

# Scientific Programming

## Lecture A09 – Programming Paradigms

Andrea Passerini

Università degli Studi di Trento

2025/05/28

Acknowledgments: Alberto Montresor, Stefano Teso

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.



## Table of contents

- 1 Programming Paradigms
- 2 Object-Oriented Python
- 3 Functional programming
- 4 Declarative programming

# Programming Paradigms

## Imperative

**Imperative programming** specifies programs as sequences of statements that change the program's state, focusing on **how** a program should operate

- C, Pascal

## Object-Oriented

**Object-oriented programming** is based on the concept of "objects", which may contain data (**attributes**) and code (**methods**)

- Java, Smalltalk

## Declarative

**Declarative programming** expresses the logic of a computation without defining its control flow, focusing on **what** the program should accomplish

- SQL, Prolog

## Functional

**Functional programming** treats computation as the evaluation of mathematical functions, avoiding mutable state

- Haskell, OCaml, ML

# Python

Python is multi-paradigm

- Python is **imperative/procedural**, because programs are described as sequences of statements
- Python is **object-oriented**, because every piece of data is an object and new data types can be defined
- Python is **functional**, thanks to list comprehensions (maps and filters) and thanks to lambda functions
- Some libraries of Python are **declarative**, like Matplotlib

# Table of contents

- 1 Programming Paradigms
- 2 Object-Oriented Python**
- 3 Functional programming
- 4 Declarative programming

# The power 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
- Create our own classes of objects on top of Python's basic classes

## What are objects – Recap

- Python supports many different kinds of data

```
1234 3.14159 "Hello" [1, 5, 7, 11, 13]
```

```
{"CA": "California", "MA": "Massachusetts"}
```

- Each of them is an **object**, i.e. an **instance** of a type (or class)
  - 1234 is an instance of an **int**
  - "hello" is an instance of a **string**
- Every entity in Python is an object: including primitive types, functions, classes, modules, etc.

# What are types/classes – Recap

Types, or classes, are abstractions that capture:

- an internal **data representation** (primitive or composite)
  - **Data attributes**, also called **fields**
  - Think of labels that describe the content of the objects belonging to the class
  - Example: A 2-D coordinate is made up of an x-value and y-value
- an **interface** for interaction with instances of such class
  - **Function attributes**, also called **methods**
  - Think of functions used to manipulate the objects
  - Example: a distance function that computes the distance between two coordinate objects

# The lifetime of types/classes and objects

- Classes are **defined**
  - The name of data attributes is defined
  - The name and code of methods is defined
- Objects are **instantiated** from classes
- Objects are **manipulated**
- Objects are **destroyed**
  - Either implicitly through garbage collection
  - Or explicitly through the command `del`

## Objects and classes in the previous lectures

- We have not defined new types/classes
  - We have used built-in types (`int`, `list`, `dict`, etc.)
  - We have used library classes (`ndarray`, `DataFrame`)
- We have instantiated objects through:
  - built-in syntax (`L = [1,2,3,4]`)
  - class constructors (`pd.Series(["a", "b", "c"])`)
- We have manipulated objects through:
  - built-in operators (`[1,2] + [2,3]`)
  - class methods (`s.head(2)`)
- We never explicitly deleted objects (not a big deal, though...)

# Class definition

```
class Point:
```

```
    # Define attributes here
```

- The `class` keyword defines a new type
- Similar to `def`, indent code to indicate which statements are part of the class definition
- Each class **inherits** all the attributes of the predefined Python type object (more on this later)

## Class definition

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

- To define how to create an instance of object, we use a special method called `__init__` (double underscore before/after)
- `__init__` takes 1 or more parameters:
  - The first, compulsory parameter `self` is the Python mechanism to pass a reference to the object that is being created
  - `x, y` are domain parameters used to initialize the object
- `__init__` defines two data attributes:
  - `self.x` and `self.y`
- From "inside", the `."` operator is used to access attributes

## Class definition

```
c = Point(3,4)
print(c)           <__main__.Point object at 0x10dc58b00>
print(c.x, c.y)   3 4
c.x = 5
print(c.x, c.y)   5 4
```

- Creating an object is done by calling a function with the instance name and the init parameters
- As a consequence, `__init__` is called; a reference to the object (`self`) is automatically added by Python
- From "outside", the `"."` operator is used to access attributes
- Up to this point, a class is nothing more than a **named tuple**

