

Recall our first program for evaluating the formula (1.2) on page 18 in Chapter 1:

```
C = 21
F = (9/5)*C + 32
print F
```

In this program, `C` is input data in the sense that `C` must be known before the program can perform the calculation of `F`. The results produced by the program, here `F`, 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 (`C = 21`). 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 explored¹.

This chapter starts with describing three different ways of reading data into a program: (i) letting the user answer questions in a dialog in the terminal window (Chapter 3.1), (ii) letting the user provide input on the command line (Chapter 3.2), and (iii) letting the user write input data in a graphical interface (Chapter 3.4). A fourth method is to read data from a file, but this topic is left for Chapter 6.

Even if your program works perfectly, wrong input data from the user may cause the program to produce wrong answers or even crash. Checking that the input data are correct is important, and Chapter 3.3 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

¹ Programmers know that any modification of the source code has a danger of introducing errors, so it is a good rule to change as little as possible in a program that works.

modules is therefore a natural and wise thing to do. Chapter 3.5 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` folder.

3.1 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`. A simple example involving the temperature conversion program above will quickly show how to use this function.

3.1.1 Reading Keyboard Input

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
```

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, as described in Chapter 1.1.5 and Appendix E.1. 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 3.1, 3.4, and 3.6 to make sure you understand how `raw_input` behaves.

3.1.2 The Magic “eval” Function

Python has a function `eval`, which takes a string as argument and evaluates this string as a Python expression. This functionality can be used to turn input into running code on the fly. To realize what it means, we invoke an interactive session:

```
>>> 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 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'>
```

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.

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.

So, why is the `eval` function so useful? Recall the `raw_input` function, which always returns a string object, which we often must explicitly transform to a different type, e.g., an `int` or a `float`. Sometimes we want to avoid specifying one particular type. The `eval` function can then be of help: we feed the returned string from `raw_input` to `eval` and let the latter function 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):

```
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:

```
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'
```

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

```
formula = raw_input('Give a formula involving x: ')
x = eval(raw_input('Give x: '))
from math import *    # make all math functions available
result = eval(formula)
print '%s for x=%g yields %g' % (formula, x, result)
```

First, we ask the reader to provide a formula, e.g., `2*sin(x)+1`. The result is a string object referred to by the `formula` variable. Then, we ask for an `x` value, typically a real number resulting in a `float` object. The key statement involves `eval(formula)`, which in the present example evaluates the expression `2*sin(x)+1`. The `x` variable is defined, and the `sin` function is also defined because of the `import` statement. Let us try to run the program:

```
eval_formula.py
Give a formula involving x: 2*sin(x)+1
Give x: 3.14
2*sin(x)+1 for x=3.14 yields 1.00319
```

Another important application of `eval` occurs in Chapter 3.2.1.

3.1.3 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, and that we want to turn this formula into a callable Python function. That is, writing `sin(x)*cos(3*x) + x**2` as the formula, we would like to get a function

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

This is easy with `exec`:

```
formula = raw_input('Write a formula involving x: ')
code = """
def f(x):
    return %s
""" % formula
exec(code)
```

If we respond with the text `sin(x)*cos(3*x) + x**2` to the question, `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 try out such code generation on the fly. We add a `while` loop to the previous code snippet defining `f(x)` such that we can provide `x` values and get `f(x)` evaluated:

```
x = 0
while x is not None:
    x = eval(raw_input('Give x (None to quit): '))
    if x is not None:
        print 'f(%g)=%g' % (x, f(x))
```

As long as we provide numbers as input for `x`, we evaluate the `f(x)` function, but when we provide the text `None`, `x` becomes a `None` object and the test in the `while` loop fails, i.e., the loop terminates. The complete program is found in the file `user_formula.py`. Here is a sample run:

Terminal

```
user_formula.py
Write a formula involving x: x**4 + x
Give x (None to quit): 1
```

```
f(1)=2
Give x (None to quit): 4
f(4)=260
Give x (None to quit): 2
f(2)=18
Give x (None to quit): None
```

3.1.4 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 “remembers” all the parameters `A`, `a`, `omega`, and `x`, and their values. You can use `pydoc` (see page 98) 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 functions².

3.2 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.

