Walker Lab Coding Summer School

Last updated: July 5, 2023

Contents

Part I

Introduction to Programming

Chapter 1

Best Practices

This Code Summer School will cover the basics of Python and C++ as well as some best practices and good habits to develop early as programmers. Whether you intend to pursue programming professionally or merely keep it as a skill in your back pocket, these lessons will hopefully serve as a good foundation. I anticipate each day will be a couple of hours of lecture and practice, with Q&A mixed in as we go. I plan to keep it fairly informal, so questions and interruptions are fine and welcome. Also, feedback is much appreciated as it will help to refine this Summer School for following years.

"Programming" is such a hugely expansive term, covering a staggering number of techniques, languages, processes, hardware configurations, purposes, applications, scales, and so forth.

So what is programming, really?

The short version is that programming is a way of making a computer (another broad term!) perform tasks. These tasks can be as simple as the addition of two numbers, or as complex as calculating the excited state energy of a fluorescent nucleotide as it moves through a solution of water and sodium chloride ions. As the tasks become more complex, the code becomes more complex as well. Small tasks may be manageable with tiny scripts or single-file programs, while larger ones may require the inclusion of other tools, multiple files, and more complicated design principles.

For the purposes of this Summer School, we'll mostly be sticking to the introductory/entry-level stuff, but remember that the actual limits of your programs are only in the capacity of your hardware and the breadth of your imagination.

There are some rules to programming that should really be learned early on, if only because knowing them early will make the rest considerably easier. For many experienced programmers, the actual writing of code is a smaller portion of the overall process than one might expect.

1.1 Version Control, Git, and GitHub

Version control is a critically important habit to develop early on. In the simplest form, version control provides a means of keeping track of the changes you've made to your code as you go, as well as providing information about who made which changes in a collaborative project. More complicated version control can lead to things like software written for different types of hardware, or for different scales of calculations, etc. Many of us have used version control at some point in our educations, whether we realized it or not. How many times have you had a document labelled "Document_v2_Final_FINAL_FINAL_v3"? That is an example (albeit not the greatest) of version control. Keeping track of changes made and versions of your code can be a way to safeguard your work against potential losses and help you track down the source of bugs and errors that may arise.

The most commonly used tool for version control is `git`, with related websites GitHub, GitLab, and Bitbucket. For the purposes of this workshop, we'll focus on using GitHub because it's free and most (if not all) of us already have accounts there.

There are a few steps to do first if you've never used `git` on your current computer before. These will configure your computer for use with your GitHub account. If you use multiple computers (including working from a HPC/Supercomputer), these steps will need to be completed for each computer you use.

Configure your local machine with an SSH-key. This will allow your computer to connect to other **trusted** computers that you've previously designated as such, and this includes your Github account.

```
cd $HOME
ssh-keygen
```

will give the following response/prompt:

```
Generating public/private rsa key pair
Enter file in which to save the key (/home/username/.ssh/id_rsa):
```

If the file already exists, choose a new filename such as `git_rsa` or something you'll recognize.

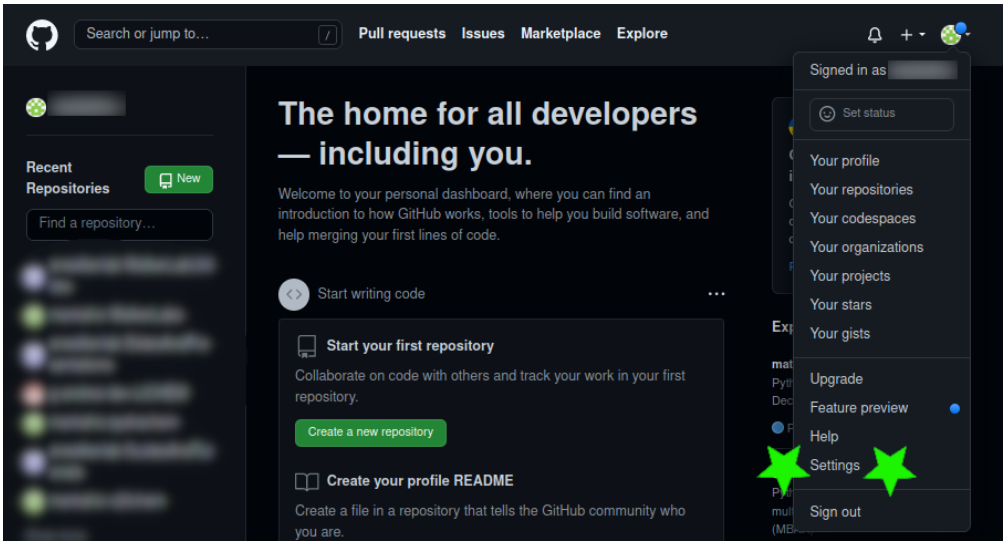
```
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
```

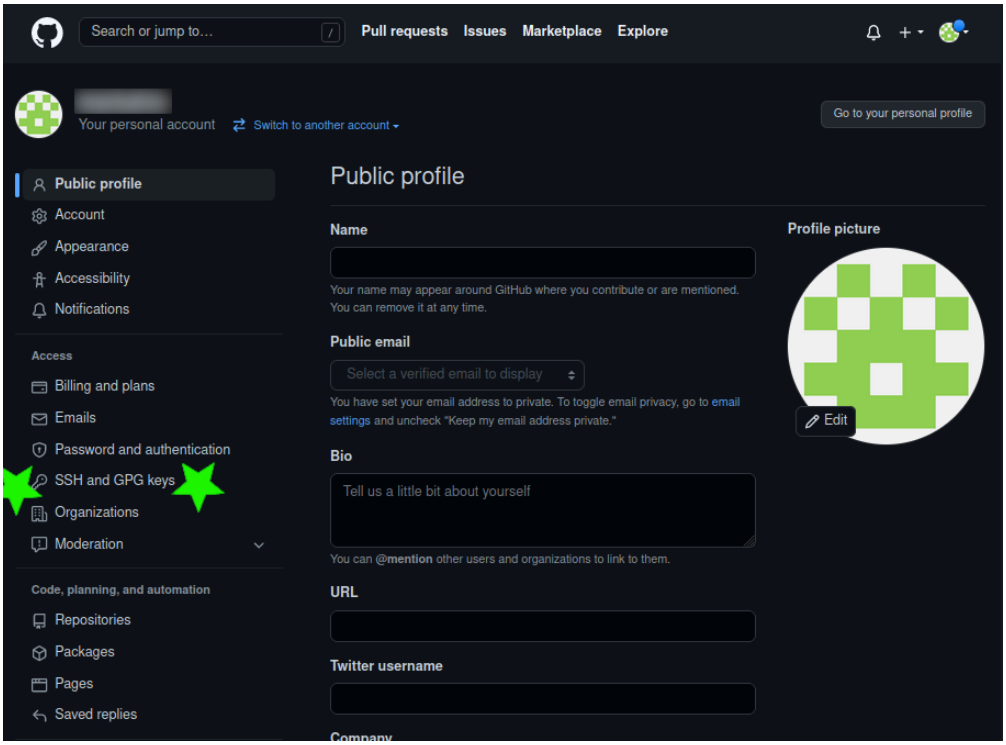
You can set a passphrase if you want, but keep in mind you'll be entering it every single time you upload changes of your code to GitHub. Some people choose not to have a passphrase for this particular aspect of their work, others do.

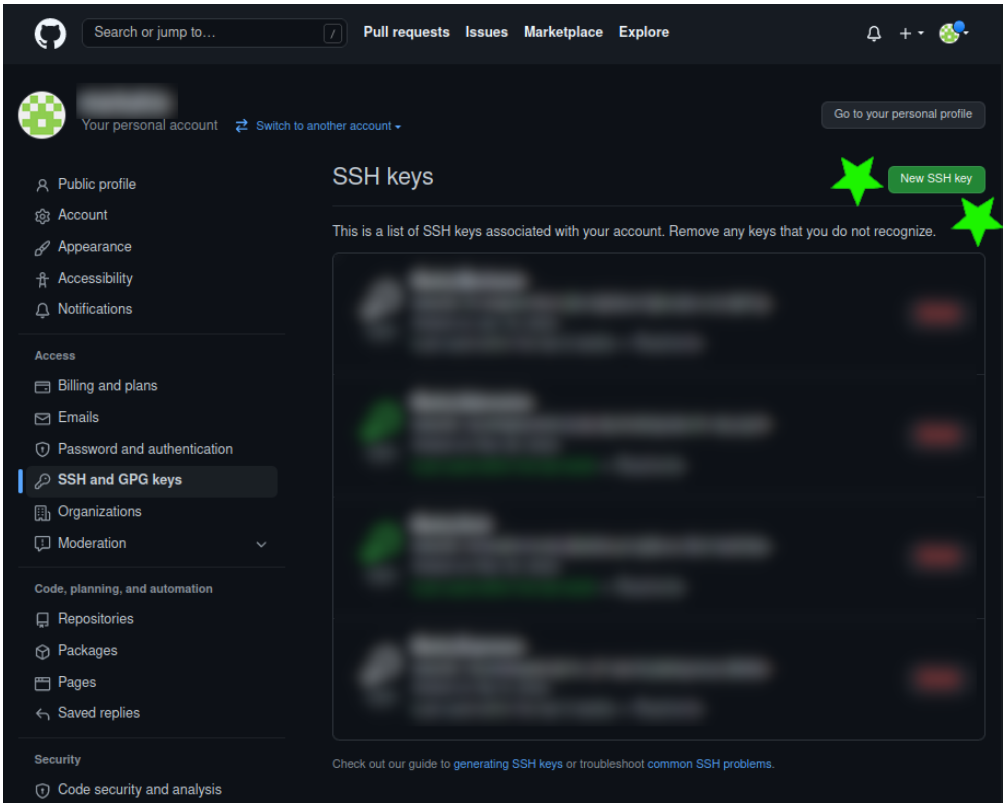
```
Your identification has been saved in /home/username/.ssh/git_rsa
Your public key has been saved in /home/username/.ssh/git_rsa.pub
The key fingerprint is:
SHA256:8U9t+r+SwCi8Xe8uu3HCjbHa7WU51A9pArzm9+F+esk username@Computer
The key's randomart image is:
+---[RSA 3072]----+
|                 |
|       .         |
|        +  . +   |
|         =. =. ..|
|      . S =*o.=..|
|       = .oBo+..o|
|        ..o+*o.*.|
|       . o.+o*D .|
|           +@Bo*o|
+----[SHA256]-----+
```

Add the SSH key to your GitHub Account to allow your computer to access it.

Go to your account settings page and click on `SSH and GPG keys`, then click on `New SSH key`.

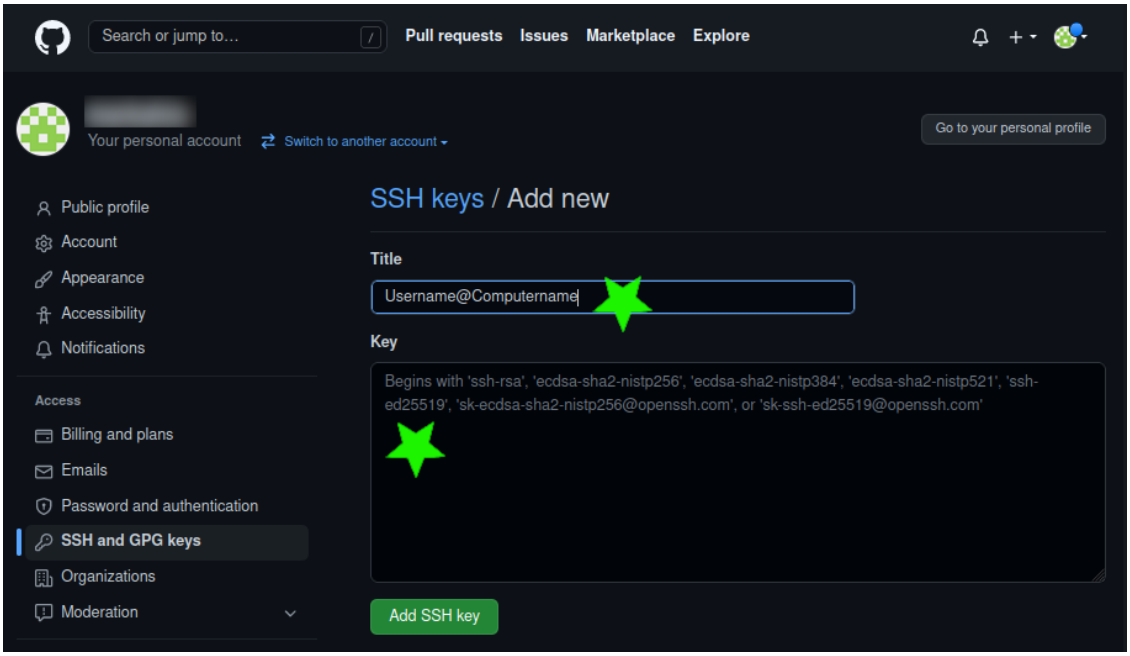

Search or jump to…
Pull requests
Issues
Marketplace
Explore
Signed in as
Set status
Your profile
Your repositories
Your codespaces
Your organizations
Your projects
Your stars
Your gists
Upgrade
Feature preview
Help
Settings
Sign out
Recent Repositories
New
Find a repository…
The home for all developers — including you.
Welcome to your personal dashboard, where you can find an introduction to how GitHub works, tools to help you build software, and help merging your first lines of code.
Start writing code
Start your first repository
Collaborate on code with others and track your work in your first repository.
Create a new repository
Create your profile README
Create a file in a repository that tells the GitHub community who you are.


Search or jump to…
Pull requests
Issues
Marketplace
Explore
Your personal account
Switch to another account
Go to your personal profile
Public profile
Account
Appearance
Accessibility
Notifications
Access
Billing and plans
Emails
Password and authentication
SSH and GPG keys
Organizations
Moderation
Code, planning, and automation
Repositories
Packages
Pages
Saved replies
Public profile
Name
Your name may appear around GitHub where you contribute or are mentioned. You can remove it at any time.
Public email
Select a verified email to display
You have set your email address to private. To toggle email privacy, go to email settings and uncheck "Keep my email address private."
Bio
Tell us a little bit about yourself
You can @mention other users and organizations to link to them.
URL
Twitter username
Company
Profile picture
Edit



You'll need some text out of a file generated by the `ssh-keygen` step. If you named the keyfile `git_rsa`, the process will have also produced a file called `git_rsa.pub`, which is the "public key" corresponding to your computer's private key. In simpler terms, the public key is like a "secret question" that the other computer can ask, that only your computer with its private "secret answer" can properly respond to, so both computers know the other is trusted with this information transfer.

Open the `git_rsa.pub` file and copy all the text into the field shown on the GitHub website here.



You'll also need to configure `git` on your computer as well. Assuming you have `git` already installed, you can begin with setting some of the initial variables.

You can configure individual repositories (projects) with these settings, or you can configure `git` globally to set your defaults. For now, we'll assume that you only have one GitHub account to manage on your computer.

```
git config --global user.name "Firstname Lastname"
git config --global user.email "username@emailserver.com"
git config --global user.user "github_username"
```

This next command may not mean too much right now, but it's useful to have right off the bat to keep things clean later on.

```
git config --global core.excludesFile '~/.gitignore'
touch ~/.gitignore
```

This tells git to ignore anything listed in the file `~/.gitignore` when maintaining version controls. This is useful for things like cached files produced by various Python scripts, compiled programs/object files from C++, and so forth. As we continue forward, we'll add some things to the global ignore, and others to repository-specific `.gitignore` files.

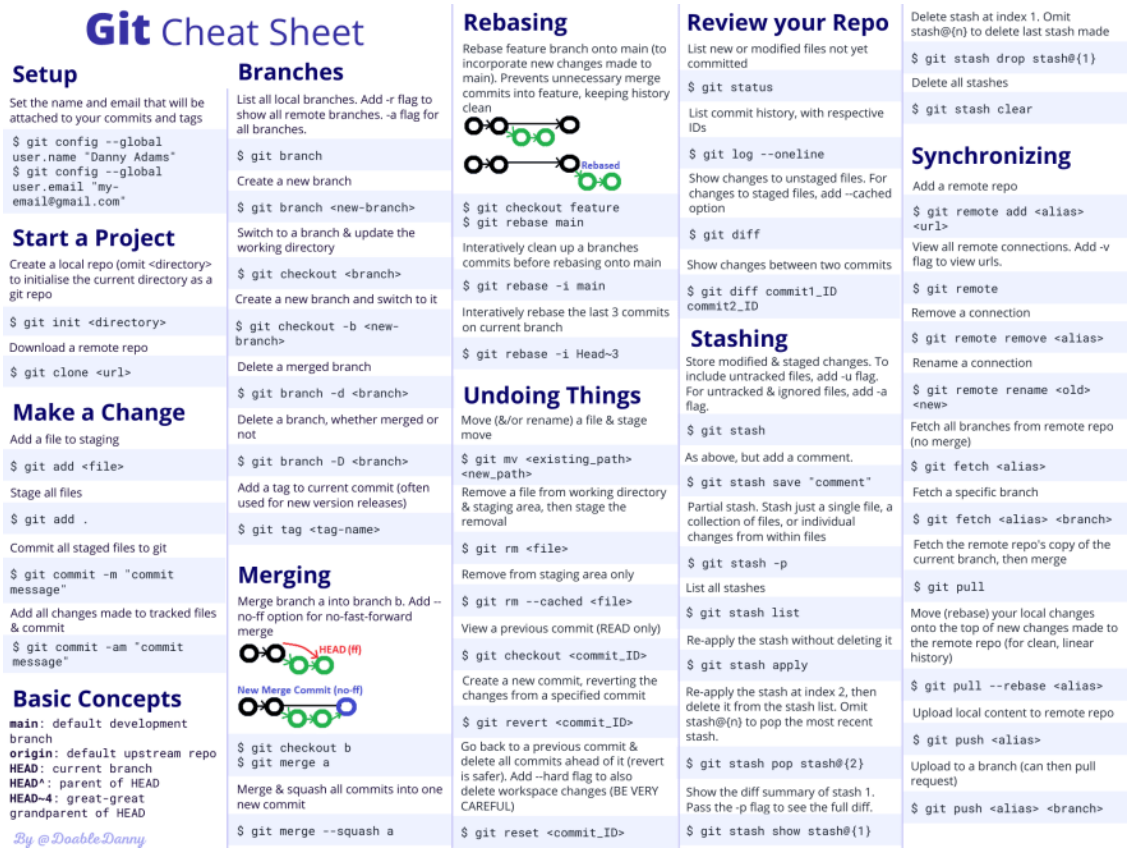
Okay, we we've configured `git` on our computers, now how about actually *using* it?

Let's say you've made some headway on designing and maybe even coding up some of your project, and you remember how important it is to maintain version controls. You can initialize the project folder `MyCodingProject` with the following command

```
git init MyCodingProject/
```

This establishes a starting point for all future versions to be compared against.

For more useful commands, check out this cheat sheet!



1.2 Code Formatting and Style

Many different research groups, organizations, and companies have what are known as "style guides" for any code produced in, by, or for the organization. These often include the simple concepts like "how many spaces constitute a `tab` in your code?" or "What information, if any, should be included in comments at the top of each file?". However, these style guides can also include more complex information beyond text formatting and up into things like "Each function definition should include comments describing the arguments to the function and what the function returns on completion" or "Test suites must be included for all additions to the codebase before they may be considered for merging".

Most programming languages have a form of something called **scope**, which is a region in the code in which certain things are true. For example, a function may have variables that only exist inside that function, and then disappear once the program exits the function's **scope**. Some languages use specific characters to define a scope, such as `C++` with `{` and `}` defining the beginning and end of a scope.

```
int main()
{
    cout << "Hello World!\n";
    if (5 < 4)
    {
        cout << "Five is less than four.\n";
    }
    return;
}
```

A common convention is to use spaces or tabs when moving into different levels of scope, however this is usually for easier reading by humans and isn't necessary for the code compiler itself.

Others, like Python, use indentations of spaces or tabs and are specifically required to change scope.

```
def myfunction(x):
    x = x + 5
    print(x)
    return

x = 10
print(x)
myfunction(x)
print(x)
```

The code snippet above shows the variable x inside the scope of `myfunction` as well as the main program. If we follow the value of `x` as the program runs, we can see that `x = 10`, which is printed out. Then, the *value* of `x` is passed into the function, which adds five and prints it out (`x = 15`). Once that is done, the function returns, and the main program's value of x is printed out again. (`x = 10`).

Let's see how that works in practice.

[2]:
```
def myfunction(x):
    x = x + 5
    print(x)
    return

x = 10
print(x)
myfunction(x)
print(x)
```

```
10
15
10
```

It is very important to keep scope in mind when using variables as counters or other housekeepers.

Many programmers use `i` as a counter variable in loops. However, sometimes it is necessary to have loops inside other loops (nested), which effectively means you have a scope inside another scope. If you use `i` in the outer loop, then change it in the inner loop, it remains changed in the outer loop and can have effects on the execution of your code.

Therefore, it is important to keep track of what variables are used during the execution of your code and how they are modified as you go.

Back to Formatting Formatting is not simply a matter of using indents or 80-characters-per-line requirements. Formatting also includes things like expected code-comments or other internal documentation. Some code development packages have the functionality built in to parse comments in the code and build human-readable documentation, but it requires the use of specific formats in the comments. See the examples below.

[3]:
```
import numpy as np
def myfunction(x,y,z):
    norm = np.linalg.norm([x,y,z])
    return norm
a = 1
b = 2
c = 3
result = myfunction(a,b,c)
```

```
print(result)
```

```
3.7416573867739413
```

The code block above has no comments in it, and so without already knowing what the individual parts are doing, it's not easy to know what is happening in the code or how to modify and manipulate it for your own purposes. If we take the same code block and add some commentary, it can be made easier.

[4]:
```
# library import
import numpy as np

# function definition
def myfunction(x,y,z):
    norm = np.linalg.norm([x,y,z])
    return norm

# Main program execution
a = 1
b = 2
c = 3
result = myfunction(a,b,c)
print(result)
```

```
3.7416573867739413
```

Now we have a little more clarity in what is happening, but it can still be made clearer. As it stands, we simply know that we're importing libraries, defining a function, and running the main program.

We can improve the commentary further by describing what is happening in the function or the steps inside the main program.

[5]:
```
# library import
import numpy as np

# function definition
def myfunction(x,y,z):
    # Arguments:
    # x - float representing the x component of a vector
    # y - float representing the y component of a vector
    # z - float representing the z component of a vector
    # Returns:
    # norm - float representing the magnitude of the vector given by the [x,y,z] values
    norm = np.linalg.norm([x,y,z])
    return norm

# Main program execution
# initialize variables
a = 1
b = 2
c = 3
# obtain the magnitude of the vector defined by [a,b,c]
result = myfunction(a,b,c)
# print out the magnitude
print(result)
```

```
3.7416573867739413
```

With this level of commentary in the code, we can easily understand what is happening in the function and the main code block. This is very useful when working on collaborative projects especially, as it can ensure that everyone is able to follow your thought process in the code and, if necessary, compare to the actual code for debugging purposes.

Another thing to consider is the larger format of a project. That is, not simply how the text is arranged in a file, but how code blocks and functions are arranged in multiple files. For smaller programs, it may not be necessary to divide the code up in this way, but for larger projects - or for standard functions you'll use in multiple separate projects - it may be easier and cleaner to keep some things separated and compartmentalized. This also makes compiling easier later on down the line.

It can also lead to smaller individual files, making debugging easier as well - most error messages include the location where the error was encountered, and it's much easier to go to `line 46 in utility.cpp` than it is to go to `line 57684 in main.cpp`. As a bonus, making changes in smaller files won't necessarily require the entire codebase to be recompiled, but rather just the small portion you modified.

Chapter 2

Thinking Like a Computer

2.1 Pseudocode

Other names for pseudocode include "algorithm development", "project management", "outlining", "planning", "thinking", and "jotting that down so I don't forget it later". Pseudocoding is effectively just writing out the stages of your program in plain language (*not code*) to ensure a clear understanding of the problem you're trying to solve. Often, programmers will begin pseudocode with a very simple set of steps they think of the problem. Each step can be explained in more and more detail as a set of smaller, more manageable steps, until eventually you wind up with a complete list of steps that can be converted to computer commands. Pseudocode can also provide some insight into ways the code might be optimized, such as by revealing opportunities to parallelise the execution, or by revealing regions where something being calculated can just be saved for reuse rather than recalculated again later.

One of the most valuable skills a programmer can develop is the ability to think like a computer. This means learning to break down larger problems and complex behaviors into smaller and smaller pieces until it becomes a collection of tiny calculations such as a sequence of additions and subtractions, or comparisons between two values.

A good habit to develop whenever beginning a new project is to first outline the expected flow of the code. Some people use a whiteboard, or scratch paper, or just a blank text document on their computer - it doesn't matter how you do it, just that you plan it out somehow before jumping into the code.

Example Project - Brownian Motion Write a program that places an arbitrary number of particles in a box of some other arbitrary size, then moves them around randomly by assigning a random x, y, and z component of their motion vector between 0 and 1.

Remember to just write in pseudocode. You can make it as complex as you like, but it should just be plain text. Try to think through the steps of the problem like a computer might.

2.2 Data and Variable Types

Different types of information can be stored for use by the computer. Some programming languages are fairly lenient about variable types and are flexible with what types of data are being provided in a given variable. Others are a bit more strict, and require explicit type declarations/definitions for each variable to ensure proper memory allocation. *If only some of that made sense, you're in the right place.*

C++ requires that every variable have a clearly defined type, which the compiler uses to ensure the program allocates the correct amount of memory at runtime. Python doesn't generally require *explicit* variable

type declarations (with some exceptions that will come later as we get into more advanced programming). However, it is still useful to know what kinds of data there is, what can be done with it, and how it's stored.

First, let's explore data types like `int`, `float`, `list`, `tuple`, and `string`.

[1]:
```
my_int    = 2
my_float  = 3.1415
my_list   = [1,3.1415,"Hello World!","pizza"]
my_tuple  = (5,6,7,8,9)
my_string = "Hello World!"
```

These examples are fairly simple.

- `my_int` is an integer, and gets treated like one. Integers are useful for things like indexes, counters, and so forth.
- `my_float` is a float (often called a "double" in other programming languages), and are regular numbers including decimals.
- `my_list` is a list of values enclosed in square brackets. Lists are indexed from zero, which means the first item in a list is "item 0". Lists are great ways to keep collections of data organized and in order, and you can extract individual values simply by including the index with the variable name: `my_list[3]` will return "pizza". You can also get values from the end of a list with negative indices. my_list`[-1]` will return "pizza" because it's the last value.
- `my_tuple` is similar to a list, except that it is a little more difficult to pull individual values from it. Tuples are useful when you need to maintain groups of values together in relation to each other, such as with (x,y,z) coordinates.
- `my_string` is a list of characters including letters, numbers, punctuation, whitespace (tabs, spaces, line breaks, etc.). The contents of a string do not include the quotation marks on either side. Strings can include quotes using *escapes* like `\"` or `\\` to include a backslash.

Variable manipulation comes in many forms and depends on the type of data contained within. Better understanding of how data types work can allow you to do some interesting things, like taking a "slice" of a string like you would from a list.

Consider the examples below.

[2]:
```
my_list = [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25]
# my_list has a length of 26 individual values
len(my_list)
```

[2]: 26

[3]:
```
my_string = "Once more into the breach!"
# my_string has a length of 26 characters including whitespace and punctuation.
len(my_string)
```

[3]: 26

Slicing Lists

One common use for lists is "slicing", where you can get a small subsection of the list. Let's say you wanted just the first five elements in `my_list`. You would use a slice. Slices are generated similar to how an individual element is called from a list, from inside square brackets. However, we can put a `:` between the starting and ending indices to get everything between. We can also use an empty space to indicate "everything". Check out the examples below.

[4]:
```
my_list[:5]
```

[4]:
```
[0, 1, 2, 3, 4]
```

[5]:
```
my_list[5:10]
```

[5]:
```
[5, 6, 7, 8, 9]
```

Note how the two results are different. The ending in the first cell is the same as the beginning of the second cell, but we don't actually get "5" in the results in the first cell. Slices go "up to" the ending value, but don't include it. Keep this in mind when working with slices. We can also combine other tricks from list manipulations, like using negative indices to go backwards from the end.

