Week 4

#Data Science/4 - Python for Data Science, AI & Development#

Reading & Writing Files with Open theory

In this section, we will use Python's built-in open function to create a file object, and obtain the data from a "txt" file. We will use Python's open function to get a file object.

We can apply a method to that object to read data from the file.
We can open the file, `Example1.txt`, as follows:

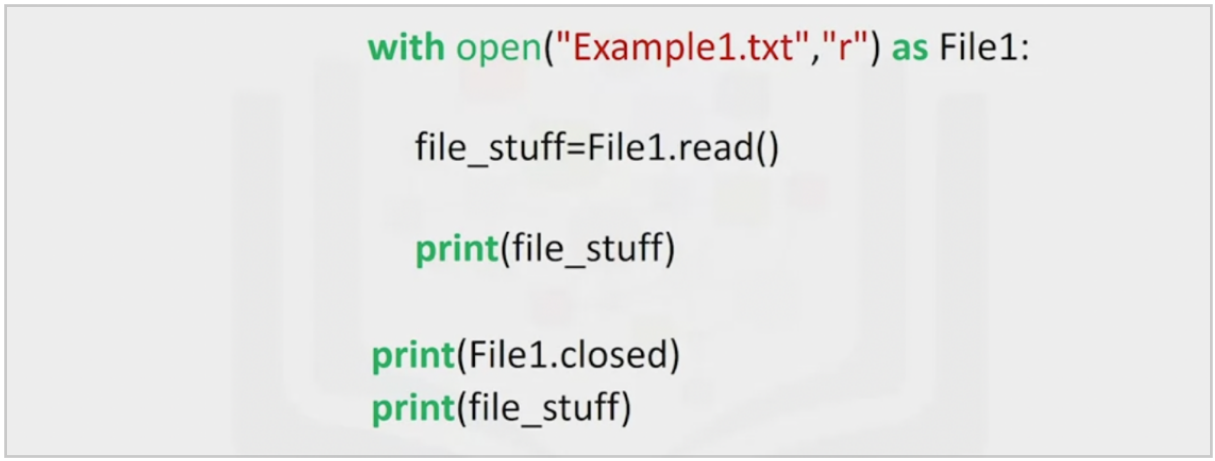
```
File1 = open("/resources/data/Example2.txt","w")
```

We use the `open` function. The **first argument** is the **file path**. This is made up of the file name, and the file directory. The **second parameter** is the **mode**. Common values used include:

- 'r' for reading,
- 'w' for writing,,
- 'a' for appending.

Finally, we have the file object.

We can now use the file object to obtain information about the file. We can use the data attribute name to get the name of the file. The result is a string that contains the name of the file. We can see what mode the object is in using the data attribute mode, and 'r' is shown representing read. You should always close the file object using the method close. This may get tedious sometimes, so let's use the "with" statement. Using a "with" statement to open a file is better practice because it automatically closes the file. The code will run everything in the indent block, then closes the file.



This code reads the file, Example1.txt. We can use the file **object** `File1`. The code will perform all operations in the indent block (store the file content and print it) then close the file at the end of the indent.

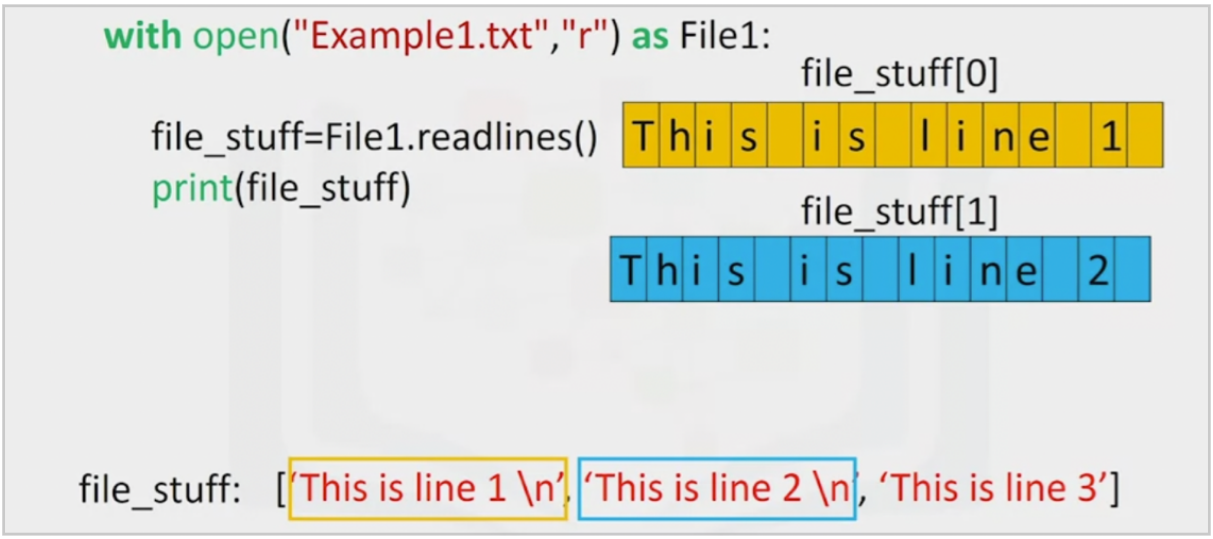
- The **method** `read` stores the values of the file in the variable `file_stuff` as a **string**.
- You can print the file content.
- You can check if the file content is closed, but you cannot read from it outside the indent.
- But you can print the file content outside the indent as well.

We can print the file content.

```
print(file_stuff)
This is line 1
This is line 2
This is line 3

file_stuff:
 This is line 1 \n This is line 2 \n This is line 3
```

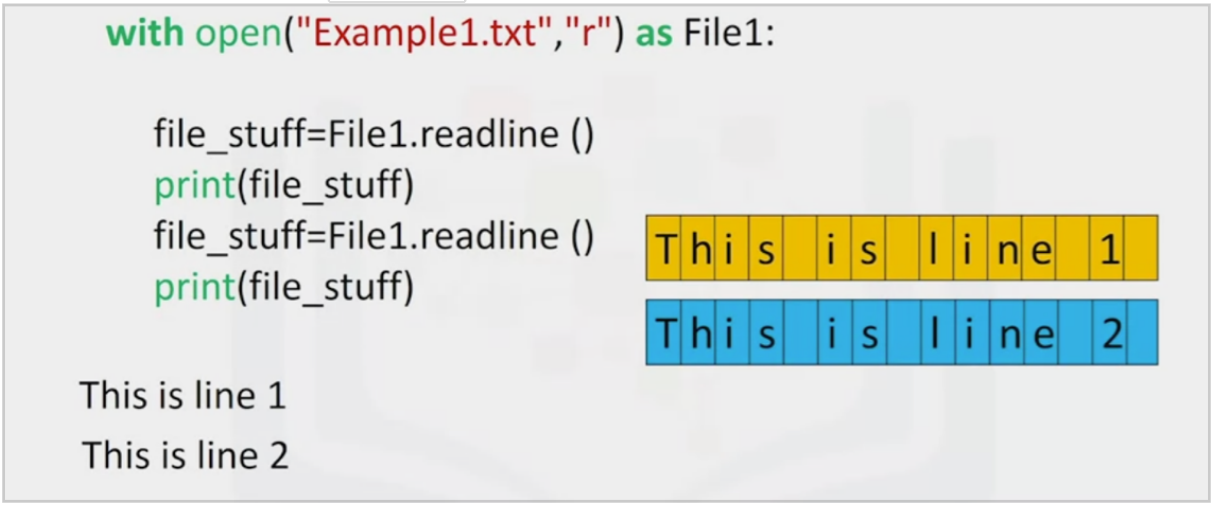
When we examine the raw string, we will see the `\n` this is how Python knows to start a new line.

We can output every line as an element in a list using the **method** `readlines` (note the plural in the method).



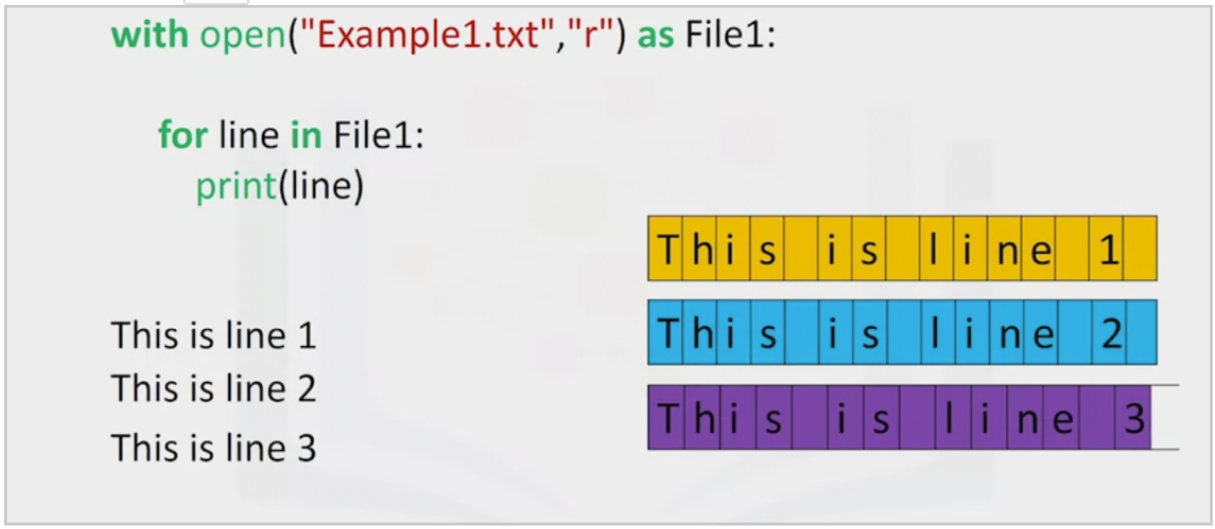
The first line corresponds to the first element in the list. The second line corresponds to the second element in the list, and so on.

We can use the **method** `readline` (note the singular) to read the first line of the file.

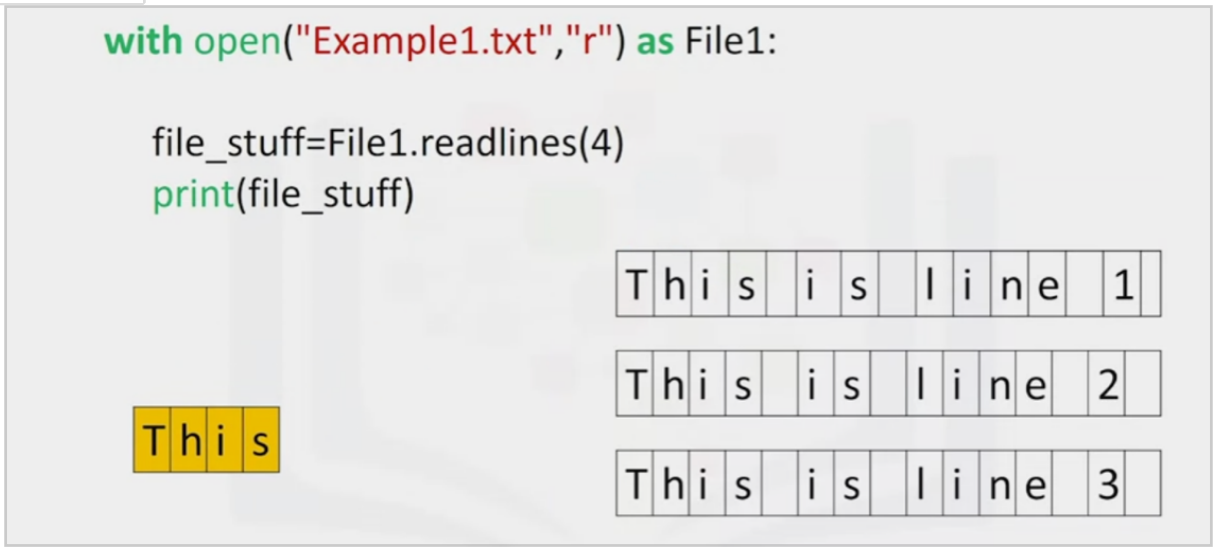


If we run this command, it will store the first line in the variable "file*stuff" then print the first line. We can use the method "readline" twice. The first time it's called, it will save the first line in the variable "file*stuff," and then print the first line. The second time it's called, it will save the second line in the variable "file_stuff," and then print the second line.

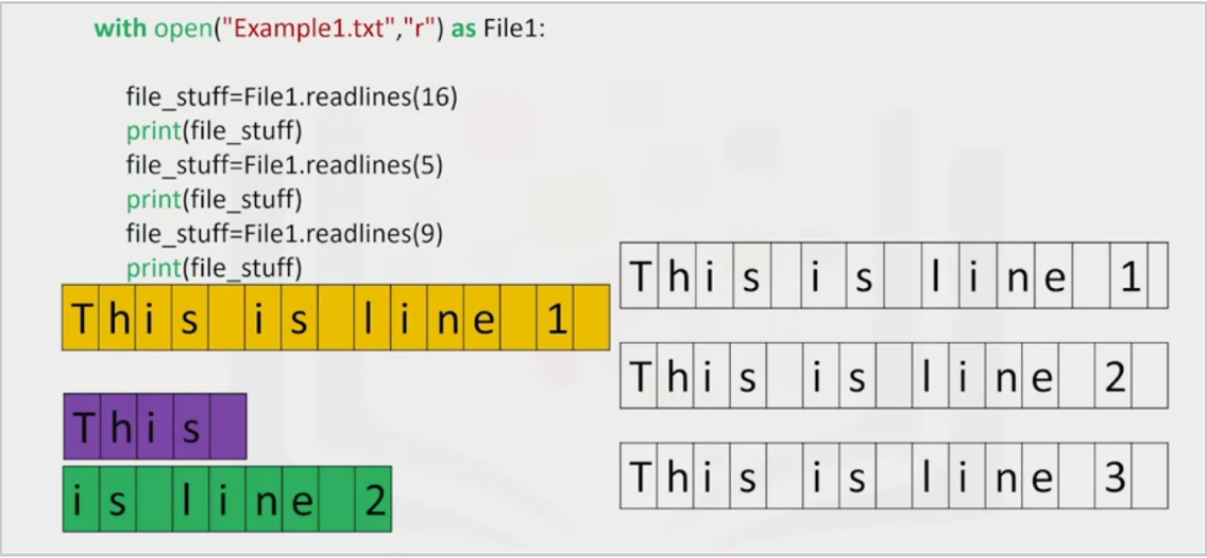
We can use a `for` loop to print out each line individually as follows.



Let's represent every character in a string as a grid. We can specify the number of characters we would like to read from the string as an **argument** to the **method** `readlines`.



When we use a **4** as an argument in the method `readlines`, **we print out the first four characters in the file**. Each time we call the method, we will progress through the text.



If we call a method with the arguments 16, the first 16 characters are printed out, and then the new line. If we call the method a second time, the next five characters are printed out. Finally, if we call the method the last time with the argument nine, the last nine characters are printed out.

Reading & Writing Files with Open Lab

Objectives

After completing this lab you will be able to:

- Read text files using Python libraries

Table of Contents

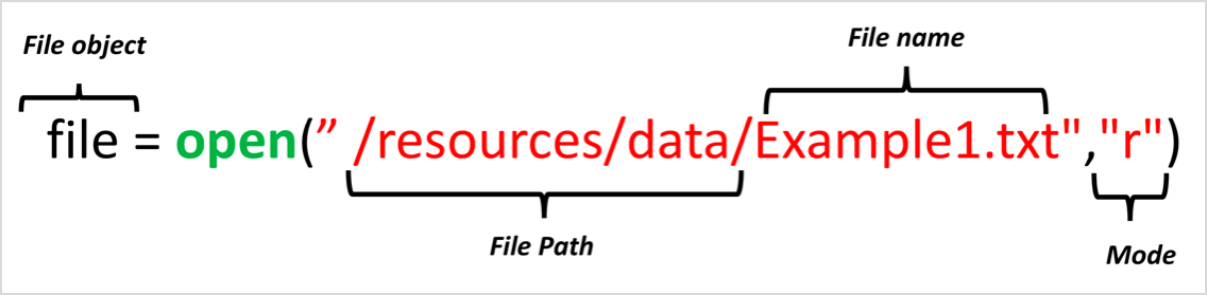
Download Data

```
import urllib.request
url = 'https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBMDeveloperSkillsNetwork-PY0101EN-SkillsNetwork/labs/Module%204/data/example1.txt'
filename = 'Example1.txt'
urllib.request.urlretrieve(url, filename)
```

('Example1.txt', <http.client.HTTPMessage at 0x7fa0b9637f40>)

Reading Text Files

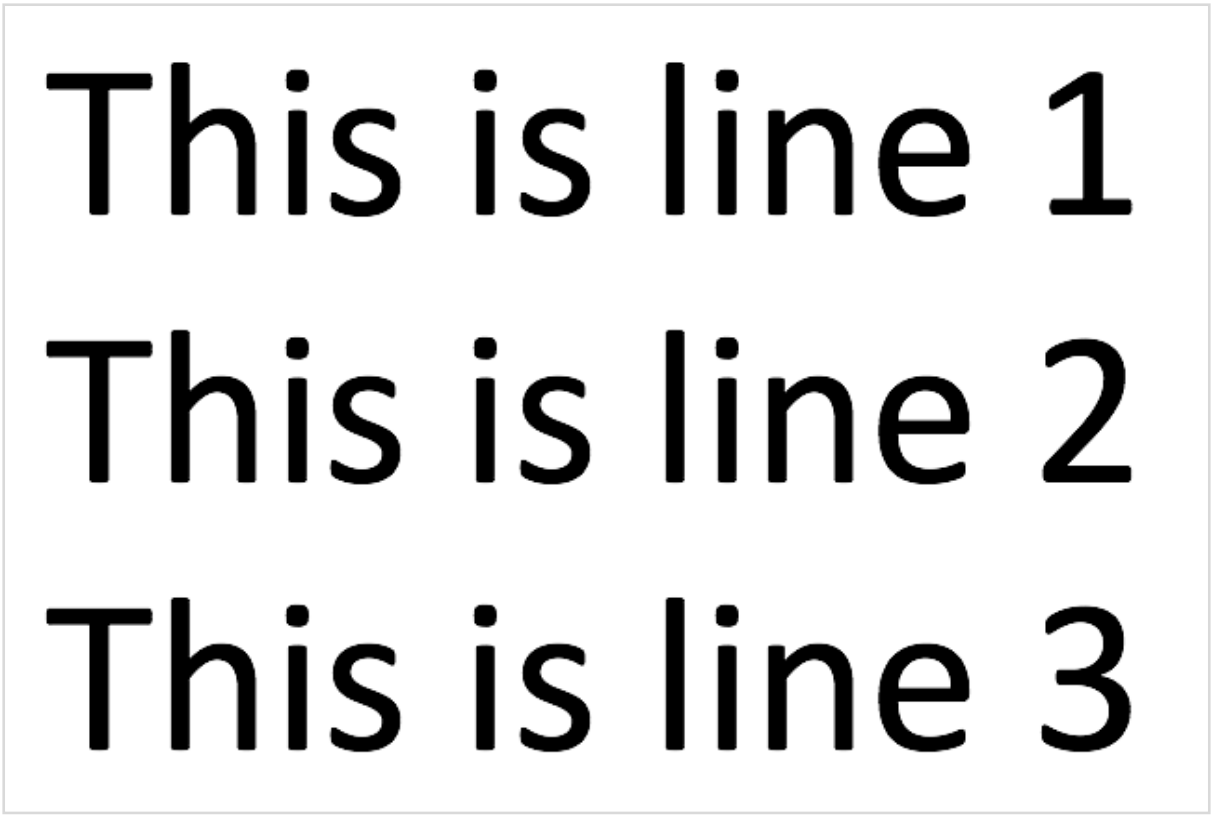
One way to read or write a file in Python is to use the built-in `open` function. The `open` function provides a **File object** that contains the methods and attributes you need in order to read, save, and manipulate the file. In this notebook, we will only cover **.txt** files. The first parameter you need is the file path and the file name. An example is shown as follow:



The mode argument is optional and the default value is `r`. In this notebook we only cover two modes:

- `r`: Read mode for reading files
- `w`: Write mode for writing files

For the next example, we will use the text file **Example1.txt**. The file is shown as follows:



We read the file:

```
# Read the Example1.txt
example1 = "Example1.txt"
file1 = open(example1, "r")
```

We can view the attributes of the file.
The name of the file:

```
# Print the path of file
file1.name
```

'Example1.txt'

The mode the file object is in:

```
# Print the mode of file, either 'r' or 'w'
```

```
file1.mode
```

'r'

We can read the file and assign it to a variable :

```
# Read the file
FileContent = file1.read()
FileContent
```

'This is line 1 \nThis is line 2\nThis is line 3'

The **/n** means that there is a new line.

We can print the file:

```
# Print the file with '\n' as a new line
print(FileContent)
```

```
This is line 1
This is line 2
This is line 3
```

The file is of type string:

```
# Type of file content
type(FileContent)
```

str

It is very important that the file is closed in the end. This frees up resources and ensures

consistency across different python versions.

```
# Close file after finish
file1.close()
```

A Better Way to Open a File

Using the `with` statement is better practice, it automatically closes the file even if the code encounters an exception. The code will run everything in the indent block then close the file object.

```
# Open file using with
with open(example1, "r") as file1:
    FileContent = file1.read()
    print(FileContent)



    This is line 1
    This is line 2
    This is line 3
```

The file object is closed, you can verify it by running the following cell:

```
# Verify if the file is closed
file1.closed
```

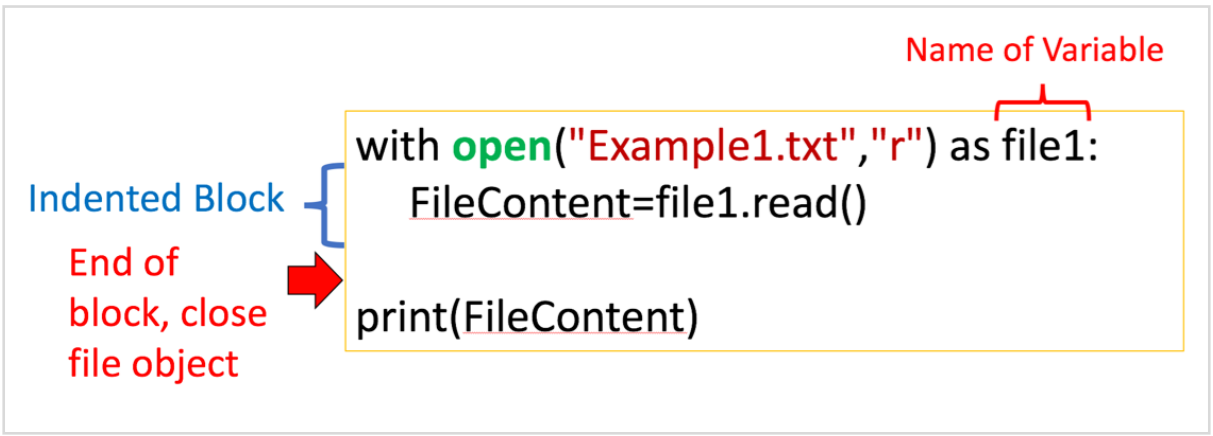
```
True
```

We can see the info in the file:

```
# See the content of file
print(FileContent)
```

```
This is line 1
This is line 2
This is line 3
```

The syntax is a little confusing as the file object is after the `as` statement. We also don't explicitly close the file. Therefore we summarize the steps in a figure:



We don't have to read the entire file, for example, we can read the first 4 characters by entering three as a parameter to the method `.read()`:

```
# Read first four characters
with open(example1, "r") as file1:
    print(file1.read(4))
```

This

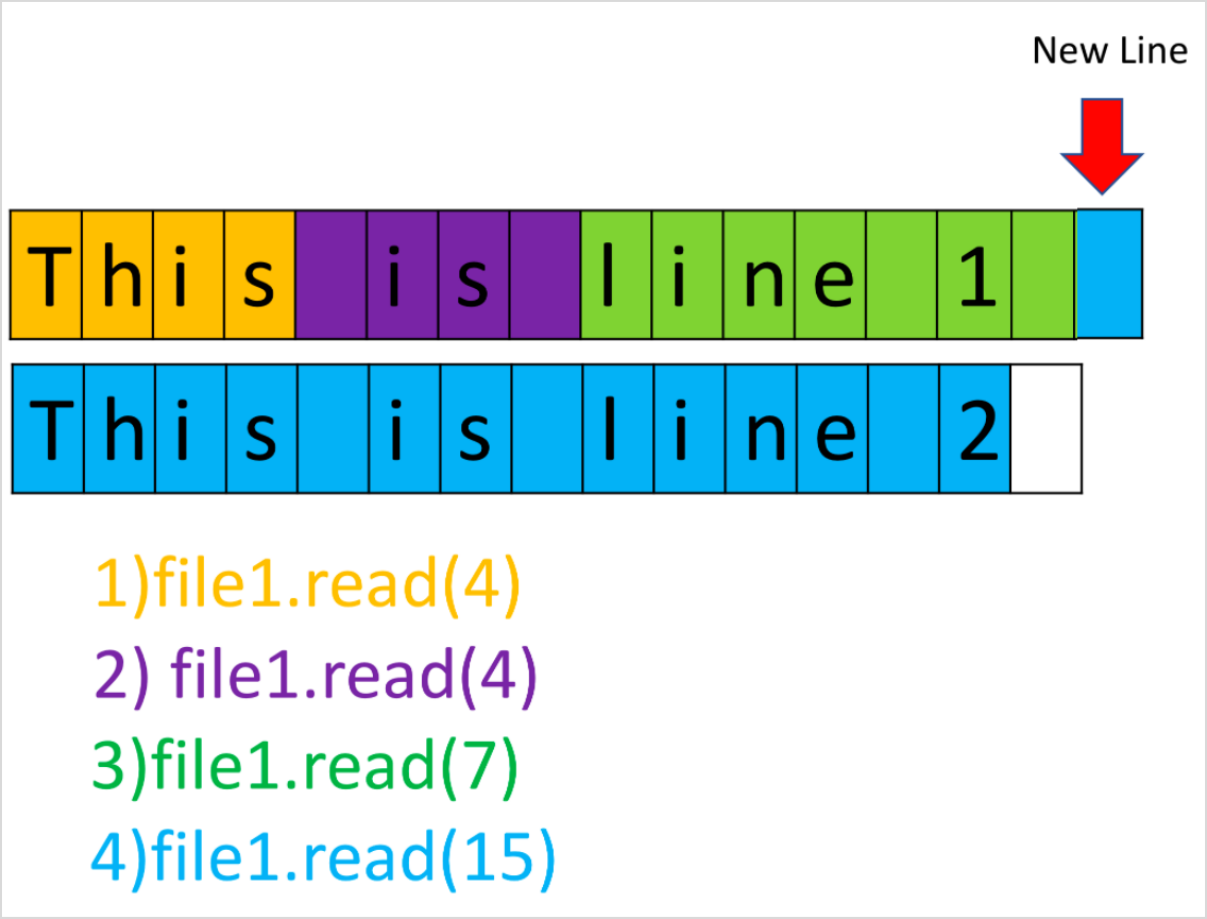
Once the method `.read(4)` is called the first 4 characters are called. If we call the method again, the next 4 characters are called. The output for the following cell will demonstrate the process for different inputs to the method `read()`:

```
# Read certain amount of characters

with open(example1, "r") as file1:
```

```
print(file1.read(4))
print(file1.read(4))
print(file1.read(7))
print(file1.read(15))
```

```
This
 is
line 1

This is line 2
```

The process is illustrated in the below figure, and each color represents the part of the file read after the method `read()` is called:



Here is an example using the same file, but instead we read 16, 5, and then 9 characters at a time:

```
# Read certain amount of characters
with open(example1, "r") as file1:
    print(file1.read(16))
    print(file1.read(5))
    print(file1.read(9))
```

```
This is line 1

This
is line 2
```

We can also read one line of the file at a time using the method `readline()`:

```
# Read one line
with open(example1, "r") as file1:
    print("first line: " + file1.readline())
```

first line: This is line 1

We can also pass an argument to `readline()` to specify the number of charecters we want to read. However, unlike `read()`, `readline()` can only read one line at most.

```
with open(example1, "r") as file1:
    print(file1.readline(20)) # does not read past the end of line
    print(file1.read(20)) # Returns the next 20 chars
```

```
This is line 1

This is line 2
This
```

