Computational Mathematics with Python

Basics

Olivier Verdier and Claus Führer

Spring 2009

1 Introduction and Motivation
- Python vs Other Languages
- Examples and Demo

2 Concepts
- Basic Types
- Variables
- Lists
- For Loop

3 Practical Information

Why Python?

Python is. . .

- Free and open source
- It is a *scripting language*, meaning that it is interpreted
- It is modern: object oriented, exception handling, dynamic typing etc.
- Plenty of libraries, in particular scientific ones: linear algebra; visualisation tools: plotting, image analysis; differential equations solving; symbolic computations; statistics ; etc.
- Many possible usages: Scientific computing (of course :-)), scripting, web sites, text parsing, etc.
- Used by YouTube, Google, NASA, Los Alamos, NSA among others

Python vs language XX

Java, C++
: Object oriented compiled languages. Very limited and extremely verbose. Low level compared to python. Few scientific libraries.

C, FORTRAN
: Very low level compiled language. Useful in some CPU critical situations.

php, ruby
: Other interpreted languages. PHP is web oriented. Ruby is as flexible as python but has no scientific library.

MATLAB
: Tool for matrix computation that evolved for scientific computing. The scientific library is huge but it is not a programming language. Extremely expensive.

Examples

Python may be used in *interactive* mode:

```
>>> x = 3
>>> y = 5
>>> print x + y
8
```

Here we solve

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \cdot x = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

```
>>> M = array([[1., 2.],
               [3., 4.]])
>>> V = array([2., 1.])
>>> x = solve(M, V)
>>> print x
[-3.   2.5]
```

More examples

Computing $e^{i\pi}$ and 2^{100}:

```
>>> print exp(1j*pi) # should return -1 :-)
(-1+1.22464679915e-16j)
>>> print 2**100
1267650600228229401496703205376L
```

Computing $\zeta(x) = \sum_{k=1}^{\infty} \frac{1}{k^x}$. For $x = 2$ we know that $\zeta(2) = \frac{\pi^2}{6}$:

```
# for x = 2:
>>> print scipy.special.zeta(2., 1)
1.64493406685
>>> print pi**2/6
1.6449340668482264
```

Demo

Demo

1 Introduction and Motivation
- Python vs Other Languages
- Examples and Demo

2 Concepts
- Basic Types
- Variables
- Lists
- For Loop

3 Practical Information

Numbers

A number may be an *integer*, a *real number* or a *complex number*. The usual operations are

- + and - addition and substraction
- * and / multiplication and division
- ** power

```
2**(2+2) # 16
1j**2 # -1
```

Strings

Strings are “lists” of characters, enclosed by simple or double quotes:

```
'valid string'
"string with double quotes"
```

Strings

Strings are “lists” of characters, enclosed by simple or double quotes:

```
'valid string'
"string with double quotes"
```

You may also use *triple quotes* for strings including multiple lines:

```
"""This is
a long,
long string"""
```

Concept: Variable

Variables

A variable is a *reference* to an object. An object may have several references. One uses the *assignment operator* = to assign a value to a variable.

Example

```
x = [3, 4] # a list object is created
y = x # this object now has two labels: x and y
del x # we delete one of the labels
del y # both labels are removed: the object is deleted
```

Concept: Lists

Lists

A python list is an *ordered* list of objects, enclosed in square brackets. One accesses elements of a list using *zero-based* indices inside square brackets.

List Examples

Example

```
L1 = [1, 2]
L1[0] # 1
L1[1] # 2
L1[2] # raises IndexError

L2 = ['a', 1, [3, 4]]
L2[0] # 'a'
L2[2][0] # 3

L2[-1] # last element: [3,4]
L2[-2] # second to last: 1
```

List Utilities

- `range(n)` creates a list with n elements, starting with zero:

```
print range(5)
[0, 1, 2, 3, 4]
```

List Utilities

- range(n) creates a list with n elements, starting with zero:

```
print range(5)
[0, 1, 2, 3, 4]
```

- len(L) gives the *length* of a list:

```
len(['a', 1, 2, 34]) # returns 4
```

List Utilities

- `range(n)` creates a list with n elements, starting with zero:

```
print range(5)
[0, 1, 2, 3, 4]
```

- `len(L)` gives the *length* of a list:

```
len(['a', 1, 2, 34]) # returns 4
```

- Use `append` to append an element to a list:

```
L = ['a', 'b', 'c']
L[-1] # 'c'
L.append('d')
L # L is now ['a', 'b', 'c', 'd']
L[-1] # 'd'
```

Comprehensive lists

A convenient way to build up lists is to use the *comprehensive lists* construct, possibly with a conditional inside.

Definition

The syntax of a comprehensive list is

```
[<expr> for <x> in <list>]
```

Example

```
L = [2, 3, 10, 1, 5]

L2 = [x*2 for x in L] # [4, 6, 20, 2, 10]

L3 = [x*2 for x in L if 4 < x <= 10] # [20, 10]
```

Comprehensive Lists in Maths

Mathematical Notation

This is very close to the mathematical notation for sets. Compare:

$$L_2 = \{2x;\ x \in L\}$$

and

```
L2 = [2*x for x in L]
```

One big difference though is that lists are *ordered* while sets aren't.

Operations on Lists

- Adding two lists *concatenates* (*sammanfoga*) them:

```
L1 = [1, 2]
L2 = [3, 4]
L = L1 + L2 # [1, 2, 3, 4]
```

Operations on Lists

- Adding two lists *concatenates* (*sammanfoga*) them:

```
L1 = [1, 2]
L2 = [3, 4]
L = L1 + L2 # [1, 2, 3, 4]
```

- Logically, multiplying a list with an integer concatenates the list with itself several times: `n*L` is equivalent to $\underbrace{L + L + \cdots + L}_{n \text{ times}}$.

```
L = [1, 2]
3 * L # [1, 2, 1, 2, 1, 2]
```

Concept: for loop

for loop

A *for loop* allows to loop through a list using an *index variable*. This variable is successively equal to all the elements in the list.

Concept: for loop

for loop

A *for loop* allows to loop through a list using an *index variable*. This variable is successively equal to all the elements in the list.

Example

```
L = [1, 2, 10]
for s in L:
  print s * 2,
# output: 2 4 20
```

Indentation

The part to be repeated in the for loop has to be properly *indented*:

```
for elt in my_list:
  do_something()
  something_else()
  etc
print "loop finished" # outside the for block
```

Repeating a Task

One *typical use* of the `for` loop is to repeat a certain task a fixed number of time:

```
n = 30
for i in range(n):
  do_something # this gets executed n times
```

1 Introduction and Motivation
- Python vs Other Languages
- Examples and Demo

2 Concepts
- Basic Types
- Variables
- Lists
- For Loop

3 Practical Information

Python Shell

- Start a python session by typing `scipython` in a unix shell
- Check that it is working with: `plot(rand(4));show()`
- A window should appear with a graph; you should be able to type other commands without having to close the graph window
- when you want to quit, write `exit()`

When you want to run python at home please follow the installation instruction on `http://www.maths.lth.se/na/python/install`

Executing Scripts

You often want to execute the contents of a file.

- We recommand to use `Kate` on the Linux machines (but any other good editor will do)
- Save your files in (for example) in `$HOME/course/`
- Type (once) in `ipython`: `cd course`
- To execute the contents of a file named `file.py` just write `execfile('file.py')` in `ipython`.

Getting Help

Some tips on how to use `ipython`:

- To get *help* on an object just type `?` after it and then return
- *Use the arrow keys to reuse the last executed commands*
- We will see later that you may use the tabulation key for *completion* in general

Computational Mathematics with Python

Booleans

Olivier Verdier and Claus Führer

Spring 2009

Conditional Expressions

Definition

A *conditional expression* is an expression that may have the value *True* or *False*.

Conditional Expressions

Definition

A *conditional expression* is an expression that may have the value *True* or *False*.

Some common operators that yield conditional expressions are:

- `==`, `!=`
- `<`, `>`, `<=`, `>=`
- One combines different boolean values with `or` and `and`
- `not` gives the *logical negation* of the expression that follows

Boolean Expression Examples

Example

```
2 >= 4  # False
2 < 3 < 4 # True
2 < 3 and 3 < 2 # False
2 != 3 < 4 or False # True
2 <= 2 and 2 >= 2 # True
not 2 == 3 # True
not False or True and False # True!
```

Note in the last example the rules when using `not`, `and`, `or`.

Concept: If statement

Concept: conditional statement

A conditional statement delimits a *block* that will be executed if the condition is true. An *optional* block, started with the keyword `else` will be executed if the condition is not fulfilled.

Concept: If statement

Concept: conditional statement

A conditional statement delimits a *block* that will be executed if the condition is true. An *optional* block, started with the keyword `else` will be executed if the condition is not fulfilled.

Example

We print the absolute value of x. Mathematically this is defined as x if $x \geq 0$ and $-x$ if $x < 0$:

```
x = ...
if x >= 0:
  print x
else:
  print -x
```

$$|x| = \begin{cases} x \text{ if} & x \geq 0 \\ -x \text{ else} & \end{cases}$$

The full for statement: `break`

break gets out of the for loop even if the list we are iterating is not exhausted.

```
for x in x_values:
  if x > threshold:
    break
  print x
```

The full for statement: `else`

else checks whether the for loop was *broken* with the `break` keyword.

```
for x in x_values:
  if x > threshold:
    break
else:
  print "all the x are below the threshold"
```

4 If Statement

5 For Loop

6 String Formatting

7 Functions

Basic string formatting

- for strings:

```
course_code = "NUMA21"
print "This course's name is %s" % course_code
# This course's name is NUMA21
```

- for integers:

```
nb_students = 16
print "There are %d students" % nb_students
# There are 16 students
```

- for reals:

```
average_grade = 3.4
print "Average grade: %f" % average_grade
# Average grade: 3.400000
```

Basics on Functions

Functions are useful to gather similar pieces of code at one place. Consider the following mathematical *function*:

$$x \mapsto f(x) := 2x + 1$$

The python equivalent is:

```
def f(x):
  return 2*x + 1
```

- the keyword `def` tells python we are defining a function
- `f` is the *name* of the function
- `x` is the *argument*, or *input* of the function
- what is after return is called the *output* of the function

Anatomy of a Function

The keyword **def** indicates the start of a function.

Function arguments are listed separated by commas. They are passed by *assignment*. More on this later.

```
def add(arg0, arg1):
    a = arg0 + arg1
    return a
```

Indentation is used to indicate the contents of the function. It is *not* optional, but a part of the syntax.

A colon (**:**) terminates the function definition.

An optional return statement specifies the value returned from the function. If return is omitted, the function returns the special value **None**.

(Source: E. Jones and T. Oliphant)

Calling a Function

Once the following function is defined:

```
def f(x):
  return 2*x + 1
```

it may now be called using:

```
f(2) # 5
f(1) # 3
# etc.
```

Concluding example

test2.py

```python
def f(x):
    y=2*x**2-x-0.5
    return 2*y

a=f(0.5)
text='a is  %s'

if a > 0:
    sign='positive'
else:
    sign='negative or zero'

print text % sign
```

Screenshot from Python editor drpython

Computational Mathematics with Python

Slicing

Olivier Verdier and Claus Führer

Spring 2009

Creating sublists

Slicing

Slicing a list between i and j is creating a copy of its element starting from element at index i and ending *just before* j.

Creating sublists

Slicing

Slicing a list between i and j is creating a copy of its element starting from element at index i and ending *just before* j.

One simple way to understand slicing

`L[i:j]` means: create a list by removing the first i elements and containing the next $j - i$ elements (for $j > i \geq 0$).

Creating sublists

Slicing

Slicing a list between i and j is creating a copy of its element starting from element at index i and ending *just before* j.

One simple way to understand slicing

`L[i:j]` means: create a list by removing the first i elements and containing the next $j-i$ elements (for $j > i \geq 0$).

Example

```
L = ['C', 'l', 'a', 'u', 's']
L[1:4] # remove one element and take three from there:
# ['l', 'a', 'u']
```

Partial slicing

One may omit the first or last bound of the slicing:

```
L = ['C', 'l', 'a', 'u', 's']
L[1:] # ['l', 'a', 'u', 's']
L[:3] # ['C', 'l', 'a']
L[-2:] # ['u', 's']
L[:-2] # ['C', 'l', 'a']
L[:] # the whole list
```

Partial slicing

One may omit the first or last bound of the slicing:

```
L = ['C', 'l', 'a', 'u', 's']
L[1:] # ['l', 'a', 'u', 's']
L[:3] # ['C', 'l', 'a']
L[-2:] # ['u', 's']
L[:-2] # ['C', 'l', 'a']
L[:] # the whole list
```

Mathematical Analogy

This is similar to half lines in $\mathbb{R}$. $[-\infty, a)$ means: take all numbers strictly lower than a; this is similar to the syntax `L[:j]`.

