Writing code

10.1 Learning outcomes

Computing and Python

- Python basics.
- Object oriented programming.
- Writing understandable code.
- Plotting.

10.2 Python reminder

10.2.1 Variable, their type and their scope

In Python you do not need to declare your variables as in many other languages. Simply assign a value to them. However, they do have a type:

```
>>> i=10
>>> type(i)
<type 'int'>
>>> n=10.0
>>> type(n)
<type 'float'>
>>> s="hello there"
>>> type(s)
<type 'str'>
>>> s=10
>>> type(s)
<type 'int'>
```

Note how the last few lines show that you are free to change the type of a variable on the fly, without warning.

10.2.2 Loops and groups

In Python (unlike many other programming languages) white space characters such as spaces, tabs and newlines are important, and have meaning. Indentation is used to make groups, e.g.

```
1 for i in [1,2,5,8]:
2     print(i)
3     print("That was fun")
```

Line 2 and 3 form the *body* of the loop, and these lines will be executed once for every value 1,2,5 and 8.

If your program fails to run correctly or produces errors, checking your white-space is usually the first thing to do. Be careful to indent consistently, that is, either use spaces or use tabs, but avoid mixing the two.

10.2.3 Lists, sets, dictionaries, tuples

Collections of variables or objects can be stored in various ways, depending on your needs. In Python *lists* (ordered collections) are declared with square brackets, while unordered *sets* with non-duplicate elements are declared with curly brackets. Compare the output and especially the order of the elements in the following:

```
>>> a = {1,5,3,5,6,6,8}
>>> b = [1,5,3,5,6,6,8]
>>> print(a)
set([8, 1, 3, 5, 6])
>>> print(b)
[1, 5, 3, 5, 6, 6, 8]
```

You can add elements to lists, or replace them,

```
>>> l = [ 1,2,3.3 ]       # a list of 4 numbers
>>> l.append(4.5)          # add an item to the list
>>> print(l)
[1, 2, 3.3, 4.5]
>>> l[3] = 5.3; print(l)  # replace item [3] by 5.3
[1, 2, 3.3, 5.3]
```

Notice that the first item has index 0, the second 1 and the index of the last item of a list of n object is $n-1$. Sets work differently: you can add to them (and duplicate elements will be ignored silently), but as they are un-ordered, it does not make sense to ask for the n-th element:

```
>>> s = {1, 3, 5, 2, 8}
>>> s.add(7); s.add(8); print(s)
{1, 2, 3, 5, 7, 8}
>>> s[3]=5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'set' object does not support item assignment
```

Once can easily extract a sublist from a list by specifying a range of indices (this is standard in Python and applies to `numpy` arrays as well, as we will see later):

```
>>> l = [0,1,2,3,4,5,6,7,8,9]
>>> l[1:3]   # select index range 1 to 2 (3 is excluded)
[1, 2]
>>> l[:3]    # select range 0 to 2 (no number before : -> 0)
[0, 1, 2]
>>> l[4:]    # select range 4 to last (no number after : -> last)
[4, 5, 6, 7, 8, 9]
>>> l[:]     # select all
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> l[3:-2]  # select from 3 to 7: -2 -> 10-2=8 which is excluded -> 7
[3, 4, 5, 6, 7]
```

Sets are special cases of *dictionaries*, which associate a label (or *key*) to each element (or *value*) in the set:

```
>>> c={"apple": 10, "orange": 20, "banana": 15, "apple": 8}
>>> c["pear"]=18
>>> print(c)
{'orange': 20, 'pear': 18, 'apple': 8, 'banana': 15}
```

Note how the value of `apple` got overwritten; consistent with the idea that dictionaries are sets, you can only have one key of any given name. Finally, Python knows about something called *tuples*, indicated by round brackets, which are ordered collections of objects of different types:

```
>>> d=("hello", 10, 3.4, "apple", 10)
>>> print(d)
('hello', 10, 3.4, 'apple', 10)
```

They are useful to pass several object to a function or get a function to return several object:


```
>>> def fct(x): return(x,x**2,x**3); # function returns x, x*x, x*x*x
>>> (x,xs,xc)=fct(2)                 # assign each value to different
    variable
>>> print(x,xs,xc)
2 4 8
```

10.2.4 Functions, modules and libraries

Instead of writing your entire program as one long list of statements, it is almost always useful to separate parts of the functionality out into functions. These functions in turn can be stored in separate files; groups of functions isolated that way are called *modules*. A different word for a module (or collection of modules) is a *library*. You will create your own libraries of functions, but we will also use libraries made by others.

Functions are defined using the `def` statement, followed by the name of the function, a list of parameters in round brackets, and a colon.

```
1 def print_us(param, another):
2     print(param)
3     print(another)
```

The function content (or function body) should be indented, just like with loops.

There are 4 ways to import functions from modules.

