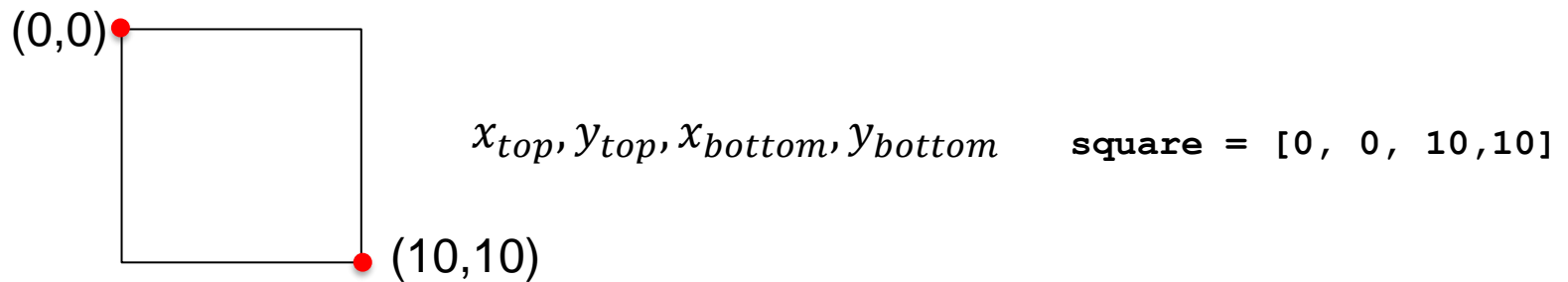


Lesson 8: Object-oriented programming

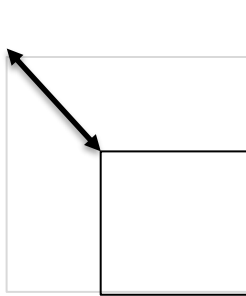
Motivation

We have seen different types of data that allowed us to represent the information that the programs needed to do their task.

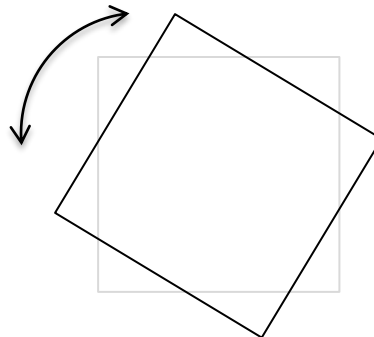
Imagine we want to represent the data that defines a square:



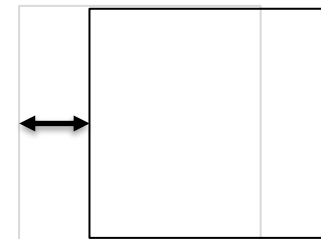
A square can have associated behaviors (functions) that modify its data:



escale



rotate



translation

Motivation

Create a model that represents an element (object) of the problem to be solved. It is composed of:

- a **set of data** that defines the element
- a collection of **operations** defined on such data

In this paradigm,

Solving the problem will consist in defining a set of objects (modules) that interact with each other

A great advantage of this approach is that the solution to a problem uses the same "nomenclature" (objects) that defines the problem itself.

Objects

Python supports different types of data:

```
1234      3.14159      "Hello"      [1, 5, 7, 11, 13]
{"CA": "California", "MA": "Massachusetts"}
```

Each of these examples is an **object** that has:

- a **type**
- an **internal representation of data** (primitive or composite)
- a set of **procedures** to interact with the object

An object is an **instance** of a type:

```
1234  is an instance of  int
"hello"  is an instance of  str
```

Object Oriented Programming

IN PYTHON, EVERYTHING IS AN OBJECT (and has a type)

In Python we can:

- Create new objects
- Manipulate objects
- Destroy objects

(explicitly using `del` or simply "forgetting" them)

```
mylist = ["Boemian Rhapsody", 1975, "Queen"]
```

```
mylist = 3          # The list mylist is not accessible anymore
```

The Python environment implements a garbage collector that recovers memory space of destroyed or inaccessible objects.

What is an Object?

An object is what is called an **abstract data**.

An abstract data is characterized by having:

- An **internal representation**: A set of data called **attributes**.
- An **interface** to interact with the object: A set of operations called **methods** (functions or procedures).

This interface allows us to define the behavior of the object by hiding the implementation.

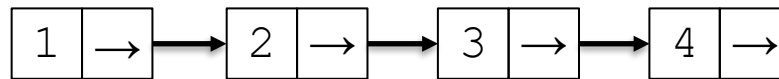
Example

`L = [1, 2, 3, 4]`

Internal representation of data.

As users, we do not need to know how it's organized.

