

- Module #1: Python Basics
  - Types
  - Expressions and Variables
  - String Operations
- Module #2: Python Data Structures
  - Lists and Tuples
  - Dictionaries
  - Sets
- Module #3: Python Programming Fundamentals
  - Conditions and Branching
  - Loops
  - Functions
  - Exception Handling
  - Objects and Classes
- Module #4: Working with Data in Python
  - Reading & Writing Files with Open
  - Pandas
  - Numpy in Python
- Module #5: APIs and Data Collection
  - Simple APIs
  - REST APIs, Web Scraping and Working with Files
  - Final Exam

## ❖ MODULE #1: Python Basics

### WEEK 1

#### Types

- Integers: `type(11)` → int or integer, they can be negative or positive
- Real numbers: `type(21.213)` → float, they include the integers but also numbers in between the integers.
- Words: `type("Hello Python 101")` → str
- You can change the type of the expression in Python, this is called typecasting, you can convert an int to a float.
  - `float(2):2.0`
  - `int(1.1):1` → you get rid of the decimals
  - `int('1'):1`
  - `int('A') → ERROR`
  - `str(1): "1"`
  - `str(4.5): '4.5'`
- Boolean can take on two values
  - `type(True): bool` → short for boolean
  - True
    - `True → int(True) → 1`
    - `True ← bool(1) ← 1`
  - False
    - `False → int(False) → 0`
    - `False ← bool(0) ← 0`

#### Expressions and Variables

- For example, basic arithmetic operations like adding multiple numbers.
  - The result in this case is 160. We call the numbers (43, 60, 16, 41) operands, and the math symbols (+, -) are called operators.
  - `25 / 5 = 5.0`, `25 / 6 = 4.166` (both are floats)
  - `25 // 5 = 5`, `25 // 6 = 4` (both are ints)
- Variables: we use them to store values
  - `my_variable = 1`
    - `my_variable:1`
  - We can use the type command in variables as well
    - `type(x):float`
  - It's good practice to use meaningful variable names (ex: `total_hr`)

#### String Operations

- A string is a sequence of characters contained within two quotes:
  - "Michael Jackson"

- You can also use single quotes:
  - 'Michael Jackson'
- A string can be spaces or digits
  - "1 2 3 4 5 6"
- A string can also be special characters
  - '@#2\_#]&\*^%\$'
- Each element in the sequence can be accessed using an index represented by the array of numbers.

◦	M	i	c	h	a	e	l		J	a	c	k	s	o	n
◦	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
◦	M	i	c	h	a	e	l		J	a	c	k	s	o	n
◦	-15	-14	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

- STRING: Slicing.
  - We can bind a string to another variable.
    - Name[0:4] = Mich
    - Name[8:12] = Jack

- STRINGS: Stride
  - Name[::2]: "McalJcsn"
  - Name[0:5:2]: "Mca"

◦	M	i	c	h	a	e	l		J	a	c	k	s	o	n
◦	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

- TUPLES: Slicing
  - We can use the len command to obtain the length of the string.
    - len("Michael Jackson") = 15 → there are 15 elements
  - We can replicate the values of a string.
    - 3 \* "Michael Jackson" = "Michael Jackson Michael Jackson Michael Jackson"
- Strings: Immutable
  - You cannot change the value of the string, but you can create a new string.
    - Name = "Michael Jackson"
    - Name = Name + " is the best"
    - Name: "Michael Jackson is the best"
- Strings: escape sequences
  - \ are meant to proceed escape sequences

- Escape sequences are strings that are difficult to input
  - `print("Michael Jackson \n is the best")`
- Similarly, `\t` represents a tab.
  - `print("Michael Jackson \t is the best")` → Michael Jackson    is the best
- If you want to place a backslash in your string, use a double backslash.
  - `print("Michael Jackson \\ is the best")` → Michael Jackson \ is the best
- We can also place an "r" in front of the string.
  - `print(r"Michael Jackson \ is the best")` → Michael Jackson \ is the best
- String Methods
  - When we apply a method to string A, we get a new string B that is different from A.
    - `A="Thriller is the sixth studio album"`
    - `B=A.upper()`
    - `B:"THRILLER IS THE SIXTH STUDIO ALBUM"`
  - The method replaces a segment of the string- i.e. a substring with a new string.
    - `A='Michael Jackson is the best'`
    - `B=A.replace('Michael','Janet')`
    - `B="Janet Jackson is the best"`
- STRINGS: Stride
  - Method find, find substrings

Name= "Michael Jackson"

M	i	c	h	a	e	l		J	a	c	k	s	o	n
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

`Name.find('el'):5`

`Name.find('Jack'):8`

- 
- What is the result of following: `"hello Mike".find("Mike")`
  - 6

## WEEK 2

### List and Tuples

- Tuples are an ordered sequence
  - Tuples are written as comma-separated elements within parentheses
  - Here is a Tuple "Ratings"
    - `Ratings=(10,9,6,5,10,8,9,6,2)`
  - Three types: Strings, integers and floats
  - `tuple1=('disco',10,1.2)`

- `type(tuple1)=tuple` [the type is tuple]
  - `Tuple1[0]: "disco"`
  - `Tuple1[1]: 10`
  - `Tuple1[2]: 1.2`
  - `Tuple1[-3]: "disco"`
  - `Tuple1[-2]: 10`
  - `Tuple1[-1]: 1.2`
- `tuple2 = tuple1 + ("hard rock",10)`
  - `("disco",10,1.2,"hard rock",10)`  
`[0,1,2,3,4]`
- Tuples: Slicing
  - `tuple2[0:3] : ('disco',10,1.2)`
    - The last index is one larger than the index you want, hence why "hard rock" is not included
  - `tuple2[3:5] : ("hard rock",10)`
    - Notice, how the last index is one larger than the last index of the tuple
  - `len(("disco",10,1.2,"hard rock",10)) = 5`
- Tuples: Immutable (we can't change them)
  - `Ratings = (10,9,6,5,10,8,9,6,2)`
  - `Ratings1=Ratings`
    - `Ratings[2] = 4` → no change
    - `Ratings = (2,10,1)` → now references another tuple
  - If we would like to manipulate a tuple we must create a new tuple instead.
    - `Ratings = (10,9,6,5,10,8,9,6,2)`
    - `RatingsSorted=sorted(Ratings) → [2,5,6,6,8,9,9,10,10]`
- Tuples: Nesting
  - A tuple can contain other tuples as well as other complex data types. This is called nesting.
  - `NT =(1,2,("pop", "rock"),(3,4),("disco",(1,2)))`
    - `NT[2]: ("pop", "rock") [1] = "rock"`
      - `NT[2][0] = "pop"`
      - `NT[2][1] = "rock"`
      - `NT[3][0] = 3`
      - `NT[3][1] = 4`
      - `NT[4][0] = "disco"`
      - `NT[4][1] = (1,2)`
    - We can access different characters in the string or various elements in the second tuple contained in the first.
      - `NT[2][1][0] = "r"`
      - `NT[2][1][1] = "o"`
      - `NT[4][1][0] = 1`
      - `NT[4][1][1] = 2`

- Lists
  - Ordered sequences
  - A List is represented with square brackets
  - List is mutable
  - Lists can contain strings, floats, integers. We can nest other lists. We also nest tuples and other data structures.
    - ["Michael Jackson",10.1,1982,[1,2],('A',1)]
  - Here is a List "L" → L = ["Michael Jackson",10.1,1982]
    - L[0]: "Michael Jackson"
    - L[1]: 10.1
    - L[2]: 1982
    - L[-3]: "Michael Jackson"
    - L[-2]: 10.1
    - L[-1]: 1982
  - Lists: Slicing → L = ["Michael Jackson",10.1,1982,"MJ",1]
    - L[3:5] : ["MJ",1]
  - L1 = L+["pop",10]
    - L1 = ["Michael Jackson",10.1,1982,"pop",10]
  - [1,2,3]+[1,1,1]
    - [1, 2, 3, 1, 1, 1]
  - Lists are mutable
    - L = ["Michael Jackson",10.1,1982] → L.extend(["pop",10])
      - L = ["Michael Jackson",10.1,1982,"pop",10]
      - Extend adds two new elements to the list
    - L = ["Michael Jackson",10.1,1982] → L.append(["pop",10])
      - L = ["Michael Jackson",10.1,1982,["pop",10]]
      - Append only adds one element to the list
    - A = ["disco",10,1.2] → A[0] = "hard rock"
      - A = ["hard rock",10,1.2]
    - A = ["disco",10,1.2] → del(A[0])
      - A : [10,1.2]
  - Convert String to List
    - "Hello Mike".split()
      - ["Hello","Mike"]
    - "A,B,C,D".split(",")
      - ["A","B","C","D"]
  - Lists: Aliasing
    - Multiple names referring to the same object is known as aliasing.
    - B[0]="hard rock", A[0]="banana"
      - A and B are referencing the same list, therefore if we change A, list B also changes.
      - If we check the first element of B after changing list A, we get banana instead of hard rock.

