна Singleton 10](#_Toc146844799)

[Пример паттерна Singleton (учебный) 13](#_Toc146844800)

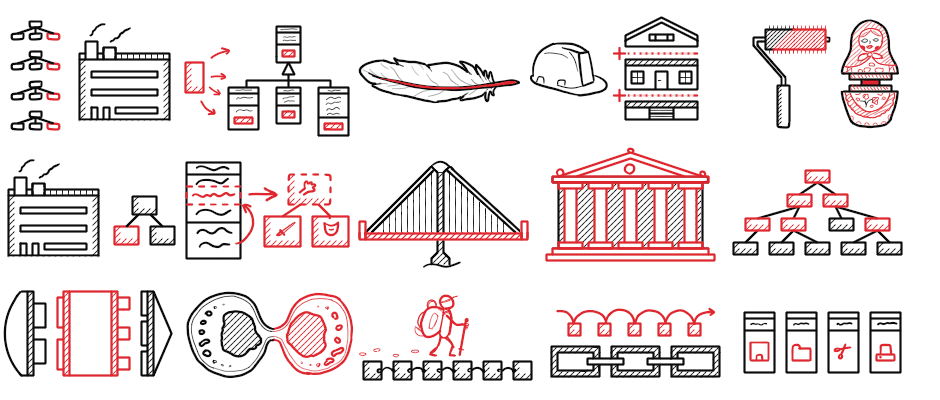
[Практическая работа Применение паттерна Фабричный метод 15](#_Toc146844801)

[Техническое задание. Игра с геометрическими фигурами. 15](#_Toc146844802)

[Разбор практической работы 17](#_Toc146844803)

# Шаблоны проектирования (паттерны банды четырех)

Философия Python базируется на хорошо продуманных лучших практиках программирования. Многие шаблоны проектирования уже встроены в язык. Разработчики используют их, даже не задумываясь. Ряд популярных паттернов очень легко реализовать благодаря динамической природе языка. А некоторые не используются в Python, так как в них нет необходимости.



Python является объектно-ориентированным языком, однако прекрасно поддерживает функциональный стиль программирования. Разработчик вовсе не обязан создавать классы и их экземпляры. Если проекту не нужны сложные структуры, нет необходимости их строить. Можно просто писать функции или даже совсем не структурированный код, чтобы быстро выполнять несложные задачи. В то же время, все элементы языка – это объекты. Даже функции, которые являются "объектами первого класса".

Что такое шаблоны проектирования?

Все началось с Банды четырех. Именно они сформулировали и подробно описали ряд способов решения распространенных проблем программирования.

Концепцию паттернов впервые описал Кристофер Александер в книге «Язык шаблонов. Города. Здания. Строительство».



Идея показалась привлекательной авторам Эриху Гамму, Ричарду Хелму, Ральфу Джонсону и Джону Влиссидесу, их принято называть «бандой четырёх» (Gang of Four).

В 1995 году они написали книгу «Design Patterns: Elements of Reusable Object-Oriented Software», в которой применили концепцию типовых паттернов в программировании. В книгу вошли 23 паттерна, решающие различные проблемы объектно-ориентированного дизайна.

В их основу были положены два принципа:

Программировать нужно для интерфейсов, а не для конкретных реализаций.

Композицию следует предпочитать наследованию.

Паттерны проектирования можно разделить по трем типам:

Паттерны порождающего типа (Creational patterns): отвечают за удобное и безопасное создание новых объектов или инстанцирование классов используя наследование или композицию.

Паттерны структурного типа (Structural patterns): позволяют определять отношения между различными элементами системы для создания более сложных структур.

Паттерны поведенческого типа (Behavioral patterns): определяют взаимодействие между объектами и классами системы и делятся на уровни объектов, классов и взаимодействующих объектов и классов.

Вообще правила, созданы для того, чтобы их улучшать и/или нарушать

Шаблон перехода дороги – остановись посмотри налево, потом направо, убедись, что безопасно, потом иди. А если дорога 4 полоска, а если дорога автомагистраль.

Согласно стандартного шаблона вы должны уже её перейти, а вы зависли в потоке машин.

Любое шаблон – это руководство к действию. А шаблонов много, банда 4-х придумала их целых 23.

Например, смысл шаблона Фабрика – скрывать логику создания новых объектов. Но в Python это не нужно, так как этот процесс динамичен по своей сути. Конечно, Фабрику можно реализовать, если есть желание. Иногда это действительно полезно, но такие случаи – больше исключение, нежели правило.

### Паттерн Строитель

Рассмотрим пример. Класс пицца. Он принимает заказ, размер, сыр, пеперони (салями) и бекон делать-не делать.

class Pizza:  
 def \_\_init\_\_(self, size, cheese, pepperoni, bacon):   
 self.\_\_size = size  
 self.\_\_cheese = cheese  
 self.\_\_pepperoni = pepperoni  
 self.\_\_bacon = bacon  
  
 def show(self):  
 recipe = 'Пицца ' + str(self.\_\_size) + " с "  
 recipe += "сыром, " if self.\_\_cheese else ""  
 recipe += "пепперони, " if self.\_\_pepperoni else ""  
 recipe += "беконом" if self.\_\_bacon else ""  
 return recipe  
pizza = Pizza(12, True, False, True)

По сути – здесь реализован антишаблон. Как не надо делать. Этот антишаблон называется телескопический конструктор. Подзорная труба. Которую использовать не нужно.

Его приходится использовать вот таким образом:

pizza = Pizza(12, True, False, True)

Сложно понять, что где-что. Не очень удобно.

