Методические указания

Урок 12.1. Основы ООП

Задачи урока:

- Познакомиться с понятием класс

0. Подготовка к уроку

До начала урока преподавателю необходимо:

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

1. Класс в Python

Учитель: Сегодня мы познакомимся с понятие ООП.

Объектно-ориентированное программирование - методология (парадигма) программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования.
В прошлых темах объекты так или иначе использовались в коде, однако сам стиль программирования был процедурным. Написание программы в исключительно таком стиле вполне возможно, однако с увеличением объема кода и усложнения логики обработки данных, объектно-ориентированный стиль дает значительные преимущества.

Написание программы используя процедурный подход рано или поздно приводит к ряду трудноразрешимых проблем.
Например., нам необходимо реализовать работу с некоторым числом окружностей.
Минимальные данные, которые нам необходимы для создания отдельной окружности - координаты ее центра x и y, а также радиус r.
Мы можем записать это как:

```
circle = [-2, 3, 10]
```

Данное решение имеет 3 недостатка:

1. Разработчик может подразумевать [x, y, r] или [r, x, y], кроме того, для обращения к параметру окружности необходимо точно знать его индекс.
2. Отсутствие контроля за значениями.

3. Малая эффективность использования существующего кода.

Третья проблема возникает в случае, когда встает необходимость одновременно работать не только с окружностями, но и с другими геометрическими фигурами.

Python - объектно-ориентированный язык, а соответственно все, что мы не видим является объектом какого либо класса. Узнать тип объекта нам поможет функция type()

```
def func():
    pass

class Test:
    pass
print(type(5))
print(type('a'))
print(type(5.5))
print(type(True))
print(type([1, 2]))
print(type((5, 4)))
print(type({1, 2}))
print(type({'a': 2}))
print(type(Test))
print(type(func))
```

Результат

```
<class 'int'>
<class 'str'>
<class 'float'>
<class 'bool'>
<class 'list'>
<class 'tuple'>
<class 'set'>
<class 'dict'>
<class 'type'>
<class 'function'
```

Как мы видит из вывода любой тип данных является объектом какого то класса, даже функция и сам класс. Кстати в данном коде впервые использован оператор pass. Данный оператор является заглушкой. Он используется в циклах, функциях, классах, условиях. Pass указывает, что например есть функция, но она ничего не делает.

Плюсы ООП в том, что декомпозиция позволяет:

- упростить архитектуру приложения.
- облегчить командную разработку (каждый разработчик занят работой над отдельным объектом кода и при взаимодействии с другими объектами ему нужно знать лишь «что объект делает», без необходимости узнавать «как он это делает»).

Разберем понятия объект(экземпляр) и класс.
Класс можно представить, как некий шаблон описывающий общие свойства и возможные действия. Например, в любой игре огромное количество персонажей и писать под каждого из них отдельный код, было бы очень накладно и ресурсоемко. Все персонажи имеют какие то общие характеристики(переменные), такие как например жизни, имя и т.п, а также набор каких то методов, например ходить, говорить, стрелять.

Метод - та же функция, но написанная внутри класса.

Объект же, если брать в пример игру, то это собранный на основе класса персонаж.

Набор полей и методов определяет интерфейс класса - способ взаимодействия с классом произвольного кода программы. Доступ к полям и методам осуществляется через указание объекта, например, в Python, используя точку.

ООП включает 3 основных принципа (свойства):

1. Инкапсуляция - как язык скрывает детали внутренней реализации объектов и предохраняет целостность данных?
2. Наследование - как язык стимулирует многократное использование кода?
3. Полиморфизм - как язык позволяет трактовать связанные объекты сходным образом?

Эти принципы мы затронем в следующих занятиях.

Для создания класса используется ключевое слово class. Давайте создадим простой класс некого персонажа.

```
class Person:
    name = 'Иван'
    age = 'Иванов'

    def say(self):
        print('Hello')
```

Учитель: Как видим создавать класс не сложнее, чем например функцию. В данном случае у нас есть класс, в котором описаны свойства нашего будущего объекта(имя, возраст), а также его действия(говорить привет)

Создадим экземпляр класса и попробуем выведем его имя и вызовем метод say()

```
class Person:
```

```
    name = 'Иван'
    age = 'Иванов'

    def say(self):
        print('Hello')

person1 = Person()
print(person1.name)
person1.say()
```

Чтобы создать экземпляр класса необходима всего одна строка. Если бы мы создавали наших персонажей без использования классов, то это заняло бы на порядок больше строк, а если бы потребовалось изменить какую либо общую характеристику, то пришлось бы менять у каждого персонажа по отдельности.

Учитель: Из предыдущего кода вытекает одна проблема: у нас заранее указаны значения для переменных. Получается у нас всегда будет одни и те же значения? Мы можем напрямую изменить значение для например, второго персонажа

```
person2 = Person()
person2.name = 'Василий'
print(person2.name)
```

Но все же хотелось бы передавать необходимые данные при создании объекта. Для решения данной проблемы нам пора познакомиться с магическим встроенным методом, который вызывается автоматически при создании объекта.

При создании экземпляра класса, как правило, требуется проводить его инициализацию (например, устанавливать начальные значения полей), для чего в Python предназначен специальный метод __init__

```
class Person:
  # Инициализирующий метод (специальный метод с __)
  def __init__(self):
    print('Метод init')


person1 = Person()
```

В данный метод мы можем передать необходимые значения и сохранить их в локальные переменные, значения которых будут разные для разных объектов.

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


person1 = Person('Иван', 15)
person2 = Person('Петр', 14)
print(person1.name)
print(person1.age)
print(person2.name)
print(person2.age)
```

Теперь мы видим, что у каждого объекта свое имя и свой возраст. Помочь программе, понять какое значение принадлежит объекту, позволяет self, которые ссылается именно на объект, который обратился к данному свойству.

Еще одним часто используемым специальным методом является специальный метод __str__, который возвращает строковое представление класса

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

  def __str__(self):
     return f'Меня зовут {self.name}'

person1 = Person('Иван')
print(person1)
```

2. Решение задач

Задача 1

Написать класс ученика класса. Описать в классе все возможные свойства, которые можно обобщить для каждого ученика, а также создать несколько методов.

Решение

```
class Person:
    def __init__(self, name,  last_name, age, num_class, level):
        self.name = name
        self.last_name = last_name
        self.age = age
        self.num_class = num_class
        self.level = level

    def do_homework(self):
```

```
        print('Делаю уроки')

    def answer_in_class(self):
        print('Отвечаю на уроке')


person1 = Person('Иван', 'Иванов', 15, '7A', 'Excellent student')
```

Дополнительно

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

Домашняя работа

Задача 1