**Тема занятия №17: Объектно-ориентированное программирование**

**1. Введение**

Python — мультипарадигмальный язык программирования. Он поддерживает разные подходы к программированию.

Один из популярных подходов к решению проблем — создание объектов. Это называется объектно-ориентированным программированием (ООП).

Объектно-ориентированное программирование (ООП) - парадигма программирования, в которой основными концепциями являются понятия объектов и классов.

Класс — тип, описывающий устройство объектов. Объект - это экземпляр класса. Класс можно сравнить с чертежом, по которому создаются объекты.

Python соответствует принципам объектно-ориентированного программирования. В python всё является объектами - и строки, и списки, и словари, и всё остальное.

Но возможности ООП в python этим не ограничены. Программист может написать свой тип данных (класс), определить в нём свои методы.

Это не является обязательным - мы можем пользоваться только встроенными объектами. Однако ООП полезно при долгосрочной разработке программы несколькими людьми, так как упрощает понимание кода.

Объектно-ориентированная парадигма имеет несколько принципов:

Данные структурируются в виде объектов, каждый из которых имеет определенный тип, то есть принадлежит к какому-либо классу.

Классы – результат формализации решаемой задачи, выделения главных ее аспектов.

Внутри объекта инкапсулируется логика работы с относящейся к нему информацией.

Объекты в программе взаимодействуют друг с другом, обмениваются запросами и ответами.

При этом объекты одного типа сходным образом отвечают на одни и те же запросы.

Объекты могут организовываться в более сложные структуры, например, включать другие объекты или наследовать от одного или нескольких объектов.

**2. Основы ООП на Python**

У объекта есть две характеристики:

* атрибуты;
* поведение.

Рассмотрим пример. Допустим, наш объект — это попугай. У попугая есть такие свойства:

Имя, возраст, цвет. Это атрибуты.

То, как попугай поет и танцует. Это поведение.

ООП предлагает писать код, который можно использовать повторно. Такой принцип называется DRY (don’t repeat yourself, «не повторяйся»).

**Класс**

Класс — это шаблон объекта.

Экземпляры классов

Инстанцировать класс в Python тоже очень просто:

**class SomeClass(object):**

**attr1 = 42**

**def method1(self, x):**

**return 2\*x**

**obj = SomeClass()**

**obj.method1(6) # 12**

**obj.attr1 # 42**

Можно создавать разные инстансы одного класса с заранее заданными параметрами с помощью инициализатора (специальный метод \_\_init\_\_). Для примера возьмем класс Point (точка пространства), объекты которого должны иметь определенные координаты:

**class Point(object):**

**def \_\_init\_\_(self, x, y, z):**

**self.coord = (x, y, z)**

**p = Point(13, 14, 15)**

**p.coord # (13, 14, 15)**

**Динамическое изменение**

Можно обойтись даже без определения атрибутов и методов:

**class SomeClass(object):**

**pass**

Кажется, этот класс совершенно бесполезен? Отнюдь. Классы в Python могут динамически изменяться после определения:

**class SomeClass(object):**

**pass**

**def squareMethod(self, x):**

**return x\*x**

**SomeClass.square = squareMethod**

**obj = SomeClass()**

**obj.square(5) # 25**

Вернемся к нашему попугаю. Если мы схематично нарисуем его на бумаге, такой набросок будет являться классом. По нему можно сделать, например, чучело попугая.

Давайте создадим класс, который описывает попугая:

**class Parrot:**

**pass**

Для объявления класса Parrot мы использовали ключевое слово class. Из классов мы получаем экземпляры, созданные по подобию этого класса.

**Объект**

Объект — это экземпляр класса. Объявленный класс — это лишь описание объекта: ему не выделяется память.

Например, экземпляра класса Parrot будет выглядеть так:

# obj — экземпляр класса Parrot

**obj = Parrot()**

Теперь разберемся, как написать класс и его объекты.

# Создаем класс и его объекты

**class Parrot:**

**# атрибуты класса**

**species = "птица"**

**# атрибуты экземпляра**

**def \_\_init\_\_(self, name, age):**

**self.name = name**

**self.age = age**

**# создаем экземпляра класса**

**kesha = Parrot("Кеша", 10)**

**cookie = Parrot("Куки", 15)**

**# получаем доступ к атрибутам класса**

**print("Кеша — {}".format(kesha.\_\_class\_\_.species))**

**print("Куки тоже {}".format(cookie.\_\_class\_\_.species))**

**# получаем доступ к атрибутам экземпляра**

**print("{} — {}-летний попугай".format(kesha.name, kesha.age))**

**print("{} — {} летний попугай".format(cookie.name, cookie.age))**

**Вывод:**

**Кеша — птица**

**Куки тоже птица**

**Кеша — 10-летний попугай**

**Куки — 15-летний попугай**

Мы создали класс Parrot. После этого мы объявили атрибуты — характеристики объекта.

Атрибуты объявлены внутри класса — в методе \_\_init\_\_. Это метод-инициализатор, который запускается сразу же после создания объекта.

После этого мы создаем экземпляры класса Parrot. kesha и cookie — ссылки на (значения) наши новые объекты.

Получить доступ к атрибуту класса можно так — \_\_class\_\_.species. Атрибуты класса для всех экземпляров класса одинаковы. Точно так же мы можем получить доступ к атрибутам экземпляра — kesha.name и kesha.age. Но вот атрибуты каждого экземпляра класса уникальны.

**Методы**

Методы — функции, объявленные внутри тела класса. Они определяют поведения объекта.

