A. Python

> *Simple is better than complex.*
> –Guido van Rossum (creator of Python)

In this appendix we will walk through some of the basics of using Python — the popular general-purpose programming language that we will use throughout this module.

For most of you this material will not be new. For example, you have probably seen it in your first year "Mathematical Programming & Skills" module. But for some of you this may be entirely new. You will have some notion of what a programming language "is" and "does", but you may never have written any code. That is all right.

If you are new to Python, don't feel that you need to work through this appendix in one go. Instead, spread the work over the first two weeks of the course and interlace it with your work on the first two chapters.

There is a lot of material in this appendix. Do not feel that you need to learn it all by hard. The idea is just that you should have seen the various language constructs once. Your familiarity with them will come automatically later when you use them throughout the course.

A.1. Why Python?

We are going to be using Python since

- Python is free,
- Python is very widely used,
- Python is flexible,
- Python is relatively easy to learn,
- and Python is quite powerful.

It is important to keep in mind that Python is a general purpose language that we will be using for Scientific Computing. The purpose of Scientific Computing is *not* to build apps, build software, manage databases, or develop user interfaces. Instead, Scientific Computing is the use of a computer programming language (like Python) along with mathematics to solve scientific and mathematical problems. For this reason it is definitely

not our purpose to write an all-encompassing guide for how to use Python. We will only cover what is necessary for our computing needs. You will learn more as the course progresses, so use this appendix just to get going with the language. To keep things as simple as possible, we will for example not use object oriented programming, so will not introduce classes and methods.

We are also definitely not saying that Python is the best language for scientific computing under all circumstances. The reason there are so many scientific programming languages coexisting is that each has particular strengths that make it the best option for particular applications. But we are saying that Python is so widely used that everyone should know Python.

There is an overwhelming abundance of information available about Python and the suite of tools that we will frequently use.

- Python https://www.python.org/,
- `numpy` (numerical Python) https://www.numpy.org/,
- `matplotlib` (a suite of plotting tools) https://matplotlib.org/,
- `scipy` (scientific Python) https://www.scipy.org/.

These tools together provide all of the computational power that we will need. And they are free!

A.2. Python Programming Basics

If you are already very practised in using Python then you can jump straight to **?@sec-python_exercises** with the coding exercises. But if you are new to Python or your Python skills are a bit rusty, then you will benefit from working through all the examples and exercises below, making sure you copy and paste all the code into your Colab notebook and run it there, and then critically evaluate and understand the output.

A.2.1. Variables

Variable names in Python can contain letters (lower case or capital), numbers 0-9, and some special characters such as the underscore. Variable names must start with a letter. There are a bunch of reserved words that you can not use for your variable names because they have a special meaning in the Python syntax. Python will let you know with a syntax error if you try to use a reserved word for a variable name.

You can do the typical things with variables. Assignment is with an equal sign (be careful R users, we will not be using the left-pointing arrow here!).

Warning: When defining numerical variables you do not always get floating point numbers. In some programming languages, if you write `x=1` then automatically `x` is saved as 1.0; a floating point number, not an integer. In Python however, if you assign `x=1` it is defined as an integer (with no decimal digits) but if you assign `x=1.0` it is assigned as a floating point number.

```
# assign some variables
x = 7 # integer assignment of the integer 7
y = 7.0 # floating point assignment of the decimal number 7.0
print("The variable x has the value", x, " and has type", type(x), ". \n")
print("The variable y has the value", y, " and has type", type(y), ". \n")
```

Remember to copy each code block to your own notebook, execute it and look at the output. To copy the code from this guide to your notebook you can use the "Copy to Clipboard" icon that pops up in the top right corner of a code block when you hover over that code block.

```
# multiplying by a float will convert an integer to a float
x = 7 # integer assignment of the integer 7
print("Multiplying x by 1.0 gives", 1.0*x)
print("The type of this value is", type(1.0*x), ". \n")
```

The allowed mathematical operations are:

- Addition: `+`
- Subtraction: `-`
- Multiplication: `*`
- Division: `/`
- Integer Division (modular division): `//` and `%`
- Exponents: `**`

That's right, the caret key, `^`, is NOT an exponent in Python (sigh). Instead we have to get used to `**` for exponents.

```
x = 7.0
y = x**2 # square the value in x
y
```

Exercise A.1. Write code to define positive integers a, b and c of your own choosing. Then calculate a^2, b^2 and c^2. When you have all three values computed, check to see if your three values form a Pythagorean Triple so that $a^2 + b^2 = c^2$. Have Python simply say True or False to verify that you do, or do not, have a Pythagorean Triple defined. **Hint:** You will need to use the `==` Boolean check just like in other programming languages.

A.2.2. Indexing and Lists

Lists are a key component to storing data in Python. Lists are exactly what the name says: lists of things (in our case, usually the entries are floating point numbers).

Warning: Python indexing starts at 0 whereas some other programming languages have indexing starting at 1. In other words, the first entry of a list has index 0, the second entry as index 1, and so on. We just have to keep this in mind.

We can extract a part of a list using the syntax `name[start:stop]` which extracts elements between index `start` and `stop-1`. Take note that Python stops reading at the second to last index. This often catches people off guard when they first start with Python.

Example A.1 (Lists and Indexing)**.** Let us look at a few examples of indexing from lists. In this example we will use the list of numbers 0 through 8. This list contains 9 numbers indexed from 0 to 8.

- Create the list of numbers 0 through 8

```
my_list = [0,1,2,3,4,5,6,7,8]
```

- Output the list

```
my_list
```

- Select only the element with index 0.

```
my_list[0]
```

- Select all elements up to, but not including, the third element of `my_list`.

```
my_list[:2]
```

- Select the last element of `my_list` (this is a handy trick!).

```
my_list[-1]
```

- Select the elements indexed 1 through 4. Beware! This is not the first through fifth element.

```
my_list[1:5]
```

- Select every other element in the list starting with the first.

```
my_list[0::2]
```

- Select the last three elements of `my_list`

```
my_list[-3:]
```

In Python, elements in a list do not need to be the same type. You can mix integers, floats, strings, lists, etc.

Example A.2. In this example we see a list of several items that have different data types: float, integer, string, and complex. Note that the imaginary number i is represented by $1j$ in Python. This use of j instead of i is common in some scientific disciplines and is just another thing that we Mathematicians will need to get used to in Python.

```
MixedList = [1.0, 7, 'Bob', 1-1j]
print(MixedList)
print(type(MixedList[0]))
print(type(MixedList[1]))
print(type(MixedList[2]))
print(type(MixedList[3]))
# Notice that we use 1j for the imaginary number "i".
```

Exercise A.2. In this exercise you will put your new list skills into practice.

