Chapter 2
Getting Started with Programming

In this text we integrate the use programming techniques and tools in our study of physics. You will therefore need to know a few programming basics in order to profit from this approach. However, if you do not have a relevant background in introductory scientific programming, do not despair. Experience shows that you can learn to program through your first physics course—many students have done this successfully and with good results. In order to prepare you for the main text, this chapter provides an introduction to programming.

2.1 A Python Calculator

We are in this text using the editor Spyder to work with Python. (Alternatively you can use iPython—type `ipython` in a terminal window to start). When you start Spyder, you get a window where you can type commands to be executed. Click on the `Console window` in the lower right corner and type:

```
>> 9*4
36
```

Notice the difference between the text *you* type, which is preceded by », and the results generated by the program, which are typeset without indentation.

Python can be used as an advanced calculator by typing expressions on the command line:

```
>> 3*2**3+4
28
```

Standard operators are plus (`+`), minus (`-`), multiplication (`*`), division (`/`), and power (`**`). Powers of ten are input using `e`:

```
>> 4.5e4
45000
>> 2.5e-10
2.5e-10
```

which also shows how Python displays numbers.

A. Malthe-Sørenssen, *Elementary Mechanics Using Python*,
Undergraduate Lecture Notes in Physics, DOI 10.1007/978-3-319-19596-4_2

Python has most mathematical functions and constants built in, such as `pi`, `cos`, `sin`, and `exp`: To use them we need to load the `pylab` module first:

```
>> from pylab import *
```

After this we are ready to use the mathematical functions:

```
>> 4*pi
12.566370614359172
```

Python uses radians for the trigonometric functions:

```
>> sin(pi/6)
0.49999999999999994
```

As you can see, Python does not always round off as you may expect, even though the answer is close to the exact answer ($sin(\pi/6) = 0.5$). You can find a list of useful syntax, functions and expressions in the summary.

Python becomes more useful when you have a formula you want to use. For example, you may want to use the formula:

$$T_F = \frac{9}{5}T_C + 32\ , \tag{2.1}$$

to find the temperature, T_F, in Fahrenheit, given the temperature T_C in centigrade. We may type this formula directly into Python

```
>> TF = 9/5*TC + 32
Traceback (most recent call last): File "<stdin >"...
NameError: name "TC" is not defined
```

Ooops. That did not work, because Python does not yet know the value of T_C. We give T_C a value and retype the formula:

```
>> TC = 40.0
>> TF = 9.0/5.0*TC + 32.0
```

To see the answer, type

```
print TF
104.0
```

Note that we are typing `.0` after each number. This is because Python differs between integers ($1, 2, 3, 4, \ldots$) and real numbers, ($1.0, 2.3, 4.9, \ldots$). Typing `.0` or only `.` (a dot) after each number ensures that the numbers are real. Also notice that integer division is not the same as real division. With integer division $9/5 = 1$, while with real numbers $9.0/5.0 = 1.8$.

Instead of retyping the formula, you can use the up arrow to find your previously typed commands and execute them again. We have now defined a variable, `TC`. You can see the value given to `TC` by typing:

```
>> print TC
40
```

Notice that we assigned a value to the variable `TF` through a calculation. We have not introduced a function for `TF`. What does this mean? It means that if you change the value of `TC`, the value of `TF` will not change automatically unless you retype

the formula for TF. You can check this by assigning TC a new value, and then ask Python for the value of TF:

```
>> TC = 50
>> print TF
104.0
```

This is an important aspect of a programming language such as Python a variable does not change value unless you assign a new value to it!

2.2 Scripts and Functions

However, we do not want to type in the whole formula each time we want to calculate a new value for T_F. Instead we can make a *script*, a group of several statements, or a *function*, similar to an internal function such as sin.

Scripts

We can group several statements into a *script*, which we can reuse. You do this by opening a new file in the File menu in Spyder: File → New file.... This opens a new window with an editor. Here you can now type (or copy) the commands we already used:

```
TC = 40.0
TF = 9.0/5.0*TC + 32.0
print TF
```

Now, we need to save the script. In the editor window you do: 'File'→'Save'. You must give the script a name and choose where to place it. This will generate a file with an extension .py - we call such a file a py-file, because it shows that the file contains a Python script/program. You run the program from the editor window by typing the F5 key. A dialog box will show up. Select Execute in current ..., leave all other options as they are and click the Run button. As a result the commands in the script are executed as if they were typed into the Python window, and the resulting output is shown in the Python window:

```
104.0
```

You can now change the value of TC in the script and rerun the script to redo the calculation for another temperature. Notice that we wrote the script so that the temperature TC is assigned inside the script. This means that if you change the value of TC on the command line, for example by typing:

```
>> TC = 45.0
```

and then run the script—the script will not use this new value of TC, but instead use the value from inside the script.

Writing scripts to solve simple problems will be our standard operating procedure throughout this text. This is an efficient way to develop a simple program, change the parameters (such as changing `TC`), and rerun the program with new parameters. While this is practical for developing short programs and solving simple problems, it is not good programming practice. In general, we encourage the development of good programming practices, but in this text we will prioritize making the code as simple as possible.

