Programação e Algoritmia
--x--
Programming and Algorithms

1 – Object-Oriented Programming

Problems

- Having developed class Person, how to create classes for Student and Teacher?

  a12345= Student (“Mário Silva”,”LMat”, 12345)

  ajst=Teacher(“António Teixeira”,“DETI”, ...)

- Classes Circle, Rectangle, Square developed in Practical classes **replicate code**

Problem – duplicated code

- In software development, **duplicating code must be avoided**
- Programmer must minimize existence of similar blocks of code

Different ways to get classes we need

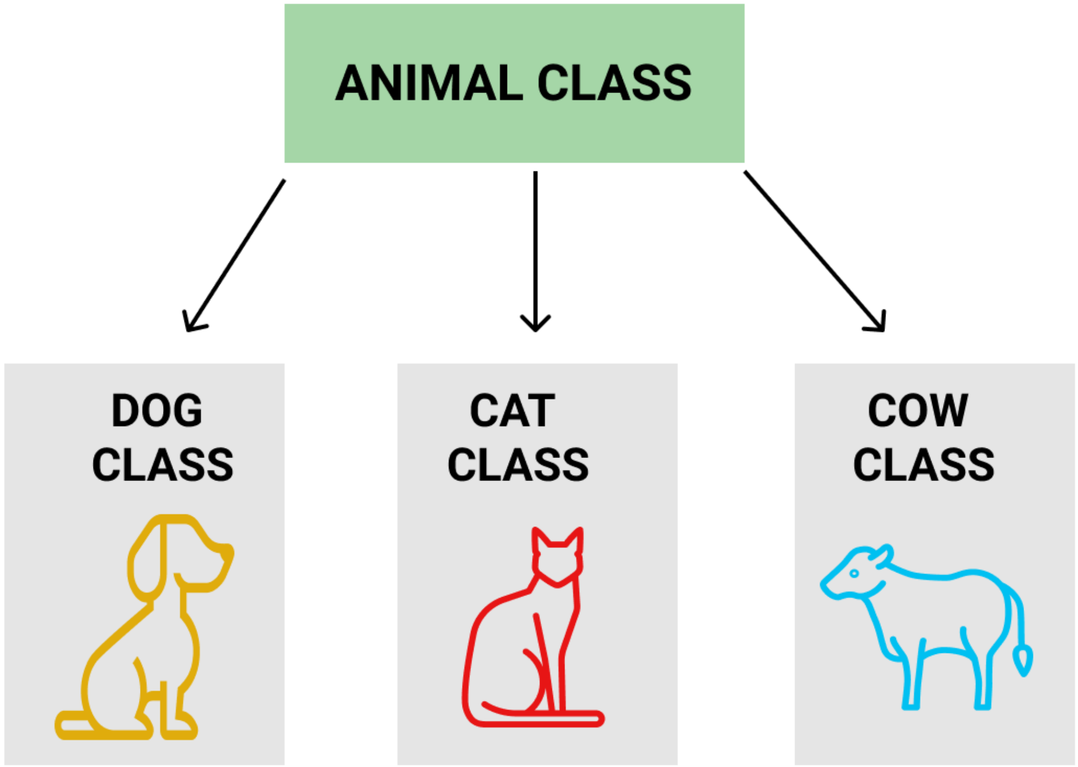
- Whenever we need a class, we can:

1. Use an existing class that meets the requirements
2. Write a new class "from scratch"
3. Reuse an existing class using **composition**
4. Reuse an existing class through **inheritance**

