Numpy is the de-facto standard for numerical arrays in Python. It arose as an effort by Travis Oliphant and others to unify the numerical arrays in Python.

Numpy provides specification of byte-sized arrays in Python. For example, below we create an array of three numbers, each of four-bytes long (32 bits at 8 bits per byte) as shown by the `itemsize` property. The first line imports Numpy as `np`, which is the recommended convention. The next line creates an array of 32 bit floating point numbers. The `itemize` property shows the number of bytes per item.

```
>>> import numpy as np # recommended convention
>>> x = np.array([1,2,3],dtype=np.float32)
>>> x
array([ 1.,  2.,  3.], dtype=float32)
>>> x.itemsize
4
```

In addition to providing uniform containers for numbers, Numpy provides a comprehensive set of unary functions (i.e., *ufuncs*) that process arrays element-wise without additional looping semantics. Below, we show how to compute the element-wise sine using Numpy,

```
>>> np.sin(np.array([1,2,3],dtype=np.float32) )
array([ 0.84147096,  0.90929741,  0.14112   ], dtype=float32)
```

This computes the sine of the input array `[1,2,3]`, using Numpy's unary function, `np.sin`. There is another sine function in the built-in `math` module, but the Numpy version is faster because it does not require explicit looping (i.e., using a `for` loop) over each of the elements in the array. That looping happens in the compiled `np.sin` function itself. Otherwise, we would have to do looping explicitly as in the following:

```
>>> from math import sin
>>> [sin(i) for i in [1,2,3]] # list comprehension
[0.8414709848078965, 0.9092974268256817, 0.1411200080598672]
```

Numpy arrays come in many dimensions. For example, the following shows a two-dimensional 2×3 array constructed from two conforming Python lists.

```
>>> x=np.array([ [1,2,3],[4,5,6] ])
>>> x.shape
(2, 3)
```

Note that Numpy is limited to 32 dimensions unless you build it for more.[2] Numpy arrays follow the usual Python slicing rules in multiple dimensions as shown below where the `:` colon character selects all elements along a particular axis.

```
>>> x=np.array([ [1,2,3],[4,5,6] ])
>>> x[:,0] # 0th column
array([1, 4])
>>> x[:,1] # 1st column
array([2, 5])
```

```
>>> x[0,:] # 0th row
array([1, 2, 3])
>>> x[1,:] # 1st row
array([4, 5, 6])
```

You can also select sub-sections of arrays by using slicing as shown below.

```
>>> x=np.array([ [1,2,3],[4,5,6] ])
>>> x
array([[1, 2, 3],
       [4, 5, 6]])
>>> x[:,1:] # all rows, 1st thru last column
array([[2, 3],
       [5, 6]])
>>> x[:,::2] # all rows, every other column
array([[1, 3],
       [4, 6]])
>>> x[:,::-1] # reverse order of columns
array([[3, 2, 1],
       [6, 5, 4]])
```

Numpy Matrices

Matrices in Numpy are similar to Numpy arrays but they can only have two dimensions. They implement row-column matrix multiplication as opposed to element-wise multiplication.

If you have two matrices you want to multiply, you can either create them directly or convert them from Numpy arrays.

For example, the following shows how to create two matrices and multiply them.

```
>>> import numpy as np
>>> A=np.matrix([[1,2,3],[4,5,6],[7,8,9]])
>>> x=np.matrix([[1],[0],[0]])
>>> A*x
matrix([[1],
        [4],
        [7]])
```

This can also be done using arrays as shown below,

```
>>> A=np.array([[1,2,3],[4,5,6],[7,8,9]])
>>> x=np.array([[1],[0],[0]])
>>> A.dot(x)
array([[1],
       [4],
       [7]])
```

Numpy arrays support element-wise multiplication, not row-column multiplication.

You must use Numpy matrices for this kind of multiplication unless use the inner product **np.dot**, which also works in multiple dimensions.

8.2 `len`

`len` is a built-in function that returns the number of characters in a string:

```
>>> fruit = 'banana'
>>> len(fruit)
6
```

To get the last letter of a string, you might be tempted to try something like this:

```
>>> length = len(fruit)
>>> last = fruit[length]
IndexError: string index out of range
```

The reason for the `IndexError` is that there is no letter in `'banana'` with the index 6. Since we started counting at zero, the six letters are numbered 0 to 5. To get the last character, you have to subtract 1 from `length`:

```
>>> last = fruit[length-1]
>>> last
'a'
```

Or you can use negative indices, which count backward from the end of the string. The expression `fruit[-1]` yields the last letter, `fruit[-2]` yields the second to last, and so on.

Or you can use negative indices, which count backward from the end of the string. The expression `fruit[-1]` yields the last letter, `fruit[-2]` yields the second to last, and so on.

Another way to write a traversal is with a `for` loop:

```
for letter in fruit:
    print(letter)
```

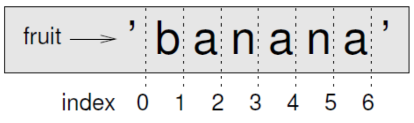


Figure 8.1: Slice indices.

8.4 String slices

A segment of a string is called a **slice**. Selecting a slice is similar to selecting a character:

```
>>> s = 'Monty Python'
>>> s[0:5]
'Monty'
>>> s[6:12]
'Python'
```

The operator `[n:m]` returns the part of the string from the "n-eth" character to the "m-eth" character, including the first but excluding the last. This behavior is counterintuitive, but it might help to imagine the indices pointing *between* the characters, as in Figure 8.1.

If you omit the first index (before the colon), the slice starts at the beginning of the string. If you omit the second index, the slice goes to the end of the string:

```
>>> fruit = 'banana'
>>> fruit[:3]
```

```
'ban'
>>> fruit[3:]
'ana'
```

If the first index is greater than or equal to the second the result is an **empty string**, represented by two quotation marks:

```
>>> fruit = 'banana'
>>> fruit[3:3]
''
```

An empty string contains no characters and has length 0, but other than that, it is the same as any other string.

Continuing this example, what do you think `fruit[:]` means? Try it and see.

8.5 Strings are immutable

It is tempting to use the `[]` operator on the left side of an assignment, with the intention of changing a character in a string. For example:

```
>>> greeting = 'Hello, world!'
>>> greeting[0] = 'J'
TypeError: 'str' object does not support item assignment
```

The "object" in this case is the string and the "item" is the character you tried to assign. For now, an object is the same thing as a value, but we will refine that definition later (Section 10.10).

The reason for the error is that strings are **immutable**, which means you can't change an existing string. The best you can do is create a new string that is a variation on the original:

```
>>> greeting = 'Hello, world!'
>>> new_greeting = 'J' + greeting[1:]
>>> new_greeting
'Jello, world!'
```

This example concatenates a new first letter onto a slice of `greeting`. It has no effect on the original string.

```
word = 'banana'
new_word = word.upper()
```

```
word = 'banana'
index = word.find('a')
index
```

In this example, we invoke `find` on `word` and pass the letter we are looking for as a parameter.

Actually, the `find` method is more general than our function; it can find substrings, not just characters:

```
>>> word.find('na')
2
```

By default, `find` starts at the beginning of the string, but it can take a second argument, the index where it should start:

```
>>> word.find('na', 3)
4
```

This is an example of an **optional argument**; `find` can also take a third argument, the index where it should stop:

```
>>> name = 'bob'
>>> name.find('b', 1, 2)
-1
```

This search fails because `b` does not appear in the index range from `1` to `2`, not including `2`. Searching up to, but not including, the second index makes `find` consistent with the slice operator.

8.9 The `in` operator

The word `in` is a boolean operator that takes two strings and returns `True` if the first appears as a substring in the second:

```
>>> 'a' in 'banana'
True
>>> 'seed' in 'banana'
False
```

For example, the following function prints all the letters from `word1` that also appear in `word2`:

```
def in_both(word1, word2):
    for letter in word1:
        if letter in word2:
            print(letter)
```

With well-chosen variable names, Python sometimes reads like English. You could read this loop, "for (each) letter in (the first) word, if (the) letter (appears) in (the second) word, print (the) letter."

