The program

What is a program?

A sequence of instructions to the computer, written in a programming language, which is somewhat like English, but very much simpler – and very much stricter!

In this course we shall use the Python language

Our first example program:

Evaluate $y(t) = v_0 t - \frac{1}{2} g t^2$ for $v_0 = 5$, $g = 9.81$ and $t = 0.6$:

$$y = 5 \cdot 0.6 - \frac{1}{2} \cdot 9.81 \cdot 0.6^2$$

Python program for doing this calculation:

```
print 5*0.6 - 0.5*9.81*0.6**2
```

How to write and run the program

- A (Python) program is plain text
- First we need to write the text in a *plain text editor*
- Use Gedit, Emacs or IDLE (*not* MS Word or OpenOffice!)
- Write the program line

```
print 5*0.6 - 0.5*9.81*0.6**2
```

- Save the program to a file (say) `ball_numbers.py`
  (Python programs are (usually) stored files ending with .py)
- Go to a terminal window
- Go to the folder containing the program (text file)
- Give this operating system command:

```
Unix/DOS> python ball_numbers.py
```

- The program prints out 1.2342 in the terminal window

About programs and programming

- When you use a computer, you always run a program
- The computer cannot do anything without being precisely told what to do, and humans write and use programs to tell the computer what to do
- Some anticipate that programming in the future may be as important as reading and writing (!)
- Most people are used to double-click on a symbol to run a program – in this course we give commands in a terminal window because that is more efficient if you work intensively with programming
- In this course we probably use computers differently from what you are used to

Storing numbers in variables

- From mathematics you are used to variables, e.g.,

$$v_0 = 5, \quad g = 9.81, \quad t = 0.6, \quad y = v_0 t - \frac{1}{2} g t^2$$

- We can use variables in a program too, and this makes the last program easier to read and understand:

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

- This program spans several lines of text and use variables, otherwise the program performs the same calculations and gives the same output as the previous program

Names of variables

- In mathematics we usually use one letter for a variable
- In a program it is smart to use one-letter symbols, words or abbreviation of words as names of variables
- The name of a variable can contain the letters a-z, A-Z, underscore `_` and the digits 0-9, but the name cannot start with a digit
- Variable names are case-sensitive (e.g., `a` is different from `A`)
- Example on other variable names in our last program:

```
initial_velocity = 5
accel_of_gravity = 9.81
TIME = 0.6
VerticalPositionOfBall = initial_velocity*TIME - \
                         0.5*accel_of_gravity*TIME**2
print VerticalPositionOfBall
```

  (the backslash allows an instruction to be continued on the next line)
- Good variable names make a program easier to understand!

Some words are reserved in Python

- Certain words have a special meaning in Python and cannot be used as variable names
- These are: `and`, `as`, `assert`, `break`, `class`, `continue`, `def`, `del`, `elif`, `else`, `except`, `exec`, `finally`, `for`, `from`, `global`, `if`, `import`, `in`, `is`, `lambda`, `not`, `or`, `pass`, `print`, `raise`, `return`, `try`, `with`, `while`, and `yield`
- There are many rules about programming and Python, we learn them as we go along with examples

Comments are useful to explain how you think in programs

Program with comments:

```
# program for computing the height of a ball
# in vertical motion
v0 = 5    # initial velocity
g = 9.81  # acceleration of gravity
t = 0.6   # time
y = v0*t - 0.5*g*t**2  # vertical position
print y
```

- Everything after `#` on a line is ignored by the computer and is known as a comment where we can write whatever we want
- Comments are used to explain what the computer instructions mean, what variables mean, how the programmer reasoned when she wrote the program, etc.

"printf-style" formatting of text and numbers

- Output from calculations often contain text and numbers, e.g.

```
At t=0.6 s, y is 1.23 m.
```

- We want to control the formatting of numbers (no of decimals, style: `0.6` vs `6E-01` or `6.0e-01`)
- So-called *printf formatting* is useful for this purpose:

```
print 'At t=%g s, y is %.2f m.' % (t, y)
```

- The printf format has "slots" where the variables listed at the end are put: `%g` $\leftarrow$ `t`, `%.2f` $\leftarrow$ `y`

Examples on different printf formats

- Examples:

```
%g          most compact formatting of a real number
%f          decimal notation (-34.674)
%10.3f      decimal notation, 3 decimals, field width 10
%.3f        decimal notation, 3 decimals, minimum width
%e or %E    scientific notation (1.42e-02 or 1.42E-02)
%9.2e       scientific notation, 2 decimals, field width 9
%d          integer
%5d         integer in a field of width 5 characters
%s          string (text)
%-20s       string, field width 20, left-adjusted
```