In the next cell, we'll get the last seven elements from the list.

[6]:
```
my_list[-7:]
```

[6]:
```
[19, 20, 21, 22, 23, 24, 25]
```

What if we wanted every third element in the list?

[7]:
```
my_list[::3]
```

[7]:
```
[0, 3, 6, 9, 12, 15, 18, 21, 24]
```

The second : indicates a "stride". This is useful when you have data that is strangely shaped (such as a long list of values that correspond to x,y,z coordinates, but aren't in a (3,n) shaped list.

Now let's combine these. We'll get every other element starting from the tenth and going up to the twentieth.

[8]:
```
my_list[10:20:2]
```

[8]:
```
[10, 12, 14, 16, 18]
```

Now let's look at strings. Strings are just lists of letters, numbers, and any other characters you can think of. With this in mind, we can do things to strings that we have done to lists.

[9]:
```
my_string
```

[9]:
```
'Once more into the breach!'
```

[10]:
```
my_string[:5]
```

[10]:
```
'Once '
```

[11]:
```
my_string[5:10]
```

[11]:
```
'more '
```

[12]:
```
my_string[-7:]
```

[12]:
```
'breach!'
```

[13]:
```
my_string[::3]
```

[13]:
```
'Oeo tt eh'
```

[14]:
```
my_string[10:20:2]
```

[14]:
```
'it h '
```

... some functions are more useful than others, but you get the idea!

Now let's look at integers and floats. In some programming languages, the difference between these two can be pretty severe. For example, in C++, dividing a double by an integer will give you a truncated integer, which means you can lose some of the information in your data if you're not careful. Thankfully, Python is a little more forgiving.

Normal division works like we might intuitively expect, where a float divided by an integer can be a float, and is therefore assumed to be.

[15]:
```
my_float/my_int
```

[15]:
```
1.57075
```

We can also force the division to return an integer value (which is useful in some situations)

In the example above, we got a value of 1.57075. If we were to round this using conventional methods, we'd get 2 However, forcing integer division with the `//` below gives us a truncated (not rounded) value of 1.0. This is also slightly deceptive, as the `.0` implies the value is a float,even though the result is a whole number. This is important to be aware of when doing mathematical work in python. Truncation just removes everything after the decimal point, while rounding actually considers the value beforehand.

[16]:
```
my_float//my_int
```

[16]:
```
1.0
```

[17]:
```
round(my_float/my_int)
```

[17]:
```
2
```

Math with Variables

Math can get incredibly complex, so it's important to remember your Order of Operations (PEMDAS) - Parentheses, Exponents, Multiplication, Division, Addition, and Subtraction.

However, in python it's a little different. Parentheses are solved first, then exponents, until everything in a given equation is reduced down to a series of terms separated by `+`,`-`,`*`, and `/`. Then, the values are processed left-to-right.

[18]:
```
1 + 2 - 3 * 4 / 5
```

[18]:
```
0.6000000000000001
```

[19]:
```
(1 + 2) - 3 * 4 / 5
```

[19]:
```
0.6000000000000001
```

[20]:
```
(1 + 2 - 3 * 4) / 5
```

[20]:
```
-1.8
```

[21]:
```
(1 + 2 - 3) * 4 / 5
```

[21]:
```
0.0
```

These are just a few examples of how order of operations affects the results. With this in mind, you can see why it's very important to keep track of what you're doing in a complex mathematical function. The next cell has a complex equation in a single line, then the same equation separated into more easily-managed terms.

[22]:
```
x=3
y=5
z=7
answer =  (x**(y/z)-x/((y+2)*z)-x)/(y*z)*x

print(answer)
```

```
-0.07452211053138932
```

Not only is that difficult to read, but it's also harder to see where errors might be arising. So we can rewrite it and create additional variables to hold small chunks

[23]:
```
x=3
y=5
z=7

# (x**(y/z)-x/((y+2)*z)-x)/(y*z)*x
p = y/z
# (x**p-x/((y+2)*z)-x)/(y*z)*x
q = x**p
# (q-x/((y+2)*z)-x)/(y*z)*x
r = y+2
# (q-x/(r*z)-x)/(y*z)*x
s = r*z
# (q-x/s-x)/(y*z)*x
t = y*z
# (q-x/s-x)/t*x
u = x/s
# (q-u-x)/t*x
v = q-u-x
# v/t*x
w = v/t
# w*x
answer = w*x

print(answer)
```

```
-0.07452211053138932
```

This may seem overengineered, but breaking down the individual terms is helpful in both programming and math, especially when it reveals certain trends, or even ways to rearrange an equation to reduce the

overall number of calculations being performed. This kind of breakdown can also be useful when you begin building larger, more complicated functions, even up to the point of creating entire programs or modules.

Booleans

Booleans are simply variables that are either `True` or `False`. They can also be interpreted as `1` and `0`. Booleans get used all the time in programming, though we may not be constantly aware of them.

For example, whenever we compare two numbers, the comparison creates a boolean

[24]:
```
3<5
```

[24]:
```
True
```

[25]:
```
3>5
```

[25]:
```
False
```

[26]:
```
3 == 5
```

[26]:
```
False
```

We can see that the responses for the different comparisons are correct. $3 < 5$ is true, while $3 > 5$ and $3 == 5$ are both false. Incidentally, the == is intentional. In Python and C++, = *assigns* a value, while == *compares* two values.

Booleans get used constantly in things like "if-else statements" or "while loops".

Dictionaries

Another python data type is the `dictionary` (or `dict` as it's written in python). The dictionary is a very useful datatype, as it can be used to store many different pieces of information in their own types.

[27]:
```
# A dictionary is denoted by { }
my_dictionary = {}

# At this point, "my_dictionary" is an empty dictionary with no keys or values␣
 ↪assigned.
# We can assign a key/value pair like this

my_dictionary["Name"] = "Mark"
my_dictionary["Age"]  = 37
my_dictionary["Job"] = "Postdoc"

# Now we can recall any of the values held in the dictionary by using the [key]. Keep␣
 ↪in mind, if a key already exists, the previous value will be overwritten.

# If you have a dictionary with keys that you don't know, you can get them like this:
key_list = [key for key in my_dictionary.keys()]
print(key_list)
# This might look strange, but it's done this way because my_dictionary.keys() is a␣
 ↪function call that returns an iterative set of single values, rather than the entire␣
 ↪list.

# You can also iterate through all the keys and values together.
```

```
for key,value in my_dictionary.items():
    print(key,"=",value)
```

```
['Name', 'Age', 'Job']
Name = Mark
Age = 37
Job = Postdoc
```

[28]:
```
# In a more relevant example to the lab (and demonstration of dictionary initialization with keys and values):

variant_prmtops = {"WT":"A3H_WT.prmtop",
                   "K121E":"A3H_K121E.prmtop",
                   "K117E":"A3H_K117E.prmtop",
                   "R124D":"A3H_R124D.prmtop"}

# Now I have a list of filenames stored, and I can recall them anytime with this
print(variant_prmtops["K121E"])

# We can also have dictionaries inside dictionaries, which can be useful for bigger datasets.

full_systems = {
"WT" : {"prmtop":"WT.prmtop","trajectory":"WT_100ns.dcd","num_residues":180,"duration":100},
"K121E" : {"prmtop":"K121E.prmtop","trajectory":"K121E_150ns.dcd","num_residues":180,"duration":150},
"K117E" : {"prmtop":"K117E.prmtop","trajectory":"K117E_200ns.dcd","num_residues":180,"duration":200}
}

print(full_systems["WT"]) ## This prints the entire dictionary

print(full_systems["WT"]["prmtop"])

# You can also store larger datasets inside dictionaries this way.  For example, let's say you have a dataset for the RMSD of an MD trajectory called "rmsd", and one for correlated motion called "correl"
import numpy as np
rmsd = np.random.rand(100)
correl = np.random.rand(50,50)

WT_analyses = {"RMSD":rmsd,"correl":correl}
```

```
A3H_K121E.prmtop
{'prmtop': 'WT.prmtop', 'trajectory': 'WT_100ns.dcd', 'num_residues': 180,
'duration': 100}
WT.prmtop
```

[29]:
```
WT_analyses["correl"]
```

[29]:
```
array([[0.07872392, 0.42709779, 0.73786009, ..., 0.14944642, 0.19055872,
        0.45752917],
```

```
       [0.14328318, 0.1483461 , 0.63825779, ..., 0.92527165, 0.9000854 ,
        0.16021734],
       [0.50201692, 0.75404792, 0.46468422, ..., 0.44324784, 0.74848111,
        0.94918908],
       ...,
       [0.64425718, 0.47652132, 0.69848477, ..., 0.19183174, 0.42951161,
        0.73094065],
       [0.2925301 , 0.79944765, 0.88059451, ..., 0.65282398, 0.82365397,
        0.56632012],
       [0.68523568, 0.71003483, 0.5788092 , ..., 0.22053449, 0.22295812,
        0.29099571]])
```

[30]:
```
WT_analyses["RMSD"]
```

[30]:
```
array([0.77547714, 0.86825809, 0.73361393, 0.53601816, 0.82347301,
       0.77017413, 0.96120335, 0.41538613, 0.59034859, 0.04939984,
       0.11903879, 0.20902424, 0.76954975, 0.20189964, 0.31954835,
       0.51617544, 0.71262053, 0.44747436, 0.17991915, 0.95935479,
       0.02334348, 0.37246611, 0.37715923, 0.31250376, 0.83167922,
       0.7179388 , 0.26188751, 0.10515804, 0.62818762, 0.82602609,
       0.98213736, 0.22903547, 0.72848045, 0.45872938, 0.26119027,
       0.05973667, 0.65432271, 0.86798405, 0.66082425, 0.0277142 ,
       0.45905896, 0.74384669, 0.29576668, 0.89319424, 0.14992499,
       0.1549802 , 0.89640709, 0.49811178, 0.07505158, 0.85436875,
       0.21030718, 0.13886558, 0.0993692 , 0.04939456, 0.71672669,
       0.00323682, 0.60787663, 0.43602982, 0.31186697, 0.02260092,
       0.41310232, 0.56875889, 0.27089672, 0.24635137, 0.25593297,
       0.65855327, 0.7809256 , 0.37840985, 0.18978786, 0.50569471,
       0.36361048, 0.32712705, 0.5499439 , 0.71682757, 0.01187723,
       0.74752752, 0.36253046, 0.4534361 , 0.75456511, 0.1084834 ,
       0.80284826, 0.66146861, 0.70693715, 0.61945641, 0.04079063,
       0.93344407, 0.71655886, 0.34367994, 0.41819506, 0.66072591,
       0.32044302, 0.3986235 , 0.87465149, 0.94527801, 0.72854001,
       0.16153919, 0.59174719, 0.56051911, 0.26658   , 0.14256879])
```

2.3 Debugging

Debugging code is often a team effort, because invariably the person who wrote the code might wind up missing some of their own mistakes that others will readily find. This is not at all an indicator of skill, intelligence, or character. This is purely because we can get tunnel vision about our own code (it happens to the best of us), and because our brains are wired to recognize *and complete* patterns. Where we might "see" a missing semicolon at the end of a line of code we wrote because we expect it to be there, someone else who didn't write our code may notice it immediately. Interestingly, debugging is actually a *highly* valued skill in industry, because fixing problems is usually far more expensive than preventing them in the first place.

It's often said that only a small portion of programming is actually writing the code - the rest is fixing the code.

Debugging is required at multiple stages of programming, and bugs can arise in many different forms.

Compilation Bugs Compilation errors are usually the easiest to solve, as they're encountered by the compiler itself and are often syntax-related ("missing semicolon on line 85") or datatype-related ("Unable to

cast string as int"), and usually include "tracebacks" which can help you figure out where exactly the error is.

Runtime Bugs There are a few kinds of runtime errors that can pop up. They are usually more complicated to unravel, depending on what caused them. The first kind is something that, while the code will compile fine, the actual execution will return an error. For example, a function that takes `x` and `y` and returns the result of `x/y` may compile just fine. But when you run the code, if somehow `y = 0`, the program will crash because of an attempt to divide by zero.

Another bug that can arise is when the results are not what was expected. For instance, if you have a program that should return the product of two numbers, and instead returns the sum of the two numbers, this is a runtime error, even though no error is actually reported. The code compiles fine and executes exactly as it is written, but it may not be what was intended. These types of bugs are why validation and testing are required for programs big and small.

Take a look at the code blocks below. One contains an example of a compilation error, the other a runtime error.

[7]:
```
int x = 45
print(x)
```

```
  File "/tmp/ipykernel_10751/2716038442.py", line 1
    int x = 45
        ^
SyntaxError: invalid syntax
```

[10]:
```
def divide(x,y):
    return x/y

x = 5
y = 1
result = divide(x,y)
print(x,y,result)

x = 5
y = 0
result = divide(x,y)
print(x,y,result)
```

```
5 1 5.0
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
/tmp/ipykernel_10751/1166490746.py in <module>
      9 x = 5
     10 y = 0
---> 11 result = divide(x,y)
     12 print(x,y,result)
     13

/tmp/ipykernel_10751/1166490746.py in divide(x, y)
      1 def divide(x,y):
----> 2     return x/y
```

```
      3
      4 x = 5
      5 y = 1

ZeroDivisionError: division by zero
```

It should be pointed out that *technically*, `Python` doesn't have compile errors since it's not compiled at all, but runs like a scripting language which merely interprets the commands line by line. However, more advanced python programming can include the actual compilation of python scripts into self-contained programs that don't require any external libraries. This is not in the scope of this workshop, however, and is merely pointed out for information.

Chapter 3

Quality Control and Optimization

3.1 Quality Control

The term "quality control" has some fairly broad implications. Depending on the specific project, it might refer to different aspects of programming, from how well the code is documented to how close to reality the results of some internal calculation get. Programs, especially large ones with multiple contributors, often need lots of careful attention to quality control. This is most important when programming for a company or organization that relies on the quality of the code to make everything else work.

Quality control also means ensuring that you (or your colleagues) are not committing some of the more common "tricks" that many programmers will boast about behind the anonymity of the internet, such as "I've included a sleep function in the code so that whenever my boss complains that the program is too slow, I can reduce the value of that function by one and then tell him I optimized the code a bit and got a speedup!" While that sounds very clever on the face of it, and like a great way to make yourself look good to your employers, the discovery of such things will usually lead to getting fired, and it also shows other programmers that your goal is to **seem** good, rather than actually **be** good. Don't do stuff like that, and discourage others from that behavior as well.

3.2 Optimization

There are many ways in which code can be optimized, depending entirely on what aspect of the problem you wish to tackle first. Ultimately, every optimization is an attempt to reduce the usage of available resources. These resources can be computational, like the amount of RAM used by a program (lookin' at you, Chrome), or the amount of processor power needed. They can also be non-computational, such as the overall amount of time needed to run a program, or the level of human involvement necessary during program execution.

Memory Management

Part of programming is the assignment of memory (RAM) to specific variables for the storage of data. Often, programmers will use variables to hold information they need to use later as a matter of convenience. However, each variable requires memory allocation, and eventually a programmer will run afoul of the hard limit that is the amount of available RAM in their computer. There are a number of things that can be done to reduce memory usage in a program.

First, you can check for variables that are assigned and never used. Most modern compilers will alert when these are encountered, but will continue to compile regardless. Second, you can see what variables are assigned and then immediately used and then never used again. If these variables occur inside the scope of a small function, it's generally not an issue - the memory will be freed as soon as the program exits that

function - however it's still good practice to reduce unnecessary memory allocation operations. Third, you can make sure that the kind of variable you are using is appropriate. For some languages, variable type is dynamically assigned and the programmer has little real influence over this. For others, however, where the programmer has control over data types, it can be helpful to use the right type for the situation. As an example, if you need a variable to act as a counter, you would use an integer which uses 4 bytes in C++, not a double which uses 8 bytes, or a long double which uses 12. These may seem like small, almost negligible differences, but at larger scales of data, these choices can pile up to real improvements.

This is also true for things like file outputs. For example, let's say you have written a program that calculates the excitation wavelength of a molecule and gets out to 12 decimal places in nanometers. Do you actually have the significant figures correct? Is it even reasonable to go out to 12 decimal places when looking at the excitation wavelength of a molecule if most spectrometers can only give an experimental value out to $\pm 1nm$? Consider these kinds of things when writing your code - file I/O takes additional computational time and storage capacity.

Processor Optimization

These kinds of optimizations are often more esoteric and are not considered as critical except in cases where certain types of calculations are being repeated hundreds of thousands of times. Consider an example where you are comparing a variable to the square root of another variable. This might look like this:

The square root function is computationally expensive compared to the squaring function (n^2). This means, from a processor optimization standpoint, it would be cheaper to have the following:

This results in the same boolean expression (True or False), but the second example takes less computational power than the first. If this is something happening hundreds of thousands of times in a program, such as when comparing distances between atoms, the difference can add up.

Parallel Processing

Many processors have multiple cores, giving them the ability to run multiple calculations at once, allowing certain tasks to be completed MUCH faster when properly assigned. These cores are often accessed by MPI (Message Passing Interface) or OpenMP, both of which allow the programmer to assign specific tasks to individual cores on a processor and then collect the results back to a main core for subsequent work.

Many tasks are "parallelizable" or even "embarassingly parallelizable", which means they can be broken up into multiple parts that can be completed independently and simultaneously without loss of accuracy in the end result. A simple example would be the summation of a hundred million numbers. If you were using a single core, you would have to add each number to the total in sequence. If each addition took a nanosecond, the summation of 100M numbers would take about 100 seconds. If, however, you were using twenty cores, and you divided the set of numbers up, each core would only have to handle 5 million numbers, and then the final 20 values would need to be added together at the end for the last value. This means the process would take only 5 seconds instead of 100. Imagine such a speedup across much longer and more complicated programs, or with greater numbers of available processing cores.

Some tasks are considered to be "nonparallelizable", which usually means that each stage of a calculation is dependent on the result of a previous stage. One example is the calculation of the position of a particle over time. If we are given a particle with an initial position, mass, velocity, and gravitational acceleration and told to calculate its position at each timestep, we cannot parallelise this because the position and velocity at T_{n+1} are dependent on the position and velocity at time T_n.

Algorithm Optimization

There is a coding challenge that Google reveals to users once they've searched enough programming-related stuff over a long-enough period of time. This challenge has multiple levels, with increasing time limits for completion. Level 1 gives you 48 hours to complete, while Level 5 gives you 22 days.

One particular level involves using a bitwise function, XOR, to "decrypt a password" (the whole challenge is in the context of rescuing bunnies from a mad scientist). However, using XOR directly will actually cause you to fail the challenge because one of the hidden tests is how much time the algorithm takes to complete. Upon researching more about XOR, I learned that there is a cyclic nature to the outputs of the command that repeats every 4 values. The result of the algorithm is therefore predictable, and the expected result can be obtained far more efficiently than actually running the algorithm. That is, rather than running the computationally expensive XOR command thousands and thousands of times to get the final result, we can simply divide the initial value by 4, take the remainder, and return a known value from that.

The purpose of this specific challenge was not just to force the user to write a program to do a task, but also to make them understand the actual math behind the code and the subsequent effects on a larger dataset.

Algorithm optimization is often one of the hardest and yet most rewarding forms of optimization, because it does not rely in any way on the code itself, but on the ability of the programmer to think through a problem and identify special characteristics of that problem.

Part II

Python

Chapter 4

Introduction and Basics

4.1 Error Handling

Error handling refers to a set of instructions the program follows if it encounters an error. If you've gotten any kind of an error in python, you've seen error handling in action.

What if you wanted to write code that would just take care of the problem if it encountered an error, rather than crashing and complaining to the user? If you can anticipate possible errors that may arise, you can account for them and work around them.

Error handling in python is based first on the `try/except` commands. Let's look at the examples below.

[8]:
```
import foobar as fb
print("Hello world!")
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
/home/mark/GH_Repositories/CodingSummerSchool/Day05_Python_WrapUp/
 ↪18_Python_Error_Handling.ipynb Cell 2' in <cell line: 1>()
----> <a href='vscode-notebook-cell:/home/mark/GH_Repositories/CodingSummerSchool/
 ↪Day05_Python_WrapUp/18_Python_Error_Handling.ipynb#ch0000000?line=0'>1</a> import␣
 ↪foobar as fb
      <a href='vscode-notebook-cell:/home/mark/GH_Repositories/CodingSummerSchool/
 ↪Day05_Python_WrapUp/18_Python_Error_Handling.ipynb#ch0000000?line=2'>3</a>␣
 ↪print("Hello world!")

ModuleNotFoundError: No module named 'foobar'
```

[4]:
```
try:
    import foobar as fb
except:
    print("foobar module is not available.")
print("Hello world!")
```

```
pytraj module is not available.
Hello world!
```

In the first example, we tried to directly import a python module that doesn't exist (on the computer or even at all), and then print "Hello World!" But because `foobar` doesn't exist, python encountered an error

and immediately crashed. In the second example, however, we included the `try/except` commands, which allows us to follow a specific set of instructions when the error is encountered, and then continue on with the program.

I commonly use this in scripts and programs that I provide to the lab that use non-standard python libraries. For example, a script may use the `mdanalysis` library, which is not included with python by default.

[ ]:
```
try:
    import mdanalysis as mda
except:
    import subprocess
    subprocess.call("conda install -y mdanalysis",shell=True)
    import mdanalysis as mda
```

The cell above tries to import `mdanalysis`, and if it fails, it loads a standard python library called `subprocess`, then uses that to install `mdanalysis`. Once that has completed, the code continues and imports mdanalysis as though it were always there.

Error handling can get more complex than this, with specific instructions being assigned to specific **types** of errors.

If you wanted to be really mean, you could include a handling of the "KeyboardInterruptException" to prevent your code from being killed by the usual `ctrl-C` method.

But don't do that.

4.2 File I/O

I/O stands for "Input/Output". Loading files into your programs, parsing the data inside, and being able to use, manipulate, transform, present, and save the results is a very important skillset.

Reading Files

Python has a few different ways to read files, but for now we'll stick with plain-text files like you'll get from various analysis programs or computational chemistry software packages. Python can also read binary data files, but that's generally more advanced and requires a bit more experience.

The number one rule to remember for any and all files is that if you **open** it, you must **close** it. A file that is opened and never closed can either wind up inaccessible to other programs or the user, or the data that was meant to be stored within can be lost.

See the example below. With the `open` function, we create a **file object** that has several useful internal functions we can take advantage of. `open()` takes two arguments - the filename first, then the type of action to be taken on the file. "r" is for "read mode", "w" is for "write mode" which starts at the beginning of the file and writes (including overwriting!), and "a" is for "append mode", which acts like "write mode" except that it adds to the end of the file rather than overwriting it.

[6]:
```
myfile = open("test_data/file_reading.txt","r")

contents = myfile.read()

print(type(contents))
print(contents)

myfile.close()
```

```
<class 'str'>
This is just a normal text file.
There are words.  There are sentences.  There are lines.

There was even a blank line.
After this sentence is the end of the file, usually called "EOF".
```

As you can see, the entire file can be read into a single string, which includes all the line breaks and everything within. This can be easiest if you have a small file, or if you know exactly the structure of the data inside.

There is also a function that can read one line at a time. This function, `readline()`, returns the *next* line in the file, up to the newline character, `\n`. We often use this if the file is large, or if there are multiple line formats in the file (such as we might see in a TeraChem output file), or if we want to keep memory usage low.

[10]:
```
myfile = open("test_data/file_reading.txt","r")

contents = myfile.readline()
print(contents)

### We've already read the first line, but if we do the same set of commands again,␣
 ↪we'll get the next line

contents = myfile.readline()
print(contents)

### And again, for the third. (see above, the third line is blank!)

contents = myfile.readline()
print(contents)

### Fourth

contents = myfile.readline()
print(contents)

myfile.close()
```

```
This is just a normal text file.

There are words.  There are sentences.  There are lines.



There was even a blank line.
```

We can also get and keep all the lines in memory stored as a `list`, with each element of the list holding one line of the file as a string. This uses a function called `readlines()`, which has the `s` at the end to keep it separate from the last function, `readline()`.

[12]:
```
myfile = open("test_data/file_reading.txt","r")
all_lines=myfile.readlines()
print(all_lines)
```

```
myfile.close()
```

```
['This is just a normal text file.\n', 'There are words.  There are sentences.
There are lines.  \n', '\n', 'There was even a blank line.\n', 'After this
sentence is the end of the file, usually called "EOF".']
```

Now you can see all the lines in the file, but presented as a list of strings instead of the formatted text. With the list structure of the `readlines()` function, you can then do more complicated things to the individual lines by iterating through the list, or choose specific lines with list-slicing like we discussed previously.

[14]:
```
for line in all_lines:
    if "There" in line:
        print(line)
```

```
There are words.  There are sentences.  There are lines.

There was even a blank line.
```

You can also further split up the lines with the `.split()` function available to any string object.

[15]:
```
for line in all_lines:
    print(line.split())
```

```
['This', 'is', 'just', 'a', 'normal', 'text', 'file.']
['There', 'are', 'words.', 'There', 'are', 'sentences.', 'There', 'are',
'lines.']
[]
['There', 'was', 'even', 'a', 'blank', 'line.']
['After', 'this', 'sentence', 'is', 'the', 'end', 'of', 'the', 'file,',
'usually', 'called', '"EOF".']
```

With this in mind, can you write a function to search the `TeraChemOutput.out` file (included in the `test_data` folder) for every instance of "RMS grad"?

[ ]:
```
### Your code goes here!
```

Now that you can find all the lines with "RMS grad" in them, can you extract the current value at each line and add it to a dataset?

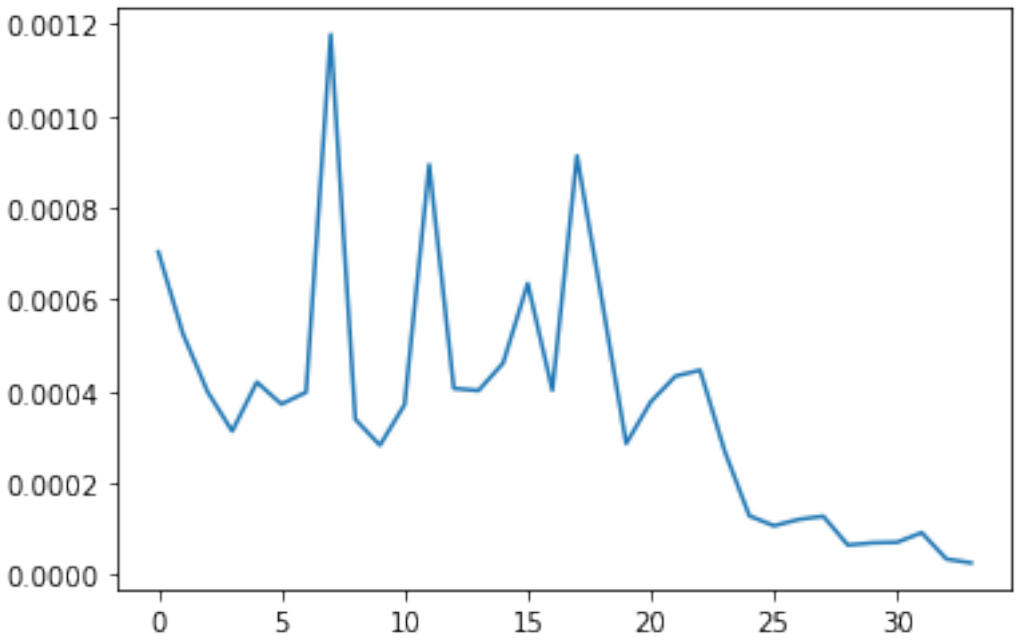
Hint: use the `append` *function, and don't forget to convert from a string to a float!*

[5]:
```
dataset = []

### Your code goes here!
```

If you were successful in finding, extracting, and storing the "RMS grad" value for each occurrence in the file, run the next cell to see that data plotted!

[6]:
```
import matplotlib.pyplot as plt
plt.plot(dataset)
```

[6]:
```
[<matplotlib.lines.Line2D at 0x7fb50353f490>]
```



You can see how useful parsing datafiles can be, especially when those datafiles aren't as neatly formatted as a CPPTRAJ output, or when there is lots of different kinds of data that you may not need or want at that moment.

But what about writing your own outputs?

Writing Files

To write to a file, you start in much the same way as reading, except you open the file in "w" or "a" mode. The file, if it doesn't already exist, will be created on `open` and saved on `close`, so again be sure to close every file you open.