Here's what you get if you compare apples and oranges:

```
>>> in_both('apples', 'oranges')
a
e
s
```

8.10 String comparison

The relational operators work on strings. To see if two strings are equal:

```
if word == 'banana':
    print('All right, bananas.')
```

Other relational operations are useful for putting words in alphabetical order:

```
if word < 'banana':
    print('Your word, ' + word + ', comes before banana.')
elif word > 'banana':
    print('Your word, ' + word + ', comes after banana.')
else:
    print('All right, bananas.')
```

Python does not handle uppercase and lowercase letters the same way people do. All the uppercase letters come before all the lowercase letters, so:

```
Your word, Pineapple, comes before banana.
```

A common way to address this problem is to convert strings to a standard format, such as all lowercase, before performing the comparison. Keep that in mind in case you have to defend yourself against a man armed with a Pineapple.

Exercises

1) Write a program that reads words.txt and prints only the words with more than 20 characters (not counting whitespace).

2) Write a function called has_no_e that returns True if the given word doesn't have the letter "e" in it. Write a program that reads words.txt and prints only the words that have no "e". Compute the percentage of words in the list that have no "e".

3) Write a function named avoids that takes a word and a string of forbidden letters, and that returns True if the word doesn't use any of the forbidden letters.
Write a program that prompts the user to enter a string of forbidden letters and then prints the number of words that don't contain any of them. Can you find a combination of 5 forbidden letters that excludes the smallest number of words?

3.1.1 Create arrays

Create ndarrays from lists. note: every element must be the same type (will be converted if possible)

```
data1 = [1, 2, 3, 4, 5]           # list
arr1 = np.array(data1)            # 1d array
data2 = [range(1, 5), range(5, 9)] # list of lists
arr2 = np.array(data2)            # 2d array
arr2.tolist()                     # convert array back to list
```

create special arrays

```
np.zeros(10)
np.zeros((3, 6))
np.ones(10)
np.linspace(0, 1, 5)            # 0 to 1 (inclusive) with 5 points
np.logspace(0, 3, 4)            # 10^0 to 10^3 (inclusive) with 4 points
```

arange is like range, except it returns an array (not a list)

```
int_array = np.arange(5)
float_array = int_array.astype(float)
```

Return Value and Parameters of `np.arange()`

NumPy `arange()` is one of the array creation routines based on numerical ranges. It creates an instance of `ndarray` with *evenly spaced values* and returns the reference to it.

You can define the interval of the values contained in an array, space between them, and their type with four parameters of `arange()`:

Python

```
numpy.arange([start, ]stop, [step, ], dtype=None) -> numpy.ndarray
```

The first three parameters determine the range of the values, while the fourth specifies the type of the elements:

1. **`start`** is the number (integer or decimal) that defines the first value in the array.
2. **`stop`** is the number that defines the end of the array and isn't included in the array.
3. **`step`** is the number that defines the spacing (difference) between each two consecutive values in the array and defaults to `1`.
4. **`dtype`** is the type of the elements of the output array and defaults to `None`.

`step` can't be zero. Otherwise, you'll get a `ZeroDivisionError`. You can't move away anywhere from `start` if the increment or decrement is `0`.

If `dtype` is omitted, `arange()` will try to deduce the type of the array elements from the types of `start`, `stop`, and `step`.

Range Arguments of `np.arange()`

The arguments of NumPy `arange()` that define the values contained in the array correspond to the numeric parameters `start`, `stop`, and `step`. You have to pass *at least one* of them.

The following examples will show you how `arange()` behaves depending on the number of arguments and their values.

Providing All Range Arguments

When working with NumPy routines, you have to import NumPy first:

Python >>>

```python
>>> import numpy as np
```

Now, you have NumPy imported and you're ready to apply `arange()`.

Let’s see a first example of how to use NumPy `arange()`:

Python >>>

```
>>> np.arange(start=1, stop=10, step=3)
array([1, 4, 7])
```

In this example, `start` is `1`. Therefore, the first element of the obtained array is `1`. `step` is `3`, which is why your second value is 1+3, that is `4`, while the third value in the array is 4+3, which equals `7`.

Following this pattern, the next value would be `10` (7+3), but counting must be ended *before* `stop` is reached, so this one is not included.

You can pass `start`, `stop`, and `step` as positional arguments as well:

Python >>>

```
>>> np.arange(1, 10, 3)
array([1, 4, 7])
```

This code sample is equivalent to, but more concise than the previous one.

The value of `stop` is not included in an array. That's why you can obtain identical results with different `stop` values:

Python >>>

```
>>> np.arange(1, 8, 3)
array([1, 4, 7])
```

This code sample returns the array with the same values as the previous two. You can get the same result with any value of `stop` strictly greater than `7` and less than or equal to `10`.

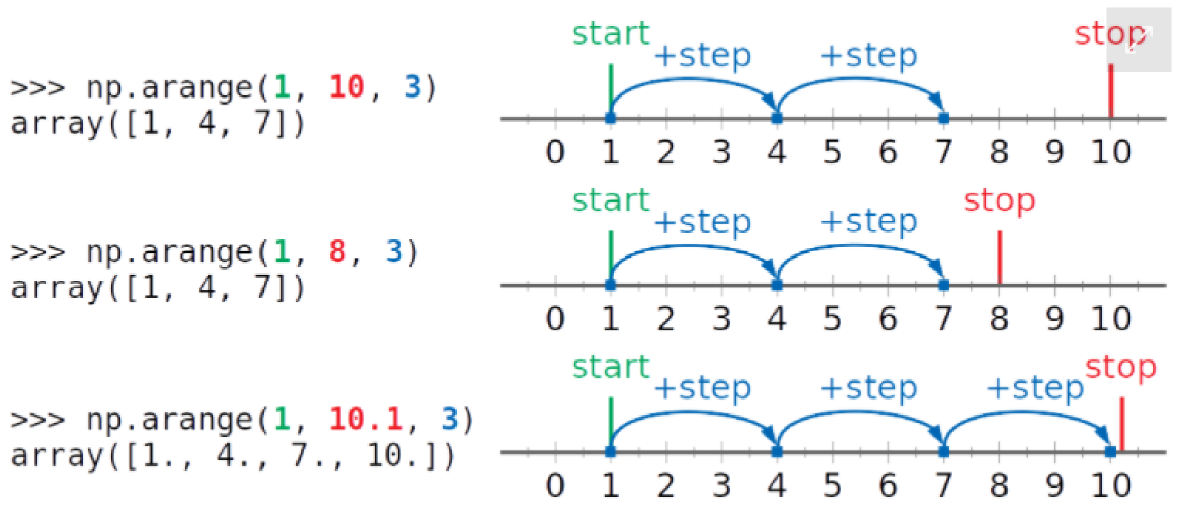
However, if you make `stop` greater than `10`, then counting is going to end after `10` is reached:

Python >>>

```
>>> np.arange(1, 10.1, 3)
array([ 1.,  4.,  7., 10.])
```

In this case, you get the array with four elements that includes `10`.

You can see the graphical representations of these three examples in the figure below:



```
>>> np.arange(1, 10, 3)
array([1, 4, 7])
```

```
>>> np.arange(1, 8, 3)
array([1, 4, 7])
```

```
>>> np.arange(1, 10.1, 3)
array([1., 4., 7., 10.])
```

`start` is shown in green, `stop` in red, while `step` and the values contained in the arrays are blue.

As you can see from the figure above, the first two examples have three values (`1`, `4`, and `7`) counted. They don't allow `10` to be included. In the third example, `stop` is larger than `10`, and it is contained in the resulting array.

Providing Two Range Arguments

You can omit `step`. In this case, `arange()` uses its default value of `1`. The following two statements are equivalent:

Python >>>

```
>>> np.arange(start=1, stop=10, step=1)
array([1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.arange(start=1, stop=10)
array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

The second statement is shorter. `step`, which defaults to `1`, is what's usually intuitively expected.

Using `arange()` with the increment `1` is a very common case in practice. Again, you can write the previous example more concisely with the positional arguments `start` and `stop`:

Python >>>

```
>>> np.arange(1, 10)
array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

This is an intuitive and concise way to invoke `arange()`. Using the keyword arguments in this example doesn't really improve readability.

Providing Negative Arguments

If you provide negative values for `start` or both `start` and `stop`, and have a positive `step`, then `arange()` will work the same way as with all positive arguments:

Python >>>

```
>>> np.arange(-5, -1)
array([-5, -4, -3, -2])
>>> np.arange(-8, -2, 2)
array([-8, -6, -4])
>>> np.arange(-5, 6, 4)
array([-5, -1,  3])
```

This behavior is fully consistent with the previous examples. The counting begins with the value of `start`, incrementing repeatedly by `step`, and ending before `stop` is reached.