We can use a loop to iterate through each line:

```
# Iterate through the lines
with open(example1,"r") as file1:
        i = 0;
        for line in file1:
            print("Iteration", str(i), ": ", line)
            i = i + 1
```

```
Iteration 0 :  This is line 1

Iteration 1 :  This is line 2

Iteration 2 :  This is line 3
```

We can use the method `readlines()` to save the text file to a list:

```
# Read all lines and save as a list
with open(example1, "r") as file1:
    FileasList = file1.readlines()
```

Each element of the list corresponds to a line of text:

```
# Print the first line
FileasList[0]
```

'This is line 1 \n'

Print the second line

FileasList[1]

```

```

```
# Print the third line
FileasList[2]
```

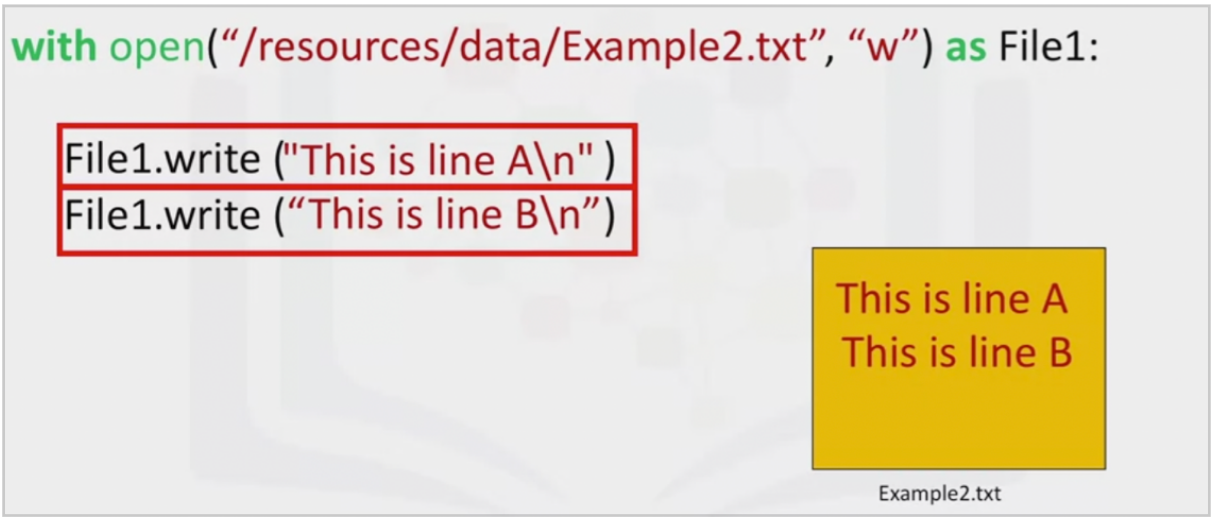
'This is line 3'

Writing Files with Open theory

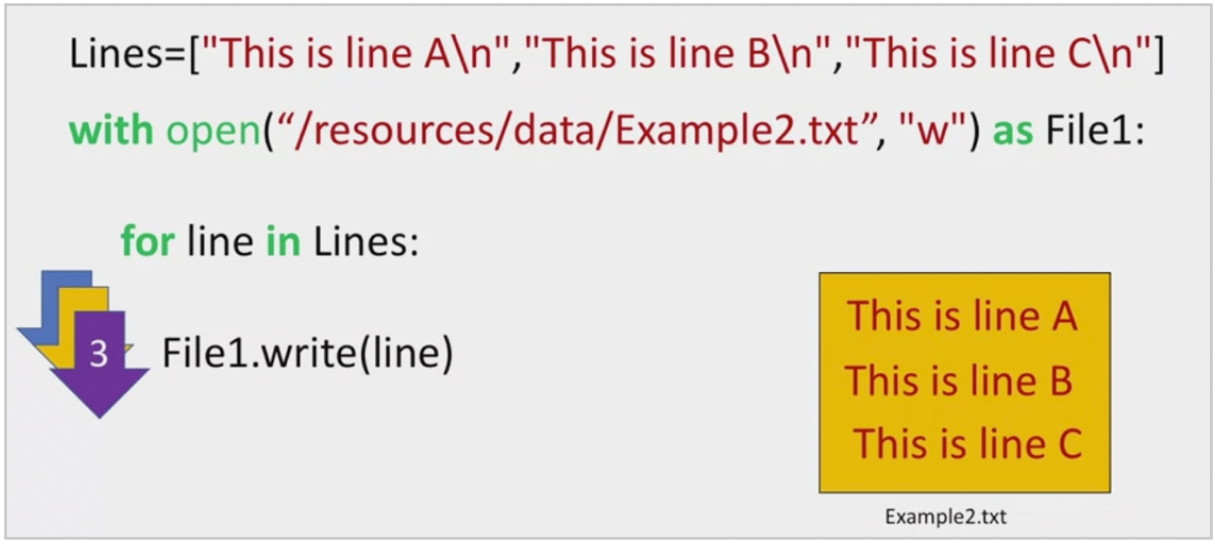
We can also write to files using the `open` function. We will use Python's open function to get a file object to create a text file. We can apply method write to write data to that file. As a result, text will be written to the file.
We can create the file Example2.txt as follows.

```
File1 = open("/resources/data/Example2.txt","w")
```

We use the open function. The first argument is the file path. This is made up of the file name. If you have that file in your directory, it will be overwritten, and the file directory. We set the mode parameter to W for writing. Finally, we have the file object.

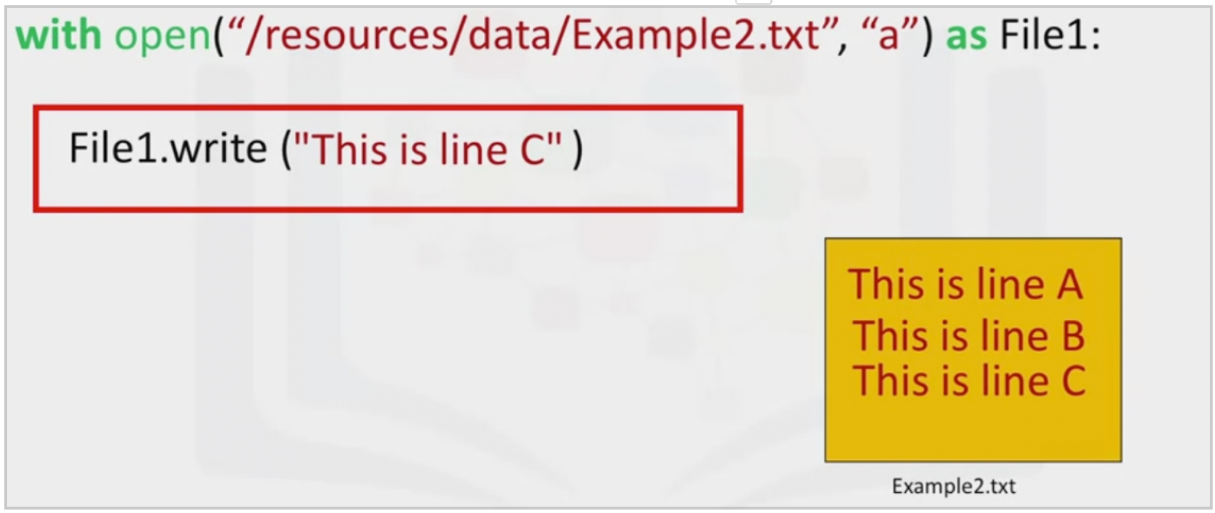


As before we use the with statement. The code will run everything in the indent block then close the file. We create the file object, File1. We use the open function. This creates a file Example2.txt in your directory. We use the method write, to write data into the file. The argument is the text we would like input into the file. If we use the write method successively, each time it is called, it will write to the file. The first time it is called, we will write, "This is line A " to represent a new line. The second time we call the method, it will write, "this is line B " then it will close the file.
We can write each element in a list to a file. As before, we use a with command and the open function to create a file.



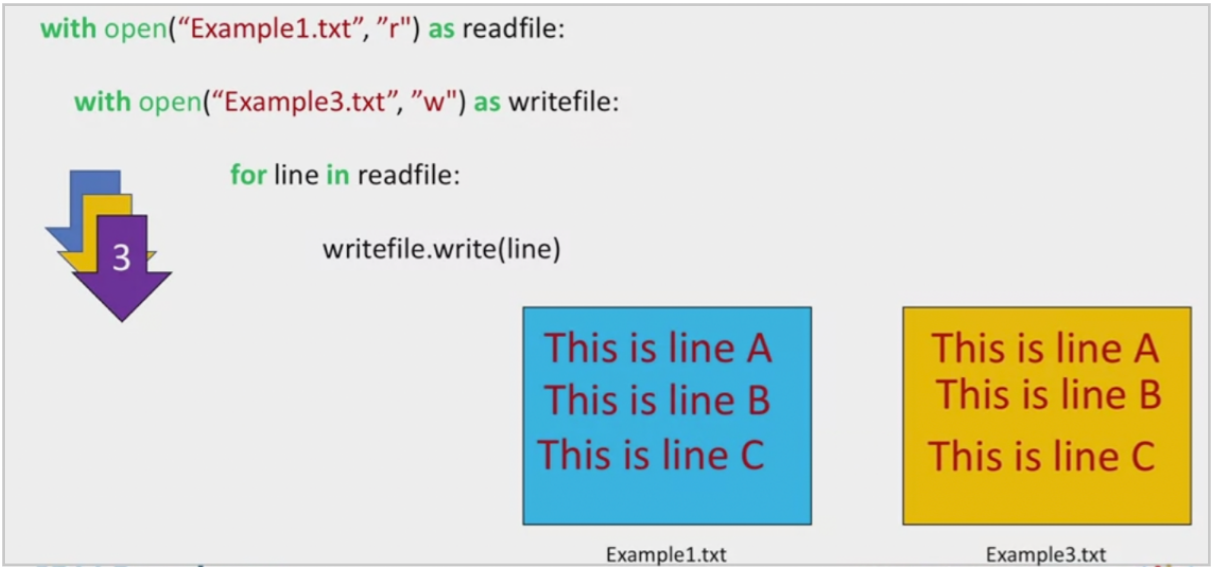
The list, Lines, has three elements consisting of text. We use a for loop to read each element of the first lines and pass it to the variable line. The first iteration of the loop writes the first element of the list to the file Example2. The second iteration writes the second element of the list and so on. At the end of the loop, the file will be closed.

We can **set the mode to appended** using a lowercase `a`.



This will not create a new file but just use the existing file. If we call the method write, it will just write to the existing file, then add "This is line C" then close the file.

We can copy one file to a new file as follows.



First, we read the file Example1 and interact with it via the file object, read file. Then we create a new file Example3 and use the file object write file to interact with it. The for loop takes a line from the file object, read file, and stores it in the file Example3 using the file object, write file. The first iteration copies the first line. The second iteration copies the second line, till the end of the file is reached. Then both files are closed.

Writing Files with Open - Lab

Objectives

After completing this lab you will be able to:

- Write to files using Python libraries

Table of Contents

Writing Files

We can open a file object using the method `write()` to save the text file to a list. To write to a file, the mode argument must be set to **w**. Let's write a file **Example2.txt** with the line: **"This is line A"**

```
# Write line to file
exmp2 = 'Example2.txt'
```

```
with open(exmp2, 'w') as writefile:
    writefile.write("This is line A")
```

We can read the file to see if it worked:

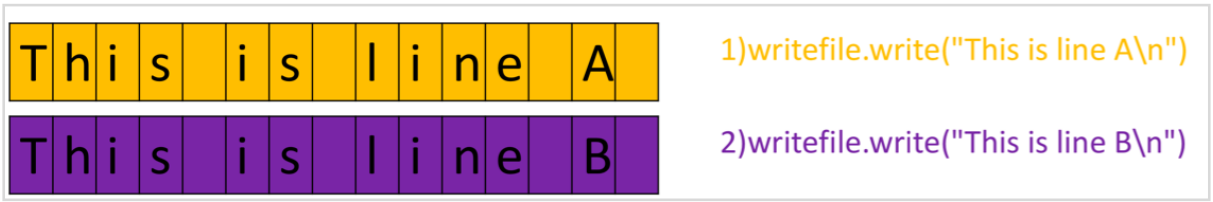
```
# Read file
with open(exmp2, 'r') as testwritefile:
    print(testwritefile.read())
```

```
This is line A
```

We can write multiple lines:

```
# Write lines to file
with open(exmp2, 'w') as writefile:
    writefile.write("This is line A\n")
    writefile.write("This is line B\n")
```

The method `.write()` works similar to the method `.readline()`, except instead of reading a new line it writes a new line. The process is illustrated in the figure. The different colour coding of the grid represents a new line added to the file after each method call.



You can check the file to see if your results are correct

```
# Check whether write to file
with open(exmp2, 'r') as testwritefile:
    print(testwritefile.read())
```

```
    This is line A
    This is line B
```

We write a list to a **.txt** file as follows:

```
# Sample list of text
Lines = ["This is line A\n", "This is line B\n", "This is line C\n"]
Lines
```

['This is line A\n', 'This is line B\n', 'This is line C\n']

```
# Write the strings in the list to text file
with open('Example2.txt', 'w') as writefile:
    for line in Lines:
        print(line)
        writefile.write(line)


    This is line A
    This is line B
    This is line C
```

We can verify the file is written by reading it and printing out the values:

```
# Verify if writing to file is successfully executed
with open('Example2.txt', 'r') as testwritefile:
    print(testwritefile.read())


    This is line A
    This is line B
    This is line C
```

However, note that setting the mode to **w** overwrites all the existing data in the file.

```
with open('Example2.txt', 'w') as writefile:
    writefile.write("Overwrite\n")
with open('Example2.txt', 'r') as testwritefile:
    print(testwritefile.read())
```

```
Overwrite
```

Appending Files

We can write to files without losing any of the existing data as follows by setting the mode argument to append: **a**. you can append a new line as follows:

```
# Write a new line to text file

with open('Example2.txt', 'a') as testwritefile:
    testwritefile.write("This is line C\n")
    testwritefile.write("This is line D\n")
    testwritefile.write("This is line E\n")
```

You can verify the file has changed by running the following cell:

```
# Verify if the new line is in the text file

with open('Example2.txt', 'r') as testwritefile:
    print(testwritefile.read())
```

```
Overwrite
This is line C
This is line D
```

```
    This is line E
```

Additional modes

It's fairly ineffecient to open the file in **a** or **w** and then reopening it in **r** to read any lines. Luckily we can access the file in the following modes:

- `r+` : Reading and writing. Cannot truncate the file.
- `w+` : Writing and reading. Truncates the file.
- `a+` : Appending and Reading. Creates a new file, if none exists.

You dont have to dwell on the specifics of each mode for this lab.

Let's try out the `a+` mode:

```
with open('Example2.txt', 'a+') as testwritefile:
    testwritefile.write("This is line E\n")
    print(testwritefile.read())
```

There were no errors but `read()` also did not output anything. This is because of our location in the file.

Most of the file methods we've looked at work in a certain location in the file. `.write()` writes at a certain location in the file. `.read()` reads at a certain location in the file and so on. You can think of this as moving your pointer around in the notepad to make changes at specific location.

Opening the file in **w** is akin to opening the .txt file, moving your cursor to the beginning of the text file, writing new text and deleting everything that follows.
Whereas opening the file in **a** is similiar to opening the .txt file, moving your cursor to the very end and then adding the new pieces of text.
It is often very useful to know where the 'cursor' is in a file and be able to control it. The following methods allow us to do precisely this -

- `.tell()` - returns the current position in bytes
- `.seek(offset,from)` - changes the position by 'offset' bytes with respect to 'from'.

From can take the value of 0,1,2 corresponding to beginning, relative to current position and end

Now lets revisit **a+**

```
with open('Example2.txt', 'a+') as testwritefile:
    print("Initial Location: {}".format(testwritefile.tell()))

    data = testwritefile.read()
    if (not data):  #empty strings return false in python
            print('Read nothing')
    else:
            print(testwritefile.read())

    testwritefile.seek(0,0) # move 0 bytes from beginning.

    print("\nNew Location : {}".format(testwritefile.tell()))
    data = testwritefile.read()
    if (not data):
            print('Read nothing')
    else:
            print(data)

    print("Location after read: {}".format(testwritefile.tell()) )


    Initial Location: 85
    Read nothing

    New Location : 0
    Overwrite
    This is line C
    This is line D
    This is line E
    This is line E
    This is line E

    Location after read: 85
```

Finally, a note on the difference between **w+** and **r+**. Both of these modes allow access to read and write methods, however, opening a file in **w+** overwrites it and deletes all pre-existing data.
To work with a file on existing data, use **r+** and **a+**. While using **r+**, it can be useful to add a `.truncate()` method at the end of your data. This will reduce the file to your data and delete everything that follows.
In the following code block, Run the code as it is first and then run it with the `.truncate()`.

```
with open('Example2.txt', 'r+') as testwritefile:
    data = testwritefile.readlines()
    testwritefile.seek(0,0) #write at beginning of file

    testwritefile.write("Line 1" + "\n")
    testwritefile.write("Line 2" + "\n")
    testwritefile.write("Line 3" + "\n")
    testwritefile.write("finished\n")
    #Uncomment the line below
    #testwritefile.truncate()
    testwritefile.seek(0,0)
    print(testwritefile.read())



    Line 1
    Line 2
    Line 3
    finished
```

Copy a File

Let's copy the file **Example2.txt** to the file **Example3.txt**:

```
# Copy file to another
with open('Example2.txt','r') as readfile:
    with open('Example3.txt','w') as writefile:
```

```
            for line in readfile:
                  writefile.write(line)
```

We can read the file to see if everything works:

```
# Verify if the copy is successfully executed
with open('Example3.txt','r') as testwritefile:
    print(testwritefile.read())



    Line 1
    Line 2
    Line 3
    finished
```

After reading files, we can also write data into files and save them in different file formats like **.txt, .csv, .xls (for excel files) etc**. You will come across these in further examples

Now go to the directory to ensure the **.txt** file exists and contains the summary data that we wrote.

Exercise

Your local university's Raptors fan club maintains a register of its active members on a .txt document. Every month they update the file by removing the members who are not active. You have been tasked with automating this with your Python skills.
Given the file currentMem, Remove each member with a 'no' in their Active coloumn. Keep track of each of the removed members and append them to the exMem file. Make sure the format of the original files in preserved. (*Hint: Do this by reading/writing whole lines and ensuring the header remains*)
Run the code block below prior to starting the exercise. The skeleton code has been provided for you, Edit only the cleanFiles function.

```
#Run this prior to starting the exercise
from random import randint as rnd
```

```
memReg = 'members.txt'
exReg = 'inactive.txt'
fee =('yes','no')

def genFiles(current,old):
    with open(current,'w+') as writefile:
        writefile.write('Membership No  Date Joined  Active  \n')
        data = "{:^13}  {:<11}  {:<6}\n"
        for rowno in range(20):
            date = str(rnd(2015,2020))+ '-' + str(rnd(1,12))+'-'+str(rnd(1,25))
            writefile.write(data.format(rnd(10000,99999),date,fee[rnd(0,1)]))

    with open(old,'w+') as writefile:
        writefile.write('Membership No  Date Joined  Active  \n')
        data = "{:^13}  {:<11}  {:<6}\n"
        for rowno in range(3):
            date = str(rnd(2015,2020))+ '-' + str(rnd(1,12))+'-'+str(rnd(1,25))
            writefile.write(data.format(rnd(10000,99999),date,fee[1]))

genFiles(memReg,exReg)
```

Start your solution below:

```
# Definisco una funzione

def cleanFiles(currentMem,exMem):
    '''
    currentMem: File containing list of current members
    exMem: File containing list of old members

    Removes all rows from currentMem containing 'no' and appends them to exMem
    '''
    # Definisco le istruzioni per aprire i due file dati in pasto alla funzione
    with open(currentMem,'r+') as writeFile:
        with open(exMem,'a+') as appendFile:
```

```
            # apro il writableFile all'inizio del file
            writeFile.seek(0)

            # leggo il contenuto del file e lo memorizzo in una variabile
sottoforma di lista
            members = writeFile.readlines()

            # definisco una variabile per l'intestazione del writableFile
prendendo i valori dalla variabile members
            headers = members[0]

            # elimino la prima riga della lista members
            members.pop(0)

            # spostiamo tutti gli i membri non attivi in un file chiamato
"inactive"
            inactive = [member for member in members if ('no' in member)]
            # questa parte di codice equivale al seguente ciclo for
            '''
            for member in members:
                if 'no' in members:
                    inactive.append(member)
            '''

            # ora bisogna creare i file con i membri separati per quelli attivi
"writeFile" e quelli non attivi "appendFile"
            writeFile.seek(0) # entro nel file dalla posizione iniziale
            writeFile.write(headers) # usando il metodo "write" cancello tutto
il contenuto del file inserendo come intestazione il contenuto della variabile
"header"

            # creo un ciclo for che analizza nuovamente la variabile "members"
e la compara con la variabile "inactive"
            for member in members:
                if (member is inactive):
                    appendFile.write(member)
                else:
                    writeFile.write(member)
            writeFile.truncate()
```

```
# Code to help you see the files
# Leave as is
memReg = 'members.txt'
exReg = 'inactive.txt'
cleanFiles(memReg,exReg)

headers = "Membership No  Date Joined  Active  \n"
with open(memReg,'r') as readFile:
    print("Active Members: \n\n")
    print(readFile.read())

with open(exReg,'r') as readFile:
    print("Inactive Members: \n\n")
    print(readFile.read())
```

```
    Active Members:

    Membership No  Date Joined  Active
        75431      2017-4-24    no
        52063      2019-3-5     yes
        99517      2016-1-24    yes
        11188      2015-3-1     no
        96793      2017-2-16    no
        26780      2016-9-21    no
        61350      2016-6-7     no
        45312      2016-12-4    yes
        78156      2020-2-5     no
        71493      2020-8-11    no
        73309      2020-3-5     no
        80649      2015-10-25   no
        33998      2020-7-15    yes
        83206      2017-12-23   no
        63368      2020-5-2     yes
        94980      2015-6-3     yes
        36024      2018-7-22    yes
        85187      2020-5-25    yes
        49541      2016-6-19    yes
```

```
        44701       2017-9-4      no

    Inactive Members:

    Membership No  Date Joined  Active
        33223       2019-5-6      no
        73422       2020-7-3      no
        53097       2017-2-9      no
```

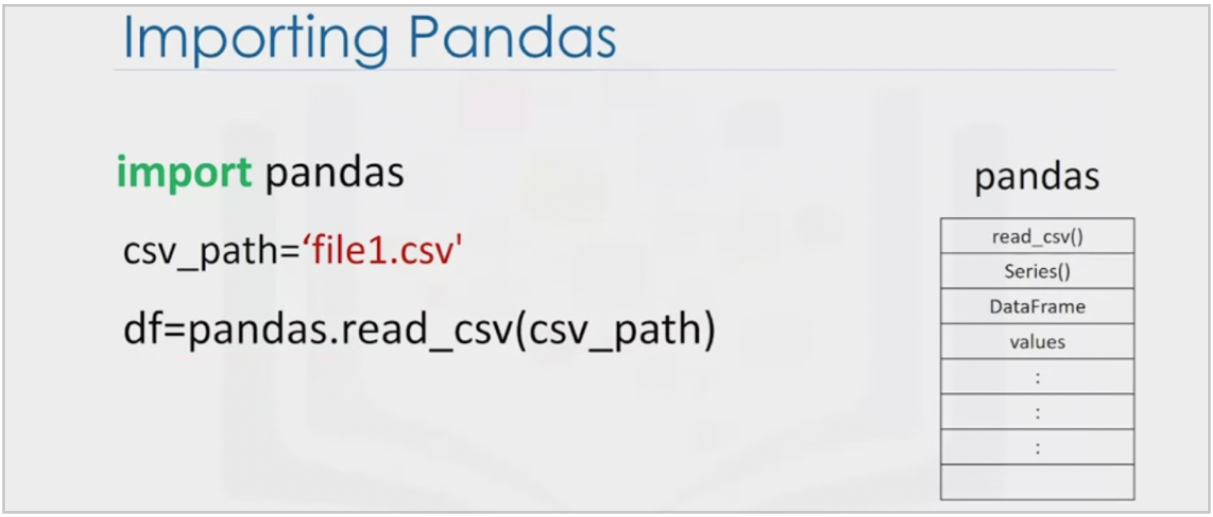
Pandas

Loading Data with Pandas

Dependencies or libraries are pre-written code to help solve problems. In this video, we will introduce Pandas, a popular library for data analysis. We can import the library or a dependency like Pandas using the following command.

- We start with the import command followed by the name of the library.
- We now have access to a large number of pre-built classes and functions. This assumes the library is installed. In our lab environment, all the necessary libraries are installed.
- Let's say we would like to load a CSV file using the Pandas built-in function, read csv.
- A CSV is a typical file type used to store data.

We simply typed the word Pandas, then a dot, and the name of the function with all the inputs. Typing Pandas all the time may get tedious.



We can use the `as` statement to shorten the name of the library. In this case, we use the standard abbreviation, `pd`. Now we type pd, and a dot, followed by the name of the function we would like to use. In this case, read_csv.

```
import pandas as pd
csv_path='file1.csv'
df= pd.read_csv(csv_path)
```

We are not limited to the abbreviation pd. In this case, we use the term banana. We will stick with pd for the rest of this video.

Let's examine this code more in-depth. One way Pandas allows you to work with data is with the **data frame**. Let's go over the process to go from a CSV file to a data frame.

Dataframes

```
csv_path='file1.csv'
df= pd.read_csv(csv_path)
df.head()
```

	Artist	Album	Released	Length	Genre	Music Recording Sales (millions)	Claimed Sales (millions)	Released.1	Soundtrack	Rating
0	Michael Jackson	Thriller	1982	0:42:19	pop, rock, R&B	46.0	65	30-Nov-82	NaN	10.0
1	AC/DC	Back in Black	1980	0:42:11	hard rock	26.1	50	25-Jul-80	NaN	9.5
2	Pink Floyd	The Dark Side of the Moon	1973	0:42:49	progressive rock	24.2	45	01-Mar-73	NaN	9.0
3	Whitney Houston	The Bodyguard	1992	0:57:44	R&B, soul, pop	27.4	44	17-Nov-92	Y	8.5
4	Meat Loaf	Bat Out of Hell	1977	0:46:33	hard rock, progressive rock	20.6	43	21-Oct-77	NaN	8.0

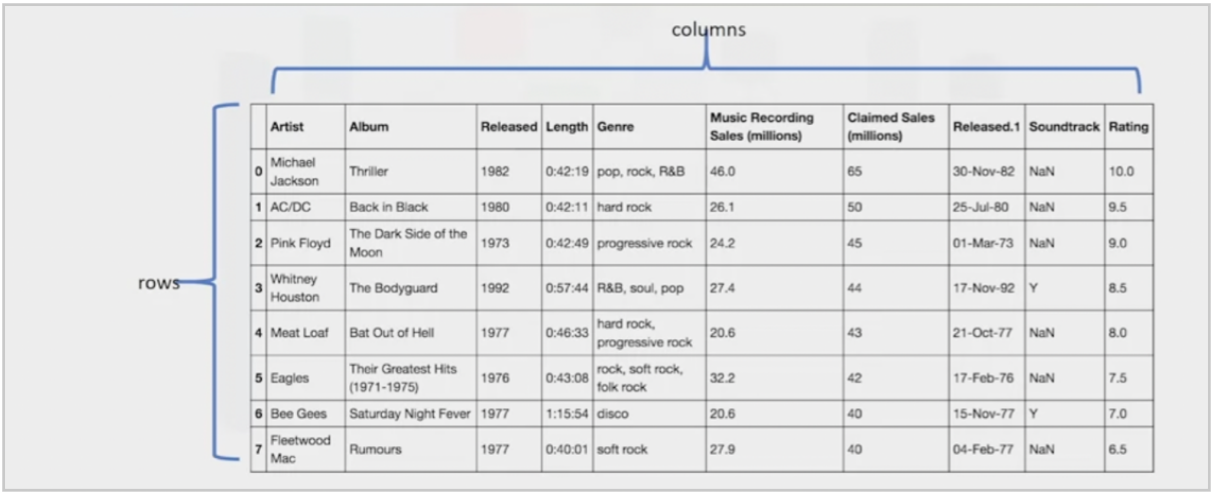
This variable stores the path of the CSV. It is used as an argument to the read_csv function. The result is stored to the variable df. This is short for data frame. Now that we have the data in a data frame, we can work with it. We can use the method head to examine the first five rows of a data frame.

