15-110 Principles of Computing – S19

Lecture 19:
Object-Oriented Programming 1

Teacher:
Gianni A. Di Caro

Python *objects* so far

- Built-in object types:

```
int, float, bool
```

```
a = 2
b = 2 + 1
c = 3.5 * b
```

Information data (values) only

- Built-in object types:

```
def my_function(args):
    # does something
    return something
```

Operations (actions) only

- Built-in object types:

```
str, list, tuple, dict, set, file
```

```
l = []
l.append(44)
l.append(22)
l.extend([1,0,9,12])
l.sort()
```

```
r = f.readline()
a = f.readlines()
f.seek(0)
f.close()
```

Information data (values)

+

Associated methods (operations) on the data

- Importing from modules and creating variables of *module type*

```
import csv
```

```
csv_data = csv.writer (f)
csv_data.writerow([12, 100, 5.5])
```

Class objects

- Built-in object types:

```
str, list, tuple, dict, set, file
```

✓ Each data type comes with a set of specialized methods, identified through the *dot notation*, to manipulate the data type itself

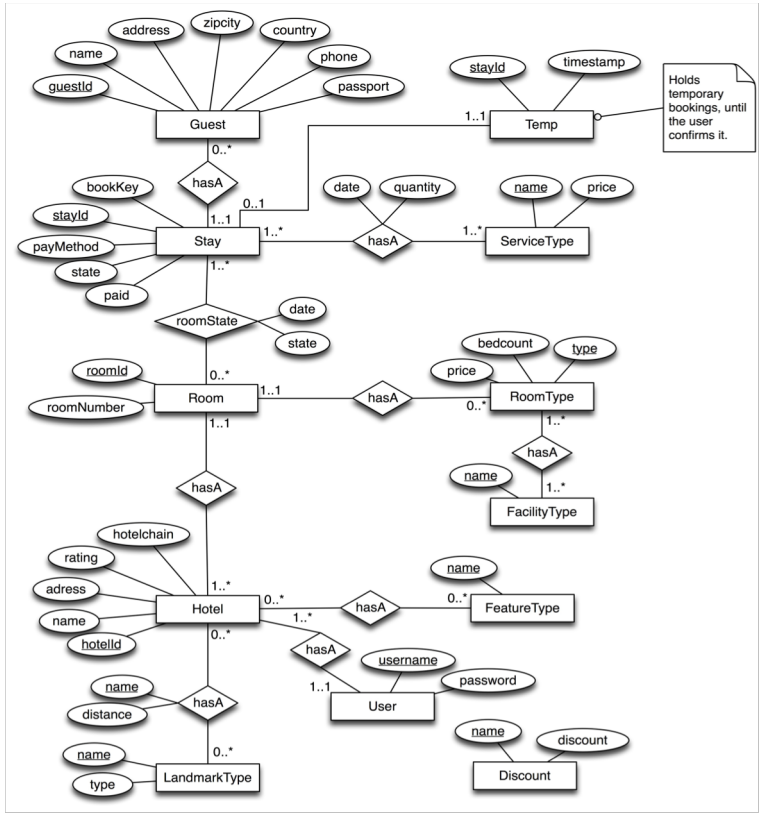
- List: `append(), insert(), extend(), remove(), pop(), count(), sort(), reverse(), copy(), …`
- Dict: `get(), keys(), values(), items(), pop(), popitem(), clear(), update(), copy(), …`
- …

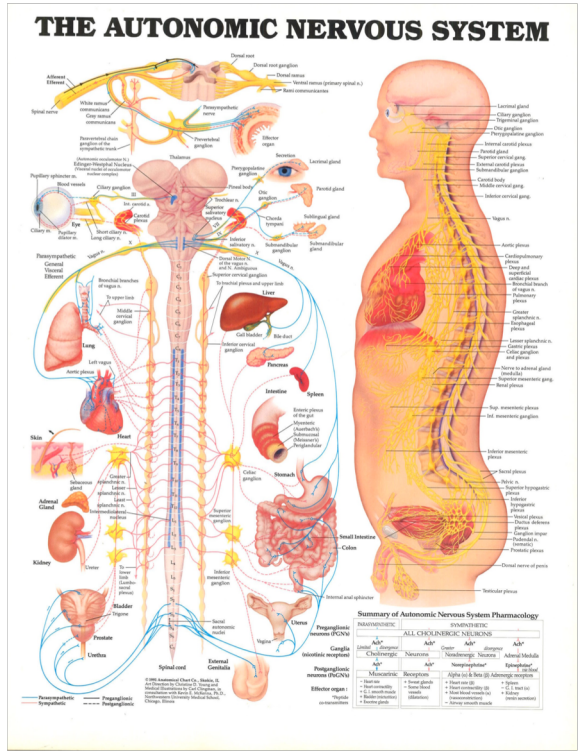
✓ Each new instance of a data type has possibly different values, but provides the same set of methods

```
l1 = []
l2 = [1,2,3]
l1.append(99)
l2.append(99)
```

