[FIM] FONDAMENTI DI INFORMATICA
per medicina e chirurgia high tech

L10: Inheritance

Dott. Giorgio De Magistris

demagistris@diag.uniroma1.it

Corso di Laurea in Medicina e Chirurgia High Tech

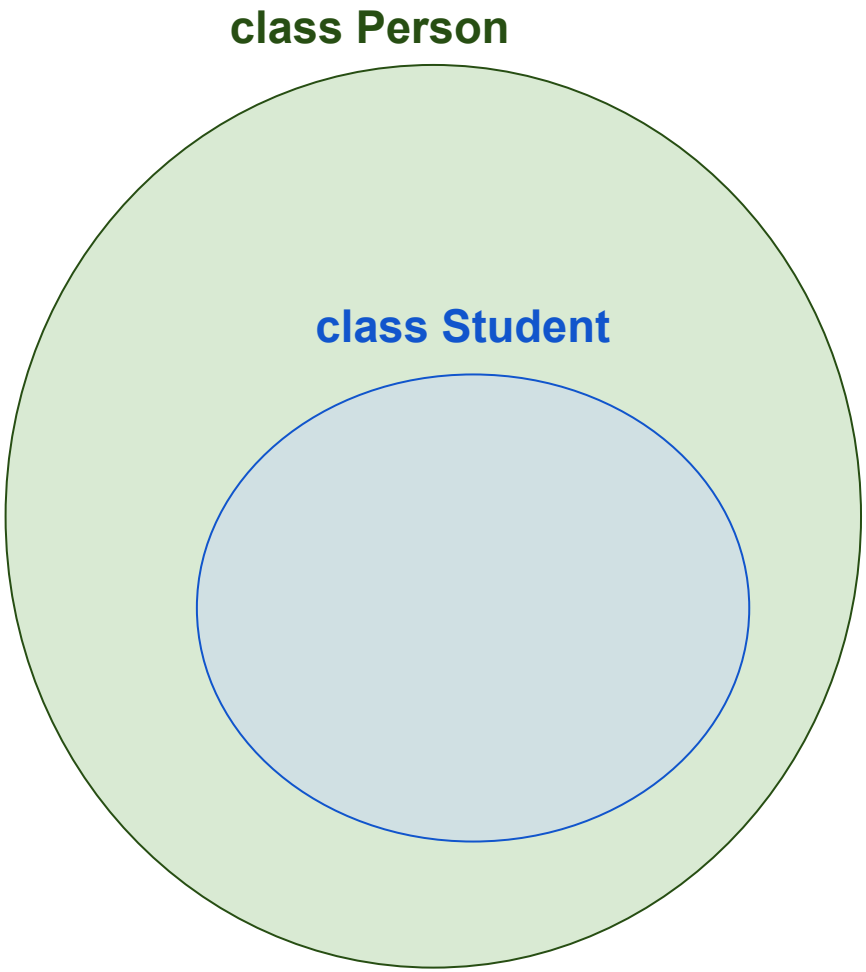
Sapienza
Università di Roma

I3S Facoltà di Ingegneria dell'Informazione, Informatica e Statistica

DIAG Dipartimento di Ingegneria Informatica, Automatica e Gestionale

Inheritance

- Allows to define a hierarchy of classes
- Class A is a subclass of class B if all instances of A can be considered instances of B
- A subclass inherits all the attributes and methods of the parent class



Example

```
class Person:
    def __init__(self,name:str,surname:str):
        self.name = name
        self.surname = surname

    def printInfo(self):
        print(f"name:{self.name},surname:{self.surname}")

    def __str__(self):
        return f"{self.name},{self.surname}"

class Student(Person):
    def __init__(self, name: str, surname: str, student_id:str):
        super().__init__(name, surname)
        self.student_id = student_id

    def printInfo(self):
        print(f"name:{self.name},surname:{self.surname},student_id:{self.student_id}")

    def __str__(self):
        return super().__str__() + f",{self.student_id}"


p = Person("Mario","Rossi")
print(p)
p.printInfo()

s = Student("Luigi","Bianchi","1234567")
print(s)
s.printInfo()
print(s.name)

print(p.student_id)
```