Functions

From a programming perspective, it is better to introduce a *function* to calculate the temperature. A user defined function acts just like a predefined mathematical function such as `sin` or `exp`. We define a function by opening a new py-file: Push `File` → `New File ...`. We define a function by typing the following into the editor:

```
def convertF(TC):
    # Converts from centigrade to Fahrenheit
    TF = 9.0/5.0*TC + 32.0
    return TF
```

and save it with the name `convertF.py`.

What do these statements do? We define a function by the command `def` followed by the name of the function, the input arguments. The line must end with a colon, `:`. Here the only argument is the temperature in centigrades, `TC`. Inside the function we must calculate the value of `TF`, because this is the value the function is supposed to calculate. Finally, we specify that the function will return the value of the variable `TF`.

We call our new function by typing:

```
>> convertF(45.0)
113.0
```

Notice that Python requires each such used-defined function to either be in the same file as the script using the function, of the function must be in a separate file, and that the file must be in the search path for Python. This means that the file `convertF.py` must be in the current directory or in the standard Python directory for this to work. I suggest that you always save the functions you need in the same directory as you save the scripts you are currently working on, and that you make new directories for each problem.

A particular feature of functions is that the internal calculations and variables used inside the function are lost as soon as the function is finished. For example, Python may use several calculation steps if we call the `sin` function, but this is hidden from us. Outside the function, we only see the result of the function. To illustrate this, we could break our short function into several steps:

```
def convertF2(TC):
    # Converts from centigrade to Fahrenheit
    ratio = 9.0/5.0
    constant = 32.0
    TF = factor*TC + constant
    return TF
```

Here, we have introduced two internal variables, `ratio` and `constant`, that are forgotten as soon as the function is finished. For example, if we type:

```
>> convertF2(45.0)
113
>> print ratio
Traceback (most recent call last): File "<stdin >", ...
NameError: name "ratio" is not defined
```

we see that Python does not know the value of `ratio` after the function has done its work.

Functions are powerful and necessary tools of more advanced programming techniques and will be gradually introduced throughout the text. But initially we try to make the programs as simple as possible, and we will therefore use simple scripting as our main tool.

2.3 Plotting Data-Sets

Python not only works as a numerical calculator, it also has advanced data visualization capabilities. For example, as part of a laboratory exercise you may have measured the volume and mass of a set of steel spheres. You number the measurements using the index, i, and record masses m_i and volumes V_i in Table 2.1, where we have used that 1 litre = 1 l = 1 dm^3.

Such a sequence of numbers are stored in an *array* (or a *vector*) in Python. We define the sequence of masses and volumes in Python using

```
>> m = array([1.0,2.0,4.0,6.0,9.0,11.0])
>> V = array([0.13,0.26,0.50,0.77,1.15,1.36])
```

We can find an individual mass value by:

```
>> print m[0]
1.0
>> m[3]
6.0
```

Table 2.1 Measurement of masses m_i as a function of volumes V_i

i	1	2	3	4	5	6
m_i (kg)	1	2	4	6	9	11
V_i	0.13 l	0.26 l	0.50 l	0.77 l	1.15 l	1.36 l

There are now 6 values for the masses, numbered `m[0]` to `m[5]`. Notice that Python starts enumeration at 0, so that 0 is the first element and 5 is the last element. We call the array `m` a 6×1 array or a vector of length 6. The volumes are stored the same way:

```
>> V[0]
0.1300
>> V[3]
0.7700
```

The enumeration of the two arrays is identical: element `m[3]` of the masses corresponds to element `V[3]` of the volumes.

The relation between m and V is illustrated by plotting V as a function of m. This is done by the `plot` command:

```
>> plot(m,V,'o')
```

where the string `'o'` ensures that a small circle is plotted at each data-point. The plot command makes a "scatter" plot—it contains a point for each of the data-points $m(i)$, $V(i)$ in the two arrays. The two arrays must therefore be the same length—they must have the same number of elements. We annotate the axes by:

```
>> xlabel('m (kg)')
>> ylabel('V (l)')
```

where the `xlabel` refers to the first array `m` in `plot(m,V,'o')`, and the `ylabel` refers to the second array—the `V` array. The resulting plot is shown in Fig. 2.1.

Where did the units (kg and liters) go when we defined the mass `m` and the volume `V`? We cannot use units when we introduce digital representations of the numbers. We can only input numbers into Python and we have to keep track of the units. This is why we specified the units along the axes in the `xlabel` and `ylabel` commands.

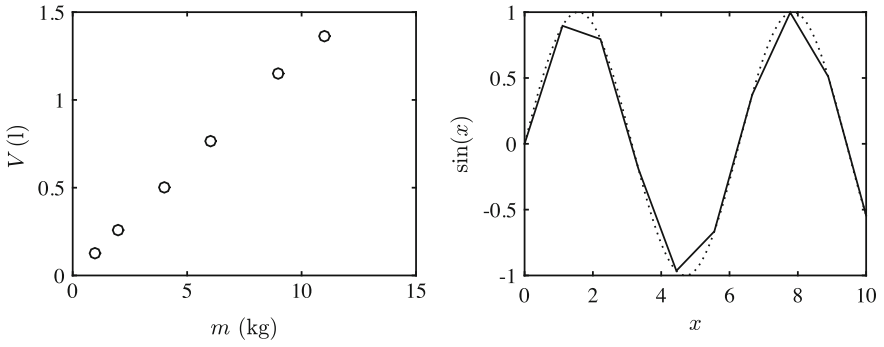

Fig. 2.1 The plot of V as a function of m (*Left*) and with the plot of $\sin(x)$ as a function of x for 10 points (*solid line*) and for 1000 points (*dotted line*) (*Right*)

