Simulating Language: Lab 1 Worksheet
Example code

In the worksheet for lab 1, under section 8, we asked you to try to write some functions. I have given some examples of the kind of functions I was expecting you to come up with below. Note that for all of these problems you could achieve similar results in a different way, and the example code I have provided is by no means perfect - for instance, all the functions that take a list will probably go wrong if you pass them an empty list. But they should hopefully give you an idea of what a working function looks like. On pages 3-8 I go through each of those functions line by line, providing detailed comments on what I am doing and why - if you don't understand what a particular piece of code below does, go to the line-by-line walk-through and find out.

Define your own functions and test them to check that they return what you expect.

1. a function which calculates and returns the product of two numbers;

```
In [1]: def product(x,y):
   ...:     return x * y

In [2]: product(3,4)
Out[2]: 12
```

2. a function which returns the first item in a list;

```
In [3]: def first_in_list(alist):
   ...:     return alist[0]

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

3. a function which returns the last item in a list;

```
In [5]: def last_in_list(alist):
   ...:     return alist[len(alist)-1]

In [6]: last_in_list([1,2,3,4])
Out[6]: 4
```

4. a function which takes a list and prints out the square of each value in the list in turn;

```
In [7]: def square_list(alist):
   ...:     for x in alist:
   ...:         print x*x

In [8]: square_list([1,2,3,4])
1
4
9
16
```

5. a function which returns the largest number in a list.

```
In [9]: def max_in_list(alist):
   ...:     current_max = alist[0]
   ...:     for x in alist[1:]:
   ...:         if x>current_max:
   ...:             current_max=x
   ...:     return current_max

In [10]: max_in_list([1,2,3,4])
Out[10]: 4

In [11]: max_in_list([9,8,7,6])
Out[11]: 9

In [12]: max_in_list([1,1,100,3])
Out[12]: 100
```

Walk-through

1. a function which calculates and returns the product of two numbers;
I'll go through this one in excruciating detail.

```
In [1]: def product(x,y):
```

Every function definition starts something like this. We use **def** to tell Python we are about to provide it with a definition of a function. We then give the function name: in this case I chose `product`, since that seems like a sensible name for a function that is going to return the product of two numbers, but I could have called it anything. Then, between brackets `(...)`, I give the arguments that this function will take: in this case, we are going to pass two numbers to the function (which it will then multiply), so it has two arguments. I decided to call them `x` and `y`, but again I could have called them anything I wanted: `val1` and `val2`, `jack` and `jill`, whatever. Finally, we have a colon **:**. The body of the function - the stuff that actually happens when we use this function - follows this colon.

```
   ...:     return x * y
```

This is the body of the function - since this function is so simple, it's just one line. Notice that, if you type this in to Canopy, it automatically indents for you after the colon at the end of the previous line: the body of the function is a block of code, and blocks of code are indicated by indentation.

We use **return** to indicate that we want this function to return some value - we are going to pass two values in to this function, we want it to pass us a single value back (namely, the product of the two numbers we passed in), and **return** is how we do this. Then we have an expression which tells it what to return: in this case, `x * y`, i.e. the product of the two numbers we passed in to the function as arguments. So when we actually use this function it will multiply whatever two numbers we give it, and return (pass back) the product of those numbers.

Remember that, when you type the second line of the function into Canopy and hit RETURN, it knows that you might want to add some more stuff to the body of the function (i.e. you might want the function to do some other stuff), and therefore it keeps you at the same level of indentation for adding more code to this code block. In order to signal to it that you have finished with that code block, you hit RETURN again, and it closes off that code block and takes you back to the prompt.

