Chapter 5
User Input and Error Handling

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 rerun the program. Of course, this is not how we are used to interacting with computer programs. Usually, a program will receive 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 can be just as efficient and, in some cases, far more suitable for processing large amounts of data and automating repetitive tasks. In this chapter we will show how we can extend our programs with simple yet powerful systems for user 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 such events without simply stopping and writing a cryptic error message to the screen. We will introduce a concept known as *exception handling*, which is a widespread system for handling errors in programs, used in Python and many other programming languages.

Finally, in this chapter, we shall see how to create our own modules that can be imported for use in other programs, just as we have done with the `math` module in previous chapters.

5.1 Reading User Input Data

So far, we have implemented various mathematical formulas that involved input variables and parameters, but all of these values have been hard-coded into the programs. To introduce a new example, consider the following formula, which gives an estimate of the atmospheric pressure p as a function of

J. Sundnes, *Introduction to Scientific Programming with Python*, Simula SpringerBriefs on Computing 6,
https://doi.org/10.1007/978-3-030-50356-7_5

altitude h:

$$p = p_0 e^{-h/h_0},$$

where p_0 is the pressure at sea level (≈ 100 kPa) and h_0 is the so-called scale height (≈ 8.4km). A Python program for evaluating this formula could look like

```
from math import exp

p0 = 100.0    #sea level pressure (kPa)
h0 = 8400     #scale height (m)

h = 8848
p = p0 * exp(-h/h0)
print(p)
```

Of course, we are usually interested in evaluating the formula for different altitudes, which, in this code, would require editing the line `h = 8848` to change the respective variable, and then rerunning the program. This solution could be acceptable for programs we write and use ourselves, but it is not how we are used to interacting with computers. In particular, if we write programs that could be used by others, editing the code this way is inconvenient and can easily introduce errors.

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

Obtaining input from questions and answers. A natural extension of this program is to allow it to ask the user for a value of `h`, and then compute and output the corresponding atmospheric pressure. A Python function called `input` provides exactly this functionality. For instance a line such as

```
input('Input the altitude (in meters):')
```

will make the program stop and display the text `Input the altitude (in meters):` in the terminal, and then continue when the user presses `Enter`. The complete code could look like

```
from math import exp

h = input('Input the altitude (in meters):')
h = float(h)

p0 = 100.0    #sea level pressure (kPa)
h0 = 8400     #scale height (m)

p = p0 * exp(-h/h0)
print(p)
```

Running the program in a terminal window could look like:

Terminal

```
Terminal> python altitude.py
Input the altitude (in meters): 2469
74.53297273796525
```

Notice in particular the line `h = float(h)`, which is an example of the type conversions mentioned in Chapter 2. 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 `amount`, since we would by trying 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 `h` an empty string), but the conversion `h = float(h)` only works if the string is a number.

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)
```

Here we convert the input text using `int(...)`, since the `range` function only accepts integer arguments. Just as in the example above, the code is not very robust, since it will break from any input that cannot be converted to an integer. Later in this chapter we will look at ways to handle such errors and make the programs more robust.

Command line arguments are words written after the program name. When working in a Unix-style terminal window (e.g., Mac, Linux, Windows PowerShell), we often provide arguments when we run a command. These arguments can be names of files or directories, for example, when copying a file with `cp`, or they can change the output from the command, such as `ls -l` to obtain more detailed output from the `ls` command. Anyone who is used to working in Unix-style terminals will be familiar with commands like these:

Terminal

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

Some commands require arguments – for instance, you receive an error message if you do not give two arguments to `cp` – while other arguments are optional. Standard Unix programs make heavy use of command line arguments, (try, for instance, typing `man ls`), because they are a very efficient

way of providing input and modifying program behavior. We will make our Python programs do the same, and write programs that can be run as

Terminal

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

where `arg1 arg2 arg3`, and so forth are input arguments to the program.

We again consider the air pressure calculation program above, but now we want the altitude to be specified as a command line argument rather than obtained by stopping and asking for input. For instance, we want to run the program as followss:

Terminal

