CHBE 230 - Computational Methods with Python

Tutorial 2

Python Crash Course II

Instructor:
Arman Seyed-Ahmadi

Today's Outline

Today, we are going to learn about:

- Booleans and logic
- Conditional statements and flow control
- Loops and repetitive operations
- Functions and modular programming
- Basic input and output
- Scientific visualization: How to make and customize plots

Booleans and Logic

- A Boolean is a type of data that can only have either one of the two values:

True or **False**

- The result of **logical operations** is always a Boolean.

Comparison operators

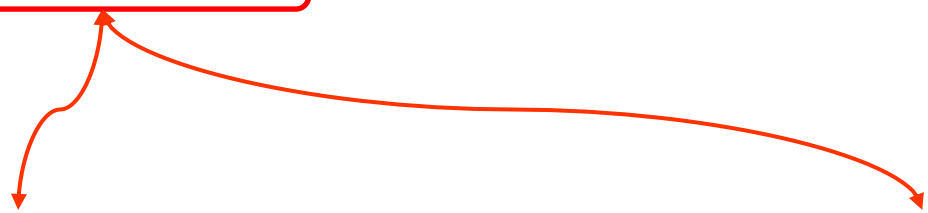
Operator	Description
<	strictly less than
<=	less than or equal
>	strictly greater than
>=	greater than or equal
==	equal
!=	not equal

Logical operators

Operator	Description
A `and` B	returns **True** if both A and B are **True**
A `or` B	returns **True** if either A or B is **True**
`not` A	returns **True** if A is **False**

Booleans and Logic

- The result of **logical operations** is always a Boolean.



Identity operators

Operator	Description
`A is B`	returns **True** if both A and B are the same object
`A is not B`	returns **True** if A and B are not the same object

Membership operators

Operator	Description
`A in B`	returns **True** if object A exists in object B
`A not in B`	returns **True** if object A doesn't exists in object B

Booleans and Logic

- Practice with Booleans: What do you think the result of the following operations would be?

1.	**True** and **True**	**1. True**
2.	**False** and **True**	**2. False**
3.	1 == 1 **and** 2 == 1	**3. False**
4.	"test" == "test“	**4. True**
5.	1 == 1 **or** 2 != 1	**5. True**
6.	**True and** 1 == 1	**6. True**
7.	**False and** 0 != 0	**7. False**
8.	**True or** 1 == 1	**8. True**
9.	"test" == "testing“	**9. False**
10.	1 != 0 **and** 2 == 1	**10. False**
11.	"test" != "testing“	**11. True**
12.	"test" == 1	**12. False**
13.	**not** (**True and False**)	**13. True**
14.	**not** (1 == 1 **and** 0 != 1)	**14. False**
15.	**not** (10 == 1 **or** 1000 == 1000)	**15. False**

Conditional Statements

- The structure of a conditional statement in Python is:

```
if condition1:
    <do something>
elif condition2:
    <do something>
else condition3:
    <do something>
```

This empty space is necessary in Python! It's called an **indent**.

- The conditions are logical expressions, which result in **True** or **False**. If the condition returns **True**, the `<do something>` will be executed. If not, that block is ignored.

This is the condition.

```
a = 10
if a > 10:
    print('Value of a is greater than 10')
else:
    print('Value of a is =< 10')
```

Conditional Statements

```
# create two boolean objects

x = False
y = True

# The validation will be "True" only if
# all the expressions generate the value "True"

if x and y:
    print('Both x and y are True')
else:
    print('x is False or y is False or both x and y are False')
```

A more complex condition made with **and**

```
n = 700

# if n is greater than 500, n is multiplied by 7,
# otherwise n is divided by 7

result = n * 7 if n > 500 else n / 7

print(result)
```

One-line if-statement

Loops for Repetitive Operations

- Computers are very good at doing things repeatedly, remember?
- These repetitive operations are called **loops**.
- There are two types of loops:

    1. **When we want to do a loop while a condition is true,**
    2. **When we want to do a loop for a specified number of times, or over specified items.**

- In programming, these two types of loops are called '**for loops**' and '**while loops**'.

While Loops

```
while condition:
    <do something>
else:
    <do something>
```

- While loops are used when the **number of iterations** is **unknown**.

```
x = 0
s = 0

while (x < 10):
    s = s + x
    x = x + 1
else:
    print('The sum of first 9 integers:', s)
```

```
[]: The sum of first 9 integers: 45
```

For Loops

```
for variable in sequence:
    <do something>
else:
    <do something>
```

- For loops are used when the **number of iterations** or **the objects we iterate over** are **known**.

```
# The list has four elements; indices start at 0 and end at 3
color_list = ["Red", "Blue", "Green", "Black"]

for c in color_list:
    print(c)
```

```
[]: Red
    Blue
    Green
    Black
```

In Python, you can loop over any object directly!

The range() Function

- The `range(start, stop, step)` function in Python can be used to easily generate a sequence of numbers. This is useful in for loops:

```
for i in range(5):
    print(i, end=' ')

[]: 0 1 2 3 4
```

```
for a in range(2, 19, 5):
    print(a)

[]: 2
    7
    12
    17
```

The `range()` function is a built-in Python function and returns a "range" object.

You cannot print a range object directly. However, you can iterate over it!

In Numpy, there is a similar function called `arange()` function.
This function returns a Numpy array object.

Controlling Loops

- **breaking:** The `break` command is used to used to stop the loop and continue executing the code after the loop:

```
while condition1:
    if condition2:
        break <or> continue
else:
    <do something else>
```

```
for variable in sequence:
    if condition:
        break <or> continue
else:
    <do something else>
```

- **continuing:** The `continue` command tells the interpreter to ignore executing the rest of the current step, and to go back to the next loop cycle:

- `break` and `continue` commands work with both **for** and **while** loops.

- When using `break`, the `else` part of the loop **will NOT be executed**.