`L[2:]`

0	1	2	3	$\cdots$	-3	-2	-1

Rule of thumb

Taking all elements from index 2 *included* amounts to *remove* the first two elements

`L[2:]`

0	1	2	3	$\cdots$	-3	-2	-1

Rule of thumb

Taking all elements from index 2 *included* amounts to *remove* the first two elements

`L[:2]`

0	1	2	3	$\cdots$	-3	-2	-1

Rule of thumb

Taking all elements until index 2 *excluded* amounts to *keep* only the first two elements

0	1	2	3	$\cdots$	-3	-2	-1

`L[:-2]`

Rule of thumb

Taking all elements until index -2 *excluded* amounts to *remove* the last two elements

`L[:-2]`

0	1	2	3	$\cdots$	-3	-2	-1

Rule of thumb

Taking all elements until index -2 *excluded* amounts to *remove* the last two elements

`L[-2:]`

0	1	2	3	$\cdots$	-3	-2	-1

Rule of thumb

Taking all elements from index -2 *included* amounts to *keep* only the last two elements

`L[2:-1]`

0	1	2	3	$\cdots$	-3	-2	-1

Rule of thumb

`L[i:-j]` amounts to *remove* the first i elements and *remove* the last j elements.

`L[2:-1]`

0	1	2	3	$\cdots$	-3	-2	-1

Rule of thumb

`L[i:-j]` amounts to *remove* the first i elements and *remove* the last j elements.

`L[2:5]`

0	1	2	3	4	5	$\cdots$	-1

Rule of thumb

`L[i:j]` amounts to *remove* the first i and *keep* the $j - i$ *next* ones.

`L[-4:-1]`

0	1	⋯	-5	-4	-3	-2	-1

Rule of thumb

`L[-i:-j]` amounts to *remove* the last j and *keep* the $i-j$ *preceding* ones.

Tuples

Definition

A *tuple* is an *immutable* list. Immutable means that it cannot be modified.

Tuples

Definition

A *tuple* is an *immutable* list. Immutable means that it cannot be modified.

Example

```
my_tuple = 1, 2, 3 # our first tuple!
len(my_tuple) # 3, same as for lists

my_tuple[0] = 'a' # error! tuples are immutable

singleton = 1, # note the comma
len(singleton) # 1
```

Packing and unpacking

One may assign several variables at once by *unpacking* a list or tuple:

```
a, b = 0, 1 # a gets 0 and b gets 1
a, b = [0, 1] # exactly the same effect
(a, b) = 0, 1 # same
[a,b] = [0,1] # same thing
```

Packing and unpacking

One may assign several variables at once by *unpacking* a list or tuple:

```
a, b = 0, 1 # a gets 0 and b gets 1
a, b = [0, 1] # exactly the same effect
(a, b) = 0, 1 # same
[a,b] = [0,1] # same thing
```

The swapping trick!

Use packing and unpacking to *swap* the contents of two variables.

```
a, b = b, a
```

Formatting Strings

Format with several arguments. Use of tuple as a second argument is *mandatory*.

```
print "My name is %s %s" % ('Olivier', 'Verdier')
```

Returning Multiple Values

A function may return several values:

```
def argmin(L): # return the minimum and index
    ...
    return minimum, minimum_index

m, i = argmin([1, 2, 0]) # m is 0, i is 2
# or:
min_info = argmin([1, 2, 0])
min_info[0] # 0
min_info[1] # 2
```

A final word on tuples

- Tuples are *nothing else* than immutable lists with a notation without brackets

A final word on tuples

- Tuples are *nothing else* than immutable lists with a notation without brackets
- In most cases lists may be used instead of tuples

A final word on tuples

- Tuples are *nothing else* than immutable lists with a notation without brackets
- In most cases lists may be used instead of tuples
- The bracket free notation is nice but *dangerous*, you should *use parenthesis when you are not sure*:

```
a, b = b, a # the swap tricks; equivalent to:
(a, b) = (b, a)
# but
1, 2 == 3, 4 # returns (1, False, 4)
(1, 2) == (3, 4) # returns False
```

Convenient Notations

- Multiple assignment

```
a = b = c = 1 # a, b and c get the same value 1
```

Convenient Notations

- Multiple assignment

```
a = b = c = 1 # a, b and c get the same value 1
```

- Increment operators

```
a += 1 # same as a = a + 1
b *= 1 # same as b = b * 1
```

Convenient Notations

- Multiple assignment

```
a = b = c = 1 # a, b and c get the same value 1
```

- Increment operators

```
a += 1 # same as a = a + 1
b *= 1 # same as b = b * 1
```

- Multiple comparisons

```
a < b < c # same as: a < b and b < c
a == b == c # same as: a == b and b == c
```

Function Documentation

Document your function using a string at the beginning:

```
def newton(f, x0):
  """Compute a zero of `f` with the Newton method;
     x0 is the initial guess"""
  ...
```

Try this out and check the IPython help for that function!

Functions are objects

Functions are *objects* like everything else. One may pass functions around as arguments, change their names and delete them.

```
def square(x):
  """Return the square of ‘x‘"""
  return x**2
square(4) # 16
sq = square # now sq is the same as square
sq(4) # 16
del square # ‘square‘ doesn’t exist anymore
print newton(sq, .2) # passing as argument
```

Named Arguments

Definition

The argument of a function may be accessed in any order, provided they are *named*.

Example

Assume a function `newton` is defined as:

```
def newton(f, x0, tol):
    ...
```

Example

Assume a function `newton` is defined as:

```
def newton(f, x0, tol):
   ...
```

Now we can call it in various ways:

```
# note the order of the arguments in this call:
newton(tol=1e-3, x0=.1, f=cos)

# here the first argument is not named
newton(cos, tol=1e-3, x0=.1) # ok
newton(cos, .1, tol=1e-3) # ok

# but you can't have anonymous args after named ones
newton(f=cos, .1, tol=1e-3) # error!
```

Default Value

Definition

An argument may be given a *default value* with the equal sign.

Example

```
def newton(f, x0, tol=1e-6):
  ...
# the tolerance is set to the default 1e-6
newton(cos, .2)
newton(x0=.2, f=cos) # the same as above
```

Plot

- You have already used the command `plot`. It needs a list of x values and a list of y values. If a single list is given, the list of x values `range(len(y))` is assumed.
- You may use the keyword argument `label` to give your curves a name, and then show them using `legend`.

```
x = .2
x1 = [sin(.3*n*x) for n in range(20)]
x2 = [sin(2*n*x) for n in range(20)]
plot(x1, label='0.3')
plot(x2, label='2')
legend()
```

Computational Mathematics with Python

Exceptions, Generators

Olivier Verdier and Claus Führer

Spring 2009

Sound the alarm

Creating errors

Creating an error is called “raise an exception”. You may raise an exception like this:

```
raise Exception("Something went wrong")
```

Typical exceptions are

- TypeError
- ValueError

You already know SyntaxError and ZeroDivisionError.

Review the alarms

Reviewing the errors

You may review the errors using `try` and `except`:

```
try:
  <some code that might raise an exception>
except ValueError:
  print "Oops, a ValueError occurred"
```

Review the alarms

Reviewing the errors

You may review the errors using `try` and `except`:

```
try:
  <some code that might raise an exception>
except ValueError:
  print "Oops, a ValueError occurred"
```

Flow Control

An exception stops the flow and looks for the closest enclosing `try` block. If it is not caught it continues searching for the next `try` block.

Error messages

Golden rule

Never print error messages, *raise an exception instead*

Error messages

Golden rule

Never print error messages, *raise an exception instead*

```
def factorial(n):
  if n < 0:
    raise ValueError("A positive integer is expected")
  ...
```

Definition

Definition

A generator generates objects (to be passed to a for loop). Similar to a list except that the objects need not exist before entering the loop.

Definition

Definition

A generator generates objects (to be passed to a for loop). Similar to a list except that the objects need not exist before entering the loop.

Example

A typical generator is `xrange`: works like `range` but produces a generator instead of a sequence.

```
for i in xrange(100000000):
  if i > 10:
    break
```

Definition

Definition

A generator generates objects (to be passed to a for loop). Similar to a list except that the objects need not exist before entering the loop.

Example

A typical generator is `xrange`: works like `range` but produces a generator instead of a sequence.

```
for i in xrange(100000000):
  if i > 10:
    break
```

Infinite

Note that, just as in mathematics, generators may be infinite.

Create Generators

Creation of generators is possible with the keyword `yield`:

```
def odd_numbers(n):
  "generator for odd numbers less than n"
  for k in xrange(n):
    if k % 2:
      yield k
```

Create Generators

Creation of generators is possible with the keyword `yield`:

```
def odd_numbers(n):
  "generator for odd numbers less than n"
  for k in xrange(n):
    if k % 2:
      yield k
```

Then you call it as:

```
g = odd_numbers(10)
for k in g:
  ... # do something with k
```

Generator Tools

- `enumerate` is used to *enumerate* another generator:

```
A = ['a', 'b', 'c']
for i, x in enumerate(A):
  print i, x,
# result: 0 a 1 b 2 c
```

Generator Tools

- `enumerate` is used to *enumerate* another generator:

```
A = ['a', 'b', 'c']
for i, x in enumerate(A):
  print i, x,
# result: 0 a 1 b 2 c
```

- `reversed` creates a generator from a list by going backwards:

```
A = [0, 1, 2]
for elt in reversed(A):
  print elt,
# result: 2 1 0
```

List Filling Pattern

Common programming pattern:

```
L = []
for k in xrange(n):
  L.append(some_function(k))
```

List Filling Pattern

Common programming pattern:

```
L = []
for k in xrange(n):
  L.append(some_function(k))
```

use instead:

```
L = [function(k) for k in xrange(n)]
```

Complicated List Filling

```
L = [0,1]
for k in range(n):
  # call various functions here
  # that compute "result"
  L.append(result)
```

Complicated List Filling

```
L = [0,1]
for k in range(n):
  # call various functions here
  # that compute "result"
  L.append(result)
```

Use a generator instead:

```
def result_generator(n):
  for k in xrange(n):
    # call various functions here
    # that compute "result"
    yield result
```

Complicated List Filling

```
L = [0,1]
for k in range(n):
  # call various functions here
  # that compute "result"
  L.append(result)
```

Use a generator instead:

```
def result_generator(n):
  for k in xrange(n):
    # call various functions here
    # that compute "result"
    yield result
```

...and if you really need a list:

```
L = list(result_generator(n)) # no append needed!
```

List from a generator

To convert a generator to a list:

```
# for example:
g = xrange(10)

L = list(g)
# now L is a list with 10 elements
```

“Comprehensive” generator

Just as we had comprehensive lists, there is also *comprehensive generators*:

```
g = (n for n in xrange(1000) if not n % 100)
# generates 100, 200,
```

Zipping generators

How to make one generator out of two?

```
xg = x_generator()
yg = y_generator()

for x,y in zip(xg,yg):
  print x,y
```

Zipping generators

How to make one generator out of two?

```
xg = x_generator()
yg = y_generator()

for x,y in zip(xg,yg):
  print x,y
```

The zipped generator stops *as soon as one of the generators is exhausted*.

Out-of-bound slices

No errors for out-of-bound slices

Take notice that you *never* get index errors with out of bound slices, only, possibly, empty lists.

Example

```
L = range(4) # [0, 1, 2, 3]
L[4] # error!
L[1:100] # same as L[1:]
L[-100:-1] # same as L[:-1]
L[-100:100] # same as L[:]
L[5:0] # empty
L[-2:2] # empty WHY?
```

Altering lists

Slicing may be used to change lists.

```
L = [0, 1, 2, 3, 4]
L[0] = 'a' # standard access via index
```

- Replacement or Deletion

```
L[2:3] = [100, 200] # ['a', 1, 100, 200, 3, 4]
L[2:3] = [] # ['a', 1, 200, 3, 4]
L[3:] = [] # ['a', 1, 200]
```

- Insertion

```
L[1:1] = [1000, 2000] # ['a', 1000, 2000, 1, 100]
```

Strides

Definition

When computing slices one may also specify a *stride* which is the length of the step from one index to the other. The default stride is one.

Example

```
L = range(100)
L[:10:2] # [0, 2, 4, 6, 8]
L[::20] # [0, 20, 40, 60, 80]
L[10:20:3] # [10, 13, 16, 19]
```

Note that the stride may also be negative.

Belonging to a list

Definition

One may use the keywords **in** and **not in** to determine whether an element belongs to a list (similar to $\in$ and $\notin$ in mathematics).

