

ООП в Python

Часть III



Типы методов

В Python есть еще два типа функций, которые могут быть созданы в классе:

- Статические методы.
- Методы класса.



@classmethod
and
@staticmethod

Статический метод

- Может быть определен только внутри класса, но не для объектов класса.
- Его можно вызвать непосредственно из класса по ссылке на имя класса.
- Он не может получить доступ к атрибутам класса
- Статический метод связан с классом. Таким образом, он не может изменить состояние объекта.
- Он также используется для разделения служебных методов для класса.
- Все объекты класса используют только одну копию статического метода.

Есть два способа определить статический метод в Python:

Использование метода **staticmethod()**

Использование декоратора **@staticmethod**.

Определение Static Method in Python

#Перед реализацией метода нужно добавить декоратор
@staticmethod

```
class Calc:  
    @staticmethod  
    def add(arg1, arg2):  
        return arg1 + arg2  
Calc.add(2,3)
```

После объявления класса сделать обычный метод
статическим

```
class Employee:  
    def sample(x):  
        print('Inside static method', x)
```

```
Employee.sample = staticmethod(Employee.sample)  
# call static method  
Employee.sample(10)
```

Вызов через класс

```
class DB:  
    # Определяем статический метод используя декоратор  
    @staticmethod  
    def get_conn():  
        print("Получим дискриптор соединения с БД!")  
  
    # Вызов метода  
conn = DB.get_conn()
```

Вызов через объект

```
class DB:  
    # Определяем статический метод используя декоратор  
    @staticmethod  
    def get_conn():  
        print("Получим дискриптор соединения с БД!")  
  
test_system_db = DB()  
test_system_db.get_conn()
```

При таком вызове не происходит подкапотной передачи `self`. А значит нет доступа к атрибутам объекта.

А можно как то это исправить ?

Учим статический метод работать с экземпляром класса

```
class DB:  
  
    def __init__(self):  
        self.name = "TestSystem"  
  
    @staticmethod  
    # Определяем переменную которая будет принимать объект  
    def get_conn(self):  
  
        print(f"Получим дискриптор соединения с БД  
        {self.name}!")  
  
test_sysytem_db = DB()  
test_sysytem_db.get_conn(test_sysytem_db)
```

Вызов статического метода из обычного

```
class DB:  
    # Определяем статический метод используя декоратор  
    @staticmethod  
    def get_conn():  
        print("Получим дискриптор соединения с БД!")  
    def get_session(self):  
        # вызов статического метода  
        conn = DB.get_conn()  
  
test_system_db = DB()  
test_system_db.get_session()
```

