

PROGRAMMING IN PYTHON I

Classes



Andreas Schörgenhumer
Institute for Machine Learning

Copyright Statement

This material, no matter whether in printed or electronic form, may be used for personal and non-commercial educational use only. Any reproduction of this material, no matter whether as a whole or in parts, no matter whether in printed or in electronic form, requires explicit prior acceptance of the authors.

Contact

Andreas Schörgenhumer

Institute for Machine Learning
Johannes Kepler University
Altenberger Str. 69
A-4040 Linz

E-Mail: schoergenhumer@ml.jku.at

Write mails only for personal questions

[Institute ML Homepage](#)

CLASSES IN PYTHON



Motivation

- Often, we want reusability and modularity of our code
 - Easier software design
 - Easier data modeling

Motivation

- Often, we want reusability and modularity of our code
 - Easier software design
 - Easier data modeling
- **Object-oriented programming** (OOP) tries to increase reusability and modularity
 - Programming paradigm based on the concept of **objects**

Motivation

- Often, we want reusability and modularity of our code
 - Easier software design
 - Easier data modeling
- **Object-oriented programming** (OOP) tries to increase reusability and modularity
 - Programming paradigm based on the concept of **objects**
- **Objects** (in OOP)
 - Combination of variables, functions and data structures
 - Can contain state (data) and behavior (methods)

Motivation

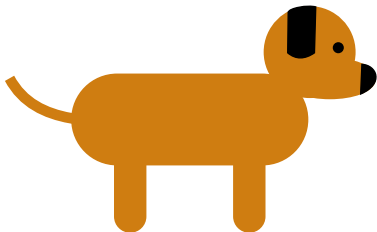
- Often, we want reusability and modularity of our code
 - Easier software design
 - Easier data modeling
- **Object-oriented programming** (OOP) tries to increase reusability and modularity
 - Programming paradigm based on the concept of **objects**
- **Objects** (in OOP)
 - Combination of variables, functions and data structures
 - Can contain state (data) and behavior (methods)
- **(Data) Attributes** (aka **fields**)
 - State (data) associated with an object

Motivation

- Often, we want reusability and modularity of our code
 - Easier software design
 - Easier data modeling
- **Object-oriented programming** (OOP) tries to increase reusability and modularity
 - Programming paradigm based on the concept of **objects**
- **Objects** (in OOP)
 - Combination of variables, functions and data structures
 - Can contain state (data) and behavior (methods)
- **(Data) Attributes** (aka **fields**)
 - State (data) associated with an object
- **Methods**
 - Behavior (functions) provided by an object

Objects: Example (1)

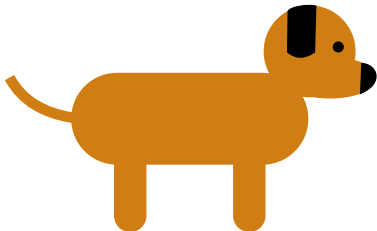
- Example: We want to create an object to describe a dog named “Bello”



Objects: Example (2)

- Our dog object can have attributes that hold values describing the name and fur color

Attributes

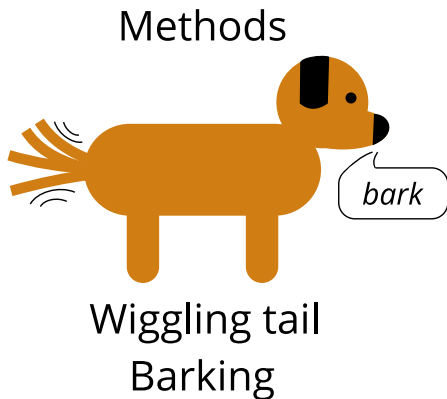


Name: "Bello"

Fur color: "brown"

Objects: Example (3)

- Our dog object can have methods that execute wiggling of its tail and barking



Classes (1)

