ТЕХНОЛОГИЧЕСКАЯ КАРТА ЗАНЯТИЯ

Тема занятия: Основы объектно-ориентированного программирования на Python

Аннотация к занятию: на данном занятии обучающиеся познакомятся с основами объектно-ориентированного программирования на Python и областью их применения.

Цель занятия: формирование у обучающихся представления об основах объектно-ориентированного программирования на Python. Знакомство с тремя принципами ООП: наследованием, полиморфизмом, инкапсуляцией.

Задачи занятия:

- познакомить обучающихся с основами объектно-ориентированного программирования на Python;
- сформировать понятия наследования, полиморфизма, инкапсуляции.

Ход занятия

Этап занятия	Время	Деятельность педагога	Комментарии, рекомендации для педагогов
Организационный этап	2 мин.	Добрый день! Добро пожаловать на урок!	Приветствие. Создание в классе атмосферы психологическог о комфорта
Постановка цели и задач занятия. Мотивация учебной деятельности обучающихся	10 мин.	В самом начале обучения Python наши программы ограничивались простым калькулятором и выводом Hello, world! в консоль. Они занимали мало строк, и отследить логику алгоритма было просто. Но прочесть программу из 50 или 200 строк кода (особенно, когда в нём много переменных!) уже сложнее, а ведь в настоящих масштабных продуктах десятки тысяч строк! **Вопрос для обсуждения** Что же делать, когда код начинает расти в объёме? **Ответы обучающихся** Сегодня мы познакомимся с объектно-ориентированным подходом — важным инструментом любого программиста. Цель объектного подхода — представить программу как взаимодействие набора объектов. Его часто пользуют, когда код	Способствовать обсуждению мотивационных вопросов

<table>
<tr><td></td><td></td><td>начинает расти в объёме: это повышает его читаемость и актуальность.</td><td></td></tr>
<tr><td>Изучение нового материала</td><td>55 мин.</td><td>Рассмотрим принципы ООП на примере умного дома. Представьте себе квартиру. В прихожей находится умный пылесос, на кухне — умные розетки и чайник, в гостиной — умные лампочки, голосовой помощник в колонке.

<table><tr><td></td><td>Пылесос</td><td>Колонка</td><td>Лампочка</td></tr><tr><td>Действие</td><td>Начало уборки
Конец уборки</td><td>Какая погода
Закажи пиццу</td><td>Включить свет
Выключить свет</td></tr></table>
Каждое из устройств уникально и выполняет разные действия, но все они объединены в одну сеть. Умная колонка может включить радио или поставить будильник, пылесос — начать уборку, а лампочка — светиться любым цветом.

Все они могут работать сообща. Но на этапе создания умных устройств никто не мог предположить, какие девайсы окажутся друг с другом в одной сети.
По этой причине разработчики заложили в них всё необходимое для самостоятельной работы и независимости от внешних условий. Такая скрупулёзность объяснима: стоимость «умных» гаджетов значительно превышает «глупый» аналог.

При проектировании объекта в Python стоит использовать схожий принцип: если мы хотим создать объект «пылесос», необходимо встроить в него все функции, которые обеспечат самостоятельную работу устройства. Для его запуска нам останется лишь сказать:</td><td>Для справки:
Сайт:
https://colab.research.google.com/drive/1A6VuFvCPNCGv3_Fho-xhgYcFYgrxEzNk?usp=sharing

Перед уроком рекомендуется ознакомиться с материалами, представленным и на сайте.</td></tr>
</table>

Пылесос.начни_уборку()

Согласитесь, это похоже на просьбу пылесоса запустить собственную функцию, ведь в русском языке запись выглядит так:
Пылесос, начни уборку!
Заметьте: в конце обращения появились скобки — это место для записи аргументов внутри функции, а вместо запятой из прямой речи в коде появилась точка. Что она означает?
Точку используют, чтобы обозначить «поиск внутри себя». С помощью этой команды мы обращаемся к пылесосу с просьбой найти и включить функцию уборки.

Если мы попросим пылесос включить музыку, программа выдаст ошибку. К сожалению, пылесосы не умеют петь, и такой функции у них нет.

