Week 2

Sun Jun
with slides from Hans Petter Langtangen
Suppose we want to make a table of Celsius and Fahrenheit degrees:

<table>
<thead>
<tr>
<th>Temperature (°C)</th>
<th>Temperature (°F)</th>
</tr>
</thead>
<tbody>
<tr>
<td>-20</td>
<td>-4.0</td>
</tr>
<tr>
<td>-15</td>
<td>5.0</td>
</tr>
<tr>
<td>-10</td>
<td>14.0</td>
</tr>
<tr>
<td>-5</td>
<td>23.0</td>
</tr>
<tr>
<td>0</td>
<td>32.0</td>
</tr>
<tr>
<td>5</td>
<td>41.0</td>
</tr>
<tr>
<td>10</td>
<td>50.0</td>
</tr>
<tr>
<td>15</td>
<td>59.0</td>
</tr>
<tr>
<td>20</td>
<td>68.0</td>
</tr>
<tr>
<td>25</td>
<td>77.0</td>
</tr>
<tr>
<td>30</td>
<td>86.0</td>
</tr>
<tr>
<td>35</td>
<td>95.0</td>
</tr>
<tr>
<td>40</td>
<td>104.0</td>
</tr>
</tbody>
</table>

How can a program write out such a table?
We know how to make one line in the table:

\[
\begin{align*}
C &= -20 \\
F &= \frac{9.0}{5}C + 32 \\
\text{print } C, \ F
\end{align*}
\]

We can just repeat these statements:

\[
\begin{align*}
C &= -20; \quad F = \frac{9.0}{5}C + 32; \quad \text{print } C, \ F \\
C &= -15; \quad F = \frac{9.0}{5}C + 32; \quad \text{print } C, \ F \\
\ldots \\
C &= 35; \quad F = \frac{9.0}{5}C + 32; \quad \text{print } C, \ F \\
C &= 40; \quad F = \frac{9.0}{5}C + 32; \quad \text{print } C, \ F
\end{align*}
\]

Very boring to write, easy to introduce a misprint

When programming becomes boring, there is usually a construct that automates the writing

The computer is very good at performing repetitive tasks!

For this purpose we use *loops*
A while loop executes repeatedly a set of statements as long as a boolean condition is true

```python
while condition:
    <statement 1>
    <statement 2>
    ...
<first statement after loop>
```

- All statements in the loop must be indented!
- The loop ends when an unindented statement is encountered
The while loop for making a table

print '------------------' # table heading
C = -20 # start value for C
dC = 5 # increment of C in loop
while C <= 40: # loop heading with condition
    F = (9.0/5)*C + 32 # 1st statement inside loop
    print C, F # 2nd statement inside loop
    C = C + dC # last statement inside loop
print '------------------' # end of table line
The program flow in a while loop

- \( C = -20 \)
- \( dC = 5 \)

while \( C \leq 40 \):
  \[
  F = \frac{9.0}{5} \times C + 32
  \]
  print \( C \), \( F \)
  \( C = C + dC \)

Let us simulate the while loop by hand

First \( c \) is -20, \(-20 \leq 40\) is true, therefore we execute the loop statements

Compute \( F \), print, and update \( c \) to -15

We jump up to the while line, evaluate \( C \leq 40 \), which is true, hence a new round in the loop

We continue this way until \( c \) is updated to 45

Now the loop condition \( 45 \leq 40 \) is false, and the program jumps to the first line after the loop – the loop is over
An expression with value true or false is called a boolean expression

Examples: $C = 40$, $C \neq 40$, $C \geq 40$, $C > 40$, $C < 40$

- $C == 40$  # note the double ==, C=40 is an assignment!
- $C != 40$
- $C >= 40$
- $C > 40$
- $C < 40$

We can test boolean expressions in a Python shell:

```python
>>> C = 41
>>> C != 40
True
>>> C < 40
False
>>> C == 41
True
```
Combining boolean expressions

Several conditions can be combined with and/or:

while condition1 and condition2:
    ...

while condition1 or condition2:
    ...