Table 2.2 Sequence of i, x_i and $\sin(x_i)$

i	1	2	3	4	5	...	n
x_i	0.0	0.1	0.2	0.3	0.4	...	10.0
$\sin(x_i)$	sin(0.0)	sin(0.1)	sin(0.2)	sin(0.3)	sin(0.4)	...	sin(10.0)

2.4 Plotting a Function

Python cannot plot a function such as $\sin(x)$ directly. We must first generate two sequences of numbers, one sequence for the x'es and one sequence for the corresponding values of $\sin(x)$, and then plot the two sequences against each other. While this may sound complicated, Python has functions that ensure that you can almost directly write the mathematical expression into Python.

Loops

We want to make a sequence of x'es, such as 0.0, 0.1, 0.2, 0.3, ... etc., and then for each x_i we want to calculate the corresponding value for $\sin(x_i)$ in Table 2.2, where we generate x_i from 0.0 to 10.0 in steps of 0.1.

How do we generate such an array in Python? First, we have to generate the array.[1] How many elements do we need? Going from 0.0 to 10.0 in steps of 0.1 we need:

$$n = \frac{10.0 - 0.0}{0.1} + 1 \,, \tag{2.2}$$

steps, where we have added one in order to include the last step (otherwise we would stop at 9.9 instead of at 10.0. We define an array of this length by:

```
>> n = int(ceil((10.0-0.0)/0.1)+1)
>> x = zeros((n,1),float)
```

Here, the function `ceil()` rounds up. Notice the use of `int` in order to ensure that we return an integer and not a real. The function `zeros((n,1),float)` generates and returns an array of size `n` by `1` which is filled with zeros. Now we need to fill the array:

```
>> x[0] = 0.0
>> x[1] = 0.1
>> x[2] = 0.2
>> x[3] = 0.3
>> x[4] = 0.4
...
```

[1] In Python it is not necessary to define the size of the array before it is filled. We could just fill it as we go along, but this is not good coding practice, it will lead to very slow codes for large arrays, and may cause surprising errors in your programs. We will therefore always predefine the size of arrays.

Fortunately, there is a more efficient way of doing this—by using a `for`-loop. A `for`-loop allows us to loop through a list of values 0, 1, 2, ..., $n-1$ for the variable `i`, and then execute a set of commands at each step—exactly what we need. We can replace the long list of `x[0] = 0.0` etc. by the loop:

```
>> for i in range(n):
...     x[i] = i*0.1
```

If you type this in, nothing will execute until you press `Enter` twice. Note also that you need to indent the first line after the colon in the line with the `for`-statement. It is only the set of commands that are indented that are part of the loop and that are run several times. Indenting the line means that you add four spaces to the beginning of the line (compared with `for` in the `for`-statement). Indentation is the way Python recognizes a block of commands.

You can check the generated values of `x` by:

```
>> print x
[[ 0. ]
 [ 0.1]
 [ 0.2]
 [ 0.3]
 [ 0.4]
 [ 0.5]
...
```

Notice how we specify the range of the loop, by specifying a sequence of numbers by use of the `range(n)` function. Typing `range(n)` at the command prompt gives you exactly the list of values for `i`. In Python it is important that `n` is an integer. If we had not been careful to define `n` as an integer above, we would have need to do it here by typing `range(int(n))`.

Now, let us put this into a small script. And let us also calculate the value for the function $\sin(x)$:

```
from pylab import *
x0 = 0.0, x1 = 10.0, dx = 0.1
n = int(ceil((x1-x0)/dx) + 1)
x = zeros((n,1),float)
y = zeros((n,1),float)
for i in range(n):
    x[i] = x0 + i*dx
    y[i] = sin(x[i])
plot(x,y)
show()
```

Which both generates and plots the function $\sin(x)$. The variables `x0`, `x1`, and `dx` provide the start, stop and step of the x-values used. You can now change them and rerun the script to generate plots for other ranges or with other resolutions.

Notice that `y[i] = sin(x[i])` must appear inside the loop—that is before the end. Otherwise it would only be executed once, using the value `i` had at the end of the loop. Putting commands outside a loop that should be inside a loop is a common mistake—sometimes also done by experienced programmers.

The `While`-Loop

The `for`-loop is probably the loop-structure you will use the most, but there are also other tools for making a loop, such as the `while`-loop. In the `while`-loop the commands inside the loop are executed until the expression in the `while` command is true. It does not automatically update a counter either. For example, we can change the script to calculate $\sin(x)$ to use a `while`-loop instead by the following changes:

```
...
y = zeros((n,1),float)
i = 0
while i<=n:
    x[i] = x0 + i*dx
    y[i] = sin(x[i])
    i = i + 1
...
```

Notice that we must assign `i=0` before the loop and `i=i+1` inside the loop, since we now need to update the counter "manually" inside the loop. We also introduce an "expression" `i<=n` which may be false (having the value 0) or true (having the value 1). The loop continues until the expression becomes false. Notice that a common source of error is to generate `while` loops that continue forever, for example because you have forgotten to update the counter inside the loop. You will notice this because your program never ends: Python will never stop or plot your results.

