PYTHON MANUAL:
A LEARNING GUIDE FOR STRUCTURAL ENGINEERING STUDENTS

BY

KENNEDY A. GOMEZ, MEILEEN YEE, AARON C. DEWEY

SENIOR PROJECT REPORT

California Polytechnic State University, San Luis Obispo, CA
December 2023

Senior Project Advisor & Co-author:
Anahid A. Behrouzi, PhD

About the Authors

Aaron Dewey, Meileen Yee, and Kennedy Gomez

Kennedy, Aaron, and Meileen were fourth year undergraduate students studying Architectural Engineering (ARCE) at California Polytechnic State University in San Luis Obispo at the time of writing the Python manual for their ARCE senior project with advising from faculty member Anahid Behrouzi. The authors share an interest in structural engineering and its intersection with computer science and are passionate about helping students learn and use Python. While working on the manual, all three authors leveraged their unique programming background and skill set. As students, the authors were familiar with the curriculum and the use of Python in ARCE lab classes. They wanted to create a tool that would be useful for students just like themselves. The Python manual also aimed to motivate students to learn Python by showing the relevance and value of computer programming in structural engineering industry. Kennedy, Aaron, and Meileen hope this will be a resource future ARCE students use for years to come.

Acknowledgments

We would like to thank our professor and senior project advisor Anahid Behrouzi for presenting us with the idea for this manual and guiding us through the process. They would also like to acknowledge faculty member Peter Laursen for his technical review and feedback on this document. In addition, we would like to thank all the students and industry professionals who took the time to respond to our surveys.

Abstract

This Python programming manual is a resource to assist architectural, civil, or structural engineering students as they learn to create scripts to solve various structural analysis and dynamics problems. Beyond providing guidance on Python libraries that enable numerical and symbolic mathematics (NumPy, SciPy, SymPy), the manual also focuses on the creation of tabular and plot outputs useful to communicate results through technical reports (Pandas, Matplotlib).

The content of this document was developed based on detailed review of curriculum for three architectural engineering computing courses typically offered to upper class students at Cal Poly, San Luis Obispo as well as surveys with students who had completed the courses and industry members. The authors recognize that programming is challenging enough without having to use a textbook with example problems from other unfamiliar technical fields or where each Python library has its own separate documentation website to navigate.

The document includes 24 chapters on how to use both the Spyder integrated development environment and a range of Python programming topics. The chapters include explanations, graphics, and examples related to structural engineering so that students can follow and apply programming skills to their coursework, but also appreciate its broader utility for their careers. The goal with this resource is to address students' common knowledge gaps in Python along with building confidence and motivating them to learn how to program, better equipping them for success in the computing courses and in leveraging programming as a tool when they enter industry.

Table of Contents

1. Introduction

As someone going into the field of structural engineering, you may be asking: why do I need to know how to code? While it may not seem like an integral part of the job, you are more than likely applying code every single day. Be it structural analysis software, drafting applications, or a simple Excel spreadsheet; structural designers are constantly utilizing code to make their work more efficient, organized, and accurate. Having a deeper understanding of how coding language works allows us to not only make better use of applications but also extend their capabilities to solve any number of problems. Having this skill can also set you apart and add value to your team by potentially saving hundreds of hours on repetitive and detailed tasks that can be expedited by developing scripts to automate this work.

In fact, we conducted a survey of industry professionals, and you may be surprised to learn that *100%* of structural engineers who responded say they have coded as a part of their position. We learned about so many applications; for example, Cal Poly civil engineering alumni Jesse Bluestein, PE developed a program that takes AutoCAD plans and builds a full RISA 3D model from it. Several other survey respondents mentioned developing programs that automatically set up and run load cases through analysis software, that generate spreadsheets to organize and store analysis tool results, or even programs that determine the amount of embodied carbon in various floor build-ups. The possibilities are endless.

Implementing Python coding activities into the lab courses of the Cal Poly architectural engineering (ARCE) curriculum encourages students to always look for and create efficient solutions, like these, going forward. It also requires a deeper understanding of engineering topics since you need to deconstruct problems to create a solution in the form of code. In the Cal Poly ARCE department, Python is used in conjunction with courses in matrix structural analysis, structural dynamics, as well as various graduate level topics. This manual will look to provide a general overview of Python basics and go more in depth with respect to these course topics to ultimately help students as they enter their structural engineering careers with:

1. Manipulating, postprocessing, and visualizing large datasets
2. Automating repetitive calculations
3. Transferring data between CAD and structural analysis platforms

and more…

As we get started with programming, a brief summary on Python (What is it? Why is it so popular?) can be viewed at https://www.youtube.com/watch?v=Y8Tko2YC5hA .

1.1 Using this Manual

Coding in the context of this document refers to writing Python scripts that are executed from the command line rather than writing software (i.e., compiled programs). This manual is divided into sections each focusing on individual coding concepts; however, concepts in later sections may rely on information already explained prior.

Each section will start with a brief overview and then delve into examples and explanations formatted as follows:

```
# Blue boxes indicate areas containing Python code
# A horizontal black line separates input code from output code
# Output code is also prefaced with a ‘>>’


print('Hello World!')
```

White text boxes within blue sections provide commentary. This is not a part of the code itself.

```
>>Hello World!
```

Users of this manual can copy and paste input code directly from examples into the command window of the Spyder (Python coding environment) to produce the output code shown. `Courier font` is also used to indicate coding language outside of these blue sections.

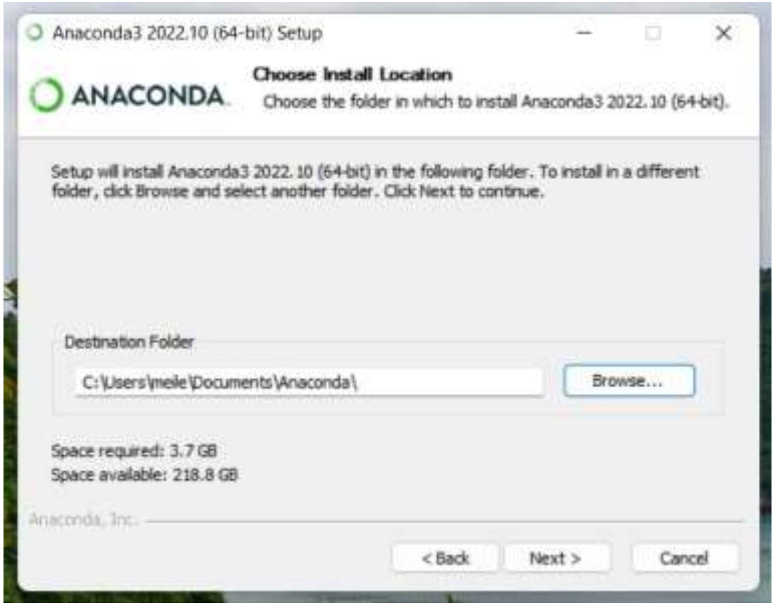
Green boxes indicate a *Motivation Station*. These sections are meant to provide some inspiration for how these coding topics can or have been used! A lot of the information found here is based on responses from a survey of professional engineers and how they use coding to solve problems in structural engineering.

2. Downloading Anaconda

An Integrated Development Environment (IDE) is a software through which you utilize Python. There are multiple IDEs, and we recommend using Spyder. Download Spyder through Anaconda so that you can access all libraries that come with Anaconda.

Follow these steps to download Anaconda on your Windows laptop or computer (note the exact language and organization on the Anaconda website may have changed since the publication of this manual, but the process should be largely similar):

1. Go to Anaconda's website > **Products** > **Anaconda Distribution** or click here: https://www.anaconda.com/products/distribution
2. Under "Product Distribution" click "Download"
3. Go to your downloads and open the Anaconda installer application
4. Follow the prompts, and choose a file path and destination folder that you will remember



Anaconda3 2022.10 (64-bit) Setup

Choose Install Location

Choose the folder in which to install Anaconda3 2022.10 (64-bit).

Setup will install Anaconda3 2022.10 (64-bit) in the following folder. To install in a different folder, click Browse and select another folder. Click Next to continue.

Destination Folder: C:\Users\meile\Documents\Anaconda\

Space required: 3.7 GB
Space available: 218.8 GB

5. Once Anaconda is downloaded, go to the destination folder that you chose previously and open this application to access Python

Follow these steps to download Anaconda on your Mac laptop or computer:

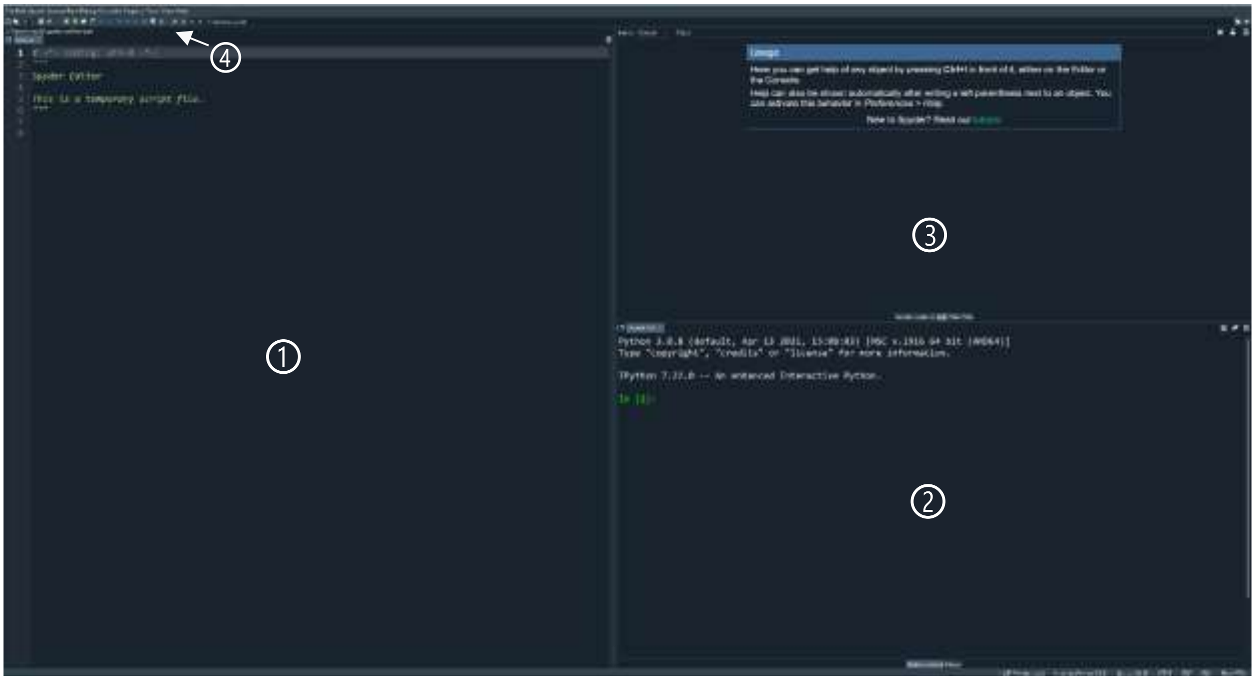
1. Go to Anaconda's website > **Products** > **Anaconda Distribution** or click here: https://www.anaconda.com/products/distribution
2. Under "Get Additional Installers" click the Apple icon
3. Click on "64-Bit Graphical Installer (688 MB)"
4. Go to your downloads and open the Anaconda installer application
5. Follow what is prompted, and choose a file path and destination folder that you will remember
6. Once Anaconda is downloaded, go to the destination folder that you chose previously and open the app to access Python

3. Spyder Interface

This section provides an overview of how to utilize different aspects of the Spyder interface, manage your files, and set up basic structure of your code.

3.1 Spyder Interface Layout

The picture below displays what Spyder looks like when you first open the software.



① **Script window**: This is where you type the script of your code and run your code.

② **Output/Command window**: This is where your code will output or “print” results when it is run. You can also utilize the command window to troubleshoot by typing individual lines of code of operations and functions to see an immediate result (see Section 23.3 for how to use the command window to troubleshoot errors).

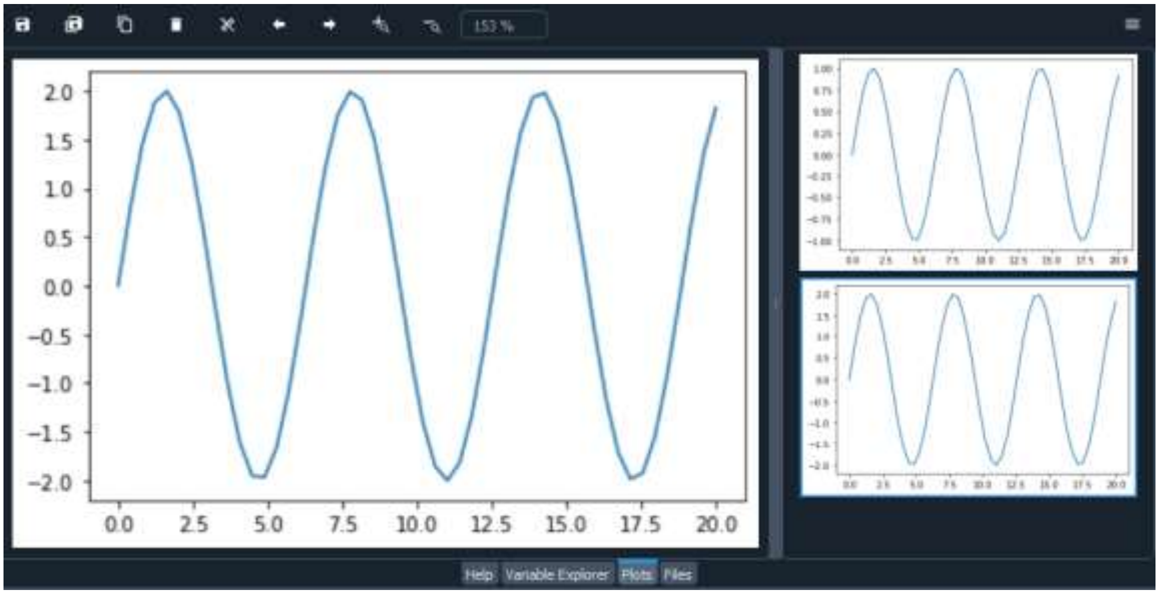
Quick tips:

- Type `clear` in the command window to clear it of prior commands and output
- Hit the up arrow on your keyboard to repeat the last command (or hit the up arrow multiple times to access earlier commands)

③ **Variable Explorer/Help/Plots/Files/Find window**: Out of these five different tools in this window, you will mostly utilize Variable explorer (see Section 4 for more information on variables) and Plots (see Sections 18 and 19 for more information on plotting) to see your variables and plots.

Name	Type	Size	Value
bee	str	5	sting
m	int	1	50
n	list	1	[50]
p	Array of int32	(3, 1)	[[1] [2]]
t	Array of int32	(2, 3)	[[1 2 3] [4 5 6]]

Variable explorer: This is where you can view all the variables *after* running the lines in your script where they are defined. The variable explorer is helpful because it summarizes all the relevant information about your variables, including the name, data type, size, and value(s) of each. You can also double-click on any array in variable explorer to see a grid format of the array.



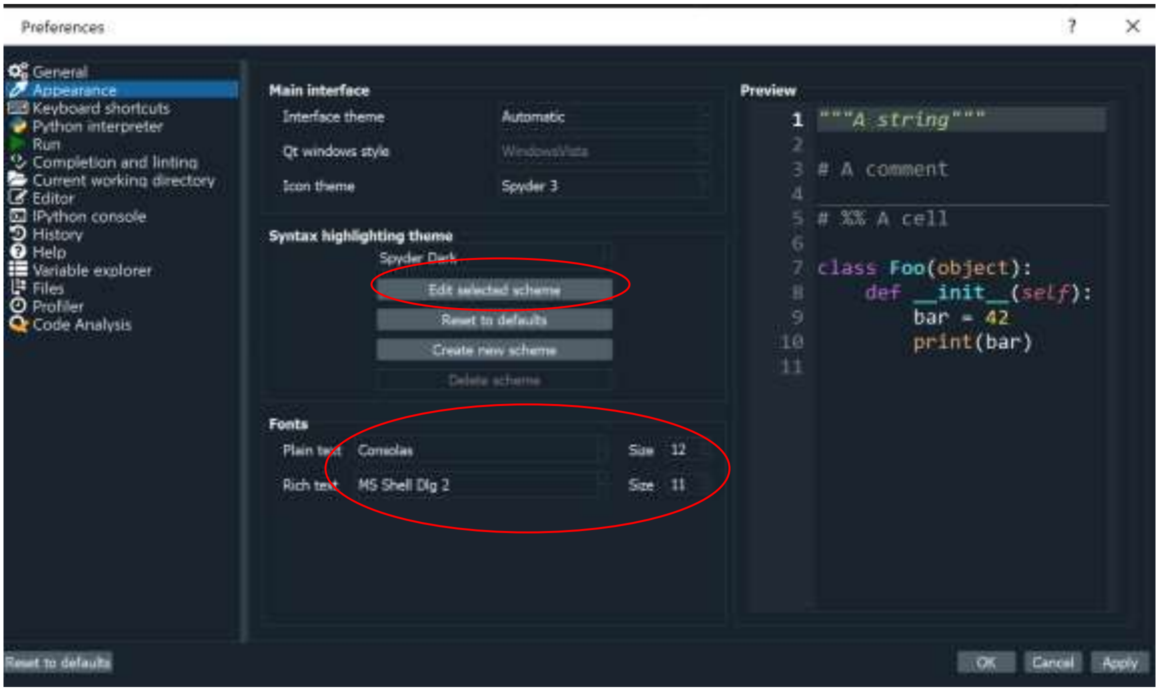
Plots: This is where you can all the plots that have been generated after running the lines in your script where they are defined. It is possible to zoom in/out as well as save or copy the plot from this interface into a report. However, it is not possible to modify the formatting of the plot within this "inline" window. This is possible by changing the plot viewing preference (see Section 3.2 for instructions).

④ **Top Ribbon Icons**: Some of the ribbon icons that will be useful have been circled in red.

- The green play button will run your entire script
- The icon to the right of it will run the current cell (see Section 3.4 for how to use cells)
- The white icon with the cursor will run the current line or a highlighted section of code
- The four arrows pointing outward or inward will make Python full screen on your device or will exit full screen
- The wrench will open your preferences

3.2 Setting Preferences

To change your font size, font type, or color theme, go to the top ribbon: **Preferences** (the wrench symbol) > **Appearance**.



Interface theme: Automatic
Qt windows style: WindowsVista
Icon theme: Spyder 3
Syntax highlighting theme: Spyder Dark
Plain text: Consolas Size: 12
Rich text: MS Shell Dlg 2 Size: 11

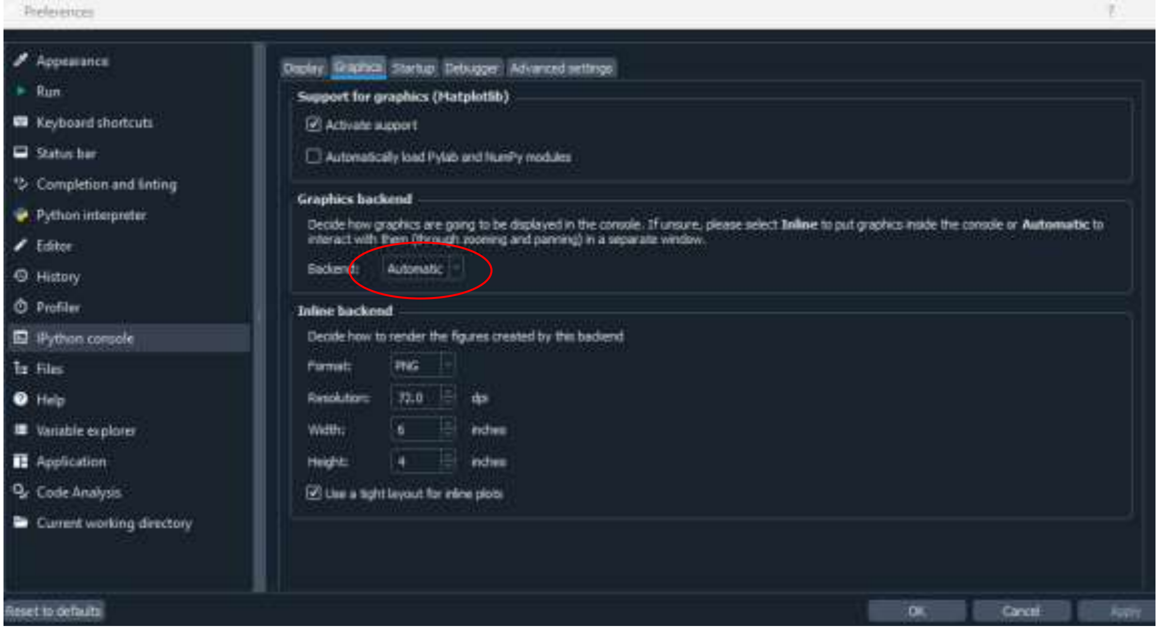
```
"""A string"""

# A comment

# %% A cell

class Foo(object):
    def __init__(self):
        bar = 42
        print(bar)
```

Changing font: Select a font from the dropdown under “Fonts.” “Plain text” is the text that appears in script, output, and variable explorer windows; whereas “Rich text” is text that appears in the Help window.

Changing font size: Select a font size from the dropdowns next to “Size.” See the above statement on “Plain text” versus “Rich text.”

Changing background/text color theme: Choose from the dropdown under “Syntax highlighting theme,” where “Spyder Dark,” is the default theme. For a coding environment with a white background and distinct font colors you can consider trying the theme “IDLE”. As a note, the visuals and associated descriptions in this manual are produced in the “Spyder Dark” theme.

To change your plot viewing window, go to the top ribbon: **Preferences** (the wrench symbol) > **IPython console** > **Graphics** > **Graphics backend** > **Automatic**.



Support for graphics (Matplotlib)

☑ Activate support

☐ Automatically load Pylab and NumPy modules

Graphics backend

Decide how graphics are going to be displayed in the console. If unsure, please select **Inline** to put graphics inside the console or **Automatic** to interact with them (through zooming and panning) in a separate window.

Backend: Automatic

Inline backend

Decide how to render the figures created by this backend

Format: PNG

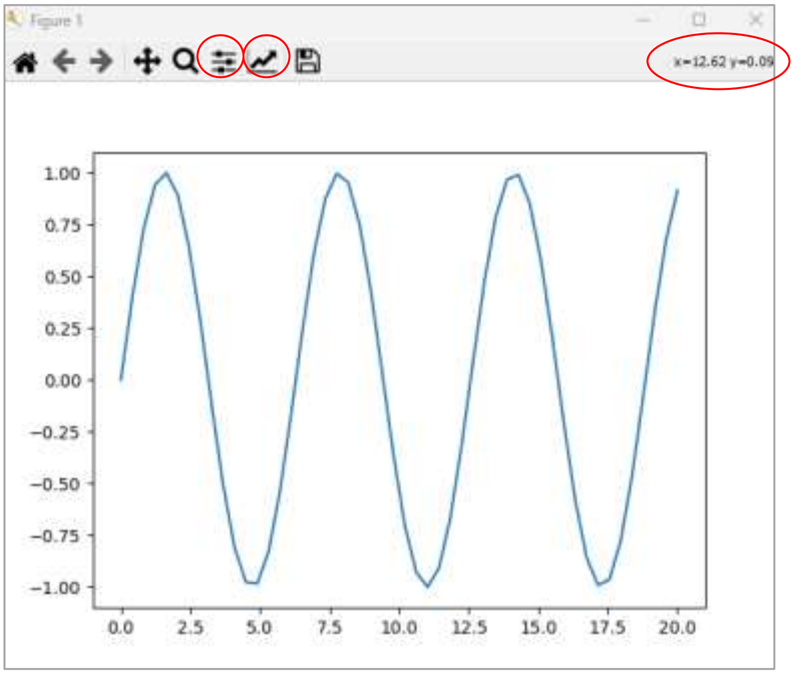
Resolution: 72.0 dpi

Width: 6 inches

Height: 4 inches

☑ Use a tight layout for inline plots

This will open the figure in a separate window. It may be necessary to click on a new taskbar icon that appears at the bottom of your computer screen to open. Other than panning /zooming and saving the figure a few other useful functions are circled in red and described below.



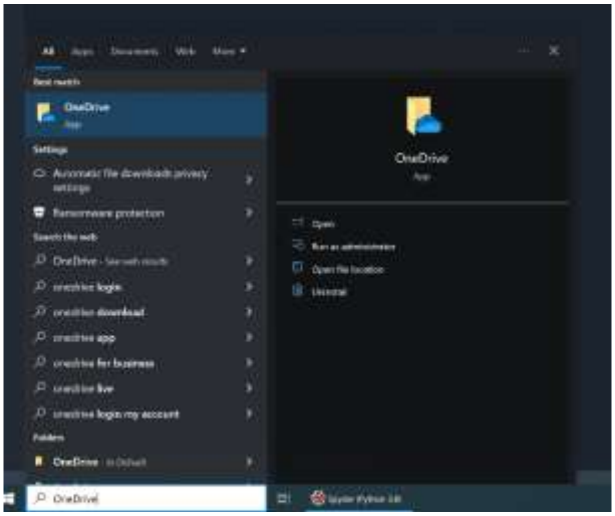
Changing the size of the graph within the figure: selecting the icon with slider bars allows you to change the border dimension and other spacing of the graph in the figure.

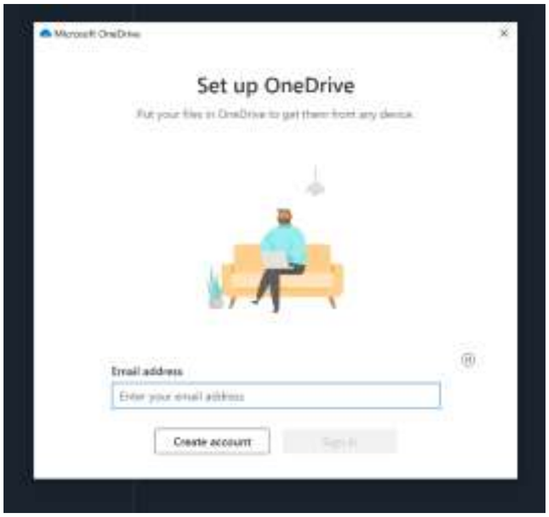
Changing graph formatting: selecting the icon with the curve on graph axes allows you to change the axes settings including specifying lower and upper bound values, a scale for axis tick mark spacing (linear vs. logarithmic), figure title, and axes labels. It is also possible to change curve parameters including the name to use in the legend, as well as line and marker styles/size/color. You can also turn the legend on/off from this interface.

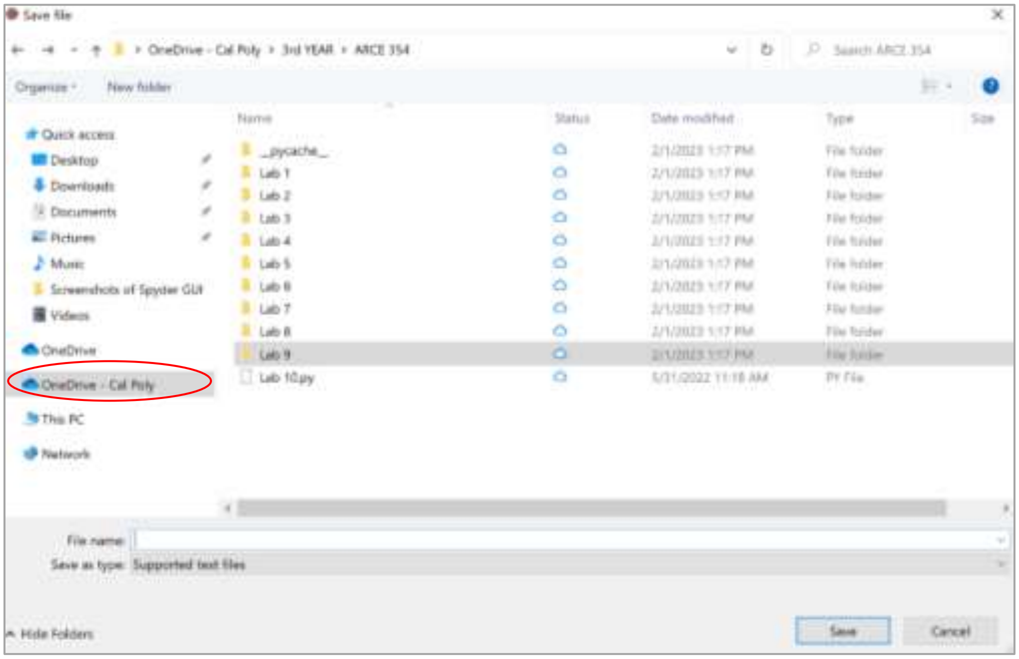
Displaying x-y coordinates: the display in the upper right of the figure window will update with new x-y coordinates when moving your mouse cursor over any location within the figure space, which can be helpful to identify critical points in the graph while analyzing data.

3.3 Managing Files

Saving a Python File: When saving a Python file, we recommend saving it to your OneDrive for easy access on different devices. Follow these steps to save a Python file to your Cal Poly OneDrive on a school computer:







1. Open the OneDrive application on the computer
2. Sign into OneDrive using your Cal Poly email
3. In Python, go to **File** > **Save File**
4. Save your Python file under "OneDrive – Cal Poly" (make sure it includes " – Cal Poly")

Sharing a Python File: To share a Python file with yourself or another person, utilize a shared folder on OneDrive or email the Python file in a zipped folder, as non-zipped Python files cannot be emailed. Follow these steps to save a Python file in a zipped folder/as a zipped file:

1. In Python, go to **File** > **Save File**
2. Save your Python file wherever you want
3. In File Explorer, right click on the Python file and click on "Compress to ZIP file"

3.4 Setting Up Your Code

Now that you know how to save your file, you can start on your code! Here is a basic layout of how we recommend setting up your code.

```
# -*- coding: utf-8 -*-
"""
#Author: Musty the Mustang
#Date: Wed, Feb 1, 2023
#Course: ARCE 412, Lab #0
#Description: How to set up your code
"""
#%% Import libraries

import numpy as np

#%% Set Variables

a = 32  #first variable
b = 70  #second variable
n = np.array([a],[b])   #array

#%% Problem 1


```

The lines of code shown above have four main parts: the header, importing libraries, setting your variables, and the rest of your code.

① **Header**: In the portion of your code between the lines with """, include your name, date, course, lab number of the assignment, and a description of what tasks your code is executing in this Python file. Using a header is good bookkeeping practice, especially as you enter industry if you share code you produced with others in your structural engineering office or even when referring back to code you may have written years prior.

② **Importing libraries**: If you will be using libraries in your code, which is highly likely, import them at the beginning of your code. You can import a library at any point before a line in which you are using the library, but importing the libraries at the top of your code makes for good organization. Rename them using a short abbreviation for your convenience in the form `import [library] as [abbreviation]`. See Section 15 for more information on libraries.

③ **Setting variables**: As with importing libraries, you can set a variable at any point before a line that uses the variable, but we recommend defining important variables at the beginning of your code for good organization so that you can easily view and modify these variables (described in Section 4 and 17).

④ **The rest of your code**: The rest of your code can be set up to your preference, depending on what problems you are solving or what you are programming.

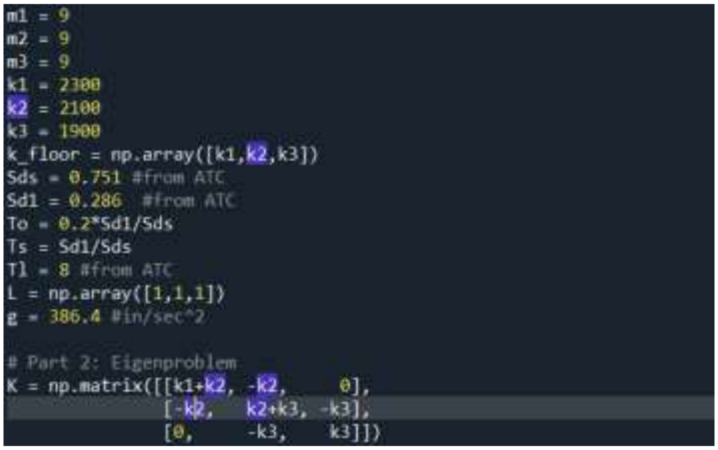
There are two other elements shown above that will be useful to utilize:

Comments: Commenting allows you to make notes without interfering with execution of your code. It can be utilized on its own line to explain what an entire section of code does or next to a single line with text (for example, to indicate the units used for variables). To make comments, simply type `#` followed by text. Comments will be shown in gray text.

Cells: Cells break up your code visually and are a helpful organizational tool. A cell can be individually run using a button in the top ribbon (see Section 3.1 for useful ribbon buttons), which is helpful for troubleshooting issues in your code (see Section 23 for troubleshooting your code). To set a cell, type `#%%` and the following lines will be included in that cell. Visually this is delineated with a horizontal gray line at the beginning of the cell.

Other Quick Tips

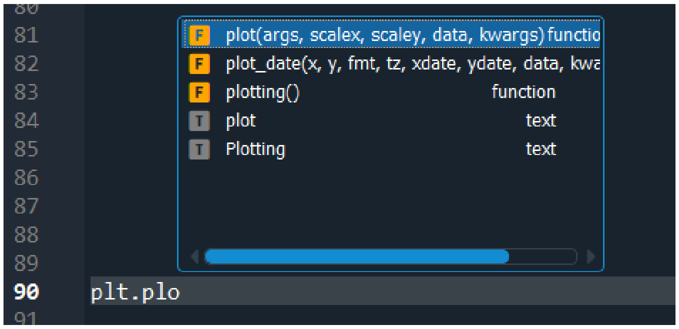
- When your cursor is placed just to the right of a variable, that variable will be highlighted in purple/blue wherever it appears in your code.



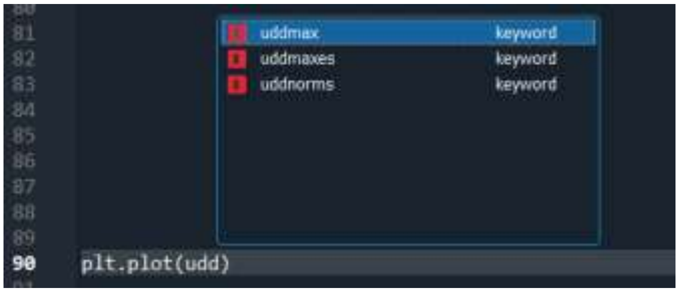
```
m1 = 9
m2 = 9
m3 = 9
k1 = 2300
k2 = 2100
k3 = 1900
k_floor = np.array([k1,k2,k3])
Sds = 0.751 #from ATC
Sd1 = 0.286  #from ATC
To = 0.2*Sd1/Sds
Ts = Sd1/Sds
Tl = 8 #from ATC
L = np.array([1,1,1])
g = 386.4 #in/sec^2

# Part 2: Eigenproblem
K = np.matrix([[k1+k2, -k2,     0],
               [-k2,   k2+k3, -k3],
               [0,     -k3,    k3]])
```

- Autocomplete: Sometimes a small gray box will pop up while you are typing your code in Spyder to try to help you autocomplete to either functions that exist in the libraries you have imported or variables you have already defined. It is recommended that users examine the contents of the pop up to help determine what inputs they need to provide to functions and to correctly reference previously defined variables.

  If the autocomplete does not appear to be working then update **Preferences** (the wrench symbol) > **Help** > **Automatic connections** and toggle the desired plug-in to show an object's help information.



```
plt.plo
```



```
plt.plot(udd)
```

4. Variables

Variables can contain any type of data and they allow programmers to abstract and manipulate data. When data is assigned to a variable, the variable is internally associated with that data and can then be used in place of the data. Data types are described in depth in Section 6.

