Classes

Procedure Oriented Programming → functions

vs

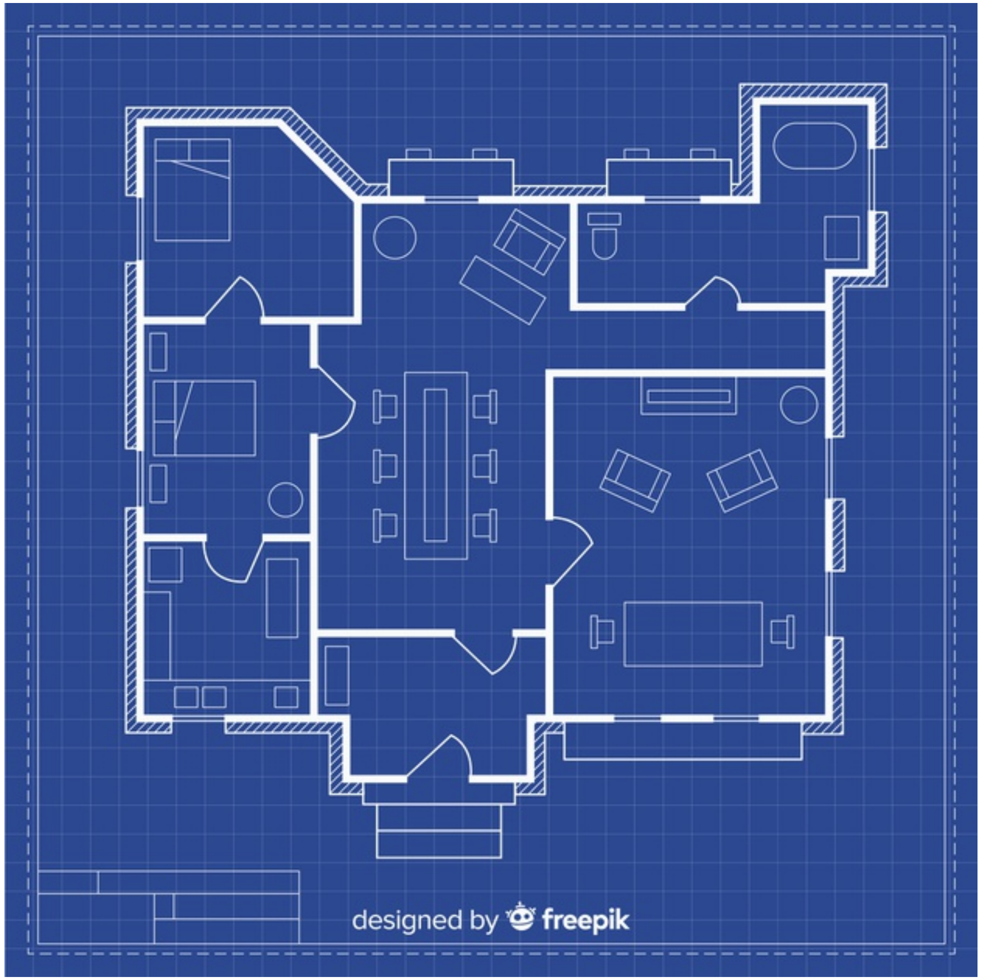
Object Oriented Programming → **objects**

an object → a collection of **attributes** (**variables**) and **methods** (**functions**)

a class → a “**blueprint**” for the object

Classes

a “**blueprint**” (sketch, prototype) for **an object**



we can build many houses from the sketch,
i.e., we can **create many objects** of **a class**

every created **object** is called **an instance** of **a class**

Definition of a class

an example:

```python
class Point2D:
    """Simple class for representing a point in 2D."""
    def __init__(self, x, y):
        """Create a new Point at x, y."""
        self.x = x
        self.y = y

    def translation(self, dx, dy):
        """Moving the point by dx and dy in the x and y direction."""
        self.x += dx
        self.y += dy
```

Definition of a class

an example:

```
class Point2D:
    """Simple class for representing a point in 2D."""
    def __init__(self, x, y):
        """Create a new Point at x, y."""
        self.x = x
        self.y = y

    def translation(self, dx, dy):
        """Moving the point by dx and dy in the x and y direction."""
        self.x += dx
        self.y += dy
```