Controlling Loops

```
# Declaring the tuple

numbers = (1, 2, 3, 4, 5, 6, 7, 8, 9)
num_sum = 0
count = 0

for x in numbers:
    num_sum += x
    count += 1

    if count == 5:
        break

print("Sum of first", count,
"integers is:", num_sum)
```

```
[]: Sum of first 5 integers is: 15
```

```
for x in range(7):
    if (x == 3 or x==6):
        continue
    print(x, end=' ')
```

```
[]: 0 1 2 4 5
```

Functions

- So far, we have used many functions such as the **`print()`** and **`range()`** functions.
- A function takes some arguments and does something for you.
- A function is a piece of code that is **reusable**.
- The code inside a function is only executed when the function is called.
- How can we define functions of our own?

```
def function_name(arg1, arg2, …):
    <do something>
    return <objects>
```

More on Functions

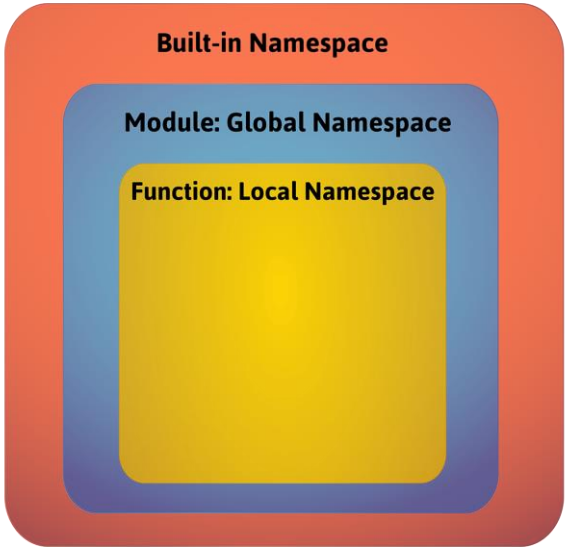
- Functions have their own **local namespace**.
- You cannot access the variables used inside a function from the outside.
- However, you can access variables in the global namespace from within a function.

```
a = 1.43

def func1():
    print(a)

func1()
```

```
[]: 1.43
```

```
def func1():
    a = 1.43

print(a)
```

```
[]: Error!
```



Function Arguments

- Functions take data from the outside world through **arguments** that are supplied to them.

- In Python, there are four types of arguments:

    1. Required (positional) arguments
    2. Default arguments
    3. Keyword arguments
    4. Variable number arguments

Required & Default Arguments

- Required (positional) arguments:

```
def divide(a, b):
    return a / b
```

In this case, even if no value is given to a particular argument, we will not see an error. This is because the function knows the **default** value for that argument.

- Arguments `a` and `b` are **required**. If you don't supply them, you will get an error:

```
divide(1, 2)
```

```
[]: 0.5
```

```
divide(1)
```

```
[]: Error!
```

- We can, however, specify **default** values for our arguments:

```
def divide(a=1, b=1):
    return a / b

divide(2)
```

```
[]: 2.0
```

```
def divide(a=1, b=1):
    return a / b

divide()
```

```
[]: 1.0
```

Keyword Arguments

- In the previous example, if we change the order of `a` and `b`, we will get completely different outputs:

```
divide(1, 2)
```

```
[]: 0.5
```

```
divide(2, 1)
```

```
[]: 2.0
```

- In order to avoid this, it is much better to use **keyword** arguments:

```
divide(a=1, b=2)
```

```
[]: 0.5
```

```
divide(b=2, a=1)
```

```
[]: 0.5
```

- In the above examples, even though we have changed the order of the arguments, the function still does what we think it should do.

Lambda Functions

- A **lambda** function is another method of defining a function instead of the `def` method:

```
variable1 = lambda <arg1, arg2, …> : expression
```

```
magnitude = lambda x, y : np.sqrt(x**2 + y**2)
```

```
magnitude(1,1)
```

```
[]: 1.4142135623730951
```

This is equivalent to:

```
def magnitude(x, y): return np.sqrt(x**2 + y**2)
```

- Lambda functions can have multiple inputs and multiple outputs:

```
div_mul = lambda x, y : (x / y, x * y)
```

- In general, use of the `def` method is recommended.

Basic Input

- We can read input from the user with the following command in Python:

```
variable = input(prompt)
```

- Remember that the output of the **input** function is of **string** type. Before using the output in calculations, we have to convert it to numerical types:

```
b_year = input('Please enter your birth year:')
b_year = int(b_year) # Do the conversion first
print(f'You are {2019 - b_year} years old!')
```

```
[]: Please enter your birth year: 1995
    You are 24 years old!
```

Basic Output

- We've already seen and used the function

```
print(*objects)
```

- This function has puts spaces between the printed objects by default, and also moves to the next line. We can change the default arguments, though:

```
print(*objects, sep=' ', end='\n', file=sys.stdout)
```

```
a, b, c = True, 30, 'Canada'
print(a, b)
print(c)
```

```
[]: True 30
    Canada
```

This is the default behavior (spaces between objects, new line after each print).

```
print(a, b, sep='_', end=' ')
print(c)
```

```
[]: True_30 Canada
```

Escape Characters

- With the print command, you've probably noticed the use of a special combination such as `\n` for the `end` argument. This is called an **escape character**.

```
print(*objects, sep=' ', end='\n', file=sys.stdout)
```

Escape character!

`\ + <newline>`	Backslash and newline ignored
`\\`	Backslash (\)
`\'`	Single quote (')
`\"`	Double quote (")
`\n`	New line
`\t`	Tab space

- We can precede strings with the letter `r` (r for raw text) just as we did with `f` (formatted strings). The `r` tells Python to treat everything as raw text and not look for escape characters.

```
a, b, c = 1.5, 30, 'Canada'
print('The variable value is: \n{a}')
print(f'The variable value is: \n{a}')
print(r'The variable value is: \n{a}')
```

```
[]: The variable value is:
{a}
[]: The variable value is:
1.5
[]: The variable value is: \n{a}
```

