Please participate in the evaluation via QR Code or via the following Link: https://www.eva.fau.de

Evaluation of Introduction to Computer Science

Password: QPSYH

Every semester our chair offers a training for the SAP certificate TS410 – Integrated Business Processes in SAP S4/Hana.

TS410 - Training

Discounted price:

- 300€ per Participant, who is enrolled at FAU

Registration and further information:

- https://www.is.rw.fau.eu/teaching/courses/ts410/

The course is a block seminar, which takes place at the beginning of every semester, before the lectures start.

In summer, the course takes place in the first two weeks of April.

Contact(s):

- Pepe Bellin (pepe.bellin@fau.de)
- Annina Liessmann (annina.liessmann@fau.de)

Friedrich-Alexander-Universität
School of Business,
Economics and Society

Week 7 – Object-oriented programming

Introduction to Computer Science | WS22/23

Chair of Digital Industrial Service Systems | Prof. Dr. Martin Matzner & Prof. Dr. Andreas Harth

After this lecture, students understand the foundational paradigms of object orientation and inheritance.

Learning objectives

- **Understand object-oriented thinking**
- **Know the core concepts of object-oriented programming**
- **Effectively use classes and objects to structure your code**
- **Be aware of the advantages object-oriented programming brings with it**
- **Write more reusable code**

Agenda

Programming Paradigms

Software- and application development mostly relies on imperative programming.

Major programming paradigms

Declarative
Programmers decide what to compute.

- Logic
- Event-driven
- Functional

Imperative
Programmers decide how to compute.

- Procedural
- Object-oriented

Most programming languages support multiple paradigms.

Only found at edge-cases for IntroCS

Major focus of IntroCS

Procedural programming lets programmers define a program's sequential procedure.

Procedural programming

Procedural programming

- Sequential programming
- Related to a human's perception
- Among the "fastest" customizable programming languages
- Only little additional software needed

Object-oriented programming facilitates a logic that claims: everything is an object.

Comparison of procedural and object-oriented programming

Procedural programming

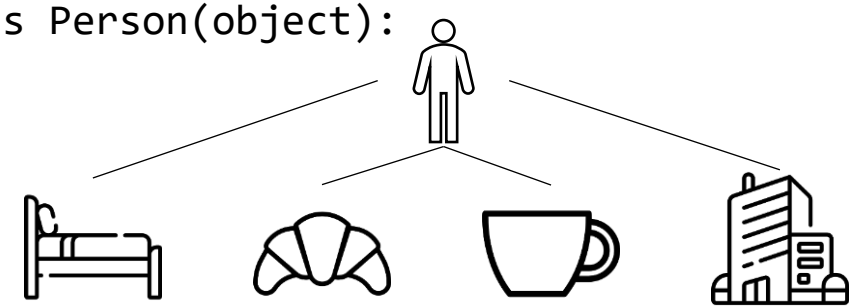
- Sequential programming
- Related to a human's perception
- Among the "fastest" programming languages.
- Procedures accomplish tasks

```
void daily_routine(Person *p)
```

Object-oriented programming

- **Functionality** is **bound** to objects
- A different way of perceiving an environment
- Everything is an object
- Objects accomplish tasks

```
class Person(object):
```



Simplicity, as in “close to real-world perceptions”, is a key advantage of OOP

Fundamentals of OOP

- **bundle data and functionality into objects** that operate with well-defined interfaces
- **divide-and-conquer** development
  - implement and test behavior of each object separately
  - increased modularity reduces complexity
- objects make it easy to **reuse** code
  - many Python modules define objects
  - each object has a separate environment (no collision on function names)
  - inheritance allows subclasses to use, redefine or extend a selected subset of a superclass’ behavior

Object-oriented programming in Python

You have used arrays and structs in C, and lists and dictionaries in Python to bundle data.

Bundling Data

Arrays in C

```
//rectangles

float width[3];
float height[3];
```

Lists / Dictionary in Python

```
rectangle = [width, height];
rectangle = {“width”: width, “height”: height}
```

Data Structures in C

```
typedef struct {

    float width;

    float height;

} rectangle;
```

Objects

Objects bundle data and functionality in a single scope.

Objects

- Python supports many kinds of data
- An object is an **instance** of a class

instance	class / type
5	int
“Hello“	str
a.append()	function
4.1495	float
…	…

State <-> Attributes

- Use `<object>.` to access any attribute of an object
- (Data) attributes are variables which define an object

Behavior <-> Methods

- Use `<object>.method` to call methods of an object
- Methods are **procedural attributes** that facilitate the interaction with an object

Everything in Python is an object. Objects consist of data attributes and methods.

