Ch.4: User input and error handling

Ole Christian Lingjærde, Dept of Informatics, UiO

15 September 2017 (PART 2)

Today's agenda

- A small quiz
- Short recapitulation of `eval` and `exec`.
- Live-programming of exercises 3.7, 4.1, 4.2, 4.3
- Writing data to file; exceptions; creating modules

Quiz 1

Which of these prints out the number 0?

- `x = [i for i in range(10)]; print(x[0])`
- `x = [i for i in range(1,10)]; print(x[0])`
- `x = [0, 23, 63]; y = x[-2]; print(y)`
- `x = [0, 23, 63] + [0, 33]; print(x[3])`
- `v = 110\%55; print('%d' % v)`
- `eps = 1e-16; x = (1+eps)-1; print('%g' % x)`
- `s = '3.0423'; print(s[2:2])`

Quiz 2

Suppose we have defined the functions below in Python. How would you test these functions (i.e. what special cases would you test them for)?

- A function `f(x)` to calculate $\exp(\cos(x))$ by a series expansion.
- A function `solve(f)` to find an `x` such that `f(x)=0`
- A function `solve2(f)` to find all `x` such that `f(x)=0`
- A function `h(f)` to calculate the definite integral $\int_0^1 f(x)\,dx$.
- A function `h(f,g)` to calculate the maximal distance between the functions `f(x)` and `g(x)` for any `x` in the interval `[0,1]`.

Short recap of the function eval

Whenever Python encounters an expression during program execution, it is *evaluated* to determine the value. Example:

```
x = 1.5 * math.sin(0.5)
x = x**2 - 3
a = [3,4,6] + [2,3]
s = 'Result: %g' % 5.0
```

An equivalent way of writing these examples is to put quotes around the expressions and call the function `eval`:

```
x = eval("1.5 * math.sin(0.5)")
x = eval("x**2 - 3")
a = eval("[3,4,6] + [2,3]")
s = eval("'Result: %g' % 5.0")
```

In the fourth line, we must use a different type of quotes than in the original expression (otherwise, Python gets confused).

The result of `eval(expr)` depends on the context

It is important to understand that the value produced by `eval(expr)` can depend on variables and functions defined *outside* the expression `expr`. Example: suppose we have

```
expr = 'f(x-1) + f(x)'
```

Case A

```
f = lambda x: x**2
x = 10
res = eval(expr)
print('Result: %g' % res)  # Printout: 'Result: 181'
```

Case B

```
f = lambda x: x
x = 5
res = eval(expr)
print('Result: %g' % res)  # Printout: 'Result: 9'
```

Making an expression into a function

We can convert an expression into a function if we know the name of the variable used in the expression. Example:

```
# Define an expression
expr = 'x + 3*x - 5*x**2'     # Name of variable is x

# Convert expression into a function of x
def f(x):
    res = eval(expr)
    return res

# Use the function
print('f(%g) = %g' % (0.5, f(0.5)))
```

Question: what would have happened here if the variable used in the expression was actually `y` rather than `x`?

If we use the wrong variable name

```
# Define an expression
expr = 'y + 3*y - 5*y**2'     # Name of variable is y

# We try to convert the expression into a function of x
def f(x):
      res = eval(expr)
      return res

# Use the function
print('f(%g) = %g' % (0.5, f(0.5)))
```

```
Terminal> python myprog.py
NameError                                 Traceback (most recent call
<ipython-input-9-303f7906c038> in <module>()
      8
      9 # Use the function
---> 10 print('f(%g) = %g' % (0.5, f(0.5)))
<ipython-input-9-303f7906c038> in f(x)
      4 # We try to convert the expression into a function of x
      5 def f(x):
----> 6       res = eval(expr)
      7       return res
      8
<string> in <module>()
NameError: name 'y' is not defined
```

Why is `eval` useful?

- It allows us to build *new expressions* during program execution and then have them evaluated.
- We can even read expressions from the user, or from file, and have them evaluated in the program.

Example 1: find maximal value of expression

Task: write a Python program `maxval.py` that takes an expression E(x) as command-line argument and finds the maximal value of the expression on the interval [0,1].

maxval.py

```
import sys
from math import *
expr = sys.argv[1]
xval = [float(i)/1000 for i in range(0,1001)]
yval = [eval(expr) for x in xval]
print('Maximal value on [0,1]: %5.2f' % max(yval))
```

Running the program

```
Terminal> python maxval.py "log(x+1)*cos(x)"
Maximum value on [0,1]:  0.41
```

Example 2: print table of expression values

Task: write a Python program `ftable.py` that takes an expression E(x) and a positive number as command-line arguments and prints the value of the expression in the n points 1/n, 2/n, ..., n/n.

ftable.py

```
import sys
from math import *
expr = sys.argv[1]
n = int(sys.argv[2])
xval = [float(i)/n for i in range(1,n+1)]
yval = [eval(expr) for x in xval]
for x,y in zip(xval,yval):
    print("%5.2f %5.2f" % (x,y))
```

