Introduction to Python for Computational Physics

Prof. Brian D'Urso

University of Pittsburgh
Department of Physics and Astronomy

Week 1

What is Python?

- A high-level language.
  - Built-in high level data structures.
  - Object oriented.
- An interpreted language.
  - You don't compile your programs.
  - Exception framework with tracebacks (no segmentation faults).
  - Automatic memory management (no malloc or free).
  - Dynamic typing, dynamic binding.
- Huge standard library with all sorts of functionality.
- Extensible and embeddable.
- Cross platform and free.
- Great as both a scripting/glue language and for full-blown application development.

Installation

Lab computers have the software pre-installed in Windows. If you want to use your personal machine:

- Mac
  - Install Parallels Desktop: http://technology.pitt.edu/software/for-students-software/parallelsdesktop-st.html
  - Install Windows and follow installation below.
- Windows
  - Python, libraries, and related utilities. Download and install pythonxy - Windows only! see http://code.google.com/p/pythonxy/
- Linux/Mac OS: Download and compile/install the packages.

Running Python

There are many ways Python can be used:

- Interactively:
  - Run the python program with no arguments and get:

```
Python 2.6.6 (r266:84297, Aug 24 2010, 18:46:32)
[MSC v.1500 32 bit (Intel)] on win32
Type "help", "copyright", "credits" ...
>>>
```

  - Useful for tests, debugging, and for demonstrations.
  - This is where we'll start today.
- Non-interactively:
  - Write a script (a text file) and run it.
  - Used for programs with more than a line or two.

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.
```

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

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

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
```

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!!

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
```

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)

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
```

The minimal need-to-know about vectors

- Vectors are known from high school mathematics, e.g., point (x, y) in the plane, point (x, y, z) in space
- In general, a vector v is an n-tuple of numbers: $v = (v_0, \ldots, v_{n-1})$
- There are rules for various mathematical operations on vectors, read the book for details (later?)
- Vectors can be represented by lists: v_i is stored as `v[i]`

Vectors and arrays are concepts in this chapter so we need to briefly explain what these concepts are. It takes separate math courses to understand what vectors and arrays really are, but in this course we only need a small subset of the complete story. A learning strategy may be to just start using vectors/arrays in programs and later, if necessary, go back to the more mathematical details in the first part of Ch. 4.

The minimal need-to-know about arrays

- Arrays are a generalization of vectors where we can have multiple indices: $A_{i,j}$, $A_{i,j,k}$ – in code this is nothing but nested lists, accessed as `A[i][j]`, `A[i][j][k]`
- Example: table of numbers, one index for the row, one for the column

$$\begin{bmatrix} 0 & 12 & -1 & 5 \\ -1 & -1 & -1 & 0 \\ 11 & 5 & 5 & -2 \end{bmatrix} \qquad A = \begin{bmatrix} A_{0,0} & \cdots & A_{0,n-1} \\ \vdots & \ddots & \vdots \\ A_{m-1,0} & \cdots & A_{m-1,n-1} \end{bmatrix}$$

- The no of indices in an array is the *rank* or *number of dimensions*
- Vector = one-dimensional array, or rank 1 array
- In Python code, we use Numerical Python arrays instead of lists to represent mathematical arrays (because this is computationally more efficient)

Storing (x,y) points on a curve in lists/arrays

- Collect (x, y) points on a function curve $y = f(x)$ in a list:

```
>>> def f(x):
...     return x**3       # sample function
...
>>> n = 5                 # no of points in [0,1]
>>> dx = 1.0/(n-1)        # x spacing
>>> xlist = [i*dx for i in range(n)]
>>> ylist = [f(x) for x in xlist]

>>> pairs = [[x, y] for x, y in zip(xlist, ylist)]
```

- Turn lists into Numerical Python (NumPy) arrays:

```
>>> import numpy as np
>>> x2 = np.array(xlist)      # turn list xlist into array
>>> y2 = np.array(ylist)
```

Make arrays directly (instead of lists)

- Instead of first making lists with x and $y = f(x)$ data, and then turning lists into arrays, we can make NumPy arrays directly:

```
>>> n = 5                          # number of points
>>> x2 = np.linspace(0, 1, n) # n points in [0, 1]
>>> y2 = np.zeros(n)               # n zeros (float data type)
>>> for i in xrange(n):
...     y2[i] = f(x2[i])
...
```

- `xrange` is similar to `range` but faster (esp. for large `n` – `xrange` does not explicitly build a list of integers, `xrange` just lets you loop over the values)
- List comprehensions create lists, not arrays, but we can do

```
>>> y2 = np.array([f(xi) for xi in x2])  # list -> array
```

The clue about NumPy arrays (part 1)

- Lists can hold any sequence of any Python objects
- Arrays can only hold objects of the same type
- Arrays are most efficient when the elements are of basic number types (`float`, `int`, `complex`)
- In that case, arrays are stored efficiently in the computer memory and we can compute very efficiently with the array elements

