Introduction to NumPy


What is NumPy?

- NumPy is an acronym for "Numeric Python"
- Open source extension module for Python
- Provides fast precompiled functions for mathematical and numerical routines.
- Powerful data structures for efficient computation of multi-dimensional arrays and matrices.
- The module supplies a large library of high-level mathematical functions to operate on these matrices and arrays.
- NumPy is usually not installed by default. It can be downloaded from the website: http://www.numpy.org


COMPARISON BETWEEN CORE PYTHON AND NUMPY

When we say "Core Python", we mean Python without any special modules, i.e. especially without NumPy.

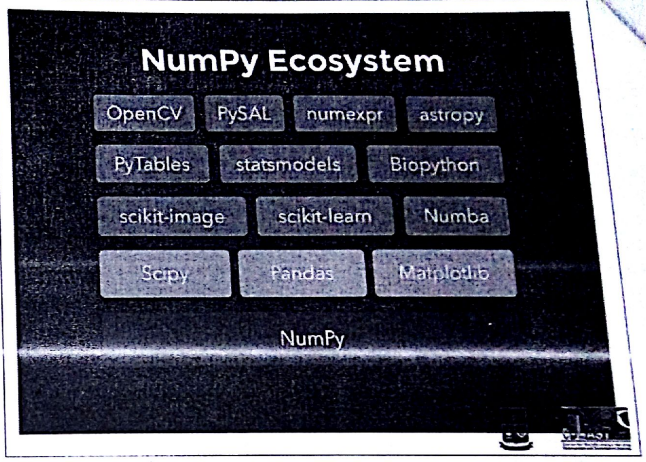
The advantages of Core Python:

- high-level number objects: integers, floating point
- containers: lists with cheap insertion and append methods, dictionaries with fast lookup

Advantages of using Numpy with Python:

- array oriented computing
- efficiently implemented multi-dimensional arrays
- designed for scientific computation


NumPy is the foundation of the Python scientific stack



A SIMPLE NUMPY EXAMPLE

Before we can use NumPy we will have to import it. It has to be imported like any other module:

```
>>> import numpy
```

But you will hardly ever see this. Numpy is usually renamed to np:

```
>>> import numpy as np
```

We have a list with values, e.g. temperatures in Celsius:

```
>>> cvalues = [25.3, 24.8, 26.9, 23.9]
```

We will turn this into a one-dimensional numpy array:

```
>>> C = np.array(cvalues)
>>> print C
```

A SIMPLE NUMPY EXAMPLE (Cont.)

Let's assume, we want to turn the values into degrees Fahrenheit. This is very easy to accomplish with a numpy array. The solution to our problem can be achieved by simple scalar multiplication:

```
>>> print C * 9 / 5 + 32
```

Compared to this, the solution for our Python list is extremely awkward:

```
>>> fvalues = []
>>> for i in cvalues:
        fvalues.append(i*9/5 + 32)
>>> print fvalues
```

CREATION OF EVENLY SPACED VALUES

There are functions provided by Numpy to create evenly spaced values within a given interval.

One uses a given distance 'arange' and the other one 'linspace' needs the number of elements and creates the distance automatically


ARANGE


The syntax of arange:
`arange([start,] stop[, step,], dtype=None)`

`arange` returns evenly spaced values within a given interval. The values are generated within the half-open interval '[start, stop)' If the function is used with integers, it is nearly equivalent to the Python built-in function range, but arange returns an ndarray rather than a list iterator as range does.
If the 'start' parameter is not given, it will be set to 0. The end of the interval is determined by the parameter 'stop'.
Usually, the interval will not include this value, except in some cases where 'step' is not an integer and floating point round-off affects the length of output ndarray. The spacing between two adjacent values of the output array is set with the optional parameter 'step'.
The default value for 'step' is 1. If the parameter 'step' is given, the 'start' parameter cannot be optional, i.e. it has to be given as well. The type of the output array can be specified with the parameter 'dtype'. If it is not given, the type will be automatically inferred from the other input arguments.

