Programing in Python

Lecture 9 - Classes

Instructor: Zhandos Yessenbayev

Outline

- Program Flow
- Object-Oriented Programming
- Python Classes
- Class Example

Program Flow

- Programming patterns we saw:
    - Sequential code (operators, assignment)
    - Conditional code (if statements)
    - Repetitive code (loops)
    - Store and reuse (functions)
    - Modular programming (modules)

Program Flow

```
import my_module

try:
    fhand = open("input.txt")
except:
    print('File cannot be opened')
    exit()

def greet():
    print('Hello, world!’)

for line in fhand:
    if line.startswith('Hello'):
        greet()

fhand.close()
```

Modules

Conditionals

Functions

Loops

Conditionals

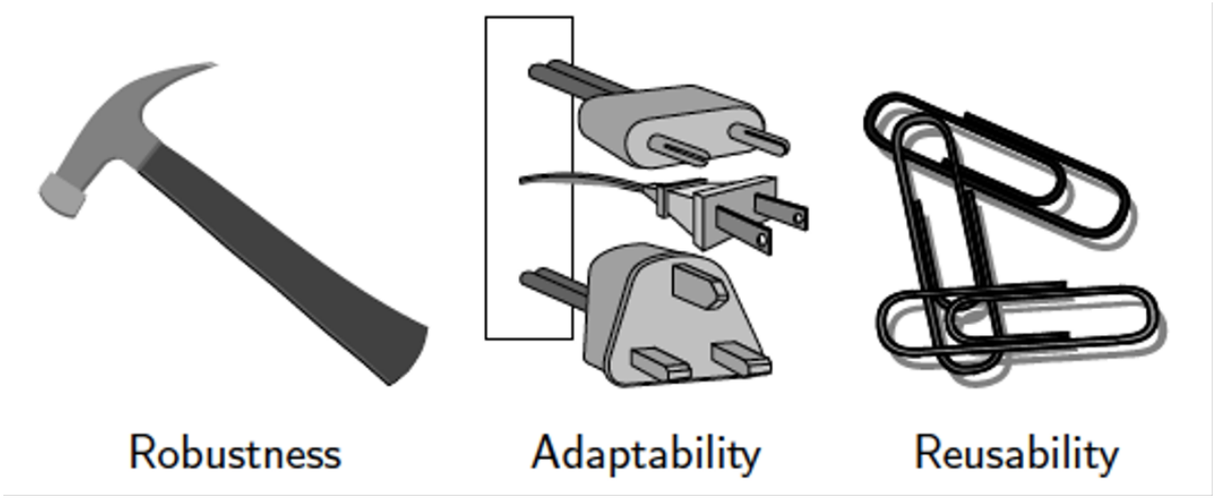
Sequential

Object-Oriented Programming

- **Object-Oriented Programming (OOP)** is a programming paradigm based on the concept of "**objects**", which may contain **data** and **methods**.

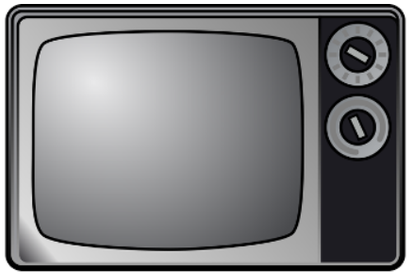
OOP Design Goals:

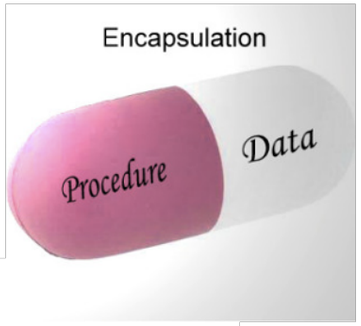
- robustness
  - Gracefully handle failures
- adaptability
  - Evolve as necessary
- reusability
  - Reuse the same code

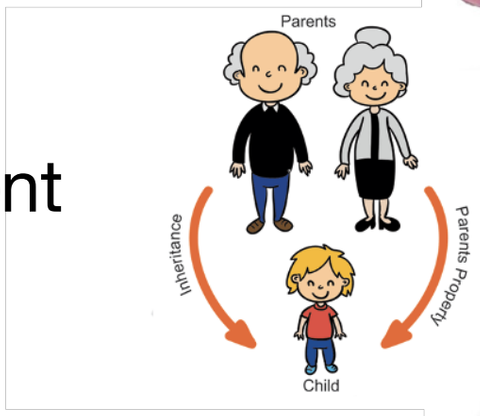


OOP Principles

- **Abstraction**
  - Exposing the properties that best describe an object
- **Encapsulation**
  - Combining data and methods while hiding the details
- **Inheritance**
  - Ability to inherit the properties from a parent object
- **Polymorphism**
  - Ability to use, override or extend the parent's behavior