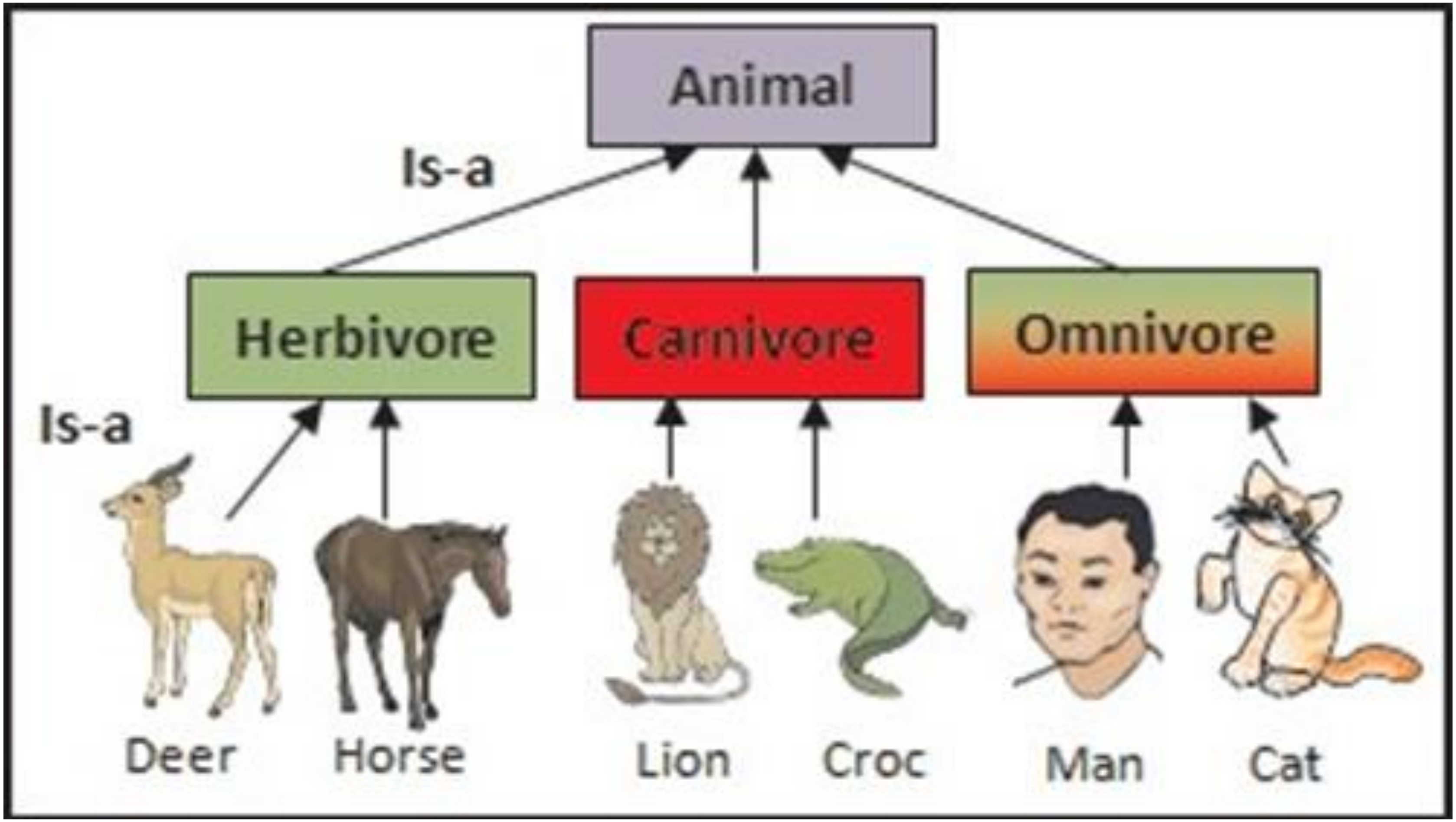
Coding example

Circle IS-A Figure, …

Inheritance in real world



Inheritance in the real world (2)