If you store this function in a file `myfunctions.py` you can later re-use it from a different python file by *importing* it. There are various different ways to do this, each with their own advantages and disadvantages. The one causing the least problems in large programs is

Method 1: `import`

```
1 import myfunctions
2
3 myfunctions.print_us("hello", "world")
```

The 1st line imports the functionality of the `myfunctions.py` file; note that you do *not* write the `.py` extension. In the 3rd line the `print_us` function is then called. Note how you need to prefix the function name with the name of the module and a period. This always makes it clear where a given function comes from.

For modules which have long names, one can import a module and give it a simpler name (or *alias*),

Method 2: `import` with alias

```
1 import myfunctions as mf
2
3 mf.print_us("hello", "world")
```

We will often use this in the practicals (see e.g. the matplotlib examples below). The disadvantage is of course that it may make code harder to read: not everyone always agrees on the best short name for a library written by others.

If you are sure that the functions which you are importing do not have a name which already exists, you can also import them like

Method 3: `from ... import`

```
1 from myfunctions import print_us
2
3 print_us("hello", "world")
```

Note how this pulls in only one function, and it now becomes available without having to pre-fix it with the module name (see line 3). However, if a function `print_us` already existed it will be overwritten without warning. Finally, if you want to import all functions from a given module, you can also do

Method 4: `from ... import *`

```
1 from myfunctions import *
2
3 print_us("hello", "world")
```

This can be very dangerous, as you do often not know in advance which functions a module declares, and their names may clash with names of functions which you or the Python system have defined already. In such cases, the old definitions will be silently overwritten!

Finally, we will see in the practicals that libraries written by other people are often so complicated that they contain sub-modules which you need to import separately. An example is `matplotlib`, which contains a separate module `pyplot`. If you want to use the latter, you need to explicitly write `import matplotlib.pyplot`; just writing `import matplotlib` will not do the trick.

10.3 Vectors and matrices

The de-facto standard for doing vector and matrix algebra (linear algebra) in Python is to use the `numpy` library, usually by doing '`import numpy as np`' at the top of your file. There are various ways to create vectors or matrices, for instance by giving their components explicitly or by specifying that you want an vector or matrix filled with zeros or ones,

```
>>> a = np.array( [2,3,4] )         # 3-component vector
>>> a = np.array([(1.5,2),(4,2)])   # 2x2 matrix
>>> np.zeros( 3 )}                  # 3-component vector with 0s
>>> np.zeros( (3,4) )               # 3x4 matrix with 0s
>>> np.zeros( 20, dtype=np.int )}   # 20-component integer vector of 0s
>>> np.ones( 5 )                    # 5-component vector with 1s
>>> np.arange(5, 20, 2)             # [ 5, 7, 9, ..., 19]
>>> np.linspace(1.5, 7.5, 4)        # [ 1.5,  3.5,  5.5, 7.5]
```

Be aware that there is an `np.empty` function which creates vector and matrices with uninitialised (arbitrary) values; it is best to avoid this and prefer the `np.zeros` function.

10.3.1 Basic array operations

When working with numpy arrays, arithmetic operations, tests and special functions are applied element by element. Note that this means that the `*` operator multiplies objects component-by-component; it is not the inner product of matrix-vector product!

```
>>> a = np.array([1.0,2.0]); b= np.array([-2.0,5.0])
>>> a+b
    array([-1.,  7.])
>>> b-a
    array([-3.,  3.])
>>> a*b
    array([-2., 10.])
>>> a/b
    array([-0.5,  0.4])
>>> a < b
    array([False,  True], dtype=bool)
>>> np.exp(a)
    array([ 2.71828183,  7.3890561 ])
>>> a.sum()   # add all the elements
    3.0
>>> a.min()   # smallest element
    1.0
>>> a.max()   # largest element
    2.0
>>> b.prod()  # multiply all the elements
    -10.0
```

10.3.2 Accessing array elements and ranges

Array element can be accessed exactly like lists. The first item is `0`, and the last one `n-1`. For one-dimensional arrays (vectors), a few examples are

```
>>> a = np.array([1.,2.,3.])
>>> a[0] = -1
>>> a
    array([-1.,  2.,  3.])
>>> b = a            # b is a reference to a
>>> b[1] = -2.0      # this modifies a as well
>>> a
    array([-1., -2.,  3.])
>>> b = a.copy()     # b is now an independent copy of a
>>> b[1] = 5.0
>>> a
    array([-1., -2.,  3.])
>>> b
    array([-1.,  5.,  3.])
```

Be aware of the difference between line 5 and line 9! Things work essentially the same way for two-dimensional arrays (matrices),

```
1  >>> a = np.array(([1.,2.],[-2,1]))
2  >>> a[0,0] = -1
3  >>> a
4      array([-1.,2.],[-2,1])
5  >>> b = a              # b is a reference to a
6  >>> b[1,1] = 3.0       # this modifies a as well
7  >>> a
8      array([-1.,2.],[-2,3])
9  >>> b = a.copy()       # b is now an independent copy of a
10 >>> b[1,1] = 5.0
11 >>> a
12     array([-1.,2.],[-2,3])
13 >>> b
14     array([-1.,2.],[-2,5])
```