- **Classes** define objects (like a blue-print). Creating a new class means creating a new **type**

Classes (1)

- **Classes** define objects (like a blue-print). Creating a new class means creating a new **type**
- An **object** is an **instance of a class/type** that exists **uniquely** among all other objects

Classes (1)

- **Classes** define objects (like a blue-print). Creating a new class means creating a new **type**
- An **object** is an **instance of a class/type** that exists **uniquely** among all other objects
- Example: Assume we want to describe multiple dogs

Classes (1)

- **Classes** define objects (like a blue-print). Creating a new class means creating a new **type**
- An **object** is an **instance of a class/type** that exists **uniquely** among all other objects
- Example: Assume we want to describe multiple dogs
 - We would first create a dog class

Classes (1)

- **Classes** define objects (like a blue-print). Creating a new class means creating a new **type**
- An **object** is an **instance of a class/type** that exists **uniquely** among all other objects
- Example: Assume we want to describe multiple dogs
 - We would first create a dog class
 - The dog class would contain attributes and methods that are used to describe a dog

Classes (1)

- **Classes** define objects (like a blue-print). Creating a new class means creating a new **type**
- An **object** is an **instance of a class/type** that exists **uniquely** among all other objects
- Example: Assume we want to describe multiple dogs
 - We would first create a dog class
 - The dog class would contain attributes and methods that are used to describe a dog
 - If we want to describe an individual dog, we create an instance of our dog class (a new object)

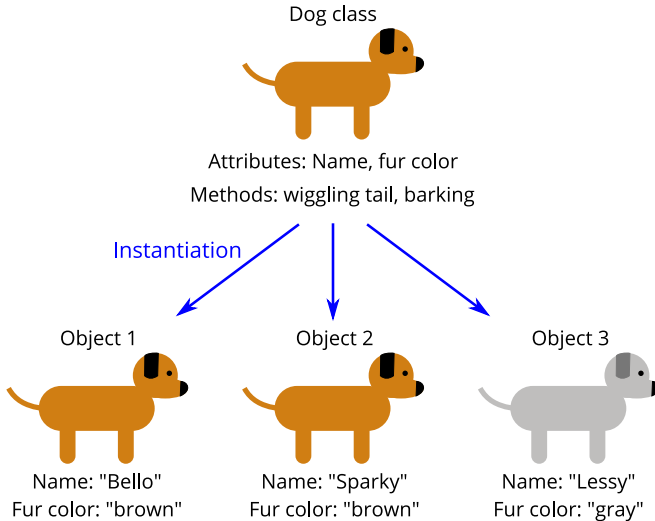
Classes (1)

- **Classes** define objects (like a blue-print). Creating a new class means creating a new **type**
- An **object** is an **instance of a class/type** that exists **uniquely** among all other objects
- Example: Assume we want to describe multiple dogs
 - We would first create a dog class
 - The dog class would contain attributes and methods that are used to describe a dog
 - If we want to describe an individual dog, we create an instance of our dog class (a new object)
 - Each instance is an individual object and contains a copy of the attributes and methods from our dog class

Classes (1)

- **Classes** define objects (like a blue-print). Creating a new class means creating a new **type**
 - An **object** is an **instance of a class/type** that exists **uniquely** among all other objects
 - Example: Assume we want to describe multiple dogs
 - We would first create a dog class
 - The dog class would contain attributes and methods that are used to describe a dog
 - If we want to describe an individual dog, we create an instance of our dog class (a new object)
 - Each instance is an individual object and contains a copy of the attributes and methods from our dog class
- We can reuse the code for a dog object for every dog!

