Simulating Language: Lab 1 Worksheet

1. Python and IDLE

During this course we will be using a simplified subset of the programming language **Python** to build and run the simulations. We're using Python because its code is concise and clear, and the meaning of the code is transparent. It's a simplified subset, because we want to avoid using Python-specific features as much as possible, so that you could (if you wanted) take our code and translate it into another programming language without difficulty.

Python is available on the computers in the lab, and is freely available for you to download and install on your own computers as well: we are using the Enthought Python Distribution, which you can download by following the link on the course website. The easiest way to run Python is through a development environment like **IDLE**, which you will find on the Windows computers in the lab under **Start-All Programs-Enthought-IDLE**. Find IDLE on your computer and start it, then use it to enter the examples in this document.

This document provides a brief summary of the basic features of Python which we will need to start this course. These will be supplemented with other notes as we need them to run the simulations.

It's important to become comfortable with using Python as quickly as you can, so make sure that you understand everything in this document.

2. Evaluation

Python is an interactive language. The prompt (>>>) indicates that Python is waiting for some input from the user. You can use Python just like a calculator, using the familiar mathematical operators (`+ - * /`) to build an expression. When you press ENTER, Python **evaluates** the expression you entered, and prints the result.

Within IDLE, you can use **ALT-P** (hold down the 'alt' key, press 'p') to access previous commands you have typed.

```
>>> 1 + 3
4
>>> 7 - 5
2
>>> 2 * 3 - 5
1
>>> 2 * (3 - 5)
-4
```

Enter a few expressions at the prompt like those above, and check that the results are what you expect. Make sure you understand how bracketing expressions affects their evaluation.

```
>>> 7 / 3
2
>>> 9 / 2
4
```

The numbers we've used so far are all **integers**, which means that apparently strange things can happen with division: if the first integer is not exactly divisible by the second integer, then the result is truncated.

To avoid this, we can use **floating-point** numbers, which are represented in standard decimal or scientific notation. It isn't necessary to include a zero after the decimal point. Inevitably, floating-point numbers can only be stored to a certain level of precision (depending on your computer), so you will occasionally encounter rounding errors like the one in the example.

```
>>> 9.0 / 2.0
4.5
>>> 9. / 2.
4.5
>>> 7.0 / 3.0
2.3333333333333335
```

It's important to be aware of the difference between integers and floating-point numbers in your simulations!

3. Variables

We use **variables** to store and manipulate data. Variable names begin with an alphabetic character, with subsequent characters being letters, numbers, or the underscore (_). Python is case-sensitive, so `candidates` and `Candidates` are the names of **different** variables (to avoid confusion, it's a good idea to use a standard style for variable names in your programs).

To assign a value to a variable, use the equals sign = followed by the value. The same value can be assigned to multiple variables by separating them each with the equals sign. (Note: unlike in some other programming languages, variables are automatically declared when first assigned a value, so there is no need for separate variable declaration.)

```
>>> x = 2
>>> y = 1
>>> a = b = 0
>>> x
2
>>> z = y + x
>>> z
3
```

```
>>> x = x + 1
>>> x
3
>>> x += 3
>>> x
6
>>> x / 2
3
>>> x
6
```

On the left, we see two ways to change the value of a variable based on its existing value: the statement `x = x + 1` adds one to the original value of `x`, and stores the result in `x`, overwriting the original value. The compound operator += is a useful shorthand for this behaviour, and has exactly the same effect.

Python does not give any response to a variable assignment, because nothing is evaluated. To check the value of a variable, type its name at the prompt, and Python evaluates it as an expression.

4. Lists

Often, we want to store multiple related values in a single data structure, and lists are the most basic way to accomplish this. A list is enclosed in square brackets [ ], with commas separating the individual elements of the list.

```
>>> mylist = [1, 2, 3, 4]
>>> mylist
[1, 2, 3, 4]
>>> mylist[0]
1
>>> mylist[1] = 5
>>> mylist
[1, 5, 3, 4]
>>> mylist[3]
4
```

These individual elements can be accessed and assigned new values by using the index operator [n], where [n] refers to the nth element in the list. Note that in Python, indexes start from **zero, not one** (i.e. the 'first' element in a list has index 0).

```
>>> mylist[0:2]
[1, 5]
>>> mylist [:2]
[1, 5]
>>> mylist[1:]
[5, 3, 4]
```

A sequential subset of the list can be obtained by using the slicing operator [m:n], which returns all items starting from the mth element, and up to (but **not including**) the nth element. Either m or n can be omitted, in which case the beginning (or end, respectively) of the list is assumed in place of the missing value.

```
>>> mylist
[1, 5, 3, 4]
>>> len(mylist)
4
```

The function **len(list)** returns the number of items in the list, or its length. For more details about how functions are used, see section 8 below.

```
>>> mylist
[1, 5, 3, 4]
>>> mylist.append(6)
>>> mylist
[1, 5, 3, 4, 6]
>>> mylist.remove(3)
>>> mylist
[1, 5, 4, 6]
```

Single items can be added to the end of an existing list using the list function **list.append(item)**. The list function **list.remove(item)** removes the first occurrence of item from list.