Rule 1: \texttt{C1 and C2} is True if both \texttt{C1} and \texttt{C2} are True

Rule 2: \texttt{C1 or C2} is True if one of \texttt{C1} or \texttt{C2} is True

Examples:

```python
>>> x = 0; y = 1.2
>>> x >= 0 and y < 1
False
>>> x >= 0 or y < 1
True
>>> x > 0 or y > 1
True
>>> x > 0 or not y > 1
False
>>> -1 < x <= 0  # -1 < x and x <= 0
True
>>> not (x > 0 or y > 0)
False
```
Exercise

• Write a program, with a while-loop, that asks users to input 10 numbers (one each time) output the sum of the numbers so far once a number is entered.

enter a number: 5
sum is: 5

enter a number: 6
sum is 11

...
Sometimes we want to perform different actions depending on a condition.
Consider the function

\[
f(x) = \begin{cases} 
\sin x, & 0 \leq x \leq \pi \\
0, & \text{otherwise} 
\end{cases}
\]

In a Python implementation of \( f \) we need to test on the value of \( x \) and branch into two computations:

```python
def f(x):
    if 0 <= x <= pi:
        return sin(x)
    else:
        return 0
```

In general (the `else` block can be skipped):

```python
if condition:
    <block of statements, executed if condition is True>
else:
    <block of statements, executed if condition is False>
```
We can test for multiple (here 3) conditions:

```python
if condition1:
    <block of statements>
elif condition2:
    <block of statements>
elif condition3:
    <block of statements>
else:
    <block of statements>
<next statement>
```
Example on multiple branches:

\[ N(x) = \begin{cases} 
0, & x < 0 \\
x, & 0 \leq x < 1 \\
2 - x, & 1 \leq x < 2 \\
0, & x \geq 2
\end{cases} \]

```python
def N(x):
    if x < 0:
        return 0
    elif 0 <= x < 1:
        return x
    elif 1 <= x < 2:
        return 2 - x
    elif x >= 2:
        return 0
```
A common construction is

```python
if condition:
    variable = value1
else:
    variable = value2
```

This test can be placed on one line as an expression:

```python
variable = (value1 if condition else value2)
```

Example:

```python
def f(x):
    return (sin(x) if 0 <= x <= 2*pi else 0)
```
• If a non-Boolean expression is given as a condition (in if, while, for, etc.), it is evaluated and converted to True/False automatically.
  – ‘’’, None, [], 0 are converted to False
  – “None”, [1,2], 4 are converted to True
Exercise

• Write a program which takes two positive integers and returns their greatest common divisor.
So far, one variable has referred to one number (or string)

Sometimes we naturally have a collection of numbers, say degrees $-20, -15, -10, -5, 0, \ldots, 40$

Simple solution: one variable for each value

$C_1 = -20$
$C_2 = -15$
$C_3 = -10$
$\ldots$
$C_{13} = 40$

(stupid and boring solution if we have many values)

Better: a set of values can be collected in a list

$C = [-20, -15, -10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40]$ Now there is one variable, $c$, holding all the values
A list consists of elements, which are Python objects.

We initialize the list by separating elements with comma and enclosing the collection in square brackets:

\[ L_1 = [-91, 'a string', 7.2, 0] \]

Elements are accessed via an index, e.g. \[ L_1[3] \] (index=3)

List indices are always numbered as 0, 1, 2, and so forth up to the number of elements minus one.

```python
mylist = [4, 6, -3.5]
print(mylist[0])
4
print(mylist[1])
6
print(mylist[2])
-3.5
len(mylist) # length of list
3
```
Some interactive examples on list operations:

```python
>>> C = [-10, -5, 0, 5, 10, 15, 20, 25, 30]
>>> C.append(35)  # add new element 35 at the end
>>> C
[-10, -5, 0, 5, 10, 15, 20, 25, 30, 35]
>>> C = C + [40, 45]  # extend C at the end
>>> C
[-10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40, 45]
>>> C.insert(0, -15)  # insert -15 as index 0
>>> C
[-15, -10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40, 45]
>>> del C[2]  # delete 3rd element
>>> C
[-15, -10, 0, 5, 10, 15, 20, 25, 30, 35, 40, 45]
>>> del C[2]  # delete what is now 3rd element
>>> C
[-15, -10, 5, 10, 15, 20, 25, 30, 35, 40, 45]
>>> len(C)  # length of list
11
```
More examples in an interactive Python shell:

```python
>>> C.index(10) # index of the first element with value 10
3
>>> 10 in C # is 10 an element in C?
True
>>> C[-1] # the last list element
45
>>> C[-2] # the next last list element
40
>>> texfile, logfile, pdf = somelist
>>> texfile
'book.tex'
>>> logfile
'book.log'
>>> pdf
'book.pdf'
```
List Operations (part 4)

