



**Sebastian von Alfthan
Jussi Enkovaara
Martti Louhivuori**



Python in High-performance Computing

January 25-27, 2016

PRACE Advanced Training Centre

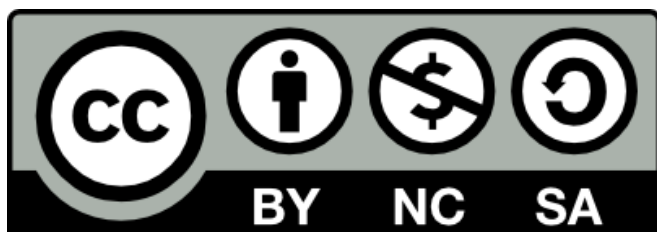
CSC – IT Center for Science Ltd, Finland

```
import sys, os
try:
    from Bio.PDB import PDBParser
    __biopython_installed__ = True
except ImportError:
    __biopython_installed__ = False

__default_bfactor__ = 0.0      # default B-factor
__default_occupancy__ = 1.0    # default occupancy level
__default_segid__ = ''        # empty segment ID

class EOF(Exception):
    def __init__(self): pass

class FileCrawler:
    """
    Crawl through a file reading back and forth without loading
    anything to memory.
    """
    def __init__(self, filename):
        try:
            self.__fp__ = open(filename)
        except IOError:
            raise ValueError, "Couldn't open file '%s' for reading." % filename
        self.tell = self.__fp__.tell
        self.seek = self.__fp__.seek
    def prevline(self):
        try:
            self.prev()
```



All material (C) 2016 by the authors.

This work is licensed under a **Creative Commons Attribution-NonCommercial-ShareAlike 3.0**
Unported License, <http://creativecommons.org/licenses/by-nc-sa/3.0/>

Agenda

Tuesday

9:00-9:45	Introduction to Python
9:45-10:30	Exercises
10:30-10:45	Coffee Break
10:45-11:15	Control structures
11:15-12:15	Exercises
12:15-13:00	Lunch break
13:00-13:30	Functions and modules
13:30-14:30	Exercises
14:30-14:45	Coffee Break
14:45-15:15	File I/O and text processing
15:15-16:15	Exercises

Wednesday

9.00-9.45	Object oriented programming with Python
9.45-10.30	Exercises
10.30-10.45	Coffee break
10:45-11:15	NumPy – fast array interface to Python
11:15-12:15	Exercises
12.15-13.00	Lunch break
13.00-13:30	NumPy (continued)
13:30-14:30	Exercises
14.30-14.45	Coffee break
14.45-15.15	NumPy (continued)
15:15-16:15	Exercises

Thursday

9:00-9:45	Visualization with Python
9:45-10:30	Exercises
10:30-10:45	Coffee Break
10:45-11:30	Scipy-package for scientific computing
11:30-12:15	Exercises
12:15-13:00	Lunch break
13:00-13:30	C extensions – integrating efficient C routines in Python
13:30-14:30	Exercises
14:30-14:45	Coffee break
14:45-15:45	MPI and Python – mpi4py
15:45-16:15	Exercises

INTRODUCTION TO PYTHON



What is Python?

- Modern, interpreted, object-oriented, full featured high level programming language
- Portable (Unix/Linux, Mac OS X, Windows)
- Open source, intellectual property rights held by the Python Software Foundation
- Python versions: 2.x and 3.x
 - 3.x is not backwards compatible with 2.x
 - This course uses 2.x version

Why Python?

- Fast program development
- Simple syntax
- Easy to write well readable code
- Large standard library
- Lots of third party libraries
 - Numpy, Scipy, Biopython
 - Matplotlib
 - ...

Information about Python

- www.python.org
- H. P. Langtangen, “Python Scripting for Computational Science”, Springer
- www.scipy.org
- matplotlib.sourceforge.net
- mpi4py.scipy.org

FIRST GLIMPSE INTO THE PYTHON

Python basics

- Syntax and code structure
- Data types and data structures
- Control structures
- Functions and modules
- Text processing and IO

Python program

- Typically, a .py ending is used for Python scripts, e.g. *hello.py*:

```
hello.py
print "Hello world!"
```

- Scripts can be executed by the *python* executable:

```
$ python hello.py
Hello world!
```

Interactive python interpreter

- The interactive interpreter can be started by executing *python* without arguments:

```
$ python
Python 2.4.3 (#1, Jul 16 2009, 06:20:46)
[GCC 4.1.2 20080704 (Red Hat 4.1.2-44)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> print "Hello"
Hello
>>>
```

- Useful for testing and learning

Python syntax

- Variable and function names start with a letter and can contain also numbers and underscores, e.g. “my_var”, “my_var2”
- Python is case sensitive
- Code blocks are defined by indentation
- Comments start by # sign

```
example.py
# example
if x > 0:
    x = x + 1 # increase x
    print("increasing x")
else:
    x = x - 1
    print "decreasing x"
print("x is processed")
```

Data types

- Python is dynamically typed language
 - no type declarations for variables
- Variable does have a type
 - incompatible types cannot be combined

```
example.py
print "Starting example"
x = 1.0
for i in range(10):
    x += 1
y = 4 * x
s = "Result"
z = s + y # Error
```

Numeric types

- Integers
- Floats
- Complex numbers
- Basic operations
 - + and -
 - *, / and **
 - implicit type conversions
 - be careful with integer division !

```
>>> x = 2
>>> x = 3.0
>>> x = 4.0 + 5.0j
>>>
>>> 2.0 + 5 - 3
4.0
>>> 4.0**2 / 2.0 * (1.0 - 3j)
(8-24j)
>>> 1/2
0
>>> 1./2
0.5
```

Strings

- Strings are enclosed by " or '
- Multiline strings can be defined with three double quotes

```
strings.py
s1 = "very simple string"
s2 = 'same simple string'
s3 = "this isn't so simple string"
s4 = 'is this "complex" string?'
s5 = """This is a long string
expanding to multiple lines,
so it is enclosed by three "s."""
```

Strings

- + and * operators with strings:

```
>>> "Strings can be " + "combined"
'Strings can be combined'
>>>
>>> "Repeat! " * 3
'Repeat! Repeat! Repeat!'
```

Data structures

- Lists and tuples
- Dictionaries

List

- Python lists are dynamic arrays
- List items are indexed (index starts from 0)
- List item can be any Python object, items can be of different type
- New items can be added to any place in the list
- Items can be removed from any place of the list

Lists

- Defining lists
- Accessing list elements
- Modifying list items

```
>>> my_list1 = [3, "egg", 6.2, 7]
>>> my_list2 = [12, [4, 5], 13, 1]
```

```
>>> my_list1[0]
3
>>> my_list2[1]
[4, 5]
>>> my_list1[-1]
7
```

```
>>> my_list1[-2] = 4
>>> my_list1
[3, 'egg', 4, 7]
```

Lists

- Adding items to list
- Accessing list elements
- + and * operators with lists

```
>>> my_list1 = [9, 8, 7, 6]
>>> my_list1.append(11)
>>> my_list1
[9, 8, 7, 6, 11]
>>> my_list1.insert(1,16)
>>> my_list1
[9, 16, 8, 7, 6, 11]
>>> my_list2 = [5, 4]
>>> my_list1.extend(my_list2)
>>> my_list1
[9, 16, 8, 7, 6, 11, 5, 4]
```

```
>>> [1, 2, 3] + [4, 5, 6]
[1, 2, 3, 4, 5, 6]
>>> [1, 2, 3] * 2
[1, 2, 3, 1, 2, 3]
```


