Методические указания

Урок 14.1. Основы ООП

Задачи урока:

- Аргументы по умолчанию
- Инкапсуляция

0. Подготовка к уроку

До начала урока преподавателю необходимо:

1) Просмотреть, как ученики справились с домашним заданием
2) Прочитать методичку

1. Аргументы по умолчанию

Учитель: Сегодня мы продолжим знакомство с классами и начнем с аргументов по умолчанию. На прошлом занятии при создании экземпляра класса мы передавали необходимые значения в скобках(скорость, цвет), Но бывают случаи когда какие то аргументы могут быть необязательными, как пример иметь по умолчанию какое то значение. Например если пользователь не указал при создании объекта цвет, то мы можем указать чтобы он брался из значения по умолчанию. Подобное, кстати работает и с обычными функциями. Давайте рассмотрим пример у классов, а заодно и функций.

```
def get_name(name='Иван'):
    print(name)

get_name()
get_name('Петр')
```

Результат

```
Иван
Петр
```

```
class Car:
    def __init__(self, speed, color='Yellow'):
        self.speed = speed
        self.color = color
```

```
car1 = Car(100)
car2 = Car(90, 'Blue')

print(f'{car1.color=}')
print(f'{car2.color=}')
```

Результат

```
car1.color='Yellow'
car2.color='Blue'
```

Как видим все работает аналогично. При использовании значения по умолчанию, если пользователь не укажет данное значение, то оно будет по умолчанию. **Важно**: если вы начали указывать для какого то аргумента значение по умолчанию, то все остальные аргументы после него должны тоже иметь значения по умолчанию, иначе возникнет ошибка. Единственное возможно, что вам здесь не понятно это строки в print. Ничего магического тут нет, так работает f-строка. Мы можем указать сразу после переменной знак равно и тогда выводится не только значение, но и сама переменная.

Давайте рассмотрим более сложный вариант

```
class Car:
    def __init__(self, speed, color='Yellow', owner=None) -> None:
        self.speed = speed
        self.color = color
        self.owner = owner

    def say_owner(self):
        if self.owner:
            print(f'Владелец {self.owner}')
        else:
            print('У данного автомобиля нет владельца')


car1 = Car(100, 'green', 'Иван')
car2 = Car(90, 'Blue')
car1.say_owner()
car2.say_owner()
```

В примере выше мы в методе say_owner проверяем если есть какое то значение, то вывести нам его, иначе вывести сообщение о отсутствии владельца.
Появляется еще одна проблема, а что если я хочу указать у первого автомобиля владельца, а цвет оставить по умолчанию? Попробуйте так сделать. Какую логическую ошибку вы видите? Как думаете как можно ее исправить?

На самом деле если мы хотим передавать значения не во все аргументы или же передаем не по порядку, то мы можем напрямую указать для какого аргумента какое значение

```
class Car:
    def __init__(self, speed, color='Yellow', owner=None) -> None:
        self.speed = speed
        self.color = color
        self.owner = owner

    def say_owner(self):
        if self.owner:
            print(f'Владелец {self.owner}')
        else:
            print('У данного автомобиля нет владельца')


car1 = Car(speed=100, owner='Иван')
car2 = Car(90, 'Blue')
car1.say_owner()
car2.say_owner()
```

Отлично с аргументами разобрались.

2. Инкапсуляция

Учитель: Инкапсуляция — ограничение доступа к составляющим объект компонентам (методам и переменным). Инкапсуляция делает некоторые из компонент доступными только внутри класса.
В Python конечно же инкапсуляция скорее условна и является соглашением на уровне разработчиков, так как нет конкретных модификаторов доступа, запрещающих изменение значений внутри класса, но по крайней мере усложняет данное действие.

Рассмотрим несколько примеров

```
class Person:
    _age = 15

person1 = Person
print(person1._age)
person1._age = 14
print(person1._age)
```

В данном случае мы указали перед переменной нижнее подчеркивание. Данный символ показывает, что данный атрибут(или метод) не предназначен для использования вне класса.

Конечно же это как мы видим не запрещает его использовать - это скорее просто соглашение между программистами

Теперь чуть более защищенный способ

```
class Person:
    _age = 15
    def __say_hello():
        print('Привет')

person1 = Person
print(person1._age)
person1.__say_hello()
```

В данном случае у нас есть метод с двумя нижними подчеркиваниями. При запуске данного кода возникает ошибка AttributeError: type object 'Person' has no attribute '__say_hello', что говорит, что данного атрибута у класса Person нет.
Можем ли мы получить доступ к атрибуту/методу в данном случае? Да может. Для этого нам всего лишь необходимо указать после объекта название класса с нижним подчеркиванием, а сразу после него название метода.

```
class Person:
    _age = 15
    def __say_hello():
        print('Привет')

person1 = Person
print(person1._age)
# person1.__say_hello()
person1._Person__say_hello()
```

Как мы видим приватность атрибутов и методов в python - скорее условность и договоренность между разработчиками.

Также мы можем использовать специальные декораторы для написания более защищенного приложения. Тему декораторов мы пройдем далее, но это не помешает нам воспользоваться несколькими для примера. Давайте создадим опять же класс человека, который будет иметь имя.

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

Так как мы ‘спрятали’ наше имя и пользователь не знает о данном поле, мы с вами создадим метод, который будет выводит наше имя пользователю

```
def name(self):
  return self._name
```

Давайте декорируем данный метод специальным декоратором,который позволит обращаться к данному методу без вызова. Как к обычной переменной

```
@property
def name(self):
  return self._name
```

Декоратор, по сути это функция обертка. Как работает декоратор и как мы можем написать собственный, мы узнаем на занятии отведенном соответствующей теме.

Теперь мы можем обращаться к методу, как к обычной переменной

```
a = MyClass('Ivan')
print(a.name)
```

Если же нам необходимо позволить пользователю изменять имя мы создадим метод, который будет принимать новое значение имени и заменять старое значение. Для того, чтобы пользователь изменял имя с помощью выражение **a.name = 'Sergey'**, мы воспользуемся еще одним декоратором, который будет так называемым сеттером. Сеттер - это метод, который используется для установки значения свойства. Для этого имя метода мы сделаем точно таким же, как и у предыдущего метода. А декорируем с помощью собачки указав имя метода и сеттер **@name.setter**

```
@name.setter
def name(self, value):
  self._name = value
```

Готово. Теперь мы можем как получать имя, так и изменять

```
a = MyClass('Ivan')
print(a.name)
a.name = 'sergey'
print(a.name)
```

3. Решение задач

Задача 1

Написать класс, который животного, который имеет аргументы по умолчанию и приватные атрибуты или методы

```
class Animal:
    def __init__(self, name, breed='Без породы') -> None:
        self.name = name
```

```
        self.breed = breed

    def __say_breed(self):
        print(self.breed)

cat = Animal('Барсик', 'Сибирский')
cat._Animal__say_breed()
dog = Animal('Тузик')
dog._Animal__say_breed()
```

Дополнительно

Если на уроке остается время, то ученикам можно предложить начать прорешивать домашнее задание.

Домашняя работа

Задача 1