Ch.4: User input and error handling

Joakim Sundnes[1,2]

[1]Simula Research Laboratory
[2]University of Oslo, Dept. of Informatics

Sep 6, 2019

So far, all the values we have assigned to variables have been written directly into our programs. If we want a different value of a variable, we need to edit the code and re-run the program. Of course, this is not how we are used to interacting with computer programs. Usually, a program will recieve some input from users, most often through a graphical user interface (GUI). However, although GUIs dominate in modern human-computer interaction, other ways of interacting with computer programs may be just as efficient, and in some cases far more suitable for processing large amounts of data and for automating repetitive tasks. In this chapter we will show how we can extend our programs with simple yet powerful systems for userr input. In particular, we will see how a program can *receive command-line arguments* when it is run, how to make a program *stop and ask for user input*, and how a program can *read data from files*.

A side-effect of allowing users to interact with programs is that things will often go wrong. Users will often provide the wrong input, and programs should be able to handle this without simply stopping and writing a cryptic error message to the screen. We will introduce a concept known as *exception handling*, and try-except blocks, which is a widespread system for handling errors in programs, and used in Python and many other programming languages.

Finally in this chapter, we shall see how we can *create our own modules*, which can be imported used in other programs just as we have done with modules like `math` and `sys`.

1 Reading input data from users

So far, have implemented various mathematical formulas that involved input variables and parameter, but all of these values have been hardcoded into the programs. Consider for instance the formula

$$y = v_0 t - 0.5gt^2,$$

which we translated into the Python program

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

Programs such as this, but typically including slightly more advanced formulas, are frequently used in a scientific setting. In many cases, it is interesting to vary one or more of the model parameters, to study the effect on the output. In this code, evaluating the formula for a different v_0 or a different t would require editing the code to change the respective value. While this may be an acceptable solution for programs we write and use ourselves, it is not how we are used to interact with computers. In particular, if we write programs that may be used by others, editing the code is inconvenient and may easily introduce errors.

For our programs to be robust and usable they need to be able to read relevant input data from the user. There are several ways this can be accomplished, and we will here consider three different ones, each with its strengths and weaknesses. We will (i) create programs that stop and asks for input from the user, and then continue the execution when the input is received, (ii) enable our programs to receive *command line arguments*, i.e arguments provided when we run the program, and (iii) make the programs read input data from files.

Getting input from questions and anwsers. Consider again the simplest version of our temperature conversion program

```
C = 21
F = (9.0/5)*C + 32
print(F)
```

A natural extension of this program is to let it ask the user for a value of `C`, and then compute and ouput the corresponding Fahrenheit value. A Python function called `input` provides exactly this functionality. For instance a code line like

```
input('What is C?')
```

will make the program stop and display the question `What is C?` in the terminal, and then continue when the user presses `Enter`. For the example at hand, the complete code could look like

```
C = input('C=? ')   # C becomes a string
C = float(C)            # convert to float so we can compute
F = (9./5)*C + 32
print F
```

Running in a terminal window:

```
Terminal> python c2f_qa.py
C=? 21
69.8
```

Notice in particular the line `C = float(C)`. This line is important because the `input` function will always return a text string, which must be converted to an actual number before we can use it in computations. Forgetting this line in the

code above will lead to an error in the line that calculates `F`, since we would try to multiply a string with a float. From these considerations we can also imagine how easy it is to break the program above. The user can type any string, or simply press enter (which makes `C` an empty string), but the conversion `C = float(C)` only works if the string is actually a number. Later in this chapter we shall look at ways to handle such errors and make the programs more robust.

As another example, consider a program that asks the user for an integer `n` and prints the `n` first even numbers:

```
n = int(input('n=? '))

for i in range(1, n+1):
    print 2*i

# or alternatively:
"""
for i in range(2, 2*n+1, 2):
    print(i)
"""
```

Just as the example above, the code is not very robust since it will break from any input that cannot be converted to an integer. We will look at ways to improve this later in the chapter.

Command-line arguments are words written after the program name. When working in a Unix-style terminal window (i.e. Mac, Linux, Windows powershell, etc) we will very often provide arguments when we run a command. These arguments may for instance be names of files or directories, for instance when copying a file with `cp`, or they may change the output from the command, such as `ls -l` to get more detailed output from the `ls` command. We frequently use commands like

```
Terminal> cp -r yourdir ../mydir
Terminal> ls -l
terminal> cd ../mydir
```

Some commands require arguments, for instance you get an error message if you don't give two arguments to `cp`, while other arguments are optional. Standard Unix programs (`rm`, `ls`, `cp`, ...) make heavy use of command-line arguments, (try for instance `man ls`), because they are a very efficient way to provide input. We will make our Python programs do the same. For instance, we want a Python program to be run like

```
Terminal> python myprog.py arg1 arg2 arg3 ...
```

where `arg1 arg2 arg3` etc. are input arguments to the program.

