JupyterLab + Python

This project looks at teaching JupyterLab [GG16] and Python (and a little bit of Matplotlib as well)

This project started on 5/19/2021, so it is still in development. Feel free to provide suggestions.

Notebook Introduction

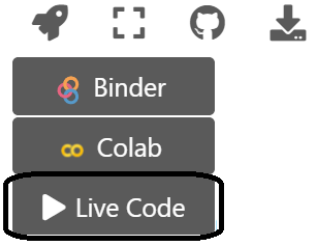
How to Use this Notebook

This notebook allows you to both follow the text and interact with the code directly.

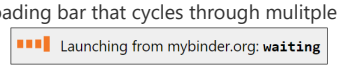
At the top page, you will see:

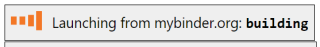
Select the icon on the left, the rocket ship:

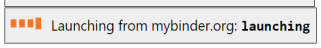
Then go down to “Live Code”:



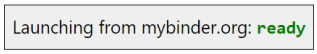
You should see at the top of the page a loading bar that cycles through mulitple states.







Then, finally:



Try this for yourself!

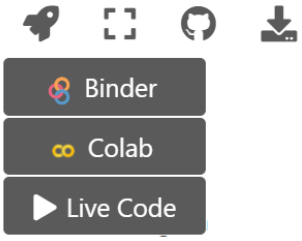
Here is a basic line in Python, after setting to Live Code. You can edit the Python code **directly in the notebook** 😀 😀 😀

```
2 + 2
```

```
4
```

Export to Binder or Google Colab

As you can see you can also export these notebooks to Binder or Google Colab.



This will take you to their respective websites but you can work with them there, if you would like!

References

I know it tradition to have the refences at the end of books, but when you are standing on the shoulders of giants. You thank them first.

[GG16]
B Granger and J Grout. Jupyterlab: building blocks for interactive computing. *Slides of presentation made at SciPy*, 2016.

[Mun14]
Tamara Munzner. *Visualization analysis and design*. CRC press, 2014.

[Wic10]
Hadley Wickham. A layered grammar of graphics. *Journal of Computational and Graphical Statistics*, 19(1):3–28, 2010.

Thank you!

Also, a huge *thank you* to Adam Lavely (https://github.com/adamlavely) for developing some of the intial notebooks!

Introduction to JupyterLab

Where am I? (JupyterLab Notebook)

Jupyter is a powerful suite of tools that allows us to do many things.

Jupyter is capable of running **Ju**lia, **Pyt**hon and **R**, as well as some other things.

Cells

Each box is called a cell.

Two types of Cells

Text

Text Cells allow you to add text (via Markdown), which includes tables, images, links, bullet lists, numbered lists, LaTeX, blockquote, among other things.

Table

```
| This | is   |
|------|------|
|   a  | table|
```

This is

a table

Image

```
![Wheat Field with Cypresses](images/vangogh.jpg)
```

Link

```
[Attribution](https://www.metmuseum.org/art/collection/search/436535)
```

Vincent van Gogh / Public domain The Metropolitan Museum of Art, New York - Purchase, The Annenberg Foundation Gift, 1993 - Attribution

Bullet List

```
* I am a
    * bullet
* list
```

- I am a
    - bullet
- list

Numbered List

```
1. I am a
  1. numbered
1. list
```

1. I am a
2. numbered
3. list

LaTeX

```
$$e=mc^2$$
```

\[e=mc^2\]

Blockquotes

```
> This is a blockquote.
```

> This is a blockquote.

Code

Cells can be run using the Run button ► or selecting one of the run options under the Run menu.

Try this out! You can change what is in the cell and rerun the same cell, which is useful for debugging.

```
2 + 2
```

```
4
```

Your turn: In a new cell, figure out what **5315 + 5618** is.

```
## remove and type out 5315 + 5618
## then hit the play button
```

Introduction to Python

In this section, I wanted to introduce a few basic concepts and give an outline of this section.

Comments in Python

In Python, we can create comments in the code itself. Considering we can use markdown language (as you see here 😁), we won't use this too much in this notebook. Though, here is an example.

Basically, you use the... umm... hashtag? Number sign? Pound sign?

This thing -> #

```
# I am a comment in Python
# Here is 2 + 2
2 + 2
# As you can see, these are not "computed" using Python.
# We are just comments for the person looking at this.
# Or... you!
```

```
4
```

Print Function

We will being using...

```
print()
```

...several times in this notebook.

print() is a function to print out strings, variables, numbers, functions, etc.

Let's use the classic example.

```
print( "hello, world!" )
```

```
hello, world!
```

OR

```
print("hello, world!")
```

```
hello, world!
```

print() can do some fun things as well. As in, giving it more than one thing to print with commas between them. This will print both things with spaces.

```
print( "hello,", "world!" )
```

```
hello, world!
```

Help Function

The...

```
help()
```

... function is exactly what it is. It is a function to help you understand the basic usage of another function.

```
help(print)
```

```
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
```

Resources

Highly suggest looking for answers using StackOverflow

Common Errors

One of the most common errors in Python is the dreaded

```
2 + 2
 3 + 3

  File "<ipython-input-1-0dcc020fd5cb>", line 2
    3 + 3
    ^
IndentationError: unexpected indent
```

Why does this occur? Well, because Python uses spacing or tabs to distinguish where things like loops, functions, and if/else statements start and end. So, if you add an extra space or tab at the beginning of the statement, you will see this message. If you do, check your spacing.

> **Note**
>
> Python can get weird with this issue. As you can, technically, start code wherever as long as you are consistent. The next cell shows an example of this... oddity.

```
            2+2
            3+3
```

```
6
```

still works!

Learning about Variables

When we are developing our idea, we sometimes need to use values multiple times or change the value based on our code. This concept is where variables become very helpful. Let's look at an example.

In this example, we are adding a few numbers together. In this instance, if all we care about is getting the result (similar to a calculator). Then variables are not needed.

```
5 + 3 + 16
```

```
24
```

But let's look at an example where we need to get the circumference of a circle using multiple radii. The equation for the circumference of a circle is: \(C = 2 \pi r\)

Let's say the radius is 5

```
2 * 3.14159265359 * 5
```

```
31.4159265359
```

OK, how about radius 10 and 11 and 4 and ... Well, in this example, we might not want to rewrite 3.14159265359 over and over. So, in this case, we want to create a variable for this, and we will call it pi.

```
pi = 3.14159265359
```

Now, every time we reference the variable called **pi** it will refer to the number **3.14159265359**

Let's try those radii again (10, 11, 4)

```
2 * pi * 10
```

```
62.8318530718
```

```
2 * pi * 11
```

```
69.11503837898
```

```
2 * pi * 4
```

```
25.13274122872
```

By the way, if you happen to get an error:

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

Make sure you go to the cell that has

```
pi = 3.14159265359
```

and run this cell *first* then try the other calculations.

Type of Variables

There are multiple types of variables. The most common (and the ones we will talk about) are:

- Integers (whole numbers)
- Float (Floating points or numbers with a decimal)
- Text
- Lists
- Dictionaries

The nice thing about Python is that we do **not** need to specify (or declare) which type we are using. Python will figure this out for us!

BUT FIRST, a quick detour...

We need to talk about Camel Casing.

Camel Case

Variable names must be one continuous string of letters/numbers. So, let's say we wanted to create a variable called "number of kittens." Instead calling this variable *number of kittens*, I would call it *numberOfKittens*. Why the capitalization? Because it makes it easier to separate the words in the name. As in, *numberofkittens* vs. *numberOfKittens*. We have a fun name for this: camel case.
File:CamelCase new.svg. (2020, April 15). Wikimedia Commons, the free media repository. Retrieved 15:25, June 3, 2020 from https://commons.wikimedia.org/w/index.php?title=File:CamelCase_new.svg&oldid=411544943.

Integers or int

As mentioned, integers are whole numbers. Let's create an example. How about we use our numberOfKittens. We will then set this value to 0. As in, we have 0 kittens.

```
numberOfKittens = 0
```

One thing we might want to do is to have Python tell us what **type** this variable is. Well, Python has a function for this called

```
type()
```

```
type( numberOfKittens )
```

```
int
```

So this checks out, we made an int, and it is showing us we have an int.

Now, once we have a variable, it is not static. We can change the value as much as we need to. Running the next cell will continually add 10 to our original variable.

Try running this a few times.

```
numberOfKittens = numberOfKittens + 10
numberOfKittens
```

```
10
```

or

in more human-readable terms.

numberOfKittens (new value 10) = numberOfKittens (originally 0) + 10

numberOfKittens is now 10

or

Floating points or floats

Floats are similar to integers, but with more precision. Float comes from a Floating point or a number with a decimal point.

This example starts at 0, but note that this is .0 Adding the decimal tells Python that we should have a float value instead of an integer.

```
aFloatVariable = .0
```

Let's again, check the variable type.

```
type( aFloatVariable )
```

```
float
```

Looks good.

And again, we will add 10 to this. There is something specific interesting here; see if you spot it.

aFloatVariable = aFloatVariable + 10 aFloatVariable

If you guessed "mixing a float and an integer," you got it. Let's see an example.

Mixing integers and floats

In Python (3, more specifically), the variable will always take the form of the most precision. So, by default, a float.

```
letsSeeWhatHappens = numberOfKittens + aFloatVariable
letsSeeWhatHappens
```

```
10.0
```

We can force variables to be a certain type. We call this 'type-cast' and can be used to:

- make an integer into a float
- a float to an integer
- an integer to a string (we have not discussed this yet)
- a float to a string (we have not discussed this yet)
- etc...

type-cast

> **Note**
>
> type-cast is temporary. If you do not use a type-cast, the variable will revert to its original variable type.

Let's switch our numberOfKittens to a float using

```
float()
```

and turn our aFloatVariable to an integer using

```
int()
```

```
float(numberOfKittens)
```

```
10.0
```

```
int(aFloatVariable)
```

```
0
```

Common Question

What happens when you convert a float like .5 to an integer? Does it round up or down?

Well let's see what happens.

for value 0.0 we will get 0
for value 0.1 we will get 0
for value 0.2 we will get 0
for value 0.3 we will get 0
for value 0.4 we will get 0
for value 0.5 we will get 0
for value 0.6 we will get 0
for value 0.7 we will get 0
for value 0.8 we will get 0
for value 0.9 we will get 0

So, in conclusion. It will always round down.

String or str

So, up to this point, we started our conversation working with numbers. Well, what about the other things that are not numbers... like text? Well, for text, we use something called a String or str.

Strings allow us to capture a single character up to thousands of characters (actually, much more than this). Let's go through a traditional example of "Hello, World!" but with my slight spin to it.

```
helloStatement = "Hello, everyone!"
```

As you can see, can capture text and other alphanumeric and special characters. There are several unique functions for strings but first, let's double-check and see what type we from our helloStatement.

```
type( helloStatement )
```

```
str
```

Not too surprising, we see this is type str or string.

Note

For those coming from another programming language. Sometimes other programming languages will have a specific designation for a single character string or, as it is called, a character. Python has a one-size-fits-all label for text, and that is a string. Here, let me prove it.

```
singleCharacter = "a"
type( singleCharacter )
```

```
str
```

String Indexing/String Slicing

One of the first ways to interact with our string is to take a look at individual characters by using their **index**.

The **index** is position (or multiple positions) for each character in the string. So, if we look at our string, we have Hello, everyone! If we wanted to see the first letter *H*, we could reference this using the index or the position where the letter is in the string.

```
helloStatement[1]
```

```
'e'
```

ohh.. wait a minute. We were expecting the letter *H*, but we got *e*. What happened?

> **Note**
>
> For indexes, we always start at the number 0. So, 0 is the first thing, 1 is the second thing, and so on.

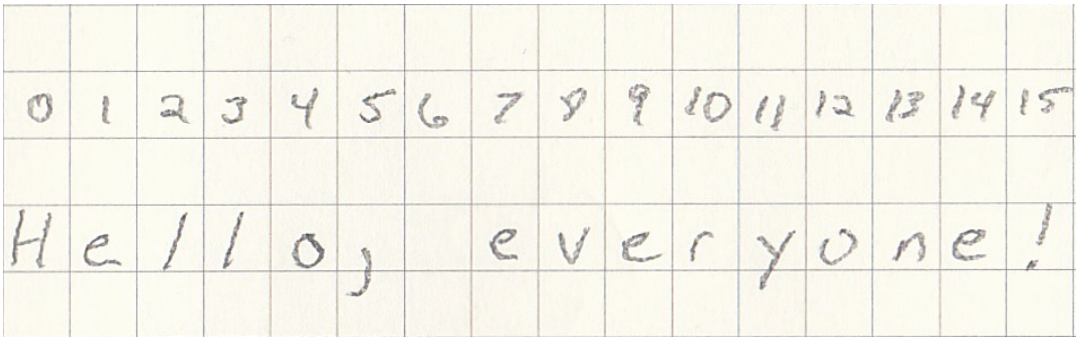
Let's try this again.

```
helloStatement[0]
```

```
'H'
```

There we go!

Visually, this is how the string looks to Python.



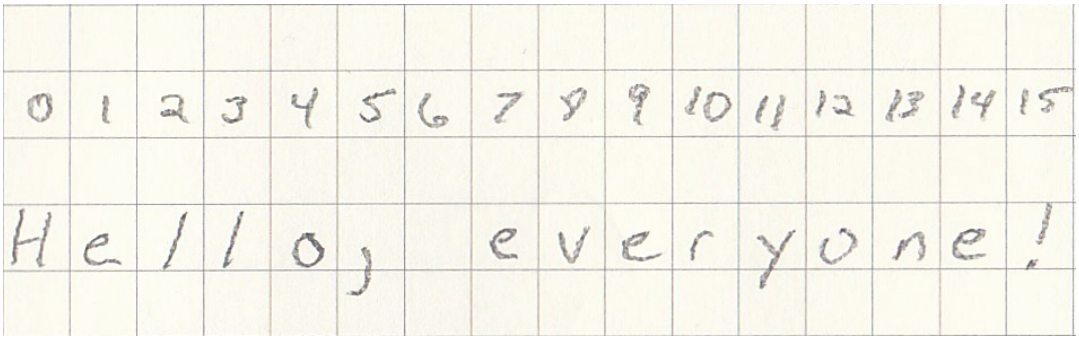
Index value: 0

Indexing Multiple Letters

```
print( helloStatement[0:5] )
```

```
Hello
```

Wait a second!



The way you should think of this is:

```
helloStatement[0 : 5 - 1]
helloStatement[(starting number) to (ending number - 1)]
```

There is also a shortcut way of writing this, without the 0.

```
print( helloStatement[:5] )
```

```
Hello
```

```
print( helloStatement[5:] )
```

```
, everyone!
```

String functions

Find

```
print( helloStatement.find("one") )
print( helloStatement.find("me") )
print( helloStatement.find("hello") )
print( helloStatement.find("Hello") )
```

```
12
-1
-1
0
```

> **Note**
>
> When using *.find()*, if the function can NOT find the sequence of the letters given. It will return -1.

Formatting

```
print( helloStatement.capitalize() )
print( helloStatement.lower() )
```

```
Hello, everyone!
hello, everyone!
```

Split

```
print( helloStatement.split(" ") )
```

```
['Hello,', 'everyone!']
```

> **Note**
>
> *.split()* will eventually become your best friend. *.split()* is a **great** function to use when using uniquelly spaced data. As in comma separated values or CSV.