Lists

- It is possible to access slices of lists

```
>>> my_list1 = [0, 1, 2, 3, 4, 5]
>>> my_list1[0:2]
[0, 1]
>>> my_list1[:2]
[0, 1]
>>> my_list1[3:]
[3, 4, 5]
>>> my_list1[0:6:2]
[0, 2, 4]
>>> my_list1[::-1]
[5, 4, 3, 2, 1, 0]
```

- Removing list items

```
>>> second = my_list1.pop(2)
>>> my_list1
[0, 1, 3, 4, 5]
>>> second
2
```

Tuples

- Tuples are immutable lists
- Tuples are indexed and sliced like lists, but cannot be modified

```
>>> t1 = (1, 2, 3)
>>> t1[1] = 4
Traceback (most recent call last):
  File "<stdin>", line 1, in
<module>
TypeError: 'tuple' object does not support item assignment
```

Dictionaries

- Dictionaries are associative arrays
- Unordered list of key - value pairs
- Values are indexed by keys
- Keys can be strings or numbers
- Value can be any Python object

Dictionaries

- Creating dictionaries

```
>>> grades = {'Alice': 5, 'John': 4, 'Carl': 2}
>>> grades
{'John': 4, 'Alice': 5, 'Carl': 2}
```

- Accessing values

```
>>> grades['John']
4
```

- Adding items

```
>>> grades['Linda'] = 3
>>> grades
{'John': 4, 'Alice': 5, 'Carl': 2, 'Linda': 3}
>>> elements = {}
>>> elements['Fe'] = 26
>>> elements
{'Fe': 26}
```

Variables

- Python variables are always references
- my_list1** and **my_list2** are references to the same list
 - Modifying **my_list2** changes also **my_list1**!
- Copy can be made by slicing the whole list

```
>>> my_list1 = [1,2,3,4]
>>> my_list2 = my_list1
```

```
>>> my_list2[0] = 0
>>> my_list1
[0, 2, 3, 4]
```

```
>>> my_list3 = my_list1[:]
>>> my_list3[-1] = 66
>>> my_list1
[0, 2, 3, 4]
>>> my_list3
[0, 2, 3, 66]
```

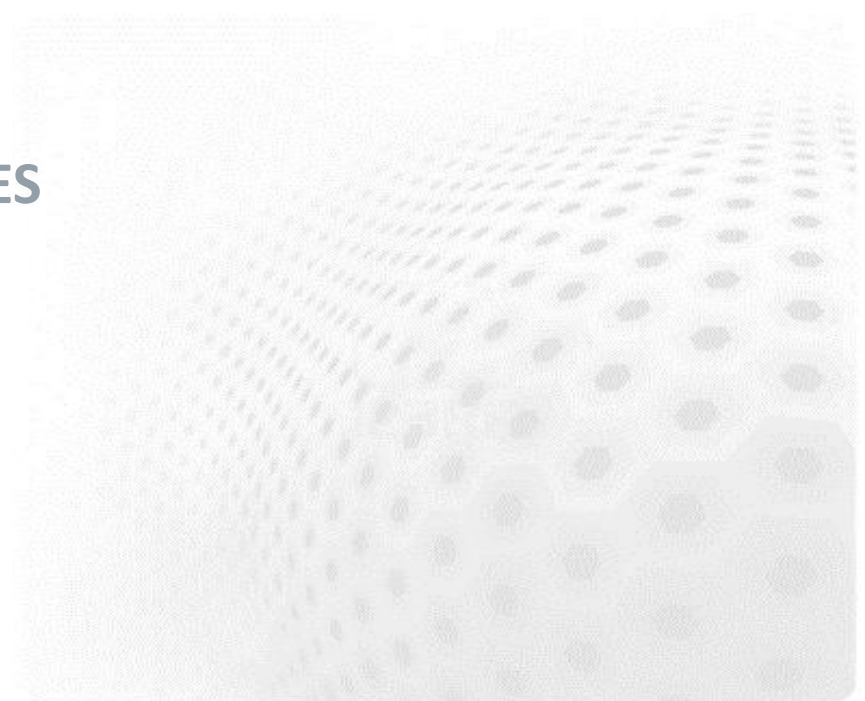
What is object?

- Object is a software bundle of data (=variables) and related methods
- Data can be accessed directly or only via the methods (=functions) of the object
- In Python, **everything** is object
- Methods of object are called with the syntax: **obj.method**
- Methods can modify the data of object or return new objects

Summary

- Python syntax: code blocks defined by indentation
- Numeric and string datatypes
- Powerful basic data structures:
 - Lists and dictionaries
- Everything is object in Python
- Python variables are always references to objects

CONTROL STRUCTURES



Control structures

- **if – else** statements
- **while** loops
- **for** loops
- Exceptions

if statement

- **if** statement allows one to execute code block depending on condition
- code blocks are defined by indentation, standard practice is to use four spaces for indentation

```
example.py
if x > 0:
    x += 1
    y = 4 * x
numbers[2] = x
```

- boolean operators:
==, !=, >, <, >=, <=

if statement

- there can be multiple branches of conditions

```
example.py
if x == 0:
    print "x is zero"
elif x < 0:
    print "x is negative"
elif x > 100000:
    print "x is large"
else:
    print "x is something completely different"
```

- Python does not have switch statement

while loop

- **while** loop executes a code block as long as an expression is True

```
example.py
x = 0
cubes = {}
cube = 0
while cube < 100:
    cubes[x] = cube
    x += 1
    cube = x**3
```

for loop

- **for** statement iterates over the items of any sequence (e.g. list)

```
example.py
cars = ['Audi', 'BMW', 'Jaguar', 'Lada']

for car in cars:
    print "Car is ", car
```

- In each pass, the loop variable **car** gets assigned next value from the sequence
 - Value of loop variable can be any Python object

for loop

- Many sequence-like Python objects support iteration
 - Dictionary: "next" values are dictionary keys

```
example.py
prices = {'Audi' : 50, 'BMW' : 70, 'Lada' : 5}

for car in prices:
    print "Car is ", car
    print "Price is ", prices[car]
```

- (later on: file as sequence of lines, "next" value of file object is the next line in the file)

for loop

- Items in the sequence can be lists themselves

```
example.py
coordinates = [[1.0, 0.0], [0.5, 0.5], [0.0, 1.0]]
for coord in coordinates:
    print "X=", coord[0], "Y=", coord[1]
```

- Values can be assigned to multiple loop variables

```
example.py
for x, y in coordinates:
    print "X=", x, "Y=", y
```

- Dictionary method **items()** returns list of key-value pairs

```
example.py
prices = {'Audi': 50, 'BMW' : 70, 'Lada' : 5}
for car, price in prices.items():
    print "Price of", car, "is", price
```

break & continue

- **break** out of the loop

```
example.py
x = 0
while True:
    x += 1
    cube = x**3
    if cube > 100:
        break
```

```
example.py
sum = 0
for p in prices:
    sum += p
    if sum > 100:
        print "too much"
        break
```

- **continue** with the next iteration of loop

```
example.py
x = -5
cube = 0
while cube < 100:
    x += 1
    if x < 0:
        continue
    cube = x**3
```

```
example.py
sum = 0
for p in prices:
    if p > 100:
        continue
    sum += p
```


exceptions

- Exceptions allow the program to handle errors and other “unusual” situations in a flexible and clean way
- Basic concepts:
 - Raising an exception. Exception can be raised by user code or by system
 - Handling an exception. Defines what to do when an exception is raised, typically in user code.
- There can be different exceptions and they can be handled by different code

exceptions in Python

- Exception is caught and handled by **try - except** statements

```
example.py
my_list = [3, 4, 5]
try:
    fourth = my_list[4]
except IndexError:
    print "There is no fourth element"
```