Python objects

Every object

- has a **type** that is a certain **class**
- Use `type(obj)` to identify the class of an object
- can **create new objects** of some type
- can **manipulate objects**
- can **destroy objects**

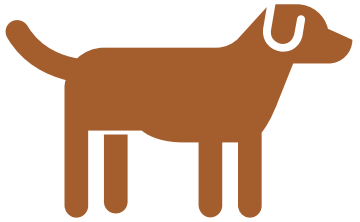
Objects are a **data abstraction** that captures…

An **internal representation**

- Through *data attributes*

An **interface** for interacting with the object

- Through methods (*object-related functions*)



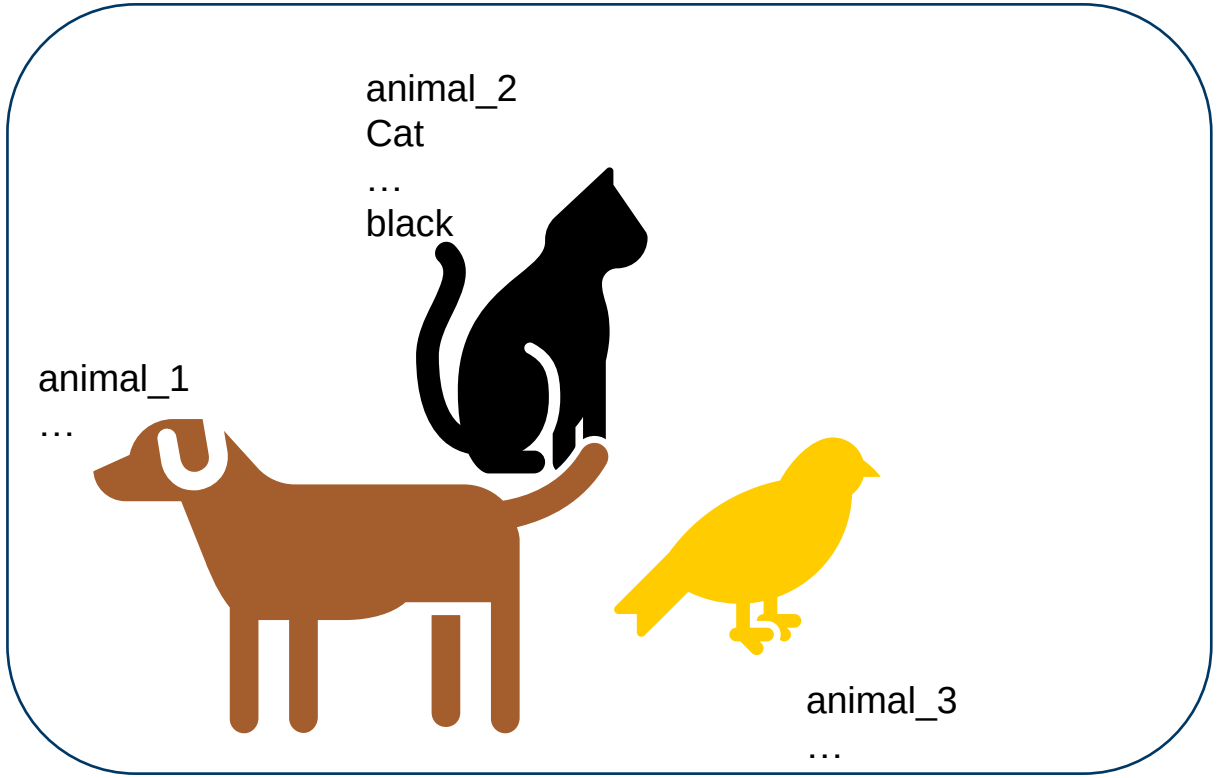
name = Bert
species = dog
color = brown

make_noise()
play_with()

Object-oriented programmers group similar objects into classes.

Classes

OOP allows us to **illustrate real-life** through **grouping** of objects of the **same type**



Animal
name species color
make_noise() play_with()

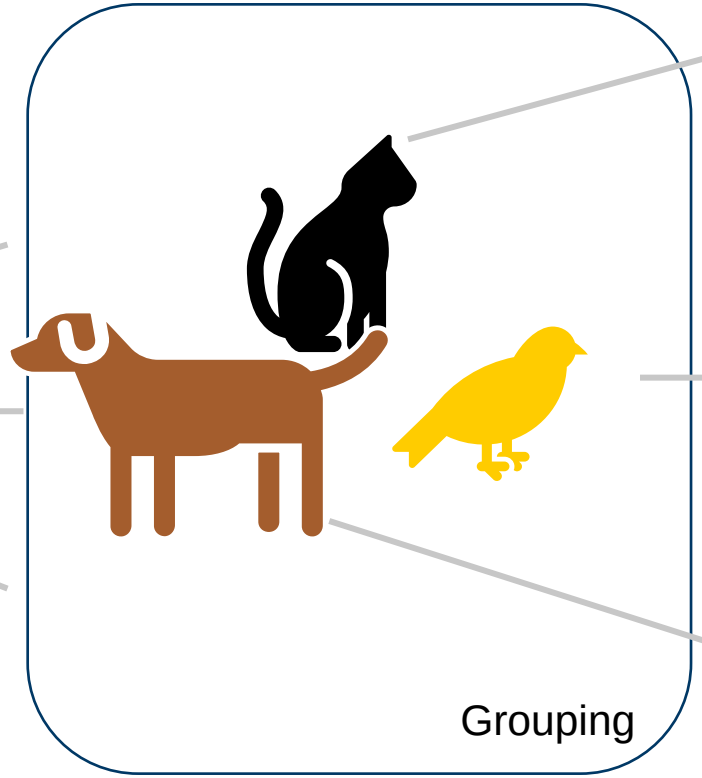
Classes are like blueprints for objects, so they abstract multiple different objects to a higher level.