ARANGE (Cont.)

```
>>> import numpy as np
>>> a = np.arange(1, 10)
>>> print a
>>> # compare to range:
>>> x = range(1,10)
>>> print x
>>> # x is an iterator
>>> print list(x)
>>> # some more arange examples:
>>> x = np.arange(10.4)
>>> print x
>>> x = np.arange(0.5, 10.4, 0.8)
>>> print x
>>> x = np.arange(0.5, 10.4, 0.8, int)
>>> print x
```


LINSPACE

The syntax of linspace:
`linspace(start, stop, num=50, endpoint=True, retstep=False)`

`linspace` returns an ndarray, consisting of 'num' equally spaced samples in the closed interval [start, stop] or the half-open interval [start, stop). If a closed or a half-open interval will be returned, depends on whether 'endpoint' is True or False. The parameter 'start' defines the start value of the sequence which will be created.
'stop' will the end value of the sequence, unless 'endpoint' is set to False. In the latter case, the resulting sequence will consist of all but the last of 'num + 1' evenly spaced samples. This means that 'stop' is excluded.
Note that the step size changes when 'endpoint' is False. The number of samples to be generated can be set with 'num', which defaults to 50. If the optional parameter 'endpoint' is set to True (the default), 'stop' will be the last sample of the sequence. Otherwise, it is not included.


LINSPACE (Cont.)

```
>>> import numpy as np
>>> # 50 values between 1 and 10:
>>> print np.linspace(1, 10)
>>> # 7 values between 1 and 10:
>>> print np.linspace(1, 10, 7)
>>> # excluding the endpoint:
>>> print np.linspace(1, 10, 7, endpoint=False)
```


LINSPACE (Cont.)

We haven't discussed one interesting parameter so far. If the optional parameter 'retstep' is set, the function will also return the value of the spacing between adjacent values. So, the function will return a tuple ('samples','step'):

```
>>> import numpy as np
>>> samples, spacing = np.linspace(1, 10, retstep=True)
>>> print(spacing)
>>> samples, spacing = np.linspace(1, 10, 20,
endpoint=True, retstep=True)
>>> print(spacing)
>>> samples, spacing = np.linspace(1, 10, 20,
endpoint=False, retstep=True)
>>> print(spacing)
```


Time Comparison Between Python Lists and NumPy Arrays

One of the main advantages of NumPy is its advantage in time compared to standard Python. Let's look at the following functions:

```
import time
size_of_vec = 1000

def pure_python_version():
    t1 = time.time()
    X = range(size_of_vec)
    Y = range(size_of_vec)
    Z = []
    for i in range(len(X)):
        Z.append(X[i] + Y[i])
    return time.time() - t1
```

```
def numpy_version():
    t1 = time.time()
    X = np.arange(size_of_vec)
    Y = np.arange(size_of_vec)
    Z = X + Y
    return time.time() - t1
```


Time Comparison Between Python Lists and NumPy Arrays (Cont.)

Let's call these functions and see the time consumption:

```
t1 = pure_python_version()
t2 = numpy_version()
print t1, t2
print "Numpy is in this example " + str(t1/t2) + "
faster!"
```

We will get the following output, if we execute the code above:

```
0.00020909309387207 2.05039978027343e-05
Numpy is in this example 10.19767441860465 faster!
```


CREATING ARRAYS


ZERO-DIMENSIONAL ARRAYS

It's possible to create multidimensional arrays in numpy. Scalars are zero dimensional. In the following example, we will create the scalar 42. Applying the ndim method to our scalar, we get the dimension of the array. We can also see that the type is a "numpy.ndarray" type.

```
x = np.array(42)
print type(x)
print np.ndim(x)
```

The above code will return the following result:

```
<class 'numpy.ndarray'>
0
```


ONE-DIMENSIONAL ARRAYS

We have already encountered a 1-dimenional array - better known to some as vectors - in our initial example. What we have not mentioned so far, but what you may have assumed, is the fact that numpy arrays are containers of items of the same type, e.g. only integers. The homogenous type of the array can be determined with the attribute "dtype", as we can learn from the following example:

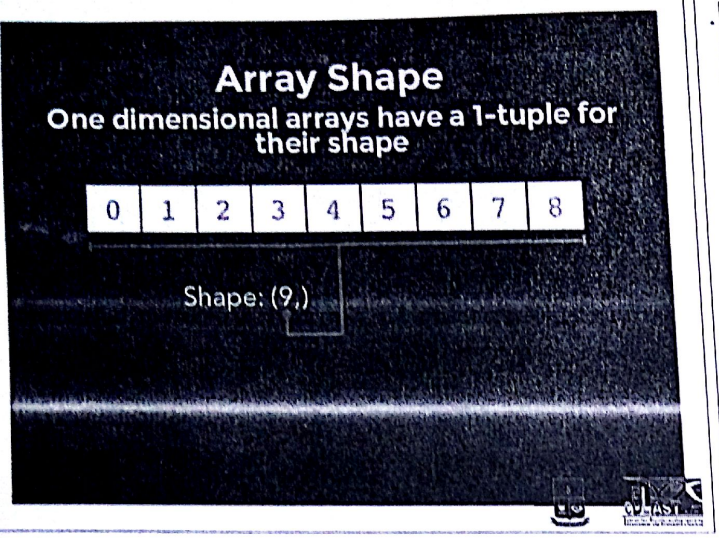
```
F = np.array([1, 1, 2, 3, 5, 8, 13, 21])
V = np.array([3.4, 6.9, 99.8, 12.8])
print F.dtype
print V.dtype
print np.ndim(F)
print np.ndim(V)
```

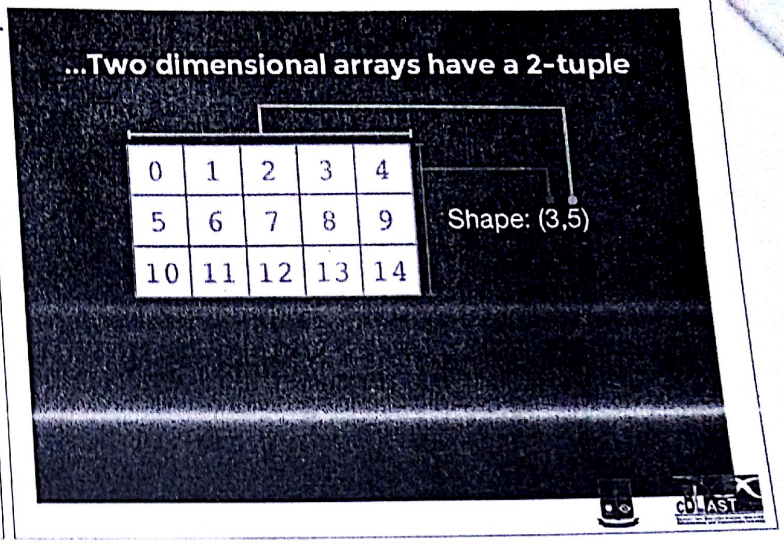

TWO- AND MULTIDIMENSIONAL ARRAYS

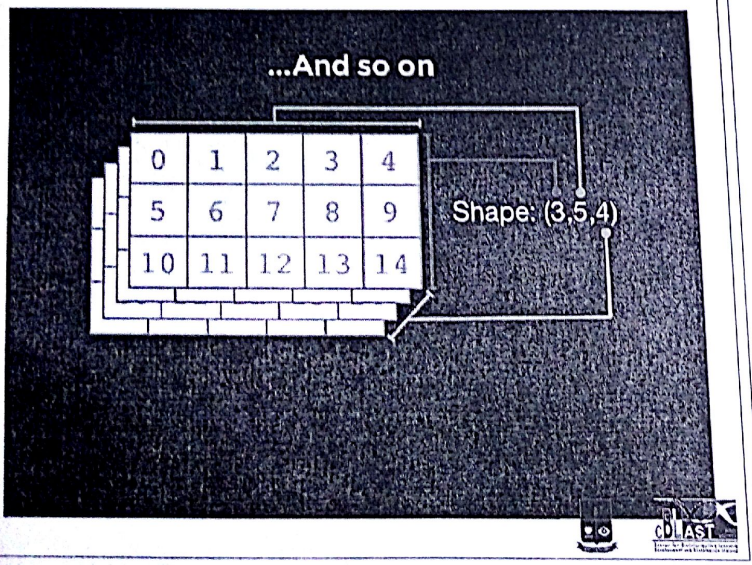
Of course, arrays of NumPy are not limited to one dimension. They are of arbitrary dimension. We create them by passing nested lists (or tuples) to the array method of numpy.

```
A = np.array([   [3.4, 8.7, 9.9],
                 [1.1, -7.8, -0.7],
                 [4.1, 12.3, 4.8]])
print A
print A.ndim

B = np.array([   [[111, 112], [121, 122]],
                 [[211, 212], [221, 222]],
                 [[311, 312], [321, 322]] ])
print B
print B.ndim
```