- User code can also **raise** an exception

```
example.py
if solver not in ['exact', 'jacobi', 'cg']:
    raise RuntimeError('Unsupported solver')
```

List comprehension

- useful Python idiom for creating lists from existing ones without explicit **for** loops
- creates a new list by performing operations for the elements of list:

newlist = [op(x) for x in oldlist]

```
>>> numbers = range(6)
>>> squares = [x**2 for x in numbers]
>>> squares
[0, 1, 4, 9, 16, 25]
```

- a conditional statement can be included

```
>>> odd_squares = [x**2 for x in numbers if x % 2 == 1]
>>> odd_squares
[1, 9, 25]
```

FUNCTIONS AND MODULES



Functions and modules

- defining functions
- calling functions
- importing modules

Functions

- function is block of code that can be referenced from other parts of the program
- functions have arguments
- functions can return values

Function definition

```
function.py
def add(x, y):
    result = x + y
    return result

u = 3.0
v = 5.0
sum = add(u, v)
```

- name of function is **add**
- **x** and **y** are arguments
- there can be any number of arguments and arguments can be any Python objects
- return value can be any Python object

Keyword arguments

- functions can also be called using keyword arguments

```
function.py
def sub(x, y):
    result = x - y
    return result

res1 = sub(3.0, 2.0)
res2 = sub(y=3.0, x=2.0)
```

- keyword arguments can improve readability of code

Default arguments

- it is possible to have default values for arguments
- function can then be called with varying number of arguments

```
function.py
def add(x, y=1.0):
    result = x + y
    return result

sum1 = add(0.0, 2.0)
sum2 = add(3.0)
```

Modifying function arguments

- as Python variables are always references, function can modify the objects that arguments refer to

```
>>> def switch(mylist):
...     tmp = mylist[-1]
...     mylist[-1] = mylist[0]
...     mylist[0] = tmp
...
>>> l1 = [1,2,3,4,5]
>>> switch(l1)
>>> l1
[5, 2, 3, 4, 1]
```

- side effects can be wanted or unwanted

Modules

- modules are extensions that can be imported to Python to provide additional functionality, e.g.
 - new data structures and data types
 - functions
- Python standard library includes several modules
- several third party modules
- user defined modules

Importing modules

- **import** statement

```
example.py
import math
x = math.exp(3.5)

import math as m
x = m.exp(3.5)

from math import exp, pi
x = exp(3.5) + pi

from math import *
x = exp(3.5) + sqrt(pi)

exp = 6.6
from math import *
x = exp + 3.2 # Won't work,
              # exp is now a function
```

Creating modules

- it is possible to make imports from own modules
- define a function in file `mymodule.py`

```
mymodule.py
def incx(x):
    return x+1
```

- the function can now be imported in other .py files:

```
test.py
import mymodule

y = mymodule.incx(1)
```

```
test.py
from mymodule import incx

y = incx(1)
```

Summary

- functions help in reusing frequently used code blocks
- functions can have default and keyword arguments
- additional functionality can be imported from modules

FILE I/O AND TEXT PROCESSING



File I/O and text processing

- working with files
- reading and processing file contents
- string formatting and writing to files

Opening and closing files

- opening a file:
`myfile = open(filename, mode)`
 - returns a handle to the file

```
>>> fp = open('example.txt', 'r')
>>>
```

Opening and closing files

- file can be opened for
 - reading: `mode='r'`
(file has to exist)
 - writing: `mode='w'`
(existing file is truncated)
 - appending: `mode='a'`
- closing a file
 - `myfile.close()`

```
example.py
# open file for reading
infile = open('input.dat', 'r')

# open file for writing
outfile = open('output.dat', 'w')

# open file for appending
appfile = open('output.dat', 'a')

# close files
infile.close()
```

Reading from files

- a single line can be read from a file with the `readline()` - function

```
>>> infile = open('inp', 'r')
>>> line = infile.readline()
```

- it is often convenient to iterate over all the lines in a file

```
>>> infile = open('inp', 'r')
>>> for line in infile:
...     # process lines
```

Processing lines

- generally, a line read from a file is just a string
- a string can be split into a list of strings:

```
>>> infile = open('inp', 'r')
>>> for line in infile:
...     line = line.split()
```

- fields in a line can be assigned to variables and added to e.g. lists or dictionaries

```
>>> for line in infile:
...     line = line.split()
...     x, y = float(line[1]), float(line[3])
...     coords.append((x,y))
```

Processing lines

- sometimes one wants to process only files containing specific tags or substrings

```
>>> for line in infile:
...     if "Force" in line:
...         line = line.split()
...         x, y, z = float(line[1]), float(line[2]), float(line[3])
...         forces.append((x,y,z))
```

- other way to check for substrings:
 - `str.startswith()`, `str.endswith()`
- Python has also an extensive support for regular expressions in `re` -module

String formatting

- Output is often wanted in certain format
- The string object has `.format` method for placing variables within string
- Replacement fields surrounded by `{}` within the string

```
>>> x, y = 1.6666, 2.3333
print "X is {0} and Y is {1}".format(x, y)
X is 1.6666 and Y is 2.3333
>>> print "Y is {1} and X is {0}".format(x, y)
Y is 2.3333 and X is 1.6666
```

- Possible to use also keywords:

```
>>> print "Y is {val_y} and X is {val_x}".format(val_x=x, val_y=y)
Y is 2.3333 and X is 1.6666
```

String formatting

- Presentation of field can be specified with `{i:[w][.p][t]}`
 - `w` is optional minimum width
 - `.p` gives optional precision (=number of decimals)
 - `t` is the presentation type
- some presentation types
 - `s` string (normally omitted)
 - `d` integer decimal
 - `f` floating point decimal
 - `e` floating point exponential

```
>>> print "X is {0:6.3f} and Y is {1:6.2f}".format(x, y)
X is  1.667 and Y is   2.33
```

Writing to a file

- data can be written to a file with **print** statements
- file objects have also a **write()** function
- the **write()** does not automatically add a newline

```
output.py
outfile = open('out', 'w')
print >> outfile, "Header"
print >> outfile, "{0:6.3f} {0:6.3f}".format(x, y)

outfile = open('out', 'w')
outfile.write("Header\n")
outfile.write("{0:6.3f} {0:6.3f}".format(x, y))
```

- file should be closed after writing is finished

Differences between Python 2.X and 3.X

- **print** is a function in 3.X

```
differences.py
print "The answer is", 2*2 # 2.X
print("The answer is", 2*2) # 3.X

print >>sys.stderr, "fatal error" # 2.X
print("fatal error", file=sys.stderr) # 3.X
```

- in 3.X some dictionary methods return “views” instead of lists.
 - e.g. `k = d.keys()`; `k.sort()` does not work, use `k = sorted(d)` instead
- for more details, see <http://docs.python.org/release/3.1/whatsnew/3.0.html>

Summary

- files are opened and closed with **open()** and **close()**
- lines can be read by iterating over the file object
- lines can be split into lists and check for existence of specific substrings
- string formatting operators can be used for obtaining specific output
- file output can be done with **print** or **write()**

Useful modules in Python standard library

- **math** : “non-basic” mathematical operations
 - **os** : operating system services
 - **glob** : Unix-style pathname expansion
 - **random** : generate pseudorandom numbers
 - **pickle** : dump/load Python objects to/from file
 - **time** : timing information and conversions
 - **xml.dom / xml.sax** : XML parsing
 - + many more
- <http://docs.python.org/library/>

Summary

- Python is dynamic programming language
- flexible basic data structures
- standard control structures
- modular programs with functions and modules
- simple and powerful test processing and file I/O
- rich standard library

