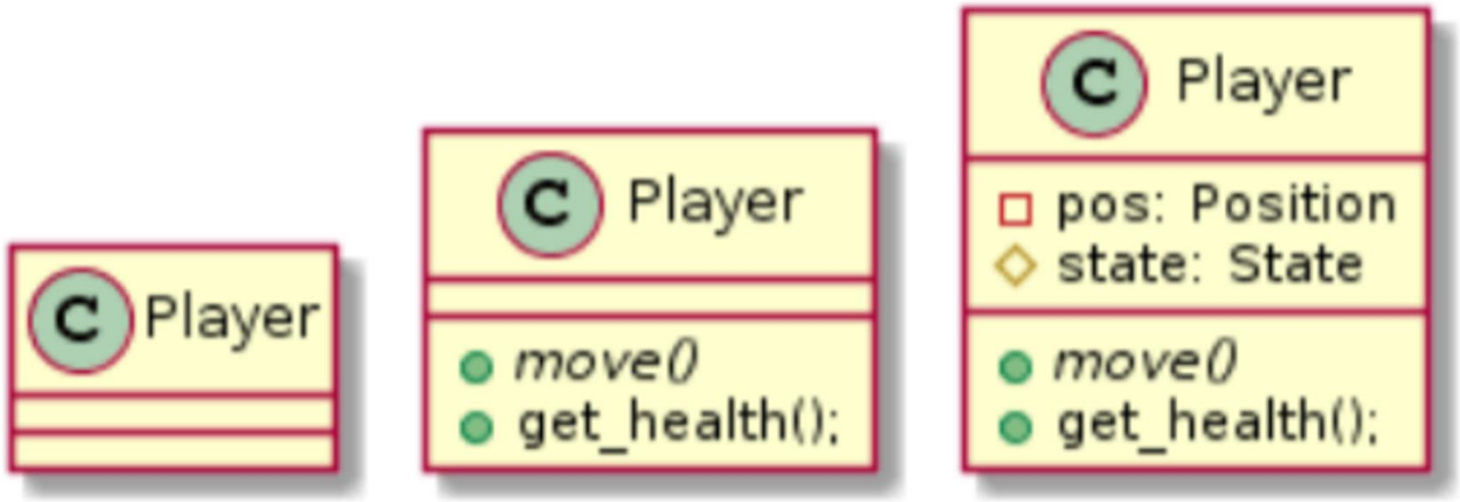
Программирование на Python.

Тема №4.2. Абстрактные классы.