- B[0]: "banana"
- You can clone list A by using the following syntax. Variable A references one list. Variable B references a new copy or clone of the original list.
  - A[0]=["hard rock", 10, 1.2], B=A[:]
  - A=["hard rock", 10, 1.2], A[0]="banana" → B[0] "hard rock" (doesn't change)

## Dictionaries

- Dictionaries are a type of collection in Python.
- A list is integer indexes and elements. A dictionary has keys (first value) and values.
- Dictionaries are denoted with curly Brackets {}
- The keys have to be immutable and unique
- The values can be immutable, mutable and duplicates
  - {"key1": 1, "key2": "2", "key3": [3, 3, 3], "key4": (4, 4, 4), ('key5'): 5}
- Each key and value pair is separated by a comma
- Example:
 

```
{ "Thriller": 1982, "Back in Black": 1980, "The Dark Side of the Moon": 1973, "The Bodyguard": 1992 }
```

  - The album title is the key, and the value is the released data
    - DICT["Thriller"]: 1982
    - DICT["Back in Black"]: 1980
    - DICT["The Dark Side of the Moon"]: 1973
  - We can delete an entry as follows
    - del(DICT['Thriller'])
  - Using the "in" command as follows
    - 'The Bodyguard' in DICT = True
    - "Starboy" in DICT = False
  - We can use the method "keys" to get the keys
    - DICT.keys() = ["Thriller", "Back in Black", "The Dark Side of the Moon", "The Bodyguard"]
  - We can use the method "values" to get the values
    - DICT.values() = ["1982", "1980", "1973", "1992"]

## Sets

- Sets are a type of collection
  - This means that like lists and tuples you can input different Python types
- Unlike lists and tuples they are unordered
  - This means sets do not record element position

- Sets only have unique elements
  - This means there is only one of a particular element in a set
- Creating a set
  - Set1={"pop","rock","soul","hard rock","rock","R&B","rock","disco"}
    - Set 1={"rock","R&B","disco","hard rock","poop","soul"}
    - Duplicate items won't be present
  - album\_list = ["Michael Jackson","Thriller","Thriller",1982]
    - album\_set = set(album\_list)
    - album\_set : {'Michael Jackson','Thriller',1982}
  - type(set([1,2,3])) = set
  - Set Operations → A = {"Thriller", "Back in Black", "AC/DC"}
    - A.add("NSYNC")
    - A: {"AC/DC", "Back in Black", "NSYNC", "Thriller"}
    - A.remove("NSYNC")
    - A: {"AC/DC", "Back in Black", "Thriller"}
    - "AC/DC" in A = True
    - "Who" in A = False
  - Mathematical set operations →
    - album\_set\_1 = {"AC/DC", "Back in Black", "Thriller"}
    - album\_set\_2 = {"AC/DC", "Back in Black", "The Dark Side of the Moon"}
    - Ampersand (&) to find the intersection of the two sets
    - album\_set\_3 = album\_set\_1 & album\_set\_2 →  
album\_set\_3: {"AC/DC", "Back in Black"}
    - album\_set\_1.union(album\_set\_2) =  
{"AC/DC", "Back in Black", "The Dark Side of the Moon"}
    - album\_set\_3.issubset(album\_set1) = True

### WEEK 3

#### **Conditions and Branching**

- Comparison operations compare some value or operand. Then based on some condition, they produce a Boolean.
  - a=6
    - a==7 → False
    - a==6 → True
  - i=6
    - i>5 → True
  - i=5
    - i>=5 → True
    - i==2 → False
  - i=2
    - i<6 → True
    - i!=6 → True



- "AC/DC"=="Michael Jackson" → False
  - "AC/DC"!="Michael Jackson" → True
- 'a'=='A' → True (it's case sensitive)
- The if Statement
  - if(age>18):
    - print("You can enter")
- The else Statement
  - if(age>18):
    - print("You can enter")
  - else
    - print("Go see MeatLoaf")
- The elif Statement
  - if(age>18):
    - print("You can enter")
  - elif(age==18):
    - print("Go see Pink Floyd")
  - else:
    - print("go see MeatLoaf")
- Logic Operators
  - not
    - If the input is true, the result is a false
      - not(True) → False
    - If the input is false, the result is a true
      - not(False) → True
  - or
    - An or statement is only False if all the Boolean values are False
    - album\_year = 1990
    - If(album\_year < 1980) or (album\_year > 1989)
      - print("The Album was made in the 70's or 90's")
    - else
      - print("The Album was made in the 1980's")
  - and
    - album\_year = 1983
    - If(album\_year > 1979) and (album\_year < 1990)
      - print("The Album was made in the 1980's")

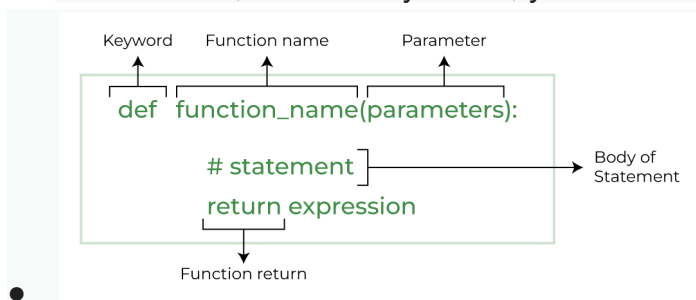
## Loops

- range(N)
  - range(3)
    - [0,1,2]
  - range(10,15)
    - [10,11,12,13,14]
- for

- squares="red","yellow","green","purple","blue"]
  - for i in range(0,5):
    - squares[i]="white"
- squares=["red","yellow","green"]
  - for square in squares:
    - square
- while loops
  - similar to for loops but instead of executing a statement a set number of times, a while loop will only run if a condition is met
  - squares=['orange','orange','purple','orange','blue']
    - Newsquares=[] → (indeterminate size)
    - i=0
    - while(squares[i]=='orange'):
      - Newsquares.append(squares[i])
      - i=i+1

## Functions

- The function is just a piece of code you can reuse. You can implement your own function, but in many cases, you use other people's functions.



