INF1100 Lectures, Chapter 4: Input Data and Error Handling

Hans Petter Langtangen

Simula Research Laboratory

University of Oslo, Dept. of Informatics

September 06, 2011
Example:

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

- Input data are explicitly set (hardcoded)
- To change input data, we need to *edit* the program.
- This is considered bad programming (because editing programs may easily introduce errors!)
- Rule: read input from user – do not edit a correct program.
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 in programs?

- Until now: hardcoded initialization of variables
- Ch. 3: Ask the user questions and read answers
- Ch. 3: Read command-line arguments
  
  Unix/DOS> python myprog.py arg1 arg2 arg3 ...
  Unix/DOS> rm -i -r temp projects univ

  Unix programs (rm, ls, cp, ...) make heavy use of command-line arguments, see e.g. man ls)

- Beyond scope: Read data from special fields in windows on the screen
  (most Windows, Mac and Linux programs work this way)
- Ch. 7: Read data from file
Sample program:

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

Idea: let the program ask the user a question "C=?", read the user's answer, assign that answer to the variable `C`

This is easy:

```python
C = raw_input('C=? ')             # C becomes a string
C = float(C)
F = (9./5)*C + 32
print F
```

Testing:

```
Unix/DOS> python c2f_qa.py
C=? 21
69.8
```
Read \( n \) from the keyboard:

\[
\begin{align*}
n &= \text{int}(\text{raw\_input}(\text{'}n=? \text{')})) \\
\text{for } i \text{ in range}(2, 2*n+1, 2): \\
&\quad \text{print } i \\
\# \text{ or:} \\
&\quad \text{print range}(2, 2*n+1, 2) \\
\# \text{ or:} \\
&\quad \text{for } i \text{ in range}(1, n+1): \\
&\quad \quad \text{print } 2*i
\end{align*}
\]
The magic `eval` function

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

- Example `r = eval('1+1')` is the same as `r = 1+1`

- Some other examples:
  ```python
  >>> r = eval('1+2')
  >>> r
  3
  >>> type(r)
  <type 'int'>
  >>> r = eval('[1, 6, 7.5]')
  >>> r
  [1, 6, 7.5]
  >>> type(r)
  <type 'list'>
  ```
Task: write \( r = \text{eval}(\ldots) \) that is equivalent to
\[
r = '\text{math programming}'
\]

Must use quotes to indicate that 'math programming' is string, plus extra quotes:
\[
\begin{align*}
r &= \text{eval}(''\text{'math programming''}') \\
# or \\
r &= \text{eval}(''\text{''math programming''}'')
\end{align*}
\]

What if we forget the extra quotes?
\[
r = \text{eval}('\text{math programming}')
\]

is the same as
\[
r = \text{math programming}
\]

but then Python thinks \text{math} and \text{programming} are two variables...combined with wrong syntax
This program adds two input variables:

```python
i1 = eval(raw_input('operand 1: '))
i2 = eval(raw_input('operand 2: '))
r = i1 + i2
print '%s + %s becomes %s
with value %s' % (type(i1), type(i2), type(r), r)
```

We can add integer and float:

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

We can add two lists:

```
Unix/DOS> 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]
```

Note: $r = i1 + i2$ becomes the same as

$r = [1,2] + [-1,0,1]$
This great flexibility also quickly breaks programs...

Unix/DOS> 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

Unix/DOS> 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

Unix/DOS> 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'
- `eval(s)` evaluates an *expression* `s`
- `eval('r = 1+1')` is illegal because this is a statement, not only an expression (assignment statement: `variable = expression`)
- ...but we can use `exec` for complete statements:
  ```python
  statement = 'r = 1+1'  # store statement in a string
  exec(statement)
  print r
  ```
  will print 2
- For longer code we can use multi-line strings:
  ```python
  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
  ```
What can exec be used for?