```
Terminal> python altitude_cml.py 2469
74.53297273796525
```

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 altitude.py`, are automatically stored in a list named `sys.argv` (short for argument values) and can be accessed from there:

```
import sys
from math import exp

h = sys.argv[1]
h = float(h)

p0 = 100.0    #sea level pressure (kPa)
h0 = 8400     #scale height (m)

p = p0 * exp(-h/h0)
print(p)
```

Here, we see that we pull out the element with index one from the `sys.argv` list, and convert it to a float. Just as the 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 your Python program is run. The first element, `sys.argv[0]` is the name of the `.py`-file containing the program. The remainder of the list is made up of whatever words we type after the program filename. Words separated by a space become separate elements in the list. A nice way to gain 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

```
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 each word being treated as a separate list element.

5.2 Flexible User Input with `eval` and `exec`

Generally, the safest way to handle input data in the form of text strings is to convert it to the specific variable type needed in the program. We did this above, using the type conversions `int(...)` and `float(...)`, and we will see below how such conversions can be made failproof and handle imporper user input. However, Python also offers a couple of more flexible functions to handle input data, namely, `eval` and `exec`, which are nice to know about. Extensive use of these functions is not recommended, 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. Starting with `eval`, this function simply takes a string `s` as input and evaluates it as a regular Python expression, just as if it were written directly into the program. Of course, `s` must be a legal Python expression, otherwise the code stops with an error message. The following interactive Python session illustrates how `eval` works:

```
>>> 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` gives much more flexibility, of course, since the string is stored in a variable and can be read as input.

A small Python program using `eval` 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 types. If we save the code in a file `add_input.py` and run it from the terminal, we can, for instance, add integer and float numbers, as in:

Terminal

```
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, as follows:

Terminal

```
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, that is, using `float(i1)`, `int(i1)`, and so on, but that would require much more programming to correctly process the input strings. The `eval` function makes such flexible input handling extremely compact and efficient, but it also quickly breaks if the input is slightly wrong. Consider the following examples:

Terminal

```
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

```
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 others to use or that we expect to use ourselves over a longer time frame.

The other "magic" text handling function is named `exec`, and it is fairly similar to `eval`. However, whereas `eval` evaluates an *expression*, `exec` executes a string argument as one or more complete statements. For instance, if we define a string `s = "r = 1+1"`, `eval(s)` is illegal, since the value of `s` (`"r = 1+1"`) is a statement (an assignment), and not a Python expression. However, `exec(s)` will work fine and is the same as including the line `r = 1+1` directly in the code. The following code illustrates 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 directly in our program. After the call to `exec` we have defined the function `f(t)` and can call this function in the usual way. Although this example does not seem very useful, the flexibility of `exec` becomes more apparent if we combine it with actual user input. For instance, consider the following code, which asks the user to type a mathematical expression involving x and then embeds 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` that evaluates the numerical derivative of a mathematical expression $f(x)$ for a given value of x. The mathematical expression and the x value will be given as command line arguments. The program could be used as follows:

Terminal

```
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 could 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.

5.3 Reading Data from Files

Scientific data are often available in files, and reading and processing data from files have always been important tasks in programming. The data science revolution that we have witnessed in recent years has only increased their importance further, since all data analysis starts with being able to read data from files and store them in suitable data structures. 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 do not 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 an 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, to read a file we could also simply write `infile = open('data.txt')`. However, including the `r` can be a good habit, since it makes the purpose of the line more obvious to anyone reading the code. In the second line, we enter a regular 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`, and inside the for loop we add any code we want for processing this line. When there are no more lines in the file, the for loop ends, just as when looping over a regular list. 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 can make a difference when writing to files.

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

```
infile = open('data.txt', 'r')    # open file
mean = 0
lines = 0
for line in infile:
    number = float(line)          # line is string
    mean = mean + number
    lines += 1
mean = mean/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. An alternative way of opening a file, which many will consider more modern, is by using the following code:

```
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 it does essentially the same thing as the line `infile = open(...)` in the first example. One important difference is that, if we use `with` we see that all file reading and processing must be put inside an indented block of code, and the file is automatically closed when this block has been completed. Therefore,

the use of `with` to open files is quite popular, and you are likely to see it in Python programs you encounter. The keyword `with` has other uses in Python that we will not cover in this book, but it is particularly common and convenient for reading files and therefore worth mentioning here.