Chaining Functions

```
print( helloStatement )
print( helloStatement[0:5].capitalize() )
print( helloStatement.find("hello") )
print( helloStatement[:5].lower().find("hello") )
```

```
Hello, everyone!
Hello
-1
0
```

Concatenating Strings

When you want to put two strings together, we say you *concatenate* the strings. There are multiple ways of doing this but presented are what I believe to be the three most common ways.

+ Method

This is the most straightforward method of the three, but there can be some issues. You simply add a plus sign + between your strings. Let's take a look at this.

```
print ( "hello, " + "everyone!")
```

```
hello, everyone!
```

This works fine, but when you add a number to this idea. We run into issues.

```
print ( "hello, " + "every" + 1 + "!")

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-41-1f53f06cad5c> in <module>
----> 1 print ( "hello, " + "every" + 1 + "!")

TypeError: can only concatenate str (not "int") to str
```

In this case we need to *type-cast* the integer as a string using

```
str()
```

```
print ( "hello, " + "every" + str(1) + "!")
```

```
hello, every1!
```

% Method

This is my favorite method out of the three. Let's see how this works with the same example.

In this case, we use a %s (s = string) for each string we want to embed in our overall string.

```
print ( "%s, %s" % ("hello", "everyone") )
```

```
hello, everyone
```

There are three parts to this.

The format

-

"%s, %s"

```
*The break*
* ```python
%
```

The fill

-

("hello", "everyone")

```
We have two %s, meaning we need to feed it with two strings.
```


0:00

(No Audio) Video of how % concatenate works with substituting strings.

OK, but what about numbers?

```
print ( "%s, %s%s%s" % ("hello","every",1,"!") )
```

```
hello, every1!
```

Still works! This reason is why I like this method. You pick the formating and feed in the strings.

join() Method

The .join() method uses a function called

```
.join()
```

This is a create function to be aware of, as it will allow you the ability to join strings with a specific, static format. What do I mean by static formatting? Well, unlike the % method, that can be formatted exactly how I want it. The .join() method requires a specific pattern. Example time!

```
print ( " ".join(["hello, ", "everyone!"]) )
```

```
hello,  everyone!
```

There are two parts to this.

The splitter

-

" "

The fill
* ```python
.join(["hello, ", "everyone!"])

Notice that the join has the brackets around it. Technically, you are feeding this an array or list (we have not talked about this yet). This function again, like *.split()*, will be a great asset to you in the future.

Let's show this with our number again.

```
print ( " ".join(["hello, ", "every", 1, "!"]) )

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-54-e926f0c4c025> in <module>
----> 1 print ( " ".join(["hello, ", "every", 1, "!"]) )

TypeError: sequence item 2: expected str instance, int found
```

The same issue as before, we need to type-cast.

```
print ( " ".join(["hello, ", "every", str(1), "!"]) )
```

```
hello,  every 1 !
```

Notice the spaces? Again, we are saying with *the splitter* what each string is going to be seperated by, so in this case, everything will be split by spaces.

Booleans

Booleans are used to do comparisions (true/false), (1/0), (yes/no)

```
someCondition = True
type( someCondition )
```

```
bool
```

Boolean Logic

We will talk about boolean logic more in the next section (Comparisons)

```
(someCondition == False)
```

```
False
```

```
if (False):
    print( "yes for False!" )
if (True):
    print( "yes for True!" )
```

```
yes for True!
```

> **Note**
>
> A more "traditional" way to do booleans is to use 0 and 1. In Python, any number other than 0 is True. Including negative numbers and decimals.

```
if (0):
    print( "yes for 0!" )
if (1):
    print( "yes for 1!" )
if (2):
    print( "yes for 2!" )
if (-3):
    print( "yes for -3!" )
if (.4):
    print( "yes for .4!" )
```

```
yes for 1!
yes for 2!
yes for -3!
yes for .4!
```

Lists

Lists (or also known as Arrays) are exactly that. A list of data.

There are two options for creating a *List*.

1. Define the list initially

```
groceryList = ["apple", "banana", "eggs"]
print( groceryList )
```

```
['apple', 'banana', 'eggs']
```

1. Create a list and add to it using

```
.append()
```

```
groceryList = []
groceryList.append("apple")
groceryList.append("banana")
groceryList.append("eggs")
print( groceryList )
```

```
['apple', 'banana', 'eggs']
```

Note

For indexes, we always start at the number 0. So, 0 is the first thing, 1 is the second thing, and so on.

```
print( groceryList[2] )
print( groceryList[0] )
print( groceryList[1] )
```

```
eggs
apple
banana
```

So what happens if we use an *index* outside of our list?

```
print( groceryList[3] )

---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-44-0a77fb05d512> in <module>
    print( groceryList[3] )

IndexError: list index out of range
```

> **Note**
>
> Typically, going through an array, one index at a time is not how we want to use lists. We will talk about going through lists using a *loop* in an upcoming notebook.

Dictionary

Dictionaries are used to index based on a specific key. As in:

dictionary[“street adddress” (key)] = “123 Apple St.” (value)

```
personalInformation = {}
personalInformation["streetAddress"] = "123 Apple St."
personalInformation["firstName"] = "Patrick"
personalInformation["lastName"] = "Dudas"
print( personalInformation )
```

```
{'streetAddress': '123 Apple St.', 'firstName': 'Patrick', 'lastName': 'Dudas'}
```

Note the order.

Again, to do this more efficiently, we will be using loops (we will talk about later).

Comparison Operators

We need to be able to compare different variables. We will be working on:

- Are these things the same?
- Are these things not the same?
- How do these things compare?

We can compare any data type, and our output will be a boolean (True or False). The other things we will cover are:

- Comparing different data types
- Making multiple comparisons at once

Comparison operators are important on their own (how do these things compare?) and are also useful for sorting and switching (see the next notebook).

Are these things the same?

Numeric Comparisons

We have already initiated variables by setting something equal to something else - let’s do that here by setting kitten 🐈 equal to 10 and then setting dog 🐕 equal to kitten 🐈. Finally, 🐝 bee will be equal to 11.

So...

🐈 = 10

🐕 = 🐈

🐝 = 11

```
kitten = 10
dog = kitten
bee = 11

print( "kitten =", kitten, "; dog =", dog, "; bee = ", bee )
```

```
kitten = 10 ; dog = 10 ; bee =  11
```

The first comparison operator is ‘==’, which tests to see if two variables are equal.

```
print( "kitten =", kitten, "; dog =", dog, "; bee = ", bee )

print( "Is kitten equal to dog?")
print( kitten == dog )

print( "Is kitten equal to bee?")
print( kitten == bee )
```

```
kitten = 10 ; dog = 10 ; bee =  11
Is kitten equal to dog?
True
Is kitten equal to bee?
False
```

This tells us that kitten is equal to dog, because it returns *True* and kitten is not equal to bee, as that returns *False*.

Character Comparisons

We can also do comparisons with other variable types. Here's an example with strings instead of integers.

Let's think about some foods, how about:

- food1 = 🍏
- food2 = 🍪
- food3 = 🍏

```
food1 = 'apple'
food2 = 'cookie'
food3 = 'apple'
print( "food1=", food1,"; food2 =", food2,"; food3 = ", food3 )

print( "Is food1 equal to food2?")
print( food1 == food2 )

print( "Is food1 equal to food3?")
print( food1 == food3 )
```

```
food1= apple ; food2 = cookie ; food3 =  apple
Is food1 equal to food2?
False
Is food1 equal to food3?
True
```

Are these things different?

This is Logical... NOT!

We can also test to see if two values are not equal using the '!=' operator.

```
print( "food1 =", food1,"; food2 =", food2,"; food3 =", food3 )

print( "Is food1 not equal to food2?")
print( food1 != food2 )

print( "Is food1 not equal to food3?")
print( food1 != food3 )
```

```
food1 = apple ; food2 = cookie ; food3 = apple
Is food1 not equal to food2?
True
Is food1 not equal to food3?
False
```

This gives us the opposite of what we had before.

So, what did we learn?

🍏 == 🍏 = *True*

🍏 != 🍪 = *True*

How do these things compare?

Math Comparisons 101

We can also compare the magnitude of values using '<', '<=', '>'and '>=', which will return 'True' if the condition is being met.

```
print( "kitten =", kitten, "; dog =", dog, "; bee = ", bee )
```

```
kitten = 10 ; dog = 10 ; bee =  11
```

```
print( "Is kitten less than dog?")
print( kitten < dog )

print( "Is kitten less than or equal to dog?")
print( kitten <= dog )

print( "Is kitten greater than or equal to dog?")
print( kitten >= dog )

print( "Is kitten greater than dog?")
print( kitten > dog )
```

```
Is kitten less than dog?
False
Is kitten less than or equal to dog?
True
Is kitten greater than or equal to dog?
True
Is kitten greater than dog?
False
```

Note

We do have to watch out for our types. Characters and numerics are **not** the same.

```
TheCharacters = "10"
TheNumbers = 10

print( "Is TheNumbers equal to TheCharacters?")
print( TheNumbers == TheCharacters )
print( "TheNumbers type is ", type( TheNumbers ), "; and TheCharacters type is ", type(
TheCharacters ) )
```

```
Is TheNumbers equal to TheCharacters?
False
TheNumbers type is  <class 'int'> ; and TheCharacters type is  <class 'str'>
```

We can compare integers and floats (!) but not other disparate data types.

If you let python take care of your data-types, be warned that they could be different from what you think they are!

Toy (Car) Problem

To start thinking of these concepts from a logical perspective, let's create a toy (car) problem. Here are a bunch of toy cars with various colors and costs. Here is how they labeled.

auto:

- car
- truck

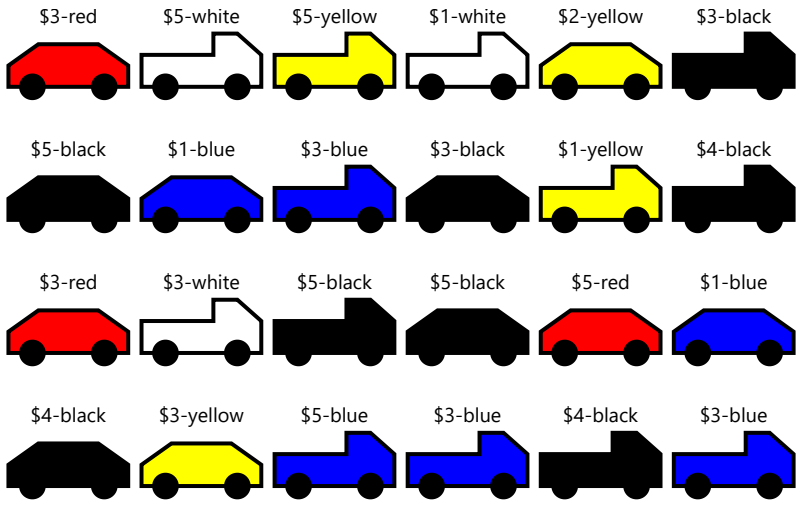
color:

- red
- blue
- yellow

- white
- black

cost:

- 1 <= cost <= 5



Python: auto == "car" search

> **Note**
>
> varible = varible is **not** the same thing as variable == variable
>
> varible = varible will **always** return true

Multiple Comparisons

We can make multiple comparisons at once by stringing the statements

- and
- not
- or

together.

The individual testable (true/false) components need to be broken apart. For example,

- If the *V* CATA bus is coming around the corner, then I need to run towards the bus stop.

requires several things for it to be true and to require running. We can break these things out with:

- If there is a vehicle coming around the corner **AND** that vehicle is a CATA bus **AND** that CATA bus is a V
  - then I need to run towards the bus stop

We will only run towards the bus stop if all of the statements are true.

AND

> **Note**
>
> the **and** operator will return True if all of the conditions are met

Let's create another scenario for this around clothes. For this, let's assume:

face = 😎

shirt = 👕

pants = 👖

```
face = "sunglasses"
shirt = "tshirt"
pants = "jeans"

print ( "Am I wearing sunglasses and jeans?" )
print (face == "sunglasses")
print (pants == "jeans")
print( (face == "sunglasses") and (pants == "jeans") )

print ( "Am I wearing sweater and jeans?" )
print (shirt == "sweater")
print (pants == "jeans")
print( (shirt == "sweater") and (pants == "jeans") )
```

```
Am I wearing sunglasses and jeans?
True
True
True
Am I wearing sweater and jeans?
False
True
False
```

We can also string as many comparisons together as we want.

```
print( (1 < 2) and (1 < 3) and (1 < 4) and (1 < 5) and (1 < 6) and (1 < 7) and (1 < 8) )
```

```
True
```

OR

> **Note**
>
> the **or** operator will return True if at least *1* of the conditions is met

```
print( "face =", face, "; shirt =", shirt, "; pants = ", pants )

print ( "Am I wearing sunglasses or jeans?" )
print (face == "sunglasses")
print (pants == "jeans")
print( (face == "sunglasses") or (pants == "jeans") )

print ( "Am I wearing sweater or jeans?" )
print (shirt == "sweater")
print (pants == "jeans")
print( (shirt == "sweater") or (pants == "jeans") )
```

```
face = sunglasses ; shirt = tshirt ; pants =  jeans
Am I wearing sunglasses or jeans?
True
True
True
Am I wearing sweater or jeans?
False
True
True
```

Not

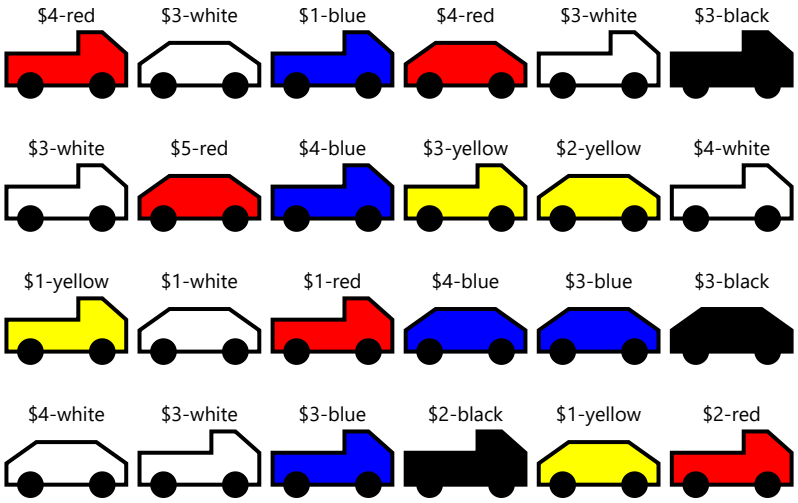
> **Note**
>
> the **not** will reverse or switch the meaning of the and/or operators

```
print( "face =", face, "; shirt =", shirt, "; pants = ", pants )

print ( "Am I wearing sunglasses and not jeans?" )
print (face == "sunglasses")
print (not (pants == "jeans"))
print( (face == "sunglasses") and not (pants == "jeans") )

print ( "Am I wearing jeans and not a sweater?" )
print (not (shirt == "sweater"))
print (pants == "jeans")
print( not (shirt == "sweater") and (pants == "jeans") )
```

```
face = sunglasses ; shirt = tshirt ; pants =  jeans
Am I wearing sunglasses and not jeans?
True
False
False
Am I wearing jeans and not a sweater?
True
True
True
```



Python: auto == "car" search

Try It Out!