1. Create the list of the first several Fibonacci numbers:

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89. \tag{A.1}$$

2. Print the first four elements of the list.
3. Print every third element of the list starting from the first.
4. Print the last element of the list.
5. Print the list in reverse order.
6. Print the list starting at the last element and counting backward by every other element.

A.2.3. List Operations

Python is awesome about allowing you to do things like appending items to lists, removing items from lists, and inserting items into lists. Note in all of the examples below that we are using the code
`variable.method`
where you put the variable name, a dot, and the thing that you would like to do to that variable. For example, `my_list.append(7)` will append the number 7 to the list `my_list`. We say that `append` is a "method" of the list `my_list`. This is a common programming feature in Python and we will use it often.

Example A.3. The `.append` method can be used to append an element to the end of a list.

```
my_list = [0,1,2,3]
print(my_list)
# Append the string 'a' to the end of the list
my_list.append('a')
print(my_list)
# Do it again ... just for fun
my_list.append('a')
print(my_list)
# Append the number 15 to the end of the list
my_list.append(15)
print(my_list)
```

Example A.4. The `.remove` method can be used to remove an element from a list.

```
# Let us remove the 3
my_list.remove(3)
my_list
```

Example A.5. The `.insert` method can be used to insert an element at a location in a list.

```
# insert the letter `A` at the 0-indexed spot
my_list.insert(0,'A')
# insert the letter `B` at the spot with index 3
my_list.insert(3,'B')
# remember that index 3 means the fourth spot in the list
my_list
```

Exercise A.3. In this exercise you will go a bit further with your list operation skills.

1. Create the list of the first several Lucas Numbers: $1, 3, 4, 7, 11, 18, 29, 47$.
2. Add the next three Lucas Numbers to the end of the list.
3. Remove the number 3 from the list.
4. Insert the 3 back into the list in the correct spot.
5. Print the list in reverse order.
6. Do a few other list operations to this list and report your findings.

A.2.4. Tuples

In Python, a "tuple" is like an ordered pair (or ordered triple, or ordered quadruple, ...) in mathematics. We will occasionally see tuples in our work in numerical analysis so for now let us just give a couple of code snippets showing how to store and read them.

We can define the tuple of numbers $(10, 20)$ in Python as follows:

Example A.6.

```
point = 10, 20
print(point, type(point))
```

We can also define a tuple with parenthesis if we like. Python does not care.

```
point = (10, 20) # now we define the tuple with parenthesis
print(point, type(point))
```

We can then unpack the tuple into components if we wish:

```
x, y = point
print("x = ", x)
print("y = ", y)
```

There are other important data structures in Python that we will not use in this module. These include dictionaries and sets. We will not cover these here because we are trying to keep things simple so that we can concentrate on Numerical Analysis instead. If you are interested in learning more about these data structures, you can find a lot of information about them in the Python documentation.

A.2.5. Control Flow: Loops and If Statements

Any time you need to do some repetitive task with a programming language you can use a loop. Just like in other programming languages, we can do loops and conditional statements in very easy ways in Python. The thing to keep in mind is that the Python language is very white-space-dependent. This means that your indentations need to be correct in order for a loop to work. You could get away with sloppy indention in other languages but not so in Python. Also, in some languages (like R and Java) you need to wrap your loops in curly braces. Again, not so in Python.

Caution: Be really careful of the white space in your code when you write loops.

A.2.5.1. `for` Loops

A `for` loop is designed to do a task a certain number of times and then stop. This is a great tool for automating repetitive tasks, but it also nice numerically for building sequences, summing series, or just checking lots of examples. The following are some examples of Python for loops.

Example A.7. Print the first 6 perfect squares.

```
for x in [1,2,3,4,5,6]:
    print(x**2)
```

Often instead of writing the list of integers explicitly one uses the `range()` function, so that this example would be written as

```
for x in range(1,7):
    print(x**2)
```

Note that `range(1,7)` produces the integers from 1 to 6, not from 1 to 7. This is another manifestation of Python's weird 0-based indexing. Of course it is only weird to people who are new to Python. For Pythonists it is perfectly natural.

You can also use `range()` to generate a sequence of numbers with a specific step size. For example, `range(1, 10, 2)` will generate the odd integers from 1 to 9.

```
for x in range(1, 10, 2):
    print(x)
```

Take careful note of the syntax for a for loop as it is the same as for other loops and conditional statements:

- a control statement,
- a colon, a new line,
- indent four spaces,
- some programming statements

When you are done with the loop, just back out of the indention. There is no need for an `end` command or a curly brace. All of the control statements in Python are white-space-dependent.

Example A.8. Print the names in a list.

```
NamesList = ['Alice','Billy','Charlie','Dom','Enrique','Francisco']
for name in NamesList:
    print(name)
```

In Python you can use a more compact notation for `for` loops sometimes. This takes a bit of getting used to, but is super slick!

Example A.9. Create a list of the perfect squares from 1 to 9.

```
# create a list of the perfect squares from 1 to 9
CoolList = [x**2 for x in range(1,10)]
print(CoolList)
# Then print the sum of this list
print("The sum of the first 9 perfect squares is", sum(CoolList))
```

`for` loops can also be used to build sequences, as can be seen in the next couple of examples.

Example A.10. In the following code we write a for loop that outputs a list of the first 7 iterations of the sequence $x_{n+1} = -0.5x_n + 1$ starting with $x_0 = 3$. Notice that we are using the command `x.append` instead of $x[n+1]$ to append the new term to the list. This allows us to grow the length of the list dynamically as the loop progresses.

```
x=[3.0]
for n in range(0,7):
    x.append(-0.5*x[n] + 1)
    print(x) # print the whole list x at each step of the loop
```

Example A.11. As an alternative to the code from the previous example we can pre-allocate the memory in a list of zeros. This is done with the clever code `x = [0] * 10`. Literally multiplying a list by some number, like 10, says to repeat that list 10 times.

Now we will build the sequence with pre-allocated memory.

```
x = [0] * 7
x[0] = 3.0
for n in range(0,6):
    x[n+1] = -0.5*x[n]+1
    print(x) # This print statement shows x at each iteration
```

Exercise A.4. We want to sum the first 100 perfect cubes. Let us do this in two ways.

1. Start off a variable called `total` at 0 and write a `for` loop that adds the next perfect cube to the running total.

2. Write a `for` loop that builds the sequence of the first 100 perfect cubes. After the list has been built find the sum with the `sum()` function.

The answer is: 25,502,500 so check your work.

Exercise A.5. Write a `for` loop that builds the first 20 terms of the sequence $x_{n+1} = 1 - x_n^2$ with $x_0 = 0.1$. Pre-allocate enough memory in your list and then fill it with the terms of the sequence. Only print the list after all of the computations have been completed.

