Python Programming 1

(CP1S02005E)

Białystok University of Technology
Faculty of Electrical Engineering

Industry Digitization, semester II
Academic year 2024/2025

Lecture no. 10 (21.05.2025)

Jarosław Forenc, PhD

Topics

- Object-Oriented Programming
  - definitions, class structure
  - constructor, attributes and methods
  - inheritance
  - multiple inheritance
  - access rights

Python - object oriented programming

- object-oriented programming is a programming paradigm in which programs are constructed by defining and manipulating objects
- in Python, object-oriented programming is based on three main elements: classes, objects, and inheritance
- a class is a template or blueprint that defines the properties and behaviors of objects (in a sense, it creates a new type)
- an object is an instance of a class (a variable of the created type)
- attributes are the properties of objects that store data; in Python, new attributes can be added to an object at any time
- methods are functions defined inside a class that operate on objects of that class; these methods have access to the object's attributes and can manipulate them
- inheritance allows the creation of new (derived) classes based on existing (base or parent) classes; the derived class inherits the attributes and methods of the base class

Python - object oriented programming (classes)

- general structure of a class definition in Python

```
class ClassName:
    # Class attributes
    variable = "value"

    # Constructor
    def __init__(self, parameters):
        self.parameters = parameters

    # Class methods
    def method(self):
        # method code
        return something
```

- a class consists of a class header and a class body
- the header begins with the keyword class, followed by the class name
- by convention, the class name starts with a capital letter
- the class body contains the attributes and methods that define the behavior and properties of objects created from that class

Python - object oriented programming (classes)

- class definition of Person, object declaration smith

```
class Person:

    def __init__(self, fname, lname, age):
        self.fname = fname
        self.lname = lname
        self.age = age

    def print(self):
        print(f"{self.fname} {self.lname}, {self.age}
                  years old")

smith = Person("John","Smith",25)
smith.print()
```

```
John Smith, 25 years old
```

Python - object oriented programming (constructor)

```
def __init__(self, fname, lname, age):
    self.fname = fname
    self.lname = lname
    self.age = age
```

- __init__ - the constructor, a special method called when a new instance (object) of the class is created

```
smith = Person("Johnn","Smith",25)
```

- double underscores at the beginning and end are a naming convention used to avoid conflicts with other methods that might be defined later with the same names
- the first parameter of the constructor should be named self - this is a reference to the instance of the class itself
- thanks to self, object instances gain access to the attributes and methods defined in the class

Python - object oriented programming (constructor)

```
def __init__(self, fname, lname, age):
    self.fname = fname
    self.lname = lname
    self.age = age
```

- __init__ - the constructor, a special method called when a new instance (object) of the class is created

```
smith = Person("Johnn","Smith",25)
```

- the constructor receives the arguments fname, lname and age, while the self argument is passed automatically
- variables preceded by self are attributes, and they are accessible to every method within the class

Python - object oriented programming (constructor)

```
def __init__(self, fname, lname, age):
    self.fname = fname
    self.lname = lname
    self.age = age
```

- __init__ - the constructor, a special method called when a new instance (object) of the class is created

```
smith = Person("Johnn","Smith",25)
```

- self.fname = fname takes the value stored in the fname parameter, assigns it to the fname attribute, and this attribute is then attached to the instance being created
- there is no return statement in the constructor, but Python automatically returns the instance of the Person class

Python - object oriented programming (methods)

```
def print(self):
    print(f"{self.fname} {self.lname}, {self.age} years old")
```

- the print() method does not require any additional data, so it has only one parameter - self
- calling the method:

```
instance_name.method_name(arguments)
```

- calling the print() method on the object instance smith:

```
smith = Person("John","Smith",25)
smith.print()
```

Python - object oriented programming (__str__)

- instead of the print() method, you can use the built-in __str__ method, which is called automatically when the object is converted to a string using the str() function or when the object is used in a context where a string is expected, such as in the print() function

```
class Person:

    def __init__(self, fname, lname, age):
        self.fname = fname
        self.lname = lname
        self.age = age

    def __str__(self):
        return f"{self.fname} {self.lname}, {self.age}
                 years old"

smith = Person("John","Smith",25)
print(smith)
```

```
John Smith, 25 years old
```

