

Углубленный Python

ООП

Воркшоп

Тарасов Артём



Основные свойства ООП

Полиморфизм: в разных объектах одна и та же операция может выполнять различные функции.

Инкапсуляция: можно скрыть ненужные внутренние подробности работы объекта от окружающего мира.

Наследование: можно создавать специализированные классы на основе базовых.

(*)**Композиция:** объект может быть составным и включать в себя другие объекты.

Инкапсуляция

```
class MyClass:
    def __init__(self):
        self._protected_var = 10    #protected attribute

        self.__private_var = 20    #private attribute

    def _protected_method(self):    #protected method
        return "protected method"

    def __private_method(self):    #private method
        return "private method"

    def public_method(self):
        result = self._protected_var
        s = ""
        i = 0
        s += self._protected_method()
        i += self.__private_var
        s += self.__private_method()
        return s
```



Наследование

```
# Определение базового класса (суперкласса)
```

```
class Animal:
    def __init__(self, name):
        self.name = name
```

```
    def speak(self):
        pass
```

```
# Определение подкласса, который наследует базовый класс
```

```
class Dog(Animal):
    def speak(self):
        return f"{self.name} говорит: Гав!"
```

```
# Создание объектов классов
```

```
animal = Animal("Животное")
dog = Dog("Бобик")
```

```
# Вызов метода speak() для каждого объекта
```

```
print(animal.speak()) # Выведет: None (нет реализации в базовом классе)
print(dog.speak())    # Выведет: Бобик говорит: Гав!
```



Полиморфизм

```
# Определение базового класса (суперкласса)
```

```
class Shape:
    def area(self):
        pass
```

```
# Определение подклассов с переопределением метода area()
```

```
class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius
```

```
    def area(self):
        return 3.14 * self.radius ** 2
```

```
class Square(Shape):
    def __init__(self, side length):
        self.side length = side length
```

```
    def area(self):
        return self.side length ** 2
```



Композиция

```
class Engine:
    def start(self):
        print("Двигатель запущен")
```

```
    def stop(self):
        print("Двигатель остановлен")
```

```
class Car:
    def __init__(self):
        self.engine = Engine() # Композиция: объект Car содержит объект
Engine
```

```
    def drive(self):
        print("Автомобиль начал движение")
        self.engine.start()
```

```
    def park(self):
        print("Автомобиль припаркован")
        self.engine.stop()
```



Основные свойства ООП

В некоторых случаях использование композиции (составных объектов) может быть более предпочтительным, чем наследование, для достижения гибкости и переиспользования кода.

Вопросы к вам:

- 1) Разница между процедурным программированием и ООП?
- 2) Какой элемент отвечает за инициализацию полей класса?
- 3) Какие типы данных допустимы в множестве?
- 4) Чем отличаются атрибуты и методы класса от атрибутов и методов объекта?
- 5) Что такое абстрактный класс и интерфейс? Как они реализуются в Python?
- 6) Для чего используются встроенные функции `isinstance()` и `issubclass()` для работы с объектами и классами?

Абстрактные классы и интерфейсы


Абстрактные классы - это классы, которые сами по себе не могут быть инстанцированы, они служат как шаблоны для других классов.

Интерфейсы - это набор методов, которые класс обязан реализовать, если он имплементирует данный интерфейс.

Абстрактные классы позволяют создавать общие атрибуты и методы для группы классов, при этом оставляя реализацию конкретных методов подклассам.

Интерфейсы, с другой стороны, определяют обязательство для классов реализовать определенные методы. Интерфейсы в Python могут быть определены как абстрактные классы с абстрактными методами, и классы должны реализовать эти методы, чтобы считаться имплементирующими данный интерфейс.

Почему ООП важен в разработке



1

Помогает нам организовать код в виде объектов, что повышает модульность приложений. Это означает, что мы можем разрабатывать и тестировать части кода независимо, что делает процесс разработки более управляемым и масштабируемым

2

Уменьшение сложности кода. С помощью абстракций и инкапсуляции мы можем скрывать детали реализации и предоставлять пользовательский интерфейс, что упрощает взаимодействие с кодом.

3

Способствует повторному использованию кода.

Ещё чуть чуть про наследование

Простое наследование: Класс, от которого произошло наследование, называется базовым или родительским. Классы, которые произошли от базового, называются потомками, наследниками или производными классами.