Example 4.1

```
a = 'this data is stored in a'
print(a)
```

```
>> this data is stored in a
```

Example 4.2

```
a = 14
b = 9
c = (a - b)*2
print(c)
```

```
>> 10
```

Notice, variables interact as though they are the data they are representing. Variables act exactly like the data they contain because the data internally replaces the variable each time the variable is evaluated. See Section 5.4 to understand the order of evaluation presented in the example.

Example 4.3

```
a = 14
a = 10
print(a)
```

```
>> 10
```

Notice variables can be overwritten so that the first data stored in the variable is lost. It is best practice to avoid overwriting variables.

4.1 Naming Variables

Variables can be named almost anything; however, it is important to use descriptive variable names for clarity. There are many different practices for naming variables and over time each programmer will develop his or her own style. Variable names have some constraints, they can only consist of alpha characters (A-Z), numbers (0-9), and underscores (_). Variable names cannot start with a number. Furthermore, variable names cannot be Python syntax terms, such as functions from a library, and will overwrite functions if they are named the same. Also note, that variables are case sensitive, so `X` is not the same as `x`.

Invalid Variable Names	Valid Variable Names	Why is the variable name invalid?
beam 2	beam_2	Names cannot contain spaces
x,0	x_0	Names cannot contain commas
for	FOR	“for” is a predefined Python term
3tree	tree3	Names cannot start with a number
#num	num	Names cannot contain hashtags

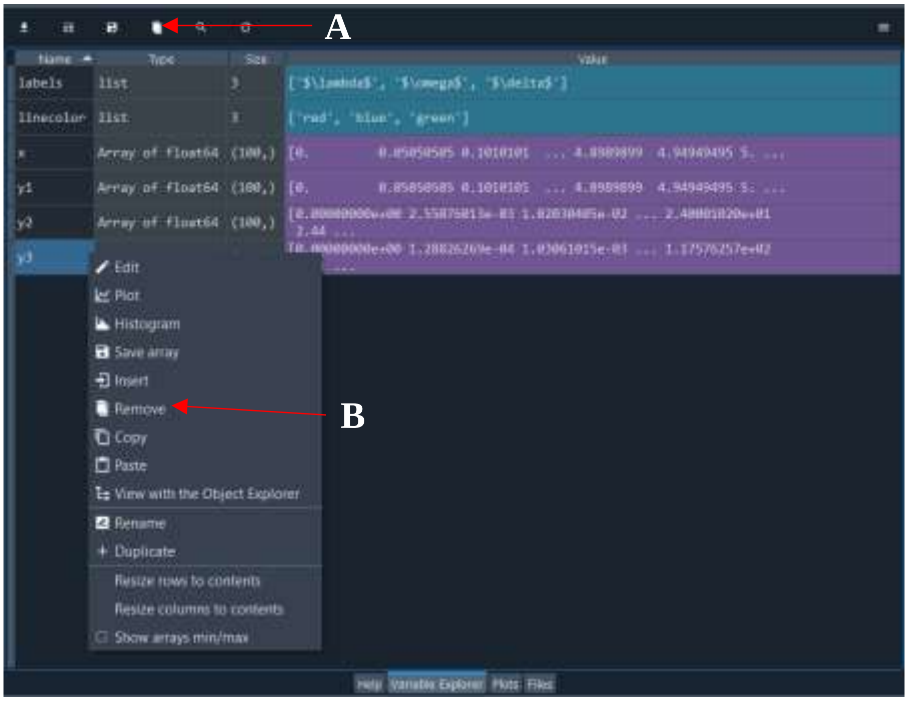
For discussion on local and global variables see Section 10.2.

4.2 Clearing

When editing and running a code several times, you can often end up with many variables, plots, or code in the command window. Python will not automatically delete old variables after rerunning code so you may be looking at inaccurate values. Therefore, it is helpful to clear old information quickly.

There are a couple ways to do this including the delete buttons in the variable explorer as shown in Figure 4.2.1.

Figure 4.2.1 Delete Button



Name	Type	Size	Value
labels	list	3	['λ', 'ω', 'δ']
linecolor	list	3	['red', 'blue', 'green']
x	Array of float64	(100,)	[0. 0.05050505 0.1010101 ... 4.8989899 4.94949495 5. ...
y1	Array of float64	(100,)	[0. 0.05050505 0.1010101 ... 4.8989899 4.94949495 5. ...
y2	Array of float64	(100,)	[0.00000000e+00 2.55076013e-03 1.02030405e-02 ... 2.40001020e+01 2.44 ...
y3			[0.00000000e+00 1.28826269e-04 1.03061015e-03 ... 1.17576257e+02 ...

A. Delete all variables
B. Delete selected variable (right click on variable for drop-down menu)

However, it may be more efficient to use the command window to delete variables. Listed below are methods for clearing the variable explorer executed via code in the command window.

Function	Description
%reset	Clears all user defined variables
del a, b	Clears chosen variables (i.e., a and b)
clear	Clears the command window
exit	Clears everything your code has created, including variables, plots, and the command window

If you want all variables to automatically delete every time you run your code (recommended), use the following lines at the beginning of your script:

```
from IPython import get_ipython
get_ipython().magic('reset -sf')
```

5. Operators and Expressions

Python supports all the standard math and logic operators as well as some other useful functions. Operators combined with data are used to form expressions. Informally, an expression is just a bit of code that produces a value when it is run.

5.1 Numeric Operators

Operator	Description	Example
+	Addition	1 + 2 = 3
-	Subtraction	4 – 2 = 2
*	Multiplication	5 * 2 = 10
/	Division	7 / 2 = 3.5
**	Exponent	2**3 = 8
%	Modulus	10 % 3 = 1
//	Floor Division	7 // 2 = 3

A few of these numeric operators are uncommon in traditional calculations but can serve a very useful purpose in programming. These are modulus (remainder in a division calculation) and floor division (division solution rounded down to the nearest integer).

5.2 Boolean Operators

Comparative operators and logic operators will always evaluate to a Boolean (either `True` or `False`).

Operator	Description	True Examples	False Examples
==	Equal	1 == 1	3 == 2
!=	Not equal	1 != 3	5 != 5
<	Less than	2 < 4	2 < 2
>	Greater than	5 > 4	3 > 4
<=	Less than or equal to	3 <= 3	6 <= 5
>=	Greater than or equal to	1 >= 0	2 >= 8
not	Switches Boolean	not False	not True
or	Evaluates to True if either are True	False or True	False or False
and	Evaluates to True only when both are True	True and True	False and True

5.3 Boolean Truth Tables

The expected outputs of Boolean operations (`and`, `or`) can be expressed diagrammatically in truth tables. Truth tables show the expected output for each corresponding input variation.

Truth Table (`and`)

`A and B`	`A = True`	`A = False`
`B = True`	`True`	`False`
`B = False`	`False`	`False`

Truth Table (`or`)

`A or B`	`A = True`	`A = False`
`B = True`	`True`	`True`
`B = False`	`True`	`False`

5.4 Order of Evaluation

The key to writing expressions is understanding the order Python will evaluate each portion or subexpression. A precedence chart shows exactly the order of operations for a programming language. The table below is an abbreviated Python precedence chart with some common expressions. When there is no preference between subexpressions, Python will evaluate the line from left to right.

Order	Operator	Description
1	( )	Parentheses
2	**	Exponent
3	* / % //	Multiply, Divide, Modulus, Floor Division
4	+ -	Add, Subtract
5	<= < > >=	Comparison Operators
6	== !=	Equality Operators
7	is is not	Identity Operators
8	in not in	Membership Operators
9	not or and	Logical Operators

The following examples diagrammatically show expressions evaluated step by step. If these operations are typed in the command window, Python will compute these steps internally and only display the final value. In these examples, lines that start with => indicate an intermediate calculation step, and only the final answer after >> will appear in the command window.

Notice expressions are evaluated from left to right when there is no subexpression preference:

```
2 + 2 + 5
=> 4 + 5
>> 9
```

Notice multiplication is evaluated before addition:

```
3 + 4 * 2
=> 3 + 8
>> 11
```

Notice the subexpression in the parentheses is evaluated first:

```
(3 + 4)  * 2
=> 7 * 2
>> 14
```

Notice conditional expression evaluate to Booleans first, and then logical operators are executed:

```
True and not 3 > 4
=> True and not False
=> True and True
>> True
```

```
5 - (7 + 2) * 3 +1
=> 5 - 9 * 3 +1
=> 5 - 27 +1
=> -22 +1
>> -21
```

Notice that the variable `num` must be evaluated as a separate step and that step does not occur until `num` is needed:

```
num = 4
2 * num - (3 + 4)
=> 2 * num - 7
=> 2 * 4 - 7
=> 8 - 7
>> 1
```

```
result = True
False or (( 3 >= 3) and result)
=> False or (True and result)
=> False or (True and True)
=> False or True
>> True
```

6. Data Types

Python, like every coding language, has different types of data. It is important to know which data type you are using because most operators and functions require a specific type of data. However, some functions and operators will work with several types of data. A resource for learning about data types: https://www.w3schools.com/python/python_datatypes.asp .

6.1 Strings

A string is a data type that stores written information. Strings can use any typed character. They often contain letters and words, but can also use numbers and special characters (see Section 20.4). To write a string enclose the contents of the text with single or double quotation marks.

```
'Words: 1'
>> 'Words: 1'
```

6.2 Integers

Integers are whole number values. Integers can be positive or negative but not fractional or decimal values. Many mathematical operations will return integers.

```
1 + 1
>> 2
```

6.3 Floats

Floats are floating point numbers, meaning that unlike integers they contain decimal values (e.g., 2.718); however, a whole number followed by a decimal containing a zero is also a float (e.g., 3.0). Floats can be positive or negative. Many mathematical expressions will return floats even if integers are used in the expression when the result is fractional in nature (e.g., 7/3). Floats cannot be larger than $\pm 1.798 \times 10^{308}$ or result of infinity (represented by inf) will result.

```
1 + 1.5
>> 2.5
```

Notice that in the example above 1 is not a float but 1.5 and 2.5 are. In most contexts, Python allows integers and floats to be used in expressions together without producing an error message.

6.4 Booleans

Booleans are the binary logical conclusions: True and False. Booleans commonly result from conditional expressions (e.g., 4 < 2 producing a result of False) and they can be assigned to variables just like strings, integers, and floats. Booleans are critical for the flow of `if`, `elif`, and `else` statements (see in Section 11).

```
6 > 4
>> True
```

6.5 Identifying Data Types

When troubleshooting, it is often useful to check the data type of variables. The Spyder interface displays all defined variables and their data types in the Variable Explorer Window (see Section 3.1). A variable’s data type can also be determined by using the `type()` function.

Example 6.5.1

```
a = 1
b = 2.718
c = 'or to take arms against a sea of troubles' # Hamlet
print(type(a))
print(type(b))
print(type(c))
```

```
>> <class ‘int’>
>> <class ‘float’>
>> <class ‘str’>
```

Data Types	Python Notation	Description	Example/Reference
Strings	str	Typed characters	'word', "$0.99", 'My name is '
Integers	int	Whole numbers	1, 5920, -23
Floats	float	Floating decimal numbers	0.5, 3.14, 1.0
Booleans	bool	Logical conclusions	True, False
Lists	list	Ordered set of elements	[0,1,2] (See Section 7)
Dictionary	dict	Unordered table of elements	{ 'a':1, ' b':2} (See Section 8)
Functions	function	Callable code	(See Section 10)

6.6 Converting Between Data Types

It is often necessary to convert one type of data to another because a function or expression requires a specific type of data. For example, to concatenate an integer or float to a string the integer or float must be converted to a string.

Float converted to string:

```
str(1.5)
```

```
>> '1.5'
```

String combined with integer converted to string:

```
'score: ' + str(98)
```

```
>> 'score: 98'
```

Float converted to integer (results in float being truncated to the whole number portion):

```
int(1.5)
```

```
>> 1
```

Integer converted to float (results in a decimal value of zero being included in number):

```
float(2)
>> 2.0
```

7. Lists

Lists are ordered containers of data. In Python, lists can hold strings, integers, floats, Booleans, or a combination of any of those. Lists can even contain other lists! Lists are an extraordinarily useful way to store data.

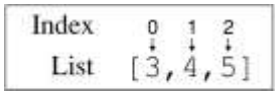
7.1 Indexing

One useful way to access the contents of a list is by indexing. Every element in a list has a position or index. The element in the first position has an index of 0. The element in the second position has an index of 1 and so on (see Example 7.1.1)

Example 7.1.1

```
L = [3,4,5]
print(L[0])
print(L[2])
```

```
>> 3
>> 5
```

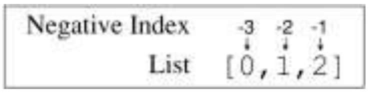


Aa list can also be indexed from the end. In Example 7.1.2, you can see that last element in a list is assigned an index of -1 the prior entries have increasing negative index values.

Example 7.1.2

```
L = [0,1,2]

print(L[-1])
```

```
>> 2
```

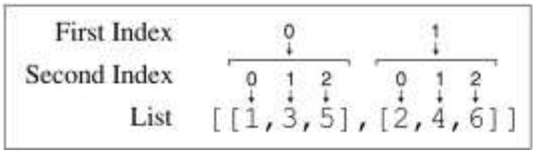


In Example 7.1.3, `L` is a list that contains two lists. The index of `0` is evaluated to select the first list then the index of `2` is evaluated to select the third element in that list.

Example 7.1.3

```
L = [[1,3,5],[2,4,6]]

print(L[0][2])
```

```
>> 5
```



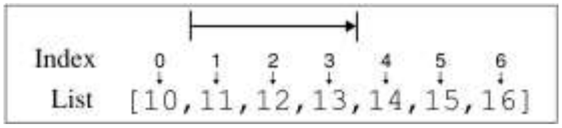
7.2 Slicing

Sometimes it is useful to slice or select a range of values in a list. The range of the desired indexes can be specified with this syntax: [first index : last index + 1]. For Example 7.2.1, where L[1:4] the indexing can be interpreted as the inequality expression: 1 ≤ index < 4.

Example 7.2.1

```
L = [10,11,12,13,14,15,16]

print(L[1:4])
```

```
>> [11,12,13]
```



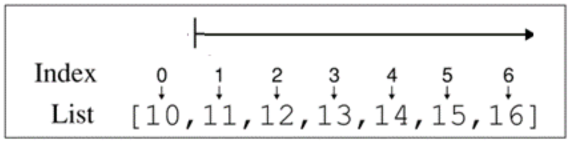
It is also possible to slice from one index to the end of the list or from the beginning of the list up to an index using an colon : (see Example 7.2.2).

Example 7.2.2

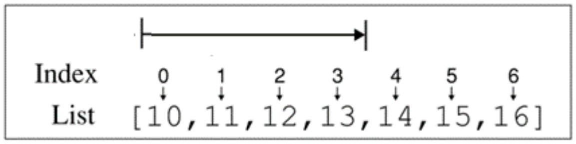
```
L = [10,11,12,13,14,15,16]

print(L[1:])

print(L[:4])
```

```
>> [11,12,13,14,15,16]

>> [10,11,12,13]
```

`print(L[1:])`



`print(L[:4])`



7.3 List Operations

There are many standard functions to help use and manipulate lists, here are three of these:

Operator	Description	Example Input	Example Output
+	Adds elements of list, makes new list	[1, 2] + [3, 4]	[1, 2, 3, 4]
.append()	Appends element to one index at the end	L = [1, 2] L.append(3)	[1, 2, 3]
len()	Returns the length of the list	len([0,1,2])	3

Example 7.3.1

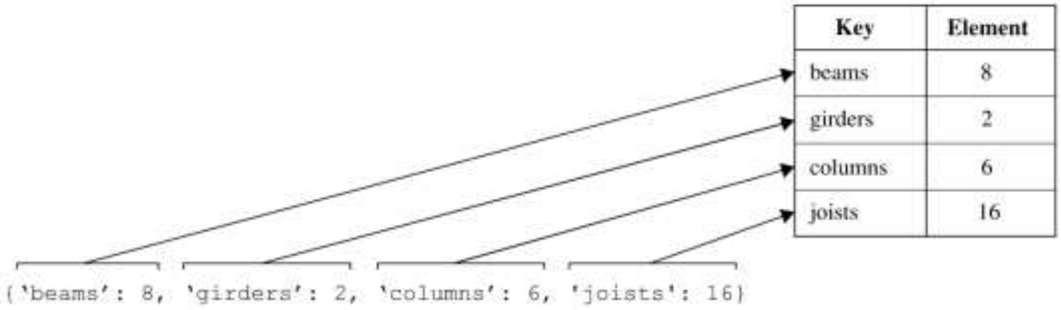
```
L1 = [7,2,6]
L2 = [0,0,0]
L3 = L1 + L2
L1.append(L2)
print(L3)
print(len(L3))
print(L1)
print(len(L1))
```

```
>> [7,2,6,0,0,0]
>> 6
>> [7,2,6,[0,0,0]]
>> 4
```

Notice the distinction between the two list combination approaches: L1 + L2 and L1.append(L2). The first approach maintains a single list and adds in the new entries, while the latter inserts a second inner list after the entries from the preexisting outer list. Since the length function only counts the number of elements in the outer list, the inner list of zeros that has been appended in L1.append(L2) is counted as one element.

8. Dictionaries

A dictionary is another data type similar to lists in that it contains a set of elements of any data type. However, dictionaries and lists differ in how they access elements. Instead of using the position index, dictionaries can use user set keys. Every element in a dictionary is accessed by a key. Unlike indexes, keys can be almost any data type (strings, integers, floats, Booleans, etc.). In the following examples all the keys are strings.



Example 8.1 Build and Access a Dictionary

```
# Create a dictionary named members with keys: beams, girders,
and columns.

members = {'beams': 8, 'girders': 2, 'columns': 6}

# Add an additional key to the members dictionary: joists.

members['joists'] = 16

# Print dictionary and data stored under individual keys.

print(members)
print(members['beams'])
print(members ['joists'])
```

```
>> {'beams': 8, 'girders': 2, 'columns': 6, 'joists': 16}
>> 8
>> 16
```

Notice elements can be inserted into the dictionary after it is created. An empty dictionary can be initiated with curly braces (`{}`) or the `dict()` function.

Example 8.2 Calculate moment and deflection of framing members using similar dictionaries.

```
# Set up dictionary entries defined by identical keys. For the
beams listed, the keys are: shape, moment of inertia, and span.

beam1 = {'shape': 'W6X25', 'I': 53.4, 'span': 240} #in^4, in
beam2 = {'shape': 'W6X20', 'I': 41.4, 'span': 120} #in^4, in
beam3 = {'shape': 'W6X15', 'I': 29.1, 'span': 150} #in^4, in

# Create the dictionary called framing from the entries above.

framing = [beam1, beam2, beam3]

# Compute moment and deflection for each beam in the framing
dictionary, treat as simply supported with uniform load.

w = 0.1 # k/in
E = 29000 # ksi
for ii in range(0,len(framing)):
    M = w*(framing[ii]['span']**2)/8
    dfc = 5*w*(framing[ii]['span']**4)/(384*E*framing[ii]['I'])
    print(framing[ii]['shape'])
    print('Moment =', str(M), 'k-in')
    print('Deflection =', str(round(dfc,2)), 'in\n')
```

```
>> W6X25
>> Moment = 720.0 k-in
>> Deflection = 2.79 in
>>
>> W6X20
>> Moment = 180.0 k-in
>> Deflection = 0.22 in
>>
>> W6X15
>> Moment = 281.25 k-in
>> Deflection = 0.78 in
```

Dictionaries help organize data that follow a similar structure. Keys for all entries in a dictionary - for framing: `shape`, `I`, and `span` - must have identical names to access data in a `for` loop (see Section 12).

Note: the `round` function was used in this script to round the deflection `dfc` to two decimal spaces, to learn more about this function visit this link.

Example 8.3 Build and access nested dictionary structure

```
# Set up dictionary entries, beams like in prior example.

beam1 = {'shape': 'W6X25', 'I': 53.4, 'span': 240} #in^4, in
beam2 = {'shape': 'W6X20', 'I': 41.4, 'span': 120} #in^4, in

# Assign beams to each bay.

bay1 = [beam1, beam1, beam2]
bay2 = [beam2, beam1, beam2]
bay3 = [beam1, beam1, beam1]

# Assign bays to a given floor.

roof_framing = [bay1, bay2, bay3]

# Set up dictionary with the building floors.

project = {'name': 'Bldg 21', 'roof': roof_framing}

# Output the value of interest - span of a beam in a bay of the
roof. Refer to the diagram below for nesting logic.

print('Span =',project['roof'][1][2]['span'], ' ft')
```

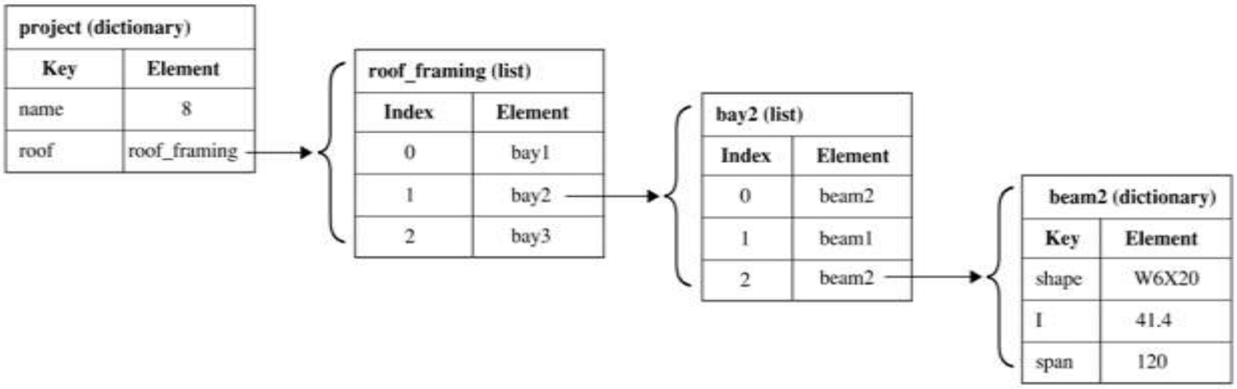
```
>> Span = 120 ft
```

Key: **span**

Second index = [2]

First index = [1]

Key: **floor**



project (dictionary)

Key	Element
name	8
roof	roof_framing

roof_framing (list)

Index	Element
0	bay1
1	bay2
2	bay3

bay2 (list)

Index	Element
0	beam2
1	beam1
2	beam2

beam2 (dictionary)

Key	Element
shape	W6X20
I	41.4
span	120

Notice how datatypes like lists and dictionaries can form complex nested structures. Accessing data within these nested structures can be challenging and it is crucial to understand how each step is evaluated. At the same time, a dictionary like Example 8.3 with its nested structure can be extremely advantageous since it would be possible to expand this data structure to have every building on our university's campus with each floor, every bay in that floor, and all the individual beam elements per bay. Even though this would be an enormous amount of data it would then be relatively easy to complete calculations and visualize information using `for` loops.

9. Built-In Functions

Python supports a few built-in functions. These functions can be used without importing libraries. Some of the common built-in functions are shown in the table below. This section only explores the function `range` in depth. For all of Python's built-in functions and more information go to https://docs.python.org/3/library/functions.html .

Function	Description	Example
abs()	Absolute Value	abs(-10) => 10
type()	Returns the data type	type('hello') => str
len()	Length (returns number of elements in a list)	len([7,8]) => 2
sum()	Sums values in a list	sum([1,5,3]) => 9
max()	Returns maximum value in a list	max([1,5,3]) => 5
min()	Returns minimum value in a list	min([1,5,3]) => 1
append()	Adds element to end of list	L = [4, 2] L.append(1) => [4,2,1]
range()	Returns sequence	(See Section 9.1)

9.1 Range

`Range` produces a sequence of integers with consistent increment and is commonly used in conjunction with a `for` loop. The `range` function can take one, two, or three integer arguments (start value, stop value, step size). Of these only the stop values is required, if no other values are given the start value will be assumed as zero and the increment size as one. The range will include the start value but excludes the stop value (start value ≤ range < stop value).

For a step size of `2`:

```
range(0,8,2) => (0,2,4,6)
```

If the step size is not specified (i.e., only two arguments), the step size is assumed to be `1`.

```
range(0,8) => (0,1,2,3,4,5,6,7)
```

If only one argument is given, the range will use a starting value of `0` and a step size of `1`.

```
range(8) => (0,1,2,3,4,5,6,7)
```

When setting up a `for` loop, the range function is commonly used with the `len()` function because it is an easy way to loop through the indexes of a list. See Section 12 on `for` loops.

```
range(len([8,2,5,6])) => (0,1,2,3)
```

```
L = [4,8,1]
for ii in range(0,len(L)):
    print(L[ii])
```

```
>> 4
>> 8
>> 1
```

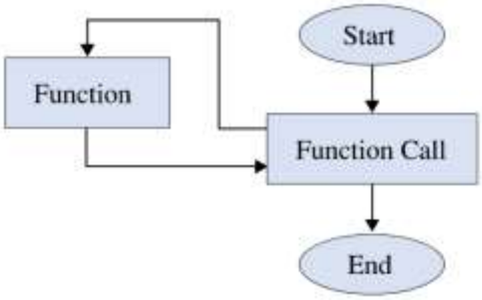
10. Functions

Functions are the heart of programing. They package up multiple lines of code to make Python script more manageable. Functions make code easier to read and less repetitive.

In Python, any data type can be passed into a function. It is crucial to know the data types that are used for the input and the output of a particular function. A function can take any number of inputs, even no inputs. If more than one input is used, the order of the inputs is critical. A brief resource for creating user-defined functions: https://www.programiz.com/python-programming/user-defined-function .

10.1 Function Structure

Functions have two parts: the function definition and the function call. Functions are often defined towards the top of the code or in a separate file. The function must be written or imported before the function call. The function call results in “entry” to the function where provided inputs are used to execute the lines of code and produce specified outputs. When the function is done running, the code continues running from where the function was called.



To define a function, start a new line of code that starts with `def` followed by a *function name*. Functions should have specific clear names that identify the purpose of the function. Function names follow the same rules as variables names (see Section 4.1) and can be overwritten by variables or other functions with the same name. The *function name* is followed by parentheses containing input variables the function will use. This initial line of code is terminated by a colon.

The next line(s) inside the function are indented and are comprised of calculations and other tasks conducted using the input variables. Also indented is the use of `return` at the end of the function definition outputs data from a function and is saved using the variable name assigned on the left-hand side of the equal sign in the function call. Data not returned will not be stored in the Variable Explorer and cannot be accessed once the function call ends. Furthermore, `return` immediately ends a function call.

Note: matching order of inputs and outputs between function definition and call are critical.

Function definition:

```
def functionname (input1, input2):
        code to execute
        return output1, output2
```

Function Call:

```
output1, output2 = functionname (input1, input2)
```

Example 10.1 demonstrates a function that requires no inputs. Any time the function is called using `hi( )` the word ‘hello’ is printed to the command window.

Example 10.1.1

```
def hi():
    print('hello')

hi()
```

```
>> hello
```

The function definition creates the function.

The function call activates the function.

Example 10.2 shows a function with a single numerical input (float or integer) with the variable name `num`. The function call provides a numerical input of 6 for `num` and this is used to compute a value for `calc`, which is then returned and assigned the variable name `result`. After running this code, you will find one variable named `result` with a value of 8 in the Variable Explorer.

Example 10.1.2

```
def add_two(num):
    calc = num + 2
    return calc

result = add_two(6)
print(result)
```

```
>> 8
```

Functions help us to avoid repeating code. Once a function is defined it can be called as many times as is needed. Example 10.3 demonstrates a function being executed multiple times, where it is carrying out the same calculation for different numerical inputs in a list. Specifically, the `for` loop is used to pull a single value from `list_of_nums` and plug it in as `num` in the function call. The function is executed and the output is appended to `list_new_nums`. This is repeated for all values in the `list_of_nums`. After running this code, you will find one variable named `list_new_nums` with five list entries in the Variable Explorer.

Example 10.1.3 Repeating Functions

```
def add_two(num):
    calc = num + 2
    return calc

list_of_nums = [8, 1, 5, 2, 2]
list_new_nums = []
for number in list_of_nums:
    list_new_nums.append((add_two(number)))
print(list_new_nums)
```

```
>> [10, 3, 7, 4, 4]
```

The `append` function adds the number to the list (see Section 7.3 List Operations).

Writing a few comments above the function definition helps to communicate what the function does and how to use it. Often these comments state the purpose of the function as well as inputs outputs. Example 10.4 demonstrates how to lay out these comments and specifically how to indicate to the user what each input/output variable means and its datatype.

When passing multiple inputs to a function, the parameters are recognized by Python using order (not variable name). For example, the first parameter in the function definition corresponds to the first argument in the function call. The second parameter corresponds to the second argument in the function call. The return statement uses order and works the same way.

After running this code, you will find two variables which are both lists named `above` and `below` in the Variable Explorer.

Example 10.1.4 Commenting on Functions

```
# Objectives: Sorts list into 2 lists by above and below value
# Inputs: list (list[int]), value of interest (int)
# Outputs: list of numbers above or equal to value (list[int])
# and list of numbers below value (list[int])

def split_list(L_nums, value):
    L_above = []
    L_below = []
    for num in L_nums:
        if num >= value:
            L_above.append(num)
        else:
            L_below.append(num)
    return L_above, L_below

above, below = split_list([9,0,4,6,1,7], 3)
print(above)
print(below)
```

```
>> [9,4,6,7]
>> [0,1]
```

Notice that a function can take any number of inputs and return any number of outputs.

Inputs can be passed to a function as variables.

10.2 Importing Functions

Storing functions in separate files can make it easier to organize your code. When you want to use a function from a separate file you must import it. There are two ways to import a function.

1. Importing the entire file that contains the function
2. Importing only the function from the file

Let us suppose we have a file named `utility_functions.py` that contains the three functions we have written in Examples 10.1-10.4.

Example 10.2.1 utility_functions.py

```
# utility_functions file

def hi():
    print('hello')

def add_two(num):
    calc = num + 2
    return calc

def split_list(L_nums, value):
    L_above = []
    L_below = []
    for num in L_nums:
        if num >= value:
            L_above.append(num)
        else:
            L_below.append(num)
    return L_above, L_below
```

To be able to import the functions from `utility_functions.py` into the main script these two files need to be stored in the same directory/folder. If the name for the file containing the functions is long, it is possible to rename it to an abbreviation during the import command. In Example 10.5.2 the `utility_functions` is abbreviated to `ufxns`.

Method 1: Importing the entire functions file

Example 10.2.2 main

```
# main file
import utility_functions as ufxns

ufxns.hi()

print(ufxns.add_two(8))
```

```
>> hello
>> 10
```

When the entire file is imported the function call requires the file name. The file name can be shortened when it is imported.

Method 2: Importing individual function(s)

Example 10.2.3 main

```
# main file
from utility_functions import hi

hi()
```

```
>> hello
```

Only the `hi` function has been imported into the main file.

The file name is not required to call the function when it is imported individually.

Example 10.2.4 main

```
# main file
from utility_functions import hi, add_two

hi()

print(add_two(8))
```

Multiple individual functions can be imported at the same time.

```
>> hello
>> 10
```

10.3 Scope

Functions can add layers to programming. A function call results in “entry” to the function and execution of the commands packaged within it and then proceeds with the rest of the main script when the function finishes evaluation. Functions are “nesting” when they call other functions.

Scope refers to the fact that variables can be defined locally (inside a function) or globally (outside functions). Functions have access to locally defined variables and any global variables that were provided as inputs.

Example 10.3.1 Locally and Globally defined Variables

```
def example(x, y, string):
    slope = y/x
    print('Local x =',x)
    print(string, slope)
    return slope

x = 5
word = 'Answer ='
result = example(2, x, word)
print('Global x =',x)
print('Result =',result)
```

The arguments (`2, x, word`) map to the positions of the parameters (`x, y, string`). Therefore, the global variable `x` maps to the local variable `y`, not `x`.

Inside the function, the locally defined `x` takes precedence over the global `x`. Outside the function the locally defined `x` does not exist and globally defined `x` takes precedence.

```
>> Local x = 2
>> Answer = 2.5
>> Global x = 5
>> Result = 2.5
```

It is best practice to avoid confusion like this in your code: where the local and global values of x have the same name but different values. Give variables unique and descriptive names between functions and your main script to provide clarity. Avoid giving variables the exact same name.

11. If, Elif, and Else Statements

Conditional structures (`if` , `elif`, and `else`) have two abilities. First, they check existing variables in the code for one or more conditional statements to produce a `True` or `False` result. Then, that result is used to make decisions that can change the path of the code. This allows for the execution of specified code under regulated conditions. For information on Boolean conditions used to develop conditional statements see Section 5.2. A resource to learn about `if-elif-else` statements: https://www.w3schools.com/python/python_conditions.asp .

11.1 If

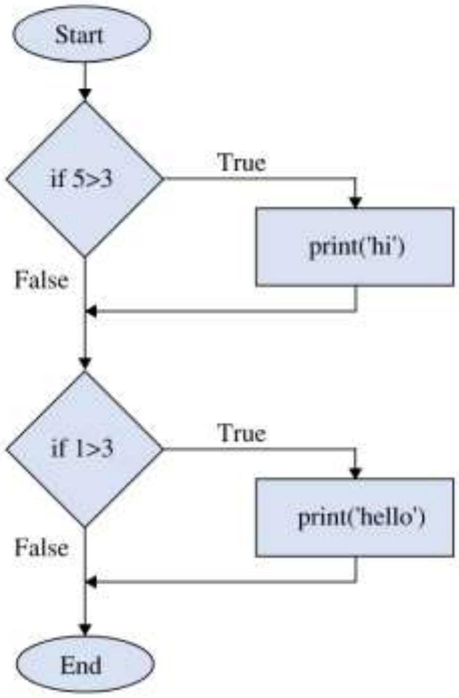
Conditional structures start with an `if` statement. If the expression on the line with `if` evaluates to `True`, the code indented under the `if` statement runs. If the expression is evaluated to `False`, the indented code under the `if` statement does not run and proceeds to check any subsequent `elif` or `else` statements and then the subsequent lines of un-indented code.

In Example 11.1.1 two independent `if` statements are evaluated. Since the first conditional statement evaluates as `True` then the text ‘hi’ is printed; however, the second conditional statement evaluates as `False` so ‘hello’ is never printed. This logic is described in the flowchart below where a diamond represents a decision-making node, and the `True`/`False` branches indicate what actions will be executed based on the result of each conditional `if` statement.

Example 11.1.1

```
if 5 > 3:
    print('hi')
if 1 > 3:
    print('hello')
```

```
>> hi
```



11.2 Elif

`elif` is an abbreviation of "else if" and works very similarly to `if`. `elif` statements must have an expression that evaluates to either `True` or `False`. Unlike `if`, `elif` is only evaluated if the previous `if` and `elif` statements evaluate to `False`.

The order of the `if` and `elif` statements is important and test cases should be developed to ensure that each of these conditional statements are executed in the desired order and produce the anticipated result. To demonstrate this, the order of the conditional statements is swapped in Example 11.2.1 and 11.2.2. The former produces the correct output, while the latter does not.

Example 11.2.1

```
a = 4
if a > 3:
    print('greater than 3')
elif a > 1:
    print('between 3 and 1')
```

```
>> greater than 3
```

Example 11.2.2

```
a = 4
if a > 1:
    print('between 3 and 1')
elif a > 3:
    print('greater than 3')
```

```
>> between 3 and 1
```

The issue in Example 11.2.2 can be resolved by using Boolean Operators like `and` to produce a two-part conditional statement shown in Example 11.2.3.