• Given \( x = [0,1,2,3,4,5] \), the following are true:
  \( x[0] == x[-6] == 0 \)
  \( x[5] == x[-1] == 5 \)
  \( x[6] \) and \( x[-7] \) give you error
  \( x[1:3] == [1,2] \)
  \( x[2:] == [2,3,4,5] \)
  \( x[:4] == [0,1,2,3] \)
  \( x[:] == [0,1,2,3,4,5] \)
  \( x + x == [0,1,2,3,4,5,0,1,2,3,4,5] == x*2 \)
Environments

• For any object,
  – type() gives its type
  – id() gives its memory address
  – its value can be obtained by evaluating the object.
  – “==” tests for equivalence of values
  – “is” tests for equivalence of ids

• Immutable objects: numbers, strings, tuples …
• Mutable objects: lists, dictionaries, …
• Learn to draw the environment diagram!
Shallow and Deep Copy

• Sharing = Saving = Out of Control
• Object copy

```python
import copy

# returns a shallow copy of x
copy.copy(x)

# returns a deep copy
copy.deepcopy(x)
```
Week 2: Cohort 2

Sun Jun

with slides from Hans Petter Langtangen
We have used many Python functions

- **Mathematical functions:**
  
  ```python
  from math import *
  y = sin(x)*log(x)
  ```

- **Other functions:**
  
  ```python
  n = len(somelist)
  ints = range(5, n, 2)
  ```

- **Functions used with the dot syntax (called methods):**
  
  ```python
  C = [5, 10, 40, 45]
  i = C.index(10)    # result: i=1
  C.append(50)
  C.insert(2, 20)
  ```

- **What is a function?** So far we have seen that we put some objects in and sometimes get an object (result) out
- **Next topic:** learn to write your own functions
Python functions

- Function = a collection of statements we can execute wherever and whenever we want
- Function can take input objects and produce output objects
- Functions help to organize programs, make them more understandable, shorter, and easier to extend
- Simple example: a mathematical function \( F(C) = \frac{9}{5}C + 32 \)
  ```python
def F(C):
    return (9.0/5)*C + 32
  
Functions start with `def`, then the name of the function, then a list of arguments (here `C`) – the function header
- Inside the function: statements – the function body
- Wherever we want, inside the function, we can ”stop the function” and return as many values/variables we want
A function does not do anything before it is called

Examples on calling the $F(C)$ function:

```python
a = 10
F1 = F(a)

F(C)
```

```python
temp = F(15.5)

print F(a+1)
```

```python
sum_temp = F(10) + F(20)
```

```python
Fdegrees = [F(C) for C in Cdegrees]
```

Since $F(C)$ produces (returns) a float object, we can call $F(C)$ everywhere a float can be used
Exercise

• Problem Set 2-1: Question 1 and 2
For loops

- We can visit each element in a list and process the element with some statements in a *for* loop.

**Example:**

```
degrees = [0, 10, 20, 40, 100]
for C in degrees:
    print 'list element:', C
    print 'The degrees list has', len(degrees), 'elements'
```

- The statement(s) in the loop must be indented!
- We can simulate the loop by hand:
- **First pass:** \( C \) is 0
- **Second pass:** \( C \) is 10 ...and so on...
- **Fifth pass:** \( C \) is 100

- Now the loop is over and the program flow jumps to the first statement with the same indentation as the `for C in degrees` line.
The table of Celsius and Fahrenheit degrees:

Celsius = [-20, -15, -10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40]

for C in Celsius:
    F = (9.0/5)*C + 32
    print C, F

The print C, F gives ugly output

Use printf syntax to nicely format the two columns:

print '%5d %5.1f' % (C, F)

Output:

-20  -4.0
-15   5.0
-10  14.0
-5   23.0
  0  32.0
  .....
 35  95.0
 40 104.0
The for loop

```python
for element in somelist:
    # process element
```

can always be transformed to a while loop

```python
index = 0
while index < len(somelist):
    element = somelist[index]
    # process element
    index += 1
```

Example: while version of the for loop on the previous slide

