Data Science and Advanced Programming — Lecture 6b

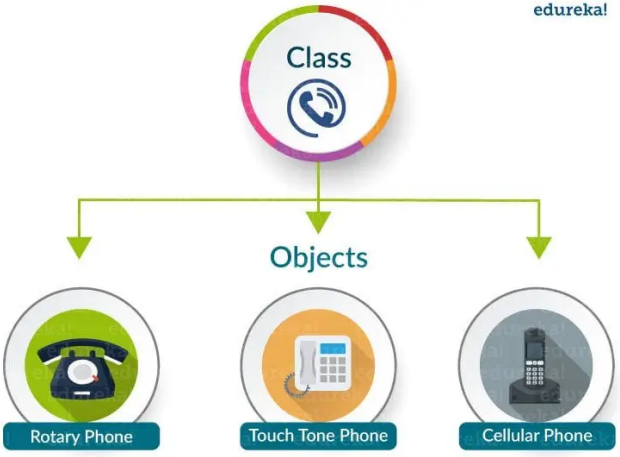
Python Fundamentals III

Simon Scheidegger
Department of Economics, University of Lausanne, Switzerland

October 20th, 2025 | 12:30 - 16:00 | Internef 263

Object Oriented Programming

https://python-textbok.readthedocs.io/en/latest/Object_Oriented_Programming.html



Object Oriented Programming

https://python-textbok.readthedocs.io/en/latest/Object_Oriented_Programming.html

- Python supports many different kinds of data
- Each of those is an object, and every object has:
  - A type
  - An internal data representation (primitive or composite)
  - A set of procedures for interaction with the object
- An object is an instance of a type
  - 1234 is an instance of an int
  - "hello" is an instance of a string

Object-oriented programming (OOP)

- EVERYTHING IN PYTHON IS AN OBJECT (and has a type).
- You can create new objects of some type.
- You can manipulate objects (e.g., append an item to a list, concatenate 2 lists, etc.).
- You can destroy objects.
  - explicitly using `del` or just "forget" about them (e.g., delete elements from a list)
  - The Python system will reclaim destroyed or inaccessible objects — called "garbage collection"

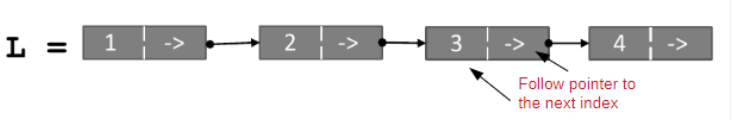
What are Objects in Programming? $\rightarrow$ "Blueprints"

Objects are a data abstraction that captures:

1. An internal representation:
   - through data attributes (e.g., what data abstractions make up an airplane, such as wings, turbines $\rightarrow$ "what data represents the plane")
2. An interface for
   - interacting with object (e.g., plane starts, lands, color of plane)
   - through methods (aka procedures/functions)
   - defining behaviors but hides implementation

An example – an object of type List

- `[1,2,3,4]` has type `list` (how is a list represented; how can you interact/what sort of operations are allowed)
- how are lists represented internally? linked list of cells:



- how to manipulate lists
  - `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()`
- internal representation should be private
- correct behavior may be compromised if you manipulate internal representation directly

The benefits of OOP

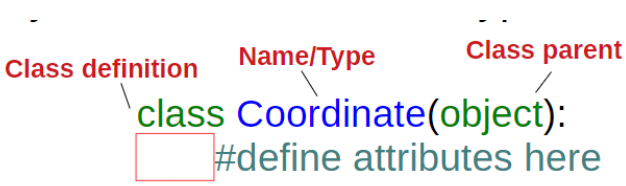
- bundle data into packages together with procedures that work on them through well-defined interfaces.
- divide-and-conquer development
  - implement and test behavior of each class separately.
  - increased modularity reduces complexity.
- classes make it easy to reuse code
  - many Python modules define new classes.
  - each class has a separate environment (no collision on function names).
  - inheritance allows subclasses to redefine or extend a selected subset of a superclass' behavior.