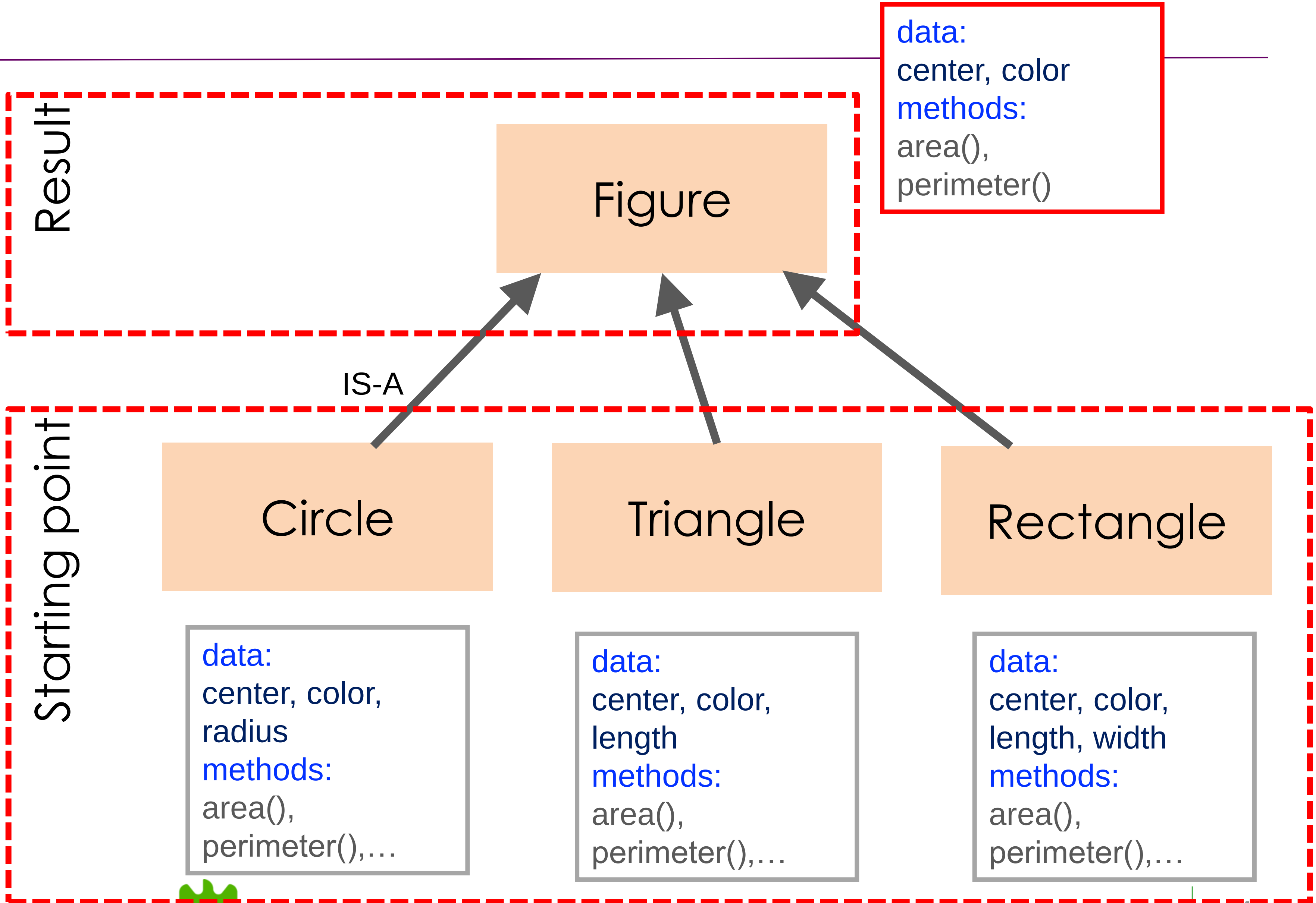
Inheritance

- The definition of inheritance relations is made by **generalization** of common data and behaviors (attributes and methods) in a superclass …
  - also called parent class

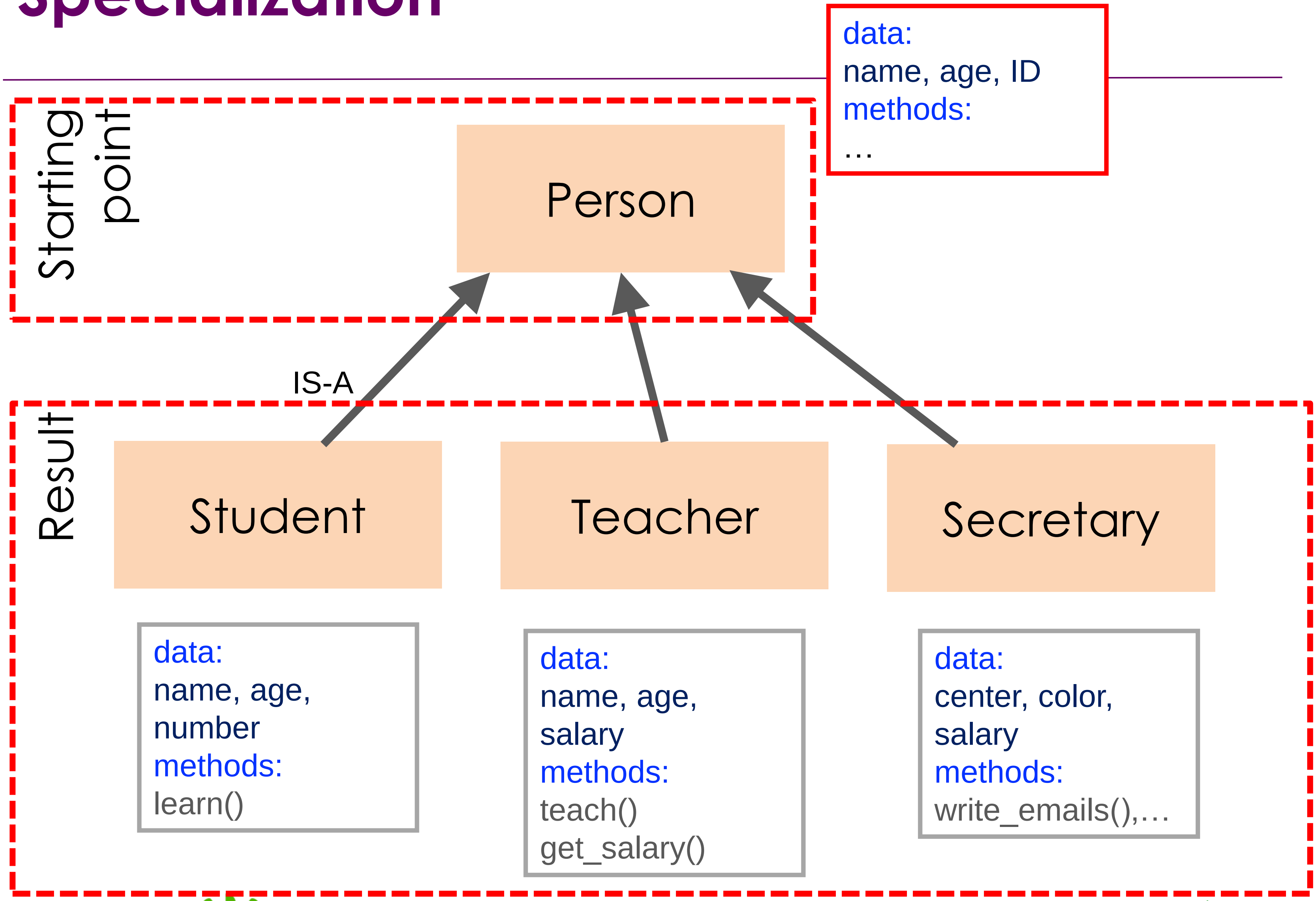
- and **specialization** of details in classes lower in the hierarchy, designated by subclasses
  - Or child classes

- It is the basis for other Object-Oriented Programming fundamental principle, polymorphism
  - Later in this class