Running the program

```
Terminal> python ftable.py "log(x)*cos(x)" 5
 0.20 -1.58
 0.40 -0.84
 0.60 -0.42
 0.80 -0.16
 1.00  0.00
```

Example 3: perform numerical differentiation

Task: write a Python program `diff.py` that takes an expression f(x) and a value x0 as command-line arguments and finds the derivative f'(x0) numerically, using the formula

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h} \quad (h \text{ small})$$

diff.py

```
import sys
from math import *
expr = sys.argv[1]
x0 = float(sys.argv[2])

def f(x):
    return eval(expr)

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

print("f(x)=%s ==> f\'(%g)=%5.2f" % (expr, x0, der(f,x0)))
```

Running the program

```
Terminal> python diff.py "exp(x)*sin(x)" 3.4
f(x)=exp(x)*sin(x) ==> f'(3.4)=-36.63

Terminal> python diff.py "tanh(x)" 1.5
f(x)=tanh(x) ==> f'(1.5)= 0.18
```

Short recap of the function exec

The function exec is analogous to eval, except that it executes a string consisting of program statements, rather than evaluating a single expression. Suppose we have:

```
x = 1.5 * math.sin(0.5)
x = x**2 - 3
a = [3,4,6] + [2,3]
s = 'Result: %g' % 5.0
```

An equivalent way of doing this is to make a string of all the statements and then call the function exec:

```
code = """
x = 1.5 * math.sin(0.5)
x = x**2 - 3
a = [3,4,6] + [2,3]
s = 'Result: %g' % 5.0
"""
exec(code)
```

Note the use of triple quotes here, which is required to define a string spanning multiple lines.

Exercise 3.7

Evaluate a sum and write a test function

- Write a Python function sum_1k(M) that returns the sum $s = \sum_{k=1}^{M} \frac{1}{k}$.
- Compute s for the case $M = 3$ by hand and write another function test_sum_1k() that calls sum_1k(3) and checks that the answer is correct.

Hint: We recommend that test_sum_1k follows the conventions of the pytest and nose testing frameworks as explained in Sects. 3.3.3 and 3.4.2 (see also Sect. H.9).

Filename: sum_func.

Exercise 4.1

Make an interactive program

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

Filename: f2c_qa.

Exercise 4.2

Read a number from the command line

Modify the program from Exercise 4.1 such that the Fahrenheit temperature is read from the command line.

Filename: f2c_cml.

Exercise 4.3

Read a number from a file

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

```
4.12 Exercises 217
Temperature data
---------------
Fahrenheit degrees: 67.2
```

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

Filename: f2c_file_read.

More about files

On the most fundamental level, a file is simply a long sequence of bits (0's or 1's):

10100100010100001001 0.....

To mean anything to a human, this sequence must be *interpreted* in some way. One way would be to consider chunks of eight bits at a time:

10100100 01010000 10010.....

Each chunk (= byte) can have $2^8 = 256$ different values.
We may decide that every time we see a particular sequence of eight bits (e.g. 10100100) we interpret it as a particular letter in the alphabet, or as some other symbol. This is the basic idea behind *text files*.

Why binary computers?

- Physically, it is more robust to store data using 0's and 1's rather than with more than two levels. Each bit can be stored using for example an electrical switch (on/off), two levels of light intensity (high/low), two distinct voltages (high/low).
- Imagine what could happen if we instead stored numbers between 0 and 99 as hundred different voltages (with imperfect equipment).
- It *is* possible to build computers based on more than two distinct states. The first modern, electronic ternary computer (three states -1,0,1) Setun was built in 1958 in the Soviet Union at the Moscow State University by Nikolay Brusentsov.
- There even exist *analog computers* which use the whole continuum of physical states (e.g. electrical voltages).

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')`.

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('%14.8f' % column)
    outfile.write('\n')
outfile.close()
```

Resulting file:

```
    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
```

Error handling

Error handling is an important topic in programming. It concerns how the program should react when something goes wrong during program execution. Things that may go wrong include e.g.:

- The user gives wrong input, for example too few command-line arguments or a value of wrong type.
- The program cannot open a file, because the file is missing or is read-protected.
- The program attemps to read a file that is wrongly formatted.

Three strategies

- **Preemptive action:** Insert check points in the program to identify error situations before they actually happen. Example: check if `x` is a number before attempting `res = x**2`.
- **Clean up afterwards:** Wait until errors happen and then take action to clean up the mess. For this to work, we have to "catch" the error before it leads to a halt in execution.
- **Do nothing:** When errors occur, let the runtime system take care of it. This essentially means that program execution stops and a standard error message is printed.

Why do anything at all?

It is tempting to do nothing to handle error situations, because error handling means more program code and more complex programs. However, there are good reasons to handle errors explicitly:

- Programs to be used by others are not likely to become popular if they crash with unintelligible error messages.
- Even during program development you may find that catching errors and giving a simple-to-understand error message helps identifying what went wrong.
- A sudden stop of program execution may result in loss of data, or corrupted files. Catching an error allows you to clean up a bit before the program stops.

Example: an error situation

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

A user can easily use the program in a wrong way:

```
Terminal> python celsius2fahrenheit.py
Traceback (most recent call last):
  File "c2f_cml.py", line 2, in ?
    C = float(sys.argv[1])
IndexError: list index out of range
```

What happened?

- The user forgot to provide a command-line argument
- `sys.argv` has then only one element, `sys.argv[0]`, which is the program name (`celsius2fahrenheit.py`)
- `sys.argv[1]` is non-existing and leads to `IndexError`

Example cont'd: preemptive action

To avoid a runtime error with a system error message, we test if the user has provided the required number of arguments:

```
# Program celsius2fahrenheit.py

import sys
if len(sys.argv) < 2:
    print('Missing argument: degrees Celcius')
    sys.exit(1)      # Abort program execution
F = 9.0*C/5 + 32
print('%gC is %.1fF' % (C, F))
```

This time, starting the program without giving a command-line argument results in an easily understandable error message:

```
Terminal> python celsius2fahrenheit.py
Missing argument: degrees Celcius
```

Handling errors by cleaning up afterwards

Normally, a runtime error immediately stops program execution. We can avoid this by putting the code inside a `try-except` block. This allows us to decide ourselves what action to take in case of an error (= exception).

Prototype for testing exceptions

```
try:
    <here we put the code that may fail>
except:
    <we come here only if an error occurs>
```

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

Example cont'd: cleaning up afterwards

This time, we try to read the variable C from the command-line. If it fails, we tell the user and abort execution:

```
# Program celsius2fahrenheit.py

import sys
try:
    C = float(sys.argv[1])
except:
    print('Missing argument: degrees Celcius')
    sys.exit(1)  # Abort the program
F = 9.0*C/5 + 32
print('%gC is %.1fF' % (C, F))
```

Execution:

```
Terminal> python celsius2fahrenheit.py
Missing argument: degrees Celcius

Terminal> python celsius2fahrenheit.py 21C
Missing argument: degrees Celcius
```

Testing for specific types of exceptions

It is good programming style to test for specific exceptions:

```
try:
    C = float(sys.argv[1])
except IndexError:
    print('Missing argument: degrees Celcius')
```

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.

Prototype for testing specific exceptions

```
try:
    <here we put the code that may fail>
except <exceptiontype 1>:
    <handle exception of type 1>
except <exceptiontype 2>:
    <handle exception of type 2>
...
```

Example cont'd: test for specific exception types

```
# Program celsius2fahrenheit.py

import sys
try:
    C = float(sys.argv[1])
except IndexError:
    print('Missing argument: degrees Celcius')
    sys.exit(1)  # Abort execution
except ValueError:
    print('Argument is not a number')
    sys.exit(1)
F = 9.0*C/5 + 32
print('%gC is %.1fF' % (C, F))
```

Executions:

```
Terminal> python celsius2fahrenheit.py
Missing argument: degrees Celcius

Terminal> python celsius2fahrenheit.py 21C
Argument is not a number
```

Raising our own exceptions

Instead of just letting Python raise exceptions, we can raise our own and tailor the message to the problem at hand. The basic syntax is `raise ExceptionType(message)`.

Example:

```
import sys
def read_C():
    try:
        C = float(sys.argv[1])
    except IndexError:
        raise IndexError('Celsius degrees must be supplied')
    except ValueError:
        raise ValueError('Degrees must be a number')
    if C < -273.15:
        raise ValueError('Temperature is outside range')
    return C

try:
    C = read_C()
except (IndexError, ValueError) as e:
    print(e); sys.exit(1)
```

Running the program

```
Terminal> python celsius2fahrenheit.py
Celsius degrees must be supplied

Terminal> python celsius2fahrenheit.py 21C
Degrees must be a number

Terminal> python celsius2fahrenheit.py -500
Temperature is outside range
```

Modules in Python

If we collect several functions in a single file, we have a module. Modules are useful for collecting related functions and data in one place. Modules are easily reused in other programs.

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

Example: creating your own module

Consider these formulas for computing with interest rates:

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

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

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

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

$$\tag{5}$$

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

We want to make a module with these four functions.

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)
```

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 'Money has doubled after %.1f years' % years
```

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
    print 'A=%g (%g) A0=%g (%.1f) n=%d (%d) p=%g (%.1f)' % \
          (present_amount(A0, p, n), A,
           initial_amount(A, p, n), A0,
           days(A0, A, p), n,
           annual_rate(A0, A, n), p)
```

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()
```

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.

Summary of reading from the keyboard and command line

Question and answer input:

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

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

Command-line input:

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

Recall: `sys.argv[0]` is the program name

Summary of reading options-value pairs

--option value pairs with the aid of argparse:

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

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

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

```
Terminal> program prog.py --parameter_1 2.1 --p2 -9
```

Summary of eval and exec

Evaluating string expressions with `eval`:

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

Executing strings with Python code, using `exec`:

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

Summary of exceptions

Handling exceptions:

```
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>
...
```

Raising exceptions:

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

Summary of file reading and writing

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

# 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

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

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