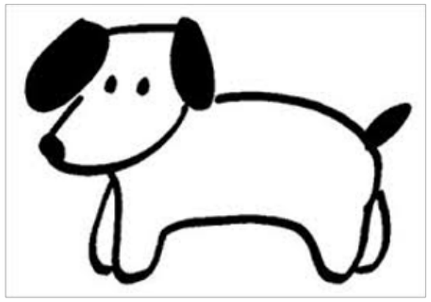
Classes and Objects

- The concepts of **classes** and **objects** are central in OOP
- **Classes** are category and description of the objects
    - data (attributes, fields, properties)
    - methods (procedures, functions)
- **Objects** are real instances of classes

Classes and Objects

- **Dog** is a concept (class) which:
    - has a *name*, *color* and *size* (properties)
    - *barks*, *eats* and *plays* (methods)





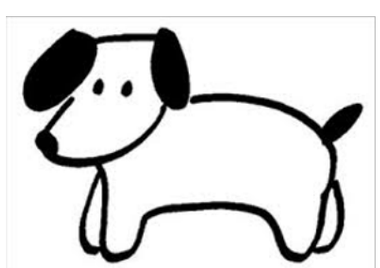
instances (objects) of class Dog

Python
Classes and Objects

- In Python, **classes** are defined using the **class** keyword.

```
>>> class Dog:    # class definition
...    pass
...
>>>
>>> Leo = Dog()  # objects creation
>>> Rex = Dog()
```

Leo



Rex

Leo, Rex are *objects* of the class Dog

Encapsulation

- In Python, a class may have there types of content:
  - **Data** (properties, attributes) - *variables* that store information
  - **Methods** - *functions* to access and modify data, to do some computations
  - **Constructor** - *special function* to create objects of the class

```
>>> class Dog:
...     name = ''
...     def bark(self):
...         print('gav gav')
...
>>> d = Dog()
>>> d.name
>>> d.name = 'Leo'
>>> d.bark()
```

self must be the first argument of a method

self keyword

- **self** is used to access *variables and functions* that belong to the class
- **self** must be the first argument of a method
- **self** is a reference to the current instance of the class

```
>>> class Dog:
...     name = ''
...     def get_name(self):
...         print(self.name)
...     def set_name(self, new_name):
...         self.name = new_name
...
>>> d = Dog()
>>> d.name
>>> d.set_name('Rex')
>>> d.get_name()
```

self must be the first parameter of a method

self is used to access variables that belong to the class

self is a reference to the current instance of the class

__init__ Function

- **__init__** is called ***constructor*** and always executed when the class is being initiated

- **__init__** is used to assign values to object properties, or other operations that are necessary to do when the object is created

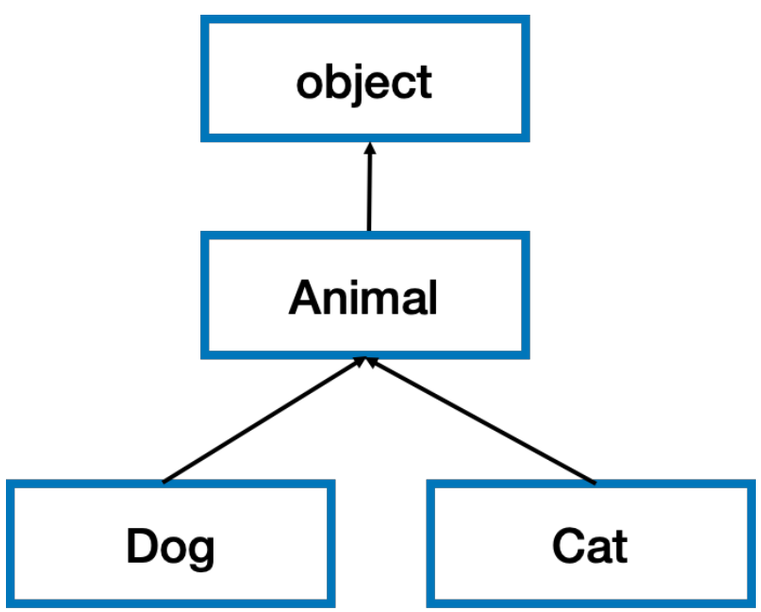
```
>>> class Dog:
...     name = ''
...     def __init__(self, name='Leo'):
...         self.name = name
...
>>> leo = Dog()
>>>
>>> rex = Dog('Rex')
>>> rex.name
```

assign values
to class properties

Inheritance

- **Inheritance** allows us to define a **child class** that inherits all the methods and properties from **parent class**.