One can also access ranges of values within an array. This works with the standard Python notation for slices, as discussed above in section 10.2.3. For one dimensional arrays

```
1  >>> a = np.array([1.,2.,3.,4.,5.])
2  >>> a[1:3]          # element 1 and 2 (3 is not included)
3      array([ 2.,  3.])
4  >>> a[2:]           # from 2 to the end
5      array([ 3.,  4.,  5.])
6  >>> a[:2]           # from 0 to 2 exclusive
7      array([ 1.,  2.])
8  >>> a[:]            # all
9      array([ 1.,  2.,  3.,  4.,  5.])
10 >>> a[0:5:2]        # Even elements. from o to 5 by step 2
11     array([ 1.,  3.,  5.])
12 >>> a[::-1]         # Negative step -> reverse
13     array([ 5.,  4.,  3.,  2.,  1.])
14 >>> b = np.array([11,12,13])
15 >>> a[1:4] = b       # assign several element at once
16 >>> a
17     array([  1.,  11.,  12.,  13.,   5.])
```

For matrices ranges can be accessed for both indices simultaneously:

```
1  >>> A = np.array(([1.,2.,3.],[11,12,13],[21,22,23]))
2  >>> A
3  array([[  1.,   2.,   3.],
4         [ 11.,  12.,  13.],
5         [ 21.,  22.,  23.]])
6  >>> A[0]          # first row
7      array([ 1.,  2.,  3.])
8  >>> A[0,1:3]    # last 2 of first row
9      array([ 2.,  3.])
10 >>> A[:,1]      # second column
11     array([  2.,  12.,  22.])
12 >>> b = np.array([0.1,0.2,0.3])
13 >>> A[2] = b    # Set 3rd row of A equal to values of b
14 >>> A
15 array([[  1. ,   2. ,   3. ],
16        [ 11. ,  12. ,  13. ],
17        [  0.1,   0.2,   0.3]])
18 >>> A[:,1] = b # Set second column of A to values of b
19 >>> A
20 array([[  1. ,   0.1,   3. ],
21        [ 11. ,   0.2,  13. ],
22        [  0.1,   0.3,   0.3]])
23 >>> A = np.zeros((4,4)) # 4x4 matrix with just zeros
24 >>> B = np.array(([1, -2],[-3,4]))
25 >>> B
26 array([[ 1, -2],
27        [-3,  4]])
28 >>> A[1:3,1:3] = B # Middle of A set to values of B
29 >>> A
30 array([[ 0.,  0.,  0.,  0.],
31        [ 0.,  1., -2.,  0.],
```

```
       [ 0., -3.,  4.,  0.],
       [ 0.,  0.,  0.,  0.]])
>>> A = np.zeros((4,4)) # Set every second element to values of B
>>> A[0::2,0::2] = B
>>> A
array([[ 1.,  0., -2.,  0.],
       [ 0.,  0.,  0.,  0.],
       [-3.,  0.,  4.,  0.],
       [ 0.,  0.,  0.,  0.]])
>>> A = np.zeros((4,4)) # Same as above but reversing order of B
>>> A[0::2,0::2] = B[::-1,::-1]
>>> A
array([[ 4.,  0., -3.,  0.],
       [ 0.,  0.,  0.,  0.],
       [-2.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  0.]])
```

10.3.3 Linear algebra

The real power of numpy lies in its functionality to do various linear algebra manipulations with vectors and matrices, for instance taking inner or outer products:

```
>>> A=np.array([[1.,2.],[3.,4.]])
>>> B=np.array([[-2.0,3.],[3.,-1.]])
>>> np.dot(A,B)   # Matrix multiplication
    array([[ 4.,  1.],
           [ 6.,  5.]])
>>> x = np.array([1.,2.,3.])
>>> y = np.array([1.,0,-2.])
>>> z = np.cross(x,y) # vector cross-product
    array([-4.,  5., -2.])
>>> np.dot(x,z)       # scalar product between 2 vectors
    0.0               # as expected
>>> np.outer(x,y)
    array([[ 1.,  0., -2.],
           [ 2.,  0., -4.],
           [ 3.,  0., -6.]])
```

You can also compute matrix inverses, determinants and eigenvalues, and solve systems of linear equations,

```
>>> A=np.array([[1.,2.],[3.,4.]])
>>> C = np.linalg.inv(A)
       array([[-2. ,  1. ],
              [ 1.5, -0.5]])
>>> np.dot(A,C)
    array([[  1.00000000e+00,   0.00000000e+00],
           [  8.88178420e-16,   1.00000000e+00]])
>>> np.linalg.eig(A) # eigen values and eigen vectors
    (array([-0.37228132,  5.37228132]),
     array([[-0.82456484, -0.41597356],
            [ 0.56576746, -0.90937671]]))
>>> np.linalg.det(A)
    -2.0000000000000004
>>> np .trace(A)
    5.0
>>> b = np.array([1.0,2.0])
>>> np.linalg.solve(A,b)   # solve A x = b
    array([ 0. ,  0.5])
```