Python - object oriented programming (example)

- a class describing a triangle

```
class Triangle:
    def __init__(self, a, h):
        self.a = a  # base
        self.h = h  # height

    def area(self):
        return (self.a * self.h) / 2

    def __str__(self):
        return f"[a = {self.a}, h = {self.h}]"

tr1 = Triangle(6, 8)
print(tr1)
print(f"Area of triangle: {tr1.area():.2f}")
```

```
[a = 6, h = 8]
Area of triangle: 24.00
```

Python - object oriented programming (example)

- access to class attributes

```
instance_name.attribute_name
```

```
tr1 = Triangle(6, 8)
tr1.a = 5
tr1.h = 9
```

- method invocation

```
instance_name.method_name()
```

```
tr1 = Triangle(6, 8)
p = tr1.area()
print(f"Area of triangle: {tr1.area():.2f}")
```

Python - object oriented programming

- it is possible to create multiple instances (objects) based on a single class

```
class Person:

    def __init__(self, fname, lname, age):
        self.fname = fname
        self.lname = lname
        self.age = age

    def __str__(self):
        return f"{self.fname} {self.lname}, {self.age}
                 years old"

smith = Person("John","Smith",25)
print(smith)

brown = Person("Kate","Brown",18)
print(brown)
```

```
John Smith, 25 years old
Kate Brown, 18 years old
```

Python - object oriented programming

- not all attributes need to have values passed during initialization - there can be attributes with default values

```
class Person:
    children = 0

    def __init__(self, fname, lname, age):
        self.fname = fname
        self.lname = lname
        self.age = age

    def __str__(self):
        return f"{self.fname} {self.lname}, {self.age}
                 years old, children: {self.children}"

smith = Person("John","Smith",25)
print(smith)
```

```
John Smith, 25 years old, children: 0
```

Python - object oriented programming

- not all attributes need to have values passed during initialization - there can be attributes with default values

```
class Person:

    def __init__(self, fname, lname, age):
        self.fname = fname
        self.lname = lname
        self.age = age
        self.children = 0

    def __str__(self):
        return f"{self.fname} {self.lname}, {self.age}
                 years old, children: {self.children}"

smith = Person("John","Smith",25)
print(smith)
```

```
John Smith, 25 years old, children: 0
```

Python - object oriented programming

- an attribute's value can be changed directly in an instance (object)

```
smith = Person("John","Smith",25)
smith.children = 2
print(smith)
```

```
John Smith, 25 years old, children: 2
```

- an attribute's value can be changed using a method

```
class Person:
    ...
    def save_children(self, number_of_children):
        self.children = number_of_children

smith = Person("John","Smith",25)
smith.save_children(3)
print(smith)
```

Python - object oriented programming

- an attribute's value can be changed directly in an instance (object)

```
smith = Person("John","Smith",25)
smith.children = 2
print(smith)
```

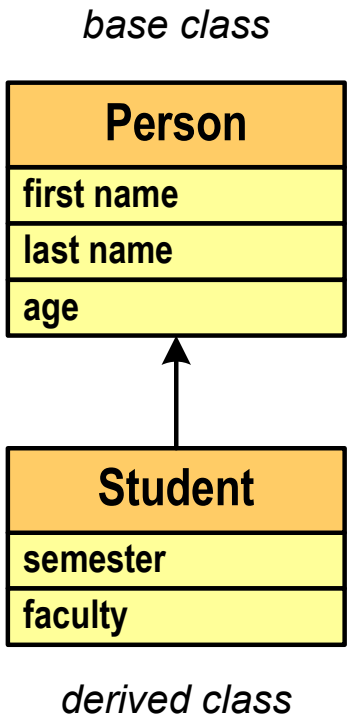
```
John Smith, 25 years old, children: 2
```