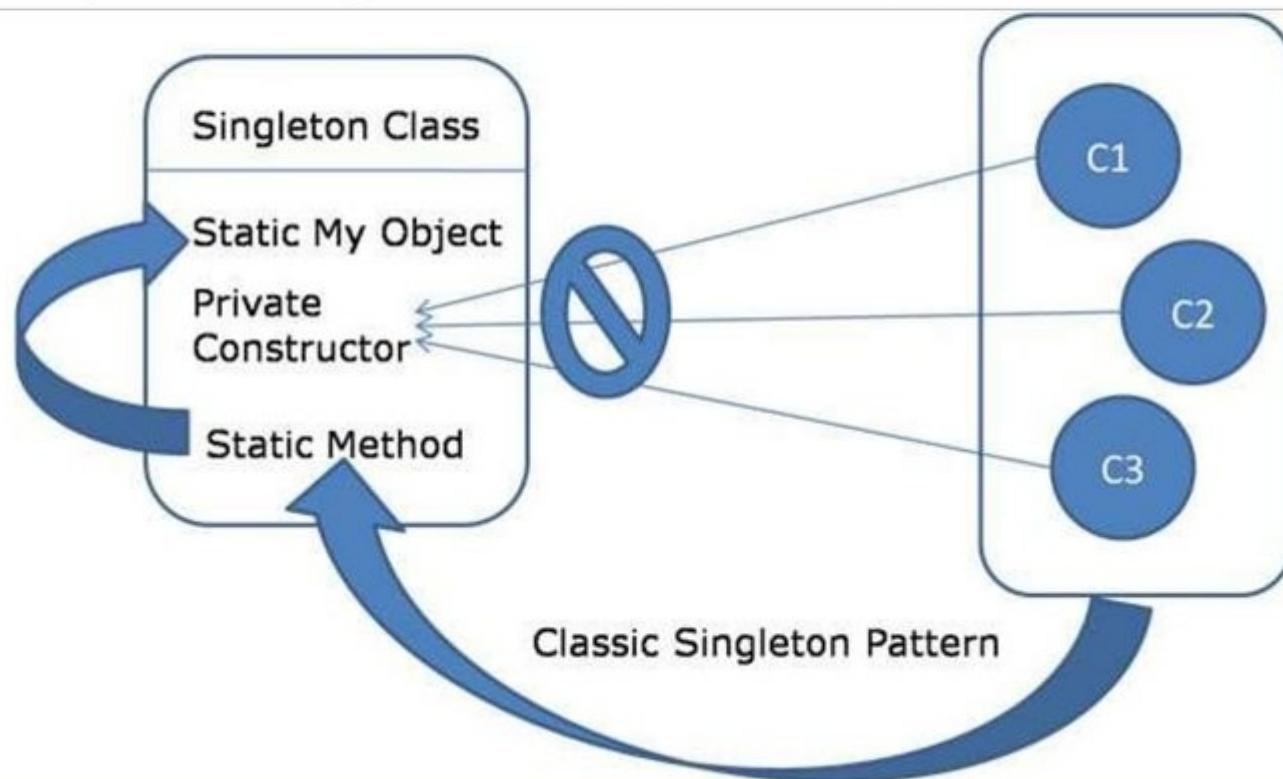


Для чего мы используем static methods ?

Паттерн Singleton

Синглтон (одиночка) – это паттерн проектирования, цель которого ограничить возможность создания объектов данного класса одним экземпляром. Он обеспечивает глобальность до одного экземпляра и глобальный доступ к созданному объекту.

Паттерн Singleton для подключения бд



```
class DB:
    __instance__ = None

    def __init__(self):
        # Проверяем конструктор на сущ. экземпляр
        if DB.__instance__ is None:
            DB.__instance__ = self
        else:
            raise Exception("We can not create another class")

    @staticmethod
    def get_instance():
        # We define the static method to fetch instance
        if not DB.__instance__:
            DB()
        return DB.__instance__

mongo = DB()
print(mongo)

my_db = DB.get_instance()
print(my_db)

another_db = DB.get_instance()
print(another_db)

new_gover = DB() ← Что будет при вызове ?
```

Методы класса

@classmethod — это метод, который получает класс в качестве неявного первого аргумента, точно так же, как обычный метод экземпляра получает экземпляр. Это означает, что вы можете использовать класс и его свойства внутри этого метода, а не конкретного экземпляра.

Метод класса

- Может быть определен только внутри класса
- Получает класс в качестве неявного первого аргумента
- Его можно вызвать непосредственно из класса по ссылке на имя класса.
- Он не может получить доступ к атрибутам класса
- Не может изменить состояние объекта.
- Все объекты класса используют только одну копию метода класса.

Есть два способа определить статический метод в Python:

Использование метода **classmethod()**

Использование декоратора **@classmethod**.

Объявление метода `@classmethod`

```
class MyClass():
    TOTAL_OBJECTS = 0
    def __init__(self):
        MyClass.TOTAL_OBJECTS = MyClass.TOTAL_OBJECTS + 1

    @classmethod
    def total_objects(cls):
        print("Total objects: ", cls.TOTAL_OBJECTS)

# Вызов через объект
my_obj1 = MyClass()
my_obj1.total_objects()
# Вызов через класс
MyClass.total_objects() ← что вернет вызов ?
```