We consider again the very simple temperature conversion program above, but now we want the Celsius degrees to be specified as a command line argument rather than by asking for input, as in

```
Terminal> python c2f_cml.py 21
69.8
```

To use command line arguments in a Python program, we need to import a module named `sys`. More specifically, the command line arguments, or in reality any words we type after the command `python c2f_cml.py`, are automatically stored in a named `sys.argv` (short for argument values), and can be accessed from there:

```
import sys
C = float(sys.argv[1])   # read 1st command-line argument
F = 9.0*C/5 + 32
print F
```

Here, we see that we pull out index one of the `sys.argv` list, and convert this to a float. Just as input provided with the `input` function above, the command line arguments are always strings and need to be converted to floats or integers before they are used in computations. The `sys.argv` variable is simply a list that is created automatically when you run your Python program. The first element, `sys.argv[0]` is simply the name of the `.py`-file with the program. The rest of the list is made up of whatever words one includes after the program file name. Words separated by space become separate elements in the list. A nice way to get a feel for the use of `sys.argv` is to test a simple program that will just print out the contents of the list, for instance by writing this simple code into the file `print_cml.py`:

```
import sys
print sys.argv
```

Running this program in different ways illustrates how the list works, for instance

```
Terminal> python print_cml.py 21 string with blanks 1.3
['print_cml.py', '21', 'string', 'with', 'blanks', '1.3']

Terminal> python print_cml.py 21 "string with blanks" 1.3
['print_cml.py', '21', 'string with blanks', '1.3']
```

We see from the second example that if we want to read in a string containing blanks as a single command line argument, we need to use quotation marks to override the default behavior of putting each word as a separate list element.

In general, the best and safest way to handle input data as above, which comes in as strings, is to convert it to the specific variable type that we need in the program. Below we shall see how such conversions can be made failsafe, so that they handle wrong input from the user. However, Python also offers a couple of very flexible functions to handle input data, named `eval` and `exec`, which are nice to know about. It is not recommended to use these functions extensively, especially not in larger programs, since the code can become messy and error-prone. However, they offer some flexible and fun opportunities for handling input data and are nice to know about. Starting with `eval`, this function simply takes a string `s` as input, and `eval(s)` will evaluate `s` as a regular Python expression [1], just as if it was written directly into the program. The following interactive Python session illustrates how eval works:

[1] This of course requires that `s` is a legal Python expression, otherwise the code stops with an error message.

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

>>> r = eval('[1, 6, 7.5] + [1, 2]')
>>> r
[1, 6, 7.5, 1, 2]
>>> type(r)
<type 'list'>
```

Here, the line `r = eval(s)` is equivalent to writing `r = 1+2`, but using `eval` of course gives much more flexibility since the string is stored in a variable and can be read as input.

Using `eval`, a small Python program can be quite flexible. Consider for instance the following code

```
i1 = eval(input('operand 1: '))
i2 = eval(input('operand 2: '))
r = i1 + i2
print(f'{type(i1)} + {type(i2)} becomes {type(r)} with value{r}')
```

This code can handle multiple input type. Say we save the code in a file `add_input.py` and run it from the terminal, we can input and add integer and float:

```
Terminal> python add_input.py
operand 1: 1
operand 2: 3.0
<type 'int'> + <type 'float'> becomes <type 'float'>
with value 4
```

or two lists:

```
Terminal> python add_input.py
operand 1: [1,2]
operand 2: [-1,0,1]
<type 'list'> + <type 'list'> becomes <type 'list'>
with value [1, 2, -1, 0, 1]
```

We could achieve similar flexibility with conventional type conversion, i.e. using `float(i1)`, `int(i1)`, etc., but it would require much more programming to correctly process the input strings. the `eval` function makes such flexible input handle extremely compact and efficient, but it also quickly breaks if we give it slightly wrong input. Consider for instance the following examples

```
Terminal> python add_input.py
operand 1: (1,2)
operand 2: [3,4]
Traceback (most recent call last):
  File "add_input.py", line 3, in <module>
    r = i1 + i2
TypeError: can only concatenate tuple (not "list") to tuple