- See the the book for more explanation and overview

Example on printf formatting in our program

- Triple-quoted strings (""") can be used for multi-line output, and here we combine such a string with printf formatting:

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

print """
At t=%f s, a ball with
initial velocity v0=%.3E m/s
is located at the height %.2f m.
""" % (t, v0, y)
```

- Running the program:

```
Unix/DOS> python ball_output2.py

At t=0.600000 s, a ball with
initial velocity v0=5.000E+00 m/s
is located at the height 1.23 m.
```

Some frequently used computer science terms

- Program or code or application
- Source code (program text)
- Code/program snippet
- Execute or run a program
- Algorithm (recipe for a program)
- Implementation (writing the program)
- Verification (does the program work correctly?)
- Bugs (errors) and debugging

Computer science meaning of terms is often different from the natural/human language meaning

Statements

- A program consists of statements

```
a = 1       # 1st statement
b = 2       # 2nd statement
c = a + b   # 3rd statement
print c     # 4th statement
```

- Normal rule: one statement per line
- Multiple statements per line is possible with a semicolon in between the statements:

```
a = 1;  b = 2;  c = a + b;  print c
```

- This is a print statement:

```
print 'y=%g' % y
```

- This is an assignment statement:

```
v0 = 3
```

- Assignment: evaluate right-hand side, assign to left-hand side

```
myvar = 10
myvar = 3*myvar    # = 30
```

Syntax

- Programs must have correct syntax, i.e., correct use of the computer language grammar rules, and no misprints
- This is a program with two syntax errors:

```
myvar = 5.2
prinnt Myvar
```

  (`prinnt` is an unknown instruction, `Myvar` is a non-existing variable)

- Python reports syntax errors:

```
      prinnt Myvar
             ^
SyntaxError: invalid syntax
```

- Only the first encountered error is reported and the program is stopped (correct error and continue with next error)

"Programming demands significantly higher standard of accuracy. Things don't simply have to make sense to another human being, they must make sense to a computer." – Donald Knuth

Blanks (whitespace)

- Blanks may or may not be important in Python programs
- These statements are equivalent (blanks do not matter):

```
v0=3
v0  =  3
v0=   3
v0 = 3
```

  (the last is the preferred formatting style of assignments)

- Here blanks do matter:

```
while counter <= 10:
    counter = counter + 1   # correct (4 leading blanks)

while counter <= 10:
counter = counter + 1       # invalid syntax
```

  (more about this in Ch. 2)

Input and output

- A program has some known input data and computes some (on beforehand unknown) output data
- Sample program:

```
v0 = 3;  g = 9.81;  t = 0.6
position = v0*t - 0.5*g*t*t
velocity = v0 - g*t
print 'position:', position, 'velocity:', velocity
```

- Input: `v0`, `g`, and `t`
- Output: `position` and `velocity`

Operating system

- An operating system (OS) is a set of programs managing hardware and software resources on a computer
- Example:

  ```
  Unix/DOS> emacs myprog.py
  ```

  `emacs` is a program that needs help from the OS to find the file `myprog.py` on the computer's disk
- Linux, Unix (Ubuntu, RedHat, Suse, Solaris)
- Windows (95, 98, NT, ME, 2000, XP, Vista)
- Macintosh (old Mac OS, Mac OS X)
- Mac OS X $\approx$ Unix $\approx$ Linux $\neq$ Windows
- Python supports cross-platform programming, i.e., a program is independent of which OS we run the program on

New formula: temperature conversion

- Given C as a temperature in Celsius degrees, compute the corresponding Fahrenheit degrees F:

$$F = \frac{9}{5}C + 32$$

- Program:

```
C = 21
F = (9/5)*C + 32
print F
```

- Execution:

```
Unix/DOS> python c2f_v1.py
53
```

- We must always check that a new program calculates the right answer(s): a calculator gives 69.8, not 53
- Where is the error?

Integer division

- $9/5$ is not 1.8 but 1 in most computer languages (!)
- If a and b are integers, a/b implies integer division: the largest integer c such that $cb \leq a$
- Examples: $1/5 = 0$, $2/5 = 0$, $7/5 = 1$, $12/5 = 2$
- In mathematics, $9/5$ is a real number (1.8) – this is called float division in Python and is the division we want
- One of the operands (a or b) in a/b must be a real number ("float") to get float division
- A float in Python has a dot (or decimals): `9.0` or `9.` is float
- No dot implies integer: `9` is an integer
- `9.0/5` yields `1.8`, `9/5.` yields `1.8`, `9/5` yields `1`
- Corrected program (with correct output 69.8):

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

Objects

- Everything in Python is an object
- Variables refer to objects