Example

```
L = ['a', 1, 'b', 2]
'a' in L # True
3 in L # False
4 not in L # True
```

Mathematical View

In mathematics one has the following notation, for two sets A and C:

$$C^A := \{\text{functions from } A \text{ to } C\}$$

One remarkable fact is that

$$C^{A\times B} \equiv (C^A)^B \equiv (C^B)^A$$

Mathematical View

In mathematics one has the following notation, for two sets A and C:

$$C^A := \{\text{functions from } A \text{ to } C\}$$

One remarkable fact is that

$$C^{A\times B} \equiv (C^A)^B \equiv (C^B)^A$$

It is indeed obvious that

$$f : (a, b) \mapsto f(a, b) \equiv a \mapsto f_a$$

where

$$f_a : b \mapsto f(a, b)$$

Practical example

We want to use the function $t \mapsto \sin(2\pi\omega t)$ for various frequencies ω. Mathematically we have a function

$$(\omega, t) \mapsto \sin(2\pi\omega t)$$

and for each ω we want the function

$$\sin_\omega : t \mapsto \sin(2\pi\omega t)$$

Practical example

We want to use the function $t \mapsto \sin(2\pi\omega t)$ for various frequencies ω. Mathematically we have a function

$$(\omega, t) \mapsto \sin(2\pi\omega t)$$

and for each ω we want the function

$$\sin_\omega : t \mapsto \sin(2\pi\omega t)$$

Python allows the following construction:

```
def make_sine(freq):
  "Make a sine function with frequency freq"
  def mysine(t):
    return sin(2*pi*freq*t)
  return mysine
```

Computational Mathematics with Python

Linear Algebra

Olivier Verdier and Claus Führer

Spring 2009

The `array` type

Lists are almost like vectors but the operations on list are *not* the linear algebra operation.

Definition

An *array* represents a vector in linear algebra. It is often initialised from a list or another vector. Operations `+`, `*`, `/`, `-` are all *elementwise*. `dot` is used for the scalar product.

Vector usage

Example

```
vec = array([1., 3.]) # a vector in the plane
2*vec # array([2., 6.])
vec * vec # array([1., 9.])
vec/2 # array([0.5, 1.5])
norm(vec) # norm
dot(vec, vec) # scalar product
```

Vectors are similar to lists

- Access vectors via their indices

```
v = array([1., 2., 3.])
v[0] # 1.
```

- The *length* of a vector is still obtained by the function `len`.

```
len(v) # 3
```

Vectors are similar to lists

- Access vectors via their indices

```
v = array([1., 2., 3.])
v[0] # 1.
```

- The *length* of a vector is still obtained by the function `len`.

```
len(v) # 3
```

- Parts of vectors using slices

```
v[:2] # array([1., 2.])
```

- Replace parts of vectors using slices

```
v[:2] = [10, 20]
v # array([10., 20., 3.])
```

Vectors are not lists! I

Operations are not the same:

- Operations + and * are *different*
- More operations are defined: -, /
- Many functions act elementwise on vectors: `exp`, `sin`, `sqrt`, etc.
- Scalar product with `dot`

Vectors are not lists! II

- Vectors have a *fixed size*: no `append` method
- Only *one type* throughout the whole vector (usually `float` or `complex` but beware of the `int` type!!)
- Vector slices are *views*:

```
v = array([1., 2., 3.])
v1 = v[:2] # v is array([1.,2.])
v1[0] = 0. # if I change v1...
v # array([0., 2., 3.]) v is changed too!
```

More examples I

```
v1 = array([1., 2., 3.]) # don't forget the dots!
v2 = array([2, 0, 1.]) # one dot is enough
v1 + v2; v1 / v2; v1 - v2; v1 * v2

3*v1
3*v1 + 2*v2
```

More examples II

```
dot(v1, v2) # scalar product
cos(v1) # cosine, elementwise

# access
v1[0] # 1.
v1[0] = 10

# slices
v1[:2] # array([10., 2.])
v1[:2] = [0, 1] # now v1 == array([0.,1.,3.])
v1[:2] = [1,2,3] # error!
```

Vectors and plotting

The `linspace` method is a convenient way to create equally spaced arrays.

```
xs = linspace(0, 10, 200) # 200 points between 0 and 10
xs[0] # the first point is zero
xs[-1] # the last is ten
```

So for example the plot of the sine function between zero and ten will be obtain by:

```
plot(xs, sin(xs))
```

Vectorised Functions

Note that *not all functions* may be applied on vectors. For instance this one:

```
def const(x):
  return 1
```

We will see later how to automatically *vectorise* a function so that it works componentwise on vectors.

Creating vectors

Some handy methods to quickly create vectors:

- zeros `zeros(n)` creates a vector of size n filled with zeros
- ones `ones(n)` is the same filled with ones
- rand `rand(n)` creates a vector randomly filled with uniform distribution between zero and one
- empty `empty(n)` creates an "empty" vector of size n (try it!)

Concatenating Vectors

Since the + operation is redefined we need a means to *concatenate* vectors. This is where the command `hstack` comes to help.
`hstack([v1,v2,...,vn])` concatenates the vectors `v1`, `v2`, ..., `vn`.

Concatenating Vectors

Since the + operation is redefined we need a means to *concatenate* vectors. This is where the command `hstack` comes to help. `hstack([v1,v2,...,vn])` concatenates the vectors `v1`, `v2`, ..., `vn`.

Symplectic permutation

We have a vector of size $2n$. We want to permute the first half with the second half of the vector with sign change:

$$(x_1, x_2, \ldots, x_n, x_{n+1}, \ldots, x_{2n}) \mapsto (-x_n, -x_{n+1}, \ldots, -x_{2n}, x_1, \ldots, x_n)$$

```
def symp(v):
  n = len(v) // 2 # use the integer division //
  return hstack([-v[-n:], v[:n]])
```

Matrices as Lists of Lists

Definition

Matrices are represented by arrays of lists of *rows*, which are lists as well.

```
# the identity matrix in 2D
id = array([[1., 0.], [0., 1.]])
# python allows this:
id = array([[1., 0.],
            [0., 1.]])
# which is more readable
```

Accessing Matrix Entries

Matrix coefficients are accessed with *two* indices:

```
M = array([[1., 2.],[3.,4.]])
M[0,0] # first line, first column: 1.
M[-1,0] # last line, first column: 3.
```

Creating Matrices

Some convenient methods to create matrices are:

- **eye** `eye(n)` is the identity matrix of size n
- **zeros** `zeros([n,m])` fills an $n \times m$ matrix with zeros
- **rand** `rand(n,m)` is the same with random values
- **empty** `empty([n,m])` same with "empty" values

Shape

The *shape* of a matrix is the tuple of its dimensions. The shape of an $n \times m$ matrix is (n,m). It is given by the method shape:

```
M = eye(3)
M.shape # (3, 3)

V = array([1., 2., 1., 4.])
V.shape # (4,) <- tuple with one element
```

Transpose

You may *switch* the two shape elements by *transposing* the matrix. The transpose of a matrix A_{ij} is a matrix B such that

$$B_{ij} = A_{ji}$$

```
A = ...
A.shape # 3,4

B = A.T # A transpose
B.shape # 4,3
```

Matrix vector multiplication

The mathematical concept of *reduction*:

$$\sum_j a_{ij} x_j$$

is translated in python in the function dot:

```
angle = pi/3
M = array([[cos(angle), -sin(angle)],
           [sin(angle), cos(angle)]])
V = array([1., 0.])
Y = dot(M, V) # the product M.V
```

Matrix vector multiplication

The mathematical concept of *reduction*:

$$\sum_j a_{ij} x_j$$

is translated in python in the function dot:

```
angle = pi/3
M = array([[cos(angle), -sin(angle)],
           [sin(angle), cos(angle)]])
V = array([1., 0.])
Y = dot(M, V) # the product M.V
```

Elementwise vs. matrix multiplication

The multiplication operator `*` is *always* elementwise. It has nothing to do with the dot operation. `A*V` is a legal operation which will be explained later on.

Solving a Linear System

If A is a matrix and b is a vector you solve the linear equation

$$A \cdot x = b$$

using `solve` which has the syntax `x = solve(A,b)`.

Solving a Linear System

If A is a matrix and b is a vector you solve the linear equation

$$A \cdot x = b$$

using `solve` which has the syntax `x = solve(A,b)`.

Example

We want to solve

$$\begin{cases} x_1 + 2x_2 & = 1 \\ 3x_1 + 4x_2 & = 4 \end{cases}$$

```
A = array([[1., 2.],
           [3., 4.]])
b = array([1., 4.])
x = solve(A,b)
dot(A, x) # should be almost b
```

Slices

Slices are similar to that of lists and vectors *except* that there are now *two dimensions*.

`M[i,:]`	a *vector* filled by the row i of M
`M[:,j]`	a *vector* filled by the column j of M
`M[2:4,:]`	slice `2:4` on the lines only
`M[2:4,1:4]`	slice on lines and columns

Slices

Slices are similar to that of lists and vectors *except* that there are now *two dimensions*.

`M[i,:]`	a *vector* filled by the row i of M
`M[:,j]`	a *vector* filled by the column j of M
`M[2:4,:]`	slice `2:4` on the lines only
`M[2:4,1:4]`	slice on lines and columns

Omitting a dimension

If you omit an index or a slice, SciPy assumes you are taking *rows only*.

`M[3]`	is the third row of M
`M[1:3]`	is a matrix with the second and third rows of M.

Altering a Matrix

You may alter a matrix using slices or direct access.

- `M[2,3] = 2.`
- `M[2,:] = <a vector>`
- `M[1:3,:]= <a matrix>`
- `M[1:4,2:5]= <a matrix>`

The matrices and vectors above *must have the right size* to “fit” in the matrix M.

Computational Mathematics with Python

Linear Algebra 2

Olivier Verdier and Claus Führer

Spring 2009

Computational Mathematics with Python

Linear Algebra 2

Olivier Verdier and Claus Führer

Spring 2009

Dot multiplication

vector vector

$$s = \sum_i x_i y_i$$

Dot multiplication

vector vector

$$s = \sum_i x_i y_i$$

matrix vector

$$y_i = \sum_j A_{ij} x_j$$

Dot multiplication

vector vector

$$s = \sum_i x_i y_i$$

matrix vector

$$y_i = \sum_j A_{ij} x_j$$

matrix matrix

$$C_{ij} = \sum_k A_{ik} B_{kj}$$

Dot multiplication

vector vector

$$s = \sum_i x_i y_i$$

matrix vector

$$y_i = \sum_j A_{ij} x_j$$

matrix matrix

$$C_{ij} = \sum_k A_{ik} B_{kj}$$

vector matrix

$$y_j = \sum_i x_i A_{ij}$$

Rank of matrix slices

When slicing the rank of the result object is as follows:

access	rank	kind
index,index	0	scalar
slice,index	1	vector
slice,slice	2	matrix

Rank of matrix slices

When slicing the rank of the result object is as follows:

access	rank	kind
index,index	0	scalar
slice,index	1	vector
slice,slice	2	matrix

Example

0	1	2	3
4	5	6	7
8	9	10	11

access	shape	rank	kind
`M[:2,1:-1]`	$(2,2)$	2	matrix

Rank of matrix slices

When slicing the rank of the result object is as follows:

access	rank	kind
index,index	0	scalar
slice,index	1	vector
slice,slice	2	matrix

Example

0	1	2	3
4	5	6	7
8	9	10	11

access	shape	rank	kind
`M[:2,1:-1]`	$(2,2)$	2	matrix
`M[1,:]`	4	1	vector

Rank of matrix slices

When slicing the rank of the result object is as follows:

access	rank	kind
index,index	0	scalar
slice,index	1	vector
slice,slice	2	matrix

Example

0	1	2	3
4	5	6	7
8	9	10	11

access	shape	rank	kind
`M[:2,1:-1]`	$(2,2)$	2	matrix
`M[1,:]`	4	1	vector
`M[1,1]`	$\emptyset$	0	scalar

Rank of matrix slices

When slicing the rank of the result object is as follows:

access	rank	kind
index,index	0	scalar
slice,index	1	vector
slice,slice	2	matrix

Example

0	1	2	3
4	5	6	7
8	9	10	11

access	shape	rank	kind
`M[:2,1:-1]`	$(2,2)$	2	matrix
`M[1,:]`	4	1	vector
`M[1,1]`	$\emptyset$	0	scalar
`M[1:2,:]`	$(1,4)$	2	matrix

Rank of matrix slices

When slicing the rank of the result object is as follows:

access	rank	kind
index,index	0	scalar
slice,index	1	vector
slice,slice	2	matrix

Example

0	1	2	3
4	5	6	7
8	9	10	11

access	shape	rank	kind
`M[:2,1:-1]`	$(2,2)$	2	matrix
`M[1,:]`	4	1	vector
`M[1,1]`	$\emptyset$	0	scalar
`M[1:2,:]`	$(1,4)$	2	matrix
`M[1:2,1:2]`	$(1,1)$	2	matrix

Reshaping

From a given tensor (vector or matrix) one may obtain another tensor by *reshaping*.