You may wonder what the point of the `while`-loop is, since it looks more cumbersome than the `for`-loop. We will use the `while` loop when we want to continue a calculation for an unknown number of steps. For example, you may want to find the motion of a falling ball up to the point where it hits the ground. However, you may not know beforehand how many steps are needed before it hits the ground. If the position of the ball is $x(t) = 1000 - 4.9\,t^2$, we can calculate the position as a function of time in steps of dt as long as x is positive using the following program:

```
from pylab import *
t0 = 0.0, t1 = 10.0, dt = 0.01
n = int(ceil((t1-t0)/dt) + 1)
t = zeros((n,1),float)
y = zeros((n,1),float)
t[0] = t0
y[0] = 100.0-4.9*t[0]**2
i = 0
while y[i]>0.0:
    i = i + 1
    t[i] = t0 + i*dt
    y[i] = 100.0-4.9*t[i]**2
#stop1
plot(t[0:i],y[0:i])
xlabel('t [s]'); ylabel('y [m]');
```

This script needs some explanation. First, we notice we update both now is `t` and `y` and not only `x` as before. In addition, we see that we calculate the value of `y[0]` before the loop starts, otherwise the loop would never start. This is also a common mistake. Therefore, ensure that you understand why and what is done before the `while`-loop starts in this script.

Vectorization

While loops are generally powerful and useful methods, there is a much simpler way to generate sequences of numbers and plot functions in Python—and the method also allows your code to follow the mathematical formulas more closely. This method is called *vectorization*.

We can make a sequence of x'es in several ways using functions that are built into Python instead of using a loop. For example, the function `linspace` generates a sequence of equidistant numbers from `0.0` to `10.0`:

```
>> x = linspace(0,10,10)
>> print x

[ 0.          1.11111111 2.22222222 3.33333333 4.44444444
  5.55555556 6.66666667 7.77777778 8.88888889 10.        ]
```

In this case we generated 10 numbers, but you are free to choose your own resolution. An alternative to specifying the number of points you want, as we do with `linspace`, is to specify the step size. The expression `r_[0.0:10.0:0.3]` returns an array starting at the value 0 and ending at 10.0 in steps of 0.3[2]:

```
>> x = arange(0.0, 10.0, 0.3)
array([ 0. , 0.3, 0.6, 0.9, 1.2, 1.5, 1.8, 2.1, 2.4, 2.7, 3. ,
        3.3, 3.6, 3.9, 4.2, 4.5, 4.8, 5.1, 5.4, 5.7, 6. , 6.3,
        6.6, 6.9, 7.2, 7.5, 7.8, 8.1, 8.4, 8.7, 9. , 9.3, 9.6,
        9.9])
```

Ok—so that was simply an easier way of generating the array `x`. Why all the fuzz? Because of a powerful feature of Python called vectorization: we can apply the function $\sin(x)$ to the whole array `x`. Python will then apply the function to each of the elements in `x` and return a new array with the same number of elements as `x`. The three lines:

```
>> x = linspace(0,10,10)
>> y = sin(x)
>> plot(x,y), show()
```

are equivalent to the program:

```
from pylab import *
x0 = 0.0
x1 = 10.0
dx = 1.0
n = int(ceil((x1-x0)/dx) + 1)
x = zeros((n,1),float)
y = zeros((n,1),float)
for i in range(n):
    x[i] = x0 + (i-1)*dx
    y(i) = sin(x[i])
plot(x,y)
show()
```

[2]Notice the small difference between the two methods: using `linspace` ensures that the first and the last numbers are included in the list, but when you use `arange(0.0, 10.0, 0.3)` the last number is 9.9 and not 10.0!.

Notice how simple the vectorized code is—it is almost identical to the mathematical formula. We only have to define the range of `x`-values before we call the `sin(x)` function. Beautiful and powerful.

The program above generates the solid line in Fig. 2.1. However, this plot has too sharp corners, because we have too few data-points. Let us generate 1000 x-values and plot $\sin(x)$ with this resolution in the same plot:

```
>> x = linspace(0,10,1000)
>> y = sin(x)
>> hold('on')
>> plot(x,y,':');
>> hold('off')
>> show()
```

The result is shown in Fig. 2.1 with a dotted line. Here, we have used a few more tricks. We use the command `hold('on')` to ensure that Python does not generate a new plot, which would remove the previous one, but instead plots the data in the same plot as we have already used. Typing `hold('off')` stops this behavior—otherwise all subsequent plots will be part of the same plot. We have also used the string `':'` to tell Python that we want a dotted line. You can find more plotting methods in the summary at the end of the chapter.

