Порождающие шаблоны в Python

23.12.2019 Editorial Team 0 Комментариев patterns 38 просмотров

Spread the love

Оригинальная статья: Darinka Zobenica – Creational Design Patterns in Python

Первая статья в небольшой серии статей, посвященных шаблонам проектирования в Python.

Порождающие шаблоны

Порождающие шаблоны, как следует из названия, имеют дело с созданием классов или объектов.

Они служат для абстрагирования от специфики классов, чтобы мы были менее зависимы от их точной реализации, или чтобы нам не приходилось иметь дело со сложной конструкцией всякий раз, когда они нам нужны, или чтобы мы обеспечивали некоторые специальные свойства при создание экземпляров.

Они очень полезны для понижения уровня зависимости между классами и управления взаимодействием пользователя с ними.

Шаблоны, описанные в этой статье:

- Factory
- Abstract Factory
- Builder
- Prototype
- Singleton
- Object Pool

Factory

Проблема

Допустим, вы создаете программное обеспечение для страховой компании, которая предлагает страхование людям, которые работают полный рабочий день. Вы сделали

приложение, используя класс под названием **Worker** .

Тем не менее, клиент решает расширить свой бизнес и теперь будет предоставлять свои услуги и безработным, хотя и с другими процедурами и условиями.

Теперь вам нужно создать совершенно новый класс для безработных, который возьмет совершенно другого конструктора! Но теперь вы не знаете, какой конструктор вызывать в общем случае, тем более, какие аргументы ему нужно передать.

Вы можете использовать некоторые уродливые условия во всем коде, где каждый вызов конструктора будет окружен операторами **if** , и вы вынуждены будете используете какую-то дорогостоящую операцию для проверки типа самого объекта.

Если во время инициализации будут возникать ошибки, вы будете тратить время на их поиск, и редактирования кода, в каждом из ста мест, в которых используются конструкторы.

Понятно что такой подход менее чем желателен, он не масштабируется и вообще неустойчив.

В качестве альтернативы, вы можете рассмотреть **фабричный шаблон** .

Решение

Фабрики используются для инкапсуляции информации о классах, и при этом для создания их экземпляров на основе определенных параметров, которые им предоставляются.

Используя фабрику, можно поменять реализацию одну на другую, просто изменив параметр, который использовался для первоначального определения исходной реализации.

Это отделяет реализацию от использования таким образом, что мы можем легко масштабировать приложение, добавляя новые реализации и просто создавая их экземпляры через фабрику – с точно такой же кодовой базой.

Если мы просто получим другую фабрику в качестве параметра, нам даже не нужно будет знать, какой класс она производит. Нам просто нужен метод единой фабрики, который возвращает класс с гарантированным набором поведений. Давайте создадим фабрику.

Для начала не забудьте включить абстрактные методы:

```
from abc import ABC, abstractmethod
```

Нам нужны созданные ранее классы для реализации некоторого набора методов, которые позволяют работать с ними единообразно. Для этого мы реализуем следующий интерфейс:

```
class Product(ABC):
    @abstractmethod
    def calculate_risk(self):
        pass
```

И теперь мы наследуем от него **Worker** и **Unemployed** :

```
class Worker(Product):
    def __init__(self, name, age, hours):
        self.name = name
        self.age = age
        self.hours = hours
    def calculate_risk(self):
        # Пожалуйста, представьте более правдоподобную реализацию
        return self.age + 100/self.hours
    def __str__(self):
        return self.name+" ["+str(self.age)+"] - "+str(self.hours)+"
class Unemployed(Product):
    def __init__(self, name, age, able):
        self.name = name
        self.age = age
        self.able = able
    def calculate_risk(self):
        if self.able:
            return self.age+10
        else:
            return self.age+30
    def __str__(self):
        if self.able:
            return self.name+" ["+str(self.age)+"] - able to work"
        else:
            return self.name+" ["+str(self.age)+"] - unable to work"
```

Теперь, когда у нас есть наши работники, давайте сделаем для них фабрику:

```
class PersonFactory:
    def get_person(self, type_of_person):
```

```
        if type_of_person == "worker":
            return Worker("Oliver", 22, 30)
        if type_of_person == "unemployed":
            return Unemployed("Sophie", 33, False)
```

Здесь для ясности мы жестко закодировали параметры, хотя обычно вы просто создаете экземпляр класса и заставляете его делать свое дело.