A.2.5.2. `while` Loops

A `while` loop repeats some task (or sequence of tasks) while a logical condition is true. It stops when the logical condition turns from true to false. The structure in Python is the same as with `for` loops.

Example A.12. Print the numbers 0 through 4 and then the word "done." we will do this by starting a counter variable, `i`, at 0 and increment it every time we pass through the loop.

```
i = 0
while i < 5:
    print(i)
    i += 1 # increment the counter
print("done")
```

Example A.13. Now let us use a while loop to build the sequence of Fibonacci numbers and stop when the newest number in the sequence is greater than 1000. Notice that we want to keep looping until the condition that the last term is greater than 1000 – this is the perfect task for a `while` loop, instead of a `for` loop, since we do not know how many steps it will take before we start the task

```
Fib = [1,1]
while Fib[-1] <= 1000:
    Fib.append(Fib[-1] + Fib[-2])
print("The last few terms in the list are:\n",Fib[-3:])
```

Exercise A.6. Write a `while` loop that sums the terms in the Fibonacci sequence until the sum is larger than 1000

A.2.5.3. `if` Statements

Conditional (`if`) statements allow you to run a piece of code only under certain conditions. This is handy when you have different tasks to perform under different conditions.

Example A.14. Let us look at a simple example of an `if` statement in Python.

```
Name = "Alice"
if Name == "Alice":
    print("Hello, Alice.  Isn't it a lovely day to learn Python?")
else:
    print("You're not Alice.  Where is Alice?")
```

```
Name = "Billy"
if Name == "Alice":
    print("Hello, Alice.  Isn't it a lovely day to learn Python?")
else:
    print("You're not Alice.  Where is Alice?")
```

Example A.15. For another example, if we get a random number between 0 and 1 we could have Python print a different message depending on whether it was above or below 0.5. Run the code below several times and you will see different results each time.

Note: We have to import the `numpy` package to get the random number generator in Python. Do not worry about that for now. We will talk about packages in a moment.

```
import numpy as np
x = np.random.uniform() # get a random number between 0 and 1
if x < 0.5:
    print(x," is less than a half")
else:
    print(x, "is NOT less than a half")
```

(Take note that the output will change every time you run it.)

Example A.16. In many programming tasks it is handy to have several different choices between tasks instead of just two choices as in the previous examples. This is a job for the `elif` command.

This is the same code as last time except we will make the decision at 0.33 and 0.67.

```
import numpy as np
x = np.random.rand(1,1) # get a random 1x1 matrix using numpy
x = x[0,0] # pull the entry from the first row and first column
if x < 0.33:
    print(x," < 1/3")
elif x < 0.67:
    print("1/3 <= ",x,"< 2/3")
else:
    print(x, ">= 2/3")
```

(Take note that the output will change every time you run it.)

Exercise A.7. Write code to give the Collatz Sequence

$$x_{n+1} = \begin{cases} x_n/2, & x_n \text{ is even} \\ 3x_n + 1, & \text{otherwise} \end{cases} \tag{A.2}$$

starting with a positive integer of your choosing. The sequence will converge[1] to 1 so your code should stop when the sequence reaches 1.

Hints: To test whether a number `x` is even you can test whether the remainder after dividing by 2 is zero with `(x % 2) == 0`. Also you will want to use the integer division `//` when calculating $x_n/2$.

---

[1]Actually, it is still an open mathematical question whether every integer seed will converge to 1. The Collatz sequence has been checked for many millions of initial seeds and they all converge to 1, but there is no mathematical proof that it will always happen. You will check the conjecture numerically in Exercise 2.27

A.2.6. Functions

Mathematicians and programmers talk about functions in very similar ways, but they are not exactly the same. When we say "function" in a programming sense we are talking about a chunk of code that you can pass parameters and expect an output of some sort. This is not unlike the mathematician's version. But unlike a mathematical function, a Python function can also have side effects, like plotting a graph for example. So Python's definition of a function is a bit more flexible than that of a mathematician.

In Python, to define a function we start with `def`, followed by the function's name, any input variables in parenthesis, and a colon. The indented code after the colon is what defines the actions of the function.

Example A.17. The following code defines the polynomial $f(x) = x^3 + 3x^2 + 3x + 1$ and then evaluates the function at a point $x = 2.3$.

```
def f(x):
    return(x**3 + 3*x**2 + 3*x + 1)
f(2.3)
```

Take careful note of several things in the previous example:

- To define the function we cannot just type it like we would see it one paper. This is not how Python recognizes functions.
- Once we have the function defined we can call upon it just like we would on paper.
- We cannot pass symbols into this type of function.[2]

Exercise A.8. Define the function $g(n) = n^2 + n + 41$ as a Python function. Write a loop that gives the output for this function for integers from $n = 0$ to $n = 39$. Euler noticed that each of these outputs is a prime number (check this on your own). Will the function produce a prime for $n = 40$? For $n = 41$?

---

[2] There is the `sympy` package if you want to do symbolic computations, but we will not use that in this module.

Example A.18. One cool thing that you can do with functions is call them recursively. That is, you can call the same function from within the function itself. This turns out to be really handy in several mathematical situations.

Let us define a function for the factorial. This function is naturally going to be recursive in the sense that it calls on itself!

```
def factorial(n):
    if n==0:
        return(1)
    else:
        return(n*factorial(n-1))
        # Note: we are calling the function recursively.
```

When you run this code there will be no output. You have just defined the function so you can use it later, as follows:

```
factorial(12)
```

Example A.19. For this next example let us define a function to calculate the next element in the sequence

$$x_{n+1} = \begin{cases} 2x_n, & x_n \in [0, 0.5] \\ 2x_n - 1, & x_n \in (0.5, 1] \end{cases} \tag{A.3}$$

and then build a loop to find the first several elements of the sequence starting at any real number between 0 and 1.

```
# Define the function
def my_seq(xn):
    if xn <= 0.5:
        return(2*xn)
    else:
        return(2*xn-1)
# Now build a sequence with this function
x = [0.125] # arbitrary starting point
for n in range(0,5): # Let us only build the first 5 terms
    x.append(my_seq(x[n]))
print(x)
```

Example A.20. A fun way to approximate the square root of two is to start with any positive real number and iterate over the sequence

$$x_{n+1} = \frac{1}{2}x_n + \frac{1}{x_n} \tag{A.4}$$

until we are within any tolerance we like of the square root of 2. Write code that defines the sequence as a function and then iterates in a while loop until we are within 10^{-8} of the square root of 2.