```
In [2]: product(3,4)
Out[2]: 12
```

Now we are using the function we have just written. We want to know what the product of 3 and 4 is, so we type the function name `product`, then in brackets provide the two arguments that product requires, i.e. the two numbers to be multiplied. Because the function does what it should do, the function returns the product of these two numbers (12) , and we see the value returned by the function on the next line.

2. a function which returns the first item in a list;

```
In [3]: def first_in_list(alist):
```

Again, we start with our function definition: in this case, I have decided to call the function `first_in_list`, and I have called the single argument that this function takes `alist`: we pass a list to the function, what we want the function to do is pass back the first element from that list.

```
   ...:     return alist[len(alist)-1]
```

Again, Canopy automatically indents the next line for us, because it knows that every **def** line is followed by a block of code (the body of the function, the bit of code that does the stuff the function is supposed to do). Once again, all we want to do is return something, so we have a **return** statement followed by the thing to be returned. In this case, the thing to be returned is `alist[0]`. This is the 0th element from the list called `alist`, in other words the “first” item from the list we passed in to the function.

Again, when we hit RETURN after typing in this line, Canopy keeps us indented in case we want to add some more code to the body of the function: we don’t, so we hit return again to get back to the prompt.

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

Now we are using our function: we type the function name, and pass it the list `[1,2,3,4]` as its argument. It returns the value `1`, which is the first element in the list - hurrah, it looks like it works.

3. a function which returns the last item in a list;

```
In [5]: def last_in_list(alist):
```

Once again, we start of with our **def** line: just like `first_in_list`, `last_in_list` takes a single argument, which should be the list we want to know the last member of.

```
   ...:     return alist[len(alist)-1]
```

Just like first_in_list, we are going to return some item from the list we are calling alist, so we have a **return** statement followed by the thing we want to return. Just like in first_in_list, we are returning some item from alist, so we have alist[...]. The interesting stuff with this function happens inside the square brackets. We want the last item from alist, and to get it we are using the len function.

len is one of the built-in functions that is automatically defined when you start Canopy. If this was a programming course, we'd ask you to define your own len function, for fun, but basic functions like this are almost always provided for you, so we won't bother with that. len(alist) will return the length of alist: so if alist has 1 element, len(alist) will return 1, if it has 4 elements, len(alist) will return 4, and so on.

We subtract 1 from the number provided by len(alist): that's what len(alist)-1 means. This is the position in the list we want: if alist only has one thing in it, it has length 1 and the last element is the 0th element in that list (which is len(alist)-1); if alist has 4 things in it, it has length 4 and the last element in that list has index 3 (which is len(alist)-1). If this seems a bit confusing, remember that the "first" item in a list has index 0.

So to recap: this line of code returns the item with index len(alist)-1 from alist, which should be the last item in that list.

```
In [6]: last_in_list([1,2,3,4])
Out[6]: 4
```

We test it out and ask it to tell us the last item from [1,2,3,4], our function returns 4 - again, looks like it works.

4. a function which takes a list and prints out the square of each value in the list in turn;
This function uses a **for** loop to work through the list.

```
In [7]: def square_list(alist):
```

A fairly standard opening line: again, this function takes a single argument.

```
   ...:     for x in alist:
```

Now we have the body of the function, which is a new code block and therefore indented. This is a **for** loop: it will work through alist, taking each element in turn from that list(starting with the 0th, then the 1st, etc), temporarily calling that element x (although we could have used some other variable - we could have called each element i, or n, or `darling`), and does something with that `x`. The thing that it actually does with each `x` from the list is given in the next line.

```
   ...:         print x*x
```

Notice first that Canopy has automatically indented this line even further: the previous line setting up the for loop is followed by a new code block (the code block we are going to execute for each `x` in `alist`), and we identify the start and end of that code block using indenting. In this case, all we are doing with `x` is printing `x` out times itself: this is what `print x*x` achieves.