Classes (2)



Classes (3)

- We can also create (**derive**) new classes from existing classes, i.e., **extend** existing classes:

Classes (3)

- We can also create (**derive**) new classes from existing classes, i.e., **extend** existing classes:
 - The new classes are referred to as **child classes** or **subclasses**

Classes (3)

- We can also create (**derive**) new classes from existing classes, i.e., **extend** existing classes:
 - The new classes are referred to as **child classes** or **subclasses**
 - The classes the subclasses are derived from are referred to as **parent classes**, **base classes** or **superclasses**

Classes (3)

- We can also create (**derive**) new classes from existing classes, i.e., **extend** existing classes:
 - The new classes are referred to as **child classes** or **subclasses**
 - The classes the subclasses are derived from are referred to as **parent classes**, **base classes** or **superclasses**
- Subclasses can **inherit** attributes and method definitions from their base classes
 - Attributes/methods from parent classes are available in child classes but can be modified/extended

Classes (4)

- Example: Assume that we now want to describe guard dogs that behave like our dog class but also have a “guard” method
 - We can derive a guard-dog class from our dog class, which inherits the attribute and method definitions from the dog class
 - We can add an additional “guard” method to our guard-dog class
 - We can now create instances of our guard-dog class

Classes in Python

- Every class in Python is (indirectly) derived from the base class **object**

Classes in Python

- Every class in Python is (indirectly) derived from the base class `object`
- We have already worked with classes in Python!
 - Example: Our integer objects are instances of the `int` class, which is derived from the `object` class

Classes in Python

- Every class in Python is (indirectly) derived from the base class **object**
- We have already worked with classes in Python!
 - Example: Our integer objects are instances of the `int` class, which is derived from the `object` class
- Classes can be created using the **class** statement
 - Class names (by convention) should be **CapWords**
 - Example: `MyNewClass`

Classes in Python

- Every class in Python is (indirectly) derived from the base class **object**
- We have already worked with classes in Python!
 - Example: Our integer objects are instances of the `int` class, which is derived from the `object` class
- Classes can be created using the **class** statement
 - Class names (by convention) should be **CapWords**
 - Example: `MyNewClass`
- Similarly to functions, classes create a **namespace**
 - Attributes and methods only exist within the class or an instance thereof

Class Syntax and Terminology

```
class Dog:
    """This class represents dogs."""

    kind = "canine"

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

    def bark(self):
        print(f"{self.name}: woof!")
```

Class Syntax and Terminology

```
class Dog:
    """This class represents dogs."""

    kind = "canine"

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

    def bark(self):
        print(f"{self.name}: woof!")
```

Class/Type name

Class Syntax and Terminology

```
class Dog:
    """This class represents dogs."""
    kind = "canine"

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

    def bark(self):
        print(f"{self.name}: woof!")
```

Class documentation

Class Syntax and Terminology

```
class Dog:
    """This class represents dogs."""

    kind = "canine"

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

    def bark(self):
        print(f"{self.name}: woof!")
```

Class attribute/field (exists only once)

Class Syntax and Terminology

```
class Dog:
    """This class represents dogs."""

    kind = "canine"

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

    def bark(self):
        print(f"{self.name}: woof!")
```

Special method for object initialization. Must be named `__init__`. The first parameter `self` references this new object.

Class Syntax and Terminology

```
class Dog:
    """This class represents dogs."""

    kind = "canine"

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

    def bark(self):
        print(f"{self.name}: woof!")
```

Object/Instance attribute/field (exists for each object).
Object attributes must be accessed with `self.attribute`
within the class definition

Class Syntax and Terminology

```
class Dog:
    """This class represents dogs."""

    kind = "canine"

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

    def bark(self):
        print(f"{self.name}: woof!")
```

Method (“belongs” to the `self` object). Object methods are bound to the object they were invoked on.

Class Syntax and Terminology

```
class Dog:
    """This class represents dogs."""

    kind = "canine"

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

    def bark(self):
        print(f"{self.name}: woof!")
```

Access of the above created object attribute `name`. The same preceding `self` must be done when calling object methods.