Чтобы проверить, как все это работает, давайте создадим экземпляр нашей фабрики и позволим ей произвести пару работников:

```
factory = PersonFactory()
product = factory.get_person("worker")
print(product)
product2 = factory.get_person("unemployed")
print(product2)
```

```
Oliver [22] - 30h/week
Sophie [33] - unable to work
```

Abstract Factory

Проблема

Вам нужно создать семейство разных предметов. Хотя они разные, они так или иначе сгруппированы по определенной характеристике.

Например, вам может понадобиться создать основное блюдо и десерт в итальянском и французском ресторане, но вы не хотите смешивать одну кухню с другой.

Решение

Идея очень похожа на обычный шаблон фабрики, единственное отличие состоит в том, что все фабрики имеют несколько отдельных методов для создания объектов, и тип фабрики определяет семейство объектов.

Абстрактная фабрика отвечает за создание целых групп объектов, наряду с их соответствующими фабриками, – но она не касается конкретных реализаций этих объектов. Эта часть оставлена для их соответствующих фабрик:

```
from abc import ABC, abstractmethod
```

```
class Product(ABC):
    @abstractmethod
    def cook(self):
        pass
class FettuccineAlfredo(Product):
    name = "Fettuccine Alfredo"
    def cook(self):
        print("Italian main course prepared: "+self.name)
class Tiramisu(Product):
    name = "Tiramisu"
    def cook(self):
        print("Italian dessert prepared: "+self.name)
class DuckALOrange(Product):
    name = "Duck À L'Orange"
    def cook(self):
        print("French main course prepared: "+self.name)
class CremeBrulee(Product):
    name = "Crème brûlée"
    def cook(self):
        print("French dessert prepared: "+self.name)
class Factory(ABC):
    @abstractmethod
    def get_dish(type_of_meal):
        pass
class ItalianDishesFactory(Factory):
    def get_dish(type_of_meal):
        if type_of_meal == "main":
            return FettuccineAlfredo()
        if type_of_meal == "dessert":
            return Tiramisu()
    def create_dessert(self):
        return Tiramisu()
class FrenchDishesFactory(Factory):
    def get_dish(type_of_meal):
        if type_of_meal == "main":
            return DuckALOrange()
        if type_of_meal == "dessert":
            return CremeBrulee()
class FactoryProducer:
    def get_factory(self, type_of_factory):
        if type_of_factory == "italian":
            return ItalianDishesFactory
        if type_of_factory == "french":
            return FrenchDishesFactory
```

Мы можем проверить результаты, создав фабрики и вызвав соответствующие методы **cook()** для всех объектов:

```
fp = FactoryProducer()
fac = fp.get_factory("italian")
main = fac.get_dish("main")
main.cook()
dessert = fac.get_dish("dessert")
dessert.cook()
fac1 = fp.get_factory("french")
main = fac1.get_dish("main")
main.cook()
dessert = fac1.get_dish("dessert")
dessert.cook()
```

```
Italian main course prepared: Fettuccine Alfredo
Italian dessert prepared: Tiramisu
French main course prepared: Duck À L'Orange
French dessert prepared: Crème brûlée
```

Builder

Проблема

Представьте что вам нужно создать робота. Робот может быть гуманоидным с четырьмя конечностями и стоящим вертикально, или он может быть похож на животное с хвостом, крыльями и т. д.

Он может использовать колеса, чтобы двигаться, или он может использовать лопасти вертолета. Он может использовать камеры, инфракрасный модуль обнаружения т.д..

А теперь представьте себе какой может быть конструктор для такого объекта:

```
def __init__(self, left_leg, right_leg, left_arm, right_arm,
             left_wing, right_wing, tail, blades, cameras,
             infrared_module, #...
             ):
    self.left_leg = left_leg
    if left_leg == None:
        bipedal = False
    self.right_leg = right_leg
    self.left_arm = left_arm
    self.right_arm = right_arm
    # ...
```

Создание этого класса было бы крайне нечитаемым, было бы очень легко ошибиться в некоторых типах аргументов, так как мы работаем в Python, и можем накапливать бесчисленное количество аргументов в конструкторе.

Кроме того, что если мы не хотим, чтобы робот реализовал все поля в классе? Что если мы хотим, чтобы у него были только ноги, а не ноги и колеса?

Python не поддерживает перегруженные конструкторы, которые помогли бы нам определить такие случаи (и даже если бы мы могли, это привело бы только к еще более грязным конструкторам).