- Len
  - album\_ratings = [10,8.5,9.5,7.0,7.0,9.5,9.0,9.5]
  - L=len(album\_ratings)
    - L:8
- Sum
  - album\_ratings = [10,8.5,9.5,7.0,7.0,9.5,9.0,9.5]
  - S=sum(album\_ratings)
    - S=70
- Sorted vs Sort
  - album\_ratings = [10,8.5,9.5,7.0,7.0,9.5,9.0,9.5]
  - sorted\_album\_rating = sorted(album\_ratings) → album\_ratings won't change
    - sorted\_album\_rating = [7.0,7.0,8.5,9.0,9.5,9.5,9.5,10.0]
  - album\_ratings.sort()
    - album\_ratings: [7.0,7.0,8.5,9.0,9.5,9.5,9.5,10.0]
- Making functions

- `def add1(a):`
  - `b=a+1`
  - `return b`
- `add1(5)`
  - `b = 6`
- `c=add1(10)`
  - `c:11`
- Multiple parameters: a function can have multiple parameters
  - `def Mult(a,b):`
    - `c=a*b`
    - `return c`
  - `Mult(2,3)`
    - `6`
  - `Mult(10,3.14)`
    - `31.4`
  - `Mult(2,"Michael Jackson")`
    - `"Michael Jackson Michael Jackson"`
  - `def MJ():`
    - `print('Michael Jackson')`

`MJ()`  
`'Michael Jackson'`
  - `def NoWork():`
    - `pass`

`print(NoWork())`  
Python will return a NONE
  - Instead → `def NoWork():`
    - `pass`
    - `return None`
  - `def add1(a):`
    - `b=a+1;`
    - `print(a,"plus 1 equals ",b)`
    - `return b`
  - We can use loops in functions →  
`def printStuff(Stuff):`
    - `for i,s in enumerate(Stuff):`
      - `print("Album", i , "Rating is", s)`
    - `album_ratings = [10.0,8.5,9.5]`  
`printStuff(album_ratings)`
      - Album 0 Rating is 10
      - Album 1 Rating is 8.5
      - Album 2 Rating is 9.5
  - `def Print(A):`
    - `for a in A:`
      - `print(a+'1')`

```

    ■ Print(['a','b','c'])
    a1
    b1
    c1

```

- Collecting arguments

```

    ○ def ArtistNames(*names):
        ■ for name in names:
            ● print(name) →
                names=("Michael Jackson","AC/DC","PinkFloyd")
            ArtistNames("Michael Jackson","AC/DC","PinkFloyd")

```

- Scope: The scope of a variable is the part of the program where that variable is accessible. Variables that are defined outside of any function are said to be within the global scope, meaning they can be accessed anywhere after they are defined.

```

    ○ def AddDC(x):
        ■ x=x+"DC"
        ■ print(x)
        ■ return(x)
    x="AC"
    z=AddDC(x) → z="ACDC"

```

- Scope: Local Variables

```

    ○ Global Scope
    def Thriller():
        ■ Date=1982
        ■ return Date
    Thriller()
    Date → Does not exist within the Global Scope

```

```

    ○ Global Scope
    def Thriller():
        ■ Date=1982
        ■ return (Date)
    Date=2017
    print(Thriller())
    1982
    print(Date) → Date = 2017

```

- Scope: Variables

```

    ○ If a variable is not defined within a function, Python will check the
      global scope
    ○ Global Scope
    def ACDC(y):
        ■ print(Rating)
        ■ return(Rating+y)
    Rating=9
    Z=ACDC(1)

```

```

    9
    print(Rating)
    9

```

- Scope: Local Variables
  - Global Scope
 

```

def PinkFloyd():
    ■ global ClaimedSales
    ■ ClaimedSales = '45 million'
    ■ return ClaimedSales

PinkFloyd()
print(ClaimedSales)
45 million
          
```

## Exception Handling

- Try... Except...Else...Finally statement
  - try:
    - getfile=open("myfile","r")
    - getfile.write("My file for exception handling.")
  - except IOError:
    - print("Unable to open or read the data in the file.")
  - else:
    - print("The file was written successfully")
  - finally:
    - getfile.close()
    - print("File is now closed.")
- Try and Except : You can use the try and except blocks to prevent your program from crashing due to exceptions.
- Here's how they work:
  - The code that may result in an exception is contained in the try block.
  - If an exception occurs, the code directly jumps to except block.
  - In the except block, you can define how to handle the exception gracefully, like displaying an error message or taking alternative actions.
  - After the except block, the program continues executing the remaining code.

### Example: Attempting to divide by zero

```


1  # using Try- except
2  try:
3      # Attempting to divide 10 by 0
4      result = 10 / 0
5  except ZeroDivisionError:
6      # Handling the ZeroDivisionError and printing an error message
7      print("Error: Cannot divide by zero")
8  # This line will be executed regardless of whether an exception occurred
9  print("outside of try and except block")

```

## Objects and classes

- Python has lots of data types
- Types:
  - int: 1, 2, 567
  - float: 1.2, 0.62
  - String: 'a', 'abc', 'The cat is yellow'
  - List: [1,2, 'abc']
  - Dictionary: {"dog":1, "Cat":2}
  - Bool: False, True
- Each is an Object
- Built-in Types in Python
  - Every object has:
    - A type
    - An internal data representation (a blueprint)
    - A set of procedures for interacting with the object (methods)
  - An object is an instance of a particular type
- Objects: Type
  - You can find the type of an object by using the command type()
- Methods
  - A class or type's methods are functions that every instance of that class or type provides
  - It's how you interact with the data in an object
  - Sorting is an example of a method that interacts with the data in the object
  - Ratings=[10,9,6,5,10,8,9,6,2]  
Ratings.sort()
    - Ratings=[2,5,6,6,8,9,9,10,10]
  - Ratings.reverse()
    - Ratings=[10,10,9,9,8,6,6,5,2]
- Classes
  - Circle
    - Data Attributes: radius, color
  - Rectangle
    - Data Attributes: color, height and width
  - Create a class
    - Circle
      - Class Definition, Name of Class and Class parent →  
class Circle (object):
- Attributes and Objects
  - Object 1: Instance of type Circle
    - Data Attributes:
      - radius=4
      - color=red

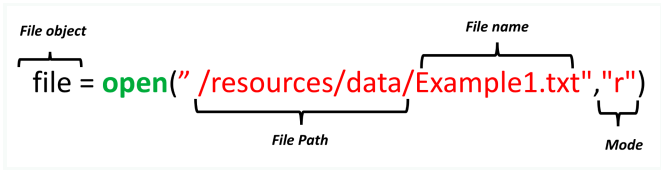
- Object 2: Instance of type Circle
  - Data Attributes:
    - radius=2
    - color=green
- Object 1: Instance of type Rectangle
  - Data Attributes:
    - height=2
    - width=2
    - color=blue
- Object 2: Instance of type Rectangle
  - Data Attributes:
    - height=1
    - width=3
    - color=yellow
- Create a class: Circle
  - class Circle(object): → Define your class
    - def \_\_init\_\_(self, radius, color):
      - self.radius=radius;
      - self.color=color; → Data attributes used to initialize each instance of the class
    - \_\_init\_\_ → Special function that tells Python you are making a new class (There are other special functions in Python to make more complex classes.)
    - self → Refers to the newly created instance of the class
    - radius, color → Special method or constructor used to initialize data attributes
    - radius and color → Can be used in the constructors body to access the values passed to the class constructor when the class is constructed
    - self.radius and self.color → We could set the value of the radius and color data attributes to the values passed to the constructor method
- Create a class: Rectangle
  - class Rectangle(object): → Define your class
    - def \_\_init\_\_(self, color, height, width):
      - self.height=height;
      - self.width=width
      - self.color=color; → Initialize the object's Data Attributes
- Create an Instance of a Class: Circle
  - How to create an object of class circle
  - RedCircle=Circle(10,"red") → (10,"red") are the attributes
  - class Circle(object): → Define your class
    - def \_\_init\_\_(self, 10, 'red'):
      - self.radius=10;

- `self.color='red';`
- 
- `RedCircle=Circle(10,'red')`
    - `RedCircle.radius=1`
    - This will change the radius to 1
  - `C1=Circle(10,"red")`
    - `C1.color="blue"`
    - This will change the color to blue
- **Methods**
    - Methods are functions that interact and change the data attributes, changing or using the data attributes of the object
    - `def add_radius(self,8)`
      - `self.radius=2+8`
      - `return (10)`
- ```
C1=Circle(2,'red')
C1.add_radius(8)
```

## WEEK 4

### Reading Files with Open

- 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:

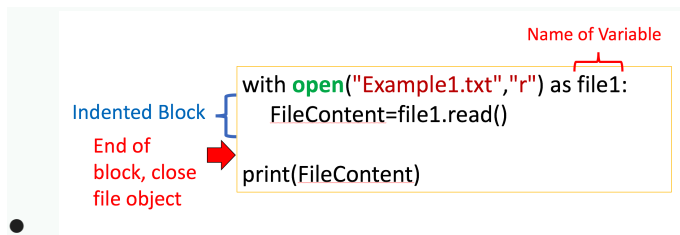


```
file = open("/resources/data/Example1.txt", "r")
```