In this case,



Sequence of nodes:

Each node points to where the next element is stored in the memory

The interface allows us to manipulate the list with a set of methods:

```
L[i], L[i:j], +  
len(), min(), max(), del(L[i])  
L.append(), L.extend(), L.count(), L.index(), L.insert(),  
L.pop(), L.remove(), L.reverse(), L.sort()
```

If we directly access data by manipulating the internal representation, the behavior may be compromised.

Advantages of object-oriented programming

- **Data is grouped with procedures** that work with this data through well-defined interfaces.
- OOP allows us to apply the "**divide and conquer**" strategy:
 - greater modularity reduces complexity
 - implement and test the behavior of each class separately
- Classes make **easier to reuse code**:
 - many Python modules define new classes
 - each class has a separate environment (without collision with function names)
 - inheritance allows us to redefine new classes

Classes: Create and use new types

It is important that we distinguish between classes and use an instance of the class.

Creating a class, implies creating a new type by:

- Defining the class name
- Defining the class attributes
- Implementing the methods

e.g. Someone wrote the code to create the class (type) List

Using a class implies:

- Creating a new instance, that is, a new object:

```
L = [1, 2]
```

- Using its methods on objects:

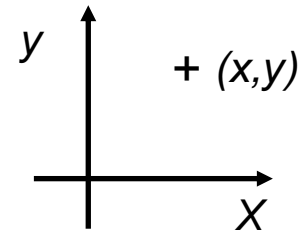
```
L.append(3)
```

```
len(L)
```

Creating a class

The instruction `class` allows us to define a new data type.
Let's define a class to represent a coordinate in a plane:

```
class Coordinate():  
    # Define attributes
```



- ① Just like when we defined a function with `def`, we indent the code to indicate which statements are part of the class definition.
- ② We assign a name to the class. This is the name of the new data type.

Creating a class: Attributes

Attributes are data and functions that "belong" to the class.

Data attributes:

- They consist of other objects that make up the class
- e.g. a coordinate consists of two numbers

Procedure Attributes: **Methods**

- They are functions that only work with that class.
- They define how to interact with the object.
- e.g. we can define a distance between two objects of the class `Coordinate`
 - but the distance between two objects of the class `list` does not make sense.

How to create an object in a class

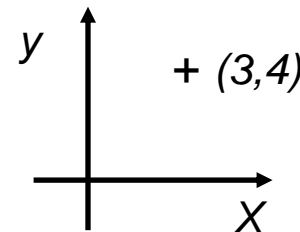
First, we need to define how to create an instance of class `Coordinate`.

e.g. We want to create an object with the information of the point (3,4)

We use a special method, called the **constructor of the class**, to initialize the data attributes.

The constructor is denoted by `__init__` (with double `_`):

```
class Coordinate():  
    def __init__(1) (self, x, y):  
        self.c_x = x (2) (3)  
        self.c_y = y  
        (4)
```



- ① Constructor: Special method for creating an instance. Mandatory in any class.
- ② **self**: parameter to refer to the instance of the class.
- ③ Data to initialize the object of class `Coordinate`.
- ④ Data attributes (`c_x` and `c_y`) for each object of class `Coordinate`.

How to create an object in a class

We have defined a first version of the class `Coordinate`:

```
class Coordinate():  
    def __init__(self, x, y):  
        self.c_x = x  
        self.c_y = y
```

To **create an instance**, we assign the result of the call to the constructor of the class `Coordinate` to a variable:

```
c = Coordinate(3,4)  
origin = Coordinate(0,0)
```



We call to `__init__` of class `Coordinate` and an object of `Coordinate` is created with values 3 and 4

We do **NOT** pass any argument for the parameter `self`;
Python does it automatically.

To access the attributes of an instance, we use the dot operator (`.`):

```
print(c.c_x)  
print(origin.c_x)
```



We use the dot (`.`) to access the attribute of instances `c` and `origin`

What are Methods?

- Methods are **procedural attributes**.
- They define the **behavior of the class**.
- Functions that **only work with their class**.
- Python always passes the object itself as the first argument of any method:

Convention → Always use **self** as the name of the first argument of all methods

- The **dot operator (.)** is used to access any attribute:
 - data attributes
 - methods

How to define a method in a class

We have defined a first version of the class `Coordinate`:

```
class Coordinate():
    def __init__(self, x, y):
        self.c_x = x
        self.c_y = y
```

We want to add the operation, the **method**, that allows us to calculate the distance between objects of the `Coordinate` class:

```
class Coordinate():
    def __init__(self, x, y):
        self.c_x = x
        self.c_y = y
    def distance(self, other):
        x_diff_sq = (self.c_x - other.c_x)**2
        y_diff_sq = (self.c_y - other.c_y)**2
        return (x_diff_sq + y_diff_sq)**0.5
```

- ① **self**: parameter to refer to the instance of the class.
- ② Parameter of the method. It will be an object of class `Coordinate`.
- ③ Dot operator (.) to access attributes (in this case, data attributes).
- ④ Methods behave like functions (input parameters, execute task, return results) except for **self** and the dot operator `.`

