Python in a Nutshell

Part I: Python, ipython, language and OOP

Manel Velasco,[1] PhD and Alexandre Perera,[1,2] PhD

[1]Departament d'Enginyeria de Sistemes, Automatica i Informatica Industrial (ESAII)
Universitat Politecnica de Catalunya

[2]Centro de Investigacion Biomedica en Red en Bioingenieria, Biomateriales y Nanomedicina (CIBER-BBN)
Alexandre.Perera@upc.edu Manel.Velasco@upc.edu

Introduction to Python for Engineering and Statistics
Febraury, 2013

Contents I

Contents II

Outline

The scientist's needs

- Get data (simulation, experiment control)
- Manipulate and process data.
- Visualize results... to understand what we are doing!
- Communicate results: produce figures for reports or publications, write presentations.

Specifications

- We don't want to re-program the plotting of a curve, a Fourier transform or a fitting algorithm. Don't reinvent the wheel! We need building blocks
- Easy to learn: computer science is neither our job nor our education
- The code should be as readable as a book
- Efficient code that executes quickly... but needless to say that a very fast code becomes useless if we spend too much time writing it. So, we need both a quick development time and a quick execution time
- A single environment/language for everything

Existing solutions I

- Compiled languages: C, C++, Fortran, etc.
  - Advantages:
    - Very fast. Very optimized compilers. For heavy computations, it's difficult to outperform these languages.
    - Some very optimized scientific libraries have been written for these languages. Example: BLAS (vector/matrix operations)
  - Drawbacks:
    - Painful usage: no interactivity during development, mandatory compilation steps, verbose syntax (*, **, ::, } , ; etc.), manual memory management (tricky in C). These are **difficult languages** for non computer scientists.

Existing solutions II

- Scripting languages: Matlab
  - Advantages:
    - Very rich collection of libraries with numerous algorithms, for many different domains. Fast execution because these libraries are often written in a compiled language.
    - Pleasant development environment: comprehensive and well organized help, integrated editor, etc.
    - Commercial support is available.
  - Drawbacks:
    - Base language is quite poor and can become restrictive for advanced users.
    - **Not free**

Existing solutions III

- Other scripting languages: Scilab, Octave, Igor, R, IDL, etc.
  - Advantages:
    - Open-source, free, or at least cheaper than Matlab.
    - Some features can be very advanced (statistics in R, figures in Igor, etc.)
  - Drawbacks:
    - Fewer available algorithms than in Matlab, and the language is not more advanced.
    - Some software are dedicated to one domain. Ex: Gnuplot or xmgrace to draw curves. These programs are very powerful, but they are restricted to a single type of usage, such as plotting.

Why not?

- What about Python?
  - Advantages:
    - Very rich scientific computing libraries (a bit less than Matlab, though)
    - Well thought out language, allowing to write very readable and well structured code: we “code what we think”.
    - Many libraries for other tasks than scientific computing (web server management, serial port access, etc.)
    - Free and open-source software, widely spread, with a vibrant community.
  - Drawbacks:
    - Less pleasant development environment than, for example, Matlab. (More geek-oriented).
    - Not all the algorithms that can be found in more specialized software or toolboxes.

It is not a must

You don’t need to use Python... but what the hell, **why not?**

History

History

- Python 1.0 - January 1994
  - Python 1.5 - December 31, 1997
  - Python 1.6 - September 5, 2000
- Python 2.0 - October 16, 2000
  - Python 2.1 - April 17, 2001
  - Python 2.2 - December 21, 2001
  - Python 2.3 - July 29, 2003
  - Python 2.4 - November 30, 2004
  - Python 2.5 - September 19, 2006
  - Python 2.6 - October 1, 2008
  - **Python 2.7 - July 3, 2010**
- Python 3.0 - December 3, 2008
  - Python 3.1 - June 27, 2009
  - Python 3.2 - February 20, 2011
  - Python 3.3 - September 29, 2012

Guido van Rossum

Conceived in the late 1980s by

Installation

Linux

apt-get install python

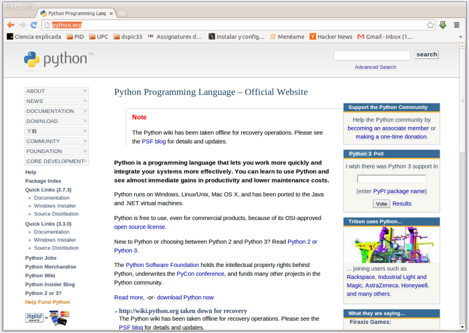
Windows

Go to
http://www.python.org/getit/
and download **Python 2.7.3 Windows Installer**