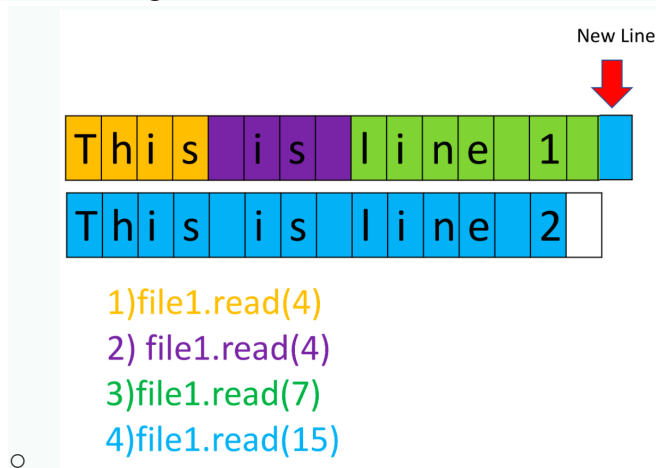
- r: Read mode for reading files
- w: Write mode for writing files
- An advantage of using the **with** statement to open a file is it automatically closes the file object
  - `with open("Example1.txt","r") as File1:`
    - `file_stuff=File.read()`
    - `print(file_stuff)`
  - `print(File1.closed)`
  - `print(file_stuff)`

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





- 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.



```
1 # Step 1: Open the file in read ('r') mode
2 file = open('file.txt', 'r')
3
4 # Step 2: Read the file content
5 content = file.read()
6
7 # Step 3: Process the content (e.g., print it)
8 print(content)
9
10 # Step 4: Close the file explicitly when done
11 file.close()
```

```
1 # Step 1: Open the file using 'with' in read ('r') mode
2 with open('file.txt', 'r') as file:
3     # Step 2: Read the file content within the 'with' block
4     content = file.read()
5
6     # Step 3: Process the content (e.g., print it)
7     print(content)
8
9 # Step 4: The file is automatically closed when the 'with' block exits
```

- Advantages of using **with** Method:
  - Automatic resource management: The file is guaranteed to be closed when you exit the `with` block, even if an exception occurs during processing.
  - Cleaner and more concise code: You don't need to explicitly call `close()`, making your code more readable and less error-prone.

```

1  # Reading and Storing the Entire Content of a File
2
3  # Using the read method, you can retrieve the complete content of a file
4  # and store it as a string in a variable for further processing or display.
5
6  # Step 1: Open the file you want to read
7  with open('File1.txt', 'r') as file:
8
9      # Step 2: Use the read method to read the entire content of the file
10     file_stuff = file.read()
11
12     # Step 3: Now that the file content is stored in the variable 'file_stuff',
13     # you can manipulate or display it as needed.
14
15     # For example, let's print the content to the console:
16     print(file_stuff)
17
18 # Step 4: The 'with' block automatically closes the file when it's done,
19 # ensuring proper resource management and preventing resource leaks.

```

- file = open('my\_file.txt', 'r')
  - line1 = file.readline() # Reads the first line
  - line2 = file.readline() # Reads the second line
  - print(line1) # Print the first line
  - if 'important' in line2:
    - print('This line is important!')
  - while True:
    - line = file.readline()
    - if not line:
      - break # Stop when there are no more lines to read
    - print(line)
  - file.close()

## Writing Files with Open

- with open("/resources/data/Example2.txt","w") as File1:
  - File1.write("This is line A")
- Additional 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 don't have to dwell on the specifics of each mode for this lab.

## Loading Data with Pandas

- Importing Pandas
  - import pandas → pre-built classes and functions like read\_csv, Series(), DataFrame, values, etc

- `csv_path='file1.csv'` → A CSV is a typical file type used to store data
- `df=pandas.read_csv(csv_path)`
- `import pandas as pd` → abbreviate pandas as pd so the code isn't tedious (we can use other terms like "banana")
  - `csv_path='file1.csv'` → This variable stores the path of the CSV, used as an argument to the `read_csv` function
  - `df=pd.read_csv(csv_path)` → The result is stored to the variable `df`, this is short for data frame
  - `df.head()` → 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 (the result is a data frame):
  - `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    |

- rows and columns
- We can create a data frame out of a dictionary (reminder; keys and values):
  - `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']}`
    - The keys correspond to the column labels (Album, Released, Length)
    - The values or lists corresponding to the rows (Thriller, 1982, 00:42:19)
  - `songs_frame=pd.DataFrame(songs)` → We then cast the dictionary to a data frame using the function `data frame`

|   | 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     |

- What python object do you cast to a dataframe?
  - Dictionary

- How would you access the first-row and first column in the dataframe df?
  - `df.ix[0,0]`
- We just put the data frame name, in this case, df, and the name of the multiple column headers enclosed in double brackets
  - `y=df[ ['Artist','Length','Genre'] ]`

|   | 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                   |

## Pandas: Working with and Saving Data

- List Unique Values
  - Pandas has the method unique to determine the unique elements in a column of a data frame.
    - 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.
    - `df['Released'].unique()`
  - 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.
  - We can use the inequality operators for the entire data frame in Pandas.
    - `df['Released']>=1980`
    - The result is a series of Boolean values.
  - We can select the specified columns in one line.
    - `df1=df[df['Released']>=1980]`

|   | 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 now have a new data frame, where each album was released after 1979
- We can save the new data frame using the method to\_csv.

- `df1.to_csv('new_songs.csv')` → The argument 'new\_songs.csv' is the name of the csv file (make sure to include .csv)

## Pandas practice

- What python object do you cast to a dataframe?
  - Dictionary
- How would you access the first-row and first column in the dataframe df?
  - `df.ix[0,0]`
- What is the proper way to load a CSV file using pandas?
  - `pandas.read_csv('data.csv')`
- Use this dataframe to answer the question

|   | 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    |

- How would you select the Genre disco? Select all that apply
  - `df.iloc[6,4]`
  - Wrong answers:
    - `df.iloc[6,'genre']`
    - `df.loc[6,5]`
    - `df.loc['Bee Gees','Genre']`
- Which will NOT evaluate to 20.6?
  - `df.loc[4,'Music Recording Sales']`
  - `df.iloc[6,'Music Recording Sales (millions)']`
  - Wrong answers:
    - `df.iloc[4,5]`
    - `df.iloc[6,5]`
- How do we select Albums The Dark Side of the Moon to Their Greatest Hits (1971-1975)?
  - `df.loc[2:5,'Album']`
  - `df.iloc[2:6,1]`

## One Dimensional Numpy

- The Basics & Array Creation
  - A numpy array or ND array is similar to a list [ ]
    - Fixed in size and integers

- `import numpy as np`  
`a=np.array([0,1,2,3,4])`
- The value of `a` is stored as follows → `a:array([0,1,2,3,4])`
  - `type(a): numpy.ndarray`
  - `a.dtype: dtype('int64')` →  
`dtype` to obtain the data type of the array's elements (64 bit int)
- `a:array([0,1,2,3,4])`
  - `a.size:5`
  - `a.ndim:1` → `ndim` is the number of array dimensions or the rank of the array (1)
  - `a.shape:(5)` → attribute `shape` is a tuple of integers indicating the size of the array in each dimension
- `b=np.array([3.1,11.,02,6.2,213.2,5.2])`
  - `type(b): numpy.ndarray`
  - `b.dtype: dtype('float64')` → elements are not integers, many other attributes
- Indexing and Slicing
  - `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])`
  - `d=c[1:4]`
    - `d:array([1,2,3])`
  - `c[3:5]=300,400`
    - `c:array([100,1,2,300,400])`
- Basic Operations: Vector Addition and Subtraction

$$\begin{aligned}
 \mathbf{u} &= \begin{bmatrix} 1 \\ 0 \end{bmatrix} & \mathbf{v} &= \begin{bmatrix} 0 \\ 1 \end{bmatrix} \\
 \mathbf{z} = \mathbf{u} + \mathbf{v} &= \begin{bmatrix} 1+0 \\ 0+1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}
 \end{aligned}$$