Множественное наследование: При множественном наследовании у класса может быть более одного предка. В этом случае класс наследует методы всех предков. Достоинства такого подхода в большей гибкости.

Множественное наследование потенциальный источник ошибок, которые могут возникнуть из-за наличия одинаковых имен методов в предках.

Множественное наследование

```
class Parent1:
    def method1(self):
        return "Метод 1 из Parent1"
```

```
class Parent2:
    def method2(self):
        return "Метод 2 из Parent2"
```

```
class Child(Parent1, Parent2):
    def method3(self):
        return "Метод 3 из Child"
```

```
child = Child()
```

```
result1 = child.method1()
result2 = child.method2()
result3 = child.method3()
```

```
# Вывод результатов
print(result1)  # Выведет: Метод 1 из Parent1
print(result2)  # Выведет: Метод 2 из Parent2
print(result3)  # Выведет: Метод 3 из Child
```



Duck Typing

Типичная **проблема**: Объект некоторого типа требуется передавать в качестве аргумента в различные функции. Разным функциям нужны разные свойства и методы этого объекта. При этом хотелось бы все эти функции сделать полиморфными, то есть способными принимать объекты разных типов.

В Питоне используется концепция, называемая Duck Typing:
“Если ЭТО ходит как утка, и крякает, как утка – значит это утка.”

Т.е., если у объекта есть все нужные функции, свойства и методы, то он подходит в качестве аргумента.