10.4 Plotting and graphics

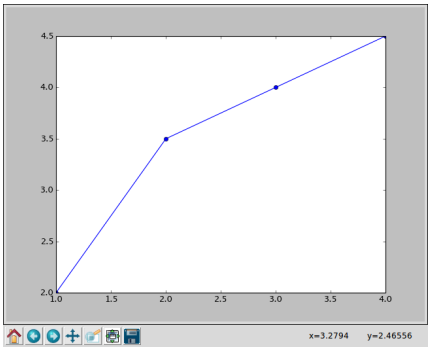
10.4.1 Introduction to matplotlib

For plotting we will always use the `matplotlib` library. This library has a very extensive set of low-level functions for construction of two- and three-dimensional plots, as well as a higher-level library which gives less flexibility but is much easier to use.This higher-level library is called `pyplot`. Unless you want to do really fancy plotting things, it is sufficient to stick to `pyplot`.

Making a plot of a set of data points in `pyplot` is as simple as

```
1 import matplotlib.pyplot as pyplot
2
3 pyplot.plot([1,2,3,4],[2,3.5,4,4.5],'-o')
4 pyplot.show()
```

The two lists correspond to the x-values and y-values of the points to be plotted. The `plt.show()` statement pops up a window like the one below,



The plotting style is determined by the `'-o'` argument to the `plot` call: the '–' symbol indicates that we want to see the dots connected with lines, and the 'o' symbol indicates that we want to draw dots at the data points. If, instead of the `pyplot.show()` statement, you use `pyplot.savefig('figure.png')`, the library will not show the figure but instead save it into a file called `figure.png`.

In order to plot a function $f(x)$, you need to generate a list of regularly spaced x-values, and another list which contains the corresponding values of $f(x)$ at those points. This is best done with the help of `np.arange` or `np.linspace` (do not create your own loops in order to create the list of x-values).

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 theta = np.arange(0., 6.28, 0.1) # array of angle values
5 f     = np.sin(theta)             # array of function values
6
7 plt.plot(theta, f) # Prepare plot f(theta)
8 plt.show()         # show the figure
```

Remember that `np.sin` acts on the entire array `theta` in one shot.

If you call `plt.plot` multiple times before you call `plt.show`, the graphs will be drawn together.

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 th1 = np.arange(0., 6.28, 0.1)
5 f1  = np.sin(t)
6
7 th2 = np.arange(0., 6.28, 0.05)
8 f2  = np.cos(t)
9
```

```
plt.plot(th1, f1,"r-",th2,f2,"g-") # f1 is read f2 is green
plt.xlabel(r'$\theta$',fontsize=22) # X labelk using latex
plt.ylabel('F',fontsize=22)        # Y label
plt.title(r'sin($\theta$) and cos($\theta$)') # Plot title using latex
# Add text inside the plot
plt.text( 1, 0.5, r'In figure',fontsize=22, color='blue')
plt.axis([0, 2*pi, -1.1, 1.1])   # specify plot range
plt.grid(True)                   # Add grid lines
plt.show()                       # show the figure
```

The example above also shows how to draw text in the plot and how to add titles and axis labels. Good plots always have axis labels! If you have many functions to plot, it is best to use some kind of construction in which the various functions sit in a list, as the example below illustrates.

```
import numpy as np
import matplotlib.pyplot as plt

x = np.arange(0., 6.28, 0.01)
# a list of 4 functions
F = [np.sin(x),np.cos(x),np.sin(x*x),x*np.sin(x)]
# a list for the colours
lines = ["r-","g-","k-","b."]

i=0;
for f in F:
  plt.plot(x,f,lines[i]) # add 1 plot at a time
  i += 1

plt.xlabel(r'x',fontsize=22)  # X
plt.ylabel('F',fontsize=22)   # Y label
plt.title(r'sin(x), cos(x), sin(x**x) and x*sin(x)')
# Add text inside the plot
plt.text( 1, -3, r'In figure',fontsize=22, color='blue')
plt.axis([0, 2*np.pi, -5, 2]) # specify plot range
plt.grid(True)                # Add grid lines
plt.show()                    # show the figure
```

10.4.2 Histograms

The following plots a histogram of the integer values of the function $\int(\sin(x) * 10)$.

```
import numpy as np
import matplotlib.pyplot as plt

x = np.arange(0., 6.28, 0.01)
F = np.array([int(x) for x in np.abs(np.sin(x)*10.0)])
F += 0.5 # shift all values by 0.5
plt.hist(F,10,(0,10)) # hist5ogram of integer distribution
plt.show()
```

10.4.3 Density Plot

Although you can plot functions of two variables (i.e. $f(x, y)$) with matplotlib (which will produce a surface in three dimensions), those plots are often better drawn as density plots.