The vectorization technique is very general, and usually allows you to translate a mathematical formula to Python almost formula by formula. For example, we plot the function

$$f(x) = x^2 e^{-ax} \sin(\pi x) \,, \tag{2.3}$$

from $x = 0$ to $x = 10$ by typing (when $a = 1$):

```
>> x = linspace(0,10,1000)
>> a = 1.0
>> f = x**2*exp(-a*x)*sin(pi*x)
>> plot(x,f), show()
```

As soon as you have learned to transcribe mathematical expressions from the mathematical notation to Python you are ready to calculate and plot any function.

The technique of vectorization is a powerful and efficient technique. Python is usually very fast at calculating vectorized commands. And we can write very elegant programs using such techniques, ensuring that the Python code follows the mathematical formulation closely, which makes the code easy to understand.

2.5 Random Numbers

Sometimes you need randomness to enter your physical simulation. For example, you may want to model the motion of a tiny dust of grain in the air bouncing about due to random hits by the air molecules, so called Brownian motion. We model this by having the grain move a *random* distance during a given time interval. How do we create random numbers on the computer? Unfortunately, we cannot generate really

random numbers, but most programs have decent pseudo-random number generators, which generate a sequence of numbers that appear to be random. In Python we can simulate the throw of a dice using

```
>> randint(6) + 1
3
```

where `randint(n)` generates a random integer between 0 and $n-1$, where each outcome has the same probability. If you type the command several times, you will get a new answer each time. Python includes several functions that return random numbers: It can generate random real numbers between 0 and 1 using the `rand` function and normal-distributed numbers (with average 0 and standard derivation 1) using the function `randn`.

2.6 Conditions

Now, if we return to discuss the motion of the grain of dust, we want to model its motion according to a simple rule: I throw a dice. If I get between 1 and 3, the grain moves a step forward, otherwise it moves a step backward. How can we handle such conditions? We need a set of conditional statements, so that we can perform a given set of commands when a particular condition is fulfilled. We need an `if`-statement:

```
if (expr):
    <statement a1>
    <statement a2>
    ..
else:
    <statement b1>
    <statement b2>
    ..
```

Here the expression `(expr)` is an expression such as `randint(6)+1>3` which may be true or false. If the expression is true, statements `a1`, `a2`, ... are executed, otherwise the statements `b1`, `b2`, ... are executed.

Let us use this to find the motion of the grain. Every time we throw the dice, the grain moves a distance $dx = \pm 1$. If the grain is at position x_i at step i, the grain will be at a position

$$x_{i+1} = x_i + dx \ . \tag{2.4}$$

at step $i+1$. We can use this rule and an `if`-statement to write the script to find the position at subsequent steps $i = 0, 1, 2, \ldots$:

```
from pylab import *
n = 1000
x = zeros(n,float)
for i in range(n):
    if (randint(6)+1<=3):
        dx = -1
    else:
        dx =  1
    x[i+1] = x[i] + dx
```

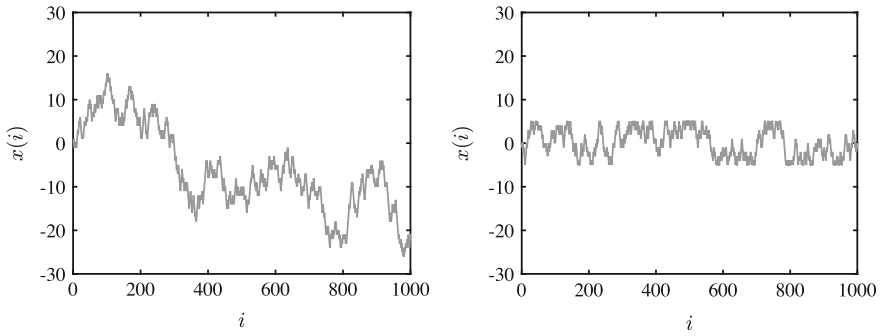

Fig. 2.2 Plot of the position $x(i)$ of a random walker (a bouncing grain of dust) as a function of the number of steps i done (*left*), and when the walker is constrained to the zone $-5 \leq x \leq +5$ (*right*)

The resulting motion is shown in Fig. 2.2.

We will use `if`-statements throughout the text, often to enforce particular conditions on the motion. For example, we could add a level of complexity to the motion of the grain by requiring that the grain moved inside a narrow channel of width 10: The grain cannot move outside a region spanning from −5 to +5:

```
#start1
from pylab import *
n = 1000
x = zeros(n,float)
for i in range(1,n):
    if (randint(6)+1<=3):
        dx = -1
    else:
        dx = +1
    x[i] = x[i-1] + dx
    if (x[i]> 5):
        x[i] =  5
    if (x[i]<-5):
        x[i] = -5
#end1
plot(x),
xlabel('i'), ylabel('x(i)')
show()
```

The resulting motion x_i as a function of i is shown in Fig. 2.2.

