Ch.4: User input and error handling

Ole Christian Lingjærde, Dept of Informatics, UiO

13 September 2017

Today's agenda

- Short recapitulation from last week
- Live-programming of exercises 2.19, 2.20, 2.21
- Reading input from the user and from files

Writing test functions

Unit testing

A crucial part of programming is to test that things work properly. One way to do this is to test each unit (= function) separately.

Test strategy

- For each function we define, we also write a *test function* to see if the function at least works properly in special cases (where we know the answer).
- Each test function is designed to stop and print an error message if the function being tested does not work properly.
- We run all the test functions, and if no error message is printed, we say that the functions pass the test.

How to stop execution and write an error message

To stop program execution and give an error message, we can use the assert statement. This has the general form

```
    assert condition, message    # Gives error if condition is False
```

Example:

```
>>> x = 5
>>> assert x > 0, "x should be positive"   # Nothing happens
>>> x = -5
>>> assert x > 0, "x should be positive"   # Generates error message
Traceback (most recent call last):

  File "<ipython-input-30-c680011d20e2>", line 1, in <module>
    assert x > 0, "x should be positive"

AssertionError: x should be positive
```

Example: test one special case

```python
def sum3(a):                 # Find sum of every 3rd element in a
    res = sum([a[i] for i in range(0,len(a),3)])
    return res

def test_sum3():             # Associated test function
    """Call sum3(a) to check that it works."""
    a = [0,1,2,3,4,5]        # Some chosen input value
    expected = 3             # What the output should be
    computed = sum3(a)
    success = (computed == expected) # Did the test pass?
    message = 'computed %s, expected %s' % (computed, expected)
    assert success, message
```

- Note 1: the last statement in the test function runs silently if the test was a success. Only if the test fails will the program stop and the message be printed out on screen.
- Note 2: observe the name of the test function: `test_` followed by the name of the function to be tested.

Example: test several special cases

```
def sum3(a):                # Find sum of every 3rd element in a
    res = sum([a[i] for i in range(0,len(a),3)])
    return res

def test_sum3():            # Associated test function
    """Call sum3(a) to check that it works."""
    inputs = [[6], [6,1], [6,1,2], [6,1,2,3]]
    answers = [6, 6, 6, 9]
    for a, expected in zip(inputs, answers):
        computed = sum3(a)
        success = (computed == expected)
        message = '%s != %s' % (computed, expected)
        assert success, message
```

Recall that zip(a, b) creates tuples (a[i],b[i]):

```
>>> zip(inputs, answers)
>>> [([6], 6), ([6, 1], 6), ([6, 1, 2], 6), ([6, 1, 2, 3], 9)]
```

We are only told if something does *not* work

A peculiar property of a test function is that it *will run silently* with no output if all tests pass. On the other hand, if a test fails, `assert` will raise an `AssertionError`.

Rules for test functions:

- Name begins with `test_`
- No arguments
- Must have an `assert success` statement, where `success` is `True` if the test passed and `False` otherwise (`assert success, msg` prints `msg` on failure)

The optional `msg` parameter writes a message if the test fails.

Performing all tests automatically

- Test frameworks, like `nose` and `pytest`, can automatically run *all* your test functions (in a folder tree) and report if any bugs have sneaked in
- Test functions are a very well established standard

```
Terminal> py.test -s .
Terminal> nosetests -s .
```

We recommend `py.test` - it has superior output.

Advice

If you have not done this already, read very carefully Section 3.4.2 in the text book (Numerical integration by Simpson's rule) which provides an example of how to perform testing in practice.
This example is also found in the previous lecture (8 Sept 2017).

Exercise 2.19

Explore round-off errors from a large number of inverse operations

```
from math import sqrt
for n in range(1, 60):
    r = 2.0
    for i in range(n):
        r = sqrt(r)
    for i in range(n):
        r = r**2
    print('%d times sqrt and **2: %.16f' % (n, r))
```

- Explain with words what the program does.
- Then run the program. Round-off errors are here completely destroying the calculations when n is large enough!
- Investigate the case when we come back to 1 instead of 2 by fixing an n value where this happens and printing out r in both for loops over i.
- Can you now explain why we come back to 1 and not 2?

Filename: repeated_sqrt.

Exercise 2.20

Explore what zero can be on a computer

Type in the following code and run it:

```
eps = 1.0
while 1.0 != 1.0 + eps:
    print('...............', eps)
    eps = eps/2.0
print('final eps: ', eps)
```

- Explain with words what the code is doing, line by line.
- Then examine the output. How can it be that the "equation" 1 != 1 + eps is not true? Or in other words, that a number of approximately size 10^{-16} (the final eps value when the loop terminates) gives the same result as if eps were zero?

Filename: machine_zero.

Exercise 2.21

Compare two real numbers with a tolerance

Run the following program:

```
a = 1/947.0 * 947
b = 1
if a != b:
    print('Wrong result!')
```

The lesson learned from this program is that one should never compare two floating-point objects directly using `a == b` or `a != b`, because round-off errors quickly make two identical mathematical values different on a computer. A better result is to test if `abs(a - b) < tol`, where `tol` is a very small number. Modify the test according to this idea.

