Software tools for Visualisation, Scientific Computing and Statistics
Course notes
Python

Christophe prud’homme

Université de Grenoble

2010-10-15

Outline

Python

- Installation
- What is Python?
- Very simple ’programs’
- Lists, Tuples, and Dictionaries
  - Tuples
  - Lists
  - Dictionaries
- IPython
- Some Math
- Classes
- Bindings
- C++/Python Bindings
  - Swig
  - Boost.Python
  - Python in C/C++
- References

Outline

Installing Python

With Debian or Ubuntu the installation is very simple

```
apt-get install python
```

Versions

Several versions of Python are available 2.4, 2.5 and 2.6.

- `python -V` gives the version of Python that will be used

```
% python -V
Python 2.5.4
```

Outline

What is Python?

- Python is an *interpreted programming language*
- What Python is, is a language which is never actually compiled in full - instead, an interpreter turns each line of code into 0s and 1s that your computer can understand this. And it is done on the fly - it compiles the bits of the program you are using as you are using them
- Allows for easy portability of code
- Supports a large library of modules, toolboxes
  - numerical methods (optimisation,...)
  - graphics
  - statistics...

More Information

- `www.python.org`
- Google
- ...

Start Python

Environment

Start the Python environment by calling Python (very much like Matlab/Octave/Scilab)

```
1167 scitools % python
Python 2.5.4 (r254:67916, Sep 26 2009, 10:32:22)
[GCC 4.3.4] on linux2
Type "help", "copyright", "credits" or "license" for more info
>>> quit
Use quit() or Ctrl-D (i.e. EOF) to exit
>>> quit()
```

Outline

Math in Python I

```
>>> 1 + 1
2

>>> 20+80
100
>>> 18294+449566
467860
(These are additions)
>>> 6-5
1
(Subtraction)
>>> 2*5
10
(Multiply, rabbits!)
>>> 5**2
25
(Exponentials e.g. this one is 5 squared)

>>> print "1 + 2 is an addition"
1 + 2 is an addition
(the print statement, which writes something onscreen)
>>> print "one kilobyte is 2^10 bytes, or", 2**10, "bytes"
```

Math in Python II

```
one kilobyte is 2^10 bytes, or 1024 bytes
(you can print sums and variables in a sentence.
The commas seperating each section are a way of
seperating clearly different things that you are printing)

>>> 21/3
7
>>> 23/3
7
>>> 23.0/3.0
7.6666...
(division, 2nd time ignoring remainder/decimals,
3rd time including decimals)
>>> 23%3
2
>>> 49%10
9
(the remainder from a division)
```

Commands

Table: Python operators

command	name	example	output
+	Addition	4+5	9
*	Multiplication	4*5	20
/	Division	18/3	6
%	Remainder	19%3	5
**	Exponent	2**4	16

Remember that thing called order of operation that they taught in maths? Well, it applies in python, too. Here it is, if you need reminding:

1. parentheses ()
2. exponents **
3. multiplication *, division and remainder
4. addition + and subtraction -

Order of operations

Here are some examples that you might want to try, if you're rusty on this:

Listing 1: Code Example 3 - Order of Operations

```
>>> 1 + 2 * 3
7
>>> (1 + 2) * 3
9
```

- In the first example, the computer calculates 2 * 3 first, then adds 1 to it. This is because multiplication has the higher priority (at 3) and addition is below that (at lowly 4).
- In the second example, the computer calculates 1 + 2 first, then multiplies it by 3. This is because parentheses (brackets, like the ones that are surrounding this interluding text ;)) have the higher priority (at 1) and addition comes in later than that.

Comments

The final thing you'll need to know to move on to multi-line programs is the comment. Type the following (and yes, the output is shown):

Listing 2: comments

```
>>> # I am a comment. Fear my wrath!

>>>
```

A comment is a piece of code that is not run. In python, you make something a comment by putting a hash in front of it. A hash comments everything after it in the line, and nothing before it. So you could type this:

Comments

A comment is a piece of code that is not run. In python, you make something a comment by putting a hash in front of it. A hash comments everything after it in the line, and nothing before it. So you could type this:

Listing 3: comment examples

```
>>> print "food is very nice" #eat me

food is very nice

(a normal output, without the smutty comment,
thankyou very much)

>>># print "food is very nice"

(nothing happens, because the code was after a comment)

>>> print "food is very nice" eat me
```

(you'll get a fairly harmless error message, because you didn't put your comment after a hash) Comments are important for adding necessary information for another programmer to

The ’while’ loop I

The following are examples of a type of loop, called the ’while’ loop:

Listing 4: The while loop

```
a = 0
while a < 10:
    a = a + 1
    print a
```

Listing 5: while loop form, and example

```
while {condition that the loop continues}:
    {what to do in the loop}
    {have it indented, usually four spaces}
{the code here is not looped}
{because it isn’t indented}
```

The ’while’ loop II

```
#EXAMPLE
#Type this in, see what it does
x = 10
while x != 0:
    print x
    x = x - 1
    print "wow, we’ve counted x down, and now it equals", x


print "And now the loop has ended."
```

Remember, to make a program, you open Python, click File > New Window, type your program in the new window, then press F5 to run.