Решение

Мы можем создать класс **Builder** , который создает наш объект и добавляет соответствующие модули нашему роботу. Вместо замысловатого конструктора мы можем создать экземпляр объекта и добавить необходимые компоненты с помощью функций.

Мы вызываем конструкцию каждого модуля отдельно, после создания объекта. Давайте продолжим и определим класс **Robot** с некоторыми значениями по умолчанию:

```
class Robot:
    def __init__(self):
        self.bipedal = False
        self.quadripedal = False
        self.wheeled = False
        self.flying = False
        self.traversal = []
        self.detection_systems = []
    def __str__(self):
        string = ""
        if self.bipedal:
            string += "BIPEDAL "
        if self.quadripedal:
            string += "QUADRIPEDAL "
        if self.flying:
            string += "FLYING ROBOT "
        if self.wheeled:
            string += "ROBOT ON WHEELS\n"
        else:
            string += "ROBOT\n"
        if self.traversal:
            string += "Traversal modules installed:\n"
        for module in self.traversal:
```

```
            string += "- " + str(module) + "\n"
        if self.detection_systems:
            string += "Detection systems installed:\n"
        for system in self.detection_systems:
            string += "- " + str(system) + "\n"
        return string
class BipedalLegs:
    def __str__(self):
        return "two legs"
class QuadripedalLegs:
    def __str__(self):
        return "four legs"
class Arms:
    def __str__(self):
        return "four legs"
class Wings:
    def __str__(self):
        return "wings"
class Blades:
    def __str__(self):
        return "blades"
class FourWheels:
    def __str__(self):
        return "four wheels"
class TwoWheels:
    def __str__(self):
        return "two wheels"
class CameraDetectionSystem:
    def __str__(self):
        return "cameras"
class InfraredDetectionSystem:
    def __str__(self):
        return "infrared"
```

Обратите внимание, что мы пропустили определенные инициализации в конструкторе и использовали вместо них значения по умолчанию. Это потому, что мы будем использовать классы **Builder** для инициализации этих значений.

Сначала мы реализуем абстрактный **Builder** , который определяет наш интерфейс для сборки:

```
from abc import ABC, abstractmethod
class RobotBuilder(ABC):
    @abstractmethod
    def reset(self):
        pass
    @abstractmethod
```

```
    def build_traversal(self):
        pass
    @abstractmethod
    def build_detection_system(self):
        pass
```

Теперь мы можем реализовать несколько видов **Builders** , которые подчиняются этому интерфейсу, например, для Android и для автономного автомобиля:

```
class AndroidBuilder(RobotBuilder):
    def __init__(self):
        self.product = Robot()
    def reset(self):
        self.product = Robot()
    def get_product(self):
        return self.product
    def build_traversal(self):
        self.product.bipedal = True
        self.product.traversal.append(BipedalLegs())
        self.product.traversal.append(Arms())
    def build_detection_system(self):
        self.product.detection_systems.append(CameraDetectionSystem(
class AutonomousCarBuilder(RobotBuilder):
    def __init__(self):
        self.product = Robot()
    def reset(self):
        self.product = Robot()
    def get_product(self):
        return self.product
    def build_traversal(self):
        self.product.wheeled = True
        self.product.traversal.append(FourWheels())
    def build_detection_system(self):
        self.product.detection_systems.append(InfraredDetectionSyste
```

Заметьте, как они реализуют одни и те же методы, но под ними по-разному строится структура объектов, и конечному пользователю не нужно разбираться в деталях этой структуры?

Конечно, мы могли бы создать робота, который может иметь как ножки, так и колеса, и пользователь должен был бы добавить все это по отдельности, но мы также можем создать очень специфические конструкторы, которые добавляют только один подходящий модуль для каждой «детали».

Давайте попробуем использовать **AndroidBuilder** для создания андроидного робота:

```
builder = AndroidBuilder()
builder.build_traversal()
builder.build_detection_system()
print(builder.get_product())
```

Запуск этого кода даст:

```
BIPEDAL ROBOT
Traversal modules installed:
- two legs
- four legs
Detection systems installed:
- cameras
```

А теперь давайте воспользуемся **AutonomousCarBuilder** для сборки автомобиля:

```
builder = AutonomousCarBuilder()
builder.build_traversal()
builder.build_detection_system()
print(builder.get_product())
```

Запуск этого кода даст:

```
ROBOT ON WHEELS
Traversal modules installed:
- four wheels
Detection systems installed:
- infrared
```

Подобная инициализация намного более понятна и читаема по сравнению с грязным конструктором, и у нас есть гибкость в добавлении модулей, которые мы хотим.

