Ch.5: User input and error handling

Joakim Sundnes[1,2]

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

Sep 13, 2021

0.1 Plan for week 37

Monday 13/9

- Exercises from last week:
  - 2.15, 3.20, 3.28
  - 3.23 left as self study
- Exercise 2.19, 2.20, 2.21 (roundoff errors)
- Exercise 3.7 (function and test function)
- Reading input from users:
  - Stop and ask for input
  - Command-line arguments
  - Reading from files

Thursday 16/9

- Exercise 4.1, 4.2, 4.3, 4.4 (user input, file read/write)
- Handling errors with try-except
- Exercise 4.5

All exercises are from H.P. Langtangen's "A Primer on..."

0.2 Programs until now hardcode input data

Example; evaluate the barometric formula

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

```
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)
```

Note:

- The input altitude is hardcoded (explicitly set)
- Changing input data requires *editing*
- This is considered bad programming
  (because editing programs may easily introduce errors)
- Rule: read input from user - avoid editing a correct program

0.3 How do professional programs get their input?

- Consider a web browser: how do you specify a web address? How do you change the font?
- You don't need to go into the program and edit it...

How can we specify input data?

- Ask the user questions and read answers
- Read command-line arguments
- Read data from a file

0.4 What about GUIs?

- Most programs today fetch input data from *graphical user interfaces* (GUI), consisting of windows and graphical elements on the screen: buttons, menus, text fields, etc.
- Why don't we learn to make such type of programs?
  - GUI demands much extra complicated programming
  - Experienced users often prefer command-line input
  - Programs with command-line or file input can easily be combined with each other, this is difficult with GUI-based programs
- Command-line input will probably fill all your needs in university courses

0.5 Alternative 1: Stop and ask for input

Sample program:

```
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 in a terminal window:

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

0.6 Another example of using `input`

Ask the user for an integer `n` and print the `n` first even numbers:

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

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

Notice the conversion of the input (i.e.'int','float'), which is needed since all input is initially a text string.

0.7 Alternative 2: Command-line arguments

Example command-line arguments:

```
Terminal> python myprog.py arg1 arg2 arg3 ...
Terminal> cp -r yourdir ../mydir
Terminal> ls -l
```

Unix programs (`rm`, `ls`, `cp`, ...) make heavy use of command-line arguments, (see e.g. `man ls`). We shall do the same.

0.8 How to use a command-line argument in our sample program

The user wants to specify `h` as a *command-line argument* after the name of the program when we run the program:

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

Command-line arguments are the "words" after the program name, and they are stored in the list `sys.argv`:

```
import sys
from math import exp

h = float(sys.argv[1])

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

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

0.9 Command-line arguments are separated by blanks

Here is another program `print_cml.py`:

```
import sys; print(sys.argv)
```

Demonstrations:

```
Terminal> python print_cml.py 1 2 3
['print_cml.py', '1', '2', '3']

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

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

Note 1: use quotes, as in `"string with blanks"`, to override the rule that command-line arguments are separated by blanks.

Note 2: all list elements are surrounded by quotes, demonstrating that command-line arguments are strings.

0.10 Alternative conversion with the magic `eval` function

- `eval(s)` evaluates a string object `s` as if the string had been written directly into the program
- Gives a more flexible alternative to converting with `float(s)`

```
>>> 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'>
```

0.11 With eval, a little program can do much

Program `input_adder.py`:

```
i1 = eval(input('Give input: '))
i2 = eval(input('Give input: '))
r = i1 + i2
print (f'{type(i1)} + {type(i2)} becomes {type(r)} \nwith value {r}')
```

0.12 This great flexibility also quickly breaks programs...

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

Terminal> python input_adder.py
operand 1: 4
operand 2: 'Hello, World!'
Traceback (most recent call last):
  File "add_input.py", line 3, in <module>
    r = i1 + i2
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

0.13 A similar magic function: exec

- `eval(s)` evaluates an *expression* `s`
- `eval('r = 1+1')` is illegal because this is a statement, not only an expression
- ...but we can use `exec` to turn one or more complete statements into live code:

```
statement = 'r = 1+1'   # store statement in a string
exec(statement)
print(r)                 # prints 2
```

For longer code we can use 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
```

0.14 Reading data from files

Scientific data are often available in files. We want to read the data into objects in a program to compute with the data.

Example on a data file.

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

One number on each line. How can we read these numbers?

0.15 Reading a file line by line

Basic file reading:

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

Compute the mean values of the numbers in the file:

```
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
infile.close()
mean = mean/lines
print(mean)
```

0.16 Alternative way to open a file

The modern `with` statement:

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

Notice:

- All the code for processing the file is an indented block
- The file is automatically closed

0.17 Alternative ways to read a file

Read all lines at once into a list of strings (lines):

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

for line in lines:
    # process line
```

Reading the whole file into a string:

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

0.18 Most data files contain text mixed with numbers

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
```

How do we read such a file?