[7]:
```
myfile = open("test_data/writing_outputs.txt","w")

myfile.write("This is the text I am writing. ")
myfile.write("Notice how there is no newline after the last sentence... ")
myfile.write("You have to include the newline character explicitly when writing to a␣
 ↪file.\n")
myfile.write("\n")
myfile.write("Like that!\n")

myfile.close()
```

You can also write data with specific formatting, producing nice neat columns of data that can make it easier to read and analyze. You can use string-formatting to specify the allowed width of sections of a line.

Consider the following set of data:

[11]:
```
people = [  {"Name":"Bobby","Age":25,"Class":"First-Year","GPA":3.95},
            {"Name":"Charlie","Age":27,"Class":"Second-Year","GPA":3.87},
            {"Name":"David","Age":26,"Class":"First-Year","GPA":4}]
```

```
myfile = open("test_data/people_test.txt","w")
myfile.write(f"{'Name':<20} {'Age':>10} {'Class':<20} {'GPA':>10}\n")
myfile.write("-------------------------------------------------------------------\n")
for person in people:
    myfile.write(f"{person['Name']:<20} {person['Age']:>10} {person['Class']:<20}␣
 ↪{person['GPA']:>10.4f}\n")
myfile.close()
```

If you open the file we've just created, it should look like this:

```
Name                          Age Class                          GPA
-------------------- ---------- -------------------- ----------
Bobby                          25 First-Year                 3.9500
Charlie                        27 Second-Year                3.8700
David                          26 First-Year                 4.0000
```

Notice that the first column is left justified and has empty spaces out to the specified length of 20 characters. Next, "Age" is right-justified with 10 spaces total. Class is left-justified again with twenty, and GPA is right-justified *with trailing zeros*, which we specified with the `.4f` addition, indicating we wanted four decimal places and that the value is to be treated like a float. I also included some blank spaces between each of the variables being printed 1) to show you where each section begins and ends, and 2) to illustrate how you should be careful when using justifications so you don't wind up with Age and Class stuck together as a single value.

Also notice that there is a `\n` character at the end of each line in the code. This ensures that the text file is properly separated by lines - python won't write what you don't tell it to write.

What if we want to add to a file that already exists? Simple, use "append mode", or open with "a".

[12]:
```
people = [  {"Name":"Esther","Age":24,"Class":"First-Year","GPA":3.96},
            {"Name":"Frances","Age":28,"Class":"Second-Year","GPA":3.99},
            {"Name":"Gloria","Age":27,"Class":"First-Year","GPA":4}]

myfile = open("test_data/people_test.txt","a")
for person in people:
    myfile.write(f"{person['Name']:<20} {person['Age']:>10} {person['Class']:<20}␣
 ↪{person['GPA']:>10.4f}\n")
myfile.close()
```

Now if we reopen the file we created, we can see the addition of the next three people to the list! This can be useful for things like logging results over the course of a longer calculation, where you might not want the file to be kept open and in memory for the entire time, but still want to add to it on occasion.

File I/O can help you find, organize, and process your data in a way that makes sense to you.

4.3 System Interaction

One of the included modules in python is the `subprocess` module, which allows you to execute shell commands on a Linux system from within your python script. You can also use it to read the outputs from these commands. There are a few different options available to manage these kinds of system/program interactions.

subprocess.call() The `subprocess.call()` function is fairly straightforward. It is useful if you need to execute a command without concern for the output to the terminal of that command.

[1]:
```
import subprocess

subprocess.call("ls -lrth",shell=True)
```

```
total 48K
-rw-rw-r-- 1 mark mark    0 Jun 13 16:37 19_Python_System_Interaction.ipynb
-rw-rw-r-- 1 mark mark    0 Jun 13 16:38 20_Python_Final_Project.ipynb
drwxrwxr-x 2 mark mark 4.0K Jun 17 16:45 test_data
-rw-rw-r-- 1 mark mark  33K Jun 17 17:01 17_Python_File_IO.ipynb
-rw-rw-r-- 1 mark mark 4.8K Jun 17 17:35 18_Python_Error_Handling.ipynb
```

[1]:
```
0
```

In a notebook environment, the results of the `subprocess.call()` command are simply printed to the output of the cell. In a regular system environment, this output is printed to the terminal. The important thing to remember, however, is that neither of those options allow for the capture and storage of the output by python. In the example above, we as a user can see the output of our `ls -lrth` command, but our **program** cannot.

I often use the `call()` function when calling external programs from within my python scripts, specifically when those programs will produce their own output files that I can use later. In these scenarios, the outputs to the terminal are not as important.

Also, note that `subprocess.call` includes a keyword argument `shell=True`. This is *required* to have the command actually run in your shell environment.

subprocess.Popen() The other kind of system interaction requires the capture and storage of the terminal outputs. The `Popen` command can be used in the same way as `call`, but with some additional keywords and with the functionality included to capture the response from the commands.

[2]:
```
import subprocess

proc = subprocess.Popen("ls -lrth",shell=True,stdout=subprocess.PIPE,stderr=subprocess.
 ↪PIPE)
output,errors = proc.communicate()
```

[8]:
```
print(output)
```

```
b'total 48K\n-rw-rw-r-- 1 mark mark    0 Jun 13 16:37
19_Python_System_Interaction.ipynb\n-rw-rw-r-- 1 mark mark    0 Jun 13 16:38
20_Python_Final_Project.ipynb\ndrwxrwxr-x 2 mark mark 4.0K Jun 17 16:45
test_data\n-rw-rw-r-- 1 mark mark  33K Jun 17 17:01
17_Python_File_IO.ipynb\n-rw-rw-r-- 1 mark mark 4.8K Jun 17 17:35
18_Python_Error_Handling.ipynb\n'
```

[9]:
```
print(errors)
```

```
b''
```

You will note that when the `output` and `errors` variables are printed, they have a `b` at the front of the string. This is an important thing to be aware of - these outputs are *technically* in binary format, not plain-text. Fortunately, the python `print` function can translate from binary to text. However, working with binary data can be slightly more difficult in some ways, and it might be easier for us to convert to plain strings before parsing information. We can use the `decode` function for this.

[12]:
```
decoded = output.decode("utf-8")

print(decoded)
```

```
total 48K
-rw-rw-r-- 1 mark mark    0 Jun 13 16:37 19_Python_System_Interaction.ipynb
-rw-rw-r-- 1 mark mark    0 Jun 13 16:38 20_Python_Final_Project.ipynb
drwxrwxr-x 2 mark mark 4.0K Jun 17 16:45 test_data
-rw-rw-r-- 1 mark mark  33K Jun 17 17:01 17_Python_File_IO.ipynb
-rw-rw-r-- 1 mark mark 4.8K Jun 17 17:35 18_Python_Error_Handling.ipynb
```

The variable `output` that we got from the `communicate` function has its own internal function called `decode`, which we call with the argument `"utf-8"`, indicating we want to use that particular Unicode formatting. Note that the resulting string does NOT have the `b` at the front, and is printed like any other string. We can also use this data like any other string, with `split` commands and other types of parsing to get the information we want from the output.

Anyone who has looked into the `TCFreeze` code will recognize the `subprocess` module as an integral part of the larger algorithm. `TCFreeze` uses `subprocess.call()` to use TeraChem at every iteration of the process.

Try out some of your own commands to get a feel for how the different interactions work.

Chapter 5

Libraries

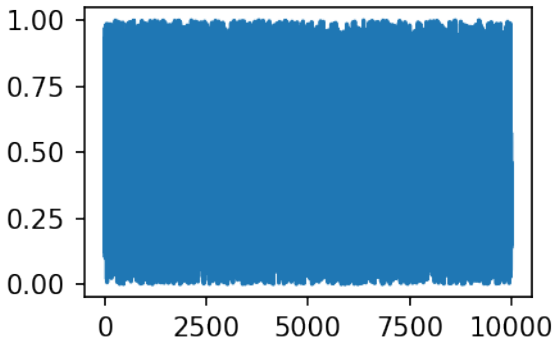
5.1 Library - Matplotlib

Let's assume you've got some data. Good data that your advisor will be pleased with. Mythical Data. How should you present this data?

1. Prepare the figure environment with `matplotlib.pyplot`

[1]:
```
import matplotlib.pyplot as plt
import numpy as np

#### GENERATING RANDOM DATA!
my_data=np.random.rand(10000) # this is just a dataset of 10,000 random values between␣
 ↪0 and 1

fig = plt.figure(figsize=[3,2],dpi=150) # this initializes a Figure with the size 3 in.
 ↪ wide by 2 in. tall at 300dpi
ax = fig.add_subplot(1,1,1) # this adds a set of axes as a subplot to the figure.

# The numbers 1,1,1 indicate the arrangement of subplots should be 1 row, 1 column,␣
 ↪and the current axes are in cell 1.
# if we used (3,2,4), this would be 3 rows, 2 columns, and in cell 4.  Cells are␣
 ↪numbered left-to-right, then top-to-bottom.
# [ ] [ ]
# [ ] [X]
# [ ] [ ]
# This is where cell 4 would be in the (3,2,4) example.

ax.plot(my_data)
```

[1]:
```
[<matplotlib.lines.Line2D at 0x7f775e5faa90>]
```

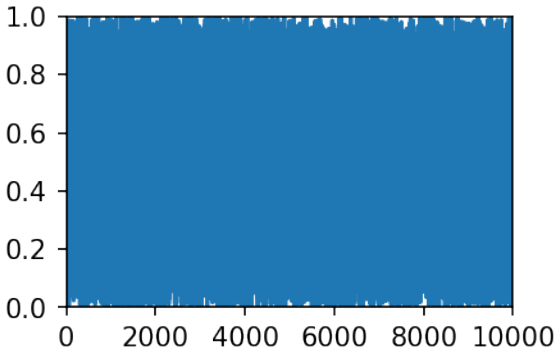


Lovely.

Let's mess with the x- and y-axis limits

[2]:
```
fig = plt.figure(figsize=[3,2],dpi=150)
ax = fig.add_subplot(1,1,1)
ax.plot(my_data)
#### NEW FUNCTIONALITY!
ax.set_xlim(0,10000)
ax.set_ylim(0,1)
```

[2]:
```
(0.0, 1.0)
```

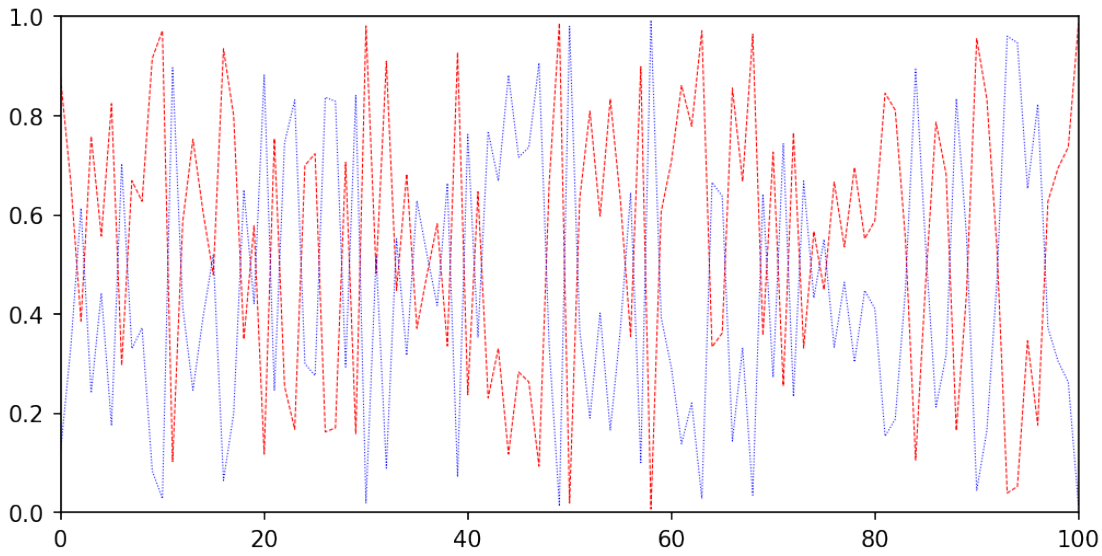


Cool. Now we can adjust the axes to ensure a) our data is presented how we want it, and b) we don't have empty space around it, which can be confusing sometimes. Now let's adjust the color, thickness, and style of the line by adding keywords to the `ax.plot` function call. I'll also adjust the x-axis down to `[0,100]` to let us see the line a little more easily - we'll address other methods of dealing with this later. We can also mess with the figure size.

[3]:
```
fig = plt.figure(figsize=[8,4],dpi=150)
ax = fig.add_subplot(1,1,1)
ax.set_xlim(0,100)
ax.set_ylim(0,1)
#### NEW FUNCTIONALITY!
```

```
ax.plot(my_data,color="red",lw=0.5,linestyle="--")
ax.plot(1-my_data,color="blue",lw=0.5,linestyle=":")
```

[3]: `[<matplotlib.lines.Line2D at 0x7f775dc7a8e0>]`

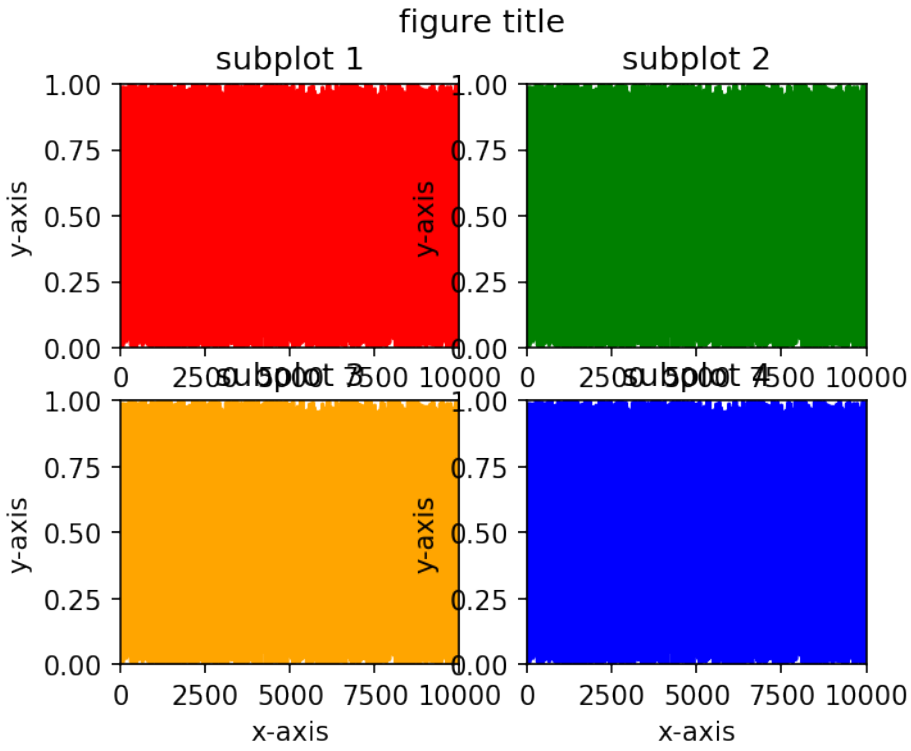


We can also add titles to each axis, to the subplot, or to the entire figure.

[7]:
```
fig = plt.figure(figsize=[5,4],dpi=150)

ax = fig.add_subplot(2,2,1)
ax.plot(my_data,color="red")
ax.set_xlim(0,10000)
ax.set_ylim(0,1)
#### NEW FUNCTIONALITY!
ax.set_xlabel("x-axis")
ax.set_ylabel("y-axis")
ax.set_title("subplot 1")

ax = fig.add_subplot(2,2,2)
ax.plot(my_data,color="green")
ax.set_xlim(0,10000)
ax.set_ylim(0,1)
#### NEW FUNCTIONALITY!
ax.set_xlabel("x-axis")
ax.set_ylabel("y-axis")
ax.set_title("subplot 2")

ax = fig.add_subplot(2,2,3)
ax.plot(my_data,color="orange")
ax.set_xlim(0,10000)
ax.set_ylim(0,1)
#### NEW FUNCTIONALITY!
```

```
ax.set_xlabel("x-axis")
ax.set_ylabel("y-axis")
ax.set_title("subplot 3")

ax = fig.add_subplot(2,2,4)
ax.plot(my_data,color="blue")
ax.set_xlim(0,10000)
ax.set_ylim(0,1)
#### NEW FUNCTIONALITY!
ax.set_xlabel("x-axis")
ax.set_ylabel("y-axis")
ax.set_title("subplot 4")

fig.suptitle("figure title")
```

[7]: `Text(0.5, 0.98, 'figure title')`

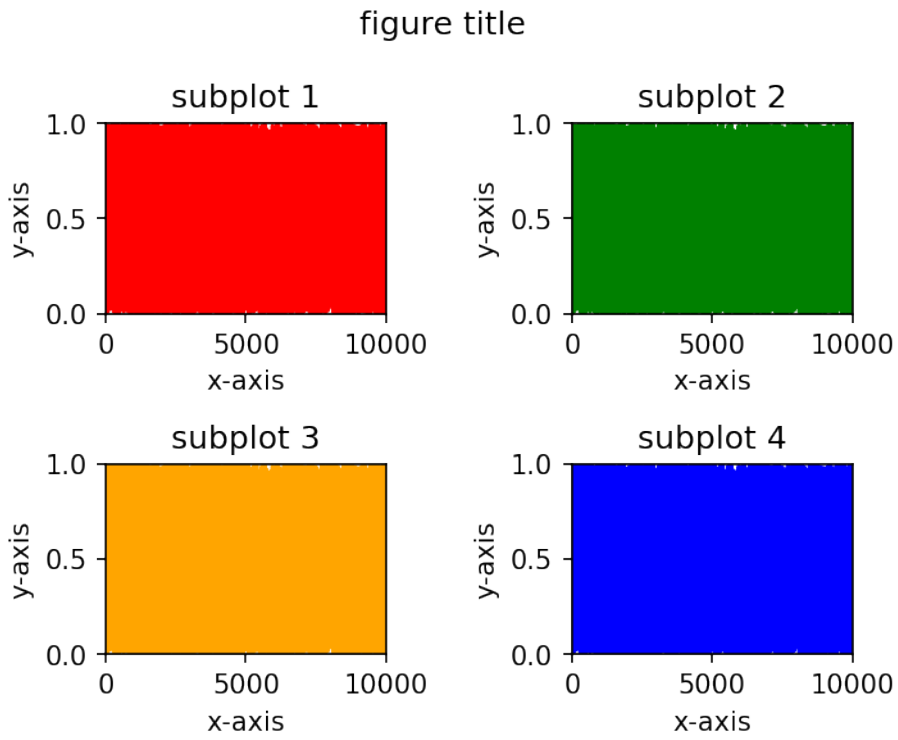


Now we can see a problem. The individual subplots are overlapping each other and making the figure an unreadable mess.

let's add one line at the end of the cell

[6]:
```
fig = plt.figure(figsize=[5,4],dpi=150)

ax = fig.add_subplot(2,2,1)
ax.plot(my_data,color="red")
ax.set_xlim(0,10000)
```

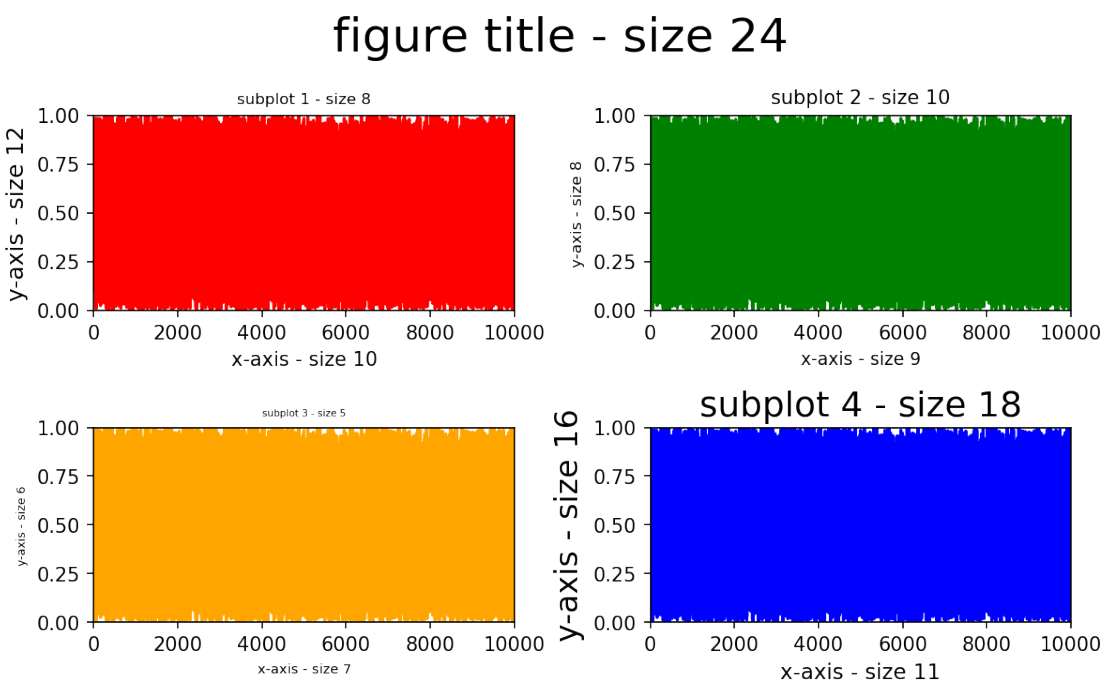
```
ax.set_ylim(0,1)
ax.set_xlabel("x-axis")
ax.set_ylabel("y-axis")
ax.set_title("subplot 1")

ax = fig.add_subplot(2,2,2)
ax.plot(my_data,color="green")
ax.set_xlim(0,10000)
ax.set_ylim(0,1)
ax.set_xlabel("x-axis")
ax.set_ylabel("y-axis")
ax.set_title("subplot 2")

ax = fig.add_subplot(2,2,3)
ax.plot(my_data,color="orange")
ax.set_xlim(0,10000)
ax.set_ylim(0,1)
ax.set_xlabel("x-axis")
ax.set_ylabel("y-axis")
ax.set_title("subplot 3")

ax = fig.add_subplot(2,2,4)
ax.plot(my_data,color="blue")
ax.set_xlim(0,10000)
ax.set_ylim(0,1)
ax.set_xlabel("x-axis")
ax.set_ylabel("y-axis")
ax.set_title("subplot 4")

fig.suptitle("figure title")
#### NEW FUNCTIONALITY!
fig.tight_layout()
```



Well, they're no longer overlapping each other. But now it looks ridiculous and not very useful. A larger figure size might help. We can also mess with font sizes of various elements.

[8]:
```
#### NEW FUNCTIONALITY!
fig = plt.figure(figsize=[8,5],dpi=150)

ax = fig.add_subplot(2,2,1)
ax.plot(my_data,color="red")
ax.set_xlim(0,10000)
ax.set_ylim(0,1)
#### NEW FUNCTIONALITY!
ax.set_xlabel("x-axis - size 10",fontsize=10)
#### NEW FUNCTIONALITY!
ax.set_ylabel("y-axis - size 12",fontsize=12)
#### NEW FUNCTIONALITY!
ax.set_title("subplot 1 - size 8",fontsize=8)

ax = fig.add_subplot(2,2,2)
ax.plot(my_data,color="green")
ax.set_xlim(0,10000)
ax.set_ylim(0,1)
#### NEW FUNCTIONALITY!
ax.set_xlabel("x-axis - size 9",fontsize=9)
#### NEW FUNCTIONALITY!
ax.set_ylabel("y-axis - size 8",fontsize=8)
#### NEW FUNCTIONALITY!
ax.set_title("subplot 2 - size 10",fontsize=10)
```

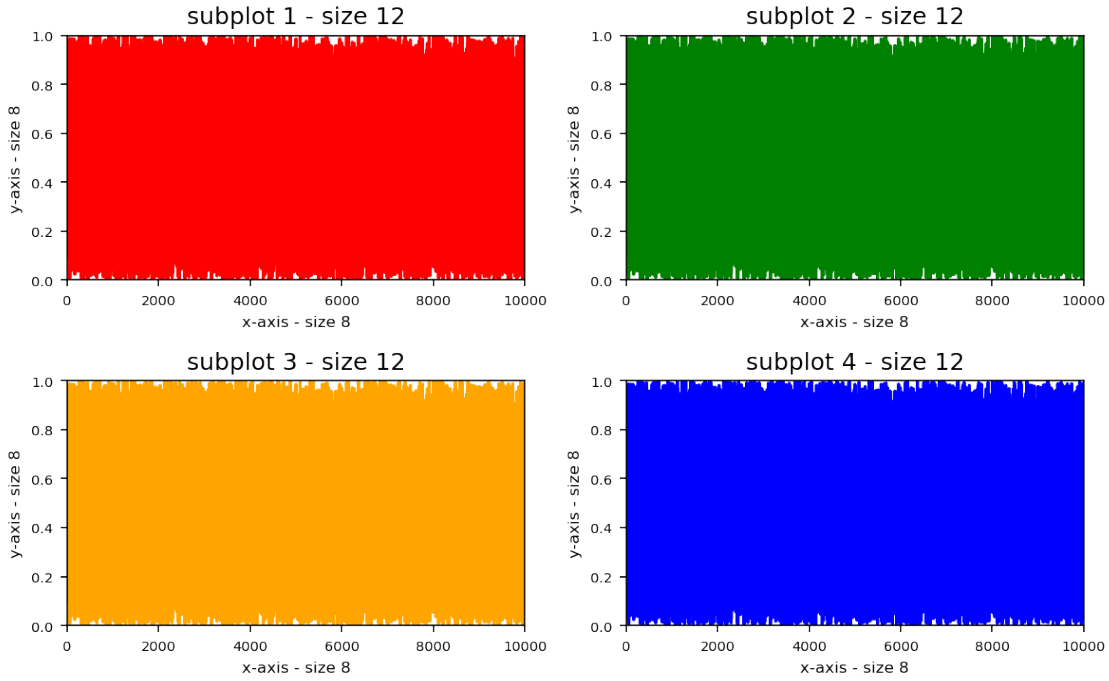
```
ax = fig.add_subplot(2,2,3)
ax.plot(my_data,color="orange")
ax.set_xlim(0,10000)
ax.set_ylim(0,1)
#### NEW FUNCTIONALITY!
ax.set_xlabel("x-axis - size 7",fontsize=7)
#### NEW FUNCTIONALITY!
ax.set_ylabel("y-axis - size 6",fontsize=6)
#### NEW FUNCTIONALITY!
ax.set_title("subplot 3 - size 5",fontsize=5)

ax = fig.add_subplot(2,2,4)
ax.plot(my_data,color="blue")
ax.set_xlim(0,10000)
ax.set_ylim(0,1)
#### NEW FUNCTIONALITY!
ax.set_xlabel("x-axis - size 11",fontsize=11)
#### NEW FUNCTIONALITY!
ax.set_ylabel("y-axis - size 16",fontsize=16)
#### NEW FUNCTIONALITY!
ax.set_title("subplot 4 - size 18",fontsize=18)

#### NEW FUNCTIONALITY!
fig.suptitle("figure title - size 24",fontsize=24)
fig.tight_layout()
```



You get the idea. It's considered good practice to have the figure title as the largest fontsize, then subplot titles, then axis labels, then axis ticks. This helps keep things clean. In the next cell, I'll just remove the figure

title. For most multi-plot figures, you won't need it anyway. And let's adjust the fontsize of the actual ticks, since we didn't do that above.