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cm as cm

N=256
Xmin = -5
Xmax = 5
Ymin = -5
Ymax = 5
X = np.linspace(Xmin,Xmax,N) # array of X values
Y = np.linspace(Ymin,Ymax,N) # array of Y values
Z = np.empty((N,N))          # array for f(x,y)

```



```
for i in range(0,N):
  for j in range(0,N):
    # Function to plot: f = X**2 *exp(-X*2-Y*2)
    Z[j][i] = X[i]**2 *np.exp(-(X[i]**2+Y[j]**2))

# create the plot
plt.imshow(Z, extent = (Xmin, Xmax, Ymin, Ymax), cmap = cm.hot,aspect='auto')
plt.colorbar() # add colour bar on the right
plt.show()     # show the figure
```

10.4.4 Dynamically updating plots

When you call `pyplot.show()`, the plot appears in a window, and your program will halt until you close that window. This is not always what you want. Sometimes you want to display a figure, but update it dynamically as the computation proceeds. In order to do this, you need to call the `pyplot.ion()` method to turn on "interactive" mode. An example follows:

pyplot_bounce.py

This program shows a plot of two data sets which change as time progresses.

```
import matplotlib.pyplot as pyplot
import math

# Turn on interactive mode and show the plot window. We do
# this only once.
pyplot.ion()
pyplot.show()

t=0
while True:
    # Tell pyplot what we want to plot: two data sets, which both depend
    # on the value of the 't' parameter.
    pyplot.plot([0, 1, 2],   [0,  math.sin(t), 0], '-o')
    pyplot.plot([0, 1.5, 2], [0, -math.sin(t), 0], '-o')
    # Set the range of y-values.
    pyplot.ylim([-1,1])
    # Draw everything.
    pyplot.draw()
    # Remove the two drawing commands from the stack.
    pyplot.cla()
    # Update time.
    t+=0.1
```

Note the difference between telling the system what you want to plot (in this case: two different data sets for every iteration of the loop) and actually drawing this (the `pyplot.draw()` function call). Also note that you need to remove the drawing commands again after you have drawn the figure, otherwise subsequent frames will show the previous frames as well (try removing `pyplot.cla()` and see what happens).

10.5 Object oriented programming

There is a programming method which allows us to safely reuse existing code without the need to make explicit copies of existing files. It also allows us to bag variables together in a single object which can then be used easily. The programming method which allows us to do all this is called *object oriented programming*. Python contains basic objects like `float`, `strings` or `lists` but it also allows us to create our own objects like matrices, which is done by the numpy library, or any other object we fancy.

Rather than starting with a complex example like the population dynamics problem of chapter 2, we will illustrate what object oriented programming can be used for using a simple example of rectangles that can be modified and drawn on a figure.

Before we start, here is a small reminder on how to plot lines and polygons using python. The `plot` function takes 2 lists, `X` and `Y` in our example, corresponding to the x and y coordinates of the points between which the lines must be drawn. If `X` and `Y` have 2 elements each, the `plot` function draws a single line. If they have `N` elements each, the `plot` function

draws N-1 lines. The 2nd argument, b-, is a format: the letter specifies the colour , b for blue, and - requests continuous lines.

Run the program demo_rect.py and make sure you understand how it works.

file: demo_rect.py

```
import matplotlib.pyplot as plt

# coordinates of the rectangle
x1 = 0
y1 = 0
x2 = 3
y2 = 2

# list of coordinates. The first point must be repeated at the end
# to draw the last segment.
X = [x1,x2,x2,x1,x1]
Y = [y1,y1,y2,y2,y1]

plt.plot(X,Y,'b-')    # join the 5 points with lines
plt.axis('equal')     # ensure a square is a square on the screen
plt.margins(0.1, 0.1) # add some space on the edges
plt.show()            # display on screen
```

Coding task 10.1:

Copy the file demo_rect.py and name it demo_plot.py. Modify demo_plot.py so that it plots on the same figure a red triangle with edges at (0,0), (1,0) and (0,1) and a green square of size 0.5 with its bottom corner at (1,1). Hint: use 4 lists, 2 for each figure, and call the function plt.plot twice before calling plt.show().

10.5.1 A simple example of classes

For our example we will start by considering rectangles which we want to display on figures. The assumption is that we might have to plot a large number of rectangles of different colours, each with different positions and sizes. One could of course keep a number of variables to store parameter values of each rectangles, but this can quickly become tedious. The code would look something like:

```
x1 = 1;   y1 = 3; lx1 = 2; ly1 = 4;
x2 = 1.5; y2 = 4; lx2 = 5; ly2 = 2;
...
x64 = 5.3; y64 = -3.4; lx64 = 0.1; ly64 = 10;

plt.plot([x1,x1+lx1,x1+lx1,x1,x1],[y1,y1,y1+ly1,y1+ly1,y1],"k-")
plt.plot([x2,x2+lx2,x2+lx2,x2,x2],[y2,y2,y2+ly2,y2+ly2,y2],"b-")
...
plt.plot([x64,x64+lx64,x64+lx64,x64,x64],[y64,y64,y64+ly64,y64+ly64,y64],"r-"
    )
```

and we would like to replace it by something of the form