Abstraction

Animal class

Animal
name species color
make_noise() play_with()

Blueprint



name = Lucy
species = cat
color = black

make_noise()
play_with()

name = Birdie
species = bird
color = yellow

make_noise()
play_with()

name = Bert
species = dog
color = brown

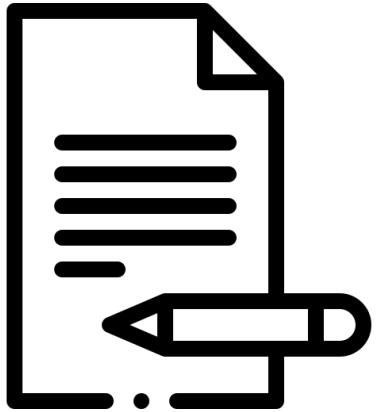
make_noise()
play_with()

Since classes are blueprints for objects, it’s important distinguish between creating and using classes.

Defining vs. using the class

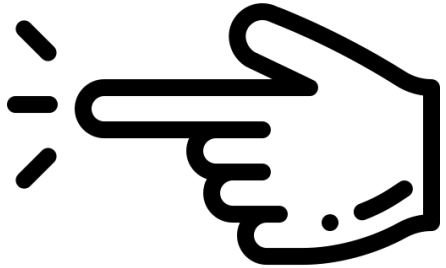
Creating the class involves

- Defining a class name
- Defining a constructor
- Defining class-scope variables



Classes support two kinds of **operations**

- **Instantiation** to create instances (a specific object) of a class
- **Attribute references** use the class name and dot-notation to access a class-scope variable



You can define your own classes, or types, using the keyworld class.

Defining a class

```
class Animal(object):
  # define attributes here
  pass
```

```
>>> animal_1 = Animal()
>>> animal_2 = Animal()
```

- `class` starts a class definition
- Code inside `class` is indented, similarly to `def`
- Use `pass` to create an “empty“ class
- Use `ClassName()` to create an object of class `ClassName`

As with structs, we can attach data to objects within their related class definitions in the constructor: `self.data = data`

Class anatomy - Data attributes

- Use **constructor,** which is a **special method** called **`__init__`** to initialize **data attributes**
- **What are attributes?**
  - Data that **belongs** to the class
    - Think of data as other objects that make up the class
    - *For example, an Animal can have a name, is of a species and has a color.*

```c
typedef struct
{
    char name[20];
    char species[30];
    char color[20];
}Animal;
```

```python
class Animal(object):
  def __init__(self, name, species, color):
    self.name = name
    self.species = species
    self.color = color
```

To instantiate an object, we call the constructor every time a new instance is created: `ClassName()`

Class anatomy – Constructor

Why do we add data to an object in a special method “Constructor”?

- **The Constructor** (`__init__()` method) is called every time an object is created, so an object of the class will always have executed the constructor, before it can be used by another object.

```
class Animal(object):
  def __init__(self, name, species, color):
    # Create the .name attribute and set it to name parameter
    self.name = name
    self.species = species
    self.color = color
    print("The __init__ method was called")

# __init__ is implicitly called
animal_3 = Animal("Birdie", "bird", "yellow")
```

```
The __init__ method was called
```

Calling the constructor creates one instance of a class. Parameters can be added as data attributes: `ClassName(params)`

Class Anatomy – Data attributes and Constructor

- Data attributes of an instance are called **instance variables**

```python
class Animal(object):
  def __init__(self, name, species, color):
    self.name = name
    self.species = species
    self.color = color

  def play_with(self, object):
    print(f"{self.name} the {self.species} is playing with {object}.")
```

```python
# create a new object of type Animal
# and pass name, species and color to __init__
animal_1 = Animal("Bert", "dog", "brown")

# use dot operator to access any attribute of animal_1
print(animal_1.species)
```

```
dog
```

Assign newly created objects to variables, otherwise you might lose access to an object.

Procedural programming defines functions to create reusable functionality, while OOP uses methods to achieve the same.

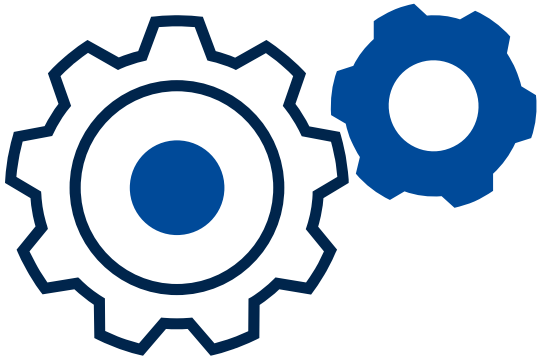
Class anatomy – methods

What are methods?