Counting Backwards

Sometimes you'll want an array with the values decrementing from left to right. In such cases, you can use `arange()` with a negative value for `step`, and with a `start` greater than `stop`:

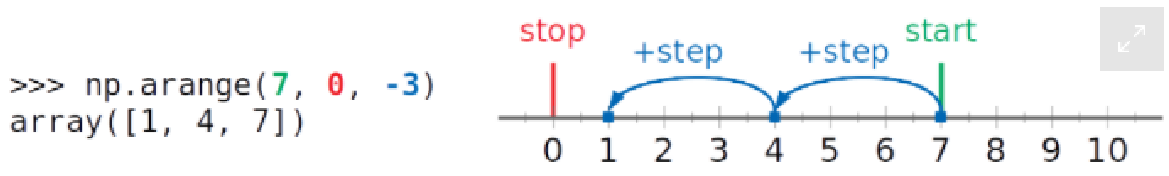
Python >>>

```
>>> np.arange(5, 1, -1)
array([5, 4, 3, 2])
>>> np.arange(7, 0, -3)
array([7, 4, 1])
```

In this example, notice the following pattern: the obtained array starts with the value of the first argument and decrements for `step` towards the value of the second argument.

In the last statement, `start` is `7`, and the resulting array begins with this value. `step` is `-3` so the second value is 7+(−3), that is `4`. The third value is 4+(−3), or `1`. Counting stops here since `stop` (`0`) is reached before the next value (`-2`).

You can see the graphical representations of this example in the figure below:



```
>>> np.arange(7, 0, -3)
array([1, 4, 7])
```

Again, `start` is shown in green, `stop` in red, while `step` and the values contained in the array are blue.

This time, the arrows show the direction from right to left. That's because `start` is greater than `stop`, `step` is negative, and you're basically counting backwards.

The previous example produces the same result as the following:

```python
>>> np.arange(1, 8, 3)[::-1]
array([7, 4, 1])
>>> np.flip(np.arange(1, 8, 3))
array([7, 4, 1])
```

However, the variant with the negative value of `step` is more elegant and concise.

Getting Empty Arrays

There are several edge cases where you can obtain empty NumPy arrays with `arange()`. These are regular instances of `numpy.ndarray` without any elements.

If you provide equal values for `start` and `stop`, then you'll get an empty array:

```
Python                                                    >>>

>>> np.arange(2, 2)
array([], dtype=int64)
```

This is because counting ends before the value of `stop` is reached. Since the value of `start` is equal to `stop`, it can't be reached and included in the resulting array as well.

What is Matplotlib?

Matplotlib is a low level graph plotting library in python that serves as a visualization utility.

Matplotlib was created by John D. Hunter.

Matplotlib is open source and we can use it freely.

Matplotlib is mostly written in python, a few segments are written in C, Objective-C and Javascript for Platform compatibility.

Installation of Matplotlib

If you have Python and PIP already installed on a system, then installation of Matplotlib is very easy.

Install it using this command:

```
C:\Users\Your Name>pip install matplotlib
```

If this command fails, then use a python distribution that already has Matplotlib installed, like Anaconda, Spyder etc.

Import Matplotlib

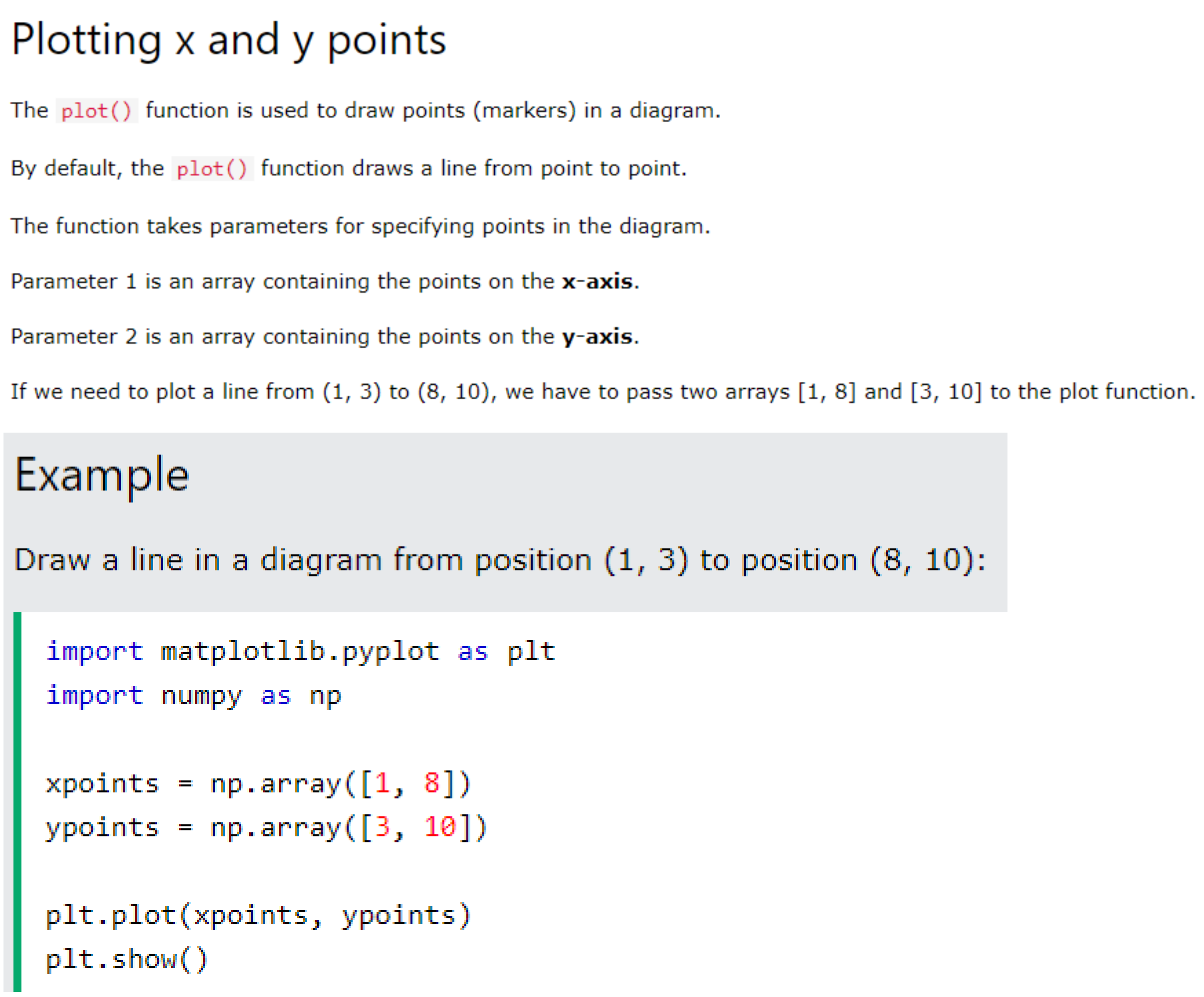
Once Matplotlib is installed, import it in your applications by adding the `import module` statement:

```
import matplotlib
```

```
#TO PRINT MATPLOTLIB VERSION INSTALLED ON YOUR SYSTEM
import matplotlib
print(matplotlib.__version__)
```

Draw a line in a diagram from position (0,0) to position (6,250):

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

xpoints = np.array([0, 6])
ypoints = np.array([0, 250])

plt.plot(xpoints, ypoints)
plt.show()
```



Plotting x and y points

The `plot()` function is used to draw points (markers) in a diagram.

By default, the `plot()` function draws a line from point to point.

The function takes parameters for specifying points in the diagram.

Parameter 1 is an array containing the points on the **x-axis**.

Parameter 2 is an array containing the points on the **y-axis**.

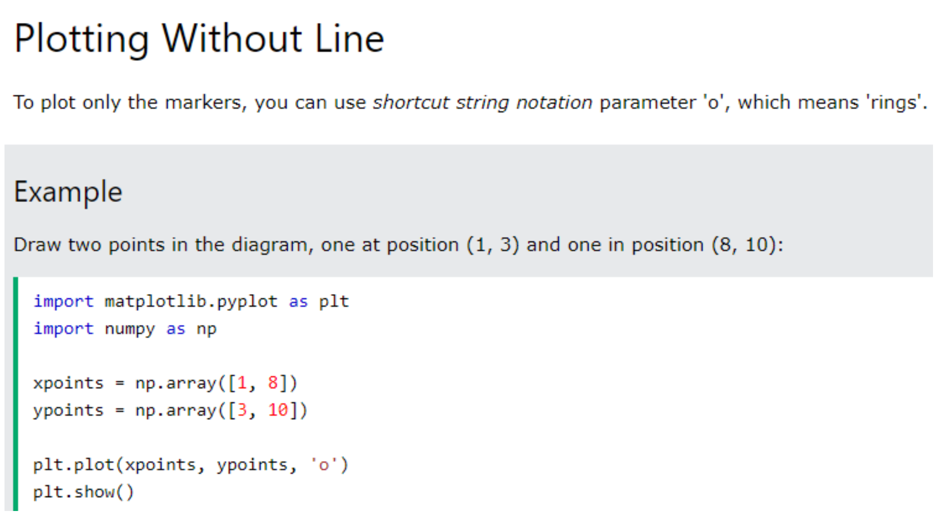
If we need to plot a line from (1, 3) to (8, 10), we have to pass two arrays [1, 8] and [3, 10] to the plot function.

Example

Draw a line in a diagram from position (1, 3) to position (8, 10):

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

xpoints = np.array([1, 8])
ypoints = np.array([3, 10])

plt.plot(xpoints, ypoints)
plt.show()
```