Try to fill in code to fulfill the request! Here are some variables used in the exercise

```
dogA_color = 'brown'
dogA_mass = 42
dogA_sex = 'male'
dogA_age = 5
dogA_name = 'chip'

dogB_color = 'white'
dogB_mass = 19
dogB_sex = 'female'
dogB_age = 2
dogB_name = 'lady'
```

Is dogA the same color as dogB? (False)

```
# Example:
print( dogA_color == dogB_color )
```

```
False
```

Does dogA have the same name as dogB? (False)

```
# Try it out here:
```

Is dogA older than dogB? (True)

```
# Try it out here:
```

Is dogA the same sex as dogB? (False)

```
# Try it out here:
```

Is dogA heavier than dogB and have a different name than dogB? (True)

```
# Try it out here:
```

Does dogA have a different age than dogB and not a different sex than dogB? (False)

```
# Try it out here:
```

If-Else Conditions

We can condition our data using if-else statements and switch cases. If-else statements allow us to do different things if a certain criterion is met or not. We can count the odds and evens in our someNumbers list.

if

The *if* statement starts with if and then lists a condition that may or may not is met. If the condition is true, we do what is listed. If it is not, we move on.

Our example here is straightforward; if answer is greater than 30, print something.

```
answer = 42

if answer > 30:
    print( "This number is greater than 30")
```

```
This number is greater than 30
```

OK, same concept.

```
answer = 42

if answer > 50:
    print( "This number is greater than 50")
```

Note

Note the structure of a Python if/else statement where some languages use { } to denote the start and end of the if/else statement. Python uses spaces.

if (condition): <-colon

<- space or tab

Anything that is also spaced or tab is *part* of the if statement.

Where the if Starts and Ends

As mentioned in our note, the if/else statement uses spacing to indicate where it starts and ends. To highlight this, let's look at an example.

```
print("Into the If/Else!")

if (10 < 2):
    print("In the If/Else!")

    print("Still in the If/Else!")




    print("How do I get out of here!?")

print("Out of the If/Else!")
```

```
Into the If/Else!
Out of the If/Else!
```

else

In these examples, only the numbers that are greater than 30 and 50 will get any response. We can add a response for values that do not meet the conditional statement found within the if using an *else* statement.

```
answer = 42

if answer > 30:
    print( answer, "> 30")
else:
    print( answer, "< 30")

if answer > 50:
    print( answer, "> 50")
else:
    print( answer, "< 50")
```

```
42 > 30
42 < 50
```

elif (else if)

If-else statements can also be stacked together to allow for additional sorting using multiple conditions. The way this is done in python is by using

```
elif
```

This will chain conditions, but once one condition is true. It will stop ✋

Let's take a look at an example.

```
favoriteColor = "Yellow"

if (favoriteColor == "Red"):
    print ("My favorite color is red.")
elif (favoriteColor == "Orange"):
    print ("My favorite color is orange.")
elif (favoriteColor == "Yellow"):
    print ("My favorite color is yellow.")
elif (favoriteColor == "Green"):
    print ("My favorite color is green.")
elif (favoriteColor == "Blue"):
    print ("My favorite color is blue.")
elif (favoriteColor == "Indigo"):
    print ("My favorite color is indigo.")
elif (favoriteColor == "Violet"):
    print ("My favorite color is violet.")
else:
    print ("I don't have a favorite color.")
```

```
My favorite color is yellow.
```

Switch Cases

Switch Cases are specialized if-else statements - the criteria must be met precisely. Let's work on an example that changes between month numbers and month names. We first set up a dictionary that will be used for the evaluation. We will use numbers as the keys and use the abbreviations as the values.

```
monthSwap =
{1:'Jan',2:'Feb',3:'Mar',4:'Apr',5:'May',6:'Jun',7:'Jul',8:'Aug',9:'Sep',10:'Oct',11:'Nov',12:'
Dec'}
```

```
print( "monthSwap = ", monthSwap)

# Print out the 5th month
print( "\n#-# Key = 5")
print( monthSwap.get(5,"oooooops!") )

# Print out the 19th month
print( "\n#-# Key = 19")
print( monthSwap.get(19,"oooooops!") )
```

```
monthSwap =  {1: 'Jan', 2: 'Feb', 3: 'Mar', 4: 'Apr', 5: 'May', 6: 'Jun', 7: 'Jul', 8: 'Aug',
9: 'Sep', 10: 'Oct', 11: 'Nov', 12: 'Dec'}

#-# Key = 5
May

#-# Key = 19
oooooops!
```

Loops

One of the programming features is that we have many options to do the same tasking multiple times. The three methods we will be looking at are:

- Functions (later notebook)
- For loops
- While Loops

For Loops

Loops allow us to do the same thing to each item in a list or array. One of the most basic types of loops is a *for loop* - this allows us to iterate over any sequence.

We set up a for loop using 2 things:

- loop variable - the value of the sequence currently being used
- sequence - the data we iterate over

The sequence can be any list. We set up *for loop* using the *for* and *in* keywords, a colon, and all of the code within the *for loop* indented.

```
exampleList = ['a', 'niner', 6, 6.1, 'V@@@', 1001/2, 42]

print( exampleList )
```

```
['a', 'niner', 6, 6.1, 'V@@@', 500.5, 42]
```

Now, before we talked about accessing elements in a list or array by their index. Meaning, if we wanted to print this out, we would need to...

```
print( exampleList[0] )
print( exampleList[1] )
print( exampleList[2] )
print( exampleList[3] )
print( exampleList[4] )
print( exampleList[5] )
print( exampleList[6] )
```

```
a
niner
6
6.1
V@@@
500.5
42
```

> ### Looping Over Values

Very time consuming and frustrating 😵.

Loops make this soooooooo much easier. There are three parts to a *for loop*.

```
for variable_name_we_make_up in our_list_name:
    do_something_with_each_value( variable_name_we_make_up )
```

As stated, variable_name_we_make_up is something we makeup and is used to represent the value as we loop through our, well,... loop.

```
groceryList = ["apple", "banana", "eggs"]
```

Remember me?

```
groceryList = ["apple", "banana", "eggs"]

for itemInOurList in groceryList:
    print (itemInOurList)
```

```
apple
banana
eggs
```

Like mentioned, we name the variable. Here is the same idea again.

```
groceryList = ["apple", "banana", "eggs"]

for steve in groceryList:
    print (steve)
```

```
apple
banana
eggs
```

Going back to our original list. See how much easier it is to print these values?

```
for item in exampleList:
    print (item)
```

```
a
niner
6
6.1
V@@@
500.5
42
```

Looping Over Indices

Sometimes, it's helpful to iterate using indices. For example, linear algebra heavy calculations will almost always use indices to make working with vectors and matrices easier.

We can use the

```
len()
```

and

```
range()
```

functions to show the length and create indices. We can then iterate using the index rather than the values. Let's show off these functions.

```
groceryList = ["apple", "banana", "eggs"]
print ( len(groceryList) )
```

```
3
```

```
print ( range(3) )
```

```
range(0, 3)
```

Note

range() can be a bit misleading. The range is always one less than what you might expect. Meaning, *range(0,3)* goes from 0 to 1 to 2 to... that's it. So when using *range()* think about it as *range(starting number, ending number - 1)*

```
for index in range(len(groceryList)):
    print("index:",index,"value:",groceryList[index])
```

```
index: 0 value: apple
index: 1 value: banana
index: 2 value: eggs
```

You may have noticed that the second line is indented. Like we saw before with If/Else statements. This indent is how we indicate what is in the loop. Our loop can have many lines (all indented). The first line that isn't indented indicates we are out of the loop. This indent is the python syntax for in and out of the loop; other coding languages use other things such as braces {}. Note that blank lines don't matter, just indentation.

```
print( "Starting the loop" )
for val in groceryList:
    print( "\t", "item:", val )

    print( "\t", "Inside the loop" )
print( "Outside the loop" )
```

```
Starting the loop
         item: apple
         Inside the loop
         item: banana
         Inside the loop
         item: eggs
         Inside the loop
Outside the loop
```

While loops

For loops are used when you have something you are iterating over - you know the length. You can use a while loop if you don't know the number of times something will run. The while loop code requires a conditional statement; the loop will continue to run as long as this is true and will not run again as soon as it is false.

Conceptual Example

You can think about taking a test in two different ways.

> Scenario: You are looking through your junk drawer for your sunglasses

For loop:

```
for item in junk_drawer:
    if (item == "sunglasses"):
        "put them on" 😎
    else:
        "keep looking"
```

While loop:

```
while item != "sunglasses":
    "keep looking"
    item = "some item in the junk drawer"
"put them on" 😎
```

Can you see where each has their unique take on looping? Of course, you don't; you are wearing sunglasses indoors. Take them off first, then check out their uniqueness.

The condition being set by the while statement will cause this to run as long as the statement is true.

```
counting = 0

while (counting < 10):
    print ( "before:", counting )
    counting = counting + 1
    print ("\t","after:",counting)
```

```
before: 0
         after: 1
before: 1
         after: 2
before: 2
         after: 3
before: 3
         after: 4
before: 4
         after: 5
before: 5
         after: 6
before: 6
         after: 7
before: 7
         after: 8
before: 8
         after: 9
before: 9
         after: 10
```

One thing to note is that the while loop won't ever be entered if the condition is false when the statement begins as false.

```
startAtTen = 10

while (startAtTen < 10):
    print ( "before:", startAtTen )
    counting = counting + 1
    print ("\t","after:",startAtTen )
```

😈 A VERY MEAN Example 😈

Let's see where we can use this type of loop, in this 😈 VERY MEAN Example 😈. We are creating a set of 30 random numbers from 1 to 50. The *while* will run until it hits its first even number and print this out. Can you spot its MEAN intention?

```
import random
randomList = [random.randrange(1, 50, 1) for i in range(30)]
print ( randomList[0:5] )

index = 0
print ("start loop")
while ( randomList[index] % 2 ):
    index = index + 1
print ( "the first even number is:", randomList[index])
```

```
[26, 20, 24, 33, 28]
start loop
the first even number is: 26
```

So why is this very mean?! Look at our warning.

> ⚠️ **Warning**
>
> While loops will keep iterating as long as the statement stays true. Infinite loops are caused by a condition that always stays true. Use the stop button (◻ but filled in) to stop this erroneous code. Here is an example of this type of code.

```
counting = 0

while (counting < 0):
    print ( "This the loop that never ends. Yes, it goes on and on, my friend!" )
    print ( "Some people started looping it not knowing what it was, " )
    print ( "and they'll continue looping it forever just because..." )
    counting = counting + 1
```

This is 😈 A VERY MEAN Example 😈 because it is possible to have a set without a single even number. The odds of picking an even or an odd is a coin flip (50%). Now do this 30 times. What are the odds of flipping a coin 30 times without a single “Tails?”

$\frac{1}{2}$ = 1 coin

$\frac{1}{2} * \frac{1}{2}$ = 2 coins

$\frac{1}{2} * \frac{1}{2} * \frac{1}{2}$ = 3 coins

$(\frac{1}{2})^n$ = n coin

$(\frac{1}{2})^{30}$ = 30 coin = $(\frac{1}{1073741824})$ OR one in 1 billion, 73 million, 741 thousand, 824.

Meaning, a person out of 1073741824 will have an infinite loop!

MUAHAHAHA!!!

Your Turn

Try to fill in code to fulfill the request! Here is a variable used in the excercises

```
aListOfNumbers = [6, 3, 4, 5, 7, 8, 9 ]
```

Write a function that returns the length of aListOfNumbers as well as the maximum value. Hint: max() is a built-in function

```
# Try it here:
```

Use a for loop to add up all of the numbers in aListOfNumbers.

```
# Try it here:
```

Use a while loop to find the first number in aListOfNumbers that is both greater than 5 and a multiple of 4.

```
# Try it here:
```

Count the number of values in aListOfNumbers that are:

- even
- odd and divisible by three
- odd and not divisible by three

using if, elif and else.

```
# Try it here:
```

Create a dictionary with keys 1-8 corresponding to the words one, two, three, etc. Loop through aListofNumbers to print out the word corresponding to the digit and provide a default value of 'Not Found' if the key is not contained within the dictionary. You should get: six three four five seven eight Not Found

```
# Try it here:
```

Loading a Library

Module or Library?

Modules are python's way of organizing functions, variables and constructors, similar to libraries in other languages. In this section, we will look at:

- Using existing python modules
- Building our own modules
- Finding the things that are within modules

Built in Modules

Python uses modules to make additional functionality available. Modules can be thought of as libraries with many functions, data types, and characteristics that can be used once loaded.

We load modules using the import statement:

- Highly recommend import using a name (import module as name)
- Use the name to keep multiply defined functions separate
- You can import only individual functions from a module
- You can also rename functions.

```
# Import all functions using a name
import numpy as np
# We then use the name to refer to functions from this module
print( np.sin( 1./2. * np.pi ) )

# We can also import just some of the functions, as well as change their names
from math import cos as mathCos
print( mathCos( np.pi ) )
```

```
1.0
-1.0
```

Some common python modules are:

- numpy
- matplotlib
- math
- scipy
- pandas

Modules based on their topic can be found: https://wiki.python.org/moin/UsefulModules

Some modules are already included on the system. You may have to add or update some yourself. Python uses pip for module addition, which includes dependencies. Typically users will put modules in their own space using –user, rather than install them globally. For example, to add cython and to update matplolib you would run in a cell:

```
!pip install cython --user

!pip install matplotlib --user --upgrade
```

Homade Modules

You can also build your own modules. To do this, place what you want in a python file (.py) in your working directory. Note that you need an empty file named **init**.py in this directory as well. You can then load your module just like the built-in modules. For example, if a file called adamsTrigs.py contained:

```
import math as mt
def printTrigVals(angle):
    print( angle,mt.sin(angle),mt.cos(angle),mt.tan(angle) )
```

then I can load the module and use this function locally by either importing everything in the file, or just the function we are interested in.

```
import adamsTrigs as aT
aT.printTrigVals( np.pi / 3. )

from adamsTrigs import printTrigVals as trigVals
trigVals( np.pi / 5)
```

```
1.0471975511965976 0.8660254037844386 0.5000000000000001 1.7320508075688767
0.6283185307179586 0.5877852522924731 0.8090169943749475 0.7265425280053608
```

Helpful hints for modules

Dir

Python has a built-in command called dir to show the directory (contents) of a module. This command provides lots of info, including the function names.

```
import adamsTrigs as aT
dir( aT )
```

```
['__builtins__',
 '__cached__',
 '__doc__',
 '__file__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
 'mt',
 'printTrigVals']
```

We can also use dir to see what is currently available to use:

```
dir()
```

```
['In',
 'Out',
 '_',
 '_3',
 '__',
 '___',
 '__builtin__',
 '__builtins__',
 '__doc__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
 '_dh',
 '_i',
 '_i1',
 '_i2',
 '_i3',
 '_i4',
 '_ih',
 '_ii',
 '_iii',
 '_oh',
 'aT',
 'exit',
 'get_ipython',
 'mathCos',
 'np',
 'quit',
 'trigVals']
```

Main

We can also define code that will only run if the file is being run directly. This will not run if the file is being loaded by another.

```
if __name__ == "__main__":
    # Make a figure of the local data
    # Print out some local variables to provide examples for later use
    print( "This is being run from here" )
```

```
This is being run from here
```

This is often used in files to read in data, where individual data sets are read into discrete files, which are then read into the main function as modules. You can make figures showing individual data sets to test if you are reading correctly in the read file. Rather than make these each time you include that data read file as a module, you can put the plotting in an if statement like this.

Check yourself

Call the math version of tan() mathTan and print out tangent of pi/2. (Hint, pi can come from math or numpy).

```
# Try it here
```

Does numpy include functions called log10 and banana?

```
# Try it here
```

Creating a Function

Functions allow us to do repeated tasks easily by writing the code only once. Functions will have a name, inputs, and outputs and can be called anywhere the task is repeated.