## Defining methods

```
class Point:
```

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

```
    def distanceFromOrigin(self):  
        return ( self.x**2 + self.y**2 )**0.5
```

- The method computes the distance of the point from the origin.
- Python always passes the object as the first argument
  - BTW, the name `self` is just a convention, but an important one
- Again, the `"."` operator is used to access attributes

## Invoking methods

```
p = Point(7, 6)
print(p.distanceFromOrigin())
```

9.21954445729

- Method attributes are accessed through the dot notation, as usual

<http://interactivepython.org/courselib/static/thinkcspy/ClassesBasics/AddingOtherMethodstoourClass.html>

# Encapsulation

## Encapsulation

*The process of compartmentalizing the elements of an abstraction that constitute its structure and behavior. Encapsulation serves to separate the contractual interface of an abstraction and its implementation.*

[G. Booch]

How it works:

- We hide the details of the implementation that are not supposed to be visible outside (e.g., the internal coordinates)
- We provide methods to interact with them (e.g, read / write)

# Encapsulation – Java Example

```
public class Point {  
  
    private int x;  
    private int y;  
  
    public Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    public int getX() { return this.x; }  
  
    public int getY() { return this.y; }  
  
}
```

Java syntax:

- **public** means that everybody can access
- **private** means that values are accessible only internally

Methods `getX()`, `getY()` are **getters**

There are no **setters**: methods to modify the content

# Encapsulation in Python

```
class Point:
```

```
    def __init__(self, x, y):
        self.__x = x
        self.__y = y
```

```
    def getX(self):
        return self.__x
    def getY(self):
        return self.__y
```

```
    def setX(self, x):
        self.__x = x
    def setY(self, y):
        self.__y = y
```

## Conventions

- Hidden attributes should start with a double underscore `--`
- Use setters/getters instead
- If no modifier methods are available, the object is immutable
- IMHO: Ugly!

```
File "lecture.py", line 18:
```

```
    print(p.__x)
```

```
AttributeError:
```

```
'Point' object has no attribute '__x'
```

# Encapsulation

- The author of a class definition could decide to change the variable names of the data attributes
- If you are accessing data attributes outside the class and the class definition changes, you may get errors
- outside of the class, use getters and setters
  - good style
  - easy to maintain code
  - prevents bugs

## Defining methods – Multiple parameters

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def distance(self, other):
        x_sq = (self.x - other.x)**2
        y_sq = (self.y - other.y)**2
        return (x_sq + y_sq)**0.5
```

- The first parameter is always a reference to the object on which the computation is performed
- The other parameters could be everything, including a reference to another object of the same type
- The dot "." notation is used to access the data attributes of both **self** and the other object

## How to use a method

```
c = Point(3,4)
origin = Point(0,0)
print(c.distance(origin))
```

### 5.0

- The method `distance()` is invoked on the object `c`
- `distance()` is called with two arguments
  - Parameter `self` is equal to `c` (added automatically)
  - Parameter `other` is equal to `origin`

## Equivalent code

```
c = Point(3,4)
origin = Point(0,0)
print(Point.distance(c, origin))
```

### 5.0

- The method `distance()` is invoked on the object `c`
- `distance()` is called with two arguments
  - Parameter `self` is equal to `c` (added automatically)
  - Parameter `other` is equal to `origin`

## Print representation of an object

```
c = Point(3,4)
```

```
print(c)
```

```
<__main__.Point object at 0x10dc58b00>
```

- **Uninformative** print representation by default
- Define a `__str__()` method for a class
- Python calls the `__str__()` method when it needs a string representation of your object
- For example, it is used by `print()` function
- You choose what it does! Say that when we print a `Point` object, want to show `<3,4>`

# Print representation of an object

```
class Point:

    def __init__(self, x, y):
        self.x = x
        self.y = y

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

c = Point(3,4)
print(c)

<3,4>
```

## Instances as return values

```
class Point:

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def halfway(self, other):
        mx = (self.x + other.x) / 2
        my = (self.y + other.y) / 2
        return Point(mx, my)
```

- Methods may return a new object, by simply calling the constructor
- This method returns a point in the middle between **self** and **other**

## Special operators

- `+`, `-`, `==`, `<`, `>`, `len()`, `print`, and many others
- You can override these to work with your class
- Define them with double underscores before/after

<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 &lt; other</code>
<code>__len__(self)</code>	<code>len(self)</code>
<code>__str__(self)</code>	<code>str(self)</code>

<https://docs.python.org/3/reference/datamodel.html#basic-customization>

## Exercise: Define a Fraction class

- Create a new type to represent a number as a fraction
- internal representation is two integers
  - numerator
  - denominator
- interface a.k.a. methods a.k.a how to interact with Fraction objects
- return the sum, product (use `__add__`, `__mul__` )
- return the inverse
- print representation
- convert to a float

## Creating and printing a fraction

```
class Fraction:

    def __init__(self, top, bottom):
        self.num = top
        self.den = bottom

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

f = Fraction(3,5)
print(f)
```

3/5

## Summing two fractions

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

```
f1=Fraction(1,4)  
f2=Fraction(1,2)  
print(f1+f2)
```

6/8

## You may need...

```
# Greatest common divisor
def gcd(a, b):
    if b > a:
        return gcd(b,a)
    while b>0:
        a, b = b, a % b
    return a
```

## Summing two fractions

```
def __add__(self, other):  
    newnum = self.num*other.den + self.den*other.num  
    newden = self.den * other.den  
    common = gcd(newnum,newden)  
    return Fraction(newnum//common,newden//common)
```

```
f1=Fraction(1,4)  
f2=Fraction(1,2)  
print(f1+f2)
```

3/4

# Inheritance

## Definition – Inheritance

*Inheritance* enables new classes to "receive" the attributes of existing classes.

```
class ChildClass(ParentClass):  
    # Additional attributes here
```

- Parent attributes are **inherited** – they are made available in the child class
- Parent attributes may be **overridden** – new version are made available in the child class
- Overridden parent attributes may be referred through the parent class' name

# Inheritance and overriding

```
class Animal:
    def __init__(self, name):
        self.name = name
    def __str__(self):
        return "Animal :" + self.name
```

```
class Cat(Animal):
    def speak(self):
        print("Meow")
    def __str__(self):
        return "Cat: " + self.name
```

```
cat = Cat("Eris")
print(cat)
cat.speak()
animal = Animal("Grumpy cat")
animal.speak()
```

- `Animal` is the parent class, `Cat` is the child class
- `Cat` inherits method `__init__()` from `Animal`
- `Cat` overrides method `__str__()` with a new version

```
Cat: Eris
```

```
Meow
```

```
AttributeError: 'Animal'
object has no attribute
'speak'
```

# Inheritance rules

- Subclass can have methods with the same name as in the superclass
- For an instance of a class, look for a method name in current class definition
- If not found, look for method name up the hierarchy (in parent, then grandparent, and so on)
- Use first method up the hierarchy that you found with that method name

## Wrapping your head around classes and types

```
cat = Cat("Eris")
print(cat)                Cat: Eris
print(type(cat))         <class '__main__.Cat'>
print(Cat)               <class '__main__.Cat'>
print(type(Cat))        <class 'type'>
print(isinstance(cat, Animal)) True
print(isinstance(cat, Cat)) True
```

- There is nothing special in a class; it is just another object of type "type", that can be inspected as any other object.

## Inheritance – Another example

```
class Person:
    def __init__(self, surname, name, gender):
        self.surname = surname
        self.name = name
        self.gender = gender

    def __str__(self):
        return self.surname+" "+self.name+" (" +self.gender+")"

class Student(Person):

    def __init__(self, surname, name, gender, mark_avg):
        Person.__init__(self,surname,name,gender)
        self.mark_avg = mark_avg

    def __str__(self):
        return Person.__str__(self)+": " + str(self.mark_avg)

student = Student("Albert", "Einstein", "M", 18.5)
print(student)
```

# Table of contents

- 1 Programming Paradigms
- 2 Object-Oriented Python
- 3 Functional programming**
- 4 Declarative programming

# Functional programming

There are three main mechanisms inherited from functional programming:

- Map
- Filter
- Reduce

You have already used the first two of them through **list comprehensions**

## Functional programming – A few examples

```
map(f, list-of-inputs)
```

Applies function `f()` to `list-of-inputs`

```
print(list(map(len, ["how", "are", "you?"])))
```

```
[3,3,4]
```

```
filter(f, list-of-inputs)
```

Returns the items in `list-of-inputs` for which function `f()` returns `True`

```
def even(x):  
    return x%2 == 0  
print(list(filter(even, range(10))))
```

```
[0,2,4,6,8]
```

# Functional programming – A few examples

```
reduce(f, list-of-inputs)
```

Applies `f()` to the first two items in the list, then it applies `f()` to the result and the third item, and so on.