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

xpoints = np.array([1, 8])
ypoints = np.array([3, 10])

plt.plot(xpoints, ypoints)
plt.show()
```



Plotting Without Line

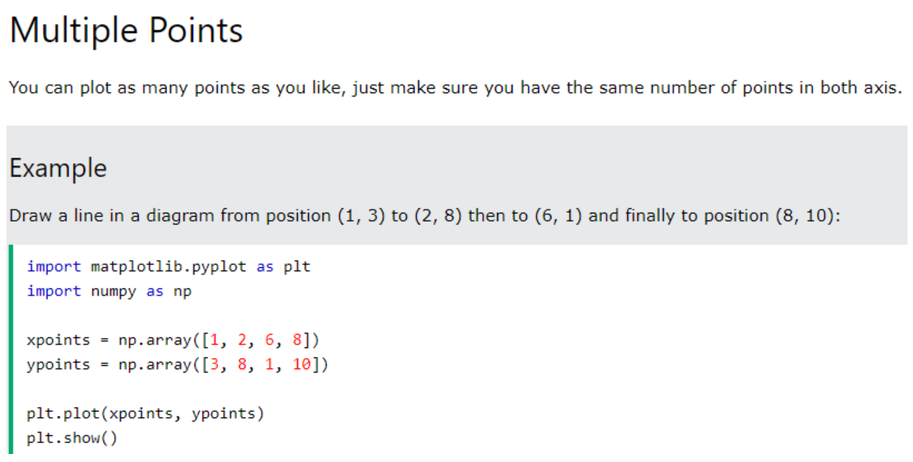
To plot only the markers, you can use *shortcut string notation* parameter 'o', which means 'rings'.

Example

Draw two points in the diagram, one at position (1, 3) and one in position (8, 10):

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

xpoints = np.array([1, 8])
ypoints = np.array([3, 10])

plt.plot(xpoints, ypoints, 'o')
plt.show()
```

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

xpoints = np.array([1, 8])
ypoints = np.array([3, 10])

plt.plot(xpoints, ypoints, 'o')
plt.show()
```



Multiple Points

You can plot as many points as you like, just make sure you have the same number of points in both axis.

Example

Draw a line in a diagram from position (1, 3) to (2, 8) then to (6, 1) and finally to position (8, 10):

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

xpoints = np.array([1, 2, 6, 8])
ypoints = np.array([3, 8, 1, 10])

plt.plot(xpoints, ypoints)
plt.show()
```

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

xpoints = np.array([1, 2, 6, 8])
ypoints = np.array([3, 8, 1, 10])

plt.plot(xpoints, ypoints)
plt.show()
```

Default X-Points

If we do not specify the points in the x-axis, they will get the default values 0, 1, 2, 3, (etc. depending on the length of the y-points.

So, if we take the same example as above, and leave out the x-points, the diagram will look like this:

Example

Plotting without x-points:

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

ypoints = np.array([3, 8, 1, 10, 5, 7])

plt.plot(ypoints)
plt.show()
```

The **x-points** in the example above is [0, 1, 2, 3, 4, 5].

Markers

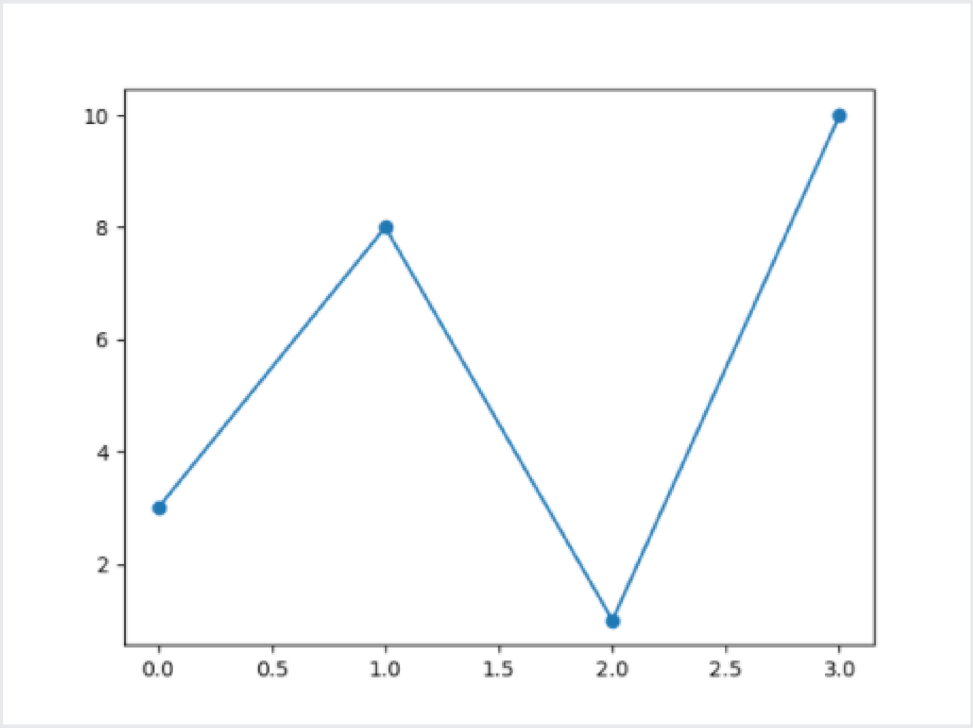
You can use the keyword argument `marker` to emphasize each point with a specified marker:

Example

Mark each point with a circle:

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

ypoints = np.array([3, 8, 1, 10])

plt.plot(ypoints, marker = 'o')
plt.show()
```

Result:



Marker Reference

You can choose any of these markers:

Marker	Description
'o'	Circle
'*'	Star
'.'	Point
','	Pixel
'x'	X
'X'	X (filled)
'+'	Plus
'P'	Plus (filled)
's'	Square
'D'	Diamond
'd'	Diamond (thin)
'p'	Pentagon
'H'	Hexagon
'h'	Hexagon

'v'	Triangle Down
'^'	Triangle Up
'<'	Triangle Left
'>'	Triangle Right
'1'	Tri Down
'2'	Tri Up
'3'	Tri Left
'4'	Tri Right
'|'	Vline
'_'	Hline

Format Strings `fmt`

You can use also use the *shortcut string notation* parameter to specify the marker.

This parameter is also called `fmt`, and is written with this syntax:

marker|line|color

Example

Mark each point with a circle:

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

ypoints = np.array([3, 8, 1, 10])

plt.plot(ypoints, 'o:r')
plt.show()
```

Line Reference

Line Syntax	Description
'-'	Solid line
':'	Dotted line
'--'	Dashed line
'-.'	Dashed/dotted line

Color Reference

Color Syntax	Description
'r'	Red
'g'	Green
'b'	Blue
'c'	Cyan
'm'	Magenta
'y'	Yellow
'k'	Black
'w'	White

Marker Size

You can use the keyword argument `markersize` or the shorter version, `ms` to set the size of the markers:

Example

Set the size of the markers to 20:

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

ypoints = np.array([3, 8, 1, 10])

plt.plot(ypoints, marker = 'o', ms = 20)
plt.show()
```

https://www.w3schools.com/python/matplotlib_markers.asp

https://www.w3schools.com/python/matplotlib_line.asp

Use a dotted line:

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

ypoints = np.array([3, 8, 1, 10])

plt.plot(ypoints, linestyle = 'dotted')
plt.show()


plt.plot(ypoints, linestyle = 'dashed')
```