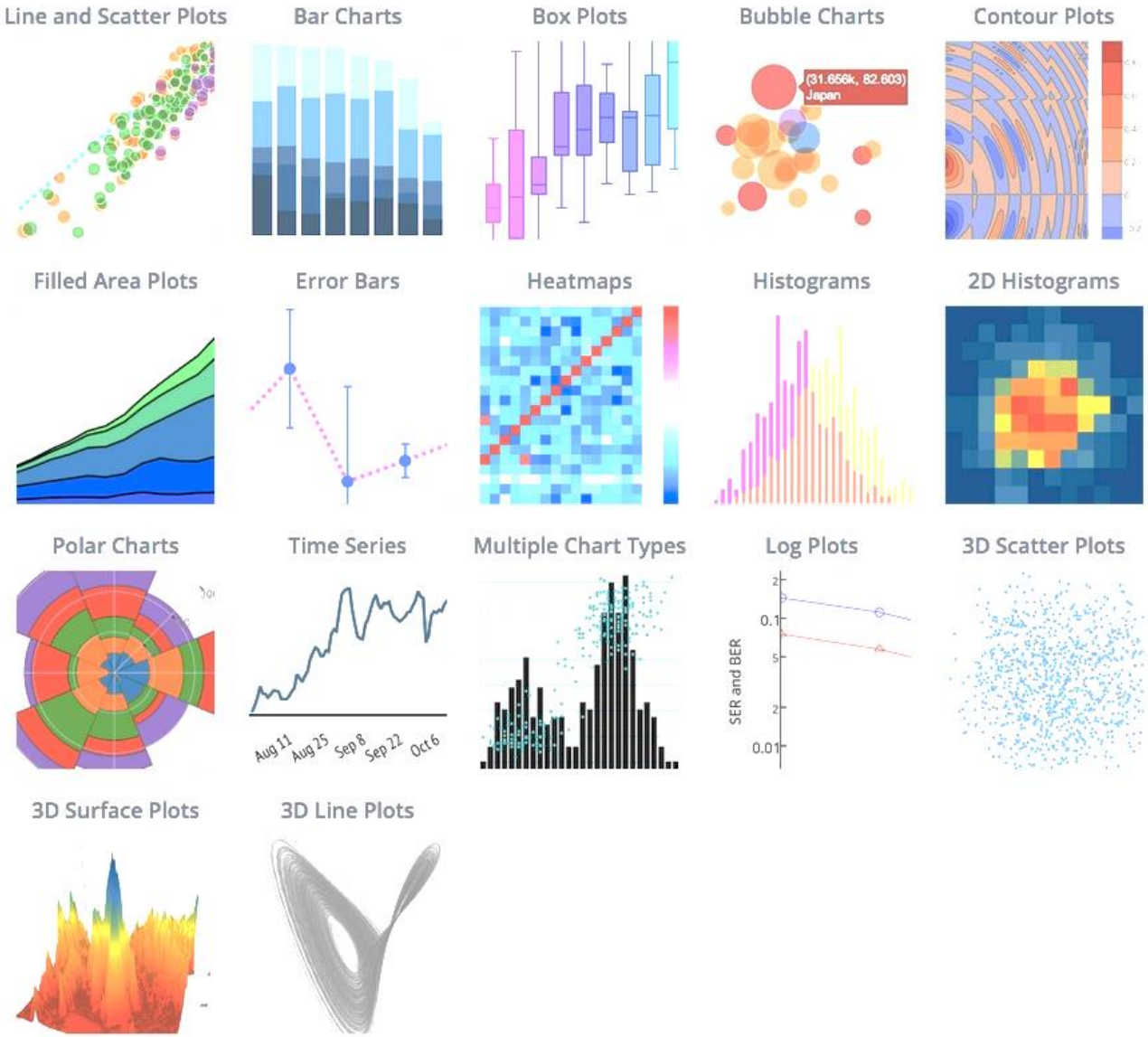
Input & Output - Numpy

- `np.save(filename, array)`: Saves a single *array* into a **binary** .npy format.
- `np.load(filename)`: loads a **binary** .npy file into Python's namespace as a numpy array.

```
import numpy as np
a = np.array([1, 2, 3, 4, 5])
np.save('a_array', a)

b = np.load('a_array.npy')
```

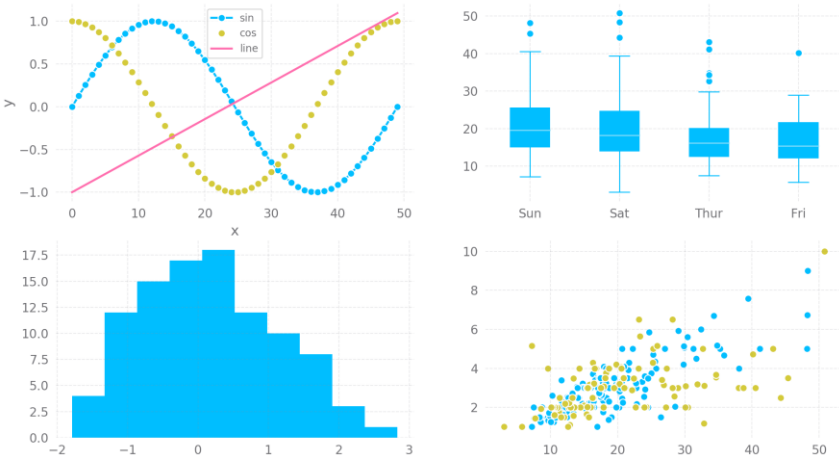
- `np.savetxt(filename, array, delimiter=' ')`: Saves a single *array* into a **text** file. The *delimiter=' '* is used to specify how to separate columns of data.
- `np.loadtxt(filename, delimiter=' ')`: loads a **text** file into Python's namespace as a numpy array. The *delimiter=' '* is used to tell Python how columns of data are separated in the text file.

Scientific Visualization



Plotting in Python

- There are a few visualization packages in Python: **matplotlib**, **plotly**, **bokeh**, etc.
- **matplotlib** is mainly used for high quality 2D graphs, while the advantage of libraries such as **plotly** is mainly in their *interactive* graphs.
- In this course, we will use the powerful **matplotlib** library for our plots.



We can import the matplotlib package with the following line:

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

- **matplotlib** was initially developed to mimic MATLAB's plotting functionality in Python.