The process for loading an Excel file is similar.

```
xlsx_path='file1.xlsx'
df= pd.read_excel (xlsx_path)
df.head()
```

	Artist	Album	Released	Length	Genre	Music Recording Sales (millions)	Claimed Sales (millions)	Released.1	Soundtrack	Rating
0	Michael Jackson	Thriller	1982	0:42:19	pop, rock, R&B	46.0	65	30-Nov-82	NaN	10.0
1	AC/DC	Back in Black	1980	0:42:11	hard rock	26.1	50	25-Jul-80	NaN	9.5
2	Pink Floyd	The Dark Side of the Moon	1973	0:42:49	progressive rock	24.2	45	01-Mar-73	NaN	9.0
3	Whitney Houston	The Bodyguard	1992	0:57:44	R&B, soul, pop	27.4	44	17-Nov-92	Y	8.5
4	Meat Loaf	Bat Out of Hell	1977	0:46:33	hard rock, progressive rock	20.6	43	21-Oct-77	NaN	8.0

We use the path of the Excel file. The function reads Excel. The result is a data frame. A data frame is comprised of rows and columns.



	Artist	Album	Released	Length	Genre	Music Recording Sales (millions)	Claimed Sales (millions)	Released.1	Soundtrack	Rating
0	Michael Jackson	Thriller	1982	0:42:19	pop, rock, R&B	46.0	65	30-Nov-82	NaN	10.0
1	AC/DC	Back in Black	1980	0:42:11	hard rock	26.1	50	25-Jul-80	NaN	9.5
2	Pink Floyd	The Dark Side of the Moon	1973	0:42:49	progressive rock	24.2	45	01-Mar-73	NaN	9.0
3	Whitney Houston	The Bodyguard	1992	0:57:44	R&B, soul, pop	27.4	44	17-Nov-92	Y	8.5
4	Meat Loaf	Bat Out of Hell	1977	0:46:33	hard rock, progressive rock	20.6	43	21-Oct-77	NaN	8.0
5	Eagles	Their Greatest Hits (1971-1975)	1976	0:43:08	rock, soft rock, folk rock	32.2	42	17-Feb-76	NaN	7.5
6	Bee Gees	Saturday Night Fever	1977	1:15:54	disco	20.6	40	15-Nov-77	Y	7.0
7	Fleetwood Mac	Rumours	1977	0:40:01	soft rock	27.9	40	04-Feb-77	NaN	6.5

We can create a data frame out of a dictionary. The keys correspond to the column labels. The values or lists corresponding to the rows.

We then **cast the dictionary to a data frame** using the function `data frame`.

```
songs = {'Album' : ['Thriller','Back in Black', 'The Dark Side of the Moon',\
'The Bodyguard','Bat Out of Hell'],
 'Released' : [1982,1980,1973,1992,1977],
'Length':['00:42:19','00:42:11','00:42:49','00:57:44','00:46:33']}

songs_frame = pd.DataFrame(songs)
```

	Album	Length	Released
0	Thriller	00:42:19	1982
1	Back in Black	00:42:11	1980
2	The Dark Side of the Moon	00:42:49	1973
3	The Bodyguard	00:57:44	1992
4	Bat Out of Hell	00:46:33	1977

We can see the direct correspondence between the table. The **keys** correspond to the **table headers**. The **values** are **lists corresponding to the rows**.

We can create a new data frame consisting of one column.

```
x=df[ ['Length'] ]
```

	Artist	Album	Release	Length	Genre	Music Recording Sales (millions)	Claimed Sales (millions)	Released.1	Soundtrack	Rating
0	Michael Jackson	Thriller	1982	0:42:19	op, rock, R&B	46.0	65	30-Nov-82	NaN	10.0
1	AC/DC	Back in Black	1980	0:42:11	ard rock	26.1	50	25-Jul-80	NaN	9.5
2	Pink Floyd	The Dark Side of the Moon	1973	0:42:49	rogressive rock	24.2	45	01-Mar-73	NaN	9.0
3	Whitney Houston	The Bodyguard	1992	0:57:44	&B, soul, pop	27.4	44	17-Nov-92	Y	8.5
4	Meat Loaf	Bat Out of Hell	1977	0:46:33	ard rock, rogressive rock	20.6	43	21-Oct-77	NaN	8.0
5	Eagles	Their Greatest Hits (1971-1975)	1976	0:43:08	ck, soft rock, lk rock	32.2	42	17-Feb-76	NaN	7.5
6	Bee Gees	Saturday Night Fever	1977	1:15:54	isco	20.6	40	15-Nov-77	Y	7.0
7	Fleetwood Mac	Rumours	1977	0:40:01	oft rock	27.9	40	04-Feb-77	NaN	6.5

x

	Length
0	0:42:19
1	0:42:11
2	0:42:49
3	0:57:44
4	0:46:33
5	0:43:08
6	1:15:54
7	0:40:01

We just put the data frame name, in this case, `df`, and the name of the column header

enclosed in double brackets. The result is a new data frame comprised of the original column.

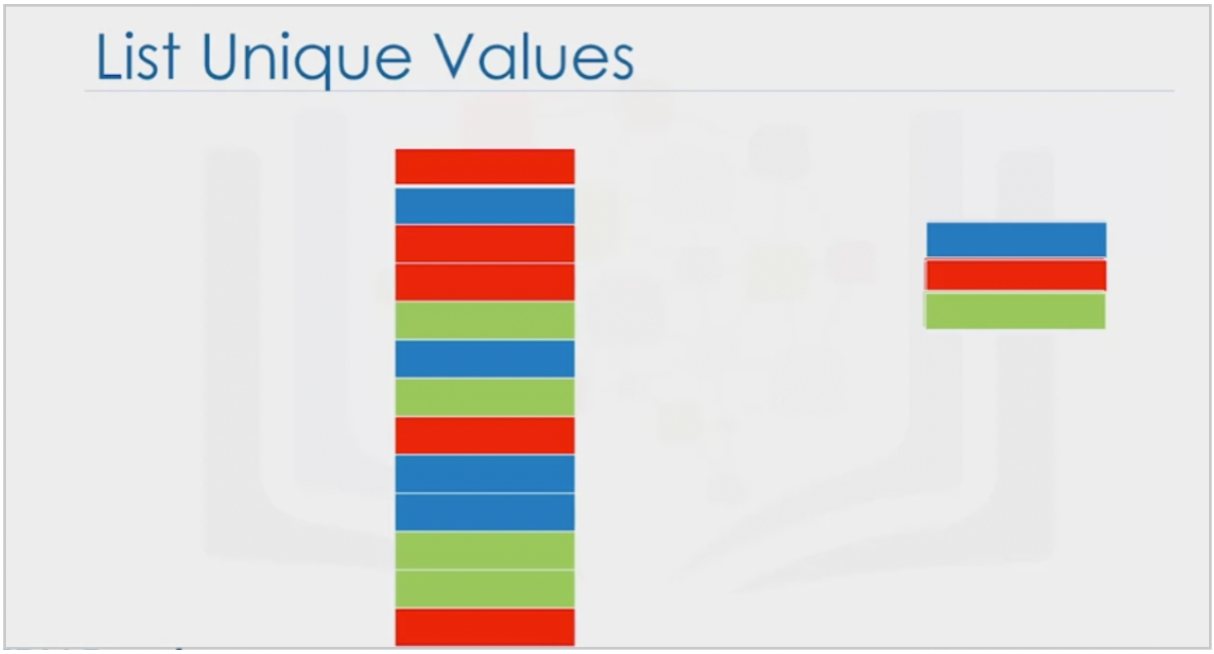
You can do the same thing for multiple columns.

	Artist	Length	Genre
0	Michael Jackson	0:42:19	pop, rock, R&B
1	AC/DC	0:42:11	hard rock
2	Pink Floyd	0:42:49	progressive rock
3	Whitney Houston	0:57:44	R&B, soul, pop
4	Meat Loaf	0:46:33	hard rock, progressive rock
5	Eagles	0:43:08	rock, soft rock, folk rock
6	Bee Gees	1:15:54	disco
7	Fleetwood Mac	0:40:01	soft rock

We just put the data frame name, in this case, df, and the name of the multiple column headers enclosed in double brackets. The result is a new data frame comprised of the specified columns.

Pandas: Working with and Saving Data

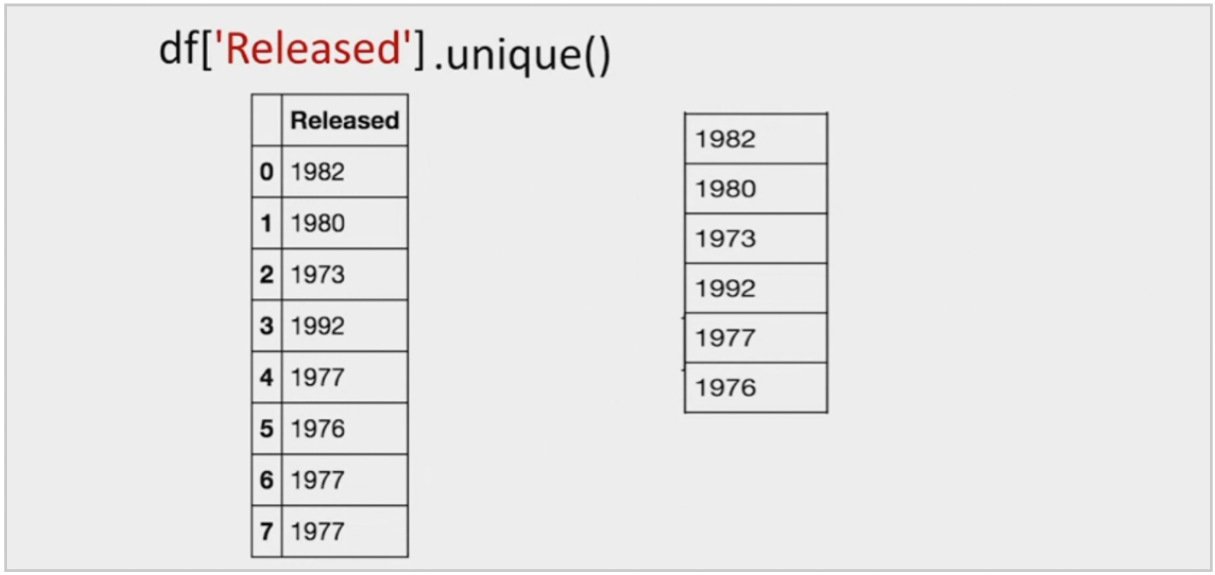
When we have a data frame we can work with the data and save the results in other formats. Consider the stack of 13 blocks of different colors. We can see there are three unique colors. Let's say you would like to find out how many unique elements are in a column of a data frame.



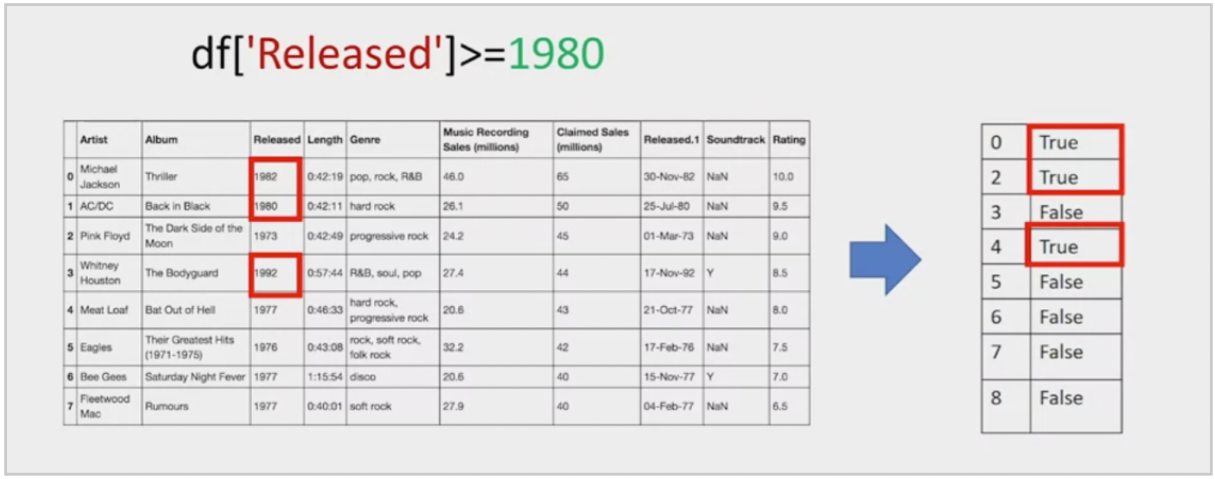
This may be much more difficult because instead of 13 elements, you may have millions. Pandas has the method `unique` to determine the unique elements in a column of a data frame.

Lets say we would like to determine the unique year of the albums in the data set. We enter the name of the data frame, then enter the name of the column released within

brackets. Then we apply the method unique. The result is all of the unique elements in the column released.



Let's say we would like to create a new database consisting of songs from the 1980s and after. We can look at the column released for songs made after 1979, then select the corresponding columns. We can accomplish this within one line of code in Pandas. But let's break up the steps. We can use the inequality operators for the entire data frame in Pandas. The result is a series of Boolean values.



For our case, we simply specify the column released and the inequality for the albums after 1979. The result is a series of Boolean values. The result is true when the condition is true and false otherwise. We can select the specified columns in one line.

	Artist	Album	Released	Length	Genre	Music Recording Sales (millions)	Claimed Sales (millions)	Released.1	Soundtrack	Rating
0	Michael Jackson	Thriller	1982	0:42:19	pop, rock, R&B	46.0	65	30-Nov-82	NaN	10.0
1	AC/DC	Back in Black	1980	0:42:11	hard rock	26.1	50	25-Jul-80	NaN	9.5
2	Pink Floyd	The Dark Side of the Moon	1973	0:42:49	progressive rock	24.2	45	01-Mar-73	NaN	9.0
3	Whitney Houston	The Bodyguard	1992	0:57:44	R&B, soul, pop	27.4	44	17-Nov-92	Y	8.5
4	Meat Loaf	Bat Out of Hell	1977	0:46:33	hard rock, progressive rock	20.6	43	21-Oct-77	NaN	8.0
5	Eagles	Their Greatest Hits (1971-1975)	1976	0:43:08	rock, soft rock, folk rock	32.2	42	17-Feb-76	NaN	7.5
6	Bee Gees	Saturday Night Fever	1977	1:15:54	disco	20.6	40	15-Nov-77	Y	7.0
7	Fleetwood Mac	Rumours	1977	0:40:01	soft rock	27.9	40	04-Feb-77	NaN	6.5

We simply use the data frames names and square brackets we placed the previously mentioned inequality and assign it to the variable df1. We now have a new data frame, where each album was released after 1979.

	Artist	Album	Released	Length	Genre	Music Recording Sales (millions)	Claimed Sales (millions)	Released.1	Soundtrack	Rating
0	Michael Jackson	Thriller	1982	00:42:19	pop, rock, R&B	46.0	65	1982-11-30	NaN	10.0
1	AC/DC	Back in Black	1980	00:42:11	hard rock	26.1	50	1980-07-25	NaN	9.5
3	Whitney Houston	The Bodyguard	1992	00:57:44	R&B, soul, pop	27.4	44	1992-11-17	Y	8.5

We can save the new data frame using the method to_csv. The argument is the name of the csv file. Make sure you include a.csv extension.

There are other functions to save the data frame in other formats.

Pandas - Lab

Objectives

After completing this lab you will be able to:

- Use Pandas to access and view data

Table of Contents

About the Dataset

The table has one row for each album and several columns.

- `artist` : Name of the artist
- `album` : Name of the album
- `released_year` : Year the album was released
- `length_min_sec` : Length of the album (hours, minutes, seconds)
- `genre` : Genre of the album
- `music_recording_sales_millions` : Music recording sales (millions in USD) on SONG://DATABASE
- `claimed_sales_millions` : Album's claimed sales (millions in USD) on SONG://DATABASE
- `date_released` : Date on which the album was released
- `soundtrack` : Indicates if the album is the movie soundtrack (Y) or (N)
- `rating_of_friends` : Indicates the rating from your friends from 1 to 10

You can see the dataset here:

Artist	Album	Released	Length	Genre	Music recording sales (millions)	Claimed sales (millions)	Released	Soundtrack	Rating (friends)
Michael Jackson	Thriller	1982	00:42:19	Pop, rock, R&B	46	65	30-Nov-82		10.0
AC/DC	Back in Black	1980	00:42:11	Hard rock	26.1	50	25-Jul-80		8.5
Pink Floyd	The Dark Side of the Moon	1973	00:42:49	Progressive rock	24.2	45	01-Mar-73		9.5
Whitney Houston	The Bodyguard	1992	00:57:44	Soundtrack/R&B, soul, pop	26.1	50	25-Jul-80	Y	7.0
Meat Loaf	Bat Out of Hell	1977	00:46:33	Hard rock, progressive rock	20.6	43	21-Oct-77		7.0
Eagles	Their Greatest Hits (1971-1975)	1976	00:43:08	Rock, soft rock, folk rock	32.2	42	17-Feb-76		9.5
Bee Gees	Saturday Night Fever	1977	1:15:54	Disco	20.6	40	15-Nov-77	Y	9.0
Fleetwood Mac	Rumours	1977	00:40:01	Soft rock	27.9	40	04-Feb-77		9.5

Introduction of `Pandas`

```
# Dependency needed to install file
!pip3 install xlrd
```

```
[33mDEPRECATION: Configuring installation scheme with distutils config files is deprecated and will no longer work in the near future. If you are using a Homebrew or Linuxbrew Python, please see discussion at https://github.com/Homebrew/homebrew-core/issues/76621
Requirement already satisfied: xlrd in /opt/homebrew/lib/python3.9/site-packages (2.0.1)
```

```
# Import required library
import pandas as pd
```

After the import command, we now have access to a large number of pre-built classes and functions. This assumes the library is installed; in our lab environment all the necessary libraries are installed. One way pandas allows you to work with data is a dataframe. Let's go through the process to go from a comma separated values (**.csv**) file to a dataframe. This variable `csv_path` stores the path of the **.csv**, that is used as an argument to the `read_csv` function. The result is stored in the object `df`, this is a common short form used for a variable referring to a Pandas dataframe.

```
# Read data from CSV file
csv_path = 'https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBMDeveloperSkillsNetwork-PY0101EN-SkillsNetwork/labs/Module%204/data/TopSellingAlbums.csv'
df = pd.read_csv(csv_path)
```

We can use the method `head()` to examine the first five rows of a dataframe:

```
# Print first five rows of the dataframe

df.head()
```

	Artist	Album	Released	Length	Genre	Music Recording Sales (millions)	Claimed Sales (millions)	Released.1	Soundtrack	Rating
0	Michael Jackson	Thriller	1982	00:42:19	pop, rock, R&B	46.0	65	1982-11-30	NaN	10.0
1	AC/DC	Back in Black	1980	00:42:11	hard rock	26.1	50	1980-07-25	NaN	9.5
2	Pink Floyd	The Dark Side of the Moon	1973	00:42:49	progressive rock	24.2	45	1973-03-01	NaN	9.0
3	Whitney Houston	The Bodyguard	1992	00:57:44	R&B, soul, pop	27.4	44	1992-11-17	Y	8.5
4	Meat Loaf	Bat Out of Hell	1977	00:46:33	hard rock, progressive rock	20.6	43	1977-10-21	NaN	8.0
5	Eagles	Their Greatest Hits (1971-1975)	1976	00:43:08	rock, soft rock, folk rock	32.2	42	1976-02-17	NaN	7.5

```
df.loc[4,'Length']
```

'0:46:33'

We use the path of the excel file and the function `read_excel`. The result is a data frame as before:

```
# Read data from Excel File and print the first five rows

xlsx_path = 'https://s3-api.us-geo.objectstorage.softlayer.net/cf-courses-data/CognitiveClass/PY0101EN/Chapter%204/Datasets/TopSellingAlbums.xlsx'
```

```
df = pd.read_excel(xlsx_path)
df.head(6)
```

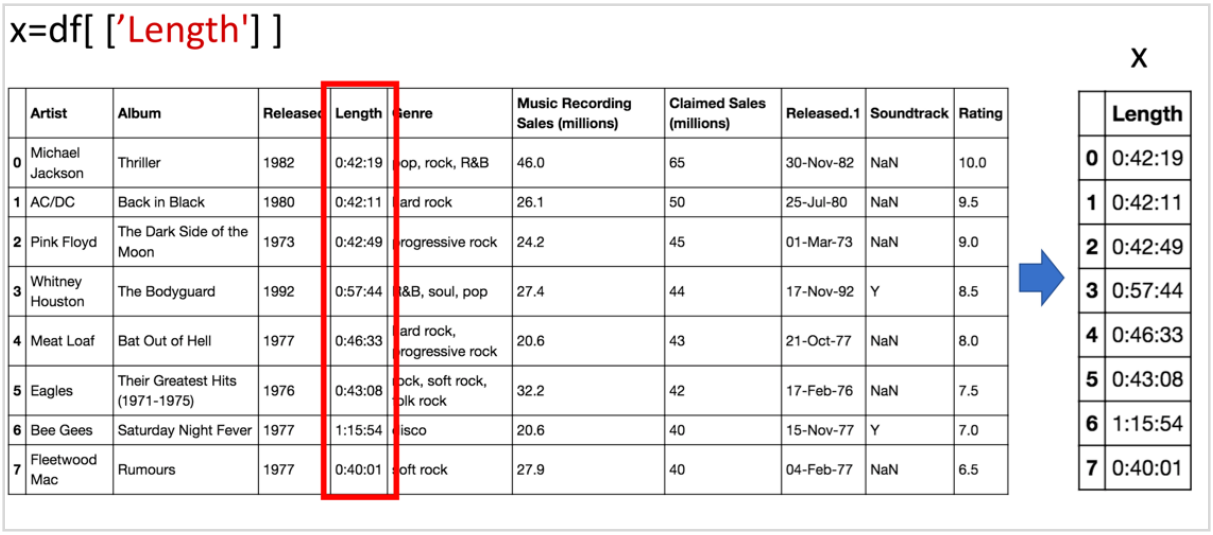
	Artist	Album	Released	Length	Genre	Music Recording Sales (millions)	Claimed Sales (millions)	Released.1	Soundtrack	Rating
0	Michael Jackson	Thriller	1982	00:42:19	pop, rock, R&B	46.0	65	1982-11-30	NaN	10.0
1	AC/DC	Back in Black	1980	00:42:11	hard rock	26.1	50	1980-07-25	NaN	9.5
2	Pink Floyd	The Dark Side of the Moon	1973	00:42:49	progressive rock	24.2	45	1973-03-01	NaN	9.0
3	Whitney Houston	The Bodyguard	1992	00:57:44	R&B, soul, pop	27.4	44	1992-11-17	Y	8.5
4	Meat Loaf	Bat Out of Hell	1977	00:46:33	hard rock, progressive rock	20.6	43	1977-10-21	NaN	8.0
5	Eagles	Their Greatest Hits (1971-1975)	1976	00:43:08	rock, soft rock, folk rock	32.2	42	1976-02-17	NaN	7.5

We can access the column **Length** and assign it a new dataframe **x**:

```
# Access to the column Length
x = df[['Length']]
x
```

	Length
0	00:42:19
1	00:42:11
2	00:42:49
3	00:57:44
4	00:46:33
5	00:43:08
6	01:15:54
7	00:40:01

The process is shown in the figure:



	Artist	Album	Released	Length	Genre	Music Recording Sales (millions)	Claimed Sales (millions)	Released.1	Soundtrack	Rating
0	Michael Jackson	Thriller	1982	0:42:19	Pop, rock, R&B	46.0	65	30-Nov-82	NaN	10.0
1	AC/DC	Back in Black	1980	0:42:11	Hard rock	26.1	50	25-Jul-80	NaN	9.5
2	Pink Floyd	The Dark Side of the Moon	1973	0:42:49	Progressive rock	24.2	45	01-Mar-73	NaN	9.0
3	Whitney Houston	The Bodyguard	1992	0:57:44	R&B, soul, pop	27.4	44	17-Nov-92	Y	8.5
4	Meat Loaf	Bat Out of Hell	1977	0:46:33	Hard rock, Progressive rock	20.6	43	21-Oct-77	NaN	8.0
5	Eagles	Their Greatest Hits (1971-1975)	1976	0:43:08	Rock, soft rock, Folk rock	32.2	42	17-Feb-76	NaN	7.5
6	Bee Gees	Saturday Night Fever	1977	1:15:54	Disco	20.6	40	15-Nov-77	Y	7.0
7	Fleetwood Mac	Rumours	1977	0:40:01	Soft rock	27.9	40	04-Feb-77	NaN	6.5

	Length
0	0:42:19
1	0:42:11
2	0:42:49
3	0:57:44
4	0:46:33
5	0:43:08
6	1:15:54
7	0:40:01

Viewing Data and Accessing Data

You can also get a column as a series. You can think of a Pandas series as a 1-D dataframe. Just use one bracket:

```
# Get the column as a series
x = df['Length']
x
```

```
0    00:42:19
1    00:42:11
2    00:42:49
3    00:57:44
4    00:46:33
5    00:43:08
6    01:15:54
7    00:40:01
Name: Length, dtype: object
```

You can also get a column as a dataframe. For example, we can assign the column **Artist**:

```
# Get the column as a dataframe
```

```
x = df[['Artist']]
type(x)
```

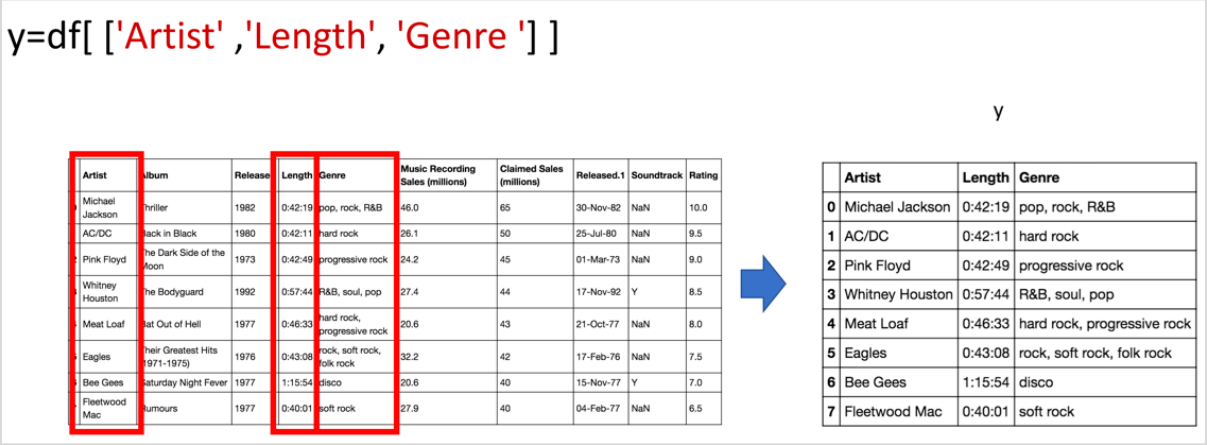
pandas.core.frame.DataFrame

You can do the same thing for multiple columns; we just put the dataframe name, in this case, `df`, and the name of the multiple column headers enclosed in double brackets. The result is a new dataframe comprised of the specified columns:

```
# Access to multiple columns
y = df[['Artist','Length','Genre']]
y
```

	Artist	Length	Genre
0	Michael Jackson	00:42:19	pop, rock, R&B
1	AC/DC	00:42:11	hard rock
2	Pink Floyd	00:42:49	progressive rock
3	Whitney Houston	00:57:44	R&B, soul, pop
4	Meat Loaf	00:46:33	hard rock, progressive rock
5	Eagles	00:43:08	rock, soft rock, folk rock
6	Bee Gees	01:15:54	disco
7	Fleetwood Mac	00:40:01	soft rock