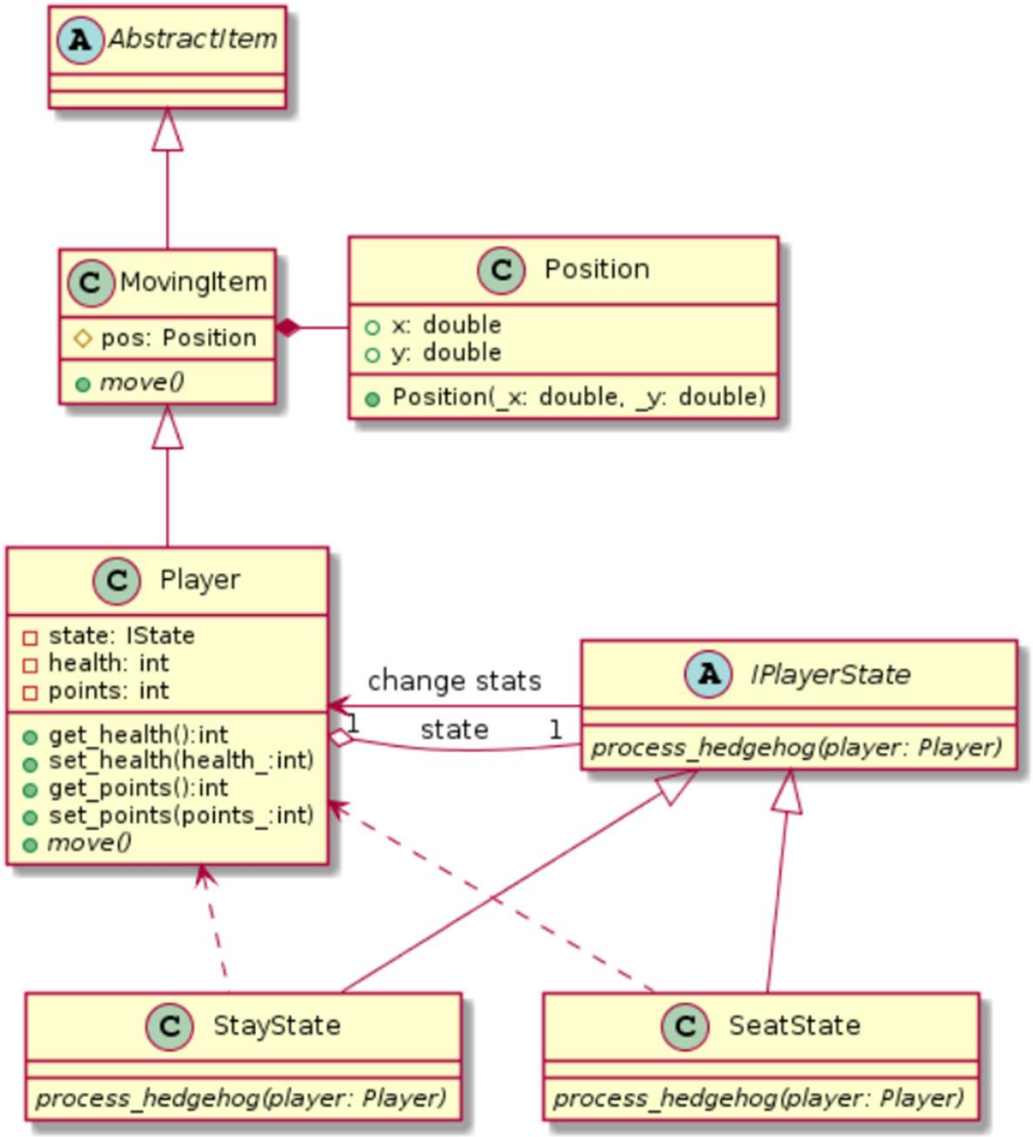
Диаграмма класса

Диаграммы классов (UML-диаграмма) может применяться для проектирования и документирования проекта



Такие диаграммы могут иметь разный уровень детализации и иллюстрировать как взаимосвязи объектов, так и их внутреннюю структуру

Пример диаграммы



Эта диаграмма отображает не только внутреннюю структуру объектов, но и отношения между классами – родительские и дочерние классы.

A – абстрактные классы, C – реальные классы.

Абстрактные классы и методы

Задачи, подобные упражнению с мебелью могут быть легко решены с помощью Абстрактных классов.

Абстрактные классы описывают сущности, не имеющие конкретного воплощения: например, сущность “мебель”. Есть конкретные реализации – стол, шкаф, тумба.

abc

Инструменты для работы с абстрактными классами описаны в модуле abc. Ключевые элементы- базовый абстрактный класс **ABC** и декоратор **@abstractmethod**

```
import abc
class Furniture(abc.ABC):
    @abc.abstractmethod
    def info(self): pass
```

Мы **наследуемся от базового абстрактного класса** и отмечаем методы абстрактного класса специальными декораторами.

Абстрактные методы не должны иметь конкретного функционала, поэтому мы вызываем в нем оператор **pass.**

При попытке создания экземпляра из абстрактного класса, мы получим ошибку:

```
import abc
class Furniture(abc.ABC):
    @abc.abstractmethod
    def info(self): pass

new = Furniture()
```

```
TypeError: Can't instantiate abstract class
 Furniture with abstract method info
```

Для чего нужны **ABC?**

Абстрактные классы позволяют создать «схему» для дальнейшей реализации дочерних классов. При помощи **ABC,** мы можем проконтролировать, что все дочерние классы имеют одинаковый интерфейс (методы с одними и теми же названиями), т.к. дочерние классы, созданные из абстрактного **ОБЯЗАНЫ** реализовать все абстрактные методы абстрактного класса

Пример

Все наследники класса **Furniture** обязаны будут определить метод info.

Если закомментировать этот метод и попытаться создать экземпляр, получим ошибку:

```
TypeError: Can't instantiate abstract class Table
with abstract method info
```

```python
import abc
class Furniture(abc.ABC):
    @abc.abstractmethod
    def info(self): pass
class Table(Furniture):
    def __init__(self,name):...
    def info(self):
        return f"Стол: {self.name}"
class Shelf(Furniture):
    def __init__(self,name):...
    def info(self):
        return f"Шкаф: {self.name}"


new = Table('кухонный')
print(new.info())
```

Конструктор, атрибуты, неабстрактные методы ABC

Абстрактные классы могут определять Конструктор, атрибуты и неабстрактные методы, которые могут применяться в классах-наследниках:

```python
import abc
class Furniture(abc.ABC):
    def __init__(self, width, height):
        self.w = width
        self.h = height
    @abc.abstractmethod
    def info(self): pass

    def appearance(self):
        print("Стоит в интерьере")
```

```python
class Table(Furniture):
    def __init__(self,width,height, name):
        super().__init__(width,height)
        self.name = name
    def info(self):
        return f"Стол: {self.name}"
```

```python
new = Table( width: 60,
             height: 100,
             name: 'кухонный')
print(new.appearance())
```