The clue about NumPy arrays (part 2)

- Mathematical operations on whole arrays can be done without loops in Python
- For example,

```
x = np.linspace(0, 2, 10001)   # numpy array
for i in xrange(len(x)):
    y[i] = sin(x[i])
```

  can be coded as

```
y = np.sin(x)                  # x: array, y: array
```

  and the loop over all elements is now performed in a very efficient C function
- Operations on whole arrays, instead of using Python for loops, is called *vectorization* and is very convenient and very efficient (and an important programming technique to master)

Vectorizing the computation of points on a function curve

- Consider the loop with computing x coordinates (`x2`) and $y = f(x)$ coordinates (`y2`) along a function curve:

```
x2 = np.linspace(0, 1, n)   # n points in [0, 1]
y2 = np.zeros(n)            # n zeros (float data type)
for i in xrange(n):
    y2[i] = f(x2[i])
```

- This computation can be replaced by

```
x2 = np.linspace(0, 1, n)   # n points in [0, 1]
y2 = f(x2)                  # y2[i] = f(x[i]) for all i
```

- Advantage: 1) no need to allocate space for `y2` (via `np.zeros`), 2) no need for a loop, 3) *much* faster computation
- Next slide explains what happens in `f(x2)`

How a vectorized function works

- Consider

```
def f(x):
    return x**3
```

- `f(x)` is intended for a number `x`, called *scalar* – contrary to vector/array
- What happens with a call `f(x2)` when `x2` is an array?
- The function then evaluates `x**3` for an array `x`
- Numerical Python supports arithmetic operations on arrays, which correspond to the equivalent operations on each element

```
from numpy import cos, exp
x**3              # x[i]**3                      for all i
cos(x)            # cos(x[i])                    for all i
x**3 + x*cos(x)   # x[i]**3 + x[i]*cos(x[i])     for all i
x/3*exp(-x*a)     # x[i]/3*exp(-x[i]*a)          for all i
```

Vectorization

- Functions that can operate on vectors (or arrays in general) are called vectorized functions (containing vectorized expressions)
- Vectorization is the process of turning a non-vectorized expression/algorithm into a vectorized expression/algorithm
- Mathematical functions in Python without `if` tests automatically work for both scalar and array (vector) arguments (i.e., no vectorization is needed by the programmer)

More explanation of a vectorized expression

- Consider `y = x**3 + x*cos(x)` with array `x`
- This is how the expression is computed:

```
r1 = x**3     # call C function for x[i]**3 loop
r2 = cos(x)   # call C function for cos(x[i]) loop
r3 = x*r2     # call C function for x[i]*r2[i] loop
y = r1 + r3   # call C function for r1[i]+r3[i] loop
```

- The C functions are highly optimized and run very much faster than Python `for` loops (factor 10-500)
- Note: `cos(x)` calls `numpy`'s `cos` (for arrays), not `math`'s `cos` (for scalars) if we have done `from numpy import cos` or `from numpy import *`

Summarizing array example

- Make two arrays `x` and `y` with 51 coordinates x_i and $y_i = f(x_i)$ on the curve $y = f(x)$, for $x \in [0, 5]$ and $f(x) = e^{-x}\sin(\omega x)$:

```
from numpy import linspace, exp, sin, pi

def f(x):
    return exp(-x)*sin(omega*x)

omega = 2*pi
x = linspace(0, 5, 51)
y = f(x)    #  or  y = exp(-x)*sin(omega*x)
```

- Without `numpy`:

```
from math import exp, sin, pi

def f(x):
    return exp(-x)*sin(omega*x)

omega = 2*pi
n = 51
dx = (5-0)/float(n)
x = [i*dx for i in range(n)]
y = [f(xi) for xi in x]
```

Assignment of an array does not copy the elements!

- Consider this code:

```
a = x
a[-1] = q
```

- Is `x[-1]` also changed to `q`? Yes!
- `a` refers to the same array as `x`
- To avoid changing `x`, `a` must be a copy of `x`:

```
a = x.copy()
```

- The same yields slices:

```
a = x[r:]
a[-1] = q   # changes x[-1]!
a = x[r:].copy()
a[-1] = q   # does not change x[-1]
```

In-place array arithmetics

- We have said that the two following statements are equivalent:

```
a = a + b    # a and b are arrays
a += b
```

- Mathematically, this is true, but not computationally
- `a = a + b` first computes `a + b` and stores the result in an intermediate (hidden) array (say) `r1` and then the name `a` is bound to `r1` – the old array `a` is lost
- `a += b` adds elements of `b` *in-place* in `a`, i.e., directly into the elements of `a` without making an extra `a+b` array
- `a = a + b` is therefore less efficient than `a += b`