- class statement **(a docstring, i.e., a brief description of a class)**
- Write the constructor or init method
- Use self to refer to attributes and methods
- The basic attributes are defined in the constructor
- You can define more methods like *translation*

x in **self.x** means that there is an **attribute named "x"**

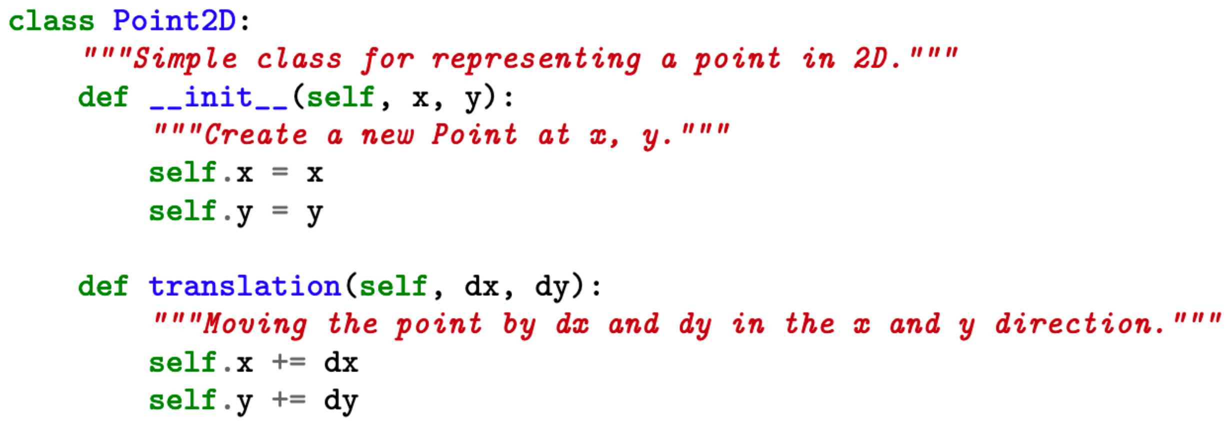
in contrast, **x** in **__init__(self,x,y)** is just a **local variable** that is assigned value when the user makes an instance of a class

Creating objects (instances of class)

```python
class Point2D:
    """Simple class for representing a point in 2D."""
    def __init__(self, x, y):
        """Create a new Point at x, y."""
        self.x = x
        self.y = y

    def translation(self, dx, dy):
        """Moving the point by dx and dy in the x and y direction."""
        self.x += dx
        self.y += dy
```

Creating objects (instances of class)



```
class Point2D:
    """Simple class for representing a point in 2D."""
    def __init__(self, x, y):
        """Create a new Point at x, y."""
        self.x = x
        self.y = y

    def translation(self, dx, dy):
        """Moving the point by dx and dy in the x and y direction."""
        self.x += dx
        self.y += dy
```

using the **class Point2D**:

p1 = Point2D(2,3) **Created object p1**— calling the class as a function called the constructor

p2 = Point2D(-1.1,42) **Created another object p2** — data of p1 and p2 are independent