Resources

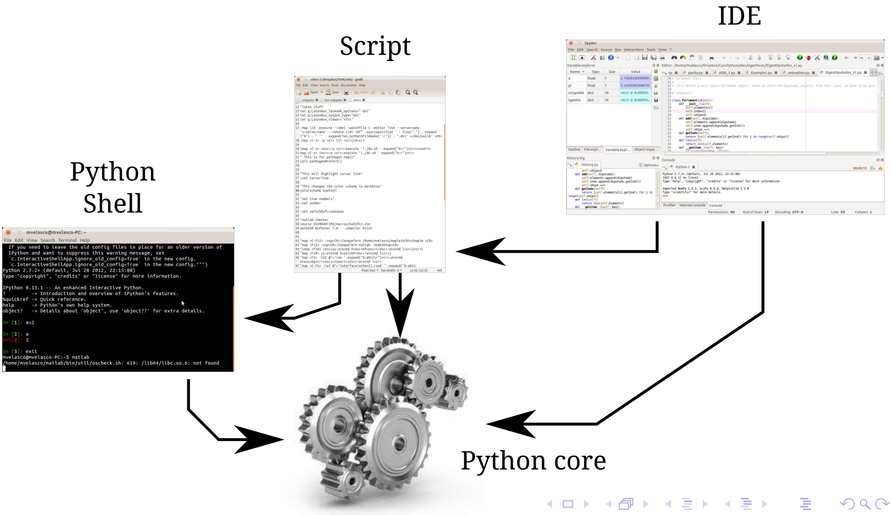
HELP!!!

http://python.org



Outline

Workflow



Python core

Python Core

Python is open, is just an specification, thus there are many Python implementations:

CPython	The default (C, C++)
CLPython	Lisp implementation of Python
Jython	The java implementation of Python
PyPy	The python implementation of Python
IronPython	C# implementation

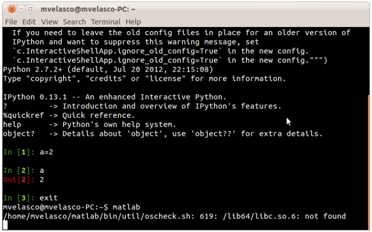
Python Core

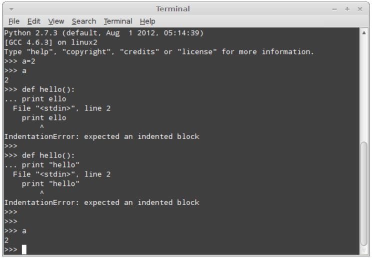
Python Shell

Python Shell

There are many tools to drive directly with Python, the most remarkable are:

- **CLIPython** The default one
- **IPython** Enhanced (VERY enhanced) default shell





Text editors

Script editors

Any text editor is well suited for creating scripts with python, we recommend some features on it:

- Tab substitution
- Code snippets
- Autocompletion

In the Linux wild, Vim and Emacs are both well suited.

IDEs

Most Valuable IDEs

- **Spyder**: The Matlab-like environment, scientist oriented. Scientist oriented
- **Eclipse-PyDEV**: Big project oriented

DEMO

Notebook

An HTML Notebook IPython

The IPython Notebook consists of two related components:

- An JSON based Notebook document format for recording and distributing Python code and rich text.
- A **web-based user interface** for authoring and running notebook documents.

DEMO

Outline

First step

STEP 1

Start the interpreter and type in

```
>>> print "Hello, world"
Hello, world
```

Welcome to Python,
you just executed your first Python instruction, congratulations!

Second step

STEP 2

To get yourself started, type the following stack of instructions

```
>>> a = 3
>>> b = 2*a
>>> type(b)
<type 'int'>
>>> print b
6
>>> a*b
18
>>> b = 'hello'
>>> type(b)
<type 'str'>
>>> b + b
'hellohello'
>>> 2*b
'hellohello'
```

Second step

STEP 2

To get yourself started, type the following stack of instructions

```
>>> a = 3
>>> b = 2*a
>>> type(b)
<type 'int'>
>>> print b
6
>>> a*b
18
>>> b = 'hello'
>>> type(b)
<type 'str'>
>>> b + b
'hellohello'
>>> 2*b
'hellohello'
```

Observe that

- We do not declare variables (hurrah!!!!!)
- Variable type may be changed on the fly (hurrah!!!, hurrah!!!)
- There is a way to overload operators (hurrah!, hurrah!, hurrah!!!)
- There is a function that tell us the type of a variable.

Types

Integer

```
>>> 1+1
2
>>> a=4
```

Float

```
>>> c=2.1
>>> 3.5/c
1.6666666666666665
```

Boolean

```
>>> 3 > 4
False
>>> test = (3 > 4)
>>> test
False
>>> type(test)
<type 'bool'>
```

Complex

```
>>> a=1.5+0.5j
>>> a.real
1.5
>>> a.imag
0.5
>>> import cmath
>>> cmath.phase(a)
0.3217505543966422
```

Basic Calculator

A Python shell can therefore replace your pocket calculator, with the basic arithmetic operations +, -, *, /, % (modulo) natively implemented:

```
>>> 7 * 3.
21.0
>>> 2**10
1024
>>> 8 % 3
2
```

WARNING!