Было бы удобнее и правильнее сделать так

pizza = PizzaMaker(12)\  
 .cheese()\  
 .pepperoni()\  
 .bacon()\  
 .make()

Обязательный параметр размер. Всё остальное функционально – через точку. Т.е. может быть или не быть. Причем это можно делать в любом порядке.

Как это сделать. Это придумано в шаблоне под названием builder

def \_\_init\_\_(self, size):  
 self.\_size = size  
 self.\_cheese = False  
 self.\_pepperoni = False  
 self.\_bacon = False  
 def cheese(self):  
 self.\_cheese = True  
 return self  
 def pepperoni(self):  
 self.\_pepperoni = True  
 return self  
 def bacon(self):  
 self.\_bacon = True  
 return self  
 def make(self):  
 return self

def \_\_repr\_\_(self):  
 return f'Пицца {self.\_size},\  
 Бекон:{"да" if self.\_bacon == 1 else "нет"},\  
 Сыр:{"да" if self.\_cheese == 1 else "нет"},\  
 Пеперони :{"да" if self.\_pepperoni == 1 else "нет"}'

pizza = PizzaMaker(12)\  
 .cheese()\  
 .pepperoni()\  
 .bacon()\  
 .make()

print(pizza)

Взяли рецепт. Отдает в производство и в изготовления. Описываем пицу. Выбираем, что у нас есть.

class Pizza:  
 def \_\_init\_\_(self, maker: PizzaMaker):  
 self.\_\_size = maker.\_size  
 self.\_\_cheese = maker.\_cheese  
 self.\_\_pepperoni = maker.\_pepperoni  
 self.\_\_bacon = maker.\_bacon  
 def \_\_str\_\_(self):  
 recipe = 'Пицца:' + str(self.\_\_size) + ", с "  
 recipe += "сыром, " if self.\_\_cheese else ""  
 recipe += "пепперони, " if self.\_\_pepperoni else ""  
 recipe += "беконом" if self.\_\_bacon else ""  
 return recipe  
  
  
  
order = Pizza(pizza)  
print(order)

Шаблон на то и шаблон, чтобы действовать примерно.

Еще один пример.

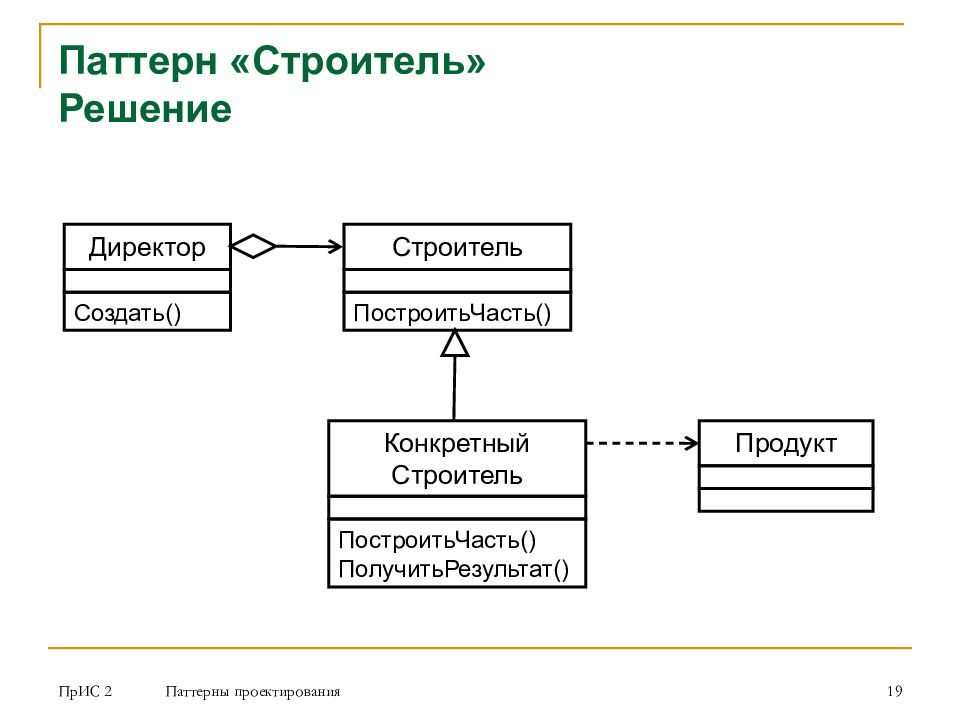
class Phone:  
 def \_\_init\_\_(self):  
 self.os = None  
 self.camera = None  
 self.battery = None  
 self.screen = None

Снова подзорная труба. Класс Phone имеет четыре свойства: os, camera, battery, и screen. Для каждого из этих свойств может быть выбрано одно из нескольких возможных значений. Например, для свойства os может быть выбрано одно из трех значений: «iOS», «Android», или «Windows». Для того, чтобы создать объект класса Phone, необходимо задать значения для каждого из свойств.