[9]:
```
fig = plt.figure(figsize=[8,5],dpi=150)

ax = fig.add_subplot(2,2,1)
ax.plot(my_data,color="red")
ax.set_xlim(0,10000)
ax.set_ylim(0,1)
ax.set_xlabel("x-axis - size 8",fontsize=8)
ax.set_ylabel("y-axis - size 8",fontsize=8)
ax.set_title("subplot 1 - size 12",fontsize=12)
#### NEW FUNCTIONALITY!
plt.xticks(fontsize=7)
plt.yticks(fontsize=7)

ax = fig.add_subplot(2,2,2)
ax.plot(my_data,color="green")
ax.set_xlim(0,10000)
ax.set_ylim(0,1)
ax.set_xlabel("x-axis - size 8",fontsize=8)
ax.set_ylabel("y-axis - size 8",fontsize=8)
ax.set_title("subplot 2 - size 12",fontsize=12)
#### NEW FUNCTIONALITY!
plt.xticks(fontsize=7)
plt.yticks(fontsize=7)

ax = fig.add_subplot(2,2,3)
ax.plot(my_data,color="orange")
ax.set_xlim(0,10000)
ax.set_ylim(0,1)
ax.set_xlabel("x-axis - size 8",fontsize=8)
ax.set_ylabel("y-axis - size 8",fontsize=8)
ax.set_title("subplot 3 - size 12",fontsize=12)
#### NEW FUNCTIONALITY!
plt.xticks(fontsize=7)
plt.yticks(fontsize=7)

ax = fig.add_subplot(2,2,4)
ax.plot(my_data,color="blue")
ax.set_xlim(0,10000)
ax.set_ylim(0,1)
ax.set_xlabel("x-axis - size 8",fontsize=8)
ax.set_ylabel("y-axis - size 8",fontsize=8)
ax.set_title("subplot 4 - size 12",fontsize=12)
#### NEW FUNCTIONALITY!
plt.xticks(fontsize=7)
plt.yticks(fontsize=7)

fig.tight_layout()
```

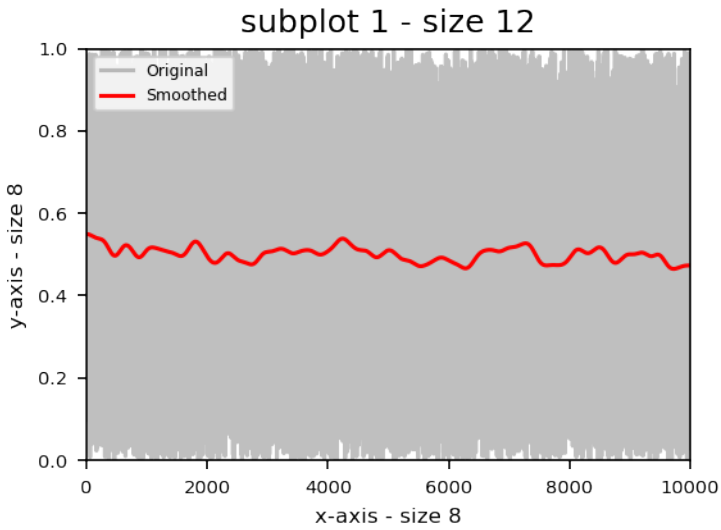


We've figured out how to plot data as a line plot here, but the data happens to be very difficult to read. It's being overwhelmed by the sheer quantity of data here, and ultimately we can't really see much. Fortunately, there are lots of functions we can use to try to smooth the data out, or extract trends. One example is the `gaussian_filter` function included with the SciPy module.

We'll smooth our data using the gaussian filter, then plot it over the original data, which we will color grey and make 50% transparent using the "alpha" keyword. I'll also add labels for each set of data, which will be shown in the legend.

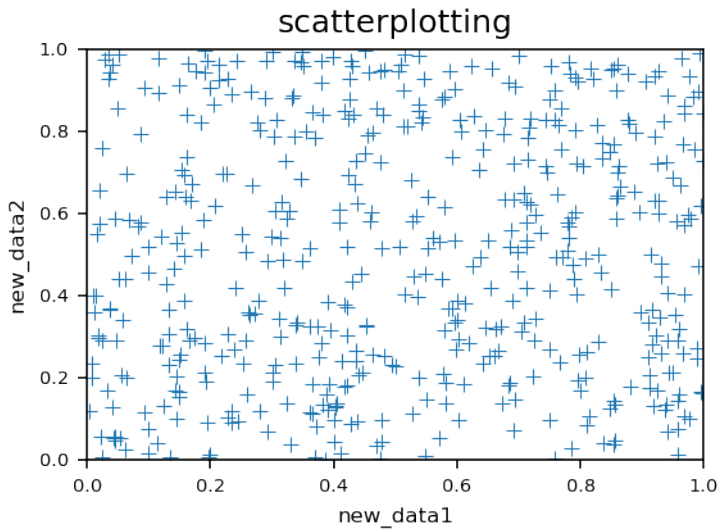
[10]:
```
#### NEW IMPORT!
from scipy.ndimage import gaussian_filter
#### NEW FUNCTIONALITY!
smoothed_data = gaussian_filter(my_data,sigma=100) # here we smooth our original over␣
 ↪100 values.

fig = plt.figure(figsize=[4,3],dpi=150)
ax = fig.add_subplot(1,1,1)
ax.plot(my_data,color="grey",alpha=0.5,label="Original")
#### NEW DATA!
ax.plot(smoothed_data,color="red",label="Smoothed")
ax.set_xlim(0,10000)
ax.set_ylim(0,1)
ax.set_xlabel("x-axis - size 8",fontsize=8)
ax.set_ylabel("y-axis - size 8",fontsize=8)
ax.set_title("subplot 1 - size 12",fontsize=12)
plt.xticks(fontsize=7)
plt.yticks(fontsize=7)
#### NEW FUNCTIONALITY!
ax.legend(loc=2,fontsize=6)
```

```
fig.tight_layout()
```



What if we wanted to plot something that wasn't a dependent variable like above? What if we wanted to compare two different measurements over time to see how they related? A scatter plot might come in handy. First, let's generate some new data. Then we'll use what we've learned above, but use the scatter plot function instead of `plot`, which defaults to a line.

[11]:
```
new_data1=np.random.rand(500)
new_data2=np.random.rand(500)

fig = plt.figure(figsize=[4,3],dpi=150)
ax = fig.add_subplot(1,1,1)
#### NEW FUNCTIONALITY!
ax.scatter(new_data1,new_data2,marker="+",lw=0.5)
ax.set_xlim(0,1)
ax.set_ylim(0,1)
ax.set_xlabel("new_data1",fontsize=8)
ax.set_ylabel("new_data2",fontsize=8)
ax.set_title("scatterplotting",fontsize=12)
plt.xticks(fontsize=7)
plt.yticks(fontsize=7)
fig.tight_layout()
```

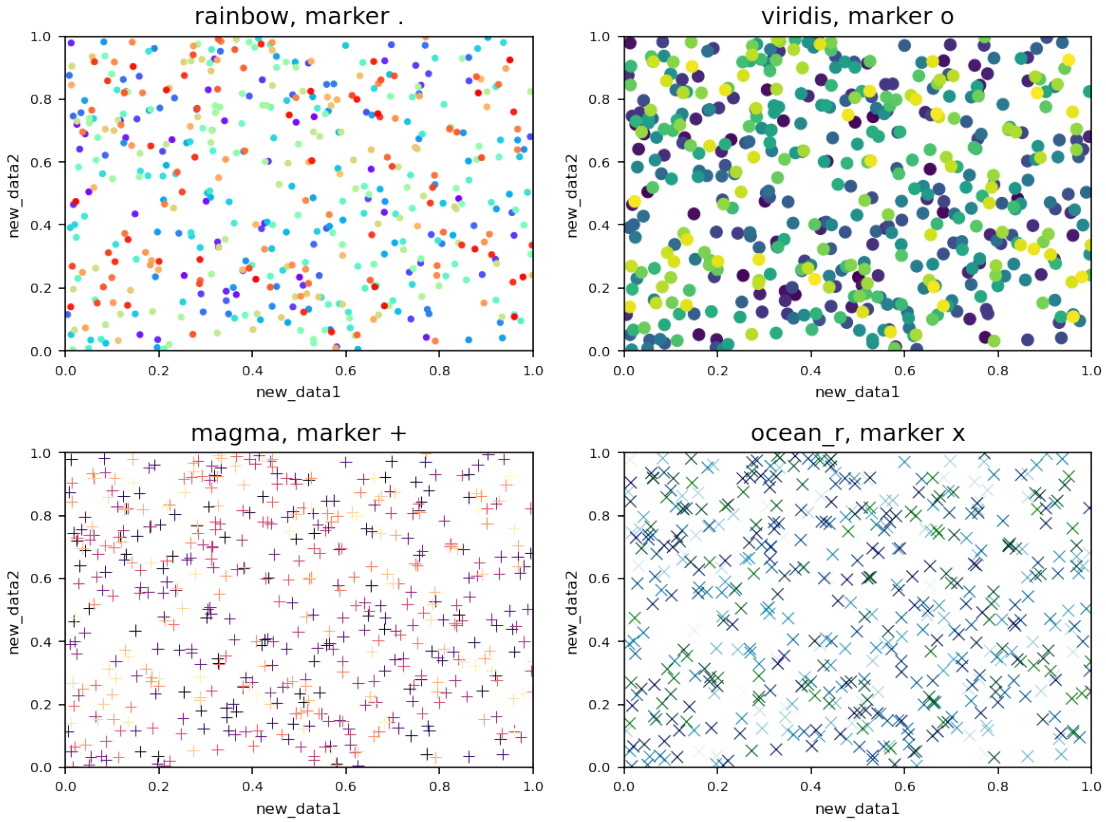


This shows us that our two sets of randomly generated data look ... random. What if we wanted to show the order in which these datapoints occur in the lists, with earlier values shown in one color and later values in a different color?

MatPlotLib has colormaps for this sort of thing! We can color each point based on a value. Since we already have the x and y values in the plot, it seems a waste to color based on those. However, we can create a list, z, made up of 500 values between 0 and 1, using `np.linspace`, and then pass that to the scatter plot function for the color value, along with the `cmap` keyword to choose the specific colormap we want (check out MatPlotLib documentation for a huge selection of colormaps - or for how to build your own!)

[12]:
```
new_data1=np.random.rand(500)
new_data2=np.random.rand(500)
#### NEW FUNCTIONALITY!
z=np.linspace(0,1,500)

fig = plt.figure(figsize=[8,6],dpi=150)
ax = fig.add_subplot(2,2,1)
#### NEW FUNCTIONALITY!
ax.scatter(new_data1,new_data2,marker=".",lw=0.5,c=z,cmap="rainbow")
ax.set_xlim(0,1)
ax.set_ylim(0,1)
ax.set_xlabel("new_data1",fontsize=8)
ax.set_ylabel("new_data2",fontsize=8)
ax.set_title("rainbow, marker .",fontsize=12)
plt.xticks(fontsize=7)
plt.yticks(fontsize=7)

ax = fig.add_subplot(2,2,2)
#### NEW FUNCTIONALITY!
ax.scatter(new_data1,new_data2,marker="o",lw=0.5,c=z,cmap="viridis")
ax.set_xlim(0,1)
ax.set_ylim(0,1)
ax.set_xlabel("new_data1",fontsize=8)
```

```
ax.set_ylabel("new_data2",fontsize=8)
ax.set_title("viridis, marker o",fontsize=12)
plt.xticks(fontsize=7)
plt.yticks(fontsize=7)

ax = fig.add_subplot(2,2,3)
#### NEW FUNCTIONALITY!
ax.scatter(new_data1,new_data2,marker="+",lw=0.5,c=z,cmap="magma")
ax.set_xlim(0,1)
ax.set_ylim(0,1)
ax.set_xlabel("new_data1",fontsize=8)
ax.set_ylabel("new_data2",fontsize=8)
ax.set_title("magma, marker +",fontsize=12)
plt.xticks(fontsize=7)
plt.yticks(fontsize=7)

ax = fig.add_subplot(2,2,4)
#### NEW FUNCTIONALITY!
ax.scatter(new_data1,new_data2,marker="x",lw=0.5,c=z,cmap="ocean_r")
ax.set_xlim(0,1)
ax.set_ylim(0,1)
ax.set_xlabel("new_data1",fontsize=8)
ax.set_ylabel("new_data2",fontsize=8)
ax.set_title("ocean_r, marker x",fontsize=12)
plt.xticks(fontsize=7)
plt.yticks(fontsize=7)


fig.tight_layout()
```

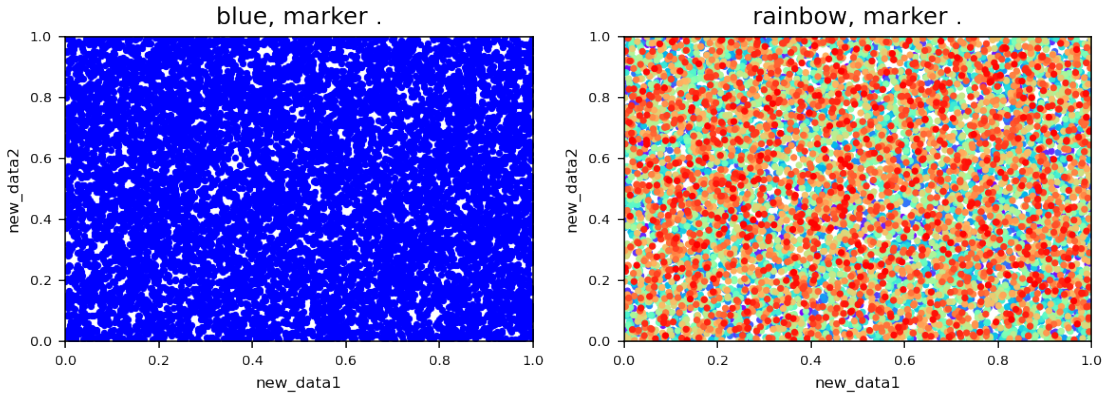


In the example above, you might have noticed one of the colormaps was `ocean_r`. MatPlotLib has the functionality built in to reverse any given colormap by adding `_r` to the name. This offers a quick way to reverse the colors being used for values of interest rather than trying to rearrange your data.

Now, scatter plots are great if your data isn't overly dense for the area in which it lies. In the examples above, we've used 500 data points scattered about. What if that were 10,000 data points?

[13]:
```
new_data1=np.random.rand(10000)
new_data2=np.random.rand(10000)
#### NEW FUNCTIONALITY!
z=np.linspace(0,1,10000)

fig = plt.figure(figsize=[8,3],dpi=150)
ax = fig.add_subplot(1,2,1)
#### NEW FUNCTIONALITY!
ax.scatter(new_data1,new_data2,marker=".",lw=0.5,c="blue")
ax.set_xlim(0,1)
ax.set_ylim(0,1)
ax.set_xlabel("new_data1",fontsize=8)
ax.set_ylabel("new_data2",fontsize=8)
ax.set_title("blue, marker .",fontsize=12)
plt.xticks(fontsize=7)
plt.yticks(fontsize=7)

ax = fig.add_subplot(1,2,2)
```

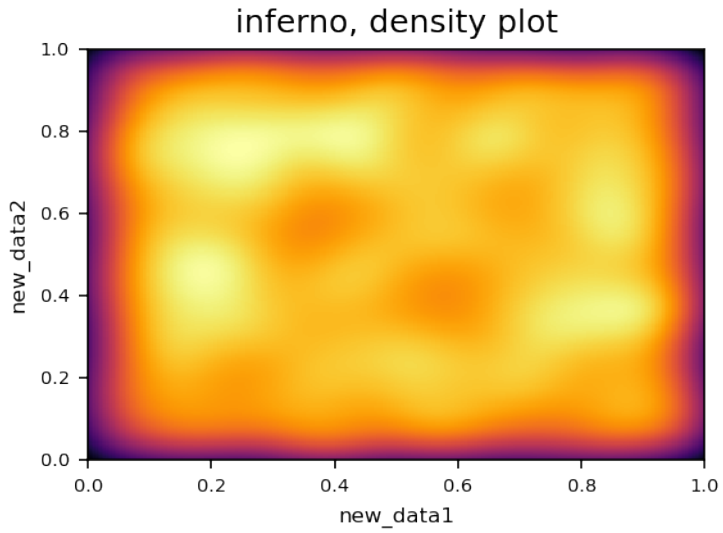
```
#### NEW FUNCTIONALITY!
ax.scatter(new_data1,new_data2,marker=".",lw=0.5,c=z,cmap="rainbow")
ax.set_xlim(0,1)
ax.set_ylim(0,1)
ax.set_xlabel("new_data1",fontsize=8)
ax.set_ylabel("new_data2",fontsize=8)
ax.set_title("rainbow, marker .",fontsize=12)
plt.xticks(fontsize=7)
plt.yticks(fontsize=7)

fig.tight_layout()
```



Even with the rainbow mapping, we simply have too much data to really see what's going on in the plot. Situations such as these might suggest we use a different approach - the Density Plot.

[14]:
```
#### NEW IMPORT!
from scipy.stats import kde
nbins=300
#### NEW FUNCTIONALITY!
k = kde.gaussian_kde([new_data1,new_data2])
X, Y = np.mgrid[new_data1.min():new_data1.max():nbins*1j, new_data2.min():new_data2.
 ↪max():nbins*1j]
Z = k(np.vstack([X.flatten(), Y.flatten()]))


fig = plt.figure(figsize=[4,3],dpi=150)
ax = fig.add_subplot(1,1,1)
#### NEW FUNCTIONALITY!
ax.pcolormesh(X, Y, Z.reshape(X.shape), shading='auto',cmap="inferno")
ax.set_xlim(0,1)
ax.set_ylim(0,1)
ax.set_xlabel("new_data1",fontsize=8)
ax.set_ylabel("new_data2",fontsize=8)
ax.set_title("inferno, density plot",fontsize=12)
plt.xticks(fontsize=7)
plt.yticks(fontsize=7)
```

```
fig.tight_layout()
```

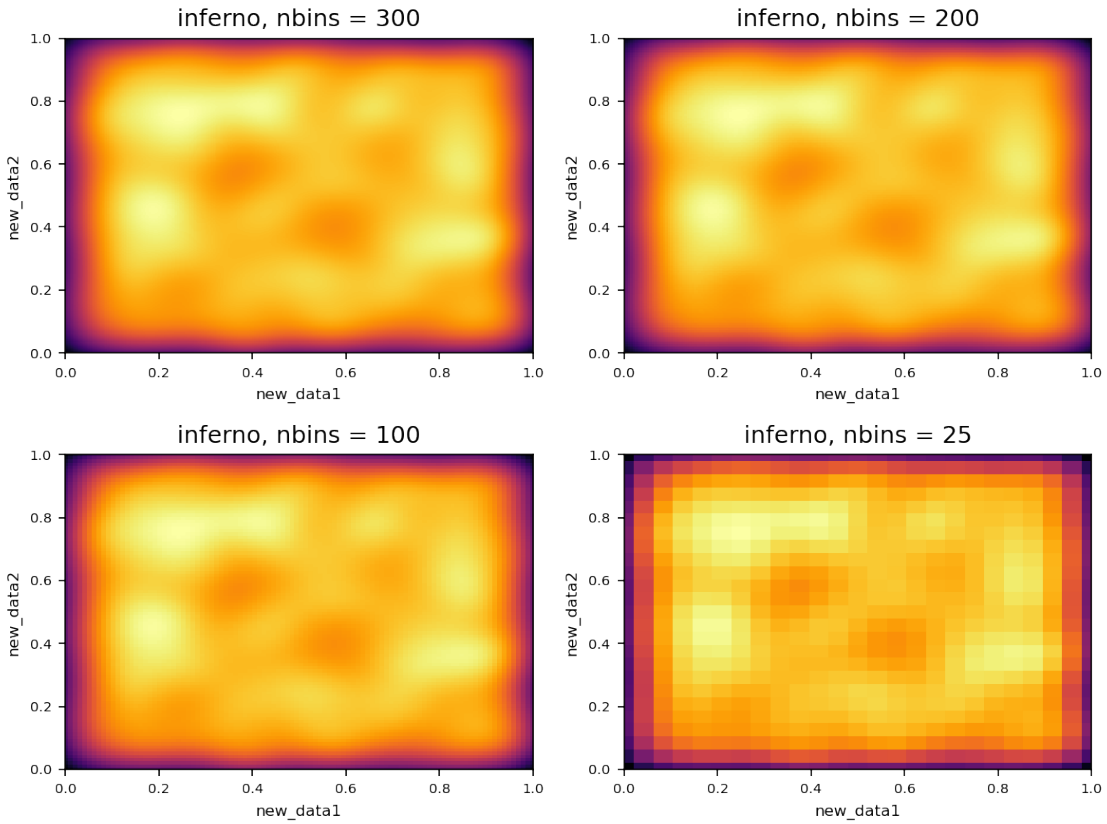


You might have noticed,when running that last cell, that it took a little longer than normal. This is because the data is being processed from individual data points into a mesh where each point is connected to adjacent points. Notice how there are regions of different color shown around the plot. While you cannot distinguish individual data points anymore, you can see trends and regions of higher or lower density of data.

The number of bins affects the smoothness of the density plot, but at the cost of differentiation. The example below shows how different bins produce different plots.

[15]:
```
#### NEW IMPORT!
from scipy.stats import kde
fig = plt.figure(figsize=[8,6],dpi=150)

#### NEW FUNCTIONALITY!
k = kde.gaussian_kde([new_data1,new_data2])
nbins=300
X, Y = np.mgrid[new_data1.min():new_data1.max():nbins*1j, new_data2.min():new_data2.
 ↪max():nbins*1j]
Z = k(np.vstack([X.flatten(), Y.flatten()]))
ax = fig.add_subplot(2,2,1)
ax.pcolormesh(X, Y, Z.reshape(X.shape), shading='auto',cmap="inferno")
ax.set_xlim(0,1)
ax.set_ylim(0,1)
ax.set_xlabel("new_data1",fontsize=8)
ax.set_ylabel("new_data2",fontsize=8)
ax.set_title("inferno, nbins = 300",fontsize=12)
plt.xticks(fontsize=7)
plt.yticks(fontsize=7)


k = kde.gaussian_kde([new_data1,new_data2])
```

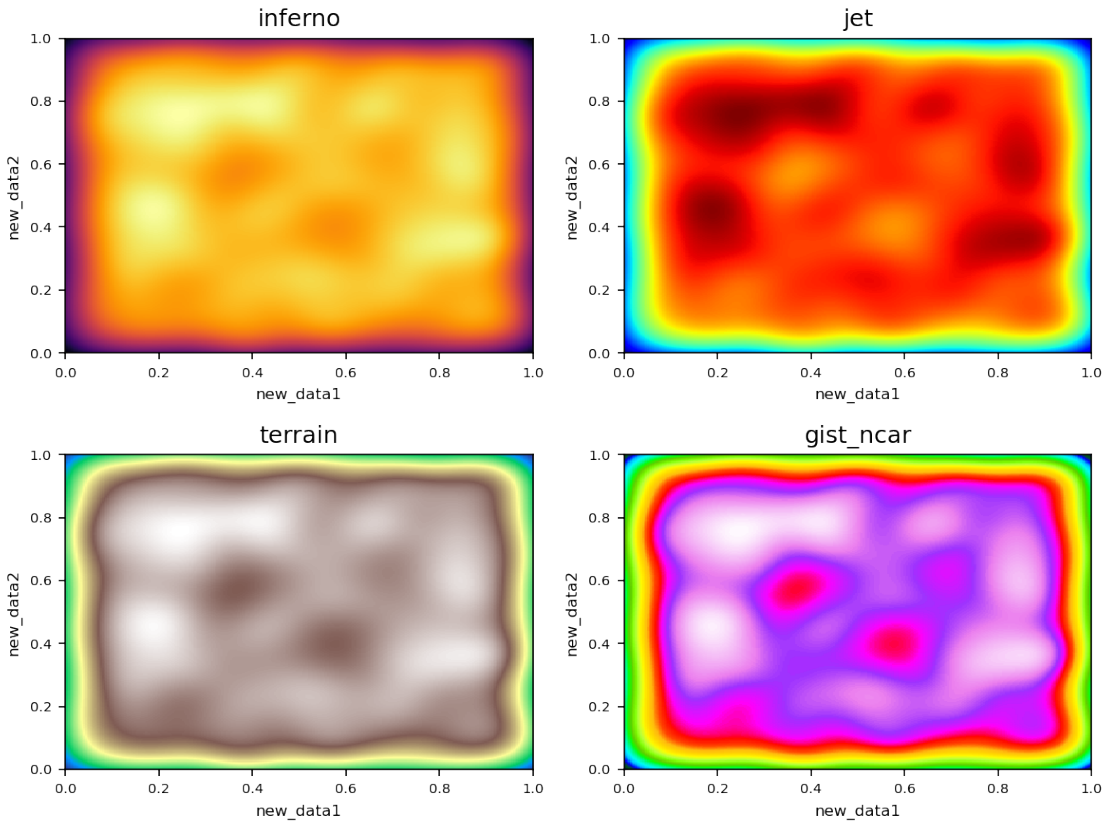
```
nbins=200
X, Y = np.mgrid[new_data1.min():new_data1.max():nbins*1j, new_data2.min():new_data2.
 ↪max():nbins*1j]
Z = k(np.vstack([X.flatten(), Y.flatten()]))
ax = fig.add_subplot(2,2,2)
ax.pcolormesh(X, Y, Z.reshape(X.shape), shading='auto',cmap="inferno")
ax.set_xlim(0,1)
ax.set_ylim(0,1)
ax.set_xlabel("new_data1",fontsize=8)
ax.set_ylabel("new_data2",fontsize=8)
ax.set_title("inferno, nbins = 200",fontsize=12)
plt.xticks(fontsize=7)
plt.yticks(fontsize=7)

k = kde.gaussian_kde([new_data1,new_data2])
nbins=100
X, Y = np.mgrid[new_data1.min():new_data1.max():nbins*1j, new_data2.min():new_data2.
 ↪max():nbins*1j]
Z = k(np.vstack([X.flatten(), Y.flatten()]))
ax = fig.add_subplot(2,2,3)
ax.pcolormesh(X, Y, Z.reshape(X.shape), shading='auto',cmap="inferno")
ax.set_xlim(0,1)
ax.set_ylim(0,1)
ax.set_xlabel("new_data1",fontsize=8)
ax.set_ylabel("new_data2",fontsize=8)
ax.set_title("inferno, nbins = 100",fontsize=12)
plt.xticks(fontsize=7)
plt.yticks(fontsize=7)

k = kde.gaussian_kde([new_data1,new_data2])
nbins=25
X, Y = np.mgrid[new_data1.min():new_data1.max():nbins*1j, new_data2.min():new_data2.
 ↪max():nbins*1j]
Z = k(np.vstack([X.flatten(), Y.flatten()]))
ax = fig.add_subplot(2,2,4)
ax.pcolormesh(X, Y, Z.reshape(X.shape), shading='auto',cmap="inferno")
ax.set_xlim(0,1)
ax.set_ylim(0,1)
ax.set_xlabel("new_data1",fontsize=8)
ax.set_ylabel("new_data2",fontsize=8)
ax.set_title("inferno, nbins = 25",fontsize=12)
plt.xticks(fontsize=7)
plt.yticks(fontsize=7)

fig.tight_layout()
```



You can also try with different colormaps until you find the one that's right for the data you're presenting. For example, the `terrain` colormap runs blue-brown-green-white similar to a topological map showing altitudes!