```
rec1 = Rectangle(1, 3, 2, 4)
rec1.draw("k-")
rec2 = Rectangle(1.5, 4, 5, 2)
rec2.draw("b-")
...
rec64 = Rectangle(5.3, -3.4, 0.1, 10)
rec64.draw("r-")
```

which is somewhat more compact but also a lot easier to read. Notice that we have created 64 rectangles, all of which have their own attributes.

Each of the rec1 ...rec64 is what we call an *object*: something which contains its own variables (the corners of the rectange) as well as functions which use or modify these variables (the draw function for instance). As you can see, you can have multiple objects, each with different values of their variables, sharing the same functions. The description of precisely what goes into each of these objects is called a *class* definition. A class definition

So we will start by creating a definition for a class which we will call `Rectangle`. The convention in Python is for class names to start with an upper case letter; see also 10.6.1. We will store this class definition in the module file `rectangle.py`. Again, by convention, module names which contain class definitions are lower-case versions of the class names. The variables which will go into this class are `x` and `y` for the position of the lower left corner of the rectangle and `lx` and `ly` for it sizes. Our class definition will start as follows:

file: `rectangle.py`

```
1  import matplotlib.pyplot as plt
2
3  class Rectangle:
4      def __init__(self,x,y,Lx,Ly):
5          """
6          Constructor: initialise a few internal variables
7          :param x : left coordinate of rectangle
8          :param y : lower coordinate of rectangle
9          :param Lx : horizontal size
10         :param Ly : vertical size
11         """
12         self.x = x
13         self.y = y
14         self.Lx = Lx
15         self.Ly = Ly
```

The first line is needed to be able to plot the rectangle later, so the first line of interest is the 3rd one. It starts the definition of a class which we call `Rectangle`. Below it, we start a function, `__init__`, which will be used/called to initialise the member of the class. When one creates a rectangle as in the code above (`rec1 = Rectangle(1,3,2,4)`) the function `__init__` is called and the parameters `x`, `y` , `lx` and `ly` are given respectively the values `1`,`3`,`2` and `4`. All function in a classes must have `self` as their first argument, but the value of that parameter is always set automatically by python when the function is called. This is a syntax imposed by python.

Lines 12 to 15, the parameter values are then saved into the class variables which are all preceded by the word `self`. For example `self.x = x` initialises the class variable `self.x` with the value of the parameter `x`. `self.x` is a variable stored in the class object while `x` only exists during the function call of `__init__`.

The most important class function in our example is `draw()`:

file: `rectangle.py`

```
25     def draw(self,format="k-"):
26         """Plot the rectangle using the given pyplot format
27         :param format : format for plot function
28         """
29         x_list = [self.x,self.x+self.Lx,self.x+self.Lx,self.x,self.x]
30         y_list = [self.y,self.y,self.y+self.Ly,self.y+self.Ly,self.y]
31         plt.plot(x_list,y_list,format)
```

If we ignore `self`, it only takes one optional argument : `format`, which is set to `'k-'` if no argument is given. `draw()` simply draws the rectangle using the pyplot library. It creates 2 lists containing the coordinates of the 4 edges of the rectangle and draws lines between them (the coordinates of the first point must be repeated to draw the 4th line) and then calls the `pyplot` `plot` function using the `format` passed as an argument. Notice that the variables of the `Rectangle` are prefixed with the word `self.` . This is how we must refer to the class variable both to retrieve their value and to modify them.

The class variables can also be modified : the `move` function translate the rectangle by a specified amount:

file: `rectangle.py`

```
17     def move(self,dx,dy):
18         """ Translate the rectangle by dx and dy
19         :param dx : horizontal displacement
20         :param dy : vertical displacement
21         """
22         self.x += dx
23         self.y += dy
```


Class functions can also call each other as illustrated by the function `draw_moved` which moves the rectangle, draws it and then moves it back to its original position. Notice that the class functions are also prefixed by `self.`.

file: `rectangle.py`

```
33      def draw_moved(self,dx,dy,format="k-"):
34          """ Draw rectangle in position shifted by (dx,dy)
35          :param dx : horizontal displacement
36          :param dy : vertical displacement
37          """
38          self.move(dx,dy)   # translate the rectangle
39          self.draw(format)  # plot the displaced rectangle
40          self.move(-dx,-dy) # translate the rectangle back
```

We can then use the class as follows:

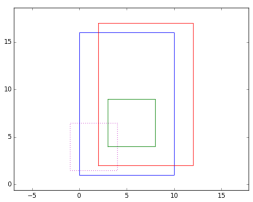
file: `rectangle.py`

