Python

Классы

Для чего используются классы?

- Создание сложных структур данных.
- Наследование: одни классы могут “наследовать” общие свойства базового класса. Такие свойства могут быть реализованы один раз и затем многократно применяться частично или полностью всеми потомками.
- Композиция: один объект, может содержать другие объекты и использовать их для выполнения соответствующих вычислений. Каждый компонент системы может быть реализован в виде класса, который определяет собственное поведение и взаимосвязи.

Классы

- Класс - способ описания сущности, определяющий состояние и поведение, зависящее от этого состояния.

- Классы - служат “фабриками” экземпляров. Атрибуты классов обеспечивают поведение всех экземпляров, сгенерированных из них.

Состояния классов:

- Кот: кличка, цвет, сытость, усталость.
- Автомобиль: мощность, цвет, тип топлива, расход топлива, максимальный объем топлива, текущий запас топлива, пробег.

Классы

Поведение класса:

- Кот: поесть, поиграть, поспать.
- Автомобиль: проехать X километров, заправить топливо.

Объекты

- Объект (экземпляр) – это отдельный представитель класса, имеющий конкретное состояние и поведение, полностью определяемое классом.

Примеры:

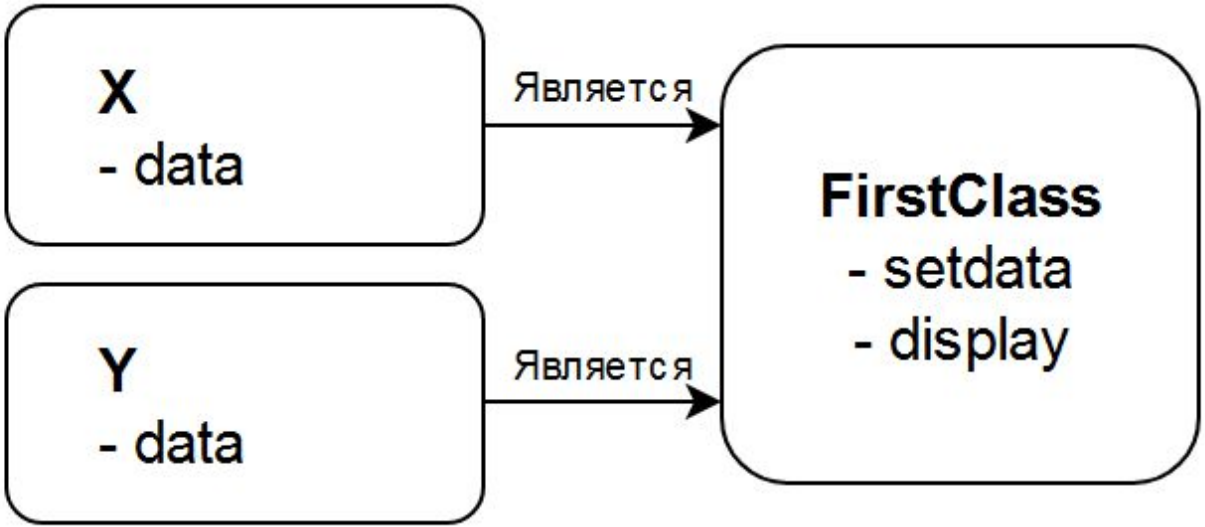
- Коты Рыжик и Барсик будут объектами (экземплярами) класса Кот.
- У кота Рыжик свои значения атрибутов кличка, цвет, сытость и усталость, у Барсик свои.
- Оба могут поесть, поиграть, поспать, так как эти методы определены в классе Кот.

Классы - простой пример

```
class FirstClass:  # Определяем объект класса
    def setdata(self, value): # Определяем метод класса
        self.data = value # self - это экземпляр
    def display(self):
        print(self.data)

# Создаем 2 экземпляра класса FirstClass
x = FirstClass()
y = FirstClass()
```

Классы - простой пример



Классы - простой пример

```
x.setdata("King Arthur") # Вызываем метод: self - это x
y.setdata(3.14159) # Выполняется FirstClass.setdata (y, 3,14159)

x.display() # self.data отличается в каждом экземпляре
y.display() # Выполняется FirstClass.display(y)

x.data = 'New value' # Можно получать/устанавливать атрибуты
x.display()

x.anothername = 'spam' # Так тоже можно, но обычно не нужно :)
```

Конструктор класса

Пусть у нас есть класс, для которого name и surname являются обязательными.

```
class Person:
    def setName(self, n, s):
        self.name = n
        self.surname = s

p1 = Person()
p1.setName("Bill", "Ross") # Можем забыть вызвать метод
```

Конструктор класса

Конструктор - это специальный метод класса, имеющий название __init__, который вызывается при создании объекта данного класса.

