Lab 2: Lists, Loops and Animation

One of the great things about computers is their ability to repeat tasks quickly, accurately and without getting bored. We will take advantage of this capability many times during this course. This lab will introduce loops, which perform such repetition, and use them to create animations. First, however, we’ll take a look at some of the different kinds of objects SageMath works with.

Types of Things

In the previous lab, you worked with numbers and functions. You also made plots, and in the process of making them, you encountered words enclosed in quotation marks. For example, when you want to make a plot red, you have to put the word “red” in quotation marks. Such words are called strings, which is short for “character strings”. Strings allow computers to handle words, phrases, and typographic symbols.

Strings can include numbers, not just letters. But when numbers are treated as strings, they act quite differently from regular numbers.

Exercise 1. Enter the following code into SageMath and compare the outputs.

```
>>>a=5
>>>show(a)
>>>b="5"
>>>show(b)
```

Exercise 2. What happens when you add 1 to a? To b?

In the above exercises, the variables a and b look the same when displayed using show but act very differently when used in an arithmetical calculation. This happens because a is the integer 5 while b is the character string “5”. We say these variables have different types, which just means they’re different kinds of things. The type of a is “integer”, while the type of b is “string”.

This explains what you saw in Exercise 2. Adding 1 to an integer is not a problem, but adding 1 to a character string makes no sense and results in an error.

Actually, addition is defined for strings in SageMath. Here’s how it works.

```
>>"5"+"1"
```
You can see that the + symbol (or “operator”) still embodies the idea of “putting things together”. However, “putting things together” means something different for integers than for character strings, so the exact meaning of + changes depending on the types of objects it’s acting on. (Programmers say this makes + an overloaded operator.)

**Exercise 3.** Give another example of a type of SageMath object for which addition is defined and explain what addition means for that type of data.

**Remark.** There is a little bit of vocabulary you should be aware of, as you may see it online and in the SageMath documentation. While the fundamental idea to understand is that different types of objects behave differently even though they may look alike, programmers often use the word “type” for simple objects like strings and numbers and “class” for more complex ones like graphs. Just think “type” when you see “class” and you’ll be fine.

You can find out what the type of an object is using the `type` command.

```python
>>> type(a)
<type 'sage.rings.integer.Integer'>
```

This output means that `a` is a SageMath integer.

```python
>>> type(b)
<type 'str'>
```

This output means that `b` is a character string.

**Exercise 4.** Find the types of the number 0.2 and the graph of \( f(x) = x^2 \).

**Lists**

Scientific data and the outputs of model simulations often come in the form of lists of numbers. SageMath gives us many tools for working with such lists and tables.

You make a list by enclosing its elements, separated by commas, in square brackets:

```python
["Bacteria", "Protists", "Plants", "Fungi", "Animals"]
[2,3,5,7,11,13]
```

Each element of a list can be accessed by its position in the list, typically called its index. In Python, indexing starts with 0, so the first element of a list with \( k \) elements has index 0 and the last element has index \( k - 1 \).
Example 1. Enter the list of biological kingdoms into SageMath and call it ```kingdoms```.
```python
>> kingdoms = ["Bacteria", "Protists", "Plants", "Fungi", "Animals"]
```
To access the first element of this list, enter:
```python
>> kingdoms[0]
'Bacteria'
```

Exercise 5. A bacteria population is doubling every hour. Its sizes at different times are 100, 200, 400 and 800. Make a list of these values.

```
[100, 200, 400, 800]
```

Exercise 6. Assign the list of bacteria population sizes to the variable `bacteria`.

Exercise 7. Find the type of the variable `bacteria` from the previous exercise.


You can add an element to the end of a list using ```listname.append(element)```.
(The generic names `listname` or `list` are just placeholders for the real name of your list.) For instance if you wanted to add the string "Archaea" to the list named `kingdoms`, the code would look something like this:
```python
>> kingdoms.append("Archaea")
```
Note that the above code does not output anything to the screen. This is because the `append()` function only tells the computer to save its input to specified list. To see the result, we would have to type `kingdoms` and evaluate the cell.

Exercise 9. Append the number 1600 to `bacteria` and call it to display its value. **Don't paste or retype any output.**

```
[100, 200, 400, 800, 1600]
```

Exercise 10. What is the next value of the population? Append it to the list.

Exercise 11. What would happen in the example above if we did `kingdoms.append("Archaea")` twice before viewing `kingdoms`? Try this out and explain why you got the result that you did.

Plotting lists

To plot the entries in a list, use the `list_plot` function. If you give this function a single list of numbers as an input, it will plot each number against its position in the list. For example, the command `list_plot(bacteria)` plots the list of
population sizes you just created in Example 6, producing the graph below.