Для более гибкого создания объектов класса Phone в соответствии с требуемыми параметрами, мы можем использовать паттерн Строитель, который позволит нам разделить процесс создания объекта на несколько шагов. Для этого мы можем определить класс PhoneBuilder, который будет предоставлять удобный интерфейс для задания значений каждого свойства объекта Phone, позволяя создавать объекты с теми параметрами, которые нам нужны.

class PhoneBuilder:  
 def \_\_init\_\_(self):  
 self.phone = Phone()  
  
 def set\_os(self, os):  
 self.phone.os = os  
 return self  
  
 def set\_camera(self, camera):  
 self.phone.camera = camera  
 return self  
  
 def set\_battery(self, battery):  
 self.phone.battery = battery  
 return self  
  
 def set\_screen(self, screen):  
 self.phone.screen = screen  
 return self  
  
 def get\_phone(self):  
 return self.phone

В классе PhoneBuilder мы определяем методы для пошагового задания значений каждого свойства объекта класса Phone. Каждый метод возвращает ссылку на объект self, что позволяет использовать цепочку вызовов методов. Когда все значения свойств заданы, мы вызываем метод get\_phone, который возвращает объект класса Phone со всеми заданными значениями.



Таким образом, класс PhoneBuilder предоставляет удобный и гибкий способ создания объектов класса Phone с нужными параметрами.

Для создания объекта класса Phone с помощью класса PhoneBuilder необходимо выполнить следующий код:

builder = PhoneBuilder()  
phone = builder.set\_os("Android").set\_camera("12 MP").set\_battery("3500 mAh").set\_screen("6.2 inches").get\_phone()

В данном примере мы создали объект builder класса PhoneBuilder, который позволяет нам пошагово задать значения для каждого из свойств объекта класса Phone. Затем мы вызвали методы set\_os, set\_camera, set\_battery, и set\_screen, чтобы задать значения для каждого из свойств. В конце мы вызвали метод get\_phone, который вернул нам готовый объект

Использование паттерна Строитель — это один из способов повысить качество кода и улучшить его структуру. Он может быть использован вместе с другими паттернами проектирования и является необходимым инструментом в арсенале каждого разработчика Python.

### Фабричный метод (Factory Method)

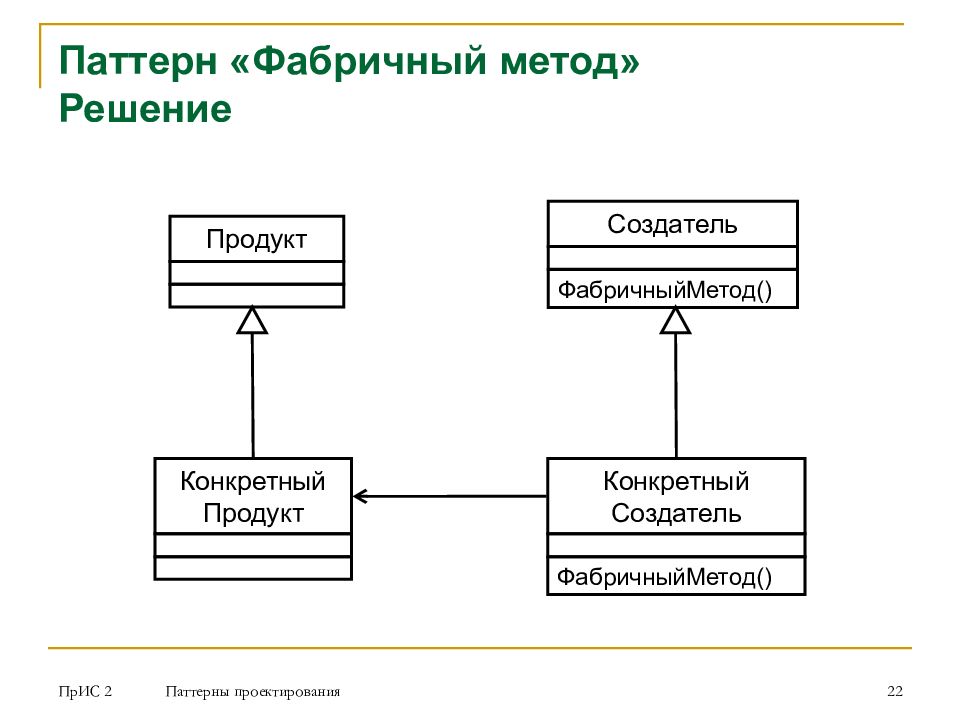
Фабричный метод (Factory Method) - это порождающий паттерн проектирования, который позволяет создавать объекты без указания их конкретных классов. Вместо этого он предоставляет интерфейс для создания объектов в суперклассе, который может быть переопределен в подклассах для изменения типа создаваемого объекта.

from abc import ABC, abstractmethod  
  
class Animal(ABC):  
 @abstractmethod  
 def speak(self):  
 pass  
  
class Dog(Animal):  
 def speak(self):  
 return "Woof!"  
  
class Cat(Animal):  
 def speak(self):  
 return "Meow!"  
  
class AnimalFactory:  
 def create\_animal(self, animal\_type):  
 if animal\_type == "Dog":  
 return Dog()  
 elif animal\_type == "Cat":  
 return Cat()  
 else:  
 return None  
  
factory = AnimalFactory()  
  
animal = factory.create\_animal("Dog")  
print(animal.speak()) *# Output: Woof!*animal = factory.create\_animal("Cat")  
print(animal.speak()) *# Output: Meow!*