There are functions that are built into python; for example, we have already been using the type() function, which tells us the type of variable we are using. Note that print is also a function!

```
aVal = 10.0
print( type( aVal ) )
```

```
<class 'float'>
```

Functions have four typical parts:

- Name - what you call your function
- Input arguments - what you provide
- Outputs - what the function gives back
- Math/Magic - what the function does

Creating Our Own Function

In python, we use def to define a function with the function name and inputs followed by a colon. The python function is then separated from the rest of the code by a tab. Some languages use braces rather than indentation.

```
def functionName( inputs ):
    # Operate on the inputs
    ouputs = inputs + 5
    # Return what we want to back
    return outputs;
```

Let's look at an example function, which changes degrees Fahrenheit to Celsius.

```
def changeFromFToC( farVal ):
    cVal = (farVal - 32.0) * 5.0 / 9.0
    return cVal
```

Here, our function name is *changeFromFToC*, the input is *farVal*, the temperature in Fahrenheit, the output is *cVal*, and the temperature in Celsius. We can print or store the output from the function. Note that the function has to be defined before we use it - the cell with the function definition has to have run before we can call the function.

```
print( "Change 14 deg F to Celsius" )
print( changeFromFToC( 14 ) )

print( "Change from 68 deg F to Celsius" )
niceTempC = changeFromFToC( 68 )
print( niceTempC )
```

```
Change 14 deg F to Celsius
-10.0
Change from 68 deg F to Celsius
20.0
```

Your turn! What is the temperature today? Convert it to Celsius.

For those who have the temperature in Celsius and want to convert it to Fahrenheit. Define a new function to do this.

Multiple inputs and outputs

Here is an example of multiple outputs. We can actually work the output in a couple of different ways.

Multiple Output Function

```
def changeFromFToCAndK( farVal ):
    # Change the temperature from Fahrenheit to Celsius and Kelvin
    cVal = (farVal - 32.0) * 5.0 / 9.0
    kVal= cVal + 273.15
    return cVal, kVal
```

Output: List

```
def changeFromFToCAndK( farVal ):
    # Change the temperature from Fahrenheit to Celsius and Kelvin
    cVal = (farVal - 32.0) * 5.0 / 9.0
    kVal= cVal + 273.15
    return cVal, kVal

print( "Change 14 deg F to Celsius and Kelvin" )
print( changeFromFToCAndK( 14 ) )

print( "Change 32 deg F to Celsius and Kelvin" )
freezing = changeFromFToCAndK( 32 )
print( freezing[0] )
print( freezing[1] )
```

```
Change 14 deg F to Celsius and Kelvin
(-10.0, 263.15)
Change 32 deg F to Celsius and Kelvin
0.0
273.15
```

Output: Multiple Variables

```
print( "Change 212 deg F to Celsius and Kelvin" )
boilingC, boilingK = changeFromFToCAndK( 212 )
print( boilingC )
print( boilingK )
```

```
Change 212 deg F to Celsius and Kelvin
100.0
373.15
```

Multiple Input Function

```
def changeFromFToCOrK( farVal, tempType ):
    if (tempType == "C"):
        return (farVal - 32.0) * 5.0 / 9.0
    elif (tempType == "K"):
        return ((farVal - 32.0) * 5.0 / 9.0) + 273.15
    else:
        return "invalid temperature type"
```

```
print ( changeFromFToCOrK(70,"C") )
```

```
21.11111111111111
```

```
print ( changeFromFToCOrK(70,"K") )
```

```
294.26111111111106
```

```
print ( changeFromFToCOrK(70,"W") )
```

```
invalid temperature type
```

Functions calling other functions

Functions can call other functions. Here we add three using a function to add two and then adding one.

```
def addTwo( inValAddTwo ):
    # Add two to the input
    return inValAddTwo + 2

def addThree( inValAddThree ):
    # Add three two the input
    return addTwo( inValAddThree ) + 1

# Add three to four
print( "Add three to four using addThree (which adds 1) but calls addTwo (which adds 2)" )
print( addThree( 4 ) )
```

```
Add three to four using addThree (which adds 1) but calls addTwo (which adds 2)
7
```

Recursive Functions

Functions can also call themselves - these are called recursive functions. See how we calculate the factorial by subtracting one and calling the function again.

```
def factorialRecursive( facIn ):
    print ("In factorialRecursive() and the current number is:", facIn)
    if facIn == 1:
        return facIn
    else:
        return facIn * factorialRecursive( facIn - 1)

print( "Use a recursive function to calculate the factorial of 5" )
print( factorialRecursive( 5 ) )
```

```
Use a recursive function to calculate the factorial of 5
In factorialRecursive() and the current number is: 5
In factorialRecursive() and the current number is: 4
In factorialRecursive() and the current number is: 3
In factorialRecursive() and the current number is: 2
In factorialRecursive() and the current number is: 1
120
```

Function Gotcha! 😖

Note

The biggest gotcha on functions is with variable scope:

- Variables defined in a function are not accessible from the outside
- Functions have access to more than just the variables passed in

```
def addAnAnimal( animal ):
    print ("\t","in the function")
    print ("\t","I have access to dog:",dog)
    print ("\t","I have access to animal:",animal)
    newValue = animal + 1
    print ("\t","I have access to newValue:",newValue)
    return newValue

print ("outside the function")
dog = 10
print("dog:", dog)
print ("function output:",addAnAnimal( dog ))
```

```
outside the function
dog: 10
         in the function
         I have access to dog: 10
         I have access to animal: 10
         I have access to newValue: 11
function output: 11
```

If we would add:

```
print (newValue)
```

to the bottom, we would end up with this:

```
def addAnAnimal( animal ):
    print ("\t","in the function")
    print ("\t","I have access to dog:",dog)
    print ("\t","I have access to animal:",animal)
    newValue = animal + 1
    print ("\t","I have access to newValue:",newValue)
    return newValue

print ("outside the function")
dog = 10
print("dog:", dog)
print ("function output:",addAnAnimal( dog ))
print (newValue)
```

outside the function

dog: 10

in the function

I have access to dog: 10

I have access to animal: 10

I have access to newValue: 11

function output: 11

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-32-07cce689eb00> in <module>
     11 print("dog:", dog)
     12 print ("function output:",addAnAnimal( dog ))
---> 13 print (newValue)

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

Your Turn

Try to fill in code to fulfill the request! Here is a variable used in the excercises.

```
aListOfNumbers = [6, 3, 4, 5, 7, 8, 9 ]
```

Write a function that returns the length of aListOfNumbers as well as the maximum value. Hint: max() is a built-in function

```
## try here!
```

Let's create a shopping list!

First, let's just create a list

```
shoppingList = []
```

Let's create a function to add new items to my shopping list.

```
def addToShoppingList(itemToAdd):
    shoppingList.append(itemToAdd)
```

Walking around my kitchen I notice we are missing milk.

```
addToShoppingList("milk")
```

We have 3 eggs left (always want 6 on hand).

```
numberOfEggs = 3
if (numberOfEggs < 6): addToShoppingList("eggs")
```

I want to make spaghetti for dinner (need sauce, spaghetti, and bread).

```
spaghettiRecipe = ["sauce","spaghetti","bread"]
for ingredient in spaghettiRecipe:
    addToShoppingList(ingredient)
```

Let's see what we have so far.

```
print( shoppingList )
```

```
['milk', 'eggs', 'sauce', 'spaghetti', 'bread']
```

Not the easiest to read. Let's try this another way:

```
for item in shoppingList:
    print( item )
```

```
milk
eggs
sauce
spaghetti
bread
```

Second, let's add some prices

So I want to keep an eye on my budget, so I note all the prices.

First, I will set all prices to 0.

```
shoppingPrices = {}
for item in shoppingList:
    shoppingPrices[item] = 0
print( shoppingPrices )
```

```
{'milk': 0, 'eggs': 0, 'sauce': 0, 'spaghetti': 0, 'bread': 0}
```

Now I will manually add prices for each item.

```
shoppingPrices['spaghetti'] = .99
shoppingPrices['eggs'] = 1.99
shoppingPrices['sauce'] = 2.15
shoppingPrices['milk'] = 2.85
shoppingPrices['bread'] = 1.49
```

```
print( shoppingPrices )
```

```
{'milk': 2.85, 'eggs': 1.99, 'sauce': 2.15, 'spaghetti': 0.99, 'bread': 1.49}
```

Also, I don't want to manually put in values.

For this example, I will just use random values. Let's see an example of how to do this:

```
# first, import the random library
import random
# to create a random integer, use randint and the range (lowest values, highest values (-1))
print( random.randint(1,21) )
# now make this into a decimal price
# example for eggs, let's assume the price can go from 1.50 to 2.50 (NOTE the 251)
print( random.randint(150,251)/100 )
```

```
17
1.95
```

why are we only getting two options (1 or 2)?

We need a decimal! So let's type-cast using float().

```
print( random.randint(150,251)/float(100) )
print( random.randint(150,251)/float(100) )
print( random.randint(150,251)/float(100) )
print( random.randint(150,251)/float(100) )
```

```
1.92
2.42
1.55
2.11
```

Let's define are ranges:

- Spaghetti (.75 - 1.00)
- Eggs (1.50 - 2.50)
- Sauce (2.00 - 3.00)
- Milk (2.50 - 4.00)
- Bread (1.25 - 1.75)

```
def createRandomPrice(item):
    if (item=="spaghetti"): return random.randint(75,100)/float(100)
    if (item=="eggs"): return random.randint(150,250)/float(100)
    if (item=="sauce"): return random.randint(200,300)/float(100)
    if (item=="milk"): return random.randint(250,400)/float(100)
    if (item=="bread"): return random.randint(125,175)/float(100)
```

Third, some serious code here

Idea!! I will go to multiple stores to get the best prices!

Here is what I need:

We will need a list of stores (Wegmans, Wal-Mart, Giant, Trader Joes).

For each store, we will need a list of the stores and our shopping list.

Let's start with our store names:

```
storeNames = ["Wegmans","Wal-Mart","Giant","Trader Joes"]
```

```
for storeName in storeNames:
    print( storeName )
```

```
Wegmans
Wal-Mart
Giant
Trader Joes
```

OK, let's test this:

```
for item in shoppingList:
    print( item, createRandomPrice( item ) )
```

```
milk 3.18
eggs 1.57
sauce 2.73
spaghetti 0.89
bread 1.34
```

This is what we want in the end. For each store, we have a price for each item.

food	**Wegmans**	**Wal-Mart**	**Giant**	**Trader Joes**
spaghetti	x.xx	x.xx	x.xx	x.xx
eggs	x.xx	x.xx	x.xx	x.xx
sauce	x.xx	x.xx	x.xx	x.xx
milk	x.xx	x.xx	x.xx	x.xx
bread	x.xx	x.xx	x.xx	x.xx

First, we need a Dictionary for each item and for each item another Dictionary for the store/prices.

```
shoppingPricesByStores = {}
for item in shoppingList:
    shoppingPricesByStores[item] = {}
print( shoppingPricesByStores )
```

```
{'milk': {}, 'eggs': {}, 'sauce': {}, 'spaghetti': {}, 'bread': {}}
```

Now for each item, we set both the store name and the random price.

```
for item in shoppingPricesByStores:
    shoppingPricesByStores[item] = {}
    for storeName in storeNames:
        shoppingPricesByStores[item][storeName] = createRandomPrice(item)
print( shoppingPricesByStores )
```

```
{'milk': {'Wegmans': 2.53, 'Wal-Mart': 3.17, 'Giant': 3.87, 'Trader Joes': 2.95}, 'eggs':
{'Wegmans': 1.93, 'Wal-Mart': 1.99, 'Giant': 2.15, 'Trader Joes': 2.45}, 'sauce': {'Wegmans':
2.5, 'Wal-Mart': 2.35, 'Giant': 2.31, 'Trader Joes': 2.54}, 'spaghetti': {'Wegmans': 0.85,
'Wal-Mart': 0.96, 'Giant': 0.92, 'Trader Joes': 0.78}, 'bread': {'Wegmans': 1.52, 'Wal-Mart':
1.72, 'Giant': 1.74, 'Trader Joes': 1.65}}
```

Finally, we need to sort. We use a sorting function (sorted).

Let's try first with just sorted (shoppingPricesByStores[item]).

```
print( sorted( shoppingPricesByStores ) )
```

```
['bread', 'eggs', 'milk', 'sauce', 'spaghetti']
```

Hmm… not what we are looking for. We want to sort based on prices per store.

```
for item in shoppingPricesByStores:
    print( item )
    print( sorted( shoppingPricesByStores[item] ) )
```

```
milk
['Giant', 'Trader Joes', 'Wal-Mart', 'Wegmans']
eggs
['Giant', 'Trader Joes', 'Wal-Mart', 'Wegmans']
sauce
['Giant', 'Trader Joes', 'Wal-Mart', 'Wegmans']
spaghetti
['Giant', 'Trader Joes', 'Wal-Mart', 'Wegmans']
bread
['Giant', 'Trader Joes', 'Wal-Mart', 'Wegmans']
```

This is sorting based on the name of the store; we need to sort based on the amount.

There is actually a function we can use, called items(), which as it sounds, gives us all items in our dictionary.

```
shoppingPricesByStores.items()
```

```
dict_items([('milk', {'Wegmans': 2.53, 'Wal-Mart': 3.17, 'Giant': 3.87, 'Trader Joes':
2.95}), ('eggs', {'Wegmans': 1.93, 'Wal-Mart': 1.99, 'Giant': 2.15, 'Trader Joes': 2.45}),
('sauce', {'Wegmans': 2.5, 'Wal-Mart': 2.35, 'Giant': 2.31, 'Trader Joes': 2.54}),
('spaghetti', {'Wegmans': 0.85, 'Wal-Mart': 0.96, 'Giant': 0.92, 'Trader Joes': 0.78}),
('bread', {'Wegmans': 1.52, 'Wal-Mart': 1.72, 'Giant': 1.74, 'Trader Joes': 1.65})])
```

So let's try using that.

```
for item in shoppingPricesByStores:
    print( item )
    print( shoppingPricesByStores[item] )
    print( sorted( shoppingPricesByStores[item].items() ) )
```

```
milk
{'Wegmans': 2.53, 'Wal-Mart': 3.17, 'Giant': 3.87, 'Trader Joes': 2.95}
[('Giant', 3.87), ('Trader Joes', 2.95), ('Wal-Mart', 3.17), ('Wegmans', 2.53)]
eggs
{'Wegmans': 1.93, 'Wal-Mart': 1.99, 'Giant': 2.15, 'Trader Joes': 2.45}
[('Giant', 2.15), ('Trader Joes', 2.45), ('Wal-Mart', 1.99), ('Wegmans', 1.93)]
sauce
{'Wegmans': 2.5, 'Wal-Mart': 2.35, 'Giant': 2.31, 'Trader Joes': 2.54}
[('Giant', 2.31), ('Trader Joes', 2.54), ('Wal-Mart', 2.35), ('Wegmans', 2.5)]
spaghetti
{'Wegmans': 0.85, 'Wal-Mart': 0.96, 'Giant': 0.92, 'Trader Joes': 0.78}
[('Giant', 0.92), ('Trader Joes', 0.78), ('Wal-Mart', 0.96), ('Wegmans', 0.85)]
bread
{'Wegmans': 1.52, 'Wal-Mart': 1.72, 'Giant': 1.74, 'Trader Joes': 1.65}
[('Giant', 1.74), ('Trader Joes', 1.65), ('Wal-Mart', 1.72), ('Wegmans', 1.52)]
```

First, I will define a function (I will explain this more in a second).

