Introduction to Python

Eduardo Rodrigues
University of Glasgow

Course given at Glasgow, 12-18 March 2008

Part I

General introduction to Python

Course outline

Part I: General introduction to Python

- working environment, basics
- main data types, functions, classes, modules

Part II: ROOT and RooFit from Python

- interfacing Python with C++ HEP software
- working with ROOT and RooFit in Python

Part III: GaudiPython

- the Gaudi software framework for ATLAS and LHCb
- basic functionality
- examples

Part I: general introduction to Python

I. What is Python? Why should I use it?

II. Environment set-up

III. Getting started

IV. Functions

V. Native data types

VI. Looping techniques

VI. Modules

VII. Files

VIII. Classes

IX. Errors and exceptions

X. Documentation

What is Python?

What is Python?

- An Object Oriented (OO) scripting language
- Interpreted – no compilation needed
- Dynamically typed – no explicit type declaration
- High-level language
- Everything is an object

Python is 2 things:

- An interpreter, for scripts, running from the command line
- An interactive shell that can evaluate any (Python) statement/expression

Why should I use Python?

- **Very easy to learn and use, powerful, elegant**
- **Fast learning curve**
- **Extensive set of modules “available on the market”**
- **Ideal for interactive work/analysis, testing, prototyping new ideas, …**
- **Allows to work with different applications at the same time (“glue”)**
- **Same code/script can run under Linux, Windows, Mac**
- **No edit-compile-run/debug cycle. Make changes and test on the spot**

- ***Not convinced yet? Relax, wait and see … ;-)***

Environment set-up

Setting up the environment

We will be using Python 2.5 !

Windows:

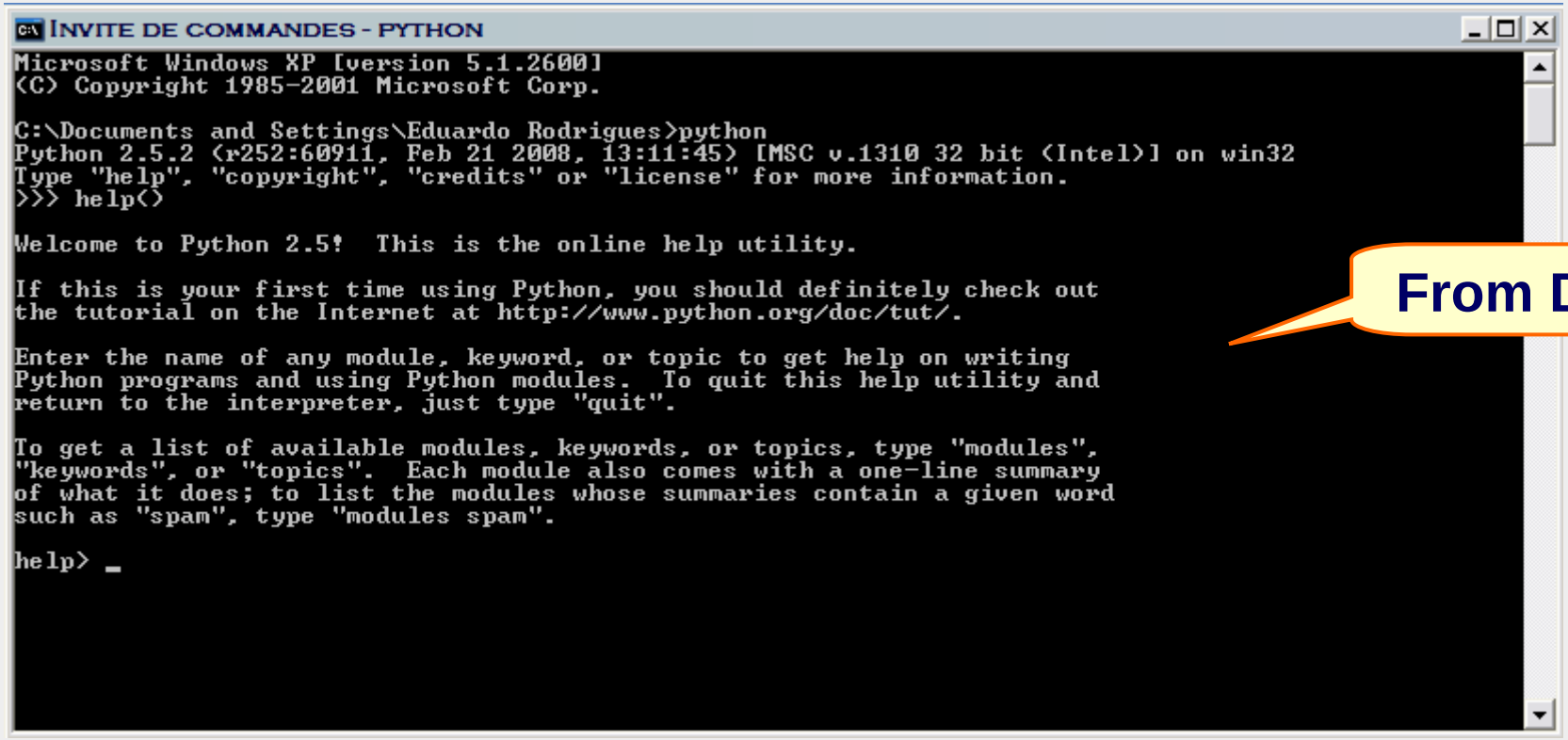
- Go to the download area at www.python.org
- Download the version of the installer you want and run it, following the straightforward instructions …
- Set up environment variables such as PYTHONPATH,

  add PYTHONPATH to PATH

Linux:

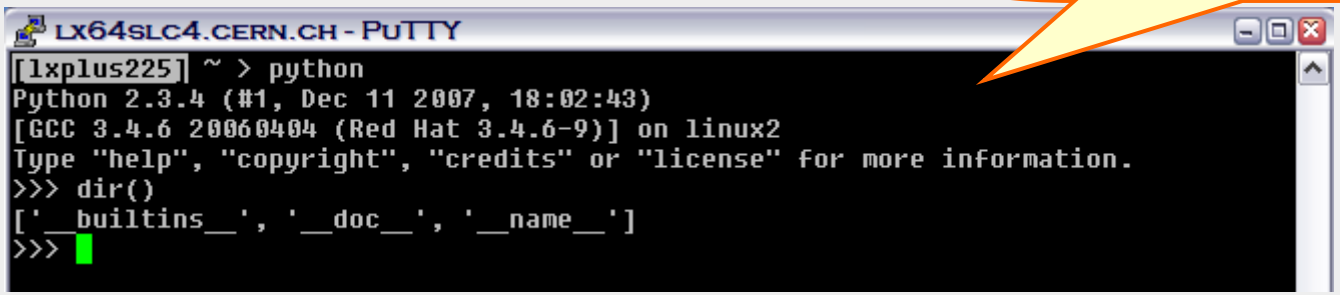
- Virtually installed on every machine
- Just try typing *python* at the prompt …

Invoking the interpreter (1/3)