Объекты в Python
Как создать новый объект в Python? Синтаксис объекта напоминает функцию: мы по очереди указываем ключевое слово class, название и двоеточие.
Создадим простой объект лампочки с двумя параметрами:

- brightness — уровень яркости (от 0 до 10),
- status — состояние (включена/выключена).
-

```
class Lamp:
    status = False
    brightness = 5
```

Вопрос для обсуждения
Подождите, но ведь мы говорили про объектное программирование. Причём здесь классы?

Ответы школьников
Чтобы лучше это понять, проведём аналогию с производством. Представьте, что у нас есть целая конвейерная линия умных пылесосов. Они производятся автоматически, по определённому чертежу. Класс — это такой чертёж, который описывает объект и процесс его создания, а объект — его материальное воплощение, готовый пылесос.

Как чертёж не является готовым устройством, так и сам класс не принимает участие в программе, но на его основе можно напечатать сколько угодно объектов. Класс — это тип данных, который определён в языке, а объект — переменная этого класса.

«Напечатаем» свой первый объект. Так как объект — это переменная, ему необходимо присвоить имя:

```
Lamp_hallway = Lamp
```

Рассмотрим параметры объекта. Чтобы к ним обратиться, поставим точку после имени объекта и укажем название параметра:

```
Lamp_hallway = Lamp

print("Лампочка имеет статус ", Lamp_hallway.status)
print("Ее яркость ", Lamp_hallway.brightness)
```

Запрашивать статус переменных вручную неудобно. Автоматизируем процесс: создадим функцию внутри объекта, куда перенесём две предыдущие строки кода. Когда нам нужно будет узнать состояние переменных, будет достаточно вызвать одну функцию. Обратите внимание: когда класс работает со своими собственными переменными, перед ними ставится self.

```
class Lamp:
    ...
    status = False
    brightness = 5

    def info1(self):
        print("Лампочка имеет статус ", self.status)
        print("Ее яркость ", self.brightness)



Lamp_hallway = Lamp()
Lamp_hallway.info1()
```

Self используют для представления объекта класса. Благодаря ему объект может получить доступ к атрибутам и методам своего класса.
Получается, когда мы пишем подобную строчку:

```
Lamp_hallway.info()
```

Для объекта она выглядит так:

```
Lamp.info(Lamp_hallway)
```

Функция написана внутри класса и справляется со своей задачей. Вот только если попытаться вызвать её не при помощи объекта, а напрямую, программа выдаст ошибку:

```
info()
```

```
123 ×
C:\Users\tsarc\PycharmProjects\Ses\venv\Scripts\python.exe C:/Users/tsarc/Pychar
Traceback (most recent call last):
  File "C:/Users/tsarc/PycharmProjects/Ses/123.py", line 59, in <module>
    info()
NameError: name 'info' is not defined

Process finished with exit code 1
```

По определению функции, она должна быть видима в любой области программы. Но если записать её внутри класса, то вызвать её сможет только копия этого класса, то есть объект.
Чтобы не возникало путаницы, все функции, помещённые внутрь объектов, называют методами. Посмотрим на их отличия.

Функция:

- фрагмент кода, к которому можно обратиться из другого места программы;
- может существовать сама по себе в программе;
- не всегда принимает или возвращает значения;
- функция (параметр1, параметр2).

		Метод: ● всегда является функцией; ● не может существовать вне класса или объекта; ● не всегда принимает или возвращает значения; ● объект.метод(параметр1, параметр2). Эти отличия можно трактовать иначе: если в программе есть метод, значит где-то неподалёку бродит его объект. Переименование не обошло и переменные. Смысл переменных — хранить значение так, чтобы его можно было посмотреть в любой момент. При этом переменная внутри объекта принадлежит только ему. Переменные внутри объекта называют параметром или полем класса. Повторю: несмотря на то, что мы обращаемся к полям с одинаковым названием, Python понимает, что у каждого объекта значение этого поля уникально. Следовательно, два объекта одного класса могут сосуществовать независимо друг от друга. Рассмотрим пример более сложного класса. Представьте, что одна из лампочек перегорела, и мы пошли искать замену. В магазине обнаружилось, что у лампочки намного больше параметров, чем мы думали. Модифицируем наш класс и добавим к нему новые параметры: ● вольтаж, ● название производителя, ● мощность в ваттах, ● тип цоколя, ● тип лампочки.	

```python
class Lamp:
    ...
    manufacturer = "Неизветсно"
    voltage = 0
    type = "Неизветсно"
    watts = 0
    type_lampshade = ""

    status = False
    brightness = 5

    def info(self):
        print("Лампочка имеет статус ", self.status)
        print("Ее яркость ", self.brightness)
```