Example 11.2.3

```
a = 4
if a > 1 and a < 3:
    print('between 3 and 1')
elif a > 3:
    print('greater than 3')
>> between 3 and 1
```

`if 1 < a < 3:`

Another alternative to create a lower and upper bound check without needing a Boolean operator.

11.3 Else

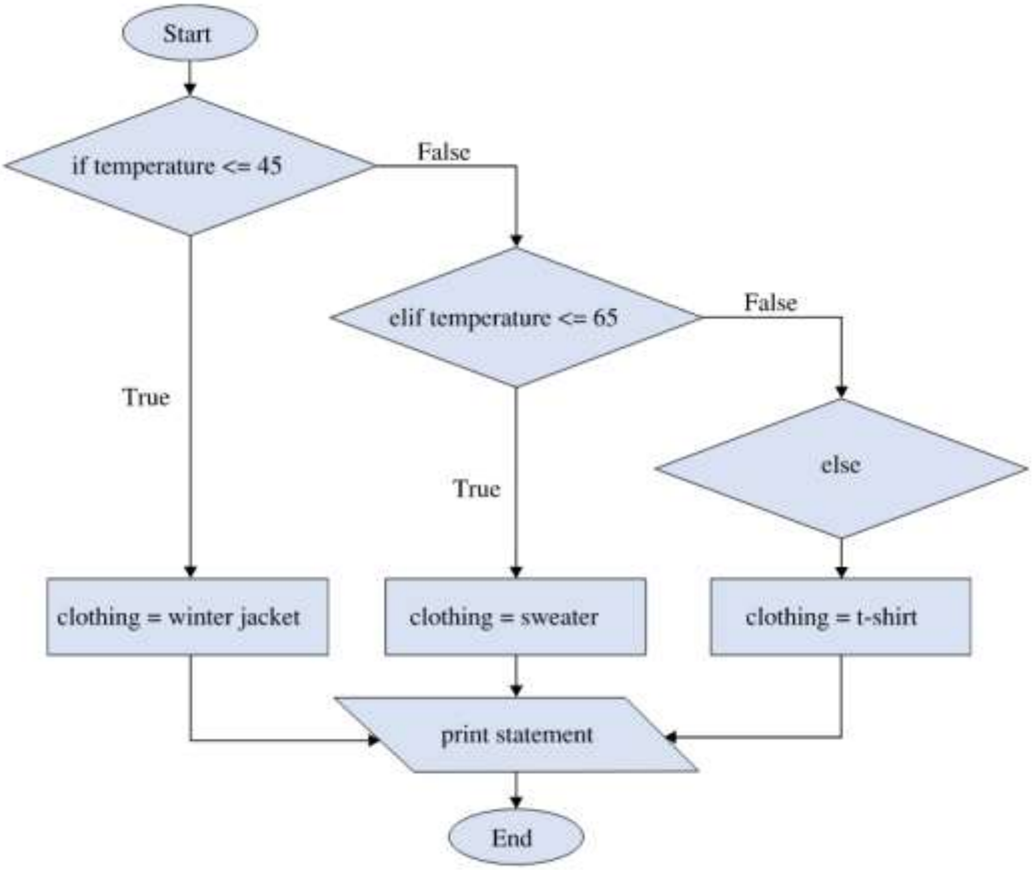
Unlike `if` and `elif`, `else` is never accompanied by an expression. The code indented under the else statement will run every time all the `if` and `elif` statements evaluate to `False`. An `else` statement allows for a “catch all” option, anything that passes the `if` and `elif` statements will run the `else` statement.

Example 11.3.1

```
a = 2
if a > 3:
    print('greater than 3')
else:
    print('less than or equal to 3')
>> less than or equal to 3
```

Example 11.3.2 Determine outfit based on the weather.

```
temperature = 70 #degrees F
if temperature <= 45:
    clothing = 'winter jacket'
elif temperature <= 65:
    clothing = 'sweater'
else:
    clothing = 't-shirt'

print('The temperature is', str(temperature),
      ' F you should wear a', clothing)
>> The temperature is 70 F you should wear a t-shirt
```



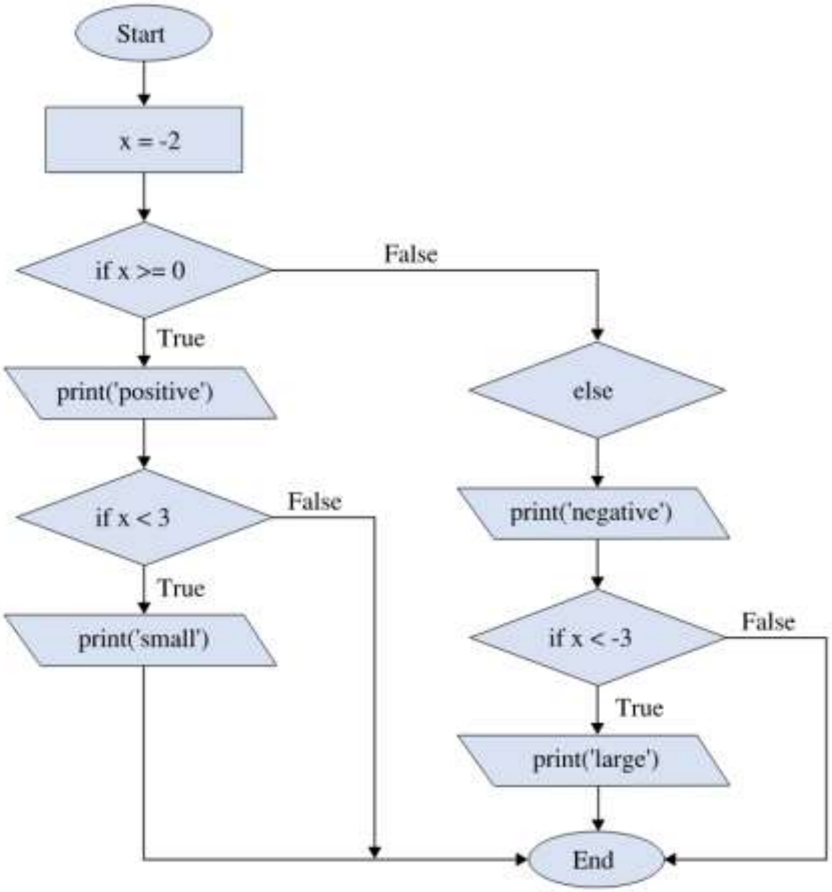
11.4 Nesting

`If` statements can be nested to add additional branching paths. In Example 11.4.1 the first assessment is whether a number is positive or negative, and the second is whether it is greater than |±3|. There are different commands that get executed based on the `True/False` result at each of these decision-making nodes as illustrated in the flowchart below.

Example 11.4.1

```
x = -2
if x >= 0:
    print('positive')
    if x < 3:
        print('small')
else:
    print('negative')
    if x < -3:
        print('large')
```

```
>> negative
```


Start
x = -2
if x >= 0
False
True
print('positive')
else
if x < 3
False
print('negative')
True
print('small')
if x < -3
False
True
print('large')
End

12. For Loops

`For` loops cycle or iterate through code. They are useful for repeated operations because they execute the same code on each iteration and are frequently used to iterate through lists. However, Python also allows iteration through other variable types like 1-D arrays, 2-D matrices, dictionaries with nested information, and other objects. This manual will focus on the use of `for` loops to iterate through lists and arrays. A resource to learn about `for` loops: https://www.w3schools.com/python/python_for_loops.asp .

12.1 For Loop Structure

To define a `for` loop, start a new line of code that starts with the word `for` followed by a *counter* named according to convention described in Section 4.1. This is followed by the word `in` and a list of values (referred to as *range* list) that the counter will use to iterate. The most common approach to setting the *range* list uses the range function described in Section 9.1. This initial line is terminated by a colon, and the next line(s) inside the `for` loop are indented.

```
for counter in range:
    code to execute
```

When the loop starts, the *counter* variable is created and assigned the value of the first entry in the *range* list. Then the indented code below the `for` loop runs and upon completion the second element in the *range* list is assigned to the *counter* variable, overwriting the first. The `for` loop continues iterating until there are no more entries in the *range* list. In this way, `for` loops are bounded loops since they will stop automatically after iterating exactly the number of times as specified in the *range* list. Then the un-indented code below the `for` loop block will run.

Suggestion for naming counters: use variable names that are easy to find with Ctrl + F and reserved to use as `for` or `while` loop counters in your code like double letters (ii, jj) or ‘num’.

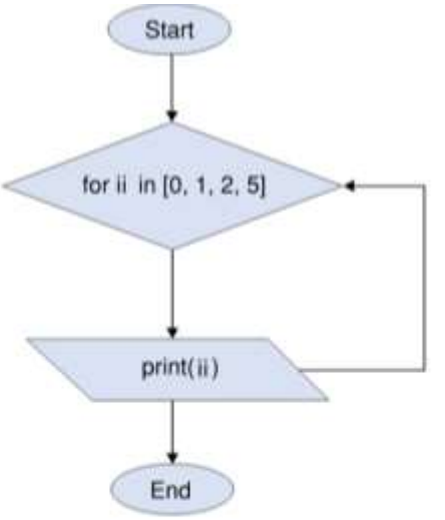
In Example 12.1, the *range* list `[0,1,2,5]` contains four elements, so the `for` loop will iterate four times and execute a print statement on each iteration. This is illustrated in the flowchart below, where the diamond is used to indicate the `for` statement evaluation.

Example 12.1.1

```
for ii in [0,1,2,5]:
    print(ii)
```

```
>> 0
>> 1
>> 2
>> 5
```

After you have run Example 12.1, notice how the *counter* variable `ii` is redefined each time the `for` loop iterates: first `ii = 0`, then `ii = 1`, then `ii = 2`, and finally `ii = 5`.



In Example 12.2, the *range* list `L =[9,-1,2]` contains three elements so there will be three iterations executing two calculations and a print statement. Note that the `loopnum += 1` is shorthand for incrementing the variable by one, equivalent to `loopnum = loopnum + 1`.

Example 12.1.2 Using a `for` Loop to Sum a List

```
L = [9,-1,2]
total = 0
loopnum = 1
for num in L:
    total = total + num
    print('Loop #',loopnum,'total =',total)
    loopnum += 1
print('Final total=',total)
```

An iterable object can be passed as a variable.

After the final loop finishes `total` is printed again

```
>> Loop # 1 total = 9
>> Loop # 2 total = 8
>> Loop # 3 total = 10
>> Final total = 10
```

It is common to use indexing with a `for` loop. One method of creating an iterable object of list indexes is to use the built in functions `len()` and `range()`, see Section 7.1 and 9.

In Example 12.1.3, `len(L)` produces a value of 3 and so `x` = `range`(3) results in the *counter* `x` being assigned the *range* list (0, 1, 2) and values that print on each loop are `L`[0], `L`[1], and `L`[2].

Example 12.1.3 Iterating by index using `range()`

```
L = [0.4,2.2,0.1]
for x in range(0,len(L)):
    print(L[x])
```

```
>> 0.4
>> 2.2
>> 0.1
```

12.2 Nested Loops

Nested loops are very useful since they enable iteration through nested lists (like matrices). The interior `for` loop will run through its entire range each time the exterior loop iterates once.

In Example 12.2.1, the exterior loop only iterates twice, once for each list in L. The interior loop iterates through each element inside those lists (the first list L[0] = [20,75] has two elements and the second list L[1] = [10,11,12] has three). The interior `for` loop can handle lists with different lengths because it measures the length of each list with the `len()` function.

Example 12.2.1 Iterating Through a Nested List

```
L = [[20,75], [10,11,12]]
for aa in range(0,len(L)):
    for bb in range(0,len(L[aa])):
        print('a:',aa, 'b:',bb, 'item:', L[aa][bb])
```

```
>> a: 0 b: 0 item: 20
>> a: 0 b: 1 item: 75
>> a: 1 b: 0 item: 10
>> a: 1 b: 1 item: 11
>> a: 1 b: 2 item: 12
```

When a `for` loop is being used to calculate and store values in a new list, it is necessary to pre-allocate a storage location of a known size for that list.

Example 12.2.2 shows one pre-allocation method `numpy.zeros` where an array or matrix of a given size is prefilled with zeros if it is known that the `for` loop will generate numerical values (float or integer). This is essentially equivalent to `[0]*len ( ).` Another option for mixed datatypes `[None]*len( ).`If the size of an array or matrix is not known, `append( )` from Example 10.1.3 can be used. See Section 15.1 for details on NumPy library functions.

Example 12.2.2 Pre-allocating a List using `numpy.zeros()`

```
import numpy as np
matrix = np.zeros((3, 3))
print('Preallocated matrix = \n', matrix)
for row in range(0,len(matrix)):
    for col in range(0,len(matrix[row])):
        matrix[row][col] = 1.62
print('\n Filled matrix =\n', matrix)
```

```
>> Preallocated matrix =
>>  [[0. 0. 0.]
>>   [0. 0. 0.]
>>   [0. 0. 0.]]
>>
>> Filled matrix =
>>  [[1.62 1.62 1.62]
>>   [1.62 1.62 1.62]
>>   [1.62 1.62 1.62]]
```

12.3 Break

A `break` statement immediately ends and exits the loop. Aside from `for` loops, `break` statements can be useful in terminating a `while` loop if it does not seem to be converging on an answer and continuing to iterate indefinitely. More about `while` loops in Section 13.

In Example 12.3.1 the expectation is that since `L = [0,1,2,3,4,5]` contains 6 entries that will be used for *counter* value `num`, then the for loop should complete 6 iterations. However, once the *counter* value `num` is found to be equal 3 then the `if` statement will evaluate to be `True` triggering the `break` statement and terminating the `for` loop. Then any lines of code that are un-indented after the `for` loop will be executed next.

Example 12.3.1

```
L = [0,1,2,3,4,5]
for num in L:
    if num == 3:
        break
    print(num)
```

```
>> 0
>> 1
>> 2
```

12.4 Continue

A `continue` statement skips the rest of the current iteration of the loop and starts the next iteration of the loop.

The structure of Example 12.4.1 is similar to that of Example 12.3.1, but this time when the *counter* value `num` is found to be equal 3 then the `if` statement will evaluate to be `True` triggering the `continue` statement and skipping to the next iteration such that the only printed value that is missing is 3.

Example 12.4.1

```
L = [0,1,2,3,4,5]
for num in L:
    if num == 3:
        continue
    print(num)
```

```
>> 0
>> 1
>> 2
>> 4
>> 5
```

13. While Loops

`While` loops, like `for` loops, cycle or loop through the same code multiple times. However, there are key differences between `while` and `for` loops. `While` loops are unbounded loops which means they do not have a preset number of iterations, rather a `while` loop is controlled by a condition that evaluates to a Boolean (`True` or `False`). As long as the condition evaluates to `True` the loop continues to run and terminates only when it evaluates to `False`.

`While` loops are especially useful for problems that require iterative solutions governed by a convergence criterion such as in structural dynamics where you are finding the first eigenvalue (natural frequency) and eigenvector (modeshapes) of a multi-degree-of-freedom system or for nonlinear structural analysis methods.

13.1 While Loop Structure

To define a `while` loop, initialize the variable that will be used in the condition. Note that this variable must produce a `True` result the first time the condition is evaluated, or the code contained in the `while` loop will never be executed. Then start a new line of code that starts with the word `while` followed by a condition using Boolean Operators described in Section 5.2. This initial line is terminated by a colon, and the next line(s) inside the `while` loop are indented. Inside the `while` loop there must be at least one statement that updates the value of the variable used in the condition, or the statement will keep evaluating as `True` and the `while` loop will continue executing indefinitely. The `while` loop will iterate until the condition evaluates to False. Then the un-indented code below the `while` loop block will run.

```
condition variable = value
while condition:
    code to execute, must include update to condition variable
```

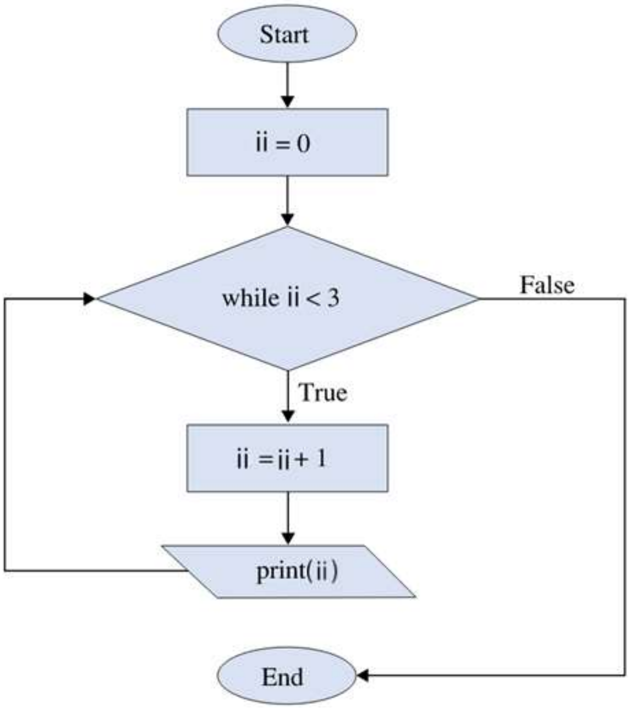
Suggestion for naming condition variable (if also being used as a counter): use variable names that are easy to find with Ctrl + F and reserved to use as `for` or `while` loop counters in your code like double letters (ii, jj) or ‘num’.

In Example 13.1.1, `ii` is the variable used to evaluate the condition and count how many times the `while` loop runs. This value is printed on each iteration of the loop. Since it is incremented by a value of 1 on each loop, in a few iterations `ii` exceeds the value of 3 producing a `False` outcome for the condition which results in termination of the `while` loop. This is illustrated in the flowchart below, where the diamond is used to indicate the `while` condition evaluation.

Example 13.1.1

```
ii = 0
while ii < 3:
    ii = ii + 1
    print(ii)
```

```
>> 1
>> 2
>> 3
```



`While` loops provide more control than `for` loops, and can always be used in place of a `for` loop. Yet there are some cases where only `while` loops will work. In Example 13.1.2, the function `factors_of_two` counts the number of times a value can be divided by two. Note that in this example it is impossible to predetermine the number of iterations in the loop (tracked by the condition variable `divisions`). In the example, you can see that an input value of 50 would require one iteration, where an input of 32 requires five.

Example 13.1.2 Write a function to determine how many times a number is divisible by two.

```
#Function definition using while loop
def factors_of_two(num):
    divisions = 0
    while num % 2 == 0:
        num = num/2
        divisions = divisions + 1
    return divisions

#Function call to run factors_of_two for multiple input values
val = [50, 32]
ans = [None]*(len(val))
for ii in range(0,len(val)):
    ans[ii]=factors_of_two(val[ii])
    print('Value =', val[ii], ', Iterations =', ans[ii])
```

```
>> Value = 50 , Iterations = 1
>> Value = 32 , Iterations = 5
```

In Example 13.1.3, a `while` loop is used to calculate π to a specified accuracy. In this example it would also be impossible to predetermine the number of iterations and thus a `for` loop cannot be used for this application either. In this case, the absolute value of the difference `dif` between the previously `pi_0` and currently `pi_1` computed values is used to check convergence of the solution to within a value of 0.001. This condition sets an acceptable tolerance of convergence for an iterative solution process, which is a very common approach to controlling a `while` loop.

Example 13.1.3 Calculate π for a specified accuracy.

```
n = 0
pi_1 = 0
dif = 1
while abs(dif) > 0.001:
    pi_0 = pi_1
    pi_1 += (-1)**n / (2*n + 1)
    dif = pi_1 - pi_0
    n += 1
print('Final Solution =', pi_1*4)
print('# of Iterations =',n)
```

$$\sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} = \frac{\pi}{4}$$

such that:

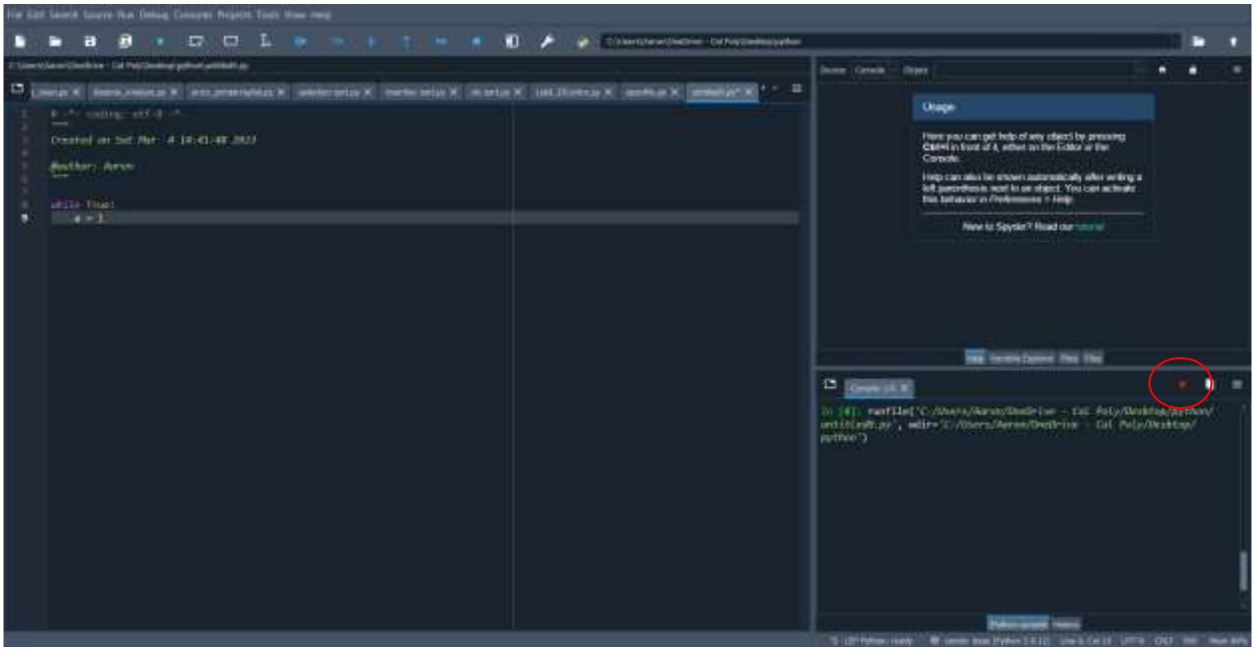
$$1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \cdots = \frac{\pi}{4}$$

```
>> Final Solution = 3.143588659585789
>> # of Iterations = 501
```

If you update the tolerance from 0.001 to 0.01, the final solution is 3.161197... in 51 iterations.

13.2 Manually Ending Program

It is possible to make a coding mistake that causes a `while` loop to iterate endlessly. If this happens the process can be manually stopped by pressing Ctrl + C or by clicking the red square above the command window (this indicator is red when your code is actively running and grayed out when not). If the code is stuck running, it cannot be rerun until manually stopped.



14. Accessing Files

It is often useful to import data from external files. Text files (.txt), data files (.dat), comma-separated value files (.csv), and Excel spreadsheets (.xlsx) are common files for storing data. To access a file, the file must be saved in the same folder as the Python file that is calling the data file, or the file path must be specified. For information specifically on extracting data from an Excel file see Section 15.4. Also, for information on how to access Excel files and read the data from within those files, see Example 20.3.2 in Section 20.3.

The following table includes common functions for reading, writing, and manipulating data from files when imported into Python. Any functions preceded by a period in this list require that the filename precede the function name. See Example 14.2 for context with .read( ).

Function	Description
numpy.loadtext( )	Opens .dat file as a list
open( )	Opens the file for a specified mode
.read( )	Reads text in file by character count
.write( )	Writes text into file
.split( )	Converts string into list of strings based on delimiter
.strip( )	Removes white space from beginning and end of file text
.join( )	Converts list to string with delimiter between elements

Example 14.1 Load .dat file

```
import numpy as np
data = np.loadtxt('ExampleData.dat')
```

Example 14.2 Load and read .txt file

```
file = open('example.txt', 'r')
text1 = file.read(5)
text2 = file.read(3)
print(text1)
print(text2)
```

```
>> abcde
>> fgh
```

Notice that the open function requires two arguments. The first is the file. The second is the mode of using the file. The most common modes are reading (`'r'`) and writing (`'w'`).

Also, notice that the `read` function internally keeps track of where it is in the file. The second `read` function does not start at `a` but at `f` because `abcde` have already been read.

Example 14.3 Read comma delimited data using split

```
raw_data = '1.31,2.06,1.86,0.95'
L_data = raw_data.split(',')
print(L_data)
```

```
>> ['1.31', '2.06', '1.86', '0.95']
```

The csv file is just a string of text with numeric values separated by commas like the `raw_data` in Example 14.3. The `split` function assembles a list using the commas as markers to split up the elements. CSV files use commas as delimiters. If the `split` function is not given an argument, it will create a list using a space as the delimiter. The `join` function does the opposite of the `split` function and concatenates strings from a list with a delimiter.

Example 14.4 Read space delimited data using split and concatenate with join

```
sentence = 'one two red blue'
L_words = sentence.split()
print(L_words)
print(' fish '.join(L_words))
```

```
>> ['one', 'two', 'red', 'blue']
>> one fish two fish red fish blue
```

Motivation Station

Operations like `split` and `join` may seem unnecessary, however they are invaluable when dealing with real files. For example, the data in a CSV file (Comma-Separated Values) is delimited by commas. `File.split(',')` can be used to extract data from a CSV file and `list.join(',')` can be used to create a CSV file.

59% of industry professionals indicate they have scripts in their office that interact with files from structural analysis software such as ETABS, SpColumn, and SAP2000

64% of industry professionals indicate they have scripts in their office that interact with files from drafting software such as Revit, AutoCAD, and Rhino.

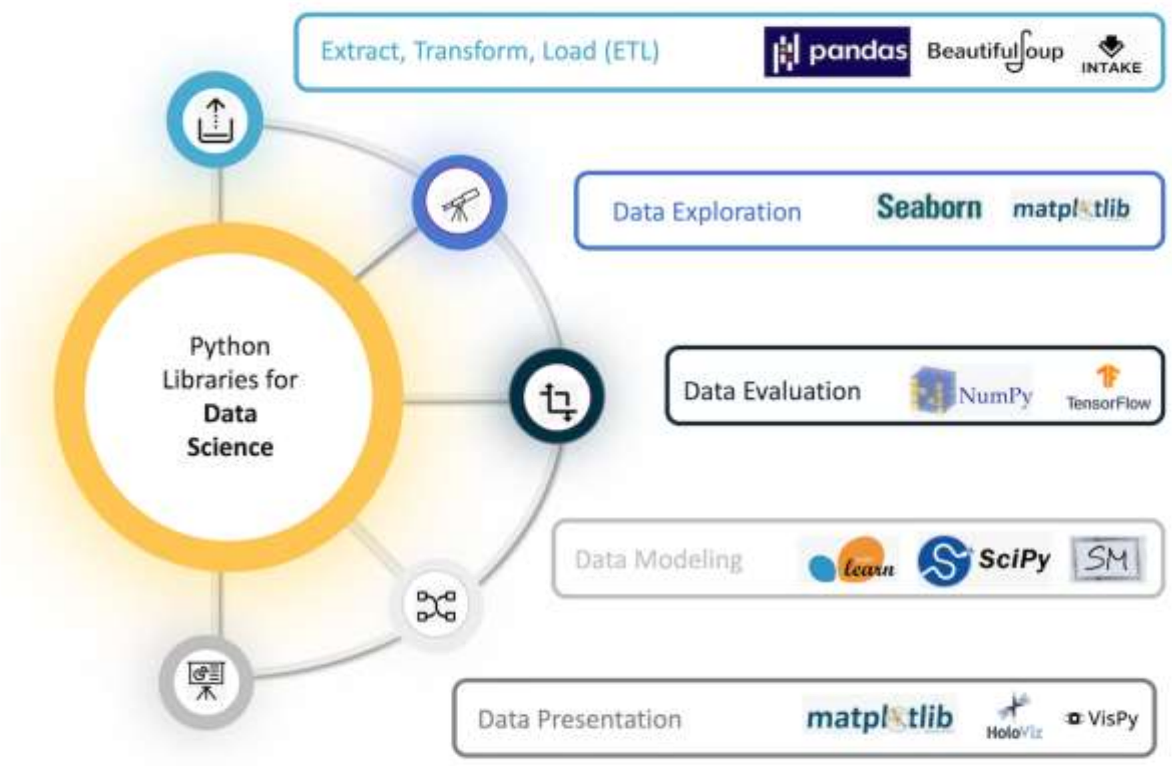
Statistics from Industry Survey: Programming in Structural Engineering

One respondent from the Industry Survey said they developed "python scripts that extract data from ETABS or manipulate ETABS input. Other Python scripts that read data from MS Office software and other software to then perform calculations and produce graphs and reports."

Another respondent said a Cal Poly alumni in their office "developed a program that takes our company-specific AutoCAD plans and builds a RISA model from it."

15. Libraries

A library is a place full of books that people can access and use at any point. In programming this is very similar: libraries store pre-compiled code that have been made available for others to access and use to save time by not having to write code from scratch to execute certain functions. The following chart summarizes a selection of the available open-source Python libraries.



Source: numpy.org

Structural engineering often includes mathematical equations, matrix manipulation, plotting and reading/writing datafiles. As such, we will take a close look at the NumPy, SciPy, Matplotlib, and Pandas libraries in this section and the SymPy library in Section 17. Additional resources on these libraries are provided in Section 24.

To use a library, you must import it at the beginning of your code and can give it an abbreviated name. That abbreviated name can then be used to call the library functions such as: numpy (np), scipy (sp), matplotlib.pyplot (plt), and pandas (pd) shown in examples in Section 15.1-15.4.

15.1 NumPy Library

NumPy is used to create array and matrices and perform various functions on them. While similar in some functionality and naming to the Math library, NumPy is better equipped to use arrays and has more functions available. Listed in the table below are some commonly used functions within the NumPy library. Note that most of the functions would need to be written with the library abbreviation followed by a period and then the function name with the needed inputs inside parenthesis, such as np.array([x_1,x_2,x_3]). Acceptable Data Types are: x = integer, float, or for some cases an array; # = integer only; and arr = an array.

Function	Description	Reference
array([x_1,x_2,x_3])	Creates an array or matrix	16.1
argmin(arr)	Returns the minimum value of arr	
argmax(arr)	Returns the maximum value of arr	
nanargmin(arr)	Returns the minimum of arr, ignoring NaNs	
nanargmax(arr)	Returns the maximum of arr, ignoring NaNs	
insert(arr,#,x)	Inserts ‘x’ into the ‘#’ spot of an array or matrix	
delete(arr,#)	Deletes element(s) in spot ‘#’ of an array or matrix	4.2
append(arr_1,arr_2)	Appends an arr_2 to the end of arr_1	7.3
around(x, #)	Rounds to nearest integer or to number of decimals defined	
zeros(#)	Defines an array or matrix of a given size full of zeros	16.1
ones(#)	Defines an array or matrix of a given size full of ones	16.1
identity(#)	Created an identity matrix of size #	16.1
real(x)	Return the real part of a complex element	16.4
arange($\#_1$,$\#_2$,$\#_3$)	Creates an array from $\#_1$ to $\#_2$ counting by $\#_3$	16.1
size(arr)	Returns the total number of values in arr	16.2
shape(arr)	Returns (number of rows, number of columns) in arr	16.2
where( )	Pulls the indices that are true for a given argument	16.2
average(arr)	Returns the average value in arr	16.2
concatenate(arr_1, arr_2)	Appends arr_2 to the end of arr_1 by rows	16.2
vstack(arr_1, arr_2)	Appends arr_2 to the end of arr_1 by rows	16.2
hstack(arr_1, arr_2)	Appends arr_2 to the end of arr_1 by columns	16.2
reshape(arr, $\#_1$,$\#_2$)	Reshapes arr to have ‘$\#_1$’ rows and ‘$\#_2$’ columns	16.3
transpose( )	Calculate the transpose of a matrix	16.3
exp(x)	Calculate the exponential elementwise	
absolute( )	Calculate the absolute value elementwise	
degrees(x)	Convert angles from x radians to degrees	
radians(x)	Convert angles from x degrees to radians	
log(x)	Natural logarithm elementwise	
divide(x_1,x_2)	Divide arguments elementwise	
randomrand(#)	Creates an array of defined shape full of random values	
randomrandn(#)	Creates an array of defined shape full of random values as per the standard normal distribution	
equal(arr_1,arr_2)	Checks arr_1 == arr_2 elementwise, outputs a boolean array	
not_equal(arr_1,arr_2)	Checks arr_1 != arr_2 elementwise, outputs a boolean array	
less(x_1,x_2)	Checks if $x_1 < x_2$, outputs a boolean array	
less_equal(x_1,x_2)	Checks if x_1 =< x_2, outputs a boolean array	
greater(x_1,x_2)	Checks if $x_1 > x_2$, outputs a boolean array	
greater_equal(x_1,x_2)	Checks if x_1 >= x_2, outputs a boolean array	
square(x)	Returns the element-wise square of the input	
sin(x)	Trigonometric sine, elementwise.	
cos(x)	Trigonometric cosine, elementwise.	
tan(x)	Trigonometric tangent, elementwise.	
sinh(x)	Hyperbolic sine, elementwise.	
cosh(x)	Hyperbolic cosine, elementwise.	

tanh(x)	Hyperbolic tangent, elementwise.
arcsin(x)	Inverse sine, elementwise
arccos(x)	Inverse cosine, elementwise
arctan(x)	Inverse tangent, elementwise
pi	Calls the number π

Example 15.1.1 Creation of array and matrix using NumPy library

```
import numpy as np

arr = np.array([1, 2, 3])
print('Array: \n',arr)

mtrx = np.array([[1, 2, 3],
                 [4, 5, 6]])
print('Matrix: \n', mtrx)
```

The numpy library can be assigned abbreviation 'np'.

Here the array function is being called using 'np'.

```
>>Array:
>>[1 2 3]
>>Matrix:
>>[[1 2 3]
>>[4 5 6]]
```

Several libraries have sub-libraries that contain a subset of functions. An example in `numpy` library is the linear algebra `linalg` sub-library. Functions within sub-libraries are called by the library abbreviation followed by a period, then the sub-library name followed by another period, and finally the function name with the needed inputs inside a set of parentheses. As an example, to compute the inverse of a matrix the code: `inverse = np.linalg.inv(mtrx).`

15.2 Matplotlib Library

Matplotlib is used to create and format graphs. Listed below are some of the more commonly used functions within its sub-library `pyplot`. Note that most of the functions would need to be written with the library abbreviation followed by a period and then the function name with the needed inputs inside parenthesis, such as `plt.plot(x,y).` Acceptable Data Types for functions are: x , y = array of integers/floats with equal length, # = integer only, and others as specified below. More on plotting with the `pyplot` sub-library in Sections 18 and 19.