3.2.1 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. That is, we write the program name, here `c2f_cml_v1.py`³, followed the Celsius temperature:

```
c2f_cml_v1.py 21
69.8
```

Terminal

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. Here there is only one argument and it is stored with index 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 write⁴

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

² This property is quite remarkable in computer science – a string formula will in most other languages be much slower than if the formula were hardcoded inside a plain function.

³ The `cml` part of the name is an abbreviation for “command line”, and `v1` denotes “version 1”, as usual.

⁴ We could write `9` instead of `9.0`, in the formula for `F`, since `C` is guaranteed to be `float`, but it is safer to write `9.0`. One could think of modifying the conversion of the command-line argument to `eval(sys.argv[1])`, and in that case `C` can easily be an `int`.

As another example, consider the `ball_variables.py` program from Chapter 1.1.7. Instead of hardcoding the values of `v0` and `t` in the program we can read the two values from the command line:

```
ball_variables2.py 0.6 5
1.2342
```

Terminal

The two command-line arguments are now available as `sys.argv[1]` and `sys.argv[2]`. The complete `ball_variables2.py` program thus looks as

```
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
```

Our final example here concerns a program that can add two input objects (file `add_cml.py`, corresponding to `add_input.py` from Chapter 3.1.1):

```
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)
```

A key issue here is that we apply `eval` to the command-line arguments and thereby convert the strings into appropriate objects. Here is an example on execution:

```
add_cml.py 2 3.1
<type 'int'> + <type 'float'> becomes <type 'float'>
with value 5.1
```

Terminal

3.2.2 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

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

Terminal

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 (note the trailing comma which prevents the usual newline, cf. page 91). 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, i.e., the text that originally appeared on the command line. Chapter 6.3.1 contains more information on `join` and many other very useful string operations.

3.2.3 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` shall 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 (the most recently modified files appear first). 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.

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

```
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

```
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.

3.2.4 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. 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 (hopefully well-known) physics formula for the location $s(t)$ of an object at time t , if the object started at $s = s_0$ at $t = 0$ with a velocity v_0 , and thereafter was subject to a constant acceleration a :

$$s(t) = s_0 + v_0 t + \frac{1}{2} a t^2. \quad (3.1)$$

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:

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

The sequence of `-option value` pairs is arbitrary.

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
```

The standard Python module `getopt` supports reading `-option value` pairs on the command line. The recipe for using `getopt` goes as follows in the present example:

```
# set default values:
s0 = v0 = 0; a = t = 1
import getopt, sys
options, args = getopt.getopt(sys.argv[1:], '',
                             ['t=', 's0=', 'v0=', 'a='])
```

Note that we specify the option names without the leading double hyphen. The trailing `=` character indicates that the option is supposed to be followed by a value (without `=` only the option and not its value can be specified on the command line – this is basically suitable for boolean variables only).

The returned options object is a list of (option, value) 2-tuples containing the `-option value` pairs found on the command line. For example, the options list may be

```
[('--v0', 1.5), ('--t', 0.1), ('--a', 3)]
```

In this case, the user specified v_0 , t , and a , but not s_0 on the command line. The `args` object returned from `getopt.getopt` is a list of all the remaining command line arguments, i.e., the arguments that are not `-option value` pairs. The `args` variable has no use in our current example.

The typical way of processing the options list involves testing on the different option names:

```
for option, value in options:
    if option == '--t':
        t = float(value)
    elif option == '--a':
        a = float(value)
    elif option == '--v0':
        v0 = float(value)
    elif option == '--s0':
        s0 = float(value)
```

Sometimes a more descriptive options, say `--initial_velocity`, is offered in addition to the short form `-v0`. Similarly, `--initial_position` can be offered as an alternative to `-s0`. We may add as many alternative options as we like:

```
options, args = getopt.getopt(sys.argv[1:], '',
                             ['v0=', 'initial_velocity=', 't=', 'time=',
                              's0=', 'initial_position=', 'a=', 'acceleration='])
```

```

for option, value in options:
    if option in ('--t', '--time'):
        t = float(value)
    elif option in ('--a', '--acceleration'):
        a = float(value)
    elif option in ('--v0', '--initial_velocity'):
        v0 = float(value)
    elif option in ('--s0', '--initial_position'):
        s0 = float(value)

```

At this point in the program we have all input data, either by their default values or by user-given command-line arguments, and we can finalize the program by computing the formula (3.1) and printing out the result:

```

s = s0 + v0*t + 0.5*a*t**2
print """
An object, starting at s=%g at t=0 with initial
velocity %s m/s, and subject to a constant
acceleration %g m/s**2, is found at the
location s=%g m after %s seconds.
""" % (s0, v0, a, s, t)

```

A complete program using the `getopt` module as explained above is found in the file `location.py` in the `input` folder.

3.3 Handling Errors

Suppose we forget to provide a command-line argument to the `c2f_cml_v1.py` program from Chapter 3.2.1:

Terminal

```

c2f_cml_v1.py
Traceback (most recent call last):
  File "c2f_cml_v1.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_cm1_v1.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.

3.3.1 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_cm1_v3.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 directly⁵ to the `except` block. 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_v3.py
You failed to provide Celsius degrees as input on the command line!

c2f_cml_v3.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_v3.py 21C
You failed to provide Celsius degrees as input on the command line!
```

⁵ This implies that `float` is not called, and `C` is not initialized with a value.

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_v4.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_v3.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.

3.3.2 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.

Example. In the program `c2f_cml_v4.py` from page 135 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.,

```

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, 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), e:
    print e          # exception message
    sys.exit(1)      # terminate execution

```

After the try-except block above, we can continue with computing $F = 9 \cdot C / 5 + 32$ 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.

3.4 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 3.1. 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. 3.1 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 3.1 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 [2, 3, 5, 7].

3.5 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.

You learned already in Chapter 1 how to import functions from Python modules. Now you will learn how to make your own 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.

3.5.1 Example: Compound Interest Formulas

The classical formula for the growth of money in a bank reads⁶

$$A = A_0 \left(1 + \frac{p}{360 \cdot 100}\right)^n, \quad (3.2)$$

where A_0 is the initial amount of money, and A is the present amount after n days with p percent annual interest rate. Equation (3.2) involves four parameters: A , A_0 , p , and n . We may solve for any of these, given the other three:

$$A_0 = A \left(1 + \frac{p}{360 \cdot 100}\right)^{-n}, \quad (3.3)$$

$$n = \frac{\ln \frac{A}{A_0}}{\ln \left(1 + \frac{p}{360 \cdot 100}\right)}, \quad (3.4)$$

$$p = 360 \cdot 100 \left(\left(\frac{A}{A_0} \right)^{1/n} - 1 \right) \quad (3.5)$$

Suppose we have implemented (3.2)–(3.5) 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,

⁶ The formula applies the so-called Actual/360 convention where the rate per day is computed as $p/360$, while n counts the actual number of days the money is in the bank. See “Day count convention” in Wikipedia for detailed information and page 238 for a Python module for computing the number of days between two dates.


```
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.

3.5.2 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.

Test Block. It is recommended to only have functions and not any statements outside functions in a module⁷. However, Python allows a special construction to let the file act both as a module with function definitions only *and* as an ordinary program that we can run, i.e., with statements that apply the functions and possibly write output. This two-fold “magic” consists of putting the application part 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

⁷ The module file is executed from top to bottom during the import. With function definitions only in the module file, there will be no calculations or output from the import, just definitions of functions. This is the desirable behavior.

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.

Often, when modules are created from an ordinary program, the original main program is used as test block. The new module file then works as the old program, but with the new possibility of being imported in other programs. Let us write a little main program for testing the `interest` module. The idea is that we assign compatible values to the four parameters and check that given three of them, the functions calculate the remaining parameter in the correct way:

```
if __name__ == '__main__':
    A = 2.2133983053266699
    A0 = 2.0
    p = 5
    n = 730
    print 'A=%g (%g)\nA0=%g (%.1f)\nn=%d (%d)\np=%g (%.1f)' % \
        (present_amount(A0, p, n), A,
         initial_amount(A, p, n), A0,
         days(A0, A, p), n,
         annual_rate(A0, A, n), p)
```

Running the module file as a program is now possible:

Terminal

```
interest.py
A=2.2134 (2.2134)
A0=2 (2.0)
n=730 (730)
p=5 (5.0)
```

The computed values appear after the equal sign, with correct values in parenthesis. We see that the program works well.

To test that the `interest.py` 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 therefore demonstrated that the file `interest.py` works both as a program and as a module.