Reshaping Example

```
A = arange(6)
```

0 1 2 3 4 5

Reshaping Example

```
A.reshape(1,6)
```

0	1	2	3	4	5

Reshaping Example

```
A.reshape(6,1)
```

0
1
2
3
4
5

Reshaping Example

```
A.reshape(2,3)
```

0	1	2
3	4	5

Reshaping Example

```
A.reshape(3,2)
```

0	1
2	3
4	5

Reshaping Trick

Note that python can *guess one of the new dimensions*. Just give a negative integer for the dimension to be guessed:

```
A = arange(12) # a vector of length 12

A.reshape(1,-1) # row matrix
A.reshape(-1,1) # column matrix

A.reshape(3,-1) # 3,4 matrix
A.reshape(-1,4) # same
```

Building Matrices

- Piling vectors
- Stacking vectors
- Stacking column matrices

Building Matrices

- Piling vectors
- Stacking vectors
- Stacking column matrices

The universal method to build matrices is `concatenate`. This function is called by several convenient functions

- `hstack` to stack matrices *horizontally*
- `vstack` to stack matrices *vertically*
- `column_stack` to stack *vectors* in columns

Stacking Vectors

```
v1 = array([1,2])
v2 = array([3,4])
```

Stacking Vectors

```
v1 = array([1,2])
v2 = array([3,4])
```

```
vstack([v1,v2])
```

1	2
3	4

Stacking Vectors

```
v1 = array([1,2])
v2 = array([3,4])
```

```
column_stack([v1,v2])
```

1	3
2	4

sum, max, min

You may perform a number of operations on arrays, either on the whole array, or column-wise or row-wise. The most common are

- *max*
- *min*
- *sum*

Example

1	2	3	4
5	6	7	8

```
A.sum()
```

36

sum, max, min

You may perform a number of operations on arrays, either on the whole array, or column-wise or row-wise. The most common are

- *max*
- *min*
- *sum*

Example

1	2	3	4
5	6	7	8

```
A.sum(axis=0)
```

The result is a *vector*

6	8	10	12

sum, max, min

You may perform a number of operations on arrays, either on the whole array, or column-wise or row-wise. The most common are

- *max*
- *min*
- *sum*

Example

1	2	3	4
5	6	7	8

```
A.sum(axis=1)
```

The result is a *vector*

10	26

19 Matrices and Vectors

- Dot Product
- Rank and Shape
- Building Matrices
- Methods

20 Other Types

- None
- Complex Numbers

None

- A function *always* returns a value. If you don't specify any, then the object `None` is returned.

Example

```
def f():
   "do nothing"
r = f()
print r
   None
```

None

- A function *always* returns a value. If you don't specify any, then the object `None` is returned.
- Sometimes you want to explicitly return `None` to get out of the function.

Example

```
def f():
   ...
   return None
```

None

- A function *always* returns a value. If you don't specify any, then the object `None` is returned.
- Sometimes you want to explicitly return `None` to get out of the function.
- `None` is also useful for *default arguments*.

Example

```
def f(arg=None):
  if arg is None:
    arg = []
  ...
```

Complex Numbers

Complex numbers are as easy to handle as real numbers. Just use the syntax `<number>j` to refer to an imaginary number.

```
I = 1j
I**2 # -1
abs(I) # 1
I*(1+I) # -1 + i
```

Complex Numbers

Complex numbers are as easy to handle as real numbers. Just use the syntax `<number>j` to refer to an imaginary number.

```
I = 1j
I**2 # -1
abs(I) # 1
I*(1+I) # -1 + i
```

The *real* and *imaginary* parts of a complex number are given by the `real` and `imag` properties

```
z = (1+1j)**2
z.real # 0
z.imag # 2
```

Complex Numbers

Complex numbers are as easy to handle as real numbers. Just use the syntax `<number>j` to refer to an imaginary number.

```
I = 1j
I**2 # -1
abs(I) # 1
I*(1+I) # -1 + i
```

The *real* and *imaginary* parts of a complex number are given by the `real` and `imag` properties or with the `numpy` function `real` and `imag` for *arrays only*.

```
z = (1+1j)**2
z.real # 0
z.imag # 2
```

```
zs = array([1., 3.]) +
1j*array([2.,4.])
real(zs) # array([1., 3.])
imag(zs) # array([2., 4.])
```

Computational Mathematics with Python

Modules, Booleans(cont.), Files, Recursions etc.

Olivier Verdier and Claus Führer

Spring 2009

Computational Mathematics with Python

Modules, Booleans(cont.), Files, Recursions etc.

Olivier Verdier and Claus Führer

Spring 2009

- Modules

First steps with Modules

Python comes along with many different *libraries*. You may also install more of those.

`numpy` and `Scipy` are examples of such libraries.

You may either

- load some objects only:

```
from numpy import array, vander
```

First steps with Modules

Python comes along with many different *libraries*. You may also install more of those.

`numpy` and `Scipy` are examples of such libraries.

You may either

- load some objects only:

```
from numpy import array, vander
```

- or load the entire library:

```
from numpy import *
```

First steps with Modules

Python comes along with many different *libraries*. You may also install more of those.

numpy and Scipy are examples of such libraries.

You may either

- load some objects only:

```
from numpy import array, vander
```

- or load the entire library:

```
from numpy import *
```

- or choose to import all the library inside a single variable

```
import numpy
...
numpy.array(...)
```

How to avoid to “destroy” functions

A usual mistake

```
from scipy.linalg import eig
     A=array([[1,2],[3,4]])
     (eig,eigvec)=eig(A)
        .....
     (c,d)=eig(B)  # raises an error, which?
```

How to avoids these unintended effects:

Better with import ... as construct

```
import scipy.linalg as sl
    A=array([[1,2],[3,4]])
    (eig,eigvec)=sl.eig(A)  # eig and sl.eig are different obj
        .....
    (c,d)=sl.eig(B)
```

- Modules

Boolean casting

Definition

Using an `if` statement with a non-boolean type *casts* it to a boolean. The rules are as follow:

bool	`False`	`True`
string	`''`	`'not empty'`
number	0	$\neq 0$
list	`[]`	`[...]` (not empty)
tuple	`()`	`(..,..)` (not empty)

Boolean casting

Definition

Using an `if` statement with a non-boolean type *casts* it to a boolean. The rules are as follow:

bool	`False`	`True`
string	`''`	`'not empty'`
number	0	$\neq 0$
list	`[]`	`[...]` (not empty)
tuple	`()`	`(..,..)` (not empty)
array	Error!	

Notice that almost *all* types will be silently cast to booleans *except* arrays.

Examples

Empty list test

```
# L is a list
if L:
  print "list not empty"
else:
  print "list is empty"
```

Examples

Empty list test

```
# L is a list
if L:
  print "list not empty"
else:
  print "list is empty"
```

Parity test

```
# n is an integer
if n % 2:
  print "n is odd"
else:
  print "n is even"
```

- Modules

Avoid Recursion!

Recursion

```
def f(N):
  if N==0: return 0
  return f(N-1)
```

This is the *simplest* recursion program. In python it chokes for $N \geq 1000$. Note that this is the *best case* since this program doesn't do *anything*.

Iteration

```
for i in xrange(
          10000000):
  pass
```

In iterative programming those loops may at least be 10000 times bigger! That is an empty `for` loop of 10 000 000 is not a problem.

Avoid Recursion!

Recursion

```
def f(N):
  if N==0: return 0
  return f(N-1)
```

This is the *simplest* recursion program. In python it chokes for $N \geq 1000$. Note that this is the *best case* since this program doesn't do *anything*.

Iteration

```
for i in xrange(
          10000000):
  pass
```

In iterative programming those loops may at least be 10000 times bigger! That is an empty `for` loop of 10 000 000 is not a problem.

Note that in some very special cases (tree traversal) recursion is almost unavoidable.

File I/O

File I/O (in- and output) is essential when

- working with measured or scanned data
- interacting with other programs
- saving information for comparions or other postprocessing needs
- .....

File objects

A file is a Python object with associated methods:

```
myfile=open('measurement.dat','r') # creating a read-only file
```

and here a generator to extract the data:

```
for line in myfile:
    data=line.split(';')
    print 'time %s sec temperature %s C' %(data[0],data[1])
```

or direct extraction into a list

```
data=list(myfile)
```

File close method

A file has to be closed before it can be reread.

```
myfile.close() # closes the file object
```

It is automatically closed when

- the program ends

Before a file is closed, you won't see any changes in it by an external editor!

File close method

A file has to be closed before it can be reread.

```
myfile.close() # closes the file object
```

It is automatically closed when

- the program ends
- the enclosing program unit (e.g. function) is left .

Before a file is closed, you won't see any changes in it by an external editor!

File Modes

```
file1=open('file1.dat','r')  # read only
file2=open('file2.dat','r+')  # read/write
file3=open('file3.dat','a')  # append (write to the end of th
file4=open('file4.dat','w') # (over-)write the file
```

The modes 'r','r+','a' require that the file exists.

File append example

```
file3=open('file3.dat','a')
file3.write('something new\n') # Note the '\n'
```

Module: pickle

read and write methods convert data to strings.
Complex data types (like arrays) cannot be written this way.
pickle solves this problem:

Pickle dump examples

```
import pickle
myfile=open('file.dat','w')
a=rand(200)
pickle.dump(a,myfile)
```

you can pickle any python object even code, e.g. functions.

Module: pickle

Pickle load examples

```
import pickle
myfile=open('file.dat','r')
a=pickle.load(myfile)  # restores the array
```

Computational Mathematics with Python

Boolean arrays, more on iterations, from datatypes top classes

Olivier Verdier and Claus Führer

Spring 2009

Boolean Arrays

One may create views of an array using *boolean arrays*.

```
B = array([[True, False],
           [False, True]])
M = array([[2, 3],
           [1, 4]])
M[B] = 0
M # [[0, 3], [1, 0]]
M[B] = 10, 20
M # [[10, 3], [1, 20]]
```

Creating boolean Arrays

It might be just as tedious to create the boolean array by hand than to change the array directly. There are however many methods to create boolean arrays.

Creating boolean Arrays

It might be just as tedious to create the boolean array by hand than to change the array directly. There are however many methods to create boolean arrays.

Any *logical operator* will create a boolean array instead of a boolean.

```
M = array([[2, 3], [1, 4]])
M > 2 # array([[False, True], [False, True]])
M == 0 # array([[False, False], [False, False]])
N = array([[2, 3], [0, 0]])
M == N # array([[True, True], [False, False]])
...
```

Creating boolean Arrays

It might be just as tedious to create the boolean array by hand than to change the array directly. There are however many methods to create boolean arrays.

Any *logical operator* will create a boolean array instead of a boolean.

```
M = array([[2, 3], [1, 4]])
M > 2 # array([[False, True], [False, True]])
M == 0 # array([[False, False], [False, False]])
N = array([[2, 3], [0, 0]])
M == N # array([[True, True], [False, False]])
...
```

This allows the elegant syntax:

```
M[M>2] = 0
# all the elements > 2 are replaced by 0
```

Comparing Arrays

Note that because array comparison create boolean arrays *one cannot compare arrays directly*.

Comparing Arrays

Note that because array comparison create boolean arrays *one cannot compare arrays directly*.

The solution is to use the methods `all` and `any`:

```
A = array([[1,2],[3,4]])
B = array([[1,2],[3,3]])
A == B # creates array([[True, True], [True, False]])
(A == B).all() # False
(A != B).any() # True
```

Boolean Operations

For the same reason as before you *cannot* use `and`, `or` nor `not` on boolean arrays! Use the following replacement operators instead:

logic operator	replacement for bool arrays
`A and B`	`A & B`
`A or B`	`A | B`
`not A`	`-A`

```
a = array([True, True, False, False])
b = array([True, False, True, False])

a and b # error!
a & b # array([True, False, False, False])
a | b # array([True, Trues, True, False])
-a # array([False, False, True, True])
```

Multiple iteration

Comprehensive double for

```
# M is a matrix
M.shape # (3,4)
flat = [M[i,j] for i in range(M.shape[0])
               for j in range(M.shape[1])]
flat # a list of length 12
```

Multiple iteration

Comprehensive double for

```
# M is a matrix
M.shape # (3,4)
flat = [M[i,j] for i in range(M.shape[0])
                for j in range(M.shape[1])]
flat # a list of length 12
```

zip

```
x_values = [1, 2, 3, 4]
y_values = [10, 20, 30, 40]
for x,y in zip(x_values, y_values):
  print "the value at %f is %f" % (x, y)
```

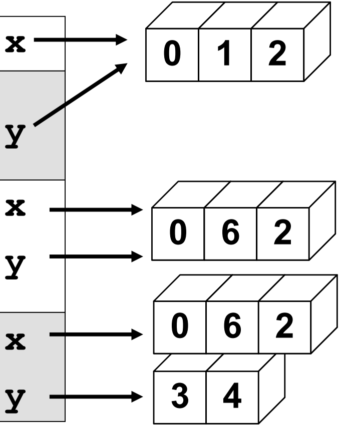
Variables are References

```
>>> x = [0, 1, 2]

# y = x cause x and y to point
# at the same list
>>> y = x

# changes to y also change x
>>> y[1] = 6
>>> print x
[0, 6, 2]

# re-assigning y to a new list
# decouples the two lists
>>> y = [3, 4]
```



(Source: E. Jones and T. Oliphant)

Equality vs Identity I

Definition

- *Identity* is the property of two *variables* to be the reference to the *same* object.
  The identity operator is `is`.
- *Equality* is the property of two *objects* to be equal.
  The equality operator is `==`.