➢ Can we generalize and extend this approach to custom-defined types? → Yes, by using ***class* objects**

Real-world is complex, populated by complex, interacting entities





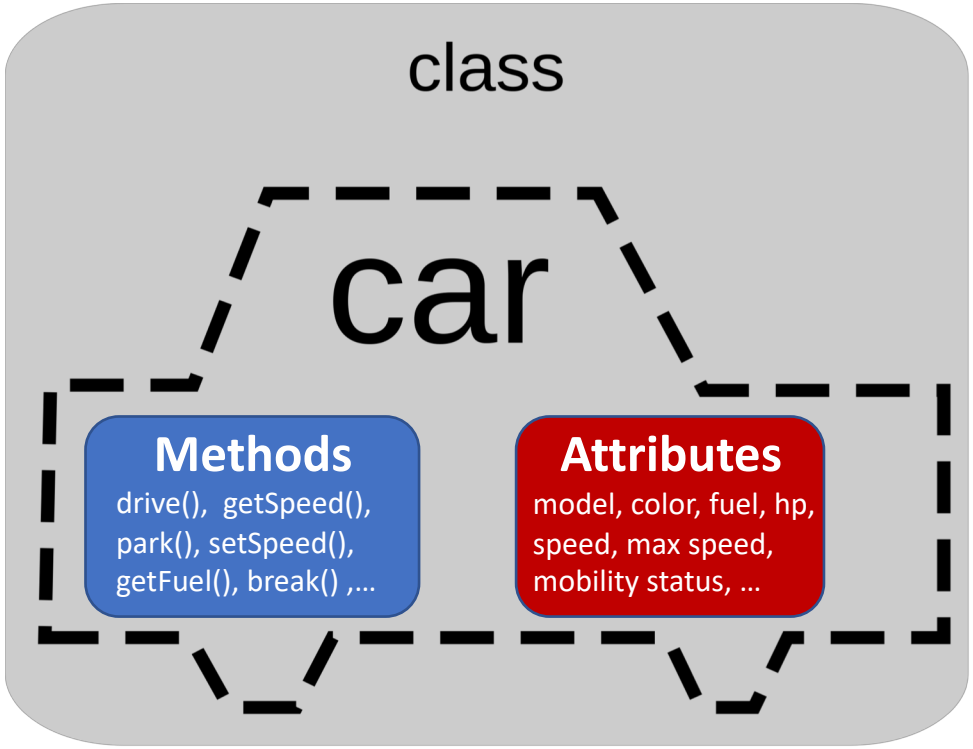
How do we use software to model, study, manage, control such complex systems in a way which is effective, efficient, maintainable, reusable, extensible, scalable, adaptable, …?

- ✓ **Object-oriented** design and programming comes into help to tackle real-world complexity: it is based on the definition of ***classes***
- ✓ *Idea:* let's pack in the same object (the class) everything that we need for describing, use, and interact with a complex entity

The blueprint analogy

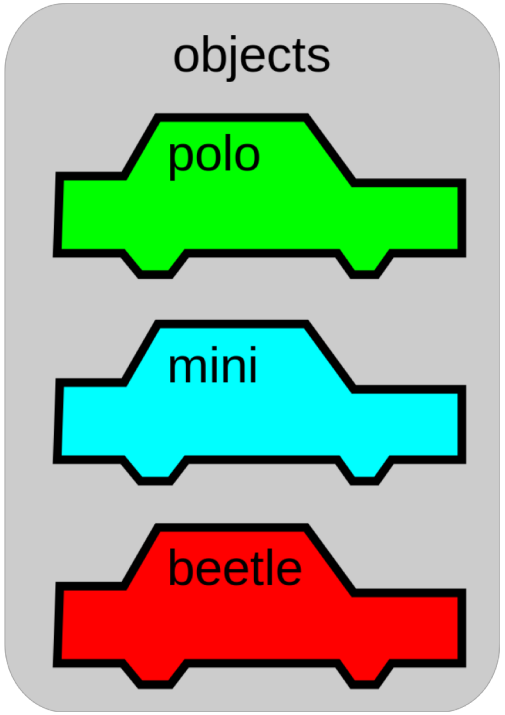
- For instance, we want to make a software to be used in the context of for road monitoring and management (whatever this could mean…)
- We first need to represent a car, since a road will feature many different cars, in different conditions of mobility