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

p1 = Person("Sam", "Baker")
```

Полный синтаксис

```
class Person:
    template = 'Name: {} Age: {}' # Определяем атрибут класса
    def __init__(self, name, age):
        self.name = name # Определяем атрибут name в объекте
        self.age = age
    def display(self): # Метод экземпляра класса
        print(self.template.format(self.name, self.age))
    @classmethod
    def from_birth_year(cls, name, year): # Метод класса
        return cls(name, date.today().year - year)
    @staticmethod
    def is_adult(age): # Статический метод
        return age > 18
```

Атрибуты объекта и класса

Атрибуты объекта: свои для каждого объекта данного класса.

Атрибуты класса:

- общие для всех экземпляров класса
- доступны через экземпляр и через класс: `Person.template`
- обращать внимание, при использовании изменяемых типов
- любые объявленные в теле класса поля и методы принадлежат классу

Атрибуты объекта и класса

Если задан атрибут класса и объекта с одинаковым именем, в первую очередь используется атрибут объекта.

```
class Warehouse:
    purpose = 'storage'
    region = 'west'
w1 = Warehouse()
print(w1.purpose, w1.region)  # Напечатает: storage west

w2 = Warehouse()
w2.region = 'east'
print(w2.purpose, w2.region)  # Напечатает: storage east
```

Методы экземпляра класса

- Принимают объект класса как первый аргумент (принято называть `self`).
- Используя `self` можем менять состояние объекта и обращаться к другим методам.
- Используя `self.__class__` получаем доступ к атрибутам класса.

Могут менять состояние объекта и состояние класса.

Методы класса

- при определении метода используется декоратор `classmethod`
- принимают класс в качестве первого аргумента (принято называть `cls`)
- `cls` указывает на класс, а не на объект (в нашем примере `Person`)

Могут менять только состояние класса.

Статические методы

- при определении метода используется декоратор `staticmethod`
- не передается `cls` или `self`

Статические методы прикреплены к классу лишь для удобства и не могут менять состояние ни класса, ни его экземпляра.

Примеры использования

```
p1 = Person('John', 24)
p1.display() # Напечатает: Name: John Age: 24

p2 = Person.from_birth_year('Jane', 1990)
p2.is_adult(p2.age) # Статические методы и методы класса можно вызывать через объект

p2.template = '### {}: {} ###'
p1.display() # ???
p2.display() # ???

Person.display() # ???
```

Примеры использования

Методы могут вызывать другие методы.