Set the line color to red:

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

ypoints = np.array([3, 8, 1, 10])

plt.plot(ypoints, color = 'r')
plt.show()
```

Plot with a 20.5pt wide line:

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

ypoints = np.array([3, 8, 1, 10])

plt.plot(ypoints, linewidth = '20.5')
plt.show()
```

Draw two lines by specifying a `plt.plot()` function for each line:

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

y1 = np.array([3, 8, 1, 10])
y2 = np.array([6, 2, 7, 11])

plt.plot(y1)
plt.plot(y2)

plt.show()
```

Draw two lines by specifiyng the x- and y-point values for both lines:

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

x1 = np.array([0, 1, 2, 3])
y1 = np.array([3, 8, 1, 10])
x2 = np.array([0, 1, 2, 3])
y2 = np.array([6, 2, 7, 11])

plt.plot(x1, y1, x2, y2)
plt.show()
```

Add labels to the x- and y-axis:

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

x = np.array([80, 85, 90, 95, 100, 105, 110, 115, 120, 125])
y = np.array([240, 250, 260, 270, 280, 290, 300, 310, 320, 330])

plt.plot(x, y)
```

```python
plt.xlabel("Average Pulse")
plt.ylabel("Calorie Burnage")

plt.show()
```

Add a plot title and labels for the x- and y-axis:

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

x = np.array([80, 85, 90, 95, 100, 105, 110, 115, 120, 125])
y = np.array([240, 250, 260, 270, 280, 290, 300, 310, 320, 330])

plt.plot(x, y)

plt.title("Sports Watch Data")
plt.xlabel("Average Pulse")
plt.ylabel("Calorie Burnage")

plt.show()
```

Set font properties for the title and labels:

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

x = np.array([80, 85, 90, 95, 100, 105, 110, 115, 120, 125])
y = np.array([240, 250, 260, 270, 280, 290, 300, 310, 320, 330])

font1 = {'family':'serif','color':'blue','size':20}
font2 = {'family':'serif','color':'darkred','size':15}

plt.title("Sports Watch Data", fontdict = font1)
plt.xlabel("Average Pulse", fontdict = font2)
plt.ylabel("Calorie Burnage", fontdict = font2)

plt.plot(x, y)
plt.show()
```

Position the Title

You can use the `loc` parameter in `title()` to position the title.

Legal values are: 'left', 'right', and 'center'. Default value is 'center'.

Position the title to the left:

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

x = np.array([80, 85, 90, 95, 100, 105, 110, 115, 120, 125])
y = np.array([240, 250, 260, 270, 280, 290, 300, 310, 320, 330])

plt.title("Sports Watch Data", loc = 'left')
plt.xlabel("Average Pulse")
plt.ylabel("Calorie Burnage")

plt.plot(x, y)
plt.show()
```

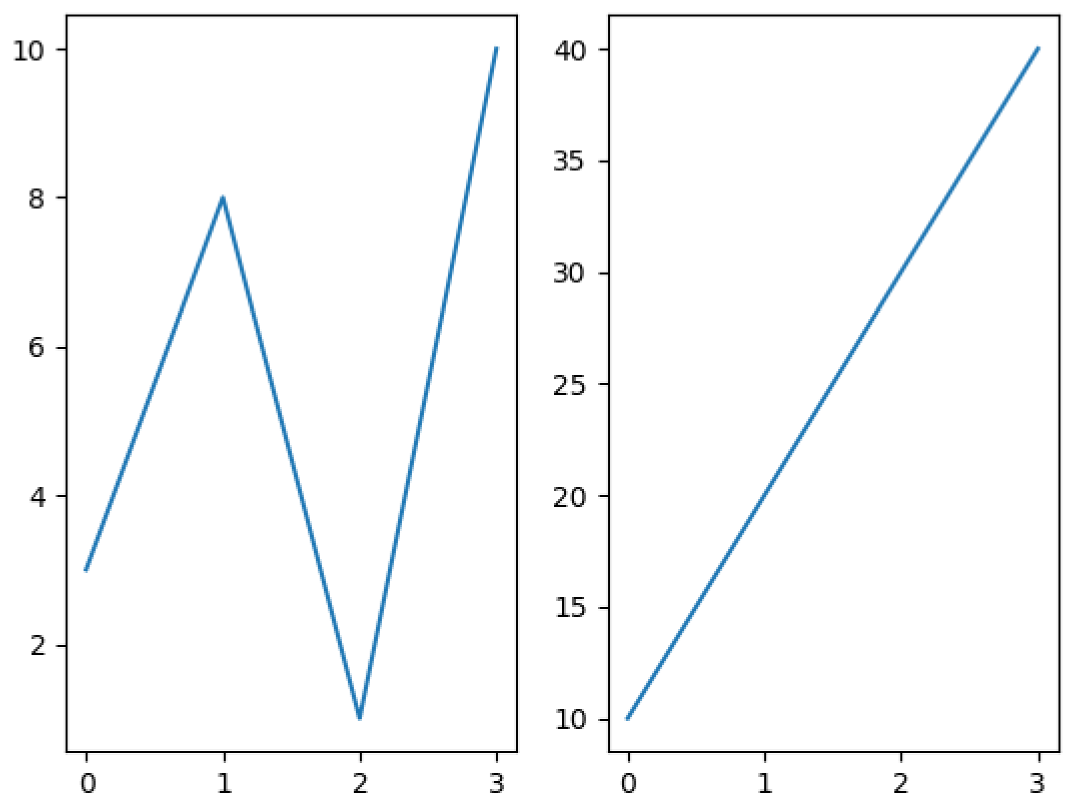
Display Multiple Plots

With the `subplots()` function you can draw multiple plots in one figure:

Draw 2 plots:

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

#plot 1:
x = np.array([0, 1, 2, 3])
y = np.array([3, 8, 1, 10])

plt.subplot(1, 2, 1)
plt.plot(x,y)

#plot 2:
x = np.array([0, 1, 2, 3])
y = np.array([10, 20, 30, 40])

plt.subplot(1, 2, 2)
plt.plot(x,y)

plt.show()
```

Result:



The subplots() Function

The `subplots()` function takes three arguments that describes the layout of the figure.

The layout is organized in rows and columns, which are represented by the *first* and *second* argument.

The third argument represents the index of the current plot.

```
plt.subplot(1, 2, 1)
#the figure has 1 row, 2 columns, and this plot is the first plot.
```

```
plt.subplot(1, 2, 2)
#the figure has 1 row, 2 columns, and this plot is the second plot.
```

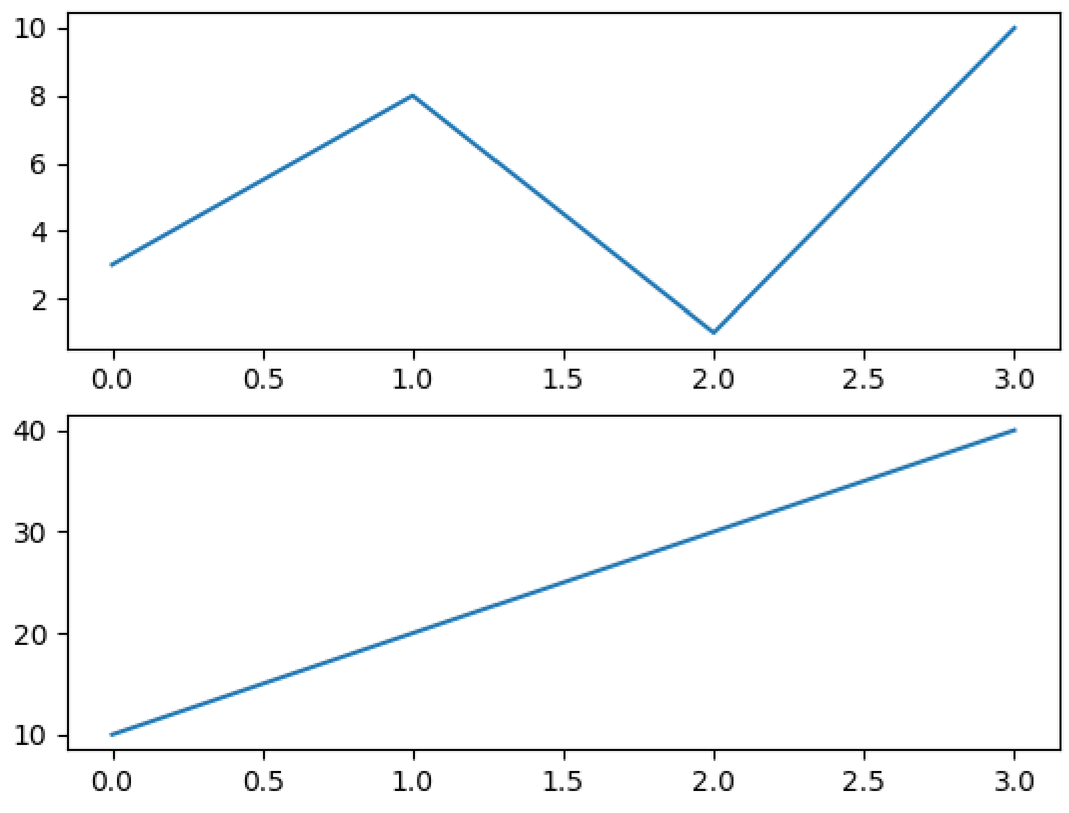
So, if we want a figure with 2 rows an 1 column (meaning that the two plots will be displayed on top of each other instead of side-by-side), we can write the syntax like this:

Draw 2 plots on top of each other:

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

#plot 1:
x = np.array([0, 1, 2, 3])
y = np.array([3, 8, 1, 10])
```

```
plt.subplot(2, 1, 1)
plt.plot(x,y)

#plot 2:
x = np.array([0, 1, 2, 3])
y = np.array([10, 20, 30, 40])

plt.subplot(2, 1, 2)
plt.plot(x,y)

plt.show()
```