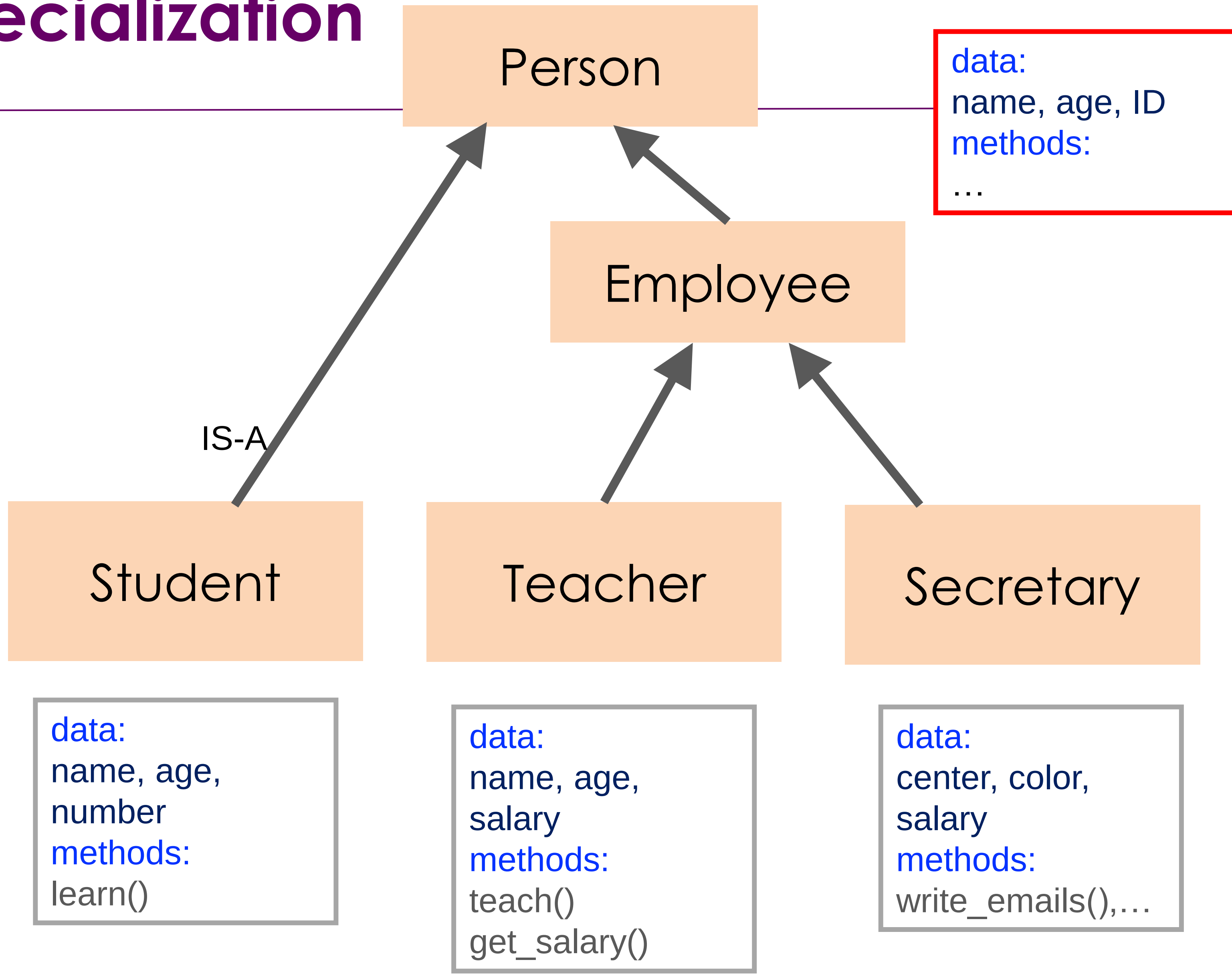
Generalization



Specialization



Specialization



Subclass and superclass

- A subclass inherits characteristics of its superclass
- A **subclass inherits attributes and methods** from the superclass
  - It can rewrite methods and data members of the superclass

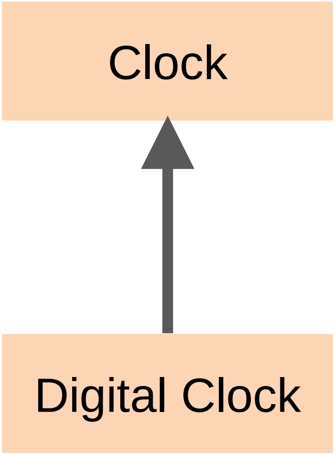
IS-A relation

- IS-A indicates specialization (inheritance)
  - i.e., when a class is a subclass of another class