The process is shown in the figure:



One way to access unique elements is the `iloc` method, where you can access the 1st row and the 1st column as follows:

```
# Access the value on the first row and the first column
df.iloc[0, 0]
```

'Michael Jackson'

You can access the 2nd row and the 1st column as follows:

```
# Access the value on the second row and the first column
df.iloc[1,0]
```

'AC/DC'

You can access the 1st row and the 3rd column as follows:

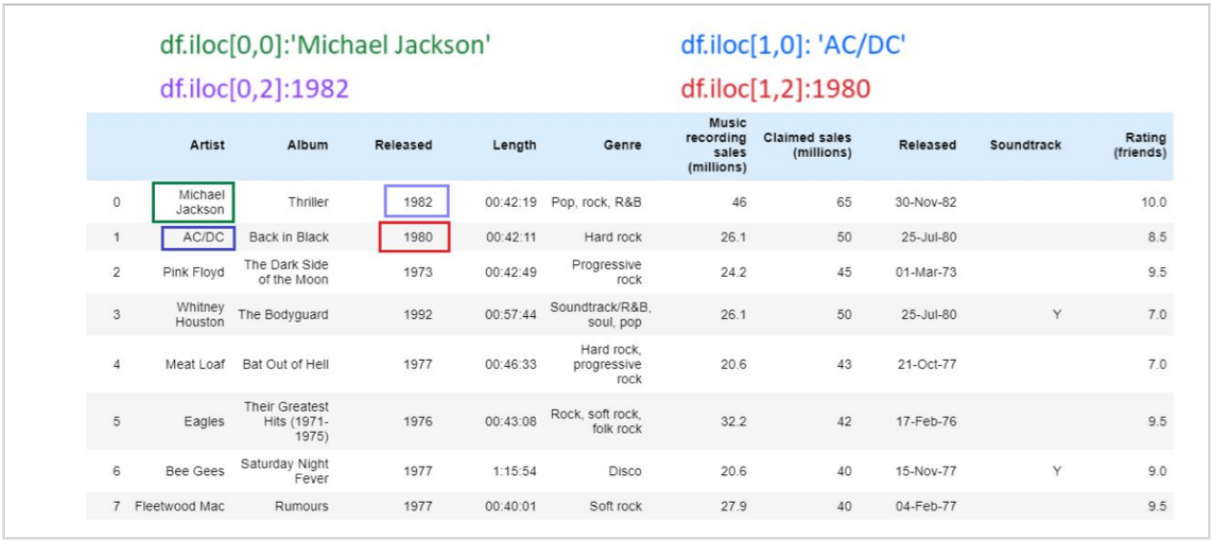
```
# Access the value on the first row and the third column
df.iloc[0,2]
```

1982

```
# Access the value on the second row and the third column
df.iloc[1,2]
```

1980

This is shown in the following image



	Artist	Album	Released	Length	Genre	Music recording sales (millions)	Claimed sales (millions)	Released	Soundtrack	Rating (friends)
0	Michael Jackson	Thriller	1982	00:42:19	Pop, rock, R&B	46	65	30-Nov-82		10.0
1	AC/DC	Back in Black	1980	00:42:11	Hard rock	26.1	50	25-Jul-80		8.5
2	Pink Floyd	The Dark Side of the Moon	1973	00:42:49	Progressive rock	24.2	45	01-Mar-73		9.5
3	Whitney Houston	The Bodyguard	1992	00:57:44	Soundtrack/R&B, soul, pop	26.1	50	25-Jul-80	Y	7.0
4	Meat Loaf	Bat Out of Hell	1977	00:46:33	Hard rock, progressive rock	20.6	43	21-Oct-77		7.0
5	Eagles	Their Greatest Hits (1971-1975)	1976	00:43:08	Rock, soft rock, folk rock	32.2	42	17-Feb-76		9.5
6	Bee Gees	Saturday Night Fever	1977	1:15:54	Disco	20.6	40	15-Nov-77	Y	9.0
7	Fleetwood Mac	Rumours	1977	00:40:01	Soft rock	27.9	40	04-Feb-77		9.5

You can access the column using the name as well, the following are the same as above:

```
# Access the column using the name
df.loc[0, 'Artist']
```

'Michael Jackson'

```
df.iloc[2:6, 1]
```

```
2           The Dark Side of the Moon
3                        The Bodyguard
4                         Bat Out of Hell
5    Their Greatest Hits (1971–1975)
Name: Album, dtype: object
```

```
# Access the column using the name
df.loc[1, 'Artist']
```

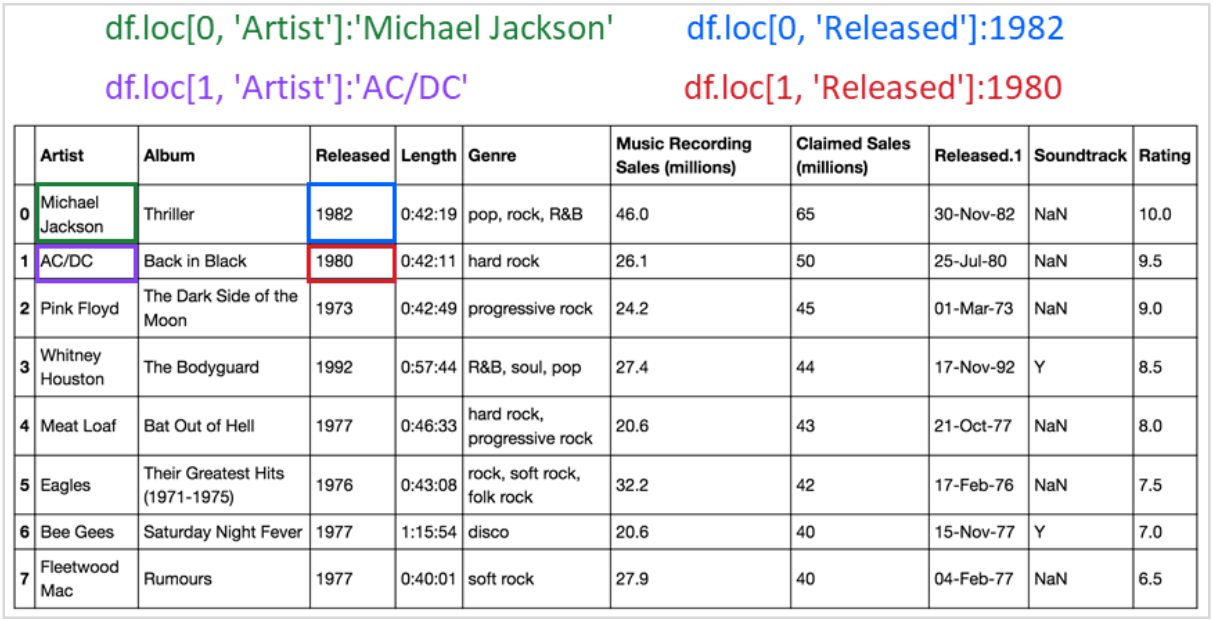
'AC/DC'

```
# Access the column using the name
df.loc[0, 'Released']
```

1982

```
# Access the column using the name
df.loc[1, 'Released']
```

1980



	Artist	Album	Released	Length	Genre	Music Recording Sales (millions)	Claimed Sales (millions)	Released.1	Soundtrack	Rating
0	Michael Jackson	Thriller	1982	0:42:19	pop, rock, R&B	46.0	65	30-Nov-82	NaN	10.0
1	AC/DC	Back in Black	1980	0:42:11	hard rock	26.1	50	25-Jul-80	NaN	9.5
2	Pink Floyd	The Dark Side of the Moon	1973	0:42:49	progressive rock	24.2	45	01-Mar-73	NaN	9.0
3	Whitney Houston	The Bodyguard	1992	0:57:44	R&B, soul, pop	27.4	44	17-Nov-92	Y	8.5
4	Meat Loaf	Bat Out of Hell	1977	0:46:33	hard rock, progressive rock	20.6	43	21-Oct-77	NaN	8.0
5	Eagles	Their Greatest Hits (1971-1975)	1976	0:43:08	rock, soft rock, folk rock	32.2	42	17-Feb-76	NaN	7.5
6	Bee Gees	Saturday Night Fever	1977	1:15:54	disco	20.6	40	15-Nov-77	Y	7.0
7	Fleetwood Mac	Rumours	1977	0:40:01	soft rock	27.9	40	04-Feb-77	NaN	6.5

You can perform `slicing` using both the index and the name of the column:

```
# Slicing the dataframe
df.iloc[0:2, 0:3]
```

	Artist	Length	Genre
0	Michael Jackson	00:42:19	pop, rock, R&B
1	AC/DC	00:42:11	hard rock
2	Pink Floyd	00:42:49	progressive rock
3	Whitney Houston	00:57:44	R&B, soul, pop
4	Meat Loaf	00:46:33	hard rock, progressive rock
5	Eagles	00:43:08	rock, soft rock, folk rock
6	Bee Gees	01:15:54	disco
7	Fleetwood Mac	00:40:01	soft rock

```
z=df.iloc[0:2, 0:3]
```

	Artist	Album	Released	Length	Genre	Music Recording Sales (millions)	Claimed Sales (millions)	Released.1	Soundtrack	Rating
0	Michael Jackson	Thriller	1982	0:42:19	pop, rock, R&B	46.0	65	30-Nov-82	NaN	10.0
1	AC/DC	Back in Black	1980	0:42:11	hard rock	26.1	50	25-Jul-80	NaN	9.5
2	Pink Floyd	The Dark Side of the Moon	1973	0:42:49	progressive rock	24.2	45	01-Mar-73	NaN	9.0
3	Whitney Houston	The Bodyguard	1992	0:57:44	R&B, soul, pop	27.4	44	17-Nov-92	Y	8.5
4	Meat Loaf	Bat Out of Hell	1977	0:46:33	hard rock, progressive rock	20.6	43	21-Oct-77	NaN	8.0
5	Eagles	Their Greatest Hits (1971-1975)	1976	0:43:08	rock, soft rock, folk rock	32.2	42	17-Feb-76	NaN	7.5
6	Bee Gees	Saturday Night Fever	1977	1:15:54	disco	20.6	40	15-Nov-77	Y	7.0
7	Fleetwood Mac	Rumours	1977	0:40:01	soft rock	27.9	40	04-Feb-77	NaN	6.5

z

	Artist	Album	Released
0	Michael Jackson	Thriller	1982
1	AC/DC	Back in Black	1980

```
# Slicing the dataframe using name
df.loc[0:2, 'Artist':'Released']
```

	Artist	Album	Released
0	Michael Jackson	Thriller	1982
1	AC/DC	Back in Black	1980
2	Pink Floyd	The Dark Side of the Moon	1973

	Artist	Album	Released	Length	Genre	Music Recording Sales (millions)	Claimed Sales (millions)	Released.1	Soundtrack	Rating
0	Michael Jackson	Thriller	1982	0:42:19	pop, rock, R&B	46.0	65	30-Nov-82	NaN	10.0
1	AC/DC	Back in Black	1980	0:42:11	hard rock	26.1	50	25-Jul-80	NaN	9.5
2	Pink Floyd	The Dark Side of the Moon	1973	0:42:49	progressive rock	24.2	45	01-Mar-73	NaN	9.0
3	Whitney Houston	The Bodyguard	1992	0:57:44	R&B, soul, pop	27.4	44	17-Nov-92	Y	8.5
4	Meat Loaf	Bat Out of Hell	1977	0:46:33	hard rock, progressive rock	20.6	43	21-Oct-77	NaN	8.0
5	Eagles	Their Greatest Hits (1971-1975)	1976	0:43:08	rock, soft rock, folk rock	32.2	42	17-Feb-76	NaN	7.5
6	Bee Gees	Saturday Night Fever	1977	1:15:54	disco	20.6	40	15-Nov-77	Y	7.0
7	Fleetwood Mac	Rumours	1977	0:40:01	soft rock	27.9	40	04-Feb-77	NaN	6.5

z

	Artist	Album	Released
0	Michael Jackson	Thriller	1982
1	AC/DC	Back in Black	1980
2	Pink Floyd	The Dark Side of the Moon	1973

Quiz on DataFrame

Use a variable `q` to store the column **Rating** as a dataframe

```
# Write your code below and press Shift+Enter to execute
q = df[['Rating']]
q
```

	Rating
0	10.0
1	9.5
2	9.0
3	8.5
4	8.0
5	7.5
6	7.0
7	6.5

Assign the variable `q` to the dataframe that is made up of the column **Released** and **Artist**:

```
# Write your code below and press Shift+Enter to execute
q = df[['Rating','Artist']]
q
```

	Rating
0	10.0
1	9.5
2	9.0
3	8.5
4	8.0
5	7.5
6	7.0
7	6.5

Access the 2nd row and the 3rd column of `df`:

```
# Write your code below and press Shift+Enter to execute
df.iloc[1,2]
```

1980

Use the following list to convert the dataframe index `df` to characters and assign it to `df_new`; find the element corresponding to the row index `a` and column `'Artist'`. Then select the rows `a` through `d` for the column `'Artist'`

```
new_index=['a','b','c','d','e','f','g','h']
df_new = df
df_new.index = new_index
df_new.loc['a','Artist']
df_new.loc['a':'d','Artist']
```

```
a    Michael Jackson
b              AC/DC
c         Pink Floyd
d    Whitney Houston
Name: Artist, dtype: object
```

Numpy in Python

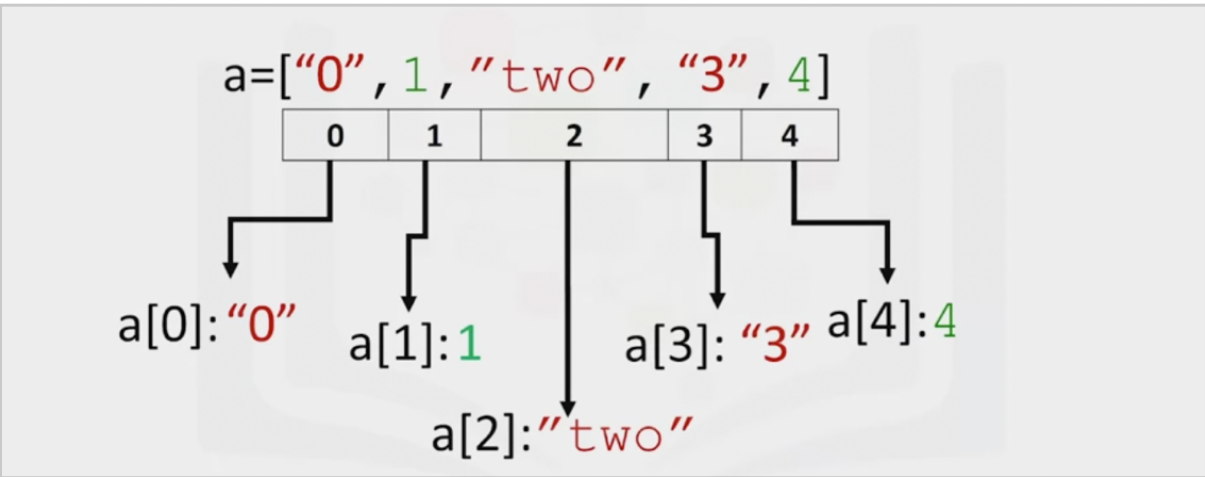
One Dimensional Numpy

In this video we will be covering numpy in 1D, in particular **ND arrays**. Numpy is a library for scientific computing. It has many useful functions. There are many other advantages like speed and memory. Numpy is also the basis for pandas. So check out our pandas video.

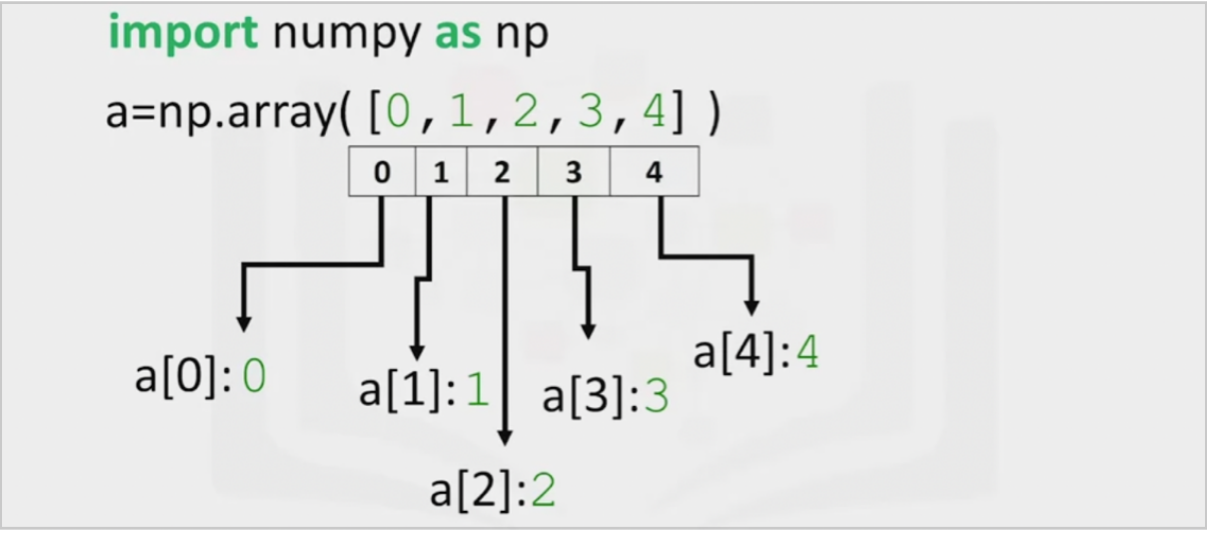
In this video we will be covering:

- the basics and array creation,
- indexing and slicing,
- basic operations,
- universal functions.

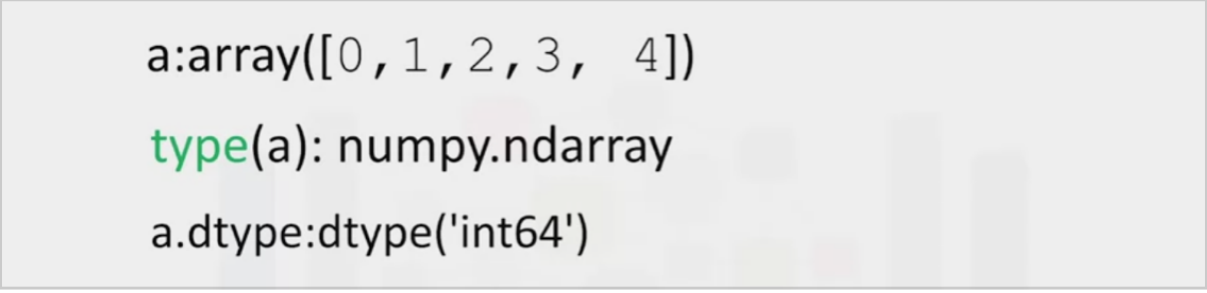
Let's go over how to create a numpy array. A Python `list` is a container that allows you to store and access data. Each element is associated with an index. We can access each element using a square bracket as follows.



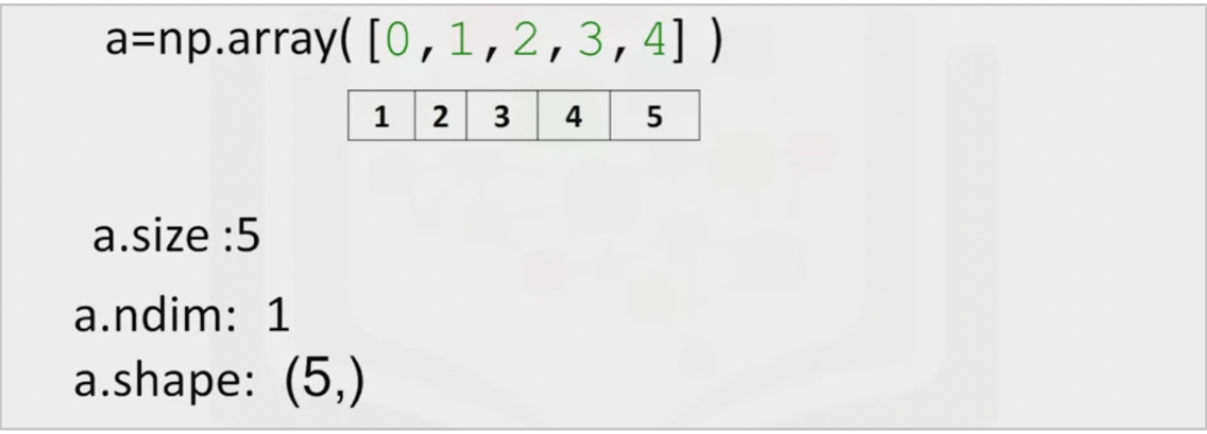
A numpy array or ND array is similar to a list. It's usually fixed in size and each element is of the same type, in this case integers. We can cast a list to a numpy array by first importing numpy. We then cast the list as follows; we can access the data via an index. As with the list, we can access each element with an integer and a square bracket.



The value of `a` is stored as follows. If we check the type of the array we get, numpy.ndarray. As numpy arrays contain data of the same type, we can use the attribute dtype to obtain the data type of the array's elements. In this case a 64-bit integer.



Let's review some basic array attributes using the array a. The attribute `size` is the number of elements in the array. As there are five elements the result is five. The next two attributes will make more sense when we get to higher dimensions, but let's review them. The attribute `ndim` represents the number of array dimensions or the rank of the array, in this case one. The attribute `shape` is a tuple of integers indicating the size of the array in each dimension.



We can create a numpy array with real numbers. When we check the type of the array, we get numpy.ndarray. If we examine the attribute D type, we see float64 as the elements are

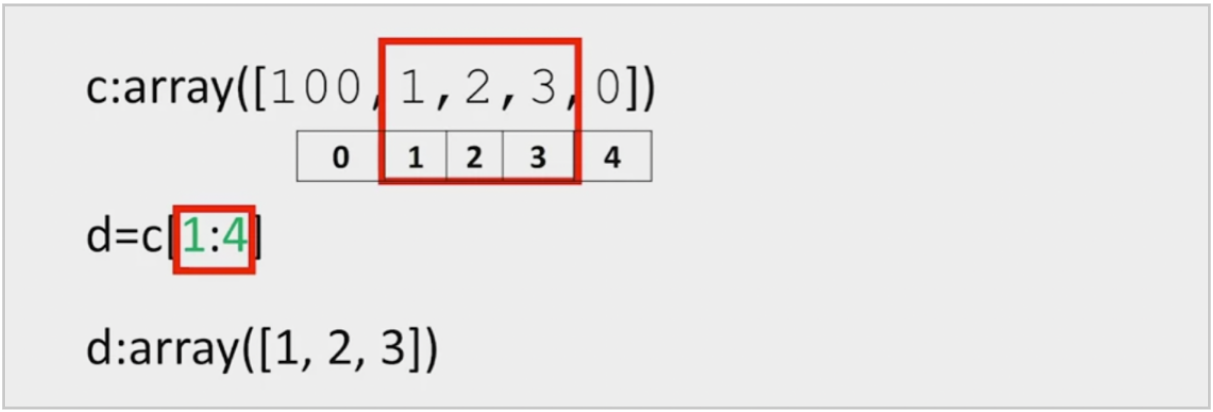
not integers. There were many other attributes, check out numpy.org.

```
b=np.array([3.1,11.02,6.2, 213.2,5.2])

type(b): numpy.ndarray
b.dtype:dtype('float64')
```

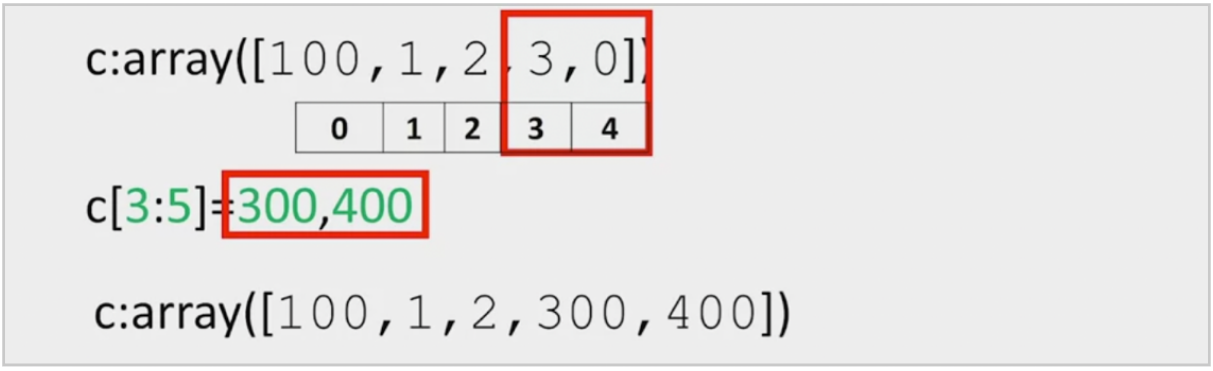
Let's review some indexing and slicing methods. We can change the first element of the array to 100 as follows. The array's first value is now 100. We can change the fifth element of the array as follows. The fifth element is now zero.

```
c=np.array([20,1,2,3, 4])
c:array([20,1,2,3, 4])

c[0]=100
c:array([100,1,2,3,4])

c[4]=0
c:array([100,1,2,3,0])
```

Like lists and tuples we can slice a NumPy array. The elements of the array correspond to the following index. We can select the elements from one to three and assign it to a new numpy array d as follows. The elements in d correspond to the index. Like lists, we do not count the element corresponding to the last index.



We can assign the corresponding indices to new values as follows. The array c now has new values. See the labs or numpy.org for more examples of what you can do with numpy.



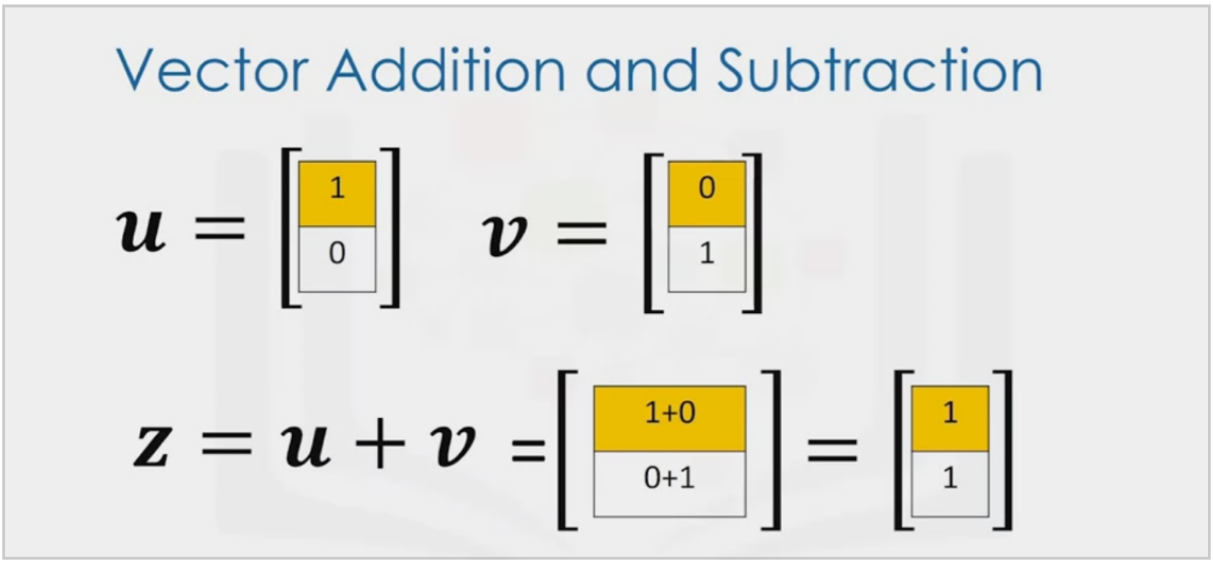
Basic Operations theory

Numpy makes it easier to do many operations that are commonly performed in data science. The same operations are usually computationally faster and require less memory in numpy compared to regular Python. Let's review some of these operations on one-dimensional arrays. We will look at many of the operations in the context of Euclidian vectors to make things more interesting.