OBJECT ORIENTED PROGRAMMING WITH PYTHON



Object oriented programming with Python

- Basic concepts
- Classes in Python
- Inheritance
- Special methods

OOP concepts

- OOP is programming paradigm
 - data and functionality are wrapped inside of an “object”
 - Objects provide methods which operate on (the data of) the object
- Encapsulation
 - User accesses objects only through methods
 - Organization of data inside the object is hidden from the user

Examples

- String as an object
 - Data is the contents of string
 - Methods could be lower/uppercasing the string
- Two dimensional vector
 - Data is the x and y components
 - Method could be the norm of vector

OOP in Python

- In Python everything is a object
 - Example: **open** function returns a file object
 - data includes e.g. the name of the file
- ```
>>> f = open('foo', 'w')
>>> f.name
'foo'
```
- methods of the file object referred by **f** are **f.read()**, **f.readlines()**, **f.close()**, ...
- Also lists and dictionaries are objects (with some special syntax)

## OOP concepts

- class
  - defines the object, i.e. the data and the methods belonging to the object
  - there is only single definition for given object type
- instance
  - there can be several instances of the object
  - each instance can have different data, but the methods are the same

## Class definition in Python

- When defining class methods in Python the first argument to method is always **self**
- **self** refers to the particular instance of the class
- **self** is not included when calling the class method
- Data of the particular instance is handled with **self**

```
students.py
class Student:
 def set_name(self, name):
 self.name = name

 def say_hello(self):
 print "Hello, my name is ", self.name
```

## Class definition in Python

```
students.py
class Student:
 def set_name(self, name):
 self.name = name
 def say_hello(self):
 print "Hello, my name is ", self.name

creating an instance of student
stu = Student()
calling a method of class
stu.set_name('Jussi')
creating another instance of student
stu2 = Student()
stu2.set_name('Matti')
the two instances contain different data
stu.say_hello()
stu2.say_hello()
```

## Passing data to object

- Data can be passed to an object at the point of creation by defining a special method **\_\_init\_\_**
- **\_\_init\_\_** is always called when creating the instance

```
students.py
class Student:
 def __init__(self, name):
 self.name = name
...
```

- In Python, one can also refer directly to data attributes

```
>>> from students import Student
>>> stu1 = Student('Jussi')
>>> stu2 = Student('Matti')
>>> print stu1.name, stu2.name
'Jussi', 'Matti'
```

## Python classes as data containers

- classes can be used for C-struct or Fortran-Type like data structures

```
students.py
class Student:
 def __init__(self, name, age):
 self.name = name
 self.age = age
```

- instances can be used as items in e.g. lists

```
>>> stu1 = Student('Jussi', 27)
>>> stu2 = Student('Martti', 25)
>>> student_list = [stu1, stu2]
>>> print student_list[1].age
```

## Encapsulation in Python

- Generally, OOP favours separation of internal data structures and implementation from the interface
- In some programming languages attributes and methods can be defined to be accessible only from other methods of the object.
- In Python, everything is public. Leading underscore in a method name can be used to suggest “privacy” for the user

## Inheritance

- New classes can be derived from existing ones by inheritance
- The derived class “inherits” the attributes and methods of parent
- The derived class can define new methods
- The derived class can override existing methods

## Inheriting classes in Python

```
inherit.py
class Student:
 ...

class PhDStudent(Student):
 # override __init__ but use __init__ of base class!
 def __init__(self, name, age, thesis_project):
 self.thesis = thesis_project
 Student.__init__(self, name, age)

 # define a new method
 def get_thesis_project(self):
 return self.thesis

stu = PhDStudent('Pekka', 20, 'Theory of everything')
use a method from the base class
stu.say_hello()
use a new method
proj = stu.get_thesis_project()
```

## Special methods

- class can define methods with special names to implement operations by special syntax (operator overloading)
- Examples
  - `__add__`, `__sub__`, `__mul__`, `__div__`
  - for arithmetic operations (+, -, \*, /)
  - `__cmp__` for comparisons, e.g. sorting
  - `__setitem__`, `__getitem__` for list/dictionary like syntax using []

## Special methods

```
special.py
class Vector:
 def __init__(self, x, y):
 self.x = x
 self.y = y

 def __add__(self, other):
 new_x = self.x + other.x
 new_y = self.y + other.y
 return Vector(new_x, new_y)

v1 = Vector(2, 4)
v2 = Vector(-3, 6)
v3 = v1 + v2
```

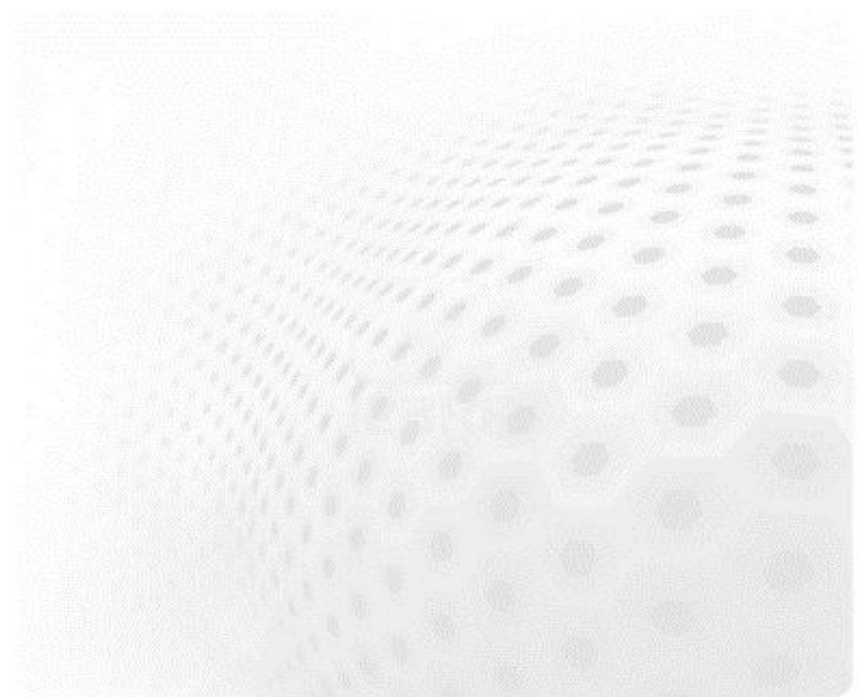
```
special.py
class Student:
 ...
 def __cmp__(self, other):
 return cmp(self.name, other.name)

students = [Student('Jussi', 27),
 Student('Aaron', 29)]
students.sort()
```

## Summary

- Objects contain both data and functionality
- class is the definition of the object
- instance is a particular realization of object
- class can be inherited from other class
- Python provides a comprehensive support for object oriented programming (“Everything is an object”)

**NUMPY**







## Numpy – fast array interface

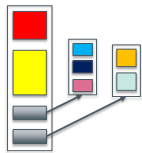
- Standard Python is not well suitable for numerical computations
  - lists are very flexible but also slow to process in numerical computations
- Numpy adds a new **array** data type
  - static, multidimensional
  - fast processing of arrays
  - some linear algebra, random numbers

## Numpy arrays

- All elements of an array have the same type
- Array can have multiple dimensions
- The number of elements in the array is fixed, shape can be changed

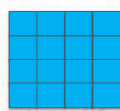
## Python list vs. NumPy array

Python list



Memory layout

NumPy array



Memory layout

## Creating numpy arrays

- From a list:

```
>>> import numpy as np
>>> a = np.array((1, 2, 3, 4), float)
>>> a
array([1., 2., 3., 4.])
>>> list1 = [[1, 2, 3], [4,5,6]]
>>> mat = np.array(list1, complex)
>>> mat
array([[1.+0.j, 2.+0.j, 3.+0.j],
 [4.+0.j, 5.+0.j, 6.+0.j]])
>>> mat.shape
(2, 3)
>>> mat.size
6
```