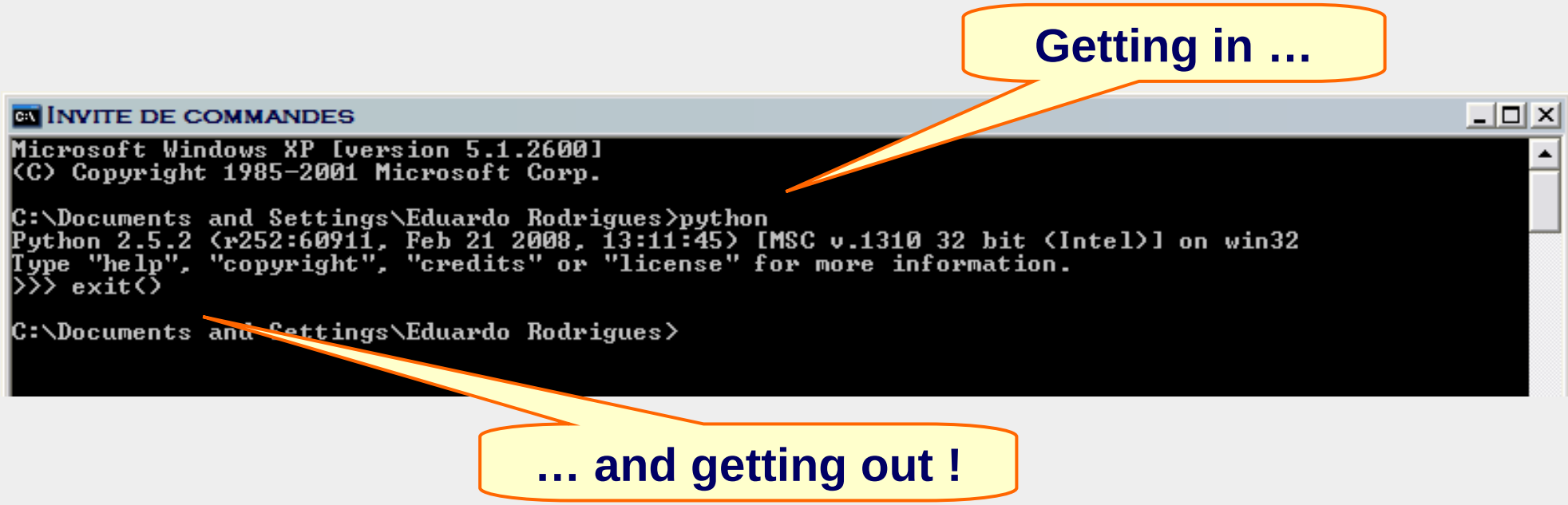


From DOS

From linux, at CERN



Invoking the interpreter (2/3)



- ✓ One can also exit the prompt with Control-D on Linux / Control-Z on Windows

- ➢ This is the most basic interactive mode
- ➢ Good for trivial tasks, but not ideal for anything else …

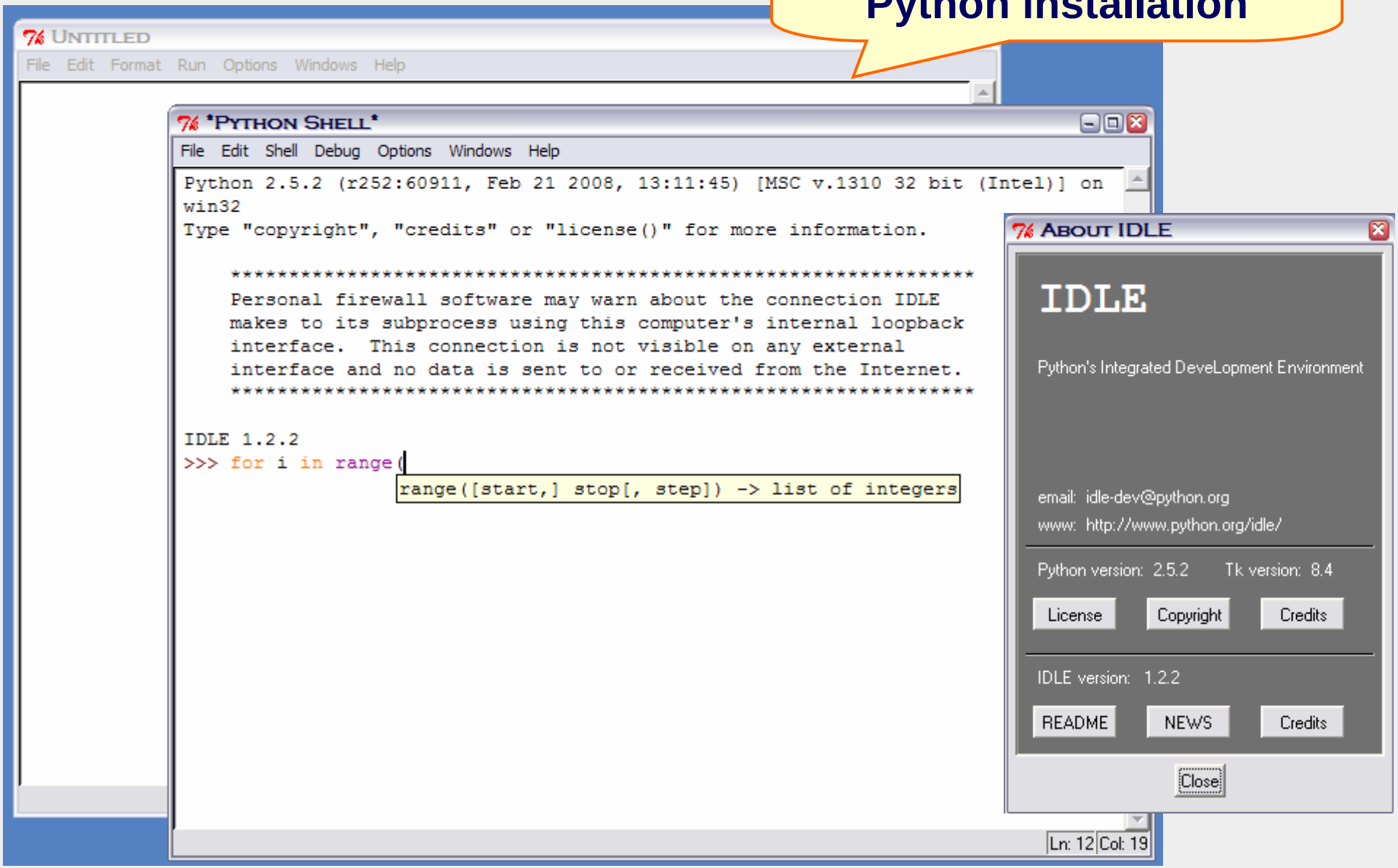
Invoking the interpreter (3/3)

```
# run a script
python my_program.py

python my_program.py > my_program.log

# run a script and enter the interactive mode
python –i my_program.py

# the interpreter has many options! Check them out with
python -h
```