Vector

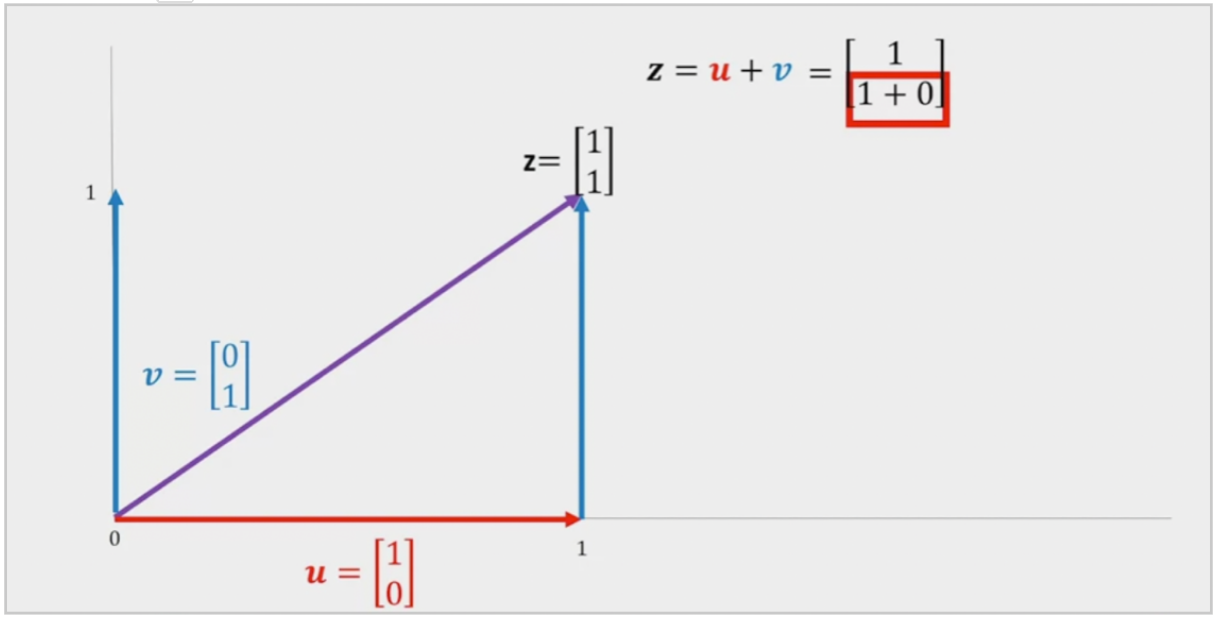
Addition

Vector **addition** is a widely used operation in data science. Consider the vector `u` with two elements, the elements are distinguished by the different colors. Similarly, consider the vector `v` with two components. In vector addition, we create a new vector in this case `z`. The first component of `z` is the addition of the first component of vectors `u` and `v`. Similarly, the second component is the sum of the second components of `u` and `v`. This new vector z is now a linear combination of the vector u and v.



Representing vector addition with line segment or arrows is helpful. The first vector is represented in red. The vector will point in the direction of the two components. The first component of the vector is one. As a result the arrow is offset one unit from the origin in the horizontal direction. The second component is zero, we represent this component in

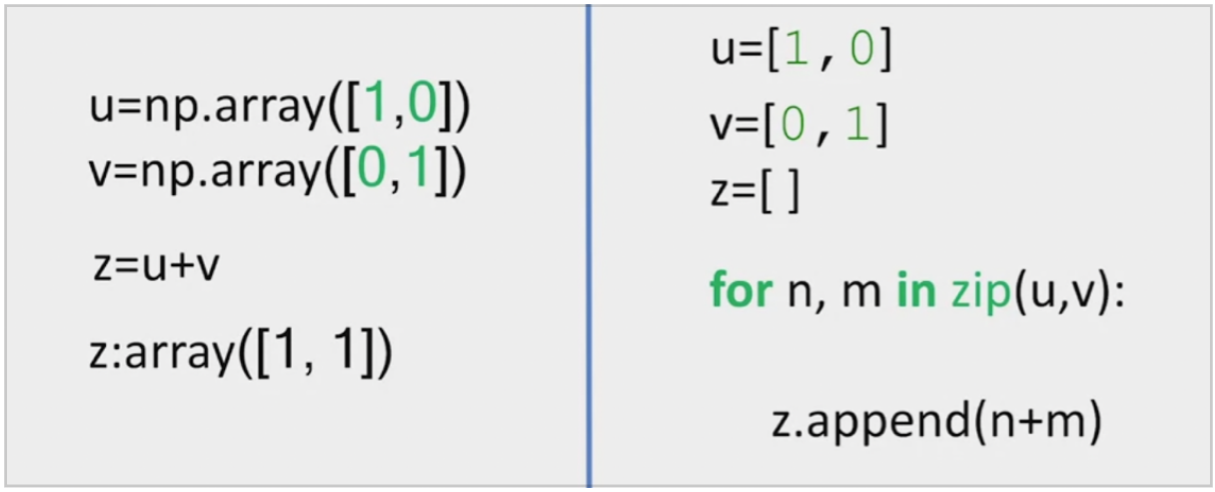
the vertical direction. As this component is zero, the vector does not point in the vertical direction. We represent the second vector in blue. The first component is zero, therefore the arrow does not point to the horizontal direction. The second component is one. As a result the vector points in the vertical direction one unit. When we add the vector `u` and `v`, we get the new vector `z`. We add the first component, this corresponds to the horizontal direction. We also add the second component. It's helpful to use the tip to tail method when adding vectors, placing the tail of the vector `v` on the tip of vector u. The new vector `z` is constructed by connecting the base of the first vector `u` with the tail of the second `v`.



The following three lines of code we'll add the two lists and place the result in the list z.

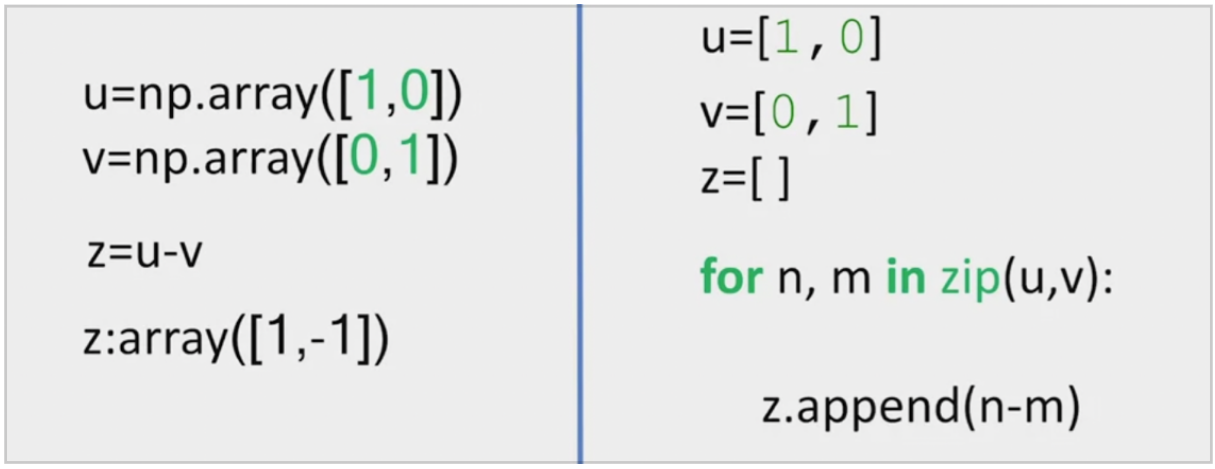
```
u=[1, 0]
v=[0, 1]

z=[ ]
for n, m in zip(u,v):

    z.append(n+m)
```

We can also perform vector addition with one line of NumPy code. It would require multiple lines to perform vector additions on two lists as shown on the right side of the screen. In addition, the numpy code will run much faster. This is important if you have lots of data.



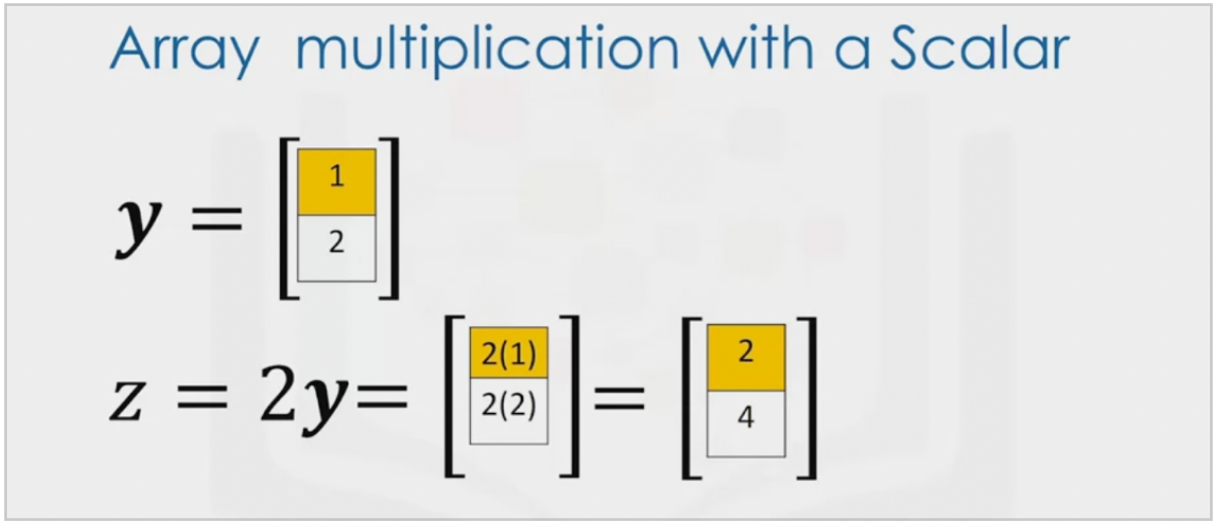
Subtraction

We can also perform vector **subtraction** by changing the addition sign to a subtraction sign. It would require multiple lines perform vector subtraction on two lists as shown on the right side of the screen.

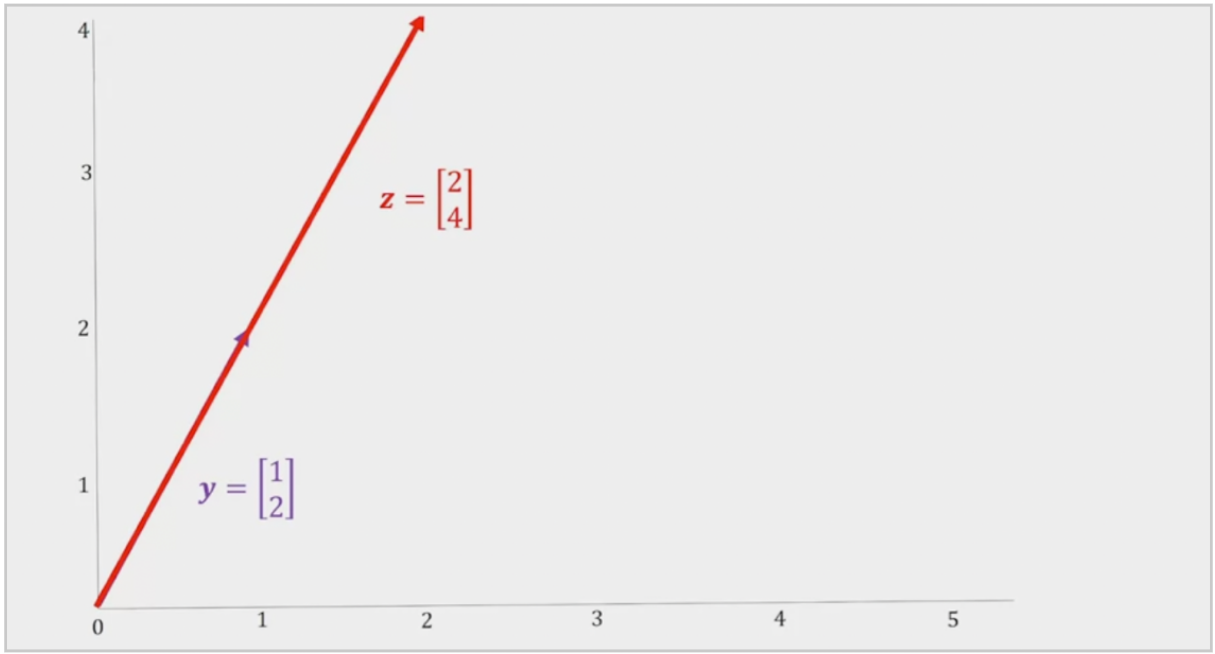


Multiplication with a scalar

Vector **multiplication with a scalar** is another commonly performed operation. Consider the vector y, each component is specified by a different color. We simply multiply the vector by a scalar value in this case two. Each component of the vector is multiplied by two, in this case each component is doubled.



We can use the line segment or arrows to visualize what's going on. The original vector y is in purple. After multiplying it by a scalar value of two, the vector is stretched out by two units as shown in red. The new vector is twice as long in each direction.



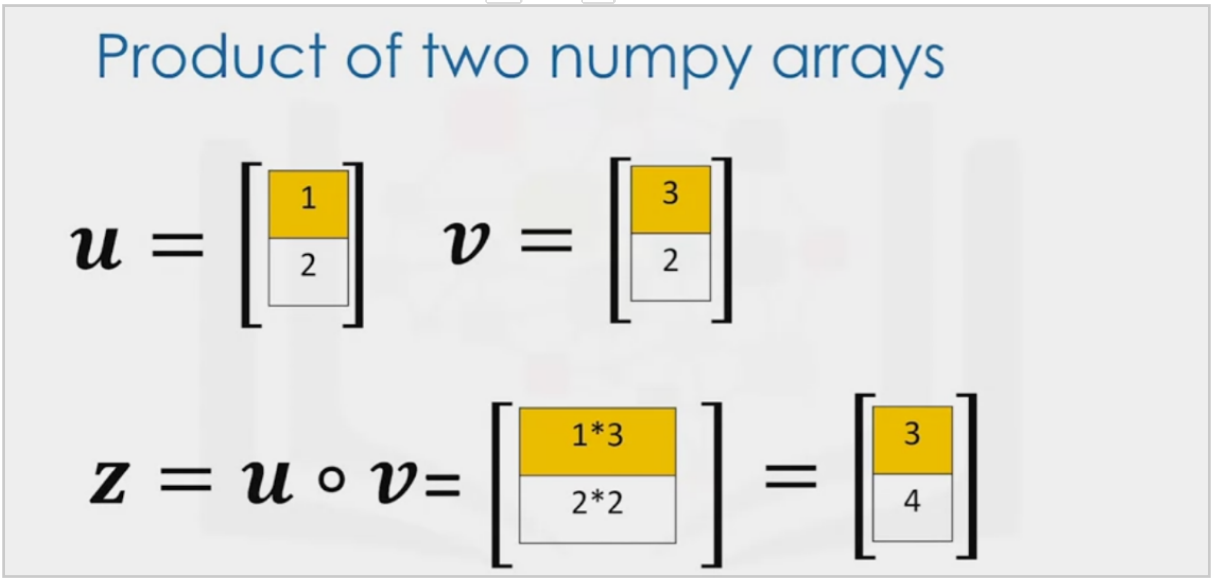
Vector multiplication with a scalar only requires one line of code using numpy. It would require multiple lines to perform the same task as shown with Python lists as shown on the right side of the screen. In addition, the operation would also be much slower.

```
y=np.array([1,2])

z=2*y

z:array([2,4])
```

```
y=[1,2]

z=[ ]

for n in y:

    z.append(2*n)
```

Hadamard product

Hadamard product is another widely used operation in data science. Consider the following two vectors, `u` and `v`. The Hadamard product of `u` and `v` is a new vector `z`. The first component of `z` is the product of the first element of `u` and `v`. Similarly, the second component is the product of the second element of `u` and `v`. The resultant vector consists of the entry wise product of `u` and `v`.



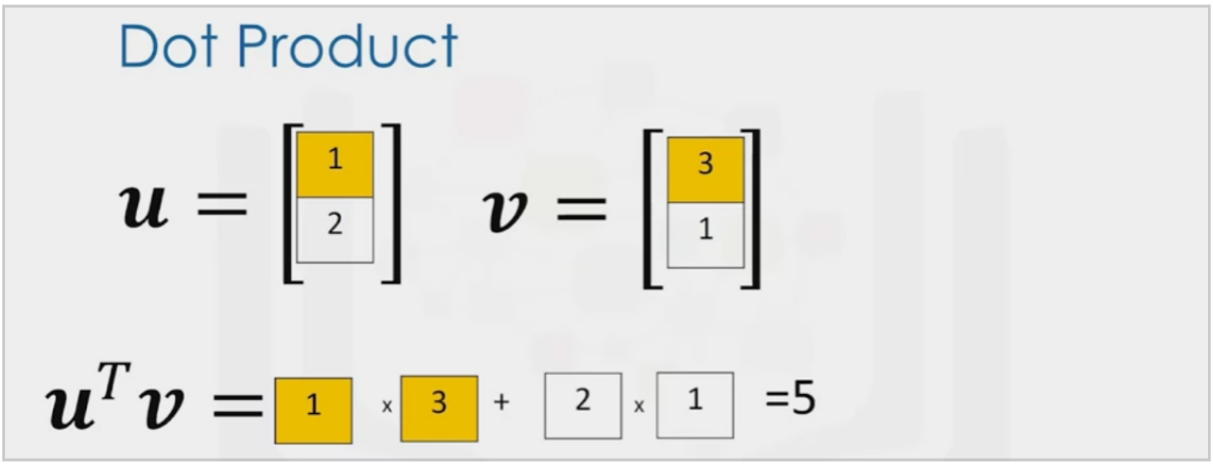
We can also perform hadamard product with one line of code in numpy. It would require multiple lines to perform hadamard product on two lists as shown on the right side of the screen.

```
u=np.array([1,2])
v=np.array([3,2])

z=u*v

z:array([3,  4])
```

```
u=[1, 2]
v=[3, 2]
z=[ ]

for n, m in zip(u,v):

    z.append(n*m)
```

Dot product (Prodotto scalare)

The **dot product** (*Prodotto Scalare*) is another widely used operation in data science. Consider the vector u and v, the dot product is a single number given by the following term and represents how similar two vectors are. We multiply the first component from v and u, we then multiply the second component and add the result together. The result is a number that represents how similar the two vectors are.



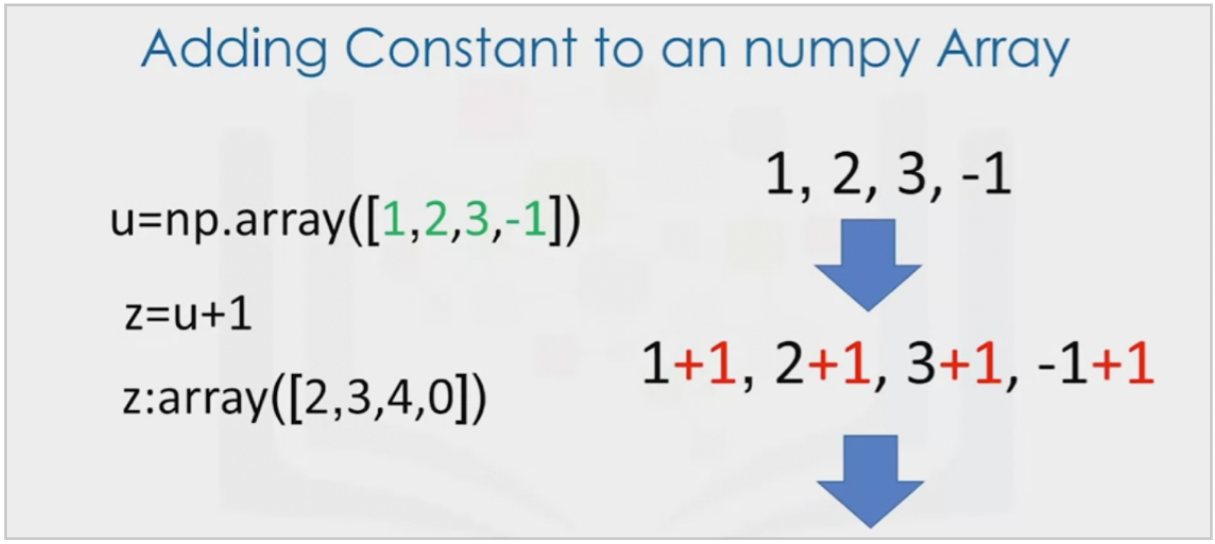
We can also perform dot product using the numpy function dot and assign it with the variable result as follows.

```
u=np.array([1,2])
v=np.array([3,1])

result =np.dot(u,v)

result :5
```

Broadcasting

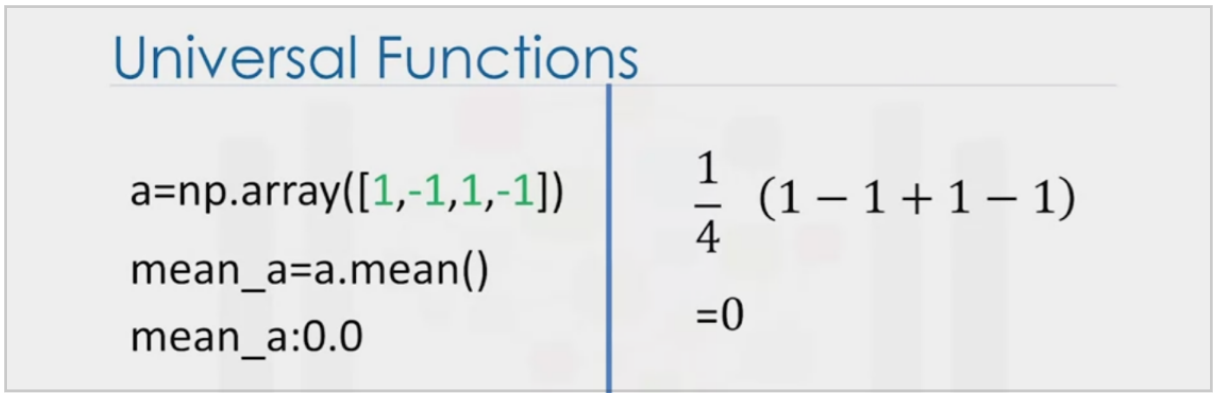
Consider the array u, the array contains the following elements. If we add a scalar value to

the array, numpy will add that value to each element. This property is known as **broadcasting**.



Universal Function

A **universal function** is a function that operates on ND arrays. We can apply a universal function to a numpy array. Consider the arrays a, we can calculate the mean or average value of all the elements in a using the method mean. This corresponds to the average of all the elements. In this case the result is zero.



There are many other functions. For example, consider the numpy arrays `b`. We can find the maximum value using the method five. We see the largest value is five, therefore the method max returns a five.

We can use numpy to create functions that map numpy arrays to new numpy arrays. Let's implement some code on the left side of the screen and use the right side of the

screen to demonstrate what's going on.

- We can access the value of pie in numpy as follows.
- We can create the following numpy array in radians. This array corresponds to the following vector.
- We can apply the function sin to the array x and assign the values to the array y. This applies the sin function to each element in the array, this corresponds to applying the sine function to each component of the vector.
- The result is a new array y, where each value corresponds to a sine function being applied to each element in the array x.

```
np.pi
x=np.array([ 0 , np.pi/2, np.pi ] )
y=np.sin(x)
y:array([ 0,1, 1.2e-16])
```

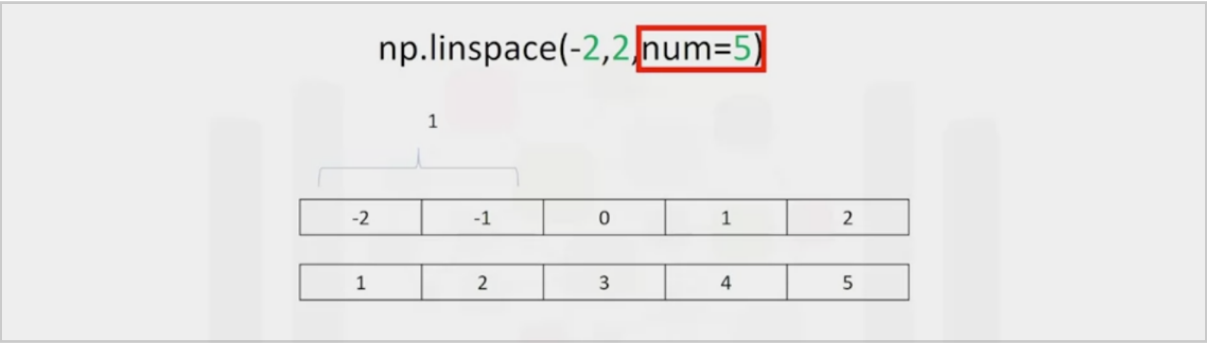
$$\pi$$

$$\boldsymbol{x} = [0, \frac{\pi}{2}, \pi]$$

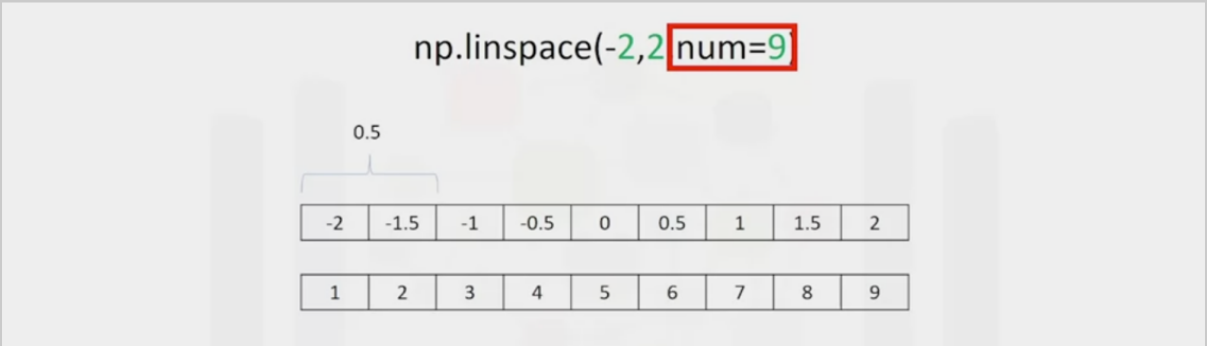
$$\boldsymbol{y} = [\sin(0), \sin(\frac{\pi}{2}), \sin(\pi)]$$

$$\boldsymbol{y} = [0,1,0]$$

A useful function for plotting mathematical functions is `line space`. Line space returns evenly spaced numbers over specified interval. We specify the starting point of the sequence, the ending point of the sequence. The parameter num indicates the number of samples to generate, in this case five. The space between samples is one.

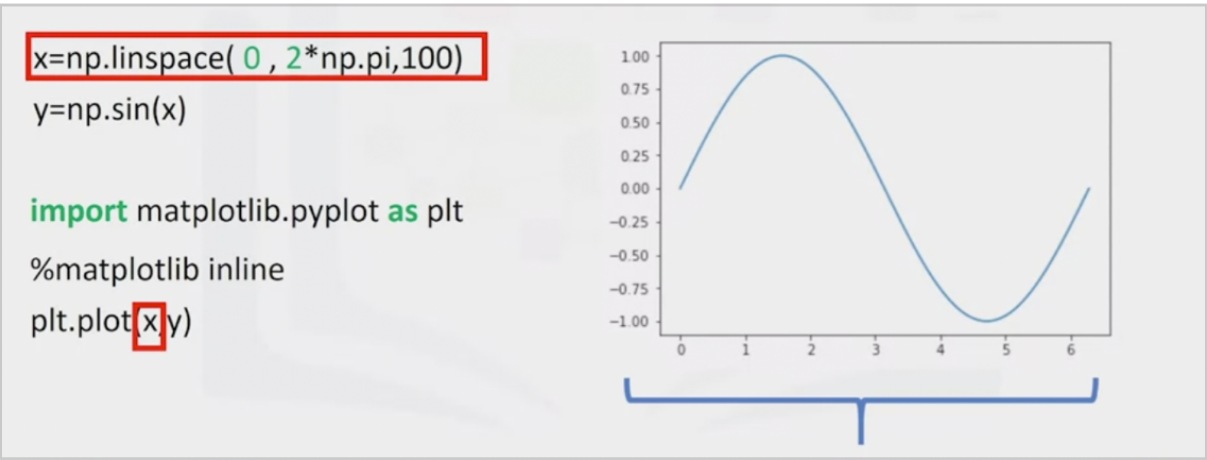


If we change the parameter num to nine, we get nine evenly spaced numbers over the integral from negative two to two. The result is the difference between subsequent samples is 0.5 as opposed to one as before.