## Creating numpy arrays

- More ways for creating arrays:

```
>>> import numpy as np
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> b = np.linspace(-4.5, 4.5, 5)
>>> b
array([-4.5, -2.25, 0. , 2.25, 4.5])
>>> c = np.zeros((4, 6), float)
>>> c.shape
(4, 6)
>>> d = np.ones((2, 4))
>>> d
array([[1., 1., 1., 1.],
 [1., 1., 1., 1.]])
```

## Indexing and slicing arrays

- Simple indexing:

```
>>> mat = np.array([[1, 2, 3], [4, 5, 6]])
>>> mat[0,2]
3
>>> mat[1,-2]
5
```

- Slicing:

```
>>> a = np.arange(5)
>>> a[2:]
array([2, 3, 4])
>>> a[:-1]
array([0, 1, 2, 3])
>>> a[1:3] = -1
>>> a
array([0, -1, -1, 3, 4])
```

## Indexing and slicing arrays

- Slicing is possible over all dimensions:

```
>>> a = np.arange(10)
>>> a[1:7:2]
array([1, 3, 5])
>>> a = np.zeros((4, 4))
>>> a[1:3, 1:3] = 2.0
>>> a
array([[0., 0., 0., 0.],
 [0., 2., 2., 0.],
 [0., 2., 2., 0.],
 [0., 0., 0., 0.]])
```

## Views and copies of arrays

- Simple assignment creates references to arrays
- Slicing creates “views” to the arrays
- Use **copy()** for real copying of arrays

```
example.py
a = np.arange(10)
b = a # reference, changing values in b changes a
b = a.copy() # true copy

c = a[1:4] # view, changing c changes elements [1:4] of a
c = a[1:4].copy() # true copy of subarray
```

## Array manipulation

- **reshape** : change the shape of array

```
>>> mat = np.array([[1, 2, 3], [4, 5, 6]])
>>> mat
array([[1, 2, 3],
 [4, 5, 6]])
>>> mat.reshape(3,2)
array([[1, 2],
 [3, 4],
 [5, 6]])
```

- **ravel** : flatten array to 1-d

```
>>> mat.ravel()
array([1, 2, 3, 4, 5, 6])
```

## Array manipulation

- **concatenate** : join arrays together

```
>>> mat1 = np.array([[1, 2, 3], [4, 5, 6]])
>>> mat2 = np.array([[7, 8, 9], [10, 11, 12]])
>>> np.concatenate((mat1, mat2))
array([[1, 2, 3],
 [4, 5, 6],
 [7, 8, 9],
 [10, 11, 12]])
>>> np.concatenate((mat1, mat2), axis=1)
array([[1, 2, 3, 7, 8, 9],
 [4, 5, 6, 10, 11, 12]])
```

- **split** : split array to N pieces

```
>>> np.split(mat1, 3, axis=1)
[array([[1],
 [4]]), array([[2],
 [5]]), array([[3],
 [6]])]
```

## Array operations

- Most operations for numpy arrays are done element-wise

— +, -, \*, /, \*\*

```
>>> a = np.array([1.0, 2.0, 3.0])
>>> b = 2.0
>>> a * b
array([2., 4., 6.])
>>> a + b
array([3., 4., 5.])
>>> a * a
array([1., 4., 9.])
```

## Array operations

- Numpy has special functions which can work with array arguments

— sin, cos, exp, sqrt, log, ...

```
>>> import numpy, math
>>> a = numpy.linspace(-math.pi, math.pi, 8)
>>> a
array([-3.14159265, -2.24399475, -1.34639685, -0.44879895,
 0.44879895, 1.34639685, 2.24399475, 3.14159265])
>>> numpy.sin(a)
array([-1.22464680e-16, -7.81831482e-01, -9.74927912e-01,
 -4.33883739e-01, 4.33883739e-01, 9.74927912e-01,
 7.81831482e-01, 1.22464680e-16])
>>> math.sin(a)
Traceback (most recent call last):
 File "<stdin>", line 1, in ?
TypeError: only length-1 arrays can be converted to Python scalars
```

## Vectorized operations

- for loops in Python are slow
- Use “vectorized” operations when possible
- Example: difference

```
example.py
brute force using a for loop
arr = np.arange(1000)
dif = np.zeros(999, int)
for i in range(1, len(arr)):
 dif[i-1] = arr[i] - arr[i-1]
```

```
vectorized operation
arr = np.arange(1000)
dif = arr[1:] - arr[:-1]
```

— for loop is ~80 times slower!

## Broadcasting

- If array shapes are different, the smaller array may be **broadcasted** into a larger shape

```
>>> from numpy import array
>>> a = array([[1,2],[3,4],[5,6]], float)
>>> a
array([[1., 2.],
 [3., 4.],
 [5., 6.]])
>>> b = array([[7,11]], float)
>>> b
array([[7., 11.]])
>>> a * b
array([[7., 22.],
 [21., 44.],
 [35., 66.]])
```

## Advanced indexing

- Numpy arrays can be indexed also with other arrays (integer or boolean)

```
>>> x = np.arange(10,1,-1)
>>> x
array([10, 9, 8, 7, 6, 5, 4, 3, 2])
>>> x[np.array([3, 3, 1, 8])]
array([7, 7, 9, 2])
```

- Boolean “mask” arrays

```
>>> m = x > 7
>>> m
array([True, True, True, False, False, ...,
 ...])
>>> x[m]
array([10, 9, 8])
```

- Advanced indexing creates copies of arrays

## Masked arrays

- Sometimes datasets contain invalid data (faulty measurement, problem in simulation)
- Masked arrays provide a way to perform array operations neglecting invalid data
- Masked array support is provided by numpy.ma module

## Masked arrays

- Masked arrays can be created by combining a regular numpy array and a boolean mask

```
>>> import numpy.ma as ma
>>> x = np.array([1, 2, 3, -1, 5])
>>>
>>> m = x < 0
>>> mx = ma.masked_array(x, mask=m)
>>> mx
masked_array(data = [1 2 3 -- 5],
 mask = [False False False True False],
 fill_value = 999999)
>>> x.mean()
2.0
>>> mx.mean()
2.75
```

## I/O with Numpy

- Numpy provides functions for reading data from file and for writing data into the files
- Simple text files
  - numpy.loadtxt
  - numpy.savetxt
  - Data in regular column layout
  - Can deal with comments and different column delimiters

## Random numbers

- The module numpy.random provides several functions for constructing random arrays
  - random: uniform random numbers
  - normal: normal distribution
  - poisson: Poisson distribution
  - ...

```
>>> import numpy.random as rnd
>>> rnd.random((2,2))
array([[0.02909142, 0.90848],
 [0.9471314 , 0.31424393]])
>>> rnd.poisson(size=(2,2))
```

## Polynomials

- Polynomial is defined by array of coefficients p
- $p(x, N) = p[0] x^{N-1} + p[1] x^{N-2} + \dots + p[N-1]$
- Least square fitting: **numpy.polyfit**
- Evaluating polynomials: **numpy.polyval**
- Roots of polynomial: **numpy.roots**
- ...

```
>>> x = np.linspace(-4, 4, 7)
>>> y = x**2 + rnd.random(x.shape)
>>>
>>> p = np.polyfit(x, y, 2)
>>> p
array([0.96869003, -0.01157275, 0.69352514])
```

