Some simple programming exercises

Here are some very introductory examples with some elementary algorithms. When examples are provided with the expected outcome, it is very important that you make sure that your code matches the examples exactly. Details matter!!

Important: Always test your code, either in the console or in source file. Make sure it gives the correct results for simple cases where you can predict the output.

Important: Remember to be both curious and precise. If you don't understand something, investigate it. If you observe something peculiar, investigate it. Don't be afraid to experiment.

Important: Do not use any fancy Python indexing rules in these exercises in case you know about them! (If you don't know them, good, ignore this.) In particular, it's not allowed to use: negative indexing, slicing, or growing/shrinking lists dynamically. One index at a time, and always non-negative, please.

1. Swap the values of two variables

Let's say that, at some point in your code, you have two variables, `a` and `b`. You want to swap their values. For example, if `a == 1` and `b == 3` you want to end up with `a == 3` and `b == 1`.

Note that doing this doesn't work (you should make sure that you understand why):

```
a = b
b = a
```

2. A simple for loop

Recall that a code that looks like this will print `"Hello"` 10 times:

```
for i in range(10):
    print("Hello")
```

Write a function called `verbose` that takes two arguments, `n` and `s`, where the first is an integer and the second a string, and prints the string `s` for `n` times.

For example, you want to have:

```
In [1]: verbose(3, "Bye")
Bye
Bye
Bye
```

Observe what happens in special cases, when `n` is `0` or negative.

3. Print the elements of a list

You are given a list, call it `l`. You can obtain the length of a list by using the built-in Python function `len`, like this:

```
n = len(l)
```

Also remember that you can access the elements of a list with the syntax `l[i]` as long as `0 <= i < len(l)`.

Also remember that, in a `for` loop, the iteration variable changes value at each loop, e.g. this code prints the square of all the numbers between `0` and `9`:

```
for i in range(10):
    print(i**2)
```

Write a function called `print_list` that takes one argument, a list `l`, and prints each element of the list.

For example, you want to get:

```
In [1]: print_list(["Do", "you", "like", "our", "owl"])
Do
you
like
our
owl
```

4. Sum the elements of a list

This time, we assume that we have been given a list `l` of numbers. We want to compute its sum.

Write a function called `sum_list` that takes one argument, a list `l`, and **returns** the sum of its elements.

For example, you want to get:

```
In [1]: sum_list([1, 3, 5, 2])
Out[1]: 11

In [2]: sum_list([-1, 3, 8])
Out[2]: 10

In [3]: sum_list([])
Out[3]: 0
```

Notice the last example: it's the sum of an empty list (which in Python is simply written as `[]`), and we want it to be `0`. My guess is that, if you manage to write a code that works correctly in the other cases, it will automatically return `0` for empty lists, without you having to do anything special about it.

Tips:

1. The resulting code should be very similar in spirit to the one that we saw in class about finding the maximum of a list. Try solving the problem with this tip alone without lokking at the others...

2. Use a value initialized at `0` and accumulate the elements of the list one at a time using a `for` loop
3. Recall that returning is not the same as printing!

5. Change signs

Again, you are given a list of numbers. We want to change the sign of all its elements.

Recall that you can set the value of an element in a list `l` by using the syntax `l[i] = x` as long as `0 <= i < len(l)`.

Write a function called `change_signs` that takes one argument, a list `l`, and changes all its signs. Then **return** the list.

For example, we want to get:

```
In [1]: l = [1, -3, -1, 0]

In [2]: change_signs(l)
Out[2]: [-1, 3, 1, 0]

In [3]: l
Out[3]: [-1, 3, 1, 0]
```

Observe that the contents of the list `l` have been modified by the function. Make sure that you understand why that happens by looking at the notes on how variables and values work, and how arguments are passed to functions.

In order to be even more sure that you understand this important point, consider this function, that works on a single scalar value instead:

```
def test(x):
    x = -x
    return x
```

In this case, if you try doing the same thing as above you get:

```
In [1]: a = 3

In [2]: test(a)
Out[2]: -3

In [3]: a
Out[3]: 3
```

Make sure that you clearly understand why the last line does not print `-3`.

6. Change odd signs

Same as above, but this time we only want to change the signs of the odd elements in the list, meaning: only of those elements that are in an odd *position*, not those that have an odd *value*.