We import the `math` package so that we get the square root function. More about packages in the next section.

```
from math import sqrt
def f(x):
    return(0.5*x + 1/x)
x = 1.1 # arbitrary starting point
print(f"{'Approximation':<20} | {'Exact':<20} | {'Absolute error':<20}")
print("-" * 68)
while abs(x-sqrt(2)) > 10**(-8):
    x = f(x)
    print(f"{x:<20} | {sqrt(2):<20} | {abs(x - sqrt(2)):<20}")
```

This example also shows how to format the output of a print statement so that it is nicely aligned in columns. The `:<20` means that the string will be left-aligned and padded with spaces to a total width of 20 characters. The `f` before the string allows us to use curly braces `{}` to insert variables into the string. You will see an alternative way for displaying tabular data in Example A.48.

Exercise A.9. The previous example is a special case of the Babylonian Algorithm for calculating square roots. If you want the square root of S then iterate the sequence

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{S}{x_n}\right) \tag{A.5}$$

until you are within an appropriate tolerance.

Modify the code given in the previous example to give a list of approximations of the square roots of the natural numbers 2 through 20, each to within 10^{-8}. This problem will require that you build a function, write a 'for' loop (for the integers 2 through 20), and write a 'while' loop inside your 'for' loop to do the iterations.

A.2.7. Lambda Functions

Using `def` to define a function as in the previous subsection is really nice when you have a function that is complicated or requires some bit of code to evaluate. However, in the case of mathematical functions we have a convenient alternative: `lambda` Functions.

The basic idea of a `lambda` Function is that we just want to state what the variable is and what the rule is for evaluating the function. This is closest to the way that we write mathematical functions. For example, we can define the mathematical function $f(x) = x^2 + 3$ in two different ways.

- Using `def`:

```
def f(x):
    return(x**2+3)
```

- Using `lambda`:

```
f = lambda x: x**2+3
```

You can see that in the Lambda Function we are explicitly stating the name of the variable immediately after the word `lambda`, then we put a colon, and then the function definition. This is somewhat similar to but also annoyingly different from the mathematicians notation $f : x \mapsto x^2 + 3$.

No matter whether we use `def` or `lambda` to define the function `f`, if we want to evaluate the function at a point, say $x = 1.5$, then we can write code just like we would mathematically: $f(1.5)$

```
f(1.5) # evaluate the function at x=1.5
```

We can also define Lambda Functions of several variables. For example, if we want to define the mathematical function $f(x, y) = x^2 + xy + y^3$ we could write the code

```
f = lambda x, y: x**2 + x*y + y**3
```

If we wanted the value $f(2, 4)$ we would now write the code `f(2,4)`.

Exercise A.10. Go back to Exercise A.8 and repeat this exercise using a `lambda` function.

Exercise A.11. Go back to Exercise A.9 and repeat this exercise using a `lambda` function.

A.2.8. Packages

Python was not created as a scientific programming language. The reason Python can be used for scientific computing is that there are powerful extension packages that define additional functions that are needed for scientific calculations.

Let us start with the `math` package.

Example A.21. The code below imports the `math` package into your instance of Python and calculates the cosine of $\pi/4$.

```
import math
x = math.cos(math.pi / 4)
print(x)
```

The answer, unsurprisingly, is the decimal form of $\sqrt{2}/2$.

You might already see a potential disadvantage to Python's packages: there is now more typing involved! Let us fix this. When you import a package you could just import all of the functions so they can be used by their proper names.

Example A.22. Here we import the entire math package so we can use every one of the functions therein without having to use the `math` prefix.

```
from math import * # read this as: from math import everything
x = cos(pi / 4)
print(x)
```

The end result is exactly the same: the decimal form of $\sqrt{2}/2$, but now we had less typing to do.

Now you can freely use the functions that were imported from the math package. There is a disadvantage to this, however. What if we have two packages that import functions with the same name. For example, in the `math` package and in the `numpy` package there is a `cos()` function. In the next block of code we will import both `math` and `numpy`, but instead we will import them with shortened names so we can type things a bit faster.

Example A.23. Here we import `math` and `numpy` under aliases so we can use the shortened aliases and not mix up which functions belong to which packages.

```
import math as ma
import numpy as np
# use the math version of the cosine function
x = ma.cos(ma.pi / 4)
# use the numpy version of the cosine function
y = np.cos(np.pi / 4)
print(x, y)
```

Both `x` and `y` in the code give the decimal approximation of $\sqrt{2}/2$. This is clearly pretty redundant in this really simple case, but you should be able to see where you might want to use this and where you might run into troubles.

Example A.24 (Contents of a package)**.** Once you have a package imported you can see what is inside of it using the `dir` command. The following block of code prints a list of all of the functions inside the `math` package.

```
import math
print(dir(math))
```

By the way: you only need to import a package once in a session. The only reason we are repeating the `import` statement in each code block is to make it easier to come back to this material later in a new session, where you will need to import the packages again.

Of course, there will be times when you need help with a function. You can use the `help` function to view the help documentation for any function. For example, you can run the code `help(math.acos)` to get help on the arc cosine function from the `math` package.

Exercise A.12. Import the `math` package, figure out how the `log` function works, and write code to calculate the logarithm of the number 8.3 in base 10, base 2, base 16, and base e (the natural logarithm).

A.3. Numerical Python with NumPy

The base implementation of Python includes the basic programming language, the tools to write loops, check conditions, build and manipulate lists, and all of the other things that we saw in the previous section. In this section we will explore the package `numpy` that contains optimized numerical routines for doing numerical computations in scientific computing.

Example A.25. To start with, let us look at a really simple example. Say you have a list of real numbers and you want to take the sine of every element in the list. If you just try to take the sine of the list you will get an error. Try it yourself.

```
from math import pi, sin
my_list = [0,pi/6, pi/4, pi/3, pi/2, 2*pi/3, 3*pi/4, 5*pi/6, pi]
sin(my_list)
```

You could get around this error using some of the tools from base Python, but none of them are very elegant from a programming perspective.

```
from math import pi, sin
my_list = [0,pi/6, pi/4, pi/3, pi/2, 2*pi/3, 3*pi/4, 5*pi/6, pi]
sine_list = [sin(n) for n in my_list]
sine_list
```

```
from math import pi, sin
my_list = [0,pi/6, pi/4, pi/3, pi/2, 2*pi/3, 3*pi/4, 5*pi/6, pi]
sine_list = [ ]
for n in range(0,len(my_list)):
    sine_list.append(sin(my_list[n]))
sine_list
```

Perhaps more simply, say we wanted to square every number in a list. Just appending the code **2 to the end of the list will fail!

```
my_list = [1,2,3,4]
my_list**2 # This will produce an error
```

If, instead, we define the list as a `numpy` array instead of a Python list then everything will work mathematically exactly the way that we intend.

```
import numpy as np
my_list = np.array([1,2,3,4])
my_list**2 # This will work as expected!
```

Exercise A.13. See if you can take the sine of a full list of numbers that are stored in a `numpy` array.