Equality vs Identity II

Example

To test whether to variables are the same *reference* you may use `is`:

```
L = ['a', 'b']
L2 = L
L is L2 # True

L2 = ['a', 'b']
L is L2 # False
L == L2 # True
```

Notice that `is` should *not* be used to compare two objects!

Copy

- To create a *copy* of an object use the function copy of the module copy:

```
c = [3, 4]
d = copy(c)
d[0] = 0
c # [3, 4]
d # [0, 4]
```

Copy

- To create a *copy* of an object use the function copy of the module copy:

```
c = [3, 4]
d = copy(c)
d[0] = 0
c # [3, 4]
d # [0, 4]
```

- For an array it is more efficient to use the *method* copy:

```
A = arange(12).reshape(3,4)
B = A.copy()
B[1,2] = 0 # only changes B, not A
```

Type of an object

Each object has a *type* that may be obtained using the function `type`.

```
x = [1, 2] # list
type(x) # returns list
y = [2, 3]
type(x) == type(y) # True
```

Checking the type

To check the type of a variable, *always* use isinstance:

```
L = [1,2]
if isinstance(L, list):
  print "L is a list"
```

Checking the type

To check the type of a variable, *always* use `isinstance`:

```
L = [1,2]
if isinstance(L, list):
  print "L is a list"
```

We have already come across different types:

- float
- int
- complex
- list
- tuple
- module
- function

Methods and Properties

Definition

Method and property A *method* or property is a function *bound* to an object. The syntax is

```
<object>.method(<arguments...>)
# or
<object>.property
```

Methods and Properties

Definition

Method and property A *method* or property is a function *bound* to an object. The syntax is

```
<object>.method(<arguments...>)
# or
<object>.property
```

- lists: `append(<obj>)`
- arrays: `shape`, `sum()`, `max()` etc.
- complex: `real`, `imag`

Why New Types?

We have already seen many types existing in python. For instance, `float`, `int` and `complex` are types.

Why New Types?

We have already seen many types existing in python. For instance, `float`, `int` and `complex` are types.

What if we want *new structures* to be represented in python? For example

- polynomials
- quaternions (for rotations)
- symbols (for symbolic calculations)
- etc.

Creating new Types

A new type is called a *class*. The syntax is as follows:

```
class Complex(object):
  def __init__(self, r, i):
    self.r = r
    self.i = i
```

Creating new Types

A new type is called a *class*. The syntax is as follows:

```
class Complex(object):
  def __init__(self, r, i):
    self.r = r
    self.i = i
```

- `object` is a *keyword*
- `__init__` is called at the creation of the object
- self is the object itself

Usage Example

One may use this class using the following syntax:

```
z = Complex(2,3)
z.r # 2
z.i # 3
```

Adding methods

A *method* is a function for a particular class.

```
class Complex(object):
...
  def module(self):
    return sqrt(self.r**2 + self.i**2)
```

Adding methods

A *method* is a function for a particular class.

```
class Complex(object):
...
  def module(self):
    return sqrt(self.r**2 + self.i**2)
```

It is used as the other methods you know:

```
z = Complex(4,3)
z.module() # 5
```

The `__init__` method

- Called at the creation of the object

```
z = Complex(2,3) # here __init__ is called
```

- Often used to *initialize* the state of the object
- optional if no initialisation has to be done

The `self` argument

All the methods get a *special first argument*. It is often called `self` by convention.

This argument contains the *object* itself. So a method is basically *a usual function* which gets an object as a first argument.

The `self` argument

All the methods get a *special first argument*. It is often called `self` by convention.

This argument contains the *object* itself. So a method is basically *a usual function* which gets an object as a first argument.

More generally, if an object `obj` is of class `cls` then the following calls are equivalent:

```
obj.method(arg1, arg2)
cls.method(obj, arg1, arg2)
```

The `self` argument

All the methods get a *special first argument*. It is often called `self` by convention.

This argument contains the *object* itself. So a method is basically *a usual function* which gets an object as a first argument.

More generally, if an object `obj` is of class `cls` then the following calls are equivalent:

```
obj.method(arg1, arg2)
cls.method(obj, arg1, arg2)
```

In the previous example the two calls are equivalent:

```
z = Complex(2,3) # the type of z is Complex
z.abs()
Complex.abs(z) # exactly equivalent to z.abs()
```

Computational Mathematics with Python

Classes

Olivier Verdier and Claus Führer

Spring 2009

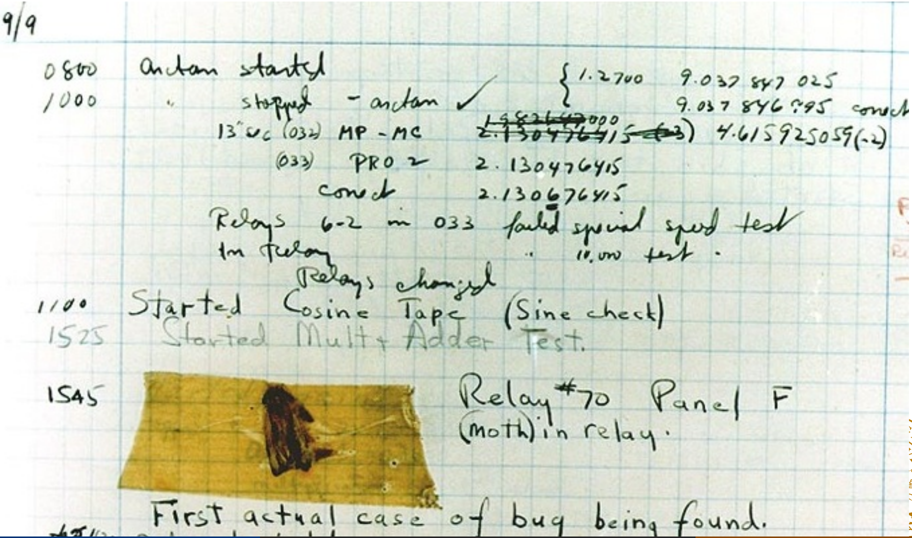
29 Debugging

- Motivation
- Stack
- Debugging Mode

30 Operator Overloading

- Operators
- Brackets

What is a Bug?


9/9
0800 Antan started
1000 " stopped - antan ✓
{ 1.2700 9.037 847 025
9.037 846 995 conct
13" uc (032) MP - MC 4.615925059(-2)
(033) PRO 2 2.130476415
conct 2.130676415
Relays 6-2 in 033 failed special speed test
in relay " 10.000 test .
Relays changed
1100 Started Cosine Tape (Sine check)
1525 Started Mult+ Adder Test.
1545 Relay #70 Panel F
(moth) in relay.
First actual case of bug being found.

Bugs in Python

There are two kinds of bug:

- An exception is raised and not catched
- The code does not function properly

Bugs in Python

There are two kinds of bug:

- An exception is raised and not catched ← Easy to fix
- The code does not function properly

Bugs in Python

There are two kinds of bug:

- An exception is raised and not catched←Easy to fix
- The code does not function properly←*More difficult*

Bugs in Python

There are two kinds of bug:

- An exception is raised and not catched←Easy to fix
- The code does not function properly←*More difficult*

We are only concerned with the first case in what folows.

The Stack

When an exception is raised you see the *stack*. It is all the functions that called the function where the exception was raised.

The Stack

When an exception is raised you see the *stack*. It is all the functions that called the function where the exception was raised.

Example

```
def f():
   ...
   g()
def g():
   ...
   h()
def h():
   raise Exception()

f()
```

The stack is now `f`, `g`, `h`.

Debugging in IPython

After an exception was raised, enter the debug mode by typing `debug`.
You are now in *debug mode*.
You may now inspect the current variables and work as usual.

Demo

Debug Commands

- **h** help
- **q** quit
- **l** shows current line
- **u** go up in the stack
- **d** go down in the stack

Short Variable Names

If you want to inspect a variable with name, for example, `h`, you must use `!h`.

Trigger the Debug Mode

- The simplest way to trigger the debug mode is to throw an exception. For example by writing 1/0 where you want the execution to stop (but you can't resume it).
- A nicer way is to use the following code

```
# at the beginning of the file:
from IPython.Debugger import Tracer
dh = Tracer()
...
dh() # triggers the debug mode
```

The advantage with that approach is that you may now *resume the execution*.

Operators

Operators (+, *, -, etc.) are defined as *methods* in python.

operator	method
+	__add__
*	__mul__
-	__sub__
/	__div__

Operators

Operators (+, *, -, etc.) are defined as *methods* in python.

operator	method
+	__add__
*	__mul__
-	__sub__
/	__div__

Example

Given two lists L1 and L2, the following are exactly equivalent:

```
L = L1 + L2
L = L1.__add__(L2)
```

Redefining Operators

In your own classes you may *define* the meaning of operators!

```
class polynomial(object):
...
  def __add__(self, other):
     ...
     return ...
```

Redefining Operators

In your own classes you may *define* the meaning of operators!

```
class polynomial(object):
...
  def __add__(self, other):
    ...
    return ...
```

You may use that code in this way:

```
p1 = polynomial(...)
p2 = polynomial(...)
p = p1 + p2 # here p1.__add__ is called
```

More Operators

Many more operators may be redefined:

- `==`, `!=`
- `<=`, `<`, `>`, `>=`
- `+=`, `-=`, `*=`, `/=`
- `%`
- etc.

You will find a complete list on the Python Quick Reference Page
(`http://rgruet.free.fr/PQR25/PQR2.5.html#SpecialMethods`)

Brackets

You may also redefine the *brackets*:

bracket	method
()	`__call__`
[]	`__getitem__`

Brackets

You may also redefine the *brackets*:

bracket	method
()	`__call__`
[]	`__getitem__`

Example

```
class polynomial(object):
...
  def __call__(self, x):
    return self.eval(x)
```

Which now may be used as:

```
p = polynomial(...)
p(3.) # value of p at 3.
```

Computational Mathematics with Python

Classes

Olivier Verdier and Claus Führer

Spring 2009

Functions

Functions are building blocks of computer programming.

Functions

Functions are building blocks of computer programming.

The disadvantage is: they are *stateless*.

Functions

Functions are building blocks of computer programming.

The disadvantage is: they are *stateless*.

A function

- takes an input
- computes, creates local variables
- *cleans everything except the output*
- returns the output
- ...and dies

Example: Output

```
def newton(f, x0, tol):
    ...
    return solution
```

Example: Output

```
def newton(f, x0, tol):
    ...
    return solution
```

But I want the number of iteration used.

Example: Output

```
def newton(f, x0, tol):
   ...
   return solution
```

But I want the number of iteration used.

Bad solution:

```
def newton(f,x0,tol):
   ...
   return solution, nb_iterations
```

Bad solution because now it *always* returns the number of iterations.

Example: Output

```
def newton(f, x0, tol):
  ...
  return solution
```

But I want the number of iteration used.

Bad solution:

```
def newton(f,x0,tol):
  ...
  return solution, nb_iterations
```

Bad solution because now it *always* returns the number of iterations.

It is as good as it gets with functions

Example: Input

```
# many arguments here:
def integrate(f, a, b, hmin, tol, ...):
```

Example: Input

```
# many arguments here:
def integrate(f, a, b, hmin, tol, ...):
```

Usage:

```
>>> integrate(f,0,1,.003,1e-6,...)
<bad result>
>>> integrate(f,0,1,.002,1e-6,...)
<bad result>
```

Example: Input

```
# many arguments here:
def integrate(f, a, b, hmin, tol, ...):
```

Usage:

```
>>> integrate(f,0,1,.003,1e-6,...)
<bad result>
>>> integrate(f,0,1,.002,1e-6,...)
<bad result>
```

When the function dies, obviously it also forgets everything about the input arguments!
You have to give the full set of input at every call.

The solution: objects

An object can perform operations on its data and *stay alive*.

The solution: objects

An object can perform operations on its data and *stay alive*.