Integer Division

```
>>> 3/2
1
```

Use floats

```
>>> 3 / 2.
1.5
>>> a = 3
>>> b = 2
>>> a / b
1
>>> a / float(b)
1.5
```

Lists

Python provides many efficient types of containers, in which collections of objects can be stored.

Lists

A list is an ordered collection of objects, that may have different types. For example

```
>>> l = [1, 2, 3, 4, 5]
>>> type(l)
<type 'list'>
```

Lists

accessing individual objects contained in the list:

```
>>> l[2]
3
```

Counting from the end with negative indices:

```
>>> l[-1]
5
>>> l[-2]
4
```

Warning Indexing starts at 0

```
>>> l[0]
1
```

Lists

Slicing

```
>>> l
[1, 2, 3, 4, 5]
>>> l[2:4]
[3, 4]
```

Warning

Warning Note that **l[start:stop]** contains the elements with indices i such as start $\leq$ i $<$ stop (i ranging from start to stop-1). Therefore, **l[start:stop] has (stop-start) elements**.

Lists

Slicing syntax: l[start:stop:step]

All slicing parameters are optional:

```
>>> l
[1, 2, 3, 4, 5]
>>> l[3:]
[4, 5]
>>> l[:3]
[1, 2, 3]
>>> l[::2]
[1, 3, 5]
```

Lists

The elements of a list may have different types:

```
>>> l = [3, 2+3j, 'hello']
>>> l
[3, (2+3j), 'hello']
>>> l[1], l[2]
((2+3j), 'hello')
```

Lists

Python offers a large panel of functions to modify lists, or query them. Here are a few examples; for more details, see http://docs.python.org/tutorial/datastructures.html#more-on-lists

Add and remove elements

```
>>> l = [1, 2, 3, 4, 5]
>>> l.append(6)
>>> l
[1, 2, 3, 4, 5, 6]
>>> l.pop()
6
>>> l
[1, 2, 3, 4, 5]
>>> l.extend([6, 7]) # extend l, in-place
>>> l
[1, 2, 3, 4, 5, 6, 7]
>>> l = l[:-2]
>>> l
[1, 2, 3, 4, 5]
```

Lists

Reverse list

```
>>> r = l[::-1]
>>> r
[5, 4, 3, 2, 1]
```

Concatenate and repeat

```
>>> r + l
[5, 4, 3, 2, 1, 1, 2, 3, 4, 5]
>>> 2 * r
[5, 4, 3, 2, 1, 5, 4, 3, 2, 1]
```

Sort (in-place)

```
>>> r.sort()
>>> r
[1, 2, 3, 4, 5]
```

Note

Methods and Object-Oriented Programming

The notation r.method() (r.sort(), r.append(3), l.pop()) is our first example of object-oriented programming (OOP). Being a list, the object r owns the method function that is called using the notation '.'

No further knowledge of OOP than understanding the notation '.' is necessary for going through this tutorial.

Note

Discovering methods in ipython tab-completion (press tab)

```
In [1]: r.
r.append    r.extend    r.insert    r.remove    r.sort
r.count     r.index     r.pop       r.reverse
```

Strings

```
s = 'Hello, how are you?'
s = "Hi, what's up"
s = '''Hello,
      how are you'''# tripling the quotes allows the
s = """Hi,
      what's up?""" # the string to span more than one line
```

Strings

Indexing strings

```
>>> a = "hello"
>>> a[0]
'h'
>>> a[1]
'e'
>>> a[-1]
'o'
```

Strings

Substitution

```
>>> 'An integer: %i; a float: %f; another string: %s' % (1, 0.1, 'string')
'An integer: 1; a float: 0.100000; another string: string'
>>> i = 102.1
>>> filename = 'processing_of_dataset_%03d.txt'%i
>>> filename
'processing_of_dataset_102.txt'
```

Challenge

5 seconds challenge

In ipython, create a list and check its methods with the tab-completion feature

Strings

Slicing

```
>>> a = "hello, world!"
>>> a[3:6] # 3rd to 6th (excluded) elements: elements 3, 4, 5
'lo,'
>>> a[2:10:2] # Syntax: a[start:stop:step]
'lo o'
>>> a[::3] # every three characters, from beginning to end
'hl r!'
```

BUT...

Strings

You can't change them in this way

```
In [1]: a = "hello, world!"
In [2]: a[2] = 'z'
---------------------------------------------------------------------------
TypeError                 Traceback (most recent call last)

/home/mvelasco/Curs_Python/<ipythonconsole> in <module>()

TypeError: 'str' object does not support item assignment
```

PAY ATTENTION

NEXT SET OF SLIDES ARE VERY IMPORTANT!!!