Boolean Expressions (Boolean... what?!?) I

Expression	Function
<	less than
<=	less that or equal to
>	greater than
>=	greater than or equal to
!=	not equal to
<>	not equal to (alternate)
==	equal to

Table: Table: Boolean operators

Dont get '=' and '==' mixed up - the '=' operator makes what is on the left equal to what is on the right. the '==' operator says whether the thing on the left is the same as what is on the right, and returns true or false.

Conditional Statements I

Conditionals are where a section of code is only run if certain conditions are met.

Listing 6: if statement and example

```
if {conditions to be met}:
    {do this}
    {and this}
    {and this}
{but this happens regardless because it isn't indented}
```

Listing 7: if statement and example

```
# EXAMPLE 1
y = 1
if y == 1:
    print "y still equals 1, I was just checking"

# EXAMPLE 2
print "We will show the even numbers up to 20"
n = 1
while n <= 20:
```

Conditional Statements II

```
    if n % 2 == 0:
        print n
    n = n + 1
print "there, done."
```

’else’ and ’elif’ - When it Ain’t True I

There are many ways you can use the **if** statement, do deal with situations where your boolean expression ends up FALSE. They are **else** and **elif**.

else simply tells the computer what to do if the conditions of **if** arent met. For example, read the following:

Listing 8: the else statement

```
a = 1
if a > 5:
    print "This shouldn’t happen."
else:
    print "This should happen."
```

elif is just a shortened way of saying **else if**. When the **if** statement fails to be true, **elif** will do what is under it **if** the conditions are met.

’else’ and ’elif’ - When it Ain’t True II

Listing 9: The elif statement

```
z = 4
if z > 70:
    print "Something is very wrong"
elif z < 7:
    print "This is normal"
```

The **if** statement, along with **else** and **elif** follow this form:

Listing 10: the complete if syntax

```
if {conditions}:
    {run this code}
elif {conditions}:
    {run this code}
elif {conditions}:
    {run this code}
else:
    {run this code}


#You can have as many or as little elif
#statements as you need anywhere from zero to very many.
```

’else’ and ’elif’ - When it Ain’t True III

```
#You can have at most one else statement
#and only after all other ifs and elifs.
```

One of the most important points to remember is that you MUST have a colon : at the end of every line with an ’if’, ’elif’, ’else’ or ’while’ in it.

Indentation I

The code MUST BE INDENTED.

If you want to loop the next five lines with a ’while’ loop, you must put a set number of spaces at the beginning of each of the next five lines. This is good programming practice in any language, but python requires that you do it.

Listing 11: Indentation

```
a = 10
while a > 0:
    print a
    if a > 5:
        print "Big number!"
    elif a % 2 != 0:
        print "This is an odd number"
        print "It isn’t greater than five, either"
    else:
        print "this number isn’t greater than 5"
        print "nor is it odd"
```

Indentation II

```
        print "feeling special?"
    a = a - 1
    print "we just made 'a' one less than what it was!"
    print "and unless a is not greater than 0, we do the loop a
print "well, it seems as if 'a' is now no bigger than 0!"
print "the loop is now over, and without furthur adue, "
print "so is this program!"
```

Indentation III

Notice the three levels of indents there:

- Each line in the first level starts with no spaces. It is the main program, and will always execute.
- Each line in the second level starts with four spaces. When there is an ’if’ or loop on the first level, everything on the second level after that will be looped/’ifed’, until a new line starts back on the first level again.
- Each line in the third level starts with eight spaces. When there is an ’if’ or loop on the second level, everything on the third level after that will be looped/’ifed’, until a new line starts back on the second level again.