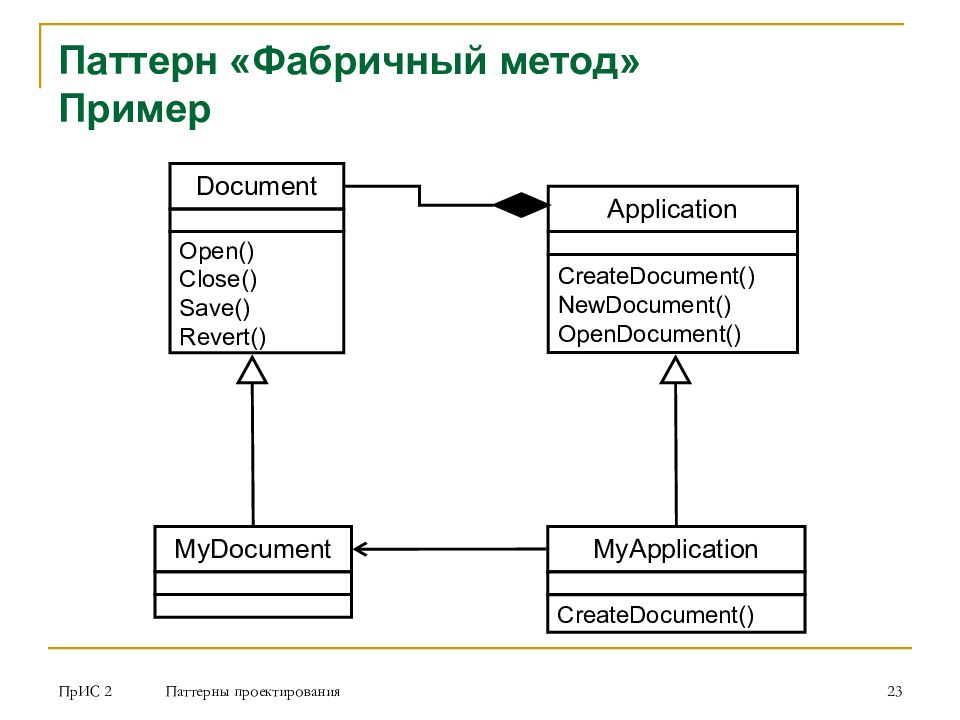
В этом примере Animal является суперклассом, который определяет абстрактный метод speak(). Dog и Cat - это подклассы, которые реализуют этот метод и возвращают соответствующий звук животного.

AnimalFactory - это фабричный метод, который создает объекты Dog или Cat в зависимости от переданного типа.

Когда factory.create\_animal("Dog") вызывается, он создает объект типа Dog и возвращает его. Аналогично, при вызове factory.create\_animal("Cat"), он создает объект типа Cat и возвращает его.



Таким образом, фабричный метод позволяет создавать объекты, не указывая их конкретных классов, что делает код гораздо более гибким и расширяемым.



К примеру шаблон синглетон, одиночка.

### Паттерн Singleton. Одиночка

Шаблон Singleton (Одиночка) один из самых часто используемых шаблонов. Его можно встретить во множестве проектов и он относительно прост для обучения. Его обязательно нужно знать и уметь его использовать.

Шаблон Singleton предоставляет механизм создания одного и только один экземпляра объекта, и предоставление к нему глобальную точку доступа.

Поэтому, Singletons обычно используются в таких случаях, как ведение журнала или операции с базой данных, диспетчера очереди печати и многих других, где существует необходимость иметь только один экземпляр, который доступен во всем приложении, чтобы избежать конфликтующих запросов на один и тот же ресурс.

Например, мы можем захотеть использовать один объект базы данных для выполнения операций с БД для обеспечения согласованности данных или один объект класса ведения журнала для нескольких служб, чтобы последовательно выгружать сообщения журнала в определенный файл журнала.

Самый простой синглтон

class Singleton:  
 \_\_instance = None  
  
  
 def \_\_new\_\_(cls):  
 if cls.\_\_instance is None:  
 cls.\_\_instance=super().\_\_new\_\_(cls)  
 return cls.\_\_instance

Класс Singleton определяется с помощью частной переменной класса под названием \_\_instance. В этой переменной будет храниться единственный экземпляр класса.

Метод \_\_new\_\_ переопределяется, чтобы проверить, имеет ли переменная класса \_\_instance значение. Если \_\_instance равно None, то класс Singleton создается впервые, поэтому новый экземпляр класса создается с помощью функции super().\_\_new\_\_(cls). Функция super() возвращает временный объект суперкласса, который затем используется для вызова его методов.

Если \_\_instance не None, это означает, что экземпляр класса Singleton уже был создан, поэтому вместо создания нового экземпляра возвращается существующий.

Преимущество использования этого паттерна в том, что он гарантирует, что в любой момент времени существует только один экземпляр класса. Это может быть полезно, когда вы хотите контролировать доступ к ресурсам или отслеживать глобальное состояние приложения.

Вкратце, цель шаблона Singleton заключаются в следующем:

• Обеспечение создания одного и только одного объекта класса

• Предоставление точки доступа для объекта, который является глобальным для программы

• Контроль одновременного доступа к ресурсам, которые являются общими

Простой способ реализации Singleton – сделать закрытым метод конструктора и создать статический метод, который выполняет инициализацию объекта. Таким образом, один объект создается при первом вызове, а класс будет всегда возвращать тот же объект при попытки новой инициализации объекта.

В Python мы реализуем эту идею по-другому, поскольку в нем нет возможности создавать приватные конструкторы.

Вот пример кода шаблона Singleton. В этом примере мы делаем две основные вещи:

Мы сделаем возможным создание только одного экземпляра класса Singleton.

Если экземпляр существует, мы всегда будем использовать уже существующий объект.

### Магический метод \_\_new\_\_. Пример паттерна Singleton

Познакомимся с еще одним магическим методом \_\_new\_\_, который вызывается непосредственно перед созданием объекта класса. Я напомню, что другой магический метод \_\_init\_\_ вызывается после создания объекта (о нем мы говорили перед этим).