Mutable and immutable types

Immutable types

- integer
- float
- complex
- boolean
- strings

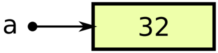
Mutable

- Lists

Immutable types

Create an immutable element

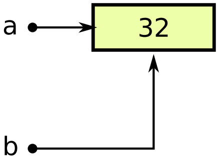
```
>>> a=32
```



Immutable types

"copy" it

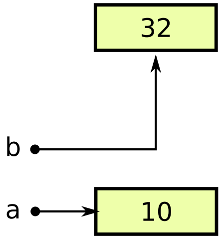
```
>>> a=32
>>> b=a
```



Immutable types

Change the original object

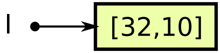
```
>>> a=32
>>> b=a
>>> a=10
>>> b
32
```



Mutable types

Create a mutable type

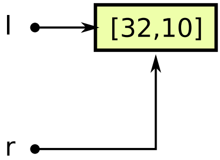
```
>>> l=[32,10]
```



Mutable types

"Copy" it

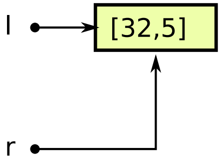
```
>>> l=[32,10]
>>> r=l
```



Mutable types

Change the original object

```
>>> l=[32,10]
>>> r=l
>>> l[1]=3
>>> r
[32, 3]
```



Challenge

1 minute challenge

Create a list A, create a list B that contains A, copy the list B into C, modify A and check C value

Visited Types

Already seen types

- boolean
- integer
- float
- complex
- string
- list

Pending Types

- Dictionary
- Tuple
- Set

Dictionary

A dictionary is basically an efficient table that maps keys to values. It is an unordered container:

```
>>> tel = {'emmanuelle': 5752, 'sebastian': 5578}
>>> tel['francis'] = 5915
>>> tel
{'sebastian': 5578, 'francis': 5915, 'emmanuelle': 5752}
>>> tel['sebastian']
5578
>>> tel.keys()
['sebastian', 'francis', 'emmanuelle']
>>> tel.values()
[5578, 5915, 5752]
>>> 'francis' in tel
True
```

Dictionary

A dictionary can have keys (resp. values) with different types:

```
>>> d = {'a':1, 'b':2, 3:'hello'}
>>> d
{'a': 1, 3: 'hello', 'b': 2}
```

Challenge

1 minute challenge

Are Dicts mutable?

Tuples

The elements of a tuple are written between parentheses, or just separated by commas:

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> u = (0, 2)
```

Sets

unordered, unique items:

```
>>> s = set(('a', 'b', 'c', 'a'))
>>> s
set(['a', 'c', 'b'])
>>> s.difference(('a', 'b'))
set(['c'])
```

Challenge

2 minutes challenge

- Are tuples mutable?
- Which are the methods of tuples?
- Are Sets mutable?
- Which are de methods of sets?

Before going on...

Built-in functions

abs()	divmod()	input()	open()	staticmethod()
all()	enumerate()	int()	ord()	str()
any()	eval()	isinstance()	pow()	sum()
basestring()	execfile()	issubclass()	print()	super()
bin()	file()	iter()	property()	tuple()
bool()	filter()	len()	range()	type()
bytearray()	float()	list()	raw_input()	unichr()
callable()	format()	locals()	reduce()	unicode()
chr()	frozenset()	long()	reload()	vars()
classmethod()	getattr()	map()	repr()	xrange()
cmp()	globals()	max()	reversed()	zip()
compile()	hasattr()	memoryview()	round()	__import__()
complex()	hash()	min()	set()	apply()
delattr()	help()	next()	setattr()	buffer()
dict()	hex()	object()	slice()	coerce()
dir()	id()	oct()	sorted()	intern()

if/then/else

If

```
>>> if 2**2 == 4:
...     print 'Obvious!'
...
Obvious!
```

Blocks are delimited by indentation

```
a = 10
if a == 1:
    print(1)
elif a == 2:
    print(2)
else:
    print('A lot')
```

```
A lot
```

Conditional Expressions¶

if object:

Evaluates to False:

- any number equal to zero (0, 0.0, 0+0j)
- an empty container (list, tuple, set, dictionary, ...)
- False, None

Evaluates to True:

- everything else (User-defined classes can customize those rules by overriding the special **nonzero** method.)

Tests equality, with logics:

```
>>> 1==1.
True
```

Tests identity: both sides are the same object:

```
>>> 1 is 1.
False
>>> a = 1
>>> b = 1
>>> a is b
True
```

For any collection b: b contains a

```
>>> b = [1, 2, 3]
>>> 2 in b
True
>>> 5 in b
False
```

If b is a dictionary, this tests that a is a key of b.

for/range

Iterating with an index:

```
>>> for i in range(4):
...     print(i)
...
0
1
2
3
```

But most often, it is more readable to iterate over values:

```
>>> for word in ('cool', 'powerful', 'readable'):
...    print('Python is %s' % word)
...
Python is cool
Python is powerful
Python is readable
```

while/break/continue¶

Typical C-style while loop (Mandelbrot problem):

```
>>> z = 1 + 1j
>>> while abs(z) < 100:
...     z = z**2 + 1
...
```

Break out of enclosing for/while loop:

```
>>> z = 1 + 1j
>>> while abs(z) < 100:
...     if z.imag == 0:
...         break
...     z = z**2 + 1
```

Continue the next iteration of a loop.:

```
a = [1, 0, 2, 4]
for element in a:
    if element == 0:
        continue
    print 1. / element
```

```
1.0
0.5
0.25
```

Advanced iteration

Iterate over any sequence

You can iterate over any sequence (string, list, keys in a dictionary, lines in a file, ...):

```
>>> vowels = 'aeiou'
>>> for i in 'powerful':
...     if i in vowels:
...         print(i),
...
```

```
>>> message = "Hello how are you?"
>>> message.split() # returns a list
['Hello', 'how', 'are', 'you?']
>>> for word in message.split():
...     print word,
...
```

Few languages (in particular, languages for scientific computing) allow to loop over anything but integers/indices. With Python it is possible to loop exactly over the objects of interest without bothering with indices you often don't care about.

Keeping track of enumeration number

Common task is to iterate over a sequence while keeping track of the item number.

Could use while loop with a counter as above. Or a for loop:

```
>>> words = ('cool', 'powerful', 'readable')
>>> for i in range(0, len(words)):
...     print(i, words[i]),
...
```

But Python provides enumerate for this:

```
>>> words = ('cool', 'powerful', 'readable')
>>> for index, item in enumerate(words):
...     print index, item,
...
```

Looping over a dictionary

Use iteritems:

```
>>> d = {'a': 1, 'b':1.2, 'c':1j}
>>> for key, val in d.iteritems():
...     print('Key: %s has value: %s' % (key, val))
...
Key: a has value: 1
Key: c has value: 1j
Key: b has value: 1.2
```

List comprehensions

Natural math

$$k = \left\{x^2, x \in \{0, 1, 2, 3\}\right\}$$

```
>>> k=[x**2 for x in range(4)]
>>> k
[0, 1, 4, 9]
```

Challenge

5 minutes challenge

Compute the decimals of π using the Wallis formula:

$$\pi = 2 \prod_{i=1}^{\infty} \frac{4i^2}{4i^2 - 1}$$

Exceptions

Exceptions are raised by errors in Python:

```
In [1]: 1/0
---------------------------------------------------------------------------
ZeroDivisionError: integer division or modulo by zero
In [2]: 1 + 'e'
---------------------------------------------------------------------------
TypeError: unsupported operand type(s) for +: 'int' and 'str'
In [3]: d = {1:1, 2:2}
In [4]: d[3]
---------------------------------------------------------------------------
KeyError: 3
In [5]: l = [1, 2, 3]
In [6]: l[4]
---------------------------------------------------------------------------
IndexError: list index out of range
In [7]: l.foobar
---------------------------------------------------------------------------
AttributeError: 'list' object has no attribute 'foobar'
```

Catching exceptions

try/except

```
In [8]: while True:
 ....:     try:
 ....:         x = int(raw_input('Please enter a number: '))
 ....:         break
 ....:     except ValueError:
 ....:         print('That was no valid number.  Try again...')
 ....:
 ....:
Please enter a number: a
That was no valid number.  Try again...
Please enter a number: 1

In [9]: x
Out[9]: 1
```

Catching exceptions

try/finally

Important for resource management (e.g. closing a file)

```
In [10]: try:
   ....:     x = int(raw_input('Please enter a number: '))
   ....: finally:
   ....:     print('Thank you for your input')
   ....:
   ....:
Please enter a number: a
Thank you for your input
---------------------------------------------------------------------------
ValueError: invalid literal for int() with base 10: 'a'
```

There are many tricks with the exceptions, but they are out of the scope of these slides

Outline

Function definition

Function blocks must be indented as other control-flow blocks.

```
In [56]: def test():
   ....:     print('in test function')
   ....:
   ....:

In [57]: test()
in test function
```

Return statement

Functions can optionally return values.

