User input and error handling

4

Consider a program for evaluating the formula $x = A\sin(wt)$:

```
from math import sin
A = 0.1
w = 1
t = 0.6
x = A*sin(w*t)
print x
```

In this program, `A`, `w`, and `t` are input data in the sense that these parameters must be known before the program can perform the calculation of `x`. The results produced by the program, here `x`, constitute the output data.

Input data can be hardcoded in the program as we do above. That is, we explicitly set variables to specific values: `A=0.1`, `w=1`, `t=0.6`. This programming style may be suitable for small programs. In general, however, it is considered good practice to let a user of the program provide input data when the program is running. There is then no need to modify the program itself when a new set of input data is to be explored. This is an important feature, because a golden rule of programming is that modification of the source code always represents a danger of introducing new errors by accident.

This chapter starts with describing three different ways of reading data into a program:

1. let the user answer questions in a dialog in the terminal window (Section 4.1),
2. let the user provide input on the command line (Section 4.2),
3. let the user provide input data in a file (Section 4.5),
4. let the user write input data in a graphical interface (Section 4.8).

Even if your program works perfectly, wrong input data from the user may cause the program to produce wrong answers or even crash. Checking

H.P. Langtangen, *A Primer on Scientific Programming with Python*,
Texts in Computational Science and Engineering 6,
DOI 10.1007/978-3-642-54959-5_4, © Springer-Verlag Berlin Heidelberg 2014

that the input data are correct is important, and Section 4.7 tells you how to do this with so-called exceptions.

The Python programming environment is organized as a big collection of modules. Organizing your own Python software in terms of modules is therefore a natural and wise thing to do. Section 4.9 tells you how easy it is to make your own modules.

All the program examples from the present chapter are available in files in the `src/input`[1] folder.

4.1 Asking questions and reading answers

One of the simplest ways of getting data into a program is to ask the user a question, let the user type in an answer, and then read the text in that answer into a variable in the program. These tasks are done by calling a function with name `raw_input` in Python 2 - the name is just `input` in Python 3.

4.1.1 Reading keyboard input

A simple problem involving the temperature conversion from Celsius to Fahrenheit constitutes our main example: $F = \frac{9}{5}C + 32$. The associated program with setting `C` explicitly in the program reads

```
C = 22
F = 9./5*C + 32
print F
```

We may ask the user a question `C=?` and wait for the user to enter a number. The program can then read this number and store it in a variable `C`. These actions are performed by the statement

```
C = raw_input('C=? ')
```

The `raw_input` function always returns the user input as a string object. That is, the variable `C` above refers to a string object. If we want to compute with this `C`, we must convert the string to a floating-point number: `C = float(C)`. A complete program for reading `C` and computing the corresponding degrees in Fahrenheit now becomes

```
C = raw_input('C=? ')
C = float(C)
F = (9./5)*C + 32
print F
```

[1] `http://tinyurl.com/pwyasaa/input`

In general, the `raw_input` function takes a string as argument, displays this string in the terminal window, waits until the user presses the Return key, and then returns a string object containing the sequence of characters that the user typed in.

The program above is stored in a file called `c2f_qa.py` (the `qa` part of the name reflects *question and answer*). We can run this program in several ways. The convention in this book is to indicate the execution by writing the program name only, but for a real execution you need to do more: write `run` before the program name in an interactive IPython session, or write `python` before the program name in a terminal session. Here is the execution of our sample program and the resulting dialog with the user:

Terminal

```
c2f_qa.py
C=? 21
69.8
```

In this particular example, the `raw_input` function reads the characters `21` from the keyboard and returns the string `'21'`, which we refer to by the variable `C`. Then we create a new `float` object by `float(C)` and let the name `C` refer to this `float` object, with value 21.

You should now try out Exercises 4.1, 4.6, and 4.9 to make sure you understand how `raw_input` behaves.

4.2 Reading from the command line

Programs running on Unix computers usually avoid asking the user questions. Instead, input data are very often fetched from the *command line*. This section explains how we can access information on the command line in Python programs.

4.2.1 Providing input on the command line

We look at the Celsius-Fahrenheit conversion program again. The idea now is to provide the Celsius input temperature as a *command-line argument* right after the program name. This means that we write the program name, here `c2f_cml.py` (`cml` for *command line*), followed the Celsius temperature:

Terminal

```
c2f_cml.py 21
69.8
```

Inside the program we can fetch the text 21 as `sys.argv[1]`. The `sys` module has a list `argv` containing all the command-line arguments to the program, i.e., all the "words" appearing after the program name when we run the program. In the present case there is only one argument and it is stored in `sys.argv[1]`. The first element in the `sys.argv` list, `sys.argv[0]`, is always the name of the program.

A command-line argument is treated as a text, so `sys.argv[1]` refers to a string object, in this case `'21'`. Since we interpret the command-line argument as a number and want to compute with it, it is necessary to explicitly convert the string to a `float` object. In the program we therefore write

```
import sys
C = float(sys.argv[1])
F = 9.0*C/5 + 32
print F
```

As another example, consider the program

```
v0 = 5
g = 9.81
t = 0.6
y = v0*t - 0.5*g*t**2
print y
```

for computing the formula $y(t) = v_0 t - \frac{1}{2}gt^2$. Instead of hardcoding the values of `v0` and `t` in the program we can read the two values from the command line:

Terminal

```
ball2_cml.py 0.6 5
1.2342
```

The two command-line arguments are now available as `sys.argv[1]` and `sys.argv[2]`. The complete `ball2_cml.py` program thus takes the form

```
import sys
t  = float(sys.argv[1])
v0 = float(sys.argv[2])
g = 9.81
y = v0*t - 0.5*g*t**2
print y
```

4.2.2 A variable number of command-line arguments

Let us make a program `addall.py` that adds all its command-line arguments. That is, we may run something like

Terminal

```
addall.py 1 3 5 -9.9
The sum of 1 3 5 -9.9 is -0.9
```

The command-line arguments are stored in the sublist `sys.argv[1:]`. Each element is a string so we must perform a conversion to `float` before performing the addition. There are many ways to write this program. Let us start with version 1, `addall_v1.py`:

```
import sys
s = 0
for arg in sys.argv[1:]:
    number = float(arg)
    s += number
print 'The sum of ',
for arg in sys.argv[1:]:
    print arg,
print 'is ', s
```

The output is on one line, but built of several `print` statements with a comma at the end to prevent the usual newline character that `print` otherwise adds to the text. The command-line arguments must be converted to numbers in the first `for` loop because we need to compute with them, but in the second loop we only need to print them and then the string representation is appropriate.

The program above can be written more compactly if desired:

```
import sys
s = sum([float(x) for x in sys.argv[1:]])
print 'The sum of %s is %s' % (' '.join(sys.argv[1:]), s)
```

Here, we convert the list `sys.argv[1:]` to a list of `float` objects and then pass this list to Python's `sum` function for adding the numbers. The construction `S.join(L)` places all the elements in the list `L` after each other with the string `S` in between. The result here is a string with all the elements in `sys.argv[1:]` and a space in between, which is the text that originally appeared on the command line.

4.2.3 More on command-line arguments

Unix commands make heavy use of command-line arguments. For example, when you write `ls -s -t` to list the files in the current folder, you run the program `ls` with two command-line arguments: `-s` and `-t`. The former specifies that `ls` is to print the file name together with the size of the file, and the latter sorts the list of files according to their dates of last modification. Similarly, `cp -r my new` for copying a folder tree `my` to a new folder tree `new` invokes the `cp` program with three command line arguments: `-r` (for recursive copying of files), `my`, and `new`. Most programming languages have support for extracting the command-line arguments given to a program.

An important rule is that *command-line arguments are separated by blanks*. What if we want to provide a text containing blanks as command-line argument? The text containing blanks must then appear inside single

or double quotes. Let us demonstrate this with a program that simply prints the command-line arguments:

```
import sys, pprint
pprint.pprint(sys.argv[1:])
```

Say this program is named `print_cml.py`. The execution

Terminal

```
print_cml.py 21 a string with blanks 31.3
['21', 'a', 'string', 'with', 'blanks', '31.3']
```

demonstrates that each word on the command line becomes an element in `sys.argv`. Enclosing strings in quotes, as in

Terminal

```
print_cml.py 21 "a string with blanks" 31.3
['21', 'a string with blanks', '31.3']
```

shows that the text inside the quotes becomes a single command line argument.

4.3 Turning user text into live objects

It is possible to provide text with valid Python code as input to a program and then turn the text into live objects as if the text were written directly into the program beforehand. This is a very powerful tool for letting users specify function formulas, for instance, as input to a program. The program code itself has no knowledge about the kind of function the user wants to work with, yet at run time the user's desired formula enters the computations.

4.3.1 The magic eval function

The `eval` functions takes a string as argument and evaluates this string as a Python *expression*. The result of an expression is an object. Consider

```
>>> r = eval('1+2')
>>> r
3
>>> type(r)
<type 'int'>
```

The result of `r = eval('1+2')` is the same as if we had written `r = 1+2` directly:

```
>>> r = 1+2
>>> r
3
>>> type(r)
<type 'int'>
```

In general, any valid Python expression stored as text in a string `s` can be turned into live Python code by `eval(s)`.

Here is an example where the string to be evaluated is `'2.5'`, which causes Python to see `r = 2.5` and make a `float` object:

```
>>> r = eval('2.5')
>>> r
2.5
>>> type(r)
<type 'float'>
```

Let us proceed with some more examples. We can put the initialization of a list inside quotes and use `eval` to make a `list` object:

```
>>> r = eval('[1, 6, 7.5]')
>>> r
[1, 6, 7.5]
>>> type(r)
<type 'list'>
```

Again, the assignment to `r` is equivalent to writing

```
>>> r = [1, 6, 7.5]
```

We can also make a `tuple` object by using tuple syntax (standard parentheses instead of brackets):

```
>>> r = eval('(-1, 1)')
>>> r
(-1, 1)
>>> type(r)
<type 'tuple'>
```

Another example reads

```
>>> from math import sqrt
>>> r = eval('sqrt(2)')
>>> r
1.4142135623730951
>>> type(r)
<type 'float'>
```

At the time we run `eval('sqrt(2)')`, this is the same as if we had written

```
>>> r = sqrt(2)
```

directly, and this is valid syntax only if the `sqrt` function is defined. Therefore, the import of `sqrt` prior to running `eval` is important in this example.

Applying eval to strings. If we put a string, enclosed in quotes, inside the expression string, the result is a string object:

```
>>>
>>> r = eval('"math programming"')
>>> r
'math programming'
>>> type(r)
<type 'str'>
```

Note that we must use two types of quotes: first double quotes to mark `math programming` as a string object and then another set of quotes, here single quotes (but we could also have used triple single quotes), to embed the text `"math programming"` inside a string. It does not matter if we have single or double quotes as inner or outer quotes, i.e., `'"..."'` is the same as `"'...'"`, because `'` and `"` are interchangeable as long as a pair of either type is used consistently.

Writing just

```
>>> r = eval('math programming')
```

is the same as writing

```
>>> r = math programming
```

which is an invalid expression. Python will in this case think that `math` and `programming` are two (undefined) variables, and setting two variables next to each other with a space in between is invalid Python syntax. However,

```
>>> r = 'math programming'
```

is valid syntax, as this is how we initialize a string `r` in Python. To repeat, if we put the valid syntax `'math programming'` inside a string,

```
s = "'math programming'"
```

`eval(s)` will evaluate the text inside the double quotes as `'math programming'`, which yields a string.

Applying eval to user input. So, why is the `eval` function so useful? When we get input via `raw_input` or `sys.argv`, it is always in the form of a string object, which often must be converted to another type of object, usually an `int` or `float`. Sometimes we want to avoid specifying one particular type. The `eval` function can then be of help: we feed the string object from the input to the `eval` function and let the it interpret the string and convert it to the right object.

An example may clarify the point. Consider a small program where we read in two values and add them. The values could be strings, floats, integers, lists, and so forth, as long as we can apply a `+` operator to the values. Since we do not know if the user supplies a string, float, integer,

or something else, we just convert the input by `eval`, which means that the user's syntax will determine the type. The program goes as follows (add_input.py):

```
i1 = eval(raw_input('Give input: '))
i2 = eval(raw_input('Give input: '))
r = i1 + i2
print '%s + %s becomes %s\nwith value %s' % \
      (type(i1), type(i2), type(r), r)
```

Observe that we write out the two supplied values, together with the types of the values (obtained by `eval`), and the sum. Let us run the program with an integer and a real number as input:

Terminal

```
add_input.py
Give input: 4
Give input: 3.1
<type 'int'> + <type 'float'> becomes <type 'float'>
with value 7.1
```

The string `'4'`, returned by the first call to `raw_input`, is interpreted as an `int` by `eval`, while `'3.1'` gives rise to a `float` object.

Supplying two lists also works fine:

Terminal

```
add_input.py
Give input: [-1, 3.2]
Give input: [9,-2,0,0]
<type 'list'> + <type 'list'> becomes <type 'list'>
with value [-1, 3.2000000000000002, 9, -2, 0, 0]
```

If we want to use the program to add two strings, the strings must be enclosed in quotes for `eval` to recognize the texts as string objects (without the quotes, `eval` aborts with an error):

Terminal

```
add_input.py
Give input: 'one string'
Give input: " and another string"
<type 'str'> + <type 'str'> becomes <type 'str'>
with value one string and another string
```

Not all objects are meaningful to add:

Terminal

```
add_input.py
Give input: 3.2
Give input: [-1,10]
Traceback (most recent call last):
  File "add_input.py", line 3, in <module>
    r = i1 + i2
TypeError: unsupported operand type(s) for +: 'float' and 'list'
```

A similar program adding two arbitrary command-line arguments reads ((add_input.py):

```
import sys
i1 = eval(sys.argv[1])
i2 = eval(sys.argv[2])
r = i1 + i2
print '%s + %s becomes %s\nwith value %s' % \
      (type(i1), type(i2), type(r), r)
```

Another important example on the usefulness of `eval` is to turn formulas, given as input, into mathematics in the program. Consider the program

```
from math import *   # make all math functions available
import sys
formula = sys.argv[1]
x = eval(sys.argv[2])
result = eval(formula)
print '%s for x=%g yields %g' % (formula, x, result)
```

Two command-line arguments are expected: a formula and a number. Say the formula given is `2*sin(x)+1` and the number is 3.14. This information is read from the command line as strings. Doing `x = eval(sys.argv[2])` means `x = eval('3.14')`, which is equivalent to `x = 3.14`, and `x` refers to a `float` object. The `eval(formula)` expression means `eval('2*sin(x)+1')`, and the corresponding statement `result = eval(formula` is therefore effectively `result = 2*sin(x)+1`, which requires `sin` and `x` to be defined objects. The result is a `float` (approximately 1.003). Providing `cos(x)` as the first command-line argument creates a need to have `cos` defined, so that is why we import all functions from the `math` module. Let us try to run the program:

Terminal

```
eval_formula.py "2*sin(x)+1" 3.14
2*sin(x)+1 for x=3.14 yields 1.00319
```

The very nice thing with using `eval` in `x = eval(sys.argv[2])` is that we can provide mathematical expressions like `pi/2` or even `tanh(2*pi)`, as the latter just effectively results in the statement `x = tanh(2*pi)`, and this works fine as long has we have imported `tanh` and `pi`.

4.3.2 The magic exec function

Having presented `eval` for turning strings into Python code, we take the opportunity to also describe the related `exec` function to execute a string containing arbitrary Python code, not only an expression.

Suppose the user can write a formula as input to the program, available to us in the form of a string object. We would then like to turn this formula

into a callable Python function. For example, writing `sin(x)*cos(3*x) + x**2` as the formula, we would make the function

```
def f(x):
    return sin(x)*cos(3*x) + x**2
```

This is easy with `exec`: just construct the right Python syntax for defining `f(x)` in a string and apply `exec` to the string,

```
formula = sys.argv[1]
code = """
def f(x):
    return %s
""" % formula
exec(code)
```

As an example, think of `"sin(x)*cos(3*x) + x**2"` as the first command-line argument. Then `formula` will hold this text, which is inserted into the `code` string such that it becomes

```
"""
def f(x):
    return sin(x)*cos(3*x) + x**2
"""
```

Thereafter, `exec(code)` executes the code as if we had written the contents of the `code` string directly into the program by hand. With this technique, we can turn any user-given formula into a Python function!