Also, I marked a couple of print statements to showcase what is going on.

All this function does is take tuple and returns the second item in the tuple.

REMEMBER, computer talk here.. 1 = 2, because it starts at 0, 1, 2,…

```
def comparePrices (itemToSort):
    print( "#From comparePrices function#" )
    print( itemToSort )
    print( "At the store "+itemToSort[0]+" the price is "+str( itemToSort[1] ) )
    print( "#End comparePrices function#" )
    return itemToSort[1]
```

We can add a "key" that tells the sorting function, which value to sort on.

So reading this in more non-technical speak:

1. For each item in dictionary called shoppingPricesByStores
2. Print item name

3. Then sort by the pair (storeName and price)
4. Use the function comparePrices to tell sorting to sort by the price.

```
for item in shoppingPricesByStores:
    print( "####################In for loop####################" )
    print( item )
    print( sorted(shoppingPricesByStores[item].items(), key=comparePrices) )
    print( "####################Out for loop####################" )
```

```
####################In for loop####################
```

```
milk
#From comparePrices function#
('Wegmans', 2.53)
At the store Wegmans the price is 2.53
#End comparePrices function#
#From comparePrices function#
('Wal-Mart', 3.17)
At the store Wal-Mart the price is 3.17
#End comparePrices function#
#From comparePrices function#
('Giant', 3.87)
At the store Giant the price is 3.87
#End comparePrices function#
#From comparePrices function#
('Trader Joes', 2.95)
At the store Trader Joes the price is 2.95
#End comparePrices function#
[('Wegmans', 2.53), ('Trader Joes', 2.95), ('Wal-Mart', 3.17), ('Giant', 3.87)]
####################Out for loop####################
####################In for loop####################
eggs
#From comparePrices function#
('Wegmans', 1.93)
At the store Wegmans the price is 1.93
#End comparePrices function#
#From comparePrices function#
('Wal-Mart', 1.99)
At the store Wal-Mart the price is 1.99
#End comparePrices function#
#From comparePrices function#
('Giant', 2.15)
At the store Giant the price is 2.15
#End comparePrices function#
#From comparePrices function#
('Trader Joes', 2.45)
At the store Trader Joes the price is 2.45
#End comparePrices function#
[('Wegmans', 1.93), ('Wal-Mart', 1.99), ('Giant', 2.15), ('Trader Joes', 2.45)]
####################Out for loop####################
####################In for loop####################
sauce
#From comparePrices function#
('Wegmans', 2.5)
At the store Wegmans the price is 2.5
#End comparePrices function#
#From comparePrices function#
('Wal-Mart', 2.35)
At the store Wal-Mart the price is 2.35
#End comparePrices function#
#From comparePrices function#
('Giant', 2.31)
At the store Giant the price is 2.31
#End comparePrices function#
#From comparePrices function#
('Trader Joes', 2.54)
At the store Trader Joes the price is 2.54
#End comparePrices function#
[('Giant', 2.31), ('Wal-Mart', 2.35), ('Wegmans', 2.5), ('Trader Joes', 2.54)]
####################Out for loop####################
####################In for loop####################
spaghetti
#From comparePrices function#
('Wegmans', 0.85)
At the store Wegmans the price is 0.85
#End comparePrices function#
#From comparePrices function#
('Wal-Mart', 0.96)
At the store Wal-Mart the price is 0.96
#End comparePrices function#
#From comparePrices function#
('Giant', 0.92)
At the store Giant the price is 0.92
#End comparePrices function#
#From comparePrices function#
('Trader Joes', 0.78)
At the store Trader Joes the price is 0.78
#End comparePrices function#
[('Trader Joes', 0.78), ('Wegmans', 0.85), ('Giant', 0.92), ('Wal-Mart', 0.96)]
####################Out for loop####################
####################In for loop####################
bread
#From comparePrices function#
('Wegmans', 1.52)
At the store Wegmans the price is 1.52
#End comparePrices function#
#From comparePrices function#
('Wal-Mart', 1.72)
```

```
At the store Wal-Mart the price is 1.72
#End comparePrices function#
#From comparePrices function#
('Giant', 1.74)
At the store Giant the price is 1.74
#End comparePrices function#
#From comparePrices function#
('Trader Joes', 1.65)
At the store Trader Joes the price is 1.65
#End comparePrices function#
[('Wegmans', 1.52), ('Trader Joes', 1.65), ('Wal-Mart', 1.72), ('Giant', 1.74)]
####################Out for loop####################
```

We can short-hand this a bit by using lambda.

Lambda is like a micro-function. We use it once and that's it.

```
for item in shoppingPricesByStores:
    print( item )
    print( sorted(shoppingPricesByStores[item].items(), key=lambda storeAmount: storeAmount[1]
) ) # Python 3
    #print( sorted(shoppingPricesByStores[item].items(), key=lambda (store,amount): (amount)) )
# Python 2
```

```
milk
[('Wegmans', 2.53), ('Trader Joes', 2.95), ('Wal-Mart', 3.17), ('Giant', 3.87)]
eggs
[('Wegmans', 1.93), ('Wal-Mart', 1.99), ('Giant', 2.15), ('Trader Joes', 2.45)]
sauce
[('Giant', 2.31), ('Wal-Mart', 2.35), ('Wegmans', 2.5), ('Trader Joes', 2.54)]
spaghetti
[('Trader Joes', 0.78), ('Wegmans', 0.85), ('Giant', 0.92), ('Wal-Mart', 0.96)]
bread
[('Wegmans', 1.52), ('Trader Joes', 1.65), ('Wal-Mart', 1.72), ('Giant', 1.74)]
```

Let's print this out a bit neater.

```
for item in shoppingPricesByStores:
    print( item )
    #for key, value in sorted(shoppingPricesByStores[item].items(), key=lambda (store,amount):
(amount)): # Python 2
    for key, value in sorted(shoppingPricesByStores[item].items(), key=lambda storeAmount:
storeAmount[1] ): # Python 3
        print( "%s: %s" % (key, value) )
    print( "\n" )
```

```
milk
Wegmans: 2.53
Trader Joes: 2.95
Wal-Mart: 3.17
Giant: 3.87


eggs
Wegmans: 1.93
Wal-Mart: 1.99
Giant: 2.15
Trader Joes: 2.45


sauce
Giant: 2.31
Wal-Mart: 2.35
Wegmans: 2.5
Trader Joes: 2.54


spaghetti
Trader Joes: 0.78
Wegmans: 0.85
Giant: 0.92
Wal-Mart: 0.96


bread
Wegmans: 1.52
Trader Joes: 1.65
Wal-Mart: 1.72
Giant: 1.74
```

A Python guessing game for jupyter

Play the existing game with the first cell! The only things here that we did not talk about in Seminar 1 are:

- isdigit() - a function to see if a string can be transformed into an integer without error
- randInt from the random module - a function to create a random integer

```
#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#
# guessingGame
# A python number guessing game
#
# Adam Lavely
# adam.lavely.psu@gmail.com
#
# Created for ICS Python Training Series
# Spring 2019
#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#

#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#
# This is a number guessing game.  Python
# uses a random number generator to get
# a target between minVal and maxVal
# and then the guesser must guess this
# number using greater/less than clues.
#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#

#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#
# Load the modules we need

# Use random to get the random number
import random as rd

#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#
# Functions required

def guessChecker( guessVal ):
    # Function to verify that the guess an integer
    if guessVal.isdigit() == True:
        return True
    else:
        print("Bad input, please use an integer")
        return False

#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#
# Background info
minVal=1
maxVal=100
goodGuesses=1
ANSWER=rd.randint( minVal, maxVal)
answerRight = 'Nope'


#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#
# Play the game
print( "Guess an integer such that", minVal, " <= X <= ", maxVal )

while answerRight == 'Nope':

    # Get the guess

    guess = input("\nEnter an integer: ")

    # Check to make sure the guess is an integer
    goodAns = guessChecker( guess )

    # Yes an integer
    if goodAns == True:
        guessInt = int( guess )

        # Check to see if guess is above the answer
        if guessInt > ANSWER:
            print("Answer is too high, please guess again")
            maxVal = guessInt
            goodGuesses = goodGuesses+1
            print( "Your current inclusive range is ", minVal, " - ", maxVal )
        elif guessInt < ANSWER:
            print("Answer is too low, please guess again")
            minVal = guessInt
            goodGuesses = goodGuesses+1
            print( "Your current inclusive range is ", minVal, " - ", maxVal )
        else:
            print("\nHurray, you guessed ", ANSWER, " in ", goodGuesses, " guesses! " )
            answerRight = 'Yep'

    # Not an integer
    else:
        print( "Please try again")
```

```
Guess an integer such that 1  <= X <=  100
```

```
---------------------------------------------------------------------------
StdinNotImplementedError                  Traceback (most recent call last)
<ipython-input-1-aef5f43d0892> in <module>
     52     # Get the guess
     53
---> 54     guess = input("\nEnter an integer: ")
     55
     56     # Check to make sure the guess is an integer

c:\users\dudas\anaconda3\envs\wintest\lib\site-packages\ipykernel\kernelbase.py in
raw_input(self, prompt)
    844         if not self._allow_stdin:
    845             raise StdinNotImplementedError(
--> 846                 "raw_input was called, but this frontend does not support input
requests."
    847             )
    848         return self._input_request(str(prompt),

StdinNotImplementedError: raw_input was called, but this frontend does not support input
requests.
```

Can you add a range checker to the code using a function? Take an initial guess of 100000001: this is an integer and so will pass the current check, but it is out of the existing range. Your updated code should limit your guesses to the current range. (Note the maxVal is set to 10 here to make debugging easier.)

```
#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#
# guessingGame
# A python number guessing game
#
# Originally written: Adam Lavely
# adam.lavely.psu@gmail.com
#
# Modified by: YOUR NAME
# YOUR EMAIL ADDRESS
#
# Created for ICS Python Training Series
# Spring 2019
#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#

#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#
# This is a number guessing game.  Python
# uses a random number generator to get
# a target between minVal and maxVal
# and then the guesser must guess this
# number using greater/less than clues.
#
# MODIFICATION:
#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#

#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#
# Load the modules we need

# Use random to get the random number
import random as rd

#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#
# Functions required

def guessChecker( guessVal ):
    # Function to verify that the guess an integer
    if guessVal.isdigit() == True:
        return True
    else:
        print("Bad input, please use an integer")
        return False

#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#
# Background info
minVal=1
maxVal=10
goodGuesses=1
ANSWER=rd.randint( minVal, maxVal)
answerRight = 'Nope'


#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#-#
# Play the game
print( "Guess an integer such that", minVal, " <= X <= ", maxVal )

while answerRight == 'Nope':

    # Get the guess

    guess = input("\nEnter an integer: ")

    # Check to make sure the guess is an integer
    goodAns = guessChecker( guess )

    # Yes an integer
    if goodAns == True:
        guessInt = int( guess )

        # Check to see if guess is above the answer
        if guessInt > ANSWER:
            print("Answer is too high, please guess again")
            maxVal = guessInt
            goodGuesses = goodGuesses+1
            print( "Your current inclusive range is ", minVal, " - ", maxVal )
        elif guessInt < ANSWER:
            print("Answer is too low, please guess again")
            minVal = guessInt
            goodGuesses = goodGuesses+1
            print( "Your current inclusive range is ", minVal, " - ", maxVal )
        else:
            print("\nHurray, you guessed ", ANSWER, " in ", goodGuesses, " guesses! " )
            answerRight = 'Yep'

    # Not an integer
    else:
        print( "Please try again")
```

Getting Started with Matplotlib

Why are we using Pandas?

Pandas actually has a few functions that are similar to D3 functions. So, we can learn two things at the same time.

```
import matplotlib.pyplot as plt
import pandas as pd
```

Getting the data

```
url="https://gist.githubusercontent.com/dudaspm/e518430a731ac11f52de9217311c674d/raw/4c2f2bd6639582a420ef321493188deebc4a575e/StateCollege2000-2020.csv"
data = []
data=pd.read_csv(url)
data = data.fillna(0) # replace all NAs with 0s
```

Viewing the data

```
data.head()
```

	DATE	DAY	MONTH	YEAR	PRCP	SNOW	TMAX	TMIN	WT_FOG	WT_THUNDER
0	1/1/2000	1	1	2000	0.00	0.0	44.0	23	0.0	0.0
1	1/2/2000	2	1	2000	0.00	0.0	52.0	23	0.0	0.0
2	1/3/2000	3	1	2000	0.01	0.0	60.0	35	0.0	0.0
3	1/4/2000	4	1	2000	0.12	0.0	62.0	54	0.0	0.0
4	1/5/2000	5	1	2000	0.04	0.0	60.0	30	0.0	0.0

Acknowledgement

Cite as: Menne, Matthew J., Imke Durre, Bryant Korzeniewski, Shelley McNeal, Kristy Thomas, Xungang Yin, Steven Anthony, Ron Ray, Russell S. Vose, Byron E.Gleason, and Tamara G. Houston (2012): Global Historical Climatology Network - Daily (GHCN-Daily), Version 3. CITY:US420020. NOAA National Climatic Data Center. doi:10.7289/V5D21VHZ 02/22/2021.

Publications citing this dataset should also cite the following article: Matthew J. Menne, Imke Durre, Russell S. Vose, Byron E. Gleason, and Tamara G. Houston, 2012: An Overview of the Global Historical Climatology Network-Daily Database. J. Atmos. Oceanic Technol., 29, 897-910. doi:10.1175/JTECH-D-11-00103.1.

Use liability: NOAA and NCEI cannot provide any warranty as to the accuracy, reliability, or completeness of furnished data. Users assume responsibility to determine the usability of these data. The user is responsible for the results of any application of this data for other than its intended purpose.