Function	Description	Reference
autoscale( )	Auto scale the axis view to the data	
axes( )	Add an axis to the current figure	
axline((x_1,y_1),(x_2,y_2))	Add an infinitely long straight line through given points	
axhline(y,x_{min},x_{max})	Add a horizontal line across the axis	
axvline(x,y_{min},y_{max})	Add a vertical line across the axis	
grid( )	Configure the grid lines	18.1
legend( )	Place a legend on the axis	18.2

plot(x,y)	Plot x vs y as lines or markers	18
polar(x,y)	Make a polar plot	
savefig( )	Save the current figure	18.4
scatter(x,y)	Make a scatter plot of x vs y	18.1
subplot( )	Add a single subplot axis to a current figure	18.3
subplot2grid(shape,loc)	Create a subplot at a specific location inside rectangular grid	
subplots($\#_1$,$\#_2$)	Create a figure and set of subplots with x_1 rows and x_2 columns (more efficient than subplot)	18.3
table( )	Add a table to the axes	
text(x,y,text)	Add text to the axes	18.1
xlabel(string)	Set the label for the x-axis	18.1
xlim(x_1,x_2)	Set the limits for the x-axis	18.1
xscale(#)	Set x-axis scale	
xticks(x,string)	Set tick locations and labels on x-axis	
ylabel(string)	Set the label for the y-axis	18.1
ylim(min,max)	Set the limits for the y-axis	18.1
yscale(#)	Set y-axis scale	
yticks(y,string)	Set tick locations and labels on y-axis	

Example 15.2.1 Creation of plot with Matplotlib.pyplot sub-library

```
import matplotlib.pyplot as plt
import numpy as np

x = np.array([1,2,3,5])
y = x**2

plt.plot(x,y)
```

Here we are importing the matplotlib.pyplot sub-library as ‘plt’. We could name a library anything we want, but the names used here are standard to what you will find in online documentation.

15.3 SciPy Library

SciPy (Scientific Python) is a scientific computation library built off NumPy with functions that simplify matrix calculations. Below are commonly used functions within the `scipy` sub-library `linalg`. All these functions are written with the library abbreviation followed by a period, the sub-library name followed by another period, and the function name with the needed inputs inside parenthesis, such as `sp.linalg.inv(mtrx)`. A matrix is the acceptable Data Type.

Function	Description	Reference
inv(mtrx)	Computes inverse of matrix	16.3
det(mtrx)	Computes determinant of matrix	16.3
norm(mtrx)	Computes matrix norm	
issymmetric(mtrx)	Checks if matrix is symmetrical	
eig(mtrx)	Solves an eigenvalue problem	16.4
eigvals(mtrx)	Computes eigenvalues	
eigh(mtrx)	Like eig(), but ensures eigenvalues are sorted	16.4

cosm(mtrx)	Compute the matrix cosine
sinm(mtrx)	Compute the matrix sine

Example 15.3.1

```
import numpy as np
import scipy as sp
from scipy import linalg

mtrx = np.array([[1,2,3],
                 [4,10,6],
                 [7,8,9]])
inverse = sp.linalg.inv(mtrx)
```

Here we are importing the scipy library as ‘sp’ and then from this library explicitly importing the ‘linalg’ sub-library.

15.4 Pandas Library

The Pandas library is also built off the NumPy library and is used for organization of data and creating tables. It is often used in conjunction with Excel to read, write, and format cells. It holds two data structures: series (one-dimensional) and data frames (two-dimensional). Below are some common functions within the library.

Function	Description	Reference
read_csv( )	Reads a csv file	
read_excel( )	Reads an Excel spreadsheet	20.3
rename( )	Renames a column or row	
drop( )	Delete columns or rows	
dropna( )	Delete missing columns or rows	
drop_duplicates( )	Drops duplicate values in row or column	
groupby( )	Groups data based on an applied function	
merge( )	Merges columns to share rows	
sort_values( )	Sorts row or column by given parameter (ie. ascending)	
fillna( )	Fills missing spots with given value	

Another way of accessing and editing Excel spreadsheets is through `xlxswriter`**.** (Note that xlxswriter may not automatically be included on Spyder and may require a pip install). Unlike Pandas, `xlxswriter` is not a library but rather a *module*. Modules in Python are like standalone files, typically with only one specific purpose. Using the analogy of a library, a module would be like a single book. They are imported and used the same way we use libraries.

Example 15.4.1

```
import pandas as pd
import xlsxwriter




# We will go into more depth on Excel in Section 20.3
```

Here we are importing the pandas library and naming it ‘pd’.

Here we are importing xlxswriter. By not redefining the name, it will have to be called by its full name.

16. Arrays and Matrices

Arrays and matrices are `numpy` objects and are some of the most commonly used data types for analysis. They differ from a list in that they contain homogeneous elements, meaning you cannot have different data types. This allows for easy storage of numerical values and makes it possible to solve more complex problems such as those involving modal analysis for structural dynamics applications. We will go over how to initialize different types of arrays and matrices, and various methods for manipulating them.

16.1 Initializing an Array or Matrix

Aside from hard coding values for an array or matrix, there are many helpful functions to quickly define matrices filled with zeros, ones, or an identity matrix which we will cover in this section.

Three approaches to initialize an array are covered in Example 16.1.1, since the goal with coding is to avoid hardcoding whenever possible. Note that `np.linspace` is inclusive of the stop value but `np.arange` is not. If a certain increment size is desired for the array, it is often easier to control by using the option `np.arange`.

Hardcoding:

var = np.array([array values separated by commas])

Using np.linspace: start value $\leq$ var $\leq$ stop value

var = np. linspace(start value, stop value, # of values in array)

Using np.arange: start value $\leq$ var $<$ stop value

var = np. arange(start value, stop value, increment size)

Example 16.1.1 Create an array from 0 to 30 counting by 3 first by hardcoding, then using `linspace` and `arange`.

```
import numpy as np

a = np.array([0,3,6,9,12,15,18,21,24,27,30])
b = np.arange(0,33,3)
c = np.linspace(0,30,11)
d = np.linspace(0,30,11, dtype = int)

print('a=',a,'\nb=', b ,'\nc=', c,'\nd=', d)
```

`linspace` will create an array of `float` objects unless you redefine the data type using `dtype`.

```
>>a= [ 0  3  6  9 12 15 18 21 24 27 30]
>>b= [ 0  3  6  9 12 15 18 21 24 27 30]
>>c= [ 0.  3.  6.  9. 12. 15. 18. 21. 24. 27. 30.]
>>d= [ 0  3  6  9 12 15 18 21 24 27 30]
```

Approaches for initializing matrices are presented in Example 16.1.2. Some like `np.zeros` are useful for pre-allocating storage space to populate during nested for loops (see Section 12.1).

Hardcoding:

mtrx = np.array([[values for 1st row separated by commas],
[values for 2nd row separated by commas]])

Matrix of Zeros, Ones, Random Numbers:

mtrx = np. zeros([# of rows, # of columns])
mtrx = np. ones([# of rows, # of columns])
mtrx = np. random.rand(# of rows, # of columns)

Identity Matrix: assumes square shape (# rows = # columns)

mtrx = np. identity(size)

Example 16.1.2 Create the following 3x3 matrices: hardcode values from 1-9, full of zeros, full of ones, full of random numbers, and the identity matrix.

```
import numpy as np

a = np.array([[1, 2, 3],
              [4, 5 ,6],
              [7, 8, 9]])
b = np.zeros([3,3])
c = np.ones([3,3])
d = np.random.rand(3,3)
e = np.identity(3)

print('a=', a,'\n\nb=', b,'\n\nc=', c,'\n\nd=', d,'\n\ne=', e)
```

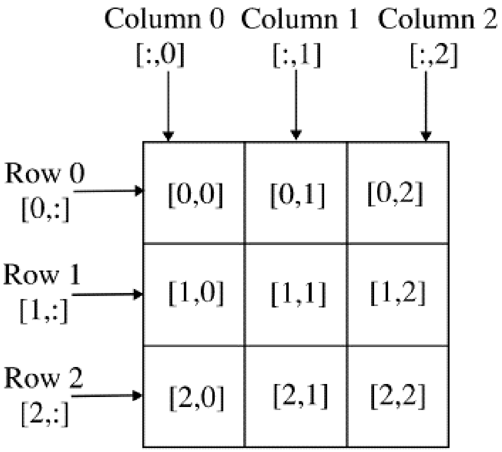
Filling an array with zeros is preferred to filling with 'NaN' or "Not a number", as this can lead to issues later since NaN is not recognized the same way as a value.

```
>>a= [[1 2 3]
>> [4 5 6]
>> [7 8 9]]
>>
>>b= [[0. 0. 0.]
>>[0. 0. 0.]
>>[0. 0. 0.]]
>>
>>c= [[1. 1. 1.]
>>[1. 1. 1.]
>>[1. 1. 1.]]
>>
>>d= [[0.15422338 0.46428684 0.24565437]
>>[0.07428929 0.52856389 0.7034907 ]
>>[0.31062592 0.22110742 0.18784994]]
>>
>>e= [[1. 0. 0.]
>>[0. 1. 0.]
>>[0. 0. 1.]
```

This will produce a different result each time it is run. `np.random.rand` produces values from 0 to 1. These can be scaled up or you can use `np.random.randint(a, size = (3,3))` which will produce values from 0 to a.

16.2 Indexing and Determining the Length of an Array or Matrix

Indexing is used to extract parts of a matrix in structural analysis, such as in static condensation or partitioning a matrix. Evaluating the length or size of a matrix is also a very common practice as this value is used to create a for loop with the appropriate number of iterations. Below is a visual of how Python assigns indices to a matrix where the convention is [row #, column #].



Indexing begins at zero for the rows and columns.

A colon can be used to call a range:

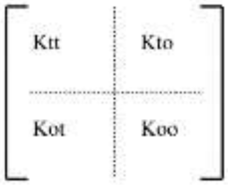
[0, :] means Row 0 and *all* Columns

[:, 1] means *all* Rows and Column 1

[0:2, 1:3] means Rows 0-1 and Columns 1-2 since $0 \leq$ rows < 2 and $0 \leq$ columns <3 (inclusive of start value but not stop value)

Also helpful is the index -1, which calls the last index of an array or matrix which is useful when you do not know the final index number.

Example 16.2.1 Given the following 4x4 stiffness matrix, print a) the first and last row, b) the total number of values in the matrix, c) the number of rows and columns using `len` and `shape`, and d) partition as shown where K_{tt}, K_{to}, K_{ot} and K_{oo} are 2x2.



```
K = np.array([[ 14.094, -1.5660, -234.90,  78.300],
              [-1.5660,  1.5660, -78.300, -78.300],
              [-234.90, -78.300,  15660,   2610.0],
              [ 78.300, -78.300,  2610.0,  5220.0]])
```

```
import numpy as np

K = np.array([[ 14.094, -1.5660, -234.90,  78.300],
              [-1.5660,  1.5660, -78.300, -78.300],
              [-234.90, -78.300,  15660,   2610.0],
              [ 78.300, -78.300,  2610.0,  5220.0]])
#Part a
first = K[0,:]
last = K[-1,:]
print('\nPart a:')
print('First Row =', first, '\nLast Row =', last)

#Part b
num = np.size(K)
print('\nPart b:')
print('\nNumber of Values = ', num)
```

```
#Part c
rows1, cols1 = np.shape(K)
print('\nPart c:')
print('Shape Fxn: # Rows = ', rows1, ', # Cols = ', cols1)

rows2 = len(K)
cols2 = len(K[0,:])
print('Length Fxn: # Rows = ', rows2, ', # Cols = ', cols2)

#Part d
Ktt = K[0:2,0:2]
Kto = K[0:2,2:4]
Kot = K[2:4,0:2]
Koo = K[2:4,2:4]
print('\nPart d:')
print('Ktt = ', Ktt, '\n\nKto = ', Kto, '\n\nKot = ', Kot,
'\n\nKoo = ', Koo)
```

Note that in all instances, the row(s) are referenced first, followed by the column(s) separated by a comma. `0:2` calls rows/columns 0 and 1.

Avoid hardcoding, if you know the indices store them in variable(s) and call them, e.j. `Koo=K[a:b,c:d]`.

```
>>Part a:
>>First Row = [  14.094   -1.566 -234.9     78.3  ]
>>Last Row = [  78.3  -78.3 2610.  5220. ]
>>
>>Part b:
>>Number of Values =  16
>>
>>Part c:
>>Shape Fxn: # Rows = 4 , # Cols = 4
>>Length Fxn: # Rows = 4 , # Cols = 4
>>
>>Part d:
>>Ktt =  [[14.094 -1.566]
>> [-1.566  1.566]]
>>
>>Kto =  [[-234.9   78.3]
>> [ -78.3  -78.3]]
>>
>>Kot =  [[-234.9  -78.3]
>> [  78.3  -78.3]]
>>
>>Koo =  [[15660.  2610.]
>> [ 2610.  5220.]]
```

If you did not know the indices to partition the matrix you could use the `where` function, which will evaluate the indices of a matrix that fit a certain condition. Another helpful function is `concatenate`, which can combine multiple arrays or matrices. Example 16.2.2 will demonstrate both of these.

Example 16.2.2 For a 4x4 matrix with random numbers between 0-100: (a) use the `where` function to find indices containing a value higher than the average value for the matrix. Print indices and values, (b) create a second 4x4 matrix and concatenate with the first matrix, first as additional rows then as additional columns. (c) delete the second column of the second matrix.

```
import numpy as np

# part a
matrix1 = np.random.randint(100,size = (4,4))
print('\nPart a:')
print('Matrix 1: \n', matrix1)

avg = np.average(matrix1)
print('\nAverage = \n', avg)

indices = np.where(matrix1 > avg)
print('\nIndices: ', indices)
print('\nValues: ', matrix1[indices])

# part b
matrix2 = np.random.randint(100,size = (4,4))
print('\nPart b:')
print('Matrix 2: \n', matrix2)

conc1 = np.concatenate((matrix1, matrix2))
conc2 = np.concatenate((matrix1, matrix2), axis=1)
print('\nConcatenate as rows: \n', conc1, '\n\nConcatenate as columns: \n', conc2)

# part c

matrix3 = np.delete(matrix2,1,1)
print('\nPart c:')
print('Deleting Second Column: \n', matrix3)
```

The `where` function only pulls the indices, to display the values you must call those values within your matrix.

`concatenate` defaults to axis 0 (adding after the rows). Setting `axis = 1` will add the values after the columns.

`np.delete` inputs are (array/matrix name, row/column number, axis number) Where axis number= 0 for rows or 1 for columns.

An alternate way to achieve the concatenation to add rows or columns, indicated above in red, is by using the `.vstack()` and `.hstack()` functions:

```
conc1 = np.vstack((matrix1, matrix2)) #adds rows
conc2 = np.hstack((matrix1, matrix2)) #adds columns
```

```
>>Part a:
>>Matrix 1:
>> [[28 14 33 26]
>> [85 31 78 81]
>> [69  5 67 97]
>> [96 96 24 89]]
>>
>>Average =
>> 57.4375
>>
>>Indices:  (array([1, 1, 1, 2, 2, 2, 3, 3, 3], dtype=int64),
>>array([0, 2, 3, 0, 2, 3, 0, 1, 3], dtype=int64))
>>
>>Values:  [85 78 81 69 67 97 96 96 89]
>>
>>Part b:
>>Matrix 2:
>> [[84  4 42 61]
>> [38 32  3 71]
>> [90  4 96 28]
>> [28 36 17 26]]
>>
>>Concatenate as rows:
>> [[28 14 33 26]
>> [85 31 78 81]
>> [69  5 67 97]
>> [96 96 24 89]
>> [84  4 42 61]
>> [38 32  3 71]
>> [90  4 96 28]
>> [28 36 17 26]]
>>
>>Concatenate as columns:
>> [[28 14 33 26 84  4 42 61]
>> [85 31 78 81 38 32  3 71]
>> [69  5 67 97 90  4 96 28]
>> [96 96 24 89 28 36 17 26]]
>>
>>Part c:
>>Deleting Second Column:
>> [[84 42 61]
>> [38  3 71]
>> [90 96 28]
>> [28 17 26]]
```

The indices are presented in two arrays since we have a 2-dimensional matrix. The first array contains the row number, the second contains the respective column number.

16.3 Performing Basic Matrix Operations

16.3.1 Adding and Subtracting Matrices

Adding or subtracting a scalar to a matrix will add or subtract each matrix element by that value and return a matrix of the same size. Adding or subtracting a matrix from another matrix of the same size will return a matrix of the same size with the sum or difference of the corresponding matrix values. Finally, adding, or subtracting arrays of the same row or column length as a matrix will add or subtract the array values in order across the rows (or columns if manipulated as shown below) of the matrix.

Example 16.3.1

```
import numpy as np
a = np.array([[1, 2, 3],
              [4, 5 ,6],
              [7, 8, 9]])
b = a+1
c = a-1
d = a+a
e = a-a

arr = np.array([1,2,3])
f = a + arr
g = (a.T + arr).T

print("a+1 = \n", b)
print("\na-1 = \n", c)
print("\na+a = \n", d)
print("\na-a = \n", e)
print("\na+arr = \n", f)
print("\na+arr column = \n", g)
```

Altneratively, `arr` could be initialized as a vertical array and you would not have to use the transpose function. Transpose of a matrix is covered in more depth in Section 16.3.3.

```
>>a+1 =
>> [[ 2  3  4]
>> [ 5  6  7]
>> [ 8  9 10]]
>>
>>a-1 =
>> [[0 1 2]
>> [3 4 5]
>> [6 7 8]]
>>
>>a+a =
>> [[ 2  4  6]
>> [ 8 10 12]
>> [14 16 18]]
```

```
>>a-a =
>> [[0 0 0]
>> [0 0 0]
>> [0 0 0]]
>>
>>a+arr =
>> [[ 2  4  6]
>> [ 5  7  9]
>> [ 8 10 12]]
>>
>>a+arr column =
>> [[ 2  3  4]
>> [ 6  7  8]
>> [10 11 12]]
```

16.3.2 Multiplying Matrices

Multiplying or dividing a matrix by a scalar will simply multiply or divide each matrix element by that value and return a matrix of the same size as shown in Example 16.3.2.

Example 16.3.2

```
import numpy as np
a = np.array([[1, 2, 3],
              [4, 5 ,6],
              [7, 8, 9]])

b = a*2
c = a/2

print('a*2= \n',b)
print('\n a/2= \n',c)
```

```
>>a*2 =
>> [[ 2  4  6]
>> [ 8 10 12]
>> [14 16 18]]
>>
>>a/2 =
>> [[0.5 1.  1.5]
>> [2.  2.5 3. ]
>> [3.5 4.  4.5]]
```

Example 16.3.3 covers three methods for multiplying two matrices: `*`, `@`, or `np.matmul()`. Using `@` and `np.matmul()` and will produce the same result, which is the dot (matrix) product between two matrices. Using the `@` symbol is preferred as it makes for a more concise code. Using * will simply return the product between the corresponding values in each matrix.

Example 16.3.3 Create identical 3x3 matrices ‘a’ and ‘b’, and 3x1 array ‘x’. Print the result of multiplying ‘a’ and ‘b’, then ‘a’ and ‘x’ using the three methods mentioned above.

```
import numpy as np
a = np.array([[1, 2, 3],
              [4, 5 ,6],
              [7, 8, 9]])

b = np.array([[1, 2, 3],
              [4, 5 ,6],
              [7, 8, 9]])
c = a*b
d = a@b
e = np.matmul(a,b)
```

```
print('\na*b =\n',c,'\n\na@b =\n',d,'\n\nnp.matmul(a,b) =\n',e)

x = np.array([1,2,3])

c = a*x
d = a@x
e = np.matmul(a,x)
print('\na*x =\n',c,'\n\na@x =\n',d,'\n\nnp.matmul(a,x) =\n',e)
```

```
 >>a*b =
 >>[[ 1  4  9]
 >> [16 25 36]
 >> [49 64 81]]
 >>
 >>a@b =
 >> [[ 30  36  42]
 >> [ 66  81  96]
 >> [102 126 150]]
 >>
 >>np.matmul(a,b) =
 >> [[ 30  36  42]
 >> [ 66  81  96]
 >> [102 126 150]]
 >>a*x =
 >> [[ 1  4  9]
 >> [ 4 10 18]
 >> [ 7 16 27]]
 >>
 >>a@x =
 >> [14 32 50]
 >>
 >>np.matmul(a,x) =
 >> [14 32 50]
```

16.3.3 Transpose of a Matrix

The transpose of a matrix is obtained by changing the rows to columns and column to rows. The main methods for achieving this are by using `.T`, `np.transpose()`, and `.reshape()`. Using `.T` and `np.transpose` works on 2-D matrices; however, they have no effect on 1-D arrays. That is where it becomes helpful to use `.reshape( )`, which reassigns the number of rows and columns.

Example 16.3.4 Create a 3x4 matrix ‘a’ and print the result of using .T and np.transpose(). Then use .reshape() to convert it into a matrix of 2 rows and 6 columns. Create a 3x1 array ‘x’ and print the result of using .T and np.transpose(), then use reshape to convert it into a 1x3 array.

```
import numpy as np

a = np.array([[1,  2,  3],
             [4,  5 ,6],
             [7,  8,  9],
             [10,11,12]])

x = np.array([1,2,3])

f = a.T
g = np.transpose(a)
h = a.reshape(2,-1)
print('\na.T =\n',f,'\n\nnp.transpose(a) =\n',g,'\n\na.reshape()
=\n',h)

f = x.T
g = np.transpose(x)
h = x.reshape(-1,1)
print('\nx.T =\n',f,'\n\nnp.transpose(x) =\n',g,'\n\nx.reshape()
=\n',h)
```

The input (2,-1) will reshape ‘a’ into a matrix with 2 rows and the -1 will result in calculating the necessary number of columns (12 entries in ‘a’ divided by 2 rows results in 6 columns).

The input (-1,1) will reshape ‘x’ into an array with -1 the calculated necessary number of rows and 1 column (3 entries in ‘x’ divided by 1 column results in 3 rows).

```
>>a.T =
>> [[ 1  4  7 10]
>> [ 2  5  8 11]
>> [ 3  6  9 12]]
>>
>>np.transpose(a) =
>> [[ 1  4  7 10]
>> [ 2  5  8 11]
>> [ 3  6  9 12]]
>>a.reshape() =
>> [[ 1  2  3  4  5  6]
>> [ 7  8  9 10 11 12]]
>>
>>x.T =
>> [1 2 3]
>>
>>np.transpose(x) =
>> [1 2 3]
>>
>>x.reshape() =
>> [[1]
>> [2]
>> [3]]
```

Note that .T and np.transpose() have no effect on a 1-D array.

16.3.4 Inverse and Determinant

The functions for the inverse and determinant of a matrix are `np.linalg.inv()` and `np.linalg.det()` respectively.

Example 16.3.5 Determine equivalent stiffness matrix ($K_{equiv} = K_{tt} – K_{to}K_{oo}^{-1}K_{to}^{T}$) of the K matrix from Example 16.2.1, then take the determinant of K_{equiv}.

```
import numpy as np

K = np.array([[ 14.094, -1.5660, -234.90,  78.300],
              [-1.5660,  1.5660, -78.300, -78.300],
              [-234.90, -78.300,  15660,   2610.0],
              [ 78.300, -78.300,  2610.0,  5220.0]])

Ktt = K[0:2,0:2]
Kto = K[0:2,2:4]
Kot = K[2:4,0:2]
Koo = K[2:4,2:4]

K_equiv = Ktt - Kto@np.linalg.inv(Koo)@Kto.T
print('K equivalent = \n', K_equiv)

print('\nDeterminant = ', round(np.linalg.det(K_equiv),4))
```

```
>> K equivalent =
>>  [[ 7.68763636 -1.13890909]
>>  [-1.13890909  0.28472727]]
>>
>> Determinant = 0.8918
```

16.4 Solving Eigenvalue Problems

The Python SciPy library contains two functions to help solve eigenvalue problems: `eig` and `eigh`. Both use two matrices as inputs and output their eigenvalues and eigenvectors. However, `eigh` automatically sorts values in ascending order while `eig` does not. This is useful in solving natural frequencies and mode shapes for structural dynamics problems where order is relevant.

Example 16.4.1 Assume the matrices represent mass and stiffness for a structure. Characterize its dynamic properties by solving for the eigenvalues (natural frequencies) and eigenvectors (modeshapes) using `eig` and `eigh`. Print the values for each and implement code that will sort the results from `eig`. Then compare to the output from `eigh` that automatically sorts results.

```
M = np.array([[.300,0,0],[0,.300,0],[0,0,1200]])
K = np.array([[100,0,6000],[0,200,0],[6000,0,3000000]])
```

```
import numpy as np
import scipy as sp
from scipy import linalg

M = np.array([[.300,0,0],[0,.300,0],[0,0,1200]])
K = np.array([[100,0,6000],[0,200,0],[6000,0,3000000]])
```

`.real` pulls just the real part of a complex value, otherwise there may be inaccurate results or a math domain type error.

```
# eig
[eigval, eigvec] = sp.linalg.eig(K,M)
print('eig result (unsorted): \n\nEigenvalues: ', eigval.real,
'\n\nEigenvectors: \n', eigvec)
```

`np.argsort` sorts values of an array from smallest to largest and outputs their indices.

```
# Sorting eig
idx = np.argsort(eigval)
eigval = eigval[idx]
eigvec = eigvec[:, idx]
print('\n\n\neig result (sorted): \n\nEigenvalues: ',
eigval.real, '\n\nEigenvectors: \n', eigvec)


# eigh
[eigval2, eigvec2] = sp.linalg.eigh(K,M)
print('\n\n\neigh result: \n\nEigenvalues: ', eigval2,
'\n\nEigenvectors: \n', eigvec2)
```

```
>> eig result (unsorted):
>>
>> Eigenvalues:  [ 288.12286552 2545.21046782  666.66666667]
>>
>> Eigenvectors:
>>  [[-0.99999745 -0.99394003  0.        ]
>>  [-0.         -0.          1.        ]
>>  [ 0.00226052 -0.10992366  0.        ]]
>>
>>
>> eig result (sorted):
>>
>> Eigenvalues:  [ 288.12286552  666.66666667 2545.21046782]
>>
>> Eigenvectors:
>>  [[-0.99999745  0.         -0.99394003]
>>  [-0.          1.         -0.        ]
>>  [ 0.00226052  0.         -0.10992366]]
```

```
>> eigh result:
>>
>> Eigenvalues:  [ 288.12286552  666.66666667 2545.21046782]
>>
>> Eigenvectors:
>>  [[-1.80736415  0.         -0.25839533]
>>  [ 0.         -1.82574186  0.         ]
>>  [ 0.00408559  0.         -0.02857694]]
```

Note: while the eigenvector results seem different between `eig` and `eigh`, even when sorted, it should be noted that these are interpreted as relative displaced shapes (mode shapes) when characterizing a structure’s dynamic response. If the eigenvectors from `eigh` are normalized by taking the values of each column of the matrix and dividing by the value in that column’s first row, the resulting matrix would be identical to that produced by `eig` after sorting.

17. SymPy Library

The SymPy library can be thought of as a cross between *sym*bols and nu*mpy*. SymPy allows us to solve the same types of problems as in NumPy but using symbolic variables rather than numbers. This is helpful when we are more interested in the generic solution rather than the specific, numerical one. Once you have a symbolic equation, SymPy can also be used to substitute in values by using the function `subs`. It is also possible to differentiate or integrate symbolic expressions.

In Example 17.1(a), we develop the symbolic expression for the equation for a line: y = mx + b. To create this symbolic expression, the `symbols` function is used to create the desired variables. For the line of code with the `symbols` function, the order of names on the left-hand side of the equals sign must match the inputs in the parentheses, that name itself does not have to match in terms of spelling or capitalization but is recommended. Following these guidelines ensures that symbolic expressions and substitution are executed correctly in the rest of the code.

Example 17.1(a) Create symbolic variables and use for a symbolic math expression

```
import sympy as sy

m, x, b = sy.symbols('m x b')
y = m*x + b
print('y =',y)
```

```
>>y = b + m*x
```

In Example 17.1(b), we can then use the `subs` function to insert a numerical value (or another SymPy symbol) into a symbolic math expression for a given symbolic variable. This does not alter the original symbolic expression, so it is necessary to save the result of the substitution under a new variable name. The first argument of `subs` function is the symbolic variable name and the second argument is the numerical value to plug in for that symbolic variable. Two examples are shown to demonstrate the difference in the syntax for substituting a single variable versus multiple variables.

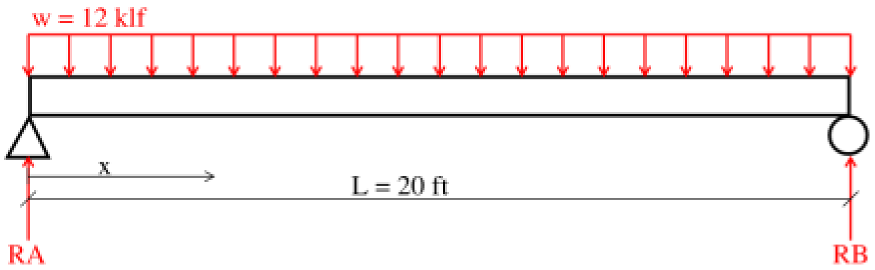
Example 17.1(b) Substitute in values for symbols in a symbolic expression

```
y2 = y.subs(m,5)
y3 = y.subs([(m, 5), (b, 10)])
print('y2 =',y2)
print('y3 =',y3)
```

Note the differences in how input is specified for single versus multiple variable substitution.

```
>>y2 = b + 5*x
>>y3 = 5*x + 10
```

Example 17.2(a) Create a script that solves for the shear and moment equations (using integration) of the following simply supported beam in terms of x, w and L. Print the symbolic equations then substitute the numerical values and solve for when x = 10 ft, and x = 15 ft.



```
import numpy as np
import sympy as sy
import matplotlib.pyplot as plt

x, w, L = sy.symbols('x w L')
# Reactions
RA = (-w*L)/2
RB = RA

# Shear
V0 = RA
V = sy.integrate(w,x) + V0
print('V(x) =', V)

# Bending Moment
M0 = 0
M = sy.integrate(V, x) + M0
print('M(x) =', M)

Vcheck = sy.diff(M, x)
print('Check V(x) =', Vcheck)

# Beam 1 solution
w1 = -12
L1 = 20

V1 = V.subs([(x,10),(w,w1),(L,L1)])
M1 = M.subs([(x,10),(w,w1),(L,L1)])
print('\nShear at x=10 (kips): ', V1)
print('Moment at x=10 (k-ft): ', M1)

V2 = V.subs([(x,15),(w,w1),(L,L1)])
M2 = M.subs([(x,15),(w,w1),(L,L1)])
print('Shear at x=15 (kips): ', V2)
print('Moment at x=15 (k-ft): ', M2)
```

Symbolic names do not have to match the variable names, though it is recommended to avoid confusion. The symbolic names are what will be displayed in the command window, while the variable name is what is used throughout the code. The order matters in this assignment.

`sy.integrate` will take the integral of the first input (w) with respect to the second input (x).

`sy.diff` will take the derivative of the first input (M) with respect to the second (x).

```
>>V(x) = -L*w/2 + w*x
>>M(x) = -L*w*x/2 + w*x**2/2
>>Check V(x) = -L*w/2 + w*x
>>
>>Shear at x=10 (kips):  0
>>Moment at x=10 (k-ft):  600
>>Shear at x=15 (kips):  -60
>>Moment at x=15 (k-ft):  450
```

Example 17.2(b) demonstrates how to substitute an array of values into a symbolic variable by using a SymPy function called `lambdify`. This essentially turns the symbolic equation into a NumPy function, allowing you to plug in and solve for an array of values. This is often used as an intermediate step to be able to plot information like deformations or forces in structural members. Plotting is covered in Sections 18-19.

Example 17.2(b) Plot shear and moment diagrams from Example 17.2(a)

```
V3 = V.subs([(w,w1),(L,L1)])
M3 = M.subs([(w,w1),(L,L1)])
```

It is possible to use `lambdify` with multiple variables, but graphing V and M the equations should have values plugged in for all variables except x (substitute in w, L).

```
V_np = sy.lambdify(x, V3, 'numpy')
M_np = sy.lambdify(x, M3, 'numpy')
```

In `lambdify` the first input is the variable(s) to define numerically, the second input is the equation/function to be evaluated, and the third input calls the numeric library to replace sympy with (e.j., `numpy`,`math`,`scipy`).

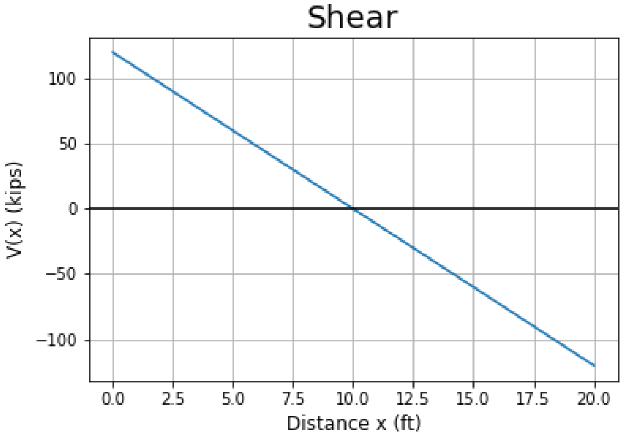
```
delta_x = .5
x_plot = np.arange(0, L1+delta_x, delta_x)

V_plot = V_np(x_plot)
M_plot = M_np(x_plot)
```

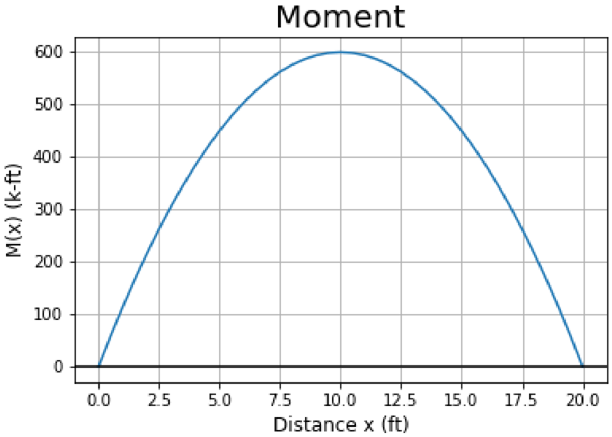
Now that the symbolic expression V3 has been lambdify-ed and saved as V_np, it possible to plug an array x_plot in using V_np(x_plot) and saving the result in an array V_plot.

```
plt.figure()
plt.plot(x_plot,V_plot)
plt.title('Shear')
plt.xlabel('Distance x (ft)')
plt.ylabel('V(x) (kips)')
plt.grid()
plt.savefig('ShearPlot.png')

plt.figure()
plt.plot(x_plot,M_plot)
plt.title('Moment')
plt.xlabel('Distance x (ft)')
plt.ylabel('M(x) (k-ft)')
plt.grid()
plt.savefig('MomentPlot.png')
```

Since x_plot and V_plot are now arrays containing numeric values with the same length, they can be plotted against each other.

Output:


Shear
V(x) (kips)
100
50
0
−50
−100
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Distance x (ft)


Moment
M(x) (k-ft)
600
500
400
300
200
100
0
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Distance x (ft)

18. Plotting Line and Scatter Plots

Python has a wide range of plotting capabilities; in this section we will cover the most commonly used plots useful for structural analysis: lines and scatter plots. There are many different libraries that can be used to plot, but we will only be covering the `matplotlib.pyplot` sub-library which is referred to with the abbreviation `plt`.

18.1 Plotting Basics

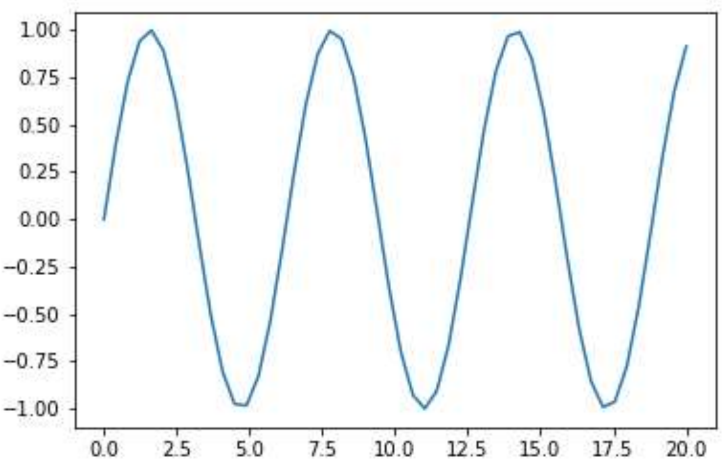
In this section we will go over how to use `plt` functions for line and scatter plots, and how to edit the general appearance of a graph.