Advantages are:

Output Methods return only the essential result, but intermediary steps are available

Input The inputs are stored and one can modify one part of the inputs

Example: Output

```
class Newton(object):
  def run(self, x0):
    ...
    # store the nb_iterations in the object:
    self.nb_iterations = ...
    return solution
```

Example: Output

```
class Newton(object):
  def run(self, x0):
    ...
    # store the nb_iterations in the object:
    self.nb_iterations = ...
    return solution
```

Usage:

```
solver = Newton(f)
solution = solver.run(.2)
# now if I want the number of iterations:
nbIter = solver.nb_iterations
```

Example: Output

```python
class Newton(object):
  def run(self, x0):
    ...
    # store the nb_iterations in the object:
    self.nb_iterations = ...
    return solution
```

Usage:

```python
solver = Newton(f)
solution = solver.run(.2)
# now if I want the number of iterations:
nbIter = solver.nb_iterations
```

Possible because the the *object* solver *stays alive* (although the *function* run dies).

Example: Input

```
class Integrator(object):
  def __init__(self, f, a, b, tol, ...):
    self.f = f
    self.a = a
    ...
  def compute(self):
    ...
```

Usage:

```
>>> sin_int = Integrator(sin, 0, 1, 1e-6)
>>> sin_int.compute()
<bad result>
>>> sin_int.tol = 1e-3
>>> sin_int.compute()
```

Now you can change arguments one at a time.

Methods

Method

A *method* is a function *bound* to an object. The syntax is

```
<object>.method(<arguments...>)
# or
<object>.property
```

Methods

Method

A *method* is a function *bound* to an object. The syntax is

```
<object>.method(<arguments...>)
# or
<object>.property
```

For instance:

- list: `append(<obj>)`
- array: `shape`, `sum()`, `max()` etc.
- complex: `real`, `imag`
- Complex: `r`, `i`

Class attributes

- Attributes in the class declaration are *always* class attributes (logical but different from C++)

```
class Newton(object):
  tol = 1e-8 # this is a *class* attribute!
N1 = Newton(f)
N2 = Newton(g)
N1.tol # 1e-8
N2.tol = 1e-4 # relax tolerance for g
N1.tol # 1e-8 only N2 was changed
```

- Class attributes are handy to simulate *default values*.

```
Newton.tol = 1e-10 # now all the Newton classes have 1e-5
N2.tol # 1e-4 because the object attribute is fetched
```

Class and object syntax

- Attributes may be added, removed and accessed at any time by anybody

```
class C:
  pass # the class is empty
c = C()
c.first = 1 # attributes created dynamically
c.second = 3
```

- You may put *any kind of code* inside a `class` statement! This code is read only *once*.

```
class C:
  2+2 # why not?
  def f(self):
    ...
  g = f # now the method g is added to the class
```

Documentation

Document classes, functions or modules by adding a string in the beginning of the indentation:

```
class myClass (object):
  """This class does this and that"""
  def f(self):
    """Does nothing"""
```

Debugging

For debugging purposes you may use the `__repr__` method:

```
class Complex(object):
  ...
  def __repr__(self):
    return "%f + %f.i" % (self.a, self.b)

# in the console:
>>>z = Complex(1,2)
>>>print z
1 + 2.i
>>>z
1 + 2.i
```

Almost everything is an object

Almost everything is an object in Python.

```
def my_sine(...):
    ...

def my_cosine(...):
    ...

# this is allowed:
my_sine.derivative = my_cosine
```

Note that:

- functions, as anything else, are *objects*
- we added a *new attribute* to an existing object

Don't

- No `while`
- No `global`
- No `map`
- No `lambda`

Don't

- No `while`
- No `global`
- No `map`
- No `lambda`

Don't use recursive programming: it is slow and unreliable.

No While

Don't use while:

```
while <condition>:
   ...
```

because you don't know whether it will ever stop.

No While

Don't use while:

```
while <condition>:
   ...
```

because you don't know whether it will ever stop.

Always use this instead:

```
max_it = 100
for k in xrange(max_it):
   ...
else:
  raise Exception("No convergence in %d iterations" % k)
```

Function arguments

A function has access to variables outside their scope. *Don't* use this feature, the outcome is not reliable.

```
tolerance = 1e-6
def algorithm():
    ...
    if abs(error) < tolerance # bad!
```

Function arguments

A function has access to variables outside their scope. *Don't* use this feature, the outcome is not reliable.

```
tolerance = 1e-6
def algorithm():
   ...
   if abs(error) < tolerance # bad!
```

Give *all* the arguments as inputs:

```
def algorithm(tolerance):
   ...
```

Errors

Don't print out errors:

```
print "The algorithm did not converge" # bad!
```

it lacks information both for humans and computers.

Errors

Don't print out errors:

```
print "The algorithm did not converge" # bad!
```

it lacks information both for humans and computers.

Throw an *exception* instead (with a message):

```
raise Exception("The algorithm did not converge")  # good!
```

For loops

Don't use:

```
for k in range(...):
  ...
  element = my_list[k]
```

unless you have a good reason to do so.

For loops

Don't use:

```
for k in range(...):
    ...
    element = my_list[k]
```

unless you have a good reason to do so.

A better way is often:

```
for element in my_list:
    ...
```

or:

```
for element in my_generator:
```

because it is easier to read.

No append

Avoid append. It is often used as:

```
my_list = []
for k in xrange(n):
  ... # compute some value here
  my_list.append(value)
```

No append

Avoid append. It is often used as:

```
my_list = []
for k in xrange(n):
  ... # compute some value here
  my_list.append(value)
```

Instead, use a *generator*:

```
def generate_values(n):
  for k in xrange(n):
    ...
    yield value

# this is much more readable:
my_list = list(generate_values(n))
```

Documentation and Testing

Do give a short documentation to all your functions.

Documentation and Testing

Do give a short documentation to all your functions.

Always add some test function:

```
def algorithm(...):
  ...

def test_algorithm():
  expected = 3
  computed = algorithm(...)
  assert expected == computed
```

The keyword `assert` will raise an exception if the statement is not evaluated to true.

Understanding the Formula

The original formula was:

$$c_i^j = \frac{c_{i+1}^{j-1} - c_i^{j-1}}{x_{i+j} - x_i}$$

with the initialisation:

$$c_i^0 = y_i$$

Understanding the Formula

The original formula was:

$$c_i^j = \frac{c_{i+1}^{j-1} - c_i^{j-1}}{x_{i+j} - x_i}$$

with the initialisation:

$$c_i^0 = y_i$$

So if we define the operator:

$$(\Delta_j x)_i = x_{i+j} - x_j$$

Then:

$$c^j = \frac{\Delta_1 c^{j-1}}{\Delta_j x}$$

Understanding the Formula

The original formula was:

$$c_i^j = \frac{c_{i+1}^{j-1} - c_i^{j-1}}{x_{i+j} - x_i}$$

with the initialisation:

$$c_i^0 = y_i$$

So if we define the operator:

$$(\Delta_j x)_i = x_{i+j} - x_j$$

Then:

$$c^j = \frac{\Delta_1 c^{j-1}}{\Delta_j x}$$

How do we program the opertor Δ_j in python?

Δ_j

Given a vector y the operator Δ_j is in python:

Δ_j

Given a vector y the operator Δ_j is in python:

```
y[j:] - y[:-j]
```

Δ_j

Given a vector y the operator Δ_j is in python:

```
y[j:] - y[:-j]
```

Isn't it simple, short and readable?

Fetching the x and y values

The interpolation points are a list of tuples:

```
xy = [(0.,1.), (1.,2.4),...]
```

How to obtain the x and y values?

Fetching the x and y values

The interpolation points are a list of tuples:

```
xy = [(0.,1.), (1.,2.4),...]
```

How to obtain the x and y values?
Use *arrays*!!

```
xy = array(xy) # now xy is an nx2 matrix
xy[:,0] # x values
xy[:,1] # y values
```

That is what matrices are for.

The Full Code of divdiff

```
x = xy[:,0] # x values
row = xy[:,1] # first row

for j in xrange(1,len(xy)):
  yield row[0]
  row = (row[1:] - row[:-1])/(x[j:] - x[:-j])
```

Computational Mathematics with Python

Case Study: Initial Value Problem

Olivier Verdier and Claus Führer

Spring 2009

35 Initial Value Problem

- Mathematical View
- Class IVPSolver

35 Initial Value Problem

- Mathematical View
- Class IVPSolver

IVP

The problem:
Find a function $u : [t_0, t_f] \rightarrow \mathbb{R}^n$ with the property

$$\dot{u}(t) = f(t, u(t)) \quad u(t_0) = u_0$$

is called an initial value problem (IVP).
f is called the right-hand side function, u_0 the initial value.

Simple Example

The problem

$$\dot{u}(t) = -3u(t) \quad u(t_0) = 5$$

has the solution

$$u(t) = \mathrm{e}^{-3(t-t_0)}5$$

Discretisation

A numerical method computes approximations to the solution at discrete times

$$t_0 < t_1 < \ldots < t_n = t_f$$

with a step size $h_i = t_{i+1} - t_i$.

Euler's Method

Euler's explicit method is defined by

$$u_{i+1} = u_i + h_i f(t_i, u_i) \quad t_{i+1} = t_i + h_i$$

where u_i is an approximation to $u(t_i)$.

Runge-Kutta's Classical Method

Here u_{i+1} is computed by passing some intermediate *stages*:

$$\begin{aligned}
U_1 &= f(t_i, u_i)\\
U_2 &= f(t_i + \frac{h_i}{2}, u_i + \frac{h_i}{2} U_1)\\
U_3 &= f(t_i + \frac{h_i}{2}, u_i + \frac{h_i}{2} U_2)\\
U_4 &= f(t_i + h_i, u_i + h_i U_3)\\
u_{i+1} &= u_i + \frac{h_i}{6}(U_1 + 2U_2 + 2U_3 + U_4)
\end{aligned}$$

init

We initialize the problem

```
class IVPSolver (object):
    def __init__(self, f, u0, t0=0):
        self.f = f    # rhs function
        self.u0= u0   # initial value
        self.t0= t0   # initial time
        self.ts = [t0]
        self.us = [u0]
```

Class Attributes

We set some predefined (default) values

```
# default values for step size, simulation time and
#  step number delimiter
    h = .01
    time = 1.
    max_steps = 10000
```

A dummy method

Some numerical methods use constant step sizes, other variable step sizes. The change of step sizes is done by a method which is provided here as a dummy method.
It will be replaced later by something meaningful, when variable stepsizes are required.

```
def adjust_step_size(self)
    pass
```

Step generator

We define a method independent step generator

```
def generator(self, t, u, tf):
    """
    Generates the (t,u) values until t > tf
    """
    for i in xrange(self.max_steps):
        t, u = self.step(t, u)
        if t > tf:
            break
        yield t, u
        self.adjust_stepsize()
    else:
        raise Exception(
        "Final time not reached within max_steps steps")
```

This generator is method independent.

Solving the equation

Solving the IVP is done by the run method

```
def run(self, time=None):
    if time is None:
        time = self.time
    # start from the last time we stopped
    t = t0 = self.ts[-1]
    u = self.us[-1]
    tus = list(self.generator(t, u, t0 + time))
    self.ts.extend(q[0] for q in tus)
    self.us.extend(q[1] for q in tus)
    self.ats = array(self.ts)
    self.aus = array(self.us)
```

Note the restarting capability.

Postprocessing

Postprocessing can be a plot or further computations:

```
def plot(self):
    """
    Plot the computed solution.
    """
    if not hasattr(self,'ats'):
        raise Exception('No data to plot.')
    plot(self.ats, self.aus, '.-')
    xlabel('time')
    ylabel('state')
    if self.f.name is not None:
        title(self.f.name)
```

Note the exception. Other possibilities ....

A method (heritage)

The stepper depends on the method:

```
class ExplicitEuler (IVPSolver):
    def step(self, t, u):
        return t + self.h, u + self.h*self.f(t, u)
```

Note, we use here a one step method.

Another method

Or the Runge–Kutta method

```
class RungeKutta4 (IVPSolver):
    def step(self, t, u):
        h=self.h
        U1 = self.f(t, u)
        U2 = self.f(t + h/2., u + h*U1/2.)
        U3 = self.f(t + h/2., u + h*U2/2.)
        U4 = self.f(t + h, u + h*U3)
        return t+h, u + h/6.*(U1 + 2.*(U2 + U3) + U4)
```

Overridding a class method

A method with local error estimation and step size control:

```
class RungeKutta34 (IVPSolver):
    tol = 1e-6
    def adjust_stepsize(self):
        self.h *= (self.tol/self.error)**(1/4)

    def step(self, t, u):
        h = self.h
        U1 = self.f(t, u)
        U2 = self.f(t + h/2., u + h*U1/2.)
        U3 = self.f(t + h/2, u + h*U2/2)
        U3_1 = self.f(t + h, u - h*U1 + 2*h*U2)
        U4 = self.f(t + h, u + h*U3)
        self.error = norm(h/6*(2*U2 + U3_1 - 2*U3 - U4))
        return t+h, u + h/6*(U1 + 2*(U2 + U3) + U4)
```

Creating an instance and running

The rhs-function

```
def my_ode(t,u):
    return -3*u
my_ode.name="u'=-3*u"
```

Creating an instance, solving and plotting:

```
rK=RungeKutta4(my_ode,-5,0)
rK.run()
rK.plot()
ee=ExplixitEuler(my_ode,-5,0)
ee.run()
ee.plot()
```

Linear Systems

More general rhs functions:

```
def make_lin(A):
    if np.isscalar(A):
        def lin(t,u):
             return A*u
    else:
        def lin(t, u):
            return dot(A,u)
    lin.exact = ....
    lin.name =  ....
    return lin
```

Computational Mathematics with Python

Broadcasting

Olivier Verdier and Claus Führer

Spring 2009

What is slow?