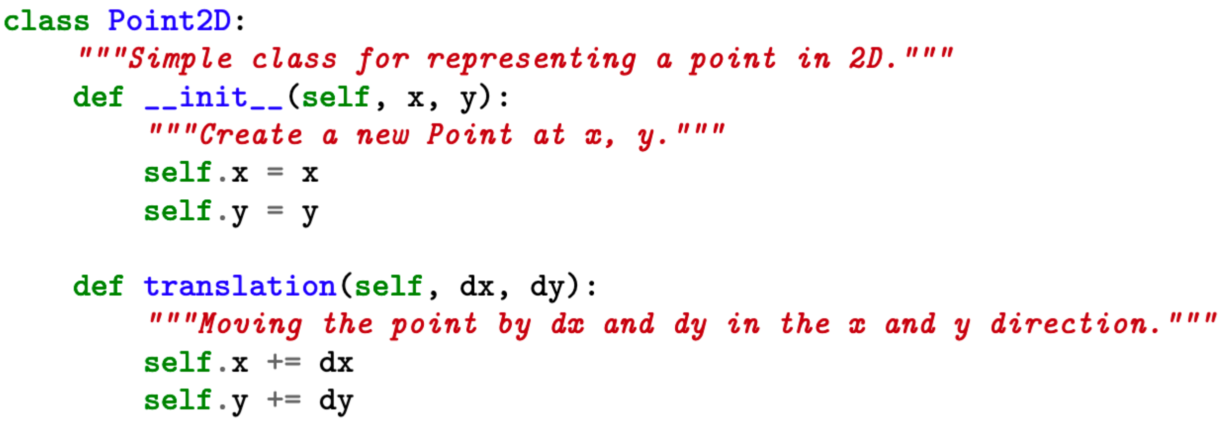
print(p1.x, p2.y) **2 42** **Accessed the attributes** —printed the x and y attributes

p1.translate(4,0) **Called the method** of the class to interface with the attributes of p1

print(p1.x, p2.y) **6 42** **Accessed the attributes** —printed the x and y attributes

p1.__dict__

Creating objects (instances of class)



```
class Point2D:
    """Simple class for representing a point in 2D."""
    def __init__(self, x, y):
        """Create a new Point at x, y."""
        self.x = x
        self.y = y

    def translation(self, dx, dy):
        """Moving the point by dx and dy in the x and y direction."""
        self.x += dx
        self.y += dy
```

using the **class Point2D**:

p1 = Point2D(2,3) **Created object p1**— calling the class as a function called the constructor
p2 = Point2D(-1.1,42) **Created another object p2** — data of p1 and p2 are independent

print(p1.x, p2.y) 2 42 **Accessed the attributes** —printed the x and y attributes
p1.translate(4,0) **Called the method** of the class to interface with the attributes of p1
print(p1.x, p2.y) 6 42 **Accessed the attributes** —printed the x and y attributes

NOTE: the command **p1.x = p1.x + 4** achieves the same as the command **p1.translate(4.0)** but is bad practice. The class should provide all the necessary methods to manipulate its attributes properly