Flexible Test Blocks. It is a good programming practice to let the test block do one or more of three things: (i) provide information on how the module or program is used, (ii) test if the module functions work properly, and (iii) 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 might be better to collect the statements in separate functions, which then are called from the test block. A convention is to let these test or documentation functions have names starting with an underscore, because such

names are not imported in other programs when doing a `from module import *` (normally we do not want to import test or documentation functions). In our example we may collect the verification statements above in a separate function and name this function `_verify` (observe the leading underscore). We also write the code a bit more explicit to better demonstrate how the module functions can be used:

```
def _verify():
    # 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)
    print 'A=%g (%g)\nA0=%g (%.1f)\nn=%d (%d)\np=%g (%.1f)' % \
        (A_computed, A, A0_computed, A0,
         n_computed, n, p_computed, p)
```

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

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

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

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

Terminal

should 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 easy⁸:

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

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
```

⁸ The `join` function on page 295 in Chapter 6.3.1, see also page 129, is more elegant and avoids the loop.

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, 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 assign value to a parameter with wrong name or forget a parameter. In those cases we call one of our four functions with uninitialized arguments. Therefore, we should embed the code above in a `try-except` block. An uninitialized variable will lead to a `NameError`, 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, 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, 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 “verify”, 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] == 'verify':
        _verify()
    else:
        init_code = ''
        for statement in sys.argv[1:]:
            init_code += statement + '\n'
        _compute_missing_parameter(init_code)
```

Note leading underscores in variable names that are to be used locally in the `interest.py` file only.

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` in `src/input` to see all elements of a good module file at once: doc string, set of functions, verification function, “main program function”, usage string, and test block.

3.5.3 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 `test.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)
```

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()
[ ..., 'annual_rate', 'days', 'initial_amount',
'present_amount', 'ln', 'sys']
```

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 `test.py`. Observe that none of the names with a leading underscore are imported. This demonstrates the importance of using a leading underscore in names for local variables and functions in a module: Names local to a module will then not pollute other programs or interactive sessions when a “star import” (`from module import *`) is performed.

Next we do

```
>>> import interest
>>> dir(interest)
['__builtins__', '__doc__', '__file__', '__name__', '__package__',
'_compute_missing_parameter', '_usage', '_verify',
'annual_rate', 'days', 'filename', 'initial_amount',
'ln', 'present_amount', 'sys']
```

All variables and functions defined or imported in the `interest.py` file are now visible, and we can access also functions and variables beginning with an underscore as long as we have the `interest.` prefix:

```
>>> interest._verify()
A=2.2134 (2.2134)
A0=2 (2.0)
n=730 (730)
p=5 (5.0)
>>> interest._filename
```

The `test.py` program works well as long as it is located in the same folder as the `interest.py` module. However, if we move `test.py` to another folder and run it, we get an error:

Terminal

```
test.py
Traceback (most recent call last):
  File "tmp.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:

- 2a. You can explicitly insert a new folder name in `sys.path` in the program that uses the module⁹:

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

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.

- 2b. Your module folders can be permanently specified in the `PYTHONPATH` environment variable¹⁰. All folder names listed in `PYTHONPATH` are automatically included in `sys.path` when a Python program starts.

⁹ In this sample path, the slashes are Unix specific. On Windows you must use backward slashes 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.

¹⁰ This makes sense only if you know what environment variables are, and we do not intend to explain that at the present stage.

3.6 Summary

3.6.1 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. Python has two modules, `getopt` and `optparse`, for interpreting command-line arguments of the form `-option value`. A simple recipe with `getopt` reads

```
import getopt, sys
try:
    options, args = getopt.getopt(sys.argv[1:], '',
        ['parameter1=', 'parameter2=', 'parameter3=',
         'p1=', 'p2=', 'p3=']) # shorter forms
```



```

except getopt.GetoptError, e:
    print 'Error in command-line option:\n', e
    sys.exit(1)

# set default values:
parameter1 = ...
parameter2 = ...
parameter3 = ...

from scitools.misc import str2obj
for option, value in options:
    if option in ('--parameter1', '--p1'):
        parameter1 = eval(value)      # if not string
    elif option in ('--parameter2', '--p2'):
        parameter2 = value            # if string
    elif option in ('--parameter3', '--p3'):
        parameter3 = str2obj(value)   # any object

```

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

```
--parameter1 --p1 --parameter2 --p2 --parameter3 --p3
```

Each option must be succeeded by a suitable value.

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.StringFunction import StringFunction
# or
from scitools.std import *

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)
```

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 Chapter 3.5.3 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.