- **Build code at run-time, e.g., a function:**

  ```python
  formula = raw_input('Write a formula involving x: '
  code = """
  def f(x):
      return %s
  """ % formula
  exec(code)
  
  x = 0
  while x is not None:
      x = eval(raw_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 f(x)!
It is common for programs to read formulas and turn them into functions so we have made a special tool for this purpose:

```python
>>> from scitools.StringFunction import StringFunction
>>> formula = 'exp(x)*sin(x)'
>>> f = StringFunction(formula)
>>> f(0)
0.0
>>> f(pi)
2.8338239229952166e-15
```

The function can have parameters: 

\[ g(t) = Ae^{-at} \sin(\omega x) \]

```python
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)
```
Consider again our Celsius-Fahrenheit program:
\[
C = 21; \ F = (9.0/5)*C + 32; \ \text{print } F
\]

Now we want to provide \( C \) as a command-line argument after the name of the program when we run the program:

Unix/DOS> python c2f_cml_v1.py 21
69.8

Command-line arguments = "words" after the program name

The list \texttt{sys.argv} holds the command-line arguments:

```python
import sys
print 'program name:', sys.argv[0]
print '1st command-line argument:', sys.argv[1] # string
print '2nd command-line argument:', sys.argv[2] # string
print '3rd command-line argument:', sys.argv[3] # string
etc.
```

The Celsius-Fahrenheit conversion program:

```python
import sys
C = float(sys.argv[1])
F = 9.0*C/5 + 32
print F
```
Compute the current location of an object, 

\[ s(t) = s_0 + v_0 t + \frac{1}{2}at^2 \]

when \( s_0 \) (initial location), \( v_0 \) (initial velocity), \( a \) (constant acceleration) and \( t \) (time) are given on the command line.

How far away is the object at \( t = 3 \) s, if it started at \( s = 1 \) m at \( t = 0 \) with a velocity \( v_0 = 1 \) m/s and has undergone a constant acceleration of 0.5 m/s\(^2\)?

Unix/DOS> python location_cml.py 1 1 0.5 3
6.25

Program:

```python
import sys
s0 = float(sys.argv[1])
v0 = float(sys.argv[2])
a = float(sys.argv[3])
t = float(sys.argv[4])
s = s0 + v0*t + 0.5*a*t*t
print s
```
Command-line arguments are separated by blanks – use quotes to override this rule!

Let us make a program for printing the command-line args.:

```python
import sys; print sys.argv[1:]
```

Demonstrations:

Unix/DOS> python print_cml.py 21 string with blanks 1.3
['21', 'string', 'with', 'blanks', '1.3']

Unix/DOS> python print_cml.py 21 "string with blanks" 1.3
['21', 'string with blanks', '1.3']

Note that all list elements are surrounded by quotes, showing that command-line arguments are strings
Many programs, especially on Unix systems, take a set of command-line arguments of the form `--option value`

```
Unix/DOS> python location.py --v0 1 --t 3 --s0 1 --a 0.5
```

Can use the module `getopt` to help reading the data:

```
s0 = 0; v0 = 0; a = t = 1  # default values
import getopt, sys
options, args = getopt.getopt(sys.argv[1:], '',
      ['t=', 's0=', 'v0=', 'a='])

# options is a list of 2-tuples (option,value) of the option-value pairs given on the command line, e.g.,
# [(’--v0’, 1.5), (’--t’, 0.1), (’--a’, 3)]

for option, value in options:
    if option == 't':
        t = float(value)
    elif option == 'a':
        a = float(value)
    elif option == 'v0':
        v0 = float(value)
    elif option == 's0':
        s0 = float(value)
```
We can allow both long and shorter options, e.g. --t and --time, and --a and --acceleration