To actually read a file after it has been opened, there are a couple of alternatives to the approach above. For instance, we can read all the 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 the list of strings. In practice there is not much difference. Usually, 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 can 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 contain 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 might want to treat each one differently. For this purpose Python's string type has a built-in method named `split` that is extremely useful. Say we define a string variable `s` with some words separated by blank spaces. 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 it as an argument to `split`. The following code gives some examples:

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

```
['This', 'is', 'a', 'typical', 'string']
['Excel;sheets;often;use;semicolon;as;separator']
['Excel', 'sheets', 'often', 'use', 'semicolon', 'as', 'separator']
```

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

To illustrate the use of `split` in the context of file data, assume we have a file with data on 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
```

Although this data file is very small, it is a fairly typical example. Often, there are one or more header lines with information that we are not really interested in processing, and the remainder 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 them in two separate 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]))
```

These steps, involving a for loop and then `split` to process each line, will be the fundamental recipe for all file processing throughout this book. It is important to understand these steps properly and well worth spending some time reading small data files and playing around with `split` to become 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 could 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.txt')
```

```
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 one, thus being ready to read the lines in which we are interested. If there are multiple header lines in the file we can 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 slicing `months[:-1]`, `rainfall[:-1]` will extract all elements from the lists up to, but not including the last one, thereby removing the yearly data from both lists.

5.4 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, which we then write 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 line break after each call by defauls, so if we do not add this explicitly, the resulting file will consist of 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 line break. The alternative way of opening files can also be used for writing, and it ensures that 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 caution when writing to files from Python programs. If you call `open(filename,'w')` with a filename that does not exist, a new file will be created; however, if a file with that name exists, it will simply be deleted and replaced by an empty file. Therefore, even if we do not 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 to the end of the file if it already exists, and create a new file if it does not exist.

For a concrete example, consider the task of writing information from a nested list to a file. We have following 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 or, otherwise, each line will just be a long string of numbers that are difficult to separate. The structure of the nested for loop is also worth stepping through in the code above. The innermost loop traverses each row, writing the numbers one by one to the file. When this inner loop is done the program moves to the next line (`outfile.write('\n')`), which writes a linebreak to the file to end the line. After this line, one pass of the outer for loop is finished 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, whereas the code for writing the line break is in the outer loop, since we only want one line break for each line.

5.5 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 can be multiple

other sources of errors as well. Python has a general set of tools for handling such errors that is commonly referred to as *exception handling*, and it used in many different programming languages. To illustrate how it works, let us return to the example with the atmospheric pressure formula:

```
import sys
from math import exp

h = sys.argv[1]
h = float(h)

p0 = 100.0; h0 = 8400
print(p0 * exp(-h/h0))
```

As mentioned above, this code can easily break if the user provides a command line argument that cannot be converted to a float, that is, any argument that is not a pure number. Potentially even worse is our program failing with a fairly cryptic error message if the user does not include a command line argument at all, as in the following:

Terminal

```
Terminal> python altitude_cml.py
Traceback (most recent call last):
  File "altitude_cml.py", line 4, in ?
    h = 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? One possible solution is to add 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.!')
    exit()  # abort

h = float(sys.argv[1])

p0 = 100.0; h0 = 8400
print(p0 * exp(-h/h0))
```

The function call `exit()` will simply abort the program, so 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

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

However, we only handle one of the potential errors, 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 intend to and, if it fails, to recover from the error. This principle uses the 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. If no errors occur inside the `try` block, the code inside the `except` block is not run and the program simply moves on to the first line after the try-except block.