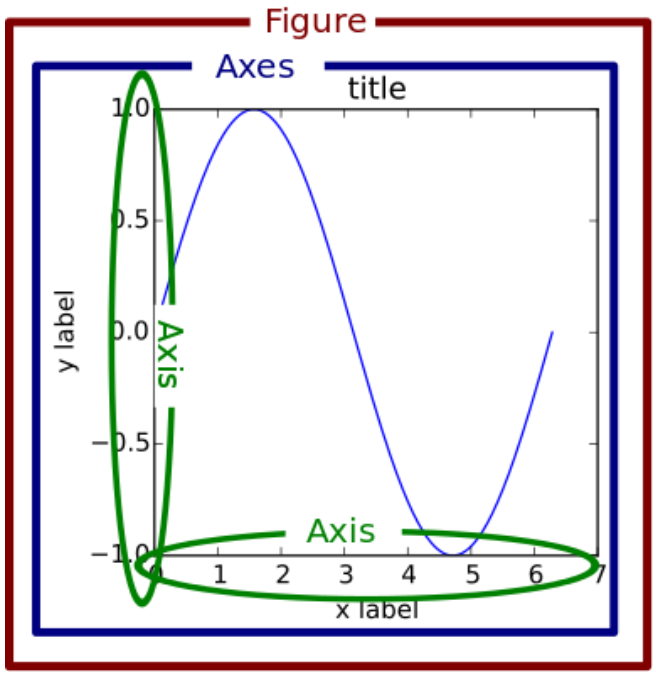
A Bit of Theory

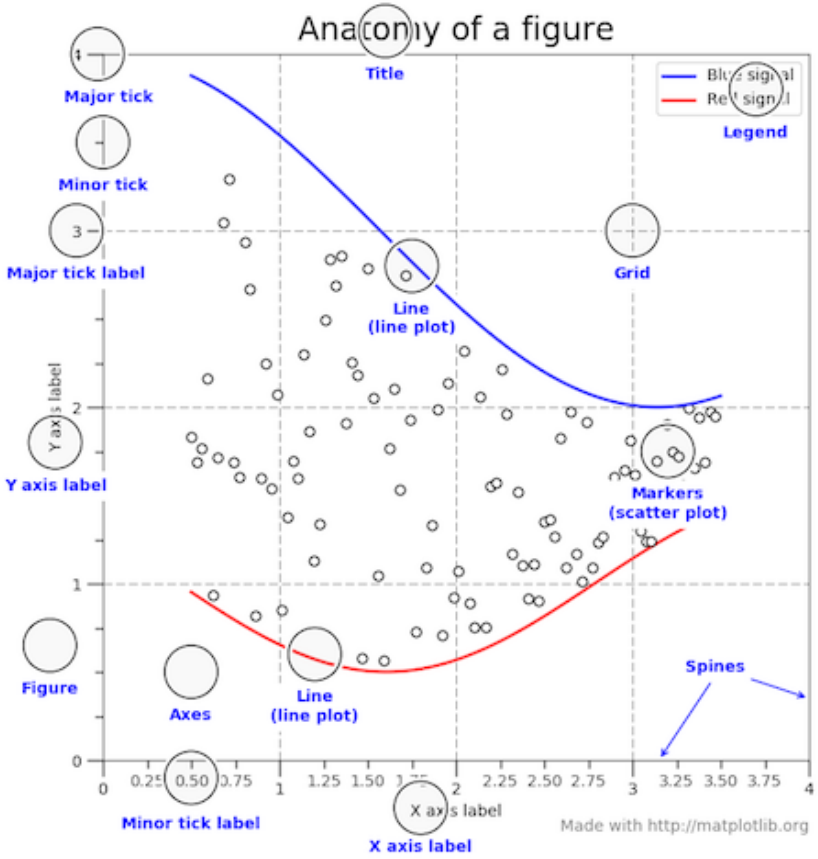
- Like with other libraries in Python, every part of a plot in matplotlib is an object, and has special methods (functions).

- There are two major ways of plotting in Python:

  - The Pythonic (object-oriented) way

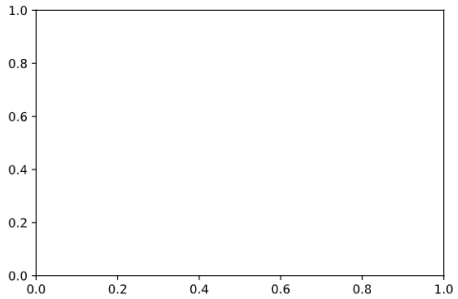
  - The MATLAB way (state-based)





Let's Do Some Plotting!

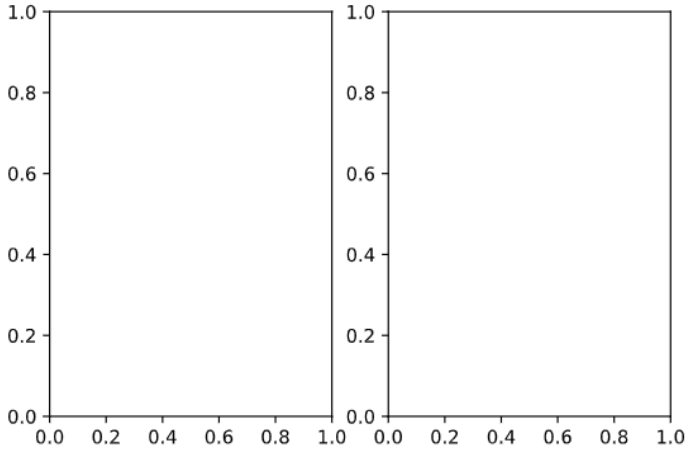
```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
```



- The `plt.subplots()` function creates and returns a single figure object and (sometimes multiple) axes object(s).
- We have easily assigned names to those objects, namely, `fig` and `ax`.

```
fig, ax = plt.subplots(nrows=1, ncols=2)
```

- It's possible to have a grid of plots in just one figure object.
- The `ax` object is now an array which contains 2 elements, one for each plot.

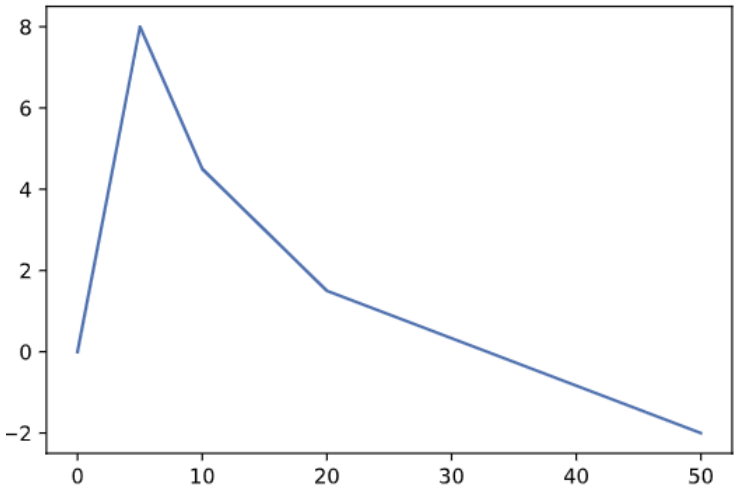


Line Plots

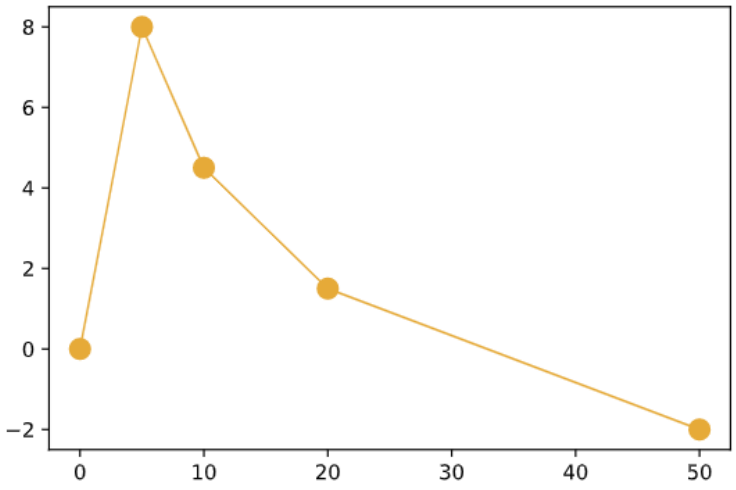
- As soon as the `fig` and `ax` objects are created, we can use the functions available to these objects to add various elements to our plots.