Здесь у вас может сразу возникнуть вопрос, зачем нужно было определять два разных метода, которые последовательно вызываются при создании экземпляров классов? Разве не достаточно одного \_\_init\_\_, чтобы выполнять начальную инициализацию объекта? Конечно, нет.

В практике программирования встречаются самые разнообразные задачи и иногда нужно что-то делать и до создания объектов. Например, реализация известного паттерна Singleton в Python, как раз делается через метод \_\_new\_\_ и мы с ним позже познакомимся.

А для начала нам нужно познакомиться с работой самого метода \_\_new\_\_. Давайте добавим его в наш класс Point. Я его перепишу в сокращенной форме:

**class** Point:

**def** \_\_new\_\_(cls, \*args, \*\*kwargs):

**print**("вызов \_\_new\_\_ для " + str(cls))

**def** \_\_init\_\_(self, x=0, y=0):

**print**("вызов \_\_init\_\_ для " + str(self))

        self.x = x

        self.y = y

Смотрите, здесь записан метод \_\_new\_\_, у которого первым идет обязательный параметр cls – это ссылка на текущий класс Point, а затем, указываются коллекции из фактических и формальных параметров, которые может принимать данная функция. Это стандартное определение метода \_\_new\_\_ в классах. В теле функции я просто сделал вывод сообщения и переменной cls.

Если теперь попробовать создать экземпляр класса:

pt = Point(1, 2)

то мы в консоли увидим только одно сообщение от метода \_\_new\_\_. То есть, второй метод \_\_init\_\_ не был вызван и, кроме того, если мы распечатаем переменную pt:

**print**(pt)

то увидим значение None, то есть, объект не был создан. Почему так произошло? В Python магический метод \_\_new\_\_ должен возвращать адрес нового созданного объекта. А в нашей программе он ничего не возвращает, то есть, значение None, что эквивалентно отказу в создании нового объекта. Именно поэтому переменная pt принимает значение None.

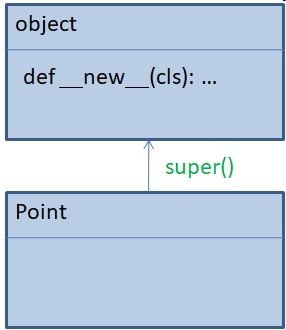
Хорошо, давайте адрес нового объекта. Но откуда мы его возьмем? Для этого можно вызвать аналогичный метод базового класса и делается это, следующим образом:

**def** \_\_new\_\_(cls, \*args, \*\*kwargs):

**print**("вызов \_\_new\_\_ для " + str(cls))

**return** super().\_\_new\_\_(cls)

Здесь функция super() возвращает ссылку на базовый класс и через нее мы вызываем метод \_\_new\_\_ с одним первым аргументом. Но, подождите! Что это за базовый класс? Мы наш класс Point ни от какого класса не наследовали? Да и вообще еще не изучали тему наследования! Да, поэтому, забегая вперед, скажу, что, начиная с версии Python 3, все классы автоматически и неявно наследуются от базового класса object:



И уже из этого базового класса мы вызываем метод \_\_new\_\_. Кстати, если метод \_\_new\_\_ не прописывать в классе Point, то будет автоматически запускаться версия базового класса.

То есть, этот метод всегда вызывается при создании нового объекта. При необходимости, мы можем его переопределять, добавляя новую логику его работы.

И то же самое относится ко всем магическим методам. Они всегда существуют у всех классов. Но переопределяем мы лишь те, что необходимо, а остальные работают по умолчанию. В этом сила базового класса object. В нем уже существует программный код, общий для всех классов языка Python. Иначе, нам пришлось бы его каждый раз прописывать заново.

Итак, теперь мы знаем откуда берется и вызывается магический метод \_\_new\_\_. Запустим программу и видим в консоли, что были вызваны оба метода \_\_new\_\_ и \_\_init\_\_ нашего класса Point, а также был успешно сформирован новый объект.

Возможно, здесь у вас остался один вопрос: а зачем нужны списки параметров \*args, \*\*kwargs в методе \_\_new\_\_? Мы, вроде, их нигде не используем? В действительности, здесь хранятся дополнительные параметры, которые мы можем указывать при создании объекта. Например, строчка:

pt = Point(1, 2)

создает объект с двумя числовыми значениями, то есть, \*args будет содержать эти два числа. По идее, мы можем реализовать в методе \_\_new\_\_ какую-либо логику с учетом значений этих аргументов.

Но, в данном случае, просто игнорируем. Используем их дальше в методе \_\_init\_\_ при инициализации объекта. То есть, аргументы 1 и 2 передаются и в метод \_\_new\_\_ и в метод \_\_init\_\_.

### Пример паттерна Singleton (учебный)

Думаю, вы в целом теперь представляете себе работу магического метода \_\_new\_\_, но остается вопрос: зачем все же он нужен? В качестве ответа я приведу пример очень известного паттерна проектирования под названием Singleton. Этот паттерн будет представлен в учебном варианте, то есть, мы его реализуем не полностью, т.к. пока отсутствуют достаточные знания.

Итак, давайте предположим, что мы разрабатываем класс для работы с БД. В частности, через него можно будет подключаться к СУБД, читать и записывать информацию, закрывать соединение:

**class** DataBase:

**def** \_\_init\_\_(self, user, psw, port):

        self.user = user

        self.psw = psw

        self.port = port

***def*** *connect(self):*

***print****(f"соединение с БД: {self.user}, {self.psw}, {self.port}")*

***def*** *close(self):*

***print****("закрытие соединения с БД")*

***def*** *read(self):*

***return*** *"данные из БД"*

***def*** *write(self, data):*

***print****(f"запись в БД {data}")*

И далее полагаем, что в программе должен существовать только один экземпляр этого класса в каждый момент ее работы. То есть, одновременно два объекта класса DataBase быть не должно. Чтобы это обеспечить и гарантировать, как раз и используется паттерн Singleton. Реализуем его для класса DataBase.