Теперь мы хотим создать пару уникальных объектов — лампочек разных производителей.

```python
Lamp1 = Lamp()
Lamp2 = Lamp()

Lamp1.manufacturer = "Китай"
Lamp1.voltage = 5
Lamp1.type = "Светодиодная"
Lamp1.watts = 10
Lamp1.type_lampshade = "e14"

Lamp2.manufacturer = "Россия"
Lamp2.voltage = 220
Lamp2.type = "Накаливаиня"
Lamp2.watts = 60
Lamp2.type_lampshade = "e27"
```

Записывать множество параметров вручную бывает утомительно, поэтому мы автоматизируем ввод.

Для автоматизации необходимо воспользоваться конструктором класса __init__(). Конструктор — это тоже метод, но необычный. Наличие знаков подчёркивания в имени метода говорит о том, что он принадлежит к группе методов перегрузки.

Их не нужно вызывать: вызовом для таких методов является определённая операция над объектом. В случае конструктора класса — это процесс создания нового объекта. Детально методы перегрузки мы рассмотрим в завершающем ролике модуля. Сейчас необходимо запомнить, что конструкцию __init__() можно использовать для помещения значений внутрь объекта. Это выглядит так:

```
class Lamp:
    def __init__(self, manufacturer, voltage, type, watts, type_lampshade):
        self.manufacturer = manufacturer
        self.voltage = voltage
        self.type = type
        self.watts = watts
        self.type_lampshade = type_lampshade

    status = False
    brightness = 5
```

Конструктор начинается с def__init__. Первым параметром выступает значение self — оно передаёт ссылку на экземпляр самого класса и создаёт уникальную переменную внутри объекта.

Раз мы используем конструктор в классе, то при создании нового объекта стоит перечислить все параметры. Это поможет избежать ошибки:

```
18  Lamp1 = Lamp()
19  Lamp2 = Lamp()
20
main ×
  File "C:\Users\d.tsarkov\PycharmProjects\Kekv\main.py", line 18, in <module>
    Lamp1 = Lamp()
TypeError: __init__() missing 5 required positional arguments: 'manufacturer', 'voltage', 'type', 'watts', and 'type_lampshade'
```

Посмотрим, как изменилась запись новых объектов.

```
18  Lamp1 = Lamp("Китай", 5, "Светодиодная", 10, "e14")
19  Lamp2 = Lamp("Россия", 220, "Накаливаиня", 60, "e27")
```

Поля, созданные при помощи конструктора, становятся динамическими, а значит, получить к ним доступ можно только после создания объекта. А вот поля яркости и статуса остались вне конструктора: они статические.

Объекты позволяют сгруппировать переменные и функции внутри одной сущности, что упрощает понимание программы. Однако не это сделало ООП таким популярным. Его смысл раскрывается в трёх принципах, которые должны выполняться всегда, вне зависимости от языков программирования и среды. Совсем как законы физики!

Наследование
Первый принцип — наследование. Это механизм, который позволяет копировать свойства и поведение других классов.

Для справки:
Сайт:
https://colab.research.google.com/drive/1OzwncrLx0HFh_p9pAR09XWXgf5Bcp0rP?usp=sharing

Благодаря ему можно расширить или модифицировать функционал существующих объектов.

Класс, передающий свои свойства, называется родителем, а принимающий — дочерним. Всё как в жизни!
Где это пригодится? Если в программе много классов с одинаковыми условиями, их можно заменить одним родителем. Например, когда мы хотим перенести устройства из нашей квартиры в программу, родительским классом станет электрическое устройство:

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

    status = False

    def On(self):
        self.status = True
        print("Устройство", self.name, "включено")

    def Off(self):
        self.status = False
        print("Устройство", self.name, "выключено")
```

В родительском классе мы опишем базу для будущих устройств: реакцию на включение, выключение и переменные, которые точно есть у любого предмета.

Наследственная связь между классами устанавливается через подкласс. При определении дочернего класса после его имени в скобках указывается родительский:

```
class Lamp(Electrical_device):
    brightness = 0

    def info(self):
        print("Лампочка имеет статус ", self.status)
        print("Ее яркость ", self.brightness)


class Cleaner(Electrical_device):
    type_of_cleaning = "Средняя"

    def cleaning(self):
        if self.type_of_cleaning == "Средняя":
            print("вжвжвжвжвжв")
        if self.type_of_cleaning == "Сильная":
            print("ВЖВЖВЖВЖЖВВЖ")
```

Класс лампочки и пылесоса унаследовали свойства своего родителя (конструктора __init()__, функции включения) и приобрели свой функционал.

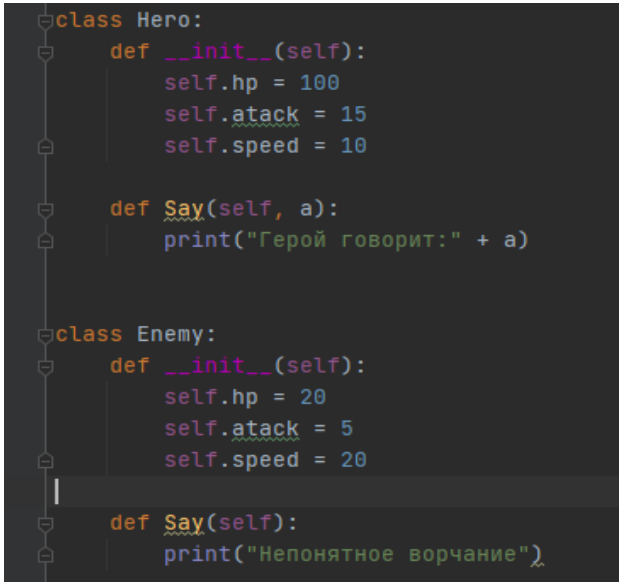
```
Cleaner1 = Cleaner(220, "Робот пылесос")
Lamp1 = Lamp(220, "умная лампочка")

Lamp1.on()
Lamp1.info()

Cleaner1.type_of_cleaning = "Сильная"
Cleaner1.cleaning()
```

```
Устройство умная лампочка включено

Лампочка имеет статус  True
Ее яркость  0

ВЖВЖВЖВЖЖВВЖ
```

Важно отметить, что наследование работает в одну сторону, и родитель не может обладать функционалом своих дочерних классов.

Полиморфизм
Дословный перевод термина «полиморфизм» с греческого — много форм. Изначальная идея полиморфизма — описывать вычисления в общем виде, чтобы их можно было использовать с любыми типами данных. Функция print() — отличный пример полиморфизма, т.к. ей не важен тип входных данных.
Динамическая типизация в Python тоже хорошо описывает полиморфизм: одна и та же переменная в программе может

		хранить в себе данные любых типов.
		`a = "5"`
		`print(a)`
		`a = 123 + 456`
		`print(a)`
		`a = [34, "a", ["erewr"], 845]`
		`print(a)`
		
		dd ×
		C:\Users\d.tsarkov\PycharmProjects\Ke
		5
		579
		[34, 'a', ['erewr'], 845]
		Существует несколько проявлений полиморфизма.
		Первое. Если в разных классах присутствуют схожие по названию переменные и методы, компилятор не выдаст ошибку. Это объяснимо: у двух классов могут быть схожие по названию, но разные по коду методы. Результаты работы таких одноимённых методов могут существенно различаться. Это позволяет не беспокоиться в случае, если у двух классов совпадают названия методов.	

		 Второе свойство основано на переопределении атрибутов, унаследованных от прошлых классов. Переопределение возникает, когда название метода или свойства дочернего объекта совпадает с родительским. При этом приоритет отдаётся дочернему. Это пригодится, когда унаследованный метод не подходит под новые требования класса:	

```python
class Hero:
    def __init__(self):
        self.hp = 100
        self.atack = 15
        self.speed = 10

    def Say(self, a):
        print("Герой говорит:" + a)


class Enemy:
    def __init__(self):
        self.hp = 20
        self.atack = 5
        self.speed = 20

    def Say(self):
        print("Непонятное ворчание")
```

```python
class Enemy:
    def __init__(self,hp,atack,speed):
        self.hp = hp
        self.atack = atack
        self.speed = speed

    def Say(self):
        print("Непонятное ворчание")

class Epick_Enemy(Enemy):
    def Fly(self):
        self.speed += 30

    def Say(self):
        print("Попробуй догони")