Example 18.1.1 Graph y = sin(x) where `x = np.linspace(0,20,50).` (See section 16.1 for more on using `np.linspace`).

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0,20,50)
y = np.sin(x)

plt.plot(x, y)
```

Plots x vs y. Both are arrays of equal lengths containing numerical values.

Output:



When no formatting parameters are given in the code, Python will automatically generate a plot with a scale that fits all your data.

Example 18.1.2 Add a plot title, axis labels and a grid to the graph from Example 18.1.1.

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0,20,50)
y = np.sin(x)

plt.figure()
plt.plot(x, y)
plt.title('Sine(x) - Line Graph')
plt.xlabel('x')
plt.ylabel('sin(x)')
plt.grid()
```

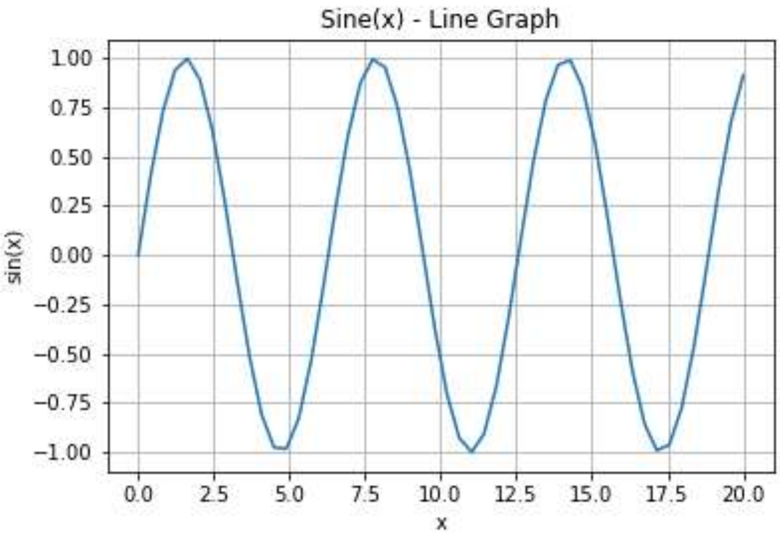
This is used to create a new figure; all plots will show up on this figure until `plt.figure()` is used to create a new one. Additional parameters can adjust figure size, colors, etc.

Creates the graph title. Additional parameters can be used to change the color, size, font, and location.

Creates axes labels. Additional parameters can change the color, size, font, and location.

Creates a grid aligning with tick marks.

Output:



Example 18.1.3 Edit the graph from Example 18.1.2 by changing the line color, style, and width. Also add markers for each point and change their color, size, and shape. Add an arrow and text box on the graph and change the x and y limits.

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0,20,50)
y = np.sin(x)

plt.figure()
plt.plot(x, y,
         color='blue',
         linestyle='dashed',
         linewidth = 3,
         marker='o',
         markerfacecolor='red',
         markersize = 6)
plt.title('Sine(x) - Line Graph')
plt.xlabel('x')
plt.ylabel('sin(x)')
plt.xlim(0, 15)
plt.ylim(-0.5, 1)
plt.arrow(11, .4, -1.5, -.2,
          head_width = 0.1,
          width = 0.01,
          color ='purple')
plt.text(10.5,.45,'A graph!')
plt.grid()
```

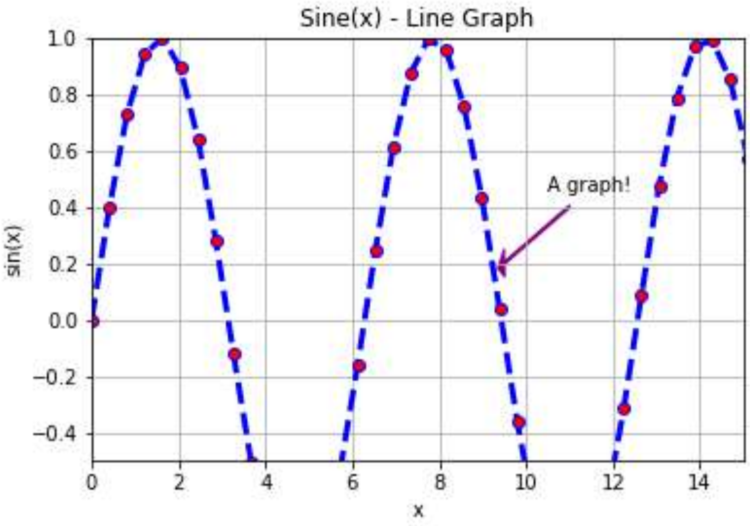
These are just some of the parameters to change the plot line and marker styles. There are many options for color, linestyle, and marker.

Changes the x and y axis limits. Will override whatever the range is for your x and y plot inputs.

Adds an arrow. The first four values indicate the arrow's x-y coordinates: that the tail is at (11, 0.4) and arrow head is 1.5 units to the left and 0.2 units down. This can be used to create vectors.

Adds text 'A graph!' at x-y coordinates (10.5, .45). Typically, it will be best *not* to hardcode x-y coordinate values but set them with respect to your data.

Output:



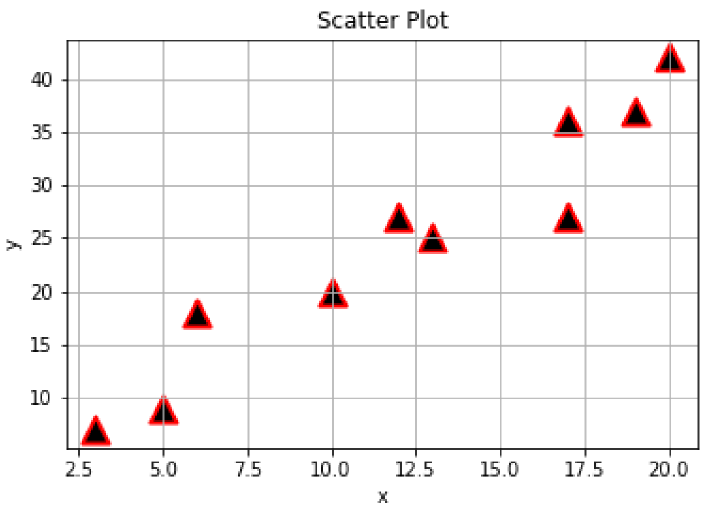
Example 18.1.4 Given the following x and y values, create a scatterplot. Edit the marker color, shape, and size.

```
x = [3, 5, 6, 10, 12, 13, 17, 17, 19, 20]
y = [7, 9, 18, 20, 27, 25, 36, 27, 37, 42]
```

```
import numpy as np
import matplotlib.pyplot as plt

x = [3, 5, 6, 10, 12, 13, 17, 17, 19, 20]
y = [7, 9, 18, 20, 27, 25, 36, 27, 37, 42]
plt.figure()
plt.scatter(x, y,
            color ='black',
            linewidths = 2,
            marker ='^',
            edgecolor ='red',
            s = 200)
plt.title('Scatter Plot')
plt.xlabel('x')
plt.ylabel('y')
plt.grid()
```

Changes the marker size.

Output:



18.2 Multiple Curves on a Single Plot

Using consecutive plot functions without defining a new figure (using `plt.figure`) will result in all data being plotted on one set of axes. Plots need not be of the same type to appear on the same figure.

Example 18.2.1 Given the following sets of x and y values, create a line plot and a scatter plot on the same axes. Provide a legend.

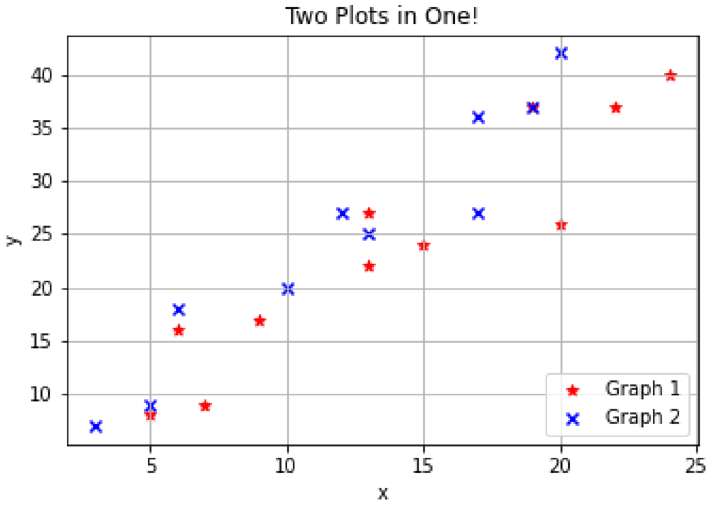
Graph 1: `x1 = [5, 7, 6, 9, 13, 13, 15, 19, 20, 22, 24]`
`y1 = [8, 9, 16, 17, 22, 27, 24, 37, 26, 37, 40]`
Graph 2: `x2 = [3, 5, 6, 10, 12, 13, 17, 17, 19, 20]`
`y2 = [7, 9, 18, 20, 27, 25, 36, 27, 37, 42]`

```
import numpy as np
import matplotlib.pyplot as plt

x1 = [5, 7, 6, 9, 13, 13, 15, 19, 20, 22, 24]
y1 = [8, 9, 16, 17, 22, 27, 24, 37, 26, 37, 40]
x2 = [3, 5, 6, 10, 12, 13, 17, 17, 19, 20]
y2 = [7, 9, 18, 20, 27, 25, 36, 27, 37, 42]


plt.figure()
plt.scatter(x1, y1, color='red', marker = '*', label='Graph 1')
plt.scatter(x2, y2, color='blue', marker = 'x', label='Graph 2')
plt.title('Two Plots in One!')
plt.xlabel('x')
plt.ylabel('y')
plt.grid()
plt.legend(loc = 'lower right')
```

The name assigned to `label` will display in the legend. Otherwise, series names can be provided in the `plt.legend` function itself.

There are several built in locations for the legend, or you can enter the precise coordinates.

Output



18.3 Subplots

Subplots allow us to create several different plots on separate axes in the same figure. Subplots work similar to a matrix layout in that for a (3,2) subplot there will be 3 rows and 2 columns of axes for a total of 6 graphs.

The use of the `for` loop in Example 18.3.1 allows four plots to be made in an efficient manner, on each loop the ‘ii’ counter is updated and used to select a different:

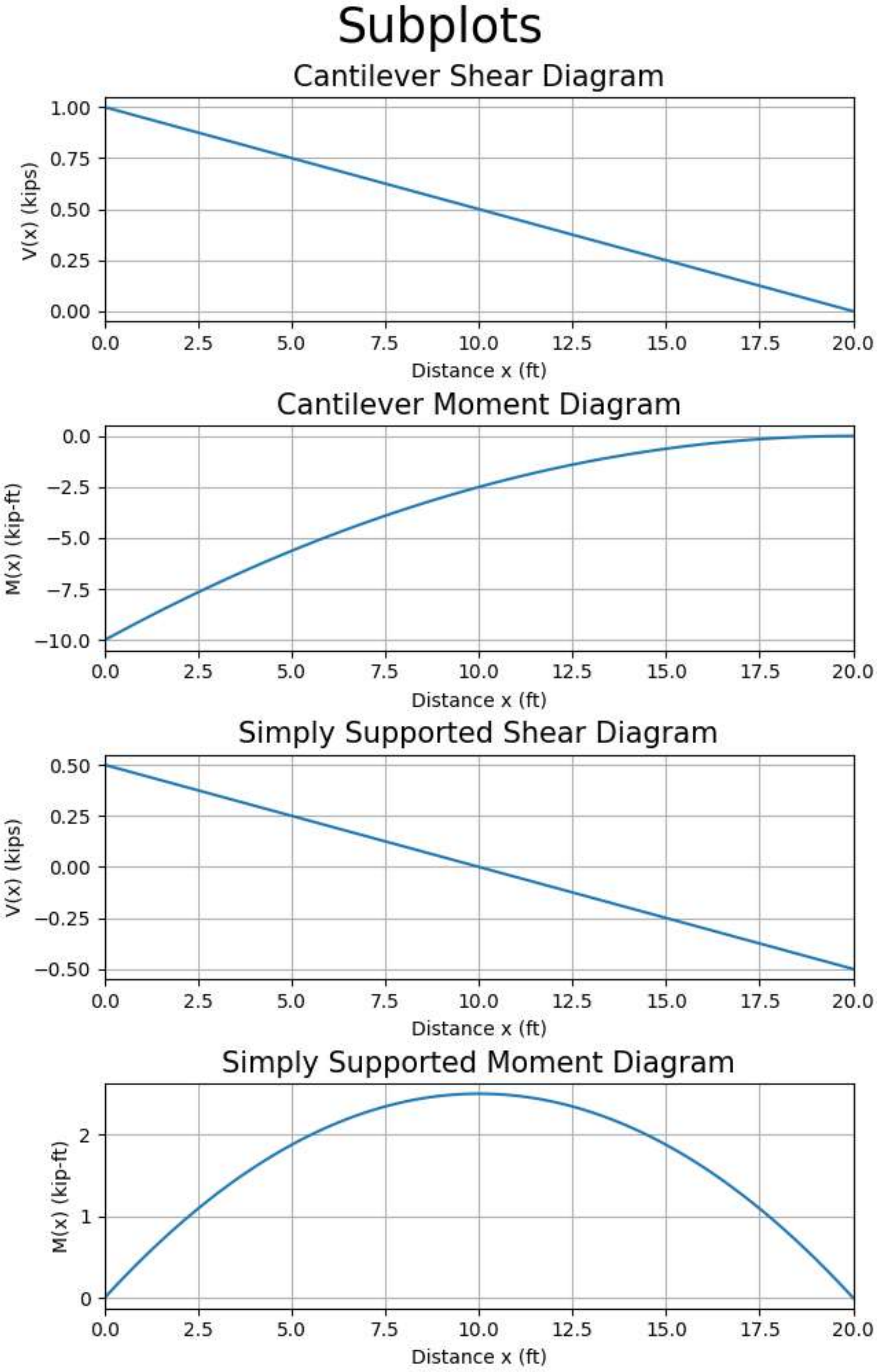
- Array from ‘y’ (list containing numpy arrays of length 100) to plot against the ‘x’ array
- Title from the ‘title’ list
- Y-axis label from the ‘ylabel’ list

Example 18.3.1 Plot the shear and moment diagrams for (a) a cantilever beam and (b) a simply supported beam. Put all four plots on a single figure in one column. (w = 0.05 klf and L = 20 ft)

```
import numpy as np
import matplotlib.pyplot as plt

w = 0.05 #klf
L = 20 #ft
x = np.linspace(0,L,100)

V_cant = w*L - w*x                      #cantilever shear
M_cant = -w*L**2/2 + w*L*x - w*x**2/2   #cantilever moment
V_ss = w*L/2 - w*x                      #simply supported shear
M_ss = w*L*x/2 - w*x**2/2               #simply supported moment

y = [V_cant, M_cant, V_ss, M_ss]
title =['Cantilever Shear Diagram', 'Cantilever Moment Diagram',
'Simply Supported Shear Diagram', 'Simply Supported Moment
Diagram']
ylabel=['V(x) (kips)', 'M(x) (kip-ft)','V(x) (kips)', 'M(x)
 (kip-ft)']


[fig1, axs] = plt.subplots(4, 1, constrained_layout=True)
fig1.suptitle('Subplots', fontsize=25)
fig1.set_figheight(10)

for ii in range(0, len(axs)):
    axs[ii].plot(x, y[ii])
    axs[ii].set_title(title[ii], fontsize=15)
    axs[ii].set_xlabel('Distance x (ft)')
    axs[ii].set_ylabel(ylabel[ii])
    axs[ii].set_xlim(min(x),max(x))
    axs[ii].grid()
```

This sets the subplot arrangement as 4 rows and 1 column.

Output:


Subplots
Cantilever Shear Diagram
V(x) (kips)
1.00
0.75
0.50
0.25
0.00
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Distance x (ft)
Cantilever Moment Diagram
M(x) (kip-ft)
0.0
−2.5
−5.0
−7.5
−10.0
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Distance x (ft)
Simply Supported Shear Diagram
V(x) (kips)
0.50
0.25
0.00
−0.25
−0.50
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Distance x (ft)
Simply Supported Moment Diagram
M(x) (kip-ft)
2
1
0
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Distance x (ft)

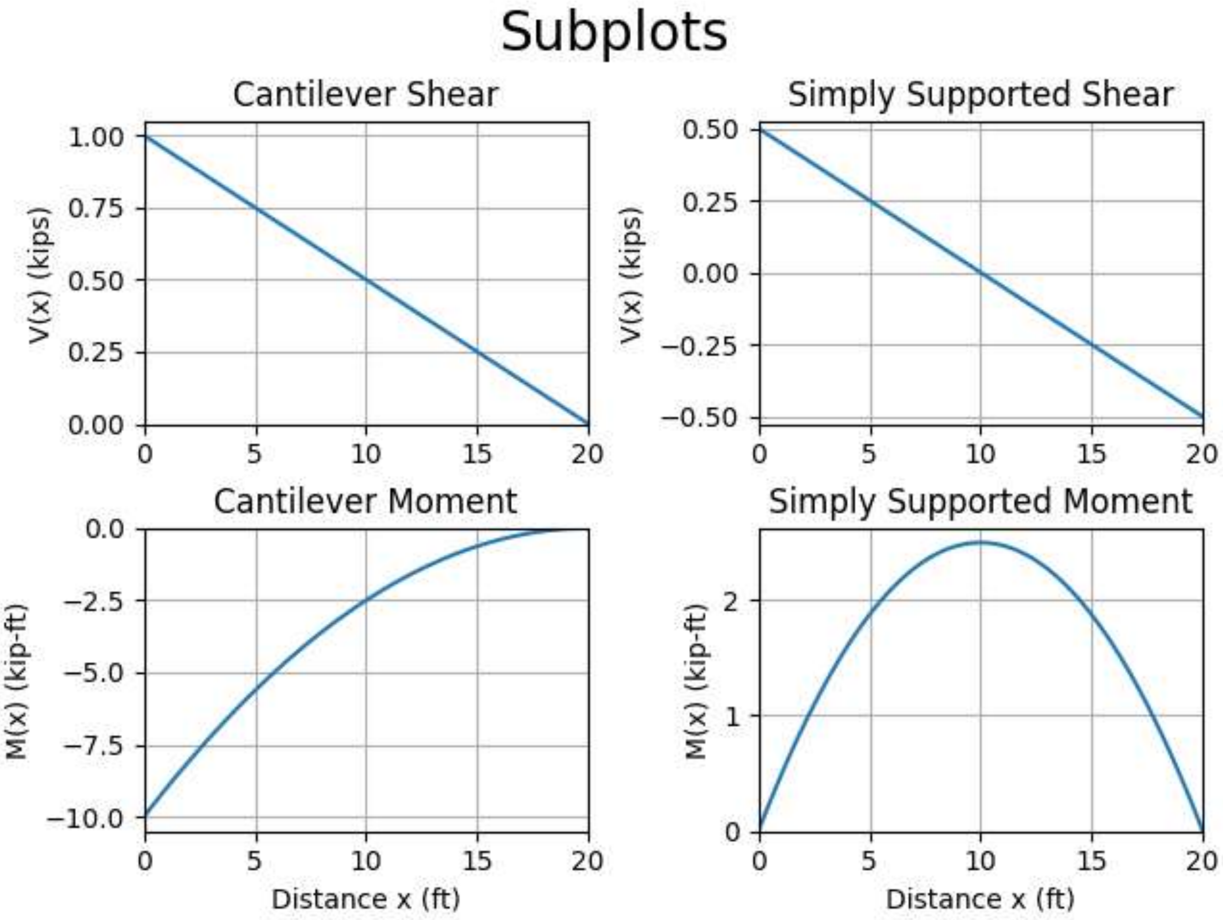
There are three major changes in Example 18.3.2 to accommodate the fact that the plots will be arranged in a 2x2 instead of 4x1 layout:

- A nested for loop is needed to create the plots in each row ‘ii’ and column ‘jj’. Consequently, each function in the nested for loop now is preceded by `axs[ii,jj]`. Information on nested for loops can be found in Section 12.2.
- The order of items in lists ‘y’, ‘title’, and ‘ylabel’ have been updated to fill in the 2x2 plots correctly with the desired information.
- The variable ‘plotnum’ was developed as a counter to track which of the 4 plots is being developed and to use the correct information from the lists ‘y’, ‘title’, and ‘ylabel’.

Example 18.3.2 Repeat Example 18.3.1, this time arranging the plots in a 2x2 with the cantilever graphs on the left and simply supported graphs on the right.

```
import numpy as np
import matplotlib.pyplot as plt

w = 0.05 #klf
L = 20 #ft
x = np.linspace(0,L,100)

V_cant = w*L - w*x                      #cantilever shear
M_cant = -w*L**2/2 + w*L*x - w*x**2/2   #cantilever moment
V_ss = w*L/2 - w*x                      #simply supported shear
M_ss = w*L*x/2 - w*x**2/2               #simply supported moment

y = [V_cant, V_ss, M_cant, M_ss]
title =['Cantilever Shear', 'Simply Supported Shear','Cantilever
Moment','Simply Supported Moment']
ylabel=['V(x) (kips)','V(x) (kips)', 'M(x) (kip-ft)', 'M(x)
(kip-ft)']

[fig2, axs] = plt.subplots(2, 2, constrained_layout=True)
fig2.suptitle('Subplots', fontsize=20)

plotnum=0
for ii in range(0, axs.shape[0]):
 for jj in range(0, axs.shape[1]):
     axs[ii,jj].plot(x, y[plotnum])
     axs[ii,jj].set_title(title[plotnum], fontsize=12)
     axs[1,jj].set_xlabel('Distance x (ft)')
     axs[ii,jj].set_ylabel(ylabel[plotnum])
     axs[ii,jj].set_xlim(min(x),max(x))
     axs[ii,jj].set_ylim(1.05*min(y[plotnum]),
     1.05*max(y[plotnum]))
     axs[ii,jj].grid()
     plotnum += 1
```

Output:


Subplots
Cantilever Shear
V(x) (kips)
1.00
0.75
0.50
0.25
0.00
0
5
10
15
20
Simply Supported Shear
V(x) (kips)
0.50
0.25
0.00
−0.25
−0.50
0
5
10
15
20
Cantilever Moment
M(x) (kip-ft)
0.0
−2.5
−5.0
−7.5
−10.0
0
5
10
15
20
Distance x (ft)
Simply Supported Moment
M(x) (kip-ft)
2
1
0
0
5
10
15
20
Distance x (ft)

18.4 Displaying and Saving a Plot

Displaying a Plot

Refer to instructions for plot display preferences in Section 3.2.

Saving a Plot

When creating a figure in Python, it will automatically display in the inline Plot window section of the Spyder interface (or a separate window if the instructions in Section 3.2 were followed). However, the figure will not be available anywhere else unless it is saved. From either the inline plot or separate window it is possible to select the save icon to store a copy of the figure, but this manual procedure would have to be executed every time a new figure is generated. The most efficient approach to automatically name and save files is to include the `plt.savefig()` function after the portion of your code that generates the plot(s).

The below line would save a previously created figure as a png file in the folder that your Python script is located. Other file types that you can save as include jpg, pdf and svg.

```
plt.savefig('FigureName.png ')
```

An alternative is shown in Example 18.4.1 which demonstrates how to automatically save your plots to a word document in the folder that your Python script is located using the `docx` library.

Example 18.4.1 Use a for loop to plot $y = x^{ii}$ for ii = 0 to ii = 3. Have each plot display automatically in a word document.

```
import numpy as np
import matplotlib.pyplot as plt
from docx import Document
from docx.shared import Inches

document = Document()
document.add_heading('Exporting Plots to Word')

x = np.linspace(0,10,100)
for ii in range(0,3):
    plt.figure()
    plt.plot(x, x**ii)
    plt.savefig('plot.png')
    document.add_picture('plot.png', width=Inches(4))

document.save('Example_18-4-1.docx')
```

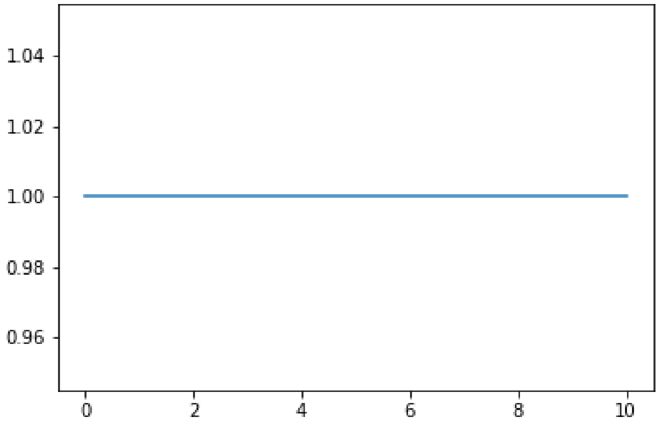
Opens a new word document.

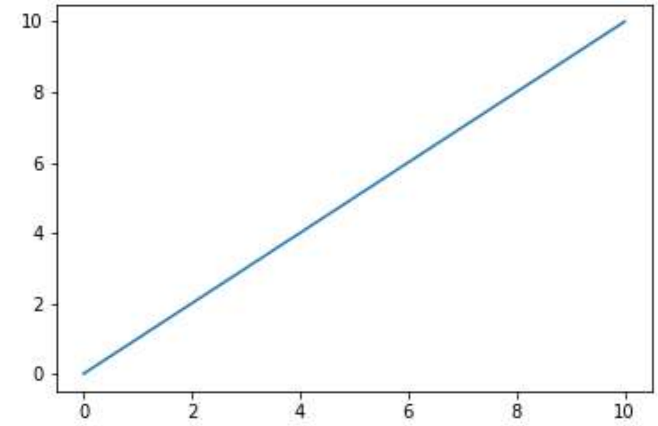
This example is simplified, but you can - and should - edit the display font, size, etc. and each individual plot.

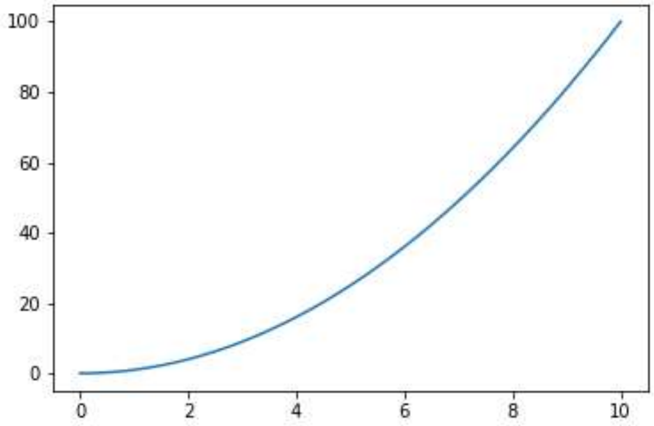
Sets the width of each image to 4 inches.

Output: The following page is what is saved as ‘Example_18-4-1.docx’

Exporting Plots to Word







18.5 Using Polyfit

The `polyfit` function within the NumPy library is used to create a line of best fit for a set of data. It takes three required parameters: x array, y array and the desired degree of polynomial to fit to the data. It creates an array of the coefficients for the best fit equations.

For `coefficients = np.polyfit(x,y,2)` then the resulting `coefficients` would be an array with three values : `[a, b, c]`, relating to the equation $ax^2 + bx + c$.

Example 18.5.1 Plot a scatter plot of $y = -x^4 + 6x^3 + 2x^2 + 3$ for $-3 \leq x \leq 6$ with 20 points. Use polyfit to create and plot 3 different lines of best fit: 1st, 2nd, and 4th degree polynomial.

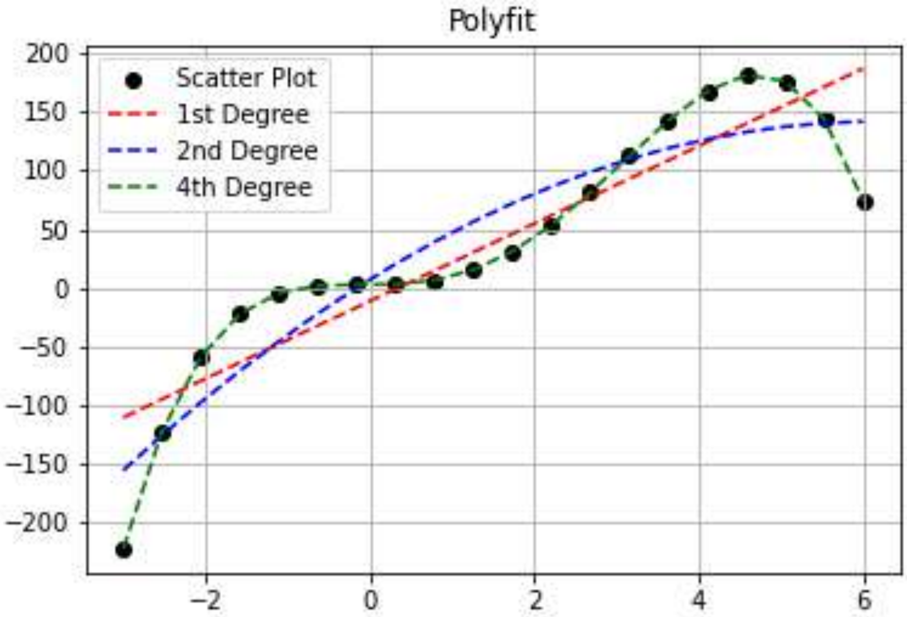
```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(-3.0, 6.0, 20)
y = -x**4 + 6*x**3 + 2*x**2 + 3

plt.figure()
plt.scatter(x, y, color='black', label='Scatter Plot')

#1st Degree polynomial
a, b = np.polyfit(x, y, 1)
y1 = a*x + b
plt.plot(x, y1, color='r', linestyle='--', label='1st Degree')

print('y1 =', a.round(2), 'x +', b.round(2) )

#2nd Degree polynomial
a, b, c = np.polyfit(x, y, 2)
y2 = a*x**2 + b*x + c
plt.plot(x, y2, color='b', linestyle='--', label='2nd Degree')

print('y2 =', a.round(2), 'x^2 +', b.round(2), 'x +', c.round())

#4th Degree polynomial
a, b, c, d, e = np.polyfit(x, y, 4)
y4 = a*x**4 + b*x**3 + c*x**2 + d*x + e
plt.plot(x, y4, color='g', linestyle='--', label='4th Degree')

print('y4 =', a.round(2), 'x^4 +', b.round(2), 'x^3 +',
c.round(2), 'x^2 +', d.round(2), 'x +', e.round(2))
plt.title('Polyfit')
plt.legend()
plt.grid()
plt.savefig('polyfit1.png')
```

Plotting the scatter plot

The output of `polyfit` is stored in individual coefficients (a, b). You can instead, store these values in an array.

```
>>y1 = 33.0 x + -11.03
>>y2 = -3.52 x^2 + 43.57 x + 7.0
>>y4 = -1.0 x^4 + 6.0 x^3 + 2.0 x^2 + 0.0 x + 3.0
```

Note that the 4th degree `polyfit` line matches our initial input.



18.6 Finding Roots

Two of the possible methods for finding the roots of a function or data set include the built-in `np.roots` function and coding a routine with the `np.where` function. The `np.roots` function is used for finding the roots of polynomials by taking the equation coefficients and returning the x-values. For example, the code to find the roots of $y = x^2 - 4$ would be as follows:

```
import numpy as np

roots = np.roots([1, 0, -4])
print(roots)
```

```
>> [ 2. -2.]
```

The disadvantages of using this method are that it only works for polynomials, you cannot set a limit on the range of data to check (it will always check for all possible roots), and the roots are not displayed in any particular order. For these reasons it may make more sense to create your own method for finding roots. This can be done in many different ways, but the following example will use the `np.where` function to find the x-values where the data changes signs.

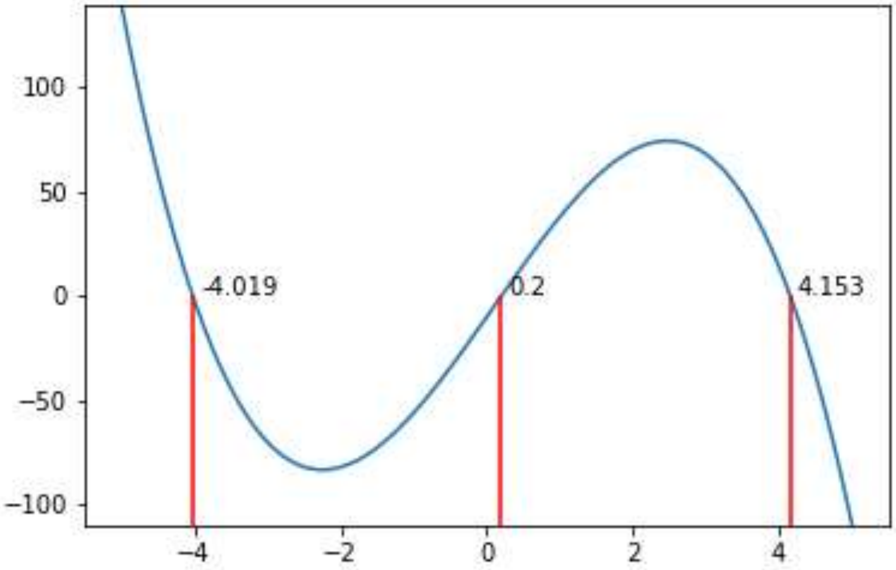
Example 18.6.1 Given the equation $y = -3x^3 + x^2 + 50x - 10$ (where $-5 < x < 5$), determine the roots using the `np.roots` and using `np.where`. Plot the results with root locations labeled.

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(-5, 5, 500)
y = -3*x**3 + x**2 + 50*x - 10

# Using np.roots
roots1 = np.real( np.roots([-3, 1, 50, -10]) ).round(3)
print('Roots from np.roots: ', roots1)

# Using np.where
index2 = np.where( np.sign(y[:-1]) != np.sign(y[1:]) )[0]
roots2 = x[index2+1].round(3)
print('Roots from np.where: ', roots2)

#Plotting Roots
plt.figure()
plt.plot(x, y)
plt.ylim(y.min(), y.max())

for ii in range(0, len(roots1)):
    plt.vlines(x = roots1[ii],ymin = y.min(),ymax=0,color='red')
    plt.text(roots1[ii]+.1,0,roots1[ii])

plt.savefig('Roots.png')
```

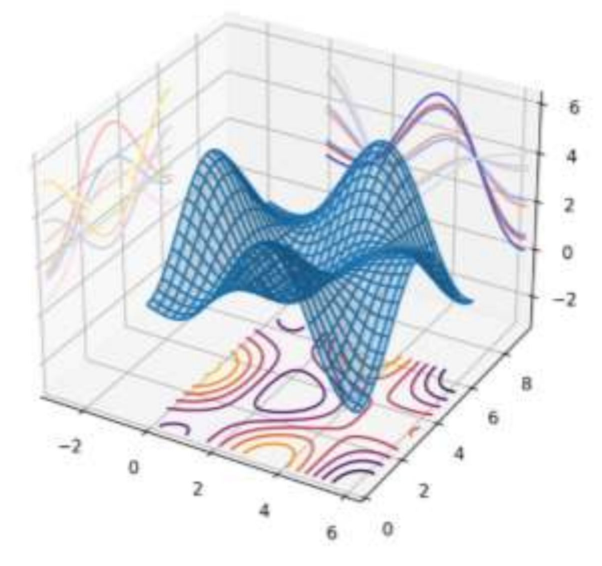
`np.where` is finding the *index* where the sign of y changes by comparing the sign of each index to the subsequent one until they are not equal.