[16]:
```
#### NEW IMPORT!
from scipy.stats import kde
fig = plt.figure(figsize=[8,6],dpi=150)

#### NEW FUNCTIONALITY!
k = kde.gaussian_kde([new_data1,new_data2])
nbins=300
X, Y = np.mgrid[new_data1.min():new_data1.max():nbins*1j, new_data2.min():new_data2.
 ↪max():nbins*1j]
Z = k(np.vstack([X.flatten(), Y.flatten()]))
ax = fig.add_subplot(2,2,1)
ax.pcolormesh(X, Y, Z.reshape(X.shape), shading='auto',cmap="inferno")
ax.set_xlim(0,1)
ax.set_ylim(0,1)
ax.set_xlabel("new_data1",fontsize=8)
ax.set_ylabel("new_data2",fontsize=8)
ax.set_title("inferno",fontsize=12)
plt.xticks(fontsize=7)
plt.yticks(fontsize=7)
```

```
ax = fig.add_subplot(2,2,2)
ax.pcolormesh(X, Y, Z.reshape(X.shape), shading='auto',cmap="jet")
ax.set_xlim(0,1)
ax.set_ylim(0,1)
ax.set_xlabel("new_data1",fontsize=8)
ax.set_ylabel("new_data2",fontsize=8)
ax.set_title("jet",fontsize=12)
plt.xticks(fontsize=7)
plt.yticks(fontsize=7)

ax = fig.add_subplot(2,2,3)
ax.pcolormesh(X, Y, Z.reshape(X.shape), shading='auto',cmap="terrain")
ax.set_xlim(0,1)
ax.set_ylim(0,1)
ax.set_xlabel("new_data1",fontsize=8)
ax.set_ylabel("new_data2",fontsize=8)
ax.set_title("terrain",fontsize=12)
plt.xticks(fontsize=7)
plt.yticks(fontsize=7)

ax = fig.add_subplot(2,2,4)
ax.pcolormesh(X, Y, Z.reshape(X.shape), shading='auto',cmap="gist_ncar")
ax.set_xlim(0,1)
ax.set_ylim(0,1)
ax.set_xlabel("new_data1",fontsize=8)
ax.set_ylabel("new_data2",fontsize=8)
ax.set_title("gist_ncar",fontsize=12)
plt.xticks(fontsize=7)
plt.yticks(fontsize=7)

fig.tight_layout()
```



Histograms

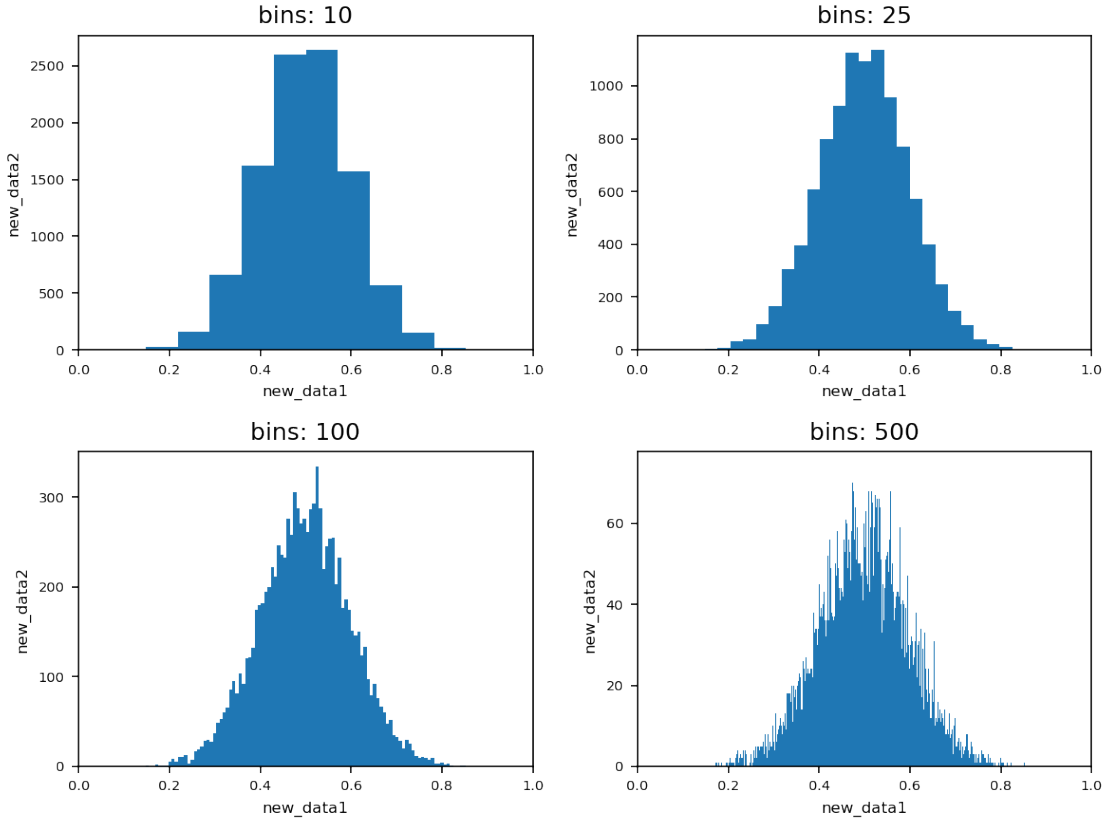
2D Density plots are great for cases where you're comparing two different data sets, but what if you want something similar for just one dataset? You can use a histogram!

In the example below, I've generated random data that follows a normal (gaussian) distribution. See how different numbers of bins can change the shape and structure of the histogram.

Note the y-axis values as well. As we increase the number of bins, the number of points in a given bin goes down on average.

[17]:
```
hist_data=np.random.normal(0.5,0.1,size=10000)
fig = plt.figure(figsize=[8,6],dpi=150)

ax = fig.add_subplot(2,2,1)
ax.hist(hist_data,bins=10)
ax.set_xlim(0,1)
ax.set_xlabel("new_data1",fontsize=8)
ax.set_ylabel("new_data2",fontsize=8)
ax.set_title("bins: 10",fontsize=12)
plt.xticks(fontsize=7)
plt.yticks(fontsize=7)

ax = fig.add_subplot(2,2,2)
ax.hist(hist_data,bins=25)
```

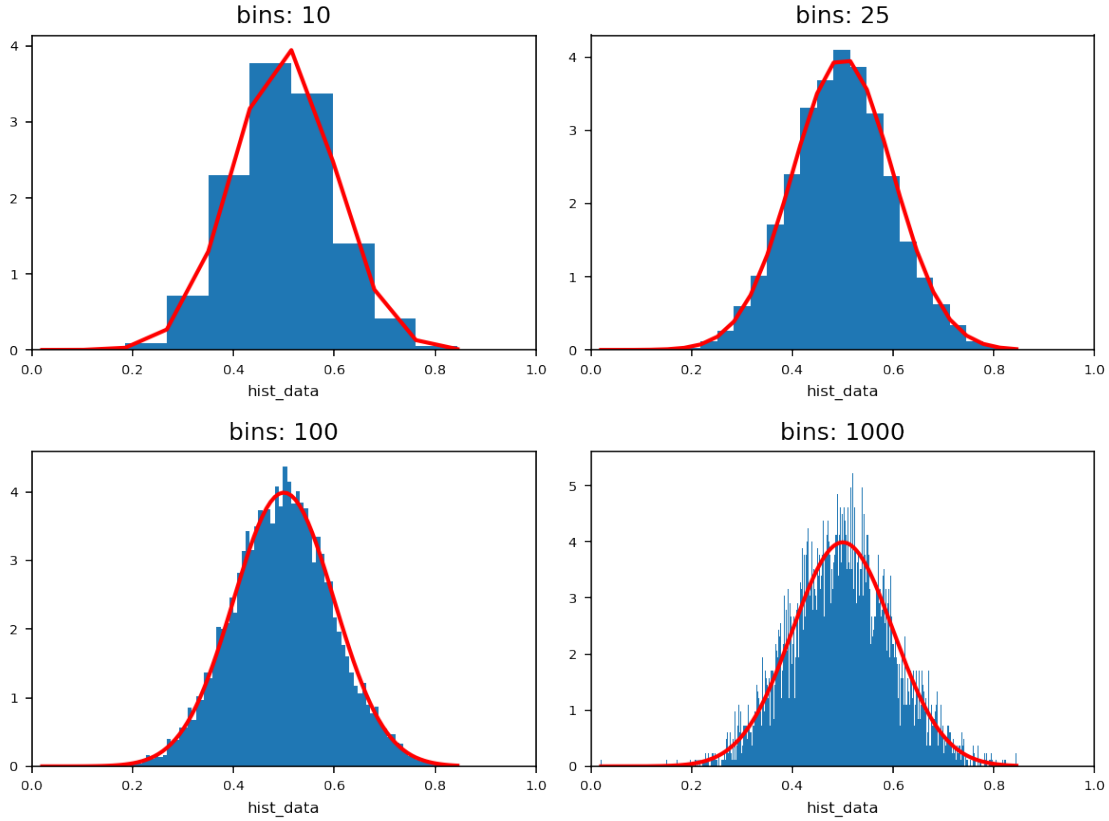
```
ax.set_xlim(0,1)
ax.set_xlabel("new_data1",fontsize=8)
ax.set_ylabel("new_data2",fontsize=8)
ax.set_title("bins: 25",fontsize=12)
plt.xticks(fontsize=7)
plt.yticks(fontsize=7)

ax = fig.add_subplot(2,2,3)
ax.hist(hist_data,bins=100)
ax.set_xlim(0,1)
ax.set_xlabel("new_data1",fontsize=8)
ax.set_ylabel("new_data2",fontsize=8)
ax.set_title("bins: 100",fontsize=12)
plt.xticks(fontsize=7)
plt.yticks(fontsize=7)

ax = fig.add_subplot(2,2,4)
ax.hist(hist_data,bins=500)
ax.set_xlim(0,1)
ax.set_xlabel("new_data1",fontsize=8)
ax.set_ylabel("new_data2",fontsize=8)
ax.set_title("bins: 500",fontsize=12)
plt.xticks(fontsize=7)
plt.yticks(fontsize=7)

fig.tight_layout()
```



Histograms can also be used to plot probability functions on the same graph. I'm also adding the `density=1` keyword argument to the histogram plot command to convert the y-axis values from the number of points in each bin to the percentage of the total. This will result in more consistent ranges and let the probability function line match with the histogram's distribution.

[18]:
```
mu = 0.5
sigma = 0.1
hist_data = np.random.normal(mu,sigma,size=10000)

fig = plt.figure(figsize=[8,6],dpi=150)

ax = fig.add_subplot(2,2,1)
nbins=10
count, bins, ignored = ax.hist(hist_data,bins=nbins, density=1)
plt.plot(bins, 1/(sigma*np.sqrt(2*np.pi))*np.exp(-(bins-mu)**2 /␣
 ↪(2*sigma**2)),linewidth=2, color='r')
ax.set_xlim(0,1)
ax.set_xlabel("hist_data",fontsize=8)
ax.set_title("bins: 10",fontsize=12)
plt.xticks(fontsize=7)
plt.yticks(fontsize=7)

ax = fig.add_subplot(2,2,2)
nbins=25
count, bins, ignored = ax.hist(hist_data,bins=nbins, density=1)
plt.plot(bins, 1/(sigma*np.sqrt(2*np.pi))*np.exp(-(bins-mu)**2 /␣
 ↪(2*sigma**2)),linewidth=2, color='r')
ax.set_xlim(0,1)
ax.set_xlabel("hist_data",fontsize=8)
ax.set_title("bins: 25",fontsize=12)
plt.xticks(fontsize=7)
plt.yticks(fontsize=7)

ax = fig.add_subplot(2,2,3)
nbins=100
count, bins, ignored = ax.hist(hist_data,bins=nbins, density=1)
plt.plot(bins, 1/(sigma*np.sqrt(2*np.pi))*np.exp(-(bins-mu)**2 /␣
 ↪(2*sigma**2)),linewidth=2, color='r')
ax.set_xlim(0,1)
ax.set_xlabel("hist_data",fontsize=8)
ax.set_title("bins: 100",fontsize=12)
plt.xticks(fontsize=7)
plt.yticks(fontsize=7)

ax = fig.add_subplot(2,2,4)
nbins=1000
count, bins, ignored = ax.hist(hist_data,bins=nbins, density=1)
plt.plot(bins, 1/(sigma*np.sqrt(2*np.pi))*np.exp(-(bins-mu)**2 /␣
 ↪(2*sigma**2)),linewidth=2, color='r')
ax.set_xlim(0,1)
ax.set_xlabel("hist_data",fontsize=8)
```

```
ax.set_title("bins: 1000",fontsize=12)
plt.xticks(fontsize=7)
plt.yticks(fontsize=7)

fig.tight_layout()
```

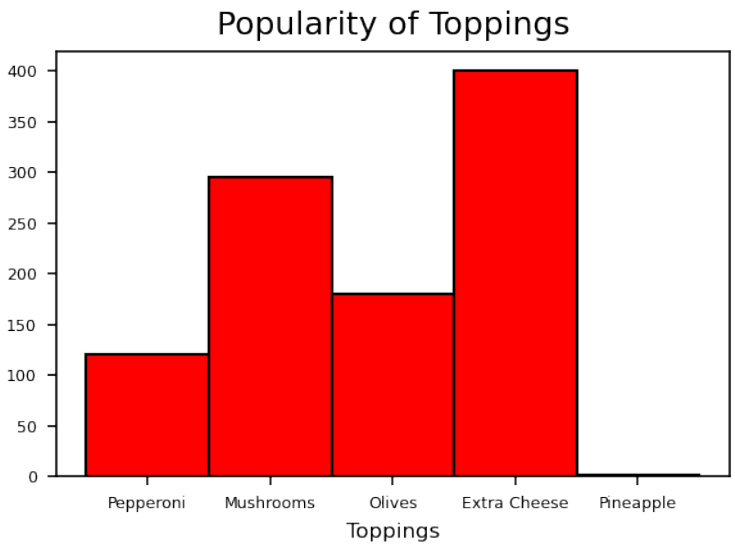


Histograms are a great lead in to bar plots, even if this text is very much not.

Consider some set of values for a bunch of individual things. Let's say you took a poll of a thousand people to figure out what pizza topping is the most popular.

You can simply provide a list of toppings and a list of values and a jab at the people who put pineapple on their pizza and produce a nice barplot!

[31]:
```
toppings = ["Pepperoni","Mushrooms","Olives","Extra Cheese","Pineapple"]
results  = [120,295,180,400,1]
fig = plt.figure(figsize=[4,3],dpi=150)
ax = fig.add_subplot(1,1,1)
ax.bar(toppings,results,width=1.0,color="red",edgecolor="black")
ax.set_xlabel("Toppings",fontsize=8)
ax.set_title("Popularity of Toppings",fontsize=12)
plt.xticks(fontsize=6)
plt.yticks(fontsize=6)

fig.tight_layout()
```



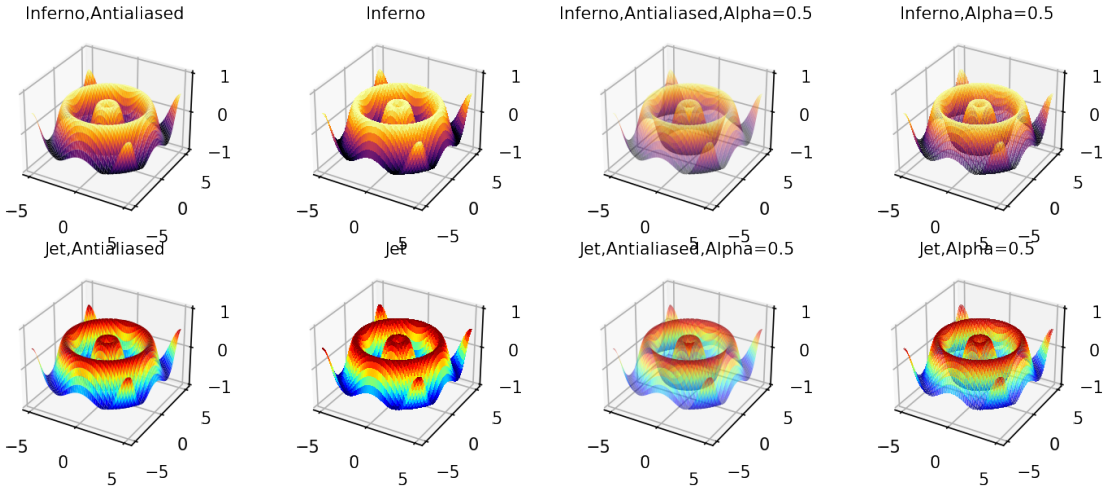
These results are valid and not up for debate. Moving on!

3-Dimensional Surfaces

MatPlotLib has the functionality to produce 3D surfaces. This means you can present values that are dependent on two variables, or even data dependent on 3 variables, using color to signify the value at a given set of x,y,z coordinates in 3D space.

[20]:
```
X = np.arange(-5, 5, 0.005)
Y = np.arange(-5, 5, 0.005)
X, Y = np.meshgrid(X, Y)
R = np.sqrt(X**2 + Y**2)
Z = np.sin(R)*np.cos(R)

fig = plt.figure(figsize=[10,4],dpi=150)
ax = fig.add_subplot(2,4,1,projection="3d")
ax.plot_surface(X, Y, Z, cmap="inferno",linewidth=0, antialiased=True,alpha=1.0)
ax.set_zlim(-1.01, 1.01)
ax.set_title("Inferno,Antialiased",fontsize=10)
ax = fig.add_subplot(2,4,2,projection="3d")
ax.plot_surface(X, Y, Z, cmap="inferno",linewidth=0, antialiased=False,alpha=1.0)
ax.set_zlim(-1.01, 1.01)
ax.set_title("Inferno",fontsize=10)
ax = fig.add_subplot(2,4,3,projection="3d")
ax.plot_surface(X, Y, Z, cmap="inferno",linewidth=0, antialiased=True,alpha=0.5)
ax.set_zlim(-1.01, 1.01)
ax.set_title("Inferno,Antialiased,Alpha=0.5",fontsize=10)
ax = fig.add_subplot(2,4,4,projection="3d")
ax.plot_surface(X, Y, Z, cmap="inferno",linewidth=0, antialiased=False,alpha=0.5)
ax.set_zlim(-1.01, 1.01)
ax.set_title("Inferno,Alpha=0.5",fontsize=10)
```

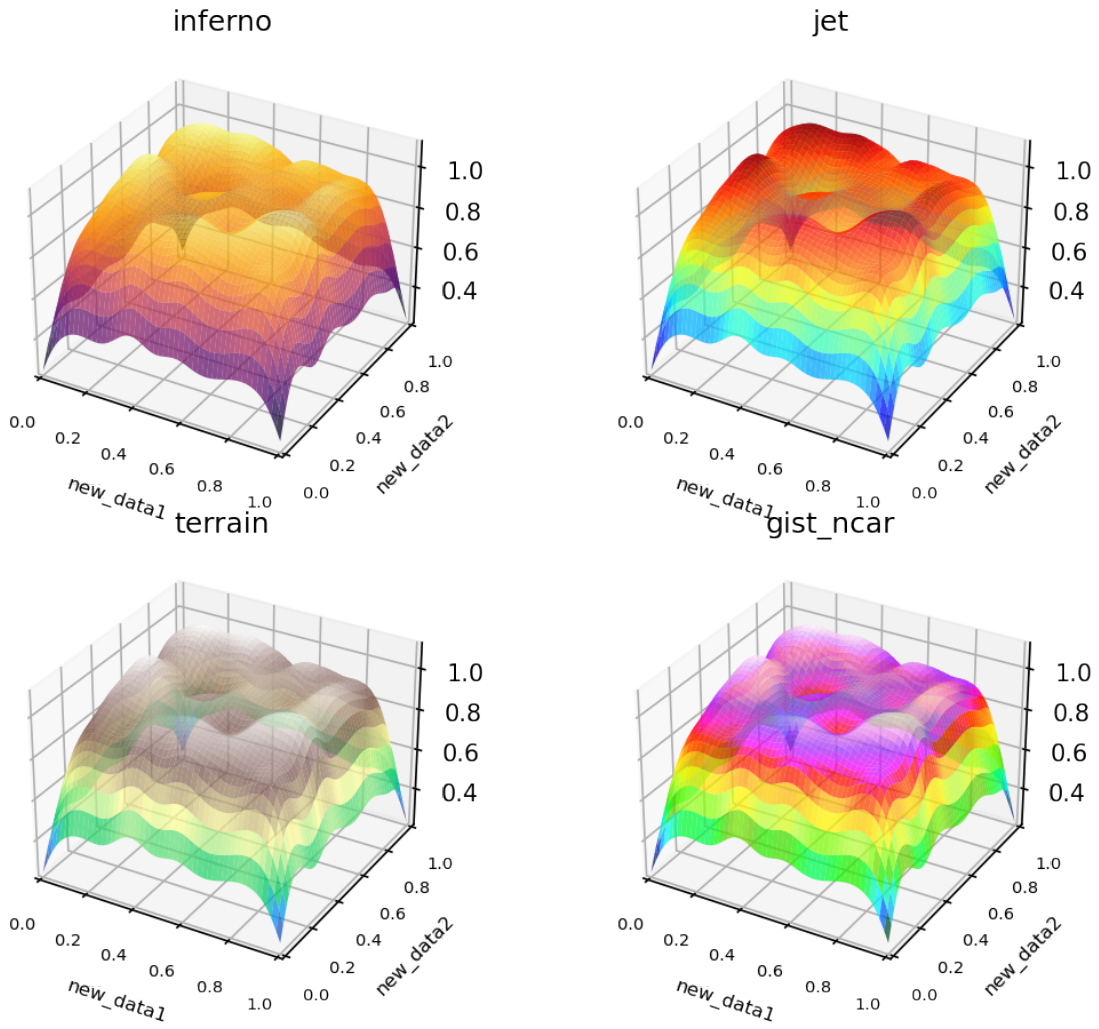
```
ax = fig.add_subplot(2,4,5,projection="3d")
ax.plot_surface(X, Y, Z, cmap="jet",linewidth=0, antialiased=True,alpha=1.0)
ax.set_zlim(-1.01, 1.01)
ax.set_title("Jet,Antialiased",fontsize=10)
ax = fig.add_subplot(2,4,6,projection="3d")
ax.plot_surface(X, Y, Z, cmap="jet",linewidth=0, antialiased=False,alpha=1.0)
ax.set_zlim(-1.01, 1.01)
ax.set_title("Jet",fontsize=10)
ax = fig.add_subplot(2,4,7,projection="3d")
ax.plot_surface(X, Y, Z, cmap="jet",linewidth=0, antialiased=True,alpha=0.5)
ax.set_zlim(-1.01, 1.01)
ax.set_title("Jet,Antialiased,Alpha=0.5",fontsize=10)
ax = fig.add_subplot(2,4,8,projection="3d")
ax.plot_surface(X, Y, Z, cmap="jet",linewidth=0, antialiased=False,alpha=0.5)
ax.set_zlim(-1.01, 1.01)
ax.set_title("Jet,Alpha=0.5",fontsize=10)


fig.tight_layout()
```



That's all well and good as an example, but how does it look with data we've already worked with? Let's try the same process, but we'll use the same dataset from the Density Plots section above.

[21]:
```
#### NEW IMPORT!
from scipy.stats import kde
fig = plt.figure(figsize=[8,6],dpi=150)

#### NEW FUNCTIONALITY!
k = kde.gaussian_kde([new_data1,new_data2])
nbins=300
X, Y = np.mgrid[new_data1.min():new_data1.max():nbins*1j, new_data2.min():new_data2.
 ↪max():nbins*1j]
Z = k(np.vstack([X.flatten(), Y.flatten()]))
ax = fig.add_subplot(2,2,1,projection="3d")
```

```
ax.plot_surface(X, Y, Z.reshape(X.shape),cmap="inferno",linewidth=0,␣
 ↪antialiased=True,alpha=0.75)
ax.set_xlim(0,1)
ax.set_ylim(0,1)
ax.set_xlabel("new_data1",fontsize=8)
ax.set_ylabel("new_data2",fontsize=8)
ax.set_title("inferno",fontsize=12)
plt.xticks(fontsize=7)
plt.yticks(fontsize=7)

ax = fig.add_subplot(2,2,2,projection="3d")
ax.plot_surface(X, Y, Z.reshape(X.shape),cmap="jet",linewidth=0,␣
 ↪antialiased=True,alpha=0.75)
ax.set_xlim(0,1)
ax.set_ylim(0,1)
ax.set_xlabel("new_data1",fontsize=8)
ax.set_ylabel("new_data2",fontsize=8)
ax.set_title("jet",fontsize=12)
plt.xticks(fontsize=7)
plt.yticks(fontsize=7)

ax = fig.add_subplot(2,2,3,projection="3d")
ax.plot_surface(X, Y, Z.reshape(X.shape),cmap="terrain",linewidth=0,␣
 ↪antialiased=True,alpha=0.75)
ax.set_xlim(0,1)
ax.set_ylim(0,1)
ax.set_xlabel("new_data1",fontsize=8)
ax.set_ylabel("new_data2",fontsize=8)
ax.set_title("terrain",fontsize=12)
plt.xticks(fontsize=7)
plt.yticks(fontsize=7)

ax = fig.add_subplot(2,2,4,projection="3d")
ax.plot_surface(X, Y, Z.reshape(X.shape),cmap="gist_ncar",linewidth=0,␣
 ↪antialiased=True,alpha=0.75)
ax.set_xlim(0,1)
ax.set_ylim(0,1)
ax.set_xlabel("new_data1",fontsize=8)
ax.set_ylabel("new_data2",fontsize=8)
ax.set_title("gist_ncar",fontsize=12)
plt.xticks(fontsize=7)
plt.yticks(fontsize=7)

fig.tight_layout()
```



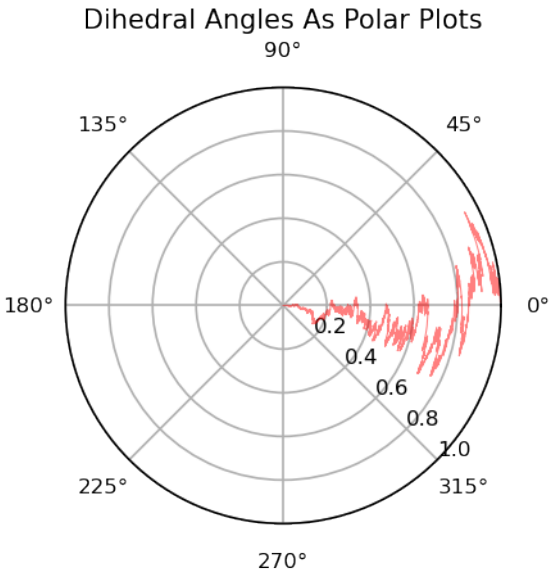
Polar Coordinates

Polar coordinates can come in very handy when you need to plot something with angle values, such as the rotation of a substituent group about a chemical bond. Because the angles are ultimately periodic, it can be important to present the data in such a way.

[30]:
```
### Random Dihedral Angles (each value is randomly adjusted from the previous value)
dihedral_data = [0]
for i in range(10000):
    move = np.random.uniform(-1,1)
    dihedral_data.append(dihedral_data[-1]+move)
```

[29]:
```
# dihedral_data is just an array of dihedrals I got from cpptraj, reported in degrees,␣
↪so I have to change to radians
plt.rcParams["font.size"] = 8

x = np.linspace(0,1,len(dihedral_data))
```

```
fig = plt.figure(figsize=[4,3],dpi=150)
ax = fig.add_subplot(1,1,1,projection='polar')
data_as_radians = [angle/180*np.pi for angle in dihedral_data] ## converting angles␣
↪from degrees to radians
ax.plot(data_as_radians,x,lw=0.5,color="red",alpha=0.5)
ax.set_rmax(1)
ax.set_rlabel_position(-45)  # Move radial labels away from plotted line
ax.grid(True)

ax.set_title("Dihedral Angles As Polar Plots", fontsize=10,y=1.10)

plt.show()

fig.tight_layout()
```



5.2 Library - Numpy

One of the most popular python libraries is `numpy`, which can be found in hundreds of other libraries. NumPy takes advantage of the speed of C++ and FORTRAN code while being usable inside python, making it extremely powerful. As their own website states, "A solution in NumPy is often clear and elegant".

We'll cover some of the basic or common functions below to get started, but keep in mind that there is quite a bit more that numpy can do to make your python-coding life easier. NumPy has no dependencies outside of python itself. It also comes standard with most python installations (like Anaconda). Loading numpy is therefore simple. The convention is to use the code below.

[3]:
```
import numpy as np
```

NumPy is most often associated with the "numpy array", which is effectively a list of elements. However, the `numpy` library takes the array to a higher level and has a lot more included functionality. Let's look at how a basic array can be made with `numpy`.

[8]:
```
my_array = np.arange(0,100)
print(my_array)
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71
 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95
 96 97 98 99]
```

We can see the use of `np.arange` above. It has two required arguments for the start and ending values. Note that in the code above, the end value is 100, but the actual list stops at 99. This means if you actually want 0 to 100, you should set an end value of 101.

We can also add a third value for stride, or how much to increment the value.

[9]:
```
my_array = np.arange(0,100,5)
print(my_array)
```

```
[ 0  5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95]
```

In the example above, we get values that are 5 apart. If we adjust the end point to 101, we'll get the last value (100) as well.

If we want to get a range of values that are evenly spaced between two endpoints *and includes both of those endpoints*, we can use the `linspace` function. The `linspace` function requires three values - start, end, and number of values to return. It is important to keep in mind how the array is created in this function. If a set of N values is requested between X and Y, it will calculate the increment as follows:

$$I = \frac{Y - X}{N - 1}$$

This will produce values of $[X, X + I, X + 2I, ..., X + (N - 1)I, Y]$, which are equally spaced.