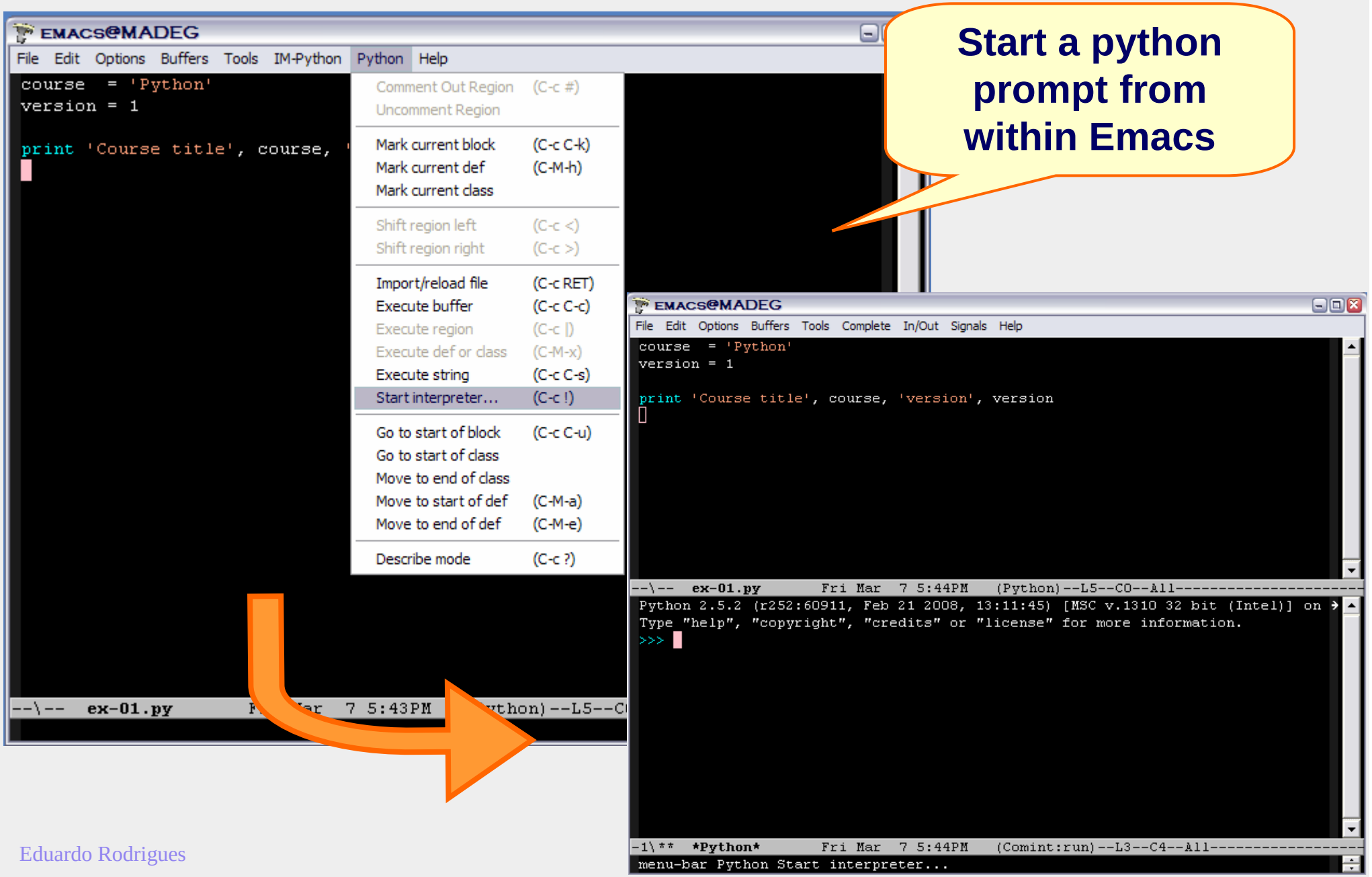
- Script files can be run in this mode
- And any (plain ASCII !) text editor can be used for file editing
- But a more interactive solution is “the way to go”
- Such as IDE …

The IDLE environment

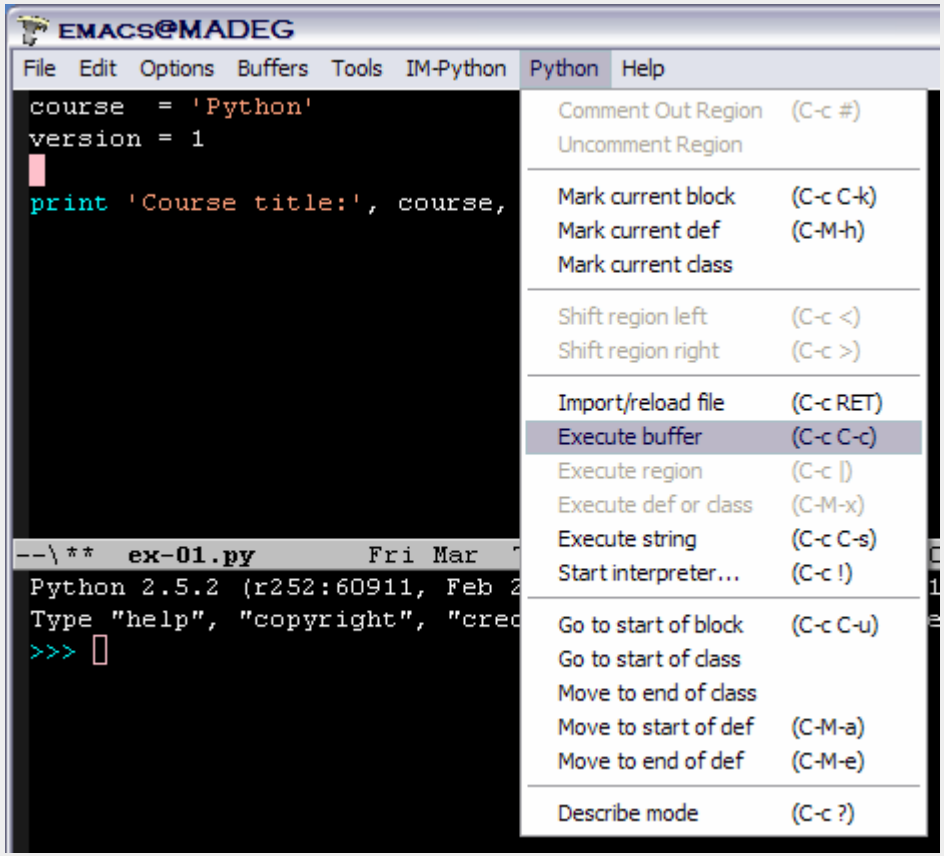
Comes with a standard Python installation



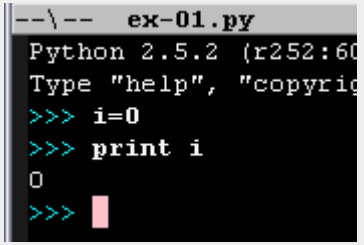
Python interpreter with Emacs (1/3)



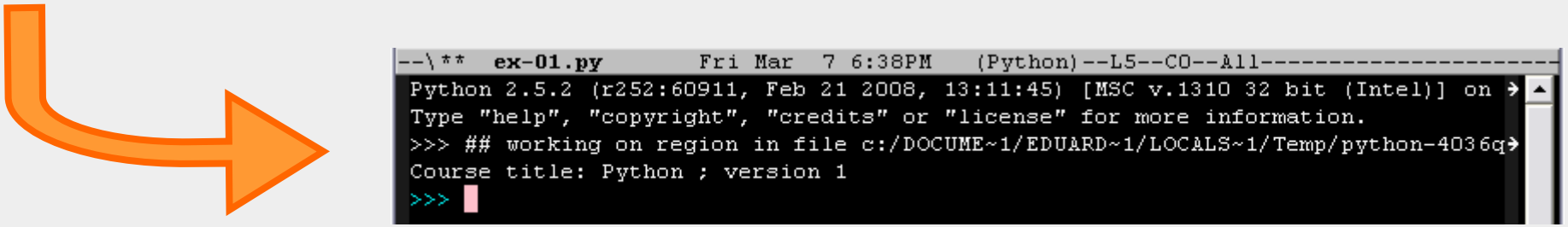
Python interpreter with Emacs (2/3)