A car class



A class is a like a *blueprint*: a design guide, an operational pattern that can be used to create multiple, independent, **instances** of class objects

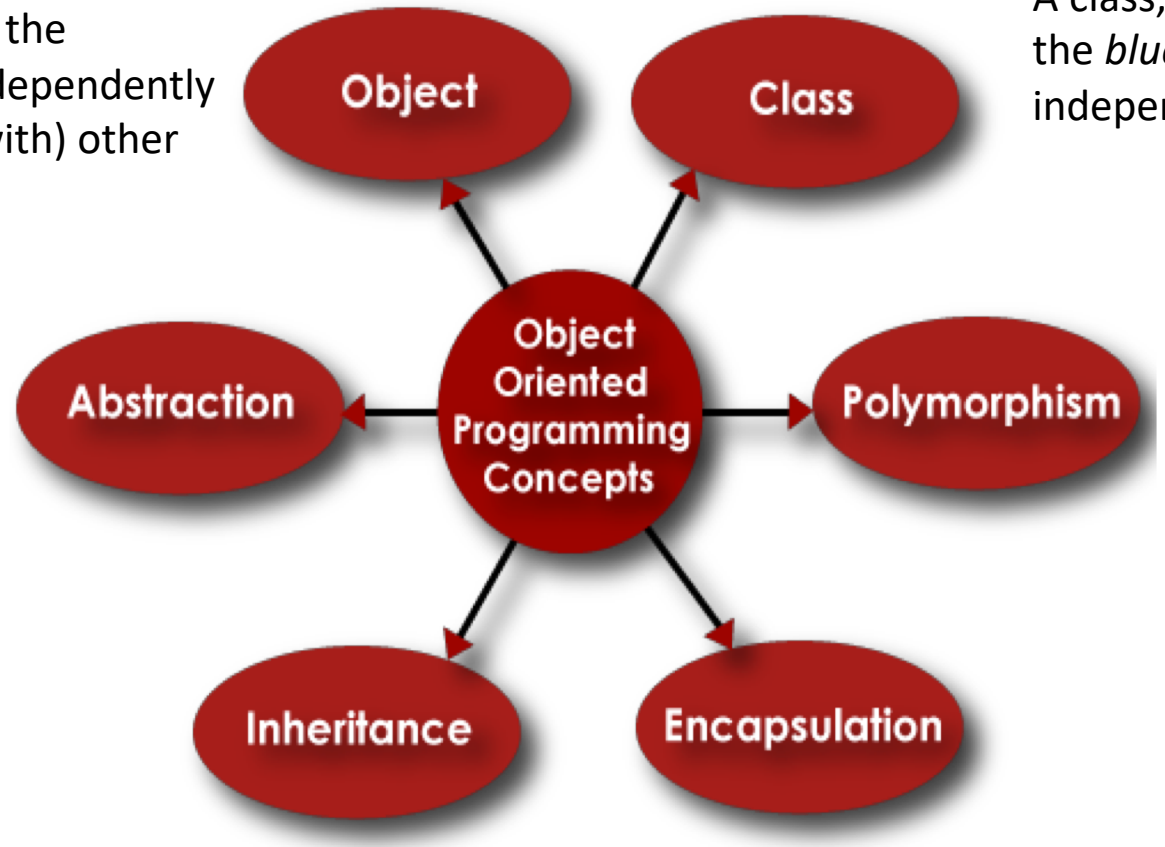
Different instances have the same design: are equipped with the same methods and attributes, but attributes might have different values



Object-Oriented (OO) design and programming

An instance of the class, with the same methods and its own values for the attributes, and which "lives" independently from (but possibly interacting with) other objects from the same class

A class, *attributes + methods* provides the *blueprint* for creating multiple independent instances of class objects



A class abstracts away implementation details, it provides the *interface* to make use of potentially complex entities and procedures

Polymorphism: taking different forms. The same method (same name, same interface) can process different input objects (forms) and produces different outputs accordingly, transparently to whom is invoking the method. Operator overloading is an expression of polymorphic behavior. Functions like sorted(), max(), min() are polymorphic. A subclass can make use of the same method of its base class, but specialized

Inheritance it allows to create a new class (a *child, sub-, derived* class) that is based upon an existing class (the *parent, base, super* class), by adding new attributes and methods on top of the existing class, (and/or by specializing existing methods). The child class *inherits* attributes and methods of the parent class.

A class can *expose* only data and methods data are strictly necessary to use and interact with the class object, internal data structures are *encapsulated* inside the class: the external user shouldn't mess up with them!