- `u=[1,0]`  
`v=[0,1]`  
`z=[]`
- `for n,m in zip(u,v):`
  - `z.append(n+m)`
- The numpy code will run much faster (especially if you have lots of data):

- `u=np.array([])`  
`v=np.array([])`  
`z=u+v`
  - `z:array([1,1])`
  - We can also use a subtraction sign → `z=u-v`, `z:array([1,-1])`

- Basic Operations: Array multiplication with a Scalar

$$\mathbf{y} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

$$\mathbf{z} = 2\mathbf{y} = \begin{bmatrix} 2(1) \\ 2(2) \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \end{bmatrix}$$

- 
- Vector multiplication with a scalar only requires one line of code using numpy:
- `y=np.array([1,2])`  
`z=2*y`
  - `z:array([2,4])`
- It would require multiple lines to perform the same task as shown:
- `y=[1,2]`  
`z=[]`  
for n in y:
  - `z.append(2*n)`

- Basic Operations: Product of two numpy arrays

$$\mathbf{u} = \begin{bmatrix} 1 \\ 2 \end{bmatrix} \quad \mathbf{v} = \begin{bmatrix} 3 \\ 2 \end{bmatrix}$$

$$\mathbf{z} = \mathbf{u} \circ \mathbf{v} = \begin{bmatrix} 1*3 \\ 2*2 \end{bmatrix} = \begin{bmatrix} 3 \\ 4 \end{bmatrix}$$

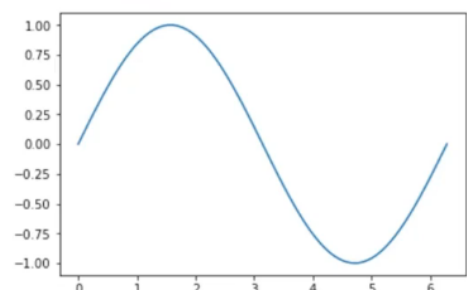
- 
- `u=np.array([1,2])`  
`v=np.array([3,2])`  
`z=u*v`
  - `z:array([3,4])`
- It would require multiple lines to perform the same task as shown:
- `u=[1,2]`  
`v=[3,2]`  
`z=[]`  
for n,m in zip(u,v)
  - `z.append(n*m)`

- Basic Operations: Dot product

$$\mathbf{u} = \begin{bmatrix} 1 \\ 2 \end{bmatrix} \quad \mathbf{v} = \begin{bmatrix} 3 \\ 1 \end{bmatrix}$$

$$\mathbf{u}^T \mathbf{v} = 1 \times 3 + 2 \times 1 = 5$$

- How would you perform the dot product between the numpy arrays  $\mathbf{u}$  and  $\mathbf{v}$ ?
  - `np.dot(u,v)`
- `u=np.array([1,2])`  
`v=np.array([3,1])`  
`result=np.dot(u,v)`  
 ■ `result:5`
- Basic Operations: Adding constant to a numpy Array
  - `u=np.array([1,2,3,-1])`  
 ■ `z=u+1`  
 ■ `z:array([2,3,4,0])`
- Universal Functions → operates on ND arrays
  - `a=np.array([1,-1,1,-1])`  
`mean_a=a.mean()`  
 ■ `mean_a:0.0`  
 ■  $\frac{1}{4}(1-1+1-1)=0$
  - `b=np.array([1,-2,3,4,5])`  
`max_b=b.max()`  
 ■ `max_b:5`
  - `np.pi` →  $\pi$   
`x=np.array([0,np.pi/2,np.pi])` → `x=[0,π/2,π]`  
`y=np.sin(x)` → `y=[sin(0),sin(π/2),sin(π)]`  
 ■ `y:array([0,1,1.2e-16])` → `y=[0,1,0]`
- Plotting Mathematical Functions
  - A useful function for plotting mathematical functions is `linspace`.  
`Linspace` returns evenly spaced numbers over specified interval
  - `np.linspace(-2,2,num=5)` → (starting point, ending point, and parameter `num` indicates the number of samples to generate, hence the space between samples is 1)
  - `np.linspace(-2,2,num=9)` → the space between samples is 0.5
  - `x=np.linspace(0,2*np.pi,100)` → x axis  
`y=np.sin(x)` → y axis  
`import matplotlib.pyplot as plt`  
`%matplotlib inline (for Jupyter Notebook)`  
`plt.plot(x,y)`





## Two Dimensional Numpy

- `a = [[11,12,13],[21,22,23],[31,32,33]]`
  - `A = np.array(a)`

A:  $\begin{bmatrix} 11 & 12 & 13 \\ 21 & 22 & 23 \\ 31 & 32 & 33 \end{bmatrix}$

- 
- `A.ndim:2`
  - Ndim (key words: number dimension) is used to obtain the number of axes or dimensions referred to as the rank
- `A.shape(3,3)`
  - First element is the amount of nested lists (rows)
  - Second element is the size of each of the nested list (columns)
- `A.size:9`
- `A: [[A[0][0],A[0][1],A[0][2],A[1][0],A[1][1],A[1][2],A[2][0],A[2][1],A[2][2]]]`

$\begin{bmatrix} A[0][0] & A[0][1] & A[0][2] \\ A[1][0] & A[1][1] & A[1][2] \\ A[2][0] & A[2][1] & A[2][2] \end{bmatrix}$

- - 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

`A[1][2]:23`

|   | 0  | 1  | 2  |
|---|----|----|----|
| 0 | 11 | 12 | 13 |
| 1 | 21 | 22 | 23 |
| 2 | 31 | 32 | 33 |

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

`A[0:2,2]:array([13, 23])`

|   | 0  | 1  | 2  |
|---|----|----|----|
| 0 | 11 | 12 | 13 |
| 1 | 21 | 22 | 23 |
| 2 | 31 | 32 | 33 |

- Sum/Addition

- $X = \text{np.array}([[1,0],[0,1]])$   
 $Y = \text{np.array}([[2,1],[1,2]])$   
 $Z = X+Y$ 
  - $Z = ([[3,1],[1,3]])$
- Multiply
  - $Y = \text{np.array}([[2,1],[1,2]])$   
 $Z = 2*Y$ 
    - $Z = ([[4,2],[2,4]])$
- Hadamard Product
  - $X = \text{np.array}([[1,0],[0,1]])$   
 $Y = \text{np.array}([[2,1],[1,2]])$   
 $Z = X*Y$ 
    - $Z = ([[2,0],[0,2]])$
- Matrix multiplication

$$A = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix} \quad B = \begin{bmatrix} 1 & 1 \\ 1 & 1 \\ -1 & 1 \end{bmatrix}$$

$0 \times 1 + 1 \times 1 + 1 \times -1$

○  $AB = \begin{bmatrix} \quad \end{bmatrix}$

$$A = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix} \quad B = \begin{bmatrix} 1 & 1 \\ 1 & 1 \\ -1 & 1 \end{bmatrix}$$

$0 \times 1 + 1 \times 1 + 1 \times 1 = 2$

○  $AB = \begin{bmatrix} 0 & 2 \end{bmatrix}$

$$A = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix} \quad B = \begin{bmatrix} 1 & 1 \\ 1 & 1 \\ -1 & 1 \end{bmatrix}$$

$1 \times 1 + 0 \times 1 + 1 \times 1 = 0$

○  $AB = \begin{bmatrix} 0 & 2 \\ 0 & \end{bmatrix}$

$$A = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix} \quad B = \begin{bmatrix} 1 & 1 \\ 1 & 1 \\ -1 & 1 \end{bmatrix}$$

$1 \times 1 + 0 \times 1 + (1) \times 1 = 2$

○  $AB = \begin{bmatrix} 0 & 2 \\ 0 & 2 \end{bmatrix}$

- $A = \text{np.array}([[0,1,1],[1,0,1]])$   
 $B = \text{np.array}([[1,1],[1,1],[-1,1]])$   
 $C = \text{np.dot}(A,B);$ 
  - $C : \text{array}([[0,2],[0,2]])$

