Contents

1 Mathematical Signs in CoCalc .......................... 1
   1.1 Equal Sign - Assigning ................................. 1
   1.2 Mathematical Operations .............................. 4
   1.3 Mathematical Comparisons ............................. 5

2 List and Dictionary Commands ........................... 7
   2.1 Defining a List ........................................... 7
   2.2 Indexes of the Components ............................. 8
       2.2.1 Positive Indexes ..................................... 8
       2.2.2 Negative Indexes ..................................... 9
   2.3 Getting the Length of the List (Number of Elements) ....... 10
   2.4 The srange Command ..................................... 11
   2.5 The zip Command ........................................ 14
   2.6 Getting the Maximum and Minimum Value of the List .......... 16
   2.7 Sorting a List in Increasing Order ..................... 17
   2.8 Adding an Element to the End of a List .................. 19
   2.9 Defining a Dictionary ................................... 20

3 Function Commands ....................................... 22
   3.1 Defining a Variable ..................................... 22
   3.2 Defining a Function ..................................... 23
       3.2.1 Defining a Mathematical Function .................... 23
       3.2.2 Defining a Sage Function ............................. 24

4 Plotting Commands ....................................... 30
   4.1 Plotting a List .......................................... 30
   4.2 Plotting a Function ..................................... 33
4.2.1 Plotting a Function in 2D ........................................ 33
4.2.2 Plotting a Function in 3D ........................................ 34
4.3 Plotting a Point ....................................................... 36
4.4 Showing a Text on the Graph ...................................... 37
4.5 Plotting a Vector Field .............................................. 38
4.6 Moving the Legend .................................................. 39

5 Repeating Commands .................................................. 44
5.1 for loop ............................................................... 44
5.2 while loop ........................................................... 53
5.3 Iteration ............................................................... 58
  5.3.1 The Method of Updating the Value Directly ............... 58
  5.3.2 The Method of Keeping All Values ......................... 61
5.4 Breaking a Loop ..................................................... 63

6 Conditional Command (if/else statement) ....................... 70

7 Solving Commands .................................................... 77
7.1 solve and assume Commands ...................................... 77
  7.1.1 solve Command ................................................. 77
  7.1.2 assume Command ............................................... 78
7.2 find_root Command ................................................ 80
7.3 Arrays and desolve_odeint Command (Simulation) ......... 82
  7.3.1 Arrays .......................................................... 82
    7.3.1.1 Define an Array ......................................... 82
    7.3.1.2 Indexes of the Elements in an Array ................. 83
  7.3.2 desolve_odeint Command .................................... 86
    7.3.2.1 Graphing a Time Series ................................. 88
    7.3.2.2 Graphing a Trajectory ................................ 88
7.4 dde_solve Command ................................................. 89

8 Special Effects Commands - Interactive and Animation .... 93
8.1 Interactive .......................................................... 93
8.2 Animation .......................................................... 95
  8.2.1 A Brief Introduction to Animation ....................... 95
  8.2.2 The Animation Command .................................... 95
9 Vector Commands

9.1 Defining a Vector ........................................... 98
9.2 Plotting a Vector ........................................... 98

10 Matrix Commands ........................................... 100

10.1 Defining a Matrix ........................................... 100
   10.1.1 Traditional Method (Row Matrix) ..................... 100
   10.1.2 Column Matrix ........................................ 101
   10.1.3 The Zero Matrix ...................................... 102
10.2 Getting the Dimensions ................................... 103
   10.2.1 General Dimensions ................................... 103
   10.2.2 Getting the Number of Rows ......................... 104
   10.2.3 Getting the Number of Columns ..................... 105
10.3 Recalling Elements from a Matrix ....................... 106
   10.3.1 Recalling a Particular Entry from Matrix ............ 106
   10.3.2 Recalling a Particular Row from Matrix .............. 107
   10.3.3 Recalling a Particular Column from Matrix .......... 108
10.4 Eigenvalues and Eigenvectors ............................ 109
   10.4.1 General Command ..................................... 109
   10.4.2 Understanding the Output of eigenvectors_right() .. 110
   10.4.3 Obtaining the Eigenvalues ............................ 111
   10.4.4 Obtaining the Eigenvectors ........................... 113
   10.4.5 Why Are the Coordinates of the Eigenvectors sometimes Non-Integers? 114

11 Derivatives and Jacobian Matrices ....................... 116

11.1 Calculating the General Derivative ...................... 116
11.2 Calculating the General Jacobian Matrix ............... 117
11.3 The sub Command - Derivative/Jacobian Matrix at a Point 119
   11.3.1 Using sub Without Using a Dictionary ............... 119
   11.3.2 Using sub Along with a Dictionary .................. 121

12 Showing Command ........................................... 123
Chapter 1

Mathematical Signs in CoCalc

1.1 Equal Sign - Assigning

Command: =

In CoCalc, “=” does not mean “equal.” It means assigning. When we write \( a = 3 \) into the CoCalc command line, it means that we are assigning the value 3 to the variable \( a \). As a result, whenever we type “a” into the subsequent command line, we essentially recall the value 3.

So what does: \( b = b + 1 \) mean? It means that we add 1 to whatever the value \( b \) is holding, then assign the new value back to the variable \( b \). For example, if we have the following code:

```
1 b = 4  # assigning the value 4 to b
2 b = b + 1  # add 1 to the value of b and assign the new value back to b
3 b  # show b
```

Analyzing the codes from above, we have:

1. The first line says that we will assign the value 4 to \( b \).
2. Then, the second line says that we will add 1 to the current value of \( b \), which is 4. Then, we will assign the sum, which is 5, back to the value of \( b \).
3. Finally, we will show the current value assigned to \( b \), which is 5.
1.1. Equal Sign - Assigning

This tool will come in handy when we want to add extra details or elements to a previously defined variable.

A very common case is the overlaying of the plots - where we literally add all the plots together to show on the same graph.

A common line of code for that is:

\[ p = \text{plot(...)} + \text{plot(...)} + \ldots \] (more on addition in the following section).

By using that line of code, we add all the plots together and assign the summation of all plots to a single variable \( p \).

For example, we trying to plot the functions \( f(x) = 2 \times x \) and \( g(x) = x^2 \) on the same graph for \(-5 \leq x \leq 5\), we have the following codes:

```python
1 f(x) = 2*x  # define the function f(x)
2 g(x) = x**2  # define the function g(x)
3 p = plot(f(x), (x, -5, 5)) + plot(g(x), (x, -5, 5), color = "red")  # plot the functions f(x) and g(x) between x = -5 and x = 5, with the graph for g(x) in red
4 show(p)  # show the plots assigned to p
```

Now, what if we want to add another plot to that collection of plots? We can retype the entire thing, but there is a simpler way than that.

Since the collection of the plots is already assigned to the single variable \( p \), we can take advantage of the assigning tool to add the new plot to \( p \) (which already holds a collection of plots), and then assign the new collection to \( p \) (just like what we did for the value of \( b \) from above).

\[ p = p + \text{plot(...)} \]

Now, let’s say we want to add a plot for \( h(x) = e^x \) in green on top of the previous plots, our new codes will look something like:
1. \( f(x) = 2^x \)  # define the function \( f(x) \)
2. \( g(x) = x^2 \)  # define the function \( g(x) \)
3. \( p = \text{plot}(f(x), (x, -5, 5)) + \text{plot}(g(x), (x, -5, 5), \text{color} = \text{"red"}) \)  # plot the functions \( f(x) \) and \( g(x) \) between \( x = -5 \) and \( x = 5 \), with the graph for \( g(x) \) in red
4. \( h(x) = e^x \)  # define the functions \( h(x) \)
5. \( p = p + \text{plot}(h(x), (x, -5, 5), \text{color} = \text{"green"}) \)  # add the plot for \( h(x) \)
6. \( \text{show}(p) \)  # show the plots assigned to \( p \)

**Note:** When we assign something to the variable, the variable will inherit its type.

For examples:

- \( a = 3 \): Since 3 is of type value, \( a \) is of type value
- \( b = \text{"hi"} \): Since \( b \) is of type strings, \( a \) is of type strings
- \( \text{mylist} = [1, 2, 3, 4] \): Since \([1, 2, 3, 4]\) is of type list, \( \text{mylist} \) is of type list
- \( p = \text{plot}(f(x), (x, 0, 5)) \): Since \( \text{plot}(f(x), (x, 0, 5)) \) is of type graphics, \( p \) is of type graphics

**Note:** Capitalization of the variable **DOES** matter. “X” is different than “x”.

Please **AVOID** using the known CoCalc commands as your variables **AT ALL COSTS**. The list of known CoCalc commands includes: \( \text{var, list, def, abs, sqrt} \), etc.
1.2 Mathematical Operations

**Command:** + - * / \^ 

These operations are used in the same way they have been since you first learn how to do mathematics.

The **most frequent mistake** is the multiplication sign:

**Incorrect:** cLH  
**Correct:** c*L*H  

**Note:** The “+” sign also has another meaning in CoCalc - it means **joining** the items of the **SAME types** together.

Let’s take a look at two simple examples.

**In our first example, we have:**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3 + 3 # add the values of 3 together</td>
</tr>
<tr>
<td></td>
<td>6</td>
</tr>
</tbody>
</table>

In this case, the “+” sign will join the two items of type Integer together. Hence, the addition is performed, and we get the value 6 as the answer.

What if we are not adding items of type Integer (or any numerical type) at all? It is doable as we already saw in the previous section, where we added two items of the type Graphics together. We will show that example again here for our second example.

**In our second example, we have:**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>f(x) = 2*x # define the functions f(x)</td>
</tr>
<tr>
<td>2</td>
<td>g(x) = x^2 # define the functions g(x)</td>
</tr>
<tr>
<td>3</td>
<td>p = plot(f(x), (x, -5, 5)) + plot(g(x), (x, -5, 5), color = “red”) # plot the functions f(x) and g(x) between x = -5 and x = 5, with the graph for g(x) in red</td>
</tr>
<tr>
<td>4</td>
<td>show(p) # show the plots assigned to p</td>
</tr>
</tbody>
</table>
As we observed, the “+” sign will join the two items of type Graphics together. Hence, the two plots are joined, leading to an overlay of graphs.

1.3 Mathematical Comparisons

**Command:**
- Equal to: == (notice the double equal sign)
- Not equal to: ! =
- Greater than: >
- At least: >=
- Smaller than: <
- At most: <=

When we do these problems, it is particular important that we pay attention to the wordings. Just some really quick examples:

<table>
<thead>
<tr>
<th>Description</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>a is equal to 3</td>
<td>a == 3</td>
</tr>
<tr>
<td>a is not equal to 3</td>
<td>a ! = 3</td>
</tr>
<tr>
<td>a is greater than 3</td>
<td>a &gt; 3</td>
</tr>
</tbody>
</table>
### 1.3. Mathematical Comparisons

<table>
<thead>
<tr>
<th>Condition</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>a is at least 3</td>
<td>$a \geq 3$</td>
</tr>
<tr>
<td>a is smaller than 3</td>
<td>$a &lt; 3$</td>
</tr>
<tr>
<td>a is at most 3</td>
<td>$a \leq 3$</td>
</tr>
</tbody>
</table>

The most frequent mistake is the equal sign:

**Incorrect:** $a = 3$

This is incorrect because as we recall, a single “=” means that we are **assigning** the value of 3 to the variable $a$.

**Correct:** $a == 3$

This is correct because the “==” sign means that we are checking if $a$ is **equal** to 3.
Chapter 2

List and Dictionary Commands

2.1 Defining a List

Command:

\[
\text{[element1, element2, etc.]}\]

where

- we will type the elements directly in the positions of \text{element1}, \text{element2}, etc.

For example, let’s say we want to define a list of time values: 0, 10, 20, 30, 40, 50 in Cocalc. Our command will be:

1 \[0, 10, 20, 30, 40, 50\] # define the list

\[0, 10, 20, 30, 40, 50\]

Now, let’s say that we want to assign the list to a variable name \text{t_list} so we don’t have to retype the list too many times, our codes will be:

1 \text{t_list} = [0, 10, 20, 30, 40, 50] # define the list and assign to the variable \text{t_list}

2 \text{t_list} # display the list assigned to \text{t_list}

\[0, 10, 20, 30, 40, 50\]
2.2 Indexes of the Components

The purpose of this command is to recall a specific element within a previously defined list.

Note: the list MUST already be assigned to a variable.

Command: list_name[index_number]

However, before we can use this command, it is important that we understand how index numbers work. There are two important rules that we will need to remember.

Two important rules regarding indexes:

1. The first item in the list will always have an index of 0.
2. The index will increase as we go from left to right and decrease as we go from right to left.

Due to Rule #2, there are two kinds of indexes: positive and negative.

2.2.1 Positive Indexes

This one is relatively self explanatory.

- The 1st element of a list has an index of 0.
- The 2nd element of a list has an index of 1.
- The 3rd element of a list has an index of 2.
... 
- The nth element of a list has an index of n-1.

For example, we have the following list:

\[ \text{list1} = [20, 21, 22, 23, 24] \]

The index for each element is as follows:

<table>
<thead>
<tr>
<th>Element</th>
<th>20</th>
<th>21</th>
<th>22</th>
<th>23</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>
Thus, if we want to recall element “22” from the list, you know that it is in the 3rd position with an index of 2. Hence, to recall the element “22” using codes, we will write:

```python
1 list_1 = [20, 21, 22, 23, 24]  # define the list list1
2 list_1[2]  # get the third element with index of 2
```

22

### 2.2.2 Negative Indexes

The way that we use negative indexes is a little less intuitive.

We will start with rule #1: “The first item in the list will always have an index of 0.”

But how do we go left from there? The only way we can is we go all the way to the back of the list, like how the snake bites its own tail. Thus, the last element of the list will have the index of -1.

- The first element of a list has an index of 0.
- The last element of a list has an index of -1.
- The second-to-last element of a list has an index of -2.
- ... 
- The second element of a list has an index of -(n-1).

Let’s use the same example from above. We have the following list:

```
list1 = [20, 21, 22, 23, 24]
```

Thus, the index for each element is as follows: (Be careful how you read the table because the first element “20” is now at the end of the list to facilitate how you read the negative indexes)

<table>
<thead>
<tr>
<th>Element</th>
<th>21</th>
<th>22</th>
<th>23</th>
<th>24</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index</td>
<td>-4</td>
<td>-3</td>
<td>-2</td>
<td>-1</td>
<td>0</td>
</tr>
</tbody>
</table>

Thus, if we want to recall the element “23” from the list, we know that it is in the second-to-last position with an index of -2. Hence, to recall the element “23” using codes, we will write:
2.3. Getting the Length of the List (Number of Elements)

Command:

\[
\text{len}(\text{list}\_a)
\]

where

- in the position of \text{list}\_a, you can either put:
  - a predefined list name, i.e. a variable to which a list is assigned
  - an actual list itself

For example, let’s say that we have the following list, \text{random}\_list:

\[
\text{random}\_list = [4, 11, 19, 0, 160, 89, 0, 29, 268, 47, 54, 3, 4, 56, 1, 614, 213, 10, 281, 248]
\]

To get the number of elements in that list, we can use one of the two following ways:

**Method #1: Use a predefined list name**

In this case, we will need to define the list in CoCalc, and then use the assigned name of the list to find the number of the elements within the list.
1. List and Dictionary Commands

Method #2: Use the actual list itself

In this case, we do **NOT** need to assign the list to a variable. Just use the list as it is.

```python
len([4, 11, 19, 0, 160, 89, 0, 29, 268, 47, 54, 3, 4, 56, 1, 614, 213, 10, 281, 248])
```

As you can see, while it is possible to use the list directly with the `len` command, it does get annoying to retype or copy/paste. Thus, it is **highly recommended** that you assign the list to a variable and use the variable from there on. It will save you a lot of time and reduce the chance of mistyping.

2.4 The `srange` Command

**Command:**

```python
srange(start_value, end_value, step_size)
```

(see the mechanism below)

**The mechanism of the `srange` command:**
The `srange` command will **generate a list** that:

1. starts with the value of `start_value`
2.4. The srange Command

2. increases/decreases in increments specified by the value of step_size
   - the list will increase if step_size > 0
   - the list will decrease if step_size < 0

3. stops one step before the value of end_value is reached

If not specified, the default value for step_size is 1.

Let’s take a look at a few examples to further understand this srange code.

Example #1: srange with an increasing step size

Let’s say that we are looking at the following code:

```
srange(0, 5, 0.5)
```

We know that this list will:

- start with the value of 0 (the start_value)
- increase in increments of 0.5 (the step_size value)
  - it increases because step_size > 0
- stop one step before the value of 5 is reached (the end_value)
  - hence, it will stop at 4.5 (and an additional step of 0.5 will mean that the value of 5 is reached, in the increasing direction)

Thus, the list generated will be:

```
[0, 0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5]
```

Example #2: srange with a decreasing step size

Let’s say that we are looking at the following code:

```
srange(5, 0, -0.5)
```

We know that this list will:

- start with the value of 5 (the start_value)
- decrease in increments of 0.5 (the step_size value)
○ it decreases because \textit{step\_size} < 0

- stop \underline{one step before} the value of 0 is reached (the \textit{end\_value})
  - hence, it will stop at 0.5 (and an additional step of 0.5 will mean that the value of 0 is reached, in the decreasing direction)

Thus, the list generated will be:

```
1 srange ( 5, 0, -0.5) # define the list with srange
```

[5, 4.5, 4, 3.5, 3, 2.5, 2, 1.5, 1, 0.5]

\textbf{Example \#3: define your own list with srange}

Let’s say that we need to obtain a list:

\[ [0, 5, 10, 15, 20] \]

We know that this list will:

- start with the value of 0
- increase in increments of 5
- stop at 20

Thus, at this point, we know that:

- our \textit{start\_value} is 0
- our \textit{step\_size} is 5

The tricky part lies at the \textit{end\_value}. Remember that srange stops \underline{one step BEFORE} the \textit{end\_value} is reached. Therefore, the \textit{end\_value} \textbf{must always be beyond} what you are trying to go for (i.e. greater than our final value if we are increasing, or less than our final value if we are decreasing).

In this case, we are increasing, so for the value of “20” to be \underline{one step BEFORE} the \textit{end\_value} with the \textit{step\_size} of 5, we know that the \textit{end\_value} is 25.

To summarize, we know that:

- our \textit{start\_value} is 0
- our \textit{step\_size} is 5
- our \textit{end\_value} is 25
Thus, our code will be:

```
1  srange ( 0, 25, 5)  # define the list with srange
  [0, 5, 10, 15, 20]
```

**Food for thoughts:** does the *end_value* in example #3 HAVE to be 25?

The answer is no. As long as our *end_value* is beyond 20 and less than 25, we are good.

If our *end_value* is beyond 25, our srange will take an additional step and reach 25, which is **NOT** what we want.

If what you just read confuses you, just remember to add an additional step beyond what you are trying to get (as you can see, the reasoning for whether or not that mark is absolute is a complicated process).