Small_Dragon = Epick_Enemy(20,30,20)
Small_Dragon.Say()
```

Инкапсуляция

Инкапсуляция — один из немногих способов скрыть свойства объектов от остального кода. Как правило, в объектно-ориентированном программировании один класс не должен иметь прямого доступа к данным другого класса: обмен информацией происходит через методы класса. В таких языках, как Java или C, инкапсуляции уделено намного больше внимания, а в Python от этого остался небольшой рудимент.

В Python существует 3 варианта доступа к данным:

- свободный режим доступа (public) (стоит по умолчанию у всех переменных и объектов в программе);
- _ — режим доступа protected (позволяет обращаться только к объектам внутри класса и во всех его дочерних классах);
- __ — режим доступа private (позволяет обращаться только к объектам внутри класса).

```
public = "я обычная перменная"
_protected = "я зашишенная переменная"
__private = "я приватная перменная"
```

Уровень доступа указывают перед переменной или функцией внутри класса.

Если мы подключим приватный метод turn_on, который будет считать, сколько раз включалось устройство, вызвать его из основной программы не получится. Компилятор сделает вид, что такого метода не существует. Таким образом можно быть уверенным, что никто не подкрутит счётчик включений и он будет показывать правильное число.

```
class Lamp(Electrical_device):
    brightness = 0
    __how_many_times_turned_on = 0

    def __turn_on(self):
        self.__how_many_times_turned_on += 1
        print("Times was turned on: ", self.__how_many_times_turned_on)

    def info(self):
        print("Лампочка имеет статус ", self.status)
        print("Ее яркость ", self.brightness)
```

```
Lamp1.turn_on()
Lamp1.__turn_on()
```

		Попытка получить приватные данные или запустить приватный метод приведёт к ошибке:
		`Traceback (most recent call last):`
		`  File "C:\Users\d.tsarkov\PycharmProjects\Kekv\main.py", line 34, in <module>`
		`  Lamp1.__turn_on()`
		`AttributeError: 'Lamp' object has no attribute '__turn_on'`
		Мы увидели некоторые формы полиморфизма у объектов. Для этих сущностей Python позволяет переопределять такие операторы, как сложение, деление, преобразование типа данных, добавление объекта в массив и прочие.	

Закрепление изученного материала	10 мин.	**Вопросы для обсуждения** ● Что такое объектно-ориентированное программирование? ● Назовите три основные принципа в ООП?	Педагог организует беседу по вопросам
Этап подведения итогов занятия (рефлексия)	8 мин.	**Вопросы для обсуждения** ● Чему я научился? ● С какими трудностями я столкнулся? ● Каких знаний мне не хватает для более глубокого понимания изученного материала? ● Достиг ли я поставленных целей и задач?	Педагог способствует размышлению обучающихся над вопросами
Информация о домашнем задании, инструктаж по его применению	5 мин.	В этом домашнем задании вам предстоит попрактиковаться в реализации собственных классов. В каждой задаче вам будет необходимо реализовать класс с тем или иным функционалом, то есть набором методов с определенными названиями.	Педагог организует беседу по вопросам

		Программы проверяются в автоматическом режиме на платформе Stepik или платформе Академии искусственного интеллекта. Здесь вам нужно будет вставлять реализацию классов прямо на платформу. Будет проверяться корректная работа всего функционала классов.	

Рекомендуемые ресурсы для дополнительного изучения:

1. Объектно-ориентированное программирование [Электронный ресурс] – Режим доступа: https://colab.research.google.com/drive/1A6VuFvCPNCGv3_Fho-xhgYcFYgrxEzNk?usp=sharing.
2. Поля классов. [Электронный ресурс] – Режим доступа: https://colab.research.google.com/drive/18Qc7cGGvy28T5NSDCaACCVsMm7Fprm_-?usp=sharing.
3. Три столпа ООП. [Электронный ресурс] – Режим доступа: https://colab.research.google.com/drive/1OzwncrLx0HFh_p9pAR09XWXgf5Bcp0rP?usp=sharing.
4. Перегрузка операторов. [Электронный ресурс] – Режим доступа: https://colab.research.google.com/drive/1S6EDzk6q_zloo2CufzSFF5FdbS-YPmKl?usp=sharing.