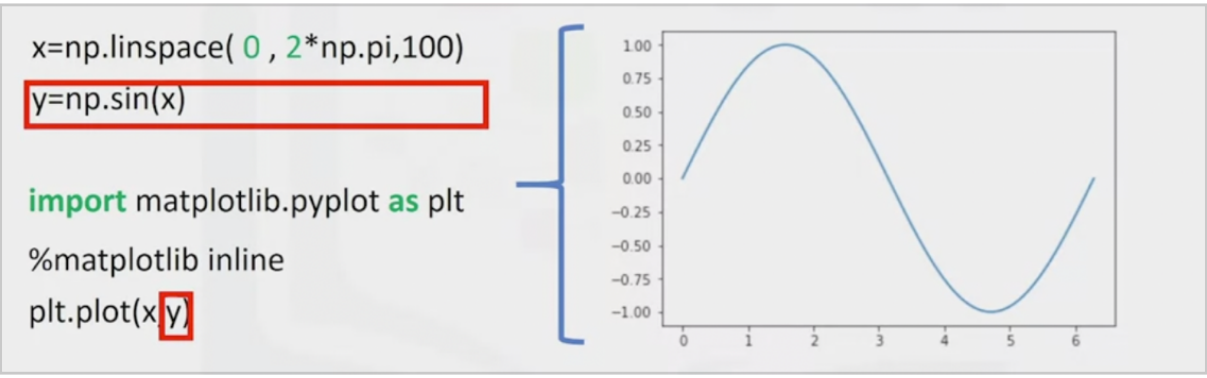


We can use the function line space to generate 100 evenly spaced samples from the interval zero to two pie. We can use the numpy function sin to map the array x to a new array y. We can import the library pyplot as plt to help us plot the function. As we are using a Jupiter notebook, we use the command matplotlib inline to display the plot. The following command plots a graph.

- The first input corresponds to the values for the horizontal or x-axis.



- The second input corresponds to the values for the vertical or y-axis.



There's a lot more you can do with numpy. Check out the labs and numpy.org for more.

One Dimensional Numpy - Lab

Objectives

After completing this lab you will be able to:

- Import and use the `numpy` library
- Perform operations with `numpy`

Table of Contents

Preparation

```
# Import the libraries
import time
import sys
import numpy as np

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

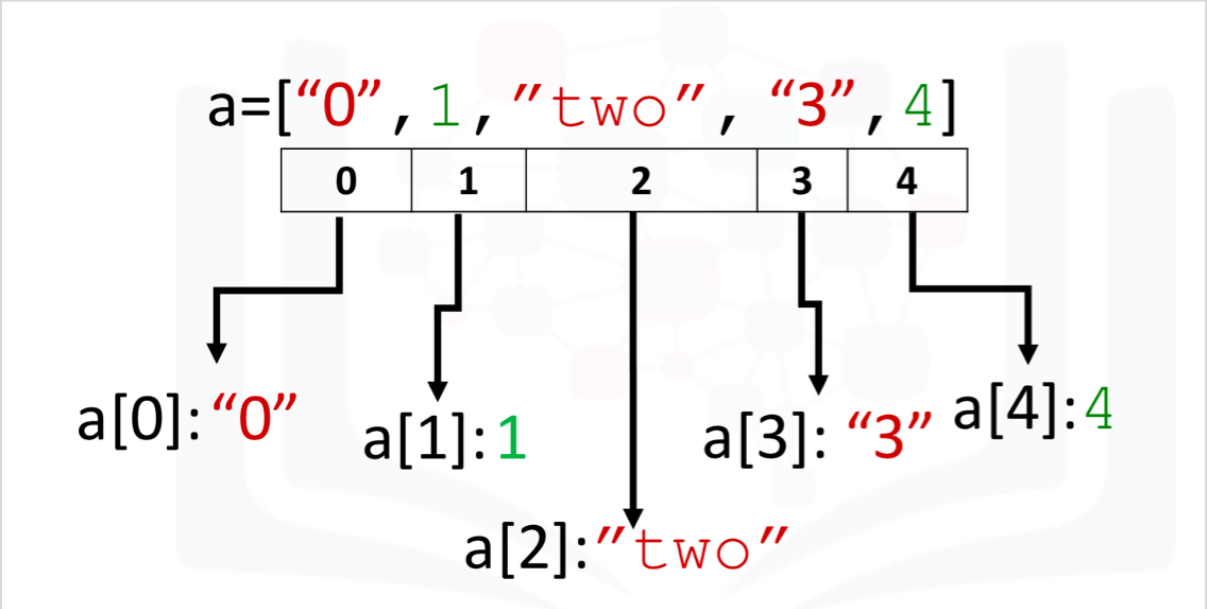
```
# Plotting functions
def Plotvec1(u, z, v):

    ax = plt.axes()
    ax.arrow(0, 0, *u, head_width=0.05, color='r', head_length=0.1)
    plt.text(*(u + 0.1), 'u')

    ax.arrow(0, 0, *v, head_width=0.05, color='b', head_length=0.1)
    plt.text(*(v + 0.1), 'v')
    ax.arrow(0, 0, *z, head_width=0.05, head_length=0.1)
```

```
    plt.text(*(z + 0.1), 'z')
    plt.ylim(-2, 2)
    plt.xlim(-2, 2)

def Plotvec2(a,b):
    ax = plt.axes()
    ax.arrow(0, 0, *a, head_width=0.05, color ='r', head_length=0.1)
    plt.text(*(a + 0.1), 'a')
    ax.arrow(0, 0, *b, head_width=0.05, color ='b', head_length=0.1)
    plt.text(*(b + 0.1), 'b')
    plt.ylim(-2, 2)
    plt.xlim(-2, 2)
```

Create a Python List as follows:

```
# Create a python list
a = ["0", 1, "two", "3", 4]
```

We can access the data via an index:



We can access each element using a square bracket as follows:

```
# Print each element
print("a[0]:", a[0])
print("a[1]:", a[1])
print("a[2]:", a[2])
print("a[3]:", a[3])
print("a[4]:", a[4])
```

```
a[0]: 0
a[1]: 1
a[2]: two
a[3]: 3
a[4]: 4
```

What is Numpy?

A numpy array is similar to a list. It's usually fixed in size and each element is of the same type. We can cast a list to a numpy array by first importing `numpy`:

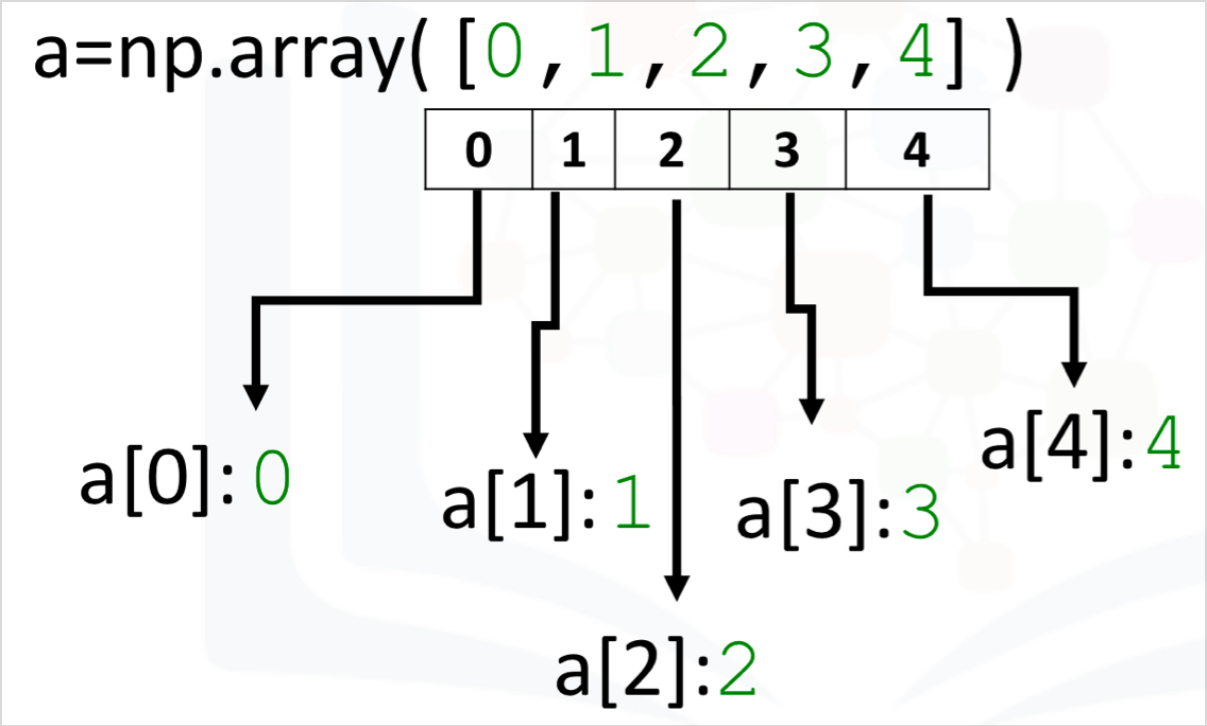
```
# import numpy library
import numpy as np
```

We then cast the list as follows:

```
# Create a numpy array
a = np.array([0, 1, 2, 3, 4])
a
```

```
array([0, 1, 2, 3, 4])
```

Each element is of the same type, in this case integers:



As with lists, we can access each element via a square bracket:

```
# Print each element
print("a[0]:", a[0])
print("a[1]:", a[1])
print("a[2]:", a[2])
print("a[3]:", a[3])
print("a[4]:", a[4])
```

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

Type

If we check the type of the array we get **numpy.ndarray**:

```
# Check the type of the array
```

```
type(a)
```

numpy.ndarray

As numpy arrays contain data of the same type, we can use the attribute "dtype" to obtain the data type of the array's elements. In this case, it's a 64-bit integer:

```
# Check the type of the values stored in numpy array
a.dtype
```

dtype('int64')

We can create a numpy array with real numbers:

```
# Create a numpy array
b = np.array([3.1, 11.02, 6.2, 213.2, 5.2])
```

When we check the type of the array we get **numpy.ndarray**:

```
# Check the type of array
type(b)
```

numpy.ndarray

If we examine the attribute `dtype` we see float 64, as the elements are not integers:

```
# Check the value type
b.dtype
```

dtype('int64')

Assign value

We can change the value of the array. Consider the array `c`:

```
# Create numpy array
c = np.array([20, 1, 2, 3, 4])
c
```

```
array([20,  1,  2,  3,  4])
```

We can change the first element of the array to 100 as follows:

```
# Assign the first element to 100
c[0] = 100
c
```

```
array([100,   1,   2,   3,   4])
```

We can change the 5th element of the array to 0 as follows:

```
# Assign the 5th element to 0
c[4] = 0
c
```

```
array([100,   1,   2,   3,   0])
```

Slicing

Like lists, we can slice the numpy array. We can select the elements from 1 to 3 and assign it to a new numpy array `d` as follows:

```
# Slicing the numpy array
d = c[1:4]
```

```
d
```

```
array([1, 2, 3])
```

We can assign the corresponding indexes to new values as follows:

```
# Set the fourth element and fifth element to 300 and 400
c[3:5] = 300, 400
c
```

```
array([100,   1,   2, 300, 400])
```

Assign Value with List

Similarly, we can use a list to select more than one specific index.
The list `select` contains several values:

```
# Create the index list
select = [0, 2, 3]
```

We can use the list as an argument in the brackets. The output is the elements corresponding to the particular indexes:

```
# Use List to select elements
d = c[select]
d
```

```
array([100,   2, 300])
```

We can assign the specified elements to a new value. For example, we can assign the values to 100 000 as follows:

```
```

```
# Assign the specified elements to new value
c[select] = 100000
c
```

array([100000,   1, 100000, 100000,   400])

Other Attributes

Let's review some basic array attributes using the array `a`:

```
# Create a numpy array
a = np.array([0, 1, 2, 3, 4])
a
```

array([0, 1, 2, 3, 4])

The attribute `size` is the number of elements in the array:

```
# Get the size of numpy array
a.size
```

5

The next two attributes will make more sense when we get to higher dimensions but let's review them. The attribute `ndim` represents the number of array dimensions, or the rank of the array. In this case, one:

```
# Get the number of dimensions of numpy array
a.ndim
```

1

The attribute `shape` is a tuple of integers indicating the size of the array in each

dimension:

```
# Get the shape/size of numpy array
a.shape
```

(5,)

```
# Create a numpy array
a = np.array([1, -1, 1, -1])
```

```
# Get the mean of numpy array
mean = a.mean()
mean
```

0.0

```
# Get the standard deviation of numpy array

standard_deviation=a.std()
standard_deviation
```

1.0

```
# Create a numpy array
b = np.array([-1, 2, 3, 4, 5])
b
```

array([-1, 2, 3, 4, 5])

```
# Get the biggest value in the numpy array
max_b = b.max()
max_b
```

5

```
# Get the smallest value in the numpy array
min_b = b.min()
min_b
```

-1

Numpy Array Operations

Array Addition

Consider the numpy array `u`:

```
u = np.array([1, 0])
u
```

array([1, 0])

Consider the numpy array `v`:

```
v = np.array([0, 1])
v
```

array([0, 1])

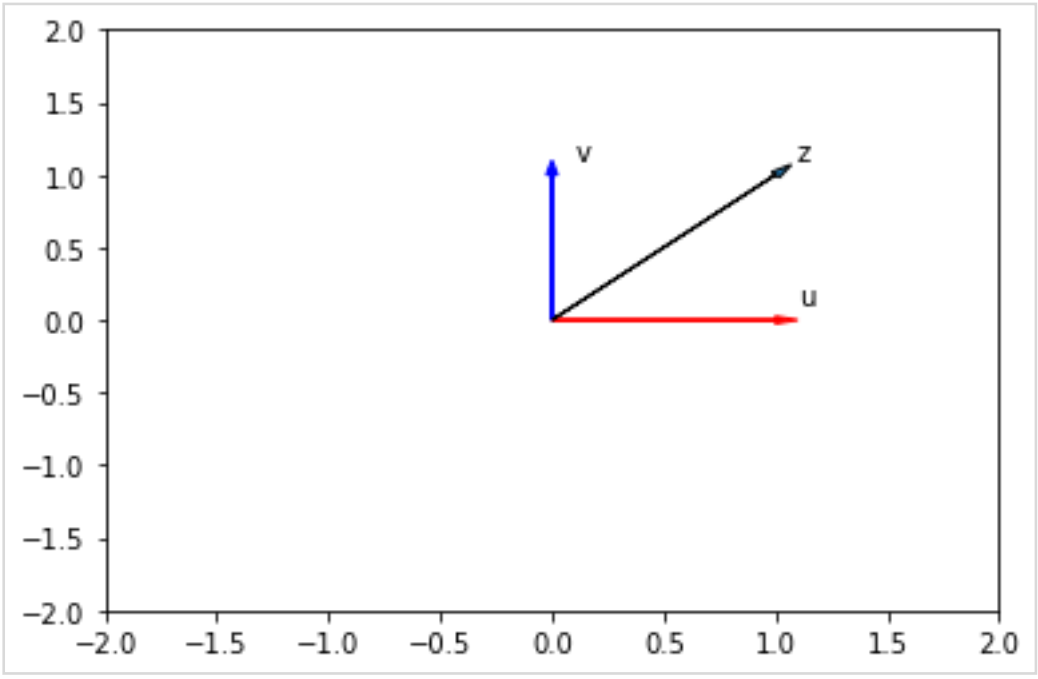
We can add the two arrays and assign it to z:

```
# Numpy Array Addition
z = u + v
z
```

array([1, 1])

The operation is equivalent to vector addition:

```
# Plot numpy arrays
Plotvec1(u, z, v)
```



Array Multiplication

Consider the vector numpy array `y`:

```
# Create a numpy array
y = np.array([1, 2])
y
```

```
array([1, 2])
```

We can multiply every element in the array by 2:

```
# Numpy Array Multiplication
z = 2 * y
z
```

```
array([2, 4])
```

This is equivalent to multiplying a vector by a scaler:

Product of Two Numpy Arrays

Consider the following array `u`:

```
# Create a numpy array
u = np.array([1, 2])
u
```

```
array([1, 2])
```

Consider the following array `v`:

```
# Create a numpy array
v = np.array([3, 2])
v
```

```
array([3, 2])
```

The product of the two numpy arrays `u` and `v` is given by:

```
# Calculate the production of two numpy arrays
```

```
z = u * v
z
```

array([3, 4])

Dot Product

The dot product of the two numpy arrays `u` and `v` is given by:

```
# Calculate the dot product
np.dot(u, v)
```

7

Adding Constant to a Numpy Array

Consider the following array:

```
# Create a constant to numpy array
u = np.array([1, 2, 3, -1])
u
```

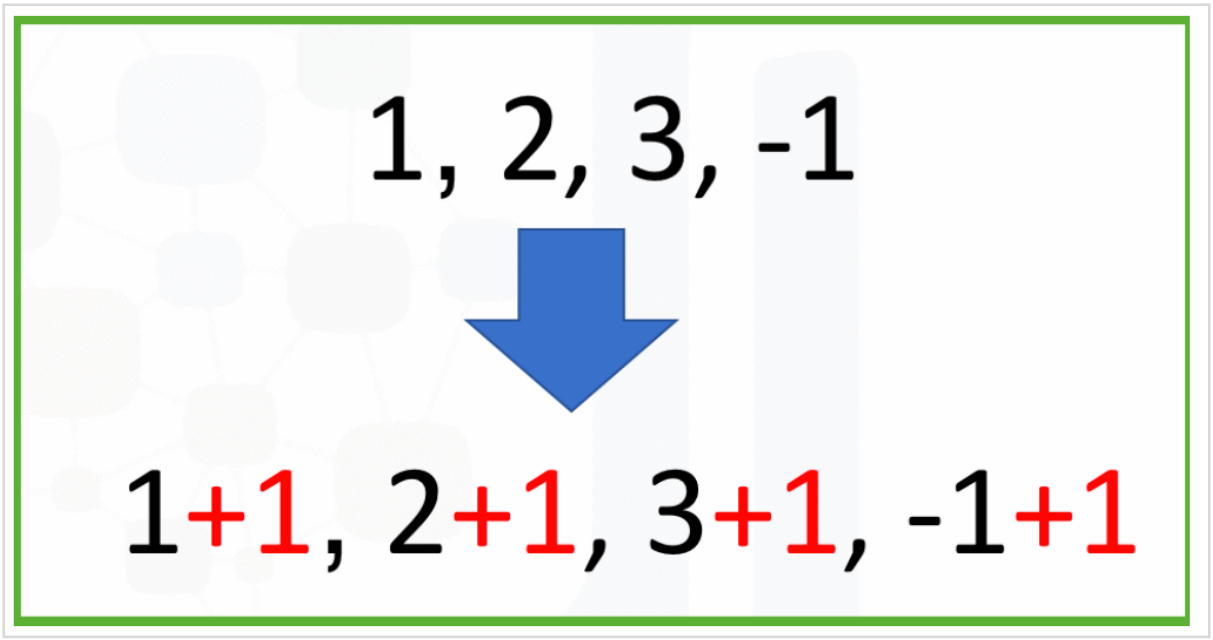
array([1, 2, 3, -1])

Adding the constant 1 to each element in the array:

```
# Add the constant to array
u + 1
```

array([2, 3, 4, 0])

The process is summarised in the following animation:



Mathematical Functions

We can access the value of `pi` in numpy as follows :

```
# The value of pi
np.pi
```

3.141592653589793

We can create the following numpy array in Radians:

```
# Create the numpy array in radians
x = np.array([0, np.pi/2 , np.pi])
```

We can apply the function `sin` to the array `x` and assign the values to the array `y`; this applies the sine function to each element in the array:

```
# Calculate the sin of each elements
y = np.sin(x)
```

```
y
```

```
array([0.0000000e+00, 1.0000000e+00, 1.2246468e-16])
```

Linspace

A useful function for plotting mathematical functions is `linspace`. Linspace returns evenly spaced numbers over a specified interval. We specify the starting point of the sequence and the ending point of the sequence. The parameter "num" indicates the Number of samples to generate, in this case 5:

```
# Makeup a numpy array within [-2, 2] and 5 elements
np.linspace(-2, 2, num=5)
```

```
array([-2., -1.,  0.,  1.,  2.])
```

If we change the parameter `num` to 9, we get 9 evenly spaced numbers over the interval from -2 to 2:

```
# Make a numpy array within [-2, 2] and 9 elements
np.linspace(-2, 2, num=9)
```

```
array([-2. , -1.5, -1. , -0.5,  0. ,  0.5,  1. ,  1.5,  2. ])
```

We can use the function `linspace` to generate 100 evenly spaced samples from the interval 0 to 2π:

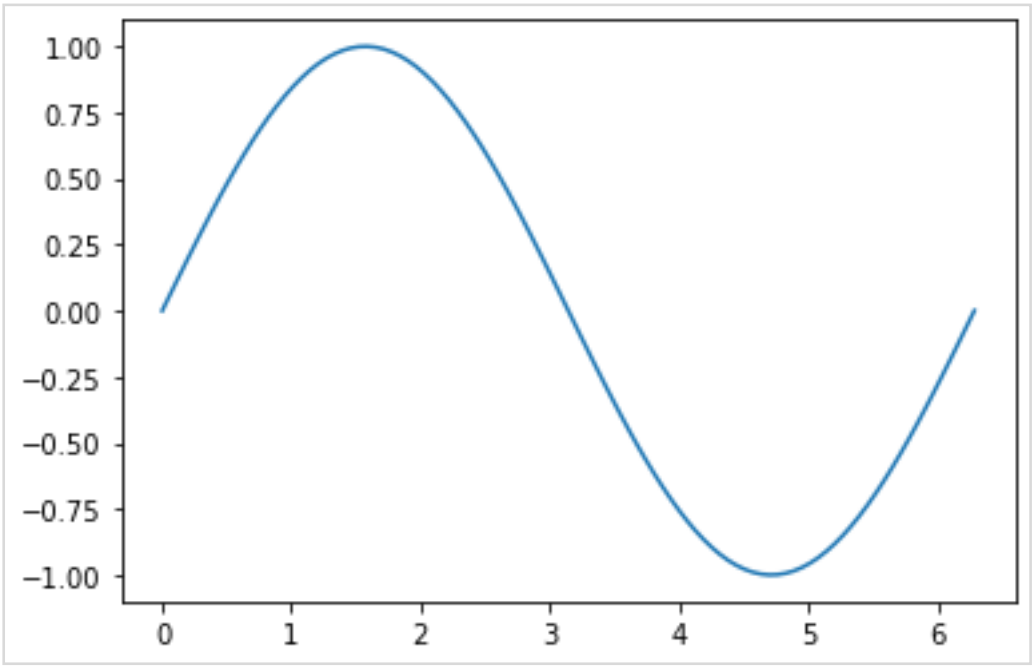
```
# Make a numpy array within [0, 2π] and 100 elements
x = np.linspace(0, 2*np.pi, num=100)
```

We can apply the sine function to each element in the array `x` and assign it to the array `y`:

```
# Calculate the sine of x list
y = np.sin(x)
```

```
# Plot the result
plt.plot(x, y)
```

```
matplotlib.lines.Line2D at 0x12c3be040
```



Quiz on 1D Numpy Array

Implement the following vector subtraction in numpy: `u-v`

```
# Write your code below and press Shift+Enter to execute
u = np.array([1, 0])
v = np.array([0, 1])
z = u - v
z
```

array([1, -1])

Multiply the numpy array `z` with -2:

```
# Write your code below and press Shift+Enter to execute
z = np.array([2, 4])
k = -2*z
k
```

array([-4, -8])

Consider the list `[1, 2, 3, 4, 5]` and `[1, 0, 1, 0, 1]`. Cast both lists to a numpy array then multiply them together:

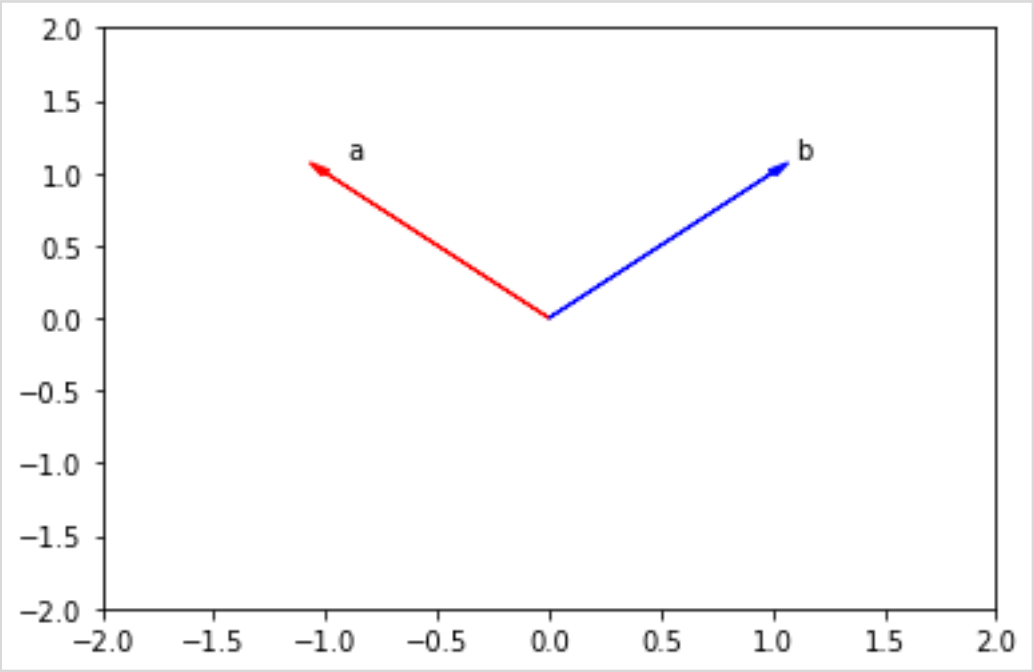
```
# Write your code below and press Shift+Enter to execute
a = np.array([1, 2, 3, 4, 5])
b = np.array([1, 0, 1, 0, 1])
c = a*b
c
```

array([1, 0, 3, 0, 5])

Convert the list `[-1, 1]` and `[1, 1]` to numpy arrays `a` and `b`. Then, plot the arrays as vectors using the fuction `Plotvec2` and find their dot product:

```
# Write your code below and press Shift+Enter to execute
u = np.array([-1,1])
v = np.array([1,1])
Plotvec2(u,v)
print("The dot product is",np.dot(u,v))
```

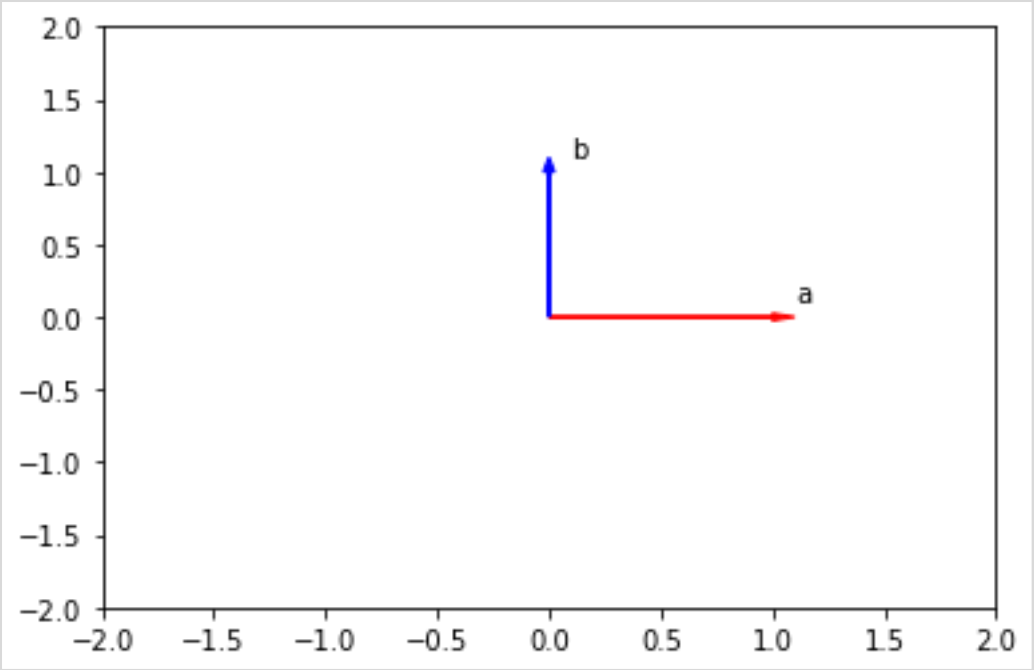
The dot product is 0



Convert the list `[1, 0]` and `[0, 1]` to numpy arrays `a` and `b`. Then, plot the arrays as vectors using the function `Plotvec2` and find their dot product:

```
# Write your code below and press Shift+Enter to execute
a = np.array([1,0])
b = np.array([0,1])
Plotvec2(a,b)
print("Their dot product is", np.dot(a,b))
```

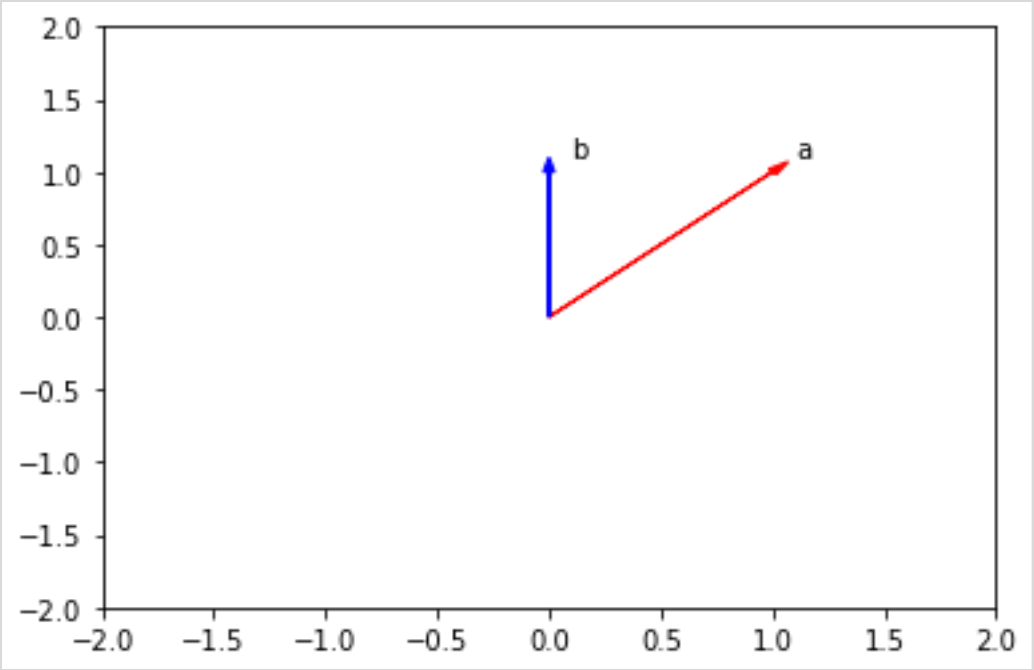
Their dot product is 0



Convert the list `[1, 1]` and `[0, 1]` to numpy arrays `a` and `b`. Then plot the arrays as vectors using the fuction `Plotvec2` and find their dot product:

```
# Write your code below and press Shift+Enter to execute
a = np.array([1,1])
b = np.array([0,1])
Plotvec2(a,b)
print("Their dot product is",np.dot(a,b))
```

Their dot product is 1



Why are the results of the dot product for `[-1, 1]` and `[1, 1]` and the dot product for `[1, 0]` and `[0, 1]` zero, but not zero for the dot product for `[1, 1]` and `[0, 1]`?

Hint: Study the corresponding figures, pay attention to the direction the arrows are pointing to.