- **Procedural** attributes
- **Functions** that only work with the specified **class**
- Allow us to **interact** with the object
- Python always passes the **object** as the **first argument**
  - By convention `self` is used as the 1st argument in method definition
- The **“.” operator** is used to access any attribute
  - Data attribute of an object
  - Method of an object

```python
class Animal(object):
  def __init__(self, name, species, color):
    self.name = name
    self.species = species
    self.color = color

  def play_with(self, object):
    print(f"{self.name} the {self.species} is playing with {object}.")
```



Python methods are defined like functions, but they require self as their first parameter: `def method(self, params)`

Class Anatomy – Methods

- We already know that **methods** are those functions that only work **within** specified **classes** – so let‘s bring it all together:

```python
class Animal(object):
  def __init__(self, name, species, color):
    self.name = name
    self.species = species
    self.color = color

  def play_with(self, object):
    print(f"{self.name} the {self.species} is playing with {object}.")
```

```
>>> animal_2 = Animal("Lucy", "cat", "black")
>>> animal_2.play_with("a rag")

Lucy the cat is playing with a rag.
```

`animal_2.play_with(“a rag“)`

- We can use a certain method by using the **object‘s name** (`animal_2`) to specify the object we want to call the method on, followed by the **method** (`play_with`) and (if required) **parameters** (“`a rag`“)

The keyword self in methods' parameters within class definitions refers to the instance of a class, which executes the method.

Class anatomy – `self` in methods

- `self` is a placeholder for a particular object used in class definition
- Don't provide argument for `self`, Python takes care of that automatically
- Python will take care of `self` when method is called from an object:

```python
class Animal(object):
  def __init__(self, name, species, color):
    self.name = name
    self.species = species
    self.color = color

  def play_with(self, object):
    print(f"{self.name} the {self.species} is playing with {object}.")
```

```
>>> animal_1 = Animal("Bert", "dog", "brown")
>>> animal_1.play_with("a ball")

Bert the dog is playing with a ball.
```

`animal_1.play_with("a ball")`
is interpreted as
`Animal.play_with(animal_1, "a ball")`

Classes consist of their definition, data attributes which are defined in a constructor, and methods for binding functionality.

Class anatomy – Wrapping Up

- Methods are function definitions within a class, but they include `self` as the first argument
- Define data attributes of a class by assigning them in the constructor `__init__` method
- Refer to attributes of an instance of a class via `self.attr`

```
class MyClass():
    # method definition in class
    # first argument is self
    def my_method1(self, other_arguments...):
        # do things here

    def my_method2(self, my_attr):
        # attribute created by assignment
        self.my_attr = my_attr
        ...
```

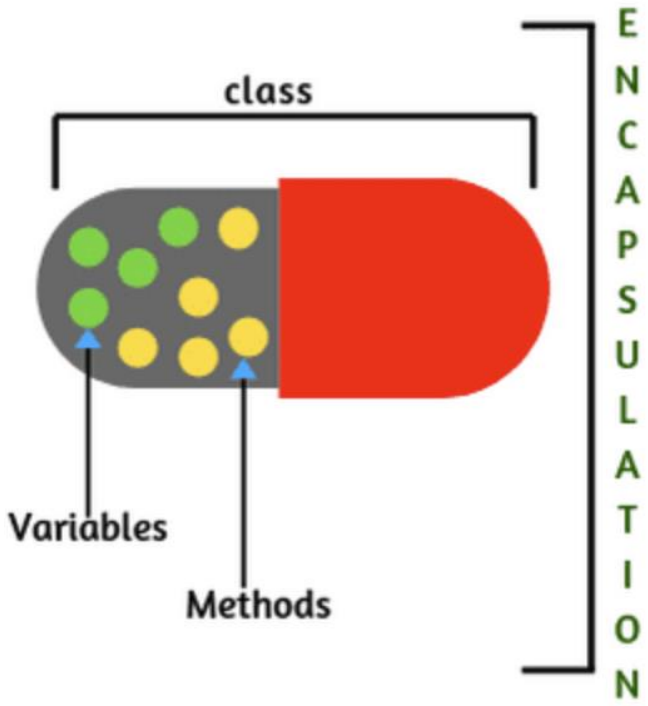
If you do not require any reference to self within a method, then you are not exploiting object-oriented programming capabilities. In such cases, you can create a function outside of a class instead of a method without breaking functionality.

Encapsulation refers to binding functionality and data to a single object.

Encapsulation

- **Encapsulation** lies at the very heart of OOP
- With encapsulation we mean **bundling** together **data attributes** and **methods** to operate on them:

```
>>> animal_2 = Animal("Lucy", "cat", "black")
>>> animal_2.play_with("a rag")
```



With encapsulation, programmers simplify reusing their code. All interfaces to class functionality (methods) are defined.

Even objects can be encapsulated within objects to increase usefulness of single objects.

Object Encapsulation

- Classes are great in bundling **data attributes** and **methods**
- This allows us to combine objects with other objects
- For example, if we think of an animal shelter, we want to have a possibility to drop off animals and mediate animals to a new home

```
class AnimalShelter(object):
  def __init__(self, max_animals):
    self.max_animals = max_animals
    self.animals = []
  def drop_off(self, animal):
    self.animals.append(animal)
  def take_home(self, animal):
    self.animals.remove(animal)
  def get_occupants(self):
    for animal in self.animals:
      print(f"{animal.name} the {animal.species}")
```

Let’s continue with the example from the slide before and do some work with our Animal Shelter.