## WEEK 5

### **Simple APIs (Part 1)**

- What is an API?
  - You use the api to communicate with the api via inputs and outputs.
  - When you create a dictionary, and then create a pandas object with the Dataframe constructor, in API lingo, this is an “instance.”
  - You then use the dataframe to communicate with the API. When you call the method head, the dataframe communicates with the API displaying the first few rows of the dataframe.
  - When you call the method, the API will calculate the mean and return the values.
- REST APIs
  - They allow you to communicate through the internet allowing you to take advantage of resources like storage, access more data, artificial intelligent algorithms, and much more.
  - The RE stands for Representational, the S stands for State, the T stands for Transfer.
  - Rest API's function by sending a request, the request is communicated via HTTP message. The HTTP message usually contains a JSON file. This contains instructions for what operation we would like the service or resource to perform. In a similar manner, API returns a response, via an HTTP message, this response is usually contained within a JSON.
  - You or your code can be thought of as a client. The web service is referred to as a resource. The client finds the service via an endpoint. We will review this more in the next section. The client sends requests to the resource and the resource (web service) sends a response to the client. HTTP methods are a way of transmitting data over the internet. We tell the Rest API's what to do by sending a request. The request is usually communicated via an HTTP message. The HTTP message usually contains a JSON file. This contains instructions for what operation we would like the service to perform. This operation is transmitted to the webservice via the internet. The service performs the operation. In the similar manner, the webservice returns a response via an HTTP message, where the information is usually returned via a JSON file. This information is transmitted back to the client.

```

!pip install pycoingecko
from pycoingecko import CoinGeckoAPI
cg = CoinGeckoAPI()
bitcoin_data = cg.get_coin_market_chart_by_id(id='bitcoin', vs_currency='usd', days=30)

```

```

bitcoin_data:
{'prices':
[[[1604138559526,
13927.73252295053],
[1604142031697,
13797.785093120927],
[1604145789426,
13917.92602548896],
[1604150065202,
13873.651848313166],
[1604152981403,
13845.28388774076],
[1604156700430,
13806.349989300157],...]]

bitcoin_data['prices']:
[[[1604138559526, 13927.73252295053],
[1604142031697, 13797.785093120927],
[1604145789426, 13917.92602548896],
[1604150065202, 13873.651848313166],
[1604152981403, 13845.28388774076]]]

```



```
data = pd.DataFrame(bitcoin_price_data, columns=['TimeStamp', 'Price'])
```

|     | TimeStamp     | Price        | Date                    |
|-----|---------------|--------------|-------------------------|
| 0   | 1604138559526 | 13927.732523 | 2020-10-31 10:02:39.526 |
| 1   | 1604142031697 | 13797.785093 | 2020-10-31 11:00:31.697 |
| 2   | 1604145789426 | 13917.926025 | 2020-10-31 12:03:09.426 |
| 3   | 1604150065202 | 13873.651848 | 2020-10-31 13:14:25.202 |
| 4   | 1604152981403 | 13845.283888 | 2020-10-31 14:03:01.403 |
| ... | ...           | ...          | ...                     |
| 716 | 1606716898166 | 18510.032786 | 2020-11-30 06:14:58.166 |
| 717 | 1606720068286 | 18509.742863 | 2020-11-30 07:07:48.286 |
| 718 | 1606724609861 | 18439.969378 | 2020-11-30 08:23:29.861 |
| 719 | 1606727338318 | 18453.345378 | 2020-11-30 09:08:58.318 |
| 720 | 1606728336000 | 18465.880043 | 2020-11-30 09:25:36.000 |

to\_datetime



```
data['Date'] = pd.to_datetime(data['TimeStamp'], unit='ms')
```

|     | TimeStamp     | Price        | Date                    |
|-----|---------------|--------------|-------------------------|
| 0   | 1604138559526 | 13927.732523 | 2020-10-31 10:02:39.526 |
| 1   | 1604142031697 | 13797.785093 | 2020-10-31 11:00:31.697 |
| 2   | 1604145789426 | 13917.926025 | 2020-10-31 12:03:09.426 |
| 3   | 1604150065202 | 13873.651848 | 2020-10-31 13:14:25.202 |
| 4   | 1604152981403 | 13845.283888 | 2020-10-31 14:03:01.403 |
| ... | ...           | ...          | ...                     |
| 716 | 1606716898166 | 18510.032786 | 2020-11-30 06:14:58.166 |
| 717 | 1606720068286 | 18509.742863 | 2020-11-30 07:07:48.286 |
| 718 | 1606724609861 | 18439.969378 | 2020-11-30 08:23:29.861 |
| 719 | 1606727338318 | 18453.345378 | 2020-11-30 09:08:58.318 |
| 720 | 1606728336000 | 18465.880043 | 2020-11-30 09:25:36.000 |

```
candlestick_data = data.groupby(data.Date.dt.date).agg({'Price': ['min', 'max', 'first', 'last']})
```


|            | Price        |              |              |              |
|------------|--------------|--------------|--------------|--------------|
|            | min          | max          | first        | last         |
| Date       |              |              |              |              |
| 2020-10-31 | 13720.425685 | 13917.926025 | 13797.785093 | 13796.028786 |
| 2020-11-01 | 13661.927858 | 13822.470734 | 13778.637638 | 13671.267179 |
| 2020-11-02 | 13313.920358 | 13786.247252 | 13786.247252 | 13554.676070 |
| 2020-11-03 | 13359.343536 | 13847.027174 | 13558.361796 | 13847.027174 |
| 2020-11-04 | 13588.251443 | 14122.045525 | 13973.977815 | 14120.620632 |
| 2020-11-05 | 14095.428885 | 15511.856787 | 14170.702323 | 15511.856787 |
| 2020-11-06 | 15413.862067 | 15855.288073 | 15570.112495 | 15619.662967 |
| 2020-11-07 | 14630.212278 | 15679.761910 | 15583.677358 | 14926.198182 |
| 2020-11-08 | 14749.879849 | 15561.599966 | 14749.879849 | 15506.766777 |

## Simple APIs (Part 2)

- API keys and endpoints
  - They will give you access to the API
  - Like many passwords, you should keep your API key secret
- Watson Speech to Text
  - filename='hello\_this\_is\_python.wav'  
with open(filename,mode="rb") as wav: → rb = read in binary
    - response = s2t.recognize(audio=wav, content\_type='audio/wav')
      - method recognize = send the audio to Watson Speech to Text service
      - parameter audio is the file object
      - content\_type is the audio file format
  - The attribute result (response.result contains a python dictionary
    - We are interested in the key 'transcript'
    - We can assign it to the variable recognize\_text as follows
    - recognized\_text: 'hello this is python'

```
response.result
{'results': [{'alternatives': [{'confidence': 0.91, 'transcript': 'hello this is python '}, {'final': True}], 'result_index': 0}]
recognized_text=response.result['results'][0]['alternatives'][0]['transcript']

recognized_text:
'hello this is python '
```



- Watson Language Translator

## REST APIs & HTTP Requests - Part 1

- Uniform Resource Locator (URL)
  - Most popular way to find resources on the web
  - Scheme: the protocol (http://)
  - Internet address or Base URL ([www.ibm.com](http://www.ibm.com))
  - Route: location on the web server (/images/IDSNlogo.png)
- Request Message
  - In the start line we have the GET method, this is an HTTP method
    - Here, it's requesting the file index.html
  - The request header passes additional information with an HTTP request
  - In the GET method the request header is empty

|                    |                                                                           |
|--------------------|---------------------------------------------------------------------------|
| Request Start line | Get/index.html HTTP/1.0                                                   |
| Request Header     | User-Agent: python-requests/2.21.0<br>Accept-Encoding: gzip, deflate<br>: |