3.6.2 Summarizing 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 to 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, many, 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 Chapter 5.1.9), 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  $i = 0, 1, 2, \dots, n$ 
   $m = (a + b)/2$ 
  if  $f(a)f(m) \leq 0$  then
     $b = m$  (root is in left half)
  else
     $a = m$  (root is in right half)
  end if
end for
 $f(x)$  has a root in  $[a, b]$ 

```

Figure 3.2 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 .

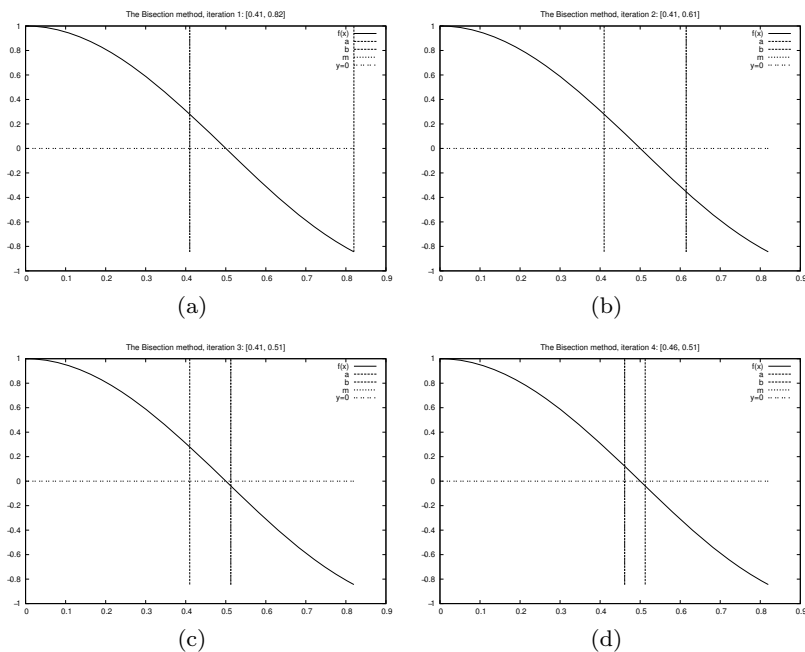


Fig. 3.2 Illustration of the first four iterations of the Bisection algorithm for solving $\cos(\pi x) = 0$. The vertical lines correspond to the current value of a and b .

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  $i = 0, 1, 2, \dots, n$ 
   $m = (a + b)/2$ 
   $f_m = f(m)$ 
  if  $f_a f_m \leq 0$  then
     $b = m$  (root is in left half)
  else
     $a = m$  (root is in right half)
     $f_a = f_m$ 
  end if
end for
 $f(x)$  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

$$n = -\frac{\ln \epsilon - \ln(b - a)}{\ln 2}. \quad (3.6)$$

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 interval¹¹.

Our final version of the Bisection algorithm now becomes

¹¹ 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.

```

 $f_a = f(a)$ 
if  $f_a f(b) > 0$  then
    error:  $f$  does not change sign in  $[a, b]$ 
end if
 $i = 0$  (iteration counter)
while  $b - a > \epsilon$ :
     $i \leftarrow i + 1$ 
     $m = (a + b)/2$ 
     $f_m = f(m)$ 
    if  $f_a f_m \leq 0$  then
         $b = m$  (root is in left half)
    else
         $a = m$  (root is in right half)
         $f_a = f_m$ 
    end if
end while
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 Python program should be quite a simple process:

```

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

eps = 1E-5
a, b = 0, 10
x, iter = bisection(f, a, b, eps)
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 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. However, if we do an import

```
from bisection_v2 import bisection
```

the import statement will run the main program in `bisection_v2.py`. We do not want to solve a particular $f(x) = 0$ example when we do an import of the `bisection` function! Therefore, we must put the main program in a test block (see Chapter 3.5.2). Even better is to collect the statements in the test program in a function and just call this function from the test block:

```

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

    eps = 1E-5
    a, b = 0, 10
    x, iter = bisection(f, a, b, eps)
    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))

if __name__ == '__main__':
    _test()

```

The complete module with the `bisection` function, the `_test` function, and the test block is found in the file `bisection.py`.

Using the Module. Suppose you want to solve $x = \sin x$ using the `bisection` module. What do you have to do? First, you reformulate

the equation as $f(x) = 0$, i.e., $x - \sin x = 0$ so that you identify $f(x) = x - \sin x$. Second, you make a file, say `x_eq_sinx.py`, where you import the `bisection` function, define the $f(x)$ function, and call `bisection`:

```
from bisection import bisection
from math import sin

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

root, iter = bisection(f, -2, 2, 1E-6)
print root
```

A Flexible Program for Solving $f(x) = 0$. The previous program hard-codes the input data $f(x)$, a , b , and ϵ to the bisection method for a specific equation. As we have pointed out in this chapter, a better solution is to let the user provide input data while the program is running. This approach avoids editing the program when a new equation needs to be solved (and as you remember, any change in a program has the danger of introducing new errors). We therefore set out to create a program that reads $f(x)$, a , b , and ϵ from the command-line. The expression for $f(x)$ is given as a text and turned into a Python function with aid of the `StringFunction` object from Chapter 3.1.4. The other parameters – a , b , and ϵ – can be read directly from the command line, but it can be handy to allow the user not to specify ϵ and provide a default value in the program instead.

The ideas above can be realized as follows in a new, general program for solving $f(x) = 0$ equations. The program is called `bisection_solver.py`:

```
import sys
usage = '%s f-formula a b [epsilon]' % sys.argv[0]
try:
    f_formula = sys.argv[1]
    a = float(sys.argv[2])
    b = float(sys.argv[3])
except IndexError:
    print usage; sys.exit(1)

try: # is epsilon given on the command-line?
    epsilon = float(sys.argv[4])
except IndexError:
    epsilon = 1E-6 # default value

from scitools.StringFunction import StringFunction
from math import * # might be needed for f_formula
f = StringFunction(f_formula)
from bisection import bisection

root, iter = bisection(f, a, b, epsilon)
if root == None:
    print 'The interval [%g, %g] does not contain a root' % (a, b)
    sys.exit(1)
print 'Found root %g\nof %s = 0 in [%g, %g] in %d iterations' % \
    (root, f_formula, a, b, iter)
```

Let us solve

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

Both equations have one root $x = 0$.

Terminal

```

bisection_solver.py "x-tanh(x)" -10 10
Found root -5.96046e-07
of x-tanh(x) = 0 in [-10, 10] in 25 iterations

bisection_solver.py "x**5-tanh(x**5)" -10 10
Found root -0.0266892
of x**5-tanh(x**5) = 0 in [-10, 10] 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 $a, b, m \rightarrow 0$, 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.

3.7 Exercises

Exercise 3.1. *Make an interactive program.*

Make a program that (i) asks the user for a temperature in Fahrenheit and reads the number; (ii) computes the corresponding temperature in Celsius degrees; and (iii) prints out the temperature in the Celsius scale. Name of program file: `f2c_qa.py`. ◇

Exercise 3.2. *Read from the command line in Exer. 3.1.*

Modify the program from Exercise 3.1 such that the Fahrenheit temperature is read from the command line. Name of program file: `f2c_cml.py`. ◇

Exercise 3.3. *Use exceptions in Exer. 3.2.*

Extend the program from Exercise 3.2 with a `try-except` block to handle the potential error that the Fahrenheit temperature is missing on the command line. Name of program file: `f2c_cml.py`. ◇

Exercise 3.4. *Read input from the keyboard.*

Make a program that asks the user for an integer, a real number, a list, a tuple, and a string. Use `eval` to convert the input string to a list or tuple. Name of program file: `objects_qa1.py`. ◇

Exercise 3.5. *Read input from the command line.*

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 (use `type` from Chapter 1.5.2). Run the program with different input: an integer, a real number, a list, and a tuple. Then try the string `"this is a string"` as a command-line argument. Why does this string cause problems and what is the remedy? Name of program file: `objects_cml.py`. ◇