Access and Instantiation

- Access of class or object attributes via `MyClass.attribute` or `my_obj.attribute`:

- `Dog.kind` `my_dog.name`

Access and Instantiation

- Access of class or object attributes via `MyClass.attribute` or `my_obj.attribute`:

- ☐ `Dog.kind` `my_dog.name`

- Instantiation/Creation of new objects/instances via `my_obj = MyClass(...)`, where `...` are the arguments that will be passed in addition to `self` to the special method `__init__(self, ...)` (if there are any):

- ☐ `d = Dog("Bello")`

- ☐ This will create a new `Dog` object (with the object attribute `name` set to `"Bello"`) and store it in the variable `d`.

Access and Instantiation

- Access of class or object attributes via `MyClass.attribute` or `my_obj.attribute`:
 - `Dog.kind` `my_dog.name`
- Instantiation/Creation of new objects/instances via `my_obj = MyClass(...)`, where `...` are the arguments that will be passed in addition to `self` to the special method `__init__(self, ...)` (if there are any):
 - `d = Dog("Bello")`
 - This will create a new `Dog` object (with the object attribute name set to "Bello") and store it in the variable `d`.
- Invocation of methods via `my_obj.method(...)`, where `...` are the arguments that will be passed in addition to `self` to method (if there are any):
 - `d.bark()`
 - This will call `Dog.bark(d)`, i.e., `self` is automatically set to the object the method was invoked on (here: `self=d`)

Inheritance

■ Syntax to extend a base class (inherit from a base class):

- ☐ `class MyClass(object)`¹
- ☐ `class MySpecializedClass(MyClass)`
- ☐ `class GuardDog(Dog)`

¹All classes inherit from the base class `object` automatically.

Inheritance

- Syntax to extend a base class (inherit from a base class):
 - `class MyClass(object)`¹
 - `class MySpecializedClass(MyClass)`
 - `class GuardDog(Dog)`
- All attributes and methods are inherited. Additional attributes and methods can be provided and behavior of existing methods can be changed (**method overriding**)

¹All classes inherit from the base class `object` automatically.

Inheritance

- Syntax to extend a base class (inherit from a base class):

- ☐ `class MyClass(object)1`
- ☐ `class MySpecializedClass(MyClass)`
- ☐ `class GuardDog(Dog)`

- All attributes and methods are inherited. Additional attributes and methods can be provided and behavior of existing methods can be changed (**method overriding**)
- Special built-in **super** for accessing the base class:

```
class GuardDog(Dog):  
    def __init__(self, name, power):  
        super().__init__(name)  
        self.power = power
```

¹All classes inherit from the base class `object` automatically.

Inheritance

- Syntax to extend a base class (inherit from a base class):

- ☐ `class MyClass(object)1`
- ☐ `class MySpecializedClass(MyClass)`
- ☐ `class GuardDog(Dog)`

- All attributes and methods are inherited. Additional attributes and methods can be provided and behavior of existing methods can be changed (**method overriding**)

- Special built-in **super** for accessing the base class:

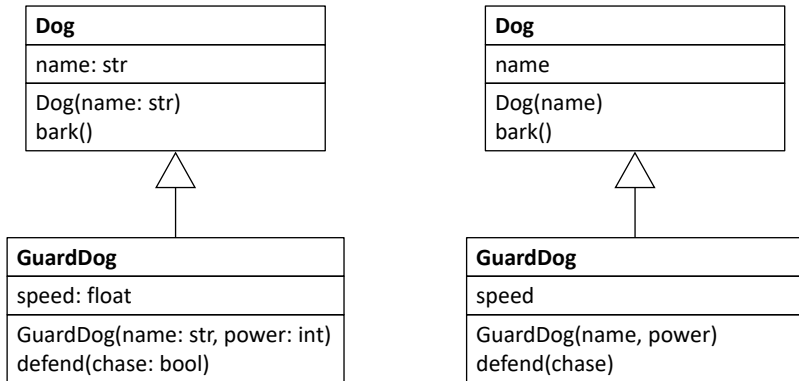
```
class GuardDog(Dog):  
    def __init__(self, name, power):  
        super().__init__(name)  
        self.power = power
```

- Python supports multiple inheritance

¹All classes inherit from the base class `object` automatically.