Объявление метода `classmethod()`

```
class Coffee:
    def __init__(self, milk, beans):
        self.milk = milk # percentage
        self.coffee = 100 - milk
        self.beans = beans

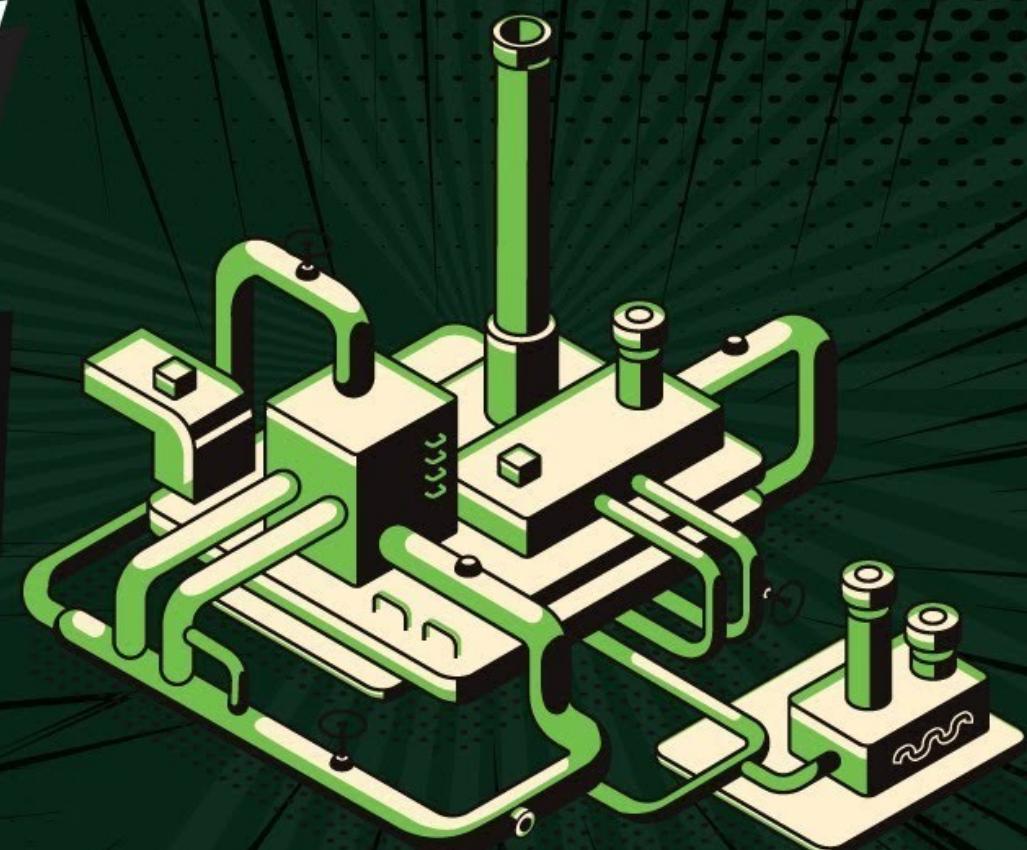
    def __repr__(self):