### 2.5 The zip Command

Very handy when we try to try to generate a list of data points where the “x-values” come from one list, “y-values” come from another list, etc.

In other words, the function of zip is to **make ONE LIST from MULTIPLE LISTS (2 and beyond)** where:

- The first elements of each list will be grouped together to form the tuple that is the first element of this newly created list
- The second elements of each list will be grouped together to form the tuple that is the second element of this newly created list
- The last elements of each list will be grouped together to form the tuple that is the last element of this newly created list

If you don’t know what “tuple” is, “tuple” is essentially a list, with the best example being a point on a graph. To draw a point on a 2D graph, you need to know the coordinates. And the collection of the coordinates, for example (2, 3), is a tuple.
**Command:**

```
zip( list1, list2, list3, etc. )
```

where

- in the positions of `list1, list2, list3,` etc., we can either put:
  - the list names, i.e. the variables to which appropriate lists are assigned
  - the actual lists themselves

For examples, let’s say that we have the following lists: a list of time marks, and a list of rabbits at those specific time marks. We want to combine the two to create a new list of data points (number of rabbits at specific time points), thus, we want to use `zip`.

Let’s say that our lists are:

```
time_list = [1, 2, 3, 4, 5]
rabbits_list = [100, 200, 300, 400, 500]
```

There are two main ways we can use `zip`:

**Method #1: Use previously defined list names**

In this case, we will need to define the lists in CoCalc, and then use the assigned names of the lists to zip the two of them together.

```
1 time_list = [1, 2, 3, 4, 5]          # define the time_list
2 rabbits_list = [100, 200, 300, 400, 500] # define the rabbits_list
3 zip(time_list, rabbits_list)        # zip the two lists together

[(1, 100), (2, 200), (3, 300), (4, 400), (5, 500)]
```

**Method #2: Use lists directly**

In this case, we do **not** need to assign the lists to variables. Just use the lists as they are.
2.6. Getting the Maximum and Minimum Value of the List

As you can see, while it is possible to use the list directly with the `zip` command, it does get annoying to retype or copy/paste. Thus, it is highly recommended that you assign the list to a variable and use the variable from there on. It will save you a lot of time and reduce the chance of mistyping.

2.6 Getting the Maximum and Minimum Value of the List

**Command:**

- `max(list_a)`
- `min(list_a)`

where

- in the position of `list_a`, we can either put:
  - a predefined list name, i.e. a variable to which a list is assigned
  - an actual list itself

Let’s take a look at an example for each method. For both examples, we will use the following list:

\[102, 61, 625, 62, 45, 61, 14, 90, 34, 21, 161, 181, 423, 630, 20, 141, 143, 12, 47, 84\]

**Method #1: Use a predefined list**

In this case, we will need to define the list in CoCalc, and then use the assigned name of the list to find the maximum or minimum value of the list.

Let’s say in this case, we will be assigning the list to the variable name, `list_1`, and we want to find the **maximum value**. Of course, we can use this method to find the minimum value as well (just switch out the command from “max” to “min”).

Thus, our commands will be:

```
zip([1, 2, 3, 4, 5], [100, 200, 300, 400, 500])  # zip the two lists together
```

\[ (1, 100), (2, 200), (3, 300), (4, 400), (5, 500) \]
1 list_1 = [ 102, 61, 625, 62, 45, 61, 14, 90, 34, 21, 161, 181, 423, 630, 20, 141, 143, 12,
47, 84]       # assign the list to the variable list_1
2 max(list_1)   # find the maximum value of the list, now represented by the
variable list_1

630

Method #2: Use the actual list itself
In this case, we do not need to assign the list to a variable. Just use the list as it is.

Let’s say in this case, we want to find the minimum value. Of course, we can use this method to
find the maximum value as well (just switch out the command from “min” to “max”).

Thus, our commands will be:

1 min( [ 102, 61, 625, 62, 45, 61, 14, 90, 34, 21, 161, 181, 423, 630, 20, 141, 143, 12,
47, 84])       # find the minimum value of the list

12

As you can see, while it is possible to use the list directly with the max and min commands, it
does get annoying to retype or copy/paste. Thus, it is highly recommended that you assign
the list to a variable and use the variable from there on. It will save you a lot of time and
reduce the chance of mistyping.

2.7 Sorting a List in Increasing Order

Command:

list_a.sort()

where

- list_a MUST be a predefined list name
For this class, this tool will come in handy when we want to sort the list that contains equilibrium points, so we can test for equilibrium type more easily (or any test that we need a sorted list).

For example, let’s say that we have the following list:

\[102, 61, 625, 62, 45, 61, 14, 90, 34, 21, 161, 181, 423, 630, 20, 141, 143, 12, 47, 84]\n
Now, let’s say we want to sort that list, our codes will be:

```python
list1 = [102, 61, 625, 62, 45, 61, 14, 90, 34, 21, 161, 181, 423, 630, 20, 141, 12, 47, 84]  # define the list and assign to the variable list1
list1.sort()                           # sort the list1 in increasing order
list1                                 # show the list
```

The way this command works is that it will sort the list, and then assigns the sorted list to the original variable that the list comes with.

In our example, it will sort the list and then assign the sorted one to the variable `list1`.

**Note:** For the sake of completeness, we should also mention that this tool is not only limited to be used for list with numerical elements. It can also be used to sort lists in alphabetical orders, like a dictionary.

Let’s take a look at the following example. We have the following list of fruits:

```
fruitslist = ["apple", "orange", "kiwi", "strawberry", "banana"]
```

We want to sort this list in alphabetical order. Thus, our codes will be:

```python
fruitslist = ["apple", "orange", "kiwi", "strawberry", "banana"]  # define the list and assign to the variable fruitslist
fruitslist.sort()              # sort the fruitslist in alphabetical order
fruitslist                   # show the list
```
2.8 Adding an Element to the End of a List

**Command:**

```python
list_a.append(element_a)
```

where

- `list_a` **MUST** be a predefined list name
- `element_a` is the element that we want to add to the **END** of `list_a`

Let’s do a quick example here. Let’s say we have the following list:

```python
list1 = [10, 15, 12, 11, 25, 27, 17]
```

We want to add an element “30” to the end of the list. We can retype, but that will take a lot of effort if the list was given to us and is long. As a result, we like to opt to use “append” as our command.

```python
1 list1 = [10, 15, 12, 11, 25, 27, 17]  # define list1
2 list1.append(30)  # add element “30” to the end of the list
3 list1  # show the new list1
```

```
[10, 15, 12, 11, 25, 27, 17, 30]
```
2.9 Defining a Dictionary

**Command:**

\[ \text{dictionary\_name} = \{ a_1:b_1, a_2:b_2, \text{etc.} \} \]

where

- \( a_1, a_2, \text{etc.} \) are the list of items (like the words)
- \( b_1, b_2, \text{etc.} \) are the list of items associated with the items in \( a_1, a_2, \text{etc.} \) (like the definitions of the words)

This tool is used when we want to store a list of items that are associated with another list of items.

It might be helpful if we think of it as a dictionary because in a sense, dictionary is a list of words with associated definitions. So in this case, \( b_1, b_2, \text{etc.} \) are the definitions for the terms \( a_1, a_2, \text{etc.} \). And just like how we use the dictionary to find a definition of a term, we can use the CoCalc dictionary to recall the value “b” that is associated with a particular value “a”.

**Note:** to recall a value of “b”, we will use the corresponding value of “a” as the the index to the list of \text{dictionary\_name}, just like how we flip through the actual dictionary and look for the word first, before we look at the definition.

Let’s take a look at an example to see how this works.

For example, let’s say that we are studying French, and we are using a dictionary to translate words from French to English. And let’s say that this “dictionary” only has 5 words in it.

<table>
<thead>
<tr>
<th>French Word</th>
<th>English Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>maison</td>
<td>house</td>
</tr>
<tr>
<td>soleil</td>
<td>sun</td>
</tr>
<tr>
<td>lune</td>
<td>moon</td>
</tr>
<tr>
<td>pays</td>
<td>country</td>
</tr>
<tr>
<td>football</td>
<td>soccer</td>
</tr>
</tbody>
</table>

Now, let’s put this into the computer memories as a CoCalc dictionary. After storing this dictionary as a CoCalc dictionary, we would like to use our dictionary to “look up” the meaning of the word
“soleil”. Our codes will look something like this:

```plaintext
2 FEdict["Soleil"]  # use the dictionary to look up the meaning of the word “soleil”
```

Sun

**Note:** the a and b values are NOT restricted to just words. They can be anything.

The application of a dictionary is a lot more widespread than you might imagine. A very good real-world example for that statement is a menu. Similar to how a dictionary allows us to recall a definition from a term, a menu allows us to look up the price of a selected dish.

Let’s put this context into codes. Let’s say that we have the following menu:

<table>
<thead>
<tr>
<th>Dish</th>
<th>Price (in $)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bolognese Spaghetti</td>
<td>14</td>
</tr>
<tr>
<td>Mushroom Pizza</td>
<td>12</td>
</tr>
<tr>
<td>Coke</td>
<td>3</td>
</tr>
</tbody>
</table>

Now, let’s say that we want to construct a dictionary, literally named “menu”, in the computer so you can use the code to recall prices later (which can prove particularly useful when we want to calculate the check at the end).

Let’s build this dictionary in codes and use it to find the price of a coke. Our codes will be:

```plaintext
1 menu = { "Bolognes Spaghetti" : 14, "Mushroom Pizza" : 12, "Coke" : 3 }  # define the menu using Sage dictionary
2 menu["Coke"]  # use the dictionary to look up the price of the Coke
```

3
Chapter 3

Function Commands

3.1 Defining a Variable

**Command:** Any of the following 4 commands are acceptable

- var(‘var1, var2, var3, etc.’)
- var(‘var1’, ‘var2’, ‘var3’, etc.)
- var(“var1, var2, var3, etc.”)
- var(“var1”, “var2”, “var3”, etc.)

**Note:**

- Capitalization DOES matter when you define a variable. The variable \( X \) is DIFFERENT than the variable \( x \).
- With the exception of the variable \( x \), we have to define ALL other variables that we want to use, **including** \( y \).

For example, we are trying to write a code to simulate the Shark-Tuna model. Let’s say we want to use the variable \( S \) and \( T \) to represent the shark and tuna populations, respectively. Our code to define these variables will be:

```
1 var(‘S, T’) # define the variables S and T
(S, T)
```
3.2 Defining a Function

There are two main ways we can define a function in Cocalc, the mathematical method and the Sage method. The functions defined by each method will be specified by the names of “mathematical functions” and “Sage functions”, respectively. While the mathematical method is simpler to use, the Sage method allows for more degree of control and complex computations.

3.2.1 Defining a Mathematical Function

**Command:**

```plaintext
func_name(variable1, variable2, etc.) = (whatever you want to accomplish here)
```

Recommended for functions that are relatively simple and easy to be defined

Great examples of functions that are relatively simple and easy to be defined are those that we have worked with since the middle school/high school age.

For example, \( g(x, y) = x^2 + y^2 \) (this function defines a circle centered at the origin)

In this case, we have:

- **func_name**: \( g \)
- **variable1**: \( x \)
- **variable2**: \( y \)
- (whatever you want to accomplish here): \( x^2 + y^2 \)

Then, we can compute the function at certain values of \( x = a \) and \( y = b \) by writing \( g(a, b) \).

Now, just plug them into the codes themselves:

```plaintext
1 g(x, y) = x^2 + y^2  # define the function g(x, y)
2 g(3, 4)  # compute the function at x = 3, y = 4
```
3.2. Defining a Function

**Note:** Unlike the other cases of using variables, we do **NOT** need to define the variables `variable1`, `variable2`, etc. before we can use them for the functions. They are automatically defined for us **WITHIN** the environment of the function. In other words, they are just **STANDING VARIABLES** that are used as **placeholders** for when we want to substitute values into the function to calculate for the value of the function at that particular state (such as substituting value of “3” in the positions of “x”, and substituting value of “4” in the positions of “y”).

However, once we leave the environment of the function (the window of codes that we are using to define and regulate the behavior of the function), the variables **will lose** its meaning.

### 3.2.2 Defining a Sage Function

**Command:**

```python
def func_name(variable1, variable2, etc.):
    (whatever you want to accomplish)
    return something
```

**Recommended for functions that are more complex**

We will start with something simple first. We can literally define the function \( f(x) = x^2 \) using the Sage function. Our command will be:

```python
1 def f(x): # start the definition of function f(x)
2     a = x^2  # calculate the value of x^2 and the assign the value back to the variable a
3     return a # return the a value, which x^2
4 f(2)    # calculate the value of the function at x = 2
```

The beauty of the Sage function is that the inputs of our function are not restricted to just numbers, as in the case of the Mathematical functions. We can literally use any other type, such as list, values, and plots, as the inputs to our Sage function. Of course, remember that the variables `variable1`, `variable2`, etc. are just **STANDING VARIABLES**. They are there to hold places for your inputs. Thus, when we define the function, just be careful of the placements. The naming of the variable is not as important, as long as we put the right
variable name in the correct location that we want our input to go to.

**Note:**

On the same line as what has already been mentioned, we can use any variable name within the function environment without having to previously defined them. It is just that once we step out of the function environment, that variable name loses its meaning. In the example above, the variable “a” does not have any meaning outside of the function. It is only a variable within the function environment that we can assign the value of $x^2$ to and then return that value via the Sage function.

Can we just write “return x^2” instead? we can. It is just that sometimes, the calculation is not as simple, so it is better that we assign to a variable that we can later just return to relatively quickly. Thus, it is recommended that the “something” in “return something” in the command guideline is actually a variable.

**Three very common mistakes in defining a Sage function:**

1. You forgot the “return” at the end. We will discuss whether or not you need the return, but when in doubt, use it.

2. You have incorrect indentation, causing incorrect calculations.

3. You forgot the “:” at the end of the defining function line.

With all of this already said, there are two other key points that we should be aware of when working with Sage functions.

**Keypoint #1:** The function will stop once it hits return, regardless if return is within the for loop/while loop or not.

We will not discuss the for loop and while loop until later, but essentially what they do is to repeat the exact command until a certain number of repetitions is reached (the **for loop**) or a condition is satisfied (the **while loop**).

However, it is worth talking about the positioning of the word “return” here. Let’s take a look at an example.

Let’s say we have the following codes:

```python
1    def f(x):  # start the definition of function f(x)
2        for i in srange(0, x, 1):  # start the for loop going from 0 to x in increments of 1
3            return i  # return the value of i
4    f(5)  # calculate the value of the function at x = 5
```
In our head, we might think that if $x = 5$, the for loop is confined with the srange of srange(0, 5, 1), which is a list of $[0, 1, 2, 3, 4]$. So, whenever we return the value of “i” from that list, our output will look something like:

```
0
1
2
3
4
```

This is incorrect. The reason is that the Sage function will terminate its operation after it executes the “return” command.

In our case, the first “return” command that the Sage function sees is for the value of $i = 0$. As a result, it will stop after it executes the “return i” line with $i = 0$.

Thus, our output will be like this:

```
def f(x):
    # start the definition of function f(x)
    for i in srange(0, x, 1):
        # start the for loop going from 0 to x in increments of 1
        return i  # return the value of i
f(5)  # calculate the value of the function at x = 5
```

Keypoint #2: We do not have to use “return”. However, it is recommended that we use “return” unless instructed otherwise.

The reasoning behind the recommendation is that the “return” command protects the type of the variable something that the function is returning. Let’s take a look at the following example for clearer demonstration.
We will define the same exact function $f(x) = x^2$ in two different ways, into the functions $f_1(x)$ and $f_2(x)$.

```python
1 def f1(x): # define function f1 using “return”
2     a = 2*x
3     return a
4
5 def f2(x): # define function f2 without using “return”
6     a = 2*x
7     print a # you “print” rather than “return”
```

At first glance, there seems to be no difference between the two functions if we try to compute for $f(2)$.

```python
1 def f1(x): # define function f1 using “return”
2     a = 2*x
3     return a
4
5 def f2(x): # define function f2 without using “return”
6     a = 2*x
7     print a # you “print” rather than “return”
8
9 f1(2) # compute f1 at x = 2
10
11 f2(2) # compute f2 at x = 2
```

Both of them yield 4 (with the first answer belongs to $f_1(2)$ and the second answer belongs to $f_2(2)$).

Now, what if we want to add 3 to the value of $f(2)$. One would think it will yield an answer of “7” (for $4 + 3$).
Let’s take a look.

```python
1 def f1(x):  # define function f1 using “return”
2     a = 2*x
3     return a

4 def f2(x):  # define function f2 without using “return”
5     a = 2*x
6     print a  # you “print” rather than “return”

9 f1(2)  # compute f1 at x = 2

f2(2)  # compute f2 at x = 2

f1(2) + 3  # add 3 to the computed value of f1(2)

f2(2) + 3  # add 3 to the computed value of f2(2)
```

As demonstrated, only the \textit{f1} can yield the correct answer, when \textit{f2} cannot. Why is that the case? It is because when we use “return” for \textit{f1}, it protects the type of the output, which is a value. Then, we can add a value to a value (since “3” is a value) to perform proper addition.

Meanwhile, when we use “print” for \textit{f2}, we have converted the output from a value to a string. We cannot add a string of “4” (though it looks exactly the same as the other answer) to a value of “3”, thus, we get an error message.

With that being said, we can use other commands when it is necessary. A very common example is to show a certain graph.
def showgraph(n):  # define the function showgraph
    p = plot(x^n, (x, 0, 5))  # plot the graph of x^n from x = 0 to x = 5 and
                        # assign the resulting plot to the variable p
    show(p)  # show the graph
showgraph(5)  # plot the graph of x^n from x = 0 to x = 5 when n = 5
Chapter 4

Plotting Commands

4.1 Plotting a List

Command:

\[
\text{list}_\text{plot}( \text{listname}, \text{size} = \text{insert\_value\_here}, \text{plotjoined} = \text{True}, \text{color} = \text{“insert\_color\_here”}, \\
\text{axes\_labels} = [\text{“variable1”, “variable2”}], \text{legend\_label} = \text{“insert\_legend\_label\_here”})
\]

where

- \text{listname} can be either an actual list or a variable to which a list was previously assigned
- \text{size} will give you the size of your points on the graph
- \text{plotjoined} = \text{True} will connect your points using linear segments
- \text{color} will give your graph your desired color - default is blue
- \text{axes\_labels} will label your axes appropriately
- \text{legend\_label} will show the legend with the plot labeled appropriately

Note:

- When we overlay the plots, only the last \text{axes\_labels} command will be taken into consideration when labeling the axes, so be very careful. If we need a refresher for overlaying plot, please visit section 1.2.
- We can either use \text{plotjoined} or \text{size} command, but NOT BOTH.