Let us now use this technique in a useful application. Suppose we have made a function for computing the integral $\int_a^b f(x)dx$ by the Midpoint rule with n intervals:

```
def midpoint_integration(f, a, b, n=100):
    h = (b - a)/float(n)
    I = 0
    for i in range(n):
        I += f(a + i*h + 0.5*h)
    return h*I
```

We now want to read a, b, and n from the command line as well as the formula that makes up the $f(x)$ function:

```
from math import *
import sys
f_formula = sys.argv[1]
a = eval(sys.argv[2])
b = eval(sys.argv[3])
if len(sys.argv) >= 5:
    n = int(sys.argv[4])
else:
    n = 200
```

Note that we import everything from `math` and use `eval` when reading the input for `a` and `b` as this will allow the user to provide values like `2*cos(pi/3)`.

The next step is to convert the `f_formula` for $f(x)$ into a Python function `g(x)`:

```
code = """
def g(x):
    return %s
""" % f_formula
exec(code)
```

Now we have an ordinary Python function `g(x)` that we can ask the integration function to integrate:

```
I = midpoint_integration(g, a, b, n)
print 'Integral of %s on [%g, %g] with n=%d: %g' % \
      (f_formula, a, b, n, I)
```

The complete code is found in integrate.py. A sample run for $\int_0^{\pi/2} \sin(x)dx$ goes like

Terminal

```
integrate.py "sin(x)" 0 pi/2
integral of sin(x) on [0, 1.5708] with n=200: 0.583009
```

(The quotes in `"sin(x)"` are needed because of the parenthesis will otherwise be interpreted by the shell.)

4.3.3 Turning string expressions into functions

The examples in the previous section indicate that it can be handy to ask the user for a formula and turn that formula into a Python function. Since this operation is so useful, we have made a special tool that hides the technicalities. The tool is named `StringFunction` and works as follows:

```
>>> from scitools.StringFunction import StringFunction
>>> formula = 'exp(x)*sin(x)'
>>> f = StringFunction(formula)   # turn formula into function f(x)
```

The `f` object now behaves as an ordinary Python function of `x`:

```
>>> f(0)
0.0
>>> f(pi)
2.8338239229952166e-15
>>> f(log(1))
0.0
```

Expressions involving other independent variables than `x` are also possible. Here is an example with the function $g(t) = Ae^{-at}\sin(\omega x)$:

```
g = StringFunction('A*exp(-a*t)*sin(omega*x)',
                   independent_variable='t',
                   A=1, a=0.1, omega=pi, x=0.5)
```

The first argument is the function formula, as before, but now we need to specify the name of the independent variable (`'x'` is default). The other parameters in the function (A, a, ω, and x) must be specified with

values, and we use keyword arguments, consistent with the names in the function formula, for this purpose. Any of the parameters `A`, `a`, `omega`, and `x` can be changed later by calls like

```
g.set_parameters(omega=0.1)
g.set_parameters(omega=0.1, A=5, x=0)
```

Calling `g(t)` works as if `g` were a plain Python function of `t`, which also stores all the parameters `A`, `a`, `omega`, and `x`, and their values. You can use `pydoc` to bring up more documentation on the possibilities with `StringFunction`. Just run

```
pydoc scitools.StringFunction.StringFunction
```

A final important point is that `StringFunction` objects are as computationally efficient as hand-written Python functions. (This property is quite remarkable, as a string formula will in most other programming languages be much slower to evaluate than if the formula were hardcoded inside a plain function.)

4.4 Option-value pairs on the command line

The examples on using command-line arguments so far require the user of the program to type all arguments in their right sequence, just as when calling a function with positional arguments in the right order. It would be very convenient to assign command-line arguments in the same way as we use keyword arguments. That is, arguments are associated with a name, their sequence can be arbitrary, and only the arguments where the default value is not appropriate need to be given. Such type of command-line arguments may have `-option value` pairs, where `option` is some name of the argument.

As usual, we shall use an example to illustrate how to work with `-option value` pairs. Consider the physics formula for the location $s(t)$ of an object at time t, given that the object started at $s = s_0$ at $t = 0$ with a velocity v_0, and thereafter was subject to a constant acceleration a:

$$s(t) = s_0 + v_0 t + \frac{1}{2} a t^2 \,. \tag{4.1}$$

This formula requires four input variables: s_0, v_0, a, and t. We can make a program `location.py` that takes four options, `-s0`, `-v0`, `-a`, and `-t`, on the command line. The program is typically run like this:

Terminal

```
location.py --t 3 --s0 1 --v0 1 --a 0.5
```

The sequence of `-option value` pairs is arbitrary. All options have a default value such that one does not have to specify all options on the command line.

All input variables should have sensible default values such that we can leave out the options for which the default value is suitable. For example, if $s_0 = 0$, $v_0 = 0$, $a = 1$, and $t = 1$ by default, and we only want to change t, we can run

Terminal

```
location.py --t 3
```

4.4.1 Basic usage of the argparse module

Python has a flexible and powerful module `argparse` for reading (parsing) `-option value` pairs on the command line. Using `argparse` consists of three steps. First, a parser object must be created:

```
import argparse
parser = argparse.ArgumentParser()
```

Second, we need to define the various command-line options,

```
parser.add_argument('--v0', '--initial_velocity', type=float,
                    default=0.0, help='initial velocity',
                    metavar='v')
parser.add_argument('--s0', '--initial_position', type=float,
                    default=0.0, help='initial position',
                    metavar='s')
parser.add_argument('--a', '--acceleration', type=float,
                    default=1.0, help='acceleration', metavar='a')
parser.add_argument('--t', '--time', type=float,
                    default=1.0, help='time', metavar='t')
```

The first arguments to `parser.add_argument` is the set of options that we want to associate with an input parameter. Optional arguments are the type, a default value, a help string, and a name for the value of the argument (`metavar`) in a usage string. The `argparse` module will automatically allow an option `-h` or `-help` that prints a usage string for all the registered options. By default, the type is `str`, the default value is `None`, the help string is empty, and `metavar` is the option in upper case without initial dashes.

Third, we must read the command line arguments and interpret them:

```
args = parser.parse_args()
```

Through the `args` object we can extract the values of the various registered parameters: `args.v0`, `args.s0`, `args.a`, and `args.t`. The name of the parameter is determined by the first option to `parser.add_argument`, so writing

```
parser.add_argument('--initial_velocity', '--v0', type=float,
                    default=0.0, help='initial velocity')
```

will make the initial velocity value appear as `args.initial_velocity`. We can add the `dest` keyword to explicitly specify the name where the value is stored:

```
parser.add_argument('--initial_velocity', '--v0', dest='V0',
                    type=float, default=0, help='initial velocity')
```

Now, `args.V0` will retrieve the value of the initial velocity. In case we do not provide any default value, the value will be `None`.

Our example is completed either by evaluating `s` as

```
s = args.s0 + args.v0*t + 0.5*args.a*args.t**2
```

or by introducing new variables so that the formula aligns better with the mathematical notation:

```
s0 = args.s0; v0 = args.v0; a = args.a; t = args.t
s = s0 + v0*t + 0.5*a*t**2
```

A complete program for the example above is found in the file `location.py`. Try to run it with the `-h` option to see an automatically generated explanation of legal command-line options.

4.4.2 Mathematical expressions as values

Values on the command line involving mathematical symbols and functions, say `-v0 'pi/2'`, pose a problem with the code example above. The `argparse` module will in that case try to do `float('pi/2')` which does not work well since `pi` is an undefined name. Changing `type=float` to `type=eval` is required to interpret the expression `pi/2`, but even `eval('pi/2')` fails since `pi` is not defined inside the `argparse` module. There are various remedies for this problem.

One can write a tailored function for converting a string value given on the command line to the desired object. For example,

```
def evalcmlarg(text):
    return eval(text)

parser.add_argument('--s0', '--initial_position', type=evalcmlarg,
                    default=0.0, help='initial position')
```

The file `location_v2.py` demonstrates such explicit type conversion through a user-provided conversion function. Note that `eval` is now taken in the programmer's namespace where (hopefully) `pi` or other symbols are imported.

More sophisticated conversions are possible. Say s_0 is specified in terms of a function of some parameter p, like $s_0 = (1 - p^2)$. We could then use

a string for `-s0` and the `StringFunction` tool from Section 4.3.3 to turn the string into a function:

```
def toStringFunction4s0(text):
    from scitools.std import StringFunction
    return StringFunction(text, independent_variable='p')

parser.add_argument('--s0', '--initial_position',
                    type=toStringFunction4s0,
                    default='0.0', help='initial position')
```

Giving a command-line argument `-s0 'exp(-1.5) + 10(1-p**2)` results in `args.s0` being a `StringFunction` object, which we must evaluate for a `p` value:

```
s0 = args.s0
p = 0.05
...
s = s0(p) + v0*t + 0.5*a*t**2
```

The file location_v3.py contains the complete code for this example.

Another alternative is to perform the correct conversion of values in our own code *after* the `parser` object has read the values. To this end, we treat argument types as strings in the `parser.add_argument` calls, meaning that we replace `type=float` by set `type=str` (which is also the default choice of `type`). Recall that this approach requires specification of default values as strings too, say `'0'`:

```
parser.add_argument('--s0', '--initial_position', type=str,
                    default='0', help='initial position')
...
from math import *
args.v0 = eval(args.v0)
# or
v0 = eval(args.v0)

s0 = StringFunction(args.s0, independent_variable='p')
p = 0.5
...
s = s0(p) + v0*t + 0.5*a*t**2
```

Such code is found in the file location_v4.py. You can try out that program with the command-line arguments `-s0 'pi/2 + sqrt(p)' -v0 pi/4'`.

The final alternative is to write an `Action` class to handle the conversion from string to the right type. This is the preferred way to perform conversions and well described in the `argparse` documentation. We shall exemplify it here, but the technicalities involved require understanding of classes (Chapter 7) and inheritance (Chapter 9). For the conversion from string to any object via `eval` we write

```
import argparse
from math import *

class ActionEval(argparse.Action):
```

```
    def __call__(self, parser, namespace, values,
                 option_string=None):
        setattr(namespace, self.dest, eval(values))
```

The command-line arguments supposed to be run through `eval` must then have an `action` parameter:

```
parser.add_argument('--v0', '--initial_velocity',
                    default=0.0, help='initial velocity',
                    action=ActionEval)
```

From string to function via `StringFunction` for the `-s0` argument we write

```
from scitools.std import StringFunction

class ActionStringFunction4s0(argparse.Action):
    def __call__(self, parser, namespace, values,
                 option_string=None):
        setattr(namespace, self.dest,
                StringFunction(values, independent_variable='p'))
```

A complete code appears in the file location_v5.py.

4.5 Reading data from file

Getting input data into a program from the command line, or from questions and answers in the terminal window, works for small amounts of data. Otherwise, input data must be available in files. Anyone with some computer experience is used to save and load data files in programs. The task now is to understand how Python programs can read and write files. The basic recipes are quite simple and illustrated through examples.

Suppose we have recorded some measurement data in the file src/input/data.txt[2]. The goal of our first example of reading files is to read the measurement values in `data.txt`, find the average value, and print it out in the terminal window.

Before trying to let a program read a file, we must know the *file format*, i.e., what the contents of the file looks like, because the structure of the text in the file greatly influences the set of statements needed to read the file. We therefore start with viewing the contents of the file `data.txt`. To this end, load the file into a text editor or viewer (one can use `emacs`, `vim`, `more`, or `less` on Unix and Mac, while on Windows, `WordPad` is appropriate, or the `type` command in a DOS or PowerShell window, and even Word processors such as LibreOffice or Microsoft Word can also be used on Windows). What we see is a column with numbers:

```
21.8
18.1
19
```

[2] `http://tinyurl.com/pwyasaa/input/data.txt`

```
23
26
17.8
```

Our task is to read this column of numbers into a list in the program and compute the average of the list items.

4.5.1 Reading a file line by line

To read a file, we first need to *open* the file. This action creates a file object, here stored in the variable `infile`:

```
infile = open('data.txt', 'r')
```

The second argument to the `open` function, the string `'r'`, tells that we want to open the file for reading. We shall later see that a file can be opened for writing instead, by providing `'w'` as the second argument. After the file is read, one should close the file object with `infile.close()`.

The basic technique for reading the file line by line applies a `for` loop like this:

```
for line in infile:
    # do something with line
```

The `line` variable is a string holding the current line in the file. The `for` loop over lines in a file has the same syntax as when we go through a list. Just think of the file object `infile` as a collection of elements, here lines in a file, and the `for` loop visits these elements in sequence such that the `line` variable refers to one line at a time. If something seemingly goes wrong in such a loop over lines in a file, it is useful to do a `print line` inside the loop.

Instead of reading one line at a time, we can load all lines into a list of strings (lines) by

```
lines = infile.readlines()
```

This statement is equivalent to

```
lines = []
for line in infile:
    lines.append(line)
```

or the list comprehension:

```
lines = [line for line in infile]
```

In the present example, we load the file into the list `lines`. The next task is to compute the average of the numbers in the file. Trying a straightforward sum of all numbers on all lines,

```
mean = 0
for number in lines:
    mean = mean + number
mean = mean/len(lines)
```

gives an error message:

```
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

The reason is that `lines` holds each line (`number`) as a string, not a `float` or `int` that we can add to other numbers. A fix is to convert each line to a `float`:

```
mean = 0
for line in lines:
    number = float(line)
    mean = mean + number
mean = mean/len(lines)
```

This code snippet works fine. The complete code can be found in the file mean1.py.

Summing up a list of numbers is often done in numerical programs, so Python has a special function `sum` for performing this task. However, `sum` must in the present case operate on a list of floats, not strings. We can use a list comprehension to turn all elements in `lines` into corresponding `float` objects:

```
mean = sum([float(line) for line in lines])/len(lines)
```

An alternative implementation is to load the lines into a list of `float` objects directly. Using this strategy, the complete program (found in file mean2.py) takes the form

```
infile = open('data.txt', 'r')
numbers = [float(line) for line in infile.readlines()]
infile.close()
mean = sum(numbers)/len(numbers)
print mean
```

4.5.2 Alternative ways of reading a file

A newcomer to programming might find it confusing to see that one problem is solved by many alternative sets of statements, but this is the very nature of programming. A clever programmer will judge several alternative solutions to a programming task and choose one that is either particularly compact, easy to understand, and/or easy to extend later. We therefore present more examples on how to read the `data.txt` file and compute with the data.

The modern with statement. Modern Python code applies the `with` statement to deal with files:

```
with open('data.txt', 'r') as infile:
    for line in infile:
        # process line
```

This snippet is equivalent to

```
infile = open('data.txt', 'r')
for line in infile:
    # process line
infile.close()
```

Note that there is no need to close the file when using the `with` statement. The advantage of the `with` construction is shorter code and better handling of errors if something goes wrong with opening or working with the file. A downside is that the syntax differs from the very classical open-close pattern that one finds in most other programming languages. Remembering to close a file is key in programming, and to train that task, we mostly apply the open-close construction in this book.

The old while construction. The call `infile.readline()` returns a string containing the text at the current line. A new `infile.readline()` will read the next line. When `infile.readline()` returns an empty string, the end of the file is reached and we must stop further reading. The following `while` loop reads the file line by line using `infile.readline()`:

```
while True:
    line = infile.readline()
    if not line:
        break
    # process line
```

This is perhaps a somewhat strange loop, but it is a well-established way of reading a file in Python, especially in older code. The shown `while` loop runs forever since the condition is always `True`. However, inside the loop we test if `line` is `False`, and it is `False` when we reach the end of the file, because `line` then becomes an empty string, which in Python evaluates to `False`. When `line` is `False`, the `break` statement breaks the loop and makes the program flow jump to the first statement after the `while` block.

Computing the average of the numbers in the `data.txt` file can now be done in yet another way:

```
infile = open('data.txt', 'r')
mean = 0
n = 0
while True:
    line = infile.readline()
    if not line:
        break
    mean += float(line)
    n += 1
mean = mean/float(n)
```

Reading a file into a string. The call `infile.read()` reads the whole file and returns the text as a string object. The following interactive session illustrates the use and result of `infile.read()`:

```
>>> infile = open('data.txt', 'r')
>>> filestr = infile.read()
>>> filestr
'21.8\n18.1\n19\n23\n26\n17.8\n'
>>> print filestr
21.8
18.1
19
23
26
17.8
```

Note the difference between just writing `filestr` and writing `print filestr`. The former dumps the string with newlines as *backslash n* characters, while the latter is a *pretty print* where the string is written out without quotes and with the newline characters as visible line shifts.

Having the numbers inside a string instead of inside a file does not look like a major step forward. However, string objects have many useful functions for extracting information. A very useful feature is *split*: `filestr.split()` will split the string into words (separated by blanks or any other sequence of characters you have defined). The "words" in this file are the numbers:

```
>>> words = filestr.split()
>>> words
['21.8', '18.1', '19', '23', '26', '17.8']
>>> numbers = [float(w) for w in words]
>>> mean = sum(numbers)/len(numbers)
>>> print mean
20.95
```

A more compact program looks as follows (`mean3.py`):

```
infile = open('data.txt', 'r')
numbers = [float(w) for w in infile.read().split()]
mean = sum(numbers)/len(numbers)
```

The next section tells you more about splitting strings.

4.5.3 Reading a mixture of text and numbers

The `data.txt` file has a very simple structure since it contains numbers only. Many data files contain a mix of text and numbers. The file `rainfall.dat` from `www.worldclimate.com`[3] provides an example:

```
Average rainfall (in mm) in Rome: 1188 months between 1782 and 1970
Jan  81.2
Feb  63.2
Mar  70.3
```

[3] `http://www.worldclimate.com/cgi-bin/data.pl?ref=N41E012+2100+1623501G1`

```
Apr  55.7
May  53.0
Jun  36.4
Jul  17.5
Aug  27.5
Sep  60.9
Oct  117.7
Nov  111.0
Dec  97.9
Year 792.9
```

How can we read the rainfall data in this file and store the information in lists suitable for further analysis? The most straightforward solution is to read the file line by line, and for each line split the line into words, store the first word (the month) in one list and the second word (the average rainfall) in another list. The elements in this latter list needs to be `float` objects if we want to compute with them.

The complete code, wrapped in a function, may look like this (file rainfall1.py):

```
def extract_data(filename):
    infile = open(filename, 'r')
    infile.readline() # skip the first line
    months = []
    rainfall = []
    for line in infile:
        words = line.split()
        # words[0]: month, words[1]: rainfall
        months.append(words[0])
        rainfall.append(float(words[1]))
    infile.close()
    months = months[:-1]      # Drop the "Year" entry
    annual_avg = rainfall[-1] # Store the annual average
    rainfall = rainfall[:-1]  # Redefine to contain monthly data
    return months, rainfall, annual_avg

months, values, avg = extract_data('rainfall.dat')
print 'The average rainfall for the months:'
for month, value in zip(months, values):
    print month, value
print 'The average rainfall for the year:', avg
```

Note that the first line in the file is just a comment line and of no interest to us. We therefore read this line by `infile.readline()` and do not store the content in any object. The `for` loop over the lines in the file will then start from the next (second) line.

We store all the data into 13 elements in the `months` and `rainfall` lists. Thereafter, we manipulate these lists a bit since we want `months` to contain the name of the 12 months only. The `rainfall` list should correspond to this `month` list. The annual average is taken out of `rainfall` and stored in a separate variable. Recall that the `-1` index corresponds to the last element of a list, and the slice `:-1` picks out all elements from the start up to, but not including, the last element.

We could, alternatively, have written a shorter code where the name of the months and the rainfall numbers are stored in a nested list:

```
def extract_data(filename):
    infile = open(filename, 'r')
    infile.readline()  # skip the first line
    data = [line.split() for line in infile]
    annual_avg = data[-1][1]
    data = [(m, float(r)) for m, r in data[:-1]]
    infile.close()
    return data, annual_avg
```

This is more advanced code, but understanding what is going on is a good test on the understanding of nested lists indexing and list comprehensions. An executable program is found in the file `rainfall2.py`.

Is it more to file reading? With the example code in this section, you have the very basic tools for reading files with a simple structure: columns of text or numbers. Many files used in scientific computations have such a format, but many files are more complicated too. Then you need the techniques of string processing. This is explained in detail in Chapter 6.

4.6 Writing data to file

Writing data to file is easy. There is basically one function to pay attention to: `outfile.write(s)`, which writes a string `s` to a file handled by the file object `outfile`. Unlike `print`, `outfile.write(s)` does not append a newline character to the written string. It will therefore often be necessary to add a newline character,

```
outfile.write(s + '\n')
```

if the string `s` is meant to appear on a single line in the file and `s` does not already contain a trailing newline character. File writing is then a matter of constructing strings containing the text we want to have in the file and for each such string call `outfile.write`.

Writing to a file demands the file object `f` to be opened for writing:

```
# write to new file, or overwrite file:
outfile = open(filename, 'w')

# append to the end of an existing file:
outfile = open(filename, 'a')
```

4.6.1 Example: Writing a table to file

Problem. As a worked example of file writing, we shall write out a nested list with tabular data to file. A sample list may take look as

```
[[ 0.75,        0.29619813, -0.29619813, -0.75      ],
 [ 0.29619813,  0.11697778, -0.11697778, -0.29619813],
 [-0.29619813, -0.11697778,  0.11697778,  0.29619813],
 [-0.75,       -0.29619813,  0.29619813,  0.75      ]]
```

Solution. We iterate through the rows (first index) in the list, and for each row, we iterate through the column values (second index) and write each value to the file. At the end of each row, we must insert a newline character in the file to get a linebreak. The code resides in the file write1.py:

```
data = [[ 0.75,        0.29619813, -0.29619813, -0.75      ],
        [ 0.29619813,  0.11697778, -0.11697778, -0.29619813],
        [-0.29619813, -0.11697778,  0.11697778,  0.29619813],
        [-0.75,       -0.29619813,  0.29619813,  0.75      ]]

outfile = open('tmp_table.dat', 'w')
for row in data:
    for column in row:
        outfile.write('%14.8f' % column)
    outfile.write('\n')
outfile.close()
```

The resulting data file becomes

```
    0.75000000    0.29619813   -0.29619813   -0.75000000
    0.29619813    0.11697778   -0.11697778   -0.29619813
   -0.29619813   -0.11697778    0.11697778    0.29619813
   -0.75000000   -0.29619813    0.29619813    0.75000000
```

An extension of this program consists in adding column and row headings:

```
            column  1     column  2     column  3     column  4
row  1    0.75000000    0.29619813   -0.29619813   -0.75000000
row  2    0.29619813    0.11697778   -0.11697778   -0.29619813
row  3   -0.29619813   -0.11697778    0.11697778    0.29619813
row  4   -0.75000000   -0.29619813    0.29619813    0.75000000
```

To obtain this end result, we need to the add some statements to the program write1.py. For the column headings we must know the number of columns, i.e., the length of the rows, and loop from 1 to this length:

```
ncolumns = len(data[0])
outfile.write('          ')
for i in range(1, ncolumns+1):
    outfile.write('%10s    ' % ('column %2d' % i))
outfile.write('\n')
```

Note the use of a nested printf construction: the text we want to insert is itself a printf string. We could also have written the text as 'column ' + str(i), but then the length of the resulting string would depend on the number of digits in i. It is recommended to always use printf constructions for a tabular output format, because this gives automatic padding of blanks so that the width of the output strings remains the same. The tuning of the widths is commonly done in a trial-and-error process.

To add the row headings, we need a counter over the row numbers:

```
row_counter = 1
for row in data:
    outfile.write('row %2d' % row_counter)
    for column in row:
```

```
        outfile.write('%14.8f' % column)
    outfile.write('\n')
    row_counter += 1
```

The complete code is found in the file `write2.py`. We could, alternatively, iterate over the indices in the list:

```
for i in range(len(data)):
    outfile.write('row %2d' % (i+1))
    for j in range(len(data[i])):
        outfile.write('%14.8f' % data[i][j])
    outfile.write('\n')
```

4.6.2 Standard input and output as file objects

Reading user input from the keyboard applies the function `raw_input` as explained in Section 4.1. The keyboard is a medium that the computer in fact treats as a file, referred to as *standard input*.

The `print` command prints text in the terminal window. This medium is also viewed as a file from the computer's point of view and called *standard output*. All general-purpose programming languages allow reading from standard input and writing to standard output. This reading and writing can be done with two types of tools, either file-like objects or special tools like `raw_input` and `print` in Python. We will here describe the file-line objects: `sys.stdin` for standard input and `sys.stdout` for standard output. These objects behave as file objects, except that they do not need to be opened or closed. The statement

```
s = raw_input('Give s:')
```

is equivalent to

```
print 'Give s: ',
s = sys.stdin.readline()
```

Recall that the trailing comma in the `print` statement avoids the newline that `print` by default adds to the output string. Similarly,

```
s = eval(raw_input('Give s:'))
```

is equivalent to

```
print 'Give s: ',
s = eval(sys.stdin.readline())
```

For output to the terminal window, the statement

```
print s
```

is equivalent to

```
sys.stdout.write(s + '\n')
```

Why it is handy to have access to standard input and output as file objects can be illustrated by an example. Suppose you have a function that reads data from a file object `infile` and writes data to a file object `outfile`. A sample function may take the form

```
def x2f(infile, outfile, f):
    for line in infile:
        x = float(line)
        y = f(x)
        outfile.write('%g\n' % y)
```

This function works with all types of files, including web pages as `infile` (see Section 6.3). With `sys.stdin` as `infile` and/or `sys.stdout` as `outfile`, the `x2f` function also works with standard input and/or standard output. Without `sys.stdin` and `sys.stdout`, we would need different code, employing `raw_input` and `print`, to deal with standard input and output. Now we can write a single function that deals with all file media in a unified way.

There is also something called *standard error*. Usually this is the terminal window, just as standard output, but programs can distinguish between writing ordinary output to standard output and error messages to standard error, and these output media can be redirected to, e.g., files such that one can separate error messages from ordinary output. In Python, standard error is the file-like object `sys.stderr`. A typical application of `sys.stderr` is to report errors:

```
if x < 0:
    sys.stderr.write('Illegal value of x'); sys.exit(1)
```

This message to `sys.stderr` is an alternative to `print` or raising an exception.

Redirecting standard input, output, and error. Standard output from a program `prog` can be redirected to a file `output` instead of the screen, by using the greater than sign:

Terminal

```
Terminal> prog > output
```

Here, `prog` can be any program, including a Python program run as `python myprog.py`. Similarly, output to the medium called *standard error* can be redirected by

Terminal

```
Terminal> prog &> output
```

For example, error messages are normally written to standard error, which is exemplified in this little terminal session on a Unix machine:

Terminal

```
Terminal> ls bla-bla1 bla-bla2
ls: cannot access bla-bla1: No such file or directory
ls: cannot access bla-bla2: No such file or directory
Terminal> ls bla-bla1 bla-bla2 &> errors
Terminal> cat errors  # print the file errors
ls: cannot access bla-bla1: No such file or directory
ls: cannot access bla-bla2: No such file or directory
```

When the program reads from standard input (the keyboard), we can equally well redirect standard input from a file, say with name `input`, such that the program reads from this file rather than from the keyboard:

Terminal

```
Terminal> prog < input
```

Combinations are also possible:

Terminal

```
Terminal> prog < input > output
```

Note. The redirection of standard output, input, and error does not work for Python programs executed with the `run` command inside IPython, only when executed directly in the operating system in a terminal window, or with the same command prefixed with an exclamation mark in IPython.

Inside a Python program we can also let standard input, output, and error work with ordinary files instead. Here is the technique:

```
sys_stdout_orig = sys.stdout
sys.stdout = open('output', 'w')
sys_stdin_orig = sys.stdin
sys.stdin = open('input', 'r')
```

Now, any `print` statement will write to the `output` file, and any `raw_input` call will read from the `input` file. (Without storing the original `sys.stdout` and `sys.stdin` objects in new variables, these objects would get lost in the redefinition above and we would never be able to reach the common standard input and output in the program.)

4.6.3 What is a file, really?

This section is not mandatory for understanding the rest of the book. Nevertheless, the information here is fundamental for understanding what files are about.

A file is simply a sequence of characters. In addition to the sequence of characters, a file has some data associated with it, typically the name of the file, its location on the disk, and the file size. These data are stored somewhere by the operating system. Without this extra information beyond the pure file contents as a sequence of characters, the operating system cannot find a file with a given name on the disk.

Each character in the file is represented as a *byte*, consisting of eight *bits*. Each bit is either 0 or 1. The zeros and ones in a byte can be combined in $2^8 = 256$ ways. This means that there are 256 different types of characters. Some of these characters can be recognized from the keyboard, but there are also characters that do not have a familiar symbol. Such characters looks cryptic when printed.

Pure text files. To see that a file is really just a sequence of characters, invoke an editor for plain text, typically the editor you use to write Python programs. Write the four characters `ABCD` into the editor, do not press the Return key, and save the text to a file `test1.txt`. Use your favorite tool for file and folder overview and move to the folder containing the `test1.txt` file. This tool may be Windows Explorer, My Computer, or a DOS window on Windows; a terminal window, Konqueror, or Nautilus on Linux; or a terminal window or Finder on Mac. If you choose a terminal window, use the `cd` (change directory) command to move to the proper folder and write `dir` (Windows) or `ls -l` (Linux/Mac) to list the files and their sizes. In a graphical program like Windows Explorer, Konqueror, Nautilus, or Finder, select a view that shows the *size* of each file (choose *view as details* in Windows Explorer, *View as List* in Nautilus, the list view icon in Finder, or you just point at a file icon in Konqueror and watch the pop-up text). You will see that the `test1.txt` file has a size of 4 bytes (if you use `ls -l`, the size measured in bytes is found in column 5, right before the date). The 4 bytes are exactly the 4 characters `ABCD` in the file. Physically, the file is just a sequence of 4 bytes on your hard disk.

Go back to the editor again and add a newline by pressing the Return key. Save this new version of the file as `test2.txt`. When you now check the size of the file it has grown to five bytes. The reason is that we added a newline character (symbolically known as *backslash n*: `\n`).

Instead of examining files via editors and folder viewers we may use Python interactively:

```
>>> file1 = open('test1.txt', 'r').read()  # read file into string
>>> file1
'ABCD'
>>> len(file1)        # length of string in bytes/characters
4
>>> file2 = open('test2.txt', 'r').read()
>>> file2
'ABCD\n'
>>> len(file2)
5
```

Python has in fact a function that returns the size of a file directly:

```
>>> import os
>>> size = os.path.getsize('test1.txt')
>>> size
4
```

Word processor files. Most computer users write text in a word processing program, such as Microsoft Word or LibreOffice. Let us investigate what happens with our four characters `ABCD` in such a program. Start the word processor, open a new document, and type in the four characters `ABCD` only. Save the document as a `.docx` file (Microsoft Word) or an `.odt` file (LibreOffice). Load this file into an editor for pure text and look at the contents. You will see that there are numerous strange characters that you did not write (!). This additional "text" contains information on what type of document this is, the font you used, etc. The LibreOffice version of this file has 8858 bytes and the Microsoft Word version contains over 26 Kb! However, if you save the file as a pure text file, with extension `.txt`, the size is down to 8 bytes in LibreOffice and five in Microsoft Word.

Instead of loading the LibreOffice file into an editor we can again read the file contents into a string in Python and examine this string:

```
>>> infile = open('test3.odt', 'r')  # open LibreOffice file
>>> s = infile.read()
>>> len(s)   # file size
8858
>>> s
'PK\x03\x04\x14\x00\x00\x08\x00\x00sKWD^\xc62\x0c\'\x00...
\x00meta.xml<?xml version="1.0" encoding="UTF-8"?>\n<office:...
" xmlns:meta="urn:oasis:names:tc:opendocument:xmlns:meta:1.0"
```

Each backslash followed by `x` and a number is a code for a special character not found on the keyboard (recall that there are 256 characters and only a subset is associated with keyboard symbols). Although we show just a small portion of all the characters in this file in the above output (otherwise, the output would have occupied several pages in this book with thousands symbols like `\x04`...), we can guarantee that you cannot find the pure sequence of characters `ABCD`. However, the computer program that generated the file, LibreOffice in this example, can easily interpret the meaning of all the characters in the file and translate the information into nice, readable text on the screen where you can recognize the text `ABCD`.

Your are now in a position to look into Exercise 4.8 to see what happens if one attempts to use LibreOffice to write Python programs.

Image files. A digital image - captured by a digital camera or a mobile phone - is a file. And since it is a file, the image is just a sequence of characters. Loading some JPEG file into a pure text editor, reveals all the strange characters in there. On the first line you will (normally) find some recognizable text in between the strange characters. This text reflects the type of camera used to capture the image and the date and time when the picture was taken. The next lines contain more information about the image. Thereafter, the file contains a set of numbers representing the image. The basic representation of an image is a set of $m \times n$ pixels, where each pixel has a color represented as a combination of 256 values

of red, green, and blue, which can be stored as three bytes (resulting in 256^3 color values). A 6-megapixel camera will then need to store $3 \times 6 \cdot 10^6 = 18$ megabytes for one picture. The JPEG file contains only a couple of megabytes. The reason is that JPEG is a *compressed* file format, produced by applying a smart technique that can throw away pixel information in the original picture such that the human eye hardly can detect the inferior quality.

A video is just a sequence of images, and therefore a video is also a stream of bytes. If the change from one video frame (image) to the next is small, one can use smart methods to compress the image information in time. Such compression is particularly important for videos since the file sizes soon get too large for being transferred over the Internet. A small video file occasionally has bad visual quality, caused by too much compression.

Music files. An MP3 file is much like a JPEG file: first, there is some information about the music (artist, title, album, etc.), and then comes the music itself as a stream of bytes. A typical MP3 file has a size of something like five million bytes or five megabytes (5 Mb). The exact size depends on the complexity of the music, the length of the track, and the MP3 resolution. On a 16 Gb MP3 player you can then store roughly $16,000,000,000/5,000,000 = 3200$ MP3 files. MP3 is, like JPEG, a compressed format. The complete data of a song on a CD (the WAV file) contains about ten times as many bytes. As for pictures, the idea is that one can throw away a lot of bytes in an intelligent way, such that the human ear hardly detects the difference between a compressed and uncompressed version of the music file.

PDF files. Looking at a PDF file in a pure text editor shows that the file contains some readable text mixed with some unreadable characters. It is not possible for a human to look at the stream of bytes and deduce the text in the document (well, from the assumption that there are always some strange people doing strange things, there might be somebody out there who, with a lot of training, can interpret the pure PDF code with the eyes). A PDF file reader can easily interpret the contents of the file and display the text in a human-readable form on the screen.

Remarks. We have repeated many times that a file is just a stream of bytes. A human can interpret (read) the stream of bytes if it makes sense in a human language - or a computer language (provided the human is a programmer). When the series of bytes does not make sense to any human, a computer program must be used to interpret the sequence of characters.

Think of a report. When you write the report as pure text in a text editor, the resulting file contains just the characters you typed in from the keyboard. On the other hand, if you applied a word processor like Microsoft Word or LibreOffice, the report file contains a large number

of extra bytes describing properties of the formatting of the text. This stream of extra bytes does not make sense to a human, and a computer program is required to interpret the file content and display it in a form that a human can understand. Behind the sequence of bytes in the file there are strict rules telling what the series of bytes means. These rules reflect the *file format*. When the rules or file format is publicly documented, a programmer can use this documentation to make her own program for interpreting the file contents (however, interpreting such files is much more complicated than our examples on reading human-readable files in this book). It happens, though, that secret file formats are used, which require certain programs from certain companies to interpret the files.

4.7 Handling errors

Suppose we forget to provide a command-line argument to the `c2f_cml.py` program from Section 4.2.1:

Terminal

```
c2f_cml.py
Traceback (most recent call last):
  File "c2f_cml.py", line 2, in ?
    C = float(sys.argv[1])
IndexError: list index out of range
```

Python aborts the program and shows an error message containing the line where the error occurred, the type of the error (`IndexError`), and a quick explanation of what the error is. From this information we deduce that the index 1 is out of range. Because there are no command-line arguments in this case, `sys.argv` has only one element, namely the program name. The only valid index is then 0.

For an experienced Python programmer this error message will normally be clear enough to indicate what is wrong. For others it would be very helpful if wrong usage could be detected by our program and a description of correct operation could be printed. The question is how to detect the error inside the program.

The problem in our sample execution is that `sys.argv` does not contain two elements (the program name, as always, plus one command-line argument). We can therefore test on the length of `sys.argv` to detect wrong usage: if `len(sys.argv)` is less than 2, the user failed to provide information on the `C` value. The new version of the program, `c2f_cml_if.py`, starts with this `if` test:

```
if len(sys.argv) < 2:
    print 'You failed to provide Celsius degrees as input '\
          'on the command line!'
    sys.exit(1)  # abort because of error
```

```
F = 9.0*C/5 + 32
print '%gC is %.1fF' % (C, F)
```

We use the `sys.exit` function to abort the program. Any argument different from zero signifies that the program was aborted due to an error, but the precise value of the argument does not matter so here we simply choose it to be `1`. If no errors are found, but we still want to abort the program, `sys.exit(0)` is used.

A more modern and flexible way of handling potential errors in a program is to *try* to execute some statements, and if something goes wrong, the program can detect this and jump to a set of statements that handle the erroneous situation as desired. The relevant program construction reads

```
try:
    <statements>
except:
    <statements>
```

If something goes wrong when executing the statements in the `try` block, Python raises what is known as an *exception*. The execution jumps directly to the `except` block whose statements can provide a remedy for the error. The next section explains the `try-except` construction in more detail through examples.

4.7.1 Exception handling

To clarify the idea of exception handling, let us use a `try-except` block to handle the potential problem arising when our Celsius-Fahrenheit conversion program lacks a command-line argument:

```
import sys
try:
    C = float(sys.argv[1])
except:
    print 'You failed to provide Celsius degrees as input '\
          'on the command line!'
    sys.exit(1)  # abort
F = 9.0*C/5 + 32
print '%gC is %.1fF' % (C, F)
```

The program is stored in the file `c2f_cml_except1.py`. If the command-line argument is missing, the indexing `sys.argv[1]`, which has an invalid index `1`, *raises an exception*. This means that the program jumps directly to the `except` block, implying that `float` is not called, and `C` is not initialized with a value. In the except block, the programmer can retrieve information about the exception and perform statements to recover from the error. In our example, we know what the error can be, and therefore we just print a message and abort the program.

Suppose the user provides a command-line argument. Now, the `try` block is executed successfully, and the program neglects the `except` block

and continues with the Fahrenheit conversion. We can try out the last program in two cases:

Terminal

```
c2f_cml_except1.py
You failed to provide Celsius degrees as input on the command line!

c2f_cml_except1.py 21
21C is 69.8F
```

In the first case, the illegal index in `sys.argv[1]` causes an exception to be raised, and we perform the steps in the `except` block. In the second case, the `try` block executes successfully, so we jump over the `except` block and continue with the computations and the printout of results.

For a user of the program, it does not matter if the programmer applies an `if` test or exception handling to recover from a missing command-line argument. Nevertheless, exception handling is considered a better programming solution because it allows more advanced ways to abort or continue the execution. Therefore, we adopt exception handling as our standard way of dealing with errors in the rest of this book.

Testing for a specific exception. Consider the assignment

```
C = float(sys.argv[1])
```

There are two typical errors associated with this statement: i) `sys.argv[1]` is illegal indexing because no command-line arguments are provided, and ii) the content in the string `sys.argv[1]` is not a pure number that can be converted to a `float` object. Python detects both these errors and raises an `IndexError` exception in the first case and a `ValueError` in the second. In the program above, we jump to the `except` block and issue the same message regardless of what went wrong in the `try` block. For example, when we indeed provide a command-line argument, but write it on an illegal form (`21C`), the program jumps to the `except` block and prints a misleading message:

Terminal

```
c2f_cml_except1.py 21C
You failed to provide Celsius degrees as input on the command line!
```

The solution to this problem is to branch into different `except` blocks depending on what type of exception that was raised in the `try` block (program c2f_cml_except2.py):

```
import sys
try:
    C = float(sys.argv[1])
except IndexError:
    print 'Celsius degrees must be supplied on the command line'
    sys.exit(1)  # abort execution
except ValueError:
```

```
    print 'Celsius degrees must be a pure number, '\
          'not "%s"' % sys.argv[1]
    sys.exit(1)

F = 9.0*C/5 + 32
print '%gC is %.1fF' % (C, F)
```

Now, if we fail to provide a command-line argument, an `IndexError` occurs and we tell the user to write the `C` value on the command line. On the other hand, if the `float` conversion fails, because the command-line argument has wrong syntax, a `ValueError` exception is raised and we branch into the second `except` block and explain that the form of the given number is wrong:

Terminal

```
c2f_cml_except1.py 21C
Celsius degrees must be a pure number, not "21C"
```

Examples on exception types. List indices out of range lead to `IndexError` exceptions:

```
>>> data = [1.0/i for i in range(1,10)]
>>> data[9]
...
IndexError: list index out of range
```

Some programming languages (Fortran, C, C++, and Perl are examples) allow list indices outside the legal index values, and such unnoticed errors can be hard to find. Python always stops a program when an invalid index is encountered, unless you handle the exception explicitly as a programmer.

Converting a string to `float` is unsuccessful and gives a `ValueError` if the string is not a pure integer or real number:

```
>>> C = float('21 C')
...
ValueError: invalid literal for float(): 21 C
```

Trying to use a variable that is not initialized gives a `NameError` exception:

```
>>> print a
...
NameError: name 'a' is not defined
```

Division by zero raises a `ZeroDivisionError` exception:

```
>>> 3.0/0
...
ZeroDivisionError: float division
```

Writing a Python keyword illegally or performing a Python grammar error leads to a `SyntaxError` exception:

```
>>> forr d in data:
...
    forr d in data:
          ^
SyntaxError: invalid syntax
```

What if we try to multiply a string by a number?

```
>>> 'a string'*3.14
...
TypeError: can't multiply sequence by non-int of type 'float'
```

The `TypeError` exception is raised because the object types involved in the multiplication are wrong (`str` and `float`).

Digression. It might come as a surprise, but multiplication of a string and a number is legal if the number is an integer. The multiplication means that the string should be repeated the specified number of times. The same rule also applies to lists:

```
>>> '--'*10   # ten double dashes = 20 dashes
'--------------------'
>>> n = 4
>>> [1, 2, 3]*n
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> [0]*n
[0, 0, 0, 0]
```

The latter construction is handy when we want to create a list of `n` elements and later assign specific values to each element in a `for` loop.

4.7.2 Raising exceptions

When an error occurs in your program, you may either print a message and use `sys.exit(1)` to abort the program, or you may raise an exception. The latter task is easy. You just write `raise E(message)`, where `E` can be a known exception type in Python and `message` is a string explaining what is wrong. Most often `E` means `ValueError` if the value of some variable is illegal, or `TypeError` if the type of a variable is wrong. You can also define your own exception types. An exception can be raised from any location in a program.

Example. In the program `c2f_cml_except2.py` from Section 4.7.1 we show how we can test for different exceptions and abort the program. Sometimes we see that an exception may happen, but if it happens, we want a more precise error message to help the user. This can be done by raising a new exception in an `except` block and provide the desired exception type and message.

Another application of raising exceptions with tailored error messages arises when input data are invalid. The code below illustrates how to raise exceptions in various cases.

We collect the reading of `C` and handling of errors a separate function:

```
def read_C():
    try:
        C = float(sys.argv[1])
    except IndexError:
        raise IndexError\
        ('Celsius degrees must be supplied on the command line')
    except ValueError:
        raise ValueError\
        ('Celsius degrees must be a pure number, '\
        'not "%s"' % sys.argv[1])
    # C is read correctly as a number, but can have wrong value:
    if C < -273.15:
        raise ValueError('C=%g is a non-physical value!' % C)
    return C
```

There are two ways of using the `read_C` function. The simplest is to call the function,

```
C = read_C()
```

Wrong input will now lead to a raw dump of exceptions, e.g.,

Terminal

```
c2f_cml_v5.py
Traceback (most recent call last):
  File "c2f_cml4.py", line 5, in ?
    raise IndexError\
IndexError: Celsius degrees must be supplied on the command line
```

New users of this program may become uncertain when getting raw output from exceptions, because words like `Traceback`, `raise`, and `IndexError` do not make much sense unless you have some experience with Python. A more user-friendly output can be obtained by calling the `read_C` function inside a `try-except` block, check for any exception (or better: check for `IndexError` *or* `ValueError`), and write out the exception message in a more nicely formatted form. In this way, the programmer takes complete control of how the program behaves when errors are encountered:

```
try:
    C = read_C()
except Exception as e:
    print e            # exception message
    sys.exit(1)        # terminate execution
```

`Exception` is the parent name of all exceptions, and `e` is an exception object. Nice printout of the exception message follows from a straight `print e`. Instead of `Exception` we can write `(ValueError, IndexError)` to test more specifically for two exception types we can expect from the `read_C` function:

```
try:
    C = read_C()
except (ValueError, IndexError) as e:
    print e            # exception message
    sys.exit(1)        # terminate execution
```

After the `try-except` block above, we can continue with computing `F = 9*C/5 + 32` and print out `F`. The complete program is found in the file `c2f_cml.py`. We may now test the program's behavior when the input is wrong and right:

Terminal

```
c2f_cml.py
Celsius degrees must be supplied on the command line

c2f_cml.py 21C
Celsius degrees must be a pure number, not "21C"

c2f_cml.py -500
C=-500 is a non-physical value!

c2f_cml.py 21
21C is 69.8F
```

This program deals with wrong input, writes an informative message, and terminates the execution without annoying behavior.

Scattered `if` tests with `sys.exit` calls are considered a bad programming style compared to the use of nested exception handling as illustrated above. You should abort execution in the main program only, not inside functions. The reason is that the functions can be re-used in other occasions where the error can be dealt with differently. For instance, one may avoid abortion by using some suitable default data.

The programming style illustrated above is considered the best way of dealing with errors, so we suggest that you hereafter apply exceptions for handling potential errors in the programs you make, simply because this is what experienced programmers expect from your codes.

4.8 A glimpse of graphical user interfaces

Maybe you find it somewhat strange that the usage of the programs we have made so far in this book - and the programs we will make in the rest of the book - are less graphical and intuitive than the computer programs you are used to from school or entertainment. Those programs are operated through some self-explaining graphics, and most of the things you want to do involve pointing with the mouse, clicking on graphical elements on the screen, and maybe filling in some text fields. The programs in this book, on the other hand, are run from the command line in a terminal window or inside IPython, and input is also given here in form of plain text.

The reason why we do not equip the programs in this book with graphical interfaces for providing input, is that such graphics is both complicated and tedious to write. If the aim is to solve problems from mathematics and science, we think it is better to focus on this part rather than large amounts of code that merely offers some "expected"

graphical cosmetics for putting data into the program. Textual input from the command line is also quicker to provide. Also remember that the computational functionality of a program is obviously independent from the type of user interface, textual or graphic.

As an illustration, we shall now show a Celsius to Fahrenheit conversion program with a graphical user interface (often called a GUI). The GUI is shown in Figure 4.1. We encourage you to try out the graphical interface - the name of the program is `c2f_gui.py`. The complete program text is listed below.

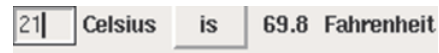

Fig. 4.1 Screen dump of the graphical interface for a Celsius to Fahrenheit conversion program. The user can type in the temperature in Celsius degrees, and when clicking on the *is* button, the corresponding Fahrenheit value is displayed.