```python
Cdegrees = [-20, -15, -10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40]
index = 0
while index < len(Cdegrees):
    C = Cdegrees[index]
    F = (9.0/5)*C + 32
    print '%5d %5.1f' % (C, F)
    index += 1
```
Let us put all the Fahrenheit values also in a list:

```python
Cdegrees = [-20, -15, -10, -5, 0, 5, 10, 15, 20, 25, 30, 35, 40]
Fdegrees = []  # start with empty list
for C in Cdegrees:
    F = (9.0/5)*C + 32
    Fdegrees.append(F)  # add new element to Fdegrees

print F prints the list
[-4.0, 5.0, 14.0, 23.0, 32.0, 41.0, 50.0, 59.0, 68.0, 77.0, 86.0, 95.0, 104.0]
```
For loops usually loop over list values (elements):

```python
for element in somelist:
    # process variable element
```

We can alternatively loop over list indices:

```python
for i in range(0, len(somelist), 1):
    element = somelist[i]
    # process element or somelist[i] directly
```

- `range(start, stop, inc)` generates a list of integers `start`, `start+inc`, `start+2*inc`, and so on up to, but not including, `stop`
- `range(stop)` is the same as `range(0, stop, 1)`

```python
>>> range(3)  # = range(0, 3, 1)
[0, 1, 2]
>>> range(2, 8, 3)
[2, 5]
```
Say we want to add 2 to all numbers in a list:

```python
>>> v = [-1, 1, 10]
>>> for e in v:
...    e = e + 2
...      
...  >>> v
[-1, 1, 10]  # unaltered!
```

Explanation: inside the loop, `e` is an ordinary (int) variable, first time `e` becomes 1, next time `e` becomes 3, and then 12 – but the list `v` is unaltered.

We have to index a list element to change its value:

```python
>>> v[1] = 4  # assign 4 to 2nd element (index 1) in v
>>> v
[-1, 4, 10]
```

To add 2 to all values we need a for loop over indices:

```python
>>> for i in range(len(v)):
...    v[i] = v[i] + 2
...      
...  >>> v
[1, 6, 12]```
List comprehensions

- Example: compute two lists in a for loop

  ```python
  n = 16
  Cdegrees = []; Fdegrees = []  # empty lists
  for i in range(n):
    Cdegrees.append(-5 + i*0.5)
    Fdegrees.append((9.0/5)*Cdegrees[i] + 32)
  ```

- Python has a compact construct, called *list comprehension*, for generating lists from a for loop:

  ```python
  Cdegrees = [-5 + i*0.5 for i in range(n)]
  Fdegrees = [(9.0/5)*C + 32 for C in Cdegrees]
  ```

- General form of a list comprehension:

  ```python
  somelist = [expression for element in somelist]
  ```

- We will use list comprehensions a lot, to save space, so there will be many more examples
List Comprehension with Condition

• General form:
  somelist = [expr. for i in list if condition]

• Example: list = [1,-2,3,-4,5,-1]
  from math import sqrt
  sqrtlist = [sqrt(i) for i in list if i >= 0]
What if we want to have a for loop over elements in Cdegrees and Fdegrees?

We can have a loop over list indices:

```python
for i in range(len(Cdegrees)):
    print Cdegrees[i], Fdegrees[i]
```

Alternative construct (regarded as more "Pythonic"):

```python
for C, F in zip(Cdegrees, Fdegrees):
    print C, F
```

Example with three lists:

```python
>>> l1 = [3, 6, 1]; l2 = [1.5, 1, 0]; l3 = [9.1, 3, 2]
>>> for e1, e2, e3 in zip(l1, l2, l3):
...     print e1, e2, e3
...             
...     3 1.5 9.1
...     6 1 3
...     1 0 2
```

What if the lists have unequal lengths? The loop stops when the end of the shortest list is reached
Exercise

• Write a function even() that takes a list and return a new list containing all even numbers in the original list. Can you use list comprehension to do this?
FUNCTIONS AND BRANCHING
Local variables in Functions

Example: sum the integers from start to stop

```python
def sumint(start, stop):
    s = 0  # variable for accumulating the sum
    i = start  # counter
    while i <= stop:
        s += i
        i += 1
    return s
```

```
print sumint(0, 10)
sum_10_100 = sumint(10, 100)
```