```
42 if __name__ == "__main__": # ONLY EXECUTE WHEN MODULE NOT IMPORTED
43      rec1 = Rectangle(0.0,1.0,10,15) # create rectangle: x=0,y=1,Lx=10,Ly=20
44      rec1.draw("b-")                 # plot rectangle in blue
45
46      rec2 = Rectangle(3.0,4.0,5,5)   # create a square
47      rec2.draw("g-")                 # and plot it in green
48
49      rec1.move(2.0,1.0)              # move the rectangle by dx=2 dy=1
50      rec1.draw("r-")                 # plot the translated rectangle in red
51
52      rec2.draw_moved(-4,-2.5,"m:")   # plot the moved square in dotted magenta
53
54      plt.axis('equal')
55      plt.margins(0.1, 0.1)
56      plt.show()
```

The first line is a trick used so that the code below is only executed when the module is not imported from another module. This is very useful for testing a module.

To create a rectangle we call the function `Rectangle`, *i.e.* the name of the class, which actually calls the class function `__init__` and return a rectangle which we store in the variable `rec1`. To draw the rectangle we must call the class function `draw()` and this is down using `rec1.draw()`, *i.e.* the variable containing the object followed by a dot and the name of the function to call. Python automatically sets the variable `self` so that it refers to the object that calls the function. The class function will thus only use and/or modify the variables of the actual object calling the function.

Notice also that the python syntax imposes us to indent the class function within the definition of the class. All the class function definitions must be indented identically. It is also good practice to comment each function with text surrounded by triple quotes (line on line 14 and 15 for example). The starting `"""` must be indented exactly like the body of the functions.



Action 10.1:

Run the program `rectangle.py`. Modify the order of the creation, modification and drawing of the different rectangles at the bottom of the program. Try to guess the changes your modifications will make and verify them by running the program.

10.5.2 Subclasses

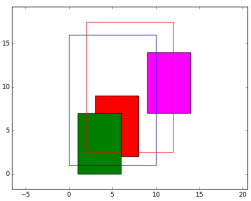
A great feature of classes is that they can be extended or customised easily avoiding a great deal of unnecessary code duplication.

To illustrate this, we will create a class containing rectangles that are filled with a colour when displayed. For this we will create a class called `RectangleFilled`. This will be very similar to the class `Rectangle` as the only class function we must change is the function

draw. We could be tempted to make a copy of the rectangle.py module and then modify it, but python allows us to do much better than that: without copying any file our new class, RectangleFilled can inherit all the variables and function of Rectangle except for those we decide to modify. This is called subclassing and is done as follows: (You do not need to understand the details on how the new draw function works at this stage.)

file: rectangle_filled.py

```
1  import matplotlib.pyplot as plt
2  import matplotlib.patches as patches
3  from rectangle import Rectangle
4
5  class RectangleFilled(Rectangle): # RectangleFilled: a subclass of Rectangle
6
7      def draw(self,format):
8          """Plot the rectangle using the given pyplot format
9          :param format : format for ploting function
10         """
11
12         # translaste "x-" into a colour name
13         cols = { "k":"black", "r":"red", "g":"green", "b":"blue",\
14                  "c":"cyan","m":"magenta","y":"yellow",}
15         colour = cols[format[0]] # use only the letter
16
17         currentAxis = plt.gca()
18         currentAxis.add_patch(patches.Rectangle(
19                             (self.x, self.y), # (x,y)
20                             self.Lx,       # width
21                             self.Ly,       # height
22                             facecolor=colour # needs a colour name
23                             ))
24
25
26 if __name__ == "__main__": # ONLY EXECUTE WHEN MODULE NOT IMPORTED
27     rec1 = Rectangle(0.0,1.0,10,15) # create rectangle: x=0,y=1,Lx=10,Ly=20
28     rec1.draw("b-")                 # plot rectangle in blue
29
30     rec2 = RectangleFilled(3.0,2.0,5,7) # create a red filled rectangle
31     rec2.draw("r-")                     # plot red rectangle
32
33     rec1.move(2.0,1.5)              # move the 1st rectangle by dx=2 dy=1.5
34     rec1.draw("r-")                 # plot the translated rectangle in red
35
36     rec2.move(-2.0,-2.0)            # move the 2nd rectangle by dx=dy=-2
37     rec2.draw("g-")                 # plot the translated rectangle
38
39     rec2.draw_moved(8.0,7.0,"m-")   # draw move in magenta
40
41     plt.axis('equal')
42     plt.margins(0.1, 0.1)
43     plt.show()
```



On line 3 we import the class Rectangle from our module rectangle and on line 5, we define the class RectangleFilled as being a subclass of the class Rectangle. This is what saves us from having to copy anything at all.

On line 7 we redefine the class function draw so that it plots a filled in rectangle (the details are not important).

Below line 27 we create some Rectangle and RectangleFilled. In line 31, rec2.draw("g-") calls the draw function from the class RectangleFilled.

Notice also that line 39 calls the class function draw_moved defined in the class Rectangle and that it itself calls the class function draw from the class RectangleFilled as rec2 is an instance of that class. (Make sure you understand this)

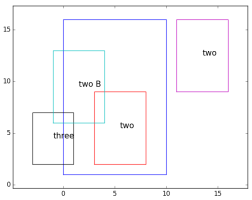
We should also point out that if we decided to add or modify a class function (to fix a bug for example) to the class Rectangle the change will be automatically made for the class RectangleFilled as well.

As another example, we will create another sub-class, called rectangle_named, of the class rectangle. The only difference is that each rectangle in that new class will have a name

that will be displayed at the centre of the rectangle. We must thus modify the class function `__init__`, which must take the rectangle as an extra argument, the class function `draw()` and we will also add the class function `reset_name` to modify the name of the rectangle.

file: `rectangle_named.py`