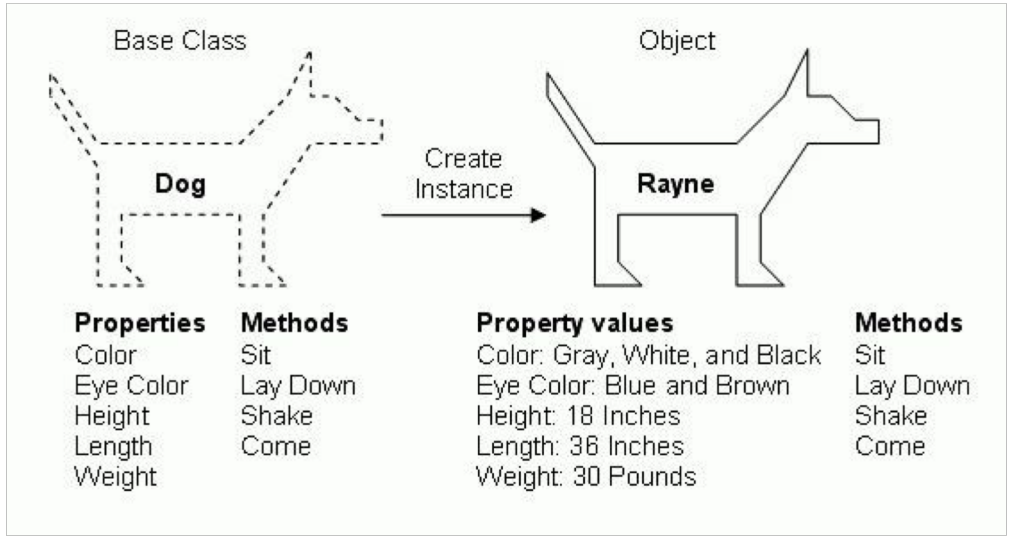
Syntax for creating a class in python

```
class ClassName:

    def __init__(self, args):
        # code for object initialization

    def one_class_method(self, args):
        # code for this method

    def another_class_method(self, args):
        # code for this method
```

- `self`: reference to the *specific class object*
- `__init__`: a *magic method*, it allows to initialize the attributes of the object, optional
- The class definition can have as many methods as we need
- Adopt the convention ClassName and not, for instance, class_name, to define the name of a class

Simple example

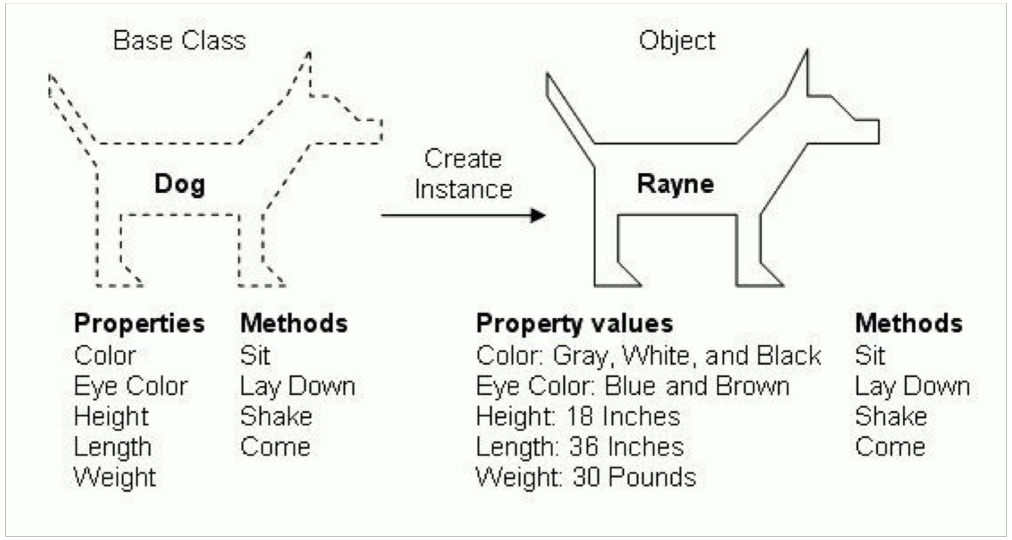
```
class Dog:
    def __init__(self, color=None, eyes=None,
                 height=None, length=None, weight=None, name=None):
        self.color = color
        self.eyes = eyes
        self.height = height
        self.length = length
        self.weight = weight
        self.name = name
        self.status = None

    def sit(self):
        if self.status != 'sitting':
            print('Sit!')
            # tell the dog to sit
            self.status = 'sitting'
        else:
            print("It's sitting already!")

    def lay_down(self):
        if self.status != 'laying down':
            print('Lay down!')
            # tell the dog to lay down
            self.status = 'laying down'
        else:
            print("It's laying down already!")
```




```
def shake(self):
        if self.status != 'shaking':
            print('Shake hands!')
            # tell the dog to shake hands
            self.status = 'shaking'
        else:
            print("It's shaking hands already!")

    def come(self):
        if self.status != 'near me':
            print('Come here!')
            # tell the dog to come close
            self.status = 'near me'
        else:
            print("It's here already!")
```

