A Couple of Useful Shortcuts

Writing mathematical functions

Previously, you learned to write functions using the keyword `def`. SageMath also provides a quick way to define mathematical functions. To do this, just give the name of the function, its independent variable or variables, and a formula defining the relationship between the dependent and independent variables. For example, the function \( f(x) = 3x^2 + 5 \) is entered into SageMath as `f(x)=3*x^2 + 5`. You can now use `f(x)` just as you would use a built-in SageMath function.

```plaintext
f(x) = 3*x^2 + 5
f(1)
8
log(f(20.0), 10.0)
3.08098704691089
```

Exercise 1. Let \( g(x) = 19x^3 \). Enter this function into SageMath and evaluate it for \( x = 0 \), \( x = 10 \) and \( x = -4 \).

Exercise 2. Define a mathematical function of your own choosing in SageMath and evaluate it at three points.

From now on, we will refer to these kinds of functions as math functions and the kind written with `def` as Python functions (because the syntax for defining them comes from Python, the programming language SageMath is based on).

Making lists automatically

Often, you will need to use lists that are much longer than what you could reasonably type by hand. Luckily, SageMath has a function called `srange` that will do this for you.\(^1\) The command `srange(n)` generates a list of numbers from 0 up to but not including \( n \), `srange(m,n)` generates a list of numbers from \( m \) to \( n \) (again, not including \( n \)), and `srange(m,n,step)` creates a list numbers from \( m \) to \( n \) in increments of \( \text{step} \).

```plaintext
srange(10)
```

\(^1\)The Python function `range` is usually faster but only works for integers.
If \( n \) is smaller than \( m \) and \( \text{step} \) is negative, \texttt{srange} will count backwards in increments of \texttt{step}.

\begin{verbatim}
>>> srange(10,1,-1)
[10, 9, 8, 7, 6, 5, 4, 3, 2]
\end{verbatim}

Note that \texttt{srange} generates a list, so there’s no need to enclose the command in square brackets to make a list.

**Exercise 3.** Generate a list containing all the integers between 1 and 100, inclusive. HINT: Make sure you haven’t accidentally made a list whose only element is another list.

**Exercise 4.** Generate a list containing the even integers between 0 and 50, inclusive.

\begin{verbatim}
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50]
\end{verbatim}

You can use the list of values generated by \texttt{srange}, like any other list, in a for loop.

**Example 1.** The following loop takes the odd numbers between 1 and 10 and doubles each of them.

\begin{verbatim}
>>> nums = srange(1,10,2)
>>> for n in nums:
...    2*n
2
6
10
14
18
\end{verbatim}

**Exercise 5.** Use a for loop and the \texttt{srange} function to print the integers between 0 and 10.

**Exercise 6.** Use a for loop and the \texttt{srange} function to compute the square roots of 10, 20, 30, and 40 as decimals.

Just as before, you can use \texttt{srange} and a for loop to make a new list.
Example 2. Make a list of the first five multiples of 3.

\[
\text{mult3} = [] \quad \text{# Create an empty list to hold the results.}
\]
\[
\text{nums} = \text{srange}(1, 6) \quad \text{# Why do we use 6?}
\]
\[
\text{for } n \text{ in nums:}
\]
\[
\text{mult3}.\text{append}(3*n)
\]
\[
\text{mult3}
\]
\[
[3, 6, 9, 12, 15]
\]

Exercise 7. What does the line `mult3.append(3*n)` do?

Exercise 8. Create a list of the first ten multiples of 4.

Exercise 9. Apply the function you defined in Exercise 2 to a list of at least five different values and make a list of the results.

Exercise 10. Plot the list you just made.

Iteration

One very powerful use of for loops is iteration — taking a value, applying a function to it, applying the same function to the resulting value, and continuing for as long as we want. For example, we can iterate the function \( f(x) = 2x \) with the initial value 1 as follows: 1, \( f(1) = 2 \), \( f(2) = 4 \), \( f(4) = 8 \), and so on. The key idea in iteration is that the output from one step becomes the input to the next step. You will use this concept many times in this course, in both coding and math.

Example 3. This code iterates the function \( f(x) = 2x \) ten times.

```python
# Define the function we want to iterate
f(x) = 2*x

# Create list of numbers defining how many times to iterate
count = srange(0, 10)

# val is the variable that will hold our intermediate calculations. We start with val being 3.
val = 3

for i in count:
    # Plug val into f and assign the result to val.
    a = f(val)
    val = a

# Show the final result
val
```
Exercise 11. What is the final result?

Exercise 12. Why is the line `val=a` necessary?

Exercise 13. Iterate $f(x) = 2x$ twenty times with 3 as the initial value and view the final result.

Exercise 14. Iterate $f(x) = 2x$ twenty times with 5 as the initial value and view the final result.

Exercise 15. Iterate $g(x) = 3x$ five times with 2 as the initial value.

Exercise 16. Rewrite Example 3 so that there is only one line in the loop body (not including the comment).

So far, we have only looked at the final result of the iteration. Usually, though, we want to keep the intermediate values, often for plotting. Thus, we will modify the iteration code above to store a list of results.

