# Python lab, lecture 2

Last week we introduced the basics of doing calculations in Python, using Python packages such as `numpy` via the `import` command, and defining our own functions. This allows us to do a single calculation using the computer. In particular, we covered: * `spyder` basics * variables * `import` * the very basics of defining functions

However, the power of using a computer is its ability to do many calculations rapidly (without getting bored). We need to be able to work with lots of things in one go in order to make Python do these many operations.

## Lists

To start with, we need an object that holds many things. Mathematically, there are many such objects: sets, groups, vectors, matrices are some examples. Each of these *contains* a number of other things. For example, the simple vector $\mathbf{v} = (0, 1, 2, 3)$ contains the four numbers $0, 1, 2, 3$.

The first Python object that we'll encounter that allows us to hold many things is a *list*. Define it as:

```
v = [0, 1, 2, 3]
print(v)
```

```
[0, 1, 2, 3]
```

The square brackets `[]` say that what follows will be a list: a collection of objects. The commas separate the different objects contained within the list.

In Python, a list can hold *anything*: it is like a mathematical set (but one which allows repetitions and which is ordered). For example:

```
w = [0, 1.2, "hello", [3, 4]]
print(w)
```

```
[0, 1.2, 'hello', [3, 4]]
```

This list holds an integer, a real number (or at least a floating point number), a string, and another list.

We can find the length of a list using `len`:

```
print(len(v))
```

```
4
```

To access individual elements of a list, use square brackets again. The elements are ordered left-to-right, **and the first element has number (or index) `0` (in Python we count starting from 0!)**:

```
print(v[0])
print(v[3])
print(w[1])
print(w[3])
```

```
0
3
1.2
[3, 4]
```

If we try to access an element that isn't in the list we get an error:

```
print(v[4])
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call
last)

<ipython-input-5-6f9feae2ef3a> in <module>
----> 1 print(v[4])


IndexError: list index out of range
```

Notice that `v[4]` asks for the element in position 4 `v` (which would be the 5th if we counted from 1). `v` only contains `len(v)` = 4 items though.

We can assign the value of elements of a list in the same way as any variable:

```
v[1] = 10
print(v)
```

```
[0, 10, 2, 3]
```

We can work with multiple elements of a list at once using the *slicing* notation:

```
print(v[0:2])
print(v[:2])
print(v[1:])
print(v[0:4:2])
print(v[::2])
```

```
[0, 10]
[0, 10]
[10, 2, 3]
[0, 2]
[0, 2]
```

The notation `[start:end:step]` means to return the entries from the `start`, up to **but not including** the `end`, in steps of length `step`. If the `start` is not included (e.g. `[:2]`) it defaults to the start, i.e. `0`. If the `end` is not included (e.g. `[1:]`) it defaults to the end (i.e., `len(...)`). If the `step` is not included it defaults to `1`.

Finally, we can use negative numbers to count from the end of the list:

```
print(v[-1])
print(v[-2])
print(v[-1:0:-1])
```

```
3
2
[3, 2, 10]
```

We can access strings (i.e. sequences of characters) in the same way:

```
s = "hello"
print(s[2:5])
```

```
llo
```

Exercise (not for the lab)

Write a function that takes a single string `s` as input and returns its characters in reverse order, except for the first.

### Loops

We now have an object that collects lots of things together. Now we want to perform a calculation on each object in the list.

Suppose we want to know $x^2 \sin(x)$ for every integer $x$ from $1$ to $5$.

We start by `import`ing the `numpy` library to get the $\sin$ function:

```
import numpy
```

Then we would create a list representing the numbers $x$:

```
numbers = [1, 2, 3, 4]
```

Then, **for each** number **in** our list of `numbers`, we want to perform the calculation. The Python syntax for this *loop* is:

```
for number in numbers:
    result = number**2 * numpy.sin(number)
    print(number, result)
print("Outside the loop")
```

```
1 0.8414709848078965
2 3.637189707302727
3 1.2700800725388048
4 -12.108839924926851
Outside the loop
```

The Python code says to take the list `numbers` and, one at a time, set the variable `number` equal to each element within `numbers` in order. The keyword `for` says to do it in order; the keyword `in` says to take the values from within the list.

The colon `:` means that the lines that follow are to be executed every time the code goes through the loop. This is exactly the syntax as was used when defining functions.

The lines to be executed are indented by four spaces. We can have many lines to be executed within the loop, as above: all must be indented. To exit the loop, stop indenting the code (as with the final `print` statement above).

We can also *nest* loops. That is, we can have loops inside loops.

```
exponents = [2, 3, 5]
for number in numbers:
    print("Within first loop")
    for exponent in exponents:
        print(number, "to the power of", exponent, "is",
number**exponent)
    print("Finished inner loop")
print("Finished all loops")
```

```
Within first loop
1 to the power of 2 is 1
1 to the power of 3 is 1
1 to the power of 5 is 1
Finished inner loop
Within first loop
2 to the power of 2 is 4
2 to the power of 3 is 8
2 to the power of 5 is 32
Finished inner loop
Within first loop
3 to the power of 2 is 9
3 to the power of 3 is 27
3 to the power of 5 is 243
Finished inner loop
Within first loop
4 to the power of 2 is 16
4 to the power of 3 is 64
4 to the power of 5 is 1024
Finished inner loop
Finished all loops
```

In the above we have often created a list of consecutive integers and looped over that. We can use the `range` function for this. When used within a loop, `range(start, end, step)` produces integers that run from `start` up to, **but not including**, the `end`, with steps of `step`. For example

```
for number in range(1, 10, 2):
    print(number)
```

```
1
3
5
7
9
```

Exercise

Using nested loops, compute

$$\sum_{n=1}^{1000} n^{-p}$$

for $p = 2, 3, 4, 5$.

### Conditional statements

At some point in our code, we may want to do a certain operation only if our variables satisfy a certain condition. This is done with the `if` statement. The syntax is similar to that for functions (which used the `def` keyword) and loops (which used the `for` keyword).

Given a list of numbers, let us use `if` to count the occurrences of the number `6`.

```
list = [1, 6, 7, 2, 4, 6, 6, 7]
counter = 0
for number in list:
    if number == 6:
        counter = counter + 1
print("Found", counter, "occurrences of the number 6")
```

```
Found 3 occurrences of the number 6
```

The instruction `counter = counter + 1` is executed only if the condition `v == 6` is satisfied. The syntax is to use the `if` keyword followed by some logical (Boolean, true of false) statement. After the condition a colon `:` is used to indicate the lines that should be executed. The lines to be executed are then indented by four spaces. Again, this follows the syntax for functions and loops. In the condition we used the `==` operator. This is different from `=`,as `==` compares two objects whereas `=` assings the value of one object to another one.

Exercise

Write a function `find_min` that takes a list `list` as input and returns the smallest number it contains.