Example

```
class Person:
    def __init__(self,name:str,surname:str):
        self.name = name
        self.surname = surname

    def printInfo(self):
        print(f"name:{self.name},surname:{self.surname}")

    def __str__(self):
        return f"{self.name},{self.surname}"

class Student(Person):
    def __init__(self, name: str, surname: str, student_id:str):
        super().__init__(name, surname)
        self.student_id = student_id

    def printInfo(self):
        print(f"name:{self.name},surname:{self.surname},student_id:{self.student_id}")

    def __str__(self):
        return super().__str__() + f",{self.student_id}"


p = Person("Mario","Rossi")
print(p)                                    // Mario,Rossi
p.printInfo()                               // name:Mario,surname:Rossi

s = Student("Luigi","Bianchi","1234567")
print(s)                                    // Luigi,Bianchi,1234567
s.printInfo()                               // name:Luigi,surname:Bianchi,student_id:1234567
print(s.name)                               // Luigi

print(p.student_id)                         // Raises AttributeError
```

Overriding

- Overriding occurs when a subclass provides a more specific implementation of a method defined in a subclass, the two methods have the same signature
- The interpreter uses the implementation provided by the most specific class

```
class Person:
    def __init__(self,name:str,surname:str):
        self.name = name
        self.surname = surname

    def printInfo(self):
        print(f"name:{self.name},surname:{self.surname}")

    def __str__(self):
        return f"{self.name},{self.surname}"

class Student(Person):
    def __init__(self, name: str, surname: str, student_id:str):
        super().__init__(name, surname)
        self.student_id = student_id

    def printInfo(self):
        print(f"name:{self.name},surname:{self.surname},student_id:{self.student_id}")

    def __str__(self):
        return super().__str__() +f",{self.student_id}"
```

Overriding

Overloading

- In statically typed languages we can have functions with the same name but different types or number of parameters, this is called overloading
- Python is a dynamically typed language, so in principle we could have at most two functions with the same name but different number of parameters
- However this is not allowed in Python, indeed the namespaces in Python are implemented with dictionaries, in which functions are addressed by their names, so if a function is redefined with a different parameter list the old function is replaced by the new function

```
>>> def foo(a,b):
...     return a+b
...
>>> globals()
{'__name__': '__main__', '__doc__': None, '__pac
kage__': None, '__loader__': <class '_frozen_imp
ortlib.BuiltinImporter'>, '__spec__': None, '__a
nnotations__': {}, '__builtins__': <module 'buil
tins' (built-in)>, 'foo': <function foo at 0x7f6
92a245700>}
>>> def foo(a,b,c):
...     return a+b+c
...
>>> globals()
{'__name__': '__main__', '__doc__': None, '__pac
kage__': None, '__loader__': <class '_frozen_imp
ortlib.BuiltinImporter'>, '__spec__': None, '__a
nnotations__': {}, '__builtins__': <module 'buil
tins' (built-in)>, 'foo': <function foo at 0x7f6
92a245790>}
>>> foo(1,2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: foo() missing 1 required positional a
rgument: 'c'
>>>
```

Operators Overloading

- A class can implement certain operations that are invoked by special syntax (such as arithmetic operations or subscripting and slicing) by defining methods with special names. This is Python's approach to operator overloading, allowing classes to define their own behavior with respect to language operators.

- These are the most common operators:

Comparison operators

```
object.__lt__(self, other)
object.__le__(self, other)
object.__eq__(self, other)
object.__ne__(self, other)
object.__gt__(self, other)
object.__ge__(self, other)
object.__bool__(self)
```

Math operators

```
object.__add__(self, other)
object.__sub__(self, other)
object.__mul__(self, other)
object.__truediv__(self, other)
object.__floordiv__(self, other)
object.__mod__(self, other)
object.__pow__(self, other[, modulo])
```

Operators for Container Objects

```
object.__len__(self)
object.__getitem__(self, key)
object.__setitem__(self, key, value)
object.__iter__(self)
object.__contains__(self, item)
```

Operators for Callable Objects

```
object.__call__(self[, args...])
```