Exercise 3.6. *Prompt the user for input to the formula (1.1).*

Consider the simplest program for evaluating (1.1):

```
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. Name of program file: `ball_qa.py`. ◇

Exercise 3.7. *Read command line input for the formula (1.1).*

Modify the program listed in Exercise 3.6 such that `v0` and `t` are read from the command line. Name of program file: `ball_cml.py`. ◇

Exercise 3.8. *Make the program from Exer. 3.7 safer.*

The program from Exercise 3.7 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. Name of program file: `ball_cml_qa.py`. ◇

Exercise 3.9. *Test more in the program from Exer. 3.7.*

Test if the `t` value read in the program from Exercise 3.7 lies between 0 and $\frac{2v_0}{g}$. If not, print a message and abort execution. Name of program file: `ball_cml_errorcheck.py`. ◇

Exercise 3.10. *Raise an exception in Exer. 3.9.*

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 3.9. The exception message should contain the legal interval for t . Name of program file: `ball_cm1_ValueError.py`.

◇

Exercise 3.11. *Look up calendar functionality.*

The purpose of this exercise is to make a program which 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 you must look up in the Python Library Reference (see Chapter 2.4.3), for calculating the weekday of a date. Name of program file: `weekday.py`.

◇

Exercise 3.12. *Use the `StringFunction` tool.*

Make the program `user_formula.py` from Chapter 3.1.3 shorter by using the convenient `StringFunction` tool from Chapter 3.1.4. Name of program file: `user_formula2.py`.

◇

Exercise 3.13. *Extend a program from Ch. 3.2.1.*

How can you modify the `add_cm1.py` program from the end of Chapter 3.1.2 such that it accepts input like `sqrt(2)` and `sin(1.2)`? In this case the output should be