Any repeated task in python is *slow*.

Slow:

```
for i in range(n):
  for j in range(n):
    v.append(A[i,j] *
x[j])
```

Slow because *interpreted* code

Fast:

```
v = dot(A,x)
```

Faster because, under the hood, it is *compiled* code.

Slow/Fast

Relatively slow:

- `for` loops
- `append` for lists
- generators
- comprehensive lists/generators

Fast, `numpy` operations:

- `dot`
- `array` operations
- `solve`
- etc.

Vectorization

To improve the performance, one has often to *vectorize*, i.e., to replace for loops by numpy functions.

```
# v is a vector
# we want to shift its values by 5

# slow:
for i in range(len(v)):
  v[i] += 5

# fast:
v += 5
```

Why Broadcasting

Take two functions $f(x)$, $g(x)$ and create a new function

$$F(t, x) = f(x) + g(t)$$

Why Broadcasting

Take two functions $f(x)$, $g(x)$ and create a new function

$$F(t, x) = f(x) + g(t)$$

You have actually *broadcast* the function f and g as follows:

$$f \to \bar{f}(t, x) = f(x)$$

$$g \to \bar{g}(t, x) = g(t)$$

and now $F = \bar{f} + \bar{g}$.

Simple Example

One of the simplest example of broadcasting in mathematics is constants being broadcast to function.
If C is a scalar one often writes:

$$f := \sin + C$$

Simple Example

One of the simplest example of broadcasting in mathematics is constants being broadcast to function.
If C is a scalar one often writes:

$$f := \sin + C$$

this is an *abuse of notation* since one should not be able to add functions and constants. Constants are however implicitly *broadcast* to functions:

$$\bar{C}(x) := C \quad \forall x$$

and now

$$f = \sin + \bar{C}$$

Mechanism

A mechanism to achieve broadcasting automatically is as follows:

1. - First *reshape* the function g to $\tilde{g}(t,0) := g(t)$
   - Then reshape f to $\tilde{f}(0,x) := f(x)$

   Now both f and g take *two* arguments.
2. - Extend f to $\bar{f}(t,x) := \tilde{f}(0,x)$
   - Extend g to $\bar{g}(t,x) := \tilde{g}(t,0)$

Several Variables

This construction extends readily to functions of several variables. Say you want to construct

$$F(x, y, z) = f(x, y) * g(z)$$

- f is broadcast to $\bar{f}(x, y, z) = f(x, y)$
- g is broadcast to $\bar{g}(x, y, z) := g(z)$

Conventions

By convention a function is automatically reshaped by adding zeros on the *left*:

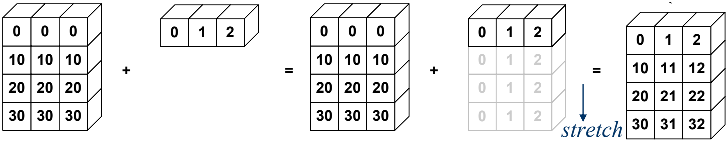
$$g(x, y) \longrightarrow g(0, 0, \ldots, 0, x, y)$$

Array Broadcasting

Arrays are just particular cases of functions.
Broadcasting is done automatically in numpy.

Example

Adding a matrix of shape (4,3) and a matrix of size (1,3). The second matrix is *extended* to the shape (4,3).



(Source: E. Jones and T. Oliphant)

The Broadcasting Problem

Problem

Given an array of shape s_1 broadcast it to the shape s_2

The Broadcasting Problem

Problem

Given an array of shape s_1 broadcast it to the shape s_2

This is done in two steps:

1. If the shape s_1 is shorter than the shape s_2 then *ones are added on the left* of the shape s_1. This is *a reshaping*.
2. When the shapes have the same length the array is *extended* to match the shape s_2 (if possible).

The Broadcasting Problem

Problem

Given an array of shape s_1 broadcast it to the shape s_2

This is done in two steps:

1. If the shape s_1 is shorter than the shape s_2 then *ones are added on the left* of the shape s_1. This is *a reshaping*.
2. When the shapes have the same length the array is *extended* to match the shape s_2 (if possible).

Example

You want to add a vector of shape (3,) to a matrix of shape (4,3). The vector needs be *broadcast*.

1. (3,) $\rightarrow$ (1,3)
2. (1,3) $\rightarrow$ (4,3)

Example

v is a vector of length n
It is to be broadcast to the shape (m, n).

1. v is *automatically reshaped* to the shape $(1, n)$
2. v is *extended* to (m, n)

Example

v is a vector of length n
It is to be broadcast to the shape (m, n).

1. v is *automatically reshaped* to the shape $(1, n)$
2. v is *extended* to (m, n)

```
M = array([[11, 12, 13, 14],
           [21, 22, 23, 24],
           [31, 32, 33, 34]])
v = array([100, 200, 300, 400])
M + v # works directly
```

Example

v is a vector of length n

It is to be broadcast to the shape (m, n).

1. v is *automatically reshaped* to the shape $(1, n)$
2. v is *extended* to (m, n)

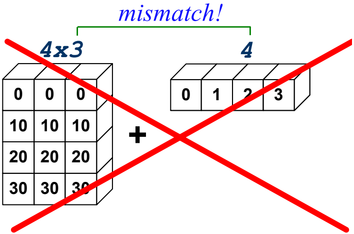
```
M = array([[11, 12, 13, 14],
           [21, 22, 23, 24],
           [31, 32, 33, 34]])
v = array([100, 200, 300, 400])
M + v # works directly
```

Result:

$$\begin{bmatrix} 111 & 212 & 313 & 414 \\ 121 & 222 & 323 & 424 \\ 131 & 232 & 333 & 434 \end{bmatrix}$$

Shape Mismatch

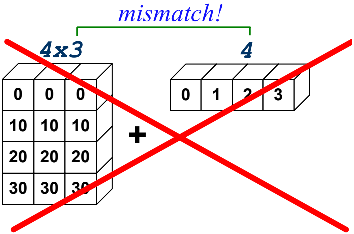
To broadcast a vector of length n to the shape (n, m) the automatic reshaping *will not work*.



(Source: E. Jones and T. Oliphant)

Shape Mismatch

To broadcast a vector of length n to the shape (n, m) the automatic reshaping *will not work*.



(Source: E. Jones and T. Oliphant)

1. one manually reshapes v to the shape $(n, 1)$
2. v is ready to be extended to (n, m)

Shape Mismatch Example

```
M = array([[11, 12, 13, 14],
           [21, 22, 23, 24],
           [31, 32, 33, 34]])
v = array([100, 200, 300])
M + v # error!
M + v.reshape(-1,1)
```

Shape Mismatch Example

```
M = array([[11, 12, 13, 14],
           [21, 22, 23, 24],
           [31, 32, 33, 34]])
v = array([100, 200, 300])
M + v # error!
M + v.reshape(-1,1)
```

Result:

$$\begin{bmatrix} 111 & 112 & 113 & 114 \\ 221 & 222 & 223 & 224 \\ 331 & 332 & 333 & 334 \end{bmatrix}$$

Typical Examples

- $M + C$ Matrix plus constant

Typical Examples

- $M + C$ Matrix plus constant
- Multiply all the *columns* of a matrix by column dependent coefficients in a vector V

```
M * V
```

Typical Examples

- $M + C$ Matrix plus constant
- Multiply all the *columns* of a matrix by column dependent coefficients in a vector V

```
M * V
```

- Multiply the *rows* of a matrix by coefficients in a vector V:

```
M * V.reshape(-1,1)
```

Typical Examples

- $M + C$ Matrix plus constant
- Multiply all the *columns* of a matrix by column dependent coefficients in a vector V

```
M * V
```

- Multiply the *rows* of a matrix by coefficients in a vector V:

```
M * V.reshape(-1,1)
```

- Compute the tensor product $M_{i,j} = V_i W_j$

```
M = V.reshape(-1,1) * W
```

Computational Mathematics with Python

Modules

Olivier Verdier and Claus Führer

Spring 2009

38 Modules
- Imports and run
- __main__ variable

39 Dictionaries and Arguments
- Dictionaries
- Function Arguments

40 Tests
- What are tests
- nosetest

38 Modules
- Imports and run
- ___main___ variable

39 Dictionaries and Arguments
- Dictionaries
- Function Arguments

40 Tests
- What are tests
- nosetest

Import

To load the contents of a file you may use `import` but the file is loaded *only once*.

Import

To load the contents of a file you may use `import` but the file is loaded *only once*.

The various syntax are

- `from module import something`
- `from module import *`
- `import module`

IPython's run command

- IPython has a special command named `run` which executes a file *as if you ran it directly in python*.
- This means that the file is executed *independently of what is already defined in IPython*.
- This is the recommended way to execute files.

You must import all you need in the executed file.

Typical Example

```
from numpy import array
...
a = array(...)
```

And in IPython: `run file`

Typical Example

```
from numpy import array
...
a = array(...)
```

And in IPython: `run file`
Everything that is defined in the file is then imported in the IPython workspace.

Where are the commands?

Where are the numpy and scipy commands?

numpy	`array`, `arange`, `linspace`, `vstack`, `hstack`, `dot`, `eye`, `zeros`
numpy.linalg	`solve`, `lstsq`, `eig`, `det`
pylab	`plot`, `legend`, `cla`
scipy.integrate	`quad`
copy	`copy`, `deepcopy`

__name__ and __main__

In a given module the special variable `__name__` is defined to the *name of the current module*.

In the command line (in IPython) this variable is set to `"__main__"` which allows the following trick:

```
# module
import ...

class ...

if __name__ == "__main__":
  # perform some tests here
```

The tests will be run *only* when the file is directly run, *not* when it is imported.

Dictionaries

A dictionary is a structure similar to lists but where *keys are (usually) strings*.
One accesses dictionaries with square brackets.

```
homework_passed = {'Svensson': True, 'Karlsson': False}

homework_passed['Svensson'] # True
# changing a value:
homework_passed['Svensson'] = False
# deleting an item
del homework_passed['Svensson']
```

Looping through Dictionaries

A dictionary is an object with the following useful mehods: `keys`, `items`, `values`.

By default a dictionary is considered as a list of *keys*:

```
for key in homework_passed:
  print "%s %s" % (key, homework_passed[key])
```

Looping through Dictionaries

A dictionary is an object with the following useful mehods: `keys`, `items`, `values`.

By default a dictionary is considered as a list of *keys*:

```
for key in homework_passed:
  print "%s %s" % (key, homework_passed[key])
```

One may also use `items` to loop through keys and values:

```
for key, value in homework_passed.items():
  print "%s %s" % (key, value)
```

Looping through Dictionaries

A dictionary is an object with the following useful mehods: `keys`, `items`, `values`.

By default a dictionary is considered as a list of *keys*:

```
for key in homework_passed:
  print "%s %s" % (key, homework_passed[key])
```

One may also use `items` to loop through keys and values:

```
for key, value in homework_passed.items():
  print "%s %s" % (key, value)
```

One may use the `keys` and `values` methods to copy or change the dictionary:

```
dict_items = zip(homework_passed.keys(),
                 homework_passed.values())
other_dict = dict(dict_items) # same dictionary
```

Function Argument List

Take the function `newton`

```
def newton(f, x0, max_iter=20):
    ...
```

Recall that this function can be called by using

- positional arguments only: `zero=newton(cos,.2,30)`
- keyword arguments only: `zero=newton(f=cos,max_iter=30,x0=.2)`
- or a mixed form: `zero=newton(cos,maxiter=30,x0=.2)`

Function Argument List

```
def newton(f, x0, max_iter=20):
   ...
```

Say that we are given a *list* with the arguments prepared:

```
L = [cos, .2]
```

Function Argument List

```
def newton(f, x0, max_iter=20):
   ...
```

Say that we are given a *list* with the arguments prepared:

```
L = [cos, .2]
```

One may transform this list into an argument list for *positional arguments* with the *single star operator*:

```
newton(L[0], L[1]) # ok but cumbersome
newton(*L) # does the same thing
```

Function Argument Dictionary

Take the function newton

```
def newton(f, x0, max_iter=20):
  ...
```

Similarly, if one is given a *dictionary* one may use for *keyword arguments* the *double star operator*:

```
D = {'x0': .3, 'f': cos}
```

Function Argument Dictionary

Take the function `newton`

```
def newton(f, x0, max_iter=20):
    ...
```

Similarly, if one is given a *dictionary* one may use for *keyword arguments* the *double star operator*:

```
D = {'x0': .3, 'f': cos}
```

One may call

```
newton(D['f'], D['x0']) # ok but cumbersome
newton(**D) # better
```

Passing arguments

Also in the definition of functions you might find these constructs. This is often used to pass parameters through a function

```
def letssee(f, x, *args,**keywords):
    return f(x,*args,**keywords)

def look(x,y,z,u):
    print y,z
    print u
    return x**2
```

A call

```
L=[1,2]
D={'u':15}
letssee(look,3,*L,**D)
```

gives

```
 1 2
15
Out[35]: 9
```

Why Tests?