Terminal> python add_input.py
```

```
operand 1: one
Traceback (most recent call last):
  File "add_input.py", line 1, in <module>
    i1 = eval(input('operand 1: '))
  File "<string>", line 1, in <module>
NameError: name 'one' is not defined
```

In the first of these examples we try to add a tuple and a list, which one could easily imagine would work, but Python does not allow this and therefore the program breaks. In the second example we try to make the program add two strings, which usually works fine, for instance `"one" +"one"` becomes the string `"oneone"`. However, the eval function breaks when we try to input the first string. To understand why, we need to think about what the corresponding line really means. We try to make the assignment `i1 = eval('one')`, which is equivalent to writing `i1 = one`, but this line does not work unless we have already defined a variable named one. A remedy to this problem is to input the strings with quotation marks, as in the following example

```
Terminal> python add_input.py
operand 1: "one"
operand 2: "two"
<class 'str'> + <class 'str'> becomes <class 'str'> with value onetwo
```

These examples illustrate the benefits of the `eval` function, and also how it easily breaks programs and is generally not recommended for "real" programs. It is useful for quick prototypes, but should usually be avoided in programs that we expect to be used by others, or that we expect to use ourselves over a longer time frame.

The other "magic" input function, named `exec` is similar to `eval`, but while `eval` evaluates an *expression*, `exec` will execute its string argument as one or more complete statements. For instance, if we define a string `s = "r = 1+1"`, `eval(s)` is illegal since `s` is a statement (an assignment), and not a Python expression. Howver, `exec(s)` will work fine, and is the same as including the line `r = 1+1` directly in the code. The following code provides a very simple example to illustrate the difference

```
expression = '1+1'         #store expression in a string
statement = 'r = 1+1'    # store statement in a string
q = eval(expression)
exec(statement)

print(q,r)                  # results are the same
```

We can also use `exec` to execute multiple statements, for instance using multi-line strings:

```
somecode = '''
def f(t):
    term1 = exp(-a*t)*sin(w1*x)
    term2 = 2*sin(w2*x)
    return term1 + term2
'''
exec(somecode)  # execute the string as Python code
```

Here, the `exec`-line will simply execute the string somecode, just as if we had typed the code in directly. After the execution we have defined the function `f(t)` and can call the function later in the code. In itself this example is not very useful, but if we combine exec with actual input its flexibility becomes more apparent. For instance, consider the following code, which will ask the user to type a mathematical expression involving x, and then embed this expression in a Python function:

```
formula = input('Write a formula involving x: ')
code = f"""
def f(x):
    return {formula}
"""
from math import *  # make sure we have sin, cos, log, etc
exec(code)          # turn string formula into live function


#Now the function is defined, and we can ask the
#user for x values and evaluate f(x)
x = 0
while x is not None:
    x = eval(input('Give x (None to quit): '))
    if x is not None:
        y = f(x)
        print f'f({x})={y}'
```

While the program is running, the user is first asked to type a formula, which becomes a function. Then the user is asked to input `x` values until the answer is `None`, and the program evaluates the function `f(x)` for each `x`. The program works even if the programmer knows nothing about the user's choice of `f(x)` when the program is written, which demonstrates the flexibility offered by the `exec` and `eval` functions.

To consider another example, say we want to create a program `diff.py`, which evaluates the numerical derivative of a mathematical expression $f(x)$ for a given value of x. The mathematical expression and the x value shall be given as command line arguments. Usage of the program may look as follows:

```
Terminal> python diff.py 'exp(x)*sin(x)' 3.4
Numerical derivative: -36.6262969164
```

The derivative of a function $f(x)$ can be approximated with a centered finite difference:

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h},$$

for some small h. The implementation of the `diff.py` program may look like

```
from math import *
import sys

formula = sys.argv[1]
code = f"""
def f(x):
    return {formula}
"""
```

```
exec(code)
x = float(sys.argv[2])

def numerical_derivative(f, x, h=1E-5):
    return (f(x+h) - f(x-h))/(2*h)

print(f'Numerical derivative: {numerical_derivative(f, x)}')
```

Again we see that the flexibility of the `exec` function enables us to implement fairly advanced functionality in a very compact program.

2 Reading data from files

Scientific data are often available in files, and reading and processing data from files has always been an important task in programming. The *data science revolution* that we have seen in recent years only increases this importance further. To start with a simple example, consider a file named `data.txt` containing a single column of numbers:

```
21.8
18.1
19
23
26
17.8
```

We assume that we know in advance that there is one number per line, but we don't know the number of lines. How can we read these numbers into a Python program?

The basic way to read a file in Python is to use the function `open`, which takes a file name as argument. The following code illustrates its use:

```
infile = open('data.txt', 'r')    # open file
for line in infile:
    # do something with line
infile.close()                    # close file
```

Here, the first line opens the file `data.txt` for reading, as specified with the letter `r`, and creates a file object named `infile`. If we want to open a file for writing, which we will consider later, we have to use `open('data.txt','w')`. The default is `r`, so for reading a file we could also write `infile = open('data.txt')`. However, including the `r` can be a good bad habit since it makes the purpose of the line more obvious to anyone reading the code. In the second line we enter a usual for-loop, which will treat the object `infile` as a list-like object, and step through the file line by line. For each pass through the for-loop, a single line of the file is read and stored in the string variable `line`. Inside the for-loop we add the code we want for extracting information and processing this line. When there are no more lines in the file the for-loop ends. The final line `infile.close()` closes the file and makes it unavailable for further reading. This line is not very important when reading from files, but it is a good habit to always include it, since it may make a difference when we write to files.

To return to the concrete data file above, say the only processing we want to do is to compute the mean values of the numbers in the file. The complete code could look like this:

```
infile = open('data.txt', 'r')    # open file
mean = 0
for line in infile:
    number = float(line)          # line is string
    mean = mean + number
mean = mean/len(lines)
print(f'The mean value is {mean}')
```

This is a standard way to read files in Python, but as usual in programming there are multiple ways to do things. One alternative for opening a file is to use the following construction:

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

The first line, using `with` and `as` probably does not look familiar, but does essentially the same thing as the `infile = ...` file considered above. One important exception is that if we use `with` we see that all file reading and processing is put inside an indented block of code, and the file is automatically closed when this block is over. For this reason the use of `with` to open files is quite popular, and you are likely to see examples of this in Python programs you encounter. The keyword `with` has other uses in Python too, which we will not cover in this book, but it is particularly common and convenient for reading files and therefore worth mentioning here.

For actually reading the file, after it has been opened, there are also a couple of alternatives to the approach above. For instance, we may read all lines into a list of strings (lines), and then process the list items one by one:

```
lines = infile.readlines()
infile.close()
for line in lines:
    # process line
```

This approach is very similar to the one used above, but here we are done working directly with the file after the first line, and the for-loop instead traverses a list of strings. In practice there is not much difference. Usually, reading and processing files line by line is very convenient, and our good friend the for-loop makes such processing quite easy. However, for files with no natural line structure it may sometimes be easier to read the entire text file into a single string:

```
text = infile.read()
# process the string text
```

The `data.txt` file above contained a single number for each line, which is usually not the case. More often, each line contains many data items, typically both text and numbers, and we want to treat each one differently. For this purpose Python's string class has a built-in method `split`, which is extremely useful. Say we define a string variable `s` with some words separated by blanks.

Then, calling `s.split()` will simply return a list containing the individual words in the string. By default the words are assumed to be separated by blanks, but if we want a different separator we can pass this as an argument to `split`. The following code gives some examples:

```
s = "This is a typical string"
csv = "Excel;sheets;often;use;semicolon;as;separator"
print(s.split())
print(csv.split())
print(csv.split(';'))
```

We see that the first attempt to split the `csv` does not work very well, since the string contains no spaces and the result is therefore a list of length one. Specifying the right separator, as in the last line, solves the problem.

To use `split` in the context of file data, assume we have a file with data about rainfall:

```
Average rainfall (in mm) in Rome: 1188 months between 1782 and 1970
Jan  81.2
Feb  63.2
Mar  70.3
Apr  55.7
May  53.0
Jun  36.4
Jul  17.5
Aug  27.5
Sep  60.9
Oct  117.7
Nov  111.0
Dec  97.9
Year 792.9
```

This is a fairly typical example of a data file. Often, there is one or more header lines with information that we are not really interested in, and the rest of the lines contain a mix of text and numbers. How can we read such a file? The key to processing each line is to use `split` to separate the two words, and for instance store this in two seaparate lists for later processing:

```
months = []
values = []
for line in infile:
    words = line.split()  # split into words
    months.append(words[0])
    values.append(float(words[1]))
```

This recipe, based on a for-loop and then `split` to process each line, will be the fundamental recipe for all file processing throughout this course. It is important to understand it properly, and well worth spending some time reading small data files and playing around with `split` to get familiar with its use. To write the complete program for reading the rainfall data, we must also account for the header line and the fact that the last line contains data of a different type. The complete code may look like:

```
def extract_data(filename):
    infile = open(filename, 'r')
    infile.readline() # skip the first line
```

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

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

This code is merely a combination of tools and functions that we have already introduced above and in earlier chapters, so nothing is truly new. Note however how we skip the first line with a single call to `infile.readline()`, which will simply read the first line and move to the next, thereby being ready to read the lines we are interested in. If there were multiple header lines we would simply add multiple `readline` calls to skip whatever we don't want to process. Notice also how list slicing is used to remove the yearly data from the lists. Negative indices in Python lists run backward starting from the last element, so `annual_avg = rainfall[-1]` will extract the last value in the `rainfall` list and assign it to `annual_avg`. The list slicings `months[:-1]`, `rainfall[:-1]` will extract all elements from the lists up to, but not including the last, thereby removing the yearly data from both lists.

Writing data to files. Writing data to files follows the same pattern as reading. We open a file for writing, and typically use a for-loop to traverse the data we and write it to the file using `write`:

```
outfile = open(filename, 'w')  # 'w' for writing

for data in somelist:
    outfile.write(sometext + '\n')

outfile.close()
```

Notice the inclusion of `\n` in the call to `write`. Unlike `print`, a call to `write` will not by default add a linebreak after each call, so if we don't add this explicitly the resulting file will consist a single long line. It is often more convenient to have a line-structured file, and for this we include the `\n`, which adds a linebreak. The alternative way of opening files can also be used for writing, and ensures the file is automatically closed:

```
with open(filename, 'w') as outfile:  # 'w' for writing
     for data in somelist:
              outfile.write(sometext + '\n')
```

One should use some caution when writing to files from Python programs. If you call `open(filename,'w')` with a file name not exist, a new file will be created.

However, if the file already exists it will simply be deleted and replaced by an empty file. So even if we don't actually write any data to the file, simply opening it for reading will erase all its contents. A safer way to write to files is to use 'open(filename,'a'), which will *append* data at the end of the file if it already exists, and create a new file if it does not exist.

To look at a concrete example, consider the task of writing information from a nested list to a file. We have the nested list (rows and columns):

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

To write these data to a file in tabular form, we follow the steps outlined above and use a nested for-loop (one for-loop inside another) to traverse the list and write the data. The following code will do the trick:

```
with open('tmp_table.dat', 'w') as outfile:
    for row in data:
        for column in row:
            outfile.write(f'{column:14.8f}')
        outfile.write('\n')
```

The resulting file looks like:

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

The nicely aligned columns are caused by the format specifier given to the f-string in the `write` call. The code will work fine without the format specifier, but the columns will not be aligned, and we also need to add a space (' ') after every number, otherwise each line will just be a long string of numbers that are difficult to separate. Another part worth stepping through in the code above is the structure of the nested for-loop. The innermost loop traverses each row, writing the numbers one by one to the file. When this over the program moves to the next line, which writes a linebreak to the file to end the line. Now we have finished one pass of the outer for-loop, and the program moves to the next iteration and the next line in the table. The code for writing each number belongs inside the innermost loop, while writing the linebreak is done in the outer loop.

3 Handling errors in programs

As demonstrated above, allowing user input in our programs will often introduce errors, and as our programs grow in complexity there may be multiple other sources of errors as well. Python has a general set of tools for handling such errors, which is commonly referred to as *exception handling* and is used in many different programming languages. As a motivating example, let us consider a

simple program that reads a Celsius temperature from the command line and prints the temperature in degrees Fahrenheit:

```
import sys
C = float(sys.argv[1])
F = 5./9*C + 32
print(F)
```

As we mentioned above, such a code can easily break if the user provides a command line argument that cannot be converted to a float, i.e. any argument that is not a pure number. Even worse, our program will fail with a fairly cryptic error message if the user does not include a command line argument at all, as in:

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

How can we fix such problems, and make the program more robust with respect to user errors? We could for instance by adding an if-test to check if any command line arguments have been included:

```
import sys
if len(sys.argv) < 2:
    print 'You failed to provide a command-line arg.!'
    sys.exit(1)  # abort
C = float(sys.argv[1])
F = 9.0*C/5 + 32
print(F)
```

This extension solves part of the problem. The program will still stop if it is used incorrectly, but it will provide a more sensible and useful error message:

```
Terminal> python c2f_cml_if.py
You failed to provide a command-line arg.!
```

However, we only solved one potential problem, and using if-tests to test for every possible error can lead to quite complex programs. Instead it is common in Python and many other languages to *try* to do what we indend to, and if it fails, we recover from the error. This principle makes use of a `try-except` block, which has the following general structure:

```
try:
    <statements we intend to do>
except:
    <statements for handling errors>
```

If something goes wrong in the `try` block, Python will raise an *exception* and the execution jumps to the `except` block. Inside the `except` block we need to add our own code for *catching* the exception, basically to detect what went wrong and try to fix it.

Improving the temperature conversion program with try-except. To apply the try-except idea to the conversion program, we try to read `C` from the command-line and convert it to a float, and if this fails we tell the user what went wrong and stop the program:

```
import sys
try:
    C = float(sys.argv[1])
except:
    print 'You failed to provide a command-line arg.!'
    sys.exit(1)  # abort
F = 9.0*C/5 + 32
print(F)
```

One may argue that this is not very different from the one using the if-test, but we shall see that the try-except block gives some benefits. First, we can try to run this program with different input, which immediately reveals a problem:

```
Terminal> python c2f_cml_except1.py
You failed to provide a command-line arg.!