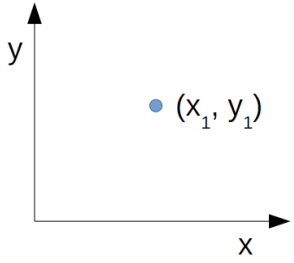
Creating and using your own Types with Classes

- make a distinction between creating a class and using an instance of the class.
- creating the class involves
  - defining the class name.
  - defining class attributes.
  - for example, someone wrote code to implement a list class.
- using the class involves
  - creating new instances of objects.
  - doing operations on the instances.
  - for example, `L=[1,2]` and `len(L)`.

How to define your own Types

- use the class keyword to define a new type:





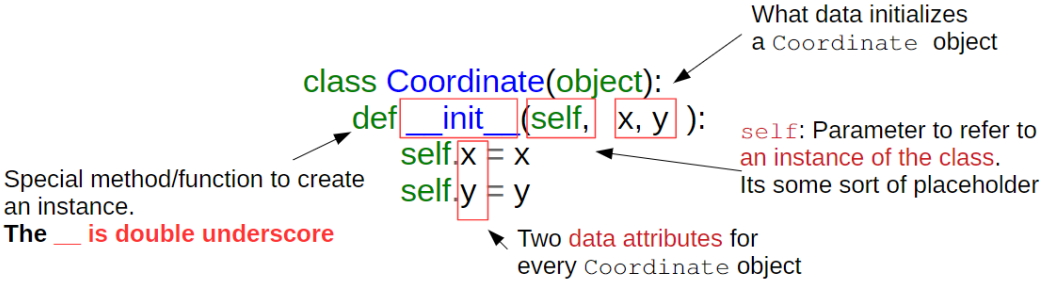
- similar to `def`, indent code to indicate which statements are part of the class definition
- the word `object` means that `Coordinate` is a Python object and inherits all its attributes (inheritance follows later in this course)
- `Coordinate` is a subclass of object
- `object` is a superclass of `Coordinate`

Attributes — data and procedures

- Attributes are data and procedures that "belong" to the class
- data attributes
  - think of data as other objects that make up the class
  - for example, a coordinate is made up of two numbers
- methods (procedural attributes)
  - think of methods as functions that only work with this class
  - how to interact with the object
  - for example you can define a distance between two coordinate objects but there is no meaning to a distance between two list objects

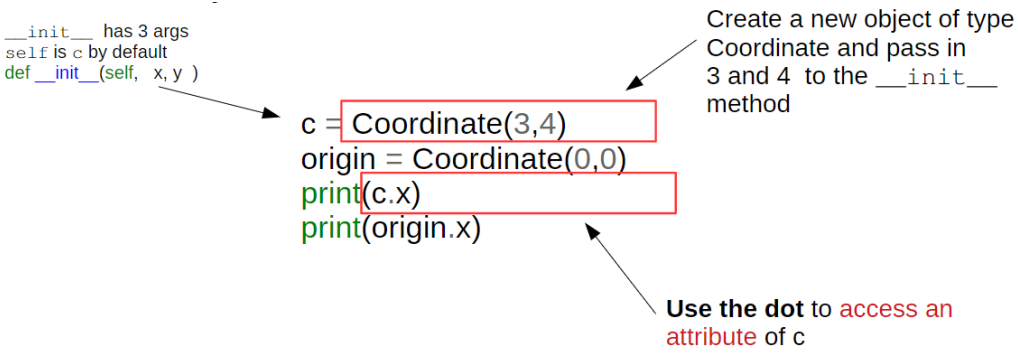
How to create an instance of a class

- first have to define how to create an instance of an object
- use a special method called `__init__` to initialize some data attributes



Creating an instance of a class

- Data attributes of an instance are called instance variables.
- Don't provide argument for `self`, Python does this automatically.



What is a method?

- Procedural attribute, like a function that works only with this class.
- Python always passes the object as the first argument
  - convention is to use self as the name of the first argument of all methods.
- the "." operator is used to access any attribute
  - a data attribute of an object.
  - a method of an object.