- `ax.plot(x, y, marker, linestyle, color, markersize, markerfacecolor, linewidth, **kwargs)`: This function plots numerical data with a variety of different marker and linestyles.

```
x = [0, 5, 10, 20, 50]
y = [0, 8, 4.5, 1.5, -2]

fig, ax = plt.subplots()
ax.plot(x, y)
```



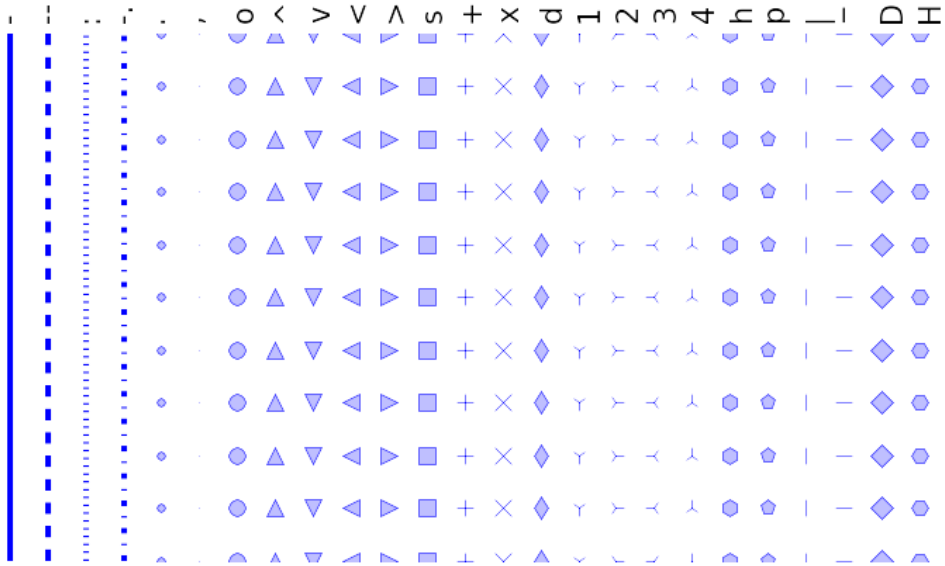

```
ax.plot(x, y, marker='o',
        color='orange',
        markersize=10,
        linewidth=1,
        linestyle='-')
```



Line and Marker Styles & Colors

- You can decide about the line and marker styles in matplotlib according to the this table, and customize your plots to your needs.

- `linestyle='-'`, `markerstyle='-'`



- You can also specify the line and marker colors from the comprehensive table on the left:

- `color='red'`, `facecolor='red'`, `edgecolors='red'`

Scatter Plots

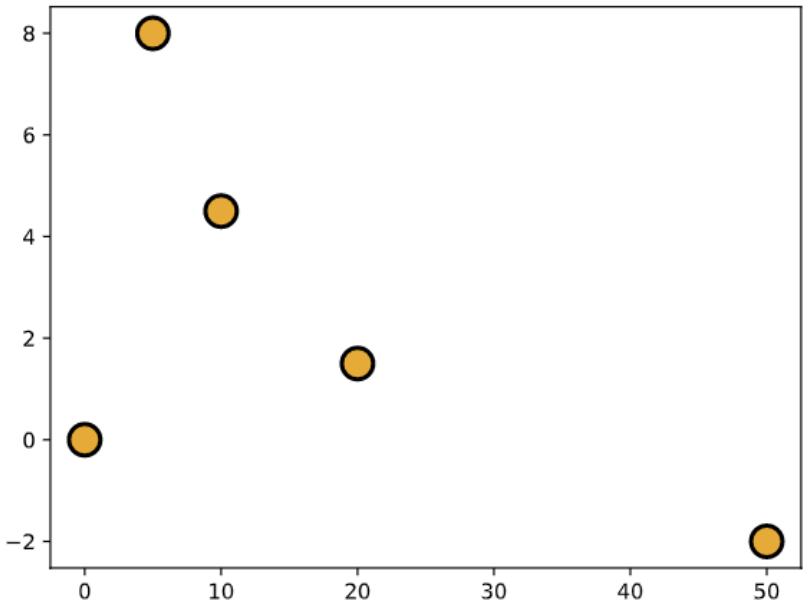
- `ax.scatter(x, y, s, c, linewidth, edgecolors, facecolors, alpha)`: This function draws a scatter plot of pairs `(x, y)` with size `s` and color `c`, and transparency of `0 < alpha < 1`.

- The main difference between `ax.plot()` and `ax.scatter()` is that

  - With `ax.plot`, the data points are usually **ordered** and connected with lines.

  - However, `ax.scatter()` only recognizes pairs of data point. The size and color of each point can be controlled **individually**.

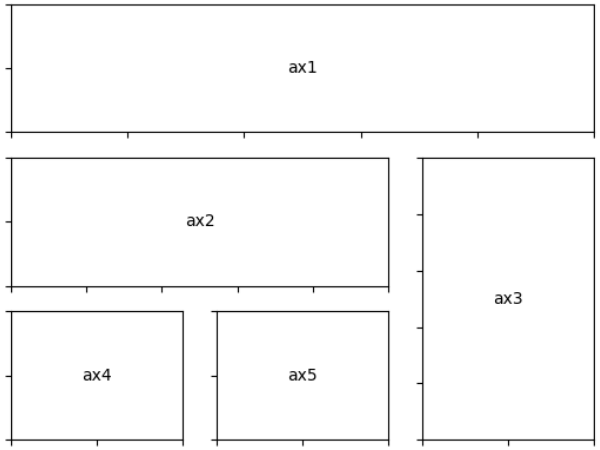
```
x = [5, 0, 10, 20, 50]
y = [8, 0, 4.5, 1.5, -2]

fig, ax = plt.subplots()
ax.scatter(x, y,
           s=15**2,
           edgecolors='black',
           facecolors='orange',
           linewidth=2,
          )
```



Complex Plot Layouts

- We have already seen that we can create **regular** grids of many plots with the following line:

```
fig, ax = plt.subplots(nrows=3, ncols=2)
```



- What if we want grids like this one?