Terminal> python c2f_cml_except1.py 21C
You failed to provide a command-line arg.!
```

Regardless of what goes wrong inside our try-block, Python will raise an exception which needs to be handled by the except-block. In the first case the problem was that there were no arguments, i.e. `sys.argv[1]` does not exist, which gives an `IndexError`. This is handled correctly by our code. In the second case the indexing of `sys.argv` goes fine, but the conversion to a float fails since Python does not know how to convert the string `21C`. This is a different type of error, known as a `ValueError`, and we see that it is not treated very well by our except block. We can improve the code by letting the except-block test for different types of errors, and handle each one differently:

```
import sys
try:
    C = float(sys.argv[1])
except IndexError:
    print('No command-line argument for C!')
    sys.exit(1)  # abort execution
except ValueError:
    print(f'C must be a pure number, not {sys.argv[1]}')
    sys.exit(1)

F = 9.0*C/5 + 32
print(C)
```

Executions:

```
Terminal> python c2f_cml_v3.py
No command-line argument for C!

Terminal> python c2f_cml_v3.py 21C
Celsius degrees must be a pure number, not "21C"
```

Of course, a drawback of this approach is that we need to predict in advance what could possibly go wrong inside the try-block, and write code to handle

these errors. With some experience this is usually not very difficult. Python has many builtin error types, but only a few that are likely to occur and and need to be considered in the programs of this course. In the code above, if the try-block would lead to a different exception than what we catch in our except-block, the code will simply end with a standard error message from Python. If we want to avoid this behavior, and catch all possible exceptions, we could add a generic except block such as

```
except:
    print('Something went wrong in reading input data!')
    sys.exit(1)
```

Such block should be added after the `except ValueError` block in the code above, and will catch any exception that is not an IndexError nor a ValueError. In this particular case it is difficult to imagine what kind of error that would be, but if it occurs it will be caught and handled by our generic except-block.

The programmer can also raise exceptions. In the code above, the exceptions were raised by standard Python functions, and we wrote the code to catch them. Instead of just letting Python raise exceptions, we can raise our own and tailor the message to the problem at hand. We provide two examples on this:

- catching an exception, but raising a new one with an improved (tailored) error message
- raising an exception because of wrong input data

The basic syntax both for raising and re-raising an exception is `raise ExceptionType(message)`. The following code includes both examples.

```
def read_C():
    try:
        C = float(sys.argv[1])
    except IndexError:
        # re-raise, but with specific explanation:
        raise IndexError(
          'Celsius degrees must be supplied on the command line.')
    except ValueError:
        # re-raise, but with specific explanation:
        raise ValueError(
         'Degrees must be number, not "{sys.argv[1]}".')

    # C is read correctly as a number, but can have wrong value:
    if C < -273.15:
        raise ValueError('C={C} is a non-physical value!')
    return C
```

The two except-blocks here are similar to the ones we used above, but instead of catching the exception and handling them (i.e. stopping the program), the blocks here will equip the exceptions with a more specific error message, and then pass them on for Python to handle them. For this particular case the difference is not very large, and one may argue that our first approach is simpler

and therefore better, but in larger programs it may often be better to re-raise exceptions and handle them elsewhere. The last part of the function is different, since the error raised here is not an error as far as Python is concerned. Our conversion formula can handle any value of `C`, but we know that it only makes sense for `C > -273.15`. If a lower `C` temperature is entered, we therefore raise a `ValueError` with a specific error message, even if it is not a Python error in the usual sense.

The following code shows how we can use the function above, and how we can catch and print the error message provided with the exceptions. The construction `except <error> as e` is used to be able to access the error and use it inside the except-block.