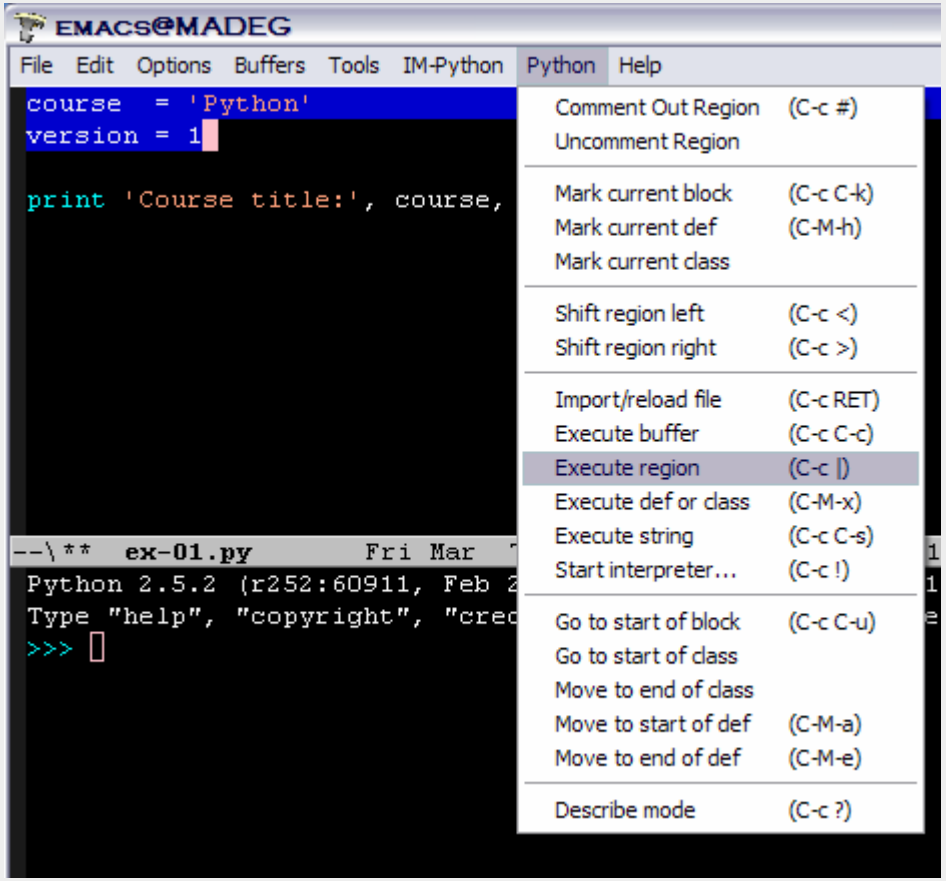
Simply at the prompt



Execute buffer (full script file) on the prompt

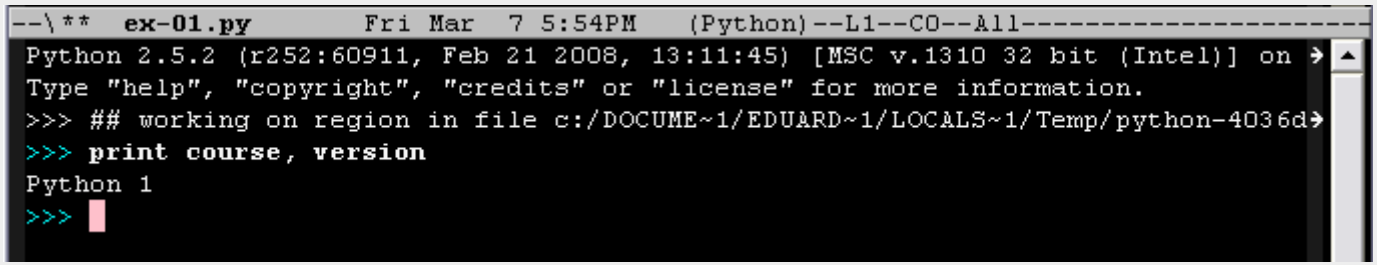


Python interpreter with Emacs (3/3)



Neat way of editing and running in the same environment

Execute a region (a part of the script file) on the prompt



Getting started

Python as a calculator (1/2)

```
>>> # this is a comment
>>> 2 + 2
4
>>> 3/2  # integer division returns the floor
1
>>> -3/2
-2
>>> 3/2.
1.5
>>> 1+2*3
7
>>> (1+2)*3
9
>>> 15 % 10
5
>>> print 5.4 % 2
1.4
>>> (6-4)**2
4
>>> (6-4)**2.
4.0
```

Operators

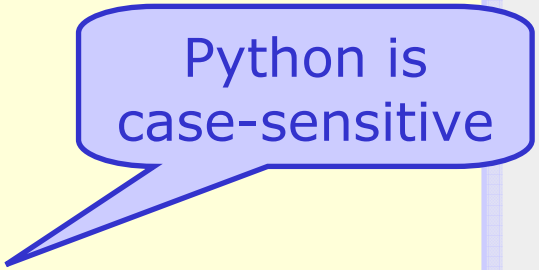
+	add
-	subtract
*	multiply
/	divide
%	remainder
**	exponentiate

Python as a calculator (2/2)

```
>>> 100.
100.0
# in interactive mode the last printed expression is assigned
# to the variable "_" ! This can simplify calculations …
>>> _ + 7.5
107.5
>>> _ * 1.026
110.295
>>> round(_,1)
110.3

>>> old_savings = 1000.
>>> _
110.3
>>> old_savings
1000.0
>>> _
1000.0
```

Assignment to variables

```
>>> width = 5
>>> length = 2
>>> area = width * length
>>> area
10

>>> bla = area
>>> bla
10

# same value simultaneously assigned to several variables
>>> x = y = z = 0
>>> x, y, z
(0, 0, 0)
# several variables assigned different values simultaneoously
>>> x, y, z = 1, 2, 3
>>> x, y, z
(1, 2, 3)

>>> x = 1 ; y = 2 ; z = 3
>>> x, y, z
(1, 2, 3)
```



Basic data types - numbers

```
>>> x = 123          # integer “int”

>>> x = 123l         # integer “long”; can use “l” or “L”

>>> x = 010          # octal number
>>> x
8

>>> x = 0xF          # hexadecimal number
>>> x
15

>>> x = 123.         # double float (“single” float does not exist)
>>> x = 1.23E+2      # double in scientific format; can also use “e”

>>> x = 1 + 2j           # complex number; can use “j” or “J”
>>> x = complex(1,2)     # complex number
>>> x
(1+2j)
>>> x.real, x.imag
(1.0, 2.0)
>>> x.conjugate()
(1-2j)

>>> x = None         # null type
```

Basic data types - booleans

```
>>> x = True
>>> y = False

>>> x
True

# avoid using built-in types as variable names !
# Python will happily accept it …
# but you the lost the built-in type definition !
>>> True
True
>>> False
False

>>> True = 123
>>> True
123
```

- Particularly relevant for *comparisons* …

Basic data types – strings (1/4)

```
>>> name = 'My first string.'
>>> name
'My first string.'

>>> name = "My first string."
>>> name
'My first string.'

>>> "It can't be so simple !"
"It can't be so simple !"
>>> 'It can\'t be so simple !'
"It can't be so simple !"

>>> '"Yes it is", I would say.'
'"Yes it is", I would say.'
>>> "\"Yes it is\", I would say."
'"Yes it is", I would say.'

>>> 'Isn\'t this pushing the "escaping" to extremes?'
'Isn\'t this pushing the "escaping" to extremes?'
>>> "Isn't this pushing the \"escaping\" to extremes?"
'Isn\'t this pushing the "escaping" to extremes?'
```

Strings can be defined with single or double quotes

A string containing the same quote as the enclosing one needs to have it escaped

Single quotes are more important than double quotes

Basic data types – strings (2/4)

```
# the backslash indicates continuation on the next line
>>> bla = ‘A multi-line ’ \
... ‘string definition’
>>> bla
‘A multi-line string definition’

# the same can be obtained with concatenation
>>> bla = ‘A multi-line ’ +\
... ‘string definition’
>>> bla
‘A multi-line string definition’

# a multi-line string with triple quotes; can also use '''
>>> abc = “““a
... b
...  c.”””
>>> abc
‘a\nb\n c.’
>>> print abc # notice the space before the “c”
a
b
 c.
```

Basic data types – strings (3/4)

```
# concatenation
>>> ‘Eduardo’ + ‘ Rodrigues’
‘Eduardo Rodrigues’

>>> ‘Eduardo’ ‘ Rodrigues’
‘Eduardo Rodrigues’
>>> ‘Eduardo’ “ Rodrigues”
‘Eduardo Rodrigues’

# multiplication
>>> ‘ah ‘ * 3
‘ah ah ah ’

# some string methods
>>> name = ‘Eduardo Rodrigues’

>>> name.upper()
‘EDUARDO RODRIGUES’

>> name.replace( ‘ ’, ‘,’ )
‘Eduardo,Rodrigues’
>>> name
‘Eduardo Rodrigues’
```