Notice that the $x$-coordinate of the first point is 0, not 1. This happens because SageMath starts counting at zero, so the index of the first element of a list is 0.

**Exercise 12.** Plot the list $[3,5,7,9,11]$.

Often, we will need to plot lists of points. For example, suppose we have the points $(1,2)$, $(2,1)$, $(3,4)$, and $(4,3)$, with the first number in the ordered pair an $x$-coordinate and the second a $y$-coordinate. How do we plot these points in SageMath?

First, we enter the list of points:

```python
>>> g = [(1,2), (2,1), (3,4), (4,3)]
```

Then, we use the `list_plot` function to produce Figure 1a:

```python
>>> list_plot(g)
```

This plot is technically correct, but the points are a little hard to see. To make them more noticeable, we might color them red and change their size. The command `list_plot(g, color="red", size=30)` produces Figure 1b on the following page.
Exercise 13. Define your own list of pairs of values and plot it. Make sure your plot is legible.
Whether you’re plotting a list of numbers using `list_plot()` or a mathematical function using `plot()`, you can label the axes of the plot using the `axes_labels` plotting option.

**Example 2.** Adding axes labels to our plot of bacteria population sizes.

```python
list_plot(bacteria, axes_labels=['time', 'population'])
```

Notice that `axes_labels` is a variable that we set equal to a list of labels. In SageMath, square brackets always mean that a list is involved.

Another feature of `list_plot` is the ability to connect the points of the list together. This is accomplished by using the `plotjoined` option:

**Example 3.** Making the bacteria graph plot joined and adding axes labels:

```python
list_plot(bacteria, axes_labels=['time', 'population'], plotjoined=True)
```

Note that the value of `plotjoined` is either `True` or `False`. This type of data is called a `boolean`. You will learn more about this data type in the future.

Now that we know how to label axes and join points, we should do so whenever it is reasonable to do so. We should join points whenever we want to see a curve. Labeling axes is a good way to keep track of which values correspond to which axes, especially when we plot lists against each other.
However, `list_plot` requires a list of points, not two lists of numbers, as input. To avoid typing long lists of points and all the required parentheses by hand, we turn to the function `zip`. This function takes two lists and turns them into a list of ordered pairs. (It can also take more than two lists and turn them into a list of n-tuples.) For example:

```python
>>> zip([1,2,3], [4,5,6])
[(1, 4), (2, 5), (3, 6)]
>>> zip([1,2,3], [4,5,6], [7,8,9])
[(1, 4, 7), (2, 5, 8), (3, 6, 9)]
```

This is the kind of input that `list_plot` needs. It's common to nest the `zip` command inside the `list_plot` command, as below.

```python
>>> list_plot(zip([1,2,3], [4,5,6]))
```

This means the same thing as:

```python
>>> pairs = zip([1,2,3], [4,5,6])
>>> list_plot(pairs)
```

Exercise 14. You are studying populations of hippos and crocodiles in a river in Africa. Over five years, the hippo population at your study site has been 62, 81, 75, 90 and 67. In the same years, the crocodile population has been 20, 34, 18, 25 and 31. Plot the system’s states in hippo-crocodile space, labeling your axes appropriately and making the points red.

The `zip` function is very helpful in plotting time series graphs. All you need to do is make a list of time values and zip it with the values of your state variable.
Exercise 15. The list in Exercise 10 gives the size of a population of bacteria at one-hour intervals. Since one hour is 1/24 of a day, create a list of time points for these observations with time in days. Then, plot a time series graph of the population.

Having developed some basic tools, we will now use them to work with real data. Your worksheet contains lists called wt5_time, wt5_heartrate and wt5_temp. These lists contain heart rate and body temperature data for a wild type (control) rat, measured over 72 hours as part of a real study of circadian rhythms.

Exercise 16. Plot time series of the rat’s heart rate and body temperature, using different colors. Make sure the plot uses the given time values, not just 0, 1, 2,... and that your axes are labeled.

Exercise 17. Compare the plots and describe any relationships you observe.

Exercise 18. Plot the data as a trajectory in temperature-heart rate space. Make sure to label your axes.

For loops

Suppose you wanted to print out a series of sentences listing your favorite foods. You might use the print command and write:

```python
print "Pizza is one of my favorite foods."
print "Chocolate is one of my favorite foods."
print "Green curry is one of my favorite foods."
```

This works, but it's rather tedious, especially if you like many kinds of food. A shortcut would be useful.

Looking at the example, you can see that the print command and the string " is one of my favorite foods." are the same in every line. The only thing that changes is the name of the food. It would be convenient if we could just make a list of the foods and insert them into the code one at a time.