Я пропишу в нем специальный атрибут (на уровне класса):

\_\_instance = None

который будет хранить ссылку на экземпляр этого класса. Если экземпляра нет, то атрибут будет принимать значение None. А, затем, чтобы гарантировать создание строго одного экземпляра, добавим в класс магический метод \_\_new\_\_:

**def** \_\_new\_\_(cls, \*args, \*\*kwargs):

**if** cls.\_\_instance **is** None:

            cls.\_\_instance = super().\_\_new\_\_(cls)

**return** cls.\_\_instance

Работает этот метод очевидным образом. Мы проверяем атрибут класса \_\_instance. Причем, для обращения к нему используем параметр cls – ссылку на текущий класс. Подробнее я еще освещу этот момент. Далее, проверяем, если значение равно None, то вызываем метод \_\_new\_\_ базового класса и тем самым разрешаем создание объекта. Иначе, просто возвращаем ссылку на ранее созданный экземпляр. Как видите, все достаточно просто.

И пропишем еще один магический метод – финализатор \_\_del\_\_, который будет обнулять атрибут \_\_instance перед уничтожением объекта, чтобы мы могли, при необходимости, создать новый.

Все, простейший вариант паттерна Singleton готов. Правда он имеет один изъян. Смотрите, если попробовать создать два экземпляра:

db = DataBase('root', '1234', 80)

db2 = DataBase('root2', '5678', 40)

**print**(id(db), id(db2))

то их id ожидаемо будут равны. То есть, ссылки db и db2 действительно ведут на один объект. Но, если выполнить метод:

db.connect()

db2.connect()

то увидим значения: 'root2', '5678', 40 – аргументы при повторном создании класса. По идее, если объект не создается, то и локальные свойства его также не должны меняться. Почему так произошло? Все просто. Мы здесь действительно видим первый объект. Но при повторном вызове DataBase() также был вызван магический метод \_\_init\_\_ с новым набором аргументов и локальные свойства изменили свое значение. Конечно, мы можем здесь поставить «костыль» (как говорят в программисты) и дополнительно в классе прописать флаговый атрибут, например:

\_\_is\_exist = False

специально для метода \_\_init\_\_, чтобы не выполнять его если объект уже создан. Но я даже не буду дописывать такую программу. Слишком уж костыльно получается. Правильнее было бы здесь переопределить еще один магический метод \_\_call\_\_, о котором мы еще будем говорить. А пока оставим нашу реализацию паттерна Singleton в таком виде.

Я, надеюсь, что из этого занятия вы поняли, как работает магический метод \_\_new\_\_ и зачем он нужен.

class Singleton(object):

def \_\_new\_\_(cls):

if not hasattr(cls, 'instance'):

cls.instance = super(Singleton, cls).\_\_new\_\_(cls)

return cls.instance

s = Singleton()

print("Object created", s)

s1 = Singleton()

print("Object created", s1)

Вывод выполнения кода должен быть примерно таким:

('Object created', <\_\_main\_\_.Singleton object at 0x10ba9db90>)

('Object created', <\_\_main\_\_.Singleton object at 0x10ba9db90>)

# Практическая работа Применение паттерна Фабричный метод

### Техническое задание. Игра с геометрическими фигурами.

Текстовая игра для школьников с геометрическими фигурами.

Дан интерфейс вот такого плана. Основная работа сделана.

Фигуры геометрические, двумерные.

**import** **abc**

**class** **IShape**(abc.ABC):

*'''Интерфейс для реализации геометрических фигур'''*

@abc.abstractmethod

**def** get\_perimeter(self):

*'''Возвращает периметр фигуры'''*

**pass**

@abc.abstractmethod

**def** get\_area(self):

*'''Возвращает площадь фигуры'''*

**pass**

@abc.abstractmethod

**def** get\_description(self):

*'''Возвращает произвольное описание фигуры'''*

**pass**

Любая фигура имеет метод get\_perimeter(self), который в*озвращает периметр фигуры,* get\_area и get\_description*.*

Нужно описать:

* Создайте классы Circle, Rectangle и Square, которые реализуют интерфейс IShape
* Создайте класс Game, который описывает логику игру

В классах Circle, Rectangle и Square необходимо описать:

* конструктор, который принимает в качестве параметров необходимые значения. Для Круга таковым является его радиус, для Прямоугольника - размеры двух его сторон, для Квадрата - размер сторон.
* методы, которые определены в интерфейсе IShape
* метод get\_description должен возвращать произвольную строку вида Я - имя\_фигуры параметры

В классе Game необходимо описать логику игру:

* Пользователю называется количество геометрических фигур участвующих в игре
* Пользователю предлагается начать игру. В случае отказа, печатается строка Спасибо за участие!
* Случайным образом создаётся объект - экземпляр одного из классов
* Созданному объекту в конструктор передаются необходимые параметры со случайными значениями
* Пользователю по очереди задаются вопросы касающиеся площади и периметра фигуры, на которые он должен ответить
* В зависимости от правильности ответа пользователя печатается строка Правильно! или Ошибка! с правильным ответом
* После каждой пары вопросов пользователю предлагается продолжить игру

Как это должно работать

>>> Привет! Мы геометрические фигуры и у нас есть 2 вопроса.