```
from Tkinter import *
root = Tk()
C_entry = Entry(root, width=4)
C_entry.pack(side='left')
Cunit_label = Label(root, text='Celsius')
Cunit_label.pack(side='left')

def compute():
    C = float(C_entry.get())
    F = (9./5)*C + 32
    F_label.configure(text='%g' % F)

compute = Button(root, text=' is ', command=compute)
compute.pack(side='left', padx=4)

F_label = Label(root, width=4)
F_label.pack(side='left')
Funit_label = Label(root, text='Fahrenheit')
Funit_label.pack(side='left')

root.mainloop()
```

The goal of the forthcoming dissection of this program is to give a taste of how graphical user interfaces are coded. The aim is not to equip you with knowledge on how you can make such programs on your own.

A GUI is built of many small graphical elements, called *widgets*. The graphical window generated by the program above and shown in Figure 4.1 has five such widgets. To the left there is an *entry* widget where the user can write in text. To the right of this entry widget is a *label* widget, which just displays some text, here "Celsius". Then we have a *button* widget, which when being clicked leads to computations in the program. The result of these computations is displayed as text in a *label* widget to the right of the button widget. Finally, to the right of this result text we have another *label* widget displaying the text "Fahrenheit". The program must construct each widget and pack it correctly into the

complete window. In the present case, all widgets are packed from left to right.

The first statement in the program imports functionality from the GUI toolkit `Tkinter` to construct widgets. First, we need to make a root widget that holds the complete window with all the other widgets. This root widget is of type `Tk`. The first entry widget is then made and referred to by a variable `C_entry`. This widget is an object of type `Entry`, provided by the `Tkinter` module. Widgets constructions follow the syntax

```
variable_name = Widget_type(parent_widget, option1, option2, ...)
variable_name.pack(side='left')
```

When creating a widget, we must bind it to a *parent widget*, which is the graphical element in which this new widget is to be packed. Our widgets in the present program have the `root` widget as parent widget. Various widgets have different types of options that we can set. For example, the `Entry` widget has a possibility for setting the width of the text field, here `width=4` means that the text field is 4 characters wide. The pack statement is important to remember - without it, the widget remains invisible.

The other widgets are constructed in similar ways. The next fundamental feature of our program is how computations are tied to the event of clicking the button *is*. The `Button` widget has naturally a text, but more important, it binds the button to a function `compute` through the `command=compute` option. This means that when the user clicks the button *is*, the function `compute` is called. Inside the `compute` function we first fetch the Celsius value from the `C_entry` widget, using this widget's `get` function, then we transform this string (everything typed in by the user is interpreted as text and stored in strings) to a `float` before we compute the corresponding Fahrenheit value. Finally, we can update (`configure`) the text in the `Label` widget `F_label` with a new text, namely the computed degrees in Fahrenheit.

A program with a GUI behaves differently from the programs we construct in this book. First, all the statements are executed from top to bottom, as in all our other programs, but these statements just construct the GUI and define functions. No computations are performed. Then the program enters a so-called *event loop*: `root.mainloop()`. This is an infinite loop that "listens" to user events, such as moving the mouse, clicking the mouse, typing characters on the keyboard, etc. When an event is recorded, the program starts performing associated actions. In the present case, the program waits for only one event: clicking the button *is*. As soon as we click on the button, the `compute` function is called and the program starts doing mathematical work. The GUI will appear on the screen until we destroy the window by click on the X up in the corner

of the window decoration. More complicated GUIs will normally have a special *Quit* button to terminate the event loop.

In all GUI programs, we must first create a hierarchy of widgets to build up all elements of the user interface. Then the program enters an event loop and waits for user events. Lots of such events are registered as actions in the program when creating the widgets, so when the user clicks on buttons, move the mouse into certain areas, etc., functions in the program are called and "things happen".

Many books explain how to make GUIs in Python programs, see for instance [4, 6, 13, 16].

4.9 Making modules

Sometimes you want to reuse a function from an old program in a new program. The simplest way to do this is to copy and paste the old source code into the new program. However, this is not good programming practice, because you then over time end up with multiple identical versions of the same function. When you want to improve the function or correct a bug, you need to remember to do the same update in all files with a copy of the function, and in real life most programmers fail to do so. You easily end up with a mess of different versions with different quality of basically the same code. Therefore, a golden rule of programming is to have one and only one version of a piece of code. All programs that want to use this piece of code must access one and only one place where the source code is kept. This principle is easy to implement if we create a module containing the code we want to reuse later in different programs.

When reading this, you probably know how to use a ready-made module. For example, if you want to compute the factorial $k! = k(k-1)(k-2)\cdots 1$, there is a function `factorial` in Python's `math` module that can be help us out. The usage goes with the `math` prefix,

```
import math
value = math.factorial(5)
```

or without,

```
from math import factorial
# or: from math import *
value = factorial(5)
```

Now you shall learn how to make your own Python modules. There is hardly anything to learn, because you just collect all the functions that constitute the module in one file, say with name `mymodule.py`. This file is automatically a module, with name `mymodule`, and you can import functions from this module in the standard way. Let us make everything clear in detail by looking at an example.

4.9.1 Example: Interest on bank deposits

The classical formula for the growth of money in a bank reads

$$A = A_0\left(1 + \frac{p}{360\cdot 100}\right)^n, \tag{4.2}$$

where A_0 is the initial amount of money, and A is the present amount after n days with p percent annual interest rate. (The formula applies the convention that the rate per day is computed as $p/360$, while n counts the actual number of days the money is in the bank, see the Wikipedia entry Day count convention[4] for explanation. There is a handy Python module `datetime` for computing the number of days between two dates.)

Equation (4.2) involves four parameters: A, A_0, p, and n. We may solve for any of these, given the other three:

$$A_0 = A\left(1 + \frac{p}{360\cdot 100}\right)^{-n}, \tag{4.3}$$

$$n = \frac{\ln \frac{A}{A_0}}{\ln\left(1 + \frac{p}{360\cdot 100}\right)}, \tag{4.4}$$

$$p = 360\cdot 100\left(\left(\frac{A}{A_0}\right)^{1/n} - 1\right). \tag{4.5}$$

Suppose we have implemented (4.2)-(4.5) in four functions:

```
from math import log as ln

def present_amount(A0, p, n):
    return A0*(1 + p/(360.0*100))**n

def initial_amount(A, p, n):
    return A*(1 + p/(360.0*100))**(-n)

def days(A0, A, p):
    return ln(A/A0)/ln(1 + p/(360.0*100))

def annual_rate(A0, A, n):
    return 360*100*((A/A0)**(1.0/n) - 1)
```

We want to make these functions available in a module, say with name `interest`, so that we can import functions and compute with them in a program. For example,

```
from interest import days
A0 = 1; A = 2; p = 5
n = days(A0, 2, p)
years = n/365.0
print 'Money has doubled after %.1f years' % years
```

How to make the `interest` module is described next.

[4] http://en.wikipedia.org/wiki/Day_count_convention

4.9.2 Collecting functions in a module file

To make a module of the four functions `present_amount`, `initial_amount`, `days`, and `annual_rate`, we simply open an empty file in a text editor and copy the program code for all the four functions over to this file. This file is then automatically a Python module provided we save the file under any valid filename. The extension must be `.py`, but the module name is only the base part of the filename. In our case, the filename `interest.py` implies a module name `interest`. To use the `annual_rate` function in another program we simply write, in that program file,

```
from interest import annual_rate
```

or we can write

```
from interest import *
```

to import all four functions, or we can write

```
import interest
```

and access individual functions as `interest.annual_rate` and so forth.

4.9.3 Test block

It is recommended to only have functions and not any statements outside functions in a module. The reason is that the module file is executed from top to bottom during the import. With function definitions only in the module file, and no main program, there will be no calculations or output from the import, just definitions of functions. This is the desirable behavior. However, it is often convenient to have test or demonstrations in the module file, and then there is need for a main program. Python allows a very fortunate construction to let the file act both as a module with function definitions only (and no main program) *and* as an ordinary program we can run, with functions and a main program.

This two-fold "magic" is realized by putting the main program after an `if` test of the form

```
if __name__ == '__main__':
    <block of statements>
```

The `__name__` variable is automatically defined in any module and equals the module name if the module file is imported in another program, or `__name__` equals the string `'__main__'` if the module file is run as a program. This implies that the `<block of statements>` part is executed if and only if we run the module file as a program. We shall refer to `<block of statements>` as the *test block* of a module.

Example on a test block in a minimalistic module. A very simple example will illustrate how this works. Consider a file `mymod.py` with the content

```
def add1(x):
    return x + 1

if __name__ == '__main__':
    print 'run as program'
    print add1(float(sys.argv[1]))
```

We can import `mymod` as a module and make use of the `add1` function:

```
>>> import mymod
>>> print mymod.add1(4)
5
```

During the import, the `if` test is false, and the only the function definition is executed. However, if we run `mymod.py` as a program,

Terminal

```
mymod.py 5
run as program
6
```

the `if` test becomes true, and the `print` statements are executed.

Tip on easy creation of a module

If you have some functions and a main program in some program file, just move the main program to the test block. Then the file can act as a module, giving access to all the functions in other files, or the file can be executed from the command line, in the same way as the original program.

A test block in the `interest` module. Let us write a little main program for demonstrating the `interest` module in a test block. We read p from the command line and write out how many years it takes to double an amount with that interest rate:

```
if __name__ == '__main__':
    import sys
    p = float(sys.argv[1])
    years = days(1, 2, p)
    print 'With p=%.2f it takes %.1 years to double' % (p, years)
```

Running the module file as a program gives this output:

Terminal

```
interest.py 2.45
With p=2.45 it takes 27.9 years to double
```

To test that the `interest.py` file also works as a module, invoke a Python shell and try to import a function and compute with it:

```
>>> from interest import present_amount
>>> present_amount(2, 5, 730)
2.2133983053266699
```

We have hence demonstrated that the file `interest.py` works both as a program and as a module.

Recommended practice in a test block

It is a good programming habit to let the test block do one or more of three things:

- provide information on how the module or program is used,
- test if the module functions work properly,
- offer interaction with users such that the module file can be applied as a useful program.

Instead of having a lot of statements in the test block, it is better to collect the statements in separate functions, which then are called from the test block.

4.9.4 Verification of the module code

Functions that verify the implementation in a module should

- have names starting with `test_`,
- express the success or failure of a test through a boolean variable, say `success`,
- run `assert success, msg` to raise an `AssertionError` with an optional message `msg` in case the test fails.

Adopting this style makes it trivial to let the tools *pytest* or *nose* automatically run through all our `test_*()` functions in all files in a folder tree. A very brief introduction to test functions compatible with pytest and nose is provided in Section 3.4.2, while Section H.6 contains a more thorough introduction to the pytest and nose testing frameworks for beginners.

A proper test function for verifying the functionality of the `interest` module, written in a way that is compatible with the pytest and nose testing frameworks, looks as follows:

```
def test_all_functions():
    # Compatible values
    A = 2.2133983053266699; A0 = 2.0; p = 5; n = 730
    # Given three of these, compute the remaining one
    # and compare with the correct value (in parenthesis)
    A_computed  = present_amount(A0, p, n)
    A0_computed = initial_amount(A, p, n)
    n_computed  = days(A0, A, p)
    p_computed  = annual_rate(A0, A, n)

    def float_eq(a, b, tolerance=1E-14):
        """Return True if a == b within the tolerance."""
        return abs(a - b) < tolerance

    success = float_eq(A_computed,  A)  and \
              float_eq(A0_computed, A0) and \
              float_eq(p_computed,  p)  and \
              float_eq(n_computed,  n)
    msg = """Computations failed (correct answers in parenthesis):
A=%g (%g)
A0=%g (%.1f)
n=%d (%d)
p=%g (%.1f)""" % (A_computed, A, A0_computed, A0,
                  n_computed, n, p_computed, p)
    assert success, msg
```

We may require a single command-line argument `test` to run the verification. The test block can then be expressed as

```
if __name__ == '__main__':
    if len(sys.argv) == 2 and sys.argv[1] == 'test':
        test_all_functions()
```

4.9.5 Getting input data

To make a useful program, we should allow setting three parameters on the command line and let the program compute the remaining parameter. For example, running the program as

Terminal

```
interest.py A0=2 A=1 n=1095
```

will lead to a computation of p, in this case for seeing the size of the annual interest rate if the amount is to be doubled after three years.

How can we achieve the desired functionality? Since variables are already introduced and "initialized" on the command line, we could grab this text and execute it as Python code, either as three different lines or with semicolon between each assignment. This is easy:

```
init_code = ''
for statement in sys.argv[1:]:
    init_code += statement + '\n'
exec(init_code)
```

(We remark that an experienced Python programmer would have created `init_code` by `'\n'.join(sys.argv[1:])`.) For the sample run above with `A0=2 A=1 n=1095` on the command line, `init_code` becomes the string

```
A0=2
A=1
n=1095
```

Note that one cannot have spaces around the equal signs on the command line as this will break an assignment like `A0 = 2` into three command-line arguments, which will give rise to a `SyntaxError` in `exec(init_code)`. To tell the user about such errors, we execute `init_code` inside a `try-except` block:

```
try:
    exec(init_code)
except SyntaxError as e:
    print e
    print init_code
    sys.exit(1)
```

At this stage, our program has hopefully initialized three parameters in a successful way, and it remains to detect the remaining parameter to be computed. The following code does the work:

```
if 'A=' not in init_code:
    print 'A =', present_amount(A0, p, n)
elif 'A0=' not in init_code:
    print 'A0 =', initial_amount(A, p, n)
elif 'n=' not in init_code:
    print 'n =', days(A0, A , p)
elif 'p=' not in init_code:
    print 'p =', annual_rate(A0, A, n)
```

It may happen that the user of the program assigns value to a parameter with wrong name or forget a parameter. In those cases we call one of our four functions with uninitialized arguments, and Python raises an exception. Therefore, we should embed the code above in a `try-except` block. An uninitialized variable will lead to a `NameError` exception, while another frequent error is illegal values in the computations, leading to a `ValueError` exception. It is also a good habit to collect all the code related to computing the remaining, fourth parameter in a function for separating this piece of code from other parts of the module file:

```
def compute_missing_parameter(init_code):
    try:
        exec(init_code)
    except SyntaxError as e:
        print e
        print init_code
        sys.exit(1)
    # Find missing parameter
    try:
        if 'A=' not in init_code:
            print 'A =', present_amount(A0, p, n)
        elif 'A0=' not in init_code:
```

```
            print 'A0 =', initial_amount(A, p, n)
        elif 'n=' not in init_code:
            print 'n =', days(A0, A , p)
        elif 'p=' not in init_code:
            print 'p =', annual_rate(A0, A, n)
    except NameError as e:
        print e
        sys.exit(1)
    except ValueError:
        print 'Illegal values in input:', init_code
        sys.exit(1)
```

If the user of the program fails to give any command-line arguments, we print a usage statement. Otherwise, we run a verification if the first command-line argument is `test`, and else we run the missing parameter computation (i.e., the useful main program):

```
_filename = sys.argv[0]
_usage = """
Usage: %s A=10 p=5 n=730
Program computes and prints the 4th parameter'
(A, A0, p, or n)""" % _filename

if __name__ == '__main__':
    if len(sys.argv) == 1:
        print _usage
    elif len(sys.argv) == 2 and sys.argv[1] == 'test':
        test_all_functions()
    else:
        init_code = ''
        for statement in sys.argv[1:]:
            init_code += statement + '\n'
        compute_missing_parameter(init_code)
```

Executing user input can be dangerous

Some purists would never demonstrate `exec` the way we do above. The reason is that our program tries to execute whatever the user writes. Consider

Terminal

```
input.py 'import shutil; shutil.rmtree("/")'
```

This evil use of the program leads to an attempt to remove all files on the computer system (the same as writing `rm -rf /` in the terminal window!). However, for small private programs helping the program writer out with mathematical calculations, this potential dangerous misuse is not so much of a concern (the user just does harm to his own computer anyway).

4.9.6 Doc strings in modules

It is also a good habit to include a doc string in the beginning of the module file. This doc string explains the purpose and use of the module:

```
"""
Module for computing with interest rates.
Symbols: A is present amount, A0 is initial amount,
n counts days, and p is the interest rate per year.