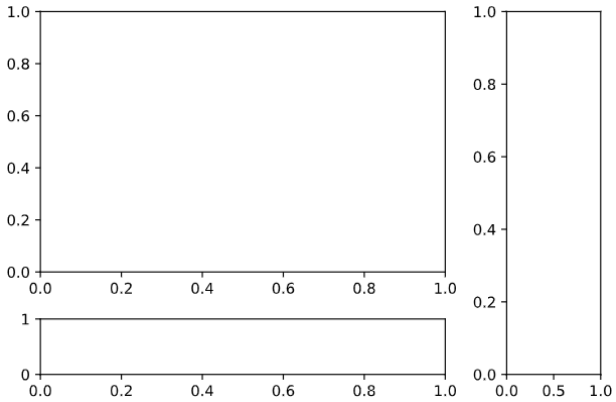
- To create **irregular** grids, we use a slightly different code:

```
fig = plt.figure()

ax1 = plt.subplot2grid(shape=(4, 4), loc=(0, 0), colspan=3, rowspan=3)
ax2 = plt.subplot2grid(shape=(4, 4), loc=(3, 0), colspan=3)
ax3 = plt.subplot2grid(shape=(4, 4), loc=(0, 3), rowspan=4)
```

```
fig.tight_layout()
```

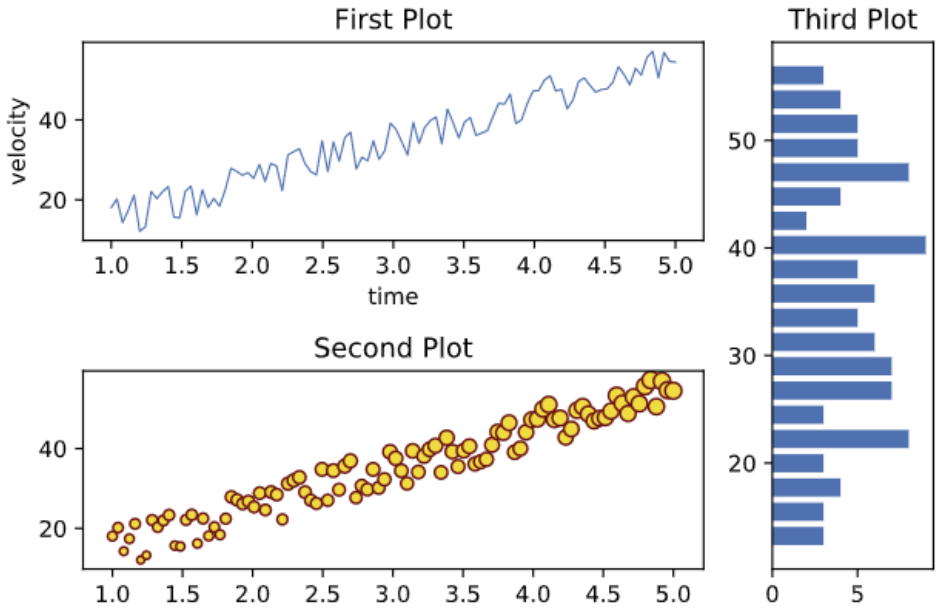
This method automatically adjust the spacings so that the overlapping is minimized.

Since we want to create the axes separately, first we have to create a figure object.



Titles and Axis Labels

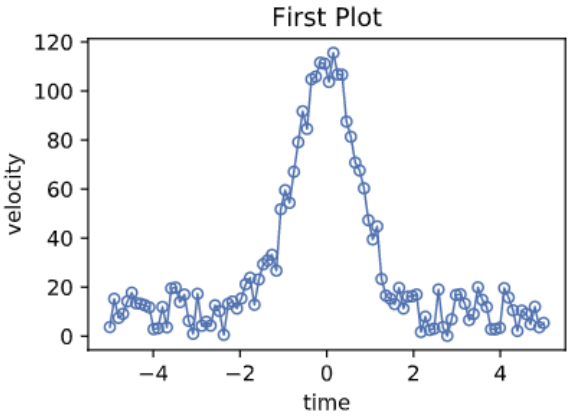
```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(1, 5, 100)
y = 10 * x + 10 * np.random.rand(100)

fig = plt.figure()

ax1 = plt.subplot2grid(shape=(4, 4),
                       loc=(0, 0),
                       rowspan=2,
                       colspan=3)
ax2 = plt.subplot2grid(shape=(4, 4),
                       loc=(2, 0),
                       rowspan=2,
                       colspan=3)
ax3 = plt.subplot2grid(shape=(4, 4),
                       loc=(0, 3),
                       rowspan=4)

ax1.plot(x, y, linewidth=0.75)
ax1.set_title('First Plot')
ax1.set_ylabel('velocity')
ax1.set_xlabel('time')
```

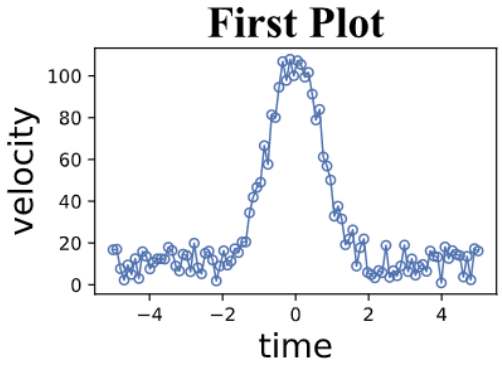



```
ax2.scatter(x, y, s=y,
            edgecolors='maroon',
            facecolor='gold')
ax2.set_title('Second Plot')

ax3.hist(y, bins=20,
         orientation='horizontal',
         rwidth=0.75)
ax3.set_title('Third Plot')

fig.tight_layout()
```

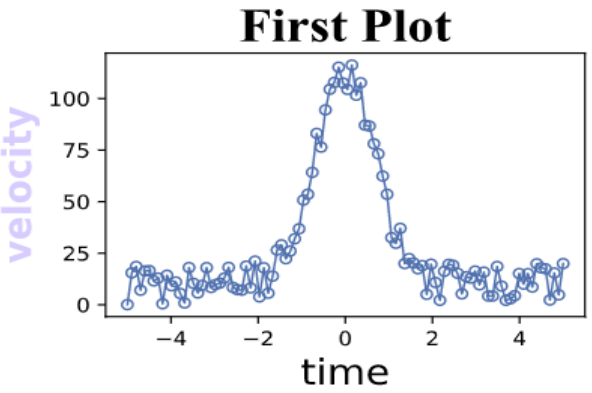
Customizing Plot Text

- Any of the functions for the title and x and y labels accept arguments for specifying different font size, color, weight, transparency, font name, and font family.