Animal shelter example

Program:

```
# Initialize animals
animal_1 = Animal("Bert", "dog", "brown")
animal_2 = Animal("Lucy", "cat", "black")
# Initialize Animal Shelter
shelter = AnimalShelter(5)
shelter.drop_off(animal_1)
shelter.drop_off(animal_2)
shelter.get_occupants()
shelter.take_home(animal_2)
print("After successful placement:")
shelter.get_occupants()
```

Output:

```
Bert the dog
Lucy the cat
After successful placement:
Bert the dog
```

Use CamelCase for classes, lower_case for methods and attributes, and keep self as self.

Best practices

1. **Initialize an object and its attributes** in `__init__()`
2. **Naming**
   - `UpperCase` for class, `lower_case` for methods and attributes
3. **Keep** `self` as `self`

```
class MyClass():
    # This works but isn't recommended
    def my_method(person, attr):
        person.attr = attr
```

4. **Use docstrings**

```
class MyClass():
    """The class's docstring"""

    def my_method(self):
        """The method's docstring"""
```

Like structs in procedural programming, objects can not be printed automatically.

Printing Objects

```
class Animal():
  ...
animal_1 = Animal("Bert", "dog", "brown")
print(animal_1)
```

Output:

```
<__main__.Animal object at 0x7f0535281970>
```

- **Uninformative** print representation by default
- Define `show` **method** for specific class
- You choose what it does!

Printing an object out of the box looks like printing a pointer in C.
When printing a struct Person, the following line will **provide informative output:**

```
printf("Name: %s, Number: %i\n", p->name, p->number)
```

In Python, printing an object requires **printing the desired data attributes.**

To print an object in a more informative way, defining a method can help.

Printing objects – Custom method

```
class Animal():
  ...
  def show(self):
    print(self.name, self.species, self.color)

animal_1 = Animal("Bert", "dog", "brown")
animal_1.show()
```

Output:

```
Bert dog brown
```

- Instead of using `print`, we access our newly defined **`show` method** by using dot notation
- This allows us to **`customize`** the `print` representation of objects, though being **more informative**

Objects are instances of a class and a program can check whether an object is an instance of a particular class.

Classes and instances

```
animal_2 = Animal("Lucy", "cat", "black")
print(animal_2)
print(type(animal_2))
```

```
This animal is a black cat that listens to the name Lucy
<class '__main__.Animal'>
```

```
print(Animal)
print(type(Animal))
```

```
<class '__main__.Animal'>
<class 'type'>
```

```
print(isinstance(animal_2, Animal))
```

```
True
```

Let's first ask for the type of an object instance

- `print(animal_2)`
  - return of the `__str__` method
- `print(type(animal_2))`
  - the type of object animal_2 is a class Animal

This is due to

- `print(Animal)`
  - an Animal is a class
- `print(type(Animal))`
  - an Animal class is a type of object

Use `isinstance()` to check if an object is an Animal

Instantiating an object from a class using a constructor creates an object which encapsulates data and procedural attributes.

Fundamentals of OOP

- **Classes are blueprints of objects.**
- Whereas **instances are objects** that are created based on a class's **constructor.**
- Constructor: **def __init__(self[, …])**
- **Encapsulation refers to bundling data into objects** that share
  - **Data attributes** and
  - **Procedural attributes (methods)** that operate on those common attributes
- Creating our **own classes of objects** on top of Python's basic classes
  - **Self** in method definitions refers to a **specific instance** of a class

Abstraction and Information Hiding

Using “.”-notation to access data attributes of an object can result in unexpected usage and is a security issue.

Instances and dot notation

Instantiation allows us to create a new object (`animal_1`) of the class (`Animal`)

```
animal_1 = Animal("Bert", "dog", "brown")
```

Dot notation is used to access attributes (data and methods) however it is **better to use getter** and **setter** methods

```
animal_1.color
animal_1.get_color()
```

Providing getters and setters in your classes increases security, as you can preprocess in- and output of your objects.

Getter and setter methods

- In OOP **getter** and **setter** methods are used to access and edit data outside of classes:

```
class Animal(object):
  def __init__(self, name, species, color):
    self.name = name
    self.species = species
    self.color = color
  def get_color(self):
    return self.color
  def set_color(self, color):
    self.color = color
```

- **Getter** methods:
  - You can **abstract attribute names**, if you want to prevent users, to change or know them.
  - You can **aggregate information**. For instance, you could calculate a Persons age from his or her date of birth.
- **Setter** methods:
  - You can **check**, whether the correct **data type** is passed to your method.
  - You can **check semantic correctness**.
  - For instance: an ID is an Integer and cannot be a negative number.

Private, Protected and Public are the access modifiers in Python

Information hiding in OOP

Data hiding refers to access-management in-between different objects during runtime. For reasons of **security, safety and robustness**, programmers **may hide critical methods and attributes from external objects.**

In general, there are three access modifiers for attributes/properties:

- **Private:** indicated by a double underscore `self.__attribute`
  - Private attributes **cannot** be accessed from outside a class.
- **Protected**: Indicated by a single underscore `self._attribute`
  - Protected attributes should not be accessed from outside a class, other than sub-classes
  - Note that Python only sets this as convention, so it's more an indicator
- **Public**: Indicated by the absence of an underscore: `self.attribute`
  - Public attributes are always accessible

OOP allows for data abstraction by using access modifiers and getter and setter methods.