```
In [6]: def disk_area(radius):
   ...:     return 3.14 * radius * radius
   ...:

In [8]: disk_area(1.5)
Out[8]: 7.0649999999999995
```

Structure:

- the def keyword;
- is followed by the function's name, then
- the arguments of the function are given between brackets followed by a colon.
- he function body ;
- and return object for optionally returning values.
- By default, functions return None.

Parameters

Mandatory parameters (positional arguments)

```
In [81]: def double_it(x):
   ....:     return x * 2
   ....:

In [82]: double_it(3)
Out[82]: 6

In [83]: double_it()
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)

/Users/cburns/src/scipy2009/scipy_2009_tutorial/source/<ipython console> in <module>()

TypeError: double_it() takes exactly 1 argument (0 given)
```

Parameters

Optional parameters (keyword or named arguments)

```
In [84]: def double_it(x=2):
   ....:     return x * 2
   ....:
In [85]: double_it()
Out[85]: 4
In [86]: double_it(3)
Out[86]: 6
```

Warning

```
In [124]: bigx = 10
In [125]: def double_it(x=bigx):
   .....:     return x * 2
   .....:
In [126]: bigx = 1e9  # Now really big
In [128]: double_it()
Out[128]: 20
```

Parameters

More involved example implementing python's slicing:

```
In [98]: def slicer(seq, start=None, stop=None, step=None):
   ....:     """Implement basic python slicing."""
   ....:     return seq[start:stop:step]
   ....:

In [101]: rhyme = 'one fish, two fish, red fish, blue fish'.split()

In [102]: rhyme
Out[102]: ['one', 'fish,', 'two', 'fish,', 'red', 'fish,', 'blue', 'fish']

In [103]: slicer(rhyme)
Out[103]: ['one', 'fish,', 'two', 'fish,', 'red', 'fish,', 'blue', 'fish']

In [104]: slicer(rhyme, step=2)
Out[104]: ['one', 'two', 'red', 'blue']

In [105]: slicer(rhyme, 1, step=2)
Out[105]: ['fish,', 'fish,', 'fish,', 'fish']

In [106]: slicer(rhyme, start=1, stop=4, step=2)
Out[106]: ['fish,', 'fish,']
```

Parameters and mutability

5 minutes challenge

Check the behaviour of mutable and no mutable parameters and determine if parameters are passed by reference or by value

Parameters and mutability

5 minutes challenge, solution

```
>>> def try_to_modify(x, y, z):
...     x = 23
...     y.append(42)
...     z = [99] # new reference
...     print(x)
...     print(y)
...     print(z)
...
>>> a = 77    # immutable variable
>>> b = [99]  # mutable variable
>>> c = [28]
>>> try_to_modify(a, b, c)
23
[99, 42]
[99]
>>> print(a)
77
>>> print(b)
[99, 42]
>>> print(c)
[28]
```

Global variables

Variables declared outside the function can be referenced within the function:

```
In [114]: x = 5

In [115]: def addx(y):
   .....:     return x + y
   .....:

In [116]: addx(10)
Out[116]: 15
```

But..

This doesn't work:

```
x=5
In [117]: def setx(y):
   .....:     x = y
   .....:     print('x is %d' % x)
   .....:
   .....:
In [118]: setx(10)
x is 10
In [120]: x
Out[120]: 5
```

This works:

```
x=5
In [121]: def setx(y):
   .....:     global x
   .....:     x = y
   .....:     print('x is %d' % x)
   .....:
   .....:
In [122]: setx(10)
x is 10
In [123]: x
Out[123]: 10
```

Variable number of parameters

Special forms of parameters:

- ***args** any number of positional arguments packed into a tuple
- ****kwargs** any number of keyword arguments packed into a dictionary

```
In [35]: def variable_args(*args, **kwargs):
   ....:     print 'args is', args
   ....:     print 'kwargs is', kwargs
   ....:

In [36]: variable_args('one', 'two', x=1, y=2, z=3)
args is ('one', 'two')
kwargs is {'y': 2, 'x': 1, 'z': 3}
```

Docstrings

Documentation about what the function does and it's parameters. General convention:

```
In [67]: def funcname(params):
   ....:     """Concise one-line sentence describing the function.
   ....:
   ....:     Extended summary which can contain multiple paragraphs.
   ....:     """
   ....:     # function body
   ....:     pass
   ....:

In [68]: funcname ?
Type:           function
Base Class: <type 'function'>
String Form:        <function funcname at 0xeaa0f0>
Namespace:  Interactive
File:           /home/mvelasco/Curs_Python/.../<ipython console>
Definition: funcname(params)
Docstring:
    Concise one-line sentence describing the function.