Call your function `change_signs_odd`. As before, return the list from the function.

Note: in this exercise and in all the exercises that follow, we want the function to modify the list that is being passed as an argument and to return the same list, always.

For example, we want to get:

```
In [1]: change_signs_odd([1, -3, -1, 0, 2, 2, -8, 12])
Out[1]: [1, 3, -1, 0, 2, -2, -8, -12]
```

Notice how the function affected the values at indices `1`, `3`, `5` and `7`.

What happens if you pass an empty list? (Note: in Python, an empty list is simply written like this: `[]`.)

Tip: a number is odd if the reminder of the division by `2` is `1`, and it is even if the remainder is `0`. Use the appropriate Python operator, and use a conditional expression *inside* the loop.

7. Rotate a list

We start getting slightly more advanced now.

This time, we are given a list that we want to "rotate", meaning that we want to put the first element at the end, and shift all the other elements down by one position. See the example below.

Call your function `rotate`, let it take one argument, a list `l`, and return it when you're done.

For example, you want to get:

```
In [1]: rotate([1, 2, 3, 4])
Out[1]: [2, 3, 4, 1]

In [2]: rotate([3, 5, 8, -1, 2, 0])
Out[2]: [5, 8, -1, 2, 0, 3]

In [3]: rotate([1])
Out[3]: [1]

In [4]: rotate([])
Out[4]: []
```

Observe the last two examples, as they are particularly important: they are "corner cases". My guess is that when you will first manage to solve correctly the first three cases, your code will still have some problems in the last case. You need to use a so-called "special case" to deal with that: use a conditional at the beginning of your function.

Tips:

1. Do not worry about the special case at first, deal with it when you have done the rest.
2. Recall that the first element of a list has index `0` and the last one hast index `len(l)-1` (unless the list is empty!!)
3. An empty list is recognized by the fact that its length is `0`.

4. Here is the general scheme that you should use (for non empty lists!) (don't read this if you want to come up with a solution by yourself): save the first element in an auxiliary variable `x`; do the shift one element at a time (but stop just before the last elment!); put back `x` (what used to be the first element) at the end of the list.
5. For the empty-list case, you don't actually need to do anything: just make sure that the list is returned at the end.

8. Back-rotate a list

Same as before, but we want to do the rotation in the other direction: put the last element at the front, and shift all the rest up by one position.

Call your function `backrotate`.

This is trickier because now you want to operate on the list starting from the end, instead of the beginning. (Without using Python's negative indexing!!)

If things work, then for any list `l` you should have that `rotate(backrotate(l))` and `backrotate(rotate(l))` return the list as it was, i.e. that `backrotate` and `rotate` are inverses one of the other.

For example, you want to get:

```
In [1]: backrotate([1, 2, 3, 4])
Out[1]: [4, 1, 2, 3]

In [2]: backrotate([3, 5, 8, -1, 2, 0])
Out[2]: [0, 3, 5, 8, -1, 2]

In [3]: backroate([1]))
Out[3]: [1]

In [4]: backrotate([])
Out[4]: []
```

Tips:

1. The last element of the list is at index `len(l)-1`. The penultimate is at index `len(l)-2`. Then `len(l)-3`. Therefore, `len(l)-1-i` walks the list in reverse if `i` ranges between `0` and `len(l)-1`, as in `range(len(l))`.
2. Alternatively, you can get a reversed range by using a "step" of `-1` instead of the default, which is `1`. Read the Python documentation for `range` if you want to try this. (This is also mentioned in the slides.)

9. Alternative, inefficient, back-rotate

Observe that, if a list has length `n`, then rotating it `n` times returns the starting list. This is the same as rotating it `n-1` times and then once more. This means that the operation "rotate the list `n-1` times" is the inverse operation than `rotate`. Therefore, since we had observed earlier that `backrotate` is the inverse of `rotate`, it means that we can implement `backrotate` by just calling `rotate` for `n-1` times. This is quite inefficient computationally (you can convince yourselves about that by analyzing its computational complexity and finding out that it is quadratic rather than linear), but it should still work.

Write a function `backrotate2` that does that. **Important:** Make sure that this function also works with empty lists! Compare the results with the previous one. Observe that, for small enough lists such as those that you can eneter by hand, even the slow inefficient version is basically instantaneous on a human time scale.