Let's define a Method for the "Coordinate" class

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

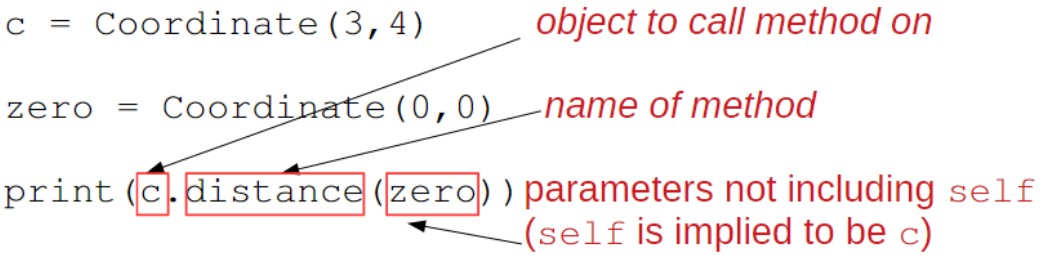
Use it to refer to any instance

Another parameter to method

Dot notation to access data

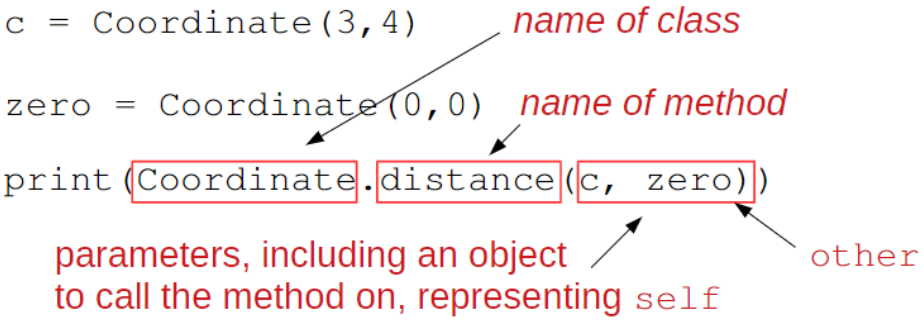
- other than self and dot notation, methods behave just like functions (take parameters, do operations, return)

How to use a Method



How to use a Method

See demo/example1.py



Print representation of an object

See demo/example1.py

```
>>> c = Coordinate(3,4)
>>> print(c)
<__main__.Coordinate object at 0x7fa918510488>
```

- Uninformative print representation by default.
- Define a `__str__` method for a class.
- Python calls the `__str__` method when used with print on your class object.
- you choose what it does! Say that when we print a Coordinate object, want to show.

```
>>> print(c)
<3,4>
```

Define your own print method

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

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

    def __str__(self):
        return "<"+str(self.x)+","+str(self.y)+">"
```

Name of special method.

Must return a string!

Wrap our heads around Types and Classes

- can ask for the type of an object instance

```
>>> c = Coordinate(3,4)
>>> print(c)
<3,4>
>>> print(type(c))
<class __main__.Coordinate>
```

return of the `__str__` method

The type of object c is a class Coordinate

- this makes sense since

```
>>> print(Coordinate)
<class __main__.Coordinate>
>>> print(type(Coordinate))
<type 'type'>
```

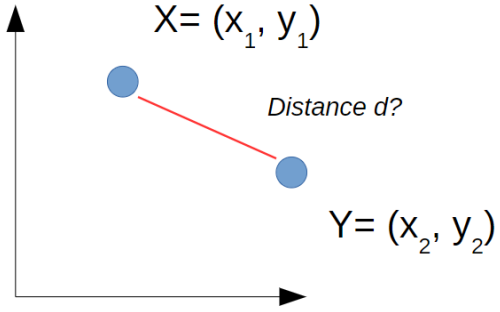
A Coordinate is a class

a Coordinate class is a type of object

- use `isinstance()` to check if an object is a `Coordinate`