```python
options, args = getopt.getopt(sys.argv[1:], '',
    ['v0=', 'initial_velocity=', 't=', 'time=','s0=', 'initial_position=', 'v0=', 'acceleration='])

for option, value in options:
    if option in ('--t', '--time'):
        t = float(value)
    elif option in ('--a', '--acceleration'):
        a = float(value)
    elif option in ('--v0', '--initial_velocity'):
        v0 = float(value)
    elif option in ('--s0', '--initial_position'):
        s0 = float(value)
```
Summary of --option value pairs

- Advantage of --option value pairs:
  - can give options and values in arbitrary sequence
  - can skip option if default value is ok

- Command-line arguments that we read as `sys.argv[1]`, `sys.argv[2]`, etc. are like positional arguments to functions: the right sequence of data is essential!

- --option value pairs are like keyword arguments – the sequence is arbitrary and all options have a default value
Handling errors in input

- A user can easily use our program in a wrong way, e.g.,
  
  Unix/DOS> python c2f_cml_v1.py  
  Traceback (most recent call last):  
  File "c2f_cml_v1.py", line 2, in ?  
  C = float(sys.argv[1])  
  IndexError: list index out of range

  (the user forgot to provide a command-line argument...)

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

  ```python
  if len(sys.argv) < 2:
      print 'You failed to provide a command-line arg.!'  
      sys.exit(1)  # abort
  F = 9.0*C/5 + 32
  print '%gC is %.1fF' % (C, F)
  ```

- Execution:

  Unix/DOS> python c2f_cml_v2.py  
  You failed to provide a command-line arg.!
Exceptions instead of 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:

```python
try:
    <statements we indend 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.

Let’s see it in an example!
Try to read $C$ from the command-line, if it fails, tell the user and abort execution:

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

Execution:

Unix/DOS> python c2f_cml_v3.py
You failed to provide a command-line arg.!

Unix/DOS> python c2f_cml_v4.py 21C
You failed to provide a command-line arg.!
Testing for a specific exception

In

```
try:
    <statements>
except:
    <statements>
```

we jump to the except block for any exception raised when executing the try block

It is good programming style to test for specific exceptions:

```
try:
    C = float(sys.argv[1])
except IndexError:
    ...
```

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:

```
Unix/DOS>> python c2f_cml_tmp.py 21C
Traceback (most recent call last):
  File "tmp.py", line 3, in <module>
    C = float(sys.argv[1])
ValueError: invalid literal for float(): 21C
```
We can test for different exceptions:

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

F = 9.0*C/5 + 32
print '%gC is %.1fF' % (C, F)
```

Execution:

```
Unix/DOS> python c2f_cml_v3.py
No command-line argument for C!

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

Example:
```python
def read_C():
    try:
        C = float(sys.argv[1])
    except IndexError:
        raise IndexError('Celsius degrees must be supplied on the command line')
    except ValueError:
        raise ValueError('Celsius degrees must be a pure number, not "%s"' % sys.argv[1])
    # C is read correctly as a number, but can have wrong value:
    if C < -273.15:
        raise ValueError('C=%g is a non-physical value!' % C)
    return C
```
Calling the function in the main program:

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

Examples on running the program:

Unix/DOS> c2f_cml.py
Celsius degrees must be supplied on the command line

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

Unix/DOS> c2f_cml.py -500
C=-500 is a non-physical value!

Unix/DOS> c2f_cml.py 21
21C is 69.8F
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
- GUI is an advantage for novice users
- Experienced users often prefer command-line input (it’s much quicker and can be automated)
- The authors of a program are very experienced users...
- Programs with command-line or file input can easily be combined with each other, this is difficult with GUI-based programs

Assertion: command-line input will probably fill all your needs in university courses

But let’s have a look at GUI programming!
A graphical Celsius-Fahrenheit conversion program

- The Celsius degrees can be filled in as a number in a field.
- Clicking the "is" button computes the corresponding Fahrenheit temperature.
The GUI code

```python
from Tkinter import *
root = Tk()
C_entry = Entry(root, width=4)
C_entry.pack(side='left')
Cunit_label = Label(root, text='Celsius')
Cunit_label.pack(side='left')

def compute():
    C = float(C_entry.get())
    F = (9./5)*C + 32
    F_label.configure(text='\%g' % F)
compute = Button(root, text=' is ', command=compute)
compute.pack(side='left', padx=4)

F_label = Label(root, width=4)
F_label.pack(side='left')
Funit_label = Label(root, text='Fahrenheit')
Funit_label.pack(side='left')

root.mainloop()
```
We have used modules:

```python
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
```