Functions I

To define your own functions use the **def** operator:

Listing 12: The def operator

```
def function_name(parameter_1,parameter_2):
    {this is the code in the function}
    {more code}
    {more code}
    return {value to return to the main program}
{this code isn't in the function}
{because it isn't indented}
# remember to put a colon ":" at the end
# of the line that starts with 'def'
```

`function_name` is the name of the function. You write the code that is in the function below that line, and have it indented. (We will worry about `parameter_1` and `parameter_2` later, for now imagine there is nothing between the parentheses.

Functions return values

Functions run completely independent of the main program. Remember when I said that when the computer comes to a function, it doesn't see the function, but a value, that the function returns? Here's the quote:

Listing 13: using return

```
# Below is the function
def hello():
    print "hello"
    return 1234

# And here is the function being used
print hello()
```

Passing Parameters to functions I

Listing 14: Defining functions with parameters

```
def function_name(parameter_1,parameter_2):
    {this is the code in the function}
    {more code}
    {more code}
    return {value (e.g. text or number) \
           to return to the main program}
```

Where `parameter_1` and `parameter_2` are between the parentheses

Passing Parameters to functions II

Listing 15: how parameters work

```
def funnyfunction(first_word,second_word,third_word):
    print "The word created is: " + first_word \
          + second_word +third_word
    return first_word + second_word + third_word
```

When you run the function above, you would type in something like this:

```
funnyfunction("meat","eater","man")
```

A Calculator Program I

```
# calculator program

# NO CODE IS REALLY RUN HERE, IT IS ONLY TELLING US
# WHAT WE WILL DO LATER
# Here we will define our functions
# this prints the main menu, and prompts for a choice
def menu():
    #print what options you have
    print "Welcome to calculator.py"
    print "your options are:"
    print " "
    print "1) Addition"
    print "2) Subtraction"
    print "3) Multiplication"
    print "4) Division"
    print "5) Quit calculator.py"
    print " "
    return input ("Choose your option: ")

# this adds two numbers given
def add(a,b):
    print a, "+", b, "=", a + b
```

A Calculator Program II

```
# this subtracts two numbers given
def sub(a,b):
    print b, "-", a, "=", b - a

# this multiplies two numbers given
def mul(a,b):
    print a, "*", b, "=", a * b

# this divides two numbers given
def div(a,b):
    print a, "/", b, "=", a / b

# NOW THE PROGRAM REALLY STARTS, AS CODE IS RUN
loop = 1
choice = 0
while loop == 1:
    choice = menu()
    if choice == 1:
        add(input("Add this: "),input("to this: "))
    elif choice == 2:
        sub(input("Subtract this: "),input("from this: "))
    elif choice == 3:
```

A Calculator Program III

```
        mul(input("Multiply this: "),input("by this: "))
    elif choice == 4:
        div(input("Divide this: "),input("by this: "))
    elif choice == 5:
        loop = 0

print "Thankyou for using calculator.py!"

# NOW THE PROGRAM REALLY FINISHES
```

Outline

Lists, Tuples, and Dictionaries

- Lists are what they seem - a list of values. Each one of them is numbered, starting from zero - the first one is numbered zero, the second 1, the third 2, etc. You can remove values from the list, and add new values to the end. Example: Your many cats' names.
- Tuples are just like lists, but you can't change their values. The values that you give it first up, are the values that you are stuck with for the rest of the program. Again, each value is numbered starting from zero, for easy reference. Example: the names of the months of the year.
- Dictionaries are similar to what their name suggests - a dictionary. In a dictionary, you have an 'index' of words, and for each of them a definition. In python, the word is called a 'key', and the definition a 'value'. The values in a dictionary aren't numbered - tare similar to what their name suggests - a dictionary. In a dictionary, you have an 'index' of words, and for each of them a definition. In

Tuples

Listing 16: creating a tuple

```
months = ('January','February','March','April','May','June',\
'July','August','September','October','November','  December')
```

- Note that the 'át the end of the first line carries over that line of code to the next line. It is usefull way of making big lines more readable.
- Technically you don't have to put those parentheses therebut it stops python from getting things confused.
- You may have spaces after the commas if you feel it necessary - it doesn't really matter.

Python then organises those values using a numbered index — starting from zero — in the order that you entered them in:

```
print months[2] # prints March
```

Lists

Lists are extremely similar to tuples. Lists are modifiable/mutable, so their values can be changed. Most of the time we use lists, not tuples, because we want to easily change the values of things if we need to.
Lists are defined very similarly to tuples:

Listing 17: Creating a List

```
cats = ['Tom', 'Snappy', 'Kitty', 'Jessie', 'Chester']
```

As you see, the code is exactly the same as a tuple, EXCEPT that all the values are put between square brackets, not parentheses.