Let’s take a look at two examples.
Example #1: The list of rabbits (with no time marks)

Let’s say that we have the following number of rabbits, measured over the span of 5 years (but we did not record the actual years themselves).

```
rabbits_list = [100, 200, 300, 400, 500]
```

We want to plot the number of rabbits over time. Thus, our commands will be:

```
1 rabbits_list = [ 100, 200, 300, 400, 500]  # define the list
2 p = list_plot( rabbits_list, size = 20, color = "green", axes_labels = [ "Year", "Number of Rabbits"] )  # plot the list and assign to the variable “p”
3 show(p)  # show the plot
```

Example #2: The list of rabbits (with time marks)

Now, let’s add the time marks to them. Same as before, we have the list of the number of rabbits over the 5-years span, but this time, we are also given the specific year mark.

```
rabbits_list = [100, 200, 300, 400, 500]
years_list = [1, 2, 3, 4, 5]
```

Seems like the example before? We shall see... Some will think that our code will be exactly the same as before. Let’s try that first:
4.1. Plotting a List

rabbits_list = [100, 200, 300, 400, 500] # define the list

p = list_plot(rabbits_list, size = 20, color = "green", axes_labels = ["Year", "Number of Rabbits"])

That looks reasonable... until we realize that we have 100 rabbits for year 0, and not year 1... Why is that?

The reason is, when we say list_plot(rabbits_list), what is it essentially doing is plotting: the value of the element against its own index. In example #1, we never specify what year that was, so year 0 was perfectly okay. In this case, we specifically said that we have 100 rabbits for year 1, and not 0, thus, we need a way to combine the two lists.

Since you now need a way to combine the two lists to generate a list of points to plot, we will have to use zip to combine the rabbits_list and years_list into a new list with corresponding data points (for the number of rabbits at specific years).

Note: Unless specified otherwise with the command zip, the list_plot command will plot the elements against their own respective indexes.

Let's update our codes.

We know that the years_list will specify the x-values, hence we will have it as the first list of the zip command (since the x-values are the first values in the (x,y) coordinates).
4.2 Plotting a Function

4.2.1 Plotting a Function in 2D

**Command:**

```
plot( function_name, (independent_variable, xmin, xmax), ymin = insert_value_here,
     ymax = insert_value_here, color = “insert_color_here”, axes_labels = [ “variable1”,
     “variable2”], aspect_ratio = insert_value_here, legend_label = “insert_legend_label here”)```

where

- **function_name** can be the function name or expression itself
  - e.g.: if you are trying to plot $f(x) = 2 \times x$, in the position of function name, you can put: f, f(x), OR $2 \times x$. Any of the three will work.

- **independent_variable** is the independent variable of your function, such as: t, x, N, etc.
  - it is interesting to note that for the independent_variable, you can either put the actual independent variable or the generic variable “x”. Either will work.
4.2. Plotting a Function

- **xmin** and **xmax** will define your minimum and maximum “x values” on the graph (hence the horizontal width of your graph)
- **ymin** and **ymax** will define your minimum and maximum “y values” on the graph (hence the vertical height of your graph)
- **color** will give your graph your desired color - default is blue
- **axes_labels** will label your axes appropriately
- **aspect_ratio** will scale your graph appropriately
  - A value less than 1 will make your “1 unit” on the horizontal axis shorter than your “1 unit” on the vertical axis.
  - A value equal than 1 will make your “1 unit” on the horizontal axis equal to your “1 unit” on the vertical axis.
  - A value greater than 1 will make your “1 unit” on the horizontal axis longer than your “1 unit” on the vertical axis.
- **legend_label** will show the legend with the plot labeled appropriately

Let’s take a look at an example. Let’s say we want to plot the graph for $f(x) = x^2$, in color red and in the following window: $-5 \leq x \leq 5$ and $-1 \leq y \leq 30$. Hence, our codes will be:

1. `f(x) = x^2` # define the function f(x)
2. `plot( f, (x, -5, 5), ymin = -1, ymax = 30, color = “red”)` # plot the graph

4.2.2 Plotting a Function in 3D

The command is similar to the command in 2D, except you put the “ymin” and the “ymax” into the format of (variable, min, max), and it is `plot3d` instead of `plot`. 
Command:

```
plot3d( function_name, (variable1, min1, max1), (variable2, min2, max2),
    color = "insert_color_here", aspect_ratio = insert_value_here, legend_label =
    "insert_legend_label here")
```

where

- `function_name` can be the function name or expression itself
  - e.g.: if you are trying to plot \( f(x, y) = 2x + 3y \), in the position of `function_name`, you can put: \( f \), \( f(x, y) \), OR \( 2x + 3y \). Any of the three will work.
- `variable1` and `variable2` are the two independent variables of your function, e.g.: \( x \) and \( y \)
- `min1` and `max1` will define your length of the horizontal axis defined by the `variable1`
- `min2` and `max2` will define your length of the vertical axis defined by the `variable2`
- `color` will give your graph your desired color - default is blue
- `legend_label` will show the legend with the plot labeled appropriately

**Note:** There is currently **NO** known way to label the axes in the 3D graph.

Let’s look at an example. Let’s say, we want to graph the function \( f(x, y) = 2x^2 + y^2 \), in color blue and in the following window: \(-3 \leq x \leq 3\) and \(-3 \leq y \leq 3\). Hence, our codes will be:

```
1  f(x,y) = 2*x^2 + y^2       # define the function f(x,y)
2  plot3d( f, (x, -3, 3), (y, -3, 3), color = "blue" )       # plot the graph
```
4.3 Plotting a Point

**Command:**

For 2D:

\[
\text{point}( [ x_{\text{value}}, y_{\text{value}}], \text{color} = \text{“insert\_color\_here”}, \text{size} = \text{insert\_value\_here}, \text{legend\_label} = \text{“insert\_legend\_label\_here”})
\]

For 3D:

\[
\text{point3d}( [ x_{\text{value}}, y_{\text{value}}, z_{\text{value}}], \text{color} = \text{“insert\_color\_here”}, \text{size} = \text{insert\_value\_here}, \text{legend\_label} = \text{“insert\_legend\_label\_here”})
\]

where

- \(x_{\text{value}}, y_{\text{value}}, \text{and} z_{\text{value}}\) (if 3D) are the coordinates of the point
- \text{size} will give you the size of your point on the graph
- \text{color} will give your point your desired color - default is blue
- \text{legend\_label} will show the legend with the plot labeled appropriately

Let’s do a quick example. Let’s say we want to plot a red point, size 40, at position (1,4), thus a 2D graph, our command will be:

```plaintext
1 point ( [ 1, 4], color = “red”, size = 40)  # plot the point
```

![Graph example](image)
4.4 Showing a Text on the Graph

**Command:**

For 2D:

```
    text ( "insert_words_here", [ x_value, y_value], color = "insert_color_here")
```

For 3D:

```
    text3d ( "insert_words_here", [ x_value, y_value, z_value], color = "insert_color_here")
```

where

- `insert_words_here` is any text that you want to show on the graph
- `x_value`, `y_value`, and `z_value` (if 3D) will specify the location where the text will be displayed
- `color` will give your point your desired color - default is blue

Let’s also do another quick example. In this case, we want to display the word “hi” in color green at the location specified by the point (2, 3) on a 2D graph. Our commands will be:

```
1    text ( "hi", [ 2, 3], color = "green", size = 40)  # show the text
```

![Graph showing the word 'hi' at the location (2, 3) in color green]
4.5 Plotting a Vector Field

**Command:**

```python
plot_vector_field ( [equation1, equation2]), (variable1, min1, max1), (variable2, min2, max2), color = "insert_color_here"
```

where

- *equation1* is the equation that defines the horizontal component (direction and magnitude) of the vector
  - it is the rate of change according to the variable1 (e.g. X’)
  - just as before, you can use any of the three methods to put in the position of *equation1*: \( f, f(x, y) \) or the expression itself (\( f, x, y \) are chosen arbitrarily to represent the function name and the independent variables)

- *equation2* is the equation that defines the vertical component (direction and magnitude) of the vector
  - it is the rate of change according to the variable2 (e.g. Y’)
  - just as before, you can use any of the three methods to put in the position of *equation2*: \( f, f(x, y) \) or the expression itself (\( f, x, y \) are chosen arbitrarily to represent the function name and the independent variables)

- *variable1* and *variable2* are the two independent variables of your function, e.g.: \( x \) and \( y \)

- *min1* and *max1* will define your length of the horizontal axis defined by the *variable1*

- *min2* and *max2* will define your length of the vertical axis defined by the *variable2*

- *color* will give your graph your desired color - default is black

For example, let’s say we are trying to graph the vector field that represents the following Romeo and Juliet model:

\[
\begin{align*}
R' &= -J \\
J' &= R
\end{align*}
\]

Let’s say we are also trying to plot the vectors that are in color blue and contained within the following window: \(-5 \leq R \leq 5\) and \(-5 \leq J \leq 5\).

Thus, our commands for plotting this vector field are:

```python
1 var('R, J') # define R and J as variables
2 plot_vector_field ( [ -J, R], (R, -5, 5), (J, -5, 5), color = "blue" ) # plot the vector field
```
Remember that the first output \((R, J)\) comes from the \texttt{var(‘R, J’)} command line.

4.6 Moving the Legend

As the title suggested, this command will be useful when we want to move the legend out to not block our graphs.

**Command:**

\[
\text{show ( plot\_variable, legend\_loc = ( horizontal\_value, vertical\_value ) )}
\]

where

- \textit{plot\_variable} is the variable to which the plot is assigned
- \textit{horizontal\_value} will determine the location of the legend along the horizontal line. The value will vary \texttt{from 0 to 1}.
  - \textit{horizontal\_value} = 0 $\rightarrow$ the legend will be on the left side of the graph
  - \textit{horizontal\_value} = 1 $\rightarrow$ the legend will be on the right side of the graph
- \textit{vertical\_value} will determine the location of the legend along the vertical line. The value will vary \texttt{from 0 to 1}.
  - \textit{vertical\_value} = 0 $\rightarrow$ the legend will be on the bottom side of the graph
  - \textit{vertical\_value} = 1 $\rightarrow$ the legend will be on the top side of the graph
4.6. Moving the Legend

We will use one function as an example, but we will vary the location of the legend to five different places to demonstrate the idea.

We will plot the function $f(x) = 2x$ for $0 \leq x \leq 10$.

**Bottom Left:**

We have:

- Bottom $\rightarrow$ \textit{vertical\_value} = 0
- Left $\rightarrow$ \textit{horizontal\_value} = 0

Our commands are:

```python
1  p = plot( 2*x, (x, 0, 10), legend_label = "Function 2*x") # plot the function and
    assign the plot to the variable "p"
2  show(p, legend_loc = (0, 0) ) # show the graph with the legend on the
    bottom left
```

**Bottom Right:**

We have:

- Bottom $\rightarrow$ \textit{vertical\_value} = 0
- Right $\rightarrow$ \textit{horizontal\_value} = 1

Our commands are:
Plotting Commands

1. \[ p = \text{plot}(2x, (x, 0, 10), \text{legend} \text{ label} = \text{"Function } 2x\text{"}) \] # plot the function and assign the plot to the variable “p”

2. \[ \text{show}(p, \text{legend} \text{ loc} = (0, 1)) \] # show the graph with the legend on the bottom right

Top Left:

We have:

- Top \(\rightarrow\) \text{vertical\_value} = 1
- Left \(\rightarrow\) \text{horizontal\_value} = 0

Our commands are:

1. \[ p = \text{plot}(2x, (x, 0, 10), \text{legend} \text{ label} = \text{"Function } 2x\text{"}) \] # plot the function and assign the plot to the variable “p”

2. \[ \text{show}(p, \text{legend} \text{ loc} = (1, 0)) \] # show the graph with the legend on the top left
Top Right:

We have:

- Top \( \rightarrow \) \textit{vertical value} = 1
- Right \( \rightarrow \) \textit{horizontal value} = 1

Our commands are:

1. \( p = \text{plot}(2 \times x, (x, 0, 10), \text{legend\_label} = \text{"Function 2*x"}) \)  
   # plot the function and assign the plot to the variable “p”
2. \( \text{show}(p, \text{legend\_loc} = (1, 1)) \)  
   # show the graph with the legend on the top right

Right in the Middle:

We have:

- Middle \( \rightarrow \) \textit{vertical value} = 0.5
- Middle \( \rightarrow \) \textit{horizontal value} = 0.5

Our commands are:

1. \( p = \text{plot}(2 \times x, (x, 0, 10), \text{legend\_label} = \text{"Function 2*x"}) \)  
   # plot the function and assign the plot to the variable “p”
2. \( \text{show}(p, \text{legend\_loc} = (0.5, 0.5)) \)  
   # show the graph with the legend on the right in the middle
Chapter 5

Repeating Commands

5.1 for loop

This purpose of this tool is to have the computer performing the same task over and over again for a **PREDEFINED** amount of time.

**Command:**

```
for var in list_name:
    loop body
```

where

- `var` can be any variable of your choice (that does **NOT** need to be predefined via the “var” operation)
- `list_name` can be either:
  - a variable to which a list was previously assigned, or
  - a list itself

However, before we learn how to use the *for loop*, it is important that we learn how it works.

**How the *for loop* works:**

- For the **first loop**:
  1. It will take the **first element** in the list `list_name`.
  2. It then assigns the element to the variable `var`.
  3. It will perform the commands in the *loop body* to complete the first loop.
• For the **second loop**:

1. It will take the **second element** in the list *list_name*.
2. It then assigns the element to the variable *var*.
3. It will perform the commands in the *loop body* to complete the second loop.

... 

• For the **last loop**:

1. It will take the **last element** in the list *list_name*.
2. It then assigns the element to the variable *var*.
3. It will perform the commands in the *loop body* to complete the last loop.

As you can see, the number of elements in *list_name* will determine how many loops the *for loop* will run through. More specifically, for a list *list_name* with *n* elements, we have:

<table>
<thead>
<tr>
<th>Loop</th>
<th>Element Taken from <em>list_name</em></th>
<th>Index of the Element</th>
<th>Assigning Step of the <em>for loop</em></th>
</tr>
</thead>
<tbody>
<tr>
<td>1&lt;sup&gt;st&lt;/sup&gt;</td>
<td>1&lt;sup&gt;st&lt;/sup&gt;</td>
<td>0</td>
<td><em>var = list_name[0]</em></td>
</tr>
<tr>
<td>2&lt;sup&gt;nd&lt;/sup&gt;</td>
<td>2&lt;sup&gt;nd&lt;/sup&gt;</td>
<td>1</td>
<td><em>var = list_name[1]</em></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td><em>n</em>&lt;sup&gt;th&lt;/sup&gt;</td>
<td><em>n</em>&lt;sup&gt;th&lt;/sup&gt;</td>
<td><em>n</em>-1</td>
<td><em>var = list_name[<em>n</em> - 1]</em></td>
</tr>
</tbody>
</table>

Now, there are two ways we can use *var* in your *loop body*:

1. We can omit it from the *loop body*, making *var* only a **counting tool**, telling the computer what loop it is on and when to stop, **OR**

2. We can include *var* in the *loop body*, allowing it to play a part in **both the counting and computing process**.

We will look at an example from each one to solidify this concept.

**Method #1: Using *var* simply as a counting tool**

Let’s say that we want to print the number “2” five times. Since we know the exact number of times, *for loop* is a perfect tool to use.
There are a few ways we can do this, but in this case, we will specifically use `var` simply as a counting tool.

We know the following things:

1. The `for loop` takes the elements from `list_name`, thus the number of elements in `list_name` will govern how many loops are there. **Since we need to print five times, we will need to have a list, `list_name`, that has five elements.**

2. Our task is to print the number “2” five times. In this case, our loop body simply consists of one single and simple command “print 2”.

Let’s put this together. We only need a list of five elements, and it does not matter what the elements are. So, we will just pick:

\[
\text{list1} = [1, 2, 3, 4, 5]
\]

Our commands will then be:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>list1 = [1, 2, 3, 4, 5]  # define a list with five elements</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>for i in list1:  # start the for loop with var being “i”</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>print 2  # “print 2” as the loop body command</td>
</tr>
</tbody>
</table>

Let’s walk through the results together so we can see how the variable `i` is used simply as a counting tool. We will use our table one more time. We have five lines of output, one for each loop that the `for loop` runs through.
Repeating Commands

<table>
<thead>
<tr>
<th>Loop</th>
<th>Order of Element Taken from list1</th>
<th>Actual Element Taken from list1</th>
<th>Assigning Step of the for loop</th>
<th>The Command within the loop body</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st</td>
<td>1st</td>
<td>1</td>
<td>i = 1</td>
<td>print 2</td>
<td>2</td>
</tr>
<tr>
<td>2nd</td>
<td>2nd</td>
<td>2</td>
<td>i = 2</td>
<td>print 2</td>
<td>2</td>
</tr>
<tr>
<td>3rd</td>
<td>3rd</td>
<td>3</td>
<td>i = 3</td>
<td>print 2</td>
<td>2</td>
</tr>
<tr>
<td>4th</td>
<td>4th</td>
<td>4</td>
<td>i = 4</td>
<td>print 2</td>
<td>2</td>
</tr>
<tr>
<td>5th</td>
<td>5th</td>
<td>5</td>
<td>i = 5</td>
<td>print 2</td>
<td>2</td>
</tr>
</tbody>
</table>

As you can see, also shown below for better visual representation, even though the value of \( i \) changes for each loop, the command is still the same “print 2”, leading to the same output for each for loop, independent of the value of \( i \).

