Lab Week 4

In weeks 2 and 3 we did

- lists: definition, slicing;
- `for` loops;
- `numpy` arrays;
- `numpy` linear algebra operations.

Script

In previous weeks we introduced containers such as lists and `numpy` arrays. These can be used for one essential operation in Mathematics, and particularly Operational Research: ordering and sorting things.

For example, let us take an *un-ordered* list:

```
list_1 = [3, 2, 5, 1, 4]
```

There are two ways that we can sort this. There is the `sorted` function:

```
print(sorted(list_1))
```

```
[1, 2, 3, 4, 5]
```

This function returns a *new list*. We can check that the original list is unchanged:

```
print(list_1)
```

```
[3, 2, 5, 1, 4]
```

There is also the `sort` *method*:

```
list_1.sort()
print(list_1)
```

```
[1, 2, 3, 4, 5]
```

This changes the values of the original list. It's usually slightly faster, but also usually not what we want to do.

If using a `numpy` array, there are similar functions and methods:

```
import numpy
```

```
array_1 = numpy.array([3, 2, 5, 1, 4])
print(numpy.sort(array_1))
print(array_1)
array_1.sort()
print(array_1)
```

```
[1 2 3 4 5]
[3 2 5 1 4]
[1 2 3 4 5]
```

The in-built functions are efficient and useful. What we want to do is to implement our own sorting algorithms to show how this can be done.

Unpacking and multiple assignment

We will start with a small, two-element list:

```
list_1 = [2, 1]
```

We want to sort it in ascending order. We see that to do this we need to switch the first (`0`) and second (`1`) entries.

How would you do this?

If you tried:

```
list_1[0] = list_1[1]
list_1[1] = list_1[0]
```

then it will fail:

```
print(list_1)
```

```
[1, 1]
```

The first assignment throws away the entry `2`, and it can't be recovered.

We could use a temporary variable:

```
list_1 = [2, 1]

tmp = list_1[0]
list_1[0] = list_1[1]
list_1[1] = tmp

print(list_1)
```

```
[1, 2]
```

This works, but involves more lines of code than we would like.

We can take advantage of some Python features with multiple variables. If you have multiple variables, or a container (like a list) containing multiple objects, on one side of the assignment (`=`), then Python will expand each individually.

For example, we can assign to multiple variables using unpacking:

```
list_1 = [2, 1]

a, b = list_1
print(a)
print(b)
```

```
2
1
```

We can use the same variable on both sides, and Python will "do the right thing":

```
b, a = a, b
print(a)
print(b)
```

```
1
2
```

So we can use this to flip the entries of `list_1` as needed:

```
list_1[1], list_1[0] = list_1[0], list_1[1]
print(list_1)
```

```
[1, 2]
```

Exercise

Write a function `swap` that takes a list, `unswapped`, and two integers `i, j`. It should swap the `i` th and `j` th entries of the list and return the swapped list.

Conditional statements

We still need to get the computer to check when it should swap the entries in a list. That is, how do we make it do the swap *only if* a certain condition holds?

This is the Python `if` statement. The syntax is similar to that for functions (which used the `def` keyword) and loops (which used the `for` keyword):

```
list_1 = [2, 1]
list_2 = [3, 4]

if list_1[1] < list_1[0]:
    print("Swapping entries in list 1")
    list_1[1], list_1[0] = list_1[0], list_1[1]

if list_2[1] < list_2[0]:
    print("Swapping entries in list 2")
    list_2[1], list_2[0] = list_2[0], list_2[1]

print(list_1)
print(list_2)
```

```
Swapping entries in list 1
[1, 2]
[3, 4]
```

We see that *only the indented code* for `list_1` was executed. 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.

Exercise

Write a function `max_swap` that takes the unswapped list, `unswapped`, and two integers `n` and `nmax`. It should compute two integers `child1 = 2*n+1` and `child2 = 2*(n+1)`, find which is the largest entry `unswapped[i]`, where `i` can be `n` or `child1` or `child2`. If the largest entry is not `unswapped[n]`, the `swap` function should be used to make it the largest. The `child*` entries should only be used if `child*` is less than `nmax`.