```
a = 5        # a refers to an integer (int) object
b = 9        # b refers to an integer (int) object
c = 9.0      # c refers to a real number (float) object
d = b/a      # d refers to an int/int => int object
e = c/a      # e refers to float/int => float object
s = 'b/a=%g' % (b/a)  # s is a string/text (str) object
```

- We can convert between object types:

```
a = 3                # a is int
b = float(a)         # b is float 3.0
c = 3.9              # c is float
d = int(c)           # d is int 3
d = round(c)         # d is float 4.0
d = int(round(c))    # d is int 4
d = str(c)           # d is str '3.9'
e = '-4.2'           # e is str
f = float(e)         # f is float -4.2
```

How are arithmetic expressions evaluated?

- Example: $\frac{5}{9} + 2a^4/2$, in Python written as `5/9 + 2*a**4/2`
- The rules are the same as in mathematics: proceed term by term (additions/subtractions) from the left, compute powers first, then multiplication and division, in each term
- `r1 = 5/9` ($=0$)
- `r2 = a**4`
- `r3 = 2*r2`
- `r4 = r3/2`
- `r5 = r1 + r4`
- Use parenthesis to override these default rules – or use parenthesis to explicitly tell how the rules work (smart):
  `(5/9) + (2*(a**4))/2`

Standard mathematical functions

- What if we need to compute $\sin x$, $\cos x$, $\ln x$, etc. in a program?
- Such functions are available in Python's `math` module
- In general: lots of useful functionality in Python is available in modules – but modules must be *imported* in our programs
- Compute $\sqrt{2}$ using the `sqrt` function in the `math` module:

```
import math
r = math.sqrt(2)
# or
from math import sqrt
r = sqrt(2)
# or
from math import *   # import everything in math
r = sqrt(2)
```

- Another example:

```
from math import sin, cos, log
x = 1.2
print sin(x)*cos(x) + 4*log(x)   # log is ln (base e)
```

A glimpse of round-off errors

- Let us compute $1/49 \cdot 49$ and $1/51 \cdot 51$:

```
v1 = 1/49.0*49
v2 = 1/51.0*51
print '%.16f %.16f' % (v1, v2)
```

- Output with 16 decimals becomes

```
0.9999999999999999 1.0000000000000000
```

- Most real numbers are represented inexactly on a computer
- Neither $1/49$ nor $1/51$ is represented exactly, the error is typically 10^{-16}
- Sometimes such small errors propagate to the final answer, sometimes not, and somtimes the small errors accumulate through many mathematical operations
- Lesson learned: real numbers on a computer and the results of mathematical computations are only approximate

Another example involving math functions

- The $\sinh x$ function is defined as

$$\sinh(x) = \frac{1}{2}\left(e^{x} - e^{-x}\right)$$

- We can evaluate this function in three ways:
  1) `math.sinh`,
  2) combination of two `math.exp`,
  3) combination of two powers of `math.e`

```
from math import sinh, exp, e, pi
x = 2*pi
r1 = sinh(x)
r2 = 0.5*(exp(x) - exp(-x))
r3 = 0.5*(e**x - e**(-x))
print '%.16f %.16f %.16f' % (r1,r2,r3)
```

- Output: `r1` is 267.7448940410164369, `r2` is 267.7448940410164369, `r3` is 267.7448940410163232 (!)

Interactive Python shells

- So far we have performed calculations in Python programs
- Python can also be used interactively in what is known as a shell
- Type `python`, `ipython`, or `idle`
- A Python shell is entered where you can write statements after >>> (IPython has a different prompt)
- Example:

```
Unix/DOS> python
Python 2.5 (r25:409, Feb 27 2007, 19:35:40)
...
>>> C = 41
>>> F = (9.0/5)*C + 32
>>> print F
105.8
>>> F
105.8
```

- Previous commands can be recalled and edited, making the shell an interactive calculator

Complex numbers

- Python has full support for complex numbers
- $2 + 3i$ in mathematics is written as `2 + 3j` in Python
- Examples:

```
>>> a = -2
>>> b = 0.5
>>> s = complex(a, b)  # make complex from variables
>>> s
(-2+0.5j)
>>> s*w                # complex*complex
(-10.5-3.75j)
>>> s/w                # complex/complex
(-0.25641025641025639+0.28205128205128205j)
>>> s.real
-2.0
>>> s.imag
0.5
```

- See the book for additional info

Summary of Chapter 1 (part 1)

- Programs must be accurate!
- Variables are names for objects
- We have met different object types: `int`, `float`, `str`
- Choose variable names close to the mathematical symbols in the problem being solved
- Arithmetic operations in Python: term by term (+/-) from left to right, power before * and / – as in mathematics; use parenthesis when there is any doubt
- Watch out for unintended integer division!

Summary of Chapter 1 (part 2)

- Mathematical functions like $\sin x$ and $\ln x$ must be imported from the `math` module:

```
from math import sin, log
x = 5
r = sin(3*log(10*x))
```

- Use printf syntax for full control of output of text and numbers

```
>>> a = 5.0; b = -5.0; c = 1.9856; d = 33
>>> print 'a is', a, 'b is', b, 'c and d are', c, d
a is 5.0 b is -5.0 c and d are 1.9856 33
```

- Important terms: object, variable, algorithm, statement, assignment, implementation, verification, debugging

Summarizing example: throwing a ball (problem)

- We throw a ball with velocity v_0, at an angle θ with the horizontal, from the point $(x = 0, y = y_0)$. The trajectory of the ball is a parabola (we neglect air resistance):

$$y = x \tan\theta - \frac{1}{2v_0} \frac{gx^2}{\cos^2\theta} + y_0$$

- Let us program this formula
- Program tasks: initialize input data (v_0, g, θ, y_0), import from `math`, compute y
- We give x, y and y_0 in m, $g = 9.81\text{m/s}^2$, v_0 in km/h and θ in degrees – this requires conversion of v_0 to m/s and θ to radians

Summarizing example: throwing a ball (solution)

Program:

```
g = 9.81     # m/s**2
v0 = 15      # km/h
theta = 60   # degrees
x = 0.5      # m
y0 = 1       # m

print """\
v0    = %.1f km/h
theta = %d degrees
y0    = %.1f m
x     = %.1f m\
""" % (v0, theta, y0, x)

# convert v0 to m/s and theta to radians:
v0 = v0/3.6
from math import pi, tan, cos
theta = theta*pi/180

y = x*tan(theta) - 1/(2*v0)*g*x**2/((cos(theta))**2) + y0

print 'y     = %.1f m' % y
```

Making a table; problem

- Suppose we want to make a table of Celsius and Fahrenheit degrees:

```
-20  -4.0
-15   5.0
-10  14.0
 -5  23.0
  0  32.0
  5  41.0
 10  50.0
 15  59.0
 20  68.0
 25  77.0
 30  86.0
 35  95.0
 40 104.0
```

- How can a program write out such a table?

Making a table; simple solution

- We know how to make one line in the table:

```
C = -20
F = 9.0/5*C + 32
print C, F
```

- We can just repeat these statements:

```
C = -20;  F = 9.0/5*C + 32;  print C, F
C = -15;  F = 9.0/5*C + 32;  print C, F
...
C =  35;  F = 9.0/5*C + 32;  print C, F
C =  40;  F = 9.0/5*C + 32;  print C, F
```

- 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*

The while loop

- A while loop executes repeatedly a set of statements as long as a boolean condition is true

```
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 <= 40:
    F = (9.0/5)*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

Boolean expressions

- 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:

```
>>> 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: `C1 and C2` is `True` if both `C1` and `C2` are `True`
- Rule 2: `C1 or C2` is `True` if one of `C1` or `C2` is `True`
- Examples:

```
>>> 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
```

Lists

- 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

```
C1 = -20
C2 = -15
C3 = -10
...
C13 = 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

List operations (part 1)

- 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:

```
L1 = [-91, 'a string', 7.2, 0]
```

- Elements are accessed via an index, e.g. `L1[3]` (index=3)
- List indices are always numbered as 0, 1, 2, and so forth up to the number of elements minus one

```
>>> mylist = [4, 6, -3.5]
>>> print mylist[0]
4
>>> print mylist[1]
6
>>> print mylist[2]
-3.5
>>> len(mylist)  # length of list
3
```

List operations (part 2)

- Some interactive examples on list operations:

```
>>> 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
```

List operations (part 3)

- More examples in an interactive Python shell:

```
>>> 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
>>> somelist = ['book.tex', 'book.log', 'book.pdf']
>>> texfile, logfile, pdf = somelist
>>> texfile
'book.tex'
>>> logfile
'book.log'
>>> pdf
'book.pdf'
```

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

Making a table with a for loop

- The table of Celsius and Fahreheit degrees:

```
Cdegrees = [-20, -15, -10, -5, 0, 5, 10, 15,
            20, 25, 30, 35, 40]
for C in Cdegrees:
    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
```

Translation of a for loop to a while loop

- The for loop

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

  can always be transformed to a while loop

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

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

```
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
```