SHAPE OF AN ARRAY

The function "shape" returns the shape of an array. The shape is a tuple of integers. These numbers denote the lengths of the corresponding array dimension. In other words: The "shape" of an array is a tuple with the number of elements per axis (dimension). In our example, the shape is equal to (6, 3), i.e. we have 6 lines and 3 columns.

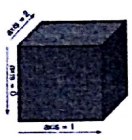
```
x = np.array([   [67, 63, 87],
                 [77, 69, 59],
                 [85, 87, 99],
                 [79, 72, 71],
                 [63, 89, 93],
                 [68, 92, 78]])
print np.shape(x)
```

There is also an equivalent array property:

```
print x.shape
```

SHAPE OF AN ARRAY (Cont.)

The shape of an array tells us also something about the order in which the indices are processed, i.e. first rows, then columns and after that the further Dimensions.
"shape" can also be used to change the shape of an array.



```
x.shape = (3, 6)
print x
```

You might have guessed by now that the new shape must correspond to the number of elements of the array,
i.e. the total size of the new array must be the same as the old one. We will raise an exception, if this is not the case:

```
x.shape = (4, 4)
```

INDEXING

Assigning to and accessing the elements of an array is similar to other sequential data types of Python, i.e. lists and tuples. We have also many options to indexing, which makes indexing in Numpy very powerful and similar to core Python. Single indexing is the way, you will most probably expect it:

```
F = np.array([1, 1, 2, 3, 5, 8, 13, 21])
# print the first element of F, i.e. the element with
the index 0
print F[0]
# print the last element of F
print F[-1]
B = np.array([  [[111, 112], [121, 122]],
                [[211, 212], [221, 222]],
                [[311, 312], [321, 322]] ])
print B[0][1][0]
```

INDEXING (Cont.)

Indexing multidimensional arrays:

```
A = np.array([  [3.4, 8.7, 9.9],
                [1.1, -7.8, -0.7],
                [4.1, 12.3, 4.8]])
print A[1][0]
```

We accessed the element in the second row, i.e. the row with the index 1, and the first column (index 0). We accessed it the same way, we would have done with an element of a nested Python list. There is another way to access elements of multidimensional arrays in numpy. There is also an alternative: We use only one pair of square brackets and all the indices are separated by Commas:

```
print A[1, 0]
```

Second way is more efficient

SLICING

Slicing in an arrya is similar to slicing of lists and tuples. The syntax is the same in numpy for one-dimensional arrays, but it can be applied to multiple dimensions as well.
The general syntax for a one-dimensional array A looks like this:

```
A[start:stop:step]
```

We illustrate the operating principle of "slicing" with some examples. We start with the easiest case, i.e. the slicing of a one-dimensional array:

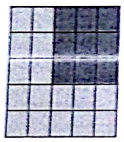
```
S = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
print S[2:5]
print S[:4]
print S[6:]
print S[:]
```

SLICING (Cont.)

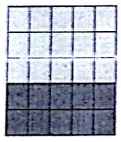
We will illustrate the multidimensional slicing in the following examples. The ranges for each dimension are separated by commas:

```
A = np.array([[11,12,13,14,15],
              [21,22,23,24,25],
              [31,32,33,34,35],
              [41,42,43,44,45],
              [51,52,53,54,55]])
```

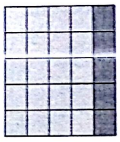
`print A[:3,2:]`



`print A[3:,:]`



`printA[:,4:]`



ARRAYS OF ONES AND ZEROS

There are two ways of initializing Arrays with Zeros or Ones. The method ones(t) takes a tuple t with the shape of the array and fills the array accordingly with ones. By default it will be filled with Ones of type float. If you need integer Ones, you have to set the optional parameter dtype to int:

```
import numpy as np
E = np.ones((2,3))
print E
F = np.ones((3,4),dtype=int)
print F
```