The reason this is being explained is shown below. If we wanted 20 values between 0 and 10, we might intuitively think it would give us increments of 0.5, so that the array would look like $[0, 0.5, 1.0, 1.5, ...]$. However, the `linspace` array includes both endpoints in the list.

[13]:
```
my_array = np.linspace(0,10,20)
print(my_array)
```

```
[ 0.         0.52631579  1.05263158  1.57894737  2.10526316  2.63157895
  3.15789474  3.68421053  4.21052632  4.73684211  5.26315789  5.78947368
  6.31578947  6.84210526  7.36842105  7.89473684  8.42105263  8.94736842
  9.47368421 10.        ]
```

[14]:
```
my_array = np.linspace(0,10,21)
print(my_array)
```

```
[ 0.   0.5  1.   1.5  2.   2.5  3.   3.5  4.   4.5  5.   5.5  6.   6.5
  7.   7.5  8.   8.5  9.   9.5 10. ]
```

Another function in `numpy` is the `roll` function. This treats the array like a closed loop and lets you move all elements some number of spaces down, looping end elements back to the start.

[20]:
```
np.roll(my_array,1) # shift everything 1 space over, brings the last element to the␣
↪front.
```

[20]:
```
array([10. ,  0. ,  0.5,  1. ,  1.5,  2. ,  2.5,  3. ,  3.5,  4. ,  4.5,
        5. ,  5.5,  6. ,  6.5,  7. ,  7.5,  8. ,  8.5,  9. ,  9.5])
```

[19]:
```
np.roll(my_array,3) # Shift everything 3 spaces over.
```

[19]:
```
array([ 9. ,  9.5, 10. ,  0. ,  0.5,  1. ,  1.5,  2. ,  2.5,  3. ,  3.5,
        4. ,  4.5,  5. ,  5.5,  6. ,  6.5,  7. ,  7.5,  8. ,  8.5])
```

numpy.linalg

NumPy has a number of submodules that are useful for certain tasks. The `linalg` submodule is short for Linear Algebra, which is focused on vector-, matrix-, and tensor-math functions. Most of the things you learn to calculate by hand in a multivariable calculus course can be done with simple functions in `numpy.linalg`

For example, if you have a vector $[1,2,3]$, you can get the magnitude of the vector by using `linalg.norm`.

[21]:
```
import numpy.linalg as la

my_vector=[1,2,3]
print(la.norm(my_vector))
```

```
3.7416573867739413
```

You can also do more complicated operations like the dot-product or cross-products

np.dot()

[25]:
```
vec1 = [1,2,3]
vec2 = [4,5,6]
dot_product = np.dot(vec1,vec2)
print(dot_product)
```

```
32
```

np.cross()

[36]:
```
mat1 = np.array([[1,2,3],[4,5,6],[7,8,9]])
mat2 = np.array([[1,2],[3,4],[5,6]])
cross_product = np.cross(mat1,mat2)

print(mat1)
print(mat2)
print(cross_product)
```

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
[[1 2]
 [3 4]
 [5 6]]
[[ -6   3   0]
 [-24  18   1]
 [-54  45   2]]
```

Random Number Generators

Numpy is also home to a good selection of random number generators that can provide different random distributions depending on what kind of random number you need. - uniform - normal (Gaussian) - exponential - binomial - etc.

Polynomial Fits

`numpy.polynomial` can provide you with data fitting tools to obtain N-degree polynomials for your data.

5.3 Library - Pandas

Pandas is a wonderful data organization and processing library. It works off of a single major underlying structure, the `DataFrame`. The `DataFrame` is an object that can be described as a "dictionary of dictionaries". Similar to how previous examples have shown that you can have nested dictionary datatypes, Pandas takes it a step further and provides tools to organize, manipulate, combine, compare, and store these multi-layer dictionaries. Pandas also includes built-in Excel Spreadsheet creation functionality, meaning that you can transform your data into more complex structures in Excel, making it easier to share with collaborators.

Let's look at an example of a simple dataframe with just a few rows and columns.

[2]:
```
import pandas as pd

df = pd.DataFrame([{"First Name":"Zee","Age":40,"Team":"Blue"},
                   {"First Name":"Charlotte","Age":45,"Team":"Red"},
                   {"First Name":"Wilbur","Age":50,"Team":"Green"}])
```

[3]:
```
display(df)
```

```
  First Name  Age   Team
0        Zee   40   Blue
1  Charlotte   45    Red
2     Wilbur   50  Green
```

[4]:
```
print(df)
```

```
  First Name  Age   Team
0        Zee   40   Blue
1  Charlotte   45    Red
2     Wilbur   50  Green
```

Notice the difference between `display` and `print` in the example above. It should be pointed out that `display` is functional in notebook environments only, not in terminals. Trying to use `display` in a terminal will result in an error.

You can add to existing dataframes by appending new rows. Each row will be made as a dictionary beforehand.

[5]:
```
newrow = {"First Name":"Iroh","Age":99,"Team":"Red"}
df = df.append(newrow,ignore_index=True)
```

[6]:
```
display(df)
```

```
  First Name  Age   Team
0        Zee   40   Blue
1  Charlotte   45    Red
```

```
2     Wilbur   50  Green
3       Iroh   99    Red
```

Notice that we had to reassign the dataframe when we added the new row. This is because all of the internal dataframe functions return a *new* dataframe object. This method ensures that dataframes are not overwritten on accident, and that data is not lost without intent. Also, we now have two rows that have a shared value in a column. We can sort the rows by specific columns, which can be used to group things together. We can also group by multiple columns to have improved organization.

[7]:
```
df2 = df.sort_values("Team",ascending=True)
display(df2)
```

```
  First Name  Age   Team
0        Zee   40   Blue
2     Wilbur   50  Green
1  Charlotte   45    Red
3       Iroh   99    Red
```

[8]:
```
df3 = df.sort_values(["Team","Age"],ascending=[True,False])
display(df3)
```

```
  First Name  Age   Team
0        Zee   40   Blue
2     Wilbur   50  Green
3       Iroh   99    Red
1  Charlotte   45    Red
```

In the first cell, we sort only by the column "Team", and put the results in ascending order. Pandas uses alphabetical sorting unless all cells in a column are *entirely* numerical.

In the second cell, we sorted by a list, which means the whole dataframe is sorted by the first element, and any rows that have the same value in that first element are then sorted by the second element, and so on. Additionally, we use a second list for the ascending values to set the ascending/descending state for each element being sorted.

What if we wanted just the values in one column? We can call the dataframe like we would a dictionary, using the square brackets and a column name.

[9]:
```
print(df["Age"])
```

```
0    40
1    45
2    50
3    99
Name: Age, dtype: int64
```

[10]:
```
print(df["Age"].values)
```

```
[40 45 50 99]
```

We can also call individual rows with the `.iloc` function. In the example below, `.iloc(0)` indicates we're iterating on the 0th axis, which is down the rows. The `[2]` indicates we want the data at index 2 in that list of rows. In the original `df` dataframe object, that is the third row.

[12]:
```
print(df.iloc(0)[2])
```

```
First Name    Wilbur
Age               50
Team           Green
Name: 2, dtype: object
```

Let's consider an example that might be more relevant to the lab. Mark is currently working on a Automated Fluorescent Nucleotide Workflow (that needs a better name with a fun acronym). Over time, this workflow is intended to produce large amounts of data for hundreds - even thousands - of molecules. We want to use this data in a machine learning model to predict possible new fluorescent nucleotides.

We can use Pandas to organize this data into a more easily managed form.

What are the different things to consider for each of the molecules? - Sugar type (Ribose / Deoxyribose) - Nucleobase (A/C/G/T/U) - Connection point on the base (C6,C5, etc.) - Tag structure and connection point. - Absorption Wavelength - Emission Wavelength - Quantum Yield

Let's assume that we don't have all that information for every single molecule just yet. In fact, that's kind of the point of machine learning, filling in the gaps of data.

[54]:
```
## A list of all the columns we want in our dataframe, corresponding to values we want␣
 ↪to keep track of for each system.
df_columns = ["Sugar","Base","ConnectionPoint","FluorescentTag",␣
 ↪"AbsorptionWavelength","EmissionWavelength","QuantumYield"]

## Initialize the dataframe without data, just column names
nucleotides = pd.DataFrame(columns=df_columns)

display(nucleotides)
```

```
Empty DataFrame
Columns: [Sugar, Base, ConnectionPoint, FluorescentTag, AbsorptionWavelength,␣
 ↪EmissionWavelength, QuantumYield]
Index: []
```

I'm including a function below to make my life easier for the example process, but if we haven't talked about functions yet, you can ignore it for now.

[55]:
```
def add_molecule(df,**kwargs):
    new_molecule = {}
    for key,val in kwargs.items():
        new_molecule[key] = val
    df = df.append(new_molecule,ignore_index=True)
    return df
```

[56]:
```
nucleotides = add_molecule(nucleotides,Sugar="Ribose",Base="C",ConnectionPoint="C5",␣
 ↪AbsorptionWavelength=380,EmissionWavelength=415,QuantumYield=.93)
nucleotides =␣
 ↪add_molecule(nucleotides,Sugar="Ribose",Base="C",FluorescentTag="Perylene",␣
 ↪ConnectionPoint="C6",EmissionWavelength=465,QuantumYield=.96)
nucleotides =␣
 ↪add_molecule(nucleotides,Sugar="Deoxyribose",Base="C",FluorescentTag="Benzopyrene",␣
 ↪ConnectionPoint="C5",AbsorptionWavelength=360,EmissionWavelength=395)
nucleotides =␣
 ↪add_molecule(nucleotides,Sugar="Deoxyribose",Base="C",FluorescentTag="Furan",␣
 ↪ConnectionPoint="C6",AbsorptionWavelength=390,QuantumYield=.75)
```

```
display(nucleotides)
```

```
         Sugar Base ConnectionPoint FluorescentTag AbsorptionWavelength  \
0       Ribose    C              C5            NaN                   380
1       Ribose    C              C6       Perylene                   NaN
2  Deoxyribose    C              C5    Benzopyrene                   360
3  Deoxyribose    C              C6          Furan                   390

  EmissionWavelength  QuantumYield
0                415          0.93
1                465          0.96
2                395           NaN
3                NaN          0.75
```

In the above cell, we can see that empty values are printed as `NaN`. We can clear those out and replace them with empty cells to make it easier to see where the data is missing.

[57]:
```
display(nucleotides.fillna(''))
```

```
         Sugar Base ConnectionPoint FluorescentTag AbsorptionWavelength  \
0       Ribose    C              C5                                  380
1       Ribose    C              C6       Perylene
2  Deoxyribose    C              C5    Benzopyrene                   360
3  Deoxyribose    C              C6          Furan                   390

  EmissionWavelength QuantumYield
0                415         0.93
1                465         0.96
2                395
3                            0.75
```

[58]:
```
display(nucleotides.
 ↪sort_values(["AbsorptionWavelength","EmissionWavelength","QuantumYield"]).fillna(""))
```

```
         Sugar Base ConnectionPoint FluorescentTag AbsorptionWavelength  \
2  Deoxyribose    C              C5    Benzopyrene                   360
0       Ribose    C              C5                                  380
3  Deoxyribose    C              C6          Furan                   390
1       Ribose    C              C6       Perylene

  EmissionWavelength QuantumYield
2                395
0                415         0.93
3                            0.75
1                465         0.96
```

Let's generate some random data to mess with.

[71]:
```
import numpy as np
base_list=["A_C2","A_C8","C_C5","C_C6","G_C2","G_C8","G_N7","U_C6","U_C5","T_C6"]
sugar_list=["Ribose","Deoxyribose"]
tag_list = ["Perylene","Benzopyrene","Furan","Naphthalene","Beta Carotene","Imidazole"]
nucleotides = pd.DataFrame(columns=df_columns)
for base in base_list:
    for sugar in sugar_list:
```

```
        for tag in tag_list:
            abs_wl = np.random.randint(355,800)
            emi_wl = np.random.randint(abs_wl,850)
            nucleotides = add_molecule(nucleotides,Sugar=sugar,Base=base.
↪split("_")[0],ConnectionPoint=base.
↪split("_")[1],FluorescentTag=tag,AbsorptionWavelength=abs_wl,EmissionWavelength=emi_wl,QuantumYield=
↪random.rand(),2))
```

[72]:
```
display(nucleotides.fillna(""))
```

```
           Sugar Base ConnectionPoint FluorescentTag  AbsorptionWavelength  \
0         Ribose    A              C2       Perylene                   561
1         Ribose    A              C2    Benzopyrene                   664
2         Ribose    A              C2          Furan                   604
3         Ribose    A              C2    Naphthalene                   424
4         Ribose    A              C2  Beta Carotene                   774
..           ...  ...             ...            ...                   ...
115  Deoxyribose    T              C6    Benzopyrene                   602
116  Deoxyribose    T              C6          Furan                   531
117  Deoxyribose    T              C6    Naphthalene                   489
118  Deoxyribose    T              C6  Beta Carotene                   697
119  Deoxyribose    T              C6      Imidazole                   389

     EmissionWavelength  QuantumYield
0                   828          0.65
1                   771          0.55
2                   737          0.09
3                   734          0.61
4                   791          0.01
..                  ...           ...
115                 808          0.38
116                 776          0.91
117                 639          0.68
118                 757          0.11
119                 529          0.09

[120 rows x 7 columns]
```

Note here that the full dataframe is not shown, but rather displays only the first five and last five rows. The dimensions of the dataframe are given below it. In this case, it's 120 rows and 7 columns.

What if we wanted to sort the data by Quantum Yield?

[73]:
```
nucleotides.sort_values("QuantumYield",ascending=False)
```

[73]:
```
           Sugar Base ConnectionPoint FluorescentTag AbsorptionWavelength  \
83   Deoxyribose    G              N7      Imidazole                   587
92   Deoxyribose    U              C6          Furan                   724
21   Deoxyribose    A              C8    Naphthalene                   513
70   Deoxyribose    G              C8  Beta Carotene                   643
29        Ribose    C              C5      Imidazole                   450
..           ...  ...             ...            ...                   ...
44   Deoxyribose    C              C6          Furan                   588
89        Ribose    U              C6      Imidazole                   415
```

```
26        Ribose    C              C5          Furan                  761
110       Ribose    T              C6          Furan                  419
4         Ribose    A              C2  Beta Carotene                  774

    EmissionWavelength  QuantumYield
83                 740          1.00
92                 846          0.99
21                 765          0.98
70                 758          0.96
29                 500          0.95
..                 ...           ...
44                 828          0.06
89                 707          0.03
26                 797          0.01
110                472          0.01
4                  791          0.01

[120 rows x 7 columns]
```

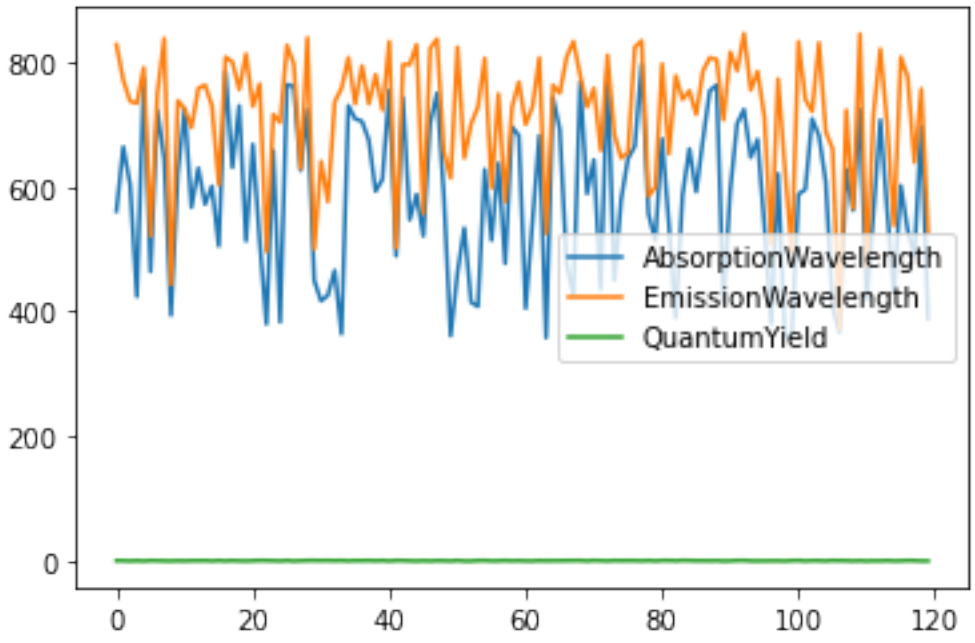
Keep in mind that the above result did not sort the dataframe in itself, it just presented the results of a sorting algorithm applied to it. Note the indices on the left side and how they're out of order.

Now, what if we wanted to plot some of the data from our dataframe?

If our dataframe is simple enough, we can just call the `plot()` function directly from it, which returns a `matplotlib Axes` object.

[83]:
```
import matplotlib.pyplot as plt
fig = plt.figure(figsize=[5,4],dpi=300)
ax = nucleotides.plot()
```

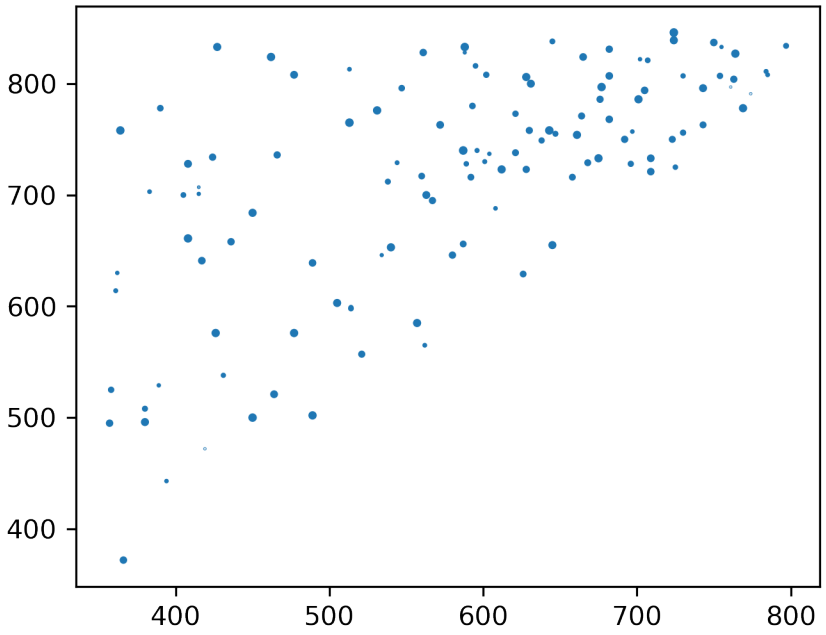
```
<Figure size 1500x1200 with 0 Axes>
```



We can also simply call columns of our data as though they were any other array of numbers (assuming the column is, in fact, strictly numerical).

[85]:
```
import matplotlib.pyplot as plt
fig = plt.figure(figsize=[5,4],dpi=300)
ax = fig.add_subplot(1,1,1)
ax.scatter(nucleotides["AbsorptionWavelength"],nucleotides["EmissionWavelength"],s=␣
 ↪nucleotides["QuantumYield"]*5)

### I have included the QuantumYield column values as the marker size variable so that␣
 ↪each point gives us the absorption/emission wavelengths and the quantum yields.
```

[85]:
```
<matplotlib.collections.PathCollection at 0x7f45f705cfd0>
```



Chapter 6

Functions

6.1 Basic Functions

A function in programming is similar to a function in math - variables go in, operations occur, results come out. In fact, I passed all my higher level math classes by learning to reinterpret complex mathematical equations as programming functions. For example, if you wanted to obtain the sum of all integers between 0 and 1000, you would do the following:

Math

$$\sum_{x=0}^{1000} x$$

Python

```
total = 0
for x in range(0,1001):
    total += x
print(total)
```

C++

```
int total = 0;
for(int x=0;x<=1000;x++)
{
    total += x;
}
cout << total << endl;
```

The functions above produce the same value. They are simply different representations of the same algorithm.

What do you notice that's different between these two programming languages?

If you find there's a task or set of instructions that you do multiple times in a program/script, you might find it useful to convert it into a function.

Functions have the benefit of ensuring that the calculations are being performed consistently - you only have to get the code right once!

However, it's also important to consider if something is **worth** turning into a function. Whenever you think about building a function, the first thing to do is see if it already exists. This is a fundamental rule for

programming and science - don't reinvent the wheel unless you can make it a lot better.

Python Functions

For today, we'll focus on python functions.

A function to return the remainder of one number divided by another. That might look somethink like this:

```
def get_remainder(number,divisor):
    if number > divisor:
        return get_remainder(number-divisor,divisor)
    else:
        return number
```

Experienced programmers will look at the function above and consider hunting me down. Rightfully so, for multiple reasons. First and most importantly, the function is unnecessary. There already exists a function to get the remainder of a number and a divisor, called "modulo", and is called using % like so:

[1]:
```
23%5
```

[1]:
```
3
```

That is so much easier than writing the function above. However, that's not the only reason the function above will drive programmers nuts: **Recursion**

Recursive functions are any function that calls itself during its execution. While there are some situations where recursion may be necessary, There is nearly always a cleaner solution.

The problems with recursion lie in the fact that such a function can call itself infinitely if not well-written, and can become computationally expensive and slow. So if you ever find yourself writing a function that calls itself, get another opinion from someone else.

Okay, let's start at the basics of writing a function in Python

Every function must be *defined* before it can be *called* (used). In Python, a function definition begins with the command `def`, followed by the name of the function and then any function *arguments* it will need. *Arguments* are named variables that will be used inside the function. This means that any needed variables that aren't given to the function as an argument must be created inside the function. After the arguments, you need a `:` to indicate that the following code is what should be executed whenever the function is called. It's also good practice to include the `return` command at the end of the function, even if nothing is sent back to where the function was called.

An example is below.

[2]:
```
def myfunction(argument1,argument2,argument3):
    print(argument1)
    print(argument2)
    print(argument3)
    return
```

This is a fairly simple function, and now that we've defined it, we can call it as much as we want.

[3]:
```
myfunction("potato","nebula","agreeable")
```

```
potato
nebula
agreeable
```

[4]:
```
myfunction(4,78.951,9001)
```

```
4
78.951
9001
```

Note that the two different calls above pass different types of data to the function, but the function is able to handle them all. Let's try a different function that does some math on the variables we pass it.

[5]:
```
def mymathfunction(arg1,arg2,arg3):
    result = arg1 + arg2 - arg3
    return result
```

Now, looking at the function above, we expect it to use numbers as arguments.

[6]:
```
answer = mymathfunction(5,10,3)
print(answer)
```

```
12
```

What if we passed data that wasn't a number?

[7]:
```
answer = mymathfunction(5,10,"cactus")
print(answer)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [7], in <cell line: 1>()
----> 1 answer = mymathfunction(5,10,"cactus")
      2 print(answer)

Input In [5], in mymathfunction(arg1, arg2, arg3)
      1 def mymathfunction(arg1,arg2,arg3):
----> 2     result = arg1 + arg2 - arg3
      3     return result

TypeError: unsupported operand type(s) for -: 'int' and 'str'
```

Now we get a type error. Fortunately, we can rewrite the function to include "type hints". If we add `:float` after the variable names in the function definition, we're telling the programmer "This variable should be of the `float` data type". Please note that it doesn't actually stop the user from passing incorrect data types to the function.

[8]:
```
def mymathfunction(arg1:float, arg2:float, arg3:float):
    result = arg1 + arg2 - arg3
    return result
```

[9]:
```
answer = mymathfunction(5,10,"cactus")
print(answer)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [9], in <cell line: 1>()
----> 1 answer = mymathfunction(5,10,"cactus")
```

```
      2 print(answer)

Input In [8], in mymathfunction(arg1, arg2, arg3)
      1 def mymathfunction(arg1:float, arg2:float, arg3:float):
----> 2     result = arg1 + arg2 - arg3
      3     return result

TypeError: unsupported operand type(s) for -: 'int' and 'str'
```

Recall the booleans we discussed yesterday. We can use them along with a *type check* in our function to first make sure the arguments to our function are of the correct type.

[10]:
```
# declare a variable as an integer.
x = 5

# check if it's an integer.
print(isinstance(x,int))

# check if it's a float.
print(isinstance(x,float))
```

```
True
False
```

We can see how the `isinstance` function (which doesn't need to be defined because it's already part of the base python) takes the variable we give it and a type, and returns a boolean response. Note that the `int` and `float` checks return different results. However, as programmers we know that python math with integers and floats is (generally) interchangable.

How can we use this to guard our functions?

In the example below, we will create an empty list for the type checks, then add the result of each check to the list. Then, if the whole list is `True`, we will proceed with the function. If it's not all `True`, we'll exit the function without doing anything.

[11]:
```
def mymathfunction(arg1:float, arg2:float, arg3:float):
    # make an empty list
    variable_type_checks = []
    # check if arg1 is a float or an integer and add the result to the list
    variable_type_checks.append( any( [isinstance(arg1,float), isinstance(arg1,int)] )␣
 ↪)
    # check if arg2 is a float or an integer and add the result to the list
    variable_type_checks.append( any( [isinstance(arg2,float), isinstance(arg2,int)] )␣
 ↪)
    # check if arg3 is a float or an integer and add the result to the list
    variable_type_checks.append( any( [isinstance(arg3,float), isinstance(arg3,int)] )␣
 ↪)

    # check if the entire list is "True"
    if all(variable_type_checks):
        result = arg1 + arg2 - arg3
        return result
    # if the above fails (the if-condition is not met), we'll do the next section␣
 ↪instead.
```

```
    elif any(variable_type_checks):
        print("Some arguments are not floats.")
        return None
    # if nothing above worked, do this.
    else:
        print("Every argument is not a float!")
        return None
```

This function is a bit larger, but it's including additional checks and conditions to ensure it works correctly.

[12]:
```
answer = mymathfunction(5,10,3)
print(answer)
```

```
12
```

[13]:
```
answer = mymathfunction(5,10,"cactus")
print(answer)
```

```
Some arguments are not floats.
None
```

[14]:
```
answer = mymathfunction("cheese","crater","cactus")
print(answer)
```

```
Every argument is not a float!
None
```

We have the three possible outcomes for this function above. In all cases, the function executed without throwing an error (which can cause the program to crash entirely), and each returned a result.

The two "failing" function calls returned the `None` datatype. In C++, this is called `NULL`. Both effectively mean there's an empty spot. However, we can still treat `None` like a data type.

[15]:
```
answer = mymathfunction("cheese","crater","cactus")

answer == None
```

```
Every argument is not a float!
```

[15]:
```
True
```

We can use the `None` comparison to check results, which can be useful in larger functions or programs.

What if there are external values that you need to change inside the function? By default, when you pass arguments to functions in Python, you're not passing the variable itself, but a copy of the value inside. Consider the function below.

[16]:
```
def f(x):
    x = x+10
    print(x)

# set x to a value
x = 5
# print the value of x
print(x)
# call the function f(x), which internally changes the value of x and then prints it
```