- Examples:
  - Student **IS A** Person
  - A DigitalClock IS A Clock

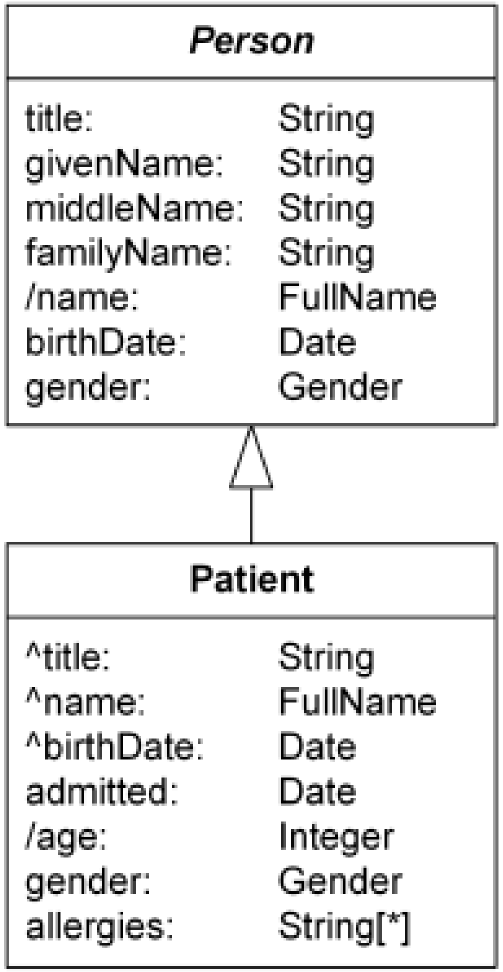
```
class Clock:
   ...


class DigitalClock (Clock)
   ...
```



UML

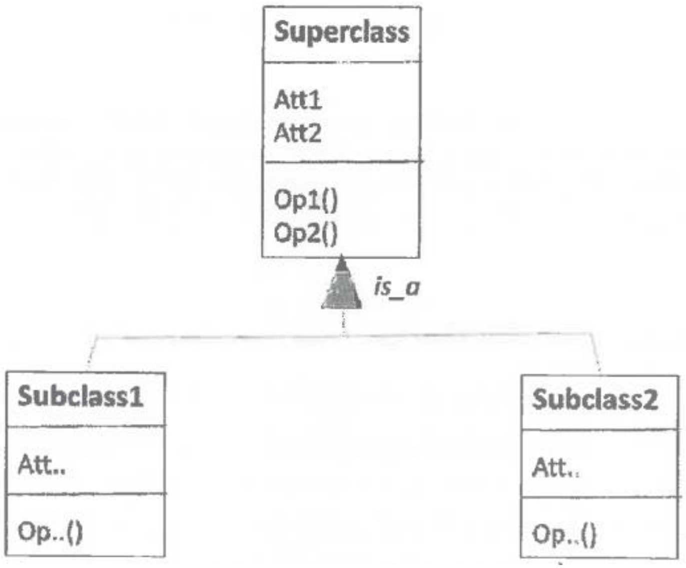
- In UML inheritance (IS-A relation) is represented by an **arrow** from the subclass to the superclass



Simple inheritance

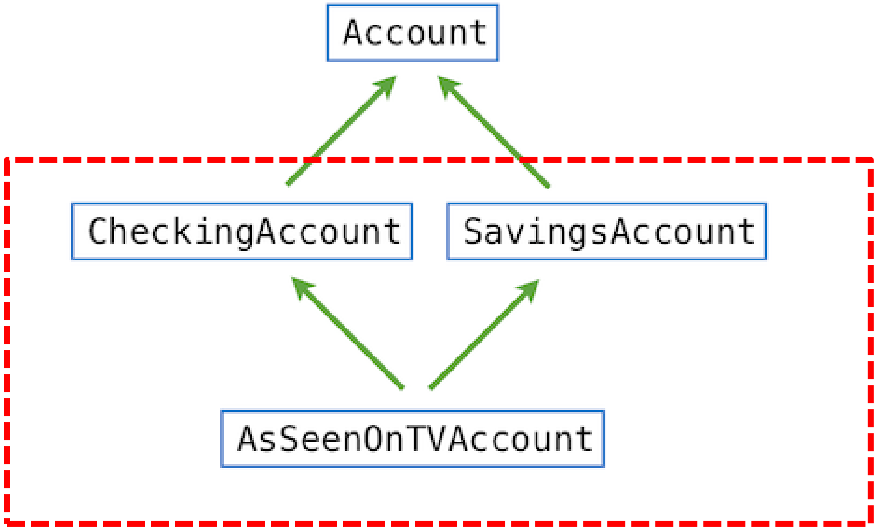
- Simple inheritance makes possible creating new classes that inherit properties and behavior of a single class, previously defined (the superclass)

- This mechanism transfers the characteristics of the superclass to derived subclasses



Multiple inheritance

- If a subclass inherits from more that one class, the mechanism is called multiple inheritance



- Multiple inheritance should be only applied in specific cases
  - Its use requires higher programming competences
  - Several experience programmers recommend not using it

Inheritance in Python

(Simple) Inheritance in Python

- The syntax to create derived classes by (simple) inheritance in Python is:

```
class SuperClass1:
    # specification of the class

class SubClass1(SuperClass1):
    # specification of the class

class SubClass2(SuperClass1):
    # specification of the class
```

Multiple inheritance in Python

```
class SuperClass1:
    # specification of the class

class SuperClass2:
    # specification of the class

class Subclass (SuperClass1, SuperClass2, …):
    # specification of the class
```

Multiple inheritance - Example

- Example

```python
class Father():
    def drive(self):
        print("Father drives his son to school")

class Mother():
    def cook(self):
        print("Mother loves to cook for her son")

class Son(Father, Mother):
    def love(self):
        print("I love my Parents")


c=Son()
c.drive()
c.cook()
c.love()
```

Object superclass

- Any time you define a new class in Python you apply inheritance
- All classes in Python have default **object** as their superclass
  - When you write

  class NewClass: is in fact class NewClass(object):

- If superclass is not explicitly specified, a class automatically inherits from objet
  - You can write class subclass(object) but not needed

object members (API)

__class__
__delattr__
__dir__
__doc__
__eq__
__format__
__ge__
__getattribute__
__gt__
__hash__
__init__
__le__
__lt__
__ne__
__new__
__reduce__
__reduce_ex__
__repr__
__setattr__
__sizeof__
__str__
__subclasshook__

__repr__ vs __str__

- Why 2 methods?