Hint: you will now see why the numpy package provides its own version of the sine function.

The package `numpy` is used in many (most) mathematical computations in numerical analysis using Python. It provides algorithms for matrix and vector arithmetic. Furthermore, it is optimized to be able to do these computations in the most efficient possible way (both in terms of memory and in terms of speed).

Typically when we import `numpy` we use `import numpy as np`. This is the standard way to name the `numpy` package. This means that we will have lots of function with the prefix "np" in order to call on the `numpy` functions. Let us first see what is inside the package

```
import numpy as np
dir(np)
```

A brief glimpse through the list reveals a huge wealth of mathematical functions that are optimized to work in the best possible way with the Python language. (We are intentionally not showing the output here since it is quite extensive, run it so you can see.)

A.3.1. Numpy Arrays, Array Operations, and Matrix Operations

In the previous section you worked with Python lists. As we pointed out, the shortcoming of Python lists is that they do not behave well when we want to apply mathematical functions to the vector as a whole. The "numpy array", `np.array`, is essentially the same as a Python list with the notable exceptions that

- In a `numpy` array every entry is a floating point number
- In a `numpy` array the memory usage is more efficient (mostly since Python is expecting data of all the same type)
- With a `numpy` array there are ready-made functions that can act directly on the array as a matrix or a vector

Let us just look at a few examples using `numpy`. What we are going to do is to define a matrix A and vectors v and w as

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}, \quad v = \begin{pmatrix} 5 \\ 6 \end{pmatrix} \quad \text{and} \quad w = v^T = \begin{pmatrix} 5 & 6 \end{pmatrix}. \tag{A.6}$$

Then we will do the following

- Get the size and shape of these arrays
- Get individual elements, rows, and columns from these arrays
- Treat these arrays as with linear algebra to
  - do element-wise multiplication
  - do matrix a vector products
  - do scalar multiplication
  - take the transpose of matrices
  - take the inverse of matrices

Example A.26 (numpy Matrices)**.** The first thing to note is that a matrix is a list of lists (each row is a list).

```
import numpy as np
A = np.array([[1,2],[3,4]])
print("The matrix A is:\n",A)
v = np.array([[5],[6]]) # this creates a column vector
print("The vector v is:\n",v)
w = np.array([[5,6]]) # this creates a row vector
print("The vector w is:\n",w)
```

Example A.27 (`.shape`)**.** The `.shape` attribute can be used to give the shape of a `numpy` array. Notice that the output is a tuple showing the size `(rows, columns)`.

```
print("The shape of the matrix A is ", A.shape)
print("The shape of the column vector v is ", v.shape)
print("The shape of the row vector w is ", w.shape)
```

Example A.28 (`.size`)**.** The `.size` attribute can be used to give the size of a `numpy` array. The size of a matrix or vector will be the total number of elements in the array. You can think of this as the product of the values in the tuple coming from the shape method.

```
print("The size of the matrix A is ", A.size)
print("The size of the column vector v is ", v.size)
print("The size of the row vector w is ", w.size)
```

Reading individual elements from a `numpy` array is the same, essentially, as reading elements from a Python list. We will use square brackets to get the row and column. Remember that the indexing all starts from 0, not 1!

Example A.29. Let us read the top left and bottom right entries of the matrix A.

```
import numpy as np
A = np.array([[1,2],[3,4]])
print(A[0,0]) # top left
print(A[1,1]) # bottom right
```

Example A.30. Let us read the first row from that matrix A.

```
import numpy as np
A = np.array([[1,2],[3,4]])
print(A[0,:])
```

Example A.31. Let us read the second column from the matrix A.

```
import numpy as np
A = np.array([[1,2],[3,4]])
print(A[:,1])
```

Notice when we read the column it was displayed as a row. Be careful. Reading a row or a column from a matrix will automatically flatten it into a 1-dimensional array.

If we try to multiply either A and v or A and A we will get some funky results. Unlike in some programming languages like MATLAB, the default notion of multiplication is NOT matrix multiplication. Instead, the default is element-wise multiplication. You may be familiar with this from R.

Example A.32. If we write the code `A*A` we do NOT do matrix multiplication. Instead we do element-by-element multiplication. This is a common source of issues when dealing with matrices and Linear Algebra in Python.

```
import numpy as np
A = np.array([[1,2],[3,4]])
print("Element-wise multiplication:\n", A * A)
print("Matrix multiplication:\n", A @ A)
```

Example A.33. If we write `A * v` Python will do element-wise multiplication across each column since v is a column vector. If we want the matrix `A` to act on `v` we write `A @ v`.

```
import numpy as np
A = np.array([[1,2],[3,4]])
v = np.array([[5],[6]])
print("Element-wise multiplication on each column:\n", A * v)
# A @ v will do proper matrix multiplication
print("Matrix A acting on vector v:\n", A @ v)
```

It is up to you to check that these products are indeed correct from the definitions of matrix multiplication from Linear Algebra.

It remains to show some of the other basic linear algebra operations: inverses, determinants, the trace, and the transpose.

Example A.34 (Transpose)**.** Taking the transpose of a matrix (swapping the rows and columns) is done with the `.T` attribute.

```
A.T # The transpose is relatively simple
```

Example A.35 (Trace)**.** The trace is done with `matrix.trace()`

```
A.trace() # The trace is pretty darn easy too
```

Oddly enough, the trace returns a matrix, not a scalar Therefore you will have to read the first entry (index `[0,0]`) from the answer to just get the trace.

Example A.36 (Determinant)**.** The determinant function is hiding under the `linalg` subpackage inside `numpy`. Therefore we need to call it as such.

```
np.linalg.det(A)
```

You notice an interesting numerical error here. You can do the determinant easily by hand and so know that it should be exactly -2. We'll discuss the source of these kinds of errors in Chapter 1.

Example A.37 (Inverse)**.** In the `linalg` subpackage there is also a function for taking the inverse of a matrix.

```
A_inv = np.linalg.inv(A)
A_inv
```

We can check that we get the identity matrix back:

```
A @ A_inv
```

Exercise A.14. Now that we can do some basic linear algebra with `numpy` it is your turn. Define the matrix B and the vector u as

$$B = \begin{pmatrix} 1 & 4 & 8 \\ 2 & 3 & -1 \\ 0 & 9 & -3 \end{pmatrix} \quad \text{and} \quad u = \begin{pmatrix} 6 \\ 3 \\ -7 \end{pmatrix}. \tag{A.7}$$

Then find

1. Bu
2. B^2 (in the traditional linear algebra sense)
3. The size and shape of B
4. $B^T u$
5. The element-by-element product of B with itself
6. The dot product of u with the first row of B

A.3.2. `arange`, `linspace`, `zeros`, `ones`, and `meshgrid`

There are a few built-in ways to build arrays in `numpy` that save a bit of time in many scientific computing settings.

