# Object Oriented Programming

Prof. Dr. Harald Gall

Dr. Carol Alexandru-Funakoshi

University of Zurich, Department of Informatics

# What is a Class? What is an Object?

Class

**Dog**

**Attributes / State**

name
breed
age
activity
is_hungry

**Functions**

setActivity(activity)
bark()

Object

**"Pelé"**
**"unknown"**
**0.5**
**"playing"**
**False**

Object

**"Rodney"**
**"Chihuahua"**
**9**
**"skateboarding"**
**False**

Object

**"Coco"**
**"Poodle"**
**2**
**"begging"**
**True**

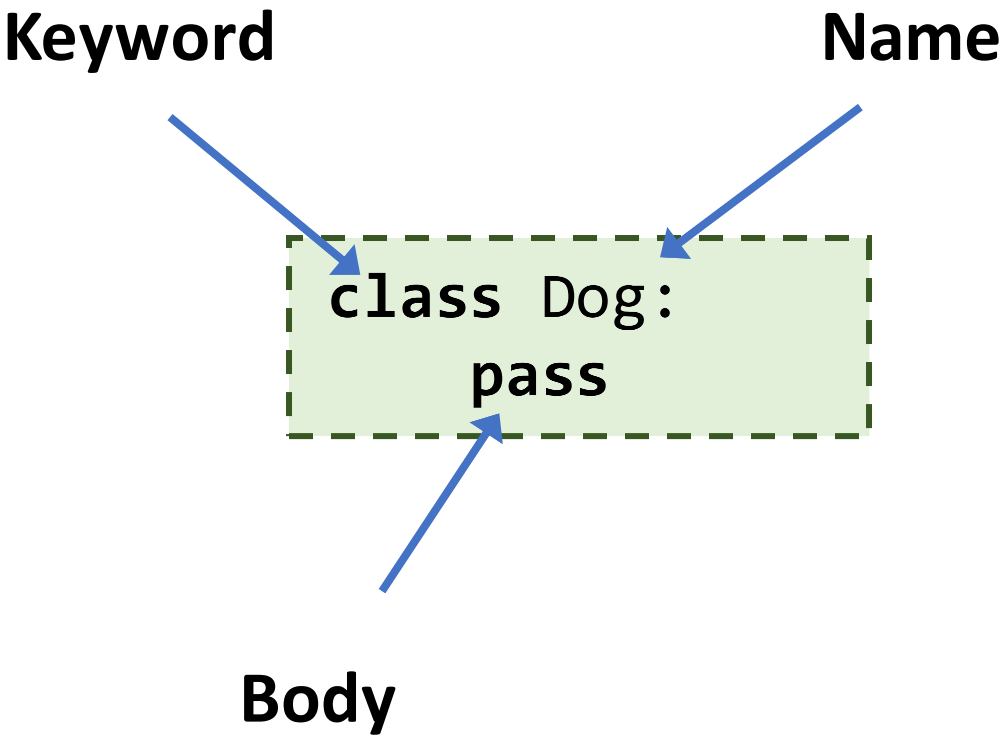
A class is like a blueprint – objects are concrete instances
Objects of the same class have the same attributes and functions

# Everything in Python is an Object!

| | |
|---|---|
| • Integers | 3 |
| • Strings | 'Hello, world!' |
| • Lists | [“dog”, “bird” , “cat”] |
| • Dictionaries | {’a': 1, ‘b’: 2, ‘c’: 3} |
| • Tuples | (1, 2) |
| • … | |