Filename: compare_floats.

Obtaining input data

A considerable part of most programming languages concerns ways of transferring data from the environment into a program, and from the program to the environment. Examples:

- An ATM exchanges info with user and bank
- A web browser exchanges info with user and the Internet
- An airplane autopilot may exchange info with satellites, ground stations, the pilot, and the plane's navigation controls.

Obtaining input data

Input data can be given:

- when we write the program (hard-coded input)
- when we start the program (command-line arguments)
- when the program is running (user interaction, file reading)

Hard-coded input

Example:

```python
# Here the input data is hard-coded as initial values:
a = 0.43
b = 6.233
c = 0.34552
d = 1.64
x = 3.5

def p(x):
    res = a + b*x + c*x**2 + d*x**3
    return res

print('p(%g) = %g' % (x, p(x)))
```

The main drawback with hard-coded input is the lack of flexibility. If we wish to change the input, we have to change the program itself which is often not acceptable in real applications.

Command line input

Command line input allows the user to supply input values at the time the program is started. This input is given as arguments (words) written after the program name.

Examples of 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.

Command line arguments provide more flexibility than hard-coded input. However, it is still quite rigid, since no event occurring *after* start of program execution can be taken into account.

Programming with command-line input

Consider the following program celsius2fahrenheit.py:

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

which we run by the command python celsius2fahrenheit.py. To make the variable C a command-line argument, we replace the above code with:

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

and run it by the command python celsius2fahrenheit.py 21.

Programming with multiple command-line inputs

Starting a program, we can add any number of words at the end of the command line, and each becomes a command-line argument. Inside the program, they are all available in the list `sys.argv`.

Example: suppose `mult.py` is defined as:

```python
import sys
x = float(sys.argv[1])
y = float(sys.argv[2])
z = x*y
print('%g * %g = %g' % (x, y, x*y))
```

Using the program:

```
Terminal> python mult.py 3.1415 0.2354
3.1415 * 0.2354 = 0.739509

Terminal> python mult.py
Traceback (most recent call last):
  File "mult.py", line 6, in <module>
    x = float(sys.argv[1])
IndexError: list index out of range
```

Command-line inputs are strings in Python

- All command-line arguments in the list `sys.argv` are strings
- Use type conversion if other types (e.g. floats) are desired.
- To treat multiple words as a single command-line argument, surround them by quotes as in `"string with blanks"`.

Example: suppose `printargs.py` is defined as:

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

Using the program (notice the first list element):

```
Terminal> python printargs.py a b c
['printargs.py', 'a', 'b', 'c']

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

User interaction input

User interaction allows the program to receive input from the user at any time during program execution. Here we focus on user interaction in the command window.

Examples of user interaction input:

```
Terminal> python celsius2fahrenheit.py
Terminal> Degrees Celcius = ? 21
Terminal> ==> Degrees Fahrenheit: 69.8
```

User interaction provides maximal flexibility for user input of data. However, it demands more of the user than hard-coded and command-line input. The user must be present to answer questions at the program's demand, which is not always practical.

Programming with user interaction

Consider the following program `celsius2fahrenheit.py`:

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

To let the user set the variable `C`, we replace the above code with:

```
C = float(input('Degrees Celcius = ? '))
F = 9.0*C/5 + 32
print('==> Degrees Fahrenheit: %g' % F)
```

Another example: print all primes up until n

```python
def isPrime(k):
    for i in range(2,k):
        if k%i == 0:
            return False
    return True

n = int(input('n = ? '))
primes = [i for i in range(2,n+1) if isPrime(i)]
print(primes)
```

Using the program:

```
Terminal> python printprimes.py
n = ? 100
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43,
47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]
```

Timing the user interaction

If a program is expected to run for a long time, one should strongly consider performing all the user interaction close to the start and/or end of program execution. This is less demanding on the user.

Examples:

- A slow installation program should be able to run smoothly without continuous watch by the user
- A dishwasher should not require input from the user multiple times during operation
- An equation solver should not require additional parameters from the user in the middle of an hour-long computation

Not always possible to follow (e.g. errors detected during execution).

Reading data from files

We frequently want to read data on files into a program. Here we focus on reading simple text files, although Python can also read files with other formats. Three steps are involved:

- Opening the file
- Reading its content
- Closing the file

Prototype pattern for file reading

```
infile = open('filename.txt', 'r')  # Opening the file
<read file content>                 # Reading its content
infile.close()                      # Closing the file
```

Reading the file content

Even simple text files can have a variety of formats, e.g.

- A single number on each line
- Multiple numbers on each line (separated by blanks)
- Multiple data types present on each line (separated by blanks)
- Rows with different formatting

Reading the content of a file must therefore be custom-made to the specific format of the file in question.

Three different approaches to read a file's content:

- Reading one line at a time: `infile.readline()`
- Reading the whole file into a list: `infile.readlines()`
- Reading the whole file into a string: `infile.read()`

Method A: reading one line at a time

We can use `readline()` to read a file one line at a time. When there is nothing more to read, the function returns `False`.

Example:

```
infile = open('data.txt', 'r')
line = infile.readline()
while line:
    print(line)      # Do something with line
infile.close()
```

Alternative (not using `readline()`):

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

Method B: reading everything into a list

We can use `readlines()` to read the entire file into a list, where each list element contains a line in the file.

Example 1:

```
infile = open('data.txt', 'r')
lines = infile.readlines()       # Reads entire file into a list
for s in lines:                  # Go through list
    print(s)                     # Print one line
infile.close()
```

Example 2:

```
>>> infile = open('data.txt', 'r')
>>> lines = infile.readlines()   # Read all lines
>>> lines
['Line 1.\n', 'Line 2.\n', 'Line 3.\n', 'Line 4.\n']
>>> infile.readline()  # no more to read
''
```

Method C: reading everything into a string

We can use `read()` to read the entire file into a string. To split the string into individual words, we can use `split()`.

Example:

```
infile = open('data.txt', 'r')
text = infile.read()        # Reads entire file into a string
words = text.split()        # Split text into words
infile.close()
```

By default, `text.split()` splits the text by arbitrary strings of whitespace characters (space, tab, newline, return, formfeed). You can also specify an arbitrary split string `s` using `text.split(s)`.

Example: file with mixed contents

File with data about rainfall:

```
Average rainfall in mm in Rome (1782-1970):
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 this file?

Example (cont'd)

First decide on a strategy

- The first line is a title and should be read separately. We decide in this case to simply skip this line (i.e. reading it, but not saving it).
- The next lines each consist of two words separated by whitespace. We can read each line with `readline()` and then split into two words and save them separately in lists `col1` and `col2` respectively.
- The last line is special and should be saved separately.
- After file reading, we move the appropriate parts of `col1` and `col2` into lists `name_month`, `rain_month` and `rain_year`.

Can we trust that all 12 months are present? Then we can use a for-loop to read all the monthly data. If not, we can simply read the file until there is nothing left.

Complete program for reading rainfall data

```python
# Open file for reading
infile = open('rainfall.dat', 'r')

# Read its content
infile.readline()      # Skip first line (not stored)
col1 = []; col2 = []   # Lists to store the data we read
for line in infile:
    words = line.split()
    col1.append(words[0])
    col2.append(float(words[1]))
infile.close()
name_month = col1[:-1] # Remove last element
rain_month = col2[:-1] # Remove last element
rain_year = col2[-1]   # The last element

# Print results
print('Monthly average rainfall:')
for name,rain in zip(name_month, rain_month):
    print('%s : %5.1f' % (name,rain))
print('Annual average rainfall: %5.1f' % rain_year)
```

The magic eval function turns a string into live code

`eval(s)` evaluates a string object `s` as if the string had been written directly into the program

```
>>> 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'>
```

Note: be careful with eval and string values

We want `r = 'math programming'`. Writing just

```
r = eval('math programming')
```

is the same as writing

```
r = math programming
```

which is an invalid expression and illegal syntax.
Remedy: must put the string inside quotes:

```
s = "'math programming'"
r = eval(s)                 # r becomes 'math programming'
```

With eval, a little program (add_input.py) can do much...

Program:

```
i1 = eval(input('Operand 1: '))
i2 = eval(input('Operand 2: '))
r = i1 + i2
print('%s + %s becomes %s\nwith value %s' % \
      (type(i1), type(i2), type(r), r))
```

We can 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]
```

This great flexibility also quickly breaks programs...

```
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(raw_input('operand 1: '))
  File "<string>", line 1, in <module>
NameError: name 'one' is not defined

Terminal> python add_input.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'
```

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
```

exec can be used for building a function at run-time

Embed user's formula in a Python function:

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

# 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(%g)=%g' % (x, y))
```

While the program is running, the user types a formula, which becomes a function, the user gives `x` values until the answer is `None`, and the program evaluates the function `f(x)`. Note: the programmer knows nothing about the user's choice of `f(x)` when she writes the program (!).

StringFunction: turn string formulas into Python functions

It is common for programs to read formulas and turn them into functions so we have made a special tool for this purpose:

```
>>> from scitools.std import StringFunction
>>> formula = 'exp(x)*sin(x)'
>>> f = StringFunction(formula)
>>> f(0)
0.0
>>> from math import pi
>>> f(pi)
2.83823929952166e-15
>>> print(str(f))
exp(x)*sin(x)
```

The function can also have parameters: $g(t) = Ae^{-at}\sin(\omega x)$

```
g = StringFunction('A*exp(-a*t)*sin(omega*x)',
    independent_variable='t', A=1, a=0.1, omega=pi, x=5)
print(g(1.2))
g.set_parameters(A=2, x=10)
print(g(1.2))
```