- an attribute's value can be changed using a method

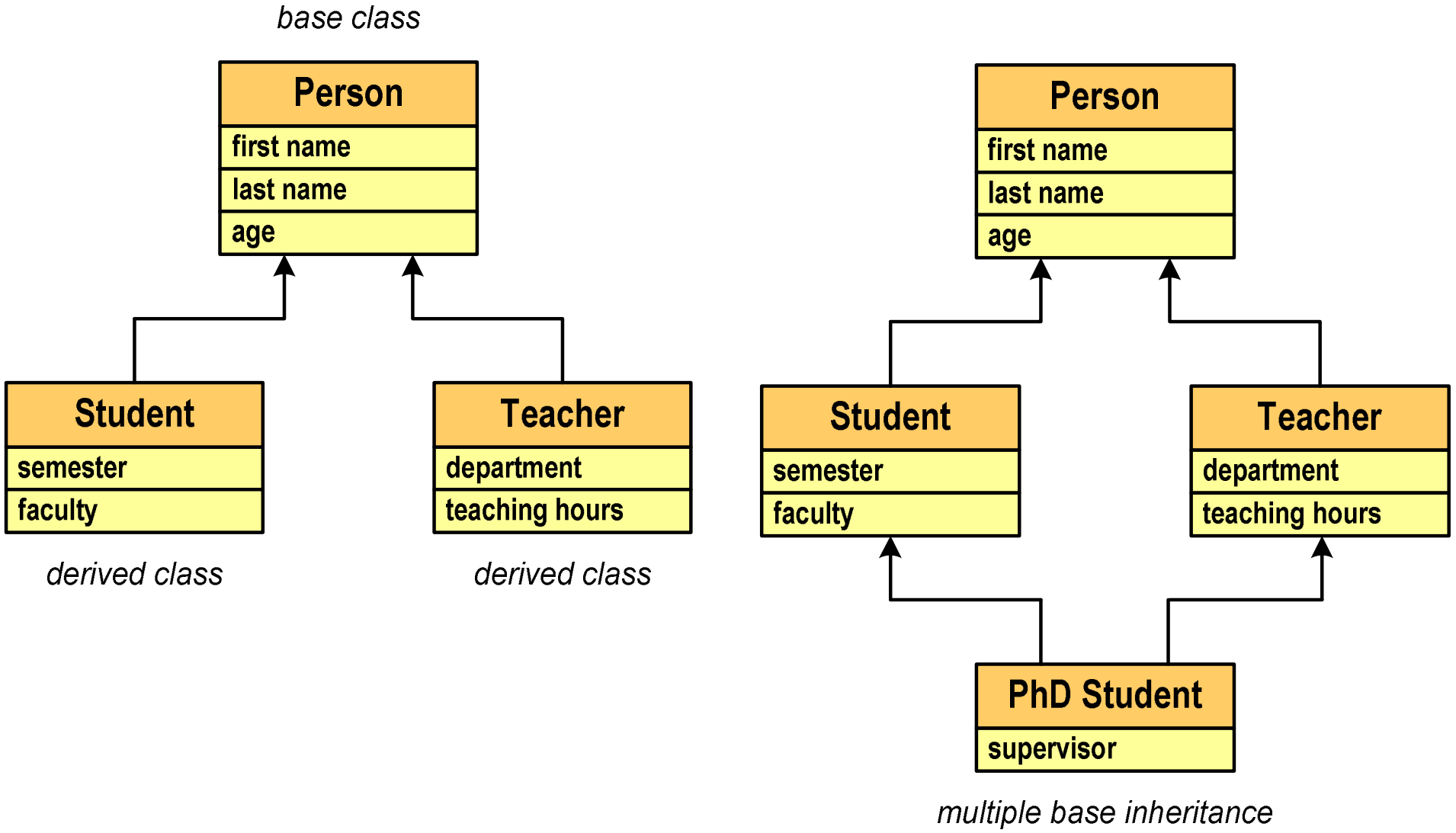
```
class Person:
    ...
    def add_child(self):
        self.children += 1

smith = Person("John","Smith",25)
smith.add_child()
print(smith)
```

Python - object oriented programming (inheritance)