Storing the table columns as lists

- Let us put all the Fahrenheit values also in a list:

```
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 loop with list indices

- For loops usually loop over list values (elements):

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

- We can alternatively loop over list indices:

```
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)`

```
>>> range(3)          # = range(0, 3, 1)
[0, 1, 2]
>>> range(2, 8, 3)
[2, 5]
```

How to change elements in a list

- Say we want to add 2 to all numbers in a list:

```
>>> 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:

```
>>> 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:

```
>>> 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

```
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:

```
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:

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

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

Traversing multiple lists simultaneously

- What if we want to have a for loop over elements in `Cdegrees` *and* `Fdegrees`?
- We can have a loop over list indices:

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

- Alternative construct (regarded as more "Pythonic"):

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

- Example with three lists:

```
>>> 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

Nested lists: list of lists

- A list can contain "any" object, also another list
- Instead of storing a table as two separate lists (one for each column), we can stick the two lists together in a new list:

```
Cdegrees = range(-20, 41, 5)
Fdegrees = [(9.0/5)*C + 32 for C in Cdegrees]

table1 = [Cdegrees, Fdegrees]  # list of two lists
```

- `table1[0]` is the `Cdegrees` list
- `table1[1]` is the `Fdegrees` list
- `table1[1][2]` is the 3rd element in `Fdegrees`

Table of coloumns vs table of rows

- The previous `table = [Cdegrees,Fdegrees]` is a table of (two) columns
- Let us make a table of rows instead, each row is a `[C,F]` pair:

```
table2 = []
for C, F in zip(Cdegrees, Fdegrees):
    row = [C, F]
    table2.append(row)

# more compact with list comprehension:
table2 = [[C, F] for C, F in zip(Cdegrees, Fdegrees)]
```

- `print table2` gives

```
[[-20, -4.0], [-15, 5.0], ......., [40, 104.0]]
```

- Iteration over a nested list:

```
for C, F in table2:
    # work with C and F from a row in table2
```