- `i` and `s` are local variables in `sumint` – these are destroyed at the end (return) of the function and never visible outside the function (in the calling program); in fact, `start` and `stop` are also local variables

- In the program above, there is one global variable, `sum_10_100`, and two local variables, `s` and `i` (in the `sumint` function)

- Read Chapter 2.2.2 in the book about local and global variables!!
Recall the formula \( y(t) = v_0 t - \frac{1}{2} g t^2 \):

We can make a Python function for \( y(t) \):

```python
def yfunc(t, v0):
    g = 9.81
    return v0*t - 0.5*g*t**2
```

# sample calls:
y = yfunc(0.1, 6)
y = yfunc(0.1, v0=6)
y = yfunc(t=0.1, v0=6)
y = yfunc(v0=6, t=0.1)

Functions can have as many arguments as you like

When we make a call \( yfunc(0.1, 6) \), all these statements are in fact executed:

```python
t = 0.1  # arguments get values as in standard assignments
v0 = 6
g = 9.81
return v0*t - 0.5*g*t**2
```
The $y(t,v_0)$ function took two arguments

Could implement $y(t)$ as a function of $t$ only:

```python
>>> def yfunc(t):
...     g = 9.81
...     return v0*t - 0.5*g*t**2
... >>> yfunc(0.6)
... NameError: global name 'v0' is not defined
```

$v_0$ must be defined in the calling program program before we call `yfunc`

```python
>>> v0 = 5
>>> yfunc(0.6)
1.2342
```

$v_0$ is a global variable

Global variables are variables defined outside functions

Global variables are visible everywhere in a program

$g$ is a local variable, not visible outside of `yfunc`
Environments for Functions

• Example:

```python
da = 100
def f(x):
da = 5
    global a
    a = 5
    f(a)
    f(a)
```

```python
da = 100
def f(x):
da = a+1
```

```python
da = 100
def f(x):
da = a+1
```

```python
da = 100
def f(x):
da = a+1
```
Functions can return multiple values

Say we want to compute $y(t)$ and $y'(t) = v_0 - gt$:

```python
def yfunc(t, v0):
g = 9.81
y = v0*t - 0.5*g*t**2
dydt = v0 - g*t
return y, dydt
```

# call:
position, velocity = yfunc(0.6, 3)

Separate the objects to be returned by comma

What is returned is then actually a tuple

```python
>>> def f(x):
...     return x, x**2, x**4
...
>>> s = f(2)
>>> s
(2, 4, 16)
>>> type(s)
<type 'tuple'>
>>> x, x2, x4 = f(2)
```
Exercise

- Problem Set: Week 2, Question 8 and 10
The function

\[ L(x; n) = \sum_{i=1}^{n} \frac{1}{i} \left( \frac{x}{1+x} \right)^i \]

is an approximation to \( \ln(1 + x) \) for a finite \( n \) and \( x \geq 1 \)

Let us make a Python function for \( L(x; n) \):

```python
def L(x, n):
    x = float(x) # ensure float division below
    s = 0
    for i in range(1, n+1):
        s += (1.0/i)*(x/(1+x))**i
    return s
```

\[ x = 5 \]

from math import log as ln
print L(x, 10), L(x, 100), ln(1+x)
We can return more: also the first neglected term in the sum and the error $(\ln(1 + x) - L(x; n))$:

```python
def L2(x, n):
    x = float(x)
    s = 0
    for i in range(1, n+1):
        s += (1.0/i)*(x/(1+x))**i
    value_of_sum = s
    first_neglected_term = (1.0/(n+1))*(x/(1+x))**(n+1)
    from math import log
    exact_error = log(1+x) - value_of_sum
    return value_of_sum, first_neglected_term, exact_error

# typical call:
x = 1.2; n = 100
value, approximate_error, exact_error = L2(x, n)
```
Let us make a table of $L(x; n)$ versus the exact $\ln(1 + x)$.

The table can be produced by a Python function.