Result:



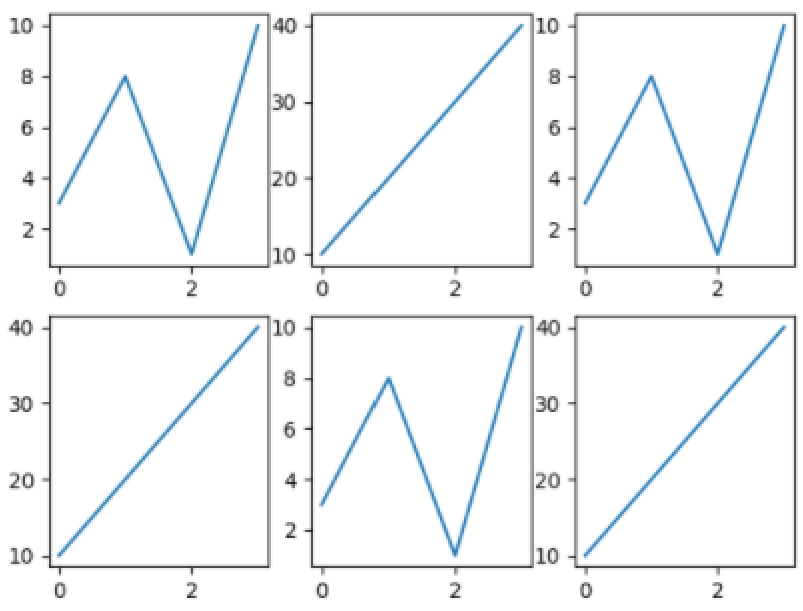
You can draw as many plots you like on one figure, just descibe the number of rows, columns, and the index of the plot.

Draw 6 plots:

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

```python
x = np.array([0, 1, 2, 3])
y = np.array([3, 8, 1, 10])

plt.subplot(2, 3, 1)
plt.plot(x,y)

x = np.array([0, 1, 2, 3])
y = np.array([10, 20, 30, 40])

plt.subplot(2, 3, 2)
plt.plot(x,y)

x = np.array([0, 1, 2, 3])
y = np.array([3, 8, 1, 10])

plt.subplot(2, 3, 3)
plt.plot(x,y)

x = np.array([0, 1, 2, 3])
y = np.array([10, 20, 30, 40])

plt.subplot(2, 3, 4)
plt.plot(x,y)

x = np.array([0, 1, 2, 3])
y = np.array([3, 8, 1, 10])

plt.subplot(2, 3, 5)
plt.plot(x,y)

x = np.array([0, 1, 2, 3])
y = np.array([10, 20, 30, 40])

plt.subplot(2, 3, 6)
plt.plot(x,y)

plt.show()
```

Result:



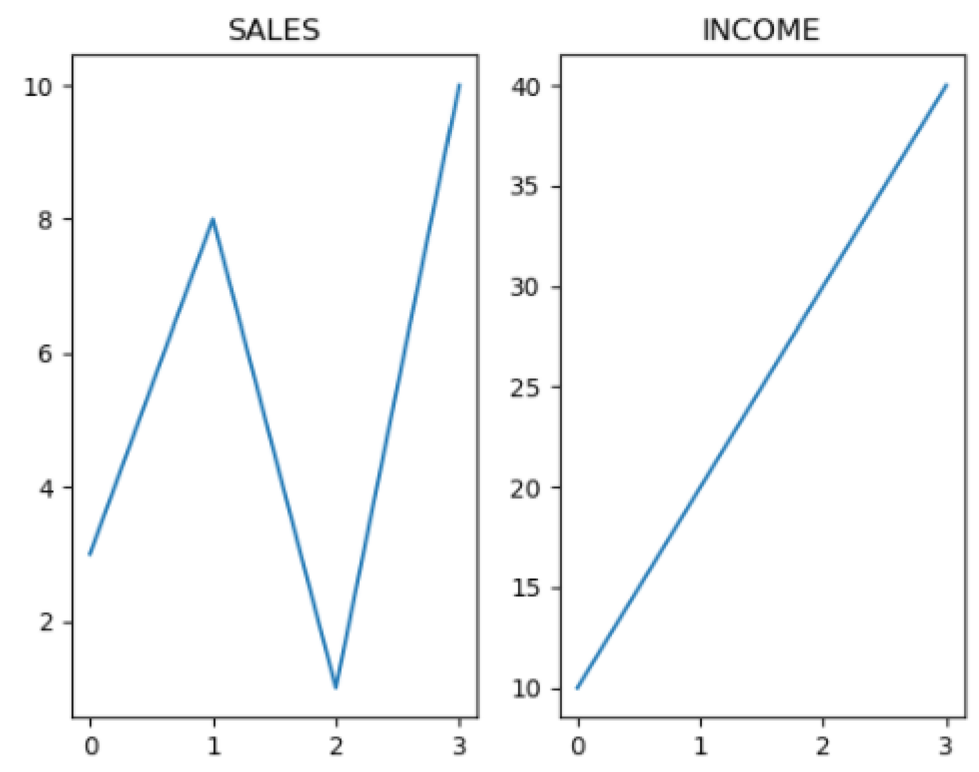
Title

You can add a title to each plot with the `title()` function:

2 plots, with titles:

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

#plot 1:
x = np.array([0, 1, 2, 3])
y = np.array([3, 8, 1, 10])

plt.subplot(1, 2, 1)
plt.plot(x,y)
plt.title("SALES")

#plot 2:
x = np.array([0, 1, 2, 3])
y = np.array([10, 20, 30, 40])