For the interested reader, we include a particularly compact formulation of the random walk, which you may have fun trying to understand.

```
>> x = cumsum(2*(randint(1,7,1000)<=3)-1)
>> plot(x), show()
```

You can research this expression by using the `help`-function in Python

```
>> help(cumsum)
```

2.7 Reading Real Data

When you work with physics you need to handle real data: NASA publishes data for the motion of most stellar objects; your mobile phone has an accelerometer; and a GPS that measures thousands of data-points in a few seconds. You do not want to type in these numbers by hand. Therefore you need to be able to read files containing numbers. For example, the motion of a sprinter running 100 m is given in the file 100m.d.[3] The file looks like this if you open it in a text editor (such as `emacs`):

```
0.0000000e+000 -2.1155775e-001
1.0000000e-002 -1.7485406e-001
2.0000000e-002 -1.3798607e-001
3.0000000e-002 -1.0095306e-001
4.0000000e-002 -6.3754256e-002
5.0000000e-002 -2.6388915e-002
...
```

A total of 972 lines of data. The first column gives the time, measured in seconds, and the second column gives the position of the runner, measured in meters. Fortunately, it is very simple to read such as file into Python. It is done by a single command:

```
>> run100m = loadtxt("run100m.d")
```

We split the data into two arrays `t` and `x` by:

```
>> t = run100m[:,0]
>> x = run100m[:,1]
```

and plot the data using

```
>> plot(t,x), show()
```

If you experience a problem where Python cannot find the file, getting an error message like:

```
>> loadtxt("run100m.d")
... IOError: [Errno 2] No such file or directory: "run100m.d"
```

It means that the file `run100m.d` is not in your current working directory.

2.7.1 Example: Plot of Function and Derivative

Problem: Plot the function

$$f(x) = e^{-x^2} , \tag{2.5}$$

and its derivative by using the formula:

$$f'(x) \simeq \frac{f(x+h) - f(x-h)}{2h} , \tag{2.6}$$

[3] http://folk.uio.no/malthe/mechbook/100m.d.

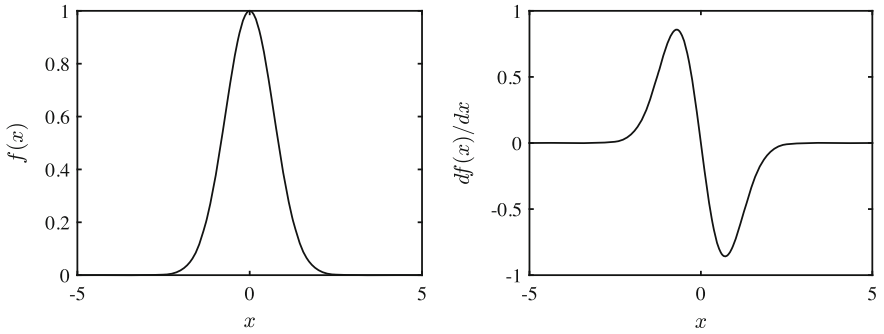

Fig. 2.3 Plot of $f(x)$ as a function of x and its derivative df/dx as a function of x calculated using a numerical method

as an approximation for the derivative on the interval $-5 \leq x \leq 5$. You may use the value $h = 0.001$ for h.

Solution: The function can be plotted directly by a vectorized approach:

```
>> from pylab import *
>> x = linspace(-5,5,1000)
>> f = exp(-x**2)
>> plot(x,f), show()
```

In order to use the numerical approximation for the derivative, we need to perform the approximation for each of the x-values in the `x`-array. We access them by a `for`-loop through the 1000 elements in the `x`-array:

```
>> h = 0.001
>> df = zeros(1000,float)
>> for i in range(1000):
...     df[i] = (exp(-(x[i]+h)**2)-exp(-(x[i]-h)**2))/(2*h)
>> plot(x,df), show()
```

The resulting plot is shown in Fig. 2.3.

Even simple problems such as these are useful to implement as scripts saved in a file, since this makes debugging—the process of finding and removing errors in the script—simpler. If you make a small mistake, you have to retype all the commands when you operate on the command line, but if you use a script, you simply make a small change in the script, rerun, and that is it.

Summary

Using Python as a calculator:

- Direct calculations are done on the command line

  ```
  >> 10.0*sin(pi/3)+4.0**3
  ```

- Defining and reusing variables

  ```
  >> a = 2.0, b = 4.5, c = a**2 + b**2
  ```

- Vectorized plotting of functions

```
>> x=linspace(0,10,0.01), y=exp(-x)*sin(x),plot(x,y),show()
```

- Vectorized operations are denoted by a leading `.` and are done *element-wise*.

Functions and scripts:

- A script is a sequence of executable commands stored in a separate `.py`-file
  - All variables are available on from the command line afterwards
  - The script is run by typing `F5` in the editor
  - Scripts allow rapid rerunning a program after changes in parameters
- A function can be part of a script of stored in a separate `.py` file. It has the syntax

```
def myfunction(a,b,c):
v = a*b*c
d = v**2
y = 2.0*d
return y
```

- The name of the function (`myfunction`) should be the name of the `.py`-file. Variables defined inside the function, such as `v` and `d` are not available outside the function

Plotting:

- You plot two arrays `t` and `x` versus each other by

```
plot(t,x,'-b'), xlabel('t (s)'), ylabel('x (m)')
```

- Plotting markers and symbols are listed in Table 2.3.
- Plotting several data-sets in the same plot:

```
plot(t1,x1,'-b',t2,x2,'-r')
```