Если в полях нашего продукта используются относительно стандартные конструкторы, мы можем даже создать так называемого директора для управления конкретными сборщиками:

```
class Director:
    def make_android(self, builder):
        builder.build_traversal()
        builder.build_detection_system()
        return builder.get_product()
    def make_autonomous_car(self, builder):
        builder.build_traversal()
```

```
        builder.build_detection_system()
        return builder.get_product()
director = Director()
builder = AndroidBuilder()
print(director.make_android(builder))
```

Запуск этого кода выдаст:

```
BIPEDAL ROBOT
Traversal modules installed:
- two legs
- four legs
Detection systems installed:
- cameras
```

Тем не менее, шаблон **Builder** не имеет большого смысла для небольших, простых классов, поскольку добавленная логика для их построения просто добавляет сложности.

Хотя, когда дело доходит до больших, сложных классов с многочисленными областями, такими как многослойные нейронные сети – паттерн **Builder** спасает жизнь.

Prototype

Проблема

Допустим нам нужно клонировать объект, но мы можем не знать его точный тип и его параметры. Все они могут быть назначены не в конструкторе или могут зависеть от состояния системы в конкретной точке во время выполнения.

Если мы попытаемся сделать это напрямую, мы добавим много ветвей зависимостей в наш код, и в итоге это может не сработать.

Решение

Шаблон проектирования **Prototype** решает проблему копирования объектов путем делегирования этой задачи самим объектам. Все объекты, которые можно копировать, должны реализовать метод **clone** и использовать его для получения точных копий самих себя.

Давайте продолжим и определим общую функцию **clone** для всех дочерних классов, а затем унаследуем ее от родительского класса:

```
from abc import ABC, abstractmethod
class Prototype(ABC):
    def clone(self):
        pass
class MyObject(Prototype):
    def __init__(self, arg1, arg2):
        self.field1 = arg1
        self.field2 = arg2
    def __operation__(self):
        self.performed_operation = True
    def clone(self):
        obj = MyObject(self.field1, field2)
        obj.performed_operation = self.performed_operation
        return obj
```

В качестве альтернативы вы можете использовать функцию **deepcopy** вместо простого назначения полей, как в предыдущем примере:

```
class MyObject(Prototype):
    def __init__(self, arg1, arg2):
        self.field1 = arg1
        self.field2 = arg2
    def __operation__(self):
        self.performed_operation = True
    def clone(self):
        return deepcopy(self)
```

Шаблон **Prototype** может быть действительно полезен в крупномасштабных приложениях, которые создают множество объектов. Иногда копирование уже существующего объекта обходится дешевле, чем создание нового.

Singleton

Проблема

Singleton – это объект с двумя основными характеристиками:

- Нельзя создавать более одного экземпляра этого объекта
- Он должен быть доступен в любом месте программы

Оба эти свойства важны, хотя на практике вы часто слышите, как люди называют что-то синглтоном, даже если у него есть только одно из этих свойств.

Наличие только одного экземпляра обычно является механизмом управления доступом к некоторому общему ресурсу. Например, два потока могут работать с одним и тем же файлом, поэтому вместо того, чтобы открывать его по отдельности, **Singleton** может предоставить им уникальную точку доступа.

Глобальная доступность важна, потому что после того, как ваш класс был создан один раз, вам нужно будет передать этот единственный экземпляр для работы с ним. И не всегда такое возможно будет повторить. Вот почему легче убедиться, что всякий раз, когда вы снова пытаетесь создать экземпляр класса, вы просто получаете тот же экземпляр, который у вас уже был.

Решение

Давайте реализуем шаблон **Singleton** , сделав объект глобально доступным и ограниченным одним экземпляром:

```
from typing import Optional
class MetaSingleton(type):
    _instance : Optional[type] = None
    def __call__(cls, *args, **kwargs):
        if cls._instance is None:
            cls._instance = super(MetaSingleton, cls).__call__(*args
        return cls._instance
class BaseClass:
    field = 5
class Singleton(BaseClass, metaclass=MetaSingleton):
    pass
```

Тип данных **Optional** , может содержать либо класс, указанный в [], либо None.

Определение метода **__call__** позволяет использовать экземпляры класса в качестве функций. Метод также вызывается во время инициализации, поэтому, когда мы вызываем что-то вроде **a = Singleton()** , вызывается метод **__call__** своего базового класса.