This function prints out text and numbers but do not need to return anything – we can then skip the final `return`.

```python
def table(x):
    print '\nx=%g, ln(1+x)=%g' % (x, log(1+x))
    for n in [1, 2, 10, 100, 500]:
        value, next, error = L2(x, n)
        print 'n=%-4d %-10g (next term: %8.2e ' \\
            'error: %8.2e)' % (n, value, next, error)
```

Output from `table(10)` on the screen:

```
x=10, ln(1+x)=2.3979
n=1 0.909091 (next term: 4.13e-01 error: 1.49e+00)
n=2 1.32231 (next term: 2.50e-01 error: 1.08e+00)
n=10 2.17907 (next term: 3.19e-02 error: 2.19e-01)
n=100 2.39789 (next term: 6.53e-07 error: 6.59e-06)
n=500 2.3979 (next term: 3.65e-24 error: 6.22e-15)
```
Consider a function without any return value:

```python
def message(course):
    print "%s is the greatest fun I’ve ever experienced\n    " % course
message(‘INF1100’)  # store the return value
```

None is a special Python object that represents an "empty" or undefined value – we will use it a lot later.

No return value implies that None is returned
Functions can have arguments of the form name=value, called keyword arguments:

```python
>>> def somefunc(arg1, arg2, kwarg1=True, kwarg2=0):
    print arg1, arg2, kwarg1, kwarg2

>>> somefunc('Hello', [1,2])  # drop kwarg1 and kwarg2
Hello [1, 2] True 0       # default values are used

>>> somefunc('Hello', [1,2], kwarg1='Hi')
Hello [1, 2] Hi 0         # kwarg2 has default value

>>> somefunc('Hello', [1,2], kwarg2='Hi')
Hello [1, 2] True Hi      # kwarg1 has default value

>>> somefunc('Hello', [1,2], kwarg2='Hi', kwarg1=6)
Hello [1, 2] 6 Hi         # specify all args
```

If we use name=value for all arguments, their sequence can be arbitrary:

```python
>>> somefunc(kwarg2='Hello', arg1='Hi', kwarg1=6, arg2=[2])
Hi [2] 6 Hello
```
Consider a function of $t$, with parameters $A$, $a$, and $\omega$:

$$f(t; A, a, \omega) = Ae^{-at} \sin(\omega t)$$

We can implement $f$ in a Python function with $t$ as positional argument and $A$, $a$, and $\omega$ as keyword arguments:

```python
from math import pi, exp, sin

def f(t, A=1, a=1, omega=2*pi):
    return A*exp(-a*t)*sin(omega*t)

v1 = f(0.2)
v2 = f(0.2, omega=1)
v2 = f(0.2, 1, 3)  # same as f(0.2, A=1, a=3)
v3 = f(0.2, omega=1, A=2.5)
v4 = f(A=5, a=0.1, omega=1, t=1.3)
v5 = f(t=0.2, A=9)
```
A program contains functions and ordinary statements outside functions, the latter constitute the main program

```python
from math import * # in main

def f(x): # in main
    e = exp(-0.1*x)
    s = sin(6*pi*x)
    return e*s

x = 2 # in main
y = f(x) # in main
print 'f(%g)=%g' % (x, y) # in main
```

The execution starts with the first statement in the main program and proceeds line by line, top to bottom.

def statements define a function, but the statements inside the function are not executed before the function is called.
Math functions as arguments to Python functions

- Programs doing calculus frequently need to have functions as arguments in other functions
- We may have Python functions for
  - numerical integration: \( \int_a^b f(x) \, dx \)
  - numerical differentiation: \( f'(x) \)
  - numerical root finding: \( f(x) = 0 \)
- Example: numerical computation of \( f''(x) \) by

\[
 f''(x) \approx \frac{f(x - h) - 2f(x) + f(x + h)}{h^2}
\]

```python
def diff2(f, x, h=1E-6):
    r = (f(x-h) - 2*f(x) + f(x+h))/float(h*h)
    return r
```

- No difficulty with \( f \) being a function (this is more complicated in Matlab, C, C++, Fortran, and very much more complicated in Java)
Application of the \texttt{diff2} function

**Code:**

```python
def g(t):
    return t**(-6)

# make table of g''(t) for 14 h values:
for k in range(1,15):
    h = 10**(-k)
    print 'h=%.0e: %.5f' % (h, diff2(g, 1, h))
```

**Output \((g''(1) = 42)\):**

```
h=1e-01: 44.61504
h=1e-02: 42.02521
h=1e-03: 42.00025
h=1e-04: 42.00000
h=1e-05: 41.99999
h=1e-06: 42.00074
h=1e-07: 41.94423
h=1e-08: 47.73959
h=1e-09: -666.13381
h=1e-10: 0.00000
h=1e-11: 0.00000
h=1e-12: -666133814.77509
h=1e-13: 66613381477.50939
h=1e-14: 0.00000
```
What is the problem? Round-off errors...