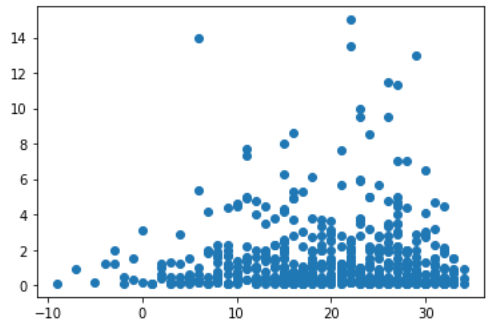
Links: https://data.noaa.gov/onestop/

https://www.ncdc.noaa.gov/cdo-web/search

Correlation

```
import matplotlib.pyplot as plt
snowdays = data[(data.SNOW>0)]

plt.scatter(snowdays.TMIN, snowdays.SNOW)
plt.show()
```



Making Plots

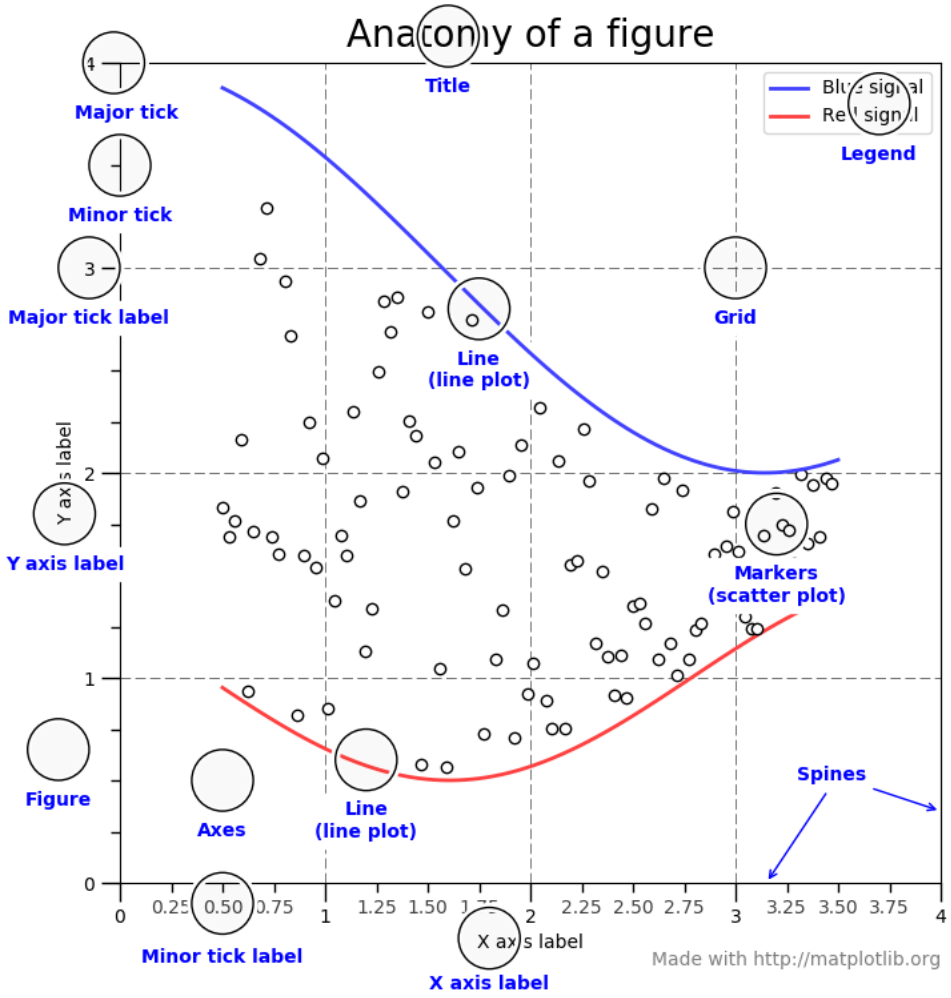
Change Over Time

Note: I worked with Adam Lavely on these notebook, so a big thank you to Adam for helping!

This section covers making plots and doing several basic things:

- Adding gridlines and a legend
- Changing the way the data is shown (linestyles and marker types)
- Adding titles and axis labels
- Creating multiple plots within the same figure
- Adding text to the plots
- Saving the plot to be used outside of the Jupyter environment
- Using the rcParams package to create uniform plots

Basic Plots



> Usage Guide — Matplotlib 3.3.4 documentation. (n.d.). MatPlotLib. Retrieved February 25, 2021, from https://matplotlib.org/stable/tutorials/introductory/usage.html

Plots are ways of showing data. We will use pyplot from the matplotlib library as it is simple and common.

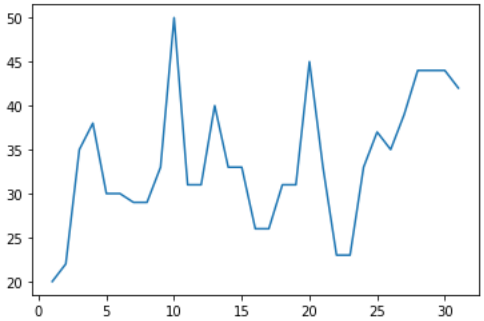
```
import matplotlib.pyplot as plt

# Some data for us to plot
xVals = data[(data.MONTH==3) & (data.YEAR==2020)].DAY #independent
yVals = data[(data.MONTH==3) & (data.YEAR==2020)].TMIN #dependent

# Creat the figure
fig, ax = plt.subplots(1)

# Put the data on the plot
ax.plot( xVals, yVals )

# Show the figure (here in Jupyter)
plt.show( fig )
plt.close( fig )
```



This seems very simple and basic, but let's discuss the objects here to more easily understand how to add new things.

- We're importing a the matplotlib.pyplot module, which gives us access to functions like subplots.
- When subplots is called without any arguments, a single plot is made
- We have an internal name for our figure (fig) and identify the specific set of axes (ax) we are using (this allows us to have multiple subplots on the same figure)
- We can then add things - think about if we are acting on the whole figure, or the individual subplot

We can add things like labels and a title to help provide context.

```
import matplotlib.pyplot as plt

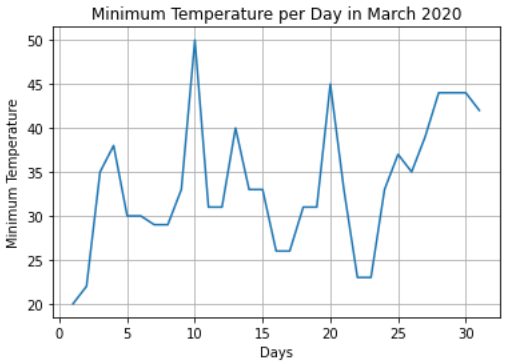
# Some data for us to plot
xVals = data[(data.MONTH==3) & (data.YEAR==2020)].DAY #independent
yVals = data[(data.MONTH==3) & (data.YEAR==2020)].TMIN #dependent

# Creat the figure
fig, ax = plt.subplots(1)

# Put the data on the plot
ax.plot( xVals, yVals )

# Add labels, a title and grid lines to the plot
ax.set_xlabel( 'Days' )
ax.set_ylabel( 'Minimum Temperature' )
plt.title( 'Minimum Temperature per Day in March 2020' )
ax.grid()

# Show the figure (here in Jupyter)
plt.show( fig )
plt.close( fig )
```



We can change how the data is shown on the plot. Some common things to change are linestyle, color and marker. Look at https://matplotlib.org/api/_as_gen/matplotlib.pyplot.plot.html#matplotlib.pyplot.plot for a complete list of options and https://matplotlib.org/2.0.2/examples/color/named_colors.html for the named colors.

You can either specify these things individually, or with shorthand notation

```
import matplotlib.pyplot as plt

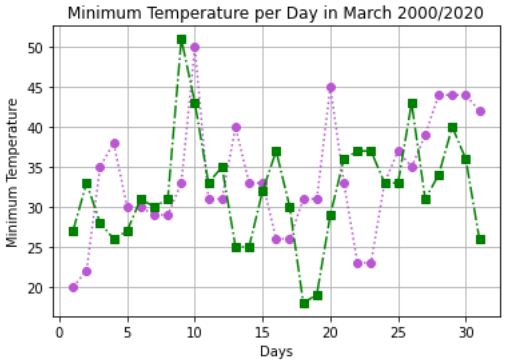
# Some data for us to plot
xVals = data[(data.MONTH==3) & (data.YEAR==2020)].DAY # independent
yVals = data[(data.MONTH==3) & (data.YEAR==2020)].TMIN # dependent
zVals = data[(data.MONTH==3) & (data.YEAR==2000)].TMIN # dependent

# Creat the figure
fig, ax = plt.subplots(1)

# Add our data and change the color, marker type and linestyle
ax.plot( xVals, yVals, color='mediumorchid', marker='o', linestyle=':' )
ax.plot( xVals, zVals, 'gs-.' )

# Add labels, a title and grid lines to the plot
ax.set_xlabel( 'Days' )
ax.set_ylabel( 'Minimum Temperature' )
plt.title( 'Minimum Temperature per Day in March 2000/2020' )
ax.grid()

# Show the figure (here in Jupyter)
plt.show( fig )
plt.close( fig )
```



We can also add legends and text to our figures.

```
import matplotlib.pyplot as plt

# Some data for us to plot
xVals = data[(data.MONTH==3) & (data.YEAR==2020)].DAY # independent
yVals = data[(data.MONTH==3) & (data.YEAR==2020)].TMIN # dependent
zVals = data[(data.MONTH==3) & (data.YEAR==2000)].TMIN # dependent

# Creat the figure
fig, ax = plt.subplots(1)

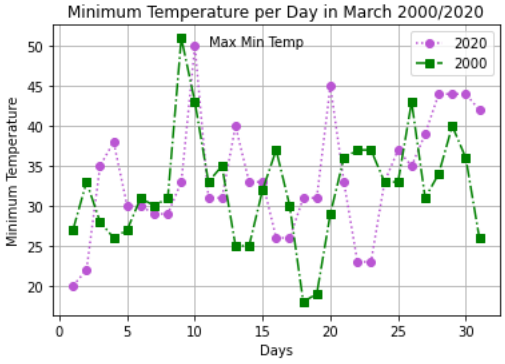
# Add our data and change the color, marker type and linestyle
ax.plot( xVals, yVals, color='mediumorchid', marker='o', linestyle=':',  label = '2020')
ax.plot( xVals, zVals, 'gs-.',  label = '2000')

# Add some text; note the starting point and rotation
plt.text( 11, 50,'Max Min Temp' )

# Add labels, a title and grid lines to the plot
ax.set_xlabel( 'Days' )
ax.set_ylabel( 'Minimum Temperature' )
plt.title( 'Minimum Temperature per Day in March 2000/2020' )
ax.grid()

# Put in the legend - we put it in location 2 (top left)
ax.legend( loc = 0)

# Show the figure (here in Jupyter)
plt.show( fig )
plt.close( fig )
```



Multiple plots on the same figure

Let's take advantage of adding an additional subplot, and change the y-axes of the different plots separately. We use the sharex to indicate that the x-axes should be the same. Note that the figure handle refers to the entire figure and the axes are individual sets of axes that occur within the figure.

```
import matplotlib.pyplot as plt

# Some data for us to plot
xVals = data[(data.MONTH==3) & (data.YEAR==2020)].DAY # independent
yVals = data[(data.MONTH==3) & (data.YEAR==2020)].TMIN # dependent
zVals = data[(data.MONTH==3) & (data.YEAR==2000)].TMIN # dependent

# Create the figure.  Note that there are two plots here, and that they share the x axes
figd, [axA, axB]  = plt.subplots( 2, sharex = True )
print(  "Type of figd:,",type(figd) )
print( "Type of axA:", type(axA) )

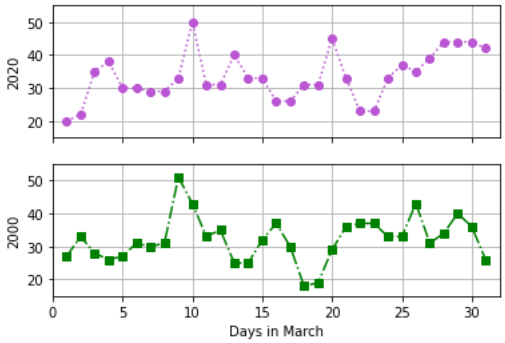
# Add data to the plots.  note that we add to the subplot we desire
axA.plot( xVals, yVals, color = 'mediumorchid', marker = 'o', linestyle = ':' )
axB.plot( xVals, zVals, 'gs-.' )

# Add the axis labels and grid
axA.set_ylabel( '2020' )
axB.set_ylabel( '2000' )
axB.set_xlabel( 'Days in March' )
axA.grid()
axB.grid()

# Set some axis limits - note we only have to set X once
axA.set_xlim( [ 0 , 32 ] )
axA.set_ylim( [ 15, 55 ] )
axB.set_ylim( [ 15, 55 ] )

# Show the figure in the Jupyter environment
plt.show( figd )
plt.close( figd )
```

```
Type of figd:, <class 'matplotlib.figure.Figure'>
Type of axA: <class 'matplotlib.axes._subplots.AxesSubplot'>
```



Saving the file to use later

NOTE: this may require one more step in Google Colab

Everything so far has just been in the jupyter notebook environment. We can save the files using the savefig function. You can then use the Home tab within jupyter to download the file to your local machine.

```
import matplotlib.pyplot as plt

# Some data for us to plot
xVals = data[(data.MONTH==3) & (data.YEAR==2020)].DAY # independent
yVals = data[(data.MONTH==3) & (data.YEAR==2020)].TMIN # dependent
zVals = data[(data.MONTH==3) & (data.YEAR==2000)].TMIN # dependent

# Create the figure.  Note that there are two plots here, and that they share the x axes
figd, [axA, axB]  = plt.subplots( 2, sharex = True )
print(  "Type of figd:,",type(figd) )
print( "Type of axA:", type(axA) )

# Add data to the plots.  note that we add to the subplot we desire
axA.plot( xVals, yVals, color = 'mediumorchid', marker = 'o', linestyle = ':' )
axB.plot( xVals, zVals, 'gs-.' )

# Add the axis labels and grid
axA.set_ylabel( '2020' )
axB.set_ylabel( '2000' )
axB.set_xlabel( 'Days in March' )
axA.grid()
axB.grid()

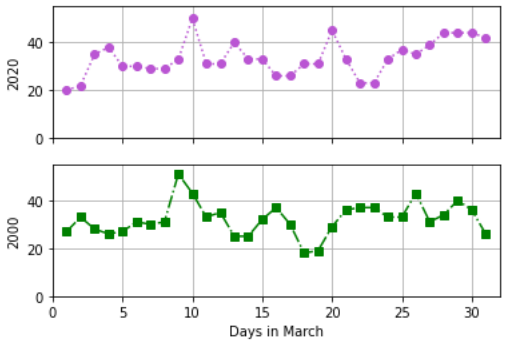
# Set some axis limits - note we only have to set X once
axA.set_xlim( [ 0 , 32 ] )
axA.set_ylim( [ 0, 55 ] )
axB.set_ylim( [ 0, 55 ] )

# Save the figure so that we have access to it outside of Jupyter
plt.savefig( 'MarchWeather.png' )

# Save the figure so that we have access to it outside of Jupyter
plt.savefig( 'MarchWeather.svg' )

# Show the figure in the Jupyter environment
plt.show( figd )
plt.close( figd )
```

```
Type of figd:, <class 'matplotlib.figure.Figure'>
Type of axA: <class 'matplotlib.axes._subplots.AxesSubplot'>
```



Publication Quality Plots

```
import matplotlib.pyplot as plt

# Some data for us to plot
xVals = data[(data.MONTH==3) & (data.YEAR==2020)].DAY # independent
yVals = data[(data.MONTH==3) & (data.YEAR==2020)].TMIN # dependent
zVals = data[(data.MONTH==3) & (data.YEAR==2000)].TMIN # dependent

# Creat the figure
fig, ax = plt.subplots(1)

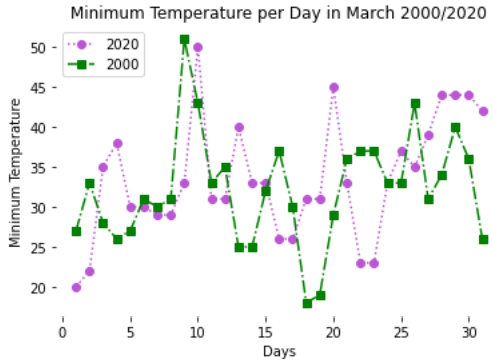
# Add our data and change the color, marker type and linestyle
ax.plot( xVals, yVals, color='mediumorchid', marker='o', linestyle=':',  label = '2020')
ax.plot( xVals, zVals, 'gs-.',  label = '2000')

# Add labels, a title and grid lines to the plot
ax.set_xlabel( 'Days' )
ax.set_ylabel( 'Minimum Temperature' )
plt.title( 'Minimum Temperature per Day in March 2000/2020' )

##################################################
ax.spines['top'].set_visible(False)
ax.spines['right'].set_visible(False)
ax.spines['bottom'].set_visible(False)
ax.spines['left'].set_visible(False)
##################################################

# Put in the legend - we put it in location 2 (top left)
ax.legend( loc = 2 )

# Show the figure (here in Jupyter)
plt.show( fig )
plt.close( fig )
```