What we have said about the method ones() is valid for the method zeros() analogously, as we can see in the following example:

```
Z = np.zeros((2,4))
print Z
```

Exercises

1. Create an arbitrary one dimensional array called "v".
2. Create a new array which consists of the odd indices of previously created array "v".
3. Create a new array in backwards ordering from v.
4. What will be the output of the following code:

```
a = np.array([1, 2, 3, 4, 5])
b = a[1:4]
print(a[1])
```

5. Create a two dimensional array called "m".
6. Create a new array from m, in which the elements of each row are in reverse order.
7. Another one, where the rows are in reverse order.
8. Create an array from m, where columns and rows are in reverse order.
9. Cut of the first and last row and the first and last column.

Biopython (Part II)

Eutils

EUtils: Entrez Programming Utilities

- Access NCBI's online services:

```
>>> from Bio import Entrez
>>> Entrez.email = "yourmail@provider.com"
```

- Request a GenBank record:

```
>>> handle = Entrez.efetch(db="protein",
id="69316", rettype="gb", retmode="text")
>>> record = SeqIO.read(handle, "gb")
```

- Specify multiple IDs in one query:

```
>>> handle = Entrez.efetch(db="protein",
id="349839,349840",rettype="fasta", retmode="text")
>>> records = SeqIO.parse(handle, "fasta")
```

Search the Entrez nucleotide database by keyword(s)

- You can do it using a combination of `Entrez.esearch()` and `Entrez.efetch()`

```
from Bio import Entrez
from Bio import SeqIO

Entrez.email = "allegra.via@uniroma1.it"
handle = Entrez.esearch(db="nucleotide", term="Homo sapiens AND mRNA
AND MapK")
records = Entrez.read(handle)
print records['Count']
FewRecords = records['IdList'][0:3]
handle = Entrez.efetch(db = "nucleotide",retmode="xml",id =
FewRecords)

records = Entrez.parse(handle)
for record in records:
    print record.keys()
    print record['GBSeq_primary-accession']
    print record['GBSeq_sequence']
    print len(records)
```

Search the Entrez Pubmed

```
from Bio import Entrez
from Bio import Medline

keyword = "PyCogent"
# search publications in PubMed
Entrez.email = "allegra.via@uniroma1.it"
handle = Entrez.esearch(db="pubmed", term=keyword)
record = Entrez.read(handle)
pmids = record['IdList']
print pmids

# retrieve Medline entries from PubMed
handle = Entrez.efetch(db="pubmed", id=pmids, rettype="medline",
retmode="text")
medline_records = Medline.parse(handle)
records = list(medline_records)

n = 1
for record in records:
if keyword in record["TI"]:
    print n, ')', record["TI"]
```

Search the Entrez protein

```
from Bio import Entrez

Entrez.email = "allegra.via@gmail.com"

# search entries by keywords
handle = Entrez.esearch(db = "protein", term = "Human AND
cancer AND p21 AND Map kinase", rettype = "fasta")
records = Entrez.read(handle)

FewRecords = records['IdList'][0:3]
print FewRecords
ID_list = ",".join(FewRecords)

# retrieve and parse entries from the database
handle = Entrez.efetch(db = "protein", id = ID_list,
rettype="fasta", retmode="xml")

records= Entrez.read(handle)
rec = list(records)
print rec[0].keys()
print rec[0]['TSeq_defline']
```

BLAST

Running Blast

Locally

From the UNIX shell: Blast command line
From a script: Using `os.system()`
Using Biopython

Over the Internet

Using `Bio.Blast.NCBIWWW`
Using Web Programming
Using a Web Browser

Running BLAST locally

- Standalone BLAST need to be installed.
- A Python script can be use using `os.system()`

```
import os
S = "blastp -query P05480.fasta -out \
blast_output -db nr.00"
os.system(S)
```

Running BLAST over the Internet

- Using `Bio.Blast.NCBIWWW`

```
from Bio.Blast import NCBIWWW
result_handle = NCBIWWW.qblast("blastn", "nr",
                "8332116")
save_file = open("qblast_blastn.out", "w")
save_file.write(result_handle.read())
save_file.close()
result_handle.close()
```

Save the output locally in xml format

```
from Bio.Blast import NCBIWWW

result_handle =
NCBIWWW.qblast("blastn","nr","8332116",
format_type="XML")

save_file = open("qblast_blastn.xml", "w")
save_file.write(result_handle.read())
save_file.close()

result_handle.close()
```

Parsing the BLAST output

You can get BLAST output in XML in various ways: for the parser it does not matter how the output was generated as long as it is in XML format.