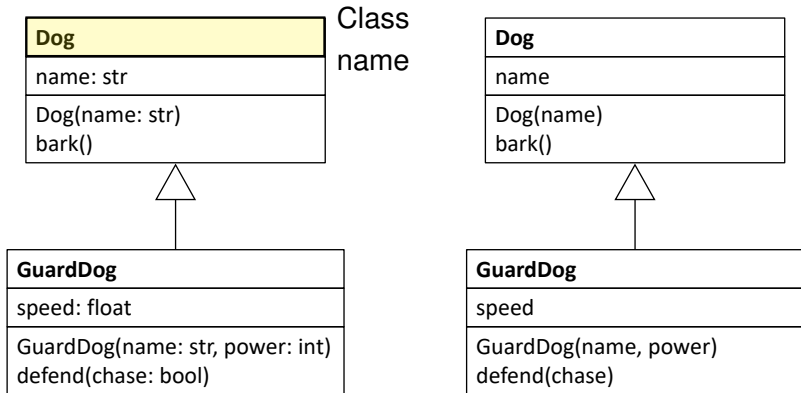
Graphical Class (Hierarchy) Notation

- UML (Unified Modeling Language) **class diagram**
- Very common. Allows to quickly model classes with instance attributes, methods, types, inheritance, etc.



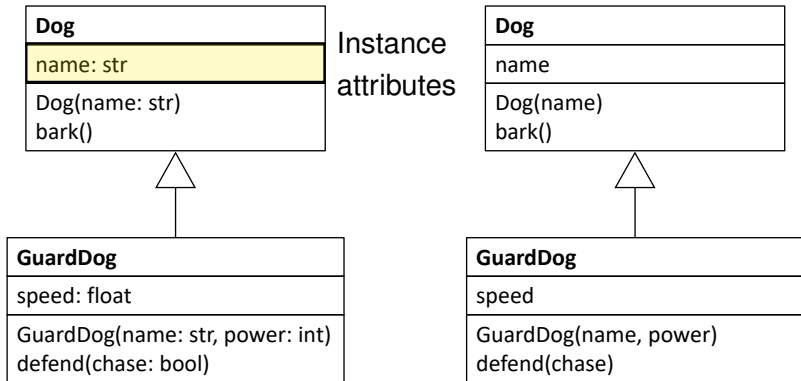
Graphical Class (Hierarchy) Notation

- UML (Unified Modeling Language) **class diagram**
- Very common. Allows to quickly model classes with instance attributes, methods, types, inheritance, etc.



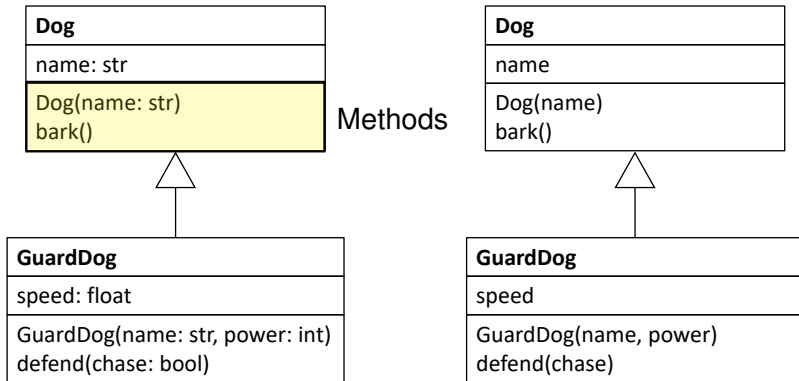
Graphical Class (Hierarchy) Notation

- UML (Unified Modeling Language) **class diagram**
- Very common. Allows to quickly model classes with instance attributes, methods, types, inheritance, etc.