`np.where` will return the values and the datatype, so calling the first index will pull only the values. Unlike `np.roots`, these values will be indices, not the actual x-values. This is why we are taking x[index] in the following line.

```
>>Roots from np.roots:  [-4.019  4.153  0.2  ]
>>Roots from np.where:  [-4.018  0.21   4.158]
```



Motivation Station



This plotting section just scratched the surface of what you can do with the Matplotlib.pyplot sub-library. Shown left is a graph that encapsulates a few more tools that exist within the library: 3D figures, colormaps, contour maps, and projections. The code for this is provided in the supplementary files in case you wanted to play around with the parameters!

There are also ways to animate plots, add images, sliders, and more that you can explore.

19. Bar Charts, Histograms and Pie Charts

In this section we will look at some of the other graph types that live within the `matplotlib.pyplot` sub-library, namely the `bar`, `hist`, and `pie` functions.

19.1 Bar Charts

Like line and scatter plots, the `bar` function takes two required parameters, the first being the x "values" (or labels) and the second being the corresponding magnitude to set the bar height. Additional parameters can change bar labels, colors, sizes, and fonts.

Example 19.1.1 Given the following data, create a bar chart showing the number of students enrolled in each course.

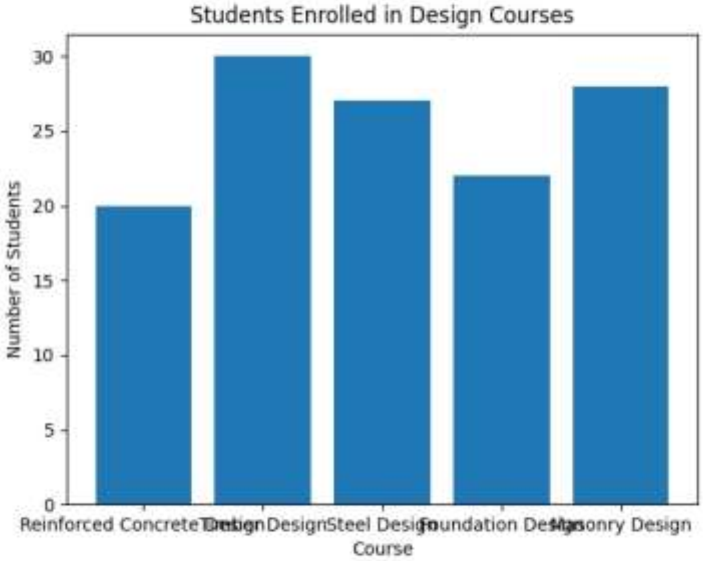
Course Name	Number of Students
Reinforced Concrete Design	20
Timber Design	30
Steel Design	27
Foundation Design	22
Masonry Design	28

```
import numpy as np
import matplotlib.pyplot as plt

x = ['Reinforced Concrete Design', 'Timber Design', 'Steel
Design', 'Foundation Design', 'Masonry Design']
y = [20, 30, 27, 22, 28]

plt.figure()
plt.bar(x,y, width=0.8)

plt.title('Students Enrolled in Design Courses')
plt.xlabel('Course')
plt.ylabel('Number of Students')

plt.savefig('BarChart1.png')
```

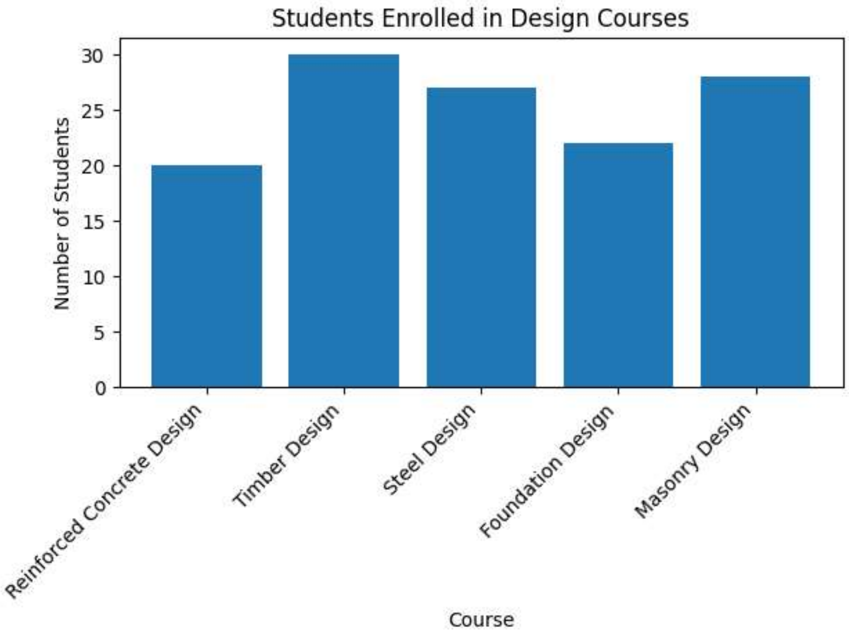
Rather than create separate lists for your data like in this example, consider using a dictionary. See next example.

The default spacing between the center of each bar is 1, so setting the width of the bars to 0.8 will leave space between them.

Output:



Notice how the course names overlap in the x-axis labels. One approach to resolving this is rotating the labels: `plt.xticks(rotation=45, ha="right")` where `ha` stands for horizontal alignment and indicates what part of the text will align with the tick mark. Adding this and `plt.tight_layout( )` prior to the `plt.savefig('BarChart1.png')` results in the output shown below while avoiding the x-axis labels from getting cut off.



Another possible solution is to make the bar chart horizontal using the `barh` function as shown in Example 19.1.2.

Example 19.1.2 Change the bar graph from example 19.1.1 to be horizontal with labeled values.

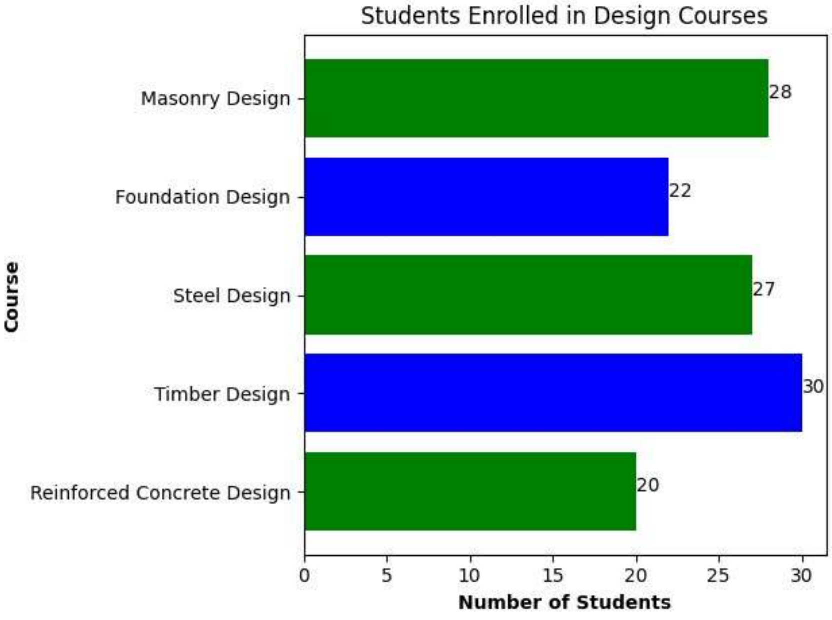
```
import numpy as np
import matplotlib.pyplot as plt

coursedata = {'Reinforced Concrete Design':20, 'Timber
Design':30, 'Steel Design':27, 'Foundation Design':22, 'Masonry
Design':28}
x = list(coursedata.keys())
y = list(coursedata.values())

plt.figure()
plt.barh(x, y, color = ['green', 'blue'])

plt.title('Students Enrolled in Design Courses')
plt.xlabel('Number of Students', weight = 'bold')
plt.ylabel('Course', weight = 'bold')


for ii in range(0,len(x)):
    plt.text(y[ii], ii, y[ii])

plt.tight_layout()
plt.savefig('BarChart2.png')
```

Data is now stored in a dictionary rather than creating individual lists. For more on dictionaries see Section 8.

This will add the value label for each bar onto the plot. Note that if you have a vertical bar graph the coordinates of the text would be switched.

If parts of your plot or labels are being cut off, using this function will compress it in a way that will fit.

Output:



Stacked Bar Charts

It is also possible to create stacked bar charts by adding a parameter `bottom` to the `bar` function. This allows us to place data atop previous values. The following example shows two subsets of data, but there can be an unlimited number of "stacks".

Example 19.1.3 Create a stacked bar chart with students in each course (juniors vs. seniors).

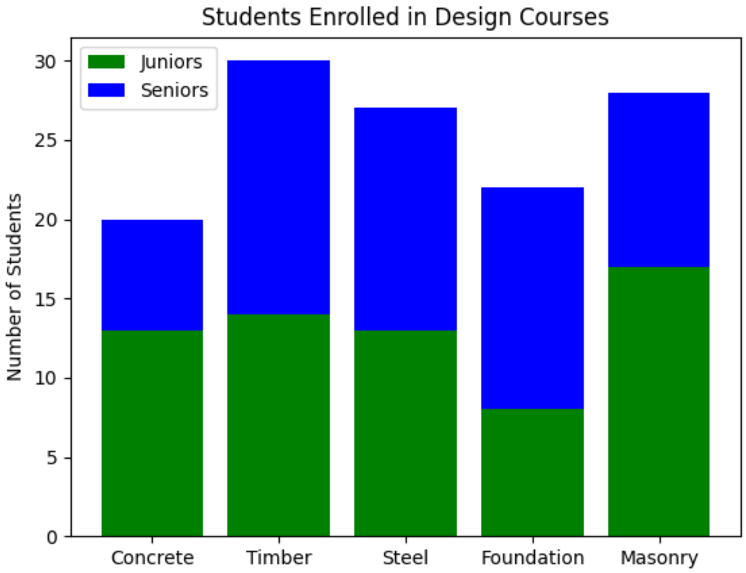
Course Name	Juniors	Seniors
Reinforced Concrete Design	13	7
Timber Design	14	16
Steel Design	13	14
Foundation Design	8	14
Masonry Design	17	11

```
import numpy as np
import matplotlib.pyplot as plt

x = ['Concrete', 'Timber', 'Steel', 'Foundation', 'Masonry']
y1 = [13, 14, 13, 8, 17] #juniors
y2 = [7, 16, 14, 14, 11] #seniors

plt.figure()
plt.bar(x, y1, color='green')
plt.bar(x, y2, bottom=y1, color='blue')

plt.title('Students Enrolled in Design Courses')
plt.ylabel('Number of Students')
plt.legend(['Juniors', 'Seniors'])
plt.savefig('BarChart3.png')
```

If converting lists into a dictionary, insert the following after defining 'x', 'y1', and 'y2':

```
# Organize List Data into Dictionary
coursedata={}
for ii in range(0,len(x)):
    coursedata[x[ii]]={}
    coursedata[x[ii]]['juniors']=y1[ii]
    coursedata[x[ii]]['seniors']=y2[ii]

# Read from Dictionary to List
x_read = list(coursedata.keys())

y1 = [0]*len(x)
y2 = [0]*len(x)
for ii in range(len(x_read)):
    y1[ii] = coursedata[x_read[ii]]['juniors']
    y2[ii] = coursedata[x_read[ii]]['seniors']
```

Output:



Multiple Bar Charts

By modifying bar widths and spacing we can plot multiple “sets” of bar charts. In the Example 19.1.4 students enrolled in courses (“categories”) will now be separated by quarter (“sets”).

Example 19.1.4 Create multiple bar charts on one plot, showing the number of students in each course for different quarters. Use a legend to indicate the courses.

Course Name	**Fall**	**Winter**	**Spring**
Reinforced Concrete Design	36	19	20
Timber Design	18	21	30
Steel Design	24	32	27
Foundation Design	18	25	22
Masonry Design	20	18	28

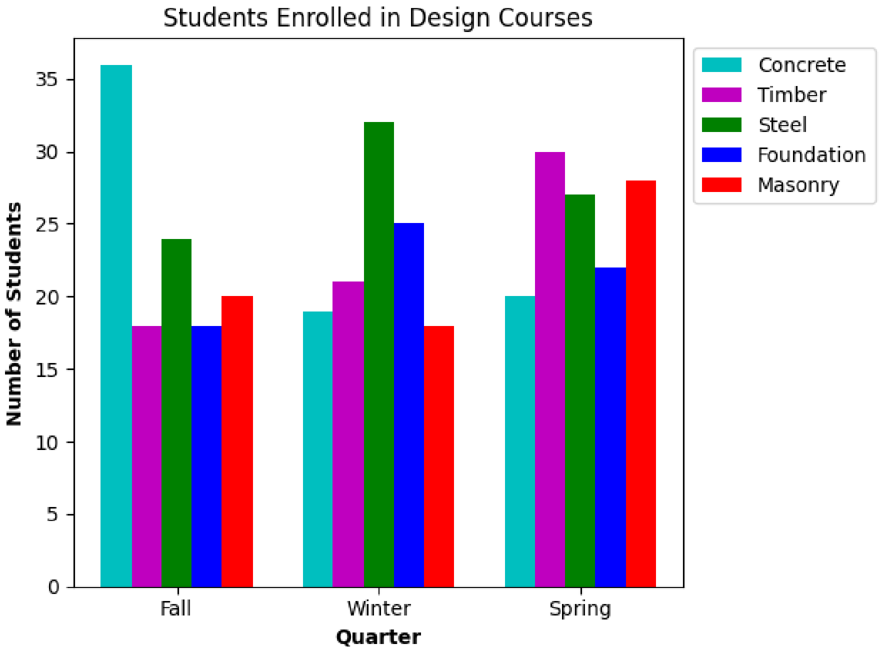
```
import numpy as np
import matplotlib.pyplot as plt

# Data in List Format
x = ['Concrete', 'Timber', 'Steel', 'Foundation', 'Masonry']
x2 =['Fall', 'Winter', 'Spring']
y1 = [36, 18, 24, 18, 20] #Fall
y2 = [19, 21, 32, 25, 18] #Winter
y3 = [20, 30, 27, 22, 28] #Spring
```

```
# Organize into Dictionary
coursedata={}
for ii in range(0,len(x)):
    coursedata[x[ii]]={}
    coursedata[x[ii]][x2[0]]=y1[ii]
    coursedata[x[ii]][x2[1]]=y2[ii]
    coursedata[x[ii]][x2[2]]=y3[ii]

# Read from Dictionary
courses = len(coursedata.keys())
quarters = len(coursedata[x[0]])

x_plot=np.arange(0,quarters)
y_plot=np.zeros([courses,quarters])

for ii in range(0,courses):
    for jj in range(0,quarters):
        y_plot[ii,jj] = coursedata[x[ii]][x2[jj]]

# Generate Plot
plt.figure()
barwidth = 0.15
colorlist=['c','m','g','b','r']

for ii in range(0,courses):
    x_plot= x_plot + barwidth
    if ii == np.floor(courses/2):
        xloc = x_plot
    plt.bar(x_plot, y_plot[ii,:],
            color = colorlist[ii],
            width = barwidth,
            label = x[ii])

plt.title('Students Enrolled in Design Courses')
plt.ylabel('Number of Students', weight = 'bold')
plt.xlabel('Quarter', weight = 'bold')
plt.xticks(xloc, x2)
plt.legend(bbox_to_anchor=(1, 1))
plt.tight_layout()
plt.savefig('BarChart4.png')
```

A good rule of thumb for choosing a bar width is to divide 1 by the number of categories (1/5 = 0.2) and subtract a small amount to create space between sets (hence 0.15).

For the location of the x-axis labels, the x-coordinates closest to the center of each set (xloc array in this example) is calculated in the `if` statement above and used here.

Output:



It is also good practice to check for color-blindness on any graphs that have multiple colors to ensure that everyone can properly interpret your data. This link provides a free color-blindness test on any image. If you are using a lot of different data sets, be sure that any adjacent data is distinguishable or use different symbols/patterns.

19.2 Histograms

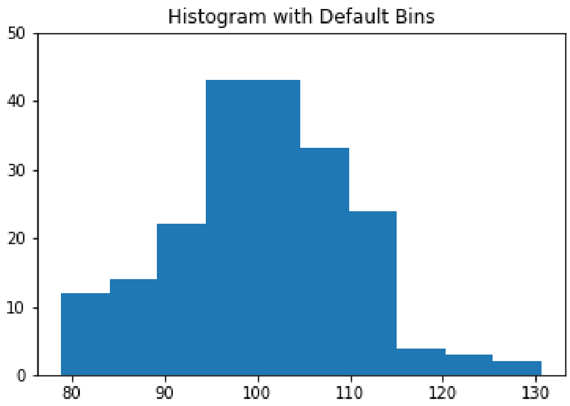
A histogram provides an approximate representation of the distribution of a data set. The `hist` function in the `matplotlib.pyplot` sub-library has one required parameter of an ‘x’ array. Additional parameters can change the number of bins and the general appearance of the plot.

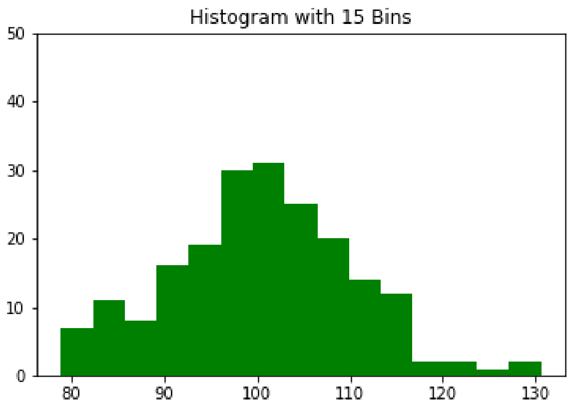
Example 19.2.1 Create a histogram of a set of data with 200 values with a mean of 100 and standard deviation of 10. (Use `np.random.normal`)

```
import numpy as np
import matplotlib.pyplot as plt

x = np.random.normal(100, 10, 200)

plt.figure()
plt.hist(x)
plt.ylim([0,50])
plt.title('Histogram with Default Bins')
plt.savefig('Histogram1.png')
```

```
plt.figure()
plt.hist(x, 15, color='green')
plt.ylim([0,50])
plt.title('Histogram with 15 Bins')
plt.savefig('Histogram2.png')
```

Output:





19.3 Pie Charts

Pie charts are used to show part-to-whole relationships for datasets. The `pie` function of the `matplotlib.pyplot` sub-library only has one required parameter and that is the relative size of each pie slice. Additional parameters can be used to add labels, change colors, designs etc.

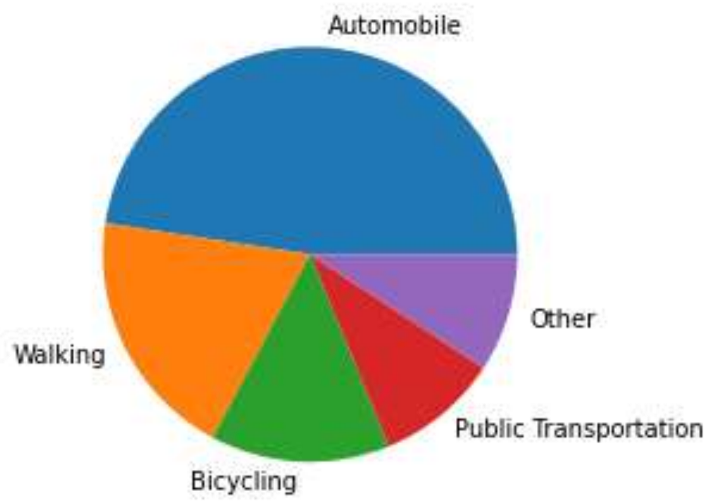
Example 19.3.1 Given the following data, create a pie chart showing the percentage of students who use each mode of transportation, labeling each slice.

Primary Mode of Transportation	**Number of Students**
Personal Automobile	119
Walking	49
Bicycling	35
Public Transportation	24
Other	23

```
import matplotlib.pyplot as plt

labels = ['Automobile', 'Walking', 'Bicycling', 'Public
Transportation', 'Other']
sizes = [119, 49, 35, 24, 23]

plt.figure()
plt.pie(sizes, labels=labels)

plt.savefig('PieChart1.png')
```

The "sizes" of each slice do not need to be percents, Python will automatically portion the pie chart accordingly.

Output:



Example 19.3.2 Edit the pie chart from the previous example to include the percentage values, custom colors, and an exploded slice.

```
import matplotlib.pyplot as plt


labels = ['Automobile', 'Walking', 'Bicycling',
          'Public Transportation', 'Other']
sizes  = [119, 49, 35, 24, 23]
colors = ['steelblue', 'dodgerblue', 'deepskyblue',
          'lightskyblue', 'aliceblue']
explode = (0, .2, 0, 0, 0)
```

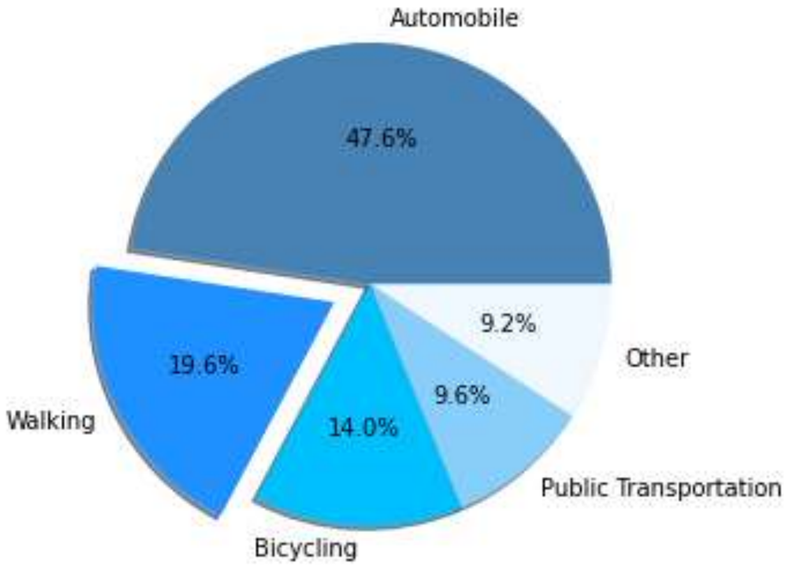
The array contains values indicating how far each slices will be moved (exploded) from pie’s center.

```
plt.figure()
plt.pie(sizes, labels=labels, colors=colors, explode=explode,
        autopct='%1.1f%%', radius=1.2, shadow=True)
```

Adds the percent values to each slice. `1.1` indicates one decimal.

```
plt.savefig('PieChart2.png')
```

Output:



20. Printing

Once a segment of code is completed and runs properly, it is helpful to print the result for inclusion in a calculation package or other report type document. There are many ways to format code output: from a simple line of text to a tabulated output printed to the Spyder command window to even creating an Excel spreadsheet with multiple tabs.

20.1 Printing Basics

The easiest way to print something out to the command window is to use the `print` function. This function can be used for many different data types, but it is necessary to convert them to strings before printing.

Example 20.1.1 Printing a variable, an array, and a matrix.

```
import numpy as np

x = 7**2
arr = np.arange(0,10,2)
mtrx = np.array([[1,2,3],
                 [4,5,6]])


print('x = ', x, '\nArray = ', arr, '\nMatrix = ', mtrx)
print('Matrix = \n', mtrx)
```

Single or double quotation marks are used to enclose string elements.

‘\n’ enters a new line as does starting a new print function.

Note that using a comma to separate elements allows you to mix different data types. If you wanted to use ‘+’ instead, it would be necessary to convert all data types to strings.

```
>> x = 49
>> Array = [0, 2, 4, 6, 8]
>> Matrix = [[1 2 3]
>> [4 5 6]]
>> Matrix =
>> [[1 2 3]
>> [4 5 6]]
```

It may be necessary adjust the print command to get the result formatted as desired. In the second instance, notice how the first row of the matrix has been moved to a new line so it is easier to read.

Often with the `print` function the `format` function is used. This creates a place holder within a string to display variables to avoid breaking up the string like in Example 20.1.1.

Example 20.1.2 Printing a variable, an array and matrix using `format` function

```
import numpy as np

x = 7**2
arr = np.arange(0,10,2)
mtrx = np.array([[1,2,3],
                [4,5,6]])
```

Leave an open bracket {} at each location where you want to print a variable in the string. End the string with `.format()` and list the variables you want to fill the brackets in order.

```
print('x = {}\nArray = {}\nMatrix = {}\n'.format(x, arr, mtrx))
print('x = {2}\nArray = {1}\nMatrix = {0}'.format(x, arr, mtrx))
```

You can also call the variable you want to print by its index. This results in a printing order of mtrx, arr, x. See below.

```
print('\n%s = %.2f' % ('Variable', x))
```

The `%` operator is another placeholder for variables. Here `%s` is for a string ‘Variable’ and `%.2f` is for a float with 2 decimals ‘x’. This shows the `%` operator can adjust decimals, convert data types, etc. See here for more on `%` and `{}`.

```
>> x = 49
>> Array = [0, 2, 4, 6, 8]
>> Matrix =
>> [[1 2 3]
>> [4 5 6]]
>>
>> x =
>> [[1 2 3]
>> [4 5 6]]
>> Array = [0, 2, 4, 6, 8]
>> Matrix = 49
>>
>> Variable = 49.00
```

Calling index `{2}` in the first bracket of the string prints `mtrx` instead of x as it is in the 2nd index of `.format()`.

20.2 Tabular Output

When outputting large amounts of data, it may be advantageous to present it in a table. There are several ways to achieve this, namely: `print` and `format` functions, Pandas library, and `tabulate` module. These are covered in Examples 20.2.1-20.2.3.

Example 20.2.1 Create a table complete with headers listing the Area, Moment of Inertia, and the Elastic and Plastic Section Modulus for all steel W6x beams using `print` and `format`.

See Section 20.5 for details on Unicode used here for superscripts.

```
import numpy as np

headers = ('Shape', 'Area (in\u00B2)', 'I (in\u2074)', 'S
(in\u00B3)', 'Z (in\u00B3)')
shapes=('W6x25','W6x20','W6x15','W6x16','W6x12','W6x9','W6x8.5')
A = np.array([7.34, 5.87, 4.43, 4.74, 3.55, 2.68, 2.52])
I = np.array([53.4, 41.4, 29.1, 32.1, 22.1, 16.4, 14.9])
S = np.array([16.7, 13.4, 9.72, 10.2, 7.31, 5.56, 5.10])
Z = np.array([18.9, 14.9, 10.8, 11.7, 8.30, 6.23, 5.73])
```

This pads the string with 12 spaces to the right (see this marked below in red). Using > instead would pad it to the left and ^ would pad both sides.

```
print('-------------------------------------------------------')
print('{:<8} {:<12} {:<12} {:<12} {}'.format(headers[0],
     headers[1],headers[2],headers[3], headers[4]))
print('-------------------------------------------------------')
```

Using the same padding as the headers will line up the elements of the table. Here we are printing each row at a time using a for loop.

```
for ii in range(0, len(shapes)):
     print('{:<8} {:<12} {:<12} {:<12} {}'.format(shapes[ii],
     A[ii],I[ii], S[ii], Z[ii]))
```

```
>>-------------------------------------------------------
>>Shape    Area (in²)   I (in⁴)      S (in³)      Z (in³)
>>-------------------------------------------------------
>>W6x25    7.34         53.4         16.7         18.9
>>W6x20    5.87         41.4         13.4         14.9
>>W6x15    4.43         29.1         9.72         10.8
>>W6x16    4.74         32.1         10.2         11.7
>>W6x12    3.55         22.1         7.31         8.3
>>W6x9     2.68         16.4         5.56         6.23
>>W6x8.5   2.52         14.9         5.1          5.73
>>-------------------------------------------------------
```

Another resource to learn more about formatting strings refer to: https://mkaz.blog/working-with-python/string-formatting/ .

Example 20.2.2 Repeat the same example using `pandas` to format the table.

```
import numpy as np
import pandas as pd

shapes =('W6x25','W6x20','W6x15','W6x16','W6x12','W6x9','W6x8.5')
A = np.array([7.34, 5.87, 4.43, 4.74, 3.55, 2.68, 2.52])
I = np.array([53.4, 41.4, 29.1, 32.1, 22.1, 16.4, 14.9])
S = np.array([16.7, 13.4, 9.72, 10.2, 7.31, 5.56, 5.10])
Z = np.array([18.9, 14.9, 10.8, 11.7, 8.30, 6.23, 5.73])
```

This data type is referred to as a ‘dictionary’. It begins with the header in quotes, referred to as ‘keys’, followed by the column data. See Section 8.

```
data = {'Area (in\u00B2)': A.tolist(),'I (in\u2074)':
I.tolist(),  'S (in\u00B3)': S.tolist(),'Z (in\u00B3)':
Z.tolist()}
```

To display elements of an array in separate columns, it must be converted to a list.

```
table = pd.DataFrame(data, shapes)
print(table)

query = 'W6x9'
print('\nShape:',query)
print(table.loc[query])
```

Panda’s `DataFrame` function will take a dictionary ‘data’ and the row (index) names from ‘shapes’ and format it in a table called a ‘dataframe’.

```
>>          Area (in²)  I (in⁴)  S (in³)  Z (in³)
>>W6x25          7.34     53.4    16.70    18.90
>>W6x20          5.87     41.4    13.40    14.90
>>W6x15          4.43     29.1     9.72    10.80
>>W6x16          4.74     32.1    10.20    11.70
>>W6x12          3.55     22.1     7.31     8.30
>>W6x9           2.68     16.4     5.56     6.23
>>W6x8.5         2.52     14.9     5.10     5.73
>>
>>Shape: W6x9
>>Area (in²)     2.68
>>I (in⁴)       16.40
>>S (in³)        5.56
>>Z (in³)        6.23
>> Name: W6x9, dtype: float64
```

An advantage of using this method is that a dataframe is its own structure, meaning you can easily call on a row using `.loc[]` and display just that row’s information.

Example 20.2.3 Repeat the same example now using `tabulate` to format the table.

```
import tabulate as tab

data = {'Shape': shapes, 'Area (in\u00B2)': A.tolist(),
'I (in\u2074)': I.tolist(),'S (in\u00B3)': S.tolist(),
'Z (in\u00B3)': Z.tolist()}

tbl1 = tab.tabulate(data, headers= 'keys')
tbl2 = tab.tabulate(data, headers= 'keys',tablefmt='fancy_grid')

print(tbl1,"\n\n", tbl2)
```

Need to update ‘`data`’ from Example 20.2.2 to include ‘shapes’ information. Note that in addition to dictionaries, `tabulate` also accepts lists, list of lists, arrays and matrcices.

```
>>Shape    Area (in²)  I (in⁴)  S (in³)  Z (in³)
>>-------  ----------  -------  -------  -------
>>W6x25          7.34     53.4    16.7     18.9
>>W6x20          5.87     41.4    13.4     14.9
>>W6x15          4.43     29.1     9.72    10.8
>>W6x16          4.74     32.1    10.2     11.7
>>W6x12          3.55     22.1     7.31     8.3
>>W6x9           2.68     16.4     5.56     6.23
>>W6x8.5         2.52     14.9     5.1      5.73
>>
>> ╒═════════╤══════════════╤═══════════╤═══════════╤═══════════╕
>> │ Shape   │   Area (in²) │   I (in⁴) │   S (in³) │   Z (in³) │
>> ╞═════════╪══════════════╪═══════════╪═══════════╪═══════════╡
>> │ W6x25   │         7.34 │      53.4 │     16.7  │     18.9  │
>> ├─────────┼──────────────┼───────────┼───────────┼───────────┤
>> │ W6x20   │         5.87 │      41.4 │     13.4  │     14.9  │
>> ├─────────┼──────────────┼───────────┼───────────┼───────────┤
>> │ W6x15   │         4.43 │      29.1 │      9.72 │     10.8  │
>> ├─────────┼──────────────┼───────────┼───────────┼───────────┤
>> │ W6x16   │         4.74 │      32.1 │     10.2  │     11.7  │
>> ├─────────┼──────────────┼───────────┼───────────┼───────────┤
>> │ W6x12   │         3.55 │      22.1 │      7.31 │      8.3  │
>> ├─────────┼──────────────┼───────────┼───────────┼───────────┤
>> │ W6x9    │         2.68 │      16.4 │      5.56 │      6.23 │
>> ├─────────┼──────────────┼───────────┼───────────┼───────────┤
>> │ W6x8.5  │         2.52 │      14.9 │      5.1  │      5.73 │
>> ╘═════════╧══════════════╧═══════════╧═══════════╧═══════════╛
```

Defining the headers creates a separate row at the top of the table.

The `tablefmt` parameter allows you to change the appearance of the table.

20.3 Printing to Excel

There are two main ways of printing to Excel: using `xlsxwriter` or `pandas`. `xlsxwriter` is preferred for *creating* Excel files with Python, as it has more functions available. However, one of its drawbacks is that it cannot be used to read or modify existing files. That is where `pandas` is much more helpful. To demonstrate how to use these tools, we will continue the previous example using W6x beams. For additional tips, see the following links for xlswriter and pandas.

Example 20.3.1 Repeat Example 20.2.1, now printing to an Excel sheet using `xlsxwriter`. Also export the information as a matrix.

```
import numpy as np
import xlsxwriter

headers = ('Shape', 'Area', 'I', 'S', 'Z')
shapes=('W6x25','W6x20','W6x15','W6x16', 'W6x12','W6x9','W6x8.5')
A = np.array([7.34, 5.87, 4.43, 4.74, 3.55, 2.68, 2.52])
I = np.array([53.4, 41.4, 29.1, 32.1, 22.1, 16.4, 14.9])
S = np.array([16.7, 13.4, 9.72, 10.2, 7.31, 5.56, 5.10])
Z = np.array([18.9, 14.9, 10.8, 11.7, 8.30, 6.23, 5.73])

mtrx =  np.array([A,I,S,Z])
```

Create a new Excel file called ‘WideFlanges’ with a sheet within that file called ‘Properties’. These are the names that will appear in Excel. In Python, they are defined as variables ‘workbook’ and ‘worksheet1’.

```
workbook = xlsxwriter.Workbook('WideFlanges.xlsx')
worksheet1 = workbook.add_worksheet('Properties')

row = 0
col = 0
```

Several formatting options exist that can change text font, size, color etc; as well as number formatting. See Fig. 20.3.1 below.

```
bold = workbook.add_format({'bold': 'true'})
num_format = workbook.add_format({'num_format': '#,##0.00'})

worksheet1.write_row(row, col, headers, bold)
worksheet1.write_column(row+1, col, shapes)
```

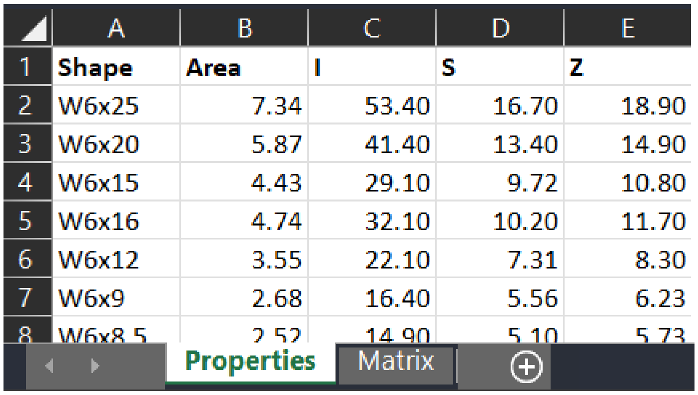
The `write` function is used to fill either a row or column. It takes the start row number, start column number, and the data. This can be followed by any formatting parameters.

```
for ii in range(0,len(mtrx)):
    worksheet1.write_column(row+1, col+(ii+1), mtrx[ii,:],
    num_format)
```

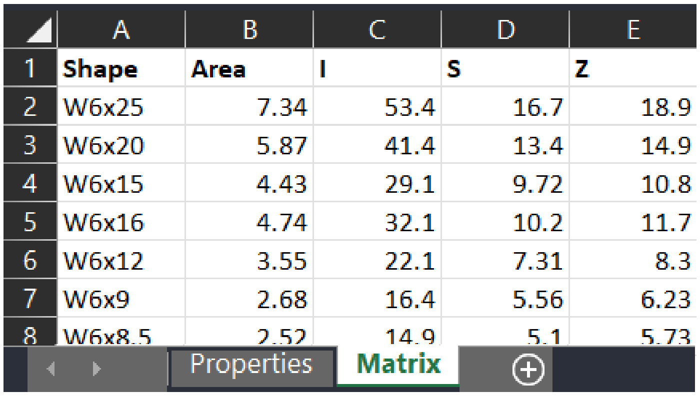
```
worksheet2 = workbook.add_worksheet('Matrix')
row = 0
col = 0

worksheet2.write_row(row, col, headers, bold)
worksheet2.write_column(row+1, col, shapes)

for ii in range(0, len(A)):
    worksheet2.write_row(ii+1, col+1, mtrx[:,ii])


workbook.close()
```

Add a new sheet called ‘Matrix’ and use a `for` loop to print the data in th matrix to Excel row by row. This is a more concise way to achieve the same result.