Listing 18: Recalling items from a list

```
print cats[2] # an element
print cats[0:2] # a range of elements
```

Lists: adding elements

To add a value to a list, you use the ’append()’ function:

Listing 19: Adding items to a list

```
cats.append(’Catherine’)
```

Listing 20: Using the append function

```
#add a new value to the end of a list:
list_name.append(value-to-add)

#e.g. to add the number 5038 to the list ’numbers’:
numbers.append(5038)
```

Lists: deleting elements

Similarly we can delete an element from a list using the **del** function:

Listing 21: Deleting an item

```
#Remove your 2nd cat, Snappy. Woe is you.
del cats[1]
print cats
print cats[1]
```

```
>>> del cats[1]
>>> print cats[1]
Kitty
>>> print cats
['Tom', 'Kitty', 'Jessie', 'Chester']
>>>
```

Creating a Dictionary

To define a dictionary, use braces {, }

Listing 22: Creating a dictionary

```
#Make the phone book:
phonebook = {'Andrew Parson':8806336, \
'Emily Everett':6784346, 'Peter Power':7658344, \
'Lewis Lame':1122345}
```

To add a new element:

Listing 23: adding a new element

```
phonebook['Gingerbread Man'] = 1234567
```

To delete an element

```
del phonebook['Andrew Parson']
```

To print the dictionary

```
print phonebook
```

A Few Examples and Functions of Dictionaries I

```
Functions of dictionaries
#A few examples of a dictionary

#First we define the dictionary
#it will have nothing in it this time
ages = {}

#Add a couple of names to the dictionary
ages['Sue'] = 23
ages['Peter'] = 19
ages['Andrew'] = 78
ages['Karren'] = 45

#Use the function has_key() -
#This function takes this form:
#function_name.has_key(key-name)
#It returns TRUE
#if the dictionary has key-name in it
#but returns FALSE if it doesn't.
#Remember - this is how 'if' statements work -
#they run if something is true
#and they don't when something is false.
```

A Few Examples and Functions of Dictionaries II

```
if ages.has_key('Sue'):
    print "Sue is in the dictionary. She is", \
ages['Sue'], "years old"

else:
    print "Sue is not in the dictionary"

#Use the function keys() -
#This function returns a list
#of all the names of the keys.
#E.g.
print "The following people are in the dictionary:"
print ages.keys()

#You could use this function to
#put all the key names in a list:
keys = ages.keys()

#You can also get a list
#of all the values in a dictionary.
#You use the values() function:
print "People are aged the following:", \
ages.values()
```

A Few Examples and Functions of Dictionaries III

```
#Put it in a list:
values = ages.values()

#You can sort lists, with the sort() function
#It will sort all values in a list
#alphabetically, numerically, etc...
#You can't sort dictionaries -
#they are in no particular order
print keys
keys.sort()
print keys

print values
values.sort()
print values

#You can find the number of entries
#with the len() function:
print "The dictionary has", \
len(ages), "entries in it"
```

For Loop I

The for loop does something for every value in a list:

Listing 24: The for Loop

```
# Example 'for' loop
# First, create a list to loop through:
newList = [45, 'eat me', 90210, "The day has come, \
       the walrus said, to speak of many things", -67]

# create the loop:
# Goes through newList, and seqentially puts each
# bit of information into the variable value,
# and runs the loop
for value in newList:
    print value
```

When the loop executes, it runs through all of the values in the list mentioned after 'in'. It then puts them into value, and executes through the loop, each time with value being something different.

For Loop II

Listing 25: A for Loop Example

```
# cheerleading program
word = raw_input("Who do you go for? ")

for letter in word:
    call = "Gimme a " + letter + "!"
    print call
    print letter + "!"

print "What does that spell?"
print word + "!"
```

- Strings are just lists with lots of characters.
- The program went through each of the letters (or values) in word, and it printed them onscreen.

Outline

Interactive Python

- Python statements can be run interactively in a Python shell
- The “best” shell is called IPython
- Sample session with IPython:

```
Unix/DOS> ipython
...
In [1]:3*4-1
Out[1]:11

In [2]:from math import *

In [3]:x = 1.2

In [4]:y = sin(x)

In [5]:x
Out[5]:1.2

In [6]:y
Out[6]:0.93203908596722629
```

Editing capabilities in IPython

- Up- and down-arrays: go through command history
- Emacs key bindings for editing previous commands
- The underscore variable holds the last output

```
In [6]:y
Out[6]:0.93203908596722629

In [7]:_ + 1
Out[7]:1.93203908596722629
```

TAB completion

- IPython supports TAB completion: write a part of a command or name (variable, function, module), hit the TAB key, and IPython will complete the word or show different alternatives:

```
In [1]: import math

In [2]: math.<TABKEY>
math.__class__          math.__str__            math.frexp
math.__delattr__        math.acos               math.hypot
math.__dict__           math.asin               math.ldexp
...
or
In [2]: my_variable_with_a_very_long_name = True

In [3]: my<TABKEY>
In [3]: my_variable_with_a_very_long_name
```

You can increase your typing speed with TAB completion!