A module is a 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, you should consider making a module containing these functions, and then you can do

```python
import mymodule
r = mymodule.my_resuable_function(arg1, arg2, arg3)
```

It's very easy to make modules: just collect the functions you want in a file, and that's a module!
Here are formulas for computing with interest rates:

\[ A = A_0 \left(1 + \frac{p}{360 \cdot 100}\right)^n \]

\[ A_0 = A \left(1 + \frac{p}{360 \cdot 100}\right)^{-n} \]

\[ n = \frac{\ln \frac{A}{A_0}}{\ln \left(1 + \frac{p}{360 \cdot 100}\right)} \]

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

\(A_0\): initial amount, \(p\): percentage, \(n\): days, \(A\): final amount

We want to make a module with these four functions
Python code for the four functions:

```python
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)
```
Collect the 4 functions in a file interest.py
Now interest.py is actually a module interest (!)
Here is a program that applies this module:

```python
# How long 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
```
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.

In our example we can have a verification as test block:

```python
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)
```
Alternative code organization:

```python
def _verify():
    # 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)
    print 'A=%g (%g)
A0=%g (%.1f)
n=%d (%d)
p=%g (%.1f)' % 
        (A_computed, A, A0_computed, A0,
         n_computed, n, p_computed, p)

if __name__ == '__main__':
    if len(sys.argv) == 2 and sys.argv[1] == 'verify':
        _verify()
```
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.

**Different techniques:**

- Add folder to `PYTHONPATH` in `.bashrc`:
  ```bash
  export PYTHONPATH=$PYTHONPATH:/Users/hpl/lib/python/mymods
  ```

- Add folder to `sys.path` in the program:
  ```python
  sys.path.insert(0, '/Users/hpl/lib/python/mymods')
  ```

- Add the module file in a directory that Python already know contains useful libraries.
• Give three parameters on the command line

• Let the program (test block) compute the fourth

```python
interest.py A0=2 A=1 n=1095    # compute p
```

• Use `exec` to execute the statements on the command line, find the missing parameter and call the appropriate function

```python
init_code = ''
for statement in sys.argv[1:]:
    init_code += statement + '\n'
exec(init_code)    # initialize input parameters

if 'A=' not in init_code:
    print 'A =', present_amount(A0, p, n)
elif 'A0=' not in init_code:
    print 'A0 =', initial_amount(A, p, n)
elif 'n=' not in init_code:
    print 'n =', days(A0, A, p)
elif 'p=' not in init_code:
    print 'p =', annual_rate(A0, A, n)
```

• Study the book for understanding all details of this module
**Question and answer input:**

```python
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:**

```python
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
--option value pairs with the aid of getopt:

import getopt
options, args = getopt.getopt(sys.argv[1:], '',
    ['parameter1=', 'parameter2=', 'parameter3=','p1=', 'p2=', 'p3='])  # shorter forms

# set default values:
parameter1 = ...
parameter2 = ...
parameter3 = ...

from scitools.misc import str2obj
for option, value in options:
    if option in ('--parameter1', '--p1'):
        parameter1 = eval(value)  # if not string
    elif option in ('--parameter2', '--p2'):
        parameter2 = value         # if string
    elif option in ('--parameter3', '--p3'):
        parameter3 = str2obj(value)  # if any object
Summary of eval and exec

- Evaluating string expressions with `eval`:
  ```python
  >>> x = 20
  >>> r = eval('x + 1.1')
  >>> r
  21.1
  >>> type(r)
  <type 'float'>
  ```

