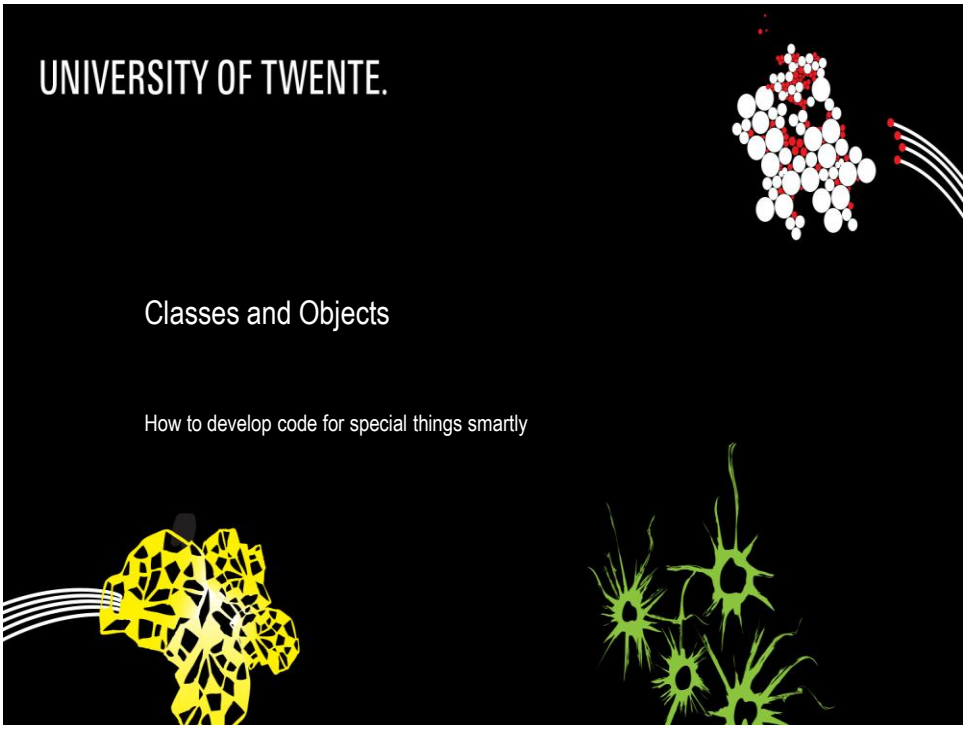


1

Main objectives

After this lecture, students will be able to:

- Explain the object-oriented programming paradigm
- Understand classes, instances and methods
- Use object-oriented techniques

2

Object-oriented Programming

The idea behind the object-oriented paradigm

- The real world is made up of **objects**, which have characteristics (modelled as *data attributes*) and behaviour (modelled as *actions or methods*)
- Similar objects can be grouped and described in one go, as a **class**
- Complex objects can be defined in terms of less complex objects in a hierarchical manner

All cars *have doors, use fuel*, and *can be steered and parked*. The class *describes the commonalities* between different cars.

Class car

---

data

wheels: list of wheel [4..4]
gear: manual | automatic
steering: electrical | mechanical
doors: list of door [2..5]
size: real
width: real
height: real
fuel: diesel | gasoline

---

actions

steer()
brake()
speed_up()
park()
….

Classes

A class defines the common things, and thus can be used as a template to create instances (aka objects)

Class car

---

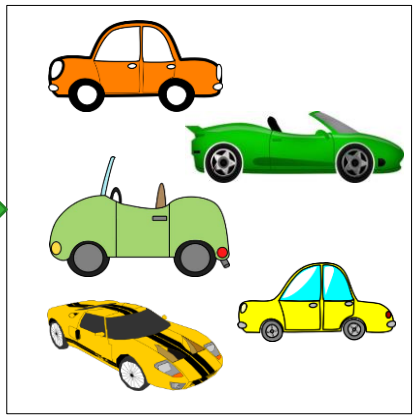
data

wheels: list of wheel [4..4]
gear: manual | automatic
steering: electrical | mechanical
doors: list of door [2..5]
size: real
width: real
height: real
fuel: diesel | gasoline

---

actions

steer()
brake()
speed_up()
park()
….



Programming pragmatics

Obviously, the car class is not so useful for geocomputing perhaps. It helps to explain the notions of o-o though.