Illustration of table of columns

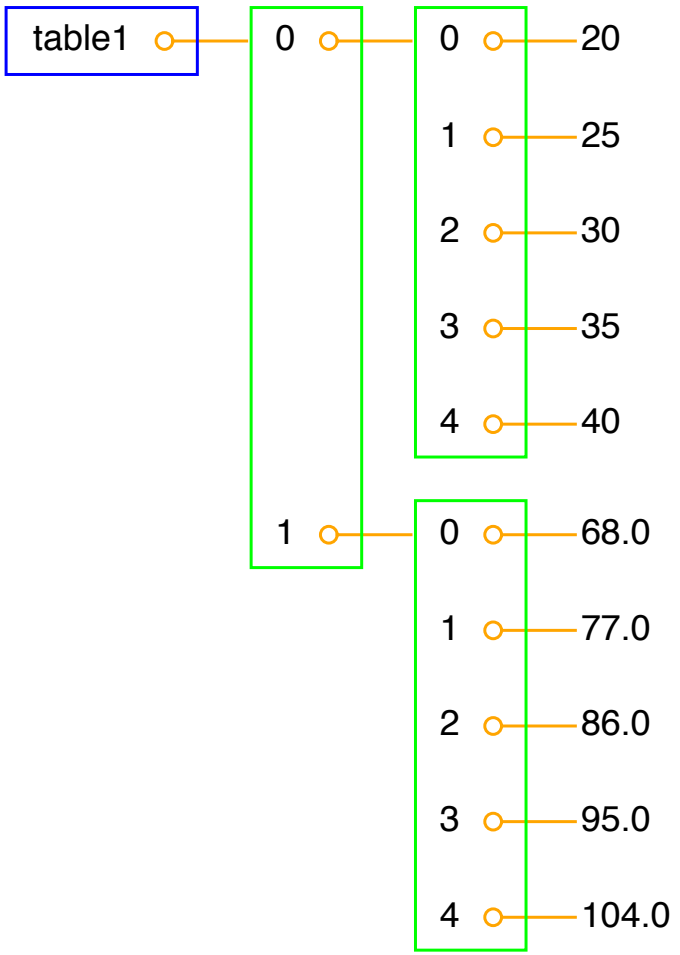
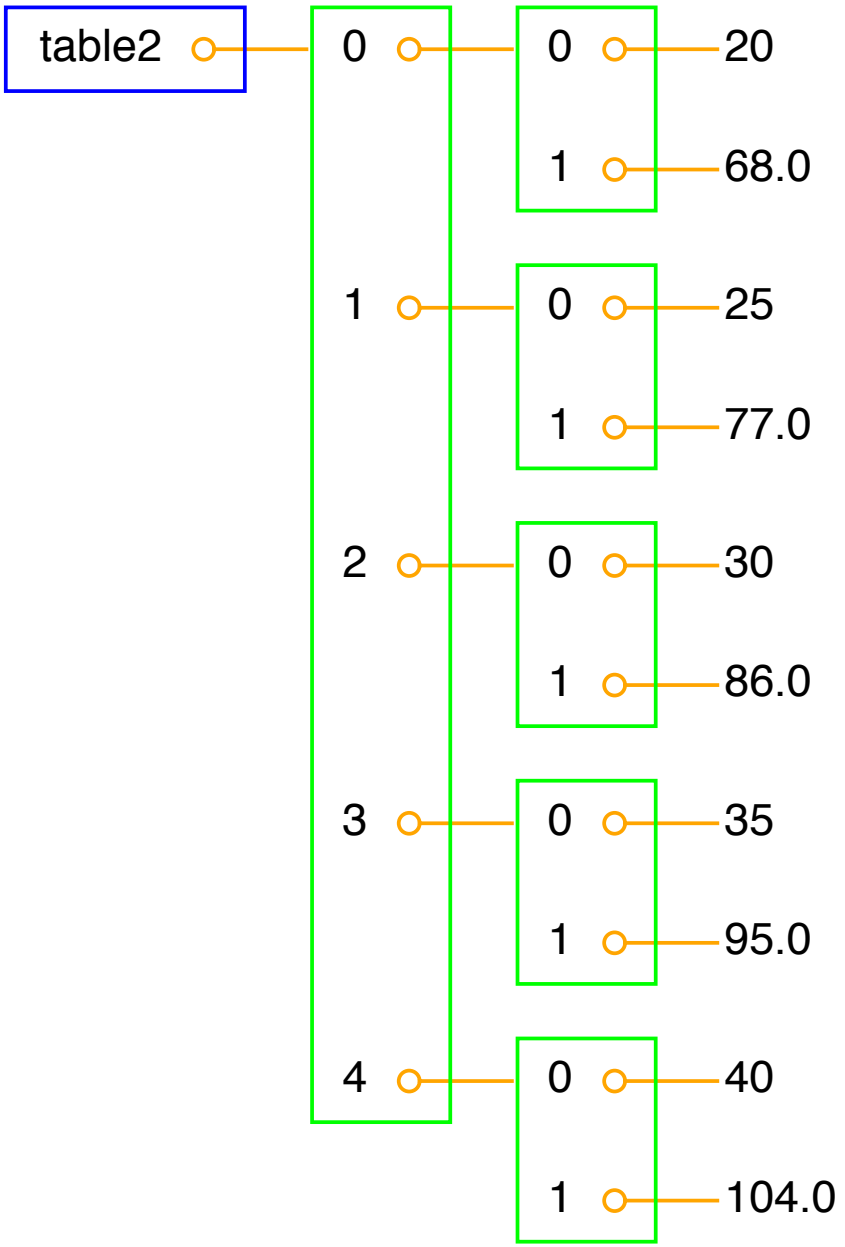


Illustration of table of rows



Extracting sublists (or slices)

- We can easily grab parts of a list:

```
>>> A = [2, 3.5, 8, 10]
>>> A[2:]   # from index 2 to end of list
[8, 10]

>>> A[1:3]  # from index 1 up to, but not incl., index 3
[3.5, 8]

>>> A[:3]   # from start up to, but not incl., index 3
[2, 3.5, 8]

>>> A[1:-1] # from index 1 to next last element
[3.5, 8]

>>> A[:]    # the whole list
[2, 3.5, 8, 10]
```

- Sublists/slices are copies of the original list!

What does this code snippet do?

```
for C, F in table2[Cdegrees.index(10):Cdegrees.index(35)]:
    print '%5.0f %5.1f' % (C, F)
```

- This is a for loop over a sublist of `table2`
- Sublist indices: `Cdegrees.index(10)`, `Cdegrees.index(35)`, i.e., the indices corresponding to elements 10 and 35, i.e., we want to run over rows in `table2` starting with the one where `C` is 10 and ending in the row before the row where `C` is 35
- Output:

```
   10  50.0
   15  59.0
   20  68.0
   25  77.0
   30  86.0
```

More general nested lists

- We have seen one example on a nested list: a table with n rows, each row with a `[C, F]` list of two elements
- Traversal of this list (`table2`):

```
for C, F in table2:
    # work with C and F (columns in the current row)
```

- What if we have a more general list with m rows, n columns, and maybe not the same number of columns in each row?