Strings have many more methods.
To be continued …

Basic data types – strings (4/4)

```
>>> name = 'Eduardo Rodrigues'

# string indexing
>>> name[3]     # gives the 4th character (counting starts at 0)
'a'
>>> name[-4]     # gives the 4th character from the end
'g'

# string slicing
>>> name[1:3]
'du'
>>> name[:3]    # gives the first 3 characters; same as name[0:3]
'Edu'
>>> name[3:]    # gives everything but the first 3 characters
'ardo Rodrigues'
# consequence: name[:i] + name[i:] returns the original string
>>> name[1:3]
'du'
>>> name[-3:]    # gives the last 3 characters
'ues'
>>> name[:-3]    # gives everything but the last 3 characters
'Eduardo Rodrig'

>>> name[10:2]
''
```

Introducing the print statement

```
>>> print ‘Eduardo Rodrigues’

>>> print “Eduardo Rodrigues”

>>> print “So far ...” ‘so good.’  # both strings are concatenated
So far ...so good.
>>> print “So far ...”, ‘so good.’ # watch the extra space !
So far ... so good.

>>> print ‘’

>>> print ‘Hi ’ \
... ‘there !’
Hi there !

>>> print ‘a =’, 2
a = 2

# formatting “à la C” – the other usual formatting letters exist
>>> print ‘Result:%8.2f’ % 1.456
Result:    1.46
>>> print ‘Result:%8.2f +/-%5.2f’ % ( 1.456, 0.035 )
Result:    1.46 +/- 0.04
```

Basics of flow control - if, elif, else

```
>>> yesNo = True
>>> if yesNo:
...     print ‘Yes’
... else:
...     print ‘No’
...
Yes

>>> if yesNo: print ‘Yes’
... else: print ‘No’
...
Yes
```

The “else” is optional

```
>>> if x < 0 :
...     print ‘Negative’
... elif x == 0 :
...     print ‘Zero’
... else :
...     print ‘Positive’
```

The number of “elif” is arbitrary

Introspection (1/2)

One of the great benefits of using Python !

The magic function “dir”

```
# on a fresh new Python prompt:
>>> dir()
['__builtins__', '__doc__', '__name__']
```

```
>>> help(dir)
Help on built-in function dir in module __builtin__:

dir(...)
    dir([object]) -> list of strings

    Return an alphabetized list of names comprising (some of) the attributes
    of the given object, and of attributes reachable from it:

    No argument:  the names in the current scope.
    Module object:  the module attributes.
    Type or class object:  its attributes, and recursively the attributes of
        its bases.
    Otherwise:  its attributes, its class's attributes, and recursively the
        attributes of its class's base classes.
```

Introspection (2/2)

- Python can ask its objects all sorts of questions

```
dir( <object> )

type( <object> )

isinstance( <object>, <type specification> )
```

```
# examples:
dir( 1 ), dir( list ), dir( 'a string' )

type( 1 ), type( 1L ), type( [] ), type( 'a string' )

isinstance( 1, int ), isinstance( [], list )
```

- There is also a "helper" function:

```
help( <object> )
```

Some common built-in functions

- Conversion functions: to integer, longs, floats, strings, ...

```
int( 4. ), int( -3.45E+5 )
long( 10.6 ), long( '123' )

int( '10', 2 )     # conversion from string allows a base
long( '17', 8 )

float( 10L ), float( '123.45' )

str( 123 )

bool(), bool( 0 ), bool( '' )     # all return False
bool( 1 ), bool( '123' )          # all return True
```

- Length of a string, number of items of a sequence, etc.

```
len( '123' )
```

- Do not forget to try `help()` on these functions ...

Reserved keywords

and	elif	global	or
assert	else	if	pass
break	except	import	print
class	exec	in	raise
continue	finally	is	return
def	for	lambda	try
del	from	not	while

- Not too many compared e.g. with C++ …

Functions

Functions (1/2)

- Python has functions, just like any other language
- Defined by the keyword `def`
- No return type specified, nor (optional) parameters type

```
# definition of a simple function
def SumOfSquares( x, y ):
    return x*x + y*y
```

Indentation matters !

```
# call the function
>>> a = 2
>>> b = 3
>>> c = SumOfSquares( a, b )
>>> c
13
```

Note on indentation:

- Python's way of defining blocks, i.e groups of statements
- Be careful with indentation: spaces are not the same as tabs !
- It is the amount of indentation that defines what is in which block

Functions (2/2)

- **Function overloading:**

```
# one can assign a function to a “variable”
>>> radius = SumOfSquares
>>> type( radius )
<type ‘function’>

>>> radius( 1, 1 )
2
>>> radius( 1, 1. )
2.0
```

One gets function overloading for free !

- **Recursive functions:**

```
def fibonacci( n ):
    if n < 3 :
        return 1
    return fibonacci( n-1 ) + fibonacci( n-2 )
```

Doc strings, help() and __doc__

- The help(<object>) helper function returns the documentation string (“doc string”) of objects
- Any literal string defined at the top of a function/class/method/module definition is taken to be its doc string

```
def SumOfSquares( x, y ):
    """Returns the sum of the squares.

    Takes 2 number arguments."""
    return x*x + y*y
```

```
>>> help(SumOfSquares)
Help on function SumOfSquares in module __main__:

SumOfSquares(x, y)
    Returns the sum of the squares.

    Takes 2 number arguments.

>>> print SumOfSquares.__doc__
Returns the sum of the squares.

   Takes 2 number arguments.
```

Native data types

Sequences

Basics:

- Sequences can be seen as sophisticated arrays
- Some are mutable, others immutable (e.g. strings)
- Most useful! You will hardly ever write a script that does not use a kind of sequence at some point …

Main sequences in Python:

- Lists, tuples

- Dictionaries are slightly different, more like associative arrays

Lists (1/3)

- Can be seen as ***mutable*** arrays
- List items may be heterogeneous, i.e. may be of different type

```
# create an empty list – 2 ways
>>> l1 = []
>>> l2 = list()

>>> l1
[]
>>> l1 == l2
True

# create a list with a few elements (of different types)
>>> x = 1. + 2.5j
>>> l = [ 1, 2, 3, 'a', x, True, None ]
>>> l
[ 1, 2, 3, 'a', (1+2.5j), True, None ]

>>> len(l)    # get the number of items
7
```

Lists (2/3)

```
# adding elements to a list
>>> l = [ 1, 2, 3 ]
>>> l.extend( [ 4 ] )
>>> l
[1, 2, 3, 4]
>>> l.extend( [ 5, 6 ] )
>>> l
[1, 2, 3, 4, 5, 6]

>>> l = [ 1, 2, 3 ]
>>> l.append( 4 )
>>> l
[1, 2, 3, 4]
>>> l.append( [ 5, 6 ] )
>>> l
[1, 2, 3, 4, [5, 6]]        # example of a nested list
```

"extend" method takes *iterables* as input

"append" takes *objects* as input

- Objects of type `list` have many methods. Check them with e.g. `dir(list)`
- The append and pop methods are very useful to create stacks or queues

Lists (3/3)

```
>>> l = [ 3, 2, 1, 0 ]

# concatenation and replication
>>> l + [ -1, -2 ]
[3, 2, 1, 0, -1, -2]
>>> [ 1, 2] * 3
[1, 2, 1, 2, 1, 2]

# access to elements or slices just like with strings ...
>>> l[0]
3
>>> l[1:3]
[2, 1]
>>> l[-1]
0
# ... though one can change each list element, unlike with strings
>>> l[0] = [ 'a', 'b' ]
>>> l
[['a', 'b'], 2, 1, 0]

# delete element(s)
>>> del l[0]
>>> del l[:2]
>>> l
[0]
```