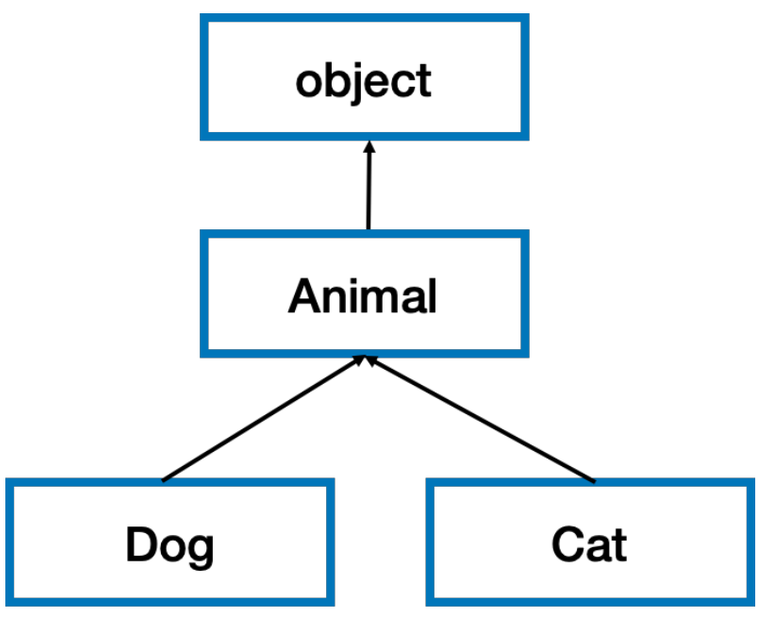
- In Python, all classes inherit from a special class **object**




```
>>> dir(object)
>>> ['__class__', '__delattr__', '__dir__',
'__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__gt__', '__hash__',
'__init__', '__init_subclass__', '__le__',
'__lt__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__']
```

Inheritance

- **parent** class defines core attributes and methods

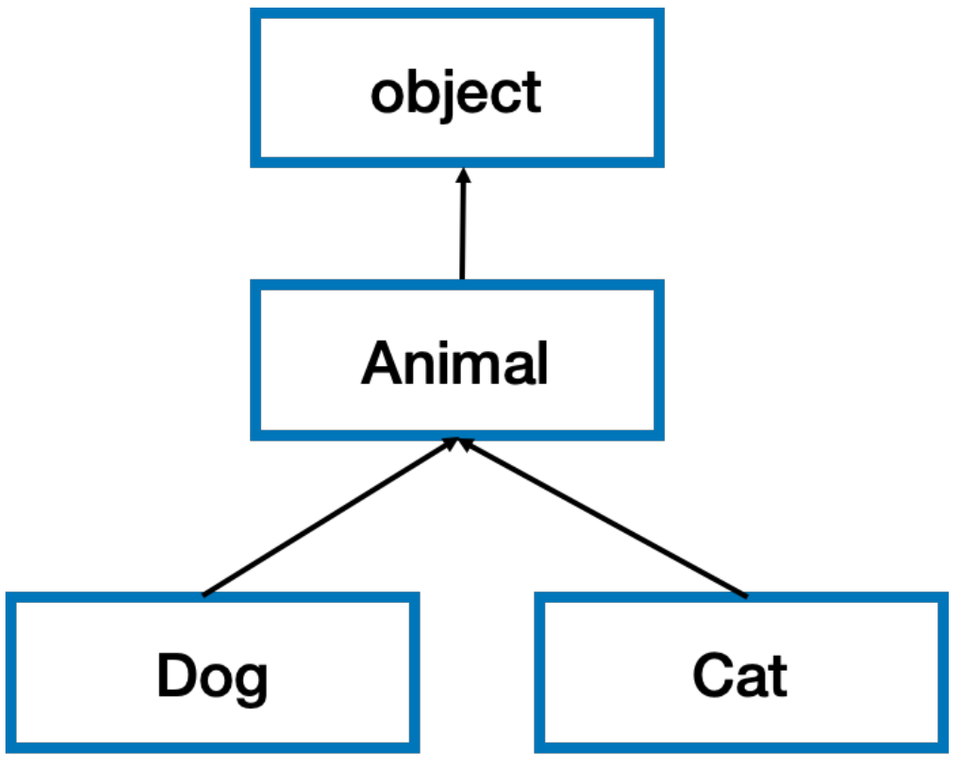


Parent class **Animal**

```
>>> class Animal():
...     name = ''
...     def __init__(self, name='animal'):
...         self.name = name
...
>>> a = Animal()
>>> a.name
```

Inheritance

- **child** class *inherits* all the methods and properties from **parent** class and can *extend* it.