```
from functools import reduce
def mul(x,y):
    return x*y
print(reduce(mul, range(1,5)))
```

24

Multiplies all the items included in the range.

Equivalent to:

```
res = 1
for x in range(2,5):
    res = res*x
print(res)
```

But also to:

```
from functools import reduce
print(reduce(int.__mul__, range(1,5)))
```

## Lambda functions

Creating and naming a function is not needed, though. You can use (anonymous) **lambda functions**.

```
lambda input-parameters: expression
```

The examples above can be rewritten as follows:

```
from functools import reduce
print(list(filter(lambda x: x%2==0, range(10))))
print(reduce(lambda x,y: x*y, range(1,5)))
```

## Lambda functions and sorting

`list.sort()` accepts a `key` argument to specify a function to be called on each list element prior to make comparisons

```
# Sort case-independent
```

```
L = ['a', 'Andrew', 'from', 'is', 'string', 'test', 'This']  
L.sort(key=str.lower)
```

```
# Sort by third field
```

```
students = [ ('john','A',15), ('jane','B',12), ('tom','B',10) ]  
students.sort(key=lambda student: student[2])
```

```
# Sort by distance from origin, from closer to further
```

```
points = [ Point(1,2), Point(3,4), Point(4,1) ]  
points.sort(key=lambda point: point.distanceFromOrigin())
```

# Table of contents

- 1 Programming Paradigms
- 2 Object-Oriented Python
- 3 Functional programming
- 4 Declarative programming

# Declarative programming

- In Python, declarative programming is used in some of the libraries.
- We already mentioned Matplotlib
- We have a brief look at **regular expressions**

# Regular expressions

## Definition

A **regular expression** (or **regex**) is a string that encodes a string **pattern**. The pattern specifies which strings do match the regex.

A regex consists of both normal and special characters:

- Normal characters match themselves.
- Special characters match sets of other characters.

A string matches a regex if it matches all of its characters, in the sequence in which they appear.

# Regular expression syntax

Character	Meaning
text	Matches itself
( <b>regex</b> )	Matches the regex <code>regex</code> (i.e. parentheses don't count)
^	Matches the start of the string
\$	Matches the end of the string or just before the newline
.	Matches any character except a newline
<b>regex</b> ?	Matches 0 or 1 repetitions of <code>regex</code> (longest possible)
<b>regex</b> *	Matches 0 or more repetitions of <code>regex</code> (longest possible)
<b>regex</b> +	Matches 1 or more repetitions of <code>regex</code> (longest possible)
<b>regex</b> { <i>m,n</i> }	Matches from <i>m</i> to <i>n</i> repetitions of <code>regex</code> (longest possible)
[...]	Matches a set of characters
[ <i>c1-c2</i> ]	Matches the characters "in between" <i>c1</i> and <i>c2</i>
[^...]	Matches the complement of a set of characters
<b>r1</b>   <b>r2</b>	Matches both <b>r1</b> and <b>r2</b>

There are many more special symbols that can be used into a regex. We will restrict ourselves to the most common ones.

# Examples

- The regex `^something` matches all strings that start with "something", for instance "something better".
- The regex `worse$` matches all strings that end with "worse", for instance "I am feeling worse".
- The “anything goes” character `.` (the dot) matches all characters except the newline:
  - `."` matches all strings that contain at least one character
  - `"..."` matches all strings that contain at least three characters
  - `"^.$"` matches all those strings that contain exactly one character
  - `"^...$"` matches all those strings that contain exactly three characters.

## Examples

- The “optional” character `?` matches zero or more repetitions of the preceding regex.
  - `"words?"` matches both `"word"` and `"words"`: the last character of the `"words"` regex (that is, the `"s"`) is now optional.
  - `"(optional)?"` matches both `"optional"` and the empty string.
  - `"he is (our)? over(lord!)"` matches the following four strings: `"he is over"`, `"he is our over"`, `"he is overlord!"`, and `"he is our overlord!"`.

Parenthesis `()` are used to specify the scope of the `?`

# Examples

- The characters "\*" and "+" match zero or more or one or more repetitions of the preceding regex, respectively:
  - "Python!\*" matches all of the following strings: "Python", "Python!", "Python!!", "Python!!!!", etc.
  - "(column)+" matches: "column ", "column column ", etc. but not the empty string ""
  - "I think that (you think that (I think that)\*)+ this regex is cool" will match things like
    - "I think that you think that this regex is cool", as well as
    - "I think that you think that I think that you think that I think that this regex is cool", and so on.
- The “from  $n$  to  $m$ ” regex  $n,m$  matches from  $n$  to  $m$  repetitions of the previous regex
  - "(AB )2,3" matches "AB AB " and "AB AB AB ".

## Examples

Regexes can also match entire sets of characters (or their complement); in other words, they match all strings containing at least one of the characters in the set.

- "[abc]" matches strings that contain "a", "b", or "c". It does not match the string "zzzz".
- "[^abc]" matches all characters except "a", "b", and "c".
- "[a-z]" matches all lowercase alphabetic characters.
- "[A-Z]" matches all uppercase alphabetic characters.
- "[0-9]" matches all numeric characters from 0 to 9 (included).
- "[2-6]" matches all numeric characters from 2 to 6 (included).
- "[^2-6]" matches all characters except the numeric characters from 2 to 6 (included).
- "[a-zA-Z0-9]" matches all alphanumeric characters.

## Examples of powerful regexes

- `"^ATOM[ ]+[0-9]+ [0-9]+ [0-9]+"`:
  - A regex that only matches strings that start with "ATOM", followed by one or more space, followed by three space-separated integers.
  - "ATOM 30 42 12" matches
- `"[0-9]+(\.[0-9]+)?"`
  - A regex that matches a floating-point number in dot-notation:
  - "123" or "2.71828"
- `"[0-9]+(\.[0-9])?e[0-9]+"`
  - A regex that matches a floating-point number in mathematical format
  - "6.022e23". (It can be improved!)

## The re module

The re module of the standard Python library allows to deal with regular expression matching, for instance checking whether a given string matches a regular expression, or how many times a regular expression occurs in a string.

Returns	Method	Meaning
MatchObject	<code>match(regex, str)</code>	Match a regular expression regex to the beginning of a string
MatchObject	<code>search(regex, str)</code>	Search a string for the presence of a regex
list	<code>findall(regex, str)</code>	Find all occurrences of a regex in a string

## Example