Tuples (1/3)

- Can be seen as ***immutable*** arrays
- Tuple items may be heterogeneous, i.e. may be of different type

```
# create an empty tuple – 2 ways
>>> t1 = ()
>>> t2 = tuple()

>>> t1
()
>>> t1 == t2
True

# create a tuple with a single element – watch the comma !
>>> t3 = ( 0, )
>>> t4 = 0,
# create a list with a few elements (of different types)
>>> x = 1. + 2.5j
>>> t = ( 1, 2, 3, 'a', x, True, None )
>>> t
( 1, 2, 3, 'a', (1+2.5j), True, None )

>>> len(t)    # get the number of items
7
```

Tuples (2/3)

```
 >>> t = ( 3, 2, 1, 0 )

 # concatenation and replication
 >>> t + ( -1, -2 )
 (3, 2, 1, 0, -1, -2)
 >>> ( 1, 2) * 3
 (1, 2, 1, 2, 1, 2)

 # access to elements or slices just like with lists ...
 >>> t[0]
 3
 >>> t[1:3]
 (2, 1)
 >>> t[-1]
 0
# ... though one cannot change elements nor delete them !

# delete the whole tuple
 >>> del t
```

Tuples (3/3)

```
# tuple packing ...
>>> t = 1, 2, 3
>>> t
(1, 2, 3)
 >>> type(t)
<type 'tuple'>

# ... and unpacking
>>> x, y, z = t
>>> x
1
>>> y
2
>>> z
3

# swapping variables contents
>>> a, b, = 1, 2
>>> a, b = b, a
>>> print a, b
2 1
```

Dictionaries (1/3)

- Can be seen as ***labelled mutable*** arrays,

  or an unordered set of pairs (key, value)
- Dictionaries are indexed by key (sequences indexed by numbers)
  - keys have to be immutable objects of any kind

```
# create an empty dictionary – 2 ways
>>> d1 = {}
>>> d2 = dict()

>>> d1
{}
>>> d1 == d2
True

# create a dictionary with a few elements (of different types)
>>> d = { 'None': 0, 'a': 1, 'b' : 'second' }
>>> d
{'a': 1, 'None': 0, 'b' : 'second'}

>>> len(d)    # get the number of items
3
```

Dictionaries (2/3)

```
>>> tel = { ‘Me’: 1000, ‘You’: 2000, ‘NextDoor’ : 3000 }

# checking the keys and the values
>>> tel.keys()
[‘Me’, ‘You’, ‘NextDoor’]
>>> tel.values()
[1000, 2000, 3000]

>>> tel.has_key( ‘NextDoor’ )
True
>>> tel.has_key( ‘Blabla’ )
False
>>> ‘Me’ in tel
True

# accessing, changing, adding information
>>> tel[ ‘You’ ]
2000
>>> tel[ ‘You’ ] = 4000
>>> tel [ ‘You’ ]
>>> 4000
>>> tel[ ‘Secret’] = 9999
>>> tel
{'Me': 1000, 'You': 4000, 'Secret': 9999, 'NextDoor': 3000}
```

Dictionaries (3/3)

```
>>> d = {‘Me’: 1000, ‘You’: 4000, ‘NextDoor’: 3000, ‘Secret’: 9999 }

# deletion and “resetting”
>>> del d[ ‘Secret’ ]
>>> d
{'Me': 1000, 'You': 4000, 'NextDoor': 3000 }
>>> d.clear()
>>> d
{}
>>> del d
```

- Objects of type `dict` have many methods. Check them with e.g. `dir(dict)`
- In particular it provides handy methods to iterate over keys and values …

Looping techniques

while

```
>>> i = 0
>>> while i < 10 :
...     print i,
...     i += 1
...
0 1 2 3 4 5 6 7 8 9
>>> i
10
```

Notice the trailing comma

Notice the "+=" à la C

```
>>> l = [ 0, 1, 2, 3, 4, 5 ]
>>> while l:
...     l.pop()
...
5
4
3
2
1
0
>>> l
[]
```

Simple way of iterating over elements of a sequence

for

- Looping over a sequence of numbers

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(1, 4)
[1, 2, 3]
>>> range( -1, 6, 2)
[-1, 1, 3, 5]

>>> for i in range(1,4) :
...     print i,
1 2 3
```

- Powerful iterator over elements of sequences

```
my_list = [ 1, 2, 3, 'a', (1,2), True, None ]
>>> for elm in my_list :
...     print elm,
...
1 2 3 a (1, 2) True None

>>> for i in range( len(my_list) ) :
...     print i, my_list[i]
```

continue, break, pass

```
>>> for i in range ( 10, -100, -1 ) :
...     if i == 0:
...         continue       # continue with the next iteration
...     elif i % 2 == 0 :
...         print i,
...     elif i < -10 :
...         break          # break out of the enclosing loop
...
10 8 6 4 2 -2 -4 -6 -8 -10
```

```
# a trivial infinite loop
>>> while True :
...     pass    # the pass statement does ... nothing
...
```

Modules

What are modules?

- Programming languages have libraries; Python has modules
- Natural / simple way of grouping related functionality
- Main means of extending Python
- Modules define their namespace
- The Python Standard Library (see later) is a collection of modules

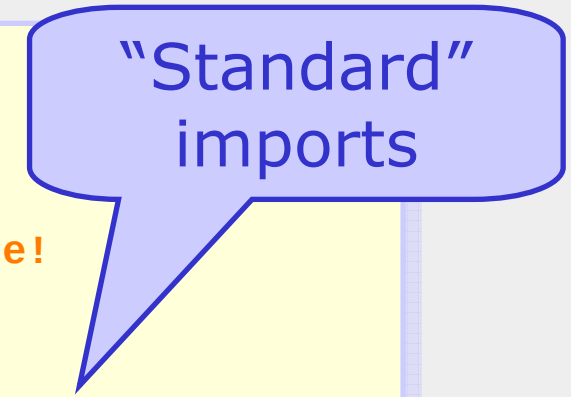
There are modules … and modules

Different types of modules:

- **Built-in modules:**
  - always available
  - popular examples are: sys, math, time
- **Standard library modules:**
  - come with a standard Python installation
  - examples: os, urllib, etc.
- **Third-party modules:**
  - examples: PyRoot, the module for ROOT
- **User-defined modules:**
  - up to the user to make them available ;-)

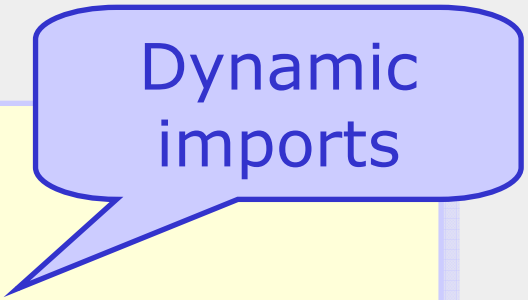
Importing modules (1/3)

```
# simple import
import math
import math, cmath

# simple import renaming the module – use with care!
import math as my_math

# import a method/function from a module
from math import sqrt
from math import sin, cos

# import a single method/function renaming it – use with care!
from math import sqrt as my_sqrt

# import all into to global namespace
from math import *

# import a sub-module from a module
from os import path

# import method/function from a sub-module
from os.path import isfile
```



Importing modules (2/3)

```
# built-in function __import__
# allows to programmatically import a module
__import__( 'math' )
__import__( "cmath" )