    Extended summary which can contain multiple paragraphs.
```

Functions are objects

Functions are first-class objects, which means they can be:

- assigned to a variable
- an item in a list (or any collection)
- passed as an argument to another function

Example

```
In [38]: va = variable_args

In [39]: va('three', x=1, y=2)
args is ('three',)
kwargs is {'y': 2, 'x': 1}
```

Challenge

10 min challenge: Fibonacci

Write a function that displays the n first terms of the Fibonacci sequence, defined by:

$u_0 = 1; u_1 = 1$

$u_{(n+2)} = u_{(n+1)} + u_n$

15 minutes challenge: QuickSort

Implement the quicksort algorithm, as defined by wikipedia

Decorators as function wrapper

Function can be decorated by using the decorator syntax for functions:

```
@mydecorator           # (2)
def function():        # (1)
    pass
```

```
def mydecorator(f)
    return f()
def function():                  # (1)
    pass
function = mydecorator(function)   # (2)
```

Decorators as function wrappers

Example

```
def helloSolarSystem(original_function):
    def new_function():
        original_function()  # the () after "original_function" causes original_function to be called
        print("Hello, solar system!")
    return new_function

def helloGalaxy(original_function):
    def new_function():
        original_function()  # the () after "original_function" cause original_function to be called
        print("Hello, galaxy!")
    return new_function

@helloGalaxy
@helloSolarSystem
def hello():
    print ("Hello, world!")

# Here is where we actually *do* something!
hello()
```

Checkout the result of this structure

Debug with decorators

Just for fun

```
def debug(f):
    def my_wrapper(*args,**kwargs):
        call_string = "%s called with *args: %r, **kwargs: %r " % (f.__name__, args, kwargs)
        ret_val=f(*args,**kwargs)
        call_string+=repr(ret_val)
        if debugging:
            print call_string
        return ret_val
    return my_wrapper

@debug
def recursive(k):
    if k>1:
        return k*recursive(k-1)
    else:
        return 1

debugging=False
recursive(3)
debugging=True
recursive(3)
```

Scripts

First script

A sequence of instructions that are executed each time the script is called.

Instructions may be e.g. copied-and-pasted from the interpreter (but take care to respect indentation rules!).

```
message = "Hello how are you?"
for word in message.split():
    print word
```

Scripts

in Ipython, the syntax to execute a script is %run script.py. For example,

```
In [1]: %run test.py
Hello
how
are
you ?

In [2]: message
Out[2]: 'Hello how are you?'
```

From de command line

```
mvelasco->mvelasco-PC:~/Curs_Python\ $  python test.py
Hello
how
are
you ?
```

Scripts

Standalone scripts may also take command-line arguments

in file.py:

```
import sys
print sys.argv
```

when executed

```
\ $  python file.py test arguments
['file.py', 'test', 'arguments']
```

Modules

Importing objects from modules

```
In [1]: import os

In [2]: os
Out[2]: <module 'os' from '/usr/lib/python2.6/os.pyc'>

In [3]: os.listdir('.')
Out[3]:
['conf.py',
 'basic_types.rst',
 'control_flow.rst',
 'functions.rst',
 'python_language.rst',
 'reusing.rst',
 'file_io.rst',
 'exceptions.rst',
 'workflow.rst',
 'index.rst']
```

Try to check how many functions are there in os with tab-completion and ipython

Modules

Alternatives to full import

Import only some functions

```
In [4]: from os import listdir
```

Or a shorthand

```
In [5]: import numpy as np
```

Modules

Actually, all the scientific computing tools we are going to use are modules:

```
>>> import numpy as np # data arrays
>>> np.linspace(0, 10, 6)
array([  0.,   2.,   4.,   6.,   8.,  10.])
>>> import scipy # scientific computing
```

My own module

```
"A demo module."

def print_b():
    "Prints b."
    print 'b'
def print_a():
    "Prints a."
    print 'a'
c = 2
d = 2
```

```
In [1]: import demo
In [2]: demo.print_a()
a
In [3]: demo.print_b()
b
```

Try this in ipython

```
In [4]: demo ?
In [5]: who
In [6]: whos
In [7]: dir(demo)
In [8]: demo.     #tab-completion
```

Modules

Warning:Module caching

‘main’ and module loading

A script and a Module

```
def print_a():
    "Prints a."
    print 'a'


if __name__ == '__main__':
    print_a()
```

```
In [12]: import demo2
In [13]: %run demo2
a
```

Input and Output

To write in a file:

```
>>> f = open('workfile', 'w') # opens the workfile file
>>> type(f)
<type 'file'>
>>> f.write('This is a test \nand another test')
>>> f.close()
```

Input and Output

To read from a file

```
In [1]: f = open('workfile', 'r')

In [2]: s = f.read()

In [3]: print(s)
This is a test
and another test

In [4]: f.close()
```

Input and Output

Iterating over a file

```
In [6]: f = open('workfile', 'r')

In [7]: for line in f:
...:     print line
...:
...:
This is a test
and another test
In [8]: f.close()
```

Challenge

10 Minutes challenge

Write a script that reads a file with a column of numbers and calculates the min, max and sum

Challenge

10 minutes challenge

Write a module that performs basic trigonometric functions using Taylor expansions

OS module: Operating system functionality

Directory and file manipulation

Current directory:

```
In [17]: os.getcwd()
Out[17]: '/Users/cburns/src/scipy2009/scipy_2009_tutorial/source'
```

List a directory:

```
In [31]: os.listdir(os.curdir)
Out[31]:
['.index.rst.swo',
 '.python_language.rst.swp',
 '.view_array.py.swp',
 '_static',
 '_templates',
 'basic_types.rst',
 'conf.py',
 'control_flow.rst',
 'debugging.rst',
 ...
```

OS module: Operating system functionality

Make a directory

```
In [32]: os.mkdir('junkdir')
In [33]: 'junkdir' in os.listdir(os.curdir)
Out[33]: True
```

Rename the directory:

```
In [36]: os.rename('junkdir', 'foodir')
In [37]: 'junkdir' in os.listdir(os.curdir)
Out[37]: False
In [38]: 'foodir' in os.listdir(os.curdir)
Out[38]: True
In [41]: os.rmdir('foodir')
In [42]: 'foodir' in os.listdir(os.curdir)
Out[42]: False
```

Delete a file:

```
In [44]: fp = open('junk.txt', 'w')
In [45]: fp.close()
In [46]: 'junk.txt' in os.listdir(os.curdir)
Out[46]: True
In [47]: os.remove('junk.txt')
In [48]: 'junk.txt' in os.listdir(os.curdir)
Out[48]: False
```

os.path: path manipulations

os.path provides common operations on pathnames.

```
In [70]: fp = open('junk.txt', 'w')
In [71]: fp.close()
In [72]: a = os.path.abspath('junk.txt')
In [73]: a
Out[73]: '/Users/cburns/src/scipy2009/scipy_2009_tutorial/source/junk.txt'
In [74]: os.path.split(a)
Out[74]: ('/Users/cburns/src/scipy2009/scipy_2009_tutorial/source','junk.txt')
In [78]: os.path.dirname(a)
Out[78]: '/Users/cburns/src/scipy2009/scipy_2009_tutorial/source'
In [79]: os.path.basename(a)
Out[79]: 'junk.txt'
In [80]: os.path.splitext(os.path.basename(a))
Out[80]: ('junk', '.txt')
In [84]: os.path.exists('junk.txt')
Out[84]: True
In [86]: os.path.isfile('junk.txt')
Out[86]: True
In [87]: os.path.isdir('junk.txt')
Out[87]: False
In [88]: os.path.expanduser('~/local')
Out[88]: '/Users/cburns/local'
In [92]: os.path.join(os.path.expanduser('~'), 'local', 'bin')
Out[92]: '/Users/cburns/local/bin'
```

Other OS services

Running an external command

```
In [3]: os.system('ls *.tex')
commondefs.tex CursP_1.tex  CursP_3.tex
CursP_4.tex     format.tex   header.tex
```

Walking a directory

```
In [4]: for dirpath, dirnames, filenames in
os.walk(os.curdir):
   ...:     for fp in filenames:
   ...:         print os.path.abspath(fp)
   ...:
/home/mvelasco/Dropbox/Curs_Python/CursP_3.log
/home/mvelasco/Dropbox/Curs_Python/CursP_4.out
/home/mvelasco/Dropbox/Curs_Python/syllabus.odt
/home/mvelasco/Dropbox/Curs_Python/format.tex
/home/mvelasco/Dropbox/Curs_Python/CursP_3.pdf
/home/mvelasco/Dropbox/Curs_Python/tags
/home/mvelasco/Dropbox/Curs_Python/CursP_3.vrb
```

glob: Pattern matching on files

```
In [5]: import glob
In [6]: glob.glob('*.tex')
Out[6]:
['format.tex',
 'CursP_4.tex',
 'header.tex',
 'CursP_1.tex',
 'CursP_3.tex',
 'commondefs.tex']
```

sys module: system-specific information

```
In [8]: import sys
In [9]: sys.platform
Out[9]: 'linux2'
In [10]: sys.version
Out[10]: '2.7.3 (default, Aug  1 2012, 05:14:39) \n[G
In [11]: sys.prefix
Out[11]: '/usr'
```

Object-oriented programming

OOP

We are not going to use OOP in this course, but we provide some snippets of code just to know the structure of class declaration

Object-oriented programming

Class Declaration

```
>>> class Student(object):
...     def __init__(self, name):
...         self.name = name
...     def set_age(self, age):
...         self.age = age
...     def set_major(self, major):
...         self.major = major
...
>>> anna = Student('anna')
>>> anna.set_age(21)
>>> anna.set_major('physics')
```

Class extension

```
>>> class MasterStudent(Student):
...     internship = 'mandatory, from March to June'
...
>>> james = MasterStudent('james')
>>> james.internship
'mandatory, from March to June'
>>> james.set_age(23)
>>> james.age
23
```