```
class Bag:
    def __init__(self):
        self.data = []
    def add(self, x):
        self.data.append(x)
    def addtwice(self, x):
        self.add(x)
        self.add(x)
```

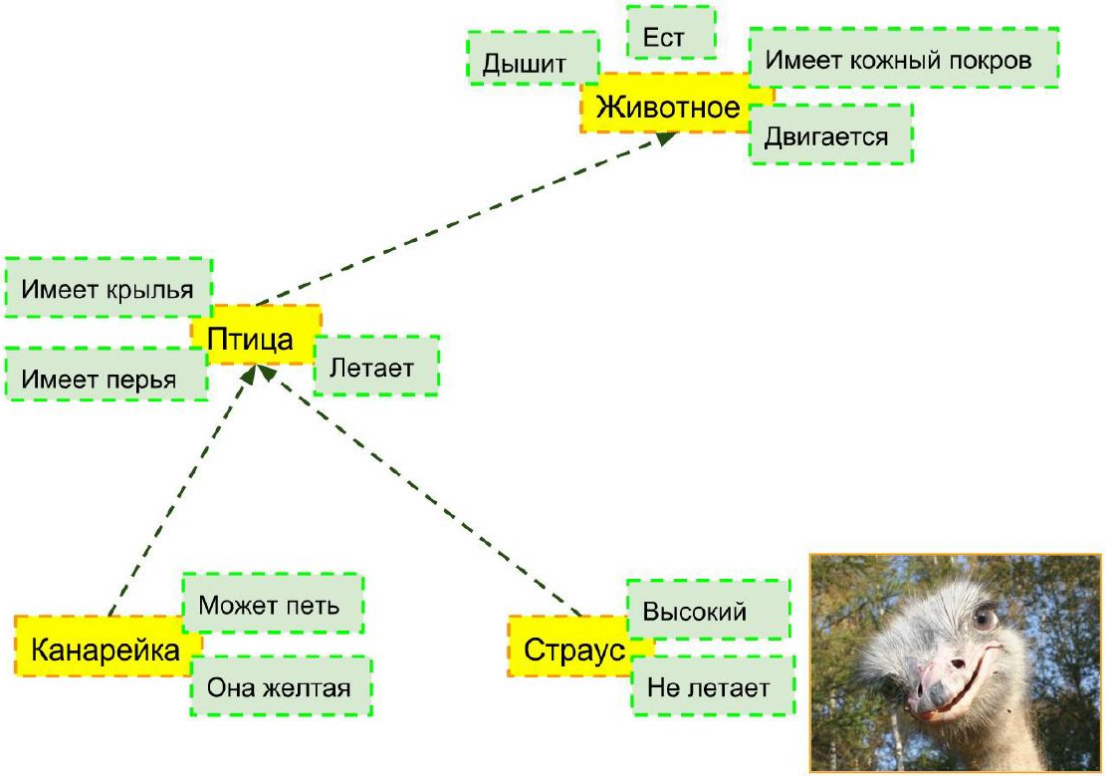
Python это не Java

```
def make_title(self):
    if self.title == 'Mister':
        return 'Mr. ' + self.surname

class Person:
    pass

john = Person()
john.title = 'Mister'
john.surname = 'Peterson'
Person.get_name = make_title
print(john.get_name())  # ???
```

Наследование



Наследование

Наследование – это свойство системы, позволяющее описать новый класс на основе уже существующего с частично или полностью заимствующейся функциональностью.

Класс, от которого производится наследование, называется **базовым** или **родительским**.

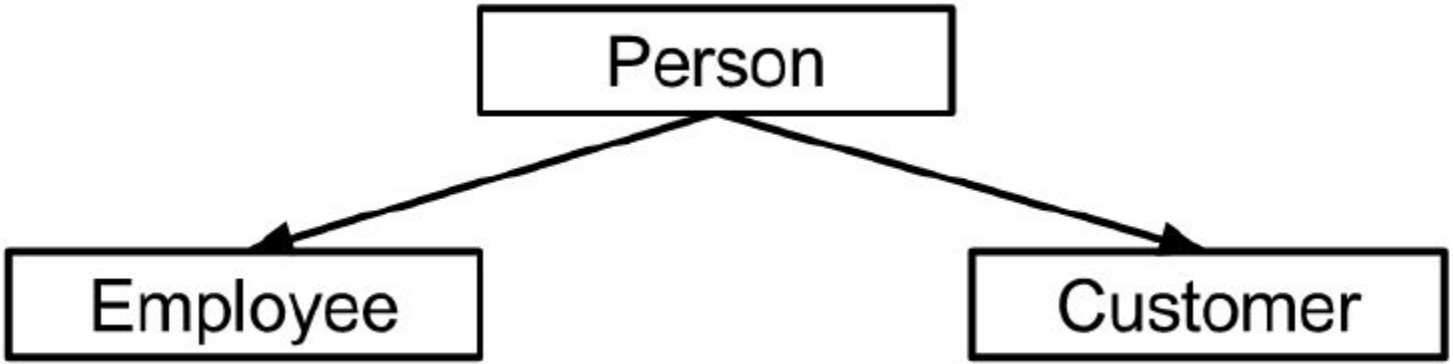
Новый класс – **потомком**, **наследником** или п**роизводным** классом.

Отношения - “является”.

Наследование

```
class DerivedClassName(BaseClassName):
    <statement-1>
    ...
    <statement-N>
```

Наследование



Наследование

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

class Employee(Person):
    def __init__(self, name, job_title):
        super().__init__(name)
        self.job_title = job_title

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

Связывание классов - замещение