0.19 Processing each line with `split()`

- The key idea to process each line is to split the line into words
- Python's `split` method is extremely useful for splitting strings

General recipe:

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

0.20 Become familiar with the `split()` method!

- By default, `split()` will split words separated by space
- We can specify any string `s` as separator: `split(s)`

```
>>> line = 'Oct  117.7'
>>> words = line.split()
>>> words
['Oct', '117.7,']
>>> type(words[1])   # string, not a number!
<type 'str'>
>>> line2 = 'output;from;excel'
>>> line2.split(';')
['output', 'from', 'excel']
```

0.21 Complete program for reading rainfall data

```
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)
```

0.22 Writing data to file

Basic pattern:

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

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

outfile.close()
```

Can *append* text to a file with `open(filename, 'a')`.

0.23 Example: Writing a table to file

Problem: We have a 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      ]]
```

Write these data to file in tabular form

Solution:

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

0.24 Back to a simple program that reads from the command line

Code:

```
import sys
from math import exp

h = float(sys.argv[1])

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

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

Next topic: How to handle wrong input from the user?

0.25 The program stops with a strange error message if the command-line argument is missing

A user can easily use our program in a wrong way, e.g.,

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

Why?

1. The user forgot to provide a command-line argument
2. `sys.argv` has then only one element, `sys.argv[0]`, which is the program name (`altitude.py`)
3. Index 1, in `sys.argv[1]`, points to a non-existing element in the `sys.argv` list
4. Any index corresponding to a non-existing element in a list leads to `IndexError`

0.26 We should handle errors in input

How can *we* take control, explain what was wrong with the input, and stop the program without strange Python error messages?

```
# Program altitude_if.py

import sys
if len(sys.argv) < 2:
    print 'You failed to provide a command-line arg.!'
    sys.exit(1)  # abort
...

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

0.27 Exceptions as an alternative to if tests

- Rather than test *if something is wrong, recover from error, else do what we indended to do*, 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

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

If something goes wrong in the `try` block, Python raises an *exception* and the execution jumps immediately to the `except` block.

0.28 The temperature conversion program with try-except

Try to read `h` from the command-line, if it fails, tell the user, and abort execution:

```
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))
```

Execution:

```
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.!
```

0.29 It is good programming style to test for specific exceptions

It is good programming style to test for specific exceptions:

```
try:
    h = float(sys.argv[1])
except IndexError:
    print 'You failed to provide a command-line arg.!'
```

If we have an index out of bounds in `sys.argv`, an `IndexError` exception is raised, and we jump to the `except` block.

If any other exception arises, Python aborts the execution:

```
Terminal> python altitude_cml_except1.py 2469m
Traceback (most recent call last):
  File "altitude.py", line 3, in <module>
    C = float(sys.argv[1])
ValueError: invalid literal for float(): 2469m
```

0.30 Improvement: test for `IndexError` and `ValueError` exceptions

```
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))
```

Executions:

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

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

0.31 The programmer can raise exceptions

- 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
- Baisc syntax: `raise ExceptionType(message)`

0.32 Examples on re-raising exceptions with better messages

```
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
```

0.33 Calling the previous function and running the program

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

Executions:

```
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
```

0.34 Making your own modules

We have frequently used modules like `math` and `sys`:

```
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
```

Characteristics of modules:

- Collection of useful data and functions
  (later also classes)
- Functions in a module can be reused in many different programs
- If you have some general functions that can be handy in more than one program, make a module with these functions
- It's easy: just collect the functions you want in a file, and that's a module!

0.35 Case on making our own module

Here are formulas for computing with interest rates:

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

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

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

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

A_0: initial amount, p: percentage, n: days, A: final amount

We want to make a module with these four functions.

0.36 First we make Python functions for the formuluas

```
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)
```

0.37 Then we can make the module file

- Collect the 4 functions in a file `interest.py`
- Now `interest.py` is actually a module `interest` (!)

Example on use:

```
# How long time does it take to double an amount of money?

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:.1f} years')
```

0.38 Adding a test block in a module file

- Module files can have an if test at the end containing a *test block* for testing or demonstrating the module
- The test block is not executed when the file is imported as a module in another program
- The test block is executed *only* when the file is run as a program

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

In our case:

```
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})')
```

0.39 Test blocks are often collected in functions

Let's make a real *test function* for what we had in 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()
```

0.40 How can Python find our new module?

- If the module is in the same folder as the main program, everything is simple and ok
- Home-made modules are normally collected in a common folder, say `/Users/hpl/lib/python/mymods`
- In that case Python must be notified that our module is in that folder

Technique 1: add folder to `PYTHONPATH` in `.bashrc`:

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

Technique 2: add folder to `sys.path` in the program:

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

Technique 3: move the module file in a directory that Python already searches for libraries.