- For $h < 10^{-8}$ the results are totally wrong
- We would expect better approximations as $h$ gets smaller
- Problem: for small $h$ we add and subtract numbers of approx equal size and this gives rise to round-off errors
- Remedy: use float variables with more digits
- Python has a (slow) float variable with arbitrary number of digits
- Using 25 digits gives accurate results for $h \leq 10^{-13}$
- Is this really a problem? Quite seldom – other uncertainties in input data to a mathematical computation makes it usual to have (e.g.) $10^{-2} \leq h \leq 10^{-6}$
Exercise

• The map function takes a function and a list, and returns a new list which is the result of applying the function to each item in the list. Write your own version of the map function, myMap, using list comprehension.
If tests:

    if x < 0:
        value = -1
    elif x >= 0 and x <= 1:
        value = x
    else:
        value = 1

User-defined functions:

    def quadratic_polynomial(x, a, b, c):
        value = a*x*x + b*x + c
        derivative = 2*a*x + b
        return value, derivative

    # function call:
    x = 1
    p, dp = quadratic_polynomial(x, 2, 0.5, 1)
    p, dp = quadratic_polynomial(x=x, a=-4, b=0.5, c=0)

Positional arguments must appear before keyword arguments:

    def f(x, A=1, a=1, w=pi):
        return A*exp(-a*x)*sin(w*x)
An integral
\[ \int_a^b f(x) \, dx \]
can be approximated by Simpson’s rule:

\[ \int_a^b f(x) \, dx \approx \frac{b - a}{3n} \left( f(a) + f(b) + 4 \sum_{i=1}^{n/2} f(a + (2i - 1)h) + 2 \sum_{i=1}^{n/2-1} f(a + 2ih) \right) \]

Problem: make a function `Simpson(f, a, b, n=500)` for computing an integral of \( f(x) \) by Simpson’s rule. Call `Simpson(...)` for \( \frac{3}{2} \int_0^\pi \sin^3 x \, dx \) (exact value: 2) for \( n = 2, 6, 12, 100, 500 \).
def Simpson(f, a, b, n=500):
    """
    Return the approximation of the integral of \( f \)
    from \( a \) to \( b \) using Simpson’s rule with \( n \) intervals.
    """

    h = (b - a)/float(n)

    sum1 = 0
    for i in range(1, n/2 + 1):
        sum1 += f(a + (2*i-1)*h)

    sum2 = 0
    for i in range(1, n/2):
        sum2 += f(a + 2*i*h)

    integral = (b-a)/(3*n)*(f(a) + f(b) + 4*sum1 + 2*sum2)
    return integral
def Simpson(f, a, b, n=500):
    if a > b:
        print ‘Error: a=%g > b=%g’ % (a, b)
        return None

    # Check that n is even
    if n % 2 != 0:
        print ‘Error: n=%d is not an even integer!’ % n
        n = n+1  # make n even

    # as before...
    ...
    return integral
def h(x):
    return (3./2)*sin(x)**3

from math import sin, pi

def application():
    print 'Integral of 1.5*sin^3 from 0 to pi:'
    for n in 2, 6, 12, 100, 500:
        approx = Simpson(h, 0, pi, n)
        print 'n=%3d, approx=%18.15f, error=%9.2E' % application()
Property of Simpson’s rule: 2nd degree polynomials are integrated exactly!

```python
def verify():
    """Check that 2nd-degree polynomials are integrated exactly.""
    a = 1.5
    b = 2.0
    n = 8
    g = lambda x: 3*x**2 - 7*x + 2.5  # test integrand
    G = lambda x: x**3 - 3.5*x**2 + 2.5*x  # integral of g
    exact = G(b) - G(a)
    approx = Simpson(g, a, b, n)
    if abs(exact - approx) > 1E-14:  # never use == for floats!
        print "Error: Simpson’s rule should integrate g exactly"

    verify()
```

Exercise

• Problem Set: the rest of the problems
Happy Lunar New Year

YEAR OF SNAKE