# Working with Classes in Python

# Class definition in Python (simplest case)



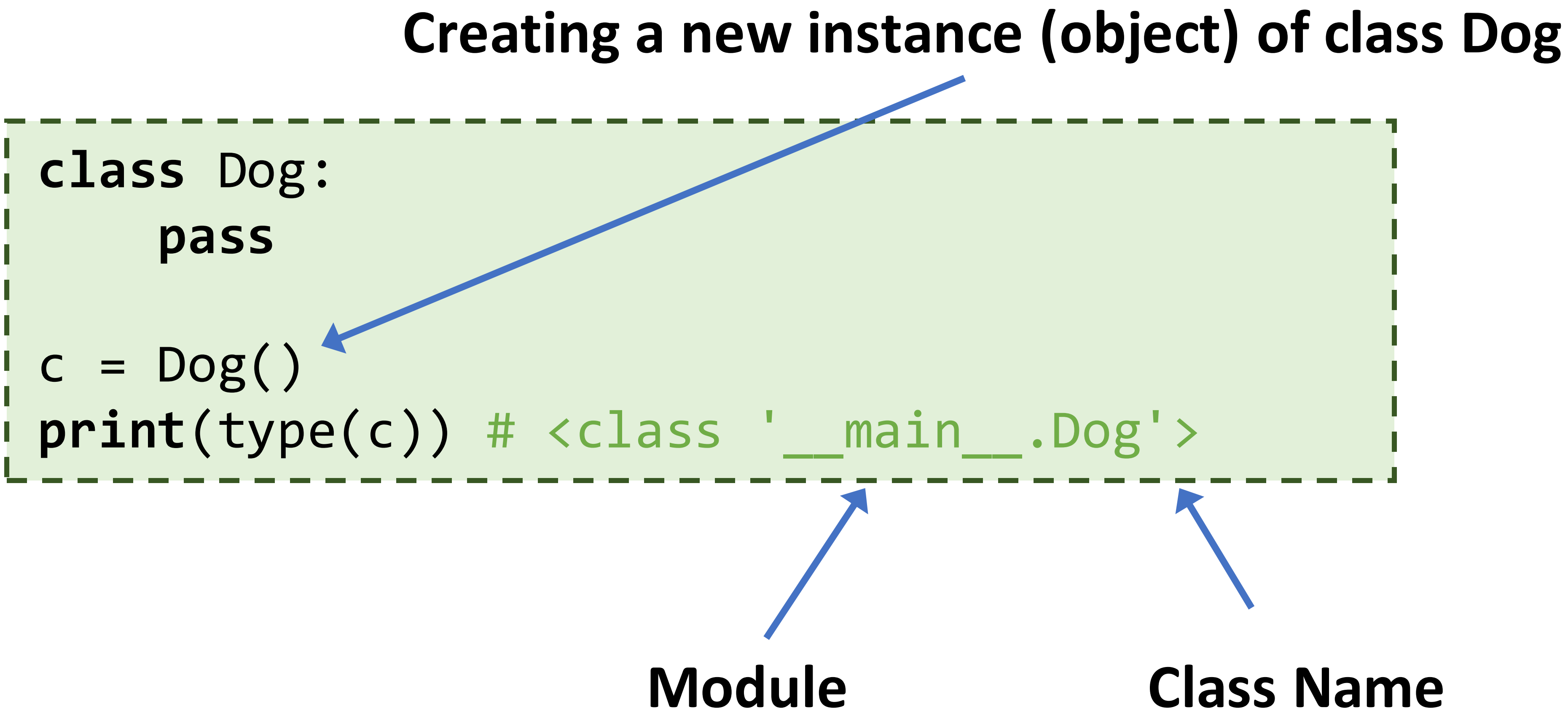
Keyword

Name

```
class Dog:
    pass
```

Body

# The type of an object is its class



Creating a new instance (object) of class Dog

```
class Dog:
    pass

c = Dog()
print(type(c)) # <class '__main__.Dog'>
```

Module

Class Name

# Using a constructor to define data attributes

- When creating an object (a.k.a. instantiation, construction), the `__init__` function is called. This is used to:
  - Define data attributes upon object creation
  - Run any other code that needs to be run upon object creation
- The first parameter, `self`, points to the instance
- Data attributes are also called instance variables

```
class Dog:
    def __init__(self, name, breed, age, is_hungry=False):
        self.name = name
        self.breed = breed
        self.age = age
        self.activity = "idle" if age > 3 else "playing"
        self.is_hungry = is_hungry
```