Outline

A first math script

```
#!/usr/bin/env python
import sys, math       # load system and math module
r = float(sys.argv[1]) # extract the 1st command-line argument
s = math.sin(r)
print "Hello, World! sin(" + str(r) + ")=" + str(s)
```

Loading data into lists

- Read input file into list of lines

```
lines = ifile.readlines()
```

- Now the 1st line is `lines[0]`, the 2nd is `lines[1]`, etc.
- Store `x` and `y` data in lists:

```
# go through each line,
# split line into x and y columns

x = []; y = []   # store data pairs in lists x and y

for line in lines:
    xval, yval = line.split()
    x.append(float(xval))
    y.append(float(yval))
```

See `python/datatrans2.py`

Listing 26: A Least Square Example

```
#!/usr/bin/env python
import sys

try:
    infilename = sys.argv[1];  outfilename = sys.argv[2]
except:
    print "Usage:",sys.argv[0], "infile outfile"; sys.exit(1)

ifile = open(infilename, 'r')  # open file for reading
lines = ifile.readlines()      # read file into list of lines
print lines
ifile.close()

# go through each line and split line into x and y columns:
x = []; y = []   # store data pairs in two lists x and y
for line in lines:
    print line
    xval, yval = line.split()
    x.append(float(xval)); y.append(float(yval))
```

datatrans II

```
from math import exp
def myfunc(y):
    return (y**5*exp(-y) if y >= 0 else 0.0)

ofile = open(outfilename, 'w') # open file for writing
for i in range(len(x)):
    fy = myfunc(y[i])  # transform y value
    ofile.write('%g  %12.5e\n' % (x[i],fy))
ofile.close()
```

Executing Datatrans

- Method 1: write just the name of the scriptfile:

```
./datatrans2.py infile outfile
# or
datatrans2.py infile outfile
```

  if . (current working directory) or the directory containing datatrans1.py is in the path

- Method 2: run an interpreter explicitly:

```
python datatrans2.py infile outfile
```

  Use the first python program found in the path This works on Windows too (method 1 requires the right assoc/ftype bindings for .py files)

Loop over list entries

- For-loop in Python:

```
for i in range(start,stop,inc):
    ...
for j in range(stop):
    ...
```

generates

```
i = start, start+inc, start+2*inc, ..., stop-1
j = 0, 1, 2, ..., stop-1
```

- Loop over (x,y) values:

```
ofile = open(outfilename, 'w') # open for writing

for i in range(len(x)):
    fy = myfunc(y[i])  # transform y value
    ofile.write('%g  %12.5e\n' % (x[i], fy))

ofile.close()
```

Computing with Arrays

- x and y in datatrans2.py are lists
- We can compute with lists element by element (as shown)
- However: using Numerical Python (NumPy) arrays instead of lists is much more efficient and convenient
- Numerical Python is an extension of Python: a new fixed-size array type and lots of functions operating on such arrays

A first glimpse at NumPy

- Import

```
from scitools import *
from scitools.numpytools import *
x = sequence(0, 1, 0.001)  # 0.0, 0.001, 0.002, ..., 1.0
x = sin(x)                 # computes sin(x[0]), sin(x[1])
```

- `x=sin(x)` is 13 times faster than an explicit loop:

```
for i in range(len(x)):
    x[i] = sin(x[i])
```

because sin(x) invokes an efficient loop in C

More on NumPy Arrays

- Multi-dimensional arrays can be constructed:

```
x = zeros(n, Float)  # array with indices 0,1,...,n-1
x = zeros((m,n), Float)    # two-dimensional array
x[i,j] = 1.0               # indexing
x = zeros((p,q,r), Float)  # three-dimensional array
x[i,j,k] = -2.1
x = sin(x)*cos(x)
```

- We can plot one-dimensional arrays:

```
from scitools.all import *  # import numpy and much of sci
x = sequence(0, 2, 0.1)
y = x + sin(10*x)
plot(x, y)
```