- **__str__** is tried by user-friendly displays
  - Such as print()
- The **__repr__** method should in principle return a string that could be used as executable code to recreate the object

- If no **__str__** is present, Python falls back on **__repr__**
  - Bot not vice versa

- More information:
  - https://stackoverflow.com/a/2626364/4244835

Methods and Inheritance

What happens to Methods when we use inheritance ?

Inheritance of Methods

- In inheritance, methods can be:

- **Inherited** / kept unchanged
  - Definition at superclass will be used
  - No need to define in derived class

- **Extended**
  - Have new added functionalities

- **Redefined**
  - Definition changed

Inherited methods

```
class Person:
    def __init__(self, name):
        self.__name = name
    def get_name(self):
        return self.__name
    def __str__(self):
        return("PERSON:")

class Student(Person):
    def __init__(self, name, nmec):
        super().__init__(name)
        self.__nmec = nmec
    def get_num(self):
        return self.__nmec

stu = Student("Andreia", 55678)
print (f"{stu} : {stu.get_name()}, {stu.get_num()}")

Output:
PERSON: : Andreia, 55678
```

Python starts searching **get_name()** in Student.

As it does not find it, continues search in superclass.

Extended methods

```python
class Person:
    def __init__(self, name):
        self.__name = name
    def get_name(self):
        return self.__name
    def __str__(self):
        return "PERSON:"

class Student(Person):
    def __init__(self, name, nmec):
        super().__init__(name)
        self.__nmec = nmec
    def get_num(self):
        return self.__nmec
    def __str__(self):
        return super().__str__()+" STUDENT"

stu = Student("Andreia", 55678)
print (f"{stu} : {stu.get_name()}, {stu.get_num()}")
```

```
Output:
PERSON: STUDENT : Andreia, 55678
```

Redefined (override) methods

```python
class Person:
    def __init__(self, name):
        self.__name = name
    def get_name(self):
        return self.__name
    def __str__(self):
        return("PERSON:")

class Student(Person):
    def __init__(self, name, nmec):
        super().__init__(name)
        self.__nmec = nmec
    def get_num(self):
        return self.__nmec
    def __str__(self):
        return("STUDENT:")

stu = Student("Andreia", 55678)
print (f"{stu} : {stu.get_name()}, {stu.get_num()}")
```

```
Output:
STUDENT: : Andreia, 55678
```

Override

- As shown in previous example, a class can rewrite (override) certain methods, to implement its specific needs
  - This process is call overriding

- Override is motivated by the need to comply with derived class specificities

- The syntax for override is exemplified at right

```
# define class (superclass)
class SuperClass1:
    def sMethod(self):
        print('... superclass method')

# define derived class (subclass)
class SubClass1(SuperClass1):
    def sMethod(self):
        print('... sublass method')

# main program
sup = SuperClass1()
sub = SubClass1()
# invoke method
sup.sMethod()
# invoke in sub
sub.sMethod()

Output:
... superclass method
... sublass method
```

Constructors (__init__)

- As other methods, can be kept, extended and overridden

- Quite often they are extended
  - Invoking super().__init__()
  - Followed by needed specific code for the derived class

Constructors

```
class Art:
    def __init__(self):
        print("Art constructor")

class Drawing(Art):
    def __init__(self):
        print("Drawing constructor")

class Cartoon(Drawing):
    def __init__(self):
        print("Cartoon constructor")

c = Cartoon()

Output:
Cartoon constructor
```

Contructors (cont.)

```
class Art:
    def __init__(self):
        #super().__init__()
        print("Art constructor")

class Drawing(Art):
    def __init__(self):
        super().__init__()
        print("Drawing constructor")

class Cartoon(Drawing):
    def __init__(self):
        super().__init__()
        print("Cartoon constructor")

c = Cartoon()
```

Output:
Art constructor
Drawing constructor
Cartoon constructor

Contructors - Alternative

```python
class Art:
    def __init__(self):
        #super().__init__()
        print("Art constructor")

class Drawing(Art):
    def __init__(self):
        Art.__init__(self)
        print("Drawing constructor")

class Cartoon(Drawing):
    def __init__(self):
        Drawing.__init__(self)
        print("Cartoon constructor")

c = Cartoon()
```

Output:
Art constructor
Drawing constructor
Cartoon constructor

Class Relations

Relations between Classes

- Part of the class modeling process consists of:
  - Identify candidates for being classes
  - **Identify relationships** between these

- Relations:
  - IS-A (inheritance)
  - HAS-A

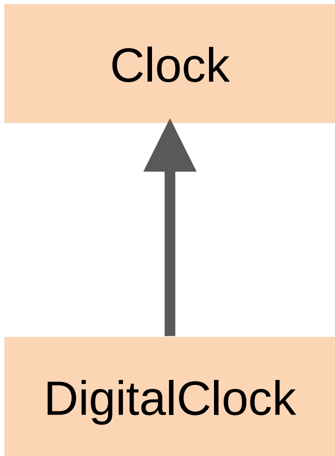
Inheritance (IS-A)

- IS-A indicates specialization (inheritance) that is, when a class is a subtype of another class.