- Executing strings with Python code, using `exec`:
  ```python
  exec(""
  def f(x):
      return %s
  """ % sys.argv[1])
  ```
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\n        (’z=%s is negative - cannot do log(z)’ % z)
Nonlinear algebraic equations like

\[ x = 1 + \sin x \]

\[ \tan x + \cos x = \sin 8x \]

\[ x^5 - 3x^3 = 10 \]

are usually impossible to solve by pen and paper

Numerical methods can solve these easily

There are general algorithms for solving \( f(x) = 0 \) for "any" \( f \)

The three equations above correspond to

\[ f(x) = x - 1 - \sin x \]

\[ f(x) = \tan x + \cos x - \sin 8x \]

\[ f(x) = x^5 - 3x^3 - 10 \]
We shall learn about a method for solving $f(x) = 0$

A solution $x$ of $f(x) = 0$ is called a *root* of $f(x)$

Outline of the the next slides:
- Formulate a method for finding a root
- Translate the method to a precise algorithm
- Implement the algorithm in Python
The Bisection method

- Start with an interval \([a, b]\) in which \(f(x)\) changes sign.
- Then there must be (at least) one root in \([a, b]\).
- Halve the interval:
  - \(m = (a + b)/2\); does \(f\) change sign in left half \([a, m]\)?
  - Yes: continue with left interval \([a, m]\) (set \(b = m\)).
  - No: continue with right interval \([m, b]\) (set \(a = m\)).
- Repeat the procedure.

After halving the initial interval \([p, q]\) \(n\) times, we know that \(f(x)\) must have a root inside a (small) interval \(2^{-n}(q - p)\).

- The method is slow, but very safe.
- Other methods (like Newton’s method) can be faster, but may also fail to locate a root – bisection does not fail.
Solving $\cos \pi x = 0$: iteration no. 1

The Bisection method, iteration 1: [0.41, 0.82]
Solving \( \cos \pi x = 0 \): iteration no. 2

The Bisection method, iteration 2: [0.41, 0.61]
Solving $\cos \pi x = 0$: iteration no. 3

The Bisection method, iteration 3: [0.41, 0.51]
Solving $\cos \pi x = 0$: iteration no. 4

The Bisection method, iteration 4: [0.46, 0.51]
We need to translate the mathematical description of the Bisection method to a Python program.

An important intermediate step is to formulate a precise algorithm.

Algorithm = detailed, code-like formulation of the method.

```python
for i = 0, 1, 2, ..., n
    m = (a + b)/2 (compute midpoint)
    if f(a)f(m) ≤ 0 then
        b = m (root is in left half)
    else
        a = m (root is in right half)
    end if
end for

f(x) has a root in [a, b]
```
The algorithm can be made more efficient

- $f(a)$ is recomputed in each if test
- This is not necessary if $a$ has not changed since last pass in the loop
- On modern computers and simple formulas for $f(x)$ these extra computations do not matter
- However, in science and engineering one meets $f$ functions that take hours or days to evaluate at a point, and saving some $f(a)$ evaluations matters!
- Rule of thumb: remove redundant computations (unless the code becomes much more complicated, and harder to verify)
New, more efficient version of the algorithm

Idea: save $f(x)$ evaluations in variables

$$f_a = f(a)$$

for $i = 0, 1, 2, \ldots, n$

$$m = (a + b)/2$$

$$f_m = f(m)$$

if $f_a f_m \leq 0$ then

$$b = m \text{ (root is in left half)}$$

else

$$a = m \text{ (root is in right half)}$$

$$f_a = f_m$$

end if

end for

$f(x)$ has a root in $[a, b]$
How to choose \( n \)? That is, when to stop the iteration?