Child class **Dog**

```
>>> class Dog(Animal):
...     pass
...
>>> d = Dog('Rex')
>>> d.name
```

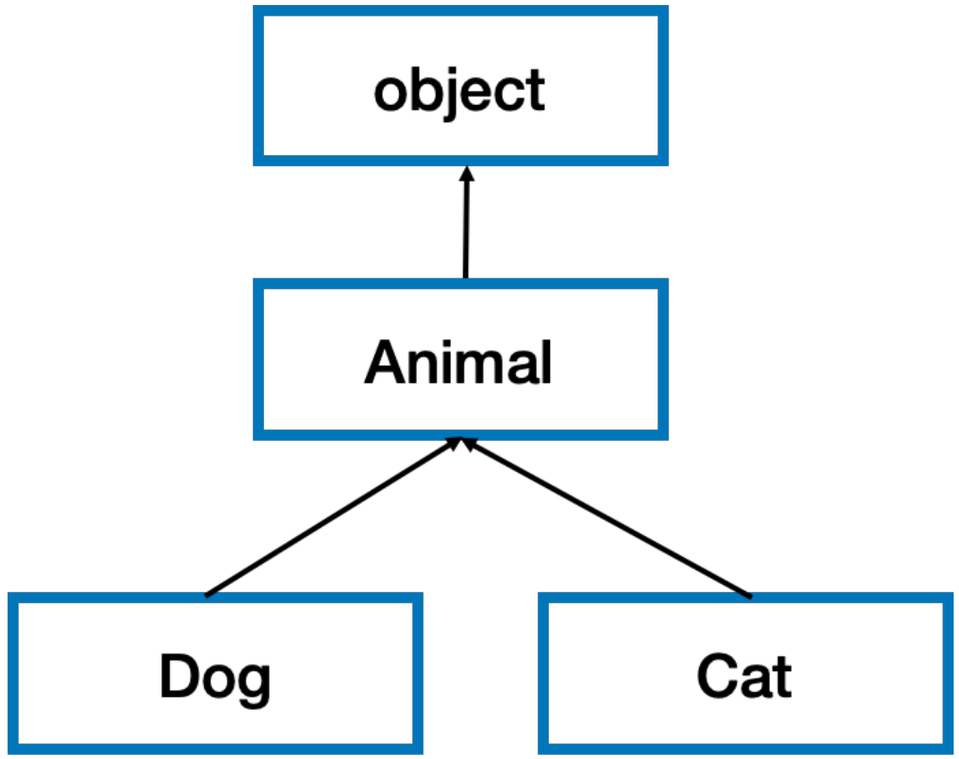
Child class **Cat**

```
>>> class Cat(Animal):
...     color = 'white'
...
>>> c = Cat('Kitty')
>>> c.name
>>> c.color
```

Polymorphism

- **Polymorphism** allows Python to decide on which object's method to use at runtime

- Same methods can be ***redefined*** in child classes



Polymorphism

```
>>> class Animal():
...     name = ''
...     def __init__(self, name='animal'):
...         self.name = name
...
```

```
>>> class Dog(Animal):
...     name = ''
...     def __init__(self, name='dog'):
...         self.name = name
...
>>> class Cat(Animal):
...     name = ''
...     def __init__(self, name='cat'):
...         self.name = name
...
>>> d = Dog()          # c = Cat()
>>> d.name.            # c.name
```

super keyword

- **super( )** function accesses the parent class

Child class **Cat**

```
>>> class Cat(Animal):
...    color = ‘white’
...    def __init__(self, name, color):
...        super().__init__(name)  # initialize parent
...        self.color = color      # initialize child
...
>>> c = Cat(‘Kitty’, ‘black’)
>>> c.name
>>> c.color
```

Class Example

- Create a class **Person** with relevant attributes and methods
- Create two classes **Student** and **Teacher** which are subclasses of **Person**
- Demonstrate *polymorphism* using these classes

Thanks!