plt.subplot(1, 2, 2)
plt.plot(x,y)
plt.title("INCOME")
plt.show()
```

Result:



Super Title

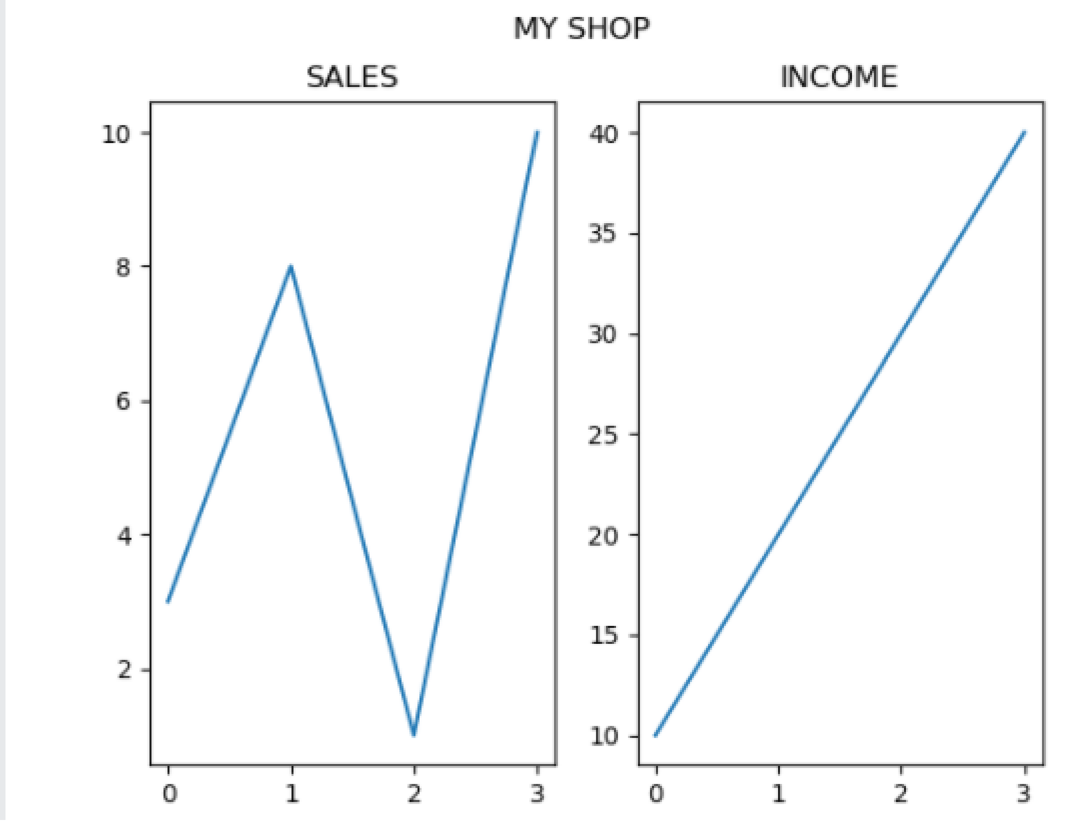
You can add a title to the entire figure with the `suptitle()` function:

Add a title for the entire figure:

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

#plot 1:
x = np.array([0, 1, 2, 3])
y = np.array([3, 8, 1, 10])

plt.subplot(1, 2, 1)
plt.plot(x,y)
plt.title("SALES")
```

```
#plot 2:
x = np.array([0, 1, 2, 3])
y = np.array([10, 20, 30, 40])

plt.subplot(1, 2, 2)
plt.plot(x,y)
plt.title("INCOME")

plt.suptitle("MY SHOP")
plt.show()
```

Result:



Creating Scatter Plots

With Pyplot, you can use the `scatter()` function to draw a scatter plot.

The `scatter()` function plots one dot for each observation. It needs two arrays of the same length, one for the values of the x-axis, and one for values on the y-axis:

A simple scatter plot:

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

x = np.array([5,7,8,7,2,17,2,9,4,11,12,9,6])
y = np.array([99,86,87,88,111,86,103,87,94,78,77,85,86])
```

```
plt.scatter(x, y)
plt.show()
```

The observation in the example above is the result of 13 cars passing by.

The X-axis shows how old the car is.

The Y-axis shows the speed of the car when it passes.

Are there any relationships between the observations?

It seems that the newer the car, the faster it drives, but that could be a coincidence, after all we only registered 13 cars.

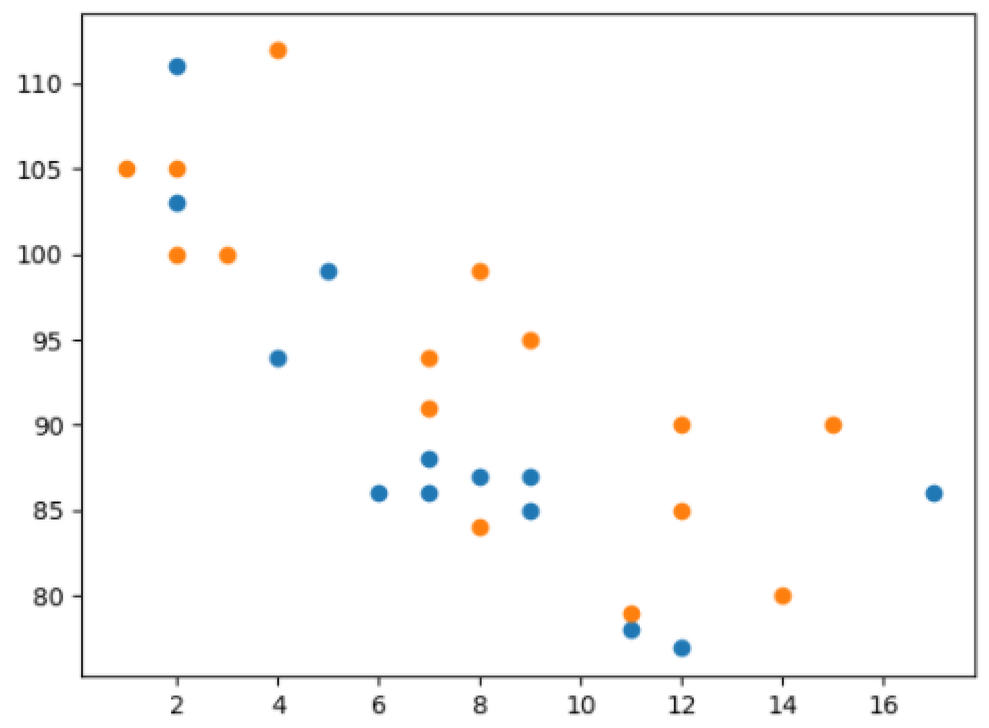
Compare Plots

In the example above, there seems to be a relationship between speed and age, but what if we plot the observations from another day as well? Will the scatter plot tell us something else?

Draw two plots on the same figure:

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

#day one, the age and speed of 13 cars:
x = np.array([5,7,8,7,2,17,2,9,4,11,12,9,6])
y = np.array([99,86,87,88,111,86,103,87,94,78,77,85,86])
plt.scatter(x, y)

#day two, the age and speed of 15 cars:
x = np.array([2,2,8,1,15,8,12,9,7,3,11,4,7,14,12])
y = np.array([100,105,84,105,90,99,90,95,94,100,79,112,91,80,85])
plt.scatter(x, y)

plt.show()
```

Result:



Note: The two plots are plotted with two different colors, by default blue and orange, you will learn how to change colors later in this chapter.

By comparing the two plots, I think it is safe to say that they both gives us the same conclusion: the newer the car, the faster it drives.

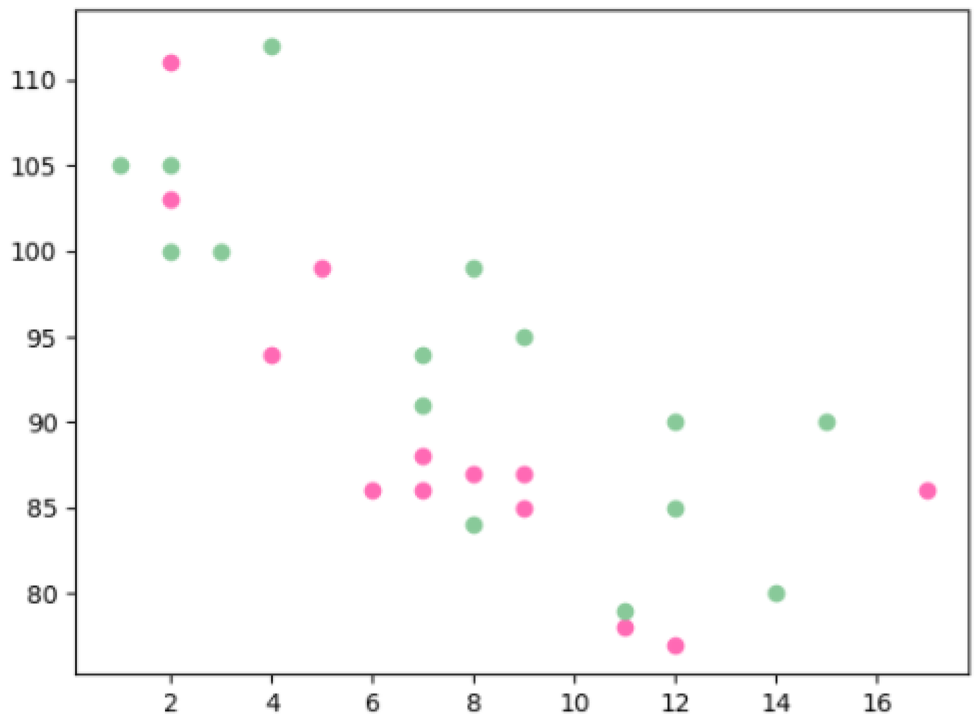
Colors

You can set your own color for each scatter plot with the `color` or the `c` argument:

Set your own color of the markers:

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

x = np.array([5,7,8,7,2,17,2,9,4,11,12,9,6])
y = np.array([99,86,87,88,111,86,103,87,94,78,77,85,86])
plt.scatter(x, y, color = 'hotpink')

x = np.array([2,2,8,1,15,8,12,9,7,3,11,4,7,14,12])
y = np.array([100,105,84,105,90,99,90,95,94,100,79,112,91,80,85])
plt.scatter(x, y, color = '#88c999')

plt.show()
```

Result:



Color Each Dot

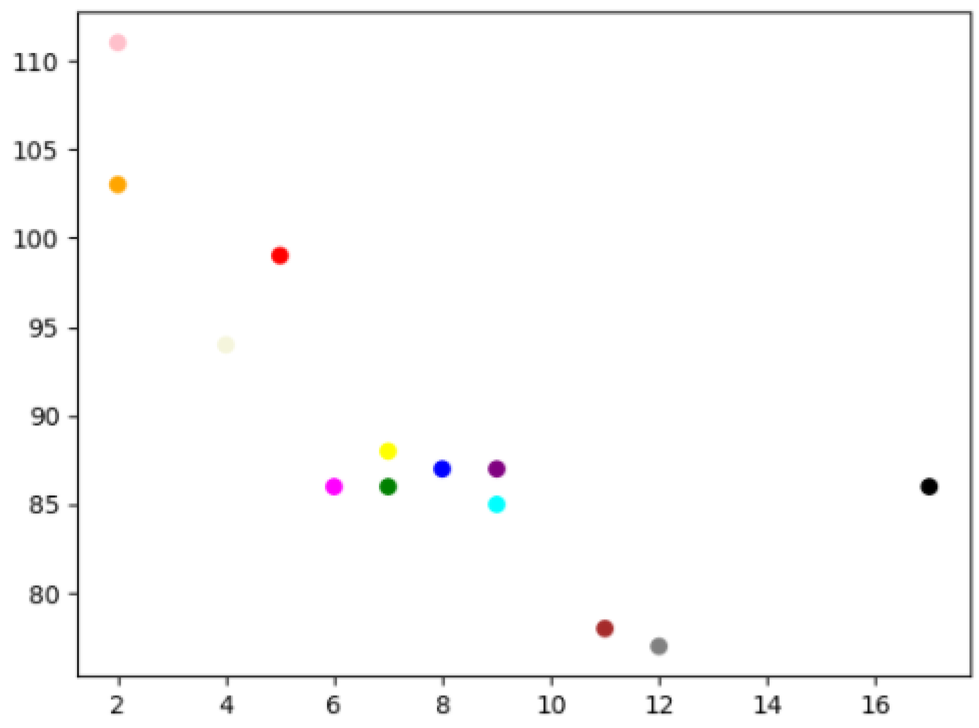
You can even set a specific color for each dot by using an array of colors as value for the `c` argument:

Note: You *cannot* use the `color` argument for this, only the `c` argument.

Set your own color of the markers:

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

x = np.array([5,7,8,7,2,17,2,9,4,11,12,9,6])
y = np.array([99,86,87,88,111,86,103,87,94,78,77,85,86])
colors =
np.array(["red","green","blue","yellow","pink","black","orange","purple
","beige","brown","gray","cyan","magenta"])

plt.scatter(x, y, c=colors)

plt.show()
```

Result:



https://www.w3schools.com/python/matplotlib_scatter.asp

Size

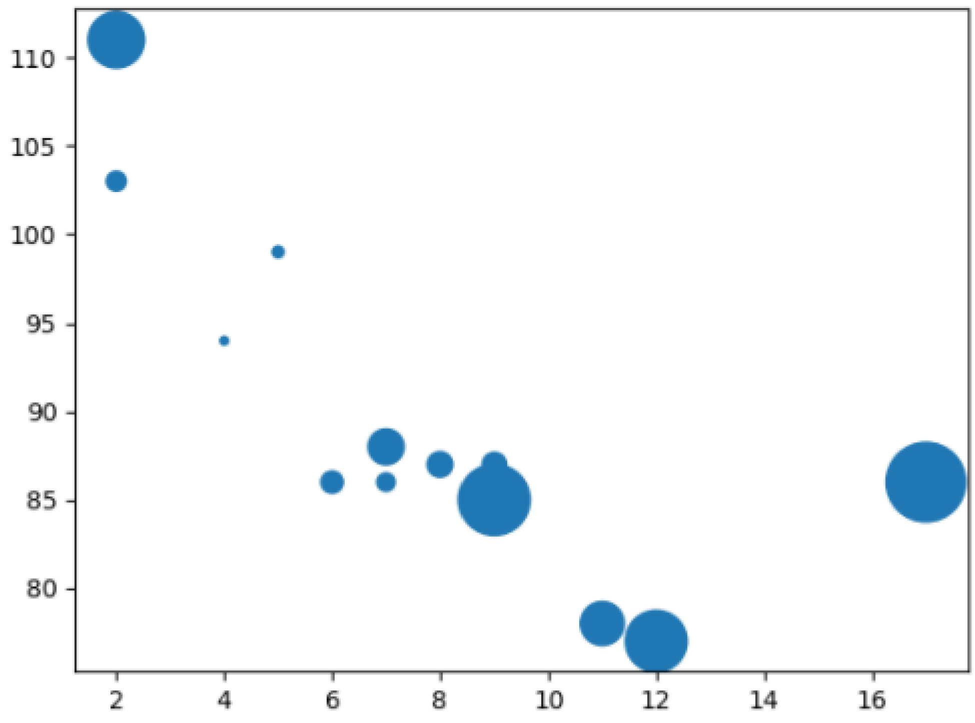
You can change the size of the dots with the `s` argument.

Just like colors, make sure the array for sizes has the same length as the arrays for the x- and y-axis:

Set your own size for the markers:

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

x = np.array([5,7,8,7,2,17,2,9,4,11,12,9,6])
y = np.array([99,86,87,88,111,86,103,87,94,78,77,85,86])
sizes = np.array([20,50,100,200,500,1000,60,90,10,300,600,800,75])

plt.scatter(x, y, s=sizes)

plt.show()
```

Result:



SCATTER PLOT

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

girls_grades = [89, 90, 70, 89, 100, 80, 90, 100, 80, 34]
boys_grades = [30, 29, 49, 48, 100, 48, 38, 45, 20, 30]
grades_range = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]

plt.scatter(grades_range, girls_grades, color = ‘r’)

plt.scatter(grades_range, boys_grades, color = ‘g’)

plt.xlabel(‘Grades Range’)

plt.ylabel(‘Grades Scored’)

plt.show()
```

2)

```
import matplotlib.pyplot as plt %matplotlib inline

import matplotlib.pyplot as plt
%matplotlib inline
# Plot plt.plot([1,2,3,4,10])
```

The `plt.plot` accepts 3 basic arguments in the following order: **(x, y, format).**

This format is a short hand combination of `{color}{marker}{line}` .

For example, the format `'go-'` has 3 characters standing for: ‘green colored dots with solid line’. By omitting the line part (‘-’) in the end, you will be left with only green dots (‘go’), which makes it draw a scatterplot.

Few commonly used short hand format examples are:

* `'r*--'` : ‘red stars with dashed lines’
* `'ks.'` : ‘black squares with dotted line’ (‘k’ stands for black)
* `'bD-.'` : ‘blue diamonds with dash-dot line’.

For a complete list of colors, markers and linestyles, check out the `help(plt.plot)` command.

```
1) # 'go' stands for green dots
plt.plot([1,2,3,4,5], [1,2,3,4,10], 'go')
plt.show()

2) # Draw two sets of points
plt.plot([1,2,3,4,5], [1,2,3,4,10], 'go')
# green dots
plt.plot([1,2,3,4,5], [2,3,4,5,11], 'b*')
# blue stars
plt.show()

3)
plt.plot([1,2,3,4,5], [1,2,3,4,10], 'go', label='GreenDots')

plt.plot([1,2,3,4,5], [2,3,4,5,11], 'b*', label='Bluestars')
plt.title('A Simple Scatterplot')

plt.xlabel('X')
plt.ylabel('Y')

# legend text comes from the plot's label parameter.
plt.legend(loc='best')
plt.show()
```