```
try:
    C = read_C()
except (IndexError, ValueError) as e:
    # print exception message and stop the program
    print(e)
    sys.exit(1)
```

We can run the code in the terminal to confirm that we get the correct error messages:

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

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

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

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

4 Making your own modules

So far in this course we have frequently used modules like `math` and `sys`, by importing them into our code:

```
from math import log
r = log(6)   # call log function in math module

import sys
x = eval(sys.argv[1])  # access list argv in sys module
```

Modules are extremely useful in Python program, as they contain a collection of useful data and functions (later also classes), that we can reuse in our code. But what if you have written some general and useful functions yourself, that you would like to reuse in more than one program? In such cases it would be convenient to make our own module that we can import in other programs when we need it. Fortunately, this task is very simple in Python; just collect the functions you want in a file, and that's a module!

To look at a specific example, say we want to create a module containing useful formulas for computing with interest rates. We have the mathematical formulas

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

$$A_0 = A\left(1 + \frac{p}{360 \cdot 100}\right)^{-n}, \tag{2}$$

$$n = \frac{\ln \frac{A}{A_0}}{\ln\left(1 + \frac{p}{360 \cdot 100}\right)}, \tag{3}$$

$$p = 360 \cdot 100\left(\left(\frac{A}{A_0}\right)^{1/n} - 1\right), \tag{4}$$

where A_0 is the initial amount, p the interest rate (percent), n is days and A is the final amount. We now want to implement these formulas as Python functions and make a module of them. We write the functions in the usual way:

```
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)
```

If we now save these functions in a file `interest.py`, it becomes a module that we can import as we are used to with the builtin Python modules. As an example of use, say we want to know how much time it takes to double an amount of money, for an interest rate of 5%. The `days` function in the module provides the right formula, and we can import and use it in our program:

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

We can add a *test block* to a module file. If we try to run the module file above with `python interest.py` from the terminal, no output is produced since the functions are never called. Sometimes it can be useful to be able to add some examples of use in a module file, which demonstrate how the functions are called and used. However, if we add regular function calls, print statements and other code to the file, this code will be run when we import the module, which is usually not what we want. The solution is to add such example code in a *test block* at the end of the module file. The test block includes an if-test to

check if the file is imported as a module or run as a regular Python program. The code inside the test block is executed only when the file is run as a program, not when it is imported as a module into another program. The structure of the if-test and the test block looks as follows:

```
if __name__ == '__main__': # this test defineds the test block
    <block of statements>
```

The key is the first line, which checks the value of the builtin variable `__name__`. This string variable is automatically created and is always defined when Python runs. (Try putting `print(__name__)` inside one of your programs or type it in an interactive session.) Inside an imported module, `__name__` holds the name of the module, while in the main program its value is `"__main__"`.

For our specific case, the complete test block may look like

```
if __name__ == '__main__':
    A = 2.2133983053266699
    A0 = 2.0
    p = 5
    n = 730
    A_ = present_amount(A0, p, n)
    A0_ = initial_amount(A, p, n)
    d_ = days(A0, A, p)
    p_ = annual_rate(A0, A, n)
    print(f'A={A_} ({A}) A0={A0_} ({A}) n={n_} ({n}) p={p_} ({p})')
```

A test block is often included simply for demonstrating and documenting how a model is used, or it is used in files that we sometimes use as stand-alone programs and sometimes as modules. As indicated by the name, they are also frequently used to test modules. Using what we learnt about test functions in the previous chapter, we can do this by writing a standard test function that tests the functions in the module, and then calling this function from the test block:

```
def test_all_functions():
    # Define compatible values
    A = 2.2133983053266699; A0 = 2.0; p = 5; n = 730
    # Given three of these, compute the remaining one
    # and compare with the correct value (in parenthesis)
    A_computed = present_amount(A0, p, n)
    A0_computed = initial_amount(A, p, n)
    n_computed = days(A0, A, p)
    p_computed = annual_rate(A0, A, n)
    def float_eq(a, b, tolerance=1E-12):
        """Return True if a == b within the tolerance."""
        return abs(a - b) < tolerance

    success = float_eq(A_computed,  A)  and \
              float_eq(A0_computed, A0) and \
              float_eq(p_computed,  p)  and \
              float_eq(n_computed,  n)
    assert success  # could add message here if desired

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

Since we have followed the naming convention of test functions, the function will be called if we run for instance `pytest interest.py`, but since we call it from inside the test block the test can also be run simply by `python interest.py`. In the latter case the test will produce no output unless there are errors.

