<table>
<thead>
<tr>
<th>1</th>
<th>Practical session 04</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Exceptions</td>
<td>3</td>
</tr>
<tr>
<td>1.2</td>
<td>Lists as arrays</td>
<td>4</td>
</tr>
<tr>
<td>1.3</td>
<td>Python Modules</td>
<td>4</td>
</tr>
<tr>
<td>1.4</td>
<td>Numpy arrays</td>
<td>5</td>
</tr>
<tr>
<td>1.5</td>
<td>TASKS</td>
<td>6</td>
</tr>
</tbody>
</table>
Files for this practical can be found at http://becksteinlab.physics.asu.edu/pages/courses/2013/SimBioNano/03/

Contents:
The three virtues of a programmer are
Laziness  Impatience  Hubris
— Lary Wall (inventor of the Perl programming language)

1.1 Exceptions

Exceptions are a way to stop control flow if something bad has happened:

```python
raise Exception("description")
raise ValueError("description")
raise TypeError("description")
raise IOError(number, "description")
```

Exceptions can be trapped with `try ... except`:

```python
try:
    # some python code that might fail
    f = open(filename)
except IOError:
    print("Problem with opening \$r" % filename)
    print("Trying default file")
    f = open("default.xyz")
```

One can add cleanup in a `finally` clause, which is *always* execute, no matter what happens in `try`:

```python
output = open(filename, "w")
try:
    for x,y,z in coordinates:
        output.write("%f %f %f
" % (x,y,z))
finally:
    output.close()
```

This makes sure that the final is always closed and completely written to disk, even if something bad happens in between.

For files their’s an even more concise way to write the above with the `with` statement (since Python 2.5):

```python
with open(filename, "w") as output:
    for x,y,z in coordinates:
        output.write("%f %f %f\n" % (x,y,z))
```

When the `with` block ends, the file is automatically closed (the file object knows what to do when the block ends!)
1.2 Lists as arrays

Make a pseudo coordinate array for 4 “atoms”:

```python
>>> x0 = [[0, 1, 2], [3, 4, 5], [6, 7, 8], [9, 10, 11]]
```

Access coordinates of 3rd atom (index 2!):

```python
>>> x0[2]
[6, 7, 8]
```

Access y coordinate of atom 2:

```python
>>> x0[2][1]
7
```

Access all z coordinates:

```python
>>> z0 = []
... for x in x0:
...     z0.append(x[2])
...print z0
[2, 5, 8, 11]
```

Better with “list comprehensions” (implicit loop):

```python
>>> z0 = [x[2] for x in x0]
[2, 5, 8, 11]
```

Subtracting two positions (connecting vector), eg 0 --> 1: r = b - a

```python
a = x0[0]
b = x0[1]
r = [b[i] - a[i] for i in xrange(3)]
```

Length of the vector \( \sqrt{r \cdot r} \):

```python
import math
d = math.sqrt(sum([ri**2 for ri in r]))
```

**Note:** `import` loads a module, a library of additional functions and objects. These additional functions can be accessed with the `module_name-DOT-function_name` syntax.

See `help('math')` to learn more about the math module.

1.3 Python Modules

Code re-use is key to efficient coding. You put useful code into functions (or classes for object oriented programming). These code pieces are collected in “libraries” or *modules* as they are called in Python.

For instance, the `read_xyz()` and `write_xyz()` are in a file `mdIO.py` in your current directory. Then you can access them with

```python
import mdIO

atoms, coordinates = mdIO.read_xyz(filename)
mdIO.write_xyz(filename, atoms, coordinates)
```
You can also import individual functions:

```python
from mdIO import read_xyz, write_xyz
from math import sqrt
```

Finally, you can even rename functions (or modules) when importing:

```python
import numpy as np
from mdIO import read_xyz as rxyz
```

Imports and renamed objects are only available in the file where you imported them.

### 1.4 Numpy arrays

**NumPy** is a Python package for scientific computing. Fundamentally, it provides efficient array structures. They are much better and more performant than oridnary lists.

Use NumPy package:

```python
>>> import numpy
>>> numpy.array([1,2,3])
```

However, we are lazy and don’t want to type too much:

```python
>>> import numpy as np
>>> x = np.array([[0,1,2],[3,4,5],[6,7,8],[9,10,11]])
```

3rd atom:

```python
>>> x[2]
array([6, 7, 8])
```

y of 3rd atom:

```python
>>> x[2][1]
7
>>> x[2,1]
7
```

z of all atoms:

```python
>>> x[:,2]
array([ 2, 5, 8, 11])
```

subtracting two positions:

```python
>>> a = x[0]; b = x[1]
>>> r = b - a
>>> print r
array([3, 3, 3])
```

or shorter:

```python
>>> x[1] - x[0]
```

length of distance:

```python
>>> r*r # element-wise!
>>> np.sqrt(np.sum(r*r))
```
Also note multiplication with a scalar (element-wise):

```python
>>> 10*r
array([30, 30, 30])
```

Arrays are not “vectors” and 2-d arrays are not “matrices”: all operations are done element-wise.

Other array operations:

```python
a+b
a += b  # a = a + b
a/b
a *= 10  # a = 10*a
```

All are done element-by-element.

### 1.4.1 Advanced: Broadcasting

(Not discussed in class, here for completeness sake.)

Size of arrays (“shape”):

```python
>>> x
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
```

```python
>>> x.shape
(4, 3)
```

```python
>>> y = np.array([-1,1,100])
array([-1, 1, 100])
```

```python
>>> y.shape
(3,)
```

Broadcasting replicates a smaller array as many times as to match a bigger one. This only works if the last dimensions match:

```python
>>> x * y
array([[  0,   1,  200],
        [ -3,   4,  500],
        [ -6,   7,  800],
        [ -9,  10, 1100]])
```

```python
>>> x[0] * y
array([  0,   1,  200])
```

```python
>>> x[1] * y
array([ -3,   4,  500])
```

You need to play around with it to get your hand around... there’s more to that than what we touched on. (E.g., what happens for `array([[1],[2],[3]]) * array([-1,1])` and why?)

### 1.5 TASKS

See material at http://becksteinlab.physics.asu.edu/pages/courses/2013/SimBioNano/04/
1. **mdIO**: use `np.array` in `read_xyz()` and `write_xyz()`: edit `mdIO_v0.py` to create `mdIO.py`. `mdIO` will be your module to read and write structures (and later trajectories).

2. **Periodic boundary conditions**: Write a function

   ```python
def mimg(x, box):
    # That maps x into the interval [-box/2, box/2].
    # Extend your function to 3D (assume for the moment that you have a cubic box)
    (a) First make x a point `r = np.array([x, y, z])` and have the function return another point 3D.
    (b) Then change it so that x can be a whole coordinate list `np.array([[x1, y1, z1], [x2, y2, z2], ...])`.
    (c) **Bonus extension**: Generalize to an orthorhombic cell so that the box lengths are `box = np.array([L1, L2, L3])`.

    Test version 2 on the coordinates in `Ar_rho=0.0333_fcc_unwrapped.xyz`. Find the box dimensions from the line containing "box" in the file.

    Visualize the input and output system in **VMD**:
    (a) Launch /Applications/VMD
    (b) File -> New Molecule: Browse for the file and Load
    (c) Graphics -> Representations: Drawing Method: VDW

    **See Also**:

    Possibly useful functions: `numpy.around()` (used to be called `numpy.round()`), `numpy.ceil()`, `numpy.floor()`, `numpy.rint()`