Example A.38. The `np.arange` (array range) function is great for building sequences.

```
import numpy as np
x = np.arange(0,0.6,0.1)
x
```

`np.arange` builds an array of floating point numbers with the arguments `start, stop,` and `step`. Note that the `stop` value itself is not included in the result.

Example A.39. The `np.linspace` function builds an array of floating point numbers starting at one point, ending at the next point, and have exactly the number of points specified with equal spacing in between: `start, stop, number of points`.

```
import numpy as np
y = np.linspace(0,5,11)
y
```

In a linear space you are always guaranteed to hit the stop point exactly, but you do not have direct control over the step size.

Example A.40. The `np.zeros` function builds an array of zeros. This is handy for pre-allocating memory.

```
import numpy as np
z = np.zeros((3,5)) # create a 3x5 matrix of zeros
z
```

If you already have an array and want to create an array of zeros with the same shape, you can use `np.zeros_like(array)`.

```
import numpy as np
x = np.linspace(0,5,11)
z = np.zeros_like(x)
z
```

Similarly there are the functions `np.ones` and `np.ones_like` to build arrays of ones.

Example A.41. The `np.meshgrid` function builds two arrays that when paired make up the ordered pairs for a 2D (or higher D) mesh grid of points. This is handy for building 2D (or higher dimensional) arrays of data for multi-variable functions. Notice that the output is defined as a tuple.

```
import numpy as np
x, y = np.meshgrid(np.linspace(0,5,6), np.linspace(0,5,6))
print("x = ", x)
print("y = ", y)
```

The thing to notice with the `np.meshgrid()` function is that when you lay the two arrays on top of each other, the matching entries give every ordered pair in the domain.

If the purpose of this is not clear to you yet, don't worry. You will see it used a lot later in the module.

Exercise A.15. Now it is time to practice with some of these `numpy` functions.

a. Create a `numpy` array of the numbers 1 through 10 and square every entry in the list without using a loop.

b. Create a 10×10 identity matrix and change the top right corner to a 5. Hint: `np.identity()`

c. Find the matrix-vector product of the answer to part (b) and the answer to part (a).

d. Change the bottom row of your matrix from part (b) to all 3's, then change the third column to all 7's, and then find the 5^{th} power of this matrix.

A.4. Plotting with Matplotlib

A key part of scientific computing is plotting your results or your data. The tool in Python best-suited to this task is the package `matplotlib`. As with all of the other packages in Python, it is best to learn just the basics first and then to dig deeper later. One advantage to using `matplotlib` in Python is that it is modelled off of MATLAB's plotting tools. People coming from a MATLAB background should feel pretty comfortable here, but there are some differences to be aware of.

A.4.1. Basics with `plt.plot()`

We are going to start right away with an example. In this example, however, we will walk through each of the code chunks one-by-one so that we understand how to set up a proper plot.

Below we will mention some tricks for getting the plots to render that only apply to Jupyter Notebooks. If you are using Google Colab then you may not need some of these little tricks.

Example A.42 (Plotting with matplotlib)**.** In the first example we want to simply plot the sine function on the domain $x \in [0, 2\pi]$, colour it green, put a grid on it, and give a meaningful legend and axis labels. To do so we first need to take care of a couple of housekeeping items.

- Import `numpy` so we can take advantage of some good numerical routines.
- Import `matplotlib`'s `pyplot` module. The standard way to pull it in is with the nickname `plt` (just like with `numpy` when we import it as `np`).

In Jupyter Notebooks the plots will not show up unless you tell the notebook to put them "inline." Usually we will use the following command to get the plots to show up. You do not need to do this in Google Colab. The percent sign is called a *magic* command in Jupyter Notebooks. This is not a Python command, but it is a command for controlling the Jupyter IDE specifically.

```
%matplotlib inline
```

Now we will build a `numpy` array of x values (using the `np.linspace` function) and a `numpy` array of y values from the sine function.

- Next, build the plot with `plt.plot()`. The syntax is: `plt.plot(x, y, 'color', ...)` where you have several options that you can pass (more on that later).

- We send the plot label directly to the plot function. This is optional and we could set the legend up separately if we like.
- Then we will add the grid with `plt.grid()`
- Then we will add the legend to the plot
- Finally we will add the axis labels
- We end the plotting code with `plt.show()` to tell Python to finally show the plot. This line of code tells Python that you are done building that plot.

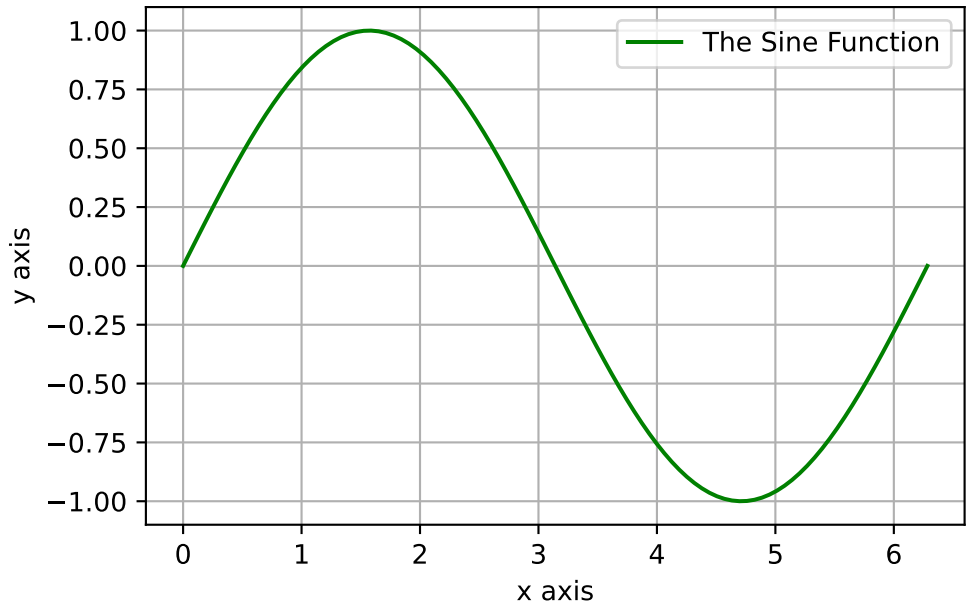

Figure A.1.: The sine function

Example A.43. Now let us do a second example, but this time we want to show four different plots on top of each other. When you start a figure, `matplotlib` is expecting all of those plots to be layered on top of each other. (Note:For MATLAB users, this means that you do not need the `hold on` command since it is automatically "on.")