Next, I want to move the labels to the end of the lines.

```
import matplotlib.pyplot as plt

# Some data for us to plot
xVals = data[(data.MONTH==3) & (data.YEAR==2020)].DAY # independent
yVals = data[(data.MONTH==3) & (data.YEAR==2020)].TMIN # dependent
zVals = data[(data.MONTH==3) & (data.YEAR==2000)].TMIN # dependent

# Creat the figure
fig, ax = plt.subplots(1)

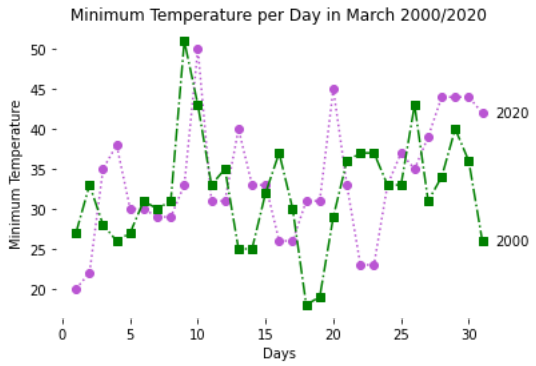
# Add our data and change the color, marker type and linestyle
ax.plot( xVals, yVals, color='mediumorchid', marker='o', linestyle=':',  label = '2020')
ax.plot( xVals, zVals, 'gs-.',  label = '2000')

# Add labels, a title and grid lines to the plot
ax.set_xlabel( 'Days' )
ax.set_ylabel( 'Minimum Temperature' )
plt.title( 'Minimum Temperature per Day in March 2000/2020' )

ax.spines['top'].set_visible(False)
ax.spines['right'].set_visible(False)
ax.spines['bottom'].set_visible(False)
ax.spines['left'].set_visible(False)

##################################################
ax.text(32, yVals.iloc[-1], '2020', horizontalalignment='left', verticalalignment='center')
ax.text(32, zVals.iloc[-1], '2000', horizontalalignment='left', verticalalignment='center')
##################################################


# Show the figure (here in Jupyter)
plt.show( fig )
plt.close( fig )
```



Let's make the Axis Titles larger

```
import matplotlib.pyplot as plt

# Some data for us to plot
xVals = data[(data.MONTH==3) & (data.YEAR==2020)].DAY # independent
yVals = data[(data.MONTH==3) & (data.YEAR==2020)].TMIN # dependent
zVals = data[(data.MONTH==3) & (data.YEAR==2000)].TMIN # dependent

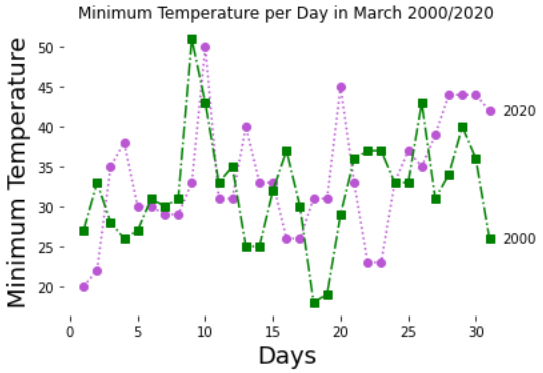
# Creat the figure
fig, ax = plt.subplots(1)

# Add our data and change the color, marker type and linestyle
ax.plot( xVals, yVals, color='mediumorchid', marker='o', linestyle=':',  label = '2020')
ax.plot( xVals, zVals, 'gs-.',  label = '2000')

##################################################
# Add labels, a title and grid lines to the plot
ax.set_xlabel( 'Days' , fontsize=18)
ax.set_ylabel( 'Minimum Temperature' , fontsize=18)
plt.title( 'Minimum Temperature per Day in March 2000/2020' )
##################################################
ax.spines['top'].set_visible(False)
ax.spines['right'].set_visible(False)
ax.spines['bottom'].set_visible(False)
ax.spines['left'].set_visible(False)

ax.text(32, yVals.iloc[-1], '2020', horizontalalignment='left', verticalalignment='center')
ax.text(32, zVals.iloc[-1], '2000', horizontalalignment='left', verticalalignment='center')

# Show the figure (here in Jupyter)
plt.show( fig )
plt.close( fig )
```



Considering we moved the legend titles to the end of each line. This might not be as important, but if wanted to get some “print friendly colors,” we should use colorbrewer.

https://scitools.org.uk/iris/docs/v1.1/userguide/plotting_a_cube.html#brewer-colour-palettes

http://colorbrewer2.org

```
import matplotlib.cm as brewer
for colors in brewer.datad:
    print( colors )
```

```
Blues
BrBG
BuGn
BuPu
CMRmap
GnBu
Greens
Greys
OrRd
Oranges
PRGn
PiYG
PuBu
PuBuGn
PuOr
PuRd
Purples
RdBu
RdGy
RdPu
RdYlBu
RdYlGn
Reds
Spectral
Wistia
YlGn
YlGnBu
YlOrBr
YlOrRd
afmhot
autumn
binary
bone
brg
bwr
cool
coolwarm
copper
cubehelix
flag
gist_earth
gist_gray
gist_heat
gist_ncar
gist_rainbow
gist_stern
gist_yarg
gnuplot
gnuplot2
gray
hot
hsv
jet
nipy_spectral
ocean
pink
prism
rainbow
seismic
spring
summer
terrain
winter
Accent
Dark2
Paired
Pastel1
Pastel2
Set1
Set2
Set3
tab10
tab20
tab20b
tab20c
```

```
import matplotlib.pyplot as plt

# Some data for us to plot
xVals = data[(data.MONTH==3) & (data.YEAR==2020)].DAY # independent
yVals = data[(data.MONTH==3) & (data.YEAR==2020)].TMIN # dependent
zVals = data[(data.MONTH==3) & (data.YEAR==2000)].TMIN # dependent

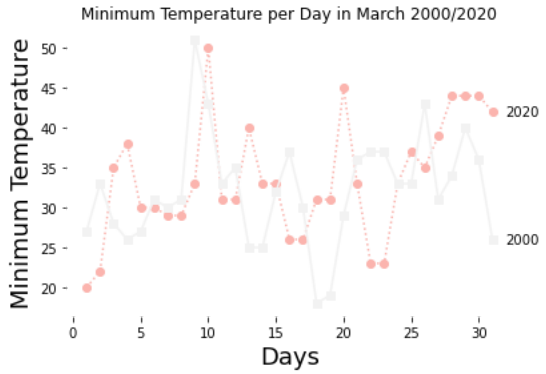
# Creat the figure
fig, ax = plt.subplots(1)
colors = brewer.get_cmap('Pastel1',2)
# Add our data and change the color, marker type and linestyle
ax.plot( xVals, yVals, marker='o', linestyle=':',  label = '2020', color=colors(0))
ax.plot( xVals, zVals, marker='s', linestyle='-',  label = '2000', color=colors(1))

# Add labels, a title and grid lines to the plot
ax.set_xlabel( 'Days' , fontsize=18)
ax.set_ylabel( 'Minimum Temperature' , fontsize=18)
plt.title( 'Minimum Temperature per Day in March 2000/2020' )

ax.spines['top'].set_visible(False)
ax.spines['right'].set_visible(False)
ax.spines['bottom'].set_visible(False)
ax.spines['left'].set_visible(False)

ax.text(32, yVals.iloc[-1], '2020', horizontalalignment='left', verticalalignment='center')
ax.text(32, zVals.iloc[-1], '2000', horizontalalignment='left', verticalalignment='center')

# Show the figure (here in Jupyter)
plt.show( fig )
plt.close( fig )
```



Finally, we might want to highlight a certain data points with a line or lines.

```
import matplotlib.pyplot as plt

# Some data for us to plot
xVals = data[(data.MONTH==3) & (data.YEAR==2020)].DAY # independent
yVals = data[(data.MONTH==3) & (data.YEAR==2020)].TMIN # dependent
zVals = data[(data.MONTH==3) & (data.YEAR==2000)].TMIN # dependent



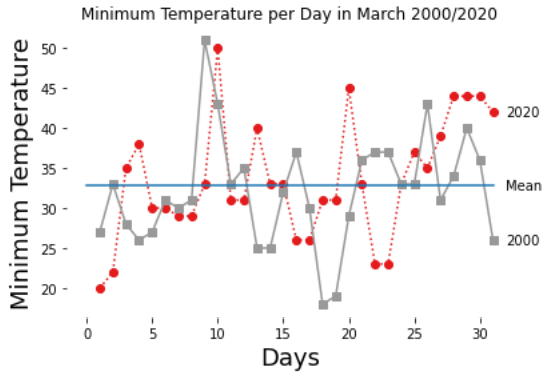
# Creat the figure
fig, ax = plt.subplots(1)
colors = brewer.get_cmap('Set1',2)
# Add our data and change the color, marker type and linestyle
ax.plot( xVals, yVals, marker='o', linestyle=':',  label = '2020', color=colors(0))
ax.plot( xVals, zVals, marker='s', linestyle='-',  label = '2000', color=colors(1))

##################################################
meanValue = pd.concat([yVals,zVals]).mean()
plt.plot([0, 31],[meanValue, meanValue]) # [x1, x2], [y1, y2]
##################################################
# Add labels, a title and grid lines to the plot
ax.set_xlabel( 'Days' , fontsize=18)
ax.set_ylabel( 'Minimum Temperature' , fontsize=18)
plt.title( 'Minimum Temperature per Day in March 2000/2020' )

ax.spines['top'].set_visible(False)
ax.spines['right'].set_visible(False)
ax.spines['bottom'].set_visible(False)
ax.spines['left'].set_visible(False)

ax.text(32, yVals.iloc[-1], '2020', horizontalalignment='left', verticalalignment='center')
ax.text(32, zVals.iloc[-1], '2000', horizontalalignment='left', verticalalignment='center')
ax.text(32, meanValue, 'Mean', horizontalalignment='left', verticalalignment='center')

# Show the figure (here in Jupyter)
plt.show( fig )
plt.close( fig )
```



Changing the size of the figure.

```
import matplotlib.pyplot as plt

##################################################
plt.rcParams["figure.figsize"] = (15,10)
#plt.rcParams["figure.figsize"] = (8,6)
##################################################

# Some data for us to plot
xVals = data[(data.MONTH==3) & (data.YEAR==2020)].DAY # independent
yVals = data[(data.MONTH==3) & (data.YEAR==2020)].TMIN # dependent
zVals = data[(data.MONTH==3) & (data.YEAR==2000)].TMIN # dependent



# Creat the figure
fig, ax = plt.subplots(1)
colors = brewer.get_cmap('Set1',2)
# Add our data and change the color, marker type and linestyle
ax.plot( xVals, yVals, marker='o', linestyle=':',  label = '2020', color=colors(0))
ax.plot( xVals, zVals, marker='s', linestyle='-',  label = '2000', color=colors(1))

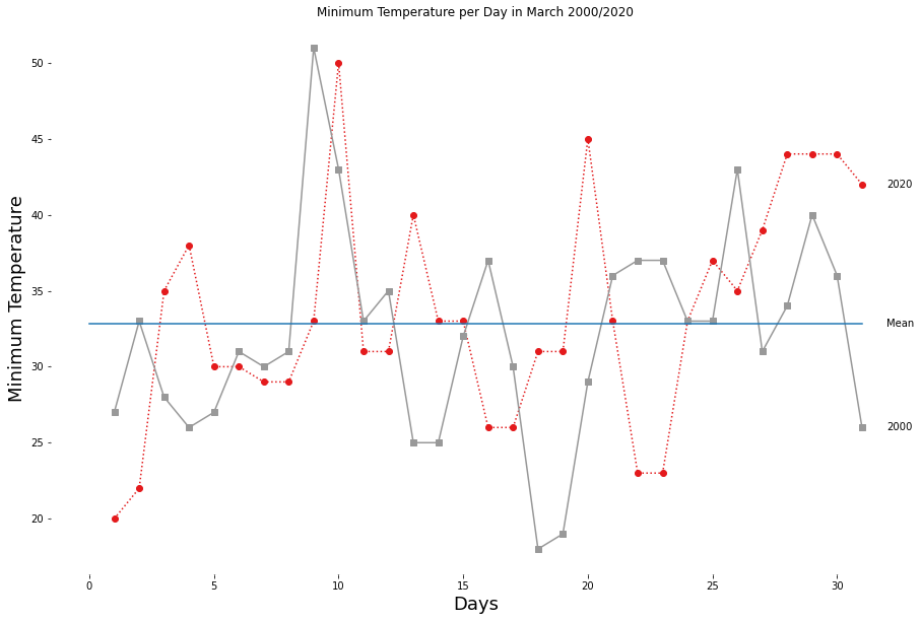
meanValue = pd.concat([yVals,zVals]).mean()
plt.plot([0, 31],[meanValue, meanValue]) # [x1, x2], [y1, y2]

# Add labels, a title and grid lines to the plot
ax.set_xlabel( 'Days' , fontsize=18)
ax.set_ylabel( 'Minimum Temperature' , fontsize=18)
plt.title( 'Minimum Temperature per Day in March 2000/2020' )

ax.spines['top'].set_visible(False)
ax.spines['right'].set_visible(False)
ax.spines['bottom'].set_visible(False)
ax.spines['left'].set_visible(False)

ax.text(32, yVals.iloc[-1], '2020', horizontalalignment='left', verticalalignment='center')
ax.text(32, zVals.iloc[-1], '2000', horizontalalignment='left', verticalalignment='center')
ax.text(32, meanValue, 'Mean', horizontalalignment='left', verticalalignment='center')

# Show the figure (here in Jupyter)
plt.show( fig )
plt.close( fig )
```



first = January, February, and March second = April, May and June third = July, August, and September fourth = October, November, and December

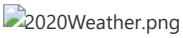
To update the figure size and font, use rcParams:

plt.rcParams[“figure.figsize”] = (14,12)

plt.rcParams.update({‘font.size’: 12})

For the color, I used colorbrewer, set “tab20”

The y-limits were all [20 , 100]



Tables to Idioms

Acknowledgement

Slides created by and for: *Munzner, T. (2014). Visualization analysis and design. CRC press.* [Mun14]

Used by permission of the author.

Image from: *Wickham, H. (2010). A layered grammar of graphics. Journal of Computational and Graphical Statistics, 19(1), 3-28.* [Wic10]

[GG16]
B Granger and J Grout. Jupyterlab: building blocks for interactive computing. *Slides of presentation made at SciPy*, 2016.

[Mun14]
Tamara Munzner. *Visualization analysis and design*. CRC press, 2014.

[Wic10]
Hadley Wickham. A layered grammar of graphics. *Journal of Computational and Graphical Statistics*, 19(1):3–28, 2010.