Given three of these parameters, the fourth can be
computed as follows:

    A  = present_amount(A0, p, n)
    A0 = initial_amount(A, p, n)
    n  = days(A0, A, p)
    p  = annual_rate(A0, A, n)
"""
```

You can run the `pydoc` program to see a documentation of the new module, containing the doc string above and a list of the functions in the module: just write `pydoc interest` in a terminal window.

Now the reader is recommended to take a look at the actual file `interest.py` to see all elements of a good module file at once: doc strings, a set of functions, a test function, a function with the main program, a usage string, and a test block.

4.9.7 Using modules

Let us further demonstrate how to use the `interest.py` module in programs. For illustration purposes, we make a separate program file, say with name `doubling.py`, containing some computations:

```
from interest import days

# How many days does it take to double an amount when the
# interest rate is p=1,2,3,...14?
for p in range(1, 15):
    years = days(1, 2, p)/365.0
    print 'With p=%d%% it takes %.1f years to double the amount' %\
    (p, years)
```

What gets imported by various import statements? There are different ways to import functions in a module, and let us explore these in an interactive session. The function call `dir()` will list all names we have defined, including imported names of variables and functions. Calling `dir(m)` will print the names defined inside a module with name `m`. First we start an interactive shell and call `dir()`

```
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__']
```

These variables are always defined. Running the IPython shell will introduce several other standard variables too. Doing

```
>>> from interest import *
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__',
 'annual_rate', 'compute_missing_parameter', 'days',
 'initial_amount', 'ln', 'present_amount', 'sys',
 'test_all_functions']
```

shows that we get our four functions imported, along with `ln` and `sys`. The latter two are needed in the `interest` module, but not necessarily in our new program `doubling.py`.

The alternative `import interest` actually gives us access to more names in the module, namely also all variables and functions that start with an underscore:

```
>>> import interest
>>> dir(interest)
['__builtins__', '__doc__', '__file__', '__name__',
 '__package__', '_filename', '_usage', 'annual_rate',
 'compute_missing_parameter', 'days', 'initial_amount',
 'ln', 'present_amount', 'sys', 'test_all_functions']
```

It is a habit to use an underscore for all variables that are not to be included in a `from interest import *` statement. These variables can, however, be reached through `interest._filename` and `interest._usage` in the present example.

It would be best that a statement `from interest import *` just imported the four functions doing the computations of general interest in other programs. This can be archived by deleting all unwanted names (among those without an initial underscore) at the very end of the module:

```
del sys, ln, compute_missing_parameter, test_all_functions
```

Instead of deleting variables and using initial underscores in names, it is in general better to specify the special variable `__all__`, which is used by Python to select functions to be imported in `from interest import *` statements. Here we can define `__all__` to contain the four function of main interest:

```
__all__ = ['annual_rate', 'days', 'initial_amount', 'present_amount']
```

Now we get

```
>>> from interest import *
['__builtins__', '__doc__', '__name__', '__package__',
 'annual_rate', 'days', 'initial_amount', 'present_amount']
```

How to make Python find a module file. The `doubling.py` program works well as long as it is located in the same folder as the `interest.py` module. However, if we move `doubling.py` to another folder and run it, we get an error:

Terminal

```
doubling.py
Traceback (most recent call last):
  File "doubling.py", line 1, in <module>
    from interest import days
ImportError: No module named interest
```

Unless the module file resides in the same folder, we need to tell Python where to find our module. Python looks for modules in the folders contained in the list `sys.path`. A little program

```
import sys, pprint
pprint.pprint(sys.path)
```

prints out all these predefined module folders. You can now do one of two things:

1. Place the module file in one of the folders in `sys.path`.
2. Include the folder containing the module file in `sys.path`.

There are two ways of doing the latter task. Alternative 1 is to explicitly insert a new folder name in `sys.path` in the program that uses the module:

```
modulefolder = '../../pymodules'
sys.path.insert(0, modulefolder)
```

(In this sample path, the slashes are Unix specific. On Windows you must use backslashes and a raw string. A better solution is to express the path as `os.path.join(os.pardir, os.pardir, 'mymodules')`. This will work on all platforms.)

Python searches the folders in the sequence they appear in the `sys.path` list so by inserting the folder name as the first list element we ensure that our module is found quickly, and in case there are other modules with the same name in other folders in `sys.path`, the one in `modulefolder` gets imported.

Alternative 2 is to specify the folder name in the `PYTHONPATH` environment variable. All folder names listed in `PYTHONPATH` are automatically included in `sys.path` when a Python program starts. On Mac and Linux systems, environment variables like `PYTHONPATH` are set in the `.bashrc` file in the home folder, typically as

```
export PYTHONPATH=$HOME/software/lib/pymodules:$PYTHONPATH
```

if `$HOME/software/lib/pymodules` is the folder containing Python modules. On Windows, you launch *Computer - Properties - Advanced System Settings - Environment Variables*, click under *System Variable*, write in `PYTHONPATH` as variable name and the relevant folder(s) as value.

How to make Python run the module file. The description above concerns importing the module in a program located anywhere on the

system. If we want to run the module file as a program, anywhere on the system, the operating system searches the `PATH` environment variable for the program name `interst.py`. It is therefore necessary to update `PATH` with the folder where `interest.py` resides.

On Mac and Linux system this is done in `.bashrc` in the same way as for `PYTHONPATH`:

```
export PYTH=$HOME/software/lib/pymodules:$PATH
```

On Windows, launch the dialog for setting environment variables as described above and find the `PATH` variable. It already has much content, so you add your new folder value either at the beginning or end, using a semicolon to separate the new value from the existing ones.

4.9.8 Distributing modules

Modules are usually useful pieces of software that others can take advantage of. Even though our simple `interest` module is of less interest to the world, we can illustrate how such a module is most effectively distributed to other users. The standard in Python is to distribute the module file together with a program called `setup.py` such that any user can just do

Terminal

```
Terminal> sudo python setup.py install
```

to install the module in one of the directories in `sys.path` so that the module is immediately accessible anywhere, both for import in a Python program and for execution as a stand-alone program.

The `setup.py` file is in the case of one module file very short:

```
from distutils.core import setup
setup(name='interest',
      version='1.0',
      py_modules=['interest'],
      scripts=['interest.py'],
      )
```

The `scripts=` keyword argument can be dropped if the module is just to be imported and not run as a program as well. More module files can trivially be added to the list.

A user who runs `setup.py install` on an Ubuntu machine will see from the output that `interest.py` is copied to the system folders `/usr/local/lib/python2.7/dist-packages` and `/usr/local/bin`. The former folder is for module files, the latter for executable programs.

Remark

Distributing a single module file can be done as shown, but if you have two or more module files that belong together, you should definitely create a *package* [26].

4.9.9 Making software available on the Internet

Distributing software today means making it available on one of the major project hosting sites such as Googlecode, GitHub, or Bitbucket. You will develop and maintain the project files on your own computer(s), but frequently push the software out in the cloud such that others also get your updates. The mentioned sites have is very strong support for collaborative software development.

Since many already have a Gmail or Google account, we briefly describe how you can make your software available at Googlecode.

1. Go to `http://googlecode.com`.
2. Sign in with your Gmail/Google username and password.
3. Click on *Create a new project*.
4. Fill in project name, summary, and description.
5. Choose a *version control system*. Git is recommended.
6. Select a license for the software. Notice that on Googlecode a software project must be available to the whole world as open source code. Other sites (Bitbucket and GitHub) allows private projects.
7. Press *Create project*.
8. Click on *Source* and then on *Checkout*.
9. Go to some appropriate place in your home folder tree where you want to store this Googlecode project.
10. Copy and paste the command under *Option 1* and run it in a terminal window (this requires that Git is installed on your system).

You now have a folder with the same name as the project name. Copy `setup.py` and `interst.py` to the folder and move to the folder. It is good to also write a short `README` file explaining what the project is about. Alternatively, you can later extend the description on the Googlecode web page for the project. Run

Terminal

```
Terminal> git add .
Terminal> git commit -am 'First registration of project files'
Terminal> git push origin master
```

You are now prompted for the `googlecode.com` password, which you can click on right under the *Option 1* command on the web page.

The above `git` commands look cryptic, but these commands plus 2-3 more are the essence of how programmers today work on software projects, small or big. I strongly encourage you to learn more about version control systems and project hosting sites [12]. The tools are in nature like Dropbox and Google Drive, just much more powerful when you collaborate with others.

Your project files are now stored in the cloud at http://code.google.com/p/project-name. Anyone can get the software by the listed `git clone` command you used above, or by clicking on the `zip` link under *Source* and *Browse* to download a zip file.

Every time you update the project files, you need to register the update at the Googlecode project site by

Terminal

```
Terminal> git commit -am 'Description of the changes you made...'
Terminal> git push origin master
```

The files at Googlecode are now synchronized with your local ones.

There is a bit more to be said here to make you up and going with this style of professional work [12], but the information above gives you at least a glimpse of how to put your software project in the cloud and opening it up for others. The Googlecode address for the particular `interest` module described above is http://code.google.com/p/interest-primer.

4.10 Summary

4.10.1 Chapter topics

Question and answer input. Prompting the user and reading the answer back into a variable is done by

```
var = raw_input('Give value: ')
```

The `raw_input` function returns a string containing the characters that the user wrote on the keyboard before pressing the Return key. It is necessary to convert `var` to an appropriate object (`int` or `float`, for instance) if we want to perform mathematical operations with `var`. Sometimes

```
var = eval(raw_input('Give value: '))
```

is a flexible and easy way of transforming the string to the right type of object (integer, real number, list, tuple, and so on). This last statement will not work, however, for strings unless the text is surrounded by quotes when written on the keyboard. A general conversion function that turns any text without quotes into the right object is `scitools.misc.str2obj`:

```
from scitools.misc import str2obj
var = str2obj(raw_input('Give value: '))
```

Typing, for example, `3` makes `var` refer to an `int` object, `3.14` results in a `float` object, `[-1,1]` results in a `list`, `(1,3,5,7)` in a `tuple`, and `some text` in the string (`str`) object `'some text'` (run the program `str2obj_demo.py` to see this functionality demonstrated).

Getting command-line arguments. The `sys.argv[1:]` list contains all the command-line arguments given to a program (`sys.argv[0]` contains the program name). All elements in `sys.argv` are strings. A typical usage is

```
parameter1 = float(sys.argv[1])
parameter2 = int(sys.argv[2])
parameter3 = sys.argv[3]          # parameter3 can be string
```

Using option-value pairs. The `argparse` module is recommended for interpreting command-line arguments of the form `-option value`. A simple recipe with `argparse` reads

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('--p1', '--parameter_1', type=float,
                    default=0.0, help='1st parameter')
parser.add_argument('--p2', type=float,
                    default=0.0, help='2nd parameter')

args = parser.parse_args()
p1 = args.p1
p2 = args.p2
```

On the command line we can provide any or all of these options:

```
--parameter_1 --p1 --p2
```

where each option must be succeeded by a suitable value. However, `argparse` is very flexible can easily handle options without values or command-line arguments without any option specifications.

Generating code on the fly. Calling `eval(s)` turns a string `s`, containing a Python expression, into code as if the contents of the string were written directly into the program code. The result of the following `eval` call is a `float` object holding the number 21.1:

```
>>> x = 20
>>> r = eval('x + 1.1')
>>> r
21.1
>>> type(r)
<type 'float'>
```

The `exec` function takes a string with arbitrary Python code as argument and executes the code. For example, writing

```
exec("""
def f(x):
    return %s
""" % sys.argv[1])
```

is the same as if we had hardcoded the (for the programmer unknown) contents of `sys.argv[1]` into a function definition in the program.

Turning string formulas into Python functions. Given a mathematical formula as a string, `s`, we can turn this formula into a callable Python function `f(x)` by

```
from scitools.std import StringFunction

f = StringFunction(s)
```

The string formula can contain parameters and an independent variable with another name than `x`:

```
Q_formula = 'amplitude*sin(w*t-phaseshift)'
Q = StringFunction(Q_formula, independent_variable='t',
                   amplitude=1.5, w=pi, phaseshift=0)
values1 = [Q(i*0.1) for t in range(10)]
Q.set_parameters(phaseshift=pi/4, amplitude=1)
values2 = [Q(i*0.1) for t in range(10)]
```

Functions of several independent variables are also supported:

```
f = StringFunction('x+y**2+A', independent_variables=('x', 'y'),
                   A=0.2)
x = 1; y = 0.5
print f(x, y)
```

File operations. Reading from or writing to a file first requires that the file is opened, either for reading, writing, or appending:

```
infile  = open(filename, 'r')   # read
outfile = open(filename, 'w')   # write
outfile = open(filename, 'a')   # append
```

or using `with`:

```
with open(filename, 'r') as infile:   # read
with open(filename, 'w') as outfile:  # write
with open(filename, 'a') as outfile:  # append
```

There are four basic reading commands:

```
line    = infile.readline()   # read the next line
filestr = infile.read()       # read rest of file into string
lines   = infile.readlines()  # read rest of file into list
for line in infile:           # read rest of file line by line
```

File writing is usually about repeatedly using the command

```
outfile.write(s)
```

where `s` is a string. Contrary to `print s`, no newline is added to `s` in `outfile.write(s)`.

After reading or writing is finished, the file must be closed:

```
somefile.close()
```

However, closing the file is not necessary if we employ the `with` statement for reading or writing files:

```
with open(filename, 'w') as outfile:
    for var1, var2 in data:
        outfile.write('%5.2f %g\n' % (var1, var2))
# outfile is closed
```

Handling exceptions. Testing for potential errors is done with `try-except` blocks:

```
try:
    <statements>
except ExceptionType1:
    <provide a remedy for ExceptionType1 errors>
except ExceptionType2, ExceptionType3, ExceptionType4:
    <provide a remedy for three other types of errors>
except:
    <provide a remedy for any other errors>
...
```

The most common exception types are `NameError` for an undefined variable, `TypeError` for an illegal value in an operation, and `IndexError` for a list index out of bounds.

Raising exceptions. When some error is encountered in a program, the programmer can raise an exception:

```
if z < 0:
    raise ValueError('z=%s is negative - cannot do log(z)' % z)
r = log(z)
```

Modules. A module is created by putting a set of functions in a file. The filename (minus the required extension `.py`) is the name of the module. Other programs can import the module only if it resides in the same folder or in a folder contained in the `sys.path` list (see Section 4.9.7 for how to deal with this potential problem). Optionally, the module file can have a special `if` construct at the end, called test block, which tests the module or demonstrates its usage. The test block does not get executed when the module is imported in another program, only when the module file is run as a program.

Terminology. The important computer science topics and Python tools in this chapter are

- command line
- `sys.argv`
- `raw_input`
- `eval` and `exec`
- file reading and writing
- handling and raising exceptions
- module
- test block

4.10.2 Example: Bisection root finding

Problem. The summarizing example of this chapter concerns the implementation of the Bisection method for solving nonlinear equations of the form $f(x) = 0$ with respect to x. For example, the equation

$$x = 1 + \sin x$$

can be cast in the form $f(x) = 0$ if we move all terms to the left-hand side and define $f(x) = x - 1 - \sin x$. We say that x is a *root* of the equation $f(x) = 0$ if x is a solution of this equation. Nonlinear equations $f(x) = 0$ can have zero, one, several, or infinitely many roots.

Numerical methods for computing roots normally lead to approximate results only, i.e., $f(x)$ is not made exactly zero, but very close to zero. More precisely, an approximate root x fulfills $|f(x)| \leq \epsilon$, where ϵ is a small number. Methods for finding roots are of an iterative nature: we start with a rough approximation to a root and perform a repetitive set of steps that aim to improve the approximation. Our particular method for computing roots, the Bisection method, guarantees to find an approximate root, while other methods, such as the widely used Newton's method (see Section A.1.10), can fail to find roots.

The idea of the Bisection method is to start with an interval $[a, b]$ that contains a root of $f(x)$. The interval is halved at $m = (a + b)/2$, and if $f(x)$ changes sign in the left half interval $[a, m]$, one continues with that interval, otherwise one continues with the right half interval $[m, b]$. This procedure is repeated, say n times, and the root is then guaranteed to be inside an interval of length $2^{-n}(b - a)$. The task is to write a program that implements the Bisection method and verify the implementation.

Solution. To implement the Bisection method, we need to translate the description in the previous paragraph to a precise algorithm that can be almost directly translated to computer code. Since the halving of the interval is repeated many times, it is natural to do this inside a loop. We start with the interval $[a, b]$, and adjust a to m if the root must be in

the right half of the interval, or we adjust b to m if the root must be in the left half. In a language close to computer code we can express the algorithm precisely as follows:

```
for i = 0,1,2, ..., n:
    m = (a + b)/2
    if f(a)*f(m) <= 0:
        b = m  # root is in left half
    else:
        a = m  # root is in right half

# f(x) has a root in [a,b]
```

Figure 4.2 displays graphically the first four steps of this algorithm for solving the equation $\cos(\pi x) = 0$, starting with the interval $[0, 0.82]$. The graphs are automatically produced by the program `bisection_movie.py`, which was run as follows for this particular example:

Terminal

```
bisection_movie.py 'cos(pi*x)' 0 0.82
```

The first command-line argument is the formula for $f(x)$, the next is a, and the final is b.