```
In [8]: square_list([1,2,3,4])
1
4
9
16
```

We test our new function, it behaves as expected.

5. a function which returns the largest number in a list.
This function is actually quite complex. Before I go through the code, I'll explain the basic procedure that my code uses. Often, if you want to write a function that does something interesting, you should start by working out a step-by-step procedure for how this thing is to be achieved - once you know what you are doing, the code should be easy(ish!) to write, but if you try to write the code without really knowing what you are trying to do you'll just end up in a mess.

In this case, my general idea is that I am going to work through the list of numbers, left to right, and keep a note of the highest value I have encountered so far. I'll call this something like "current max". So as I go through the list I know what current max is. For each new element in the list, I compare it to current max. If it's greater than current max, then it becomes my new current max: I forget what the old current max was, and replace it with this new value I have just encountered. On the other hand, if this new number is not greater than current max, I don't have to do anything: I just forget about this new value, and just move on down the list, keeping current max as it was.

Finally, when I have worked through this entire list, I have to remember to return my current max: this is what the function is supposed to do, and by the time I have gone through the list I will definitely have encountered and remembered the highest value in that list, which I am calling current max, so I just return that.

I think that'll work. The only other thing I have to decide is what value of current max I should start with. I guess I could choose some extremely low value that's probably going to be lower than anything in the list, but that's a bit risky (what if I guess wrong?), so instead the sensible thing to do is take the very first item from the list as my initial highest value, then work down the rest of the list, checking to see if I can find anything higher.

OK, now I have a plan I can work through the code, explaining how it executes the procedure I have just outlined.

```
In [9]: def max_in_list(alist):
```

As usual, I start by naming my function and its argument.

```
   ...:     current_max = alist[0]
```

I introduce a new variable, which I call `current_max`. This is where I am storing the current maximum value that I have encountered. Initially I assign this variable the value `alist[0]`, i.e. the 0th element in the list I am working with.

```
   ...:     for x in alist[1:]:
```

Now we have a **for** loop, which is going to help me work through the list. Compare this line with the equivalent line in my `square_list` function: it basically looks the same, but instead of working through all of `alist`, I am going to work through `alist[1:]`. `alist[1:]` is the list of elements in `alist` from index 1 to the end - in other words everything in `alist` apart from element 0. Check back over the notes in the worksheet where this "splice" notation is explained to refresh your memory. It wouldn't actually matter if I just worked through all of `alist` (i.e. replaced this line of code with `for x in alist:`), but since I've already looked at element 0 in the list it seems a bit of a waste to look at it again.

So, to recap, we're going to take every element from the rest of the list in turn, call it `x`, and do something with it.

```
   ...:         if x>current_max:
```

Canopy has indented us a bit more: we are now inside the code block that is executed for every `x` in `alist[1:]`. And what we have is a conditional statement: so we compare the element `x` to `current_max`, and if `x` is greater than `current_max` we do whatever it says in the next code block.

```
   ...:             current_max=x
```

And what we do, if our condition is met, is overwrite `current_max` with `x`: so if the value we are currently considering (which we are calling `x`) is greater than the highest value we have encountered so far (which we are calling `current_max`), then we store `x` as our new `current_max` (and forget whatever the old value of `current_max` was).

```
   ...:     return current_max
```

The indenting at this point gets interesting: we have actually come out **two** levels, which means we are out of the body of the **if** conditional, and also out of the body of the **for** loop: so in other words, this line of code is executed once the for loop has completely finished. This line of code simply states that we return the value of `current_max` that we have ended up with after working through the list using the for loop. Remember, this was the plan: work through the list, keep a note of the maximum value encountered, and then when we have gone all the way through the list we return that value.

```
In [10]: max_in_list([1,2,3,4])
Out[10]: 4

In [11]: max_in_list([9,8,7,6])
Out[11]: 9

In [12]: max_in_list([1,1,100,3])
Out[12]: 100
```

We test our function, it works. Notice that I have done a few checks here - I am pretty confident the code is right, because I planned it carefully before I wrote it and I know what every line does, but nonetheless I am going to check that it can definitely find the maximum value even if it is first in the list, or last, or buried in the middle.