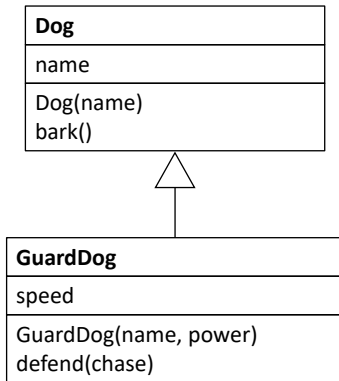
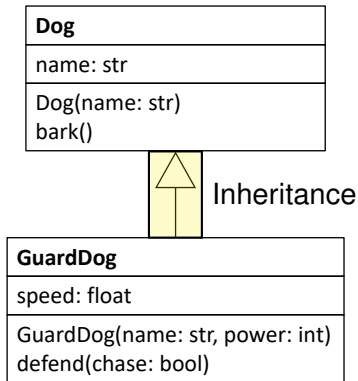
Graphical Class (Hierarchy) Notation

- UML (Unified Modeling Language) **class diagram**
- Very common. Allows to quickly model classes with instance attributes, methods, types, inheritance, etc.



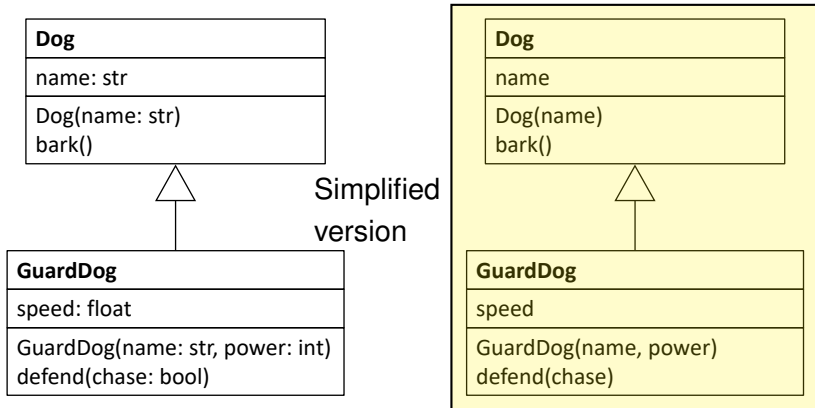
Graphical Class (Hierarchy) Notation

- UML (Unified Modeling Language) **class diagram**
- Very common. Allows to quickly model classes with instance attributes, methods, types, inheritance, etc.



Graphical Class (Hierarchy) Notation

- UML (Unified Modeling Language) **class diagram**
- Very common. Allows to quickly model classes with instance attributes, methods, types, inheritance, etc.



Useful Builtin Functionality

- **type(x)**: Returns the class/type of x

Useful Builtin Functionality

- **type(x)**: Returns the class/type of x
- **isinstance(x, y)**: Check if x is an instance of class/type y or of a subclass thereof
 - `isinstance(123, int) → True`
 - `isinstance(my_dog, Dog) → True`
 - `isinstance(my_guard_dog, Dog) → True`
 - `isinstance(my_dog, GuardDog) → False`

Useful Builtin Functionality

- **type(x)**: Returns the class/type of `x`
- **instance(x, y)**: Check if `x` is an instance of class/type `y` or of a subclass thereof
 - ☐ `instance(123, int) → True`
 - ☐ `instance(my_dog, Dog) → True`
 - ☐ `instance(my_guard_dog, Dog) → True`
 - ☐ `instance(my_dog, GuardDog) → False`
- **issubclass(x, y)**: Check if `x` is a class or subclass of class/type `y`
 - ☐ `issubclass(Dog, Dog) → True`
 - ☐ `issubclass(GuardDog, Dog) → True`
 - ☐ `issubclass(Dog, GuardDog) → False`