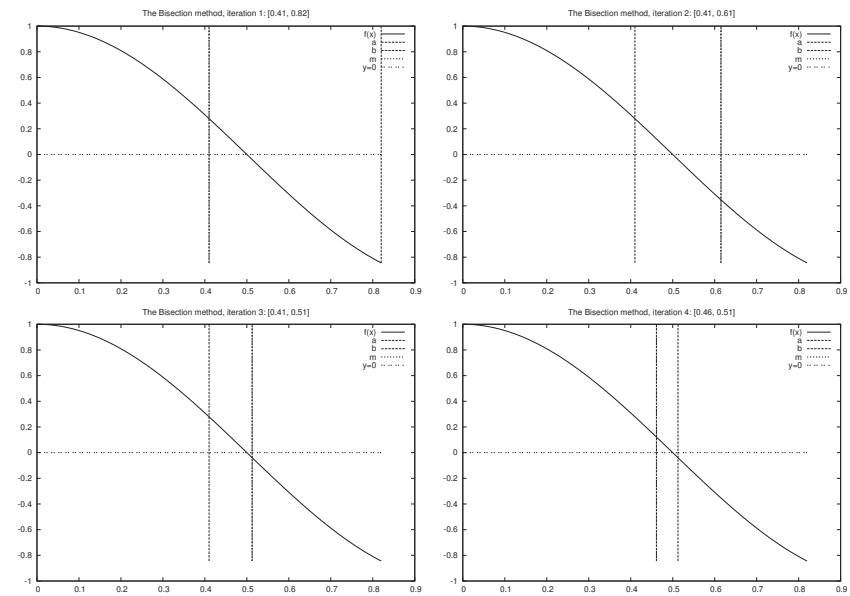

Fig. 4.2 Illustration of the first four iterations of the Bisection algorithm for solving $\cos(\pi x) = 0$. The vertical lines correspond to the current value of a and b.

In the algorithm listed above, we recompute $f(a)$ in each `if`-test, but this is not necessary if a has not changed since the last $f(a)$ computations. It is a good habit in numerical programming to avoid redundant work. On modern computers the Bisection algorithm normally runs so fast that we can afford to do more work than necessary. However, if $f(x)$

is not a simple formula, but computed by comprehensive calculations in a program, the evaluation of f might take minutes or even hours, and reducing the number of evaluations in the Bisection algorithm is then very important. We will therefore introduce extra variables in the algorithm above to save an $f(m)$ evaluation in each iteration in the `for` loop:

```
f_a = f(a)
for i = 0,1,2, ..., n:
    m = (a + b)/2
    f_m = f(m)
    if f_a*f_m <= 0:
        b = m   # root is in left half
    else:
        a = m   # root is in right half
        f_a = f_m

# f(x) has a root in [a,b]
```

To execute the algorithm above, we need to specify n. Say we want to be sure that the root lies in an interval of maximum extent ϵ. After n iterations the length of our current interval is $2^{-n}(b-a)$, if $[a,b]$ is the initial interval. The current interval is sufficiently small if

$$2^{-n}(b-a) = \epsilon,$$

which implies

$$n = -\frac{\ln \epsilon - \ln(b-a)}{\ln 2}\,. \tag{4.6}$$

Instead of calculating this n, we may simply stop the iterations when the length of the current interval is less than ϵ. The loop is then naturally implemented as a `while` loop testing on whether $b - a \leq \epsilon$. To make the algorithm more foolproof, we also insert a test to ensure that $f(x)$ really changes sign in the initial interval. This guarantees a root in $[a,b]$. (However, $f(a)f(b) < 0$ is not a necessary condition if there is an even number of roots in the initial interval.)

Our final version of the Bisection algorithm now becomes

```
f_a=f(a)
if f_a*f(b) > 0:
    # error: f does not change sign in [a,b]

i = 0
while b-a > epsilon:
    i = i + 1
    m = (a + b)/2
    f_m = f(m)
    if f_a*f_m <= 0:
        b = m  # root is in left half
    else:
        a = m  # root is in right half
        f_a = f_m

# if x is the real root, |x-m| < epsilon
```

This is the algorithm we aim to implement in a Python program.

A direct translation of the previous algorithm to a valid Python program is a matter of some minor edits:

```
eps = 1E-5
a, b = 0, 10

fa = f(a)
if fa*f(b) > 0:
    print 'f(x) does not change sign in [%g,%g].' % (a, b)
    sys.exit(1)

i = 0   # iteration counter
while b-a > eps:
    i += 1
    m = (a + b)/2.0
    fm = f(m)
    if fa*fm <= 0:
        b = m  # root is in left half of [a,b]
    else:
        a = m  # root is in right half of [a,b]
        fa = fm
    print 'Iteration %d: interval=[%g, %g]' % (i, a, b)

x = m          # this is the approximate root
print 'The root is', x, 'found in', i, 'iterations'
print 'f(%g)=%g' % (x, f(x))
```

This program is found in the file bisection_v1.py.

Verification. To verify the implementation in `bisection_v1.py` we choose a very simple $f(x)$ where we know the exact root. One suitable example is a linear function, $f(x) = 2x - 3$ such that $x = 3/2$ is the root of f. As can be seen from the source code above, we have inserted a `print` statement inside the `while` loop to control that the program really does the right things. Running the program yields the output

```
Iteration 1: interval=[0, 5]
Iteration 2: interval=[0, 2.5]
Iteration 3: interval=[1.25, 2.5]
Iteration 4: interval=[1.25, 1.875]
...
Iteration 19: interval=[1.5, 1.50002]
Iteration 20: interval=[1.5, 1.50001]
The root is 1.50000572205 found in 20 iterations
f(1.50001)=1.14441e-05
```

It seems that the implementation works. Further checks should include hand calculations for the first (say) three iterations and comparison of the results with the program.

Making a function. The previous implementation of the bisection algorithm is fine for many purposes. To solve a new problem $f(x) = 0$ it is just necessary to change the `f(x)` function in the program. However, if we encounter solving $f(x) = 0$ in another program in another context, we must put the bisection algorithm into that program in the right place. This is simple in practice, but it requires some careful work, and it is easy to make errors. The task of solving $f(x) = 0$ by the bisection algorithm is much simpler and safer if we have that algorithm available as a function

in a module. Then we can just import the function and call it. This requires a minimum of writing in later programs.

When you have a "flat" program as shown above, without basic steps in the program collected in functions, you should always consider dividing the code into functions. The reason is that parts of the program will be much easier to reuse in other programs. You save coding, and that is a good rule! A program with functions is also easier to understand, because statements are collected into logical, separate units, which is another good rule! In a mathematical context, functions are particularly important since they naturally split the code into general algorithms (like the bisection algorithm) and a problem-specific part (like a special choice of $f(x)$).

Shuffling statements in a program around to form a new and better designed version of the program is called *refactoring*. We shall now refactor the `bisection_v1.py` program by putting the statements in the bisection algorithm in a function `bisection`. This function naturally takes $f(x)$, a, b, and ϵ as parameters and returns the found root, perhaps together with the number of iterations required:

```
def bisection(f, a, b, eps):
    fa = f(a)
    if fa*f(b) > 0:
        return None, 0

    i = 0   # iteration counter
    while b-a > eps:
        i += 1
        m = (a + b)/2.0
        fm = f(m)
        if fa*fm <= 0:
            b = m  # root is in left half of [a,b]
        else:
            a = m  # root is in right half of [a,b]
            fa = fm
    return m, i
```

After this function we can have a test program:

```
def f(x):
    return 2*x - 3   # one root x=1.5

x, iter = bisection(f, a=0, b=10, eps=1E-5)
if x is None:
    print 'f(x) does not change sign in [%g,%g].' % (a, b)
else:
    print 'The root is', x, 'found in', iter, 'iterations'
    print 'f(%g)=%g' % (x, f(x))
```

The complete code is found in file `bisection_v2.py`.

Making a test function. Rather than having a main program as above for verifying the implementation, we should make a test function `test_bisection` as described in Section 4.9.4. To this end, we move the statements above inside a function, drop the output, but instead make a boolean variable `success` that is `True` if the test is passed and

`False` otherwise. Then we do `assert success, msg`, which will abort the program if the test fails. The `msg` variable is a string with more explanation of what went wrong the test fails. A test function with this structure is easy to integrate into the widely used testing frameworks nose and pytest, and there are no good reasons for not adopting this structure. The code checking that the root is within a distance ϵ to the exact root becomes

```
def test_bisection():
    def f(x):
        return 2*x - 3   # one root x=1.5

    x, iter = bisection(f, a=0, b=10, eps=1E-5)
    success = abs(x - 1.5) < 1E-5  # test within eps tolerance
    assert success, 'found x=%g != 1.5' % x
```

Making a module. A motivating factor for implementing the bisection algorithm as a function `bisection` was that we could import this function in other programs to solve $f(x) = 0$ equations. We therefore need to make a module file `bisection.py` such that we can do, e.g.,

```
from bisection import bisection
x, iter = bisection(lambda x: x**3 + 2*x -1, -10, 10, 1E-5)
```

A module file should not execute a main program, but just define functions, import modules, and define global variables. Any execution of a main program must take place in the test block, otherwise the `import` statement will start executing the main program, resulting in very disturbing statements for another program that wants to solve a different $f(x) = 0$ equation.

The `bisection_v2.py` file had a main program that was just a simple test for checking that the `bisection` algorithm works for a linear function. We took this main program and wrapped in a test function `test_bisection` above. To run the test, we make the call to this function from the test block:

```
if __name__ == '__main__':
    test_bisection()
```

This is all that is demanded to turn the file `bisection_v2.py` into a proper module file `bisection.py`.

Defining a user interface. It is nice to have our `bisection` module do more than just test itself: there should be a user interface such that we can solve real problems $f(x) = 0$, where $f(x)$, a, b, and ϵ are defined on the command line by the user. A dedicated function can read from the command line and return the data as Python object. For reading the function $f(x)$ we can either apply `eval` on the command-line argument, or use the more sophisticated `StringFunction` tool from Section 4.3.3. With `eval` we need to import functions from the `math` module in case the user

have such functions in the expression for $f(x)$. With `StringFunction` this is not necessary.

A `get_input()` for getting input from the command line can be implemented as

```
def get_input():
    """Get f, a, b, eps from the command line."""
    from scitools.std import StringFunction
    try:
        f = StringFunction(sys.argv[1])
        a = float(sys.argv[2])
        b = float(sys.argv[3])
        eps = float(sys.argv[4])
    except IndexError:
        print 'Usage %s: f a b eps' % sys.argv[0]
        sys.exit(1)
    return f, a, b, eps
```

To solve the corresponding $f(x) = 0$ problem, we simply add a branch in the `if` test in the test block:

```
if __name__ == '__main__':
    import sys
    if len(sys.argv) >= 2 and sys.argv[1] == 'test':
        test_bisection()
    else:
        f, a, b, eps = get_input()
        x, iter = bisection(f, a, b, eps)
        print 'Found root x=%g in %d iterations' % (x, iter)
```

Desired properties of a module

Our `bisection.py` code is a complete module file with the following generally desired features of Python modules:

- other programs can import the `bisection` function,
- the module can test itself (with a pytest/nose-compatible test function),
- the module file can be run as a program with a user interface where a general rooting finding problem can be specified in terms of a formula for $f(x)$ along with the parameters a, b, and ϵ.

Using the module. Suppose you want to solve $x/(x-1) = \sin x$ using the `bisection` module. What do you have to do? First, you must reformulate the equation as $f(x) = 0$, i.e., $x/(x-1) - \sin x = 0$, or maybe multiply by $x - 1$ to get $f(x) = x - (x-1)\sin x$.

It is required to identify an interval for the root. By evaluating $f(x)$ for some points x one can be trial and error locate an interval. A more convenient approach is to plot the function $f(x)$ and visually inspect

where a root is. Chapter 5 describes the techniques, but here we simply state the recipe. We start `ipython -pylab` and write

```
In [1]: x = linspace(-3, 3, 50)  # generate 50 coordinates in [-3,3]

In [2]: y = x - (x-1)*sin(x)

In [3]: plot(x, y)
```

Figure 4.3 shows $f(x)$ and we clearly see that, e.g., $[-2, 1]$ is an appropriate interval.

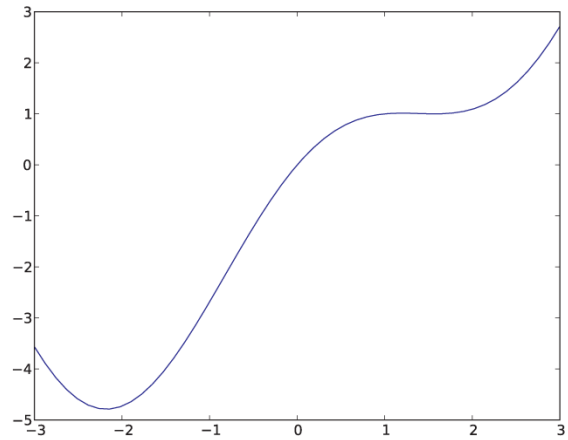

Fig. 4.3 Plot of $f(x) = x - \sin(x)$.

The next step is to run the Bisection algorithm. There are two possibilities:

- make a program where you code $f(x)$ and run the `bisection` function, or
- run the `bisection.py` program directly.

The latter approach is the simplest:

Terminal

```
bisection.py "x - (x-1)*sin(x)" -2 1 1E-5
Found root x=-1.90735e-06 in 19 iterations
```

The alternative approach is to make a program:

```
from bisection import bisection
from math import sin

def f(x):
    return x - (x-1)*sin(x)

x, iter = bisection(f, a=-2, b=1, eps=1E-5)
print x, iter
```

Potential problems with the software. Let us solve

- $x = \tanh x$ with start interval $[-10, 10]$ and $\epsilon = 10^{-6}$,
- $x^5 = \tanh(x^5)$ with start interval $[-10, 10]$ and $\epsilon = 10^{-6}$.

Both equations have one root $x = 0$.

Terminal

```
bisection.py "x-tanh(x)" -10 10
Found root x=-5.96046e-07 in 25 iterations

bisection.py "x**5-tanh(x**5)" -10 10
Found root x=-0.0266892 in 25 iterations
```

These results look strange. In both cases we halve the start interval $[-10, 10]$ 25 times, but in the second case we end up with a much less accurate root although the value of ϵ is the same. A closer inspection of what goes on in the bisection algorithm reveals that the inaccuracy is caused by round-off errors. As $a, b, m \to 0$, raising a small number to the fifth power in the expression for $f(x)$ yields a much smaller result. Subtracting a very small number $\tanh x^5$ from another very small number x^5 may result in a small number with wrong sign, and the sign of f is essential in the bisection algorithm. We encourage the reader to graphically inspect this behavior by running these two examples with the `bisection_plot.py` program using a smaller interval $[-1, 1]$ to better see what is going on. The command-line arguments for the `bisection_plot.py` program are `'x-tanh(x)' -1 1` and `'x**5-tanh(x**5)' -1 1`. The very flat area, in the latter case, where $f(x) \approx 0$ for $x \in [-1/2, 1/2]$ illustrates well that it is difficult to locate an exact root.

Distributing the bisection module to others. The Python standard for installing software is to run a `setup.py` program,

Terminal

```
Terminal> sudo python setup.py install
```

to install the system. The relevant `setup.py` for the `bisection` module arises from substituting the name `interest` by `bisection` in the `setup.py` file listed in Section 4.9.8. You can then distribute `bisection.py` and `setup.py` together.

4.11 Exercises

Exercise 4.1: Make an interactive program

Make a program that asks the user for a temperature in Fahrenheit degrees and reads the number; computes the corresponding temperature in Celsius degrees; and prints out the temperature in the Celsius scale. Filename: `f2c_qa.py`.