>>> Играем? Y/N: y

>>> Я - окружность с радиусом 7

>>> Укажите мою площадь: 153.86

>>> Правильно!

>>> Укажите мой периметр: 100

>>> Ошибка! Правильный ответ: 43.96

>>> Играем? Y/N: y

>>> Я - прямоугольник со сторонами 7 и 1

>>> Укажите мою площадь: 7

>>> Правильно!

>>> Укажите мой периметр: 16

>>> Правильно!

>>> Играем? Y/N: n

>>> Спасибо за участие!

Советы по описанию класса Game:

* Создайте статический метод get\_shape, который создаёт и возвращает экземпляр нужного класса
* Создайте статический метод calculate, который задаёт вопросы и принимает и проверяет ответы
* Создайте метод класса run, который вызывает методы get\_shape и calculate
* Создайте метод класса play, который запускает основной цикл и вызывает метод run
* Для начала игры вызовите метод play

### Разбор практической работы

1 этап

import abc  
  
  
class IShape(abc.ABC):  
 *'''Интерфейс для реализации геометрических фигур'''* @abc.abstractmethod  
 def get\_perimeter(self):  
 *'''Возвращает периметр фигуры'''* pass  
  
 @abc.abstractmethod  
 def get\_area(self):  
 *'''Возвращает площадь фигуры'''* pass  
  
 @abc.abstractmethod  
 def get\_description(self):  
 *'''Возвращает произвольное описание фигуры'''* pass  
  
  
**class Circle(IShape):  
 PI = 3.14  
  
 def \_\_init\_\_(self, radius):  
 self.\_\_radius = radius  
  
 def get\_perimeter(self):  
 return 2 \* self.\_\_class\_\_.PI \* self.\_\_radius  
  
 def get\_area(self):  
 return self.\_\_class\_\_.PI \* self.\_\_radius \*\* 2  
  
 def get\_description(self):  
 return f'Я окружность с радиусом {self.\_\_radius}'  
  
  
class Rectangle(IShape):  
 def \_\_init\_\_(self, width, height):  
 self.\_\_width = width  
 self.\_\_heigth = height  
  
 def get\_width(self):  
 return self.\_\_width  
  
 width = property(get\_width) *# для чего ?* def get\_height(self):  
 return self.\_\_heigth  
  
 heigth = property(get\_height) *# для чего ?* def get\_perimeter(self):  
 return 2 \* (self.width + self.heigth)  
  
 def get\_area(self):  
 return self.width \* self.heigth  
  
 def get\_description(self):  
 return f'Я прямоугольник {self.\_\_width}х {self.\_\_heigth}'  
  
  
class Square(Rectangle):  
 def \_\_init\_\_(self, side):  
 super().\_\_init\_\_(side, side)  
  
 def get\_description(self):  
 return f'Я квадрат размером {self.width}'  
  
  
  
r1 = Rectangle(10, 20)  
s1 = Square(10)  
print(s1.get\_perimeter())  
print(s1.get\_description())**

Всё работает!? Всё понятно?

Далее пошла бизнес-логика.

В классе.

2 этап

class Game:  
 QUESTION\_COUNT = 2 #кол-во вопросоа  
  
 def \_\_init\_\_(self): *# чтобы не создавать экземпляры данного класса* raise Exception('Нельзя создавать экземпляр данного класс') *# поскольку это алгоритм, это БЛ-игры* @staticmethod  
 def \_\_get\_shape():  
 pass  
  
 @staticmethod  
 def \_\_calculate(string, answer): *# площадь и заранее вычисленный ответ* pass  
  
 @classmethod  
 def \_\_run(cls): *# единственная разница класс передается или нет* pass  
 @classmethod  
 def play(cls): *# единственный открытый метод, который будет вызываться* pass  
  
  
Game.play() *# единственная строка, запускающая игры (БЛ)*

далее описываем согласно ТЗ

@classmethod  
def play(cls): *# единственный открытый метод, который будет вызываться* print(f'Привет! Мы фигуры и у нас есть {cls.QUESTION\_COUNT} вопроса')  
 while True:  
 is\_game\_over = input('Играем? Y/N:').strip()  
 if is\_game\_over.upper()=="N":  
 break  
 cls.\_\_run()  
 print('Спасибо за участие!')

Далее

@classmethod  
def \_\_run(cls): *# единственная разница класс передается или нет* shape = cls.\_\_get\_shape()

и фабричный метод

@staticmethod  
def \_\_get\_shape(): *# метод фабрика* type = randrange(0,2)  
 if type == 0:  
 return Circle(randrange(1,10))  
 elif type == 1:  
 return Rectangle(randrange(1, 10),randrange(1, 10) )  
 else:  
 return Square(randrange(1, 10))

Причем генерить рандом мы можем и в методе run и передавать его в \_\_get\_shape.