Lists are very important structures in Python, and it is worth spending some time to make sure you understand how they work and are comfortable with them.

1. Make a list of the first four even numbers (2,4,6,8) called evens and check that it is stored correctly.
2. Check the length of evens using the len function.
3. What happens when you evaluate evens[4]? Why?
4. Extract the middle two numbers from evens.
5. Append the number 10 to the end of evens, and check that it is stored correctly.
6. Create a new list of integers and use it to verify that the remove function does indeed remove the first occurrence of an item from a list.

Lists can themselves also contain other lists, which provides an easy way to create complex and flexible data structures. In the example, the second element of `complex_list` is itself a list. To access the individual items of this sub-list, just use the index operator again, as shown. In principle, there is no limit to the nesting of lists.

```
>>> complex_list = [1, [2, 3], 4]
>>> complex_list[0]
1
>>> complex_list[1]
[2, 3]
>>> complex_list[1][1]
3
```

```
>>> lista = [1, 2, 3, 4]
>>> listb = [5, 6, 7]
>>> listc = [lista,listb]
```

How long is the list `listc` (left)? Make sure you understand why.

5. Code Blocks and Conditionals

Blocks of code in Python are identified by **indentation**, not by curly brackets or begin/end markers as in other languages. The beginning of a block is marked by an increase in indentation, and the end is marked by a return to the previous level of indentation; all the lines of code in the same block must therefore be indented at the same level. This makes Python code easy to read, but can be tricky to get used to at first. Within IDLE, some aspects of indentation are handled automatically, but when it comes to writing more complex programs you'll have to do indentation manually, using either tabs or multiple spaces. **Never** mix tabs and spaces in indentation - use either one or the other!

A conditional statement is used to execute a block of code only in certain circumstances (when the conditional expression is true); it is introduced by the word **if**, followed by the conditional expression, and a colon (**:**) to introduce the conditional code block. If the conditional expression is true, then the code block is executed.

Conditional expressions in Python use the comparison operators == (equal to), < (less than), <= (less than or equal to), > (greater than), >= (greater than or equal to), **!**= (not equal to), and they can be joined together into complex expression using **and**, **or** and **not**. Note particularly the operator equal to (**==**), which uses **two** equals signs; one equals sign assigns a value to a variable!

In this example, the conditional expression is `x == 0`. If this statement is true, then the indented code block below is executed. When typing this example, note that IDLE will helpfully indent the next line after the conditional expression automatically, as it knows that another indented code block must follow. Also note that you will need to press ENTER twice in order to get back to the prompt after typing the final line of code, because of the indentation; after the first ENTER, IDLE assumes that you might want to add some more code to the conditional code block.

```
>>> x = 0
>>> if x == 0:
        y = 1

>>> y
1
```

```
>>> x = 4
>>> y = 3
>>> if x <= y:
        z = 1
else:
        z = 2

>>> z
2
```

You can also use an **else:** statement, followed by another code block which is executed when the conditional expression is **false**, as shown on the left. After pressing ENTER after `z = 1`, IDLE again assumes that you might want to add some more code, so you will need to press DELETE in order to put the **else:** statement on the following line at the right indentation. Again, IDLE will helpfully indent the next line after the **else:** statement automatically, as it knows that another indented code block must follow.

6. For-Loops

The **for** loop can be used to run one code block repeatedly for a set of elements. In the example, each of the elements in the list `[1,3,5]` is taken in turn, its value is assigned to the variable n, and then the following code block is executed with this value of n (in this case, print each value multiplied by 2).

```
>>> for n in [1, 3, 5]:
        print n * 2

2
6
10
```

The **print** statement is used to print data to the screen; the statement is followed by details of the variables you want to be printed.

```
>>> range(1,4)
[1, 2, 3]
>>> range(-1,2)
[-1, 0, 1]
>>> range(3)
[0, 1, 2]
>>> for n in range(3):
        print n, n + 3
0 3
1 4
2 5
```

Because the for loop uses a list of elements to iterate over, it does not act exactly like a traditional counting loop would. In order to simulate a counting loop, we need to generate an appropriate list of numbers. The **range(x,y)** function is useful for this; it creates a list of numbers from x up to (**but not including**) y. If x is omitted then a start value of zero is helpfully assumed. A list generated by range can be used directly in a for loop, as in the example.