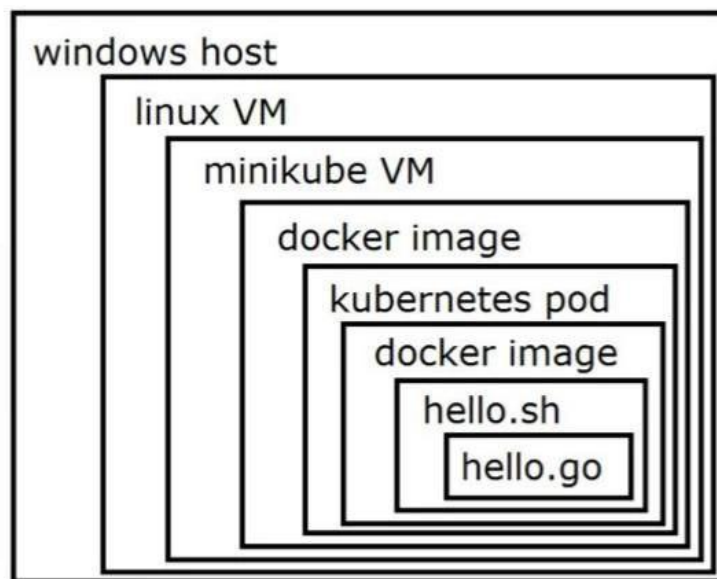
Например, в функцию

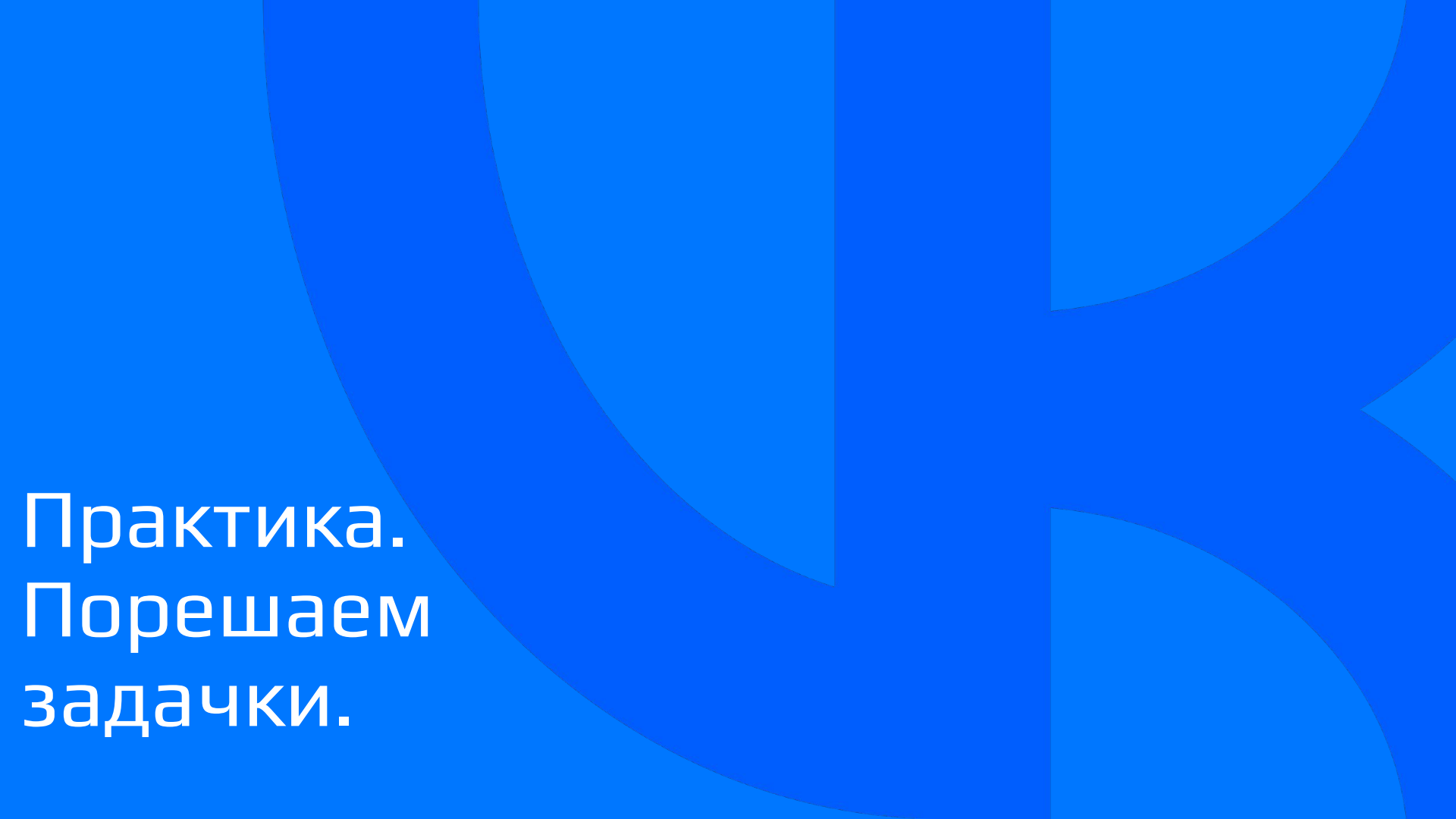
```
def f(x):  
    return x.get_value_()
```

Можно передавать объект любого типа, лишь бы у него был метод `get_value()`.

Помоги Даше-разработчице
понять, на каком уровне протекает
абстракция

[Translate Tweet](#)





Практика.
Порешаем
задачи.

deepcopy

```
# Глубокая копия объекта создает новый объект, который является полностью  
независимой копией исходного объекта.
```

```
import copy
```

```
original_list = [1, [2, 3], [4, 5]]  
copied_list = copy.deepcopy(original_list)
```

```
# Теперь изменение copied_list не влияет на original_list и наоборот.
```

```
copied_list[1][0] = 99  
print(original_list)    # Выведет: [1, [2, 3], [4, 5]]  
print(copied_list)      # Выведет: [1, [99, 3], [4, 5]]
```

Менеджеры контекста

```
import sys
class LookingGlass:
    def enter(self) -> str:
        self.original_write = sys.stdout.write
        sys.stdout.write = self.reverse_write
        return "Some String"
```

```
    def reverse_write(self, text):
        self.original_write(text[::-1])
```

```
    def exit(self, exc_type, exc_value, traceback):
        sys.stdout.write = self.original_write
        if exc_type is ZeroDivisionError:
            print("Please DO NOT divide by zero!")
            ## сообщаем интерпретатору, что было обработано исключение
        return True
```

```
with LookingGlass() as what:
    print("Alice, Kitty and Snowdrop")
    print(what)
```

```
>>> pordwonS dna yttiK ,ecilA
>>> gnirts emoS
```



«Хрупкий словарь»



Давайте создавать
код, который меняет
мир, и вдохновлять
следующее поколение
разработчиков.



education



Q&A.
С радостью
отвечу на ваши
вопросы.