```
list1 = [ 1, 2, 3, 4, 5]  # define a list with five elements
for i in list1:  # start the for loop with var being “i”
    print 2  # “print 2” as the loop body command
```

```python
2 ← i = 1
2 ← i = 2
2 ← i = 3
2 ← i = 4
2 ← i = 5
```
5.1. for loop

**Method #2: Using var simply as a counting and computing tool**

Let’s say we have the following list:

```python
animals_list = ["dog", "cats", "hamsters", "rabbits", "fishes"]
```

Now the task is that we want to print each pet on a separate line. Since it is a repeating task (printing a particular pet) for a specific number of times (5 times), we know that the for loop is an excellent tool to use.

At this point, there are two ways we can solve this.

**Solution #2A: Direct extraction**

In direct extraction, we will assign the element directly to the var, and we can do that by using `animals_list` as our list_name. By using `animals_list` in that position, we achieve two goals simultaneously:

1. We know that we need a way to end the for loop when it prints the last element of `animals_list`. Thus, we need a list that is as long as `animals_list` to restrict the number of loops (5 loops). What is a better list than `animals_list` itself?

2. We know that the for loop will assign the elements from a particular list to var. If we use `animals_list`, we can have the for loop assign the elements from `animals_list` to var. For each loop passing, var is assigned to a specific pet, so during that loop, all we have to do is to “print var” to print out the name of the pet.

Our commands are then:

```python
1 animals_list = ["dogs", "cats", "hamsters", "rabbits", "fishes"] # define the animals_list
2 for a in animals_list: # start the for loop with var being “a”
3 print a # “print a” as the loop body command

dogs
cats
hamsters
rabbits
fishes
```
Let’s walk through the results together so we can see how the variable \( a \) is used as a counting and computing tool. We will use our table one more time. We have five lines of output, one for each loop that the \textit{for loop} runs through.

<table>
<thead>
<tr>
<th>Loop</th>
<th>Order of Element Taken from \textit{animals_list}</th>
<th>Actual Element Taken from \textit{animals_list}</th>
<th>Assigning Step of the \textit{for loop}</th>
<th>The Command within the \textit{loop body}</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1\textsuperscript{st}</td>
<td>1\textsuperscript{st}</td>
<td>dogs</td>
<td>( a = \text{dogs} )</td>
<td>\textit{print} ( a )</td>
<td>dogs</td>
</tr>
<tr>
<td>2\textsuperscript{nd}</td>
<td>2\textsuperscript{nd}</td>
<td>cats</td>
<td>( a = \text{cats} )</td>
<td>\textit{print} ( a )</td>
<td>cats</td>
</tr>
<tr>
<td>3\textsuperscript{rd}</td>
<td>3\textsuperscript{rd}</td>
<td>hamsters</td>
<td>( a = \text{hamsters} )</td>
<td>\textit{print} ( a )</td>
<td>hamsters</td>
</tr>
<tr>
<td>4\textsuperscript{th}</td>
<td>4\textsuperscript{th}</td>
<td>rabbits</td>
<td>( a = \text{rabbits} )</td>
<td>\textit{print} ( a )</td>
<td>rabbits</td>
</tr>
<tr>
<td>5\textsuperscript{th}</td>
<td>5\textsuperscript{th}</td>
<td>fishes</td>
<td>( a = \text{fishes} )</td>
<td>\textit{print} ( a )</td>
<td>fishes</td>
</tr>
</tbody>
</table>

As you can see, as shown below for better visual representation, as the value of \( a \) changes for each loop, the command “\textit{print} \( a \)” will lead to a different output for each \textit{for loop}, dependent on the value of \( a \).

1

\begin{verbatim}
animals_list = [ "dogs", "cats", "hamsters", "rabbits", "fishes"]

# define the animals_list
for a in animals_list:
    # start the for loop with var being “a”
    print a  # “print a” as the loop body command

dogs ← a = “dogs”
cats ← a = “cats”
hamsters ← a = “hamsters”
rabbits ← a = “rabbits”
fishes ← a = “fishes”
\end{verbatim}
Solution #2B: Indirect extraction

The line of thought for “Indirect extraction” comes from the idea that if we want to extract the elements from *animals_list*, we will simply use the index. Thus,

<table>
<thead>
<tr>
<th>In order to print the pet...</th>
<th>You will need to extract the element...</th>
<th>Which can be extracted using the command...</th>
<th>And then use the following command to print...</th>
</tr>
</thead>
<tbody>
<tr>
<td>dogs</td>
<td>dogs (1st element, index of 0)</td>
<td>animals_list[0]</td>
<td>print animals_list[0]</td>
</tr>
<tr>
<td>cats</td>
<td>cats (2nd element, index of 1)</td>
<td>animals_list[1]</td>
<td>print animals_list[1]</td>
</tr>
<tr>
<td>hamsters</td>
<td>hamsters (3rd element, index of 2)</td>
<td>animals_list[2]</td>
<td>print animals_list[2]</td>
</tr>
<tr>
<td>rabbits</td>
<td>rabbits (4th element, index of 3)</td>
<td>animals_list[3]</td>
<td>print animals_list[3]</td>
</tr>
</tbody>
</table>

As you notice, the only thing that really changes is the number within the *animals_list[...]* command.

You also know that during the *for loop*, the only thing that changes from one loop to another is the *var*.

So let’s change that command to be more generic and adaptable to our *for loop*:

```
print animals_list[var]
```

You know that *var* will be assigned to the following values: 0, 1, 2, 3, 4.

But you also know that, according to the structure of the *for loop*, *var* will be assigned values that come from *list_name*. So we need a *list_name* that contains the elements: 0, 1, 2, 3, 4.

There are two ways we can do this:

<table>
<thead>
<tr>
<th>Method</th>
<th>Example Command Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Typing</td>
<td>list1 = [0, 1, 2, 3, 4]</td>
</tr>
<tr>
<td>Using srange</td>
<td>list1 = srange(0, 5)</td>
</tr>
</tbody>
</table>

Let’s put everything into commands:
animals_list = ["dogs", "cats", "hamsters", "rabbits", "fishes"]

# define the animals_list

list1 = range(0, 5)  # define the list that contains elements to be assigned to var

for k in list1:  # start the for loop with var being "k"
    print animals_list[k]  # "print animals_list[k]" as the loop body command

    dogs
    cats
    hamsters
    rabbits
    fishes

Let’s walk through the results together, so we can see how the variable \( a \) is used as a counting and computing tool for indirect extraction. We will use our table, again. We have five lines of output, one for each loop that the for loop runs through.

Remember that our list1 is our list_name in this case.

<table>
<thead>
<tr>
<th>Loop</th>
<th>Order of Element Taken from list1</th>
<th>Actual Element Taken from list1</th>
<th>Assigning Step of the for loop</th>
<th>The Command within the loop body</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st</td>
<td>1st</td>
<td>0</td>
<td>( k = 0 )</td>
<td>print animals_list[k]</td>
<td>dogs</td>
</tr>
<tr>
<td>2nd</td>
<td>2nd</td>
<td>1</td>
<td>( k = 1 )</td>
<td>print animals_list[k]</td>
<td>cats</td>
</tr>
<tr>
<td>3rd</td>
<td>3rd</td>
<td>2</td>
<td>( k = 2 )</td>
<td>print animals_list[k]</td>
<td>hamsters</td>
</tr>
<tr>
<td>4th</td>
<td>4th</td>
<td>3</td>
<td>( k = 3 )</td>
<td>print animals_list[k]</td>
<td>rabbits</td>
</tr>
<tr>
<td>5th</td>
<td>5th</td>
<td>4</td>
<td>( k = 4 )</td>
<td>print animals_list[k]</td>
<td>fishes</td>
</tr>
</tbody>
</table>
for loop

Each loop, the command “print animals_list[k]” will lead to a different output for each for loop, dependent on the value of \( k \). However, it is indirect, because it does not assign the elements from \( \text{animals_list} \) to \( k \) directly.

```python
animals_list = ["dogs", "cats", "hamsters", "rabbits", "fishes"]
# define the animals_list

list1 = srange(0, 5)  # define the list that contains elements to be assigned to var

for k in list1:  # start the for loop with var being "k"
    print animals_list[k]  # "print animals_list[k]" as the loop body command
```

dogs ← \( k = 0 \) ⇒ animals_list[k] = animals_list[0] = “dogs”
cats ← \( k = 1 \) ⇒ animals_list[k] = animals_list[1] = “cats”
hamsters ← \( k = 2 \) ⇒ animals_list[k] = animals_list[2] = “hamsters”
rabbits ← \( k = 3 \) ⇒ animals_list[k] = animals_list[3] = “rabbits”
fishes ← \( k = 4 \) ⇒ animals_list[k] = animals_list[4] = “fishes”

As confusing as indirect extraction might seem (relative to direct extraction), indirect extraction is a lot more versatile and can be especially useful in the concept of iteration.

For direct extraction, you have to extract ALL elements IN THE ORDER that they appear within the list.

Meanwhile, for indirect extraction, you are NOT confined to those requirements. You can extract out of order, extract only specific elements, etc. All you have to remember for indirect extraction is that the elements of your list_name are the indexes of the elements, from the list of interest (e.g. animals_list), that you want to extract.

Two very common mistakes in using a for loop:

1. You have incorrect indentation, causing incorrect calculations.
2. You forgot the “;” at the end of the defining function line.
5.2 while loop

This purpose of this tool is to have the computer performing the same task over and over again UNTIL A CERTAIN CONDITION IS REACHED (i.e. you do not know how many loops you have to go through in order to reach the condition).

**Command:**

```python
while conditions:
    loop body
```

where

- **conditions** are the conditions for which the *while loop* will keep running
  - it is important to remember that you want the *while loop* to keep running until you get a desired condition. In other words, as long as the conditions are NOT what you want, the *while loop* is active. As a result, you will put the UNDESIRABLE conditions in the position of *conditions*.

Just as in the case of the *for loop*, before we learn how to use the *while loop*, it is important that we learn how it works.

**How the *while loop* works:**

- For the **first loop**:
  1. It will check the conditions. If the conditions are satisfied for running (hence not your desirable condition), the *loop body* will be executed (i.e. a certain task will be performed). If not, the *while loop* command is terminated.

- For the **second loop**:
  1. It will check the conditions. If the conditions are satisfied for running (hence not your desirable condition), the *loop body* will be executed (i.e. a certain task will be performed). If not, the *while loop* command is terminated.

... (and it will keep going until the *while loop* found that the conditions are no longer satisfied for running (hence you obtain your desirable conditions)).

Let’s do an example. Let’s say we start with \( n = 5 \), and we want to keep adding 1 to \( n \) until \( n \) reaches 8.

So, when does the *while loop* run? It will run when the value of \( n \) is less than 8.
Then, when the value of $n$ is less than 8, we will add 1 to it, leading to the command line:

\[ n = n + 1 \quad \text{add 1 to } n \text{ and assign back to } n \text{ to update its value} \]

Our commands are then:

1. \( n = 5 \)  # assign 5 to the value of n
2. 
3. while \( n < 8 \):  # start the while loop and it will keep running as long as \( n < 8 \)
4. \( n = n + 1 \)  # add 1 to \( n \) and assign the new value back to \( n \)
5. 
6. \( n \)  # show the value of \( n \)

For instructional purposes, we will two more lines of codes to discuss in details what happens during each loop. More specifically, for each loop, we will print the value of \( n \) before and after the addition. Our commands are then:

1. \( n = 5 \)  # assign 5 to the value of n
2. 
3. while \( n < 8 \):  # start the while loop and it will keep running as long as \( n < 8 \)
4. print \( n \)  # show the value of \( n \) before the addition
5. \( n = n + 1 \)  # add 1 to \( n \) and assign the new value back to \( n \)
6. print \( n \)  # show the value of \( n \) after the addition
7. 
8. \( n \)  # show the value of \( n \)

And here are our results:
So let’s explain the reasoning behind each line of output that we saw.

The last output, the bottom “8”, belongs to the bottom command “n”, and thus will not be explained further. The other outputs belong to the *while loop*.

<table>
<thead>
<tr>
<th>Loop</th>
<th>Value of $n$ at the beginning of the <em>while loop</em></th>
<th>$n &lt; 8$?</th>
<th>The Command within the <em>loop body</em></th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1&lt;sup&gt;st&lt;/sup&gt;</td>
<td>5</td>
<td>Yes</td>
<td>print $n$</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$n = n + 1$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>print $n$</td>
<td>6</td>
</tr>
<tr>
<td>2&lt;sup&gt;nd&lt;/sup&gt;</td>
<td>6</td>
<td>Yes</td>
<td>print $n$</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$n = n + 1$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>print $n$</td>
<td>7</td>
</tr>
</tbody>
</table>
Just as in the case of the for loop, we will put our explanations directly into the answer box for better visual representation:

<table>
<thead>
<tr>
<th></th>
<th>3rd</th>
<th>7</th>
<th>Yes</th>
<th>Yes</th>
<th>print n</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>n = n + 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>print n</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4th</td>
<td>8</td>
<td>No</td>
<td>No</td>
<td>Not executed</td>
<td>8</td>
</tr>
</tbody>
</table>

Now, let’s talk about some of the common mistakes. One of the mistakes are so common that it deserves a special example to illustrate it.

**Four very common mistakes in using a for loop:**

1. You have incorrect indentation, causing incorrect calculations.
2. You forgot the “;” at the end of the defining function line.
3. You mixed up the conditions: put the desirable conditions, rather than undesirable
Repeating Commands

conditions in the position of *conditions*. Remember, you want the *while loop* to run as long as the conditions are not what you want, and stop when they are.

4. You have an infinite loop.

That last mistake is so common and so frustrated that it deserves an example for demonstration. Let’s take a look at the following codes:

```python
n = 1  # assign 1 to the value of n

while n < 10:  # start the while loop and it will keep running as long as n < 10
    print 5  # print the value 5
```

Obviously this code will give the infinite loop. But why is that?

Recall that the *while loop* will run as long as \( n < 10 \). So here is what happens:

- **First loop**: \( n = 1 \rightarrow n < 10 \rightarrow \text{the while loop is executed} \)
- **Second loop**: \( n = 1 \rightarrow n < 10 \rightarrow \text{the while loop is executed} \)
- and so on...

As you can see here, our codes do not have any method to change the value of \( n \). Thus, as we continue on, the value of \( n \) is always equal to 1, causing the *while loop* to always be active, leading to an infinite loop.

Thus, to kill the infinite loop, we need a way to update the value of \( n \), so it will rise to be above 10 after a certain point.

One such example is to keep adding one to it. In that example, our codes can look something like this:

```python
n = 1  # assign 1 to the value of n

while n < 10:  # start the while loop and it will keep running as long as n < 10
    print 5  # print the value 5
    n = n + 1  # update the value of n
```
5.3 Iteration

In the context of this course, iteration is the series of action of applying a function to a value, getting a new result, then applying the function to that new result, and repeating. In other words, we will get the following sequence of results by iterating \( n \) times:

\[
x_0 \xrightarrow{f(x_0)} x_1 \xrightarrow{f(x_1)} x_2 \xrightarrow{f(x_2)} \ldots \xrightarrow{f(x_{n-1})} x_n
\]

There are two ways we can do iteration, and unlike the previous sections, for this section, we will derive the general structure of the commands for each method, rather than give them out right away.

For a lot of the times, we will tell you how many iterations you will have to do. Since you know that you have to repeat the same action \( x_n = f(x_{n-1}) \) for a definite amount of time, the for loop is an excellent tool for iteration. With that being said, there will be certain situations where you will have to use the while loop, but that is rarely touched within the context of this class.

5.3.1 The Method of Updating the Value Directly

This method is relatively simple. However, it can **ONLY** be applied when you only care about getting the final result and do not need to recall any of the previous values in the iteration series.

This method stems the combination of ideas that:

- If we only care about the last values, why do we bother keeping all the values in the middle.
- We know that CoCalc allows us to update the value assigned to a variable, i.e. the concept of \( n = n + 1 \) that allows us to add 1 to the current value of \( n \) and assign back to \( n \). Thus, \( n \) then has a new value that is one more than its old value.

We know that for iteration,

\[
n_{new} = f(n_{old})
\]

With the self-assignment concept mentioned earlier, this method is centralized around the following single command:

\[
n = f(n)
\]

where we apply \( f \) to \( n \), and assign the result back to \( n \), so that this “new” \( n \) will be the “old” \( n \) for the next iteration. One example of that function that we saw earlier is \( f(x) = x + 1 \), so that when you write \( n = f(n) \), it is practically the same as \( n = n + 1 \).

So from what we have said, we already have the following skeleton of our work:
Repeating Commands

```python
a = some_value  # initial value of a

f(x) = ...  # define the function f(x)

for i in some_list:  # start the for loop for iteration
    a = f(a)  # iteration

a  # show the final value of a
```

**Note:** When you define the function (in line #3), the variable that you put within `f(...)` can be **anything but the variable** `a` (or anything you choose in line #1).

Now, what remains to be filled is `some_list`. To know this, we need to discuss the following items:

1. **What is our `some_list`?** We know that the number of elements in `some_list` will determine how many loops it will run through. Let’s say that we need `n` loops for `n` iterations. As a result, our `some_list` needs to have `n` elements.

2. **Do we want `i` to play a role in our loop body?** We know that all we need in our loop body is simply `a = f(a)`, thus, we don’t need `i` to play a role in our loop body. We know that `i` gets assigned values that are elements from `some_list`. Since we don’t need `i` to play a role loop body, we do not really care what are the actual elements in the `some_list`. All we need is that `some_list` contains exactly `n` elements for `n` iterations.

One very quick method to generate a list is the `srange` command. We know that the command:

```
srange(0, n)
```

will give a list of `[0, 1, ... n-1]`. Thus, that list contains exactly `n` elements, which is what we need. Let’s now update our commands, to give us the general skeleton:
Let’s now apply this concept to an example. Let’s say that you want to model the squirrel population at UCLA. And let’s also say hypothetically that the squirrel population doubles annually. You are currently a freshman in college and like to predict how many squirrels are there when you graduate (3 years from now), given that the current number of squirrels is 20.

We know the following items:

- The current number of squirrels is 20 → \( a = 20 \).
- You want to predict the number of squirrels 3 years from now → \( n = 3 \).
- You know that the number of squirrels doubles every year \( a_{\text{new}} = 2 \times a_{\text{old}} = f(a_{\text{old}}) \rightarrow f(a) = 2 \times a \rightarrow f(x) = 2 \times x \).

Thus, our commands are:

```
1 a = 20       # initial value of a = 20
2 f(x) = 2*x    # define the function f(x) = 2*x
3 for i in srange(0, 3):   # start the for loop for n = 3 iterations
   a = f(a)        # iteration
4 a               # show the final value of a
```

160

So by the time you graduate, the squirrel population has grown from 20 to 160.
5.3.2 The Method of Keeping All Values

Method #1 was relatively simple, but what if we want to keep track of all the values and graph them to see patterns over time? Method #1 will not be able to give us that.

While method #2 is slightly more complex, it allows for greater degree of data analysis, such as recalling specific elements or graphing to observe patterns.

To utilize method #2, we know the following five items:

1. We need something to store all results obtained from iterations → we should use a list.

2. The first element of that list will be the initial condition.

3. We need to iterate \( n_{\text{new}} = f(n_{\text{old}}) \). This step is relatively simple and self-explanatory.

4. After each iteration, we should add the newly obtained element to the list, so that element can be used as the “\( n_{\text{old}} \)” for the next step → we should use append.

5. We know that the \( n_{\text{new}} \) of this iteration is always added to the end of the list to become the \( n_{\text{old}} \) for the next iteration. Thus, at the beginning of each iteration, to obtain \( n_{\text{old}} \), we can simply extract the last element of the list → this can be obtained using the index of -1.

Knowing all of these items, we drafted the following skeleton:

```python
list1 = [some_value]  # start the list to store all values from iterations, with the first value being the initial condition

f(x) = ...  # define the function f(x)

for i in some_list:  # start the for loop for iteration
    a_old = list1[-1]  # extract the last element from list1 and make it the “old” value for this loop of iteration
    a_new = f(a_old)  # iteration
    list1.append(a_new)  # add the “new” value to the end of the list

list_plot(list1)  # plot all values of the iterations
```

Same as before, what remains to be filled is `some_list`. To know this, we need to discuss the following items:

1. **What is our `some_list`?** We know that the number of elements in `some_list` will determine
how many loops it will run through. Let’s say that we need \( n \) loops for \( n \) iterations. As a result, our `some_list` needs to have \( n \) elements.

2. **Do we want \( i \) to play a role in our loop body?** We know that based on the way that we currently structure our codes, we don’t need \( i \) to play a role in our loop body.

We know that \( i \) gets assigned values that are elements from `some_list`. Since we don’t need \( i \) to play a role loop body, we do not really care what are the actual elements in the `some_list`. All we need is that `some_list` contains exactly \( n \) elements for \( n \) iterations.

Hence, we opt to use the `srange` command, and based on our previous discussion, we know that `srange(0, n)` will give a list of \([0, 1, \ldots, n-1]\) that contains exactly \( n \) elements, which is what we need.

Let’s now update our commands, to give us the general skeleton:

```python
list1 = [some_value]  # start the list to store all values from iterations, with the
first value being the initial condition

f(x) = ...  # define the function f(x)

for i in srange(0, n):  # start the for loop for n iterations
    a_old = list1[-1]  # extract the last element from list1 and make it the “old”
    value for this loop of iteration
    a_new = f(a_old)  # iteration
    list1.append(a_new)  # add the “new” value to the end of the list1

list_plot(list1)  # plot all values of the iterations
```

Let’s now apply this method to the same example before. Let’s say that you want to model the squirrel population at UCLA. And let’s also say hypothetically that the squirrel population doubles annually. You are currently a freshman in college and like to see how the number of squirrels will change for each year that you are here (i.e. plot them over the years), given that the current number of squirrels is 20.

We know the following items:

- The current number of squirrels is 20 \( \rightarrow \) `list1 = [20]`.
- You want to predict the number of squirrels 3 years from now \( \rightarrow n = 3 \).
- You know that the number of squirrels doubles every year \( a_{\text{new}} = 2 \cdot a_{\text{old}} = f(a_{\text{old}}) \rightarrow f(a) = 2 \cdot a \rightarrow f(x) = 2 \cdot x \).
Thus, our commands are:

```python
list1 = [20]  # start the list to store all values from iterations, with the first value being the initial condition of 20 squirrels

f(x) = 2*x  # define the function f(x) = 2*x

for i in srange(0, 3):  # start the for loop for n = 3 iterations
    a_old = list1[-1]  # extract the last element from list1 and make it the “old” value for this loop of iteration
    a_new = f(a_old)  # iteration
    list1.append(a_new)  # add the “new” value to the end of the list1

list_plot(list1)  # plot all values of the iterations
```

Just remember that for your work, you should label the axes appropriately and use “zip” if necessary.

### 5.4 Breaking a Loop

**Command:** break

It is used to break out from the **MOST IMMEDIATE** loop (either *for loop* or *while loop*).

We will look at two examples to demonstrate the use of this command, with the second one focusing...
5.4. Breaking a Loop

on what we mean by saying “most immediate loop”.

**Example #1: Simple demonstration of the use of “break”**

We will take a look at these codes:

```
for i in srange(0, 100):  # start the for loop for 100 times
    print i               # print the value of i
    break                 # break the loop
```

The output for the command is simply “0” because during the first loop, the computer assigns 0, the first element of the list generated via `srange(0, 100)`, to `i`. Then, it sees:

1. “print i” → the computer will print out the current value of `i` within that loop, which is 0 → “0” is printed in the output box

2. “break” → the computer then terminates the loop, and since that was the only loop of the commands, the work required is now finished → no additional output

**Example #2: Demonstration of how “break” only terminates the most immediate loop**

When we say “most immediate”, we mean that the loop that the “break” commands takes direct part of, so the **indentation** is key.

The following two sets of commands mean two different things:

**Command Set #1:**

```
for i in srange(0, 3):  # start the for loop for i
    print i            # print the value of i
    for j in srange(10, 12):  # start the for loop for j
        print j          # print the value of j
    break               # break the loop
```

Revised Fall 2019   LS 30 Comprehensive Coding Manual   Click Here to Go To TOC
Command Set #2:

```python
for i in srange(0, 5):  # start the for loop for i
    print i  # print the value of i
for j in srange(10, 15):  # start the for loop for j
    print j  # print the value of j
    break  # break the loop
```

In the absence of the “break” command:

You see that there are two for loops: one on the outside, and one on the inside. As the outside for loop runs, it will follow the following tasks:

1. See the line “print i” → the computer will print out the current value of i within that loop
2. See the inside for loop → it will allow the inside for loop to be active:
   (a) The computer then sees the line “print j” → the computer will print out the current value of j within that loop
   (b) It will repeat or terminate if the number of loops ends
3. Repeat or terminate if the number of loops ends

Essentially, it will:

1. Print the first value of i (first outside for loop)
2. Print all the values of j (during the execution of the first outside for loop)
3. Print the next value of i (second outside for loop)
4. Print all the values of j (during the execution of the second outside for loop)
5. ...
6. Print the last value of i (last outside for loop)
7. Print all the values of j (during the execution of the last outside for loop)

Thus, the output that we will be looking at is:
Now, let’s put the “break” command in.
**Command Set #1:**

```python
for i in srange(0, 3):  # start the for loop for i
    print i  # print the value of i
    for j in srange(10, 12):  # start the for loop for j
        print j  # print the value of j
    break  # break the loop
```

In this case, as the outside *for loop* runs, it will follow the following tasks:

1. See the line “print i” → the computer will print out the current value of *i* within that loop
2. See the inside *for loop* → it will allow the inside *for loop* to be active:
   (a) The computer then sees the line “print j” → the computer will print out the current value of *j* within that loop
   (b) It will repeat or terminate if the number of loops ends
3. See the line “break” (look carefully at the indentation, if you need to, to see that the “break” command belongs to the outside *for loop*, the → the *for loop* is terminated → no additional output)

Essentially, it will:

1. Print the first value of *i* (*first* outside *for loop*)
2. Print all the values of *j* (during the execution of the *first* outside *for loop*)
3. Stop, since it hits the “break” command

Putting everything together into the coding format, we have the following output:

```python
for i in srange(0, 5):  # start the for loop for i
    print i  # print the value of i
    for j in srange(10, 15):  # start the for loop for j
        print j  # print the value of j
    break  # break the loop
```
5.4. Breaking a Loop

0 ← print \( i \) for the 1\(^{\text{st}}\) outside loop

10 ← print \( j \) from the 1\(^{\text{st}}\) inside loop, as the 1\(^{\text{st}}\) outside loop is executing

11 ← print \( j \) from the 2\(^{\text{nd}}\) inside loop, as the 1\(^{\text{st}}\) outside loop is executing

12 ← print \( j \) from the 3\(^{\text{rd}}\) inside loop, as the 1\(^{\text{st}}\) outside loop is executing

Command Set #2:

1 for \( i \) in \text{srange}(0, 5): \quad \# \text{start the for loop for } i
2 \quad \text{print } i \quad \# \text{print the value of } i
3 \quad \text{for } j \text{ in } \text{srange}(10, 15): \quad \# \text{start the for loop for } j
4 \quad \quad \text{print } j \quad \# \text{print the value of } j
5 \quad \quad \text{break} \quad \# \text{break the loop}

In this case, as the outside for loop runs, it will follow the following tasks:

1. See the line “print \( i \)” → the computer will print out the current value of \( i \) within that loop

2. See the inside for loop → it will allow the inside for loop to be active:
   
   (a) The computer then sees the line “print \( j \)” → the computer will print out the current value of \( j \) within that loop
   
   (b) After, it sees the line “break” (look carefully at the indentation, if you need to, to see that the “break” command belongs to the inside for loop, the → the for loop is terminated → no additional output)

3. Repeat or terminate if the number of loops ends

Essentially, it will:

1. Print the first value of \( i \) (first outside for loop)

2. Print the first the value of \( j \) before the inside for loop is terminated via the “break” command (this is during the execution of the first outside for loop)

3. Print the second value of \( i \) (second outside for loop)

4. Print the first the value of \( j \) before the inside for loop is terminated via the “break” command (this is during the execution of the second outside for loop)
5. ...

6. Print the last value of \(i\) (last outside for loop)

7. Print the first the value of \(j\) before the inside for loop is terminated via the “break” command (this is during the execution of the last outside for loop)

Putting everything together into the coding format, we have the following output:

```python
for i in srange(0, 5):  # start the for loop for i
    print i  # print the value of i
    for j in srange(10, 15):  # start the for loop for j
        print j  # print the value of j
        break  # break the loop
```

0 ← print \(i\) for the 1\textsuperscript{st} outside loop

10 ← print \(j\) from the 1\textsuperscript{st} inside loop, as the 1\textsuperscript{st} outside loop is executing

1 ← print \(i\) for the 2\textsuperscript{nd} outside loop

10 ← print \(j\) from the 1\textsuperscript{st} inside loop, as the 2\textsuperscript{nd} outside loop is executing

2 ← print \(i\) for the 3\textsuperscript{rd} outside loop

10 ← print \(j\) from the 1\textsuperscript{st} inside loop, as the 3\textsuperscript{rd} outside loop is executing

The most common mistake involving the “break” command is improper indentation. As demonstrated above, the indentation of the “break” command is very important, since it will determine which loop will be cut short, so be very extra careful when using it.
Chapter 6

Conditional Command (if/else statement)

**Command:**

if *condition_1*:
    perform *tasks_1*
elif *condition_2*:
    perform *tasks_2*
    (insert as many elif statements as you want)
else:
    perform *tasks_n*

where

- in each position of *condition_1, condition_2*, etc., you can have smaller conditions joined together via “AND” or “OR”.

This purpose of this tool is to have the computer execute certain tasks only under certain conditions.

There are a few important points to make regarding the mechanism of the if/else statement.

**Notes regarding its mechanism:**

1. The if/else statement will only execute a certain task if the condition stated on the line immediately above it results in a “true” result (e.g. 2 = 2 is “true”). For example, it will perform *tasks_1* if *condition_1* is “true.” As a result, it is important to carefully check when you use “AND” or “OR” in your conditions.

   - For “AND”, the condition is “TRUE IF AND ONLY IF EVERY small
Conditional Command (if/else statement)

- Conditions are “true.” (e.g., if \( a = 5 \) and you are checking for \( a > 0 \) and \( a < 10 \), it will result in “true” for both smaller conditions, hence the large condition - \( a > 0 \) and \( a < 10 \) - is “true”).

  - For “OR”, the condition is **TRUE** **IF AT LEAST ONE** small condition is “true.” (e.g., if \( a = 20 \) and you are checking for \( a > 0 \) or \( a < 10 \), it will result in “true” for \( a > 0 \), hence the large condition - \( a > 0 \) or \( a < 10 \) - is “true”).

2. The computer will read the if/else statements from top to bottom, and then left to right. Thus, it is important that the order of your conditions are correct.

3. The computer will enter the “else” portion **IF ALL** of the conditions from the above “if” and “elif” lines are **NOT** satisfied.

4. A complete set of if/else statement starts with the “if” and ends with the “else” (with the “elif” in between).

   - It is okay to not have “elif” or “else” if it is not necessary.
   - However, it is important to not confuse between “elif” and “if”. **While “elif” links between the conditions, “if” will start a brand new set.**

We will do a few examples to illustrate this mechanism.

**Example #1: A classic set up of the if/else statement (illustration for #1-3)**

Let’s say you have a certain value for \( n \), and you want to do one of the three things, depending on the value of \( n \):

1. Multiply \( n \) by 3 if \( n < 0 \)
2. Multiply \( n \) by 5 if \( 0 \leq n \leq 10 \)
3. Multiply \( n \) by 7 if \( n > 10 \)

Skeleton wise, we know we need three things:

1. A line to assign a value to \( n \) (the value that was given prior to if/else statement)
2. A 3-parts if/else statement, one part for each condition stated above
3. A line to show the final value of \( n \) to check that our if/else statement works appropriately

**A line to assign a value to \( n \):**

This is straight-forward.

```plaintext
1 n = some_value # assign a value to n
```
A 3-parts if/else statement:

There are a few ways we can structure the if/else statement. This is just an example. We will go in the order that it was stated in the problem statement.

Multiply $n$ by 3 if $n < 0$:
Since we just start an if/else statement, we need to start with “if”:

```python
1 if n < 0:  # if n < 0
2 n = 3 * n   # multiply the value of n by 3 then assign to itself
```

Multiply $n$ by 5 if $0 \leq n \leq 10$:
For this condition, we know that it is not the opening condition, so “if” cannot be used. However, it is also not the last condition, so “else” cannot be used. Thus, we have to use “elif” to make it a part of the 3-parts conditional statement.

We also know that we have to use the and condition because $n$ has to be between 0 and 10, and we cannot do a three-way comparison (like how the problem statement was written).

```python
1 elif n >= 0 and n <= 10:  # if n is between 0 and 10, inclusively
2 n = 5 * n   # multiply the value of n by 5 then assign to itself
```

Multiply $n$ by 7 if $n > 10$:
This is the last condition of our command, so it is natural to think that we can use “else” for this condition. However, it is also important to think why we can use “else” as well (in rare conditions, you cannot use “else”, so it is good to be careful in any case).

In this case, we can use “else” because if the two top conditions are not satisfied, the last condition is automatically true. More specifically, if $n$ is not less than 0 and is not between 0 and 10 (inclusively), it has to be greater than 10. Thus, it is the same to write “Multiply $n$ by 7 if $n > 10$” or “Multiply $n$ by 7, otherwise.” Since “else” means “otherwise” in the if/else statement, it is okay to use “else” in this case.

Let’s say that the last sentence is “Multiply $n$ by 7 if $n > 15$” instead. If the top two conditions are not satisfied, it does not mean that that last condition is satisfied, as there exists $n = 11, 12, 13, 14, 15$ where they do not satisfy any of the three conditions. In this case, there is a small window between 11 and 15, inclusively, that the problem does not specify what to do with $n$. If we write “else” there and $n = 12$ for example, our code will enter the “else” condition and multiply $n$ by 7, even though the problem did not tell us to do so (thus, we incorrectly solve the problem).

With all of the items considered, we decide that it is appropriate to use “else,” thus our codes are:

```python
1 else:  # otherwise
2 n = 7 * n  # multiply the value of n by 7 then assign to itself
```
A line to show the final value of $n$ to check that our if/else statement works:

This is also straight-forward.

```python
1 n # show the value of n
```

**Let’s put everything together** and apply to $n = 3$.

Since $n = 3$, it falls into the category of $0 \leq n \leq 10$. Thus, we expect the value to be multiplied by 5, and it is what we observe.

```python
1 n = 3 # assign a value to n
2
3 if n < 0: # if n < 0
4   n = 3 * n # multiply the value of n by 3 then assign to itself
5 elif n >= 0 and n <= 10: # if n is between 0 and 10, inclusively
6   n = 5 * n # multiply the value of n by 5 then assign to itself
7 else: # otherwise
8   n = 7 * n # multiply the value of n by 7 then assign to itself
9
10 n # show the value of n
```

15

**Example #2: It is not required to have “elif” or “else” (illustration for #4a)**

We have the following problem statement:

```python
print n if n > 10
```

From the problem statement, we know we need two items:

1. A line to assign a value to $n$ (the value that was given prior to if/else statement)
2. A 1-part if/else statement, only one for $n > 10$

**A line to assign a value to $n$:**

This is straight-forward.

```python
1 n = some_value # assign a value to n
```
A 1-part if/else statement, only one for $n > 10$:

This is also straight-forward.

```python
1 if n > 10:  # if n > 10
2 print n      # print the value of n
```

Putting the two together and test for $n = 20$ (which is $n > 10$), we have:

```python
1 n = 20       # assign a value to n
2
3 if n > 10:   # if n > 10
4 print n      # print the value of n
```

Since $n > 10$, the computer enters the “if” statement to print the value of $n$, which is 20.

However, the interesting question here is: what if $n < 10$? We will test for $n = 5$. If $n = 5$, we have:

```python
1 n = 5       # assign a value to n
2
3 if n > 10:  # if n > 10
4 print n     # print the value of n
```

As we can see, if $n < 10$, the computer will not enter the “if” statement. The “print” statement is therefore not executed. Since there is no other command line that is applicable to this condition, no output is produced in the green box.

As you can see here, the “elif” and “else” commands are not needed as they are not applicable to the problem statement. They are not as crucial to the statement as the “if” command.
Example #3: “if” always starts a brand new if/else statement (illustration for #4b)

Let’s use the problem statement of example #1, with a slight modification.

1. Multiply \( n \) by 3 if \( n < 0 \)
2. Multiply \( n \) by 5 if \( 0 \leq n \leq 10 \)
3. Multiply \( n \) by 7, otherwise

We will skip the problem set up (as it is similar to that from example #1), but there is a slight modification: we will use “if” instead of “elif” for \( 0 \leq n \leq 10 \).

We will use the new set up to test for \( n = -5 \).

If \( n = -5 \), we would expect the \( n \) value to be multiplied by 3 and become -15. Let’s take a look:

```python
1 n = -5    # assign a value to n
2
3 if n < 0:    # if n < 0
4    n = 3 * n  # multiply the value of n by 3 then assign to itself
5 if n >= 0 and n <= 10:    # if n is between 0 and 10, inclusively
6    n = 5 * n  # multiply the value of n by 5 then assign to itself
7 else:    # otherwise
8    n = 7 * n  # multiply the value of n by 7 then assign to itself
9
10 n    # show the value of n
```

-105

It is not the \(-15\) that we expected because the second “if” splits our “3-parts” if-else statement into two different statements:
if n < 0:  # first if/else statement
    n = 3 * n

if n >= 0 and n <= 10:  # second if/else statement
    n = 5 * n
else:
    n = 7 * n

So, as our n value goes through the commands, it passes through two different if/else statement:

1. **First if/else statement**: This if/else statement states that if n < 0, multiply it by 3. If not, do nothing (remember that “elif” and “else” are not necessary, as shown in example #2).
   - Since n = -5, n < 0. As a result, it is multiplied by 3 → \( n = -15 \).

2. **Second if/else statement**: This if/else statement states that if 0 ≤ n ≤ 10, multiply it by 5. Otherwise, multiply by 7. This “else”
   - It is important to note that by the time n arrives at this second if/else statement, its value is -15 (NOT -5).
   - Since n = -15, it falls into the “else” condition. As a result, it is multiplied by 7 → \( n = -105 \).

At the end of our commands, we obtain: \( n = -105 \).

**As you can see, we illustrate that “if” always starts a brand new if/else statement.**

### Most common errors involving if/else statement:

1. You have incorrect indentation, causing incorrect read and execution.
2. You have improper use of “if”, “elif” and/or “else”, leading to improper break in statement or the situation where not all conditions are accounted for.
3. You have improper ordering of conditions, causing incorrect read and execution.
4. You forget the “:” at the end of each line, causing a syntax error.
Chapter 7

Solving Commands

7.1 solve and assume Commands

7.1.1 solve Command

Command:

\[ \text{solve( equation, var)} \]

where

- \( \text{equation} \) is the equation you are trying to solve for, and you can use any of the following three formats (e.g. you are trying to solve for \( f(x) = 0 \), and \( f(x) = 2x \))
  - \( f(x) == 0 \)
  - \( f == 0 \)
  - \( 2^x == 0 \)

- \( \text{var} \) is the independent variable that you are trying to solve for

Let’s do an example. Let’s say we want to solve for the equation:

\[ (x - 1) * (x + 1) = 0 \]

Our commands will be:
7.1. solve and assume Commands

```python
1 sol = solve( (x-1)*(x+1) == 0, x)  # solve the equation and assign the solutions to the variable sol
2 sol  # display the solutions
[ x == -1, x == 1 ]
```

Note: The output of the `solve` command is a list. As a result, the number of solutions is equal to the number of the elements of the resulting list.

In our case, to see the number of solutions, we can use:

```python
1 sol = solve( (x-1)*(x+1) == 0, x)  # solve the equation and assign the solutions to the variable sol
2 len(sol)  # check the length of the list, which is equal to the number of solutions
2
```

Common mistakes encountered for the `solve` command:

1. You forgot to use “`var(...)`” when necessary.
2. You confuse between “`=`” and “`==`”. Remember in Cocalc:
   - “`=`” means assigning.
   - “`==`” means equal.

7.1.2 assume Command

If you remember from Algebra 2, there are functions where there exist solutions in the imaginary space. For example, the solutions to the following equation are:

\[ f(x) = -x^4 + 5x^2 + 36 \rightarrow x = -3, 3, -2i, 2i \]

In the context of this class, there will be times where you want to filter out solutions that are imaginary or negative, since it is not possible to have imaginary or negative populations. This is
where the *assume* command comes in.

The purpose of the *assume* command is to filter the solutions.

**Command:** assume(*insert_condition_here*)

**Note:** You have to use the *assume* command **BEFORE** you solve the equations.

Two very common assume commands that we will use in this class will be:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>assume(x, &quot;real&quot;)</code></td>
<td>only take “real” solutions, and eliminate all imaginary ones</td>
</tr>
<tr>
<td><code>assume(x &gt;= 0)</code></td>
<td>only take positive solutions, and eliminate all negative ones</td>
</tr>
</tbody>
</table>

Let’s apply the *assume* command to the example shown in the beginning of this subsection.