In geocomputing, more useful classes could be those capturing geometry (classes Point, Linestring, Multipolygon, etc.) or gridded data (classes Raster, Rasterband, Rastercell). With each we would have attributes and methods, for instance

class *Linestring*
data
spatialrefsystem: integer,
vertices : list of Point
actions
is_closed(),
metric_length(),
rotate_around(c : Point)

This example provides a class that models a data container.
In o-o programming, we commonly also find classes that model some form of computing. An example could be a pathfinder class that determines an optimal between two given points. The pathfinder class could have data and actions associated.

Why would we use classes?

All programming can be done without object-orientation. O-o is an code *organizing principle* that has proven its value in efficient coding.

When object-orientation is applied properly, it often allows *more rapid code development*, and often also better quality code that fails less often. This is because o-o aims to *factor out commonalities*. It is an acquired skill to work well with o-o.

Classes *help organize the code* and the coding approach.

Classes promote code reuse, as we will see

Most GIS libraries (numpy, gdal, scikit-learn, all forthcoming) use classes

Class definition

```
class Point:
    x = 0
    y = 0
```

← capitalized name (good convention)
← *x* and *y* are shared class attributes

```
p = Point()                    # create a point object p
type(p)                        # what is p's type?
<class 'Point'>                # its type is Point
```

A class definition introduces a new type, and a type is an object too!

```
type(Point)                    # what is the type of Point?
<class 'type'>                 # its type is … type
```

This is somewhat confusing at first. Object p is a Point, and Point is a type.

Besides a type, each object p has a value, just *p*, as well as an identity, *id(p)*. The first can change and the second cannot. An identity is a unique but otherwise meaningless, large integer value.

Syntax for class definition

```
class Name(Superclass, …):
    attr = value               # shared class attribute
    def method (self,….):      # methods
        self.attr = value      # per-instance attribute
```

attributes

A class has a name, and may be based on ("derived from") one or more superclasses. If we already have class *Geometry*, we could define:

class Linestring(Geometry)

This states that linestrings are a special case of geometry.

A class itself can have variables, and its instantiations (its objects) also.

Classes and their members

Members of a type are called **instances** of that type or *objects*

Creating or initializing a new instance is called *instantiation*

```
p1 = Point()                    # instantiation, p1 is now an object
```

During instantiation the class's **constructor method** is called (more on this later)

The items of a class/instance are called its **attributes** (here x,y)

In Python, you can get/set the attributes with the dot notation:

```
p1.x = 7.0                      # set attribute
p1.y = 8.0
p1.x                            # get attribute
7.0
```

Class and object attributes

With
class Point:
  x = 0
  y = 0
we are creating two attributes at class level that are shared between all objects of this class, now and into the future. A change to *Point.x* will be seen by all objects. It is a **class attribute**.

If we next do
p = Point()
print(p.x)
We will see printed the class attribute *Point.x* because *p* does not yet have its own version of attribute *x*. It has no **object attribute**.

However, after
p.x = 6.0567
object *p* has obtained a (local) object attribute *x* that is separate from the class attribute *x*. Class attributes can thus, a.o., be used *as defaults*.

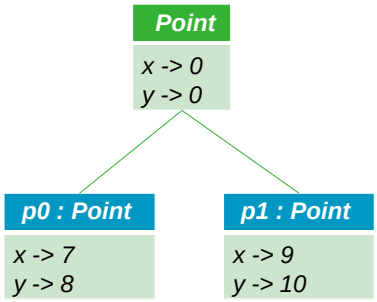
Class as a template for new objects

A class is a template for instances

A class statement creates a class object and assigns it a name

```
p0 = Point()
p0.x = 7
p0.y = 8
p1 = Point()
p1.x = 9
p1.y = 10

isinstance(p0, Point)
True
```

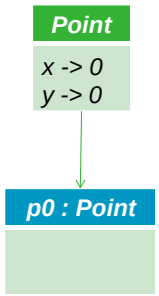


11

Object initialization

```
p0 = Point()
p0.x
0
```

What is this code doing?



12

Methods are defined inside a class

A method is a function that is defined inside a class, and that is meant to work only on instances of this class.

```
class Point:
    x = 0.0
    y = 0.0
    def distance(self, b):
        return math.hypot(self.x - b.x, self.y - b.y)
```

The *self* parameter denotes the point object to which we apply the method. The b point is the point towards which we measure distance.