- inheritance is a technique that allows the creation of new classes based on existing ones
- a class from which other classes inherit is called a base class (also known as a parent class or superclass)
- the new class is called a derived class (also known as a child class or subclass)
- the new class automatically inherits the attributes and methods of the base class
- a derived class can also define new attributes and methods



Python - object oriented programming (inheritance)



Python - object oriented programming (inheritance)

```
class Person:
    def __init__(self, fname, lname):
        self.fname = fname
        self.lname = lname

    def __str__(self):
        return f"Person: {self.fname} {self.lname}"

class Student(Person):
    def __init__(self, fname, lname, semester, faculty):
        super().__init__(fname, lname)
        self.semester = semester
        self.faculty = faculty

    def __str__(self):
        return (f"Student: {self.fname} {self.lname}, "
                f"semester: {self.semester}, faculty: "
                f"{self.faculty}")
```

Python - object oriented programming (inheritance)

- the base class must be located in the same file where the derived class is created
- the name of the base class must be included in the header of derived class

```
class Student(Person):
```

- in the __init__() method of the derived class, we call the __init__() method from the base class

```
super().__init__(fname, lname)
```

- super() is a built-in function that returns a special object (called a delegate), which allows access to methods and attributes of the base class from within the derived class
- super() is used to call the constructor or methods of the base class; the function name comes from the term "superclass"

Python - object oriented programming (inheritance)

- creating objects and calling methods

```
person = Person("John", "Smith")
student = Student("Kate", "Brown", 3, "EE")

print(person)
print(student)

print(f"First name: {student.fname}")
print(f"Last name: {student.lname}")
print(f"Semester: {student.semester}")
print(f"Faculty: {student.faculty}")
```

```
Person: John Smith
Student: Kate Brown, semester: 3, faculty: EE
Fisrt name: Kate
Last name: Brown
Semester: 3
Faculty: EE
```

Python - object oriented programming (inheritance)

- any method from the base class can be overridden

```
class Vehicle:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model

    def desc(self):
        return f"Vehicle brand {self.brand}, model {self.model}"

class Car(Vehicle):
    def __init__(self, brand, model, year):
        super().__init__(brand, model)
        self.year = year

    def desc(self):
        basic_desc = super().desc()
        return f"{basic_desc}, year of production: {self.year}"
```

Python - object oriented programming (inheritance)

- a method from the derived class overrides the visibility of a method from the base class

```
vehicle = Vehicle("Toyota", "Corolla")
car = Car("BMW", "X5", 2022)

print(vehicle.desc())
print(car.desc())
```

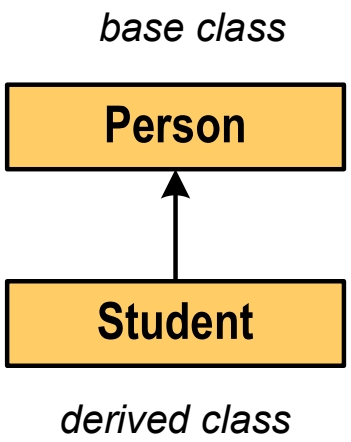
```
Vehicle brand Toyota, model Corolla
Vehicle brand BMW, model X5, year of production: 2022
```

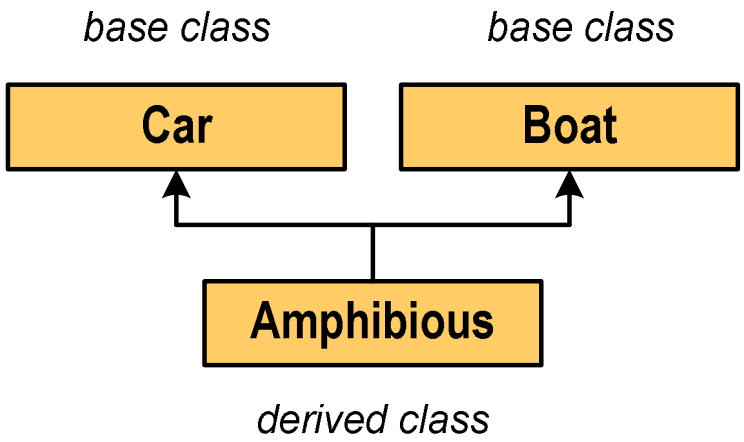
- using the super() function, one can refer to a method from the base class

```
def desc(self):
    basic_desc = super().desc()
    return f"{basic_desc}, rok produkcji: {self.rok}"
```