- Use Biopython to run BLAST over the internet
- Use Biopython to run BLAST locally
- Do the BLAST search yourself on the NCBI site through your web browser, and then save the results (choose XML format for the output)
- Run BLAST locally without using Biopython, and save the output in a file (choose XML format for the output file)

Parsing the BLAST output

- Using `Bio.Blast.NCBIXML`

```
from Bio.Blast import NCBIXML
result_handle = open("qblast_blastn.out")
#If you expect a single BLAST result
blast_record = NCBIXML.read(result_handle)
#If you have lots of results
blast_records = NCBIXML.parse(result_handle)
```

Phylogenetics

Phylogenetic tree I/O

Start with:

```
>>> from Bio import Phylo
```

Input and output of trees is just like SeqIO:

`read, parse`	single or multiple trees in Newick, Nexus and PhyloXML formats
`write`	to any of the formats supported by read/parse
`convert`	between two formats in one step

Use StringIO to load strings directly:

```
>>> from cStringIO import StringIO
>>> handle = StringIO("((A,B),(C,(D,E)));")
>>> tree = Phylo.read(handle, "newick")
```

What's in a tree?

Make a tree with branch lengths:

```
>>> tree = Phylo.read(StringIO("((A:1,B:1):2,
... (C:2,(D:1,E:1):1):1);"), "newick")
```

View the object structure of the entire tree:

```
>>> print tree
```

Draw an "ASCII-art" (plain text) representation:

```
>>> Phylo.draw_ascii(tree)
```

OK, let's do it properly now:

```
>>> Phylo.draw(tree)
```


Modify the tree

Check the tree object for its methods:

```
>>> help(tree)
```

Try a few:

```
>>> tree.get_terminals()
>>> clade = tree.common_ancestor("A", "B")
>>> clade.color = "red"
>>> tree.root_with_outgroup("D", "E")
>>> tree.ladderize()
>>> Phylo.draw(tree)
```

Protein structures

Going 3D: The PDB module

Load a structure:

```
>>> from Bio import PDB
>>> parser = PDB.PDBParser()
>>> struct = parser.get_structure('1ATP','1ATP.pdb')
```

Inspect the object hierarchy:

```
>>> list(struct)
>>> model = struct[0]
>>> list(model)
>>> chain = model['E']
>>> list(chain)
>>> residue = chain[15]
>>> list(residue)
```

Extracting a peptide sequence

Get the amino acid sequence through a Polypeptide object:

```
>>> from Bio import PDB
>>> parser = PDB.PDBParser()
>>> struct = parser.get structure('1ATP', '1ATP.pdb')

>>> ppb = PDB.PPBuilder()
>>> peptides = ppb.build peptides(struct)
>>> for pep in peptides:
... print pep.get sequence()
```


Calculating RMSD

Given two aligned structures, filter a list of target residues for high RMS deviation.

Input: - list of residue positions (integers)
- two equivalent chains from aligned protein
- models — residue numbers must match
- Minimum RMSD value (float)

Output: - list of residue positions, filtered

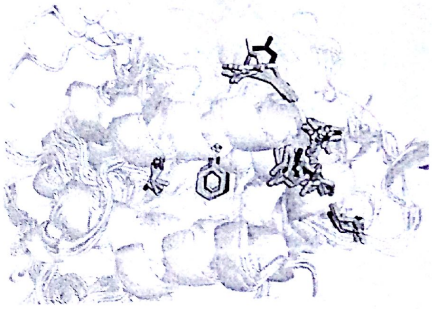
Procedure: -Extract coordinates of Cα atoms
- If available (not glycine), extract Cβ coordinates, too
- Use Bio.SVDSuperimposer to calculate the RMSD between coordinates
- Compare to the given RMSD threshold