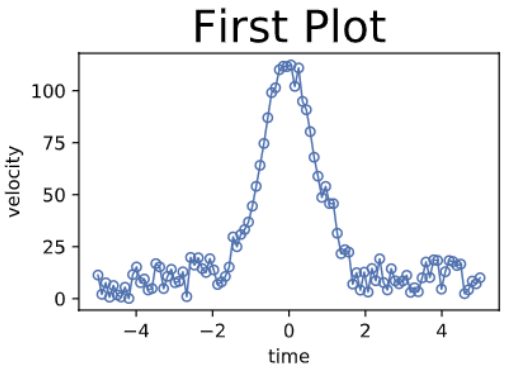





```
ax1.set_title('First Plot',
              fontsize=24,
              fontname='Times New Roman')
ax1.set_ylabel('velocity', fontsize=18)
ax1.set_xlabel('time', fontsize=18)
```




```
ax1.set_ylabel('velocity',
               fontsize=18,
               weight='bold',
               color='blue',
               alpha=0.2)
```




```
ax1.set_title('First Plot',
              fontsize=24)
```

Default Settings and rcParams

- Suppose that you want to have an enlarged font size by default. You will have to put particular keywords for font size in every script.
- There is, however, a workaround this: modifying the default values in **rcParams**.
- **rcParams** is a dictionary object in matplotlib that stores the default values (parameters) for all parts of a plot.
- You can access the default values by typing the following in the console:

```
plt.rcParams
```

- We already know how to change the values of a key in a dictionary, right?

```
plt.rcParams['font.size'] = 18
```

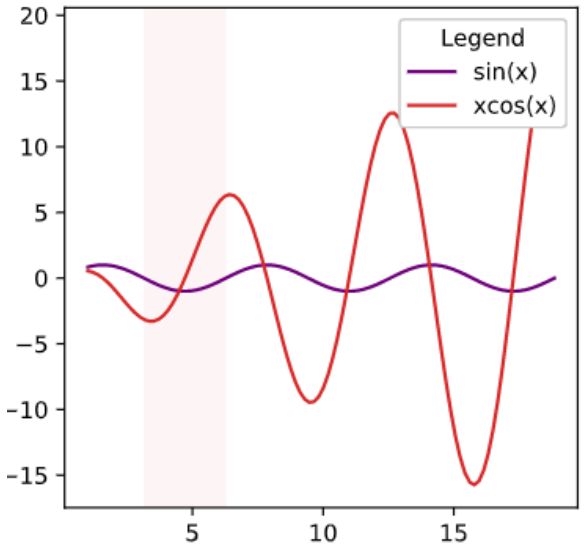
- You can reset to default values with the following line:

```
plt.rcParams.update(plt.rcParamsDefault)
```

```
figure.max_open_warning : 20,
'figure.subplot.bottom': 0.125,
'figure.subplot.hspace': 0.2,
'figure.subplot.left': 0.125,
'figure.subplot.right': 0.9,
'figure.subplot.top': 0.88,
'figure.subplot.wspace': 0.2,
'figure.titlesize': 'large',
'figure.titleweight': 'normal',
'font.cursive': ['Apple Chancery',
                 'Textile',
                 'Zapf Chancery',
                 'Sand',
                 'Script MT',
                 'Felipa',
                 'cursive'],
'font.family': ['sans-serif'],
'font.fantasy': ['Comic Sans MS',
                 'Chicago',
                 'Charcoal',
                 'Impact',
                 'Western',
                 'Humor Sans',
                 'xkcd',
                 'fantasy'],
'font.monospace': ['DejaVu Sans Mono',
                   'Bitstream Vera Sans Mono',
                   'Computer Modern Typewriter',
                   'Andale Mono',
                   'Nimbus Mono L',
                   'Courier New',
                   'Courier',
                   'Fixed',
                   'Terminal',
```

Legends

- When we plot multiple datasets in one single plots, legends can be used to identify different groups of data.
- In matplotlib, we can add the keyword `label='name'` to any plot function.
- `ax.legend(loc, framealpha, frameon, title, **kwargs)`: This function puts a legend corresponding to the labels already defined in the plot functions.



The `ax1.axvspan` function fills the area between two values of x on the plot.

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

x = np.linspace(1, 6 * np.pi, 100)

fig1, ax1 =
plt.subplots(figsize=(4,4))

ax1.plot(x, np.sin(x),
         color='purple',
         linewidth=1.5,
         label='sin(x)',
        )

ax1.plot(x, x * np.cos(x),
         color='red',
         linewidth=1.5,
         label='xcos(x)')

ax1.axvspan(np.pi, 2 * np.pi,
alpha=0.05, facecolor='red')

ax1.legend(loc='upper right',
           framealpha=1,
           title='Legend')
```

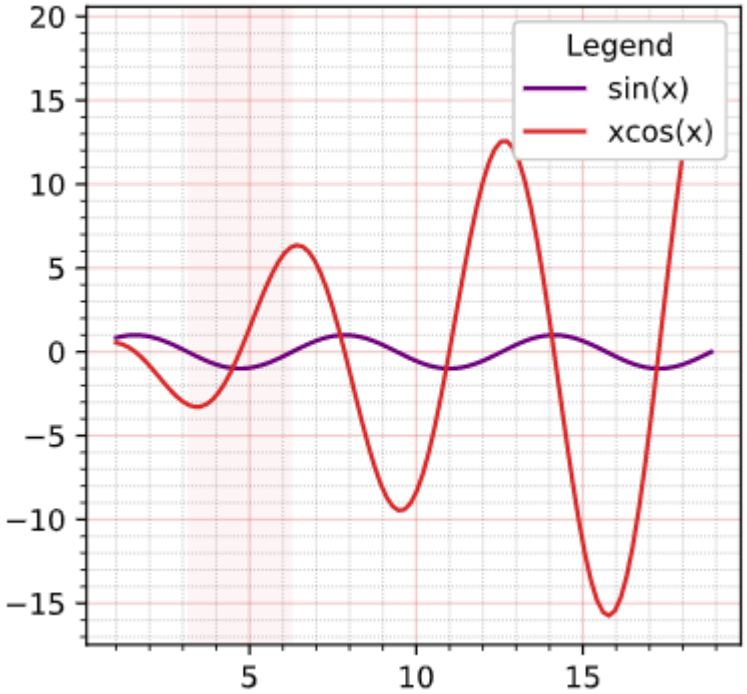
Grids

- `ax.grid(which='major', axis='both', color, linestyle, linewidth, alpha, direction, length, **kwargs)`: This function draws gridlines on the plot with specified linestyle.