We can do this using something called a for loop. One way to handle the foods example with a for loop is the following:

```python
favorites = ["Pizza", "Chocolate", "Green curry"]
for food in favorites:
    print food + " is one of my favorite foods."
```

In this example, food is a variable that takes on the value "Pizza" the first time the computer executes the statement print food + " is one of my favorite foods.", "Chocolate" the second time and "Green curry" the third time.
Exercise 19. Print out sentences listing five of your favorite books, songs or movies using a for loop.

More generally, a for loop in SageMath (or Python) has the form:

```python
for var in list:
    loop body
```

The loop body must be indented. After the loop body, go back to your previous level of indentation.

**Example 4.** A string is similar in most ways to a list. This loop will print each character in the word “dynamics” on a separate line.

```python
>>> for char in "dynamics":
    print char
d
y
a
m
i
c
s
```

*One difference is that the characters in a string can’t be changed by indexing.*

As you can see, the variable given immediately after the word `for` takes on the value of each list element in turn. First `char` was “d”, then it was “y”, then “n”, and so on. This is the key to how SageMath for loops work. The body of the loop is executed for each element in the list. The body stays the same, while the list element changes. When writing loops, think about what should stay the same and what should change.

**Example 5.** The following loop computes the base-10 logarithm of 10, 100, and 1000 in SageMath:

```python
>>> for val in [10, 100, 1000]:
    log(val, 10)
1
2
3
```

Exercise 20. Use a for loop to print your name vertically.

Exercise 21. Use a for loop to square the numbers 15, 27, 39 and 84.

225
729
1521
7056
Things to Do with Lists and Loops

Using loops to make lists

In the previous exercises, you used loops to output values. Often, it is useful to store these values in a list.

To start, you’ll need to create a list with no elements. (Think of this as tearing out a piece of paper and titling it “Groceries” when making a shopping list.) To make such an empty list, enter `listname = []`. Then, use `listname.append()` to add computed values to your list.

Example 6. This code makes a list of the first ten multiples of 2.

```python
>> mult2 = [] #Set up empty list
>> for n in [1,2,3,4,5,6,7,8,9,10]:
    > mult2.append(2*n) #Compute the n’th multiple of 2 and append to list
>> mult2 #Display the list
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

Exercise 22. Make a list containing the first five multiples of 3.

```
[3, 6, 9, 12, 15]
```

Exercise 23. Write a loop that makes a list of the squares of the numbers 0.1, 0.2, ..., 0.7. Then, plot your list.

```
[0.0100000000000000, 0.0400000000000000, 0.0900000000000000, 0.1600000000000000, 0.2500000000000000, 0.3600000000000000, 0.4900000000000000]
```

Exercise 24. Create a function and apply it to the numbers 0 through 5, inclusive. Plot the list of resulting values.

We can use for loops to process data.

Exercise 25. The time in `wt5_time` is measured in hours. Create another list in which it is given in minutes.

Exercise 26. Convert the temperatures in `wt5_temp` from Celsius to Fahrenheit. (The formula is \( F = \frac{9}{5}C + 32 \).)

Exercise 27. Plot a time series of your transformed data.

Exercise 28. Plot a trajectory of the transformed temperature data and the original heart rate data.
Animations

When investigating functions and models, it can be useful to animate their response to changes in parameters. Sage’s `animate` function allows us to easily produce such animations.

Animations are created by showing a series of still images one after the other, fast enough to create the illusion of motion. The `animate` function takes a list of plots as input and animates it.

**Example 7.** The following code shows how a change in the slope of a line affects the line’s appearance.

```python
plots = []  # Set up empty list to hold plots
slopes = [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5]  # Make a list of slopes
for m in slopes:  # For each m in slopes, create plot and add to list
    p = plot(m*x, (x,-10,10))
    plots.append(p)

a = animate(plots)  # Create the animation
show(a)  # Necessary to display the animation
```

Try this code now. The `show` command is necessary to view the animation; it can also be used with other graphics.

Oops! The code produces an animation all right, but the animation is useless because it’s the axes, not the line, that move. To stop this from happening, we can specify maximum and minimum values for $y$, fixing the $y$-axis in place.

**Example 8.** Fixing the $y$-axis in an animation

```python
plots = []  # Set up empty list to hold plots
slopes = [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5]  # Make a list of slopes
for m in slopes:  # For each m in slopes, create plot and add to list
    p = plot(m*x, (x,-10,10), ymin=-50, ymax=50)
    plots.append(p)

a = animate(plots)  # Create the animation
show(a)  # Necessary to display the animation
```

This code produces a useful animated plot.

**Exercise 29.** Change the animation in Example 8 to make the line green rather than blue.

**Exercise 30.** Change the previous animation to make the slope range from -3 to 3 in steps of 0.5.

**Exercise 31.** Rewrite the animation in Exercise 29 so that the slope of the line plotted is always 1 but the $y$-intercept ranges between -5 and 5.