## Linear algebra

- Numpy can calculate matrix and vector products efficiently: **dot**, **vdot**, ...
- Eigenproblems: **linalg.eig**, **linalg.eigvals**, ...
- Linear systems and matrix inversion: **linalg.solve**, **linalg.inv**

```
>>> A = np.array(((2, 1), (1, 3)))
>>> B = np.array((-2, 4.2), (4.2, 6)))
>>> C = np.dot(A, B)
>>>
>>> b = np.array((1, 2))
>>> np.linalg.solve(C, b) # solve C x = b
array([0.04453441, 0.06882591])
```

## Numpy performance

- Matrix multiplication  
 $C = A * B$   
matrix dimension 200
- pure python: 5.30 s
- naive C: 0.09 s
- numpy.dot: 0.01 s

## Summary

- Numpy provides a static array data structure
- Multidimensional arrays
- Fast mathematical operations for arrays
- Arrays can be broadcasted into same shapes
- Tools for linear algebra and random numbers



# VISUALIZATION WITH PYTHON







## Matplotlib

- 2D plotting library for python
- Can be used in scripts and in interactive shell
- Publication quality in various hardcopy formats
- “Easy things easy, hard things possible”
- Some 3D functionality

## Matplotlib interfaces

- Simple command style functions similar to Matlab

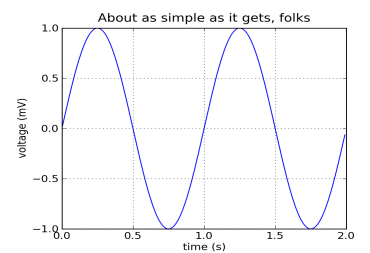
```
plot.py
import pylab as pl
...
pl.plot(x, y)
```

- Powerful object oriented API for full control of plotting

## Basic concepts

- Figure: the main container of a plot
- Axes: the “plotting” area, a figure can contain multiple Axes
- graphical objects: lines, rectangles, text
- Command style functions are used for creating and manipulating figures, axes, lines, ...
- The command style interface is stateful:
  - track is kept about current figure and plotting area

## Simple plot

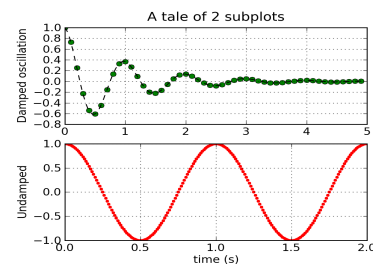


- **plot** : create a simple plot. Figure and axes are created if needed

## Interactive vs. batch mode

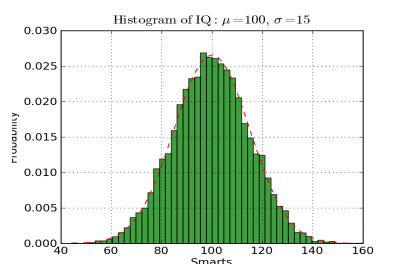
- In many installations batch mode is default
  - Figures do not show up without **show()** function
  - Batch mode is useful e.g. for writing out files during simulation and for heavy rendering
- Mode can be controlled as:
  - **ion()** : turn on interactive mode
  - **ioff()** : turn on interactive mode

## Multiple subplots



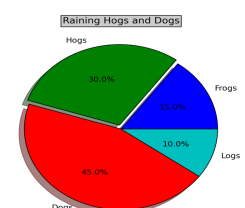
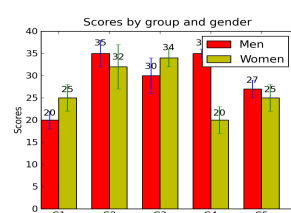
- **subplot** : create multiple axes in the figure and switch between subplots

## Histograms



- **hist** : create histogram
- Latex can be used with matplotlib

## Bar and pie charts



- **bar** : bar charts
- **pie** : pie charts

## Summary of basic functions

- Simple plot: **plot**
- Interactive vs. batch mode: **ion** / **ioff**
- Hardcopies: **savefig**
- Multiple plots: **subplot**
- Histograms: **hist**
- Bar charts: **bar**
- Pie charts: **pie**
- Switch plotting on top of existing figure: **hold**
- Contour plots: **contour**, **contourf**

## Summary

- Matplotlib provides a simple command style interface for creating publication quality figures
- Interactive plotting and different output formats (.png, .pdf, .eps)
- Simple plots, multiplot figures, decorations
- Possible to use Latex in text

## Mayavi

- General purpose, cross-platform tool for 3-D scientific data visualization
- Visualization of scalar, vector and tensor data in 2 and 3 dimensions
- Easy scriptability using Python
- Convenient functionality for rapid scientific plotting via mlab

## Simple example

- Surface described by three 2D arrays

```
>>> from mayavi import mlab
>>> from numpy import pi, sin, cos, mgrid
>>> dphi, dtheta = pi/250.0, pi/250.0
>>> [phi, theta] = mgrid[0:pi+dphi*1.5:dphi, 0:2*pi+dtheta*1.5:dtheta]
>>> m0 = 4; m1 = 3; m2 = 2; m3 = 3; m4 = 6; m5 = 2; m6 = 6; m7 = 4;
>>> r = sin(m0*phi)**m1 + cos(m2*phi)**m3 + sin(m4*theta)**m5 +
>>> cos(m6*theta)**m7
>>> x = r*sin(phi)*cos(theta)
>>> y = r*cos(phi)
>>> z = r*sin(phi)*sin(theta)
>>> mlab.mesh(x,y,z)
```

## Simple example 2

- Iso-surfaces for a 3D volume

```
>>> from mayavi import mlab
>>> import numpy as np
>>> x, y, z = np.ogrid[-5:5:64j, -5:5:64j, -5:5:64j]
>>> scalars = x * x * 0.5 + y * y + z * z * 2.0
>>> mlab.contour3d(scalars, contours=4, transparent=True)
```

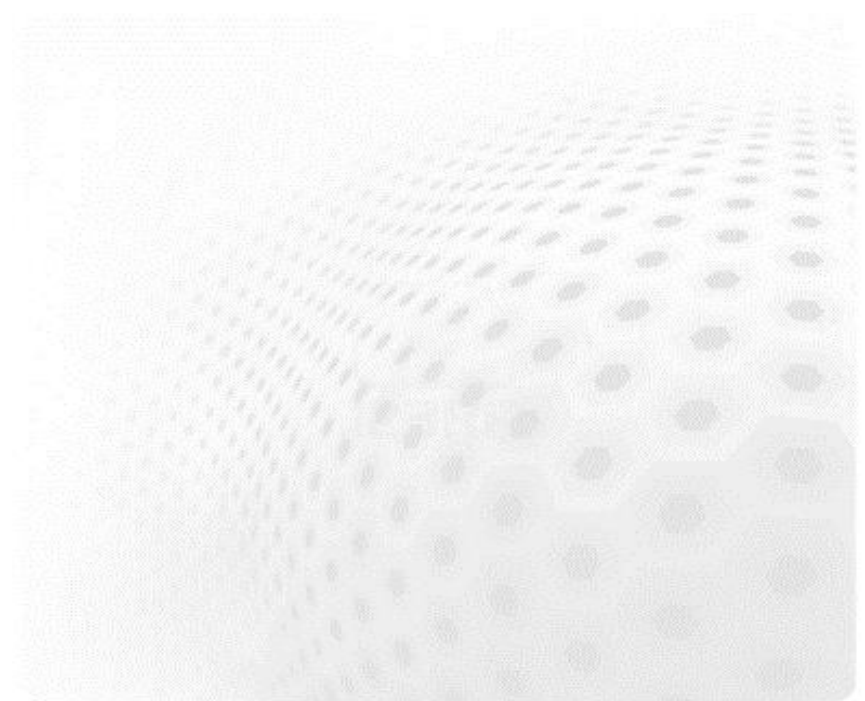
## Additional basic functions

- **imshow** : view a 2D array as an image
- **surf** : view a 2D array as a carpet plot
- **quiver3d** : plot arrows to represent vectors at data points
- **savefig** : write out a hardcopy

