Lecture 3: Lists, arrays, and optimisation with NumPy

Dr Benjamin J. Morgan[1] and Dr Andrew R. McCluskey[1,2]

[1] *Department of Chemistry, University of Bath, email: b.j.morgan@bath.ac.uk*
[2] *Diamond Light Source, email: andrew.mccluskey@diamond.ac.uk*

July 25, 2019

Aim

This lecture will introduce lists, arrays, and show how the NumPy library can be used to make your code faster.

1 Lists

The Python programming language natively includes the ability to group together a series of objects. These are `lists` and are one of the most powerful Python objects. Lists are an ordered set of objects, from which it is possible to pick all, one, or many values. A list is defined as follows,

```
# Making a list

elements = ["Hydrogen", "Helium", "Lithium", "Beryllium",
    "Boron", "Carbon", "Nitrogen", "Oxygen"]
```

Having defined the list, it is then possible to select individual items of the list by using the following syntax,

```
# Printing some items

print(elements[0], elements[4], elements[-1])
```

Note, that Python starts counting from the number 0, and using the minus sign we can ask Python to count from the end. This means that the above code should print, `"Hydrogen", "Boron", "Oxygen"`. This counting from 0 means that in the above list, the string `"Hydrogen"` would be referred to as the zeroth object in the list, while `"Helium"` would be the first.

In addition to making use of single objects from within a list, it is also possible to create sublists, for example,

```
# Just the first 4 elements

print(elements[0:4])
```

Note that above, the numbers on either side of the colon the list indices. However, rather strangely, the sublist created is **inclusive** of the first number and **exclusive** of the second. Additionally, it is possible to select non-consecutive objects from a list by placing commas between the indices,

```
# Just the gases

print(elements[0, 1, 6, 7])
```

The final point about `lists` is that the data that they hold does not all need to be the same time. For example, the list below contains a `float`, two `str`, a `complex` number and an `int`,

```
# List of many types

a_new_list = ['hello', 12.41242, 5 + 8j, 'sadness', 2]
print(a_new_list)
```

Exercise

- Create two lists, one containing names the first 8 elements in the periodic table and another containing the massive numbers for those elements. Then, using a loop, print each element name and mass number, format each print statment with the `.format()` syntax.

2 NumPy Arrays

NumPy (or `numpy` or more commonly `np`) is a library that Python can use that is designed and optimised for doing numerical operations.[1] Over this course you will be introduced to many other Python libraries, in order to use any of these you must `import` them,

```
# Import NumPy

import numpy as np
```

This asks the Python interperator to go and find the NumPy library, then in order to reduce the amount of typing (programmers are lazy), we give the library the alias `np`.

One of the most powerful features of the NumPy library is the `array`, these are similar to lists but with some important differences. Unlike a list, all of the items in a NumPy array must be of the same type; namely a NumPy data type (a list of these can be found online: `https://docs.scipy.org/doc/numpy/`

`user/basics.types.html`) which are numerical data types such as `int`, `float`, and `complex`.

The power of a NumPy array comes in the ability to perform mathematical operations incredibly efficiently. For example,[1] the summation of zero to ten million is ~ 25 times faster when using the NumPy array operation shown below when compared with a simple implementation in pure Python,

```
# The pure Python way

numbers = range(10000000)
total = 0
for i in numbers:
    total = total + i
print(total)

# The NumPy operation

import numpy as np

numbers = np.arange(10000000)
total = np.sum(numbers)
print(total)
```

Note that in the above example, the `range` function creates a list of numbers from 0 to 10000000, while the `np.arange` function creates a NumPy array containing the same values.

NumPy arrays also have a *huge* amount of additional functionality, such as the ability to easily access statistically relevant values, powerful sub-array definition (in particular for multi-dimensional arrays), data reorganisation. Some of these tools are shown below, and no doubt you will become familiar with many others throughout this course,

```
# Determine the mean and standard deviation
import numpy as np

## First get an array of 6 numbers
x1 = np.array([2, 5, 3, 7, 2, 7])

print(x1.mean(), x1.std())

## Now get a two-dimensional array of random ints from 0 to 10
x2 = np.random.randint(10, size=(3, 2))

print(x2.shape)
print(x2)
print(x2[0])
print(x2[:, 1])
print(x2[:, 0::-1])
```

[1] When running on a MacBook Air 2018 with a 1.6 GHz Intel Core i5.

```
## Lets reshape a one-dimensional array
x3 = np.arange(10)

print(x3)
print(x3.reshape((3, 3))
```

Like other numerical types in Python, it is possible to perform mathematical operations on them (`+`, `-`, `*`, `/`, etc.). Furthermore, additional operations are available from the NumPy library, such as the dot product of two arrays or the determinant of the result,

```
# Dot product ot two-dimensional arrays

a = np.array([[1, 0], [0, 1]])
b = np.array([[4, 1], [2, 2]])
c = np.dot(a, b)
print(c, np.det(c))
```

2.1 Optimisation with NumPy

It was shown above, that by replacing a Pythonic loop with a NumPy array operation it was possible to get a massive speed up in computational efficiency. This is a powerful tool of the NumPy library, that **must** be harnessed, where possible. The general advice is that when a loop is present in code, you should consider if it would be possible to replace this with an appropriate NumPy operation. Throughout the remainder of this module, we will make use of NumPy array operations over looping through lists whereever possible.

Exercise

- Write some *NumPy optimised* code that will calculate the average vibrational energy, $\bar{E}_v$, from the first N energy levels, E_l, of some diatomic molecule when,

$$\bar{E}_v = \frac{\sum_{i=0}^{N} E_l p}{\sum_{i=0}^{N} p}. \tag{1}$$

  Where, $N = 6$, the energy at each level is: 0, 1, 2, 3, 4, 5, 6 and the levels have populations, p: 4, 3, 2, 1, 0, 0 respectively.

  Consider how the pure Python version of this code would work.

3 Copying lists/arrays

An important fact to be aware of for both lists and NumPy arrays is that assigning a list to an new variable does **not** create a new list. Rather, this will create an alias to the same object in memory. In order to create a new list (or array), it is best to `copy` the original object to the new variable, as shown below,

```
# Copying lists and arrays

my_list = ['dog', 'cat', 'horse']

new_list = my_list

new_list[0] = 'giraffe'

print(new_list)
print(my_list)

copied_list = my_list.copy()

copied_list[1] = 'pig'

print(copied_list)
print(my_list)
```

Note that for a NumPy array, the function `np.copy(my_array)` should be used.

4 Problem

For this week's problem we will optimise the code written last week to calculate interatomic distances. Look back at the code that wrote last week and consider how (by using the NumPy array) it might be possible to improve the efficiency. This should involve considering again your algorithm, as it might require modification to work in an optimised fashion. Compare the results of your new optimised code to that from last week (they should be the same).

References

[1] T. E. Oliphant, *A guide to NumPy*, Trelgol Publishing USA, 2006, vol. 1.