# Instantiation and access to data attributes

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

c1 = Dog("Rodney", "Chihuahua")
print(c1.name) # prints "Rodney"
```

The `self` parameter is provided *implicitly* by Python

Use the **dot notation** to access data attributes

# Example: 2 objects of the same type/class

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

c1 = Dog("Rodney", "Chihuahua")
c2 = Dog("Coco", "Poodle")

print(c1.name) # prints "Rodney"
print(c2.breed) # prints "Poodle"
```

# Class variables (not instance variables!)

```
class Dog:
    id_seq = 0
    def __init__(self, name):
        self.name = name
        self.id = Dog.id_seq
        Dog.id_seq += 1

d = Dog("Struppi")
print(d.id)           # 0
d = Dog("Lassie")
print(d.id)           # 1
d = Dog("Fido")
print(d.id)           # 2
print(Dog.id_seq)     # 3
```

This attribute belongs to the **class**, not any of its instances!

The class variable is accessible via the **class name**, not the object!

`self.foo` -> **instance** attributes, object-specific

`ClassName.foo` -> exists only once, belongs to the **class**

# "Methods" Are Functions Defined In A Class

```
class Dog:
    def __init__(self, age):
        self.age = age
        self.is_hungry = False

    def bark(self):
        sound = "bork" if self.age > 1 else "bark"
        if self.is_hungry: sound =  "Woof, woof"
        return f"{sound}!!!"

d = Dog(0.5)
d.bark()     # bark!!!
```

Methods also have `self` as the first parameter

Methods can access data attributes

Use the **dot notation** to access methods. Call() them like any function

# "Static" Methods

- Functions that are defined inside the class but don't access any of its instance variables (therefore do not need `self`)
- they are defined using the **@staticmethod** annotation

```
class Dog:
    def __init__(self, age):
        self.age = age
        self.is_hungry = False

    @staticmethod
    def eat(item):
        return f"I ate a {item}, and it was delicious!"

a = Dog(3)
b = Dog(1)
a.eat("steak") # "I ate a steak, and it was delicious!"
b.eat("shoe")  # "I ate a shoe, and it was delicious!"
```

- Nothing to do with any particular *instance*!
- The first parameter is **not** an implicit `self`, like with normal methods!
- Use static methods for any functionality that is invariant with the instance

# Instance variables can be made "private"

- By default, instance variables in Python are *publicly* accessible
- If instance variables should only be used *privately*, they can be marked by starting their names with __ (double underscore)
- Private instance variables can only be accessed from *within* the class

```
class Dog:
    def __init__(self, name):
        self.name = name
        self.__thinking_of = ["treats", "cuddles"]
    def share_thoughts(self):
        return self.__thinking_of

o = Dog("Fido")
print(o.share_thoughts()) # prints ["treats", "cuddles"]
print(o.name)             # prints Fido
print(o.__thinking_of)    # AttributeError: 'Dog' object has
                            no attribute '__thinking_of'
```

# So what is self??

- self is a reference to the instance.

```
class Dog:
    def __init__(self, name, breed, age, is_hungry=False):
        self.name = name
        self.breed = breed
        self.age = age
        self.activity = "idle" if age > 3 else "playing"
        Self.is_hungry = is_hungry
    def bark(self): # This is a method
        sound = "bork" if self.age > 1 else "bark"
        if self.is_hungry:
            sound =  "Woof, woof"
        return f"{sound}!!!"
d = Dog("Struppi", "Foxterrier", 7, True)
print(d.bark())  # Woof, woof!!!
```

# Three Equivalent Ways of Invoking a Method

Example From Last Slide

```
d = Dog(1)    # create object
d.bark()      # this is the "normal" way of calling a method
```

d is the `self` parameter, implicitly!

```
Dog.bark(d)   # Calling the class method with object as parameter
```

d is the `self` parameter, explicitly!

```
m = Dog.bark # methods are objects, can be assigned to variables
m(d)          # m is a function, it can be called
```