## Summary

- Mayavi is easy-to-use tool for 3D visualization
- Surfaces, iso-surfaces, vector fields
- Hardcopies in various formats
- Vast set of more advanced features

SCIPY





## Scipy – Scientific tools for Python

- Scipy is a Python package containing several tools for scientific computing
- Modules for:
  - statistics, optimization, integration, interpolation
  - linear algebra, Fourier transforms, signal and image processing
  - ODE solvers, special functions
  - ...
- Vast package, reference guide is currently 975 pages
- Scipy is built on top of Numpy

## Library overview

- Clustering package (scipy.cluster)
- Constants (scipy.constants)
- Fourier transforms (scipy.fftpack)
- Integration and ODEs (scipy.integrate)
- Interpolation (scipy.interpolate)
- Input and output (scipy.io)
- Linear algebra (scipy.linalg)
- Maximum entropy models (scipy.maxentropy)
- Miscellaneous routines (scipy.misc)
- Multi-dimensional image processing (scipy.ndimage)
- Orthogonal distance regression (scipy.odr)
- Optimization and root finding (scipy.optimize)
- Signal processing (scipy.signal)
- Sparse matrices (scipy.sparse)
- Sparse linear algebra (scipy.sparse.linalg)
- Spatial algorithms and data structures (scipy.spatial)
- Special functions (scipy.special)
- Statistical functions (scipy.stats)
- Image Array Manipulation and Convolution (scipy.stsci)
- C/C++ integration (scipy.weave)

## Integration

- Routines for numerical integration
  - single, double and triple integrals
- Function to integrate can be given by function object or by fixed samples

```
integrate.py
from scipy.integrate import simps, quad, inf
x = np.linspace(0, 1, 20)
y = np.exp(-x)
int1 = simps(y, x) # integrate function given by samples

def f(x):
 return exp(-x)

int2 = quad(f, 0, 1) # integrate function object
int3 = quad(f, 0, inf) # integrate up to infinity
```

## Optimization

- Several classical optimization algorithms
  - Quasi-Newton type optimizations
  - Least squares fitting
  - Simulated annealing
  - General purpose root finding
  - ...

```
>>> from scipy.optimize import fmin
>>>
```

## Special functions

- Scipy contains huge set of special functions
  - Bessel functions
  - Legendre functions
  - Gamma functions
  - ...

```
>>> from scipy.special import jv, gamma
>>>
```

## Linear algebra

- Wider set of linear algebra operations than in Numpy
  - decompositions, matrix exponentials
- Routines also for sparse matrices
  - storage formats
  - iterative algorithms

```
sparse.py
import numpy as np
from scipy.sparse.linalg import LinearOperator, cg

"Sparse" matrix-vector product
def mv(v):
 return np.array([2*v[0], 3*v[1]])

A = LinearOperator((2,2), matvec=mv, dtype=float)
b = np.array((4.0, 1.0))
x = cg(A, b) # Solve linear equation Ax = b with conjugate gradient
```

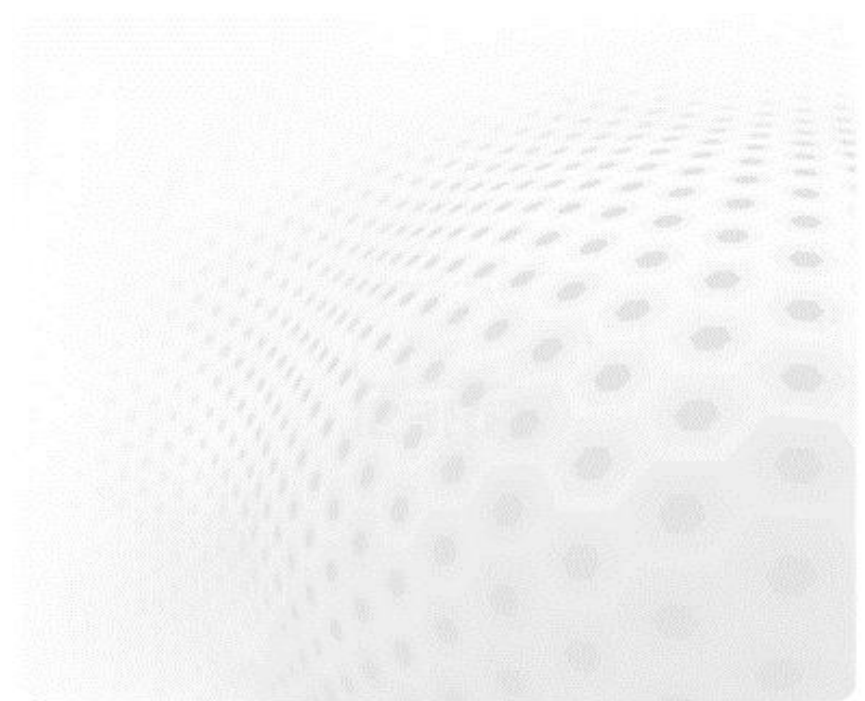
## Summary

- Scipy is vast package of tools for scientific computing
- Uses lots of NumPy in the background
- Numerical integration, optimization, special functions, linear algebra, ...
- Look Scipy documentation for finding tools for your needs!





## C - EXTENSIONS





## C - extensions

- Some times there are time critical parts of code which would benefit from compiled language
- 90/10 rule: **90 % of time is spent in 10 % of code**
  - only a small part of application benefits from compiled code
- It is relatively straightforward to create a Python interface to C-functions
  - data is passed from Python, routine is executed without any Python overheads

## C - extensions

- C routines are build into a shared library
- Routines are loaded dynamically with normal import statements

```
>>> import hello
>>> hello.world()
```

- A library **hello.so** is looked for
- A function **world** (defined in hello.so) is called

## Creating C-extension

### 1) Include Python headers

```
hello.c
#include <Python.h>
```

### 2) Define the C-function

```
hello.c
...
PyObject* world_c(PyObject *self, PyObject *args)
{
 printf("Hello world!\n");
 Py_RETURN_NONE;
}
```

- Type of function is always PyObject
- Function arguments are always the same (args is used for passing data from Python to C)
- A macro Py\_RETURN\_NONE is used for returning "nothing"

## Creating C-extension

### 3) Define the Python interfaces for functions

```
hello.c
...
static PyMethodDef functions[] = {
 {"world", world_c, METH_VARARGS, 0},
 {"honey", honey_c, METH_VARARGS, 0},
 {0, 0, 0, 0} /* "Sentinel" notifies the end of definitions */
};
```

- world** is the function name used in Python code, **world\_c** is the actual C-function to be called
- Single extension module can contain several functions (world, honey, ...)

## Creating C-extension

### 4) Define the module initialization function

```
hello.c
...
PyMODINIT_FUNC inithello(void)
{
 (void) Py_InitModule("hello", functions);
}
```

- Extension module should be build into **hello.so**
- Extension is module is imported as **import hello**
- Functions/interfaces defined in functions are called as hello.world(), hello.honey(), ...

## Creating C-extension

### 5) Compile as shared library

```
$ gcc -shared -o hello.so -I/usr/include/python2.6 -fPIC hello.c
$
$ python
Python 2.4.3 (#1, Jul 16 2009, 06:20:46)
[GCC 4.1.2 20080704 (Red Hat 4.1.2-44)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import hello
>>> hello.world()
Hello world!
```

- The location of Python headers (/usr/include/...) may vary in different systems
- Use *exercises/include\_paths.py* to find out yours!