```
# Write your code below and press Shift+Enter to execute
I vettori che danno prodotto scalare nullo sono ortogonali tra di loro.
```

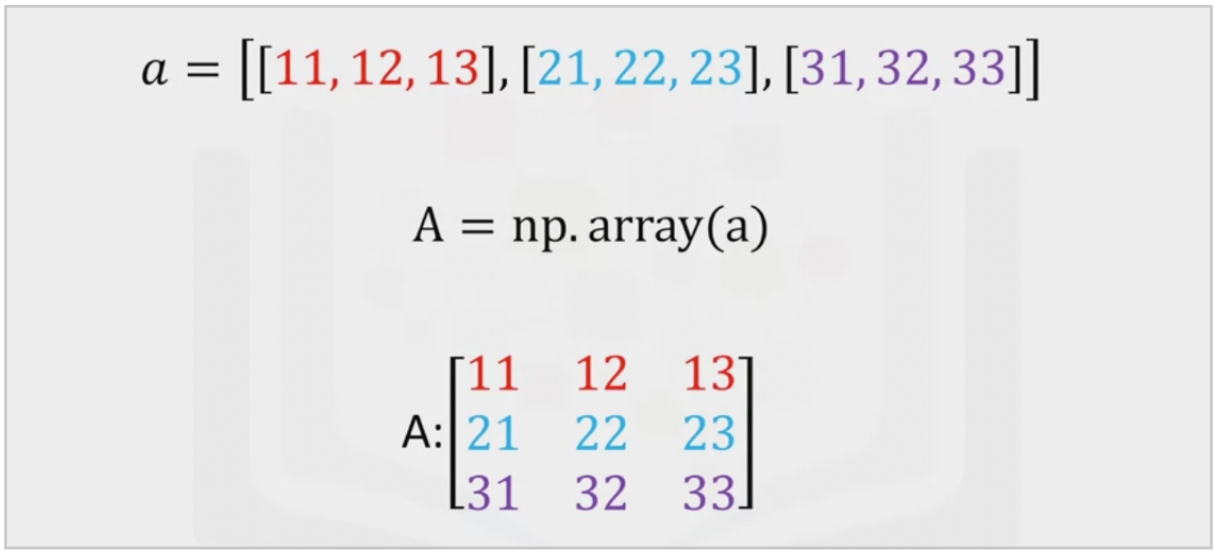
Two Dimensional Numpy

We can create numpy arrays with more than one dimension. This section will focus only on 2D arrays but you can use numpy to build arrays of much higher dimensions. In this video, we will cover:

- the basics and array creation in 2D,
- indexing and slicing in 2D,
- basic operations in 2D.

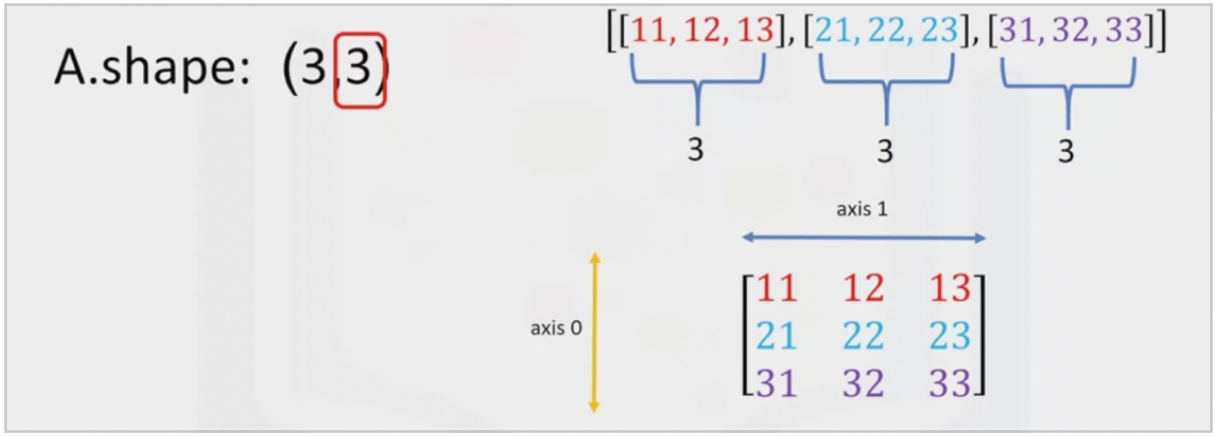
Consider the list a, the list contains three nested lists each of equal size. Each list is color-coded for simplicity. We can cast the list to a numpy array as follows. It is helpful to visualize the numpy array as a rectangular array each nested lists corresponds to a

different row of the matrix.

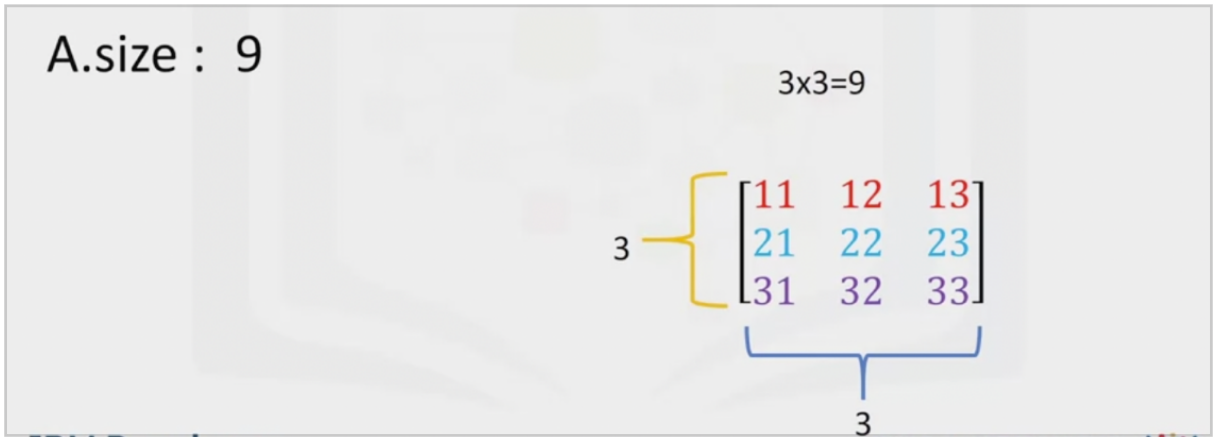


We can use the attribute `ndim` to obtain the number of axes or dimensions referred to as the `rank`. **The term rank does not refer to the number of linearly independent columns like a matrix**. It's useful to think of ndim as the **number of nested lists**. The **first list** represents the **first dimension**: this list contains another set of lists. The blu parenthesis elements represents the **second dimension** or axis. The number of lists the list contains does not have to do with the dimension but the shape of the list.

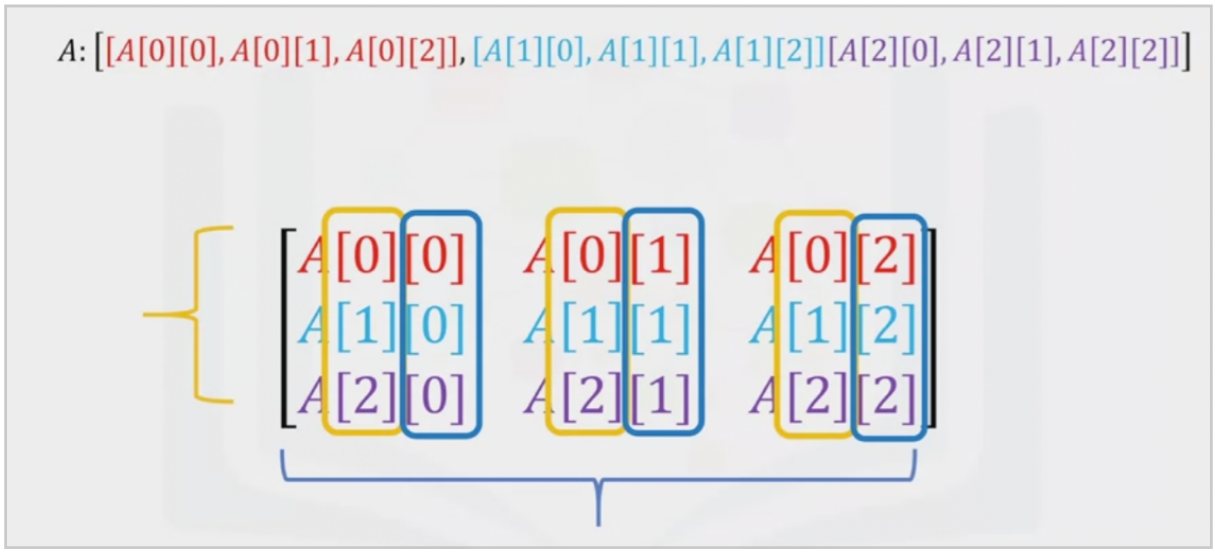
As with a 1D array, the attribute `shape` returns a tuple. It's helpful to use the rectangular representation as well. The **first element** in the tuple corresponds to the **number of nested lists** contained in the original list **or the number of rows** in the rectangular representation, in this case three. The **second element** corresponds to the **size of each of the nested list** or the **number of columns** in the rectangular array zero. The convention is to label this axis zero and this axis one as follows.



We can also use the attribute `size` to get the size of the array. We see there are three rows and three columns. Multiplying the number of columns and rows together, we get the total number of elements, in this case nine. Check out the labs for arrays of different shapes and other attributes.

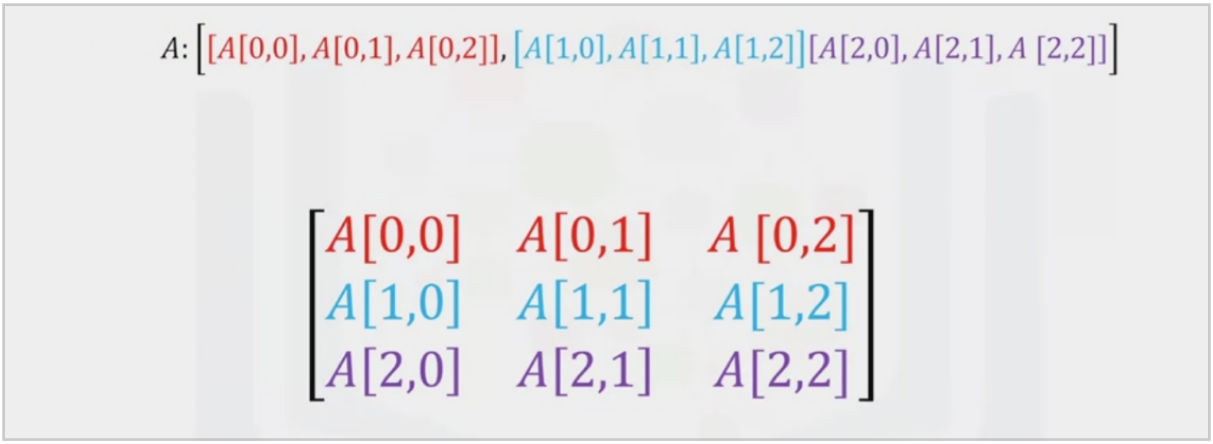


We can use **rectangular brackets** to access the different elements of the array. The following image demonstrates the relationship between the indexing conventions for the lists like representation.

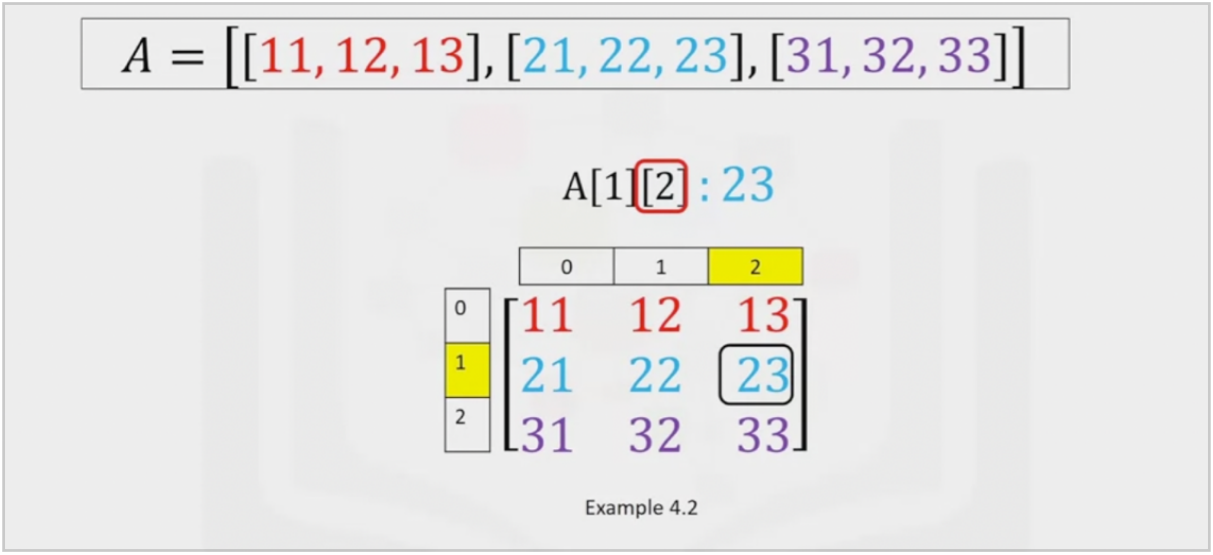


The index in the first bracket corresponds to the different nested lists each a different color. The second bracket corresponds to the index of a particular element within the

nested list. Using the rectangular representation, the first index corresponds to the row index. The second index corresponds to the column index. We could also use a single bracket to access the elements as follows.

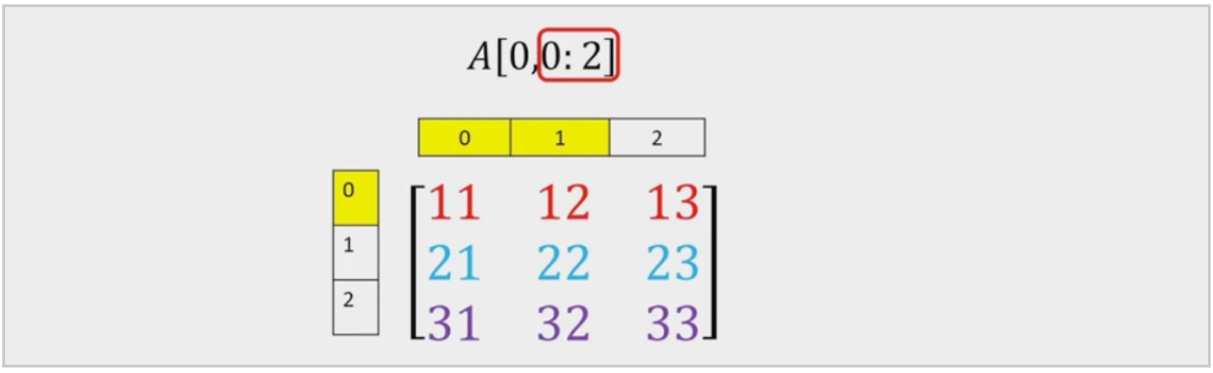


Consider the following syntax. This index corresponds to the second row, and this index the third column, the value is 23.



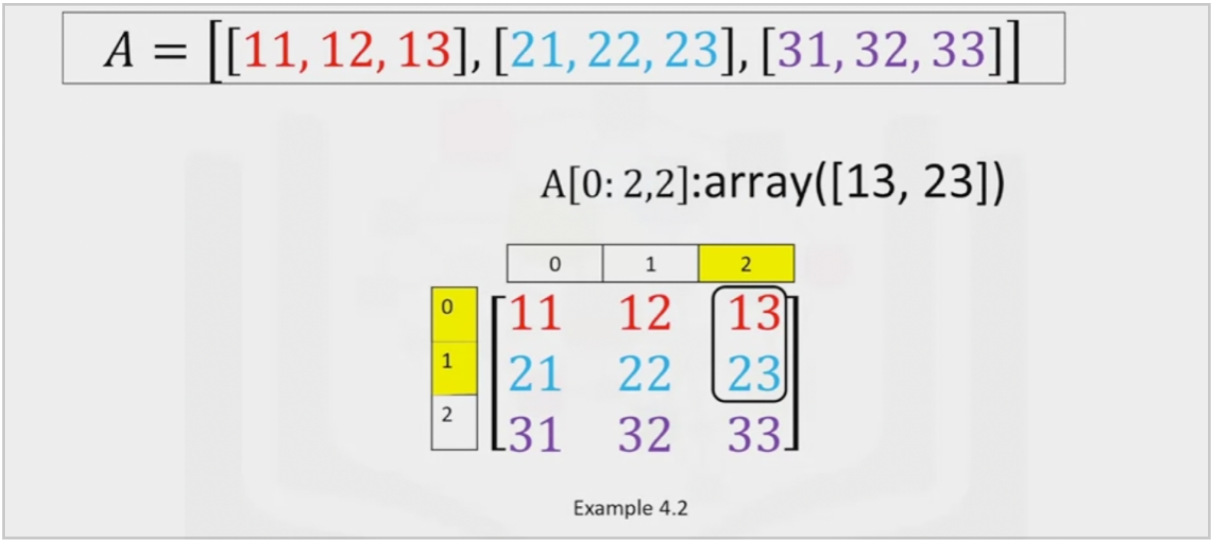
Consider this example, this index corresponds to the first row and the second index corresponds to the first column, and a value of 11.

We can also use `slicing` in numpy arrays. The first index corresponds to the first row. The second index accesses the first two columns.

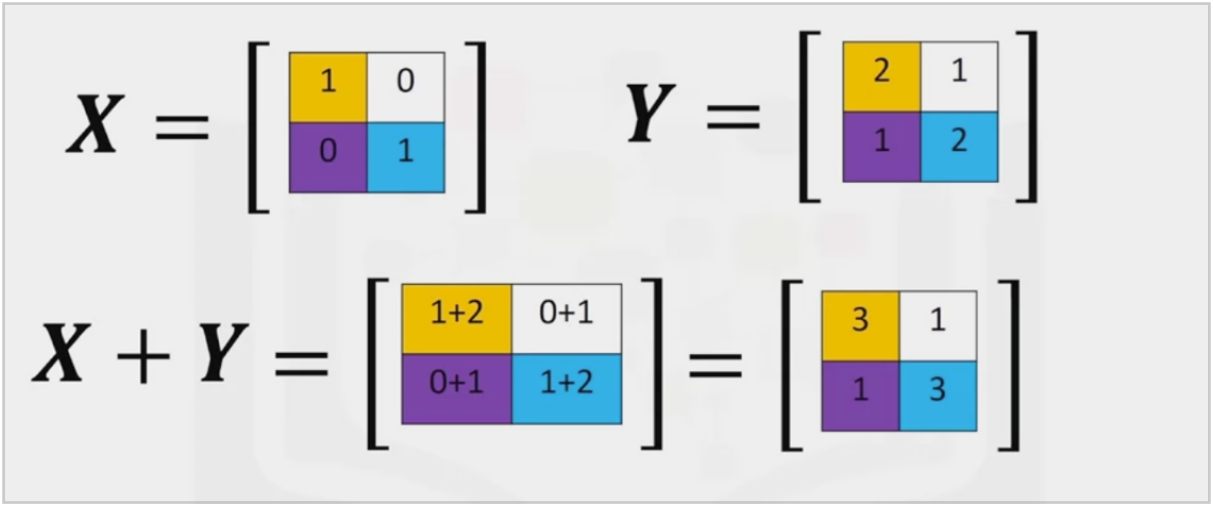


Consider this example, the first index corresponds to the first two rows. The second index

accesses the last column.



We can also add arrays, the process is identical to matrix addition. Consider the matrix X, each element is colored differently. Consider the matrix Y. Similarly, each element is colored differently. We can add the matrices. This corresponds to adding the elements in the same position, i.e adding elements contained in the same color boxes together. The result is a new matrix that has the same size as matrix Y or X. Each element in this new matrix is the sum of the corresponding elements in X and Y.



Sum

To add two arrays in numpy, we define the array in this case X. Then we define the second array Y, we add the arrays. The result is identical to matrix addition.

```
X=np.array([[1,0],[0,1]])
Y=np.array([[2,1],[1,2]])
Z=X+Y;
Z:array([[3,1],
         [1,3]])
```

$$X = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$Y = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$$

$$Z = X + Y$$

$$Z = \begin{bmatrix} 3 & 1 \\ 1 & 3 \end{bmatrix}$$

Multiplication by a scalar

Multiplying a numpy array by a scalar is identical to multiplying a matrix by a scalar. Consider the matrix Y. If we multiply the matrix by this scalar two, we simply multiply every element in the matrix by two. The result is a new matrix of the same size where each element is multiplied by two.

$$Y = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$$

$$2Y = \begin{bmatrix} 2x2 & 2x1 \\ 2x1 & 2x2 \end{bmatrix} = \begin{bmatrix} 4 & 2 \\ 2 & 4 \end{bmatrix}$$

Consider the array Y. We first define the array, we multiply the array by a scalar as follows and assign it to the variable Z. The result is a new array where each element is multiplied by two.