```
f(x)
# print the value of x
print(x)
```

```
5
15
5
```

Now, here we can see that the value of `x` *outside* the function `f(x)` never changes, even though it's changed inside. This is an example of *scope* that we discussed yesterday. What if we wanted to change the value of `x` with the function?

We can use `global` to pull variables in from the outside and use/modify them. This is particularly useful when you're wanting to change a large dataset, like a list of xyz-coordinates for a bunch of particles (as a completely innocent and totally-not-foreshadowing example).

[17]:
```
def f():
    global y
    y = y+10

y=7
print(y)
f()
print(y)
```

```
7
17
```

Now we see how the function `f()` without any arguments was able to modify the value of `y` outside itself.

Now that you know about `global`, be aware that it can be very dangerous if used improperly.

In Python, functions can pull external things inside with `global`, but nothing can be pulled from inside a function except things the function *returns*.

You can also nest function calls inside other functions, which can be useful as well.

[18]:
```
def f(x):
    return x**2 + 7

def g(x):
    return x*f(x)

print(g(-3))
print(g(4))
```

```
-48
92
```

That's a very simple example, but you can see how having smaller functions can lend itself to a modularity and reusability that can become very convenient.

6.2 Intermediate Functions - args and kwargs

What if you wanted to give a function an unknown number of arguments? Say, for example, you wanted to give it some random number of `float` values and have it multiply the reciprocals of each together? The mathematical formula would look like this:

$$\prod_{i=1} \frac{1}{x_i}$$

The Python function can use `*args` as an argument. The `*` indicates the value is going to be some arbitrary length list of values that should be collected by the function.

[29]:
```
def product_of_reciprocals(*args):
    # Start with the product set to 1 (since anything multiplied by one is itself).
    product=1
    for arg in args: # iterate through all the values given in the function call
        product = product * (1/arg) # multiply the current product value by the␣
↪reciprocal of the current arg value, and assign it to the product value
    return product
```

[30]:
```
product_of_reciprocals()
```

[30]:
```
1
```

[31]:
```
product_of_reciprocals(3,6,95)
```

[31]:
```
0.0005847953216374268
```

[32]:
```
product_of_reciprocals(1,1,2,3,5,8,11)
```

[32]:
```
0.0003787878787878788
```

[36]:
```
product_of_reciprocals(0.1, 0.2, 0.5, 1.0, 1.5, 2.0, 5.0, 7.0, 10.0002)
```

[36]:
```
0.09523619051428493
```

With `*args`, any number of values may be passed to the function.

What about unknown and arbitrary number of arguments that must be assigned to specific keywords?

Here, we can use `**kwargs`, which is short for "keyword arguments". This specifically requires that each additional argument be given as a variable and an assignment. These can be useful if there are specific things you want your function to do, but only if those arguments are present.

The `**` indicates that `kwargs` will be a `dictionary` data type, which is a list of mapped keys and values. Therefore, using the individual keywords requires a little knowledge of dictionary manipulation.

[39]:
```
def PrintKwargs(**kwargs):
    for key,val in kwargs.items():
        print("The",key,"says",val)
    return
```

[41]:
```
PrintKwargs(chicken="bawk",cow="moo",farmer="it looks like rain",dog="woof")
```

```
The chicken says bawk
The cow says moo
```

```
The farmer says it looks like rain
The dog says woof
```

What if we include an argument without assigning it to a keyword?

[43]:
```
PrintKwargs(chicken="bawk",cow="moo",farmer="it looks like rain",dog="woof","nothing␣
↪assigned")
```

```
  File "/tmp/ipykernel_98113/3123196330.py", line 1
    PrintKwargs(chicken="bawk",cow="moo",farmer="it looks like rain",dog="woof","nothin␣
↪assigned")
↪          ^
SyntaxError: positional argument follows keyword argument
```

Not great. In this case, you get an error because there's something passed to the function that isn't a keyword argument. As an aside, a "positional argument" is just the regular arguments we worked with in the examples before `*args` and `**kwargs`.

What if we wanted to account for keyword arguments AND unassigned arguments?

You can use positional arguments, `*args`, and `**kwargs` in your function calls, so long as they're in that order.

Positional arguments may also have default values assigned to them in the function definition. Any default variables should be placed at the end of the *positional arguments*, but before the `*args` and `**kwargs`.

[1]:
```
def BigFunction(x,y,z=10,*args,**kwargs):
    print("x is",x)
    print("y is",y)
    print("z is",z)
    for arg in args:
        print("Found arg: ",arg)
    for key,value in kwargs.items():
        print("keyword",key,"is",value)
```

[2]:
```
BigFunction(3,4,6,"bear","goat","llama","emu","shark",potato="mashed",dinner="ready")
```

```
x is  3
y is  4
z is 6
Found arg:  bear
Found arg:  goat
Found arg:  llama
Found arg:  emu
Found arg:  shark
keyword potato is mashed
keyword dinner is ready
```

One thing to keep in mind is that positional arguments get assigned before anything gets dumped into `*args`. Above, even though `z` has a default value, it got assigned a value of 6 from the function call because 6 was in that position. Everything afterwards was combined into `*args`.

6.3 Advanced Functions - Decorators and Wrappers

You can nest functions inside other functions. In this way, you can contain these nested functions to ensure they're only accessible inside the outer functions. This can be useful if you want to ensure that any data being used is processed in a specific way beforehand, depending on the way you call the function.

[7]:
```
def outer_function(x):
    def inner_function1(x):
        print(x,"was given.")
    def inner_function2(x):
        print(x**2,"is x squared")
    inner_function1(x)
    inner_function2(x)

outer_function(4)
```

```
4 was given.
16 is x squared
```

As you can see, `outer_function` calls its own `inner_functions` without issue. Let's try calling the `inner_functions` directly.

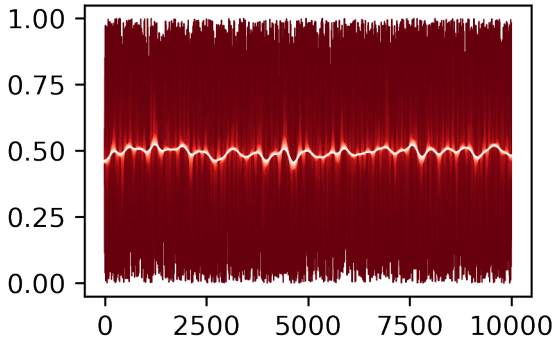
[8]:
```
inner_function1(4)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/tmp/ipykernel_269786/3089592963.py in <module>
----> 1 inner_function1(4)

NameError: name 'inner_function1' is not defined
```

Now we get an error. This is because the inner functions only exist in the scope of the outer function. These small examples illustrate the concept in a basic way, but let's consider a more complicated function.

[29]:
```
import matplotlib.pyplot as plt
from scipy.ndimage import gaussian_filter
import numpy as np
import matplotlib.cm as cm

def plot_with_smoothing(data,sigma):
    """Takes data and a smoothing factor sigma and plots the raw data plus the␣
 ↪smoothed data"""
    cmap = cm.get_cmap("Reds_r")
    fig = plt.figure(figsize=[3,2],dpi=300)
    ax = fig.add_subplot(1,1,1)
    def smoothing(data,sigma,color):
        """Plots the smoothed data"""
        ax.plot(gaussian_filter(data,sigma=sigma),color=color,lw=0.5)
    # ax.plot(data,color="grey",lw=0.5)
    for i in range(sigma):
        smoothing(data,i,cmap(i/sigma))

data = np.random.rand(10000)
```

```
plot_with_smoothing(data,100)
```



Internally defined functions can also allow you to use the same function name in many different contexts. It can be helpful when you have functions that are doing some small task a large number of times, but in a different way depending on the context. In the above example, `smoothing` works on 1-dimensional data. There are other occasions where you may have 2-dimensional data, which requires different techniques to smooth properly. Therefore, you might want to ensure that the smoothing functions for 1-d data and 2-d data are kept isolated to prevent misuse.

Internal functions can also be called as results of if-else statements or other conditional checks.

Advanced Functions - Decorators

Decorators are another version of nested functions, but in the opposite direction. Let's say you want to get timings for individual functions during program execution so you can see where the program spends the most time. With decorators, you can simple wrap your other functions in another function that grabs the start and end times for that function and prints out how long it took to run.

[46]:
```
### The function 'wrapper' takes the function as an argument, and the nested function␣
 ↪'wrapped' uses the incoming arguments in the external function.
### This allows you to pass arguments through a decorator.
### The decorator does its own stuff around the called function.
def decorator_function(func):
    def wrapper_function(*args,**kwargs):
        print("Before the function!")
        result = func(*args,**kwargs)
        print("After the function!")
        return result
    return wrapper_function


@decorator_function ### Here's where we "wrap" the new function rather than having to␣
 ↪nest the entire thing inside the decorator.
def f(x):
    print(f"The value of x is {x}.  Returning x**2")
    return x**2

answer = f(5)
print(answer)
```

```
Before the function!
The value of x is 5.  Returning x**2
After the function!
25
```

Decorators are extremely useful tools, especially with larger projects. You can also have multiple decorators on a single function, which can further expand your code!

[50]:
```
from datetime import datetime as dt

def timing_wrapper(func):
    def wrapped(*args,**kwargs):
        start = dt.now()
        print("Start time is ",start)
        result = func(*args,**kwargs)
        end = dt.now()
        print("End time is ",end)
        print("Total Time:",end - start)
        return result
    return wrapped

@timing_wrapper
@decorator_function
def g(x):
    print("X is ",x,"\nReturning -X.")
    return -x

g(6)
```

```
Start time is  2022-06-14 16:55:36.759226
Before the function!
X is  6
Returning -X.
After the function!
End time is  2022-06-14 16:55:36.759419
Total Time: 0:00:00.000193
```

[50]:
```
-6
```

This shows us that the order of our decorators matters quite a bit. The first decorator in the list is the outermost decorator, then the next one, and so forth.

You can also include additional functionality inside decorators that can make things easier in your code.

For example, let's say I have a function that opens a file and reads the first line. I might want to be sure the file exists before I try to open it. Now, this is something I can do in every function that has a file being opened and read, which means more code being written and more to sift through if something goes wrong. However, if I am consistent in my variable naming conventions (most people develop a style of their own - unless you work for a company with their own established style requirements), I can do things like check for a file's existence before passing the filename on to the reading function. Let's see how that might look.

I'll use the `glob` library that comes standard with `python`.

[57]:
```
from glob import glob

def check_file_exists(func):
```

```
    def wrapper(*args,**kwargs):
        if glob(kwargs["filename"]):
            return func(*args,**kwargs)
        else:
            print(args)
            print(kwargs)
            print("File does not exist")
    return wrapper

@check_file_exists
def read_first_line(filename):
    with open(filename) as f:
        print(f.readline())
    f.close()

read_first_line(filename="test.txt")
read_first_line(filename="09_Functions_Advanced.ipynb") ## This is this very notebook,␣
 ↪so it should return an actual result... which is just a "{".
```

```
()
{'filename': 'test.txt'}
File does not exist
{
```

It should be noted that even though the `read_first_line` function uses the positional argument "filename", if the function is called without the explicit keyword argument `filename="test.txt"`, there will be an error in the wrapper. So take the knowledge that you can mess with parameters inside a wrapper with a grain of salt - you can also make a big ol' mess.

[ ]:
```
### For future use, here's a basic template!
def decorator_name(func):
    def wrapper(*args,**kwargs):
        return func(*args,**kwargs)
    return wrapper
```

6.4 Functions Project

Let's try creating some other functions, perhaps based on the pseudocode project from last time.

1. Write a function to generate a random set of coordinates for a particle in a cube with an edge length of 20.
2. Write a function to create a list of `N` randomly placed particles.
3. Write a function to generate a random movement of a particle with each `x`, `y`, and `z` component of the motion vector as a random value between -1 and +1.
4. Write a function to make each particle in the list take `S` number of random motion steps.
5. Write a function to measure the distance between the starting and ending points of all particles.

I will include some helper functions for things like random number generation below. For all functions you'll write below, you will need to choose what arguments you need to pass to the function and how to get to the result at the `return` statement ending each function.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt

def random(lower,upper):
    return np.random.uniform(low=lower, high=upper)

def plot_distances(distances):
    plt.hist(distances, bins=100, density=1)
    plt.show()
```

[2]:
```
# Generate particle with random position [x,y,z]
def CreateParticle():
    ### Your code goes here!

    return particle
```

[3]:
```
# Create a list of N randomly generated particles
def GenerateParticleCollection():
    ### Your code goes here!

    return all_particles
```

[4]:
```
# generate a random motion between -1 and 1 in all axes and apply it to a given␣
 ↪particle.
def MoveParticle():
    ### Your code goes here!

    return new_position
```

[5]:
```
# Move all particles in your list S number of times and return the final positions.
def BrownianMotionSimulation():
    ### Your code goes here!

    return final_positions
```

[6]:
```
# Compare starting and ending positions for all particles and return the distances.
def CalculateDistances():
    ### Your code goes here!

    return distances
```

Now write a sequence of function calls, variable assignments, etc. to run through your algorithm.

[7]:
```
# This is your main "program", which will use the functions you've written and return␣
 ↪a nice histogram of the resulting distances traveled by each particle.

# Generate 10000 particles

# Move the particles 100 steps.

# Measure the distances between starting and ending positions.

# use the `plot_distances()` function call to show the results as a histogram.
```

If you got a histogram plot that looks like it could be the results of the random motion of 10,000 particles, your program probably works great!

Now that we've all completed that, let's consider some additional things about the algorithm and the *actual* execution of it.

For example, does the distance travelled have any dependency on the original position of the particles? If not, what can we do to reduce the amount of work being done by the computer? Can we reduce the entire program down to something smaller?

Programming is often thought of as "make the computer do a task", which is true, but in the larger picture of scientific programming, it's important to consider the actual goals.

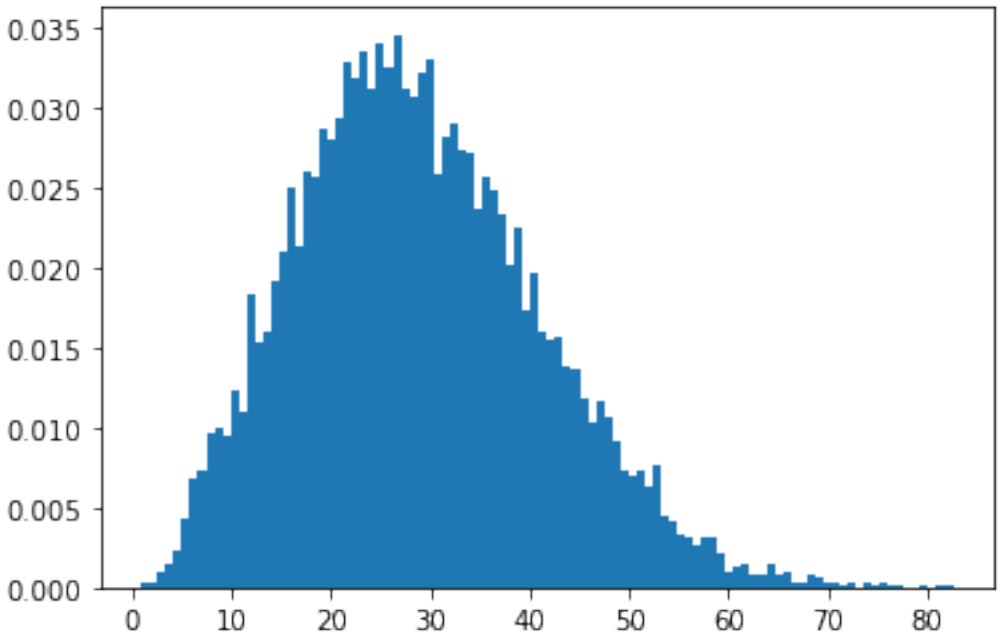
If we re-examine the actual goal of "obtain a distribution of how far particles will travel given random motion", we can reduce the algorithm down significantly.

- The x, y, and z components of motion are all generated the same way and independently of each other.
- Each particle's motion is independent of all the other particles, as is its final distance traveled.
- The final distance is independent of the starting coordinates.

With these points in mind, can you think of a new form of the algorithm?

[8]:
```
# start empty list of distances
# loop through number of particles to test
    # start particle position at 0,0,0
    # make S number of x moves, sum them up to a final x position
    # make S number of y moves, sum them up to a final y position
    # make S number of z moves, sum them up to a final z position
    # calculate norm of x,y,z coordinates
    # append norm to list of distances.
# plot histogram of distances
```

[9]:
```
distances = []
for i in range(10000):
    x=0
    y=0
    z=0
    for j in range(1000):
        x += random(-1,1)
        y += random(-1,1)
        z += random(-1,1)
    norm = np.linalg.norm([x,y,z])
    distances.append(norm)

plot_distances(distances)
```



Chapter 7

Classes

Classes are one of the most powerful, versatile, and useful tools in modern programming. A *class* is a construct that can hold internal data and has its own internal functions that can be called. Classes can also use external functions, other classes, and can range from the extremely simple to the incredibly complex.

Classes are defined in specific way, with some required functions built into all of them.

7.1 Basic Classes

Check out the example class below. Note the two functions that are already included. Of these two, `__init__` is **required** for any and all classes, otherwise the class won't actually be initialized when called, and the data you're trying to hold won't be kept.

[5]:
```
class MyClass:
    def __init__(self,name,age):
        self.name = name
        self.age  = age
    def __str__(self):
        return f"{self.name} is {self.age} years old."
```

`MyClass` is an example of a very basic class structure.

- The references to `self` are important. It is necessary when referencing variables held by the class, especially in internal functions. You may notice that the `__str__` function doesn't take any arguments except for `self`, yet it is able to use the `age` and `name` variables we stored during the `__init__` function call.
- The first class function defined is `__init__`, which is understood by Python to be the *constructor*. This function is called the very first time a new instance of `MyClass` is created.
- The next class function is `__str__`, which we can use to define what is displayed if the user calls the class object as a string, such as in a print statement. This function is optional, and only changes the default behavior of python when converting a class object into a string. See the examples below.

[7]:
```
class MyStringlessClass:
    def __init__(self,name,age):
        self.name = name
        self.age  = age

stringless = MyStringlessClass("Mark",37)
print("MyStringlessClass")
```

```
print(stringless)
print("")

not_stringless = MyClass("Mark",37)
print("MyClass")
print(not_stringless)
print("")
```

```
MyStringlessClass
<__main__.MyStringlessClass object at 0x7f00a8471ee0>

MyClass
Mark is 37 years old.
```

We can see the differences between the two classes in how they are displayed through a `print()` function. We can also create our own additional functions that can be called after the class is initially created.

We'll continue with an example class to hold information about a single atom in a system. I'll be importing `numpy` as well to make some of the internal functions a little easier to work with.

[13]:
```
import numpy as np

class Atom:
    def __init__(self,x,y,z,atomic_number,charge):
        self.x = x
        self.y = y
        self.z = z
        self.number = atomic_number
        self.charge = charge
    def distance_from_center(self):
        return np.sqrt(self.x**2 + self.y**2 + self.z**2)
    def update_location(self,x,y,z):
        self.x = x
        self.y = y
        self.z = z
    def get_location(self):
        return f"({self.x},{self.y},{self.z})"
    def str_charge(self):
        if self.charge > 0:
            return f"+{self.charge}"
        return f"{self.charge}"
    def __str__(self):
        return f"Atom located at {self.get_location()} with charge of {self.
 ↪str_charge()}"
```

[21]:
```
chloride = Atom(3.5,4.2,5.9,17,-1)
print(chloride)

lithium = Atom(1.0,1.0,1.0,3,1)
print(lithium)
```

```
Atom located at (3.5,4.2,5.9) with charge of -1.
Atom located at (1.0,1.0,1.0) with charge of +1.
```

[22]:
```
print(chloride)
print(chloride.distance_from_center())
chloride.update_location(3,3,3)
print(chloride)
print(chloride.distance_from_center())
```

```
Atom located at (3.5,4.2,5.9) with charge of -1.
8.043631020876083
Atom located at (3,3,3) with charge of -1.
5.196152422706632
```

[18]:
```
lithium.distance_from_center()
```

[18]:
```
1.7320508075688772
```

As mentioned above, classes can also use other classes inside themselves, Let's try making a class for a Molecule that uses Atoms.

First, we want to think about what additional information we need for a molecule that isn't already included in the Atoms.

The first thing that comes to mind is bonds.

We can create a class that includes a list of atoms and bonds.

[34]:
```
class Molecule:
    def __init__(self):
        self.atoms = []
        self.bonds = []
    def __str__(self):
        string  = "Molecule made of "
        string += ",".join([str(atom) for atom in self.atoms])
        return string
    def add_atom(self,x,y,z,number,charge):
        self.atoms.append(Atom(x,y,z,number,charge))
    def add_bond(self,index1,index2):
        self.bonds.append([index1,index2])
    def get_bond_distance(self,index):
        idx1,idx2 = self.bonds[index]
        x1 = self.atoms[idx1].x
        y1 = self.atoms[idx1].y
        z1 = self.atoms[idx1].z
        x2 = self.atoms[idx2].x
        y2 = self.atoms[idx2].y
        z2 = self.atoms[idx2].z
        dist = np.sqrt((x2-x1)**2 + (y2-y1)**2 + (z2-z1)**2)
        return dist
    def get_total_charge(self):
        charge = 0
        for atom in self.atoms:
            charge += atom.charge
        return charge
```

[35]:
```
sulfurhexafluoride = Molecule()
```

[36]:
```
sulfurhexafluoride.add_atom(0,0,0,16,+6)
sulfurhexafluoride.add_atom(1,0,0,9,-1)
sulfurhexafluoride.add_atom(-1,0,0,9,-1)
sulfurhexafluoride.add_atom(0,1,0,9,-1)
sulfurhexafluoride.add_atom(0,-1,0,9,-1)
sulfurhexafluoride.add_atom(0,0,1,9,-1)
sulfurhexafluoride.add_atom(0,0,-1,9,-1)
```

[38]:
```
print(sulfurhexafluoride.get_total_charge())
print(sulfurhexafluoride)
```

```
0
Molecule made of Atom located at (0,0,0) with charge of +6.,Atom located at
(1,0,0) with charge of -1.,Atom located at (-1,0,0) with charge of -1.,Atom
located at (0,1,0) with charge of -1.,Atom located at (0,-1,0) with charge of
-1.,Atom located at (0,0,1) with charge of -1.,Atom located at (0,0,-1) with
charge of -1.
```

As you can see, you can build classes of increasing complexity to serve as more manageable data structures for whatever your specific needs may be.

7.2 Inheritance

Inheritance refers to building a class that is based on a framework of another class. This can be done when you have need of classes for different subcategories of object that all fall under the same larger category. This can be thought of in terms of different people having different jobs. You can have a class, `Person` that has all the usual information about a person such as name and age, and then have a class called `Employee` which inherits from `Person`. All `Employees` are also a `Person`, but all `Persons` are not `Employees`.

Let's look at it from a scientific perspective.

[2]:
```
import numpy as np

class Point:
    def __init__(self,x,y,z):
        self.x = x
        self.y = y
        self.z = z
    def distance_from(self,point):
        return np.sqrt( (self.x-point.x)**2 + (self.y-point.y)**2 + (self.z-point.
 ↪z)**2 )

## Now we have a class that defines a point in cartesian space (x,y,z) and has a␣
 ↪function that can be used to calculate the distance from another point.
## What if we wanted a point-charge class but didn't want to have to rewrite the␣
 ↪entire class function for it?
class PointCharge(Point):
    def __init__(self,x,y,z,charge):
        super().__init__(x,y,z)
        self.charge = charge
    def force_between(self,pointcharge):
        dist = self.distance_from(pointcharge)
        k = 8.988e9
```

```
        force = k * (self.charge * pointcharge.charge) / (dist**2)
        return force

## With the PointCharge class, we included the type of class it's inheriting from,␣
 ↪Point.
## PointCharge now has all the same variables and functions as Point, without having␣
 ↪to redefine them.

p1 = Point(0,1,2)
pc1 = PointCharge(1,2,3,-1)
pc2 = PointCharge(0,0,0,+1)

print(pc2.force_between(pc1))
```

```
-642000000.0
```

Take note of how the first function of the `PointCharge` class, `__init__()` is written. The first command is `super().__init__()`, which refers to the inherited class' initialization function. Effectively, it says "do all the stuff the previous class normally does first, then do these additional steps".

We can take inheritance further. Any class can be inherited, and everything in that class comes with it, including its own inheritances.

Consider an atom. We can (roughly!) describe an atom as a point charge that also has mass. Of course they're more complicated than that, but we're taking it a step at a time.

[3]:
```
class Atom(PointCharge):
    def __init__(self,x,y,z,charge,mass):
        super().__init__(x,y,z,charge)
        self.mass = mass

    def get_accel_between(self,atom):
        force = self.force_between(atom)
        accel = force/self.mass
        return accel
```

Now we have a class that holds all the previous stages' parameters and has access to all the previous stages' functions. With this, we can build larger and larger systems and include more and more data, without having to completely rebuild each class from the ground up.

You may notice that each of the classes we built end up requiring more and more variables at their creation. But what if you wanted to have some classes that have standard values? Let's say we wanted to make an Iron(II) atom? Every iron(II) atom can be represented with the same mass (we're skipping isotopic effects for now and going with periodic table values) and same charge.

[4]:
```
class Iron(Atom):
    def __init__(self,x,y,z):
        super().__init__(x,y,z,2,55.845)

iron_atom = Iron(0,0,0)
```

Now we can create inherited classes that are somewhat more specific and also faster to initialize. In the above case, if you know an atom is going to be an Iron(II), you don't have to give the call anything for `mass` or `charge`, because those values are already known.

Try making some other classes for anything you want. You can also mix in things from previous lessons

like decorators to make your classes unique and highly customized.

7.3 Classes Project

Let's build some classes of our own!

1. Write a class to hold a dictionary of TeraChem keywords and their values. Include a function to print out these keyword/value combinations similar to how they would appear in a TeraChem input file.
2. Write a class that takes a single filename (without the extension) and creates its own set of filenames (WITH extension) to use as a TeraChem command line. Include a function that prints out the custom TeraChem command line. Include an option to send the command to the background immediately, freeing up the python interpreter for any additional work.
3. Write a set of classes to hold information about your research projects. The main class should have basic information like "Topic" or "Date Started". Then, use inheritance to create some subclasses for things like "Collaborations" or "Internal" or "Single Author" projects. You can decide what kinds of information each of these unique classes might need.

Try instantiating (*fancy word for creating*) instances of each of your new classes, call their internal functions, print out different internal variables, etc. Have fun, make a mess, go wild!

Chapter 8

Modules

Modules are self-contained larger objects in Python that can be imported and used to make your code easier to manage, easier to read, and easier to modify. If you've used python at all (including the different scripts Dr. Walker has provided at various stages of a project), you've seen and used modules already. The most easily recognized python modules are `numpy` and `matplotlib`, but there are hundreds more included by default with python, and thousands more available through various channels.

8.1 Structure of Modules

Importing Modules/Libraries

There are many ways to import modules into your code. The first example uses the command `import` to load in the module `numpy`

```
import numpy
```

With `numpy` loaded like this, we can use any of the numpy functions and submodules by calling them appropriately. For example, to use `numpy`'s `linspace` function to get 1000 numbers between 0 and 1, we'd call it like this:

```
data = numpy.linspace(0,1,1000)
```

It's also common to see imports assigned to shorter variable names.

```
import numpy as np
```

This shortened name is the standard convention, and while in this case it saves only three letters, these assignments become cleaner and easier when working in larger and more complex modules like `matplotlib`, which has dozens of its own submodules. A common shortening we see is this:

```
import matplotlib.pyplot as plt
```

If you had to type `matplotlib.pyplot.plot(data)` every time you wanted to plot your data, it would get tedious *and* introduce the possibility of typos and errors in the code.

You may encounter some well-established conventions if you find yourself searching for various python-related code help on the internet. The following is a (very not comprehensive) list of some of these conventions.

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import mdanalysis as mda
```

```
import pytraj as pt
import parmed as pmd
```

Much of the import convention can be traced to the original documentation for the various modules.

Another convention you may run across - especially if you're digging into any of Mark's code repositories on GitHub - is the use of CAPS to indicate module bases.

```
import numpy as NP
import matplotlib.pyplot as PLT
import glob as G
import subprocess as S
```

... and so forth. The specific naming convention you use is up to you, just as long as you don't mix up what you've imported. Personally, I use the caps solely because they stand out in the rest of my code, making it easier for me to keep track of where I'm pulling module-specific functionality into my own code.

Importing FROM Modules/Libraries

Let's say there's a specific function in a much larger library that you want to use, but you don't need everything else that comes with it. This is usually because of things like module loading times. If a module is considerably larger than is reasonable, and it forces your computer to lag when importing it, you can consider a different approach. In the example below, there's a module called `glob`, which has a function called `glob()` (super helpful/original, I know). I don't want all of the module, just the function. So I'll import only the function like this:

```
from glob import glob as g
```

Now I can use the `glob.glob()` function simply by using `g()`, and without having to load all the rest of the module with it. Another example some of you may have seen is the `gaussian_filter` function from `scipy.ndimage`, which gets used to smooth out rough data into cleaner curves.

```
from scipy.ndimage import gaussian_filter
```

Importing a Module Without a Name

While this is not something you should generally do, there are occasions where it might be easier to just import everything from a module without including the actual name assignment. I'm not a fan of this because it can have unexpected side effects if there are competing functions of the same name in different modules, but I feel it's better to make you aware in the first place.

```
from subprocess import *
```

This imports everything from the `subprocess` module without requiring the use of `subprocess.` before any of the function/submodule calls. Usually this is seen in code examples on the internet.