- For instance:
  - A DigitalClock IS-A Clock.

```
class Clock:
    # class definition

class DigitalClock (Clock):
    # class definition
```



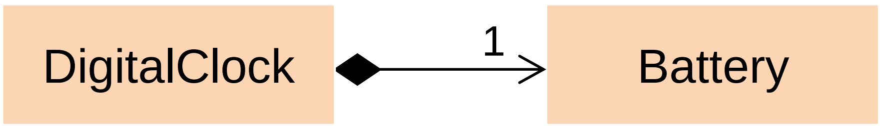
Composition (HAS-A)

- HAS-A indicates that one class is composed of objects from another class.

- For instance:
  - Forest contains (HAS-A) Tree.
  - A DigitalClock contains (HAS-A) a battery

```
class Battery_
    # class definition

class DigitalClock(Clock):
    b = Battery()
    # additional code
```



Questions

- What are the relationships between:
  - Worker, Driver, Salesman, Administrative and Accountant
  - Square, Triangle, and Pentagon
  - Teacher, Student and Employee
  - Bus, Vehicle, Wheel, Engine, Tire

Abstract Classes

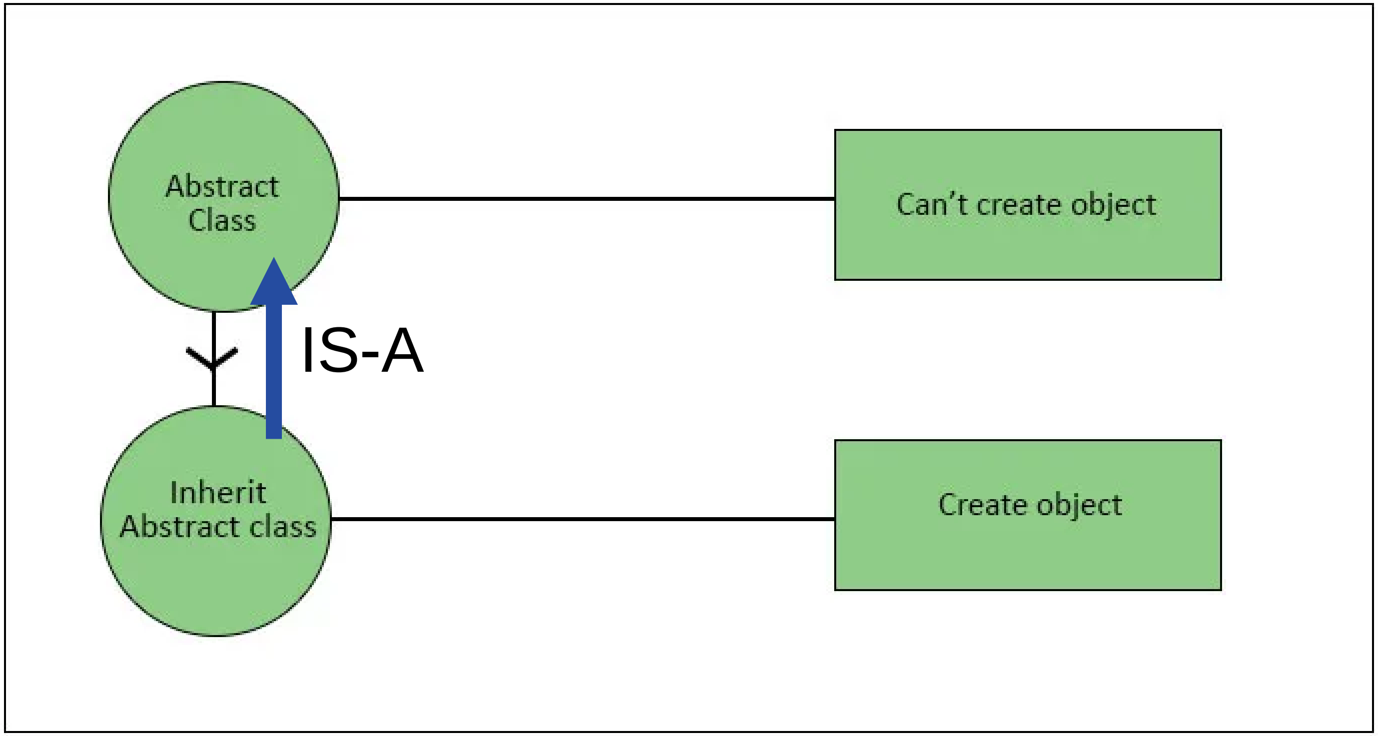
Abstract classes (and methods)

- A class is called an Abstract class if it **contains one or more abstract methods**

- An abstract method is a method that is declared, but contains **no implementation**

Abstract classes

- Abstract classes **can not be instantiated**
  - and its abstract methods must be implemented by its subclasses



Abstract classes

- Abstract classes are a way to ensure a certain level of code quality
  - because they enforce certain standards and can reduce the amount of duplicate code that we write
- They establish a connection between the base class and the concrete class
- They define a generalized structure of methods without its complete implementation
  - making the life of the programmer easy by abstracting the background process and making them focus only on important points.