Пример

Экземпляр класса **Table** теперь использует родительский **__init__**, обязан иметь реализацию абстрактного метода **info()**, и имеет доступ к родительскому методу **apperance()**

```python
import abc
class Furniture(abc.ABC):
    def __init__(self, width, height):
        self.w = width
        self.h = height
    @abc.abstractmethod
    def info(self): pass

    def appearance(self):
        print("Стоит в интерьере")
```

```python
class Table(Furniture):
    def __init__(self,width,height, name):
        super().__init__(width,height)
        self.name = name
    def info(self):
        return f"Стол: {self.name}"
```

```python
new = Table( width: 60,
             height: 100,
             name: 'кухонный')
print(new.appearance())
```

Абстрактные classmethod, staticmethod, property

Абстрактные классы могут иметь **абстрактные методы классов абстркатные статические методы** и **абстрактные дескрипторы property:**

```
class Demo(ABC):
    @classmethod
    @abstractmethod
    def my_abstract_classmethod(cls,*args):
        pass
    @staticmethod
    @abstractmethod
    def my_abstract_staticmethod(*args):
        pass

    @property
    @abstractmethod
    def value(self): pass
    @value.setter
    @abstractmethod
    def value(self, val): self.__value = val
```

Атрибуты абстрактных классов

Python 3.6+ допускает аннотирование атрибутов в абстрактных классов таким образом:

```
class Demo(ABC):
    path: str
    value: int
    @abstractmethod
    def my_abstract_method(self):
        pass
```

Вы не указываете явные значения атрибутов, только тип данных.

Важно! При такой реализации – дочерние классы могут не предоставить явного определения таким атрибутам, и вы не сможете контролировать их наличие!

Упражнение – абстрактная фабрика

Реализуйте классы для абстрактного стола, дивана и кресла: AbstractChair, AbstractTable, AbstractSofa. Каждый из объектов должен иметь абстрактные методы has_legs() и sit_on().

Создайте класс – **AbstractFurnitureFactory,** обладающий методами create_table(), create_sofa(), create_chair(), возвращающими экземпляры абстрактных объектов.

На основе этого абстрактного класса, реализуйте класс **ModernFurnitureFactory** и **LoftFurnitureFactory**, создающие мебель соответствующего типа. (Modern или Loft).

Напрмер – при вызове **LoftFurnitureFactory.create_table**() должен создаваться экземпляр **LoftTable** – не абстрактного воплощения класса **AbstractTable**.

Реализации реальных классов предметов мебели можно описать внутри классов фабрик.

__new__

Обычно, создание собственной реализации метода .__new__() необходима только тогда, когда нужно управлять созданием нового экземпляра класса на низком уровне. Теперь, если нужна кастомная реализация этого метода, то следует выполнить несколько шагов:

- Создать новый экземпляр, вызвав super().__new__() с соответствующими аргументами.
- Настроить новый экземпляр в соответствии с конкретными потребностями.
- возвратить новый экземпляр, чтобы продолжить процесс создания экземпляра.

Пример

Ниже представлен общий пример использования :

```
class SomeClass:
    def __new__(cls, *args, **kwargs):
        instance = super().__new__(cls)
        # В этом месте можно настроить свой экземпляр...
        return instance
    def __init__(self, val):
        self.val = val
```

Вызов super().__new__(cls) необходим, чтобы получить доступ к методу object.__new__() родительского класса object, который является базовой реализацией метода .__new__() для всех классов Python

Функция object.__new__() принимает только один аргумент - класс для создания экземпляра. Если вызывать object.__new__() с большим количеством аргументов, то получим исключение TypeError.

Практическое применение

Метод .__new__() фабрика случайных объектов

```python
from random import choice

class Pet:
    def __new__(cls):
        # выбираем класс случайным образом
        other = choice([Dog, Cat, Python])
        # подставляем вместо собственного класса `cls`
        # случайно выбранный `other`
        instance = super().__new__(other)
        print(f"Я {type(instance).__name__}!")
        return instance

    def __init__(self):
        print("Класс `Pet` никогда не запустится!")
```

```python
class Dog:
    def communicate(self):
        print("Гав! Гав!")

class Cat:
    def communicate(self):
        print("Мяу! Мяу!")

class Bird:
    def communicate(self):
        print("Чик! Чирик!")
```

Реализация singleton-Объекта

Singleton-классы позволяют создать только один экземпляр класса

```
class Singleton:
    _instance = None
    def __new__(cls, *args, **kwargs):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
        return cls._instance


a = Singleton()
b = Singleton()
print(a is b)       #True
```

В приведенном выше примере Singleton *не предоставляет реализацию* .__init__(). *Если когда-нибудь понадобится такой класс с методом* .__init__(), *то имейте в виду, что этот метод будет запускаться каждый раз, когда вы вызываете конструктор* Singleton(). *Такое поведение может вызвать непредсказуемые эффекты инициализации и ошибки.*