# one can also "execute an import"
exec "import os"
exec 'import sys'

exec 'from math import sqrt as my_sqrt'
```



- This allows code to dynamically import anything based e.g. on some decision …
- The standard "import" reserved word calls "__import__" behind the scenes

Importing modules (3/3)

Some mistakes / bad practice to be avoided

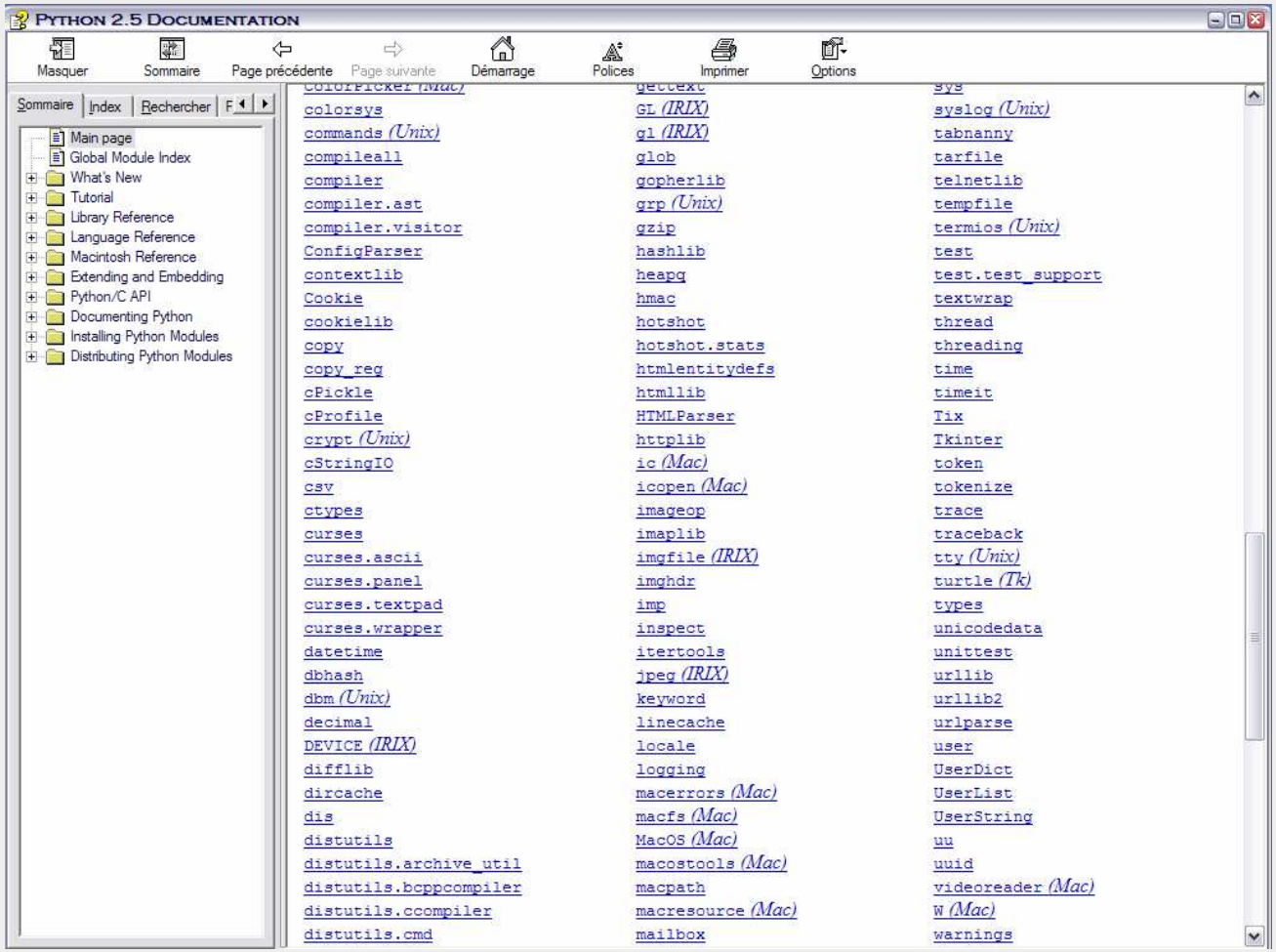
```
# let's first import the sin function from math ...
>>> from math import sin

# exception thrown (discussed later) since math module functions
# do not deal with complex numbers
>>> sin(1+2j)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't convert complex to float; use abs(z)

# let's then import the sin function from cmath ...
# no exception thrown since cmath deals with complex numbers ...
>>> from cmath import sin

>>> sin(1+2j)
(3.1657785132161682+1.9596010414216058j)
# ... this also shows that the 2nd import overwrote the 1st !
```

Some popular modules

There are very many modules !

Some popular ones are:

- sys, glob
- os, commands, shutil
- math, cmath, array
- string, StringIO, re
- getopt, optparse
- webbrowser, urllib2
- timeit



```
>>> help(os)
Help on module os:

NAME
    os - OS routines for Mac, NT, or Posix depending on what system we're on.

FILE
    c:\python25\lib\os.py

DESCRIPTION
    This exports:
      - all functions from posix, nt, os2, mac, or ce, e.g. unlink, stat, etc.
      - os.path is one of the modules posixpath, ntpath, or macpath
      - os.name is 'posix', 'nt', 'os2', 'mac', 'ce' or 'riscos'
      - os.curdir is a string representing the current directory ('.' or ':')
      - os.pardir is a string representing the parent directory ('..' or '::')
      - os.sep is the (or a most common) pathname separator ('/' or ':' or '\\')
      - os.extsep is the extension separator ('.' or '/')
      - os.altsep is the alternate pathname separator (None or '/')
      - os.pathsep is the component separator used in $PATH etc
      - os.linesep is the line separator in text files ('\r' or '\n' or '\r\n')
      - os.defpath is the default search path for executables
      - os.devnull is the file path of the null device ('/dev/null', etc.)

    Programs that import and use 'os' stand a better chance of being
    portable between different platforms.  Of course, they must then
    only use functions that are defined by all platforms (e.g., unlink
    and opendir), and leave all pathname manipulation to os.path
    (e.g., split and join).
• • •
```

A comprehensive package of modules

Files

Writing to and reading from files

```
# open(create) a (new) file and write to it
# ‘r’, ‘w’, ‘a’, for reading, writing, appending
>>> f = file( ‘my_file.txt’, ‘w’ )
>>> f
<open file 'my_file.txt', mode 'w' at 0x00B9D260>

>>> f.write( ‘Hi there!\n’ )     # newline character is relevant
>>> f.writelines( [ ‘line2\n’, ‘line3\n’] )
>>> f.close()
>>> f.closed
True

# open a file and read its contents
>>> f = file( ‘my_file.txt’ )      # in reading mode by default
>>> lines = f.readlines()
>>> f.readlines()
[]
>>> for line in lines:
...     print line
```

```
# the shortest way to “print” a file
>>> for line in file( ‘my_file.txt’ ) : print line
```

- There is more functionality to be checked: `help(file)`
- Check also the similar open built-in method for opening files …

Classes

Classes (1/3)

```
# a simple class
class MyClass:
    """My first class."""
    def __init__( self ):
        pass
    def about_me( self ):
       print 'Not much to say. I do nothing!'
```

The pre-defined "__init__" constructor is optional

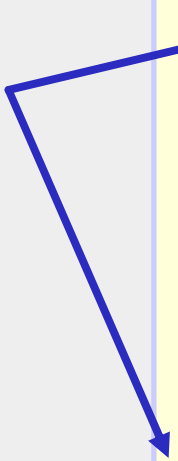
```
# a derived class
class DerivedClass( MyClass ) :
    """A class deriving from MyClass."""
    def __init__( self, username ):
        self.name = username
    def about_me( self ):
       print 'Still not much to say. Sorry.'
```

```
# the shortest class
class Minimalistic:
    pass
```

- One can also derive from the top-level base class. Try `help(object)` ...

Classes (2/3)

```
# after loading the class (Emacs “execute buffer”) in a new prompt
>>> dir()
['MyClass', '__builtins__', '__doc__', '__name__']
>>> MyClass
<class __main__.MyClass at 0x00B8FC00>

>>> print MyClass.__doc__       # doc string
My first class.