```
ax1.grid(which='major',
         linestyle='-',
         linewidth=0.75,
         color='red',
         alpha=0.25)

ax1.grid(which='minor',
         linestyle=':',
         linewidth=0.5,
         color='black',
         alpha=0.25)
```

```
ax1.minorticks_on()
```

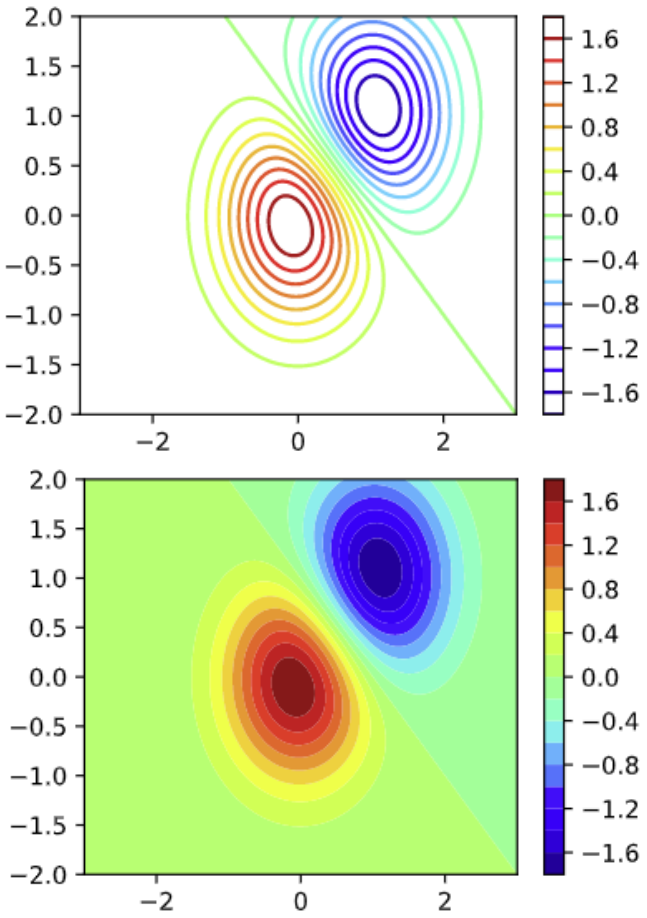
This function should be called to turn on the minor gridlines



Contour Maps

- Contour maps (plots) are very useful in visualizing 3D data.

- `ax1.contour()` creates line contours and `ax1.contourf()` creates surface contours.




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

x = np.linspace(-3.0, 3.0, 100)
y = np.linspace(-2.0, 2.0, 100)
X, Y = np.meshgrid(x, y)

Z1 = np.exp(-X**2 - Y**2)
Z2 = np.exp(-(X - 1)**2 - (Y - 1)**2)
Z = (Z1 - Z2) * 2

fig, ax1 = plt.subplots(figsize=(4,3))
cs = ax1.contourf(X, Y, Z, cmap='jet', levels=20)
cbar = fig.colorbar(cs, ax=ax1)
```

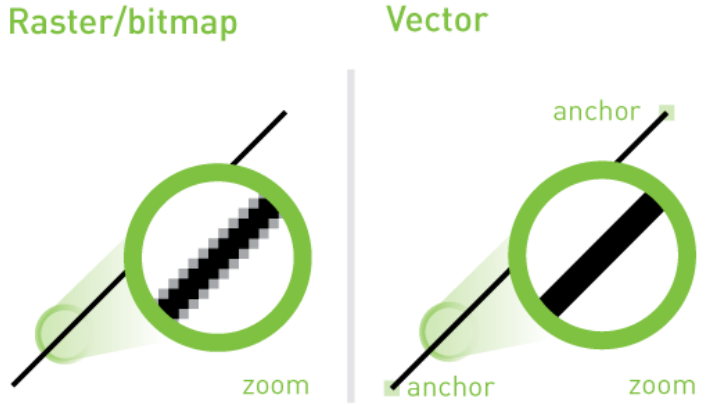
Saving Plots

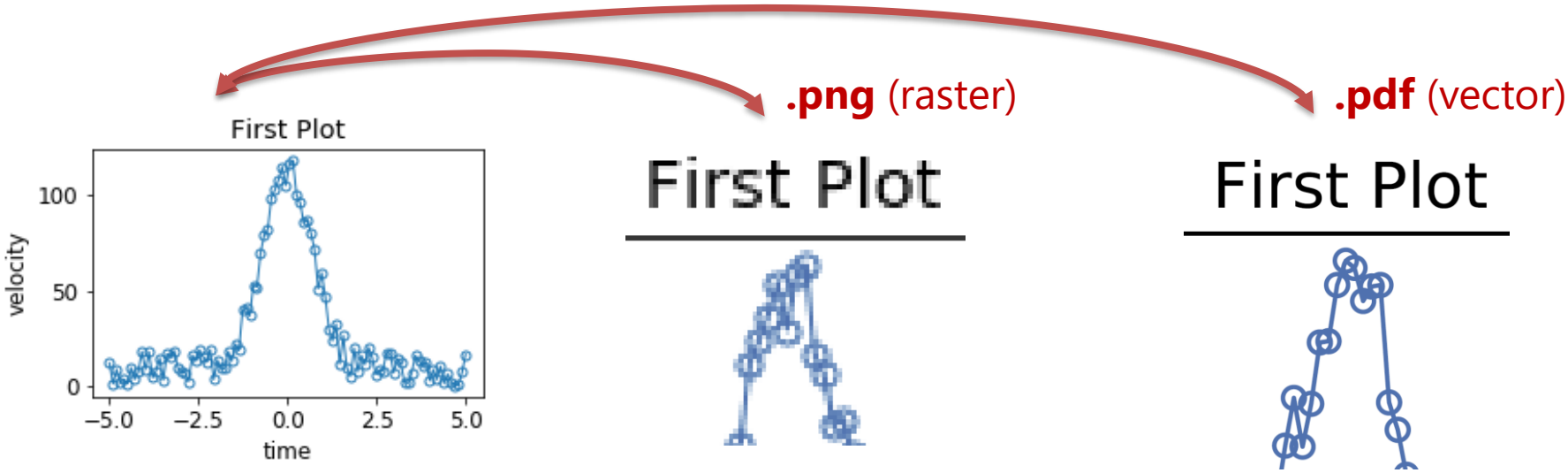
- Saving plots in matplotlib is done with the following command:

```
fig.savefig('myfigure.png')
```

- By specifying the extension of the file, we can easily determine whether the output is a **raster** or a **vector** graphics file.

Raster image = pixel data
Vector image = mathematical paths





References

1. https://www.w3resource.com/python/python-tutorial.php
2. https://www.nayuki.io/page/what-are-binary-and-text-files
3. https://realpython.com/working-with-files-in-python/
4. https://realpython.com/python-csv/
5. https://realpython.com/python-matplotlib-guide/
6. https://scipy-lectures.org/
7. https://www.webcoursesbangkok.com/blog/vector-graphics/
8. https://pynative.com/python-matplotlib-exercise/
9. https://www.math.ubc.ca/~pwalls/math-python/python/logic/