In this example we will plot

$$y_0 = \sin(2\pi x) \quad y_1 = \cos(2\pi x) \quad y_2 = y_0 + y_1 \quad \text{and} \quad y_3 = y_0 - y_1 \tag{A.8}$$

on the domain $x \in [0, 1]$ with 100 equally spaced points. we will give each of the plots a different line style, built a legend, put a grid on the plot, and give axis labels.

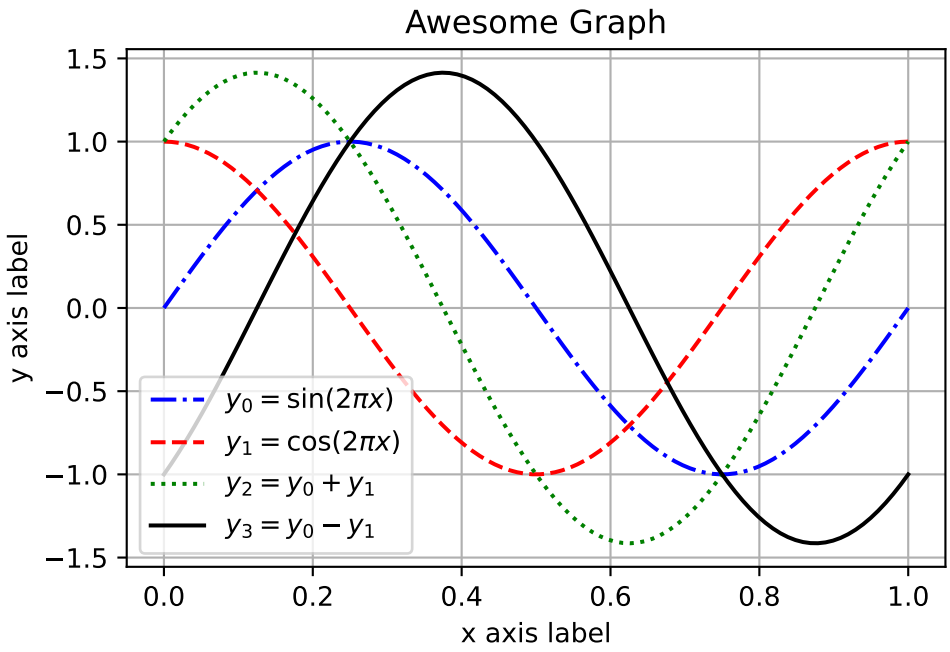

Figure A.2.: Plots of the sine, cosine, and sums and differences.

Notice the `r` in front of the strings defining the legend. This prevents the backslash that is used a lot in LaTeX to be interpreted as an escape character. These strings are referred to as raw strings.

The legend was placed automatically at the lower left of the plot. There are ways to control the placement of the legend if you wish, but for now just let Python and `matplotlib` have control over the placement.

Example A.44. Now let us create the same plot with slightly different code. The `plot` function can take several (x, y) pairs in the same line of code. This can really shrink the amount of coding that you have to do when plotting several functions on top of each other.

Exercise A.16. Plot the functions $f(x) = x^2$, $g(x) = x^3$, and $h(x) = x^4$ on the same axes. Use the domain $x \in [0, 1]$. Put a grid, a legend, a title, and appropriate labels on the axes.

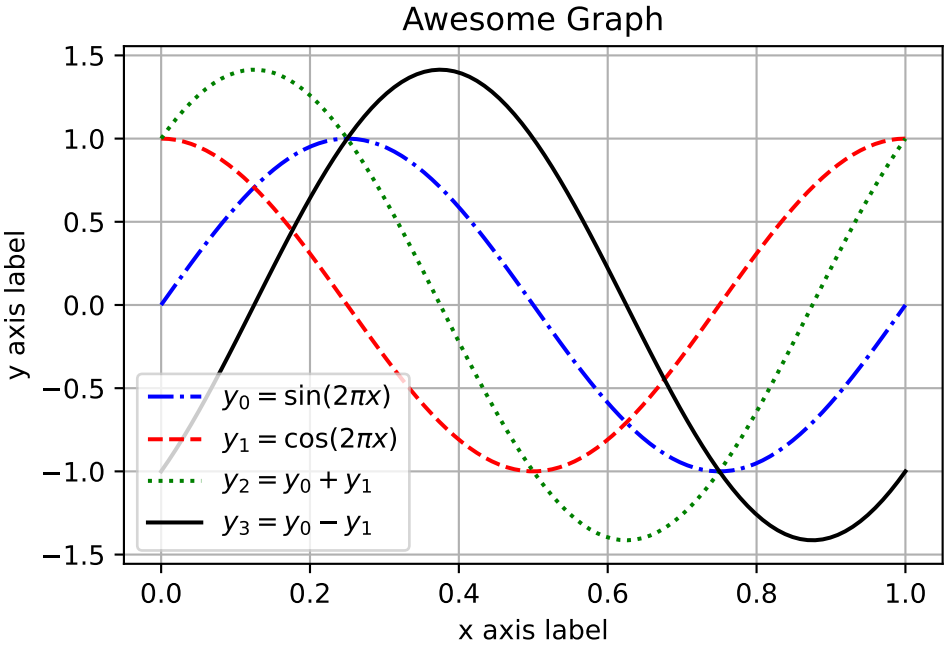

Figure A.3.: A second plot of the sine, cosine, and sums and differences.

A.4.2. Subplots

It is often very handy to place plots side-by-side or as some array of plots. The `subplots` command allows us that control. The main idea is that we are setting up a matrix of blank plots and then populating the axes with the plots that we want.

Example A.45. Let us repeat the previous exercise, but this time we will put each of the plots in its own subplot. There are a few extra coding quirks that come along with building subplots so we will highlight each block of code separately.

- First we set up the plot area with `plt.subplots()`. The first two inputs to the `subplots` command are the number of rows and the number of columns in your plot array. For the first example we will do 2 rows of plots with 2 columns – so there are four plots total.
- Then we build each plot individually telling `matplotlib` which axes to use for each of the things in the plots.
- Notice the small differences in how we set the titles and labels

- In this example we are setting the y-axis to the interval $[-2, 2]$ for consistency across all of the plots.