Shape – реализует определенный интерфейс. Посмотрим его описание

def \_\_run(cls): *# единственная разница класс передается или нет* shape = cls.\_\_get\_shape()  
 print (shape.get\_description())

Проверим:

Привет! Мы фигуры и у нас есть 2 вопроса

Играем? Y/N:Y

Я прямоугольник 9х 6

Играем? Y/N:Y

Я окружность с радиусом 4

Играем? Y/N:

Причесываем метод:

def \_\_run(cls): *# единственная разница класс передается или нет* shape = cls.\_\_get\_shape()  
 if isinstance(shape, IShape):  
 print (shape.get\_description())  
 cls.\_\_calculate('площадь', shape.get\_area())  
 cls.\_\_calculate('периметр', shape.get\_perimeter())  
 else:  
 raise TypeError('Не та фигура')

далее calculate

def \_\_calculate(string, answer): *# площадь и заранее вычисленный ответ* guess = input(f'Укажите {string} этой фигуры').strip()  
 if not guess.replace('.','',1).isdigit():  
 print('Введите число')

оптимизируем

def \_\_calculate(string, answer): *# площадь и заранее вычисленный ответ* while True:  
 guess = input(f'Укажите {string} этой фигуры').strip()  
 if not guess.replace('.','',1).isdigit():  
 print('Введите число')  
 continue  
 break

далее

@staticmethod  
def \_\_calculate(string, answer): *# площадь и заранее вычисленный ответ* while True:  
 guess = input(f'Укажите {string} этой фигуры').strip()  
 if not guess.replace('.', '', 1).isdigit():  
 print('Введите число')  
 continue  
 break  
 if float(guess) == answer:  
 print('Правильно')  
 else:  
 print(f'Ошибка. Правильный ответ - {answer}')

Весь код:

import abc  
from random import randrange  
  
  
class IShape(abc.ABC):  
 *'''Интерфейс для реализации геометрических фигур'''* @abc.abstractmethod  
 def get\_perimeter(self):  
 *'''Возвращает периметр фигуры'''* pass  
  
 @abc.abstractmethod  
 def get\_area(self):  
 *'''Возвращает площадь фигуры'''* pass  
  
 @abc.abstractmethod  
 def get\_description(self):  
 *'''Возвращает произвольное описание фигуры'''* pass  
  
  
class Circle(IShape):  
 PI = 3.14  
  
 def \_\_init\_\_(self, radius):  
 self.\_\_radius = radius  
  
 def get\_perimeter(self):  
 return 2 \* self.\_\_class\_\_.PI \* self.\_\_radius  
  
 def get\_area(self):  
 return self.\_\_class\_\_.PI \* self.\_\_radius \*\* 2  
  
 def get\_description(self):  
 return f'Я окружность с радиусом {self.\_\_radius}'  
  
  
class Rectangle(IShape):  
 def \_\_init\_\_(self, width, height):  
 self.\_\_width = width  
 self.\_\_heigth = height  
  
 def get\_width(self):  
 return self.\_\_width  
  
 width = property(get\_width) *# для чего ?* def get\_height(self):  
 return self.\_\_heigth  
  
 heigth = property(get\_height) *# для чего ?* def get\_perimeter(self):  
 return 2 \* (self.width + self.heigth)  
  
 def get\_area(self):  
 return self.width \* self.heigth  
  
 def get\_description(self):  
 return f'Я прямоугольник {self.\_\_width} х {self.\_\_heigth}'  
  
  
class Square(Rectangle):  
 def \_\_init\_\_(self, side):  
 super().\_\_init\_\_(side, side)  
  
 def get\_description(self):  
 return f'Я квадрат размером {self.width}'  
  
  
*# бизнес-логика*class Game:  
 QUESTION\_COUNT = 2  
  
 def \_\_init\_\_(self): *# чтобы не создавать экземпляры данного класса* raise Exception('Нельзя создавать экземпляр данного класс') *# поскольку это алгоритм, это БЛ-игры* @staticmethod  
 def \_\_get\_shape(): *# метод фабрика* type = randrange(0, 2)  
 if type == 0:  
 return Circle(randrange(1, 10))  
 elif type == 1:  
 return Rectangle(randrange(1, 10), randrange(1, 10))  
 else:  
 return Square(randrange(1, 10))  
  
 @staticmethod  
 def \_\_calculate(string, answer): *# площадь и заранее вычисленный ответ* while True:  
 guess = input(f'Укажите {string} этой фигуры').strip()  
 if not guess.replace('.', '', 1).isdigit():  
 print('Введите число')  
 continue  
 break  
 if float(guess) == answer:  
 print('Правильно')  
 else:  
 print(f'Ошибка. Правильный ответ - {answer}')  
  
 @classmethod  
 def \_\_run(cls): *# единственная разница класс передается или нет* shape = cls.\_\_get\_shape()  
 if isinstance(shape, IShape):  
 print(shape.get\_description())  
 cls.\_\_calculate('площадь', shape.get\_area())  
 cls.\_\_calculate('периметр', shape.get\_perimeter())  
 else:  
 raise TypeError('Не та фигура')  
  
 @classmethod  
 def play(cls): *# единственный открытый метод, который будет вызываться* print(f'Привет! Мы фигуры и у нас есть {cls.QUESTION\_COUNT} вопроса')  
 while True:  
 is\_game\_over = input('Играем? Y/N:').strip()  
 if is\_game\_over.upper() == "N":  
 break  
 cls.\_\_run()  
 print('Спасибо за участие!')  
  
  
Game.play() *# единственная строка, запускающая игры (БЛ)*r1 = Rectangle(10, 20)  
s1 = Square(10)  
print(s1.get\_perimeter())  
print(s1.get\_description())

Остались вопросы по деталям, по дизайну.

Где генерить рандом

А также можно вынести метод get\_shape в отдельный класс FabricShape.

Фабрика фигур.

Есть о чем подумать. С нормальным обоснованием.

А так всё ОК. Есть интерфейс, которые реализуется соответствующими классами.

Остались риторические вопросы.

Можно ли отнаследовать квадрат от прямоугольника?

Или правильно ли его сделать отдельным классом.