Improving the atmospheric pressure program with try-except. To apply the try-except idea to the air pressure program, we can try to read `h` 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:
    h = float(sys.argv[1])
except:
    print('You failed to provide a command line arg.!')
    exit()

p0 = 100.0; h0 = 8400
print(p0 * exp(-h/h0))
```

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

Terminal

```
Terminal> python altitude_cml_except1.py
You failed to provide a command line arg.!

Terminal> python altitude_cml_except1.py 2469m
You failed to provide a command line arg.!
```

Regardless of what goes wrong inside our try block, Python will raise an exception that needs to be handled by the except block. The problem with our code is that all possible errors will be handled the same way. In the first case, the problem is that there are no arguments, that is, `sys.argv[1]` does not exist, which leads to an `IndexError`. This situation is correctly handled

by our code. In the second case, we provide an argument, so the indexing of `sys.argv` goes well, but the conversion fails, since Python does not know how to convert the string `2469m` to a float. 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 handling each one differently:

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

p0 = 100.0; h0 = 8400
print(p0 * exp(-h/h0))
```

The following two examples illustrate how this more specific error handling works:

Terminal

```
Terminal> python altitude.py
No command line argument for h!

Terminal> python altitude.py 2469m
The altitude must be a pure number, not "2469m"
```

Of course, a drawback of this approach is that we need to guess in advance what could go wrong inside the try-block, and write code to handle all possible errors. However, with some experience, this is usually not very difficult. Python has many built-in error types, but only a few that are likely to occur and which need to be considered in the programs we encounter throughout this book. In the code above, if the try block would leads to a different exception than what we catch in our except block, the code will simply end with a standard Python error message. 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!')
    exit()
```

Such a 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 can be 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 error messages to the problem at hand. We provide two examples of such use:

- Catching an exception, but raising a new one (re-raising) with an improved (tailored) error message.
- Raising an exception because of input data that we know are wrong, although Python accepts the data.

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

```
import sys

def read_altitude():
    try:
        h = float(sys.argv[1])
    except IndexError:
        # re-raise, but with specific explanation:
        raise IndexError(
          'The altitude must be supplied on the command line.')
    except ValueError:
        # re-raise, but with specific explanation:
        raise ValueError(
         f'Altitude must be number, not "{sys.argv[1]}".')

    # h is read correctly as a number, but has a wrong value:
    if h < -430 or h > 13000:
        raise ValueError(f'The formula is not valid for h={h}')
    return h
```

Here we have defined a function to handle the user input, but the code is otherwise quite similar to the previous examples. As above, the except blocks will catch two different types of error, but, instead of handling them (i.e., stopping the program), the blocks here will equip the exceptions with more specific error messages, and then pass them on to be handled somewhere else in our program. For this particular case, the difference is not very large, and one could argue that our first approach is simpler and therefore better; however, in larger programs it can 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. We can input any value of `h` into our formula, and, unless we input a large negative number, it will not give rise to a Python error[1]. However, as an estimate of air pressure the formula is only valid in the troposphere, the lower part of the Earth's atmosphere, which extends from the lowest point on Earth (on land), at 430

[1] If we set `h` to be a large negative number, the argument for the `exp` function becomes large and positive, and leads to an `OverflowError`. However, this error will occur only for values far outside the range of validity for our air pressure estimate.

m below sea level, to around 13 km above sea level. We can therefore let the program raise a `ValueError` for any `h` outside this range, even if it does not involve 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 access the error and use it inside the except block, as follows:

```
try:
    h = read_altitude()
except (IndexError, ValueError) as e:
    # print exception message and stop the program
    print(e)
    exit()
```

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

Terminal

```
Terminal> python altitude_cml_except2.py
The altitude must be supplied on the command line.

Terminal> python altitude_cml_except2.py 1000m
Altitude must be number, not 1000m.

Terminal> python altitude_cml_except2.py 20000
The formula is not valid for h=20000.