How to use a method

We have defined the class `Coordinate`:

```
class Coordinate():
    def __init__(self, x, y):
        self.c_x = x
        self.c_y = y
    def distance(self, other):
        x_diff_sq = (self.c_x - other.c_x)**2
        y_diff_sq = (self.c_y - other.c_y)**2
        return (x_diff_sq + y_diff_sq)**0.5
```

Conventional use

```
c = Coordinate(3,4)
origin = Coordinate(0,0)

print(c.distance(origin))
```

① ② ③

- ① Object on what we make the call
- ② Name of the method
- ③ Parameters without `self` (`self` is implemented to be `c`)

Equivalent

```
c = Coordinate(3,4)
origin = Coordinate(0,0)

print(Coordinate.distance(c,origin))
```

① ② ③

- ① Name of the class
- ② Name of the method
- ③ Parameters, including an object of the class the method belongs to (it represents `self`)

How to print an object of a class

If we print an object of class `Coordinate`, Python will show:


```
In [2]: c=Coordinate(3,4)

In [3]: print(c)
<__main__.Coordinate object at 0x000001D600511EB8>
```

This information is not very helpful...

We must define **how to convert** the information of objects **to a string** representation able to be printed:

```
c = Coordinate(3,4)
print(c)
```



`<3,4>`

We do it by defining a special method called `__str__`

Python calls this method every time the print function is called on an object of the class.

Conversion to str

We define the `__str__` method of the class `Coordinate`:

```
class Coordinate():
    def __init__(self, x, y):
        self.c_x = x
        self.c_y = y
    def distance(self, other):
        x_diff_sq = (self.c_x - other.c_x)**2
        y_diff_sq = (self.c_y - other.c_y)**2
        return (x_diff_sq + y_diff_sq)**0.5
    ① def __str__(self):
        ② return ("<" + str(self.c_x) + ", " + str(self.c_y) + ">")
```

- ① Special method (double _) for creating a string.
- ② Must return a string

The method `__str__` is called by function *print* when we try to print an object of the class

Sum two objects of class `Coordinate`

If we want to add two coordinates, we can develop a method `sum`.
However...

...it would be more practical if we could add the two coordinates by **using the symbol '+'**.

The same happens with other operations such as:

`-`, `==`, `,`, `<`, `>`, `len`

Similarly to what we have done with `__str__`, we can define the behavior of classes with other operations:

<code>__add__(self, other)</code>	⇒	<code>self + other</code>
<code>__sub__(self, other)</code>	⇒	<code>self - other</code>
<code>__eq__(self, other)</code>	⇒	<code>self == other</code>
<code>__lt__(self, other)</code>	⇒	<code>self < other</code>
<code>__len__(self)</code>	⇒	<code>len(self)</code>

Sum two objects of class `Coordinate`

```
class Coordinate():
    def __init__(self, x, y):
        self.c_x = x
        self.c_y = y

    def distance(self, other):
        x_diff_sq = (self.c_x - other.c_x)**2
        y_diff_sq = (self.c_y - other.c_y)**2
        return (x_diff_sq + y_diff_sq)**0.5

    def __str__(self):
        return "<" + str(self.c_x) + "," + str(self.c_y) + ">"

    def __add__(self, other):
        x_sum = (self.c_x + other.c_x)
        y_sum = (self.c_y + other.c_y)
        return (Coordinate(x_sum, y_sum))
```

1

1 The result is a new `Coordinate` object. Thus, we need to call the class constructor.

The power of OO programming

- OOP allows **grouping objects** that share:
 - common attributes, and
 - procedures that operate with these attributes
- Use **abstraction** to differentiate between how to implement an object and how to use it.
- Construct **layers of object abstractions** that inherit behaviors from other kinds of objects.
- Create **our own classes** of objects from the basic classes of Python.

Exercise: class `Fraction`

Create a new class to represent a number as a fraction.

The internal representation must be two integers:

- numerator
- denominator

Methods \equiv interface \equiv Behavior of the objects of class `Fraction`

- add, subtract
- conversion to string (use the symbol `'/'`, e.g. `3/4`)
- conversion to float (name: `frac2float`)
- inverse fraction (name `inv`)

Exercise: class Fraction

```
class Fraction():
    def __init__(self, x, y):
        self.num = x
        self.den = y

    def __add__(self, other):
        num = (self.num * other.den + other.num * self.den)
        den = (self.den * other.den)
        return Fraction(num, den)

    def __sub__(self, other):
        num = (self.num * other.den - other.num * self.den)
        den = (self.den * other.den)
        return Fraction(num, den)

    def __str__(self):
        return str(self.num) + "/" + str(self.den)

    def frac2float(self):
        return self.num / self.den

    def inv(self):
        return Fraction(self.den, self.num)
```