```
>>> print(isinstance(c, Coordinate))
True
```

Special Operators



$X + Y = ?$

$X - Y = ?$

$\rightarrow$ We need to define such operations!

Special Operators

- `+`, `-`, `==`, `<`, `>`, `len()`, `print`, and many others
- https://docs.python.org/3/reference/datamodel.html#basiccustomization
- Like print, can override these to work with your class.
- Define them with double underscores before/after (e.g.)

An example: Fractions

`demo/example2.py`

- Create a new type to represent a number as a fraction.
- internal representation is two integers (not floats here $\rightarrow$ note the assert!).
  - Numerator.
  - Denominator.
- Interface a.k.a. methods a.k.a how to interact with Fraction objects
  - add, subtract.
  - print representation, convert to a float.
  - invert the fraction.
- Let's have a look at the code together!

A Fraction Object

demo/example2.py

```
class Fraction(object):
    """
    A number represented as a fraction
    """
    def __init__(self, num, denom):
        """ num and denom are integers """
        assert type(num) == int and type(denom) == int, "ints not used"
        self.num = num
        self.denom = denom
    def __str__(self):
        """ Returns a string representation of self """
        return str(self.num) + "/" + str(self.denom)
    def __add__(self, other):
        """ Returns a new fraction representing the addition """
        top = self.num*other.denom + self.denom*other.num
        bott = self.denom*other.denom
        return Fraction(top, bott)
    def __sub__(self, other):
        """ Returns a new fraction representing the subtraction """
        top = self.num*other.denom - self.denom*other.num
        bott = self.denom*other.denom
        return Fraction(top, bott)
    def __float__(self):
        """ Returns a float value of the fraction """
        return self.num/self.denom
    def inverse(self):
        """ Returns a new fraction representing 1/self """
        return Fraction(self.denom, self.num)
```

Another example — a set of integers as class

demo/example3.py

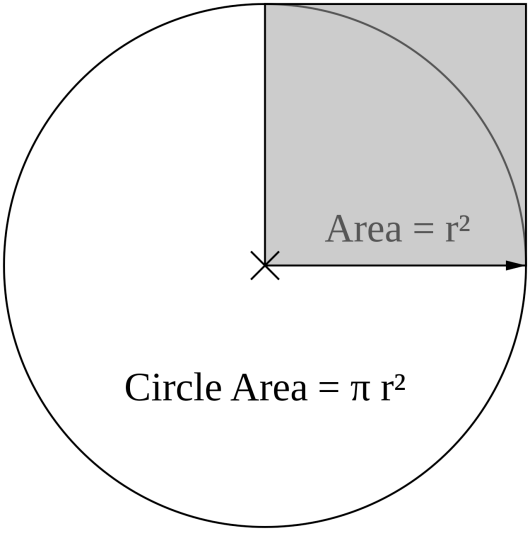
```
class intSet(object):
    """
    An intSet is a set of integers
    The value is represented by a list of ints, self.vals
    Each int in the set occurs in self.vals exactly once
    """
    def __init__(self):
        """ Create an empty set of integers """
        self.vals = []
    def insert(self, e):
        """ Assumes e is an integer and inserts e into self """
        if not e in self.vals:
            self.vals.append(e)
    def member(self, e):
        """ Assumes e is an integer
        Returns True if e is in self, and False otherwise """
        return e in self.vals
    def remove(self, e):
        """ Assumes e is an integer and removes e from self
        Raises ValueError if e is not in self """
        try:
            self.vals.remove(e)
        except:
            raise ValueError(str(e) + ' not found')
    def __str__(self):
        """ Returns a string representation of self """
        self.vals.sort()
        return '{' + ','.join([str(e) for e in self.vals]) + '}'
```

The usefulness of OOP

- bundle together objects that share
  - common attributes and
  - procedures that operate on those attributes
- use abstraction to make a distinction between how to implement an object vs how to use the object
- Build layers of object abstractions that inherit behaviors from other classes of objects.
- Create our own classes of objects on top of Python's basic classes.

Action Required — write your first Class

- Write an own class called `Circle`
- The class should:
- Take as an input the radius of the Circle
- Have a method to compute the Area of the circle: $A = \pi * r^2$
- Have a method to compute the circumference of circle, $S = 2 * \pi * r$
- Compute the ratios of the circumference and Surface for a circle with a different radius (recall: `other`)



Questions for today?