Example: table with variable no of columns (1)

- We want to record the history of scores in a game
- Each player has played the game a certain number of times
- `scores[i][j]` is a nested list holding the score of game no. `j` for player no. `i`
- Some sample code:

```
scores = []
# score of player no. 0:
scores.append([12, 16, 11, 12])
# score of player no. 1:
scores.append([9])
# score of player no. 2:
scores.append([6, 9, 11, 14, 17, 15, 14, 20])
```

- Desired printing of `scores`:

```
12  16  11  12
 9
 6   9  11  14  17  15  14  20
```

- (Think of many players, each with many games)

Example: table with variable no of columns (2)

We use two loops: one over rows and one over columns

Loops over two list indices (integers):

```
for r in range(len(scores)):
    for c in range(len(scores[r])):
        score = scores[r][c]
        print '%4d' % score,
    print
```

Or: outer loop over rows, inner loop over columns:

```
for row in scores:
    for column in row:
        score = column   # better name...
        print '%4d' % score,
    print
```

Iteration of general nested lists

List with many indices: `somelist[i1][i2][i3]...`

Loops over list indices:

```
for i1 in range(len(somelist)):
    for i2 in range(len(somelist[i1])):
        for i3 in range(len(somelist[i1][i2])):
            for i4 in range(len(somelist[i1][i2][i3])):
                value = somelist[i1][i2][i3][i4]
                # work with value
```

Loops over sublists:

```
for sublist1 in somelist:
    for sublist2 in sublist1:
        for sublist3 in sublist2:
            for sublist4 in sublist3:
                value = sublist4
                # work with value
```

Tuples: lists that cannot be changed

- Tuples are "constant lists":

```
>>> t = (2, 4, 6, 'temp.pdf')    # define a tuple
>>> t =  2, 4, 6, 'temp.pdf'     # can skip parenthesis
>>> t[1] = -1
...
TypeError: object does not support item assignment

>>> t.append(0)
...
AttributeError: 'tuple' object has no attribute 'append'

>>> del t[1]
...
TypeError: object doesn't support item deletion
```

- Tuples can do much of what lists can do:

```
>>> t = t + (-1.0, -2.0)           # add two tuples
>>> t
(2, 4, 6, 'temp.pdf', -1.0, -2.0)
>>> t[1]                           # indexing
4
>>> t[2:]                          # subtuple/slice
(6, 'temp.pdf', -1.0, -2.0)
>>> 6 in t                         # membership
True
```

Why tuples when lists have more functionality?

- Tuples are constant and thus protected against accidental changes
- Tuples are faster than lists
- Tuples are widely used in Python software
  (so you need to know about tuples!)
- Tuples (but not lists) can be used as keys is dictionaries
  (more about dictionaries later)

Summary of loops, lists and tuples

- Loops:

```
while condition:
    <block of statements>

for element in somelist:
    <block of statements>
```

- Lists and tuples:

```
mylist  = ['a string', 2.5, 6, 'another string']
mytuple = ('a string', 2.5, 6, 'another string')
mylist[1]  = -10
mylist.append('a third string')
mytuple[1] = -10  # illegal: cannot change a tuple
```

List functionality

`a = []`	initialize an empty list
`a = [1, 4.4, 'run.py']`	initialize a list
`a.append(elem)`	add `elem` object to the end
`a + [1,3]`	add two lists
`a[3]`	index a list element
`a[-1]`	get last list element
`a[1:3]`	slice: copy data to sublist (here: index 1, 2)
`del a[3]`	delete an element (index 3)
`a.remove(4.4)`	remove an element (with value `4.4`)
`a.index('run.py')`	find index corresponding to an element's value
`'run.py' in a`	test if a value is contained in the list
`a.count(v)`	count how many elements that have the value `v`
`len(a)`	number of elements in list `a`
`min(a)`	the smallest element in `a`
`max(a)`	the largest element in `a`
`sum(a)`	add all elements in `a`
`a.sort()`	sort list `a` (changes `a`)
`as = sorted(a)`	sort list `a` (return new list)
`a.reverse()`	reverse list `a` (changes `a`)
`b[3][0][2]`	nested list indexing
`isinstance(a, list)`	is `True` if `a` is a list

How to find more Python information

- The book contains only fragments of the Python language (intended for real beginners!)
- These slides are even briefer
- Therefore you will need to look up more Python information
- Primary reference: The official Python documentation at `docs.python.org`
- Very useful: The Python Library Reference, especially the index
- Example: what can I find in the `math` module?Go to the Python Library Reference index, find "math", click on the link and you get to a description of the module
- Alternative: `pydoc math` in the terminal window (briefer)
- Note: for a newbie it is difficult to read manuals (intended for experts) – you will need a lot of training; just browse, don't read everything, try to dig out the key info