```
class Super:
    def method(self):
        print('in Super.method')

class Replacer(Super) :
    def method(self):
        print('in Replacer.method')
```

Связывание классов - расширение

```
class Super:
    def method(self):
        print('in Super.method')

class Extender(Super):
    def method(self):
        print('starting Extender.method')
        super().method()
        print('ending Extender.method')
```

Связывание классов - поставщик

```
class Super:
    def delegate(self):
        self.action()

    def action():
        pass

class Provider(Super):
    def action(self):
        print('in Provider.action')
```

Ассоциация

Ассоциация – это когда один класс включает в себя другой класс в качестве одного из полей.

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

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

Ассоциация

```
class Team:
    def __init__(self, name):
        self.name = name
        self.players = []

    def add_player(self, person):
        self.players.append(person)

    def print_team(self):
        print('"{}" team:'.format(self.name))
        for player in self.players:
            print(player.get_name())
```

Ассоциация

```
p1 = Person('Александр')
p2 = Person('Николай')

team = Team('best_ever')
team.add_player(p1)
team.add_player(p2)
team.print_team()
```

```
"best_ever" team:
Александр
Николай
```

Агрегация

```
class ElectricEngine:
    ...

class Car:

    def __init__(self, engine):
        self.engine = engine


engine = ElectricEngine()
sport_car = Car(engine)
```

Композиция

```
class ElectricEngine:
    ...

class Car:

    def __init__(self):
        self.engine = ElectricEngine()

sport_car = Car()
```

Последовательность исполнения кода

```
import random

class RandomNumber:
    number = random.randint(1, 100)

print(RandomNumber().number)
print(RandomNumber().number)
```

Последовательность исполнения кода

```
import random

class RandomNumber:
    number = random.randint(1, 100)

print(RandomNumber().number)
print(RandomNumber().number)
```

```
72
72
```

Последовательность исполнения кода

```
import random

class RandomNumber:
    def __init__(self):
        self.number = random.randint(1, 100)

print(RandomNumber().number)
print(RandomNumber().number)
```

Последовательность исполнения кода

```
import random

class RandomNumber:
    def __init__(self):
        self.number = random.randint(1, 100)

print(RandomNumber().number)
print(RandomNumber().number)
```

```
22
91
```

Специальные методы

`__eq__(self, other)`	`self == other`
`__ne__(self, other)`	`self != other`
`__lt__(self, other)`	`self < other`
`__gt__(self, other)`	`self > other`
`__le__(self, other)`	`self <= other`
`__ge__(self, other)`	`self >= other`

Специальные методы

`__add__(self, other)`	`self + other`
`__sub__(self, other)`	`self - other`
`__mul__(self, other)`	`self * other`
`__floordiv__(self, other)`	`self // other`
`__truediv__(self, other)`	`self / other`
`__mod__(self, other)`	`self % other`
`__pow__(self, other)`	`self ** other`

Специальные методы

`__str__(self)`	`str(self)`
`__repr__(self)`	`repr(self)`
`__len__(self)`	`len(self)`

Специальные методы

`__getattr__(self, name)`	`obj.name`
`__setattr__(self, name, value)`	`obj.name = value`
`__delattr__(self, name)`	`del obj.name`

`__call__(self[, args...])`	`obj(args...)`

Упражнение 1

Напишите код, описывающий класс Animal, добавьте атрибуты имени животного и его сытости, метод eat - увеличивающий “сытость” животного, методы get_name и set_name, конструктор класса Animal выводящий сообщение при создании объекта.
Создайте несколько объектов и вызовите методы set_name/get_name/eat.

Упражнение 2

Создайте 2 класса разных животных, отнаследованных от класса Animal.
Определите метод make_noise, который будет уменьшать сытость животного и в зависимости от сытости выводить разное сообщение, специфичное для данного животного.

Упражнение 3

Сделать класс определяющий прямоугольный земельный участок. Реализовать операции сравнения участков по площади. Определить метод, проверяющий, можно ли сложить/вычесть 2 участка (одна из сторон совпадает по длине). Реализовать методы сложения/вычитания участков.

Домашнее задание

https://disk.yandex.ru/i/OKE-e3oj9qXpmg