- NumPy has lots of math functions and operations
- SciPy is a comprehensive extension of NumPy
- NumPy + SciPy is a kind of Matlab/Octave replacement for many people

Least Squares Example using Python.Scitools I

Listing 27: A Least Square Example

```
#!/usr/bin/env python
"""Demonstrate fitting a stright line to data with NumPy."""
# generate random data and fit a straight line

import sys
try:
    n = int(sys.argv[1])   # no of data points
except:
    n = 20

from scitools.all import *  # import numpy and much of scitool

# compute data points in x and y arrays:
# x in (0,1) and y=-2*x+3+eps, where eps is normally
# distributed with mean zero and st.dev. 0.25.
random.seed(20)
x = linspace(0.0, 1.0, n)
noise = random.normal(0, 0.25, n)
a_exact = -2.0; b_exact = 3.0
y_line = a_exact*x + b_exact
```

Least Squares Example using Python.Scitools II

```
y = y_line + noise

# create least squares system:
A = array([x, zeros(n)+1])
A = A.transpose()
result = linalg.lstsq(A, y)
# result is a 4-tuple, the solution (a,b) is the 1st entry:
a, b = result[0]

# plot:
plot(x, y, 'o',
     x, y_line, 'r',
     x, a*x + b, 'b',
     legend=('data points', 'original line', 'fitted line'),
     title='y = %g*x + %g: fit to y = %g*x + %s + normal noise
            (a, b, a_exact, b_exact),
     hardcopy='tmp.ps')
# make a PNG plot too:
hardcopy('tmp.png')

# alternative method: numpy.polyfit
a, b = polyfit(x, y, 1)
figure()
```

Least Squares Example using Python.Scitools III

```
plot(x, y, 'o',
     x, y_line, 'r',
     x, a*x + b, 'b',
     legend=('data points', 'original line', 'fitted line'),
     title='y = %g*x + %g: polyfit(x,y,1) to y = %g*x + %$ + n
            (a, b, a_exact, b_exact),
     hardcopy='tmp2.ps')
```

Outline

Defining Classes

The syntax of a class is as follows

```
# Defining a class
class class_name:
    [statement 1]
    [statement 2]
    [statement 3]
    [etc]
```

Shape Class I

Listing 28: Shape Class

```
#An example of a class
class Shape:
# constructor
    def __init__(self,x,y):
        self.x = x
        self.y = y
# data member storing description
    description = "This shape has not been described yet"
# data member storing author
    author = "Nobody has claimed to make this shape yet"
# returns area of the shape
    def area(self):
        return self.x * self.y
# returns perimeter of the shape
    def perimeter(self):
        return 2 * self.x + 2 * self.y

# returns description of the shape
    def describe(self,text):
        self.description = text
```

Shape Class II

```
# returns author
    def authorName(self,text):
        self.author = text
# scale the shape
    def scaleSize(self,scale):
        self.x = self.x * scale
        self.y = self.y * scale
```

Explanations

- The function called `__init__` is run when we create an instance of `Shape`.
- `self` is how we refer to things in the class from within itself (equivalent to `this` in C++).
- `self` is the first parameter in any function defined inside a class. Any function or variable created on the first level of indentation (that is, lines of code that start one TAB to the right of where we put class `Shape` is automatically put into `self`.
- To access these functions and variables elsewhere inside the class, their name must be preceeded with self and a full-stop (e.g. `self.variable_name`).

Object Orient Programming

- When we first describe a class, we are defining it (like with functions)
- The ability to group similar functions and variables together is called encapsulation
- The word ’class’ can be used when describing the code where the class is defined (like how a function is defined), and it can also refer to an instance of that class
- A variable inside a class is known as an Attribute
- A function inside a class is known as a method
- A class is in the same category of things as variables, lists, dictionaries, etc. That is, they are objects
- A class is known as a ’data structure’ - it holds data, and the methods to process that data.

Example with Shape class I

Listing 29: Using the Shape Class

```
rectangle = Shape(100,45)

#finding the area of your rectangle:
print rectangle.area()

#finding the perimeter of your rectangle:
print rectangle.perimeter()

#describing the rectangle
rectangle.describe("A wide rectangle, more than twice\
 as wide as it is tall")

#making the rectangle 50% smaller
rectangle.scaleSize(0.5)

#re-printing the new area of the rectangle
print rectangle.area()
```

Class Inheritance

Listing 30: Inheriting from Shape

```
# load definition of class Shape
from shape import Shape

# define subclass Square
class Square(Shape):
    def __init__(self,x):
        self.x = x
        self.y = x
```