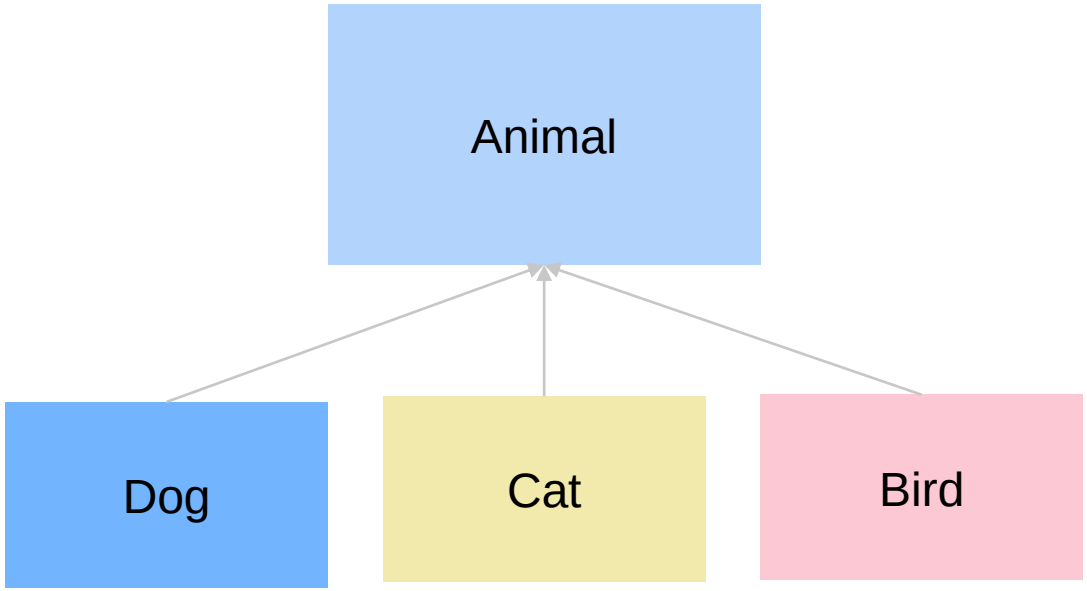
Abstraction and Information hiding

- It’s best practice to **use getter and setter methods to modify and retrieve data attributes.**
  - This way you can help other programmers to use your objects correctly
- Most OOP languages provide **three access modifiers**.
  - Make attributes **private** that are **critical to functionality**
  - Make attributes **protected**, that external users should **handle with care**
  - Make attributes **public**, that are **required for interaction**.

Inheritance

In real-life we often think of objects belonging together and being organized in hierarchies.

Reusability through hierarchies



Parent class (**superclass**)

Child class (**subclass**)

- **Inherits** all data and behaviors of parent class
- **(Can) Add** further **attributes**
- **(Can) Add** further **methods**
- **(Can) Override existing attributes / methods**

Inheritance provides a convenient mechanism for building groups of related abstractions.

Common properties amongst different types

- As you already know, types `list` and `str` each have `len` functions that mean the **same thing**

Inheritance – OOP

- Remember that **classes** are used to implement data abstractions
- **Inheritance** allows you to create a type hierarchy in which each type inherits types from above it in the hierarchy
- The class *`object`* is always at the top of hierarchy
  - in Python everything that exists at runtime is an object
  - Because `Animal` inherits all attributes of objects, programs can bind a variable to an `Animal`, `append` an `Animal` to a `list`, etc.

Inheritance provides a convenient mechanism for building groups of related abstractions.

Implementation of parent class

```
class Animal():
  def __init__(self, name, species, color):
    self.name = name
    self.species = species
    self.color = color
  def make_noise():
    print("I don't know which noise I make")
```

Everything is an object in Python, so `Animal` inherits all the properties of objects

But what does that mean?

- class *`object`* implements basic operations in Python, like binding variables, etc.

Through the concept of inheritance, we can easily reuse the attributes of a parent class.

Implementation of children classes

```
class Dog(Animal):
  def __init__(self, name, color, age):
    super().__init__(name, "dog", color)
    self.age = age
  def make_noise(self):
    print("Wuff")

class Cat(Animal):
  def __init__(self, name, color, age):
    super().__init__(name, "cat", color)
    self.age = age
  def make_noise(self):
    print("Meow")

class Bird(Animal):
  def __init__(self, name, color, age):
    super().__init__(name, "bird", color)
    self.age = age
  def make_noise(self):
    print("Chirp")
```

Parent class is `Animal`

- Call `Animal` **constructor**
- Call Animal's `constructor` method
- **Add** new data attribute `age` to `Dog` which is a string containing the dog's age

Override Animal's `make_noise` method

You might have noticed that we called the constructor of our superclass by using super().__init__(…).

super.__init__()

```
class Animal():
  def __init__(self, name, species, color):
    self.name = name
    self.species = species
    self.color = color
  def make_noise():
    print("I don't know which noise I make")
```

```
class Dog(Animal):
  def __init__(self, name, color, age):
    super().__init__(name, "dog", color)
    self.age = age
  def make_noise(self):
    print("Wuff")
```

Typical use case for super

- In a class hierarchy with single inheritance, super can be used to **refer** to **parent class without naming** it explicitly
- This makes the code more **maintainable**
- `self` is **not** needed when working with `super()`
- `super().__init__(name, “dog”, color)` **equals** to `Animal.__init__(self, name, “dog”, color)`

In addition to what subclasses inherit they can add new attributes and override attributes of superclasses.