Python - object oriented programming (inheritance)

- single inheritance - a derived class inherits from one base class
- multiple inheritance - a derived class inherits from more than one base class





Python - object oriented programming (inheritance)

```
class A:
    def __init__(self):
        self.feature_a = "Feature from class A"

    def method_a(self):
        return "Method from class A"

class B:
    def __init__(self):
        self.feature_b = "Feature from class B"

    def method_b(self):
        return "Method from class B"

class C(A, B):
    def __init__(self):
        A.__init__(self)
        B.__init__(self)
        self.feature_c = "Feature from class C"

    def method_c(self):
        return "Method from class C"
```

Python - object oriented programming (inheritance)

```
instance = C()
print(instance.feature_a)
print(instance.feature_b)
print(instance.feature_c)

print(instance.method_a())
print(instance.method_b())
print(instance.method_c())
```

```
Feature from class A
Feature from class B
Feature from class C
Method from class A
Method from class B
Method from class C
```

Python - object oriented prog. (access rights)

- Python does not have formal access modifiers like in C++ (e.g., public, protected, private)
- instead, naming conventions and certain mechanisms are used to indicate levels of access to class attributes and methods.
- by default, all attributes and methods are public - this means they are accessible from outside the class; there are no special name prefixes

```
class MyClass:
    def __init__(self):
        self.public_attribute = "Public access"

    def public_method(self):
        return "This is a public method"

instance = MyClass()
print(instance.public_attribute)  # Public access
print(instance.public_method())  # This is a public method
```

Python - object oriented prog. (access rights)

- protected attributes and methods are marked by a single underscore _ at the beginning of the name
- this is a convention indicating that the attribute or method should not be used outside the class or its subclasses, although technically it is still accessible

```
class MyClass:
    def __init__(self):
        self._protected_attribute = "Protected access"

    def _protected_method(self):
        return "This is a protected method"

instance = MyClass()
print(instance._protected_attribute)  # Access allowed,
                                      # not recommended
print(instance._protected_method())   # Access allowed,
                                      # not recommended
```

Python - object oriented prog. (access rights)

- private attributes and methods are marked by two underscores __ at the beginning of the name
- Python uses name mangling to make it more difficult to access these elements from outside the class; it modifies the name by adding a prefix _ClassName

```
class MyClass:
    def __init__(self):
        self.__private_attribute = "Private access"

    def __private_method(self):
        return "This is a private method"

instance = MyClass()
print(instance.__private_attribute)
print(instance.__private_method())
```

Python - object oriented prog. (access rights)

- private attributes and methods are marked by two underscores __ at the beginning of the name
- Python uses name mangling to make it more difficult to access these elements from outside the class; it modifies the name by adding a prefix _ClassName

```
class MyClass:
    def __init__(self):
        self.__private_attribute = "Private access"

    def __private_method(self):

instan
print
print
```

```
Traceback (most recent call last):
  File "d:\tempCodeRunnerFile.py", line 9, in <module>
    print(instance.__private_attribute)
          ^^^^^^^^^^^^^^^^^^^^^^^^^^^^
AttributeError: 'MyClass' object has no attribute
'__private_attribute'. Did you mean:
'_MyClass__private_attribute'?
```

Python - object oriented prog. (access rights)

- it is possible to access private attributes and methods via name mangling, but this is not recommended

```
class MyClass:
    def __init__(self):
        self.__private_attribute = "Private access"

    def __private_method(self):
        return "This is a private method"

instance = MyClass()

# Name mangling:
print(instance._MyClass__private_attribute)
print(instance._MyClass__private_method())
```

```
Private access
This is a private method
```

End of lecture no. 10

Thank you for your attention!