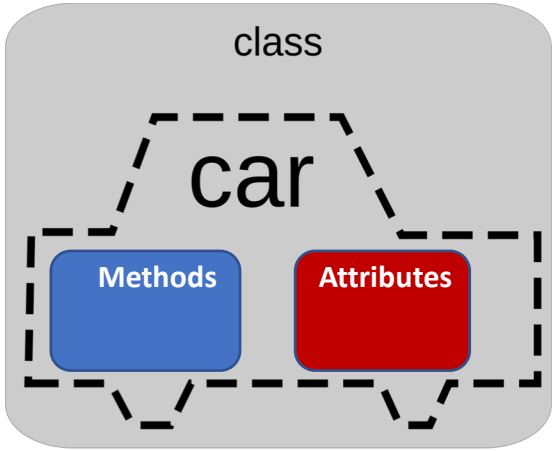
Simple example

```
class Dog:
    def __init__(self, color=None, eyes=None,
                 height=None, length=None, weight=None, name=None):
        self.color = color
        self.eyes = eyes
        self.height = height
        self.length = length
        self.weight = weight
        self.name = name
        self.status = None

    def sit(self):
        if self.status != 'sitting':
            print('Sit!')
            # tell the dog to sit
            self.status = 'sitting'
        else:
            print("It's sitting already!")

    def lay_down(self):
        if self.status != 'laying down':
            print('Lay down!')
            # tell the dog to lay down
            self.status = 'laying down'
        else:
            print("It's laying down already!")

    def shake(self):
        if self.status != 'shaking':
            print('Shake hands!')
            # tell the dog to shake hands
            self.status = 'shaking'
        else:
            print("It's shaking hands already!")

    def come(self):
        if self.status != 'near me':
            print('Come here!')
            # tell the dog to come close
            self.status = 'near me'
        else:
            print("It's here already!")
```




```
rayne = Dog('gray', 'blue', 18, 36, 30, 'Rayne')
rayne.come()
rayne.come()
rayne.lay_down()

skinny = Dog('black', 'brown', 19, 38, 27, 'Skinny')
skinny.shake()
```

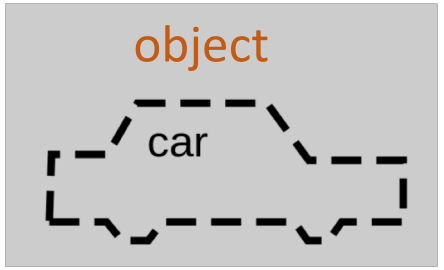
The life cycle of a class object



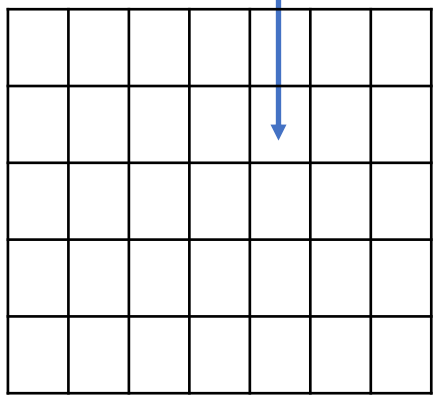
Abstract class definition
(maybe permanently stored in a module)

Blueprint

__new__
(constructor)



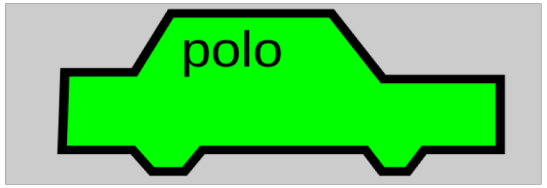
Object creation, in memory

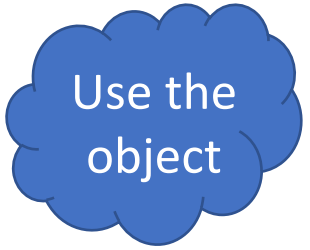


Memory space: address is stored in `self` variable

Specify attributes

__init__
(initializer)





Class methods

Dispose of the object (not the class!)

__del__
(destructor)