>>> dir(MyClass)
['__doc__', '__init__', '__module__', 'about_me']

# everything run interactively is made part of the __main__ module
>>> print MyClass.__module__
__main__

>>> type(MyClass)
<type ‘classobj’>
```



```
# after importing from module called MyClass (file “MyClass.py”)
>>> import MyClass
>>> MyClass
<module 'MyClass' from 'MyClass.py'>
>>> MyClass.MyClass
<class MyClass.MyClass at 0x00B8FBD0>
```

Classes (3/3)

```
# instantiation
>>> c = MyClass()
>>> c
<__main__.MyClass instance at 0x00C597B0>

>>> dir(MyClass)
['__doc__', '__init__', '__module__', 'about_me']

# call a class method
>>> c.about_me()
Not much to say. I do nothing!
```

```
# instantiation
>>> d = DerivedClass( ‘Eduardo’ )
>>> d
<__main__.DerivedClass instance at 0x00C5EF08>

>>> dir(DerivedClass)
['__doc__', '__init__', '__module__', 'about_me', ‘name']

# call a class method
>>> d.about_me()
Still not much to say. Sorry.
```

Errors and exceptions

Provoking errors, raising of exceptions

```
>>> print i
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'i' is not defined

>>> int('a')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'a‘

>>> sqrt('a')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: a float is required

>>> 1/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero

>>> 'You won't avoid syntax errors ...'
 File "<stdin>", line 1
   'You won't avoid syntax errors ...'
             ^
SyntaxError: invalid syntax’
```

Examples of common exceptions

Built-in exceptions

- Check the built-in exceptions in `__builtins__` . Remember

```
# on a fresh new Python prompt:
>>> dir()
['__builtins__', '__doc__', '__name__']
```

- Python's standard exceptions are detailed in the `exceptions` module
  - try `help(exceptions)` after importing the module …

- Exceptions are in reality classes. One can derive from them to define user-customized exceptions

Catching exceptions – try and except

```
# asks for input from the command line
x = input( 'Number to divide: ‘  )

try :
    print 1. / x
except TypeError :              # catches non-numbers
    print 'Please give a number!‘
except ZeroDivisionError :      # catches divisions by 0
    print 'You tried to divide by 0!‘

try :
    print 1. / x
except :                        # catches everything
    print ‘An exception was found’
```

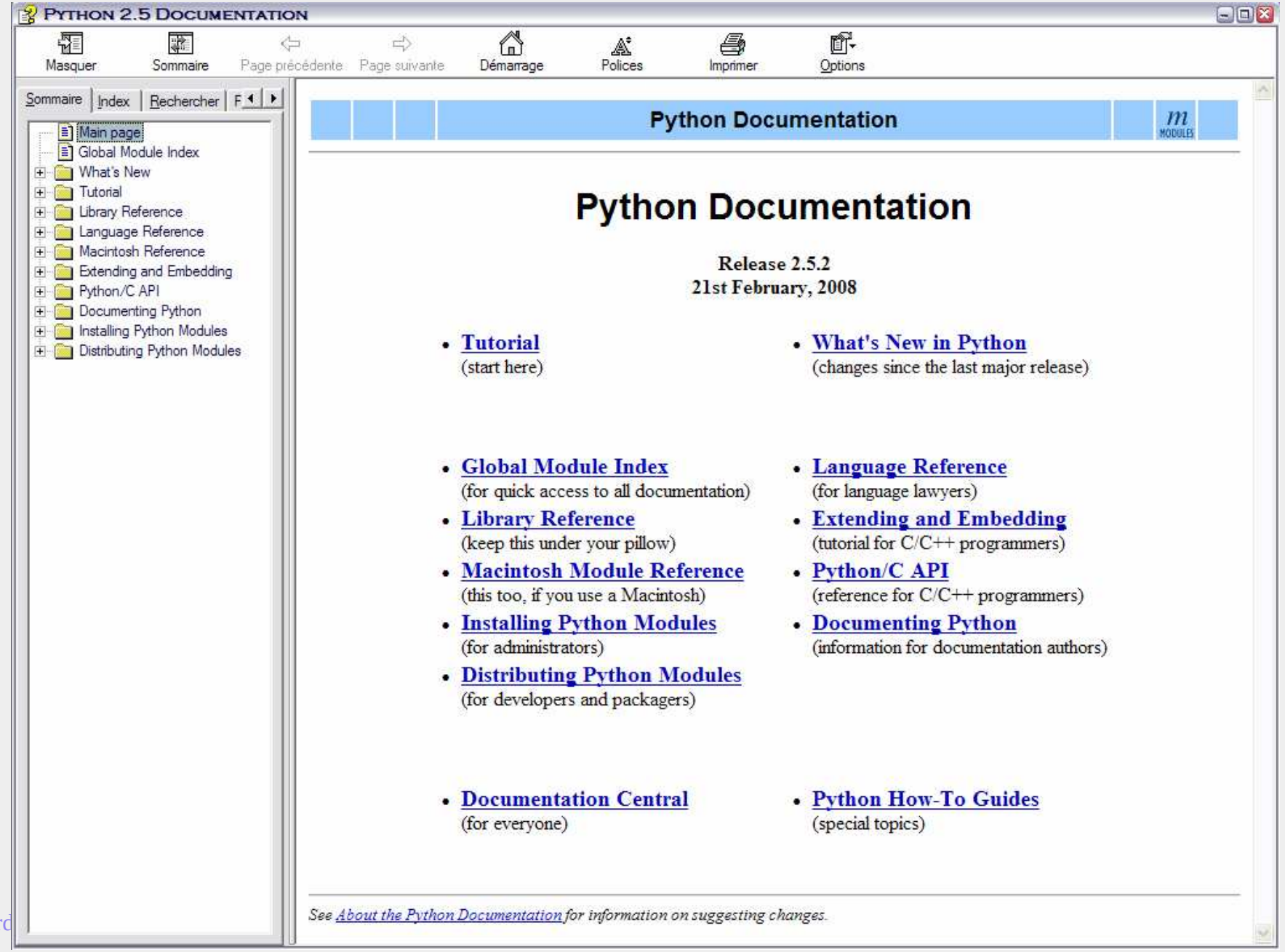
- Other reserved keywords related to exceptions are:

  `raise, finally`

- An `except` statement can also handle several exception types simultaneously. E.g. `except ( TypeError, NameError )`

Documentation

Doc, doc, doc


Python 2.5 Documentation
Masquer
Sommaire
Page précédente
Page suivante
Démarrage
Polices
Imprimer
Options
Sommaire
Index
Rechercher
Main page
Global Module Index
What's New
Tutorial
Library Reference
Language Reference
Macintosh Reference
Extending and Embedding
Python/C API
Documenting Python
Installing Python Modules
Distributing Python Modules
Python Documentation
Python Documentation
Release 2.5.2
21st February, 2008
Tutorial
(start here)
What's New in Python
(changes since the last major release)
Global Module Index
(for quick access to all documentation)
Library Reference
(keep this under your pillow)
Macintosh Module Reference
(this too, if you use a Macintosh)
Installing Python Modules
(for administrators)
Distributing Python Modules
(for developers and packagers)
Language Reference
(for language lawyers)
Extending and Embedding
(tutorial for C/C++ programmers)
Python/C API
(reference for C/C++ programmers)
Documenting Python
(information for documentation authors)
Documentation Central
(for everyone)
Python How-To Guides
(special topics)
See About the Python Documentation for information on suggesting changes.

Some suggested web sites

- www.python.org
- www.diveintopython.org
- www.scipy.org
- www.pythonware.com