Compound array expressions

- Consider

```
a = (3*x**4 + 2*x + 4)/(x + 1)
```

- Here are the actual computations:

```
r1 = x**4;  r2 = 3*r1;  r3 = 2*x;  r4 = r1 + r3
r5 = r4 + 4;  r6 = x + 1; r7 = r5/r6; a = r7
```

- With in-place arithmetics we can save four extra arrays, at a cost of much less readable code:

```
a = x.copy()
a **= 4
a *= 3
a += 2*x
a += 4
a /= x + 1
```

More on useful array operations

- Make a new array with same size as another array:

```
# x is numpy array
a = x.copy()
# or
a = zeros(x.shape, x.dtype)
```

- Make sure a list or array is an array:

```
a = asarray(a)
b = asarray(somearray, dtype=float)
```

- Test if an object is an array:

```
>>> type(a)
<type 'numpy.ndarray'>
>>> isinstance(a, ndarray)
True
```

- Generate range of numbers with given spacing:

```
>>> arange(-1, 1, 0.5)
array([-1. , -0.5,  0. ,  0.5])   # 1 is not included!
>>> linspace(-1, 0.5, 4)          # equiv. array

>>> from scitools.std import *
>>> seq(-1, 1, 0.5)               # 1 is included
array([-1. , -0.5,  0. ,  0.5,  1. ])
```

Example: vectorizing a constant function

- Constant function:

```
def f(x):
    return 2
```

- Vectorized version must return array of 2's:

```
def fv(x):
    return zeros(x.shape, x.dtype) + 2
```

- New version valid both for scalar and array `x`:

```
def f(x):
    if isinstance(x, (float, int)):
        return 2
    elif isinstance(x, ndarray):
        return zeros(x.shape, x.dtype) + 2
    else:
        raise TypeError\
        ('x must be int/float/ndarray, not %s' % type(x))
```

Generalized array indexing

- Recall slicing: `a[f:t:i]`, where the slice `f:t:i` implies a set of indices
- Any integer list or array can be used to indicate a set of indices:

```
>>> a = linspace(1, 8, 8)
>>> a
array([ 1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.])
>>> a[[1,6,7]] = 10
>>> a
array([  1.,  10.,   3.,   4.,   5.,   6.,  10.,  10.])
>>> a[range(2,8,3)] = -2   # same as a[2:8:3] = -2
>>> a
array([  1.,  10.,  -2.,   4.,   5.,  -2.,  10.,  10.])
```

- Boolean expressions can also be used (!)

```
>>> a[a < 0]             # pick out all negative elements
array([-2., -2.])
>>> a[a < 0] = a.max() # if a[i]<10, set a[i]=10
>>> a
array([  1.,  10.,  10.,   4.,   5.,  10.,  10.,  10.])
```

Summary of vectors and arrays

- Vector/array computing: apply a mathematical expression to every element in the vector/array
- Ex: `sin(x**4)*exp(-x**2)`, `x` can be array or scalar, for array the `i`'th element becomes `sin(x[i]**4)*exp(-x[i]**2)`
- Vectorization: make scalar mathematical computation valid for vectors/arrays
- Pure mathematical expressions require no extra vectorization
- Mathematical formulas involving `if` tests require manual work for vectorization:

```
scalar_result = expression1 if condition else expression2
vector_result = where(condition, expression1, expression2)
```

Array functionality

`array(ld)`	copy list data `ld` to a `numpy` array
`asarray(d)`	make array of data `d` (copy if necessary)
`zeros(n)`	make a vector/array of length `n`, with zeros (`float`)
`zeros(n, int)`	make a vector/array of length `n`, with `int` zeros
`zeros((m,n), float)`	make a two-dimensional with shape (`m`,`n`)
`zeros(x.shape, x.dtype)`	make array with shape and element type as `x`
`linspace(a,b,m)`	uniform sequence of `m` numbers between `a` and `b`
`seq(a,b,h)`	uniform sequence of numbers from `a` to `b` with step `h`
`iseq(a,b,h)`	uniform sequence of integers from `a` to `b` with step `h`
`a.shape`	tuple containing `a`'s shape
`a.size`	total no of elements in `a`
`len(a)`	length of a one-dim. array `a` (same as `a.shape[0]`)
`a.reshape(3,2)`	return `a` reshaped as 2×3 array
`a[i]`	vector indexing
`a[i,j]`	two-dim. array indexing
`a[1:k]`	slice: reference data with indices `1`,...,`k-1`
`a[1:8:3]`	slice: reference data with indices `1`, `4`,...,`7`
`b = a.copy()`	copy an array
`sin(a), exp(a), ...`	`numpy` functions applicable to arrays
`c = concatenate(a, b)`	`c` contains `a` with `b` appended
`c = where(cond, a1, a2)`	`c[i] = a1[i]` if `cond[i]`, else `c[i] = a2[i]`
`isinstance(a, ndarray)`	is `True` if `a` is an array