```
1 assume(x, "real")       # assume that the solutions are real
2 assume(x >= 0)          # assume that the solutions are positive
3
4 f(x) = -x^4 + 5*x^2 + 36   # define the function f(x)
5
6 sol = solve( f(x) == 0, x) # solve the equation and assign the solutions to the variable sol
7
8 sol                       # show the solutions
 [ x == 3 ]
```
Sometimes, we will run into a situation when the *solve* command cannot solve the equation at all. This is especially more prominent when we have an equation that contains more than one type of functions (e.g. a polynomial and an exponential, a polynomial and a sinusoidal function, etc.)

For example, we are trying to solve the equation:

\[ e^x + x^2 = 1 \]

If we try to use CoCalc, here is what we have:

```python
1 solve (eˆx + xˆ2 == 1, x)

[ x == -sqrt(-eˆx), x == sqrt(-eˆx) ]
```

These solutions are clearly not what we are looking for. This is where the *find_root* command will be useful.

The *find_root* command will give us a numerical approximation of one solution, given that we provide the command an interval that contains that particular solution of interest.

**Command:**

```
find_root( expr, min, max)
```

where

- *expr* is the expression of the equation (i.e. there is no equal sign, see note below)
- *min* is the minimum value of the range on which you are guessing there is a root
- *max* is the maximum value of the range on which you are guessing there is a root

**Note:** Finding root implies that you are interested in finding the “x” values where the function is equal to 0. Therefore, if we are trying to find something like \( f(x) = a \) (where \( a \) is a constant), we HAVE to:

1. Transform the equation \( f(x) = a \rightarrow f(x) - a = 0 \)
2. Put the entire \( f(x) - a \) in the position of *expr*
We strongly recommend that you **GRAPH THE FUNCTION FIRST** to get a sense of where the solutions are located. Then, we will choose the range that contains **ONLY ONE** possible solution.

Let’s try to use `find_root` to solve the equation:

\[ e^x + x^2 = 1 \]

**First, plot the function:**

```plaintext
1 plot (e^x + x^2, (x, -1, 1)) # plot the function
```

We are looking at where the graph of the solution cuts the imaginary horizontal line of \( y = 1 \) (since we are trying to solve for \( f(x) = 1 \)). There seems to be two solutions here: one around 0, and one around -1.

Let’s say that we are only interested in the one around -0.7.

**Second, pick an appropriate range:**

It is important that we pick a range that only contains the root of interest. If we pick the range \(-1 \leq x \leq 1\), there are two solutions: \( x \approx -1 \) and \( x \approx 0 \). Thus, we have to pick something smaller than that. We will pick the range: \(-1 \leq x \leq -0.5\).

**Third, rearrange the expression:**

\[ e^x + x^2 = 1 \rightarrow e^x + x^2 - 1 = 0 \]

Thus, our `expr` is \( e^x + x^2 - 1 \).

**Last, solve the equation:**

Our command is then:
If we use a range that contains more than one root, we will get the root that is **most positive**, **not** every root within that range (or the root that we are interested in).

So, if we use the range of $-1 \leq x \leq 1$, the solution that we will get is:

```latex
\begin{align*}
\text{find\_root}( e^x + x^2 - 1, -1, 1) & \# \text{ find the root within the indicated range} \\
4.9413955998026013e-14 & \\
\end{align*}
```

which is around 0, not around -1, like we wanted.

### 7.3 Arrays and \texttt{desolve\_odeint} Command (Simulation)

#### 7.3.1 Arrays

Before we can discuss the \texttt{desolve\_odeint command}, it is beneficial that we talk about \textit{arrays}, which is the type of output of \texttt{desolve\_odeint}. The knowledge obtained in this section will be important as we further utilize the solutions obtained from using \texttt{desolve\_odeint}.

#### 7.3.1.1 Define an Array

**THE ACTUAL COMMAND TO DEFINE AN ARRAY IS NOT IN THE SCOPE OF THIS COURSE.**

An array is a table, where we have a certain number of rows and a certain number of columns. It looks something like this:

\[
\text{array ( [ [ a_{1,1}, a_{1,2}, \ldots, a_{1,n} ], [ a_{2,1}, a_{2,2}, \ldots, a_{2,n} ], \ldots, [ a_{m,1}, a_{m,2}, \ldots, a_{m,n} ] ] )}
\]

where:
- \( m \) is the number of row
- \( n \) is the number of column
- \( a_{m,n} \) is the entry at the \( m^{th} \) row and \( n^{th} \) column

If we look closely, we see that an array is very much like a list, except it is a list of a list, where:

1. Each row is a list, such as \([ a_{1,1}, a_{1,2}, ..., a_{1,n} ]\)

2. Then, we will take all the rows and put them together in a list of the rows, to get a the table/array.

### 7.3.1.2 Indexes of the Elements in an Array

As stated above, an array is a list of a list. So, just like how we use indexes to recall a particular element within a list, we can use indexes to recall a particular element within an array. Except now, we have to use **two different indexes** to specify **one location**: one for the row and one for the column.

**Command:**

\[
\text{array name}[\text{row index, column index}]
\]

where:

- \( \text{array name} \) is the variable to which an array is assigned
- \( \text{row index} \) is the index of the row where the element is located
- \( \text{column index} \) is the index of the column where the element is located

**Two important notes regarding indexes:**

1. The index rules are the same as those of the lists (e.g. the index of the first row/column is 0, and so on).

2. If we want to extract elements from all rows of a particular column (or from all columns of a particular row), we can use “:” as an index (where “:” will indicate “all”).

We will do three examples to illustrate these points.

For all of the examples, we will be looking at an array, \( a1 \):

\[
a1 = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}
\]
Situation #1: Extracting one element from a particular location
In this case, we want to extract element “5” from \textit{a1}. We know that the element “5” is located at:

- 2\textsuperscript{nd} row $\rightarrow$ \textit{row\_index} is 1
- 2\textsuperscript{nd} column $\rightarrow$ \textit{column\_index} is 1

Thus, to extract element “5”, we will use:
Solving Commands

1. a1[1, 1] # extract element “5”

Situation #2: Extracting one row

To extract a row, we have to extract elements from ALL columns within that row.

In this case, we want to extract the top row from a1. We know that to extract the top row, we have the following indexes:

- 1st row → row_index is 0
- all columns → column_index is “:”

Thus, to extract the top row, we will use:

```
1 a1[ 0, :] # extract the top row
```

array([[ 1, 2, 3]])

Situation #3: Extracting one column

To extract a column, we have to extract elements from ALL rows within that column.

In this case, we want to extract the right-most column from a1. We know that to extract the right-most row, we have the following indexes:

- all rows → row_index is “:”
- 3rd column → column_index is 2

Thus, to extract the right-most column, we will use:
7.3. Arrays and desolve odeint Command (Simulation)

```python
1 a1[:, 2] # extract the right-most column

array([3, 6, 9])
```

**Note:** When we extract all rows/columns, we will get a list that contains the elements from the row or column of interest. We will **NOT** get that exact row or column, in the exact same formatting (e.g. we will not get a vertical column - we will get a list containing the elements from that column instead).

### 7.3.2 desolve odeint Command

**Command:**

```python
desolve_odeint ( [diffeq1, diffeq2, ...], ics = [ini1, ini2, ...], dvars = [var1, var2, ...],
                 times = time_list )
```

where

- `diffeq1, diffeq2, ...` are the differential equations that we are trying to simulate
- `ini1, ini2, ...` are the initial values of the variables (the initial condition of the system)
- `var1, var2, ...` are the state variables
- `time_list` is a sequence of time points at which a solution must be found

- The purpose of this command is to simulate the differential equations to give us approximations of the states of the system, quite similar to the Euler’s method (but not exactly the same - the exact details are beyond the scope of this class).

- The output of this command is an array, where each column is a list of the values of corresponding state variable (in the order of the variables defined in “dvars”) over time.

**Note:**

1. If we only have one variable, we don’t need to use any bracket for this command.

2. There is **NO** time point included in the array. The array is **PURELY** a table consisting of the states of the system.

Let’s do an example. Let’s say we are modeling the Romeo-Juliet love for each other:
\[
\begin{cases}
R' = -J \\
J' = R
\end{cases}
\]

Our initial condition of the system will be: \( R = 1 \) and \( J = 1 \).

We also like to find solutions at the time points from \( t = 0 \) to \( t = 100 \) in \( \Delta t = 0.1 \).

We have the following items:

- We like to find solutions at the time points from \( t = 0 \) to \( t = 100 \) in \( \Delta t = 0.1 \)
  \( \rightarrow \) \text{time\_list} = 
  \text{srange}(0, 100, 0.1)

- Our variables are: \( R \) and \( J \). Let’s just pick
  - \text{var1} is \( R \)
  - \text{var2} is \( J \)

- We have our differential equations. Since:
  - \text{var1} is \( R \rightarrow \text{diffeq1} \) \( \Rightarrow \) \text{diffeq1} = \( R'_1 = -J \)
  - \text{var2} is \( J \rightarrow \text{diffeq2} \) \( \Rightarrow \) \text{diffeq2} = \( J'_1 = R \)

- We have our initial condition. Since:
  - \text{var1} is \( R \rightarrow \text{ini1} \) \( \Rightarrow \) \text{ini1} = \( R_0 = 1 \)
  - \text{var2} is \( J \rightarrow \text{ini2} \) \( \Rightarrow \) \text{ini2} = \( J_0 = 1 \)

Our codes are then:

```python
1 var('R, J') # define R and J as symbolic CoClac variables
2
3 time_list = srange(0, 100, 0.1) # define the list of time points to obtain solutions
4
5 sol = desolve_odeint([ -J, R], dvars = [R, J], ics = [1, 1], times = time_list) # simulate the model and assign the results to the variable sol
```

\textbf{Note:} There are \textbf{four ways} you can define the differential equations to put into the positions of \textit{diffeq1}, \textit{diffeq2}, etc.

We will use the Romeo-Juliet example for demonstration. In the position of \textit{diffeq1}:

1. define \( f1(R, J) = -J \rightarrow \textit{diffeq1} \) is \( f1(R, J) \)
2. define \( f1(R, J) = -J \rightarrow \textit{diffeq1} \) is \( f1 \)
3. define \( f1 = -J \rightarrow \textit{diffeq1} \) is \( f1 \)
We will not show the results for the command since we are much less interested in the actual values, for most of the time. We are mostly interested in graphing the results, in the format of a time series or a trajectory.

### 7.3.2.1 Graphing a Time Series

A time series is a graph produced when we plot the values of the state variables against particular time points.

We know that desolve_odeint will give us an array that consists of the state values, where each column represents a particular variable.

We have the option to extract the column representing the variable of interest, in the final form of a list, and then use list_plot to plot the data. However, we know that if we do that (use list_plot to plot only that list), we will be plotting the elements against their own indexes, and not against the time points that they are supposed to.

Thus, we need to use “zip” to combine data from the time_list and the list containing the values of the variable of interest.

In the case of Romeo-Juliet model, our commands to plot the time series (given that all the other codes are already in place) will be

For Romeo Time Series: We know that “R” is the first variable in “dvars.” Thus, the column containing the states of Romeo’s will be the first column, which can be extract using \(sol[:,0]\).

1  list_plot( zip(time_list, sol[:,0]) )  # time series for Romeo

For Juliet Time Series: We know that “J” is the second variable in “dvars.” Thus, the column containing the states of Juliet will be the second column, which can be extract using \(sol[:,1]\).

1  list_plot( zip(time_list, sol[:,1]) )  # time series for Juliet

### 7.3.2.2 Graphing a Trajectory

In this case, we are plotting the state variables against each other, and there are two ways we can do this.

Two ways to graph the trajectory: Assuming that \(sol\) is the variable to which we assign the output from desolve_odeint (we can also use any other variable as we like)
1. list_plot(sol)

   - This works because sol is an array, which is a list of a list. Each inside list is actually a point that specifies the state of the system (a point on a graph can be specified by a list, e.g. [x, y, z]). So by using list_plot(sol), we are plotting a list that contains smaller lists, which are points of interest.

2. list_plot( zip(sol[:,0], sol[:,1], etc. ) )

   - This is slightly more complicated but a little more intuitive, since we can tell right away that we are extracting the list for each variable (via sol[:,0], sol[:,1], ...) and then plot them against each other to produce the trajectory.

In the context of our example, our plotting commands can be (given that all other commands are already in place):

```
list_plot( sol ) # plot the trajectory
```

OR

```
list_plot( zip(sol[:,0], sol[:,1]) ) # plot the trajectory
```

Either method will work.

## 7.4 dde_solve Command

For a lot of the times during the course, we will encounter ordinary differential equations. However, there are also times when we will need to use delayed different equations to reflect what is going on biologically (e.g. the hormone levels and response in the body). Unfortunately, the desolve_odeint command will not be able to solve such complex systems. This is why we have a function that is written by Dr. William Conley just for our class.

This tool is built to solve time-delay different differential equation.

**Note:** You **MUST** load the codes, as instructed in the lab PDF file, **BEFORE** you can use the `dde_solve` command.
where

- `eqn` is the delayed differential equation, with all time-delayed terms substituted with new variables (see special notes below)
- `statevar` is the state variable of the equation
- `delayedvars` is the dictionary containing the variables, that are substituted for the time-delayed terms, and their respective delayed time values (see special notes below)
- `history` is the initial condition
- `tmax` is the end time that you want to run the simulation to
- `timestep` is the step size of the time stamps

Special notes regarding time-delayed terms:

Due to the construction of the function, we will SUBSTITUTE ALL $X(t - \tau)$ with NEW VARIABLES, and then put those variables along with the associated $\tau$ value in a DICTIONARY, in the following format:

$$\{ \text{var1:} \tau_1, \text{var2:} \tau_2, \text{etc.} \}$$

As you can see, we need a different variable for each $\tau$ value.

Also, do NOT forget to define these variables `var1`, `var2`, etc. with the `var(...)` command.

Note:

1. This command **always** starts the simulation at $t = 0$.

2. The output of this command is a list, that contains only state values of the system and does NOT have any associated time marks. Hence, if we want to plot the time series, we will need to make a new list of time stamps and zip the two lists later, appropriately.

3. $X(t)$ is the expression for $X$ as a function of $t$. It is **NOT** $X \cdot t$. This also applies to the rest of the time-delayed terms of $X(t - \tau)$.

Recommend protocol for working with dde_solve:

1. Substituting all $X(t - \tau)$ terms in the equation

2. Make the dictionary, with the appropriate $\tau$ values being associated with the appropriate variables

3. Put everything into the dde_solve command
For example, let’s say that we want to simulate the following equation:

\[ X'(t) = 2 \cdot X(t) \cdot X(t-2) \cdot X(t-5) \]

We want to simulate from \( t = 0 \) to \( t = 3 \), with \( X(0) = 1 \) and \( \Delta t = 0.1 \).

**Step #1: Substituting all \( X(t-\tau) \) terms in your equation**

In this case, we have two time-delayed terms: \( X(t-2) \) and \( X(t-5) \).

Also, please remember that \( X(t) \) is simply just \( X \).

We will define the new variables:

\[ X1 \equiv X(t-2) \]
\[ X2 \equiv X(t-5) \]

Now, our equation becomes: \( X' = 2 \cdot X \cdot X1 \cdot X2 \)

**Step #2: Make the dictionary, with the appropriate \( \tau \) values being associated with the appropriate variables**

Since we define our new variables to be:

\[ X1 \equiv X(t-2) \]
\[ X2 \equiv X(t-5) \]

We have:

- \( \tau_1 = 2 \), and \( \tau_1 \) is associated with \( X1 \)
- \( \tau_2 = 5 \), and \( \tau_2 \) is associated with \( X2 \)

Our dictionary, \{ \textit{var1} : \textit{\tau1}, \textit{var2} : \textit{\tau2}, etc. \}, is then:

\{ \textit{X1} : 2, \textit{X2} : 5 \} 

**Step #3: Put everything into your dde_solve command**

We have:

- \textit{eqn} is the modified delayed differential equation, \( 2 \cdot X \cdot X1 \cdot X2 \)
- \textit{statevar} is the state variable, \( X \)
• `delayedvars` is the dictionary \{\textit{X1} : 2, \textit{X2} : 5\}

• `history` is the initial condition, \(X(0) = 1\)

• `tmax` is the end time that we want to run the simulation to, \(t = 3\)

• `timestep` is the step size of the time stamps, \(\Delta t = 0.1\)

Putting together, we have our commands:

```python
1 var(’X, X1, X2’) # define X, X1, X2 as CoCalc symbolic variables
2 sol = dde.solve( 2*X*X1*X2, X, {X1:2, X2:5}, 1, 3, 0.1) # solve the delayed differential equation and assign the solutions to the variable sol
```

Now, let’s say that we want to plot the time series. We know that we have to zip this list (which has no time stamp) with a list containing the appropriate time stamps.

We know that we are generating data from \(t = 0\) to \(t = 3\) in \(\Delta t = 0.1\). Thus, our time list will be:

\[
t\text{list} = \text{srange}(0, 3.1, 0.1)
\]

Our final commands are:

```python
1 var(’X, X1, X2’) # define X, X1, X2 as CoCalc symbolic variables
2 t_list = srange( 0, 3.1, 0.1) # create a time list
3 sol = dde.solve( 2*X*X1*X2, X, {X1:2, X2:5}, 1, 3, 0.1) # solve the delayed differential equation and assign the solutions to the variable sol
4 list_plot( zip( t_list, sol), plotjoined = True, axes_labels = [“Time”, “X”] ) # plot the time series
```
Chapter 8

Special Effects Commands - Interactive and Animation

8.1 Interactive

There are times when we are interested in see how changing a certain parameter will change the behavior or the visual display of the function. However, we do not want to keep having to writing new lines of codes that are so similar to each other and then just to look at one small change. That is way too much effort. This is where “interactive” will come in handy.

The “interactive” tool will allow for simple manipulation of parameters and real-time observations of the effects by using sliders.

In a sense, it is a “function” that has the following additional features:

- The interactive parameters - which include BOTH the function parameters (e.g. slope, power...) and the window parameters (i.e. xmin, xmax, ymin, ymax of the graph)
- The “show” command to display the output

**Command:**

```python
@interact
def func_name( var1 = (min1, max1, step1), var2 = (min2, max2, step2), etc.):
    interaction body
    show(final_var)
```

where

- **func_name** is the name of the “interactive” function
- **var1, var2, ...** are the parameters that we like to vary within the interaction body
• \(min_1, min_2, \ldots\) are the minimum values for the respective parameters

• \(max_1, max_2, \ldots\) are the maximum values for the respective parameters

• \(step_1, step_2, \ldots\) are the step sizes for the respective parameters, as they change from the minimum values \((min_1, min_2, \ldots)\) to the maximum values \((max_1, max_2, \ldots)\)

• \texttt{final\_var} is the variable to which the final answer, that we want to show (e.g. a plot or a value), is assigned
  - There can be \textbf{more than one} “show” lines, if we have more results that we like to display

\textbf{Note:} If we want to plot a vector with the interactive tool, we have to add the following line as an input to our command:

\[
\text{target = vector([1,1])}
\]

In other words, our commands then become:

```python
@interact
def func_name( var1 = (min1, max1, step1), etc., target = vector([1,1])):
    interaction body
    show( final\_var)
```

Let’s do a quick example. Let’s say that we want to see how the slope of the line will affect the steepness of its graph, visually.

To test this, we will go from slope of 0 to slope of 10, inclusively, and display the graph in the following window:

• 0 \(\leq\) \(x\) \(\leq\) 10

• 0 \(\leq\) \(y\) \(\leq\) 100

Our commands are then:

```python
1 @interact # start the interactive tool  
2 def steepness ( m = (0, 10) ): # change the parameter m from 0 to 10  
3 f(x) = m*x # define the function to plot  
4 p = plot( f(x), (x, 0, 10), ymin = 0, ymax = 100) # assign the plot to p  
5 show(p) # show the graph
```
8.2 Animation

While interactive is a great tool to show us changes in the graphs/values of the functions as the parameters are manipulated, there are certain times the changes are relatively difficult to observe. Those changes are best observed when we have all the plots side by side and flip through them really quickly. This is the basis of an “animation.”

8.2.1 A Brief Introduction to Animation

We believe that the best way to remember on how to write an animation in CoCalc is to quite frankly remember what an animation in real life is.

Animation, simply speaking, is a process that will allow for a “transformation” of static images to moving images. Nowadays, animation can be done by the computer, but perhaps, the best demonstration of the process lies in the old-school hand-drawn animated movies that we all love watching as we grow up - The Lion King (1994), Aladdin (1992), Mulan (1998), etc.

For each of those old-school hand-drawn animated movies, a group of artists will draw a lot of scenes, where consecutive scenes are just barely different from one to the next. Let’s say that we have a character that is trying to wave at someone else. That one action can take more than 20 drawings to make. However, when we dissect the scenes down to frame by frame, perhaps the only difference from frame 1 to frame 2 is the position of the arm, where the arm is slightly elevated in frame 2, relative to frame 1, as the person is raising their arm prior to waving at someone else.

Then, to allow for a “transformation” of static images to moving images, they will simply flip through the scenes at a rate fast enough that can fool our eyes. It is known that our eyes can only process 10-12 images per second, so if we show more than 12 scenes per second, our eyes will process the differences to be continuous motion. This is why movies are often produced at a rate of 16 or 24 frames per second.

If you like to find out more, you can watch this little clip here, showing how the cartoon flip book works to produce continuous motion: https://www.youtube.com/watch?v=UGsOeY9rW9A.

8.2.2 The Animation Command

We will carry this knowledge of how to produce an animated movie to apply to the process of building our animation.

We know that to make an animated movie, we will need to:

1. Have a blank sketchbook to store all the frames
2. Keep adding frames to the sketchbook
3. Then, flip through the sketchbook

Our commands will also then reflect these three main parts:
### 8.2. Animation

<table>
<thead>
<tr>
<th>Animated Movie</th>
<th>Animation Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>A sketchbook</td>
<td>An empty list to store all the plots</td>
</tr>
<tr>
<td>Keep adding frames</td>
<td>A <em>for loop</em> to draw new frames and add the frames to the list (that is the “sketchbook”)</td>
</tr>
<tr>
<td>Flip through the sketchbook</td>
<td>The “animate” line</td>
</tr>
</tbody>
</table>

Let’s build our general command.

```python
# create the flip book - but you can name whatever you want
flip_book = []

# start the for loop to add frames to the flip book
for var in list1:
    # perform tasks if needed
    perform certain tasks

    # draw a new frame and assign the frame to the variable new_frame
    new_frame = plotting commands

    # add the new frame to the flip book
    flip_book.append(new_frame)

# animate the frames and assign the animation to “a”
a = animate(flip_book)

# show the animation
show(a)
```

For the last line, we can also use the following command to obtain an animation in the format of gif (moving picture) rather than a video:

```python
show(a, gif = True)  # show the animation
```

Let’s use the example of the slopes of the lines in Section 7.1. As stated previously, we want to see how the slope of the line will affect the steepness of its graph, visually. To test this, we will go from slope of 0 to slope of 10, inclusively, and display the graph in the following window:

- $0 \leq x \leq 10$
- $0 \leq y \leq 100$

We know that for each frame, we want to plot a graph with a new slope. Thus, we have:
• Our *perform certain tasks* will be to define a function with a particular slope that will later be graphed

• We will then plot the graph and assign to the variable *new_frame*

Our commands will look something like this:

```python
4  f(x) = m*x  # define the function
5  new_frame = plot(f(x), (x, 0, 10))  # draw a new frame and assign the frame
```

As we are varying \( m \), we know that \( var \) also changes from loop to loop. Thus, it will be smart to make \( var \equiv m \).

From the mechanism of the *for loop*, we know that the values of \( var \) is taken from \( list1 \).

Since we also have \( var \equiv m \), it is necessary that \( list1 \) contains elements that are the values of \( m \), i.e. the slopes. Hence, \( list1 = srange(0, 10.1, 0.1) \)

Thus, our commands are:

```python
1  flip_book = []  # create the flip book
2
3  for m in srange(0, 10.1, 0.1):  # start the for loop
4    f(x) = m*x  # define the function
5    new_frame = plot(f(x), (x, 0, 10))  # draw a new frame and assign the frame
6    flip_book.append(new_frame)  # add the new frame to the flip book
7
8  a = animate(flip_book)  # animate the frames and assign the animation to “a”
9
10 show(a)  # show the animation
```
Chapter 9

Vector Commands

9.1 Defining a Vector

**Command:**

```
vector([val1, val2, etc.])
```

where

- `val1, val2,` etc. are the coordinates of the vector

(see example below to also include plotting a vector)

9.2 Plotting a Vector

**Command:**

```
plot( vector([val1, val2, etc.]), color = "insert_color_here", thickness = insert_value_here,
aspect_ratio = insert_value_here )
```

where

- `val1, val2,` etc. are the coordinates of the vector
- `color` will define the color of the vector - default is blue
- `thickness` will define the thickness of the vector
- `aspect_ratio` will scale your graph appropriately
- A value less than 1 will make your “1 unit” on the horizontal axis shorter than your “1 unit” on the vertical axis.
- A value equal than 1 will make your “1 unit” on the horizontal axis equal to your “1 unit” on the vertical axis.
- A value greater than 1 will make your “1 unit” on the horizontal axis longer than your “1 unit” on the vertical axis.

For example, let’s say we want to plot a red vector (1,2) with a thickness of 3, our commands are:

```
1 v1 = vector(1,2)            #define the vector and assign to variable v1
2 plot(v1, color = “red”, thickness = 3)      #plot the vector
```
Chapter 10

Matrix Commands

10.1 Defining a Matrix

10.1.1 Traditional Method (Row Matrix)

**Command:**

```
matrix( field, [ row1, row2, etc. ] )
```

where

- *field* is the field over which our matrix is defined over. It is often an optional component of the comment. However, if we need to compute the eigenvalues and eigenvectors of the matrix, we **MUST** use the field of RDF (we will only use “RDF” in this class).
- *row1, row2*, etc. are the lists that constitute the particular rows of the matrix of interest.

For example, if we like to write the following matrix using the row matrix method:

\[
\begin{bmatrix}
1 & 2 \\
3 & 4
\end{bmatrix}
\]

We know that:

- **First row:** [1, 2]
- **Second row:** [3, 4]

Hence, our commands will be:
1. Matrix Commands

```matlab
matrix([ [1,2], [3,4] ])  # define the matrix
```

\[
\begin{bmatrix}
1 & 2 \\
3 & 4
\end{bmatrix}
\]

However, as mentioned above, we will not be able to get eigenvalues and eigenvectors from this matrix, because we need to use the field “RDF”. To define a matrix over this field, our commands will be:

```matlab
matrix( RDF, [ [1,2], [3,4] ])  # define the matrix
```

\[
\begin{bmatrix}
1 & 2 \\
3 & 4
\end{bmatrix}
\]

10.1.2 Column Matrix

**Command:**

```matlab
column_matrix( field, [ col1, col2, etc. ] )
```

where

- `field` is the field over which our matrix is defined over. It is often an optional component of the comment. However, if we need to compute the eigenvalues and eigenvectors of the matrix, we **MUST** use the field of `RDF` (we will only use “RDF” in this class).

- `col1`, `col2`, etc. are the lists that constitute the particular columns of the matrix of interest.

This tool is especially useful when we like to construct matrices from vectors, with each vector constituting a column of the matrix.

For example, if we like to write the following matrix using the column matrix method:
10.1. Defining a Matrix

We know that:

- **First column:** [1, 3]
- **Second column:** [2, 4]

Hence, our commands will be:

```
column_matrix( [ [1,3], [2,4] ] ) # define the column matrix
```

However, as mentioned above, we will not be able to get eigenvalues and eigenvectors from this matrix, because we need to use the field “RDF”. To define a matrix over this field, our commands will be:

```
column_matrix( RDF, [ [1,3], [2,4] ] ) # define the column matrix
```

10.1.3 The Zero Matrix

**Command:**

```
zero_matrix(field, insert_number_of_row, insert_number_of_column)
```

where

- **field** is the field over which your matrix is defined over. For this class, we will use **RR** for the field of the zero matrix.
In this case, the “RR” term allows us to define the zero matrix over the real number field (technical details are not important in the context of our course).

Two important notes regarding indexes:

1. This command will define a matrix where all of its entries are zeroes.
2. This tool will come in handy especially when we are trying to make a function that will multiply two matrices together.

For example, we want to define a zero matrix with 2 rows and 3 columns, our commands are then:

```
zero_matrix(RR, 2, 3)  # define the zero matrix with 2 rows and 3 columns
```

```
[0.00000  0.00000  0.00000]
[0.00000  0.00000  0.00000]
```

10.2 Getting the Dimensions

10.2.1 General Dimensions

Command:

```
matrix_name.dimensions()
```

- `matrix_name` is either the variable to which a matrix is assigned to or the actual matrix itself

For example, we have the matrix:

```
\[
\begin{bmatrix}
1 & 2 & 3 \\
4 & 5 & 6
\end{bmatrix}
\]
```

Our commands will be:

```
1 M = matrix( [[1, 2, 3], [4, 5, 6]] )  # define the matrix using the row method and
assign the matrix to the variable M
2 M.dimensions()  # get the dimensions of the matrix
```
10.2. Getting the Dimensions

(2,3)

Note: The output of this command is a **list** of number in the format of (number of rows, number of columns).

Now, the interesting part is what if we only want to know the number of columns or the number of rows.

10.2.2 Getting the Number of Rows

There are technically two ways you can do this: either use the dimensions command or the columns command (no this is not a typo). We will only discuss the option using the dimensions command because the second option is slightly trickier and is prone to give incorrect result if you are not careful.

In the first option, recall that the output of the dimension command is a list where the # of row is the 1st element, and the # of column is the 2nd element. Thus, if we only want to know the number of rows, we can simply use the dimensions command to get the list of dimensions, then use the index “0” to recall the 1st element (which is the number of rows).

**Command:**

\[matrix\_name\].dimensions()[0]

where

- **matrix\_name** is either the variable to which a matrix is assigned to or the actual matrix itself

Let’s use the same example from before.

\[
\begin{bmatrix}
1 & 2 & 3 \\
4 & 5 & 6
\end{bmatrix}
\]

Our commands to get the number of rows will be:

1. \[
M = \text{matrix}(\left[\begin{array}{ccc}1 & 2 & 3 \\ 4 & 5 & 6\end{array}\right]) \# \text{define the matrix using the row method and assign the matrix to the variable } M
\]

2. \[
M\text{.dimensions()}[0] \# \text{get the number of rows}
\]
10.2.3 Getting the Number of Columns

Similarly, there are technically two ways you can do this: either use the dimensions command or the rows command (there is no typo here). We will only discuss the option using the dimensions command because the second option is slightly trickier and is prone to give incorrect result if you are not careful.

In the first option, recall that the output of the dimension command is a list where the # of row is the 1\textsuperscript{st} element, and the # of column is the 2\textsuperscript{nd} element. Thus, if we only want to know the number of columns, we can simply use the dimensions command to get the list of dimensions, then use the index “1” to recall the 2\textsuperscript{nd} element (which is the number of columns).

**Command:**

\[
\text{matrix\_name}.\text{dimensions()[1]}
\]

where

- \text{matrix\_name} is either the variable to which a matrix is assigned to or the actual matrix itself

Let’s use the same example from before.

\[
\begin{bmatrix}
1 & 2 & 3 \\
4 & 5 & 6
\end{bmatrix}
\]

Our commands to get the number of rows will be:

1. \texttt{M = matrix( [[1, 2, 3], [4, 5, 6]])} \# define the matrix using the row method and assign the matrix to the variable M
2. \texttt{M.dimensions()[1]} \# get the number of columns
10.3 Recalling Elements from a Matrix

10.3.1 Recalling a Particular Entry from Matrix

A matrix, in one way or another, resembles an array, which in turn is the type of the output of the \texttt{desolve} command.

Thus, much like the \texttt{sol[a,b]} terms that we use previously to recall elements from the matrix, the command to recall a specific entry of a matrix is:

\begin{center}
\textbf{Command:}
\end{center}

\begin{center}
\texttt{matrix\_name [ row\_index, column\_index ]}
\end{center}

where

- \texttt{matrix\_name} is either the variable to which a matrix is assigned to or the actual matrix itself
- \texttt{row\_index} is the index of the row where the element is located
- \texttt{column\_index} is the index of the column where the element is located

For example, let's say we have the following matrix:

\begin{equation}
\begin{pmatrix}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 9
\end{pmatrix}
\end{equation}

Now, let's say we want to claim the value “6” from the matrix, which is in the:

- 2\textsuperscript{nd} row → \texttt{row\_index} is 1
- 3\textsuperscript{rd} column → \texttt{column\_index} is 2

Thus, our commands are:

1. \begin{verbatim}
M = matrix ( [ [1,2,3] , [4,5,6] , [7,8,9] ] )  # define the matrix and assign it to the variable M
\end{verbatim}

2. \begin{verbatim}
M[1,2]  # get the value in row 2, column 3
\end{verbatim}

6
**Note:** Recall that when we assign values to variables, they will only remember the latest values that were assigned to them. We can use that to our advantage to replace an element within a matrix.

Let’s use the example from above. We have the following matrix:

\[
\begin{bmatrix}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 9
\end{bmatrix}
\]

Now, rather than recalling and visualizing the value at the position “6”, we would instead like to put the number “14” in that position. Our commands are then:

```python
1 M = matrix ( [ [1,2,3] , [4,5,6] , [7,8,9] ] )  # define the matrix and assign it to the variable M
2 M[1,2] = 14  # assign the value 14 to the location of row 2, column 3
3 M  # view the new matrix
```

\[
\begin{bmatrix}
1 & 2 & 3 \\
4 & 5 & 14 \\
7 & 8 & 9
\end{bmatrix}
\]

### 10.3.2 Recalling a Particular Row from Matrix

**Command:**