How can Python find our new module? Python has a number of designated places where it looks for modules. The first place it looks is in the same folder as the main program, so if we put our module files there it will always be found. However, this is not very convenient if we write a more general module that we plan to use from several other programs. Such modules can be put in a designated directory, say `/Users/sundnes/lib/python/mymods` or any other directory name that you choose. Then we need to tell Python that it should look for modules in this directory, otherwise it will not find the module. On Unix-like systems (Linux, Mac, etc.), the standard way to tell Python where to look is to edit the *environment variable* called `PYTHONPATH`. Environment variables are variables that hold important information used by the operating system, and `PYTHONPATH` is used to specify folders where Python should look for modules. If you type `echo $PYTHONPATH` in the terminal window you will most likely get no output, since you have not added any folders names to this variable. We can put our new folder name in this variable by running command

```
export PYTHONPATH=/Users/sundnes/lib/python/mymods
```

However, if the `PYTHONPATH` already contained some folders these would now be lost, so to be on the safe side it is better to use

```
export PYTHONPATH=$PYTHONPATH:/Users/sundnes/lib/python/mymods
```

This latter command will simply add our new folder at the end of what is already in our `PYTHONPATH` variable. To avoid having to run this command every time we want to import some modules, we can put it in the file `.bashrc`, which ensures it is run automatically when we open a new terminal window. The `.bashrc` file should be in your home directory (e.g. '/Users/sundnes/.bashrc'), and will show up with `ls -l`. (The dot at the start of the filename makes this a *hidden* file, so it will not show up simply with `ls`.) If the file is not there, you can simply create it in an editor and save it in your home directory, and the system should find it and read it automatically the next time you open a terminal window. As an alternative to editing the system-wide environment variable, we can also add our directory to the path from inside the program. Putting a line like this inside your code, before import the module, should make Python able to find it:

```
sys.path.insert(0, '/Users/hpl/lib/python/mymods')
```

As an alternative to creating our own directory for modules, and then tell Python where to find it, we can place our modules in one of the places where Python always looks for modules. The location of these vary a bit between different Python installations, but the directory itself is usually named

`site-packages`. If you have installed `numpy`[2] or another package which is not part of the standard Python distribution, you can locate the correct directory by importing this package. For instance, type the following in an interactive Python shell:

```
>>> import numpy
>>> numpy.__file__
'/Users/sundnes/anaconda3/lib/python3.7/site-packages/numpy/__init__.py'
>>>
```

The last line reveals the location of the `site-packages` directory, and placing your own modules here will ensure that Python finds them.

5 Summary of Chapter 4

Reading input from the user. We have introduced two main ways to let users give input data to our programs. We can use `input` to let the program stop and ask for input from the user:

```
var = input('Give value: ')   # var is string!

# if var needs to be a number:
var = float(var)
# or in general:
var = eval(var)
```

Alternatively, input data can be provided as command-line arguments, and accessed through the list `sys.argv`:

```
import sys
parameter1 = eval(sys.argv[1])
parameter3 = sys.argv[3]          # string is ok
parameter2 = eval(sys.argv[2])
```

Recall that `sys.argv[0]` is the program name, so the remaining list `sys.argv[1:]` contains the command-line arguments.

Using `eval` and `exec` to process input. The "magic" functions `eval` and `exec` let us run Python code in string variables as if the code was typed directly into the program. For evaluating expressions, we use `eval`:

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

While `exec` is used for running strings with complete Python code, one or multiple lines:

[2]Numpy is a package for numerical calculations, which is automatically installed if you installed Python from Anaconda. Otherwise it can be installed for instance with `pip` or other tools. We will use this package extensively in Chapter 5.

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

Try-except blocks provide a flexible way to handle errors. The structure of a try-except block looks as follows:

```
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 general idea is that we put whatever code we suspect could go wrong inside the try block, and write code to handle errors of different types inside the individual except-blocks. We have seen that we can also raise Exceptions ourselves, for instance if we encounter non-physical or otherwise wrong input data:

```
if z < 0:
    raise ValueError(
        f'z={z} is negative - cannot do log(z)')
```

Reading from and writing to files. Arguably, the most useful and relevant way to get data into a program is to read from files. Opening files for reading or writing is done with the function `open`:

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

Once it is opened, we can read the contents of a file in many ways:

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

The last option is often the most convenient. We step through the file line by line using a for-loop, processing one line for each pass of the loop. For processing lines from files, or string variables in general, the method `split` is extremely useful. By default, `split` will split the words separated by spaces in a text string into a list of words:

```
line1 = "August 16.9"
words = line1.split()
month = words[0]
temp = float(words[1])

line2 = "20.0,45,98"
numbers = line2.split(',')
numbers = [float(x) for x in numbers]
```

Writing to files is done in much the same way as reading. It is important to close the file after you are done writing:

```
# Writing
outfile.write(s)    # add \n if you need it

# Closing
infile.close()
outfile.close()
```