Exercise 4.2: Read a number from the command line

Modify the program from Exercise 4.1 such that the Fahrenheit temperature is read from the command line. Filename: `f2c_cml.py`.

Exercise 4.3: Read a number from a file

Modify the program from Exercise 4.1 such that the Fahrenheit temperature is read from a file with the following content:

```
Temperature data
----------------

Fahrenheit degrees: 67.2
```

Hint. Create a sample file manually. In the program, skip the first three lines, split the fourth line into words and grab the third word.
Filename: `f2c_file_read.py`.

Exercise 4.4: Read and write several numbers from and to file

This is a variant of Exercise 4.3 where we have several Fahrenheit degrees in a file and want to read all of them into a list and convert the numbers to Celsius degrees. Thereafter, we want to write out a file with two columns, the left with the Fahrenheit degrees and the right with the Celsius degrees.

An example on the input file format looks like

```
Temperature data
----------------

Fahrenheit degrees: 67.2
Fahrenheit degrees: 66.0
Fahrenheit degrees: 78.9
Fahrenheit degrees: 102.1
Fahrenheit degrees: 32.0
Fahrenheit degrees: 87.8
```

A sample file is `Fdeg.dat`[5]. Filename: `f2c_file_read_write.py`.

Exercise 4.5: Use exceptions to handle wrong input

Extend the program from Exercise 4.2 with a `try-except` block to handle the potential error that the Fahrenheit temperature is missing on the command line. Filename: `f2c_cml_exc.py`.

[5] `http://tinyurl.com/pwyasaa/input/Fdeg.dat`

Exercise 4.6: Read input from the keyboard

Make a program that asks for input from the user, applies `eval` to this input, and prints out the type of the resulting object and its value. Test the program by providing five types of input: an integer, a real number, a complex number, a list, and a tuple. Filename: `objects_qa.py`.

Exercise 4.7: Read input from the command line

a) Let a program store the result of applying the `eval` function to the first command-line argument. Print out the resulting object and its type.

b) Run the program with different input: an integer, a real number, a list, and a tuple.

Hint. On Unix systems you need to surround the tuple expressions in quotes on the command line to avoid error message from the Unix shell.

c) Try the string `"this is a string"` as a command-line argument. Why does this string cause problems and what is the remedy?
Filename: `objects_cml.py`.

Exercise 4.8: Try MSWord or LibreOffice to write a program

The purpose of this exercise is to tell you how hard it may be to write Python programs in the standard programs that most people use for writing text.

a) Type the following one-line program in either MSWord or LibreOffice:

```
print "Hello, World!"
```

Both Word and LibreOffice are so "smart" that they automatically edit "print" to "Print" since a sentence should always start with a capital. This is just an example that word processors are made for writing documents, not computer programs.

b) Save the program as a `.docx` (Word) or `.odt` (LibreOffice) file. Now try to run this file as a Python program. What kind of error message do you get? Can you explain why?

c) Save the program as a `.txt` file in Word or LibreOffice and run the file as a Python program. What happened now? Try to find out what the problem is.

Exercise 4.9: Prompt the user for input to a formula

Consider the simplest program for evaluating the formula $y(t) = v_0 t - \frac{1}{2}gt^2$:

```
v0 = 3; g = 9.81; t = 0.6
y = v0*t - 0.5*g*t**2
print y
```

Modify this code so that the program asks the user questions `t=?` and `v0=?`, and then gets `t` and `v0` from the user's input through the keyboard. Filename: `ball_qa.py`.

Exercise 4.10: Read parameters in a formula from the command line

Modify the program listed in Exercise 4.9 such that `v0` and `t` are read from the command line. Filename: `ball_cml.py`.

Exercise 4.11: Use exceptions to handle wrong input

The program from Exercise 4.10 reads input from the command line. Extend that program with exception handling such that missing command-line arguments are detected. In the `except IndexError` block, use the `raw_input` function to ask the user for missing input data. Filename: `ball_cml_qa.py`.

Exercise 4.12: Test validity of input data

Test if the `t` value read in the program from Exercise 4.10 lies between 0 and $2v_0/g$. If not, print a message and abort the execution. Filename: `ball_cml_tcheck.py`.

Exercise 4.13: Raise an exception in case of wrong input

Instead of printing an error message and aborting the program explicitly, raise a `ValueError` exception in the `if` test on legal `t` values in the program from Exercise 4.12. Notify the user about the legal interval for t in the exception message. Filename: `ball_cml_ValueError.py`.

Exercise 4.14: Evaluate a formula for data in a file

We consider the formula $y(t) = v_0 t - 0.5gt^2$ and want to evaluate y for a range of t values found in a file with format

```
v0: 3.00
t:
0.15592  0.28075   0.36807889 0.35 0.57681501876
0.21342619  0.0519085  0.042  0.27  0.50620017 0.528
0.2094294  0.1117  0.53012  0.3729850  0.39325246
0.21385894  0.3464815 0.57982969 0.10262264
0.29584013  0.17383923
```

More precisely, the first two lines are always present, while the next lines contain an arbitrary number of t values on each line, separated by one or more spaces.

a) Write a function that reads the input file and returns v_0 and a list with the t values.

b) Write a function that creates a file with two nicely formatted columns containing the t values to the left and the corresponding y values to the right. Let the t values appear in increasing order (note that the input file does not necessarily have the t values sorted).

c) Make a test function that generates an input file, calls the function for reading the file, and checks that the returned data objects are correct. Filename: `ball_file_read_write.py`.

Exercise 4.15: Compute the distance it takes to stop a car

A car driver, driving at velocity v_0, suddenly puts on the brake. What braking distance d is needed to stop the car? One can derive, using Newton's second law of motion or a corresponding energy equation, that

$$d = \frac{1}{2}\frac{v_0^2}{\mu g}\,. \tag{4.7}$$

Make a program for computing d in (4.7) when the initial car velocity v_0 and the friction coefficient μ are given on the command line. Run the program for two cases: $v_0 = 120$ and $v_0 = 50$ km/h, both with $\mu = 0.3$ (μ is dimensionless).

Hint. Remember to convert the velocity from km/h to m/s before inserting the value in the formula.
Filename: `stopping_length.py`.

Exercise 4.16: Look up calendar functionality

The purpose of this exercise is to make a program that takes a date, consisting of year (4 digits), month (2 digits), and day (1-31) on the command line and prints the corresponding name of the weekday (Monday, Tuesday, etc.). Python has a module `calendar`, which makes it easy to solve the exercise, but the task is to find out how to use this module. Filename: `weekday.py`.

Exercise 4.17: Use the StringFunction tool

Make the program `user_formula.py` from Section 4.3.2 shorter by using the convenient `StringFunction` tool from Section 4.3.3. Filename: `user_formula2.py`.

Exercise 4.18: Why we test for specific exception types

The simplest way of writing a `try-except` block is to test for *any* exception, for example,

```
try:
    C = float(sys.arg[1])
except:
    print 'C must be provided as command-line argument'
    sys.exit(1)
```

Write the above statements in a program and test the program. What is the problem?

The fact that a user can forget to supply a command-line argument when running the program was the original reason for using a `try` block. Find out what kind of exception that is relevant for this error and test for this specific exception and re-run the program. What is the problem now? Correct the program. Filename: `unnamed_exception.py`.

Exercise 4.19: Make a complete module

a) Make six conversion functions between temperatures in Celsius, Kelvin, and Fahrenheit: `C2F`, `F2C`, `C2K`, `K2C`, `F2K`, and `K2F`.

b) Collect these functions in a module `convert_temp`.

c) Import the module in an interactive Python shell and demonstrate some sample calls on temperature conversions.

d) Insert the session from c) in a triple quoted string at the top of the module file as a doc string for demonstrating the usage.

e) Write a function `test_conversion()` that verifies the implementation. Call this function from the test block if the first command-line argument is `verify`.

Hint. Check that `C2F(F2C(f))` is `f`, `K2C(C2K(c))` is `c`, and `K2F(F2K(f))` is `f` - with tolerance. Follow the conventions for test functions outlined in Sections 4.9.4 and 4.10.2 with a boolean variable that is `False` if a test failed, and `True` if all test are passed, and then an `assert` statement to abort the program when any test fails.

f) Add a user interface to the module such that the user can write a temperature as the first command-line argument and the corresponding temperature scale as the second command-line argument, and then get the temperature in the two other scales as output. For example, `21.3 C` on the command line results in the output `70.3 F 294.4 K`. Encapsulate the user interface in a function, which is called from the test block.
Filename: `convert_temp.py`.

Exercise 4.20: Make a module

Collect the `f` and `S` functions in the program from Exercise 3.15 in a separate file such that this file becomes a module. Put the statements making the table (i.e., the main program from Exercise 3.15) in a separate function `table(n_values, alpha_values, T)`. Make a test block in the module to read T and a series of n and α values from the command line and make a corresponding call to `table`. Filename: `sinesum2.py`.

Exercise 4.21: Read options and values from the command line

Let the input to the program in Exercise 4.20 be option-value pairs with the options `-n`, `-alpha`, and `-T`. Provide sensible default values in the module file.

Hint. Apply the `argparse` module to read the command-line arguments. Do not copy code from the `sinesum2` module, but make a new file for reading option-value pairs from the command and import the `table` function from the `sinesum2` module.
Filename: `sinesum3.py`.

Exercise 4.22: Check if mathematical identities hold

Because of round-off errors, it could happen that a mathematical rule like $(ab)^3 = a^3b^3$ does not hold exactly on a computer. The idea of testing this potential problem is to check such identities for a large number of random numbers. We can make random numbers using the `random` module in Python:

```
import random
a = random.uniform(A, B)
b = random.uniform(A, B)
```

Here, `a` and `b` will be random numbers, which are always larger than or equal to `A` and smaller than `B`.

a) Make a function `power3_identity(A=-100, B=100, n=1000)` that tests the identity `(a*b)**3 == a**3*b**3` a large number of times, `n`. Return the fraction of failures.

Hint. Inside the loop over `n`, draw random numbers `a` and `b` as described above and count the number of times the test is `True`.

b) We shall now parameterize the expressions to be tested. Make a function

```
equal(expr1, expr2, A=-100, B=100, n=500)
```

where `expr1` and `expr2` are strings containing the two mathematical expressions to be tested. More precisely, the function draws random numbers `a` and `b` between `A` and `B` and tests if `eval(expr1) == eval(expr2)`. Return the fraction of failures.

Test the function on the identities $(ab)^3 = a^3b^3$, $e^{a+b} = e^a e^b$, and $\ln a^b = b \ln a$.

Hint. Make the `equal` function robust enough to handle illegal a and b values in the mathematical expressions (e.g., $a \leq 0$ in $\ln a$).

c) We want to test the validity of the following set of identities on a computer:

- $a - b$ and $-(b - a)$
- a/b and $1/(b/a)$
- $(ab)^4$ and a^4b^4
- $(a + b)^2$ and $a^2 + 2ab + b^2$
- $(a + b)(a - b)$ and $a^2 - b^2$
- e^{a+b} and $e^a e^b$
- $\ln a^b$ and $b \ln a$
- $\ln ab$ and $\ln a + \ln b$
- ab and $e^{\ln a + \ln b}$
- $1/(1/a + 1/b)$ and $ab/(a + b)$
- $a(\sin^2 b + \cos^2 b)$ and a
- $\sinh(a + b)$ and $(e^a e^b - e^{-a}e^{-b})/2$
- $\tan(a + b)$ and $\sin(a + b)/\cos(a + b)$
- $\sin(a + b)$ and $\sin a \cos b + \sin b \cos a$

Store all the expressions in a list of 2-tuples, where each 2-tuple contains two mathematically equivalent expressions as strings, which can be sent to the `equal` function. Make a nicely formatted table with a pair of equivalent expressions at each line followed by the failure rate. Write this table to a file. Try out `A=1` and `B=2` as well as `A=1` and `B=100`. Does the failure rate seem to depend on the magnitude of the numbers a and b? Filename: `math_identities_failures.py`.

Exercise 4.23: Compute probabilities with the binomial distribution

Consider an uncertain event where there are two outcomes only, typically success or failure. Flipping a coin is an example: the outcome is uncertain and of two types, either head (can be considered as success) or tail (failure). Throwing a die can be another example, if (e.g.) getting a six is considered success and all other outcomes represent failure. Such experiments are called *Bernoulli trials*.

Let the probability of success be p and that of failure $1-p$. If we perform n experiments, where the outcome of each experiment does not depend on the outcome of previous experiments, the probability of getting success x times, and consequently failure $n-x$ times, is given by

$$B(x,n,p) = \frac{n!}{x!(n-x)!}p^x(1-p)^{n-x}\,. \tag{4.8}$$

This formula (4.8) is called the binomial distribution. The expression $x!$ is the factorial of x: $x! = x(x-1)(x-2)\cdots 1$ and `math.factorial` can do this computation.

a) Implement (4.8) in a function `binomial(x, n, p)`.

b) What is the probability of getting two heads when flipping a coin five times? This probability corresponds to $n = 5$ events, where the success of an event means getting head, which has probability $p = 1/2$, and we look for $x = 2$ successes.

c) What is the probability of getting four ones in a row when throwing a die? This probability corresponds to $n = 4$ events, success is getting one and has probability $p = 1/6$, and we look for $x = 4$ successful events.

d) Suppose cross country skiers typically experience one ski break in one out of 120 competitions. Hence, the probability of breaking a ski can be set to $p = 1/120$. What is the probability b that a skier will experience a ski break during five competitions in a world championship?

Hint. This question is a bit more demanding than the other two. We are looking for the probability of 1, 2, 3, 4 or 5 ski breaks, so it is simpler to ask for the probability c of *not* breaking a ski, and then compute $b = 1 - c$. Define *success* as breaking a ski. We then look for $x = 0$ successes out of $n = 5$ trials, with $p = 1/120$ for each trial. Compute b. Filename: `Bernoulli_trials.py`.

Exercise 4.24: Compute probabilities with the Poisson distribution

Suppose that over a period of t_m time units, a particular uncertain event happens (on average) νt_m times. The probability that there will be x such events in a time period t is approximately given by the formula

$$P(x,t,\nu) = \frac{(\nu t)^x}{x!} e^{-\nu t}\,. \tag{4.9}$$

This formula is known as the Poisson distribution. (It can be shown that (4.9) arises from (4.8) when the probability p of experiencing the event in a small time interval t/n is $p = \nu t/n$ and we let $n \to \infty$.) An important assumption is that all events are independent of each other and that the probability of experiencing an event does not change significantly over time. This is known as a *Poisson process* in probability theory.

a) Implement (4.9) in a function `Poisson(x, t, nu)`, and make a program that reads x, t, and ν from the command line and writes out the probability $P(x,t,\nu)$. Use this program to solve the problems below.

b) Suppose you are waiting for a taxi in a certain street at night. On average, 5 taxis pass this street every hour at this time of the night. What is the probability of not getting a taxi after having waited 30 minutes? Since we have 5 events in a time period of $t_m = 1$ hour, $\nu t_m = \nu = 5$. The sought probability is then $P(0, 1/2, 5)$. Compute this number. What is the probability of having to wait two hours for a taxi? If 8 people need two taxis, that is the probability that two taxis arrive in a period of 20 minutes?

c) In a certain location, 10 earthquakes have been recorded during the last 50 years. What is the probability of experiencing exactly three earthquakes over a period of 10 years in this area? What is the probability that a visitor for one week does not experience any earthquake? With 10 events over 50 years we have $\nu t_m = \nu \cdot 50$ years $= 10$ events, which implies $\nu = 1/5$ event per year. The answer to the first question of having $x = 3$ events in a period of $t = 10$ years is given directly by (4.9). The second question asks for $x = 0$ events in a time period of 1 week, i.e., $t = 1/52$ years, so the answer is $P(0, 1/52, 1/5)$.

d) Suppose that you count the number of misprints in the first versions of the reports you write and that this number shows an average of six misprints per page. What is the probability that a reader of a first draft of one of your reports reads six pages without hitting a misprint? Assuming that the Poisson distribution can be applied to this problem, we have "time" t_m as 1 page and $\nu \cdot 1 = 6$, i.e., $\nu = 6$ events (misprints) per page. The probability of no events in a "period" of six pages is $P(0, 6, 6)$.
Filename: `Poisson_processes.py`.