The constructor

```
class Point2D:
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

Initialization

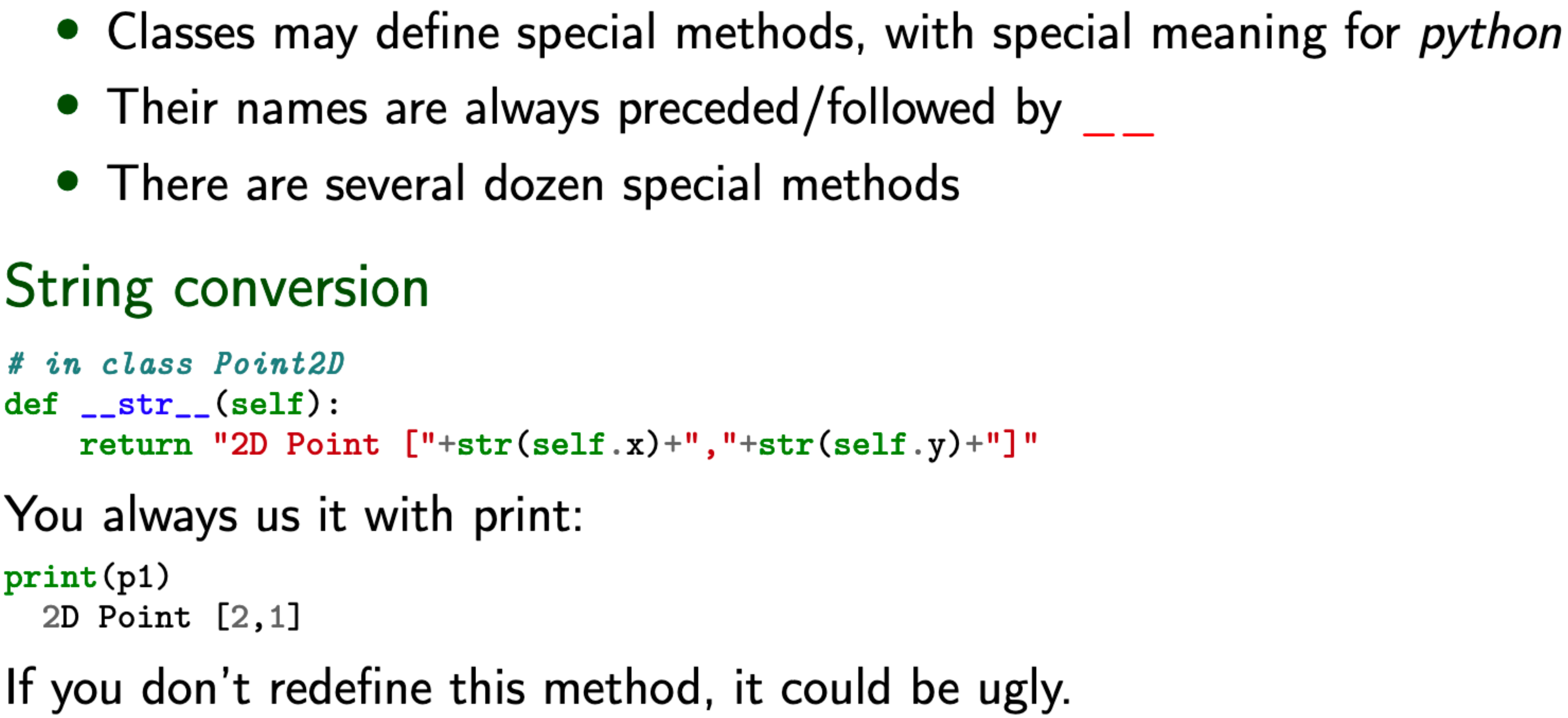
This method initializes a new instance:

- build a ready to use object
- returns a reference on it
- Called whenever a new object is created

Special methods

This is an example of a "special method", it has special meaning to Python interpreter

Special methods



- Classes may define special methods, with special meaning for *python*
- Their names are always preceded/followed by __
- There are several dozen special methods

String conversion

```
# in class Point2D
def __str__(self):
    return "2D Point ["+str(self.x)+","+str(self.y)+"]"
```

You always us it with print:

```
print(p1)
  2D Point [2,1]
```

If you don't redefine this method, it could be ugly.

NOTE: there are many more examples of special methods in the lecture notes!

Inheritance

A **tool** for introducing **new classes** which contain some attributes and methods of the class they **originate** from

The **Circle** class is called a derived class or subclass of the **Shape** class which is known as the base class or superclass

```
class Shape:
   def __init__(self,x,y): # build a Shape
       self.x = x
       self.y = y

   def translate(self,dx,dy):
      ...

   def area(self):
      raise NotImplementedError()
```

```
class Circle(Shape):              # Inherits from Shape
   def __init__(self,x,y,radius):
      Shape.__init__(self,x,y)    # First build a Shape
      self.radius = radius        # Then specialize

   def area(self):
      return math.pi*self.radius**2
```

Inheritance

A **tool** for introducing **new classes** which contain some attributes and methods of the class they **originate** from

The **Circle** class is called a derived class or subclass of the **Shape** class which is known as the base class or superclass

```python
class Shape:
   def __init__(self,x,y): # build a Shape
       self.x = x
       self.y = y

   def translate(self,dx,dy):
      ...

   def area(self):
      raise NotImplementedError()
```

```python
class Circle(Shape):                  # Inherits from Shape
   def __init__(self,x,y,radius):
      Shape.__init__(self,x,y)        # First build a Shape
      self.radius = radius            # Then specialize

   def area(self):
      return math.pi*self.radius**2
```

An instance of class **Circle** has all the attributes and methods as an instance of class **Shape** (it shares a position (**center)**, it can be **translated),** and some more.

Shape.area() is not implemented (yet), at this level of abstraction
But **Circle** is more concrete and provides the **.area()** method

We can define more subclasses, like **Rectangle**, **Square…**

NOTE: Inheritance avoids duplication of code; allows new objects which are specialized