Output:



	A	B	C	D	E
1	**Shape**	**Area**	**I**	**S**	**Z**
2	W6x25	7.34	53.40	16.70	18.90
3	W6x20	5.87	41.40	13.40	14.90
4	W6x15	4.43	29.10	9.72	10.80
5	W6x16	4.74	32.10	10.20	11.70
6	W6x12	3.55	22.10	7.31	8.30
7	W6x9	2.68	16.40	5.56	6.23
8	W6x8.5	2.52	14.90	5.10	5.73

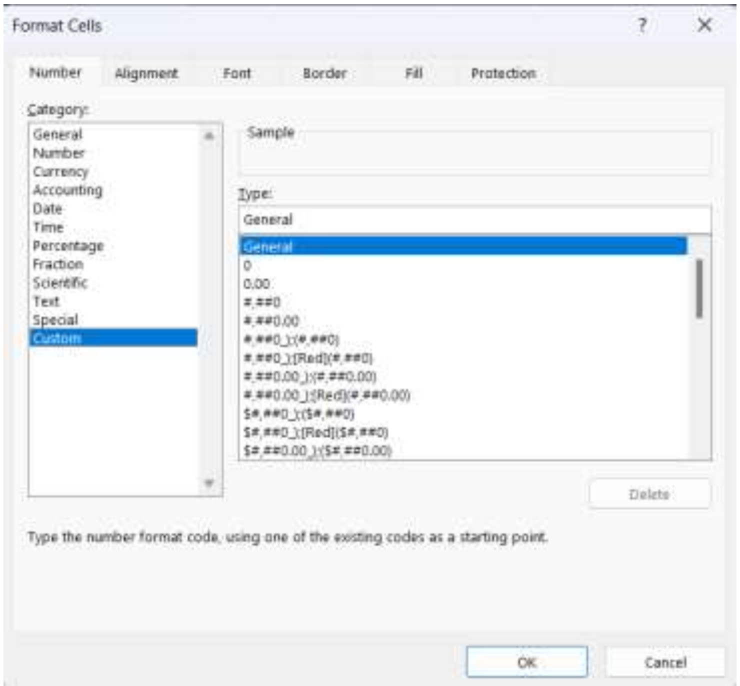


	A	B	C	D	E
1	**Shape**	**Area**	**I**	**S**	**Z**
2	W6x25	7.34	53.4	16.7	18.9
3	W6x20	5.87	41.4	13.4	14.9
4	W6x15	4.43	29.1	9.72	10.8
5	W6x16	4.74	32.1	10.2	11.7
6	W6x12	3.55	22.1	7.31	8.3
7	W6x9	2.68	16.4	5.56	6.23
8	W6x8.5	2.52	14.9	5.1	5.73

As shown in the two spreadsheets to the left, printing the data using arrays in each column and printing row by row using a matrix produced identically formatted Excel outputs.

Note: numbers in the second tab ‘Matrix’ are not formatted like in ‘Properties’ as we did not use our `num_format` parameter to enforce 2 decimals for all values.

This workbook ‘WideFlanges.xlsx’ will be saved in the same folder as the Python script. If you wanted to specify a filepath enter it when defining the workbook name:
`workbook = xlsxwriter.Workbook('C:\Users\Folder\FileName.xlsx')`



Category: Custom

Type: General

Type the number format code, using one of the existing codes as a starting point.

Fig 20.3.1 Num_Format

The `num_format` parameter follows the same naming conventions as Excel. To see a list of these options, go to the format cells option in Excel and click ‘custom’.

Example 20.3.2 Use `pandas` to extract data from the Excel file created in the previous example. Print one column and one row.

```
import pandas as pd


data = pd.read_Excel('WideFlanges.xlsx','Matrix')

print(data['Area'])

data.set_index('Shape', inplace = True)
print('\n', data.loc['W6x25'])
```

This extracts the data in the “Matrix” tab of the Excel file as a pandas dataframe.

This line allows us to call a row by its ‘Shape’ name rather than its index.

```
>> 0     7.34
>> 1     5.87
>> 2     4.43
>> 3     4.74
>> 4     3.55
>> 5     2.68
>> 6     2.52
>> Name: Area, dtype: float64
>>
>> Area      7.34
>> I        53.40
>> S        16.70
>> Z        18.90
>> Name: W6x25, dtype: float64
```

20.4 Printing Tables in Figures

There are other alternatives to produce tabular output, including placing tables below a graph in a figure or to have a standalone table as a figure, both can be executed via the `Matplotlib.pyplot` sub-library using the table function as shown in https://www.geeksforgeeks.org/matplotlib-pyplot-table-function-in-python/ and https://www.pythonpool.com/matplotlib-table/#Implementation_of_Matplotlib_table.

20.5 Printing/Displaying Special Characters

Oftentimes it is useful to print special characters, symbols, or mathematical equations in Python, either in the command window or on a plot. There are a few different methods including: Unicode and LaTeX to print symbols and special characters. Unicode is helpful when printing regular strings and LaTeX is easiest to use in the `matplotlib.pyplot` sub-library.

There are also some useful functions in Python for aiding in printing special characters, namely the ‘r’ function. This is used to print the ‘raw’ string, meaning it will cause Python to ignore escape characters.

For a full list of characters see the following links: Unicode LaTeX

Example 20.4.1 Print an array of Greek letters using Unicode, then again as a raw string.

```
# Unicode
print('\u03B1, \u03B2, \u03C0, \u00B0')

print(r'\u03B1, \u03B2, \u03C0, \u00B0')
```

```
>> α, β, π, °
>> \u03B1, \u03B2, \u03C0, \u00B0
```

Example 20.4.2 Create a plot with special characters in the axis labels and legend.

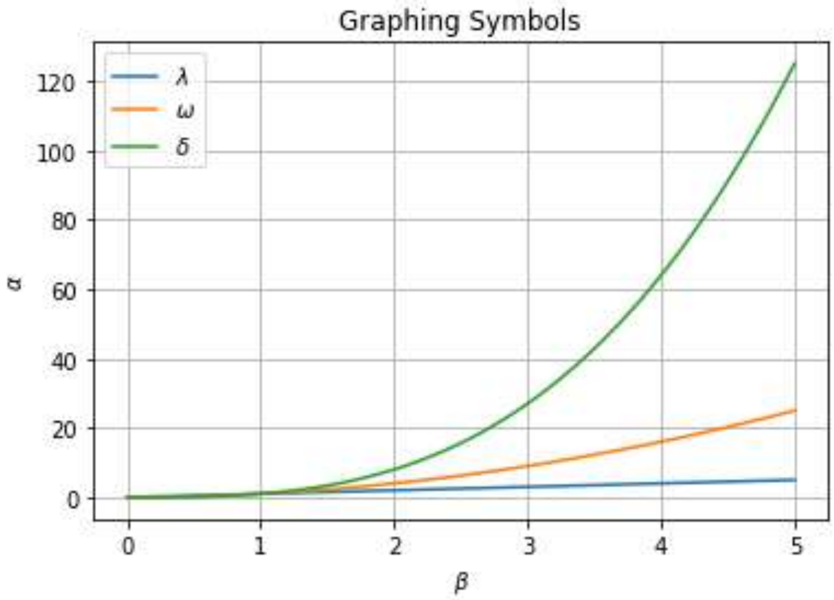
```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0,5,100)
y1 = x
y2 = x**2
y3 = x**3
y=[y1,y2,y3]

linecolor = ['red', 'blue', 'green']
labels = ['$\lambda$', '$\omega$','$\delta$']

for ii in range(0,len(y)):
    plt.plot(x,y[ii],label = labels[ii])
```

The approach to indicate when text in a string should be interpreted as LaTeX is surrounding it with `$` symbols. Printing this outside of the `matplotlib.pyplot` will not result in the same formatting.

```
plt.title('Graphing Symbols')
plt.xlabel(r'$\beta$')
plt.ylabel(r'$\alpha$')
plt.legend(prop={"size":10})
plt.grid()
plt.draw()
plt.savefig('Symbols.png')

print(labels)
```

If the LaTeX text itself is being printed instead of the special character, make the string 'raw' by inserting the letter 'r' prior to the first `$` symbol. This is necessary for \alpha and \beta as an example.

Output:



```
>> ['$\\lambda$', '$\\omega$', '$\\delta$']
```

Result of printing LaTeX outside of matplotlib.

21. User Input

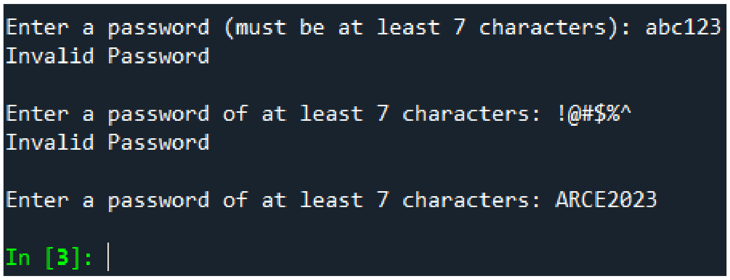
It is optimal to create code that requires very little editing should initial values change, hence why we aim to not hard code anything beyond the initial ‘user input’ or ‘givens’ section. To take this one step further, the code can be set up to ask the user to input necessary values in the command window each time it runs, so they never have to touch the baseline code. This can be helpful when sharing your program with someone who may not know how to code or if you simply do not want anyone to directly edit it.

This is done using the `input` function, which can display a prompt and take a `string` input. Example 21.1 shows how to input a string, Example 21.2 will demonstrate how to use inputs as integer values and Example 21.3 will show some more advanced uses of the `input` function.

Example 21.1 Have the user input a password of at least 7 characters. If it is less, output “invalid password” and have them reenter a password.

```
password = input('Enter a password (must be at least 7 
characters): ')

while len(password) < 7:
    print('Invalid Password')
    password=input('Enter a password of at least 7 characters:')
```

Command Window Prompts:



```
Enter a password (must be at least 7 characters): abc123
Invalid Password

Enter a password of at least 7 characters: !@#$%^
Invalid Password

Enter a password of at least 7 characters: ARCE2023

In [3]: 
```

Once the user inputs a valid 7 character password in the command window and presses ‘Enter’, a new string variable will be generated and populated with this user input and become visible in the Variable Explorer window.

Example 21.2 The following code calculates the deflection of a two-story structure. Edit this code such that E, I, La, Lb, Fa and Fb are inputted by the user and take integer values.

Original:

```
import numpy as np

E = 29000 #ksi
I = 450   #in^4
La = 100  #in
Lb = 150  #in
Fa = -3   #kips
Fb = -6   #kips

K = np.array([[12*E*I/La**3+12*E*I/Lb**3, -12*E*I/Lb**3],
             [-12*E*I/Lb**3, 12*E*I/Lb**3]])

F = np.array([[Fa],[Fb]])
u = np.linalg.inv(K)@F
print('u = ', u)
```

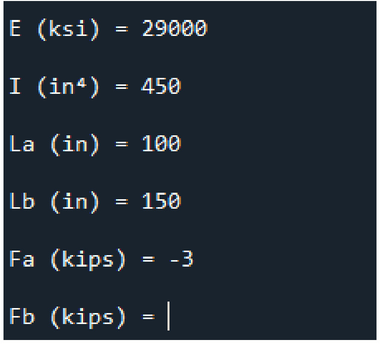
```
>> u = [[-0.05747126]
>>  [-0.18678161]]
```

Solution:

```
import numpy as np

E = int(input('E (ksi) = '))
I = int(input('I (in\u2074) = '))
La = int(input('La (in) = '))
Lb = int(input('Lb (in) = '))
Fa = int(input('Fa (kips) = '))
Fb = int(input('Fb (kips) = '))

K = np.array([[12*E*I/La**3+12*E*I/Lb**3, -12*E*I/Lb**3],
             [-12*E*I/Lb**3, 12*E*I/Lb**3]])

F = np.array([[Fa],[Fb]])
u = np.linalg.inv(K)@F
print('u = ', u)
```

The `input` function automatically converts the input into a string. To use the input as another datatype, convert it (here all inputs have been converted to integers). Otherwise, it is likely to get an error when the code is executing calculations like: `can't multiply sequence by non-int of type 'str'`.

Command Window Prompts:



```
E (ksi) = 29000

I (in⁴) = 450

La (in) = 100

Lb (in) = 150

Fa (kips) = -3

Fb (kips) = 
```

Output:

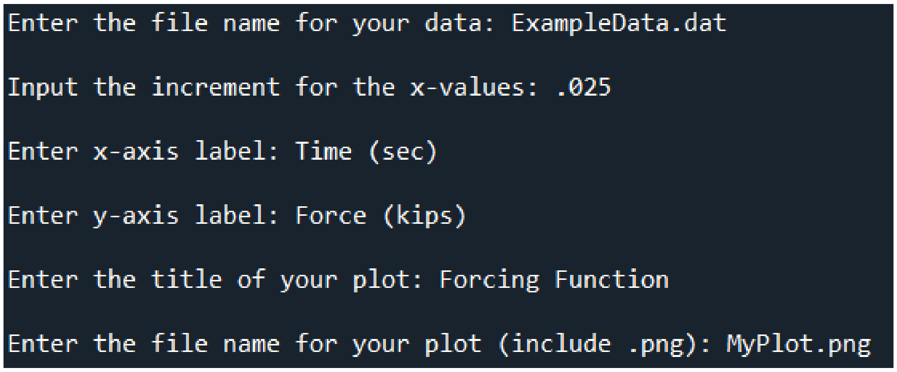
```
>> u = [[-0.05747126]
>>  [-0.18678161]]
```

Example 21.3 Create a script that allows your user to create a plot. Let them define the axis names, import data for y, set the increment value for x, and create the plot and file name. (You can find the data file used in the example in the supplementary files called ExampleData.dat.)

```
import numpy as np
import matplotlib.pyplot as plt


data = input('Enter the file name for your data: ')
increment=float(input('Input the increment for the x-values: '))

xaxis = input('Enter x-axis label: ')
yaxis = input('Enter y-axis label: ')
plotName = input('Enter the title of your plot: ')
fileName = input('Enter the file name for your plot (include
.png): ')

data = np.loadtxt(data)
timeEnd = len(data)*increment
x = np.arange(0, timeEnd, increment)

plt.figure()
plt.plot(x,data)

plt.title(plotName)
plt.xlabel(xaxis)
plt.ylabel(yaxis)
plt.grid()
plt.draw()
plt.savefig(fileName)
```

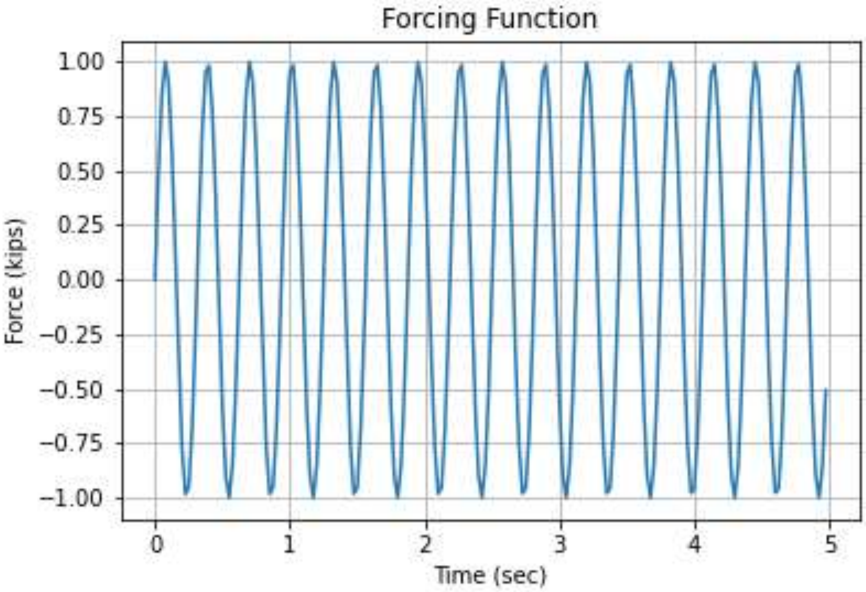
Note that the data file must be in the same folder as the Python file for this to work.

Command Window Prompts:



```
Enter the file name for your data: ExampleData.dat

Input the increment for the x-values: .025

Enter x-axis label: Time (sec)

Enter y-axis label: Force (kips)

Enter the title of your plot: Forcing Function

Enter the file name for your plot (include .png): MyPlot.png
```

Output:


Forcing Function
Force (kips)
Time (sec)
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
1
2
3
4
5

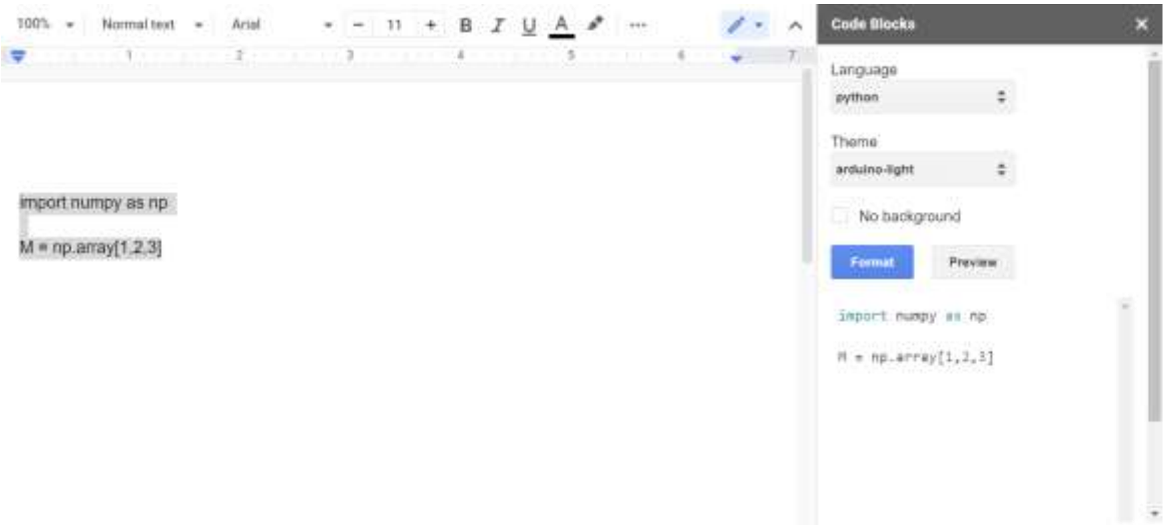
22. Script & Results Presentation in Reports

For your assignments or for professional reports in the future, you may need to transfer your script, output, and/or plots from Python onto a document. Simply copying and pasting will lose the formatting of the script, so here are the recommended ways to transfer your script and results to a document.

22.1 Transferring Script to a Word Processing Document

There is a Google extension called Code Blocks that allows you to format your script as it appears in Python in a Google Doc. Follow these steps to use Code Blocks:

1. Download the extension from Google Workspace Marketplace here: https://workspace.google.com/marketplace/app/code_blocks/100740430168
2. Copy and paste your script and/or output code from Python into Google Docs
3. In Google Docs go to **Extensions** > **Code Blocks** > **Start**
4. Select your pasted script, choose Python in the language dropdown, choose a theme, and click “Format”

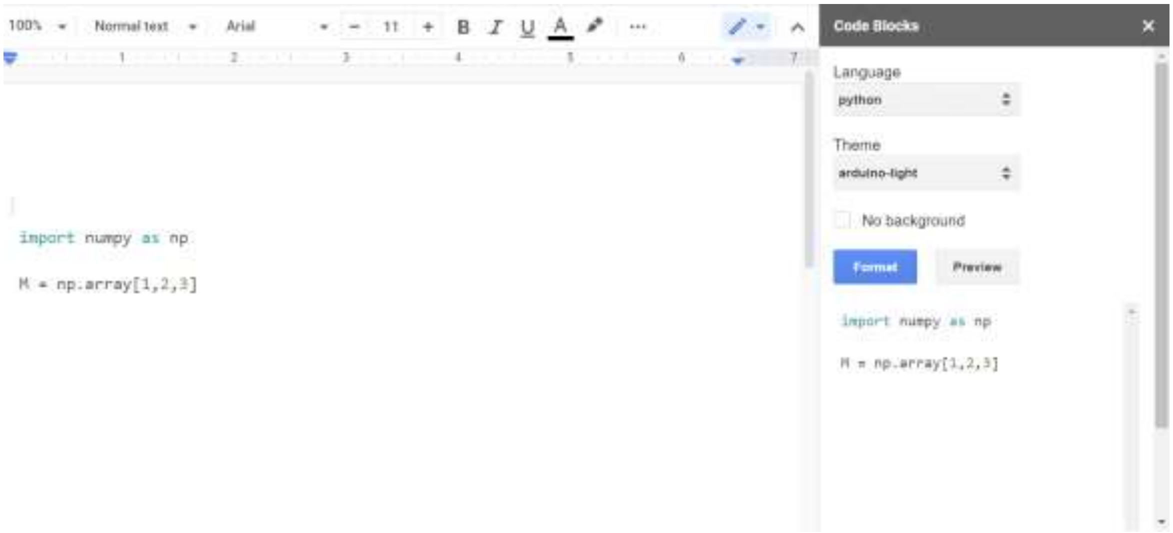


```
import numpy as np

M = np.array[1,2,3]
```

Language: python

Theme: arduino-light

```
import numpy as np

M = np.array[1,2,3]
```



```
import numpy as np

M = np.array[1,2,3]
```

Language: python

Theme: arduino-light

```
import numpy as np

M = np.array[1,2,3]
```

22.2 Exporting Plots with High Image Quality

When inserting a plot into a word document, you may find that it looks blurry, especially when you make it larger on the page. To make it look sharper, utilize `plt.savefig` with a parameter for DPI (dots per inch). A larger value of DPI will make increase the resolution.

```
plt.savefig ('name of plot', dpi = 100)
```

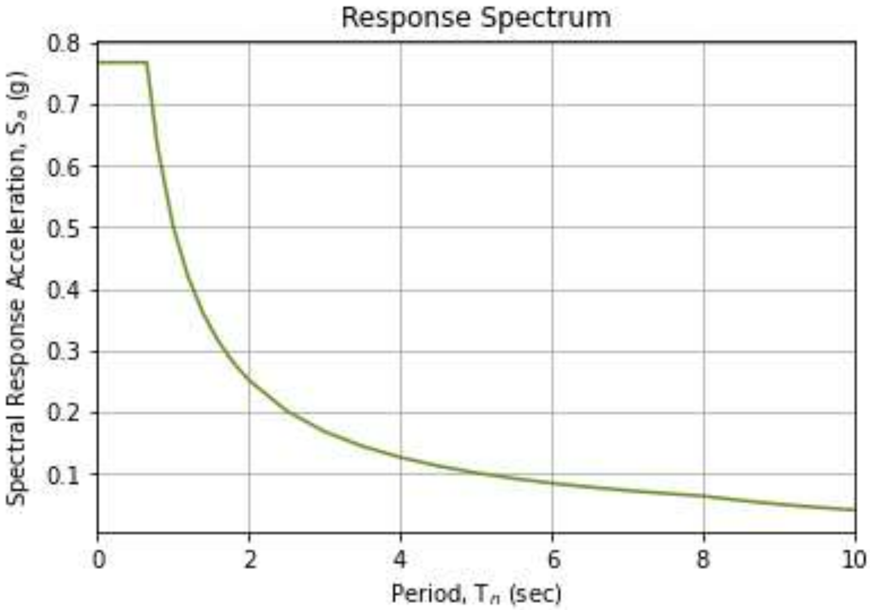


Fig. 22.2.1 *Plot with dpi not specified*

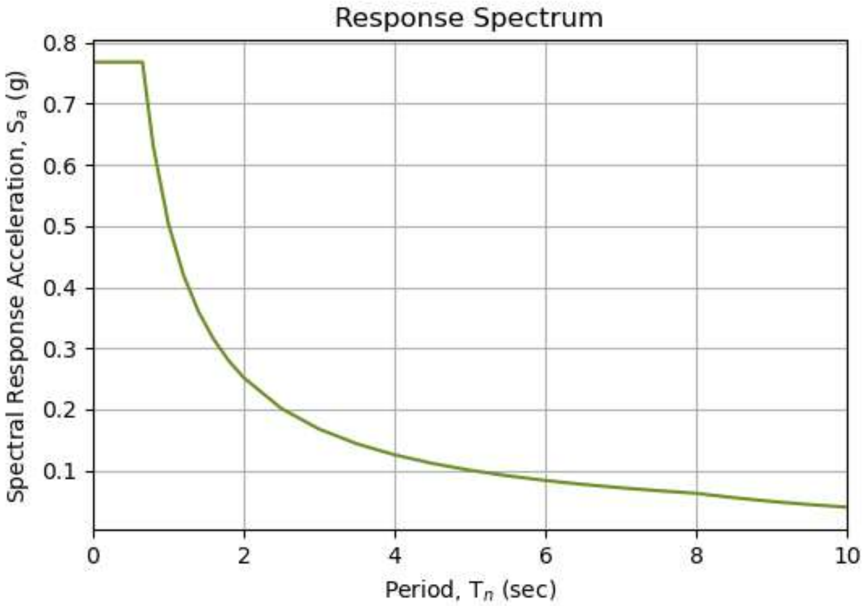


Fig 22.2.2 *Plot with dpi=100*

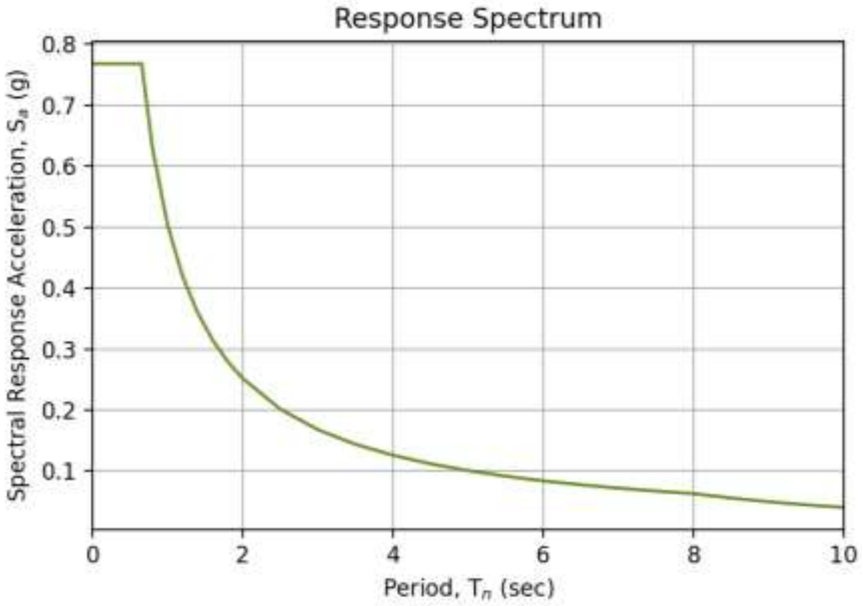


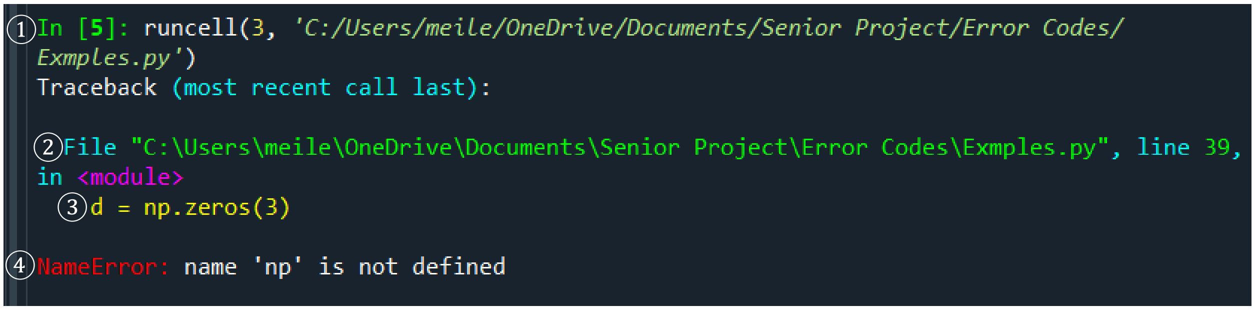
Fig 22.2.3 *Plot with dpi=200*

23. Errors & Troubleshooting

In this section we will break down error messages, explain common error types, and walk through troubleshooting methods for addressing different error types.

23.1 Deciphering Error Messages

When you receive an error message in the output window, it will most likely follow the structure shown in the picture below. Note: some Spyder color themes do not follow the same coloring convention as below, but we will be using the default color theme ("Spyder Dark") to explain error messages in this section.



```
①In [5]: runcell(3, 'C:/Users/meile/OneDrive/Documents/Senior Project/Error Codes/
Exmples.py')
Traceback (most recent call last):

②File "C:\Users\meile\OneDrive\Documents\Senior Project\Error Codes\Exmples.py", line 39,
in <module>
  ③d = np.zeros(3)

④NameError: name 'np' is not defined
```

① Each output run begins with "In [#]:" in bright green. This is followed by the line(s) you just ran as they appear in your script, or if a cell or the file was run it will say "runcell" or "runfile," along with the file path, in white and sage coloring.

② In bright blue and green it will say the file and line number on which the error occurs. Note that if individual lines or a cell was run, the line number will be relative to the selected lines that were run. Also note that the error might be fixed with a previous line of code (see **Example 23.2.1.1**). A red circle with an "x" will probably appear by the line number in your script, but not always.

③ In yellow it will show the code that is causing Python to signal an error. Again, the error might be fixed by changing other code.

④ In red is the type of error, followed by a short description in white text. The description can be confusing in some cases, so we will be focusing on how to troubleshoot based on the error type.

23.2 Common Error Message Types

In the following sub-sections, we will explain and discuss methods for fixing the following common error types:

`SyntaxError`

`NameError`

`TypeError`

`AttributeError`

`IndexError`

`ValueError`

`ImportError & ModuleNotFoundError`

For information on more error types, check out: https://www.tutorialsteacher.com/python/error-types-in-python

23.2.1 SyntaxError

What it is: A SyntaxError can be from anything regarding syntax in your code.

Common causes:

- Not having closing brackets or parentheses
- Not having a closing quotation mark
- Not using commas to separate elements in a list

How to troubleshoot: Check your syntax carefully line by line. Whenever the cursor is just to the right of a bracket or parenthesis, Python will highlight the matching parentheses or brackets in green; if it is missing an opening or closing one, it will be highlighted in red. These errors can be difficult to catch when you have been working on your code for a while, so another set of eyes from someone else may be helpful. Additionally, check the proper syntax that is needed for what you are trying to perform.

Example 23.2.1.1:

```
import numpy as np
K = np.array([[ 14.094, -1.5660, -234.90],
              [-1.5660,  1.5660, -78.300],
              [-234.90, -78.300,  15660],
              [ 78.300, -78.300,  2610.0]])

M = np.array([[ 5.0941, -7.7660],
              [-1.0960,  3.8860],
              [-93.400, -71.340]]
KM = K@M
```

```
>> runcell(1, 'C:/Users\Exmples.py')
File "C:\Users\ Exmples.py", line 10
  KM = K@M
  ^
SyntaxError: invalid syntax
```

In this example, a closing parenthesis was left out of the M array on line 9, which resulted in a `SyntaxError`. Notice how Python is calling out line 10, even though this error would need to be corrected on line 9.

23.2.2 NameError

What it is: A NameError will appear when you try to reference a variable that has not been defined above that line of code. This can also appear if you have not imported a library that you are using.

Common causes:

- Forgetting to define a variable that is being used
- Not running a line of code that defines a variable before running other lines that use that variable
- Not importing a library that is being used
- Not renaming the library when it is being called out by a different name (see **Example 23.2.2.1**)

How to troubleshoot: Make sure you define the variable in a line of code above where it is being referenced and that that line has been run. You can also use the variable explorer to see what variables exist (i.e., that have been defined AND run) in your code.

Example 23.2.2.1

```
import numpy
n = np.array([1,2,3])
```

```
>> runfile('C:/Users/meile/untitled1.py')
Traceback (most recent call last):
  File "C:\Users\meile\Documents\ARCE 354\Lab 4\untitled1.py",
line 2, in <module>
   n = np.array([1,2,3])
NameError: name 'np' is not defined
```

23.2.3 TypeError

What it is: TypeErrors occur when a function or operation you are using is being applied to an object of the wrong data type (see Section 6 for more information on data types).

Common causes:

- Putting in an object (e.g., string, array) of the wrong data type when using a library function
- Putting a counter on an uniterable object in a for loop
- Using the print command

- Calling out a function with a different number of arguments than the function was set with

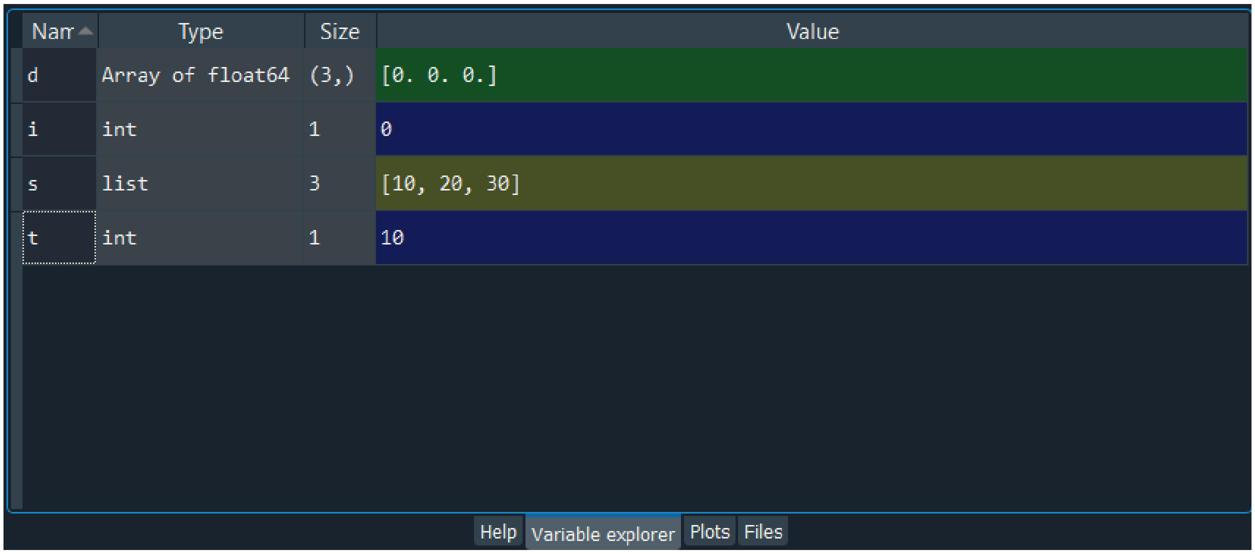
How to troubleshoot: The variable explorer will be a useful tool because it shows the data type of your variables. Check the data type that should be used with what you are performing (see Section 15 for information on libraries), and make sure your variable is the correct data type (see Section 6 for information on data types).