We have used many Python functions

- Mathematical functions:

```
from math import *
y = sin(x)*log(x)
```

- Other functions:

```
n = len(somelist)
ints = range(5, n, 2)
```

- Functions used with the dot syntax (called *methods*):

```
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 $F(C) = \frac{9}{5}C + 32$

```
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

Functions must be called

- A function does not do anything before it is called
- Examples on calling the `F(C)` function:

```
a = 10
F1 = F(a)
temp = F(15.5)
print F(a+1)
sum_temp = F(10) + F(20)
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

Local variables in Functions

- Example: sum the integers from `start` to `stop`

```
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!!~~

Python function for the "ball in the air formula"

- Recall the formula $y(t) = v_0 t - \frac{1}{2} g t^2$:
- We can make Python function for $y(t)$:

```
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:

```
t = 0.1  # arguments get values as in standard assignments
v0 = 6
g = 9.81
return v0*t - 0.5*g*t**2
```

Functions may access global variables

- The `y(t,v0)` function took two arguments
- Could implement $y(t)$ as a function of t only:

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

- `v0` must be defined in the calling program program before we call `yfunc`

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

- `v0` 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`

Functions can return multiple values

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

```
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

```
>>> 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)
```

Example: compute a function defined as a sum

- 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 $L(x;n)$:

```
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)
```

Returning errors as well from the L(x, n) function

- We can return more: also the first neglected term in the sum and the error ($\ln(1+x) - L(x;n)$):

```
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)
```

Functions do not need to return objects

- 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`

```
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)
```

No return value implies that None is returned

- Consider a function without any return value:

```
>>> def message(course):
...     print "%s is the greatest fun I've "\
...           "ever experienced" % course
...
>>> message('INF1100')
INF1100 is the greatest fun I've ever experienced
>>> r = message('INF1100)  # store the return value
INF1100 is the greatest fun I've ever experienced
>>> print r
None
```

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

Keyword arguments

- Functions can have arguments of the form name=value, called *keyword arguments*:

```
>>> 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:

```
>>> somefunc(kwarg2='Hello', arg1='Hi', kwarg1=6, arg2=[2])
Hi [2] 6 Hello
```

Example: function with default parameteres

- Consider a function of t, with parameters A, a, and ω:

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

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

```
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)
```

Doc strings

- Python convention: document the purpose of a function, its arguments, and its return values in a *doc string* – a (triple-quoted) string written right after the function header
- Examples:

```
def C2F(C):
    """Convert Celsius degrees (C) to Fahrenheit."""
    return (9.0/5)*C + 32

def line(x0, y0, x1, y1):
    """
    Compute the coefficients a and b in the mathematical
    expression for a straight line y = a*x + b that goes
    through two points (x0, y0) and (x1, y1).

    x0, y0: a point on the line (floats).
    x1, y1: another point on the line (floats).
    return: a, b (floats) for the line (y=a*x+b).
    """
    a = (y1 - y0)/(x1 - x0)
    b = y0 - a*x0
    return a, b
```

Convention for input and output data in functions

- A function can have three types of input and output data:
  - input data specified through positional/keyword arguments
  - input/output data given as positional/keyword arguments that will be modified and returned
  - output data created inside the function
- *All output data are returned, all input data are arguments*
- Sketch of a general Python function:

```
def somefunc(i1, i2, i3, io4, io5, i6=value1, io7=value2):
    # modify io4, io5, io7; compute o1, o2, o3
    return o1, o2, o3, io4, io5, io7
```

- `i1`, `i2`, `i3`, `i6`: pure input data
- `io4`, `io5`, `io7`: input and output data
- `o1`, `o2`, `o3`: pure output data

The main program

- A program contains functions and ordinary statements outside functions, the latter constitute the *main program*

```
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

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

```
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 `diff2` function

Code:

```
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 uncertainies in input data to a mathematical computation makes it usual to have (e.g.) $10^{-2} \leq h \leq 10^{-6}$

If tests

- Sometimes we want to peform 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:

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

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

```
if condition:
    <block of statements, executed if condition is True>
else:
    <block of statements, executed if condition is False>
```

If tests with multiple branches (part 1)

- We can test for multiple (here 3) conditions:

```
if condition1:
    <block of statements>
elif condition2:
    <block of statements>
elif condition3:
    <block of statements>
else:
    <block of statements>
<next statement>
```

If tests with multiple branches (part 2)

- Example on multiple branches:

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

```
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
```

Inline if tests

- A common construction is

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

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

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

- Example:

```
def f(x):
    return (sin(x) if 0 <= x <= 2*pi else 0)
```

Summary of if tests and functions

- 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)
```