```
1  import matplotlib.pyplot as plt
2  import matplotlib.patches as patches
3  from rectangle import Rectangle
4
5  class RectangleNamed(Rectangle):
6      def __init__(self,x,y,Lx,Ly,name):
7          """
8          Constructor: initialise a few internal variables
9          :param x : left coordinate of rectangle
10         :param y : lower coordinate of rectangle
11         :param Lx : horizontal size
12         :param Ly : vertical size
13         :param name: rectangle name
14         """
15         # call __init__() from parent class : Rectangle
16         super(RectangleNamed,self).__init__(x,y,Lx,Ly)
17         self.name = name
18
19     def reset_name(self,name):
20         """ Modify the rectangle name
21         :param name: rectangle new name
22         """
23         self.name = name
24
25     def draw(self,colour):
26         """Plot the rectangle using the given colour
27         :param colour: drawing colour
28         """
29         # call draw() from parent class : Rectangle
30         super(RectangleNamed,self).draw(colour)
31
32         # coordinate of middle of rectangle: where to write the name
33         x = self.x+self.Lx/2
34         y = self.y+self.Ly/2
35
36         plt.gca().text(x,y, self.name, fontsize=15)
37
38 if __name__ == "__main__": # ONLY EXECUTE WHEN MODULE NOT IMPORTED
39     rec1 = Rectangle(0.0,1.0,10,15) # create rectangle: x=0, y=1, Lx=10, Ly
           =20
40     rec1.draw("b-")                 # plot rectangle in blue
41
42     rec2 = RectangleNamed(3.0,2.0,5,7,"two") # create named rectangle
43     rec2.draw("r-")                      # plot named rectangle
44
45     rec3 = RectangleNamed(-3.0,2.0,4,5,"three") # create name rectangle
46     rec3.draw("k-")                      # plot named rectangle
47
48     rec2.draw_moved(8.0,7.0,"m-") # draw moved named rectangle
49
50     rec2.reset_name("two B")
51     rec2.draw_moved(-4.0,4.0,"c-")
52
53     plt.axis('equal')
54     plt.margins(0.1, 0.1)
55     plt.show()
```



On line 6 we redefine the class function `__init__` so that it takes an extra argument: the name of the rectangle. The function then calls the `__init__` function of the class `Rectangle` (line 16) and then saves the name in the class variable `self.name`.

The `draw()` function, line 25, starts by calling the `draw()` function from the parent class `Rectangle` to draw the edges of the rectangle and then it writes the name of the rectangle at its centre.

Homework 10.1:

List 3 useful features of classes and subclasses.

10.6 Writing readable code

10.6.1 Naming variables, functions, classes

http://visualgit.readthedocs.io/en/latest/pages/naming_convention.html

10.6.2 Long functions

When writing long function with complex functionality, a good programing practice consist in spliting the functions in 2 or more smaller function. This makes the program easier to read and to check.

10.6.3 Layout

When entering lists or function arguments, insert a space between a coma and the next item. So for example

```
a = [1,2,3,4]
b = myfunction(23,1.2,4)
```

is bad, and should be replaced by the following:

```
a = [1, 2, 3, 4]
b = myfunction(23, 1.2, 4)
```

10.6.4 Comments

Comments are for people who want to know how your code works and must describe what is being done or why it is done the way it is. Comments must not state the obvious. For example

```
x2 = x1 + dx # add x1 and dx
y2 = y1 + dy # add y2 and dy
```

is useless as one can read the code. Instead one should write

```
x2 = x1 + dx # Translate [x1,y1] by [dx,dy]
y2 = y1 + dy #
```

10.6.5 Docstrings

A nice feature of Python is that one can write comments in a module which can easily be read by user of the module without reading the source file. For example one can do

```
import numpy as np
help(np.array)
```

which will display a help message for the `numpy array function`.

We can create docstrings for our own functions and classes easily: they are enclosed between 2 sets of triple double quotes:

```
def mypower(a,b):
    """ Raises a to the power b

    :param a: function argument
    :param b: power
    """
    return(np.power(a,b))
```

Docstrings are the place where you should document the meaning of function parameters. We use the Sphynx [1] notation meaning that each parameter is listed as

```
:param VARIABLE_NAME: DESCRIPTION.
```

10.7 References

[1] Georg Brandl et al. *Sphynx: Python documentation generator*. URL: http://www.sphinx-doc.org/en/stable/.