?

Why Tests?

?

- Because you do them *anyway*
- Because it will keep your code *alive*.

Automated Tests

Automated tests

- ensure a constant (high) quality standard of your code
- serve as a documentation of the use of your code
- document the test cases → test protocol

Example

A matrix property:

Two matrices A,B are called similar, if there excists a matrix S, such that $B = S^{-1}AS$. A and B have the same eigenvalues.

A related piece of code:

```
def gen_similar(A,S):
    return dot(dot(inv(S),A),S)
```

Example (Cont.)

A test of this program can be

```
class test_sim(object):
    tol=1.e-9
    A=array([1,2,3,4]).reshape(2,2)
    S=array([1,-2,7,-4]).reshape(2,2)
    def near_real(self,a,b):
        return abs(a-b) < self.tol

    def test_eig(self):
        """
        Check to see if similar
        """
        B=gen_similar(self.A,self.S)
        assert self.near_real(norm(eig(B)[0]),norm(eig(self.A)
```

Example (Cont.)

even reaction on wrong input should be checked

```
class test_sim(object):
    ...
    S0=zeros((2,2))
    ...

    def test_eig_sing(self):
        """
        We check correct error raising
        """
        try:
            B=gen_similar(self.A,self.S0)
            flag=False
        except LinAlgError:
            flag=True
        assert flag
```

nosetests

There is a nice command which performs all tests automatically:

```
 claus@Claus-Home:~/gen_sim$ nosetests gen_sim.py
..
----------------------------------------------------------------------
Ran 2 tests in 0.390s

OK
```

Test Cases

You will want to *put your tests together*:

```
class Test_Similar(object):
  def setUp(self):
    # define the matrix here
    self.A = <some matrix>
    self.A_similar = gen_similar(A,S)

  def test_eigenvalue(self):
    assert array_almost_equal(eigvals(self.A), eigvals(self.A_s

  def test_singular(self):
    ...
```

Test discovering

`nosetest` will discover all your test automatically provided the name of the file/class/function starts with `test`:

```
nosetests test_file.py:Test_Similar.test_eigenvalue
nosetests test_file.py:Test_Similar
nosetests test_file.py
```

```
or even:
nosetests
```

Testing Tools

- `numpy.testing.assert_array_almost_equal` is very handy; use it to compare vectors or even scalars:

```
expected = array([1.,2.,3]) # or a scalar
computed = my_algorithm()
assert_array_almost_equal(computed, expected)
```

- nose has a number of assert functions: `nose.tools.assert_true`, `nose.tools.assert_raises`, etc.

Advice

When you develop, *do not test your code* in Ipython.

Advice

When you develop, *do not test your code* in Ipython.

Write your tests in a file instead.

Computational Mathematics with Python

Matlab and Concluding Remarks

Olivier Verdier and Claus Führer

Spring 2009

Matlab

Matlab has become the standard language to teach and experiment with scientific computing, especially in Sweden and the US.
Matlab is a commerical product.
There are free clones: *SciLab* and *Octave*.

Matlab

Matlab has become the standard language to teach and experiment with scientific computing, especially in Sweden and the US.

Matlab is a commerical product.

There are free clones: *SciLab* and *Octave*.

We will now focus on the difference between Matlab and Python.

41 Matlab for Python users

42 Matlab

- Syntax
- Lists, Dictionaries, Arrays
- Linear Algebra
- Functions
- Environment

43 What we didn't do in this course

- comments with %

```matlab
% matlab comment
```

```python
# python comment
```

- strings with single quotes
- blocks require no identation but are ended with end

```matlab
if z == 0
dosomething
else
dosomethingelse
end
```

```python
if z == 0:
  dosomething
else:
  dosomethingelse
#
```

Type

- There is no direct equivalent of Python types in Matlab.
- One may however use `whos` which gives information about a given variable.
- All the numbers are *floats* by default (with or without the decimal point).

Operators

The usual operators *, +, /, ==, etc. work as in Python.
The logical operators are different, as shown in the examples below.

```
2^3 % 8
2 ~= 3 % 1

1 && 0 % 0
1 || 0 % 1
~1 % 0
```

```
2**3 # 8
2 != 3 # True

True and False # False
True or False # True
not True # False
```

Printing out

Matlab will always display (echo) what you defined unless you end the assignment with a semicolon

```
% inside a program
a = 10 % prints a=10
a = 10; % silent
```

```
# inside a program
a = 10 # silent
print a # prints 10
```

Operator differences

- No multiple comparison

```
4 > 3 > 2 % false!
```

```
4 > 3 > 2 # True
```

- No increment operators (`+=`, `*=`,...)

Lists, Dictionaries, Arrays

- Python Lists correspond to MATLAB cell arrays

```
L=[[1,2,3,4],'hello']
```

```
L={[1,2,3,4],'hello}
L{1}␣%␣[1,2,3,4]
```

- Python dictionaries correspond to MATLAB structure arrays

```
D={'key1':15.,'key2':-25}
D['key1'] # 15
```

```
D.key1=15
D.key2=-25
```

- Python/numpy arrays correspond to MATLAB arrays (to some extend):

```
A=array([[1,2,3],[4,5,6]])
```

```
A=[1,2,3;4,5,6]
```

Creating matrices

The syntax is

```
M = [1 2 3; 4 5 6]
size(M) % 2 3
```

Blank or comma separates columns.
Semicolon or line break separates rows.

Creating matrices

The syntax is

```
M = [1 2 3; 4 5 6]
size(M) % 2 3
```

Blank or comma separates columns.
Semicolon or line break separates rows. Be careful, this role of the blank is ambiguous:

```
M = [1 - 1] % what will this do?
M = [1 -1] % and this?
```

Creating matrices

The syntax is

```
M = [1 2 3; 4 5 6]
size(M) % 2 3
```

Blank or comma separates columns.
Semicolon or line break separates rows. Be careful, this role of the blank is ambiguous:

```
M = [1 - 1] % what will this do?
M = [1 -1] % and this?
```

Since you may not create vectors you *must* store vectors in row or column matrices.

Transpose

The single quote is used for transposing matrices (and for delimiting strings).

```
s = 'abc'
v = [1:3] % row matrix
v' % column matrix
```

Transpose is often used to transform row matrices to column matrices and vice versa.

Shapes

- Shapes are not tuple.
- The shape is called *size*

```
size(M)
size(M)(1)  % error!
(size(M))(1)  % error!
size(M,1)  % ok

s = size(M)
s(1)  % ok
```

```
shape(M)  # or M.shape
shape(M)[0]  # ok
(shape(M))[0]  # ok


s = shape(M)
s[0]  # ok
```

Operators

Operators on matrices are *linear algebra operators*. Componentwise operators are prefixed with a dot.

```
V = [1 2 3]
W = [3 4 5]
V * W % error!
V .* W % ok
V / W % error!
V ./ W % ok
V^W % error!
V.^W % ok
```

```
V = array([1,2,3])
W = array([3,4,5])

V * W # ok

V/W # ok

V**W # ok
```

Slices

- Slices are closed intervals
- Slice indices *cannot be out of bound*.
- Last index is denoted by `end`
- There are *no half slices*

```
V = [1 2 3]
V(2:end) - V(1:end-1)

V(2:) % error!
V(2:100) % error!
```

```
V = array([1, 2, 3])
v[1:] - v[:-1]

V[1:] # ok
V[1:100] # same as V[1:]
```

for loops

- `for i=M` goes through the *columns* of the matrix M
- `1:n` creates a *row* matrix (similar to `range` in python)
- This allows the often used syntax:

```
for i=1:N
```

- Careful to create a column matrix!

```
% wrong:
c = 1:3' % 3 is transposed first!

% right:
c = (1:3)'
```

Functions in Matlab

There are *two* kinds of functions in Matlab

File Functions

- automatically reloaded
- available from everywhere

Functions in Matlab

There are *two* kinds of functions in Matlab

File Functions

- automatically reloaded
- available from everywhere

Local Functions

- available only in the file they are defined
- must be declared *after* the main file function

Functions

For all matlab functions the syntax is:

```
function [res] = fname(args)
```

```
def function_name(args):
```

For file functions *the function name is not used*; what is used is the name of the *file* instead (similar to shell scripting)

Functions

For all matlab functions the syntax is:

```
function [res] = fname(args)
```

```
def function_name(args):
```

For file functions *the function name is not used*; what is used is the name of the *file* instead (similar to shell scripting)

Functions without arguments do not require parenthesis

```
rand() % random number
rand % same thing
```

```
rand() # random number
rand # the function object
```

Single Return Values

```
function ret = f()
```

To return the value one simply *sets the variable* `ret` *to the value we want to return*.

```
ret = 2 % the function returns 2
```

Notice that the execution will continue after this, unless you write `return` (but *not* `return 2`)

Multiple Return Values I

To return multiple values you may use *structures*

```
ret = {2, 3}
```

Used as:

```
r = f
r{1} % 2
r{2} % 3
% but
f{1} % error!
```

Multiple Return Values II

But the most common way is to use the special syntax:

```
function [ret1,ret2] = f()
```

Note that *this is not like returning a tuple*. It is used as follows

```
a = f() % first value only!!!
a,b = f() % first and second values
```

feval and @

Functions are not objects so you must use a special symbol to pass it around.

```
function [res] = fname(x)
...
end
```

feval and @

Functions are not objects so you must use a special symbol to pass it around.

```
function [res] = fname(x)
...
end
```

You can't pass this function using `fname` because `fname` *executes* the function! You have to *prevent the execution* using the @ operator.

```
f = @fname
% now how to execute the function?
f(x)
```

Functions: differences with python

- Fixed return value(s)
- Multiple return values are *not tuples*
- File function name is not used
- Functions may not be defined in interactive mode
- Each function must be in its own file if it is to be publicly used
- Functions are not objects
- No default arguments
- No named arguments
- No * nor ** argument operators
- No docstring

Mathematical Functions

Most of the mathematical functions, for computing and plotting, have the same name in matlab and scipy. Examples are:

- `plot`, `legend`, `grid`, `pcolor`,...
- `eig`, `eigs`, `svd`, `qr`, ...
- `rand`, ...
- `sin`, `cos`, `tan`,...
- `det`, `norm`, ...
- `linspace`, `logspace`, `roll`, ...
- `real`, `imag`, ...
- `sum`, `max`, `min`, `mean`, `cumsum`, `cumprod`, `diff`, ...

Variables

- There is no notion of reference in matlab. Variables are always *copied* when reassigned.

```
x = [1 2 3; 4 5 6]
y = x
y(1,1) = 0
x % unchanged
```

```
x = array([[1,2,3],[4,5,6]])
y = x
y[0,0] = 0 # x is changed
x # [[0,2,3],[4,5,6]]
```

This is in particular important for function calls.

- Objects are not common (this feature was introduced recently in Matlab).

Modules

There are no modules nor namespaces in matlab. Everything that is in the search path is available.

Exception handling

```
function a=divideme(b,c)
    InfEx=MException('MATLAB:Inf','division␣by␣zero');
    a=b/c;
    if a==inf
      throw(InfEx)
    end
```

```
try
  a=divideme(1,0)
catch InfEx
  disp(InfEx.message)
end
```

```
def divideme(b,c):
    return b/c
try:
    a=divideme(1,0)
except:
    print 'Division by zero'
```

License

- Matlab is neither free nor open source
- It requires a *license* which must be purchased
- The cheapest license are network ones, which means that *you won't be able to use Matlab without internet access.*

41 Matlab for Python users

42 Matlab
- Syntax
- Lists, Dictionaries, Arrays
- Linear Algebra
- Functions
- Environment

43 What we didn't do in this course

- Graphical Interface (GUIs)
- 3D plots
- Interface with C or Fortran code
- Python for controlling system commands
- ....