Table 2.3 Line markers, colors and plotting symbols

Colors		Lines		Symbols		Symbols	
`b`	blue	`-`	solid	`.`	point	`v`	triangle (down)
`g`	green	`:`	dotted	`o`	circle	`^`	triangle (up)
`r`	red	`-.`	dashdot	`x`	x-mark	`<`	triangle (left)
`c`	cyan	`--`	dashed	`+`	plus	`>`	triangle (right)
`m`	magenta	(none)	no line	`*`	star	`p`	pentagram
`y`	yellow			`d`	diamond	`h`	hexagram
`k`	black						
`w`	white						

or

```
plot(t1,x1,'-b')
hold('on'), plot(t2,x2,'-r'), hold('off')
```

- Combining several plots into one figure:

```
subplot(2,1,1), plot(t1,x1,'-b')
subplot(2,1,2), plot(t2,x2,'-r')
```

- Saving a figure to a file: either by using the save button from the figure window, or you can save a figure as a pdf from the command line by

```
savefig('myfigure.pdf')
```

where `myfigure.pdf` is the name of the generated file.

Loops:

- `for`-loops run a counter sequentially through a list of values:

```
for i in range(100):
  x[i] = sin(i/100.0)
```

- `while`-loops run until a given expression is true

```
i = 0 while (i<100):
  i = i + 1
  x[i] = sin(i/100.0)
```

Expressions:

- `if`-statements are used to run a sequence of commands given a particular expression is true:

```
if (x>10.0):
  y =  10.0
else:
  y = -10.0
```

- Expressions return true (1) or false (0) and can be joined using logical operators such as *or* and *and* as shown in Table 2.4

Table 2.4 Expressions and operators used in Python

Expression	Name	Example	Operator	Name	Example
`==`	equal	`(x==0.0)`	`and`	logical AND	`(x==0.0)and (y>0.0)`
`!=`	not-equal	`(x!=0.0)`	`or`	logical OR	`(x!=0.0)or (y>0.0)`
`>=`	greater than or equal	`(x>=0.0)`			
`<=`	less than or equal	`(x<=0.0)`			
`>`	greater than	`(x>0.0)`			
`<`	less than	`(x<0.0)`			

Exercises

2.1 Seconds.
(**a**) Write a script that calculates the number of seconds, s, given the number of hours, h, according to the formula $s = 3600\,h$.
(**b**) Use the script to find the number of seconds in 1.5, 12 and 24 h.

2.2 Spherical mass.
(**a**) Write a script that calculates the mass of a sphere given its radius r and mass density ρ according to the formula $m = (4\pi/3)\ \rho r^3$.
(**b**) Use the script to find the mass of a sphere of steel of radius $r = 1$ mm, $r = 1$ m, and $r = 10$ m.

2.3 Angle.
(**a**) Write a function that for a point (x, y) returns the angle θ from the x-axis using the formula $\theta = \arctan(y/x)$.
(**b**) Find the angles θ for the points $(1, 1)$, $(-1, 1)$, $(-1, -1)$, $(1, -1)$.
(**c**) How would you change the function to return values of θ in the range $[0, 2\pi]$?

2.4 Unit vector.
(**a**) Write a function that returns the two-dimensional unit vector, (u_x, u_y), corresponding to an angle θ with the x-axis. You can use the formula $(u_x, u_y) = (\cos\theta, \sin\theta)$, where θ is given in radians.
(**b**) Find the unit vectors for $\theta = 0, \pi/6, \pi/3, \pi/2, 3\pi/2$.
(**c**) Rewrite the function to instead take the argument θ in degrees.

2.5 Plotting the normal distribution. The normal distribution, often called the Gaussian distribution, is given as:

$$P(x; \mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-(x-\mu)^2/(2\sigma^2)}\ , \tag{2.7}$$

where μ is the average and σ is the standard deviation.
(**a**) Make a function `normal(x,mu,sigma)` that returns the normal distribution value, $P(x, \mu, \sigma)$ as given by the formula.
(**b**) Use this function to plot the normal distribution for $-5 < x < 5$ for $\mu = 0$ and $\sigma = 1$.
(**c**) Plot the normal distribution for $-5 < x < 5$ for $\mu = 0$ and $\sigma = 2$ and for $\sigma = 0.5$ in the same plot.
(**d**) Plot the normal distribution for $-5 < x < 5$ for $\sigma = 1$ and $\mu = 0, 1, 2$ in three subplots above each other.

2.6 Plotting $1/x^n$. The function $f(x; n)$ is given as $f(x; n) = x^{-n}$.
(**a**) Make a function `fvalue(x,n)` which returns the value of $f(x; n)$.
(**b**) Use this function to plot $1/x$, $1/x^2$ and $1/x^3$ in the same plot for $-1 < x < 1$.

2.7 Plotting $\sin(x)/x^n$. The function $g(x; n)$ is given as:

$$g(x; n) = \frac{\sin(x)}{x^n} \ . \tag{2.8}$$

(**a**) Make a function `gvalue(x,n)` which returns the value of $g(x; n)$.
(**b**) Use this function to plot $\sin(x)/x$, $\sin(x)/x^2$ and $\sin(x)/x^3$ in the same plot for $-5 < x < 5$.
(**c**) Use the help function to find out how to place legends for each of the plots into the figure.