- We introduced here the notion of modules. We needed first to load the class Shape from shape.py

Outline

Bindings

- Many (e.g. C++) libraries have Python bindings: objects and functions can be created or called from Python
- Qt and VTK are such libs

Qt Hello World

Listing 31: First Qt Hello Example

```
#
# qthello1.py
#
import sys
from qt import *

app=QApplication(sys.argv)
button=QPushButton("Hello World", None)
app.setMainWidget(button)
button.show()
app.exec_loop()
```

Qt Hello World 2

Listing 32: Second Qt Hello Example

```
import sys
from qt import *
class HelloButton(QPushButton):
    def __init__(self, *args):
        apply(QPushButton.__init__, (self,) + args)
        self.setText("Hello World")
class HelloWindow(QMainWindow):
    def __init__(self, *args):
        apply(QMainWindow.__init__, (self,) + args)
        self.button=HelloButton(self)
        self.setCentralWidget(self.button)

def main(args):
    app=QApplication(args)
    win=HelloWindow()
    win.show()
    app.connect(win.button, SIGNAL("clicked()"),
                app, SLOT("quit()"))
    app.exec_loop()

main(sys.argv)
```

More on Python Qt bindings

- Define and use your own slots:

  http://techbase.kde.org/Development/Tutorials/Python_introduction_to_signals_and_slots

Outline

What do we do with Python?

- Wrap C/C++ code to easily use script languages
- Python is perfect for rapid prototyping
  - Rapid definition of classes, functions,...
  - Attention : could be slow
- Python is easy to profile
  - module time.clock() for computing delays
  - Python Profiler
- it is possible to rewrite the slow parts in C
  - Write the Python code
  - Profiler
  - ReWrite the slow parts in C
- Easy to say, trickier to do

interlanguage frameworks

- Swig : `http://www.swig.org`
- Boost.Python `http://www.boost.org/doc/libs/1_40_0/libs/python/doc/`

There might be others, the objectives are similar, the process/steps may be different

What about Conputer Graphics?

- Many libraries have a Python interface
  - VTK
  - Paraview
  - Qt
  - ...

What about Scientific Computing?

- Most libraries have a Python interface
  - Trilinos
  - Getfem++
  - Fenics
  - ...
- Python offers wrappers for many “basic” computing libraries for example through the `scipy` library but not only.
  - Linear Algebra (blas, lapack, arpack, ...)
  - FFT
  - Nonlinear solvers
- Python becomes a good replacement for Matlab and Octave.

What about Statistics?

- Many libraries/Software have a Python interface
  - R to Python and Python to R
  - Openturns
  - ...

What is SWIG?

- Swig = tool allowing for connecting C or C++ to a set of high level languages such as
  - script languages : Perl, PHP, Python, Tcl, Ruby,...;
  - other languages: C#, CLISP, Java, OCAML,...

`http://www.swig.org`

`http://www.python.org/doc/current/ext/ext.html`

"Extending Python," Guido van Rossum

We say that SWIG is a wrapper (connector).

Interface File

- We have `greet.cpp`
- We create `greet.i`
- Swig generates `greet_wrap.cxx` and `greet.py`

Interface File

One interfacing SWIG file `< greet.i >`:

```
%module mymodule  = name of the module
// code to be copied by in the interface C file
Directive %{ ... }%
Declarations
```

An Example

Listing 33: greet.cpp

```
char const* greet()
{
   return "hello, world";
}
```

Listing 34: greet.i

```
%module greet
%{
  char const* greet();
%}
  char const* greet();
```

Listing 35: hello.py

```
import greet
print  greet.greet()
```

How to use Swig?

SWIG:

```
$ swig -python -c++ greet.i
```

The command generates `greet_wrap.cxx` and `greet.py`.

`greet.py` is the module to import in python

```
$ swig -python greet.i
$ g++ -c greet.cpp
$ g++ -c greet_wrap.cxx -I/usr/include/python2.5
$ g++ -shared greet.o greet_wrap.o -o _greet.so
```

The name of the dynamic library must correspond to the name of the module with a “underscore” prefix

CMake and SWIG

Now CMake enters the game and simplifies our Life

Listing 36: CMakeLists.txt

```
find_package(SWIG )
Include( UseSWIG )

add_custom_command(OUTPUT greet_wrap.cxx
  COMMAND ${SWIG_EXECUTABLE}
  ARGS
  -python -c++
  ${PROJECT_SOURCE_DIR}/swig/greet.i)

SWIG_ADD_MODULE( greet python
  ${PROJECT_SOURCE_DIR}/swig/greet_wrap.cxx
  ${PROJECT_SOURCE_DIR}/greet.cpp)
```