Calculating RMSD [Code]

```
from Bio.SVDSuperimposer import SVDSuperimposer
from numpy import array

def filtrms(resids, refchain, cmpchain, thresh=0.5):
    super = SVDSuperimposer()
    for res in resids:
        refres = refchain[res]
        cmpres = cmpchain[res]
        coord1 = [refres['CA'].get_coord()]
        coord2 = [cmpres['CA'].get_coord()]
        if refres.has_id('CB') and cmpres.has_id('CB'):
            # Not glycine
            coord1.append(ref_res['CB'].get_coord())
            coord2.append(cmp_res['CB'].get_coord())
        super.set(array(coord1), array(coord2))
        rmsd = super.get_init_rms()
        if rmsd >= threshold:
            yield res
```

Superimposed structures, with selected deviating residues



Further Reading

The Biopython Tutorial and Cookbook contains the bulk of Biopython documentation.

It provides information to get you started with Biopython, in addition to specific documentation on a number of modules.

Visit: http://biopython.org/DIST/docs/tutorial/Tutorial.pdf

Exercise 1: Filtering DNA sequences

Input data: DNA sequencing reads in FASTQ format in file *sample1.fq*

Problem: Write a script that:
1. reads the sequences
2. writes only those reads which
   a) are >= 200 bp long
   b) do not contain any uncalled bases (i.e. 'N' s)

to an output file *sample1_filtered.fq*

Exercise 2: Calculate FASTQ stats

Input data: DNA sequences in FASTQ format from file *reads.fq*

Problem: Write a script that
a) calculates the following values for each read:
id, length, GC content, A count, T count, C count, G count, avg. quality
b) writes these as columns to a *csv* file (one row per read)

Hints:
- You can use `Bio.SeqUtils.GC` for calculating GC content
- Find base qualities for each record in record.letter_annotations["phred_quality"]

Pandas

1

Pandas DataFrames

- Pandas is a high-level data manipulation tool developed by Wes McKinney.
- It is built on the Numpy package and its key data structure is called the DataFrame.
- DataFrames allow you to store and manipulate tabular data in rows of observations and columns of variables.

2

DataFrame from Scratch

```
import pandas as pd

dict = {"country": ["Brazil", "Russia", "India", "China", "South
Africa"],
       "capital": ["Brasilia", "Moscow", "New Dehli", "Beijing",
"Pretoria"],
       "area": [8.516, 17.10, 3.286, 9.597, 1.221],
       "population": [200.4, 143.5, 1252, 1357, 52.98] }

brics = pd.DataFrame(dict)
print brics
```

3

Setting Index

- As you can see with the new brics DataFrame, Pandas has assigned a key for each country as the numerical values 0 through 4. If you would like to have different index values, say, the two letter country code, you can do that easily as well.

```
# Set the index for brics
brics.index = ["BR", "RU", "IN", "CH", "SA"]

# Print out brics with new index values
print(brics)
```


Read CSV

```
# Import pandas as pd
import pandas as pd

# Import the cars.csv
data: cars
cars =
pd.read_csv('cars.csv')

# Print out cars
print(cars)
```

Indexing DataFrames

- There are several ways to index a Pandas DataFrame. One of the easiest ways to do this is by using square bracket notation.

```
# Import pandas and cars.csv
import pandas as pd
cars = pd.read_csv('cars.csv', index_col = 0)

# Print out country column as Pandas Series
print(cars['cars_per_cap'])

# Print out country column as Pandas DataFrame
print(cars[['cars_per_cap']])

# Print out DataFrame with country and
drives_right columns
print(cars[['cars_per_cap', 'country']])
```


- Square brackets can also be used to access observations (r DataFrame. For example:

```
# Import cars data
import pandas as pd
cars = pd.read_csv('cars.csv',
index_col = 0)

# Print out first 4 observations
print(cars[0:4])

# Print out fifth, sixth, and
seventh observation
print(cars[4:6])
```

- You can also use loc and iloc to perform just about any data selection operation.
- `loc` is label-based, which means that you have to specify rows and columns based on their row and column labels.
- `iloc` is integer index based, so you have to specify rows and columns by their integer index like you did in the previous exercise.


```
# Import cars data
import pandas as pd
cars = pd.read_csv('cars.csv',
index_col = 0)

# Print out observation for Japan
print(cars.iloc[2])

# Print out observations for
Australia and Egypt
print(cars.loc[['AUS', 'EG']])
```