```
import subprocess
proc = subprocess.Popen("ls -lrth",shell=True,stdout=subprocess.PIPE, stderr=subprocess.PIPE)
```

becomes

```
from subprocess import *
proc = Popen("ls -lrth",shell=True,stdout=PIPE, stderr=PIPE)
```

Now this may seem shorter and cleaner, but in reality it exposes the user to potential problems. Consider some other libraries that have a very commonly-named `load()` function in them. Importing modules in this manner means that the `load()` function will *always* be the last one loaded by your script. Better to be safe than sorry and not do this.

Importing Files as Modules

Sometimes, we write functions or classes that are consistently useful for us, and rather than have to keep writing them into our code (or risk accidentally changing it in a way that makes it no longer useful), it can be very helpful to just load them from another file.

Python has a default `PATH` that it looks in for modules when importing them. We can add directories to this path inside a python script and then import files/modules from that directory.

Let's say we have a file full of functions we wrote called `utilities.py`:

```
import sys
sys.path.append("path/to/folder/with/file")
import utilities
```

Now we can use any of the functions we wrote by calling them like `utilities.myfunction()`. We can also use the same import tricks and shortcuts we've described above, like `import utilities as U` or even `from utilities import *`.

I'm sure you can see where this is going...

8.2 Creating a Module

As you progress in coding, you may find yourself with a whole pile of various tools and functions that do a wide array of useful things. It might become helpful to keep them all in a single location, but organized for easier access.

Python modules can be made in a fairly straightforward way. Each module is contained inside a folder, and at a minimum there will be one file in the folder called `__init__.py`. This file must be present for the module to be imported, including any submodules. Any submodules should be their own folders, each with their own `__init__.py` inside, and so forth. This can be arbitrarily deep, but be careful not to go too far down a well of submodules.

```
MyModule/
|- __init__.py
|- Submodule1/
|  |- __init__.py
|  |- blueberry.py
|  |- lemon_meringue.py
|- Submodule2/
   |- __init__.py
   |- cherry.py
   |- raspberry.py
```

This structure allows multiple import options. Each `__init__.py` file needs to include its own import commands for subdirectories and files as well. The complexity of these files can be very low, simply importing the subfolders and files as they are named, or complex, including individual function definitions and specific actions that occur during import.

The first level file, `MyModule/__init__.py`, would look something like this:

```
import Submodule1
import Submodule2
```

And the second level file, `MyModule/Submodule1/__init__.py` would look like:

```
import blueberry as bb
import lemon_meringue as lm
```

which would lead to a user's script looking like this:

```
import sys
sys.path.append("path/to/MyModule/location/")
import MyModule as mm
mm.Submodule1.bb.function()
mm.Submodule1.lm.other_function()
```

This is a fairly one-directional import structure. Each thing can really only import from below themselves in the module's structural heirarchy. However, what if a function in `cherry.py` needs to use `other_function()` from `lemon_meringue.py`? That's up one level, then over, and then back down a level. How do we get that `other_function()` into `cherry.py`? Or what if `cherry.py` needed a function in `raspberry.py`?

In the `cherry.py` file, we can include this line:

```
from ..Submodule1.lemon_meringue import other_function
from .raspberry import rasp_function
```

The dots before the modules are the key here. One dot before `raspberry` indicates that the file we're importing from is in the same directory as the current file. Two dots indicates we're up one level. Three dots would be up two levels, etc. We can go up levels and then do the normal imports of modules and submodules, including subfiles as well. This is particularly helpful when writing larger modules with lots of different tasks but use several of the same functions. It's not uncommon to have a particular set of functions that are used all over a module kept in a separate folder to prevent confusion, while also allowing them to be accessed by other submodules.

8.3 Unit Tests

It's important when creating programs to ensure each function works properly on its own and as part of the larger ecosystem. For this purpose, we'll use `pytest`, though there are other options out there.

You can create a test script with your modules to ensure that each function you make does its specific task appropriately. Each test function should be preceded by `test_` for pytest to detect and run them.

Any test scripts should include the same prefix as well. These rules ensure that `pytest` discovers the tests it is expected to run without trying to treat other files and functions as tests.

A common approach to testing is to have a specific folder for tests in the main module directory, with any necessary test data and scripts held inside.

The next cell will check to see if you have `pytest` installed, and if not, will install it for you. It should only need to run once.

[5]:
```
try:
    import pytest
except:
    !pip install -U pytest
    import pytest
```

Tests and the `assert` function Pytest makes use of the built-in `assert` function in python. Assert makes a comparison between two values and returns a `True` or `False`, which gets interpreted as a pass/fail in the test environment.

Consider the following functions in a script called `test_mystuff.py` (the cell below is the same as the contents of that file.)

[10]:
```
def cool_function(x,y):
```

```
    """We expect this function to return the numerical product of two numbers x and y.
 ↪"""
    return x*y

def test_cool_function_pass1():
    assert cool_function(3,4) == 12  ## SHOULD PASS
def test_cool_function_pass2():
    assert cool_function(2,4) == 8  ## SHOULD PASS
def test_cool_function_pass3():
    assert cool_function(4,4) == 16  ## SHOULD PASS
def test_cool_function_fail1():
    assert cool_function(3,4) == 11  ## SHOULD FAIL
def test_cool_function_pass4():
    assert cool_function(5,20) == 100  ## SHOULD PASS
def test_cool_function_pass5():
    assert cool_function(3,5) == 15  ## SHOULD PASS
```

With the function we wrote and the test functions, we can run `pytest` in the directory and it will find any and all test files, and any test functions, and check them for us. The following cell will do this for us and print out the results. Keep in mind, we wrote one of our tests to intentionally fail - this is to show you what it will look like so you'll know how to recognize it in the future.

[12]:
```
# In Jupyter/VSCode notebook environments, the ! before a normal shell command
# will direct the notebook to run that line in the shell environment rather
# than as a python command.
!pytest
```

```
============================= test session starts
==============================
platform linux -- Python 3.9.7, pytest-7.1.2, pluggy-0.13.1
rootdir: /home/mark/GH_Repositories/CodingSummerSchool/Day04_Python_Modules
plugins: anyio-2.2.0
collected 6 items


test_mystuff.py
...F..
[100%]

=================================== FAILURES ===================================
_____________________________ test_cool_function_fail1
_____________________________

    def test_cool_function_fail1():
>       assert
cool_function(3,4) == 11
E       assert 12 == 11
E        +  where 12 = cool_function(3, 4)

test_mystuff.py:12: AssertionError
========================== short test summary info ===========================
FAILED test_mystuff.py::test_cool_function_fail1 - assert 12 == 11
```

```
========================= 1 failed, 5 passed in
0.04s ==========================
```

Notice the first section, which lists the file that was tested (`test_mystuff.py`), and lists the passes and failures in order with the dot/"F". It also shows that `pytest` completed 100% of the file. Below, you see the more detailed report on the failures (nothing is given for the passes by default because... they passed).

The failures are shown so that you can see exactly *where* a function failed. You can use this method to test complex constructs like classes as well, by simply having multiple `assert` commands in your test functions. Here's another example with a basic class. The following cell is also in `test_myclass.py`

[14]:
```
class Person:
    def __init__(self,name,age,university):
        self.name = name
        self.age = age
        self.uni = university

def test_Person_PASS():
    test = Person("Mark",37,"Wayne State University")
    assert test.name == "Mark"
    assert test.age == 37
    assert test.uni == "Wayne State University"

def test_Person_FAIL():
    test = Person("Mark",37,"Wayne State University")
    assert test.name == "Mark"
    assert test.age == 37
    assert test.uni == "Wayne State Universe" ### Notice how this is not correct.
```

[15]:
```
!pytest
```

```
============================= test session starts
==============================
platform linux -- Python 3.9.7, pytest-7.1.2, pluggy-0.13.1
rootdir: /home/mark/GH_Repositories/CodingSummerSchool/Day04_Python_Modules
plugins: anyio-2.2.0
collected 8 items


test_myclass.py .F
[ 25%]
test_mystuff.py
...F..
[100%]

================================== FAILURES ===================================
_______________________________ test_Person_FAIL
_______________________________

    def test_Person_FAIL():
        test = Person("Mark",[9
```

```
4m37,"Wayne State
University")
        assert test.name ==
"Mark"
        assert test.age == 37
>       assert test.uni == "Wayne State
Universe" ### Notice how this is not
correct.
E       AssertionError: assert 'Wayne State University' == 'Wayne State
Universe'
E         - Wayne State Universe
E         ?                     ^
E         + Wayne State University
E         ?                     ^^^

test_myclass.py:17: AssertionError
___________________________ test_cool_function_fail1
___________________________

    def test_cool_function_fail1():
>       assert
cool_function(3,4) == 11
E       assert 12 == 11
E        +  where 12 = cool_function(3, 4)

test_mystuff.py:12: AssertionError
========================== short test summary info ============================
FAILED test_myclass.py::test_Person_FAIL - AssertionError: assert 'Wayne Stat...
FAILED test_mystuff.py::test_cool_function_fail1 - assert 12 == 11
========================= 2 failed, 6 passed in
0.04s ==========================
```

The `pytest` run now also gives us a summary of the different files and their failures in addition to the more thorough breakdown of each failure. Pytest can be extremely useful in finding and fixing bugs in your code, especially as the code gets bigger and more complex.

There are many more things you can do with pytest as you become more advanced in other aspects of python programming, such as testing error handling and making sure that the code handles user errors properly.

Pytest Documentation

You can also build test *classes* in addition to test *functions*. This can allow you to use more complex data structures with multiple test functions as well as ensure that the different functions of a class interact appropriately and correctly.

In most code-heavy jobs, it is generally expected that you will include tests with code you write. Thus, it's good practice to write tests alongside your actual programs and functions. It'll also make it easier to spot problems early before they become larger problems that are harder to deal with because there are more functions and classes built upon them.

Part III

C++

Chapter 9

Introduction and Basics

Chapter 10

Day 6 - Introduction to C++

Goals

1. Compile a basic C++ program
2. Identify sections of source code
3. Manage input and output in your code
4. Handle variables, including initialization, assignment, modification, and recall.
5. Read from and write to files, including parsing/processing inputs and formatting outputs.

C++ is a compiled language, which means that before a program written with it can be executed, it must be compiled from human-readable text into the binary machine language.

By convention, C++ source code files (the text you can read and edit) end with `.cpp`, compiled object files end in `.o`, and completed executables end in `.x`.

The basic compilation command we will be using is below:

```
g++ main.cpp -o output.x
```

This calls the compiler `g++` on the source file `main.cpp`, with the resulting output (`-o`) as `output.x`. The program can then be executed like any other command-line program by calling it with `./output.x`.

To start, every C++ program will have the following basic structure:

```

```

Let's break this down into some major pieces. The first section is where libraries and other required files are included. In this example, we're including the `iostream` library, which manages the input/output streams of C++, and `cstdlib`, the C Standard Library which contains a wide array of commonly used functions. As we progress, we will encounter more libraries with their own unique usefulness.

```

```

The next section is where the main program loop is written. It must be called "main" for C++ compilers to understand where the starting point of the program is when it's run. The datatype `int` is used by convention to allow for integer-based exit codes to be returned to the operating system. This is useful for error reporting - if the program fails in a way the programmer has anticipated, it can return a specific integer value to the operating system to alert it to the failure.

The **scope** of the `main()` function is defined by the open- and close- curly braces. Throughout C++, this will be how scope is defined and maintained. Inside the scope is where all the program's functions and commands are stored. At the end of any program or function (with a specific exception we will get into later), there **must be a `return` command**. Also note that at the end of every command line, there is a semicolon.

10.1 Input/Output

The most basic functionality is input from and output to the terminal. These are accomplished with `std::cin` and `std::cout`. The following example prompts the user to enter an integer value, stores it, and then prints it back out.

This example also introduces the next section on variables.

Homework

Try creating some small programs to deal with inputs and outputs from the terminal. Don't forget to compile before trying to run the programs!

10.2 Variables

The code block below introduces `int` and `double` variable types. Integers are whole numbers and doubles are floating-point numbers, or decimals, or simply floats. In C++, variables must be **declared** before they can be assigned a value, retrieved, or modified.

Notice the way each of the different sections are written. The numerical values are the same in each, with `10` and `10.` being equivalent. However, C++ code treats the former as an integer and the latter as a float. If you run the program above, you will see the different outputs for what we know to be mathematically identical operations.

It is important to keep these things in mind when working with numerical values in C++.

Constants

It is sometimes useful for us to have values stored as variables that we can adjust once rather than trying to find every instance of them in our code if we need to change them. For example, we might need to use π in a program, but may only need a few decimal places. Later on, we might find that we need more than a few decimals, and it would be easiest for us if we just had to change one value. It's also safer because we reduce the risk of introducing an error through a typo in the number if we only have to enter it once.

This is where the use of a `const` comes in handy. We add `const` before a variable type to indicate that the variable is not allowed to be changed during the execution of the program. In the example with π, it would look something like this.

In general, constants are declared outside of the scope of functions so that they can be used everywhere.

Strings

Strings are also known as arrays of characters, or chars. Both options can be used in C++, however it is important to know the differences in how they can be used. Strings in C++ have some built-in functions that can be used to find patterns and substrings, modify strings, and slice them. The code below illustrates the two different variable declarations. Note that the `char*` variable is also defined in the same line.

Also, you will notice the line `using namespace std;` in the code block below. Previous code blocks have had `std::` at the start of many of the commands. This line is being added to make it easier to use the functions contained within it. It's similar to python's `from library import *`, in that it removes the requirement to include `std::` at the start of many of the library-dependent variable types and functions. As a result, you will see things like `cout` instead of `std::cout`. This is one of the benefits of using **namespaces**, and it can be helpful later on down the line as you develop your own code.

You may also have noticed the use of `endl` rather than the newline character `"\n"`. These both produce a line break. The only difference between them is that `endl` does something called "flushing the buffer", which means that the data stored in the output buffer is forced out into the main output.

For most text-based things, this may not seem important. But if you're writing things to a file in a loop, you might want to ensure that everything is written to the file as it's generated, rather than all at once at the end of the loop. Using `endl` is the preferred option in this scenario, and it's ultimately easier.

10.3 Basic Math

Basic mathematical operators are available in C++. An example is given below. Notice that the variables are all declared on one line. This is done because all of these different variables are doubles, and the compiler can interpret the single line to mean exactly that.

When run, this program will take any two values given and return the different basic mathematical operations between them - addition, subtraction, multiplication, and division. Try writing your own mathematical algorithms!

10.4 File I/O

As we discussed before in the Python section, one of the most important things we can do with our code is learn to read and write data in files, or process that data in a way that is useful to us. The next several code blocks will illustrate some of the different ways file data can be read, processed, formatted, and written. Keep in mind that this is meant as an introduction to the concepts and not an exhaustive list of methods.

It's important to remember that whenever you open any file in C++, you must close it at some point before the end of the program. In python, there are instances where the closing of a file is automatic, so it's a little less strict. In C++, that leniency is not there. **Opened files must be closed.**

10.4.1 Reading Files

File I/O is largely controlled by the `fstream` library. When opening a file for reading, we use the variable type `ifstream`, which is short for **i**nput-**f**ile-**stream**. For files to be written, we use the variable type `ofstream`.

In the example above, we have opened the file "TestData/input_file.txt" for reading and then used it as our input stream. We then took one the first item in the stream and put it into the variable "words". Then we closed the file because we were done with it. And finally, we output the value of "words", which in our case was only the first word of the file.

If we changed the line `inFile » words;` to `inFile » words » words;`, we would get the **second** word in the file, because we assigned the first word to the variable "words", and then overwrote it with the second. You can also parse lines such that different portions of the line go to different variables. Take the following lines as an example:

```
State   Frequency       Amplitude
0       100             3.2
```

If we assume we've gotten through the file to the point that our input is sitting at the start of the second line, we could use the following code block to assign our different values.

This would assign each of the values to their respective variables. This is how many programs we use parse input files for molecular geometry, by simply reading in each line with an expected format.

The next code block shows how we can get an entire line of the file at a time. In many cases, it's easier to work with entire lines rather than trying to account for each individual word in the file. If we take entire lines at a time, we can also do things like searching for specific patterns within them.

The `getline()` function takes everything in the first argument (the inFile stream) from its current position to the next linebreak character or the EOF (end-of-file) character, and then puts that content into the second argument, in this case our variable "words".

Now let's try going through the entire file. We can open the file like normal, then put our `getline()` function inside a `while` loop. This is useful because the `getline()` function will continually iterate through all the lines in the file and return a `False` value when it gets to the end of the file. Therefore, inside the loop we can work with each line individually. In the example below, we're simply printing out each line to the terminal as we go.

If you ran this program in the actual Code Summer School folder associated with this document, you'll have gotten the entire lyrics to "I Am the Very Model of a Modern Major-General" from Gilbert and Sullivan's *Pirates of Penzance (1879)*.

What if we wanted to modify lines that had a specific pattern in them? We can use the `find()` function available to the string class to seek a pattern inside the string. We can also use some `if/else` statements to direct the code's flow. In the code block below, we simply expanded the previous loop's internal functions to include the check for the word "General" in each line. If the word is found in the current line, we run the `transform` function (found in the `algorithm` library included at the beginning) to convert the whole string to all-caps.

So every time the program encounters "General" in a line, it will print that line in all caps rather than its original format. And since the file has the lyrics to "I Am The Very Model of a Modern Major-General", we can assume it'll happen a lot.

10.4.2 Writing Files

Writing files is similar to reading files in that you have to open and close the files appropriately. Beyond that, however, writing to the file is much the same as outputting text to the terminal with `std::cout`.

The above code block opens a file called "output_file.txt", outputs two lines of text, then closes the file and exits the program. This is fairly straightforward.

However, what if you wanted to format the text in a specific and consistent way? You can use *manipulators* to achieve this.

You can change the justification and fill character with the commands shown below. Keep in mind that the fill character must be a single character. You can also set how wide each portion of the output will be by changing the `setw()` value as necessary.

Homework

Try making your own smaller program to read in information from a file, find some specific pattern, and then output a differently-formatted version of the data to a new file. You can use any of the outputs you have from your research calculations.

Chapter 11

Functions

Goals

1. Understanding of function datatypes
2. Understanding the void function
3. Differentiate between function declarations and definitions
4. Identify and manage function arguments and returns
5. Understand pass-by-value and pass-by-reference, and when to use which

There are a few vocabulary terms we'll need to establish right away with functions, starting with **function**.

In C++, a **function** is a set of instructions inside a scope that takes arguments as inputs, performs a task on them, and returns something back to wherever the function was called from. A **function call** is the point in a program where the function you wrote elsewhere is being executed. A function **definition** is where the instructions of the function are actually defined, and includes the return type, function name, arguments list (including data types for each argument), and the instructions within the scope of the function. A function **declaration** is a truncated form of a definition, which doesn't include any of the instructions, and is meant more to alert the compiler that a function exists with those parameters and is simply defined elsewhere. Without declarations, every function **MUST** be defined before they are called, and this can become difficult in more complex codebases.

11.1 Function Types

The most important thing to remember about functions in C++ is that every function must have a **return type**. These are data types like `int`,`double`, and `void`, and correspond to the type of return each function is expected to send back to wherever it was called from. Every function must have a return, with `void` functions being the only exception. Look at some example functions below (these are not complete programs, just the function code).

The first function, `add_integers` is defined with the `int` return type. Note that inside the scope, the variable being returned (`answer`) is also of the `int` type. This is important to remember, as trying to return one variable type from a function that is defined with another will cause a compiler error and your program won't run. The same is true for the second function, `multiply_doubles`. Also note that each function takes arguments of the same name, but with different types. As we've discussed before, in C++ it is important to keep track of the variable types you're working with. This is also one of the reasons that functions are so useful - once the internal code of the function is correct, you don't have to worry so much about the internal

variable types. Any variable type that you have access to (including variable types you create in your code) can be used as function types.

Void Functions

`Void` functions are generally used for functions that will manipulate or modify variables directly, or will interact with external files, but don't need to send anything back to the original point of where it was called. Aside from that distinction, however, `void` functions can be treated very much like any other function.

Note that in the above code block, there is no return command in the function. The function takes the arguments given, performs a set of instructions with the arguments, and since there is nothing that needs to be sent back from the function, the `void` function type is the correct choice.

11.2 Declarations

As mentioned above, declarations are shortened version of the function. Think of them like a preview of the function. They don't tell the whole story, but they give the compiler enough information to know what is going into the function and what should be coming back. They are also extremely useful when writing multiple versions of a function.

Why would we write multiple versions of a function?

There are occasions when certain code will work for some hardware or software, but not for others. For example, there are scientific programs that use different mathematical libraries depending on what brand of processor is in the computer. A programmer can write multiple versions of a function, with each version using specific libraries and internal instructions, so that when the program is being compiled, the only thing changing is the internal instructions of the function itself, and no additional changes need to be made to the main code base.

Function declarations take the form shown below.

Note the semicolon ending the line, and a lack of any scope following the function.

11.3 Definitions

Definitions are the complete set of instructions for a function. They begin just like a declaration, except that instead of a semicolon, they have the open- and close-curly braces with the full set of commands inside.

The full function definition above has the return type, arguments, and complete instructions inside the scope, ending with the return value.

11.4 Arguments

Arguments are the pieces of information we're passing to the function inside the parentheses whenever we call it. Each argument needs its own datatype. There are two ways in which arguments may be passed to a function, and both have very specific outcomes. In general, it's a good idea to use the first unless you specifically **need** to use the second.

Pass-By-Value

The first method of argument passing is called "pass-by-value", and is the standard way of sending information to a function. Let's see an example below.

This code block shows us two different functions, both using pass-by-value. What this means is that the program sends the value of the variable to the function, *not the variable itself*. Think of it like sending a copy of the static value at the time of the function call, rather than the original. Pass-by-value maintains the scope of the variables. In the `main()` function above, the line that reads `answer = multiple(my_var1,my_var2)` is interpreted at execution to be `answer = multiple(10.5,2.0)`. If we compile and run the code block above, we will see that the value of `my_var1` and `my_var2` do not change during the execution of the function itself, even though the value inside the function does change.

Pass-By-Reference

Pass-by-reference allows us to modify variables inside a function and have those modifications hold after exiting that function. The function, rather than taking the **value** of the variable being passed, takes the **memory location** of the variable, and changes the data at that location. Pass-by-reference is signified by including the ampersand (&) character with the variable name in the function declaration and definition. Let's look at the above code block after modifying it to be pass-by-reference.

Note that this code block is almost identical to the previous example, except for the declaration and definition of `multiply(double &var1, double &var2)`, which now has the & character included. If we compiled and ran this code, we would see the value of my_var1 would actually be changed in the main program after modification in the `multiply` function.

The most common use for pass-by-reference functions is when you have multiple values that need to be changed and that cannot be passed back as a single variable object. For example, in a molecular dynamics program, a function may be written to update information about a single atom. The information may include position (3 variables), velocity (3 variables), net force on the atom (3 variables), and so on. Additionally, pass-by-reference can be better for memory management, as your program will not require additional memory overhead to store the temporary copies of data inside each function that uses it.

Homework

Practice writing some small functions for the following scientific equations. Think about what needs to be passed to the function and what can be defined internally or used as a constant.

$P = nRT/V$	P = Pressure n = number of particles (in mol) R = gas constant T = temperature V = volume
$\langle T \rangle = -\frac{1}{2}\sum_{k=1}^{N}\langle F_k \cdot r_k \rangle$	$\langle T \rangle$ = Total Kinetic Energy N = number of particles F_k = force on the kth particle r_k = position of the kth particle
$k = Ae^{\frac{-E_a}{RT}}$	k = rate constant A = pre-exponential factor E_a = activation energy R = gas constant T = temperature

Feel free to come up with other equations to convert to functions, and don't forget to comment your code!

Chapter 12

Classes

Classes in C++ are similar to what we've previously covered in the Python section. They are objects that can hold multiple types of data and internal functions. Classes are often used for larger and more complex problems, however they can be as simple as you want.

12.1 Constructors and Destructors

Whenever a class is instantiated, the constructor is used. This is analogous to python's `__init__()` function, in that it is a set of commands that are run the moment the class object is created. Conversely, a destructor is a set of commands that are executed whenever the object is destroyed. This includes any event where the memory is released, whether from a specific command inside the code, or when the program itself finishes execution and exits.

By convention, constructors and destructors are named the same as their class, with the special addition of the ~ before the destructor's name. See the example below.

12.2 Internal Functions

Internal functions are simply functions that act on variables inside the class. An internal function may access internal variables in a class as well as take arguments like any other C++ function. If we expand the example above, we can add in a function for the area and perimeter of the square class as callable functions. Also included is a small `main()` function to show how these functions may be called.

Chapter 13

Headers and Libraries

Goals

1. Understand libraries and headers
2. Develop your own header files
3. Understand the basics of how Makefiles work

13.1 How Libraries and Header Files Work

Libraries and headers both allow you to add additional functionality to your programs without having to code them all from scratch. However, there are some differences between them.

Headers

Headers are plain-text files that contain information used in libraries, such as new data types, constants, library-specific classes, and function declarations. Headers are added to programs with the `#include` command that we're familiar with. By convention, header files end in `.h`,

Below is an example header file.

There are a few important features to point out. First, the three lines that act as "guards" are important to ensure that the compiler won't try to redefine things over and over, or bloat the final program with repeated functions. The first part of the guard has the `#ifndef` and `#define` lines. These two tell the compiler that if it has not already defined `GEOMETRY_H`, it should do so. In this way, even if you have a hundred different source files that use the header file, it will only be compiled and linked once. The final guard, `#endif`, concludes the instructions that followed the first two guards. Everything between the guards is part of that particular header/library combination.

Next is the `namespace geom` scope. This is the same concept we encountered in the earlier lessons with `std::cout` or the later command `using namespace std;`. In this case, a namespace called `geom` is created. The class contained within that namespace would be called using `geom::Circle`, and any functions inside would use a similar call. However, if our code includes `using namespace geom;` after including this header file, we can use the internal classes and functions without the `geom::` requirement, similar to how we are able to skip using `std::` in previous code examples.

Headers can be written without namespaces, however you will want to be careful of your function, class, and variable names and be sure they don't conflict with anything else in use in any of the libraries or source

codes you're using. In general, it's recommended that anything you create be contained in a namespace.

Libraries

Libraries are sometimes already compiled into machine language and are not "human-readable" files. Libraries are usually accessed during compilation of your main program as part of the linking process. Many pre-compiled libraries are available for free, and learning their use can provide improved access to hardware-specific functionality or better algorithms without having to create them all yourself. The source code for libraries is usually written in the same language and format as their associated header files. Where headers had only declarations, library source files will have the complete function definition, class definitions, etc.

Below is an example library source file to go with the above header file.

With both the `geometry.h` and `geometry.cpp` files written, a main program file might look like the following.

Chapter 14

Makefiles

14.1 Makefiles

Makefiles are a set of instructions for the compiler to use to build more complex programs from multiple files. Previously, all of our programs have been single source code files containing everything needed.

Effectively, each makefile is a list of object files that are created by compiling source code files, and include any dependencies. This way, if you make a change to one file, only that file and any dependent files are recompiled and linked, rather than the entire program (which can take hours or even days to complete, depending on the size of the program in question).

Each individual piece of the compilation process is written in a specific format, shown below.

```
```

First, we establish any necessary environment variables for the compilation using the `export` command. Then, each object to be created is listed with any dependencies. In the example above, `executable.x` (the final executable file) is dependent on the file `executable.cpp` (the source code for that particular output) and the object files `subfile1.o` and `subfile2.o`. When the makefile is run, the compiler checks to see if `executable.x` exists *and is newer than* all of its dependencies. What this means is that if any of the dependencies have been changed since the last time `executable.x` was compiled, it will be recompiled with the command on the line below, beginning with `g++`.

The compile line for each object being created begins with the compiler itself, then lists any and all source files and objects to be linked, the uses the `-o` flag to indicate the name of the output object, and `-I` for any additional file locations it should include when compiling (such as the location of all header files in the project).

If you're wondering what the `.o` files are, they're "object" files, and are effectively the intermediate steps between the raw source code of an entire project and the completed final compilation into a single executable file. Objects are compiled *portions* of the program, and when they're included in the compile command shown above, the process is called "linking".

This allows us to connect multiple separate files together without having to fully recompile everything. Each object file is listed in the Makefile in the same way as the example above, with only the filenames of dependencies and outputs being changed.