Boost.Python

Listing 37: greet.cpp

```
char const* greet()
{
   return "hello, world";
}
```

Listing 38: greet_wrapper.cpp

```
#include <boost/python.hpp>

char const* greet();

BOOST_PYTHON_MODULE(greet)
{
    using namespace boost::python;
    def("greet", greet);
}
```

Listing 39: greet.py

```
import greet
print greet.greet()
```

Building the wrapper module

Listing 40: Building the wrapper

```
include_directories(
  ${CMAKE_CURRENT_SOURCE_DIR}
  ${CMAKE_CURRENT_SOURCE_DIR}/.. )

macro( add_pylib targetname )
add_library( ${targetname}
  MODULE
  ${targetname}_wrapper.cpp ../${targetname}.cpp)
target_link_libraries( ${targetname} ${Boost_LIBRARIES})
add_custom_command(OUTPUT ${targetname}.so
  COMMAND ln  ARGS -s lib${targetname}.so ${targetname}.so )
add_custom_target(${targetname}so DEPENDS ${targetname}.so)
add_dependencies(${targetname} ${targetname}so)
endmacro( add_pylib )

add_pylib( greet )
add_pylib( classes )
```

Listing 41: executing the wrapper

```
>> python greet.py
hello, world
```

A More Complicated Example

Listing 42: classes.hpp

```
#include <iostream>
#include <string>
class A {
 public:
  A() : h(0) {}
  A( double v ): h(v) {}
  virtual ~A();
  double get() const { return h;}
  void set(double value) { h=value; }
  virtual std::string Hi() { return "Hi"; }

  double h;
};
class B : public A {
  public:
  B() : A() {}
  B( double v ): A(v) {}
  ~B();
  std::string Hi() { return "Hello"; }
};
A* factory();
```

Testing in C++

Listing 43: classes.cpp

```
#include "classes.hpp"
A::~A() {  std::cout << "calling A::~A()"; }
B::~B() { std::cout << "calling B::~B()"; }
A* factory() { return new B; }
```

Listing 44: test

```
#include "classes.hpp"
int main(int argc, char** argv ) {
  A a; std::cout << "Say " << a.Hi() << "\n";
  B b; std::cout << "Say " << b.Hi() << "\n";
  A* c = factory(); std::cout << "Say " << c->Hi() << "\n";
}
```

Listing 45: Output

```
% boost/test_classes
Say Hi
Say Hello
Say Hello
calling B::~B()calling A::~A()calling A::~A()
```

Boost.Pythonize

Listing 46: Python wrapper

```
#include <boost/python.hpp>
using namespace boost::python;
#include "classes.hpp"
BOOST_PYTHON_MODULE(classes)
{
    class_<A>("A")
    .def(init<>())
    .def(init<double>())
    .def("Hi", &A::Hi)
    .add_property("h", &A::get, &A::set)
    .def_readwrite("h", &A::h)
    ;

    class_<B, bases<A> >("B")
    .def(init<>())
    .def(init<double>())
    .def("Hi", &B::Hi)
    ;

    def("factory", factory,
        return_value_policy<manage_new_object>());
}
```

Python example

Listing 47: ex.py

```
import classes,math
a = classes.A()
print "say Hi: "+a.Hi()
a.h=math.sin(1.0)
print "h=",a.h
```

More Information

These are just simple examples, for more information see: `file:///usr/share/doc/libboost1.39-doc/` or `http://www.boost.org`

- Constructors
- Class Data Members
- Class Properties
- Inheritance
- Class Virtual Functions
- Virtual Functions with Default Implementations
- Class Operators/Special Functions

Automatic generation of wrappers

there are projects to generate automatically the wrapper for Boost.Python. See e.g. Py++

How to use Python in C/C++?

Use high level Python functions:

Listing 48: Calling Python in C/C++

```
#include <Python.h>

int main(int argc, char *argv[])
{
  Py_Initialize();
  PyRun_SimpleString("from time import time,ctime\n"
                     "print 'Today is ',ctime(time())\n");
  Py_Finalize();
  return 0;
}
```

Listing 49: CMakeLists/txt

```
add_executable(extime time.cpp )
target_link_libraries( extime ${PYTHON_LIBRARIES})
```

Outline

References

Hans-Petter Langtangen
Python Scripting for Computational Science
Springer, 2005/2006

Computing in Science and Engineering
Special Issue “Python: Batteries included”
May/June 2007