Extended functionalities of subclasses and class variables

Add new attributes

- `Dog` added the **instance** variables `age` and `dogID`
- The **instance variable** `self.dogID` is initialized using a **class variable** `tag`, that belongs to the class `Dog` rather than to instances of the class

Override attributes of superclass

- For example, `Dog` has overridden `__init__` and `make_noise`

```
class Dog(Animal):
  tag = 0

  def __init__(self, name, color, age):
    super().__init__(name, "dog", color)
    self.age = age
    self.dogID = Dog.tag
    Dog.tag += 1
  def make_noise(self):
    print("Wuff")
```

Inheritance is one of the key-concepts which make OOP so powerful. So, let's go through the process step-by-step once again.

Recap Inheritance OOP

`Dog.__init__` first **invokes** `Animal.__init__` by using `super().__init__` to initialize the **inherited instance variable** `self.name, self.species` and `self.color`

Then `self.dogID` is initialized, an **instance variable** that **instances** of `Dog` have but instances of `Animal` do not

- The **instance variable** `self.dogID` is initialized using a **class variable**, `tag`, that belongs to the class `Dog`, rather than to instances of the class
- When an instance of `Dog` is created, a new instance of `tag` is **not** created
- This allows `__init__` to ensure that each instance of `Dog` has a **unique** `ID`

```python
class Dog(Animal):
  tag = 0

  def __init__(self, name, color, age):
    super().__init__(name, "dog", color)
    self.age = age
    self.dogID = Dog.tag
    Dog.tag += 1
  def make_noise(self):
    print("Wuff")
```

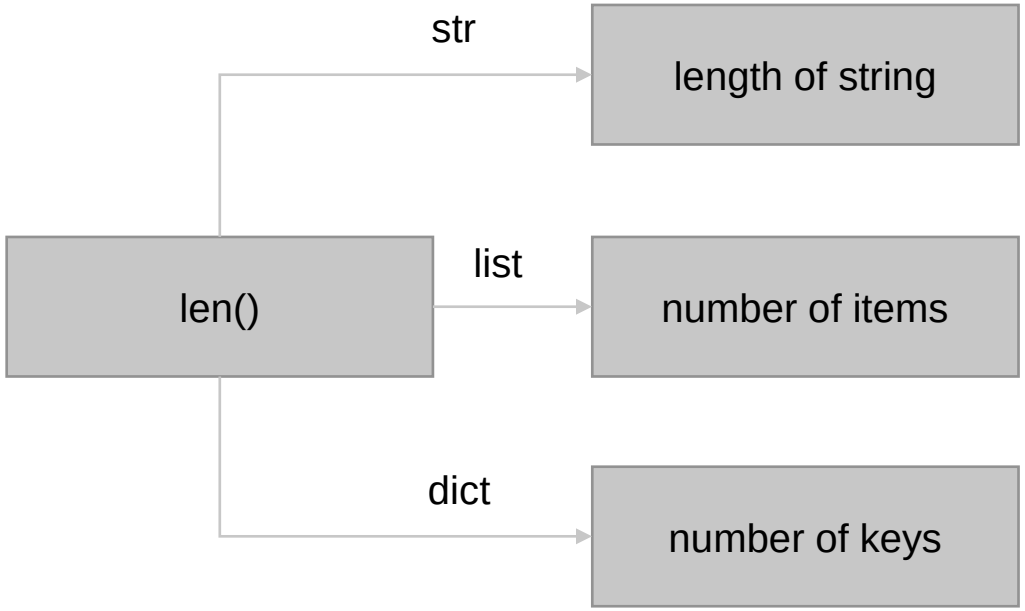
Polymorphism

Polymorphism in OOP is a powerful way to create programs and is often facilitated via inheritance.

Polymorphism in Python

Polymorphism is a very important concept, not only in OOP, but generally in programming

- It refers to the use of a **single type entity** (method, operator or object) to represent **different types** in different scenarios
- It makes programming easier and more intuitive



Dynamic typing and operator overloading are two approaches of polymorphism in Python

Dynamic typing and operator overloading

Dynamic typing

- Python utilizes **dynamic typing** (duck typing)
- No need to declare variable types before runtime
- We can use that to our advantage in regards of **flexibility**, **reusability** and **recyclability**

```
## variable a is assigned to a string
a = "hello"
print(type(a))

## variable a is assigned to an integer
a = 5
print(type(a))
```

Operator overloading

- Python objects allow us to **extend** the meaning of **default operators**, e.g., `'+'` or `'<'` by using `__add__` and `__lt__` respectively
- For example, `'+'` operator is used to add two numbers as well as to concatenate strings, which is achievable because **`'+'` operator is overloaded** by the `int` class and the `str` class

Python facilitates method overloading with default parameters and method overriding via inheritance.