```
Y=np.array([[2,1],[1,2]])

Z=2*Y;

Z:array([[4,2],
         [2,4]])
```

$$Y = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$$

$$Z = 2Y = \begin{bmatrix} (2)2 & (2)1 \\ (2)1 & (2)2 \end{bmatrix}$$

$$Z = \begin{bmatrix} 4 & 2 \\ 2 & 4 \end{bmatrix}$$

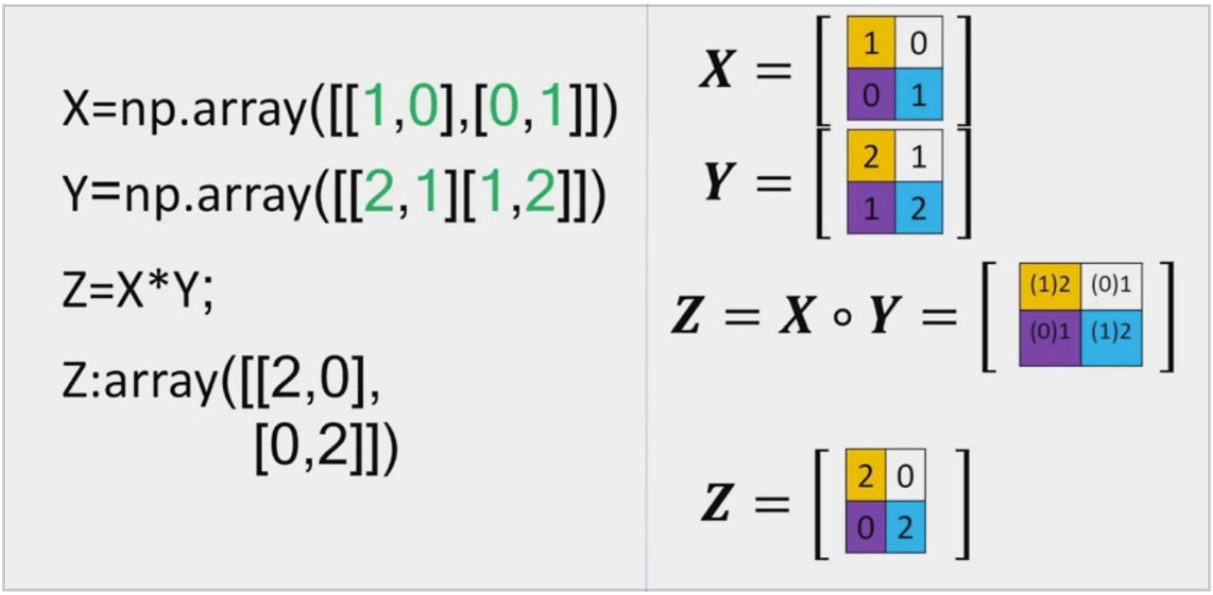
Hadamard product (Prodotto scalare)

Multiplication of two arrays corresponds to an element-wise product, or Hadamard product. Consider array X and array Y. Hadamard product corresponds to multiplying each of the elements in the same position i.e multiplying elements contained in the same color boxes together. The result is a new matrix that is the same size as matrix Y or X. Each element in this new matrix is the product of the corresponding elements in X and Y.

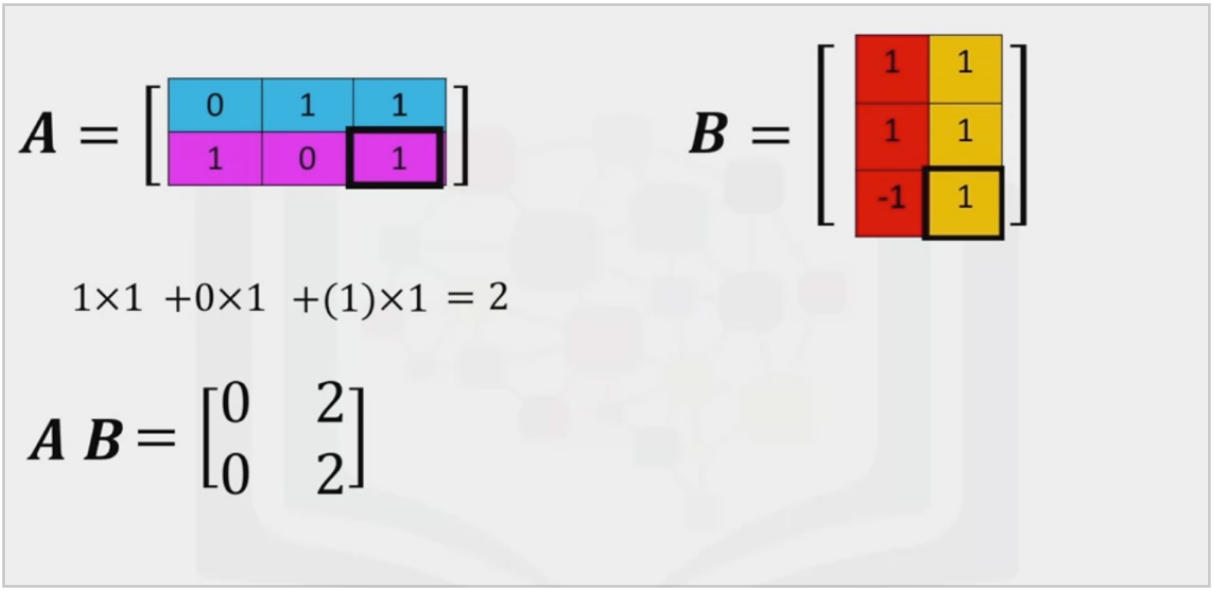
$$X = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \qquad Y = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$$

$$X \circ Y = \begin{bmatrix} (1)2 & (0)1 \\ (0)1 & (1)2 \end{bmatrix} = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}$$

Consider the array X and Y. We can find the product of two arrays X and Y in one line, and assign it to the variable Z as follows. The result is identical to Hadamard product.



We can also perform **matrix multiplication** with Numpy arrays. Matrix multiplication is a little more complex but let's provide a basic overview. Consider the matrix A where each row is a different color. Also, consider the matrix B where each column is a different color. In linear algebra, before we multiply matrix A by matrix B, we must make sure that the number of columns in matrix A in this case three is equal to the number of rows in matrix B, in this case three. From matrix multiplication, to obtain the ith row and jth column of the new matrix, we take the dot product of the ith row of a with the jth columns of B. For the first column, first row we take the dot product of the first row of A with the first column of B as follows. The result is zero. For the first row and the second column of the new matrix, we take the dot product of the first row of the matrix A, but this time we use the second column of matrix B, the result is two. For the second row and the first column of the new matrix, we take the dot product of the second row of the matrix A. With the first column of matrix B, the result is zero. Finally, for the second row and the second column of the new matrix, we take the dot product of the second row of the matrix A with the second column of matrix B, the result is two.



In numpy, we can define the numpy arrays A and B. We can perform matrix multiplication and assign it to array C. The result is the array C. It corresponds to the matrix multiplication of array A and B. There is a lot more you can do with it in numpy.

```
A=np.array([[0,1,1],[1,0,1]])
B=np.array([[1,1],[1,1],[-1,1]])
C=np.dot(A,B);
C:array([[0,2],
         [0,2]])
```

Two Dimensional Numpy - Lab

Objective

After completing this lab you will be able to:

- Operate comfortably with `numpy`
- Perform complex operations with `numpy

Table of Contents</h2>

Create a 2D Numpy Array

```
# Import the libraries
import numpy as np
import matplotlib.pyplot as plt
```

Consider the list `a`, which contains three nested lists **each of equal size**.

```
# Create a list
a = [[11, 12, 13], [21, 22, 23], [31, 32, 33]]
a
```

11, 12, 13], [21, 22, 23], [31, 32, 33

We can cast the list to a Numpy Array as follows:

```
# Convert list to Numpy Array
# Every element is the same type

A = np.array(a)
A
```

```
array([[11, 12, 13],
       [21, 22, 23],
       [31, 32, 33]])
```

We can use the attribute `ndim` to obtain the number of axes or dimensions, referred to as the rank.

```
# Show the numpy array dimensions
A.ndim
```

```
```

2

Attribute `shape` returns a tuple corresponding to the size or number of each dimension.

```
# Show the numpy array shape
A.shape
```

(3, 3)

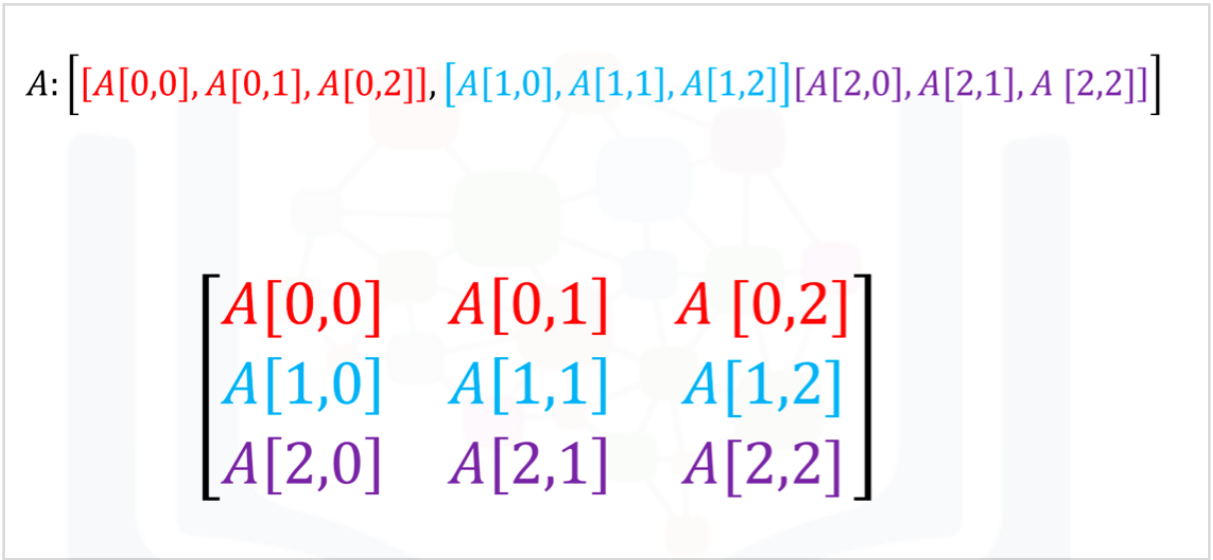
The total number of elements in the array is given by the attribute `size`.

```
# Show the numpy array size
A.size
```

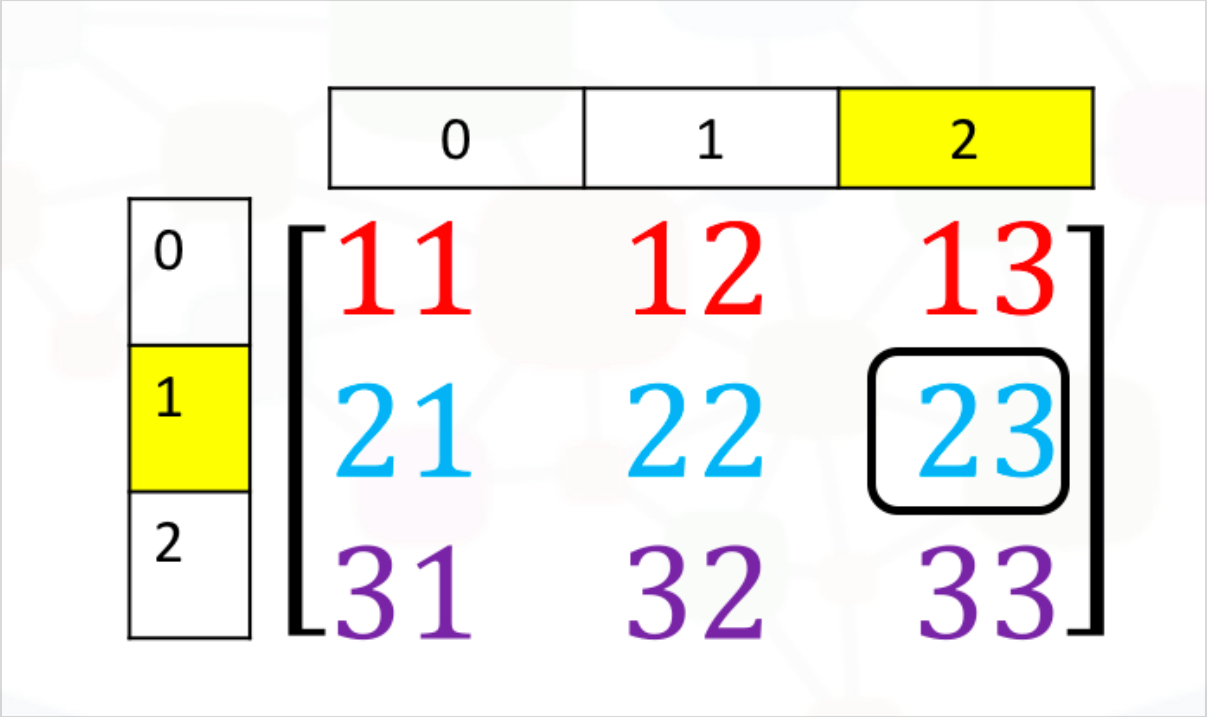
9

Accessing different elements of a Numpy Array

We can use rectangular brackets to access the different elements of the array. The correspondence between the rectangular brackets and the list and the rectangular representation is shown in the following figure for a 3x3 array:



We can access the 2nd-row, 3rd column as shown in the following figure:



We simply use the square brackets and the indices corresponding to the element we would like:

```
# Access the element on the second row and third column
A[1, 2]
```

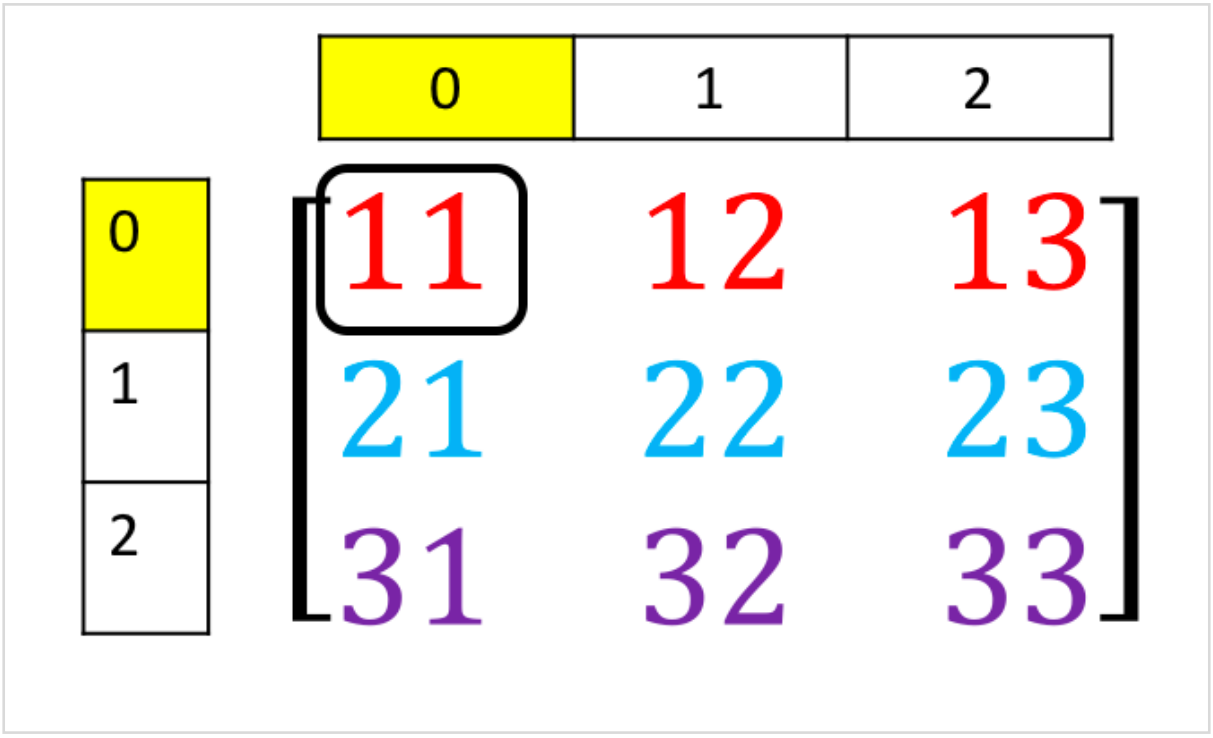
```
23
```

We can also use the following notation to obtain the elements:

```
# Access the element on the second row and third column
A[1][2]
```

```
23
```

Consider the elements shown in the following figure

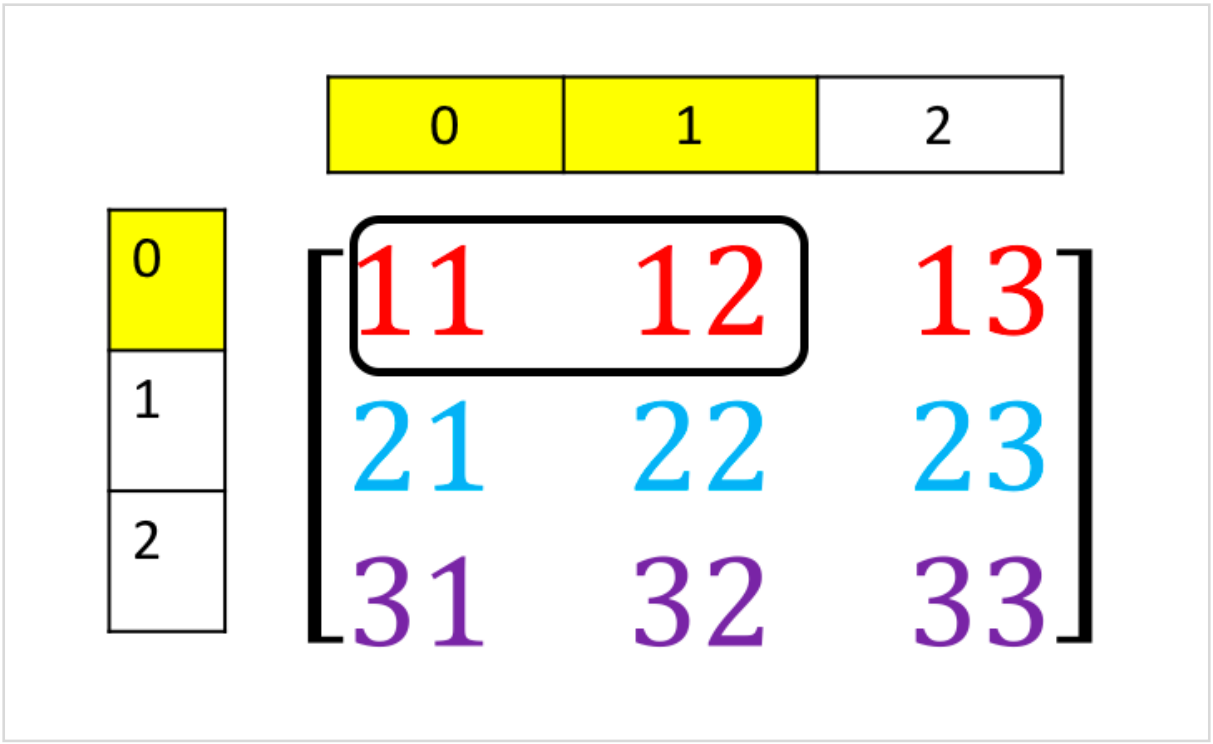


We can access the element as follows:

```
# Access the element on the first row and first column
A[0][0]
```

```
11
```

We can also use slicing in numpy arrays. Consider the following figure. We would like to obtain the first two columns in the first row



This can be done with the following syntax:

```
# Access the element on the first row and first and second columns
A[0][0:2]
```

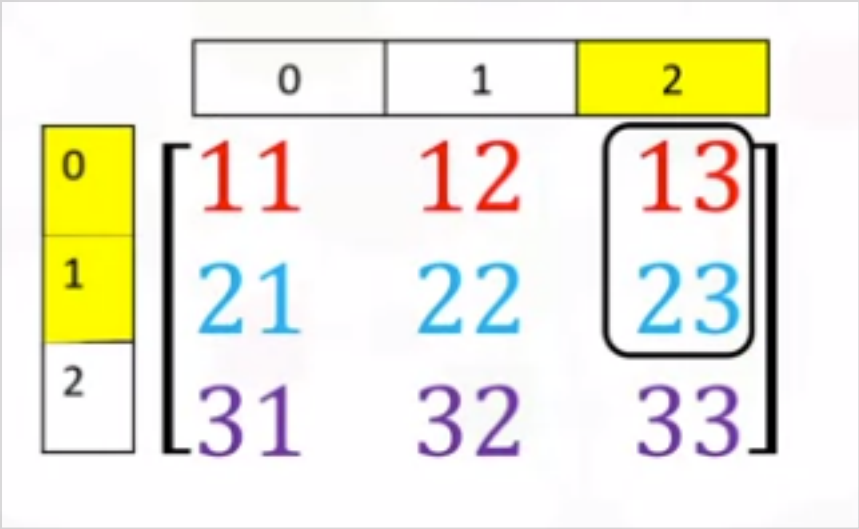
```
array([11, 12])
```

Similarly, we can obtain the first two rows of the 3rd column as follows:

```
# Access the element on the first and second rows and third column

A[0:2, 2]
```

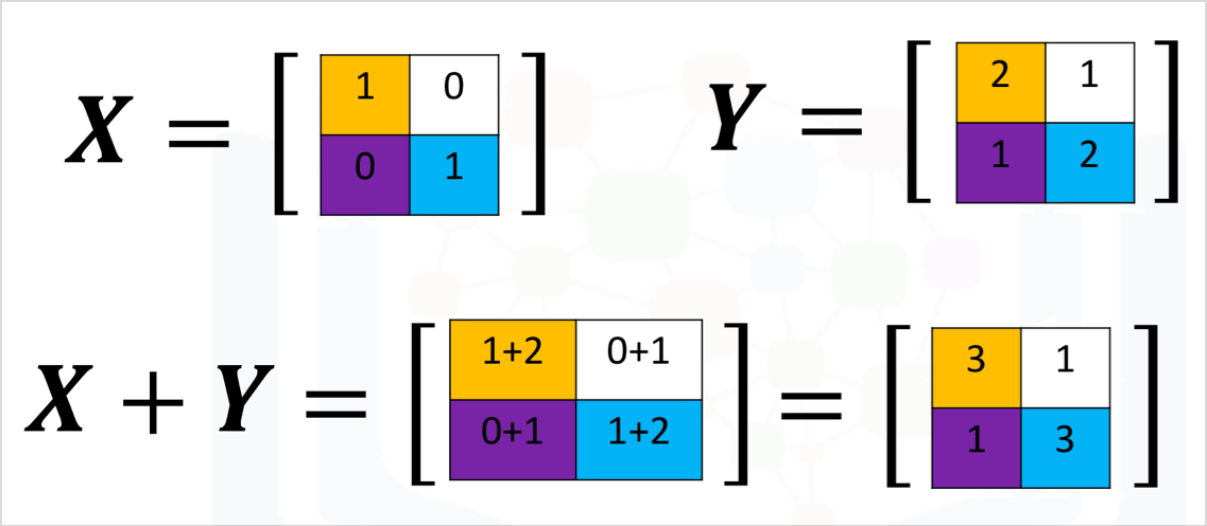
```
array([13, 23])
```

Corresponding to the following figure:



Basic Operations

We can also add arrays. The process is identical to matrix addition. Matrix addition of `X` and `Y` is shown in the following figure:



The numpy array is given by `X` and `Y`

```
# Create a numpy array X
X = np.array([[1, 0], [0, 1]])
X
```

```
array([[1, 0],
       [0, 1]])
```

```
# Create a numpy array Y
Y = np.array([[2, 1], [1, 2]])
Y
```

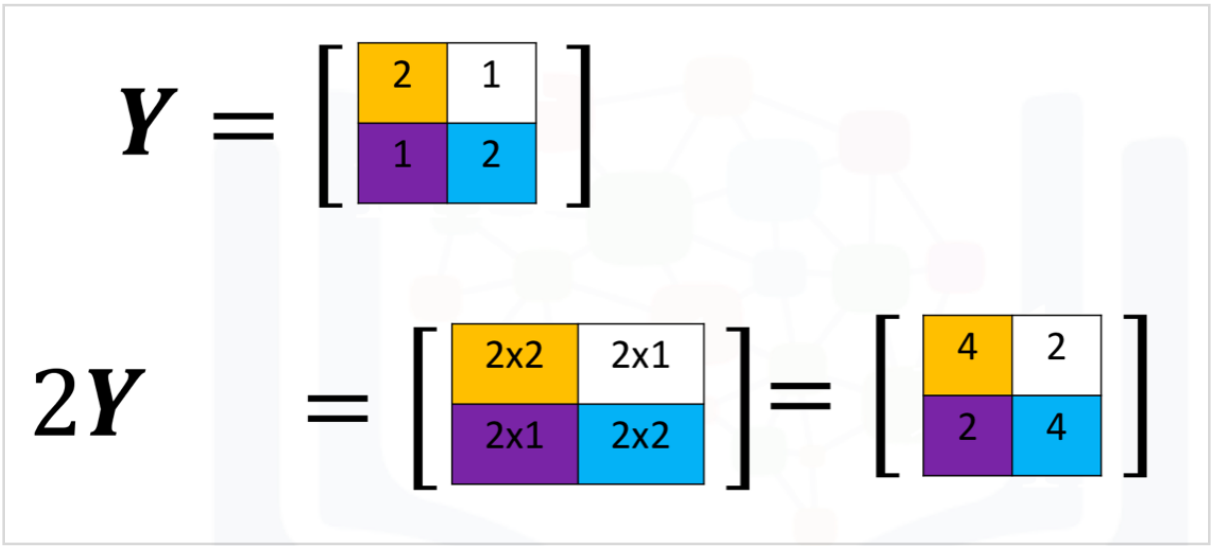
```
array([[2, 1],
       [1, 2]])
```

We can add the numpy arrays as follows.

```
# Add X and Y
Z = X + Y
Z
```

```
array([[3, 1],
       [1, 3]])
```

Multiplying a numpy array by a scaler is identical to multiplying a matrix by a scaler. If we multiply the matrix `Y` by the scaler 2, we simply multiply every element in the matrix by 2, as shown in the figure.



We can perform the same operation in numpy as follows

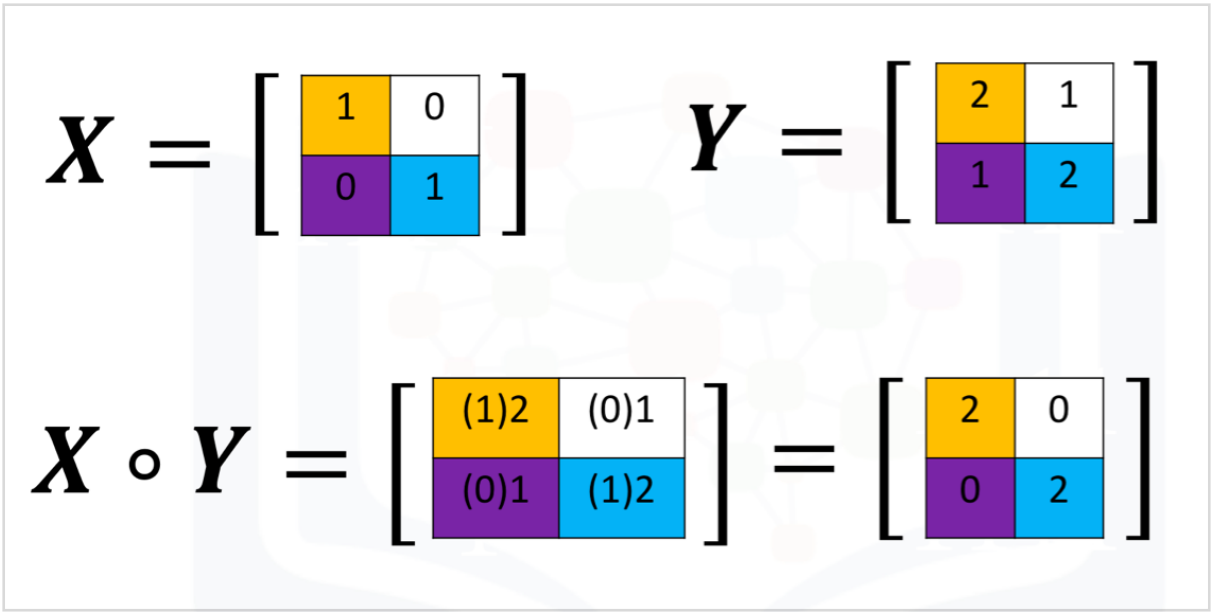
```
# Create a numpy array Y
Y = np.array([[2, 1], [1, 2]])
Y
```

```
    array([[2, 1],
           [1, 2]])
```

```
# Multiply Y with 2

Z = 2 * Y
Z
```

```
    array([[4, 2],
           [2, 4]])
```

Multiplication of two arrays corresponds to an element-wise product or Hadamard product. Consider matrix `X` and `Y`. The Hadamard product corresponds to multiplying each of the elements in the same position, i.e. multiplying elements contained in the same color boxes together. The result is a new matrix that is the same size as matrix `Y` or `X`, as shown in the following figure.



We can perform element-wise product of the array `X` and `Y` as follows:

```
# Create a numpy array Y
Y = np.array([[2, 1], [1, 2]])
Y
```

```
    array([[2, 1],
           [1, 2]])
```

```
# Create a numpy array X
X = np.array([[1,2], [1, 2]])
X
```

```
    array([[1, 2],
           [1, 2]])
```

```
# Multiply X with Y

Z = X * Y
```

```
Z
```

```
array([[2, 2],
       [1, 4]])
```

We can also perform matrix multiplication with the numpy arrays `A` and `B` as follows:

First, we define matrix `A` and `B`:

```
# Create a matrix A
A = np.array([[0, 1, 1], [1, 0, 1]])
A
```

```
array([[0, 1, 1],
       [1, 0, 1]])
```

```
# Create a matrix B

B = np.array([[1, 1], [1, 1], [-1, 1]])
B
```

```
array([[ 1,  1],
       [ 1,  1],
       [-1,  1]])
```

We use the numpy function `dot` to multiply the arrays together.

```
# Calculate the dot product
Z = np.dot(A,B)
Z
```

```
array([[0, 2],
       [0, 2]])
```

```
# Calculate the sine of Z

np.sin(Z)
```

```
array([[0.        , 0.90929743],
       [0.        , 0.90929743]])
```

We use the numpy attribute `T` to calculate the transposed matrix

```
# Create a matrix C
C = np.array([[1,1],[2,2],[3,3]])
C
```

```
array([[1, 1],
       [2, 2],
       [3, 3]])
```

```
# Get the transposed of C
C.T
```

```
array([[1, 2, 3],
       [1, 2, 3]])
```

Quiz on 2D Numpy Array

Consider the following list `a`, convert it to Numpy Array.

```
# Write your code below and press Shift+Enter to execute
a = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
A = np.array(a)
A
```

```
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

Calculate the numpy array size.

```
# Write your code below and press Shift+Enter to execute
A.size
```

12

Access the element on the first row and first and second columns.

```
# Write your code below and press Shift+Enter to execute
A[0,0:2]
```

array([1, 2])

Perform matrix multiplication with the numpy arrays `A` and `B`.

```
# Write your code below and press Shift+Enter to execute
B = np.array([[0, 1], [1, 0], [1, 1], [-1, 0]])
```

```
C = np.dot(A,B)
C
```

```
    array([[ 1,  4],
           [ 5, 12],
           [ 9, 20]])
```