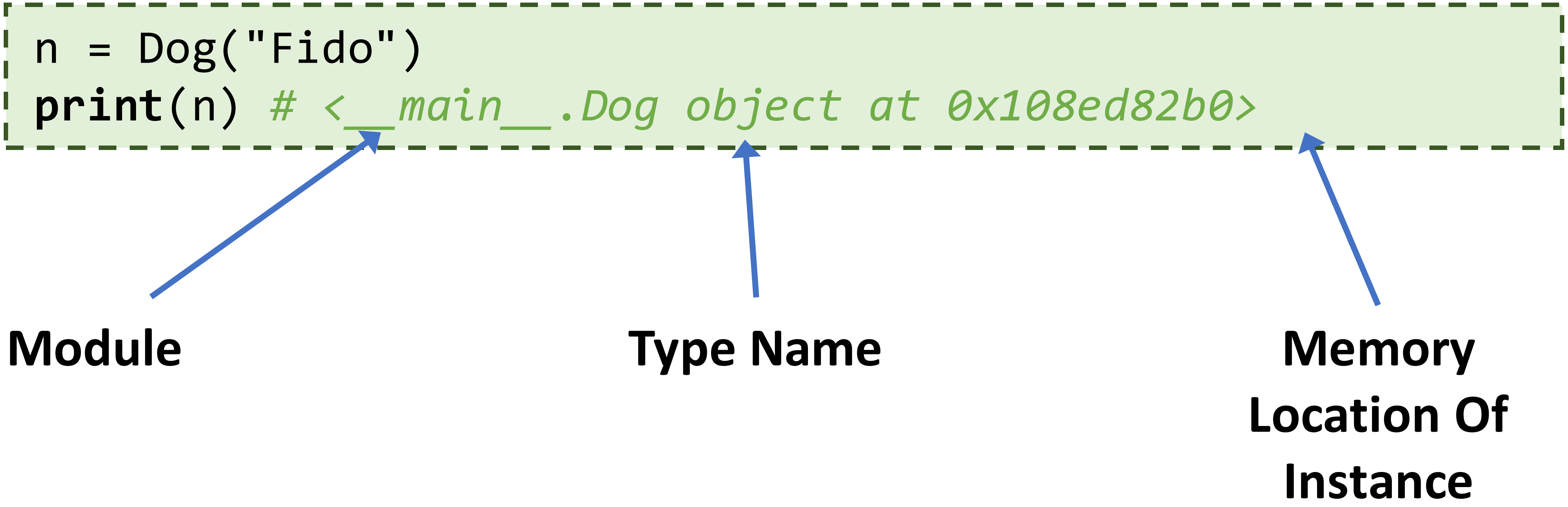
# Why "object-orientation"?

- "Object-orientation" is just one of many programming paradigms
- Arguments supporting this paradigm:
  - Our world is made up of "things", so it is a good model of the world
  - Practical considerations:
    - Facilitates testing (we can test the classes separately)
    - Encourages reuse (we can reuse classes defined by other people)
    - Users don’t need to know the inner workings of a class, they just rely on the public interface (= data- and function attributes,)
    - It is possible to hide information that users don’t need to understand
    - Class can Inherit Data and Behavior From Parent Class (Next Week)

# Special Methods

Quite Python-specific!

# By Default, Printing An Object Is Not Helpful



```
n = Dog("Fido")
print(n) # <__main__.Dog object at 0x108ed82b0>
```

Module

Type Name

Memory Location Of Instance

# Redefine __str__ to Improve Output

```python
class Dog:
    def __init__(self, name, breed, age):
        self.name = name
        self.breed = breed
        self.age = age

    def __str__(self):
        return f"{self.name} is a {self.age} year old {self.breed}"

d = Dog("Fido", "Poodle", 3)
print(d)  # Fido is a 3 year old Poodle
```

Goal of `__str__` is to create **human-readable** representation of an object.

# Why doesn’t it work in collections?

```
d = Dog("Fido", "Poodle", 3)
print(d)    # Fido is a 3 year old Poodle
print([d]) # [ <__main__.Dog object at 0x1140dd1d0> ]
```

Collections use `__repr__` to
print the contained objects.

# Define __repr__ Instead!

```
class Dog:
    def __init__(self, name, breed, age):
        self.name = name
        self.breed = breed
        self.age = age

    def __repr__(self):
        return f"<{self.name},{self.age},{self.breed}>"

d = Dog("Fido", "Poodle", 3)
print([d])  # [<Fido,3,Poodle>]
```

Goal of `__repr__` is to create **unambiguous representation** of an object.