Exercise

The function `max_swap` is part of setting up a *heap*, as required for the *heap sort* algorithm seen in lectures. Two modifications are needed to create the heap. First, it should be called on all the entries. Second, if it swaps entries, it must call itself again, to check if any of the new children of the current maximum are larger. So write a function `heapify` that adds this one line to `max_swap`.

In a heap sort, we first construct the heap. We start from the *middle* of our list, work backwards, and ensure that the children of all points in the first half of the list are smaller than their parents. With our `heapify` function, we can do that with a loop:

```
list_1 = [3, 2, 5, 1, 4]
for i in range(len(list_1)//2 - 1, -1, -1):
    list_1 = heapify(list_1, i, len(list_1))
print(list_1)
```

```
[5, 4, 3, 1, 2]
```

Let us check that we have a heap. The first entry (`0`) is the largest, as required. Its children (entries `2 n + 1 = 1`, with value `4` and `2 (n + 1) = 2` with value `3`) are both smaller than it, as required. The first child has children (entries `2 n + 1 = 3`, with value `1`, and `2 (n + 1) = 4`, with value `2`) that are both smaller than it, as required. The second child has no children within the list (both `2 n + 1 = 5` and `2 (n + 1) = 6` are too large).

As soon as we have a heap we can work from the end of the list to the beginning, swapping the entries as we go. We then recreate the heap, but only for the entries that haven't been swapped: that is, for the start of the array.

```
for i in range(len(list_1)-1, 0, -1):
    list_1 = swap(list_1, i, 0)
    list_1 = heapify(list_1, 0, i)
print(list_1)
```

```
[1, 2, 3, 4, 5]
```

Exercise

Convert this into a `heap_sort` function that sorts a list in place.

Functions and documentation

When we defined the functions above we gave them minimal documentation. We can see this documentation on the screen using the `help` function:

```
help(heap_sort)
```

```
Help on function heap_sort in module __main__:

heap_sort(unsorted)
    The heap sort algorithm
```

Alternatively, in `spyder`, go to the "Help" pane in the top right and type `bubble_sort` into the box. The help text should appear.

The documentation is needed to explain to the next person to use the function

- what it does
- how it does it
- how it should be used
- what values it returns.

A function without documentation is **fundamentally broken**, as the next person to use it will not (easily) be able to understand it. As this person is most likely to be you, the person that wrote it, then you are helping yourself by writing proper documentation.

There are various conventions for how to document functions in Python. We recommend the "numpydoc" convention, which is used within `numpy`. We will now improve the documentation for the `bubble_sort` function to illustrate:

```
def heap_sort(unsorted):
    """
    The heap sort algorithm

    Parameters
    ----------

    unsorted: list of float
        The unsorted list

    Returns
    -------

    sorted: list of float
        The sorted list (which is unsorted, sorted in place)

    Notes
    -----

    This algorithm sorts the list in place, replacing the original
entries.
    The worst case speed is O(n log(n)).
    """
    # Create the heap
    for i in range(len(unsorted)//2 - 1, -1, -1):
        unsorted = heapify(unsorted, i, len(unsorted))
    # Flatten the heap
    for i in range(len(unsorted)-1, 0, -1):
        unsorted = swap(unsorted, i, 0)
        unsorted = heapify(unsorted, 0, i)
    return unsorted
```

The documentation starts with a brief description of the algorithm. There is then a section describing the input arguments, or parameters, to the function. The name of each is given, along with its expected type, and what it means. Next is a section on the variables returned from the function, using the same conventions. Finally, a notes section allows us to add more details of the algorithm, when there may be problems, references to the literature, or examples of how to use it.

Check how this documentation appears using the `help` function, and also when using `spyder`'s help facility.

Exercise

Think of documentation like a contract. The documentation guarantees that *if* the input follows the specified form, *then* the function will return correct output in the specified form.

For the `heap_sort` function, what input causes the function to fail? Test various inputs (empty lists, single values, strings, lists of lists, etc) and see what happens. How would you tighten and/or improve the documentation to make this clear?