```
<type 'float'> + <type 'float'> becomes <type 'float'>
with value 2.34625264834
```

(Hint: Mathematical functions, such as `sqrt` and `sin`, must be defined in the program before using `eval`. Furthermore, Unix (`bash`) does not like the parentheses on the command line so you need to put quotes around the command-line arguments.) Name of program file: `add2.py`.

◇

Exercise 3.14. *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.argv[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. Name of program file: `cm1_exception.py`. ◇

Exercise 3.15. *Make a simple module.*

Make six conversion functions between temperatures in Celsius, Kelvin, and Fahrenheit: C2F, F2C, C2K, K2C, F2K, and K2F. Collect these functions in a module `convert_temp`. Make some sample calls to these functions from an interactive Python shell. Name of program file: `convert_temp.py`. ◇

Exercise 3.16. *Make a useful main program for Exer. 3.15.*

Extend the module made in Exercise 3.15 with a main program in the test block. This main program should read the first command-line argument as a numerical value of a temperature and the second argument as a temperature scale: C, K, or F. Write out the temperature in the other two scales. For example, if 21.3 C is given on the command line, the output should be 70.34 F 294.45 K. Name of program file: `convert_temp2.py`. ◇

Exercise 3.17. *Make a module in Exer. 2.39.*

Collect the functions in the program from Exercise 2.39 in a separate file such that this file becomes a module. Put the statements making the table (i.e., the main program) in a separate function `table(t, T, n_values)`, and call this function only if the module file is run as a program (i.e., include a test block, see Chapter 3.5.2). Name of program file: `compute_sum_S_module.py`. ◇

Exercise 3.18. *Extend the module from Exer. 3.17.*

Extend the program from Exercise 3.17 such that t , T , and n are read from the command line. The extended program should import the `table` function from the module `compute_sum_S_module` and not copy any code from the module file or the program file from Exercise 2.39. Name of program file: `compute_sum_S_cml.py`. ◇

Exercise 3.19. *Use options and values in Exer. 3.18.*

Let the input to the program in Exercise 3.18 be option-value pairs of the type `-t`, `-T`, and `-n`, with sensible default values for these quantities set in the program. Apply the `getopt` module to read the command-line arguments. Name of program file: `compute_sum_S_cml_getopt.py`. ◇

Exercise 3.20. *Use `optparse` in the program from Ch. 3.2.4.*

Python has a module `optparse`, which is an alternative to `getopt` for reading `-option` value pairs. Read about `optparse` in the official Python documentation, either the Python Library Reference or the Global Module Index. Figure out how to apply `optparse` to the `location.py` program from Chapter 3.2.4 and modify the program to make use of `optparse` instead of `getopt`. Name of program file: `location_optparse.py`. ◇

Exercise 3.21. *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,

from basic physics, that

$$d = \frac{1}{2} \frac{v_0^2}{\mu g}. \quad (3.7)$$

Make a program for computing d in (3.7) 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). (Remember to convert the velocity from km/h to m/s before inserting the value in the formula!) Name of program file: `stopping_length.py`. \diamond

Exercise 3.22. *Check if mathematical rules hold on a computer.*

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 this exercise is to check such rules 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`.

Make a program that reads the number of tests to be performed from the command line. Set `A` and `B` to fixed values (say -100 and 100). Perform the test in a loop. Inside the loop, draw random numbers `a` and `b` and test if the two mathematical expressions `(a*b)**3` and `a**3*b**3` are equivalent. Count the number of failures of equivalence and write out the percentage of failures at the end of the program.

Duplicate the code segment outlined above to also compare the expressions `a/b` and `1/(b/a)`. Name of program file: `math_rules_failures.py`. \diamond

Exercise 3.23. *Improve input to the program in Exer. 3.22.*

The purpose of this exercise is to extend the program from Exercise 3.22 to handle a large number of mathematical rules. Make a function `equal(expr1, expr2, A, B, n=500)` which tests if the mathematical expressions `expr1` and `expr2`, given as strings and involving numbers `a` and `b`, are exactly equal (`eval(expr1) == eval(expr2)`) for `n` random choices of numbers `a` and `b` in the interval between `A` and `B`. Return the percentage of failures. Make a module with the `equal` function and a test block which feeds the `equal` function with arguments read from the command line. Run the module file as a program to test the two rules from Exercise 3.22. Also test the rules $e^{a+b} = e^a e^b$ and $\ln a^b = b \ln a$ (take `a` from `math` `import *` in the module file so that mathematical functions like `exp` and `log` are defined). Name of program file: `math_rules_failures_cml.py`. \diamond

Exercise 3.24. *Apply the program from Exer. 3.23.*

Import the `equal` function from the module made in Exercise 3.23 and test the three rules from Exercise 3.22 in addition to the following rules:

- $a - b$ 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^ae^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 $(e^ae^b - e^{-a}e^{-b})/2$
- $\tan(a + b)$ 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 `eval` function. Choose A as 1 and B as 50. Make a nicely formatted table with a pair of equivalent expressions at each line followed by the failure rate corresponding to the B values. Does the failure rate depend on the magnitude of the numbers a and b ? Name of program file: `math_rules_failures_table.py`.

Remark. Exercise 3.22 can be solved by a simple program, but if you want to check 17 rules the present exercise demonstrates how important it is to be able to automate the process via the `equal` function and two nested loops over a list of equivalent expressions. \diamond

Exercise 3.25. *Compute 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. 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 failure $n - x$ times) is given by

$$B(x, n, p) = \frac{n!}{x!(n-x)!} p^x (1-p)^{n-x}. \quad (3.8)$$

This formula (3.8) is called the binomial distribution. The expression $x!$ is the factorial of x as defined in Exercise 2.33. Implement (3.8) in a function `binomial(x, n, p)`. Make a module containing this `binomial` function. Include a test block at the end of the module file. Name of program file: `binomial_distribution.py`. \diamond

Exercise 3.26. *Apply the binomial distribution.*

Use the module from Exercise 3.25 to make a program for solving the problems below.

1. 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.

2. 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.

3. 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?

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 .

Name of program file: `binomial_problems.py`. \diamond

Exercise 3.27. *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

$$P(x, t, \nu) = \frac{(\nu t)^x}{x!} e^{-\nu t}. \quad (3.9)$$

This formula is known as the Poisson distribution¹². 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.

¹² It can be shown that (3.9) arises from (3.8) when the probability p of experiencing the event in a small time interval t/n is $p = \nu t/n$ and we let $n \rightarrow \infty$.

Implement (3.9) 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.

1. 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?

2. 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 $\nu t_m = \nu \cdot 50 \text{ years} = 10$ 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 (3.9). 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 $P(0, 1/52, 1/5)$.

3. 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 $P(0, 6, 6)$.

◇