Example 4. This code iterates the function $f(x) = 2x$ ten times and keeps a list of the results.

```python
f(x) = 2*x  # Define the function we want to iterate
# Create list of numbers defining how many times to iterate
count = srange(0,10)
# Create a list to hold the results. The first value is 3.
vals = [3]
for i in count:
    a = vals[i]  # Grab the last item in the results list
    b = f(a)  # Plug a into the function. Put the output in b.
    vals.append(b)  # Put b into the list of results.
list_plot(vals)  # Plot the results
```

![Plot of iteration results](image)
The key to doing iteration this way is the correspondence between the value of \( i \) and the position of the last value in \( \text{vals} \). The first time the loop runs, \( i=0 \) and the last (and only) item in the list has position 0. We then append a number to \( \text{vals} \), making it one element longer. The next value of \( i \) is 1 and the position of the most recently appended value in \( \text{vals} \) is also 1. We again append a number to \( \text{vals} \), increasing its length by 1. The position of this new number is 2, as is the new value of \( i \). (For this reason, \( i \) is often called the *counting variable* or *counter.* ) The process continues until we reach the end of \( \text{count} \).

**Exercise 17.** Compare Examples 3 and 4 and explain the differences between them.

**Exercise 18.** Iterate \( 3x \) five times with 2 as the initial value, keeping all the intermediate values.

**Exercise 19.** Iterate \( \cos x \) 10 times with 0.5 as the initial value and plot the result. Does the same thing happen for other initial values?

**Exercise 20.** Iterate \( 2.5 \cos x \) 10 times with 0.5 as the initial value and plot the result. (You may want to use the plotting option `plotjoined=True` to see the behavior better.) Try several initial values. Do they act similarly? You will study this kind of unpredictable behavior, which is called chaos, later in LS 30.

Example 4 takes 8 lines of code to iterate a function and plot the results. This makes the process clear but is not how we would usually write such a program. Normally, we would make the code much more concise.

**Exercise 21.** Do the iteration and plotting in Example 4 using 7 lines of code. Comment every line to explain what it does.

**Exercise 22.** Now, do it in 6 lines. (HINT: Look for places where you can use an expression directly rather than assigning it to a variable and using the variable.)

**Exercise 23.** How short can you make this program? (Using semicolons to combine lines doesn’t count!) As before, comment every line.

**From Script to Function**

Despite all the abbreviation, iterating a function using the code you developed in the previous series of exercises means copying and modifying a hefty chunk of code. The kind of program you have, which consists of a series of commands for performing a particular task, is called a *script*. It’s often useful to turn a script into a function that can work for different inputs. If you don’t remember
how to define Python functions, now might be a good time to review that from Lab 1.

The following steps for turning scripts into functions are meant as helpful guidelines, not a lock-step procedure. You are free to follow them only in part or ignore them entirely. However, if you find yourself struggling to turn a script into a function, you should probably work through the process step by step.

1. Make sure you know exactly what you want the function to accomplish. Write this as a comment at the top of your code.

2. Identify the output the function will need to return. Write this as a comment.

3. Identify the inputs that the function will need. These will be the arguments that the function takes. Write them as another comment.

4. Come up with a name for the function that briefly describes what it does. Write the function header. This is the
   \[
   \text{def my_function_name(input1, input2, ...):}
   \]
   line that goes at the beginning of the function.

5. Copy and paste your script below the function header, as the body of the function, and indent it as necessary.

6. Copy your comment describing the function output and paste it near the end of the function, where you expect to put the `return` statement.

7. Put in the `return` statement to output whatever the function needs to output.

8. Delete any statements in the body of the function that explicitly supply data that will now be given as an input (argument) to the function. Change any references to this data so that they now refer to the correct function input (argument).

9. Go through your code and find any places where you used particular features of an example, such as the length of a list or the number of times a process should be repeated. Change them so they either (1) use the appropriate function inputs (arguments), or (2) calculate the necessary value as the function runs.

10. **Test your function** on the original example. When that works correctly, **test your function on other examples by calling it with different inputs**. Remember, the whole point of a function is that it should give the correct output for any inputs you give it, without you having to change the actual function code each time. All you should have to do is change the inputs when you “call” the function.

11. Remember, mistakes are expected, and programmers at all levels make them. The skills you learn in finding and fixing your mistakes should serve you well in other fields.
In the exercises below, you will turn Example 4 into a function that can iterate any single-variable function any number of times with any initial value.

**Exercise 24.** Copy Example 4 and make it a function called `iterate`. This function should take no inputs and return the list of values. Test your function. HINT: The header of a function that takes no inputs should have the form `def my_function_name():`

**Exercise 25.** Copy and modify your `iterate` function so that the number of times the iteration should be performed is an input. (Use a descriptive name.) Test your function with several different input values. HINT: Where does this number appear in the example?

**Exercise 26.** Copy and modify the previous version of your `iterate` function so that the initial value is also an input. As before, test your function with several different input values.

You now have a function that iterates \( f(x) = 2x \) any number of times and with any initial condition. But what if we want to iterate a different function? Python functions can take other functions as arguments simply using their names. The name of a function in SageMath does not include the list of arguments. For example, the name of \( \sin(x) \) is \( \sin \) and the name of \( f(t) \) is just \( f \). Therefore, if you wanted to run the function \( \text{myfun} \) with the function \( f \) as an argument, you would just type \( \text{myfun}(f) \).

**Exercise 27.** Copy and modify the previous version of your `iterate` function so that the function to be iterated is also an input. Test `iterate` with at least two different functions. HINT: What line of code will you need to remove when modifying the previous version of the function?

Congratulations! You’ve written a general function for performing iteration. Keep it for future use.