        return f'Milk={self.milk}% Coffee={self.coffee}%
               Beans={self.beans}'

    def cappuccino(cls):
        return cls(80, 'Arrabica')

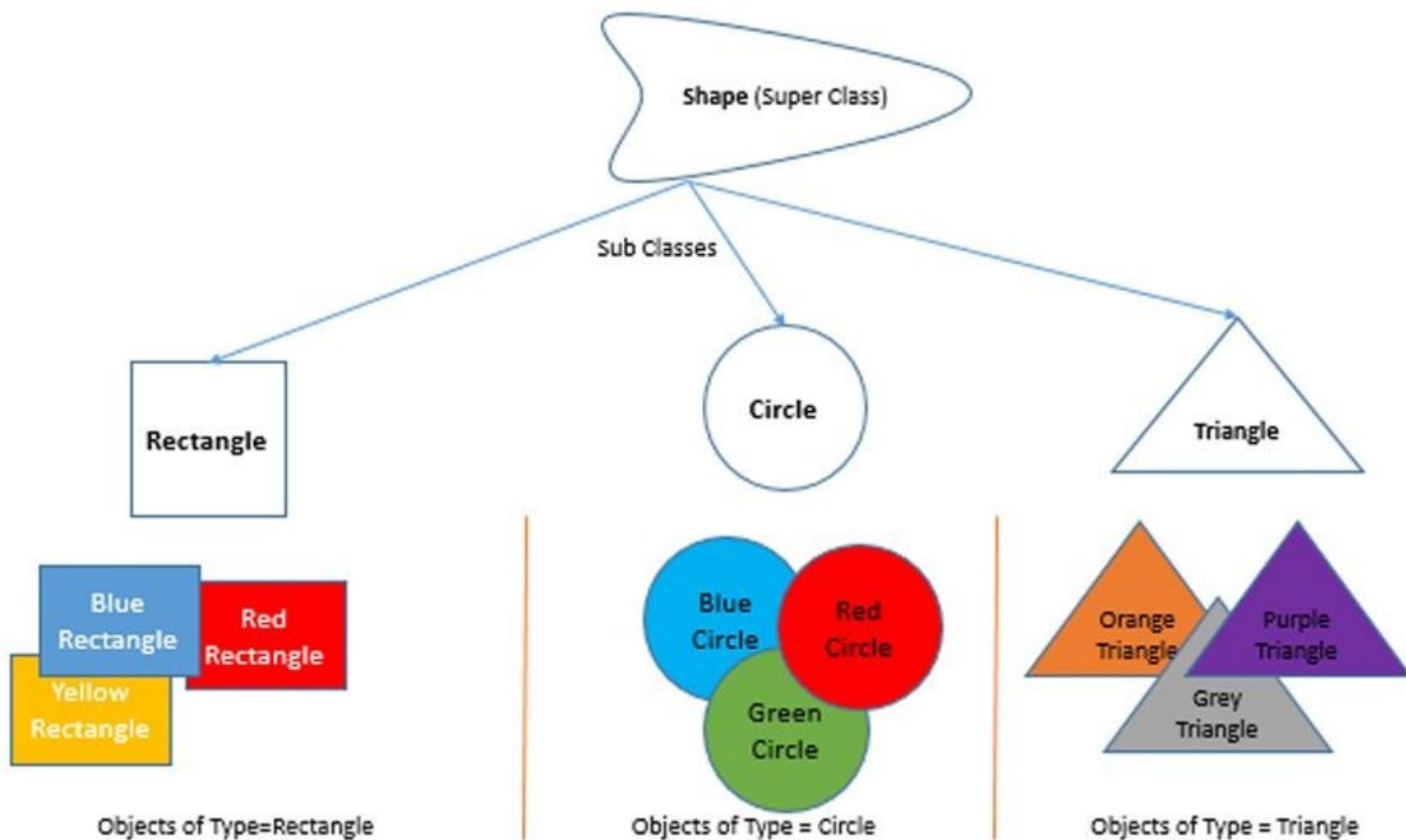
Coffee.cappuccino = classmethod(Coffee.cappuccino)
print(Coffee.cappuccino())
```