Importance of Abstract Class

- It provides the default functionality of the base classes

- It defines a common API for the set of subclasses
  - useful where a third party is providing plugins in an application

- Helpful in large code
  - when remembering many classes is difficult

Simple implementation (of abstract methods)

```
class  Figure:

   def area(self):
       raise NotImplementedError

   def  perimeter(self):
       raise NotImplementedError



f = Figure()

f.area()
```

Implementation with module abc

- To consider any class as an abstract class, the class must inherit ABC metaclass from the python built-in abc module.
  - abc module imports the ABC metaclass
  - abc stands for "abstract base classes"

```
from abc import ABC
Class Figure(ABC):
```

- Abstract methods are the methods that are declared without any implementations.

```
from abc import ABC, abstractmethod
class Figure(ABC):
    @abstractmethod
    def draw(self):
            #empty body
            pass
```

Is a decorator

Decorators are a very powerful and useful tool in Python since it allows programmers to modify the behavior of function or class

Recommendations

Inheritance – Best Practices

- Program for the interface and not for implementation
  - Interface is the set of methods made available by the class
- Look for aspects common to multiple classes and promote them to a base class
- Use inheritance judiciously
  - whenever possible favor composition

Identifying Inheritance

- Typical signs that two classes have an inheritance relationship
  - Have common aspects (data, behavior)
  - Also have different aspects
  - One is a specialization of the other

- Examples:
  - Cat is a Mammal
  - Circle is a Figure
  - Water is a Drink

Polymorphism

Polymorphism

- The word polymorphism **means having many forms**
  - The word “poly” means many and “morphs” means forms
- Quality of what can take different forms or what occurs in different ways
- A person at the same time can have different characteristics.
  - A man at the same time can be a father, a husband, an employee.
  - The same person possesses different behavior in different situations.
  - This is called polymorphism.

Polymorphism in programming

- It is possible to handle in the same way instances of different classes
- It makes **possible to send a message to an object without knowing previously its type**

Examples:

```
>>> lista = [-5,10,15,20,25]
>>> len (lista)
5
>>> len("lista")
5

>>> "lista" [:3]
"lis"
>>> lista[:3]
[-5,10,15]
```

- Method **len** without knowing in advance the type of the argument has the same behavior (returning the number of elements)

Polymorphism and inheritance

- An object of class A can be used in the place of one object of class B if A is a subclass of B

- To access (in Python) to an attribute or method is enough that the object has that attribute or method
  - Guaranteed as subclasses inherit attributes and methods from superclasses

Example

```python
import random
class Figure:
    def draw(self):
        print("I don't know how to draw myself !")

class Circle(Figure):
    def draw(self):
        print ("Circle.draw")

class Square(Figure):
    def draw(self):
        print ("Square.draw")

class Figures:

    def randShape():
        rn = random.randrange(0,2)
        if rn == 0:
            return Circle()
        elif rn ==1:
            return Square()

def main():
    figures = []
    for i in range(9):
        figures.append(Figures.randShape())

    for fig in figures:
        fig.draw()

if __name__ == '__main__':
    main()
```

```
Square.draw
Circle.draw
Square.draw
Circle.draw
Square.draw
Square.draw
Circle.draw
Square.draw
Square.draw
```

```
Circle.draw
Circle.draw
Square.draw
Circle.draw
Circle.draw
Circle.draw
Circle.draw
Circle.draw
Square.draw
```

Example (cont)

```python
import random
class Figure:

    def draw(self):
        print("I don't know how to draw myself !")

class Circle(Figure):
    def draw(self):
        print ("Circle.draw")

class Square(Figure):
    def draw(self):
        print ("Square.draw")
class Triangle(Figure):
    pass
class Figures:
    def randShape():
        rn = random.randrange(0,3)
        if rn == 0:
            return Circle()
        elif rn ==1:
            return Square()
        elif rn == 2:
            return Triangle()

def main():
    figures = []
    for i in range(9):
        figures.append(Figures.randShape())
    for fig in figures:
        fig.draw()

if __name__ == '__main__':
    main()
```

```
Square.draw
Square.draw
I don't know how to draw myself !
I don't know how to draw myself !
I don't know how to draw myself !
Square.draw
Square.draw
Square.draw
Circle.draw
```

Polymorphism in Python

- In Python polymorphism is more flexible than in other languages (e.g., Java)
  - Appearing in many situations

- It is possible to use **Duck typing**
  - an application of the duck test to determine whether an object can be used for a particular purpose

"If it walks like a duck and it quacks like a duck, then it must be a duck"

Duck Typing Example

```
class Duck:
    def swim(self):
        print("Duck swimming")

    def fly(self):
        print("Duck flying")

class Whale:
    def swim(self):
        print("Whale swimming")

for animal in [Duck(), Whale()]:
    animal.swim()
    animal.fly()
```

```
Output:
Duck swimming
Duck flying
Whale swimming
AttributeError: 'Whale' object has no
attribute 'fly'
```

- This is a simple example demonstrates how any object may be used in any context, up until it is used in a way that it does not support
- if we assume everything that can swim is a duck because ducks can swim, we will consider a whale to be a duck …
- but, if we also assume it must be capable of flying, the whale won't be a duck.

Data Structures