7. Random Numbers

In simulations, we frequently need access to random numbers; Python helpfully provides a built-in random number generator for this purpose, in a special **module**, or self-contained file. To load the **random** module containing the generator, we **import** it into Python. If you try to access the random number generator before it is imported, an error will result.

```
>>> import random
>>> list = [1, 2, 3, 4]
>>> random.choice(list)
3
>>> random.choice(list)
1
>>> random.randrange(10)
7
>>> random.random()
0.78351913714757648
```

Once the random module is imported, there are a number of different ways to access different kinds of random numbers:

- **random.choice(seq)** returns a random element from a given sequence;
- **random.randrange(n)** generates a sequence of numbers, very like range(), and then returns a random element from this generated sequence;
- **random.random()** produces a random floating-point number between 0 and 1 (including 0, but not including 1).

8. Functions

Functions are the building blocks of most programming languages, and we will use them extensively in this course; their purpose is to execute a small portion of code with a well-defined, specific function, before returning to the main program. It is a very good idea to use functions in your programs, as they can help greatly in reducing duplication of code, allowing the decomposition of a program into simpler steps, and hiding unnecessary detail from users.

To call a function, we simply type its name, together with the names of any arguments or parameters between following brackets. We have seen a number of functions already, including:

(i) the **len** function, which took the name of a list as its argument, and returned the number of items in that list;
(ii) the **range** function, which can take a single number as an argument, and return a list of numbers from 0 up to but not including that number;
(iii) the functions in the **random** module which returned different kinds of random number.

In Python, functions are defined using the **def** keyword, followed by the name of the function, the **arguments** it takes in brackets, and a colon (:) to introduce the body of the function in a code block, indented as before.

```
>>> def square(x):
        return x * x

>>> square(5)
25
>>> square(9)
81
```

Functions usually return a value, specified by the **return** statement followed by the expression which is evaluated and returned. In the example, the **square** function is **def**ined with a single argument x, and it simply returns the value of this argument multiplied by itself.

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;*
2. *a function which returns the first item in a list;*
3. *a function which returns the last item in a list;*
4. *a function which takes a list and prints out the square of each value in the list in turn;*
5. *a function which returns the largest number in a list.*

9. Plotting

We frequently want to plot the results of our simulations - to include figures in reports, but also because visualisation is often the best way to understand what is happening in a simulation. The Enthought Python Distribution comes with a library called matplotlib, which makes plotting various kinds of graphs easy.

As with the random module, we have to import a module, which in this case is called **matplotlib.pyplot**. Typing out this full module name every time we want to use a function from the module would be cumbersome, so we will give it the shorthand name of **plt** when we import it, by using **import ... as**.

There are three stages to producing a plot using this module:

1. Set up the plot, using **plt.plot(x_values_list, y_values_list).**
2. Label the axes, using **plt.xlabel(text)** and **plt.ylabel(text)**.
3. Display the plot, using **plt.show()**.

```
>>> import matplotlib.pyplot as plt
>>> x_vals = range(-1,3)
>>> y_vals =  [1, 2, 3, 4]
>>> plt.plot(x_vals, y_vals)
>>> plt.xlabel(“the x axis label”)
>>> plt.ylabel(“the y axis label”)
>>> plt.show()
```

A couple of notes:

Some of these commands return a value before going back to the Python prompt, which I have omitted from the example code above. Don’t worry about these.

Once you have displayed your plot, you have to close the plot window to get the Python prompt back.

If you don’t provide x values to plt.plot (right), matplotlib will generate them for you automatically: there will be as many x values as y values, the first value being 0.

```
>>> plt.plot(y_vals)
>>> plt.show()
```

```
>>> neg_y_vals =  [-1, -2, -3, -4]
>>> plt.plot(x_vals, y_vals,
          x_vals, neg_y_vals)
>>> plt.show()
```

You can plot multiple lines on the same graph by providing plt.plot with a series of x and y values (see left).

Once the plot window is displayed you can do several useful things including panning around the plot and zooming in (select the compass icon, left-click to pan, right-click to zoom) and saving the plot.

Experiment with plotting, panning, zooming and saving plots as image files.