Method overloading and method overriding

Method overloading

- In Python, Method overloading **does not work** as in other languages like Java or C++/#
- However, we can **set parameters** to **default** values:

```
def product(a, b, c=1):
    return a * b * c

# without defining c=1 as default
parameter, this line would throw an error
print(product(5, 10))
```

Method overriding

- Method overriding is an ability of every OOP programming language that allows **subclasses** to **override methods** of the according **superclasses** (Inheritance)

```
class Animal(object):
    def __init__(self):
        self.value = "Inside parent"
    def show(self):
        print(self.value)

class Cat(Animal):
    def __init__(self):
        self.value = "Inside children"
    def show(self):
        print(self.value)
```

To print out an Animal, we can override an object's special method __str__ method.

Lecture Add-On: Method overriding – __str__ method

```
class Animal(object):
  ...
  def __str__(self):
    return f"This animal is a {self.color} {self.species} that listens to the name {self.name}"
animal_2 = Animal("Lucy", "cat", "black")
print(animal_2.__str__())
print(animal_2)
```

`Output:`

```
This animal is a black cat that listens to the name Lucy
This animal is a black cat that listens to the name Lucy
```

- `def __str__(self):` name of a special method
- `str(self.width)` : `__str__` must **return** a **string**

Just like __str__ there are further special methods related to all objects, for instance to compare different object with each other.

Lecture Add-On: How to override operators.

- Like with print, we can **override** those operators with special methods
- Define them with **double scores** `__specialmethod__` before and after

```
__add__(self, other)
__sub__(self, other)
__eq__(self, other)
__lt__(self, other)
__len__(self)
__str__(self)
```

```
self + other
self - other
self == other
self < other
len(self)
print(self)
```

…and others which can be found here

The four core concepts of object-oriented programming are encapsulation, abstraction, inheritance, and polymorphism

Fundamentals of OOP

- **Bundle together objects** that share
  - **Common data attributes** and
  - **Procedural attributes (methods)** that operate on those common attributes
- Create our **own classes of objects** on top of Python's basic classes
- Use **abstraction** to make distinction between how to implement an object vs how to use an object
- Build **layers** of object abstractions that inherit behavior from other classes of objects
- Operators, methods and **objects can play different roles**, i.e., they can be **polymorphs**

Object-oriented programming bears the advantages: simplicity, encapsulation, reusability and modularity.

Advantages of OOP – SERM

Simplicity:

OOP **models real-world** objects in programs; thereby, the structure of a program becomes more **natural** and **intuitive**.

Encapsulation:

By nature, **objects comprise** both the **data** and **procedures** to take actions. Thus, if you have an implemented class, knowledge about its **internal functionality is not necessary**.

Recap OOP

Reusability:

Classes can often be **reused** in different contexts or programs, since their **functionality** is **independent** from a program itself.

Modularity:

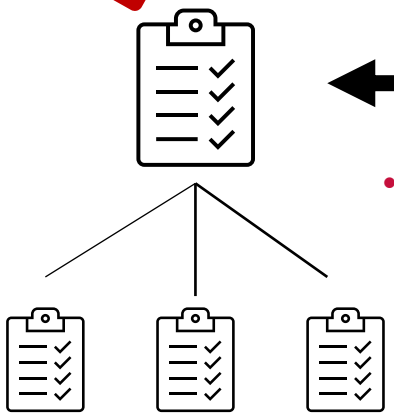
Polymorphism, **encapsulation** and **composition** make it simple to change a program. Everything you change is **bound** to the functionality of the **class** you modified.

These are the take aways from today

Recap OOP

- **Classes** facilitate code reusability

can be seen as **blueprints** for objects sharing the same attributes and behaviors



```
dog = Dog()
```

- Dinstinguish between Class creation and using a class

- Subclasses can **use**, **redefine** or **extend** enherited data or behavior

- **Encapsulation** means bounding data & functionality to objects

Attributes → States (dot-notation)
Methods → Behavior (dot-notation & arguments)

Recap OOP

- Use **type()** to get the class of an object

- If there is a python class for dogs, then **self** is one particular instance of dogs, say ***Bert***.

- **__init__(self,…)** stands for initialize and creates instances

- **super().__init__()** goes without the self and calls the superior class

After this lecture, students understand the foundational paradigms of object orientation and inheritance.

Learning Objectives (Revisited)

- **Understand object-oriented thinking**
- **Know the core concepts of object-oriented programming**
- **Effectively use classes and objects to structure your code**
- **Be aware of the advantages object-oriented programming brings with it**
- **Write more reusable code**

Friedrich-Alexander-Universität
School of Business,
Economics and Society

Chair of Digital Industrial Service Systems

Prof. Dr. Martin Matzner
Friedrich-Alexander-Universität Erlangen-Nürnberg
School of Business, Economics and Society | WiSo
wiso-is-kontakt@fau.de
twitter.com/ismama
www.is.rw.fau.eu