2.8 Logistic map. The iterative mapping $x(i+1) = r\,x(i)\ (1 - x(i))$ is called the logistic map.
(**a**) Make a function `logistic(x,r)` which returns the value of $x(i+1)$ given $x(i)$ and r as inputs.
(**b**) Write a script with a loop to calculate the first 100 steps of the logistic map starting from $x(1) = 0.5$. Store all the values in an array `x` with $n = 100$ elements and plot x as a function of the number of steps i:
(**c**) Explore the logistic map for $r = 1.0$, 2.0, 3.0 and 4.0.

2.9 Euler's method. In mechanics, we often use Euler's method to determine the motion of an object given how the acceleration depends on the velocity and position of an object. For example, we may know that the acceleration $a(x, v)$ is given as $a(x, v) = -kx - cv$. If we know the position x and the velocity v at a time $t = 0$: $x(0) = x_0 = 0$ and $v(0) = v_0 = 1$, we can use Euler's method to find the position and velocity after a small timestep Δt:

$$v_1 = v(t_0 + \Delta t) = v(t_0) + a(v(t_0), x(t_0))\Delta t \tag{2.9}$$
$$x_1 = x(t_0 + \Delta t) = x(t_0) + v(t_0)\Delta t \tag{2.10}$$
$$v_2 = v(t_1 + \Delta t) = v(t_1) + a(v(t_1), x(t_1))\Delta t \tag{2.11}$$
$$x_2 = x(t_1 + \Delta t) = x(t_1) + v(t_1)\Delta t \tag{2.12}$$

and so on. We can therefore use this scheme to find the position $x(t)$ and the velocity $v(t)$ as function of time at the discrete values $t_i = i\,\Delta t$ in time.
(**a**) Write a function `acceleration(v,x,k,C)` which returns the value of $a(x, v) = -kx - Cv$.
(**b**) Write a script that calculates the first 100 values of $x(t_i)$ and $v(t_i)$ when $k = 10$, $C = 5$, and $\Delta t = 0.01$. Plot $x(t)$, $v(t)$, and $a(t)$ as functions of time.
(**c**) What would you need to change to instead find $x(t)$ and $v(t)$ is the acceleration was given as $a(v, x) = k\sin(x) - Cv$?

2.10 Throwing two dice. You throw a pair of six-sided dice and sum the number from each of the dice: $Z = X_1 + X_2$, where Z is the sum of the results from dice 1, X_1, and dice 2, X_2. If we perform this experiment many times (N), we can find the

average and standard deviation from standard estimators from statistics. The average, $\langle Z \rangle$, of Z is estimated from:

$$\langle Z \rangle = \frac{1}{N} \sum_{j=1}^{N} Z_j \ , \tag{2.13}$$

and the standard deviation, ΔZ, is estimated from:

$$\Delta Z = \frac{1}{N-1} \left(\sum_{j=1}^{N} (Z_j - \langle Z \rangle \right)^2 \ . \tag{2.14}$$

(**a**) Write a function that returns an array of N values for Z.
(**b**) Write a function that returns an estimate of the average of an array `z` using the formula provided.
(**c**) Write a function that returns an estimate of the standard deviation of an array `z` using the formula provided.
(**d**) Find the average and standard deviation for $N = 100$ throws of two dice.

2.11 Reading data. The file trajectory.dat[4] contains a list of numbers:

```
t0 x0 y0
t1 x1 y1
.. .. ..
tn xn yn
```

corresponding to the time `t(i)` measured in seconds, and the x and y positions `x(i)` and `y(i)` measured in meters for the trajectory of a projectile.
(**a**) Read the data file into the arrays 't, `x`, and `y`.
(**b**) Plot the x and y positions as function of time in two plots above each other.
(**c**) Plot the (x, y) position of the object in a plot with x and y on the two axes.

2.12 Numerical integration of a data-set. The file velocityy.dat[5] contains a list of numbers:

```
t0 v0
t1 v1
.. .. ..
tn vn
```

corresponding to the time `t(i)` measured in seconds, and the velocity `y(i)` measured in meters per second for the trajectory of a projectile.
(**a**) Read the data file into the arrays `t`, and `v`.
(**b**) Plot $v(t)$ as function of time.

[4] http://folk.uio.no/malthe/mechbook/trajectory.dat.
[5] http://folk.uio.no/malthe/mechbook/velocityy.dat.

For a data-set `t(i)`, `v(i)`, you can estimate the function corresponding to the integral of $v(t)$ with respect to t at the times t_i using the iterative scheme:

$$y(t_1) \simeq y(t_0) + v(t_0)\,(t_1 - t_0) \tag{2.15}$$
$$y(t_2) \simeq y(t_1) + v(t_1)\,(t_2 - t_1) \tag{2.16}$$
$$\ldots \simeq \ldots \tag{2.17}$$
$$y(t_n) \simeq y(t_{n-1}) + v(t_{n-1})\,(t_n - t_{n-1}) \tag{2.18}$$

where $v(t_i) =$'v(i)' and $t_i =$'t(i)' You can assume that the motion starts at $y(t_0) =$ 0.0m at $t = t_0$.
(**c**) Write a script to calculate the time integral $y(t_i)$ of the dataset using this formula. Implement using a `for`-loop.
(**d**) Plot the position $y(t)$ and the derivative $v(t)$ as functions of time in two plots above each other.