The *self* parameter name is a coding convention, not a keyword
You cannot use or access *self* outside the scope of the method(s)

13

Encapsulation as a coding paradigm

Philosophy of object-orientation comes down to this:

- An object has a **state** through its data attribute values, as defined by its class template.
- Subtle dependency rules may exist between the data attribute values, and these should be violated. Violation will mean that the object is invalid.
- An object's *state is set* as we create the object from the class template
- An object's *state is changed* by executing methods defined in the class. Direct assignment of values to data attributes using the dot notation, other then in methods, should not be allowed unless such cannot violate the dependency rules. Thus, *methods are the shell around the object and should be the only means to change the object's state*.
- This principle is called **encapsulation**. The object's state is protected by its methods.

14

Inheritance as a code reuse mechanism

By viewing our program world as a *hierarchy of classes*, we can quickly build up data structures and functions for our purposes.

```
class Geometry():
    …
class Linestring(Geometry)
    …
```

Anything defined for geometries (attributes, methods) will now implicitly also be defined for linestrings. Class *Linestring* **inherits** these attributes/methods. If *Geometry* has attribute *spatialrefsystem: integer*, by inheritance now *Linestring* will also. Same for methods.

Class *Linestring* may have additional attributes and methods, specific to linestrings. The class may also **overwrite** inherited attributes/methods to model better the life of the linestring. *Geometry.length()* may by defintiion always return None, but *Linestring.length()* should not!

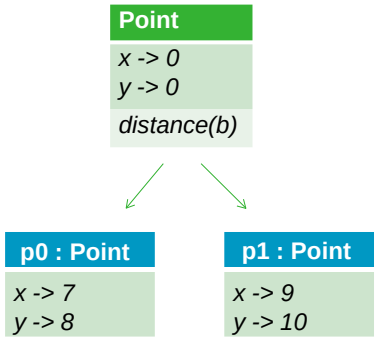
UNIVERSITY OF TWENTE.

15

METHODS

```
p0 = Point()
p0.x = 7
p0.y = 8
p1 = Point()
p1.x = 9
p1.y = 10

p0.distance(p1)
2.8284271247461903
```



UNIVERSITY OF TWENTE.

16

16

The *self* parameter

As indicated, *self* is only the (formal) formal parameter in method definitions that stands for the object on which the method is invoked.

Thus, *self* is not a data attribute of the object.

```
p0.self
AttributeError: Point instance has no attribute 'self'
```

When calling a method like *distance,* the actual value for *self* will be the object on which the method is invoked, and we do not have to indicate this. Thus, you only need to indicate a value for the second parameter:

```
p0.distance(p1)
```

This operates as if invoking *Point.distance(p0, p1)*

Other useful methods for class Point

```
# method toWKT will return the point as a string in Well-Known Text format
def toWKT(self):
    return 'POINT (%f %f)' % (self.x,self.y)

print(p0.toWKT())
POINT (9.000000 10.000000)

# method toJSON will return the point as a string in GeoJSON format
def toJSON(self):
    return json.dumps(dict(type='Point', coordinates=(self.x, self.y)))


print (p0.toJSON())
{'type': 'Point', 'coordinates': [9,10]}
```

Observe that none of these methods changes the state of the object.

State-changing methods

Here is a state changing method that translates the point in space:

```
# the move() method will move the point to another position
def move(self, dx, dy):
    self.x += dx
    self.y += dy

print(p0.move(2,12).toJSON())
{'type': 'Point', 'coordinates': [11, 22]}
```

Point p0's identity is still the same, but its state has now changed.

Changes to the class definition

In Python, you can change the class's variables (a class is just an object)

```
p0 = Point()                          # now, we have x==0 and y==0
Point.x = 6.0                         # change the shared class variable
Point.x , p0.x
(6.0, 6.0)                            #be careful, also the instances will change
                                      # (but not if p0.x has become an instance attribute)
Point.z = 0.0                         # we are adding a new class attribute
p0 = Point()
p0.z
0.0
```

Code reuse from libraries through o-o

Object-oriented programming is about code reuse.

Python libraries often provide very useful classes. These may fit your coding purposes directly, or not so precisely.

By importing a class C from a library, and basing our own definition for class D on it, with D a subclass of C, we can customize our code to purpose and reuse already written code for class C.

numpy, gdal, ogr are typical examples in geocomputing.

Summary

Classes help organize the code and coding

A class is a template for creating objects

Python has built-in classes, such as *string:* str.count()

Classes have attributes (data attributes and methods)

A method is a function associated with a class, and thus with its objects

The dot notation is used to get/set variables and call methods