Шаблон проектирования “Фабрика”

FACTORY
PATTERN



Class Object in Python



```
# Рассмотрим фабрику
class Shape:
    def draw(self):
        raise NotImplementedError('This method should
have implemented.')

class Triangle(Shape):
    def draw(self):
        print("треугольник")

class Rectangle(Shape):
    def draw(self):
        print("прямоугольник")

class ShapeFactory:
    def getShape(self, shapeType):
        if shapeType == 'Triangle':
            return Triangle()
        elif shapeType == 'Rectangle':
            return Rectangle()
        else:
            pass
obj = ShapeFactory()
trgl = obj.getShape("Triangle")
trgl.draw()
```

```
class Coffee:
    def __init__(self, milk, beans):
        self.milk = milk # percentage
        self.coffee = 100-milk # percentage
        self.beans = beans
    def __repr__(self):
        return f'Milk={self.milk}% Coffee={self.coffee}%
                           Beans={self.beans}'

@classmethod
def cappuccino(cls):
    return cls(80, 'Arrabica')

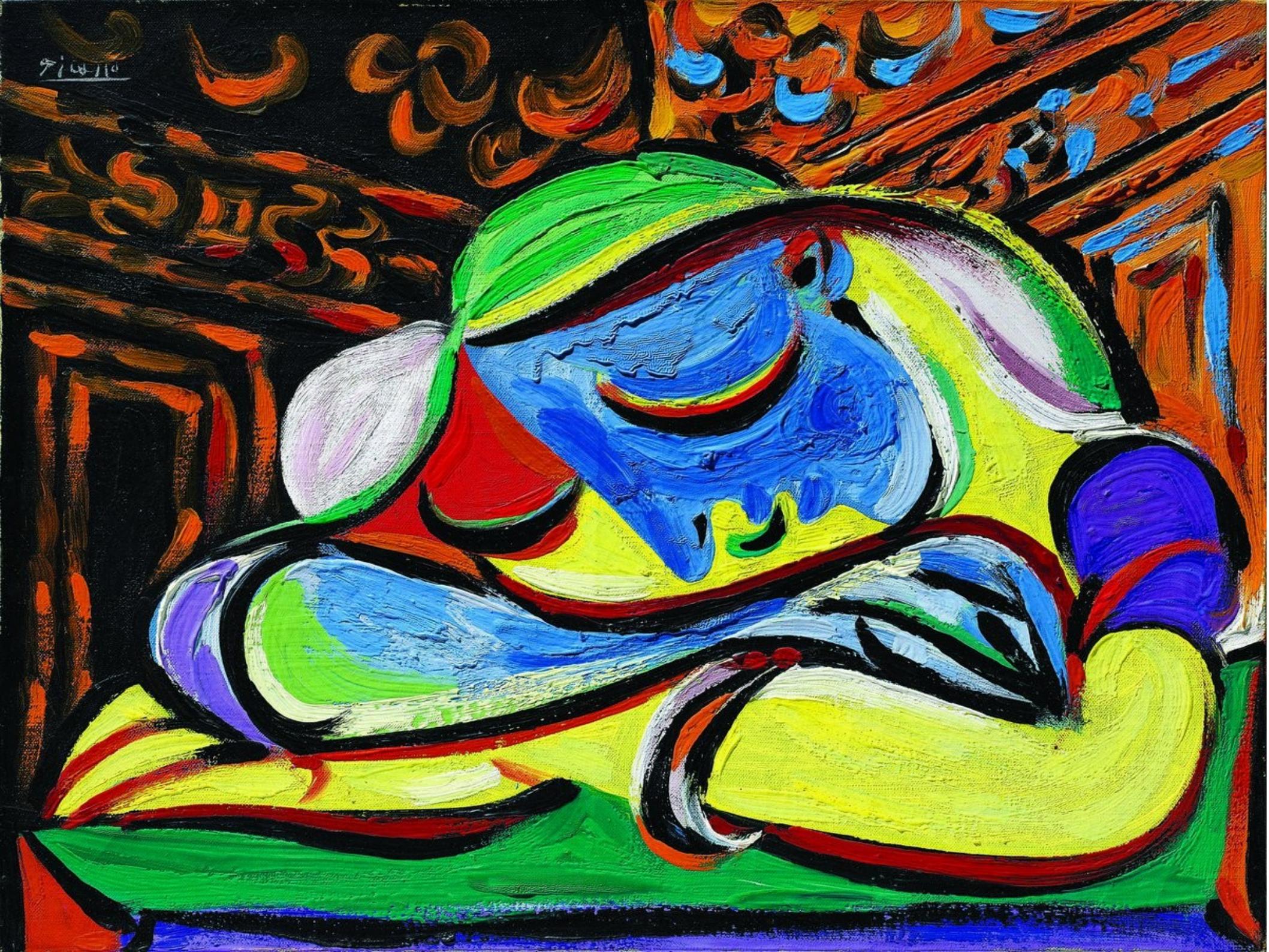
@classmethod
def espresso_macchiato(cls):
    return cls(30, 'Robusta')

@classmethod
def latte(cls):
    return cls(95, 'Arrabica')

print(Coffee.cappuccino())
print(Coffee.espresso_macchiato())
print(Coffee.latte())
```

Заключение

Декоратор аннотации `@classmethod` используется для создания фабричных методов, поскольку они могут принимать любой ввод и предоставлять объект класса на основе параметров и обработки.



Абстрактные классы и методы в Python

- **Абстрактные классы** – реализуют механизм организации объектов в иерархии, позволяющий утверждать о наличии требуемых методов.
- **Абстрактный метод** – это метод для которого отсутствует реализация. Объявляется с помощью декоратора `@abstractmethod` из модуля `abc`

Абстрактные классы и методы в Python

Чтобы объявить абстрактный класс, нам сначала нужно импортировать модуль `abc`. Давайте посмотрим на пример.

```
from abc import ABC
class abs_class(ABC):
    @abstractmethod
    def render(self):
        pass
```

Абстрактный базовый класс – класс, на основе которого нельзя создать экземпляр объекта.

Абстрактный метод – это метод, определенный в базовом классе, но он может не обеспечивать какую-либо реализацию

Абстрактный базовый класс – класс, на основе которого нельзя создать экземпляр объекта.

```
from abc import ABC
class AbsClass(ABC):
    @abstractmethod
    def render(self):
        pass

obj = AbsClass() # вызовет ошибку
```

Чтобы объявить абстрактный класс, нам сначала нужно импортировать модуль abc .

```
from abc import ABC, abstractmethod

class Absclass(ABC):
    def print(self, x):
        print("Passed value: ", x)
    @abstractmethod
    def task(self):
        print("We are inside Absclass task")

class test_class(Absclass):
    def task(self):
        print("We are inside test_class task")

test_obj = test_class()
test_obj.task()
test_obj.print("10")
```



Продолжение следует....