```
matrix_name.row ( row_index )
```

where

- *matrix_name* is either the variable to which a matrix is assigned to or the actual matrix itself
- *row_index* is the index of the row where the element is located

Be careful that we use *parentheses* instead of *brackets*. 
10.3. Recalling Elements from a Matrix

**Note:** The output of this command is the row of interest, in the format of a list.

We will use the same matrix again:

\[
\begin{bmatrix}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 9
\end{bmatrix}
\]

Let’s say, we want to extract the third row, which has an index value of 2. Our commands are then:

1. \(M = \text{matrix} ( \begin{bmatrix} 1,2,3 \end{bmatrix} , \begin{bmatrix} 4,5,6 \end{bmatrix} , \begin{bmatrix} 7,8,9 \end{bmatrix} ) \) # define the matrix and assign it to the variable M
2. \(M\text{.row}(2)\) # get the second column

\((7, 8, 9)\)

10.3.3 Recalling a Particular Column from Matrix

**Command:**

\[
\text{matrix\_name}\text{.column}\left(\text{column\_index}\right)
\]

where

- \textit{matrix\_name} is either the variable to which a matrix is assigned to or the actual matrix itself
- \textit{column\_index} is the index of the column where the element is located

Be careful that we use parentheses instead of brackets.

**Note:** The output of this command is the column of interest, in the format of a list.

We will use the same matrix again:

\[
\begin{bmatrix}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 9
\end{bmatrix}
\]
Let’s say, we want to extract the second column, which has an index value of 1. Our commands are then:

1. \[ M = \text{matrix} ( \begin{bmatrix} 1,2,3 \end{bmatrix}, \begin{bmatrix} 4,5,6 \end{bmatrix}, \begin{bmatrix} 7,8,9 \end{bmatrix} ) \] # define the matrix and assign it to the variable M
2. \[ M.\text{column}(1) \] # get the second column

\[ (2, 5, 8) \]

10.4 Eigenvalues and Eigenvectors

10.4.1 General Command

There is a different method to find eigenvalues, but to be consistent with this course, it will not be shown.

The general command to get all eigenvectors and eigenvalues is:

**Command:**

\[ \text{matrix}\.name\.eigenvectors.right() \]

where

- \text{matrix}\_name is either the variable to which a matrix is assigned to or the actual matrix itself

**Note:** Do NOT forget the RDF part when defining the matrix.

For example, let’s say we want to get the eigenvalues and eigenvectors of the matrix,

\[
\begin{bmatrix}
2 & 0 \\
0 & 3
\end{bmatrix}
\]

Our commands are then:
10.4. Eigenvalues and Eigenvectors

M = matrix ( RDF, [ [2,0] , [0,3] ] )  # define the matrix and assign it to the variable M
M.eigenvectors_right()  # get the eigenvalues and eigenvectors

[(2.0, [(1.0, 0.0)], 1), (3.0, [(0.0, 1.0)], 1)]

Now, how to extract the eigenvectors and eigenvalues from the answer? To do so, it is important that we understand the output of this command very clearly.

10.4.2 Understanding the Output of eigenvectors_right()

Two important notes regarding the output of eigenvectors_right():

1. The output of eigenvectors_right is a LIST, with each element being a TUPLE containing eigenvalues, eigenvectors, and a third number that you do not need to know for this class

2. Within each TUPLE:
   (a) the first element is the eigenvalue
   (b) the second element is a LIST containing the TUPLE that has the coordinates that defines the eigenvector associated with that eigenvalue
   (c) the third element is not important in the context of this class (it is called a “multiplicity” for those who are curious)

Let’s dissect these statements are little more clearly. Our analysis will span over this and the following two sections as well, since using these statements, we will be able to obtain our eigenvalues and eigenvectors.

Part 1: “The output of eigenvectors_right is a LIST, with each element being a TUPLE containing eigenvalues, eigenvectors, and a third number that you do not need to know for this class”

If we look at our example from above, the output has two elements:

[ (2.0, [(1.0, 0.0)], 1.0) , (3.0, [(0.0, 0.1)], 1.0) ]

And just like any list, the part highlighted in light blue has an index value of 0, and the part highlighted in light orange has an index value of 1.

Thus, for our commands, to extract each component, we will have:
First component:

```
1  M = matrix ( RDF, [ [2,0] , [0,3] ] )  # define the matrix and assign it to the variable M
2  M.eigenvectors[0]  # get the first component of the output

(2.0, [(1.0, 0.0)], 1)
```

Second component:

```
1  M = matrix ( RDF, [ [2,0] , [0,3] ] )  # define the matrix and assign it to the variable M
2  M.eigenvectors[1]  # get the second component of the output

(3.0, [(0.0, 0.1)], 1.0)
```

Notice how they have parentheses around them? That signifies that they are of the type **LIST** (also known as **TUPLE**).

However, we are not interested in those tuples. We are more interested in the actual eigenvalues and eigenvectors. This is where the second part comes in, where we will have to go inside those tuples to extract what we want.

### 10.4.3 Obtaining the Eigenvalues

From the previous part, we know that to extract a tuple that contains the eigenvalue and the associated eigenvector, all we need is:

```
matrix_name.eigenvectors_right()[tuple_index]
```

where:

- `matrix_name` is either the variable to which a matrix is assigned to or the actual matrix itself
- `tuple_index` is the index of the tuple of interest

Then, we know that:
**Part 2a:** “Within each **TUPLE**: the first element is the eigenvalues”

Thus, to get the first element of a list, our index will always be 0. This index will get **tagged on at the end** of the previous command, which gives rise to our generic command:

**Command:**

```python
matrix_name.eigenvectors_right()[tuple_index][0]
```

where:

- **matrix_name** is either the variable to which a matrix is assigned to or the actual matrix itself
- **tuple_index** is the index of the tuple of interest

In this case, “**matrix_name.eigenvectors_right()[tuple_index]**” will give us the tuple of the interest, then the end “[0]” will extract the first element within that tuple, which is the eigenvalue.

Using the example from above, we can extract:

**The first eigenvalue:**

```python
1 M = matrix ( RDF, [ [2,0], [0,3] ] ) # define the matrix and assign it to the variable M
2 M.eigenvectors_right()[0][0] # get the first eigenvalue
```

2.0

**The second eigenvalue:**

```python
1 M = matrix ( RDF, [ [2,0], [0,3] ] ) # define the matrix and assign it to the variable M
2 M.eigenvectors_right()[1][0] # get the second eigenvalue
```

3.0
10.4.4 Obtaining the Eigenvectors

How about eigenvectors? This is where part 2b comes in handy.

**Part 2b:** “Within each TUPLE: the second element is a LIST containing the TUPLE that has the coordinates that defines the eigenvector associated with that eigenvalue”

A lot of people make a very common mistake here because they read that statement too fast. A lot will think that the second element is the eigenvector, when in fact, it **IS NOT**.

Let’s use our example to demonstrate. We will look at the first element of our output, which is: 

\[(2.0, [(1.0, 0.0)], 1.0)\]

The second element of that is \([(1.0, 0.0)]\). If we look closely, we see that there are two layers: (1.0, 0.0) that is contained within the “[ ]”.

In other words, it is a list buried within another list. The **outer list**, define by the “[ ]”, quite frankly only has **one element**, which is the **inner list** of (1.0, 0.0). That element is default to have the index of “0” and is what we are after.

To summarize, to get the coordinates that defines the eigenvector, we need to:

1. go to the element of interest (first eigenvector, second eigenvector, etc.)
2. extract the **second element** of that tuple to get a list that contains the coordinates of interest → index of 1
3. extract one more time to only get the coordinates contained in that list → index of 0

As demonstrated in the part above, these new indexes will just get tagged on at the end to give us the generic command:

**Command:**

\[matrix\_name.eigenvectors\_right()[tuple\_index][1][0]\]

where:

- **matrix\_name** is either the variable to which a matrix is assigned to or the actual matrix itself
- **tuple\_index** is the index of the tuple of interest

Using the example from above, we can extract:
The coordinates of the first eigenvector:

1 M = matrix ( RDF, [[2,0], [0,3]] )  # define the matrix and assign it to the variable M
2 M.eigenvectors_right()[0][1] # get the first eigenvector

(1.0, 0.0)

The coordinates of the second eigenvector:

1 M = matrix ( RDF, [[2,0], [0,3]] )  # define the matrix and assign it to the variable M
2 M.eigenvectors_right()[1][1] # get the second eigenvector

(0.0, 1.0)

10.4.5 Why Are the Coordinates of the Eigenvectors sometimes Non-Integers?

Now, let’s say that we have the following matrix:

\[
\begin{bmatrix}
3 & 2 \\
2 & 3
\end{bmatrix}
\]

If we calculate by hand, we know that our eigenvectors are: (1, 1) and (1, −1).

However, when we use CoCalc, we get:

1 M = matrix ( RDF, [[3,2], [2,3]] )  # define the matrix and assign it to the variable M
2 M.eigenvectors_right() # get the eigenvalues and eigenvectors
\[
(5.0, [(0.7071067811865476, 0.7071067811865474), 1],
(1.0000000000000004, [(-0.7071067811865474, 0.7071067811865476)], 1))
\]

The numbers are not very pretty...

**However, the first question is: are they the same eigenvectors as those you obtain?**

The answer is yes. Remember that: **any vectors that are multiple of each other are the same vector**.

Thus, if we:

1. take the vectors obtained from CoCalc: \((0.7071067811865476, 0.7071067811865474)\) and \((-0.7071067811865476, 0.7071067811865476)\)

2. divide them by \(0.7071067811865476\) and \(-0.7071067811865476\), respectively

We will obtain \((1, 1)\) and \((1, -1)\). As a result, we demonstrate that our eigenvectors and the CoCalc eigenvectors are multiples of each other, thus they are the same vectors.

**Then, the next question is: why can’t CoCalc just give us the nice vectors?**

The answer is:

The eigenvectors given by CoCalc have a length of 1.

Let’s look at one eigenvector in particular for example. We will choose the \((1, 1)\) vector. We have:

- The length of our vector \((1, 1)\) is:
  \[\sqrt{1^2 + 1^2} = \sqrt{2} \approx 1.4\]

- The length of the CoCalc vector \((0.7071067811865476, 0.7071067811865474)\) is:
  \[\sqrt{(0.7071067811865476)^2 + (0.7071067811865476)^2} = \sqrt{1} = 1\]
Chapter 11

Derivatives and Jacobian Matrices

11.1 Calculating the General Derivative

**Command:**

```
diff ( func_name(var), var )
```

where

- `func_name` is the function
- `var` is the independent variable of `func_name`

**Note:** You should write `func_name(var)` instead of just `func_name` when you take the derivative.

For example, let’s say we want to take the derivative of the function \( f(x) = \sin(x^2) \).

Our commands are:

```
1  f(x) = sin(x^2)  # define the function
2  df = diff( f(x), x)  # take the derivative and assign the answer to the variable “df”
3  df  # show the answer
```

\( 2x\cos(x^2) \)
11.2 Calculating the General Jacobian Matrix

**Command:**

\[
\text{jacobian} \left( \left[ \text{func}_1, \text{func}_2, \text{etc.} \right], \left[ \text{var}_1, \text{var}_2, \text{etc.} \right] \right)
\]

where:

- **func}_1, \text{func}_2, \text{etc.} are the differential equations, which can either be the expressions themselves or the variables to which the expressions are assigned
- **var}_1, \text{var}_2, \text{etc.} are the variables present in the differential equations, in respective order

**Note:** We can use “( )” instead of “[ ]”.

For example, if we like to find the Jacobian matrix for the system:

\[
\begin{align*}
  x' &= 2 \cdot x \cdot y + 2 \cdot y^2 \\
  y' &= 3 \cdot x^2 - y \cdot x
\end{align*}
\]

We know that:

- **func}_1 is: 2 \cdot x \cdot y + 2 \cdot y^2
- **var}_1 is: x
- **func}_2 is: 3 \cdot x^2 - y \cdot x
- **var}_2 is: y

To obtain the Jacobian matrix, we can either write:

```python
1 var('x,y') # define x,y as the CoCalc symbolic variables
2 jacobian ( [ 2*x*y + 2*y^2, 3*x^2 - y*x ], [x, y] ) # obtain the Jacobian matrix
```

\[
\begin{bmatrix}
  2y & 2x + 4y \\
  6x - y & -x
\end{bmatrix}
\]

(remember that the first \((x,y)\) output line comes from the command of \texttt{var(‘x,y’)})
OR

```python
var('x,y') # define x,y as the CoCalc symbolic variables
f1 = 2*x*y + 2*y^2 # assign the first expression to the variable f1
f2 = 3*x^2 - y*x # assign the second expression to the variable f2
jacobian([f1, f2], [x, y]) # obtain the Jacobian matrix
```

(remember that the first (x,y) output line comes from the command of var('x,y'))

**Note:**

While it makes sense to write:

```python
var('x,y') # define x,y as the CoCalc symbolic variables
f1(x,y) = 2*x*y + 2*y^2 # assign the first expression to the variable f1
f2(x,y) = 3*x^2 - y*x # assign the second expression to the variable f2
jacobian([f1, f2], [x, y]) # obtain the Jacobian matrix
```

(i.e. express the equations as functions \( f_1 \) and \( f_2 \))

We strongly suggest that you should just assign the expressions to the variables \( f_1 \) and \( f_2 \) like in the example above, so that the output is a little neater and easier to see. If you make the expressions into functions, the answer is still correct, but a little harder to see and interpret.

If you are curious, this is the output if you define \( f_1 \) and \( f_2 \) as functions of \( x \) and \( y \):
(x,y)

\[
\begin{bmatrix}
(x,y) & 2y \\
(x,y) & 2x + 4y
\end{bmatrix}
\]

\[
\begin{bmatrix}
(x,y) & 6x - y \\
(x,y) & -x
\end{bmatrix}
\]

11.3 The sub Command - Derivative/Jacobian Matrix at a Point

This tool is useful when we try to compute the derivative or Jacobian matrix at a specific point.

There are two ways we can use the sub command - with or without using a dictionary.

For each method, we will show an example for calculating derivative at a point and an example for calculating the Jacobian matrix at a point.

**Note:** Before using the `sub` command, it is **highly recommended** that you:

1. find the general derivative/Jacobian matrix, then
2. assign the general derivative/Jacobian matrix to a variable of your choice

11.3.1 Using sub Without Using a Dictionary

**Command:**

\[
\text{var\_name}.\text{subs}\left(\text{var1} = \text{value1}, \text{var2} = \text{value2}, \text{etc.}\right)
\]

where

- `var\_name` is the variable to which an expression (e.g. general derivative or general Jacobian matrix) is assigned
- `var1`, `var2`, etc. are the variables present in the expression that we are looking to be substituted for
- `value1`, `value2`, etc. are the associated values that we are looking to substitute in for `var1`, `var2`, etc.
Example # 1: Example for Derivative

Let’s say we want to find the derivative of the function \( f(x) = \sin(x^2) \) at \( x = 1 \). Our commands are:

```python
1 f(x) = sin(x^2)  # define the function
2 df = diff ( f(x), x)  # take the derivative of f(x) where f(x) = sin(x^2) and assign
3  the result to the variable “df”
4 df.subs(x = 1)  # substitute the value 1 to the expression of the derivative of f(x)

2*cos(1)
```

Example # 2: Example for Jacobian Matrix

Let’s say we want to obtain the Jacobian matrix for the following system at point (1, 2):

\[
\begin{align*}
x' &= 2 \cdot x \cdot y + 2 \cdot y^2 \\
y' &= 3 \cdot x^2 - y \cdot x
\end{align*}
\]

```python
1 var('x,y')  # define x,y as the CoCalc symbolic variables
2 f1 = 2*x*y + 2*y^2  # assign the first expression to the variable f1
3 f2 = 3*x^2 - y*x  # assign the second expression to the variable f2
4
5 J = jacobian ( [ f1, f2 ], [x, y] )  # obtain the Jacobian matrix and assign it to the
6  variable “J”
7 J.subs(x = 1, y = 2)  # find the Jacobian matrix at x = 1 and y = 2

(x,y)

[[ 4  10 ]
 [ 4  -1 ]]

(remember that the first (x,y) output line comes from the command of `var(‘x,y’)`)
```
11.3.2 Using sub Along with a Dictionary

**Command:**

```
var_name.subs ( { var1:value1, var2:value2, etc. } )
```

where

- `var_name` is the variable to which an expression (e.g. general derivative or general Jacobian matrix) is assigned
- `var1, var2, etc.` are the variables present in the expression that we are looking to be substituted for
- `value1, value2, etc.` are the associated values that we are looking to substitute in for `var1, var2, etc.`

**Example # 1: Example for Derivative**

Let’s say we want to find the derivative of the function \( f(x) = \sin(x^2) \) at \( x = 1 \). Our commands are:

```python
1 f(x) = sin(x^2)  # define the function
2 df = diff ( f(x), x)  # take the derivative of f(x) where f(x) = sin(x^2) and assign the result to the variable “df”
3 df.subs({x:1})  # substitute the value 1 to the expression of the derivative of f(x)
```

```
2*cos(1)
```

**Example # 2: Example for Jacobian Matrix**

Let’s say we want to obtain the Jacobian matrix for the following system at point (1, 2):

\[
\begin{align*}
x' &= 2 \cdot x \cdot y + 2 \cdot y^2 \\
y' &= 3 \cdot x^2 - y \cdot x
\end{align*}
\]
11.3. The sub Command - Derivative/Jacobian Matrix at a Point

1 \texttt{var('x,y')}  \quad \# \text{define } x,y \text{ as the CoCalc symbolic variables}

2

3 \texttt{f1 = 2*x*y + 2*y^2}  \quad \# \text{assign the first expression to the variable f1}

4 \texttt{f2 = 3*x^2 - y*x}  \quad \# \text{assign the second expression to the variable f2}

5

6 \texttt{J = jacobian([f1, f2], [x, y])}  \quad \# \text{obtain the Jacobian matrix and assign it to the variable “J”}

7

8 \texttt{J.subs({x:1, y:2})}  \quad \# \text{find the Jacobian matrix at } x = 1 \text{ and } y = 2

\[
\begin{bmatrix}
4 & 10 \\
4 & -1 
\end{bmatrix}
\]

(remember that the first \((x,y)\) output line comes from the command of \texttt{var('x,y')})
Chapter 12

Showing Command

**Command:** show(\textit{variable})

Most often, we use the \texttt{show} command to display a graph, in the form of \texttt{show(p)} where \texttt{p} is the variable to which all the plots commands are assigned.

For example, we want to plot the graph of \( f(x) = x^2 \) for: \( 0 \leq x \leq 5 \) and \( 0 \leq y \leq 30 \). Our commands are:

```
1   f(x) = x^2  # define the function
2   p = plot(f(x), (x, 0, 5), ymin = 0, ymax = 30)  # plot the graph and assign the plot to the variable p
3   show(p)   # show the plot
```

However, there are two things to note here.

**Note # 1:** This command is \textbf{NOT} restricted to the type \texttt{graphics} (e.g. \texttt{plot}). It depends on the \texttt{type} of the objects assigned to the variable that takes the position of \texttt{variable}. 

123
For example, we want to **show** the third element of the list \([1, 3, 5, 7, 9]\) using the command `show()`. Our commands will be:

```python
1. list1 = [1, 3, 5, 7, 9]  # define the list and assign it to the variable list1
2. b = list1[2]  # get the third element and assign it to the variable “b”
3. show(b)  # show the value assigned to the variable “b”, which has type of “integer”
```

**Note # 2:** In the position *variable*, it does **NOT** have to be solely a variable. It can be another command or mathematical expression.

For example, we have the lists of the number of puppies and kittens in the neighborhood over a time period of five years.

```plaintext
times = [1, 2, 3, 4, 5]
puppies = [5, 10, 20, 25, 50]
kittens = [2, 6, 14, 17, 27]
```

Now, we want to plot the time series for both populations on the same graph.

```python
1. time_list = [1, 2, 3, 4, 5]  # define the list of time stamps and assign it to the variable “time_list”
2. puppies = [5, 10, 20, 25, 50]  # define the list of puppies population and assign it to the variable “puppies”
3. kittens = [2, 6, 14, 17, 27]  # define the list of kittens population and assign it to the variable “kittens”
4. p1 = list_plot( zip(time_list, puppies), axes_labels = [“time”, “puppies/kittens”] )  # graph the time series for the puppies population in blue and assign the plot to the variable p1
5. p2 = list_plot( zip(time_list, kittens), color = “red”, axes_labels = [“time”, “puppies/kittens”] )  # graph the time series for the kittens population in red and assign the plot to the variable p2
6. show(p1 + p2)  # show the overlay of the two plots
```