В Python все является объектом. Все обычные классы, которые вы пишете, а также стандартные классы, имеют **type** в качестве типа объекта. Даже **type** относится к **type** .

Это означает, что **type** является метаклассом – другие классы являются экземплярами **type** , точно так же, как переменные объекты являются экземплярами этих классов. В нашем случае **Singleton** является экземпляром **MetaSingleton** .

Все это означает, что наш метод **__call__** будет вызываться всякий раз, когда создается новый объект, и он будет предоставлять новый экземпляр, если мы еще не инициализировали его. Если у нас есть, он просто вернет уже инициализированный экземпляр.

super(MetaSingleton, cls).__call__(* args, ** kwargs) вызывает суперкласс **__call__** . Наш суперкласс в этом случае – это **type** , имеющий реализацию **__call__** , которая будет выполнять инициализацию с заданными аргументами.

Мы указали **type(MetaSingleton)** , значение, которое будет присвоено полю **_instance (cls)** .

Целью использования метакласса в этом случае, а не использрвание более простой реализацией, по сути, является возможность повторного использования кода.

В этом случае мы извлекли из него один класс, но если бы нам нужен был другой **Singleton** для другой цели, мы могли бы просто получить тот же метакласс вместо того, чтобы реализовывать по существу одно и то же.

Теперь мы можем попробовать использовать его:

```
a = Singleton()
b = Singleton()
a == b
```

```
True
```

Из-за своей глобальности целесообразно интегрировать обеспечение безопасности потоков в **Singleton** . К счастью, нам не нужно слишком много редактировать, чтобы сделать это. Мы можем просто немного отредактировать **MetaSingleton** :

```
def __call__(cls, *args, **kwargs):
    with cls._lock:
        if not cls._instance:
            cls._instance = super().__call__(*args, **kwargs)
    return cls._instance
```

Таким образом, если два потока начнут создавать экземпляр **Singleton** одновременно, один остановится на блокировке. Когда менеджер контекста снимает блокировку, другой вводит оператор **if** и видит, что экземпляр действительно уже был создан другим потоком.

Object Pool

Проблема

Допустим у нас есть класс в нашем проекте, который называется **MyClass** . **MyClass** очень полезен для нас и мы его часто используем на протяжении всего проекта, хотя и в течение коротких периодов времени.

Однако его создание и инициализация очень дороги, и наша программа работает очень медленно, потому что ей постоянно нужно создавать новые экземпляры, чтобы использовать их только для нескольких операций.

Решение

Мы можем создать пул объектов, при этом сами объекты будут создавать при создании пула. Всякий раз, когда нам нужно использовать объект типа **MyClass** , мы будем получать его из пула, далее использовать его, а затем перемещать обратно в пул для повторного использования.

Если объект имеет какое-то начальное состояние по умолчанию, освобождение всегда будет перезапускать его.

Реализуем пример **ObjectPool** и сначала определим **MyClass** :

```
class MyClass:
    # Return the resource to default setting
    def reset(self):
        self.setting = 0
class ObjectPool:
    def __init__(self, size):
        self.objects = [MyClass() for _ in range(size)]
    def acquire(self):
        if self.objects:
            return self.objects.pop()
        else:
            self.objects.append(MyClass())
            return self.objects.pop()
```

```
    def release(self, reusable):
        reusable.reset()
        self.objects.append(reusable)
```

И чтобы проверить это:

```
pool = ObjectPool(10)
reusable = pool.acquire()
pool.release(reusable)
```

Обратите внимание, что это простейшая реализация, и на практике этот шаблон может использоваться вместе с **Singleton** для обеспечения единого пула, доступного в глобальном масштабе.

Обратите внимание, что полезность этого шаблона обсуждается в языках, которые используют сборщик мусора. Распределение объектов, которые занимают только память (то есть никаких внешних ресурсов), как правило, является относительно недорогим в таких языках, в то время как много «живых» ссылок на объекты могут замедлить сборку мусора, потому что GC нужно проходить через все ссылки.

Заключение

Мы рассмотрели наиболее важные порождающие шаблоны проектирования в Python – проблемы, которые они решают, и способы их решения.

Знание шаблонов проектирования – чрезвычайно удобный набор навыков для всех разработчиков, поскольку они предоставляют решения общих проблем, возникающих в программировании.

Зная о мотивах и решениях, вы также можете избежать случайного появления анти-паттерна при попытке решить проблему.