## Full listing of hello.c

```
hello.c
#include <Python.h>

PyObject* world_c(PyObject *self, PyObject *args)
{
 printf("Hello world!\n");
 Py_RETURN_NONE;
}

static PyMethodDef functions[] = {
 {"world", world_c, METH_VARARGS, 0},
 {0, 0, 0, 0}
};

PyMODINIT_FUNC inithello(void)
{
 (void) Py_InitModule("hello", functions);
}
```

## Passing arguments to C-functions

```
hello2.c
...
PyObject* pass_c(PyObject *self, PyObject *args)
{
 int a;
 double b;
 char* str;
 if (!PyArg_ParseTuple(args, "ids", &a, &b, &str))
 return NULL;
 printf("int %i, double %f, string %s\n", a, b, str);
 Py_RETURN_NONE;
}
```

- PyArg\_ParseTuple** checks that function is called with proper arguments
  - "ids" : integer, double, string
- and does the conversion from Python to C types

## Returning values

```
hello2.c
...
PyObject* square_c(PyObject *self, PyObject *args)
{
 int a;
 if (!PyArg_ParseTuple(args, "i", &a))
 return NULL;
 a = a*a;
 return Py_BuildValue("i", a);
}
```

- Create and return Python integer from C variable. A "d" would create Python double etc.
- Returning tuple:  
**Py\_BuildValue("(ids)", a, b, str);**

## Operating with NumPy array

```
hello3.c
#include <Python.h>
#include <numpy/arrayobject.h>

PyObject* array(PyObject *self, PyObject *args)
{
 PyArrayObject* a;
 if (!PyArg_ParseTuple(args, "O", &a))
 return NULL;

 int size = PyArray_SIZE(a); /* Total size of array */
 double *data = PyArray_DATA(a); /* Pointer to data */
 for (int i=0; i < size; i++) {
 data[i] = data[i] * data[i];
 }
 Py_RETURN_NONE;
}
```

- NumPy provides API also for determining the dimensions of an array etc.

## Operating with NumPy array

- Function **import\_array()** should be called in the module initialization function when using NumPy C-API

```
hello3.c
...
PyMODINIT_FUNC inithello(void)
{
 import_array();
 (void) Py_InitModule("hello", functions);
}
```

## Tools for easier interfacing

- Cython
- SWIG
- pyrex
- f2py (for Fortran code)

## Summary

- Python can be extended with C-functions relatively easily
- C-extension build as shared library
- It is possible to pass data between Python and C code
- Extending Python:  
<http://docs.python.org/extending/>
- NumPy C-API  
<http://docs.scipy.org/doc/numpy/reference/c-api.html>

# PARALLEL PROGRAMMING WITH PYTHON USING MPI4PY







## Outline

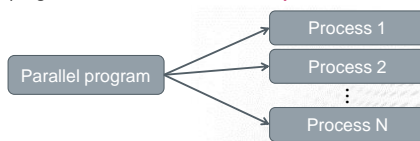
- Brief introduction to message passing interface (MPI)
- Python interface to MPI – mpi4py
- Performance considerations

## Message passing interface

- MPI is an application programming interface (API) for communication between separate processes
- The most widely used approach for distributed parallel computing
- MPI programs are portable and scalable
  - the same program can run on different types of computers, from PC's to supercomputers
- MPI is flexible and comprehensive
  - large (over 120 procedures)
  - concise (often only 6 procedures are needed)
- MPI standard defines C and Fortran interfaces
- **mpi4py** provides (an unofficial) Python interface

## Execution model in MPI

- Parallel program is launched as set of **independent, identical processes**



- All the processes contain the same program code and instructions
- Processes can reside in different nodes or even in different computers
- The way to launch parallel program is implementation dependent
  - mpirun, mpiexec, aprun, poe, ...
- When using Python, one launches N Python interpreters
  - mpirun -np 32 python parallel\_script.py

## MPI Concepts

- **rank**: id number given to process
  - it is possible to query for rank
  - processes can perform different tasks based on their rank

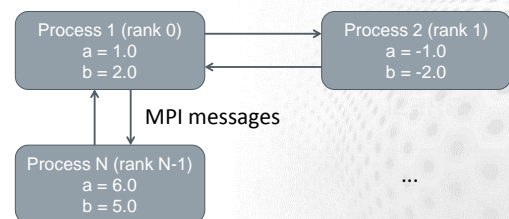
```
mpi.py
if (rank == 0):
 # do something
elif (rank == 1):
 # do something else
else:
 # all other processes do something different
```

## MPI Concepts

- **Communicator**: group containing process
  - in mpi4py the basic object whose methods are called
  - **MPI\_COMM\_WORLD** contains all the process (MPI.COMM\_WORLD in mpi4py)

## Data model

- All variables and data structures are local to the process
- Processes can exchange data by sending and receiving messages



## Using mpi4py

- Basic methods of communicator object
  - Get\_size()  
Number of processes in communicator
  - Get\_rank()  
rank of this process

```
mpi.py
from mpi4py import MPI

comm = MPI.COMM_WORLD # communicator object containing all processes
size = comm.Get_size()
rank = comm.Get_rank()

print "I am rank %d in group of %d processes" % (rank, size)
```

## Sending and receiving data

- Sending and receiving a dictionary

```
mpi.py
from mpi4py import MPI

comm = MPI.COMM_WORLD # communicator object containing all processes
rank = comm.Get_rank()

if rank == 0:
 data = {'a': 7, 'b': 3.14}
 comm.send(data, dest=1, tag=11)
elif rank == 1:
 data = comm.recv(source=0, tag=11)
```

## Sending and receiving data

- Arbitrary Python objects can be communicated with the **send** and **receive** methods of communicator
- send(data, dest, tag)**
  - data** Python object to send
  - dest** destination rank
  - tag** id given to the message
- recv(source, tag)**
  - source** source rank
  - tag** id given to the message
  - data is provided as return value
- Destination and source ranks as well as tags have to match

## Communicating NumPy arrays

- Arbitrary Python objects are converted to byte streams when sending
- Byte stream is converted back to Python object when receiving
- Conversions give overhead to communication
- (Contiguous) NumPy arrays can be communicated with very little overhead with upper case methods:
- Send(data, dest, tag)**
- Recv(data, source, tag)**
  - Note the difference in receiving: the data array has to exist in the time of call

## Communicating NumPy arrays

- Sending and receiving a NumPy array

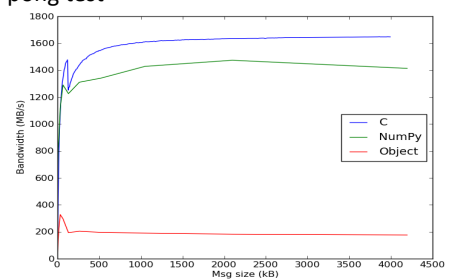
```
#p1.py
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
 data = numpy.arange(100, dtype=numpy.float)
 comm.Send(data, dest=1, tag=13)
elif rank == 1:
 data = numpy.empty(100, dtype=numpy.float)
 comm.Recv(data, source=0, tag=13)
```

- Note the difference between upper/lower case!
  - send/recv: general Python objects, slow
  - Send/Recv: continuous arrays, fast

## mpi4py performance

- Ping-pong test



## Summary

- mpi4py provides Python interface to MPI
- MPI calls via communicator object
- Possible to communicate arbitrary Python objects
- NumPy arrays can be communicated with nearly same speed as from C/Fortran