# By Default, Different Objects Are Not Equal

```python
class Coin:
    def __init__(self, value):
        self.value = value

c1 = Coin(2)
c2 = Coin(2)

print(c1 == c2) # False
```

# Redefine __eq__ to Allow Equality Check

```
class Coin:
    def __init__(self, value):
        self.value = value
    def __eq__(self, other):
        return self.value == other.value


c1 = Coin(2)
c2 = Coin(2)

print(c1 == c2) # True
```

Compare data attributes that matter

# By Default, Classes Cannot Be *Hashed*

```
n = ComplexNumber(1, 2)  # represents "one half"

d = {}
d[n] = 0.5 # TypeError: unhashable type: 'ComplexNumber'
```

# Define __hash__ to Use Instances as Keys

```
class ComplexNumber(object):
    …
    def __hash__(self):
        return (101 * self.x) + self.y

d = { ComplexNumber(1, 2): 0.5 }

print(d) # { ComplexNumber(1,2): 0.5 }
```

__hash__ **must return the same value for objects that are** __eq__**.**

https://en.wikipedia.org/wiki/Hash_function

# Other Special Methods…

- __len__ (e.g., len(o))
- __add__ (e.g., a + b)
- __sub__ (e.g., a - b)
- …

https://docs.python.org/3/reference/datamodel.html

# Classes vs. "Data classes"

# Example of a Class

```
class Car:
    def __init__(self):
        self.__speed = 0

    def accelerate(self):
        self.__speed += 1

    def break(self):
        if self.__speed > 0:
            self.__speed -= 1

    def get_state(self):
        return “running” if \
               self.__speed > 0 \
               else “standing”
```

**Objects** hide their data behind abstractions and only expose functions that operate on that data.

Programmers just need to know how to use (and what to expect from) the public interface, but no implementation details.

# Example of a Data Class

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

    def get_age(self):
        return self.__age

p = Person(“Bob”, 27)
print(p.name)
print(p.get_age())
```

**Data objects** expose their data and have no meaningful functions.

“Getters”/”Setters” introduce indirection, but they also expose Implementation details

Data classes in Python 3 have dedicated syntax. If you need a data class, consider using it: [https://docs.python.org/3/library/dataclasses.html](https://docs.python.org/3/library/dataclasses.html)

# Summary

# You should be able to answer these questions

- What is a class? How to define one?
- What is the difference between a class and an instance?
- How do we declare and access attributes?
- How do we define and call methods?
- What is the difference between…
    - a function and a method?
    - a data structure and an object?
    - a class variable, an instance variable and a local variable?
    - a static and a non-static method?
- What is information hiding and why is it important?
- What is the purpose of __str__/__repr__, __eq__, and __hash__?

# Example 1: Dog

- Write a Dog class combining all the demonstrated features
- Extend the class as follows:
  - Add a data attribute sex (can be male or female)
  - Dogs should implement the + operator. If two dogs are of different sex and are at least 1 yeare old, they can have offspring. The offspring will be a new Dog with:
    - Name: None
    - Breed: parent's breed if they are the same, "Mutt" otherwise
    - Age: 0
  - Add an instance variable `tricks` which is an empty set
  - Add a method `learn_trick(t)` which will add a trick to the `tricks` set

# Example 2: SortedDict

- Write a clas SortedDict which behaves similarly to a Python dict, but where the entries are always sorted by their key. There should be:
  - A method add(k, v) which adds a new entry. When a new entry is added and the key already exists, overwrite the entry. Entries should be ordered after the addition is finished. Sorting should rely on the < and > operators.
  - A method remove(k) which deletes an entry. If the key is not in the SortedDict, throw an IndexError.
  - A method __str__ which prints the entries
  - A method + which returns a new SortedDict which combines entries from two SortedDicts. If a key exists in both, the second value should be used.
  - A method – which returns a new SortedDict with keys of the first SortedDict without those in the second.