- Some requests have a body

- Response Message

- The response start line contains the version number followed by a descriptive phrase, in this case, HTTP/1.0 a status code 200 meaning success and the descriptive phase 'OK'
- The response header contains information
- Finally, we have the response body containing the requested file, in this case an HTML document

|                     |                                                                                                                      |
|---------------------|----------------------------------------------------------------------------------------------------------------------|
| Response Start line | HTTP/1.0 200 OK                                                                                                      |
| Response Header     | Server: Apache-Cache: UNCACHEABLE                                                                                    |
| Response Body       | <!DOCTYPE html><br><html><br><body><br><h1>My First Heading</h1><br><p>My first paragraph.</p><br></body><br></html> |

- Status Code

- The 100s are informational responses; indicates that everything is OK so far
- The 200s are successful responses
- Anything in the 400s is bad news. 401 means the request is unauthorized
- 500s stands for server errors, like 501 for not implemented

|     |                  |
|-----|------------------|
| 1XX | Informational    |
| 2xx | Success          |
| 200 | OK               |
| 3XX | Redirection      |
| 300 | Multiple Choices |
| 4XX | Client Error     |
| 401 | Unauthorized     |
| 403 | Forbidden        |
| 404 | Not Found        |

- HTTP Methods

- When an HTTP request is made, an HTTP method is sent. This tells the server what action to perform
  - GET: Retrieves data from the server
  - POST: Submits data to the server
  - PUT: Updates data already on server
  - DELETE: Deletes data from server

## REST APIs & HTTP Requests - Part 2

- Requests in Python

- Requests is a Python library that allows you to send HTTP/1.1 requests easily
- import requests  
url = 'https://www.ibm.com/'  
r=requests.get(url)  
r.status\_code:200  
r.request.headers
- r.request.body: None → As there is no body for a GET request, we get a None
- header=r.headers → You can view the HTTP response header using the attribute headers. This returns a Python dictionary of HTTP response headers. We can look at the dictionary values.
  - header['date']: 'Thu, 19 Nov 2020 15:21:47 GMT' → We can obtain the date request was sent using the key 'date'
  - header['Content-Type']: 'text/html; charset=UTF-8' → The key Content-Type indicates the type of data
  - r.encoding: 'UTF-8' → Using the response object 'r', we can also check the encoding
  - r.text[0:100] → As the Content-Type is text/html, we can use the attribute text to display the HTML in the body. We can review the first 100 characters. You can also download other content.

- Get Request

- We send a GET request to the server. Like before, we have the Base URL in the Route; we append /get. This indicates we would like to perform a GET request. This is demonstrated in the following table:

| Base URL                                             | Route                   |
|------------------------------------------------------|-------------------------|
| <a href="http://httpbin.org">httpbin.org</a>         | <a href="/get">/get</a> |
| <a href="http://httpbin.org/get">httpbin.org/get</a> |                         |

- Query string: After GET is requested we have the query string. This is a part of a URL and this sends other information to the web server
  - The start of the query is a ?, followed by a series of parameter and value pairs
  - The first parameter name is "Name" and the value is Joseph
  - The second parameter name is "ID" and the value is 123
  - Each pair, parameter and value is separated by an equal sign, "=". The series of pairs is separated by the ampersand &

| Start of Query                                                                                            | Parameter Name | Value    | Parameter Name | Value |
|-----------------------------------------------------------------------------------------------------------|----------------|----------|----------------|-------|
| ?                                                                                                         | name           | = Joseph | ID             | = 123 |
| <a href="http://httpbin.org/get?Name=Joseph&amp;ID=123">http://httpbin.org/get?Name=Joseph&amp;ID=123</a> |                |          |                |       |

- Create Query String
  - url\_get='http://httpbin.org/get'
  - payload={"name": "Joseph", "ID": "123"}

- `r=requests.get(url_get.params=payload)`
  - `r.url:'http://httpbin.org/get?name=Joseph&ID=123'`
  - `r.request.body: None`
  - `r.status_code: 200`
  - `r.text: very big text`
  - `r.headers['Content-Type']: 'application/json'`
    - `r.json()`
- Post Request
  - Like a GET request, a POST request is used to send data to a server, but the POST request sends the data in a request body, not the URL
  - In order to send the Post Request in the URL, we change the route to POST
- POST
  - `url_post="http://httpbin.org/post"`
    - This endpoint will expect data and it is a convenient way to configure an HTTP request to send data to a server
  - `payload={"name":"Joseph","ID":"123"}`
    - We have the payload dictionary
  - `r_post=requests.post(url_post,data=payload)`
    - To make a POST request, we use the `post()` function. The variable payload is passed to the parameter data
- Compare POST and GET
  - Comparing the URL using the attribute `url` from the response object of the GET and POST request
    - `print("POST request URL: ",r_post.url)`
    - `print("GET request URL: ",r.url)`
    - POST request body: `name=Joseph&ID=123`
    - GET request body: `None`
  - And finally, we can view the key form to get the payload
    - `r_post.json()['form']`  
`{'ID':'123', 'name':Joseph''}`

## API Examples (Labs)

- Get Methods
  - `get_cell(), get_city(), get_dob(), get_email(), get_first_name(), get_full_name(), get_gender(), get_id(), get_id_number(), get_id_type(), get_info(), get_last_name(), get_login_md5(), get_login_salt(), get_login_sha1(), get_login_sha256(), get_nat(), get_password(), get_phone(), get_picture(), get_postcode(), get_registered(), get_state(), get_street(), get_username(), get_zipcode()`

## Optional: HTML for Webscraping

- Table of Contents:



- Review the HTML of a basic web page
- Understand the Composition of an HTML Tag
- Understand HTML Trees
- Understand HTML Tables
- HTML Composition: It consists of text surrounded by a series of blue text elements enclosed in angle brackets called tags. The tags tell the browser how to display the content.
  - Document Type Declaration (DOCTYPE): The HTML document usually starts with a DOCTYPE declaration, which defines the version of HTML being used. For HTML5, the DOCTYPE declaration is as follows:
  - <!DOCTYPE html>
  - HTML Element: The <html> element is the root element of an HTML document and encapsulates the entire content of the page. It contains two main sections: the <head> and the <body>.
    - <head>: The <head> section contains metadata about the document, such as the page title, character encoding, linked stylesheets, scripts, and other information that doesn't directly appear on the visible page.
    - <body>: The <body> section contains the visible content of the webpage, including text, images, links, and other elements that are displayed to the user.
  - Page Title: The <title> element within the <head> section defines the title of the web page, which appears in the browser's title bar or tab.
    - <head>
      - <title>My Web Page</title>
    - </head>
  - Metadata and Links: Within the <head> section, you can include metadata such as the character encoding, author information, and links to external resources like stylesheets and icons.
    - <meta charset="UTF-8">
    - <meta name="author" content="John Doe">
    - <link rel="stylesheet" type="text/css" href="styles.css">
    - <link rel="icon" href="favicon.ico" type="image/x-icon">
  - Content Elements: Within the <body> section, you use various HTML elements to structure and display content. Some common elements include:
    - Headings (<h1>, <h2>, <h3>, etc.): Used for headings and subheadings.
    - Paragraphs (<p>): Used for text paragraphs.
    - Lists (<ul>, <ol>, <li>): Used to create unordered and ordered lists.
    - Links (<a>): Used to create hyperlinks to other pages or resources.
    - Images (<img>): Used to display images.