Terminal> python altitude_cml_except2.py 8848
34.8773231887747
```

5.6 Making Modules

So far in this course we have frequently used modules such as `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 programs, since they contain a collection of useful data and functions (as well as classes later), 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 your own module that you can import into other programs when needed. Fortunately, this task is very simple in Python; just collect the functions you want in a file, and you have a new module!

To look at a specific example, say we want create a module containing the interest formula considered earlier and a few other useful formulas for computing with interest rates. We have the mathematical formulas

$$A = P(1 + r/100)^n, \tag{5.1}$$

$$P = A(1 + r/100)^{-n}, \tag{5.2}$$

$$n = \frac{\ln \frac{A}{P}}{\ln(1 + r/100)}, \tag{5.3}$$

$$r = 100\left(\left(\frac{A}{P}\right)^{1/n} - 1\right), \tag{5.4}$$

where, as above, P is the initial amount, r is the interest rate (percent), n is the number of years, 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(P, r, n):
    return P*(1 + r/100)**n

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

def years(P, A, r):
    return ln(A/P)/ln(1 + r/100)

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

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

```
from interest import years
P = 1; r = 5
n = years(P, 2*P, p)
print(f'Money has doubled after {n} 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, to demonstrate how the functions

are called and used and give sensible output if we run the file with `python interest.py`. However, if we add regular function calls, print statements and other code to the file, this code will also be run whenever 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 if it is run as a regular Python program. The code inside the test block is then executed only when the file is run as a program, and not when it is imported as a module into another program. The structure of the if-test and the test block is as follows:

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

The key is the first line, which checks the value of the built-in 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, whereas in the main program its value is `"__main__"`.

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

```
if __name__ == '__main__':
    A = 2.31525
    P = 2.0
    r = 5
    n = 3
    A_ = present_amount(P, r, n)
    P_ = initial_amount(A, r, n)
    n_ = years(P, A, r)
    r_ = annual_rate(P, A, n)
    print(f'A={A_} ({A}) P={P_} ({A}) n={n_} ({n}) r={r_} ({p})')
```

Test blocks are often included simply for demonstrating and documenting how modules are used, or they are included 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 learned 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 simply calling this function from inside the test block:

```
def test_all_functions():
    # Define compatible values
    A = 2.31525; P = 2.0; r = 5.0; n = 3
    # Given three of these, compute the remaining one
    # and compare with the correct value (in parenthesis)
    A_computed = present_amount(P, r, n)
    P_computed = initial_amount(A, r, n)
    n_computed = years(P, A, r)
    r_computed = annual_rate(P, 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. However, if we import the module to use in another program, the test function is not run, because the variable `__name__` will be the name of the module (i.e. `interest`) and the test `__name__ == '__main__'` will be evaluated as false.

How Python finds 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; therefore, if we put our module files there, they will always be found. However, this is not very convenient if we write more general modules 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 to 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 by editing the *environment variable* called `PYTHONPATH`. Environment variables are variables that hold important information used by the operating system, and `PYTHONPATH` is used to specify the folders where Python should look for modules. If you type `echo $PYTHONPATH` in the terminal window, you will most likely obtain no output, since you have not added any folder names to this variable. We can put our new folder name in this variable by running the command

Terminal

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

However, if the `PYTHONPATH` already contained any folders, these will now be lost; therefore, to be on the safe side, it is better to use

Terminal

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

This last command will simply add our new folder to the end of what is already in our `PYTHONPATH` variable. To avoid having to run this command

every time we want to import a module, we can put it in the file `.bashrc`, to ensure that 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 be listed with `ls -a`. (The dot at the start of the filename makes it a *hidden* file, so it will not show up with just `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 systemwide environment variable, we can also add our directory to the path from inside the program. Putting a line such as this inside your code, before you import the module, should allow Python to find it:

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

As an alternative to creating your own directory for modules, and then tell Python where to find them, you can place the modules in one of the places where Python always looks for modules. The location of these varies a bit between different Python installations, but the directory itself is usually named `site-packages`. If you have installed NumPy[2] or another package that 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 there will ensure Python will find them.

[2]NumPy is a package for numerical calculations. It is not part of the standard Python distribution, but it is often installed automatically if you install Python from other sources, for instance, from Anaconda. Otherwise, it can be installed for instance, with `pip` or other tools. The NumPy package will be used extensively in the next chapter.