# Создаем метод

**class Parrot:**

**# атрибуты экземпляра**

**def \_\_init\_\_(self, name, age):**

**self.name = name**

**self.age = age**

**# метод экземпляра**

**def sing(self, song):**

**return "{} поет {}".format(self.name, song)**

**def dance(self):**

**return "{} танцует".format(self.name)**

**# создаем экземпляр класса**

**kesha = Parrot("Кеша", 10)**

**# вызываем методы экземпляра**

**print(kesha.sing("песенки"))**

**print(kesha.dance())**

**Вывод:**

**Кеша поет песенки**

**Кеша танцует**

В этой программе мы объявили два метода: sing() и dance(). Они являются методами экземпляра, потому что они вызываются объектами — например, kesha.

**ООП. Инкапсуляция**

**Введение**

Инкапсуляция — это один из столпов объектно-ориентированного программирования. Инкапсуляция просто означает скрытие данных. Как правило, в объектно-ориентированном программировании один класс не должен иметь прямого доступа к данным другого класса. Вместо этого, доступ должен контролироваться через методы класса.

Все объекты в Python инкапсулируют внутри себя данные и методы работы с ними, предоставляя публичные интерфейсы для взаимодействия.

Атрибут может быть объявлен приватным (внутренним) с помощью нижнего подчеркивания перед именем, но настоящего скрытия на самом деле не происходит – все на уровне соглашений.

**class SomeClass:**

**def \_private(self):**

**print("Это внутренний метод объекта")**

**obj = SomeClass()**

**obj.\_private() # это внутренний метод объекта**

Если поставить перед именем атрибута два подчеркивания, к нему нельзя будет обратиться напрямую. Но все равно остается обходной путь:

**class SomeClass():**

**def \_\_init\_\_(self):**

**self.\_\_param = 42 # защищенный атрибут**

**obj = SomeClass()**

**obj.\_\_param # AttributeError: 'SomeClass' object has no attribute '\_\_param'**

**obj.\_SomeClass\_\_param # 42**

Специальные свойства и методы класса, некоторые из которых вам уже знакомы, имеют двойные подчеркивания до и после имени.

Кроме прямого доступа к атрибутам (obj.attrName), могут быть использованы специальные методы доступа (геттеры, сеттеры и деструкторы):

**class SomeClass():**

**def \_\_init\_\_(self, value):**

**self.\_value = value**

**def getvalue(self): # получение значения атрибута**

**return self.\_value**

**def setvalue(self, value): # установка значения атрибута**

**self.\_value = value**

**def delvalue(self): # удаление атрибута**

**del self.\_value**

**value = property(getvalue, setvalue, delvalue, "Свойство value")**

**obj = SomeClass(42)**

**print(obj.value)**

**obj.value = 43**

Такой подход очень удобен, если получение или установка значения атрибута требует сложной логики.

Вместо того чтобы вручную создавать геттеры и сеттеры для каждого атрибута, можно перегрузить встроенные методы \_\_getattr\_\_, \_\_setattr\_\_ и \_\_delattr\_\_. Например, так можно перехватить обращение к свойствам и методам, которых в объекте не существует:

**class SomeClass():**

**attr1 = 42**

**def \_\_getattr\_\_(self, attr):**

**return attr.upper()**

**obj = SomeClass()**

**obj.attr1 # 42 &nbsp;&nbsp;**

**obj.attr2 # ATTR2**

\_\_getattribute\_\_ перехватывает все обращения (в том числе и к существующим атрибутам):

**class SomeClass():**

**attr1 = 42**

**def \_\_getattribute\_\_(self, attr):**

**return attr.upper()**

**obj = SomeClass()**

**obj.attr1 # ATTR1**

**obj.attr2 # ATTR2**

Чтобы предоставить контролируемый доступ к данным класса в Python, используются модификаторы доступа и свойства. Мы посмотрим, как действуют свойства.

Предположим, что нам нужно убедиться в том, что модель автомобиля должна датироваться между 2000 и 2018 годом. Если пользователь пытается ввести значение меньше 2000 для модели автомобиля, значение автоматически установится как 2000, и если было введено значение выше 2018, оно должно установиться на 2018. Если значение находится между 2000 и 2018 — оно остается неизменным. Мы можем создать свойство атрибута модели, которое реализует эту логику. Взглянем на пример:

**# создаем класс Car**

**class Car:**

**# создаем конструктор класса Car**

**def \_\_init\_\_(self, model):**

**# Инициализация свойств.**

**self.model = model**

**# создаем свойство модели.**

**@property**

**def model(self):**

**return self.\_\_model**

**# Сеттер для создания свойств.**

**@model.setter**

**def model(self, model):**

**if model < 2000:**

**self.\_\_model = 2000**

**elif model > 2018:**

**self.\_\_model = 2018**

**else:**

**self.\_\_model = model**

**def getCarModel(self):**

**return "Год выпуска модели " + str(self.model)**

**carA = Car(2088)**

**print(carA.getCarModel())**

Свойство имеет три части. Вам нужно определить атрибут, который является моделью в скрипте выше. Затем, вам нужно определить свойство атрибута, используя декоратор @property. Наконец, вам нужно создать установщик свойства, который является дескриптором @model.setter в примере выше.

Теперь, если вы попробуете ввести значение выше 2018 в атрибуте модели, вы увидите, что значение установлено на 2018. Давайте проверим это. Выполним следующий скрипт:

**car\_a = Car(2088)**

**print(car\_a.get\_car\_model())**

Здесь мы передаем 2088 как значение для модели, однако, если вы введете значение для атрибута модели через функцию get\_car\_model(), вы увидите 2018 в выдаче.