- We want the error in the root to be \( \epsilon \) or smaller.
- After \( n \) iterations, the initial interval \( [a, b] \) is halved \( n \) times and the current interval has length \( 2^{-n}(b-a) \). This is sufficiently small if:
  \[
  2^{-n}(b-a) = \epsilon \quad \Rightarrow \quad n = -\frac{\ln \epsilon - \ln(b-a)}{\ln 2}
  \]
- A simpler alternative: just repeat halving until the length of the current interval is \( \leq \epsilon \).
- This is easiest done with a while loop:
  ```python
  while b-a <= epsilon:
  ```
- We also add a test to check if \( f \) really changes sign in the initial interval \( [a, b] \).
$f_a = f(a)$

if $f_a f(b) > 0$ then
    error: $f$ does not change sign in $[a, b]$
end if

$i = 0$

while $b - a > \epsilon$:
    $i \leftarrow i + 1$
    $m = (a + b)/2$
    $f_m = f(m)$
    if $f_a f_m \leq 0$ then
        $b = m$ (root is in left half)
    else
        $a = m$ (root is in right half)
        $f_a = f_m$
    end if
end while

if $x$ is the real root, $|x - m| < \epsilon$
def f(x):
    return 2*x - 3  # one root x=1.5

eps = 1E-5
a, b = 0, 10

fa = f(a)
if fa*f(b) > 0:
    print f(x) does not change sign in [%g,%g].’ % (a, b)
    sys.exit(1)

i = 0  # iteration counter
while b-a > eps:
    i += 1
    m = (a + b)/2.0
    fm = f(m)
    if fa*fm <= 0:
        b = m  # root is in left half of [a,b]
    else:
        a = m  # root is in right half of [a,b]
        fa = fm

x = m  # this is the approximate root
def bisection(f, a, b, eps):
    fa = f(a)
    if fa*f(b) > 0:
        return None, 0

    i = 0  # iteration counter
    while b-a < eps:
        i += 1
        m = (a + b)/2.0
        fm = f(m)
        if fa*fm <= 0:
            b = m  # root is in left half of [a,b]
        else:
            a = m  # root is in right half of [a,b]
            fa = fm
    return m, i
If we put the `bisection` function in a file `bisection.py`, we automatically have a module, and the `bisection` function can easily be imported in other programs to solve \( f(x) = 0 \).

Verification part in the module is put in a "private" function and called from the module's test block:

```python
def _test():  # start with _ to make "private"
    def f(x):
        return 2*x - 3  # one root x=1.5

    eps = 1E-5
    a, b = 0, 10
    x, iter = bisection(f, a, b, eps)
    # check that x is 1.5

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

To the point of this lecture: get input!

- We want to provide an $f(x)$ formula at the command line along with $a$ and $b$ (3 command-line args)

- Usage:
  ```python
  python bisection_solver.py 'sin(pi*x**3)-x**2' -1 3.5
  ```

The complete application program:

```python
import sys
f_formula = sys.argv[1]
a = float(sys.argv[2])
b = float(sys.argv[3])
epsilon = 1E-6

from scitools.StringFunction import StringFunction
f = StringFunction(f_formula)

from bisection import bisection
root, iter = bisection(f, a, b, epsilon)
print 'Found root %g in %d iterations' % (root, iter)
```
import sys
try:
    f_formula = sys.argv[1]
    a = float(sys.argv[2])
    b = float(sys.argv[3])
except IndexError:
    print '%s f-formula a b [epsilon]' % sys.argv[0]
    sys.exit(1)

try:  # is epsilon given on the command-line?
    epsilon = float(sys.argv[4])
except IndexError:
    epsilon = 1E-6  # default value

from scitools.StringFunction import StringFunction
from math import *  # might be needed for f_formula
f = StringFunction(f_formula)
from bisection import bisection
root, iter = bisection(f, a, b, epsilon)
if root == None:
    print 'No root found'; sys.exit(1)
print 'Found root %g in %d iterations' % (root, iter)
Two examples: \( \tanh x = x \) and \( \tanh x^5 = x^5 \)

Can run a program for graphically demonstrating the method:

```
Unix/DOS> python bisection_plot.py "x-tanh(x)" -1 1
Unix/DOS> python bisection_plot.py "x**5-tanh(x**5)" -1 1
```

- The first equation is easy to treat
- The second leads to much less accurate results
- Why??? Run the demos!