```
import re

sequence = "AGGAGGCGTGTTGGTGGG"
match = re.search("GG.G", sequence)
if match:
    print(match.group(), (match.start(), match.end()))
else:
    print("No match!!")
```

GGAG (1, 5)

If you are interested in a single element, you can use the `MatchObject` object returned by `search()`

- `match.group()` returns the matched string
- `match.start()` returns the starting point
- `match.stop()` returns the stop point

## Example

```
import re

sequence = "AGGAGGCGTGTTGGTGGG"
for match in re.finditer("GG.G", sequence):
    s = match.start()
    e = match.end()
    print("Found", match.group(), "at", s, "-", e)
```

Found GGAG at 1 - 5

Found GGTG at 12 - 16

You can iterate over all (non-overlapping) matches using method `finditer()`

## Example

```
import re
```

```
line = """Don't forget to write your comments in the  
teaching evaluation form. You can also directly write  
to andrea.passerini@unitn.it for the first module,  
and luca.bianco@fmach.it for the second one."""
```

```
print(re.findall(r'[\w\.-]+@[ \w\.-]+', line))
```

```
['andrea.passerini@unitn.it', 'luca.bianco@fmach.it']
```

If you are interested just in the text of non-overlapping matches, you may obtain it through method `findall()`

## Example

```
import re

sequence = "AGGAGGAGTGTTCCCGGG<@GCAGGAGTGT"
match = re.search("(G.G)GG.G(...)", sequence)
if match:
    print(match.group(), (match.start(), match.end()))
    print(match.groups())
else:
    print("No match!!")
```

```
GGAGGAGTGT (1, 11)
('GGA', 'TGT')
```

re is capable to answer much more complex questions; here, we are looking for `GG.G` and we are interested in identifying what occurs before and after the match.