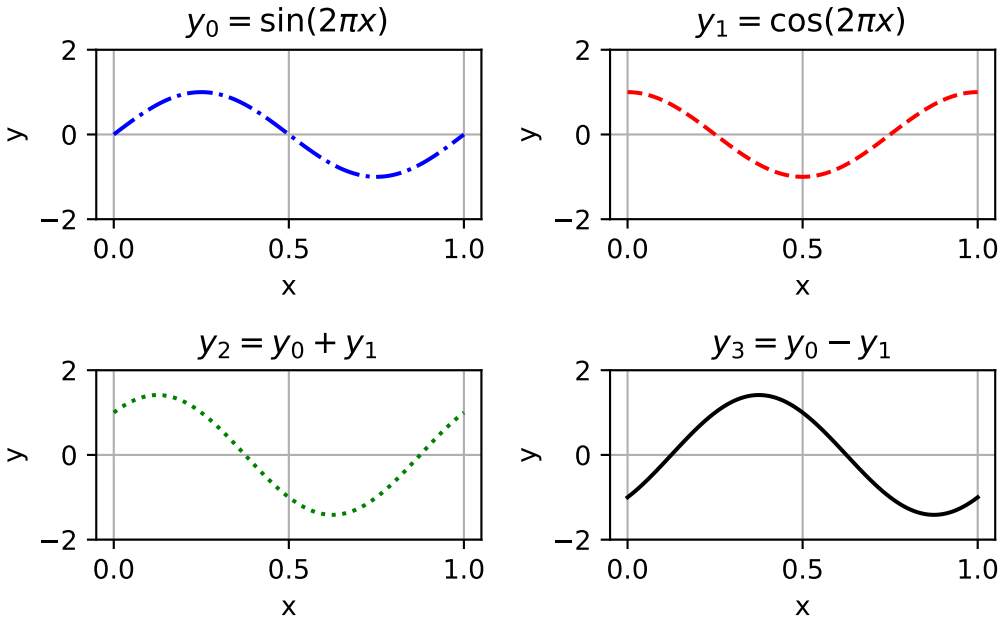

Figure A.4.: An example of subplots

The `fig.tight_layout()` command makes the plot labels a bit more readable in this instance (again, something you can play with).

Exercise A.17. Put the functions $f(x) = x^2$, $g(x) = x^3$ and $h(x) = x^4$ in a subplot environment with 1 row and 3 columns of plots. Use the unit interval as the domain and range for all three plot. Make sure that each plot has a grid, appropriate labels, an appropriate title, and the overall figure has a title.

A.4.3. Logarithmic Scaling with `semilogy`, `semilogx`, and `loglog`

It is occasionally useful to scale an axis logarithmically. This arises most often when we are examining an exponential function, or some other function, that is close to zero for much of the domain. Scaling logarithmically allows us to see how small the function is getting in orders of magnitude instead of as a raw real number. we will use this often in numerical methods.

Example A.46. In this example we will plot the function $y = 10^{-0.01x}$ on a regular (linear) scale and on a logarithmic scale on the y axis. We use the interval $[0, 500]$ on the x axis.

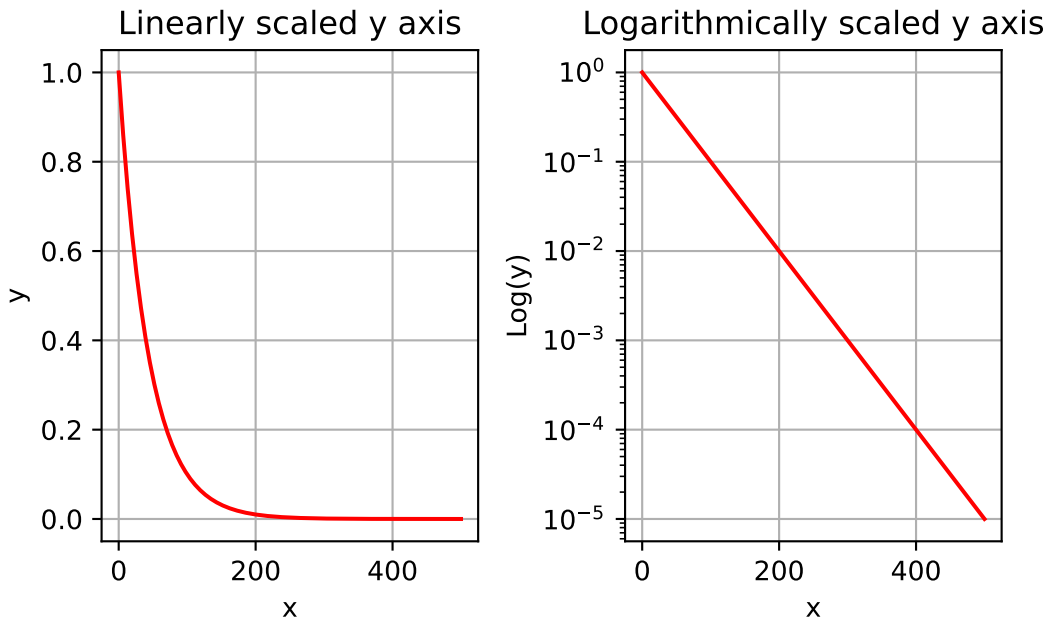

Figure A.5.: An example of using logarithmic scaling.

It should be noted that the same result can be achieved using the `yscale` command along with the `plot` command instead of using the `semilogy` command. So you could replace

```
axis[1].semilogy(x,y, 'r')
```

by

```
axis[1].plot(x,y, 'r')
axis[1].set_yscale("log")
```

to produce identical results.

Exercise A.18. Plot the function $f(x) = x^3$ for $x \in [0, 1]$ on linearly scaled axes, logarithmic axis in the y direction, logarithmically scaled axes in the x direction, and a log-log plot with logarithmic scaling on both axes. Use `subplots` to put your plots side-by-side. Give appropriate labels, titles, etc.

A.5. Dataframes with Pandas

The Pandas package provides Python with the ability to work with tables of data similar to what R provides via its dataframes. As we will not work much with data in this module, we do not need to dive deep into the Pandas package. In some of the optional exercises you will load in data from files using `pd.read_csv()`.

Example A.47.

	Time (sec)	Speed (ft/sec)
0	0	34
1	10	32
2	20	29
3	30	33
4	40	37
5	50	40
6	60	41
7	70	36
8	80	38
9	90	39

Example A.48. Pandas can also be useful to us for collecting computational results into tables for easier display. In this example we will build a table of the first 10 natural numbers and their squares and cubes. We then display the table.

	n	n^2	n^3
0	1	1	1
1	2	4	8
2	3	9	27
3	4	16	64
4	5	25	125
5	6	36	216
6	7	49	343
7	8	64	512
8	9	81	729
9	10	100	1000

This provides an alternative to how we created a tabular display with the `print()` function in Example A.20.