- Forms (<form>): Used to collect user input.
  - Tables (<table>, <tr>, <td>): Used to create structured data tables.
- Attributes: Elements can have attributes that provide additional information or modify their behavior. For example, the <a> element can have an href attribute to specify the link's destination.
  - <a href="https://www.example.com">Visit Example.com</a>
- Comments: You can add comments to your HTML code using <!-- ... -->. Comments are not visible to users but can provide explanations or notes for developers.
  - <!-- This is a comment -->
- Whitespace: HTML is forgiving when it comes to whitespace (spaces, tabs, and line breaks), so you can format your code for readability. However, excessive whitespace doesn't affect the rendered page.
- Script Elements: JavaScript code can be included within <script> elements in the <head> or <body> sections to add interactivity and functionality to the webpage.
  - <script>
    - function greet() {
      - alert('Hello, World!');
    - }
  - </script>
- Composition of an HTML Tag
  - Hyperlink Tag: <a
  - Opening and End Tags: <a href="link"> and </a>
  - Hyperlink Content: > IBM webpage </a>
  - Attributes
    - Attribute Name: href=
    - Attribute Value: "link"
- HTML Trees
  - <head>
    - <title>
  - <body>
    - <h3>, <p>, <h3>, <p>, <h3>, <p>
- HTML Tables
  - <tr>
    - <td> Pizza Place </td>
    - <td> Orders </td>
    - <td> Slices </td>
</tr>
  - <tr>
    - <td> Domino's Pizza </td>
    - <td> 10 </td>
    - <td> 100 </td>

</tr>

| Pizza Place    | Orders | Slices |
|----------------|--------|--------|
| Domino's Pizza | 10     | 100    |
| Little Caesars | 12     | 144    |

○

## Webscraping

- What is Webscraping
  - Webscraping is a process that can be used to automatically extract information from a website, and can easily be accomplished within a matter of minutes and not hours.
- BeautifulSoup
  - from bs4 import BeautifulSoup
  - html = " asfgaedgnwef "
  - soup = BeautifulSoup(html,'html5lib') → Soup object represents the document as a nested data structure
- BeautifulSoup Objects: BeautifulSoup is a Python library for pulling data out of HTML and XML files, we will focus on HTML files.
  - Tag Object

tag\_object=soup.title

tag\_object:

■ <title>Page Title</title>

### Tag Object

tag\_object=soup.title  
tag\_object:  
<title>Page Title</title>

tag\_object=soup.h3

tag\_object:  
<h3><b id="boldest">Lebron James</b></h3>

```
<!DOCTYPE html>
<html>
<head>
<title>Page Title</title>
</head>
<body>
<h3><b id="boldest">Lebron James</b></h3>
<p> Salary: $ 95,000,000 </p>
<h3> Stephen Curry</h3>
<p> Salary: $85,000, 000 </p>
<h3> Kevin Durant </h3>
<p> Salary: $73,200, 000</p>
</body>
</html>
```

○

- Parent attribute

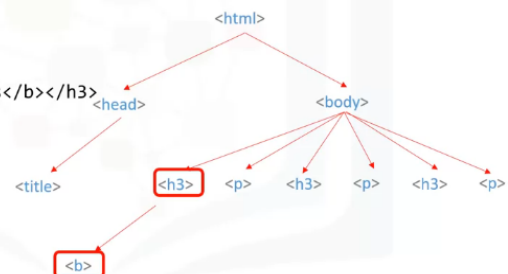
tag\_child:<b id="boldest">Lebron James</b>

parent\_tag=tag\_child.parent  
parent\_tag:

<h3><b id="boldest">Lebron James</b></h3>

tag\_object

■

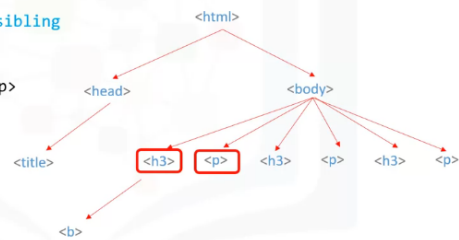


- Next-sibling attribute

```
tag_object: <h3><b id="boldest">Lebron James</b></h3>
```

```
sibling_1= tag_object.next_sibling
sibling_1:
```

```
<p> Salary: $ 92,000,000 </p>
```



- Navigable string

```
tag_child:<b id="boldest">Lebron James</b>
```

```
tag_child.attrs:
```

```
{'id': 'boldest'}
```

```
tag_child.string:
```

```
'Lebron James'
```

```
<!DOCTYPE html>
<html>
<head>
<title>Page Title</title>
</head>
<body>
<h3><b id='boldest'> Lebron James</b></h3>
<p> Salary: $ 92,000,000 </p>
<h3> Stephen Curry</h3>
<p> Salary: $85,000, 000 </p>
<h3> Kevin Durant </h3>
<p> Salary: $73,200, 000</p>
</body>
</html>
```

- find\_all

- This is a filter, you can use filters to filter based on a tag's name, it's attributes, the text of a string, or on some combination of these
- Consider the list of pizza places from the last video
- Create a BeautifulSoup Object
- Python iterable

```
table_row=table.find_all(name='tr')
```

```
table_row:
```

Pizza Place	Orders	Slices
Domino's Pizza	10	100
Little Caesars	12	144

```
[<tr><td>Pizza Place</td><td>Orders</td><td>Slices</td></tr>,
<tr><td>Domino's Pizza</td><td>10</td><td>100</td></tr>,
<tr><td>Little Caesars</td><td>12</td><td>144</td></tr>,
```

- Tag object

```
first_row =table_row[0]
first_row:
```

```
<tr><td>Pizza Place</td><td>Orders</td> <td>Slices </td></tr>
```

```
first_row.td :
```

```
<td>Pizza Place</td>
```

- Variable row / Elements

Pizza Place	Orders	Slices
Domino's Pizza	10	100
Little Caesars	12	144
Papa John's	15	166

```
for i,row in enumerate(table_rows):
    print("row", i)
    cells=row.find_all("td")
    for j,cell in enumerate(cells):
        print("column", j , "cell", cell)
```

- First, we iterate through the list “table rows,” via the variable row.
- Each element corresponds to a row in the table.
- We can apply the method find all to find all the table cells, then we can iterate through the variable cells for each row.
- For each iteration, the variable cell corresponds to an element in the table for that particular row.
- We continue to iterate through each element and repeat the process for each row.

## Working with different file formats (csv,xml,json,xlsx)

- Objectives
  - Define different file formats such as csv, xml, and json
  - Write simple programs to read and output data
  - List what Python libraries are needed to extract data
- Csv: Python Libraries
  - Pandas: by importing this library in the beginning of the code, we are then able to easily read the different file types
    - import pandas as pd
    - file="FileExample.csv"
    - df=pd.read\_csv(file)
- Using Dataframes
  - To fix the csv file problem of no headers, we can do it like this:
    - df.columns=['Name','Phone Number','Birthday']

	Name	Phone Number	Birthday
1	Bill Smith	258-541-2598	5/25/1984
2	Jerod Rue	857-968-8542	12/5/1986
3	Kris Ann	875-256-1452	5/17/1978

- Reading JSON files
  - Similar to a Python dictionary
    - import json
    - with open('filesample.json','r') as openfile:
      - json\_object=json.load(openfile)
- Reading XML files

- While the pandas library does not have an attribute to read this type of file let us explore how to parse this type of file

```

■ import pandas as pd
■ import xml.etree.ElementTree as etree
■ tree = etree.parse("fileExample.xml")
■ root=tree.getroot()
■ columns=["Name","Phone Number","Birthday"]
■ df=pd.DataFrame(columns=columns)
■ _____
■ AND THEN
■ _____

for node in root:
    name = node.find("name").text
    phonenumber = node.find("phonenumber").text
    birthday = node.find("birthday").text

    df = df.append(pd.Series([name, phonenumber,
    birthday], index = columns)
    ..., ignore_index = True)

```

### Practice Quiz

- What is the function of “GET” in HTTP requests?
  - Carries the request to the client from the requestor
- What does URL stand for?
  - Uniform Resource Locator
- What does the file extension “csv” stand for?
  - Comma Separated Values
- What is webscraping?
  - The process to extract data from a particular website

### Module 5 Graded Quiz

- What are the 3 parts to a response message?
  - Start or status line, header, and body
- What is the purpose of this line of code “table\_row=table.find\_all(name='tr')” used in webscraping?
  - It will find all of the data within the table marked with a tag “tr”
- In what data structure do HTTP responses generally return?
  - JSON
- The Python library we used to plot the chart in video/lasb is
  - Matplotlib

<https://niyander.blogspot.com/2021/06/Python%20for%20Data%20Science%20AI%20Development%20Final%20Exam%20Quiz%20Answers.html>