Example 23.2.3.1:

```
1 Import numpy as np
2 s = [10,20,30]      #steps
3 t = 10              #in/step
4 d = np.zeros(3)
5 for ii in range(0,len(s)):
6     d[ii] = t[ii]*s[ii]
7 print(d)
```

```
>> runcell(3, 'C:/Users/Exmples.py')
Traceback (most recent call last):
  File "C:\Users\Exmples.py", line 42, in <module>
    d[ii] = t[ii]*s[ii]
TypeError: 'int' object is not subscriptable
```

This TypeError tells us that we are likely using a variable incorrectly in line 6, in regard to its data type. To troubleshoot this, we should look at our variable explorer.



Nam	Type	Size	Value
d	Array of float64	(3,)	[0. 0. 0.]
i	int	1	0
s	list	3	[10, 20, 30]
t	int	1	10

In line 6 we are putting an index counter `i` to use with variables `d`, `t`, and `s`, so those variables should each be a list with multiple elements (specifically 3 elements, since the length of `s` was used to define the index counter `i`). As we can see in the variable explorer, however, `t` is an integer with one element, which is why it is not “subscriptable,” or countable.

To fix our code, we can either remove the counter on line 6, or make `t` a list of size 3 on line 3.

23.2.4 AttributeError

What it is: An AttributeError occurs when you “attempt to call an attribute of an object whose type does not support that method.” This type of error is similar to TypeError in that the correct parameter and data types must be used for what is trying to be performed.

Common causes:

- Using `.append( )` on an integer or string instead of a list

How to troubleshoot: Check the parameters for the function you are trying to perform. (See Section 15 for information on libraries and Section 6. for information on data types.)

Example 23.2.4.1

```
organs = “heart”
organs.append(“kidney”)
```

```
>> organs = "heart"
organs.append("kidney")
Traceback (most recent call last):
  File "<ipython-input-40-5709782cbd3c>", line 2, in <module>
    organs.append("kidney")
AttributeError: 'str' object has no attribute
'append'AttributeError: 'str' object has no attribute 'append'
```

This error is saying that append cannot be performed on a string, so to fix this error, you need to check the parameters of using append and adjust accordingly. `organs` can easily be changed to a list by adding brackets as shown below so the error does not occur.

```
organs = [“heart”]
organs.append(“kidney”)
```

23.2.5 IndexError

What it is: An IndexError occurs when you call an index that does not exist.

How to troubleshoot: Use the variable explorer or `np.shape( )` in the command window to check the size of the object (i.e., list, array, matrix) of whose index you are referring to, and make sure you are calling an index that exists in the object. Note that in Python an object’s first index is 0, not 1, so the index of the nth element in a list is n-1.

Example 23.2.5.1

```
L1=[1,2,3]
L1[3]
```

```
>> IndexError: list index out of range
```

The index L1[3] does not exist because the 3rd element in L1 has the index L1[2].

23.2.6 ValueError

What it is: A ValueError occurs when a function or operation has a resultant value that does not exist or when a mathematical operation cannot be performed.

Common causes:

- Performing an operation that gives a result that does not exist in the mathematical domain (e.g., taking the square root of a negative number)
- Multiplying matrices of the wrong dimensions
- Plotting arrays that are not of the same size

How to troubleshoot: Variable explorer can be used to keep track of what numbers are being used in operations and what size your arrays and matrices are. You may also need to brush up on math concepts if you are not sure why an operation cannot be performed.

23.2.7 ImportError and ModuleNotFoundError

What it is: An ImportError can occur when dealing with functions called from other files. Recall that you must use the format "`from [file name] import [function name] as [new function name]`". This error typically comes up when you put the `[function name]` as a different name.

A similar error to this is ModuleNotFoundError, which occurs when the `[file name]` is put as a different name, or when the files are saved in different folders. (See Section 10 for more information on functions.)

How to troubleshoot: Simply check that you are referring to the file name and function name correctly when calling a function from another file, and check that the files are saved in the same folder.

23.3 General Troubleshooting Tips

Here are some other general tips to help you troubleshoot:

Use the command window to hardcode. By hardcode, we mean break down your code and directly put in the value that is meant to be used. For example, replace a counter in a for loop directly with the value or variable that will pass through the loop, and repeat this for each value or variable.

Sketch out your logic. It can be helpful to literally draw out with a pencil and paper what your code is performing. This can be a flow chart, table, or anything else that helps you through your logic. This is especially helpful when programming `for` loops. An example of how to implement a pen and paper process to develop code is shown in Example 23.3.1.

Example 23.3.1 Develop a `for` loop with nested `if-elif-else` statements to plot the response spectrum (S_a vs. T_n) for any given range of T_n. Site specific seismic parameters S_{ds}, S_{d1}=0.286, and T_l are provided from the ATC Hazards Tool. Use the ASCE 7-16 parameters outlined in code section 11.4.6. (Note: T in ASCE was replaced with T_n in this example.)

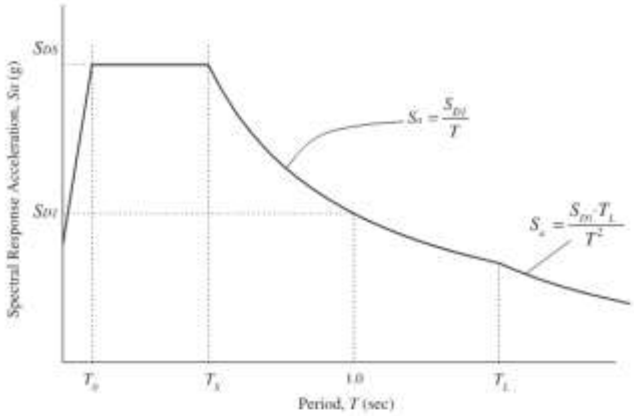
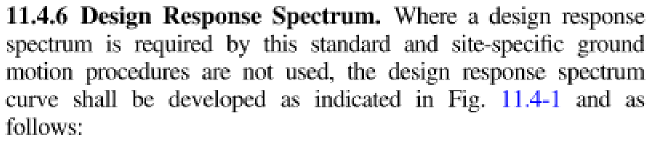
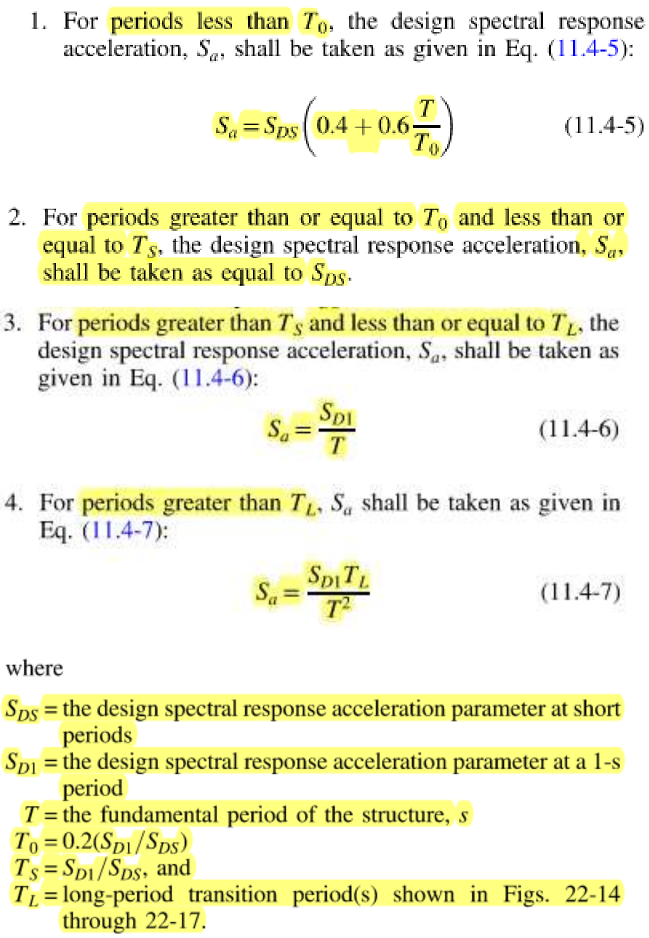


FIGURE 11.4-1 Design Response Spectrum



11.4.6 Design Response Spectrum. Where a design response spectrum is required by this standard and site-specific ground motion procedures are not used, the design response spectrum curve shall be developed as indicated in Fig. 11.4-1 and as follows:



1. For periods less than T_0, the design spectral response acceleration, S_a, shall be taken as given in Eq. (11.4-5):

$$S_a = S_{DS}\left(0.4 + 0.6\frac{T}{T_0}\right) \tag{11.4-5}$$

2. For periods greater than or equal to T_0 and less than or equal to T_S, the design spectral response acceleration, S_a, shall be taken as equal to S_{DS}.

3. For periods greater than T_S and less than or equal to T_L, the design spectral response acceleration, S_a, shall be taken as given in Eq. (11.4-6):

$$S_a = \frac{S_{D1}}{T} \tag{11.4-6}$$

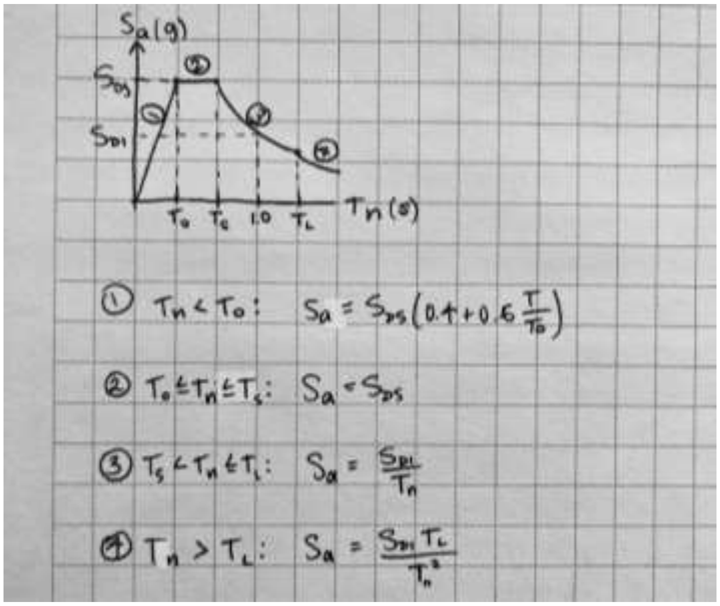
4. For periods greater than T_L, S_a shall be taken as given in Eq. (11.4-7):

$$S_a = \frac{S_{D1}T_L}{T^2} \tag{11.4-7}$$

where

S_{DS} = the design spectral response acceleration parameter at short periods
S_{D1} = the design spectral response acceleration parameter at a 1-s period
T = the fundamental period of the structure, s
$T_0 = 0.2(S_{D1}/S_{DS})$
$T_S = S_{D1}/S_{DS}$, and
T_L = long-period transition period(s) shown in Figs. 22-14 through 22-17.

Step 1) Sketch out the four scenarios of T_n and S_a values, and then type them in Python.



① $T_n < T_0$: $S_a = S_{DS}\left(0.4+0.6\frac{T}{T_0}\right)$

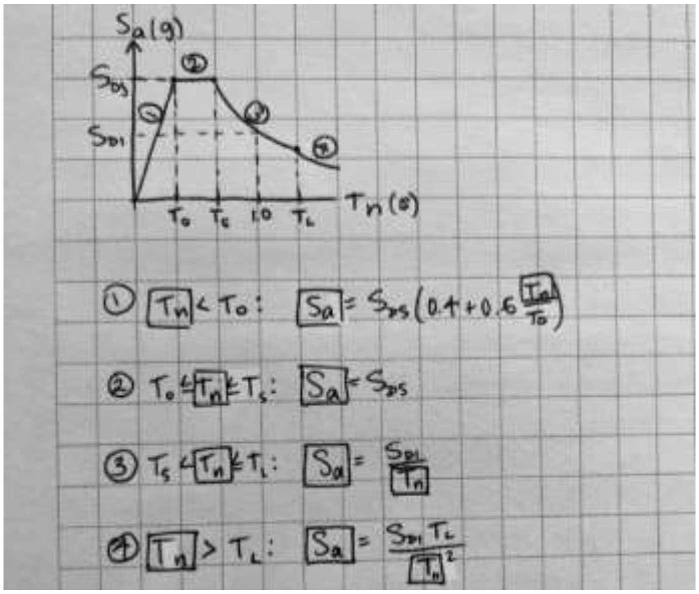
② $T_0 \le T_n \le T_s$: $S_a = S_{DS}$

③ $T_s < T_n \le T_L$: $S_a = \frac{S_{D1}}{T_n}$

④ $T_n > T_L$: $S_a = \frac{S_{D1}T_L}{T_n^2}$

```
Tn < To:
        Sa = Sds*(0.4 + 0.6*Tn/To)
Tn >= To and Tn <= Ts:
        Sa = Sds
Tn > Ts and Tn <= Tl:
        Sa = Sd1/Tn
Tn > Tl:
        Sa = Sd1*Tl/Tn**2
```

Step 2) Determine what variables are not constant (T_n, S_a), and add counters to those variables. The constant variables (S_{ds}, S_{d1}, T_l, T_o, T_s) would be defined at the start of your code as inputs.



```
Tn[ii] < To:
        Sa[ii] = Sds*(0.4 + 0.6*Tn[ii]/To)
Tn[ii] >= To and Tn[ii] <= Ts:
        Sa[ii] = Sds
Tn[ii] > Ts and Tn[ii] <= Tl:
        Sa[ii] = Sd1/Tn[ii]
Tn[ii] > Tl:
        Sa[ii] = Sd1*Tl/Tn[ii]**2
```

Step 3) Determine what kind of loop you will be using, determine the range of your counter, and place your statements in that loop.

```
Sa = np.zeros(len(Tn))

for ii in range(0,len(Tn)):
    if Tn[ii] < To:
        Sa[ii] = Sds*(0.4 + 0.6*Tn[ii]/To)
    elif Tn[ii] >= To and Tn[ii] <= Ts:
        Sa[ii] = Sds
    elif Tn[ii] > Ts and Tn[ii] <= Tl:
        Sa[ii] = Sd1/Tn[ii]
    else:
        Sa[ii] = Sd1*Tl/Tn[ii]**2
```

24. Where to Get Help & Additional Resources

If you are seeking information beyond the scope of this help manual, there are many resources on the internet, including a few that we recommend and have outlined below. For general information refer to the Python general documentation at https://docs.python.org/3/ and tutorial at https://docs.python.org/3/tutorial/index.html.

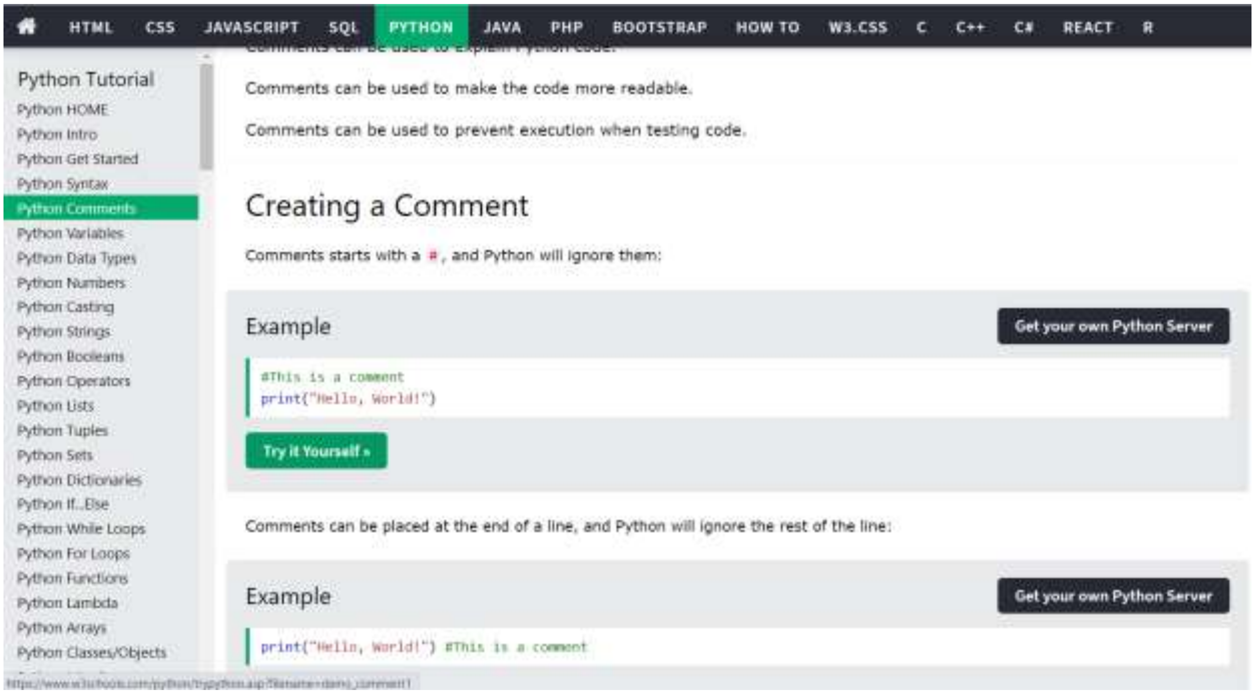
24.1 W3schools

Access W3schools' by going to their website at: https://www.w3schools.com/python/default.asp

W3schools is a beginner-friendly website that teaches you how to use different coding languages. Under the Python tab, there is lots of information and modules, so we chose a few sections to highlight that will probably be the most relevant and helpful to you. However, if you want to extend your Python or coding knowledge, there is much more to explore on W3schools. You can even take a quiz to become certified in a language!

Python Tutorials

In the side bar, there are many topics under "Python Tutorial" that the website leads you through. They give a basic overview and have multiple "Try It Yourself" features for each topic.



Comments can be used to make the code more readable.

Comments can be used to prevent execution when testing code.

Creating a Comment

Comments starts with a `#`, and Python will ignore them:

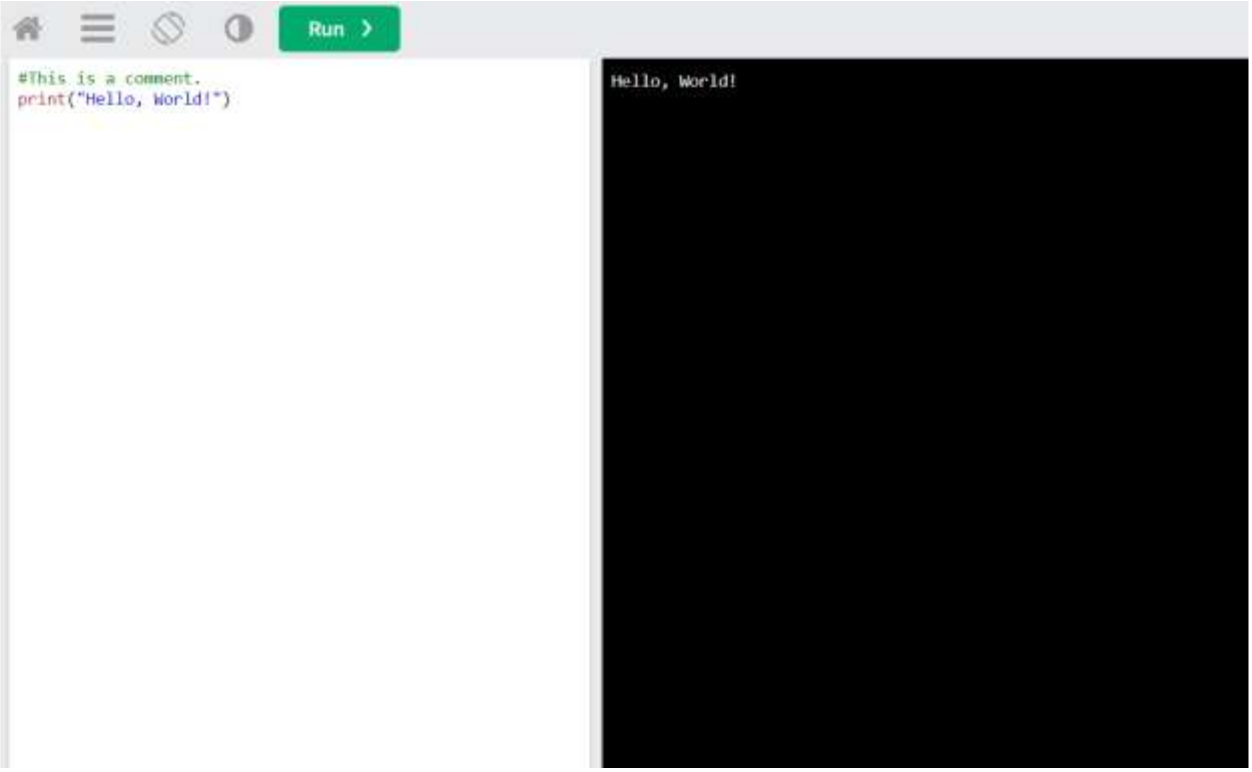
Example

```
#This is a comment
print("Hello, World!")
```

Comments can be placed at the end of a line, and Python will ignore the rest of the line:

Example

```
print("Hello, World!") #This is a comment
```

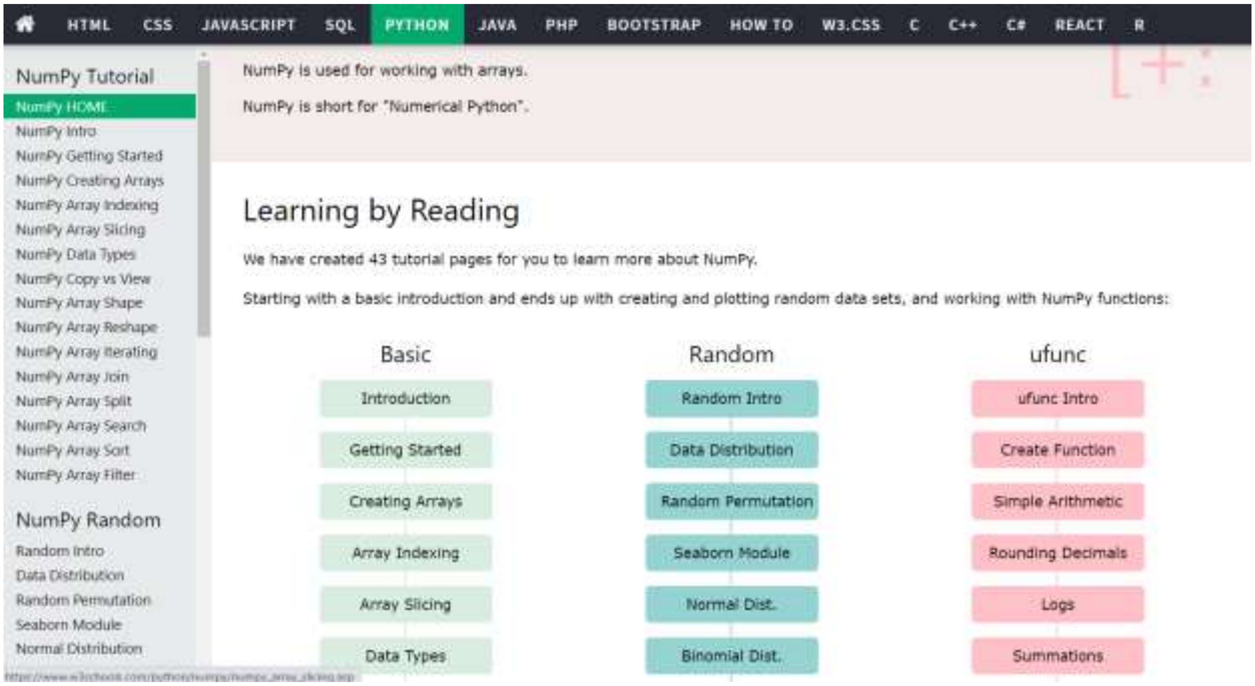


```python
#This is a comment.
print("Hello, World!")
```

```
Hello, World!
```

Python Library Modules

In the side bar under “Python Modules,” you will see some Python library modules that will bring you to a page like the one shown below for NumPy. In these modules you can find tutorials and exercises specifically for utilizing the library.



NumPy is used for working with arrays.

NumPy is short for "Numerical Python".

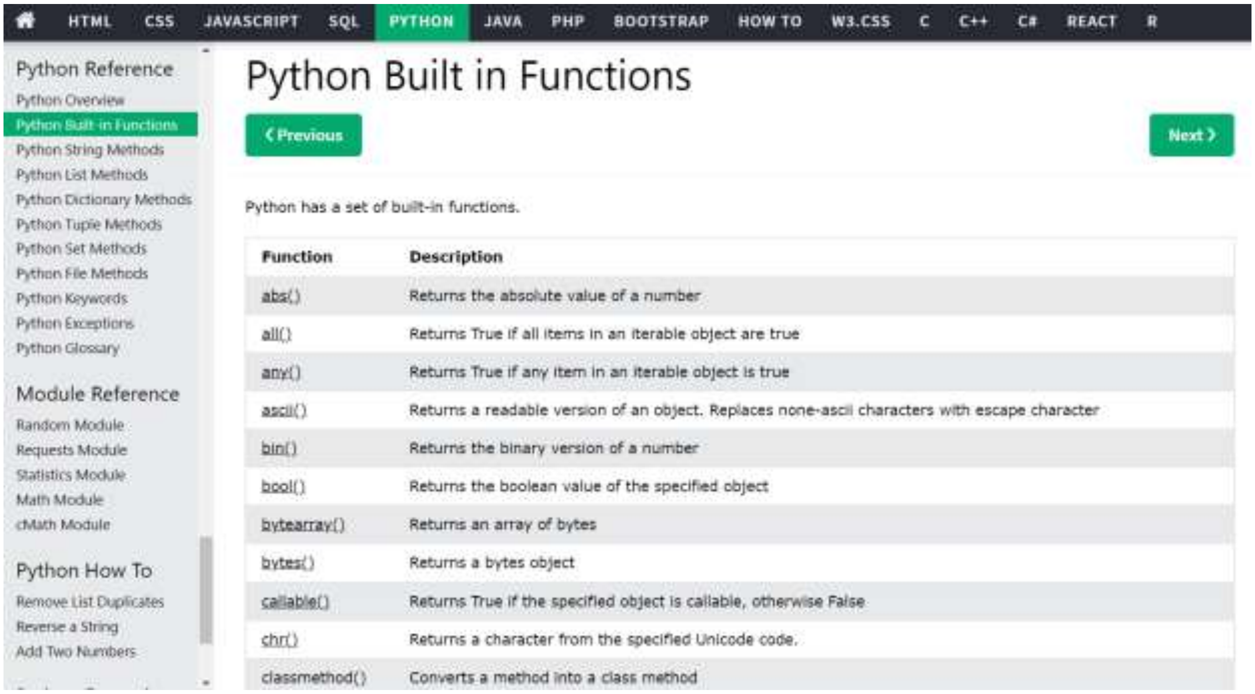
Learning by Reading

We have created 43 tutorial pages for you to learn more about NumPy.

Starting with a basic introduction and ends up with creating and plotting random data sets, and working with NumPy functions:

Python Built-in Functions

In the side bar there is a tab called “Python Built-in Functions.” Here, you can see a list of Python’s built-in functions.



Python Built in Functions

Python has a set of built-in functions.

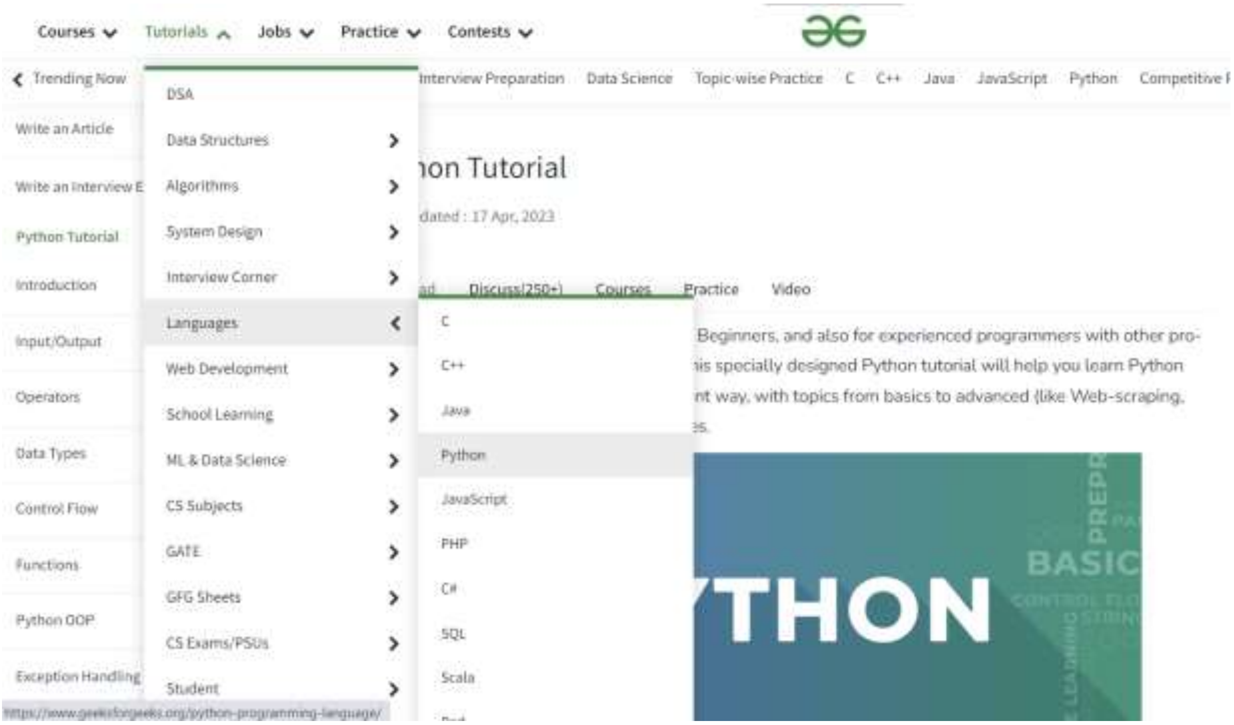
Function	Description
abs()	Returns the absolute value of a number
all()	Returns True if all items in an iterable object are true
any()	Returns True if any item in an iterable object is true
ascii()	Returns a readable version of an object. Replaces none-ascii characters with escape character
bin()	Returns the binary version of a number
bool()	Returns the boolean value of the specified object
bytearray()	Returns an array of bytes
bytes()	Returns a bytes object
callable()	Returns True if the specified object is callable, otherwise False
chr()	Returns a character from the specified Unicode code.
classmethod()	Converts a method into a class method

24.2 GeeksforGeeks

Access GeeksforGeeks by going to their website at: https://www.geeksforgeeks.org/.

Like W3schools, GeeksforGeeks is a great resource that teaches users how to code in Python, broken down by topic, using examples and written explanation. The main difference is their presentation of information—GeeksforGeeks has more written explanation and does not allow you to run example code without signing into an account (which is free). GeeksforGeeks also has many courses and tutorials on not just different programming languages, but other programming-related topics and professional skills, like algorithms and interview help. It is a great resource to expand your programming knowledge for a future career in computer science.

After going to the main Python page under **Tutorials** > **Languages** > **Python**, there is a sidebar with general topics to navigate, but you can better see all the topics by scrolling to the bottom of the main page.



hon Tutorial

dated : 17 Apr, 2023

Beginners, and also for experienced programmers with other pro-
his specially designed Python tutorial will help you learn Python
nt way, with topics from basics to advanced (like Web-scraping,
es.

Getting Started with Python Tutorial

Here are the important topics that come under Python. After completing all the important topics, you'll have a basic understanding of the Python programming language:-

Python Basics

- Python language introduction
- Python 3 basics
- Python The new generation language
- Important difference between python 2.x and python 3.x with example
- Keywords in Python | Set 1, Set 2
- Namespaces and Scope in Python
- Statement, Indentation and Comment in Python
- Structuring Python Programs
- How to check if a string is a valid keyword in Python?
- How to assign values to variables in Python and other languages
- How to print without newline in Python?
- Decision making
- Basic calculator program using Python
- Python Language advantages and applications

Django Framework

- **Django Tutorial**
- Django Basics
- Django Introduction and Installation
- Django Forms
- Views In Django
- Django Models
- Django Templates
- ToDo webapp using Django
- Django News App
- Weather app using Django

Data Analysis

- Data visualization using Bokeh
- Exploratory Data Analysis in Python
- Data visualization with different Charts in Python

24.3 Library Websites

The different libraries you use in Python have their own websites with information on how to use their library functions and perform certain tasks.

NumPy: https://numpy.org/doc/stable/user/index.html#user

SciPy: https://docs.scipy.org/doc/scipy/tutorial/index.html#user-guide

SymPy: https://docs.sympy.org/latest/index.html

Pandas: https://pandas.pydata.org/docs/user_guide/index.html

Matplotlib: https://matplotlib.org/stable/index.html

Math: https://docs.python.org/3/library/math.html

24.4 Quick Sheets

There have been quick sheets to help with learning and using common Python libraries. A selection of these pertaining to the libraries covered in this manual are listed below.

NumPy:

https://cdn.intellipaat.com/mediaFiles/2018/12/Python-NumPy-Cheat-Sheet-.pdf
https://images.datacamp.com/image/upload/v1676302459/Marketing/Blog/Numpy_Cheat_Sheet.pdf
http://datasciencefree.com/numpy.pdf
https://mathesaurus.sourceforge.net/matlab-numpy.html

SciPy:

https://images.datacamp.com/image/upload/v1676303474/Marketing/Blog/SciPy_Cheat_Sheet.pdf

Pandas:

https://intellipaat.com/mediaFiles/2018/12/Python-Pandas-Cheat-Sheet.png
https://images.datacamp.com/image/upload/v1676302827/Marketing/Blog/Data_Wrangling_Cheat_Sheet.pdf
http://datasciencefree.com/pandas.pdf

Matplotlib:

https://images.datacamp.com/image/upload/v1676360378/Marketing/Blog/Matplotlib_Cheat_Sheet.pdf

24.5 YouTube Video Tutorials

A few suggested video tutorials on Python are provided below, other video links can be found with the associated topic earlier in specific sections of the manual.

General coding: https://www.youtube.com/watch?v=N4mEzFDjqtA

Numpy: https://www.youtube.com/watch?v=GB9ByFAIAH4

Matplotlib: https://www.youtube.com/watch?v=qErBw-R2Ybk

24.6 Cloud-Based Programming Tool: Replit

Replit is a website that allows you to code if you are having issues with or do not have access to a Python reader application (i.e., Spyder) on a local device.

Access Replit by going to their website: https://replit.com/~