First the Data

```
import matplotlib.pyplot as plt
import pandas as pd

url="https://raw.githubusercontent.com/owid/covid-19-
data/master/public/data/vaccinations/us_state_vaccinations.csv"
data = []
data=pd.read_csv(url)
data = data.dropna() # removed NAs with 0s

### Remove the United States entries

data = data[data.location != "United States"]

data.total_vaccinations = data.total_vaccinations/100000
data.total_distributed = data.total_distributed/100000
data.people_vaccinated = data.people_vaccinated/100000
```

```
data.head()
```

	date	location	total_vaccinations	total_distributed	people_vaccinated	people_fully_vac
1	2021-01-13	Alabama	0.84040	3.78975	0.74792	
3	2021-01-15	Alabama	1.00567	4.44650	0.86956	
7	2021-01-19	Alabama	1.30795	4.44650	1.14319	
8	2021-01-20	Alabama	1.39200	4.83275	1.21113	
9	2021-01-21	Alabama	1.65919	4.93125	1.44429	

Acknowledgement

Max Roser, Hannah Ritchie, Esteban Ortiz-Ospina and Joe Hasell (2020) - "Coronavirus Pandemic (COVID-19)". Published online at OurWorldInData.org. Retrieved from: 'https://ourworldindata.org/coronavirus' [Online Resource]

Original Link: https://github.com/owid/covid-19-data (Accessed 3/11/2021) Source Link: https://github.com/owid/covid-19-data/tree/master/public/data/vaccinations

Hunter, J. D. (2007). Matplotlib: A 2D graphics environment. IEEE Annals of the History of Computing, 9(03), 90-95.

Today's and Yesterday's date and data

```
from datetime import datetime, timedelta

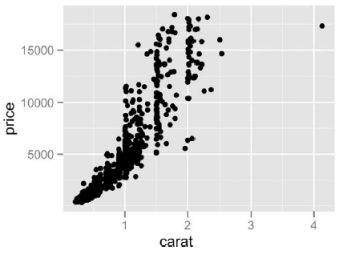
todaysDate = datetime.today()
yesterdaysDate = todaysDate  - timedelta(days=1)
yesterdaysDate = yesterdaysDate.strftime('%Y-%m-%d')
twoDays = todaysDate - timedelta(days=2)
twoDays = twoDays.strftime('%Y-%m-%d')

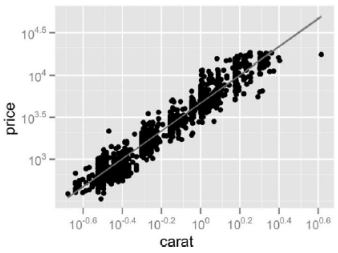
data[data.date==yesterdaysDate].head()
```

	date	location	total_vaccinations	total_distributed	people_vaccinated	people_fully_
183	2021-07-14	Alabama	34.34499	48.93240	20.00914	
368	2021-07-14	Alaska	6.84753	8.41265	3.69636	
553	2021-07-14	American Samoa	0.49040	0.54030	0.27279	
738	2021-07-14	Arizona	69.07835	82.54080	37.46711	
923	2021-07-14	Arkansas	22.97956	28.79750	13.04706	

Idiom: **scatterplot**

- ***express*** values
  - –quantitative attributes
- no keys, only values
  - –data
    - 2 quant attribs
  - –mark: points
  - –channels
    - horiz + vert position
  - –tasks
    - find trends, outliers, distribution, correlation, clusters
  - –scalability
    - hundreds of items





[A layered grammar of graphics. Wickham. Journ. Computational and Graphical Statistics 19:1 (2010), 3–28.]

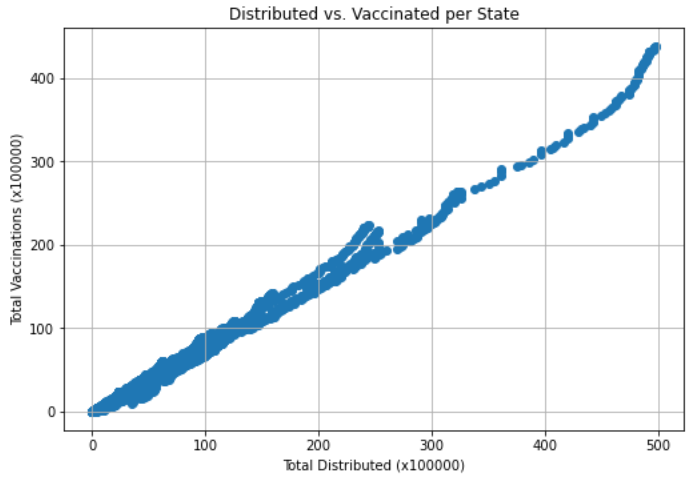
Scatter Plot — Matplotlib 3.3.4 Documentation. https://matplotlib.org/stable/gallery/shapes_and_collections/scatter.html. Accessed 14 Mar. 2021.

```
px = 1/plt.rcParams['figure.dpi']  # pixel in inches
fig, ax = plt.subplots(1,figsize=(600*px, 400*px))

ax.scatter(data.total_distributed, data.total_vaccinations)

# Add labels, a title and grid lines to the plot
ax.set_xlabel( 'Total Distributed (x100000)' )
ax.set_ylabel( 'Total Vaccinations (x100000)' )
plt.title( 'Distributed vs. Vaccinated per State' )
ax.grid()

# Show the figure (here in Jupyter)
plt.show( fig )
plt.close( fig )
```

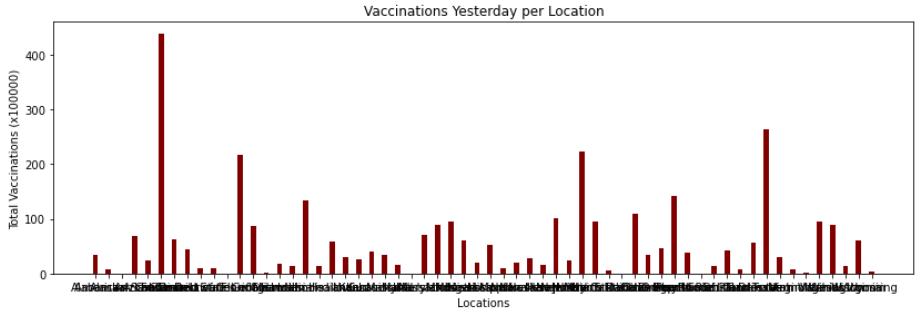


Bar Chart Idiom

Grouped Bar Chart with Labels — Matplotlib 3.3.4 Documentation. https://matplotlib.org/stable/gallery/lines_bars_and_markers/barchart.html. *Accessed 14 Mar. 2021.*

```
# creating the bar plot
yData = data[data.date==yesterdaysDate]
px = 1/plt.rcParams['figure.dpi']  # pixel in inches
fig, ax = plt.subplots(1,figsize=(1000*px, 300*px))
ax.bar(yData.location, yData.total_vaccinations, color ='maroon', width = 0.4)

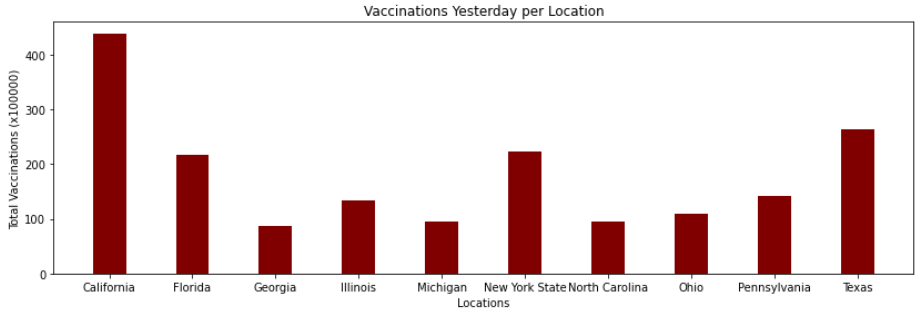
# Add labels, a title and grid lines to the plot
ax.set_xlabel( 'Locations' )
ax.set_ylabel( 'Total Vaccinations (x100000)' )
plt.title( 'Vaccinations Yesterday per Location' )
# Show the figure (here in Jupyter)
plt.show( fig )
plt.close( fig )
```



```
# creating the bar plot
top10 = ["California","Texas","Florida","New York
State","Illinois","Pennsylvania","Ohio","Georgia","North Carolina","Michigan"]
yData = data[data.date==yesterdaysDate]

top10Yest = yData[yData.location.isin(top10)]
px = 1/plt.rcParams['figure.dpi']  # pixel in inches
fig, ax = plt.subplots(1,figsize=(1000*px, 300*px))
ax.bar(top10Yest.location, top10Yest.total_vaccinations, color ='maroon', width = 0.4)

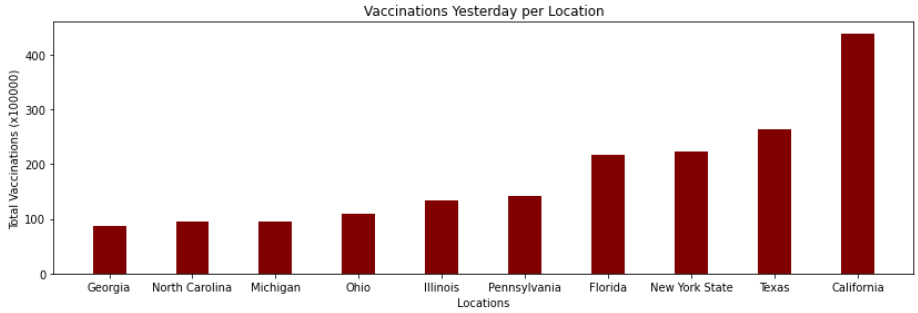
# Add labels, a title and grid lines to the plot
ax.set_xlabel( 'Locations' )
ax.set_ylabel( 'Total Vaccinations (x100000)' )
plt.title( 'Vaccinations Yesterday per Location' )
# Show the figure (here in Jupyter)
plt.show( fig )
plt.close( fig )
```




```
# creating the bar plot
top10 = ["California","Texas","Florida","New York
State","Illinois","Pennsylvania","Ohio","Georgia","North Carolina","Michigan"]
yData = data[data.date==yesterdaysDate]

sortedData = yData.sort_values(by=['total_vaccinations'])
top10Yest = sortedData[sortedData.location.isin(top10)]
px = 1/plt.rcParams['figure.dpi']  # pixel in inches
fig, ax = plt.subplots(1,figsize=(1000*px, 300*px))
ax.bar(top10Yest.location, top10Yest.total_vaccinations, color ='maroon', width = 0.4)

# Add labels, a title and grid lines to the plot
ax.set_xlabel( 'Locations' )
ax.set_ylabel( 'Total Vaccinations (x100000)' )
plt.title( 'Vaccinations Yesterday per Location' )
# Show the figure (here in Jupyter)
plt.show( fig )
plt.close( fig )
```



Paired Bar Chart

```
import numpy as np
# creating the bar plot
top10 = ["California","Texas","Florida","New York
State","Illinois","Pennsylvania","Ohio","Georgia","North Carolina","Michigan"]
yData = data[data.date==yesterdaysDate]
sortedData = yData.sort_values(by=['total_vaccinations'])
top10Yest = sortedData[sortedData.location.isin(top10)]

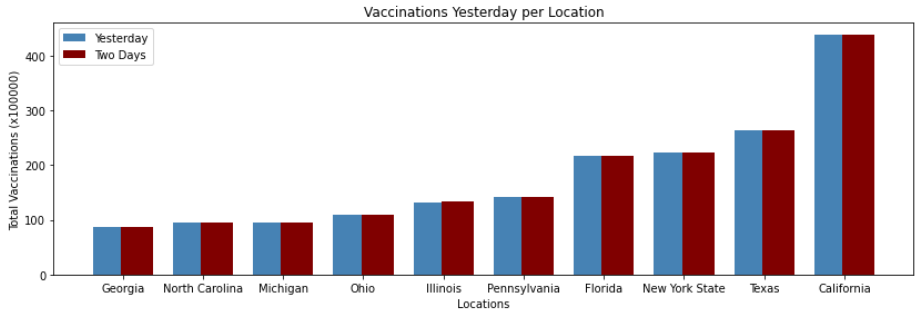
twoData = data[data.date==twoDays]
sortedData = twoData.sort_values(by=['total_vaccinations'])
top102days = sortedData[sortedData.location.isin(top10)]

x = np.arange(len(sortedData[sortedData.location.isin(top10)].location))  # the label locations
width = 0.35  # the width of the bars

px = 1/plt.rcParams['figure.dpi']  # pixel in inches
fig, ax = plt.subplots(1,figsize=(1000*px, 300*px))
ax.bar(x - width/2, top102days.total_vaccinations, color ='steelblue', width = 0.4,
label="Yesterday")
ax.bar(x + width/2, top10Yest.total_vaccinations, color ='maroon', width = 0.4, label="Two
Days")
# Add labels, a title and grid lines to the plot
ax.set_xlabel( 'Locations' )
ax.set_ylabel( 'Total Vaccinations (x100000)' )
ax.set_xticks(x)
ax.set_xticklabels(sortedData[sortedData.location.isin(top10)].location)

ax.legend()

plt.title( 'Vaccinations Yesterday per Location' )
# Show the figure (here in Jupyter)
plt.show( fig )
plt.close( fig )
```



```
import numpy as np
# creating the bar plot
top10 = ["California","Texas","Florida","New York
State","Illinois","Pennsylvania","Ohio","Georgia","North Carolina","Michigan"]
yData = data[data.date==yesterdaysDate]
sortedData = yData.sort_values(by=['total_vaccinations'])
top10Yest = sortedData[sortedData.location.isin(top10)]

twoData = data[data.date==twoDays]
sortedData = twoData.sort_values(by=['total_vaccinations'])
top102days = sortedData[sortedData.location.isin(top10)]

x = np.arange(len(sortedData[sortedData.location.isin(top10)].location))  # the label locations
width = 0.35  # the width of the bars

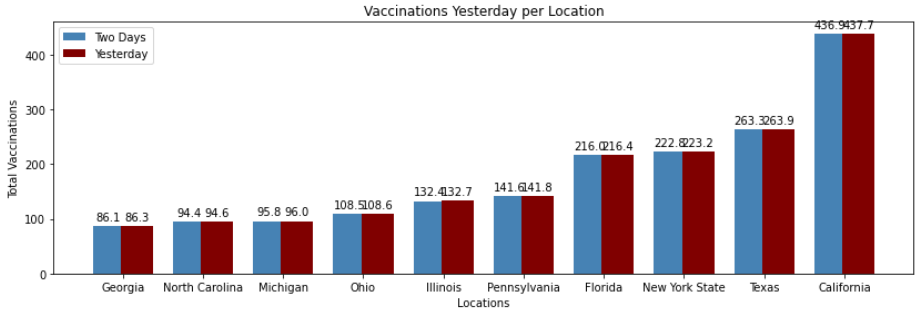
px = 1/plt.rcParams['figure.dpi']  # pixel in inches
fig, ax = plt.subplots(1,figsize=(1000*px, 300*px))
rects1 = ax.bar(x - width/2, top102days.total_vaccinations, color ='steelblue', width = 0.4,
label="Two Days")
rects2 = ax.bar(x + width/2, top10Yest.total_vaccinations, color ='maroon', width = 0.4,
label="Yesterday")
# Add labels, a title and grid lines to the plot
ax.set_xlabel( 'Locations' )
ax.set_ylabel( 'Total Vaccinations' )
ax.set_xticks(x)
ax.set_xticklabels(sortedData[sortedData.location.isin(top10)].location)

ax.legend()

def autolabel(rects):
    """Attach a text label above each bar in *rects*, displaying its height."""
    for rect in rects:
        height = rect.get_height()
        ax.annotate('{:.1f}'.format(height),
                    xy=(rect.get_x() + rect.get_width() / 2, height),
                    xytext=(0, 3),  # 3 points vertical offset
                    textcoords="offset points",
                    ha='center', va='bottom')


autolabel(rects1)
autolabel(rects2)


plt.title( 'Vaccinations Yesterday per Location' )
# Show the figure (here in Jupyter)
plt.show( fig )
plt.close( fig )
```



By Patrick M. Dudas