



# LAB 2: FINISHING UP PYTHON BASICS, NUMPY & WORKING WITH AUDIO DATA

University of Washington

ECE 241

Winter 2022

Author: Jimin Kim ([jk55@uw.edu](mailto:jk55@uw.edu))

Version: v1.5.0

# OUTLINE

## Part 1: Finishing up Python Basics

- Conditional Statements
- Loops
- Functions

## Part 2: Introduction to Numpy

- Numpy arrays
- Array operations in Numpy
- Useful Numpy functions
- Math operations with Numpy

## Part 3: Plotting with matplotlib

- Basic plotting
- Labeling your plots
- Multiple plots
- Subplots

## Part 4: Audio I.O.

- Digital audio data
- Read/write/play audio files

## Part 5: Lab Assignments

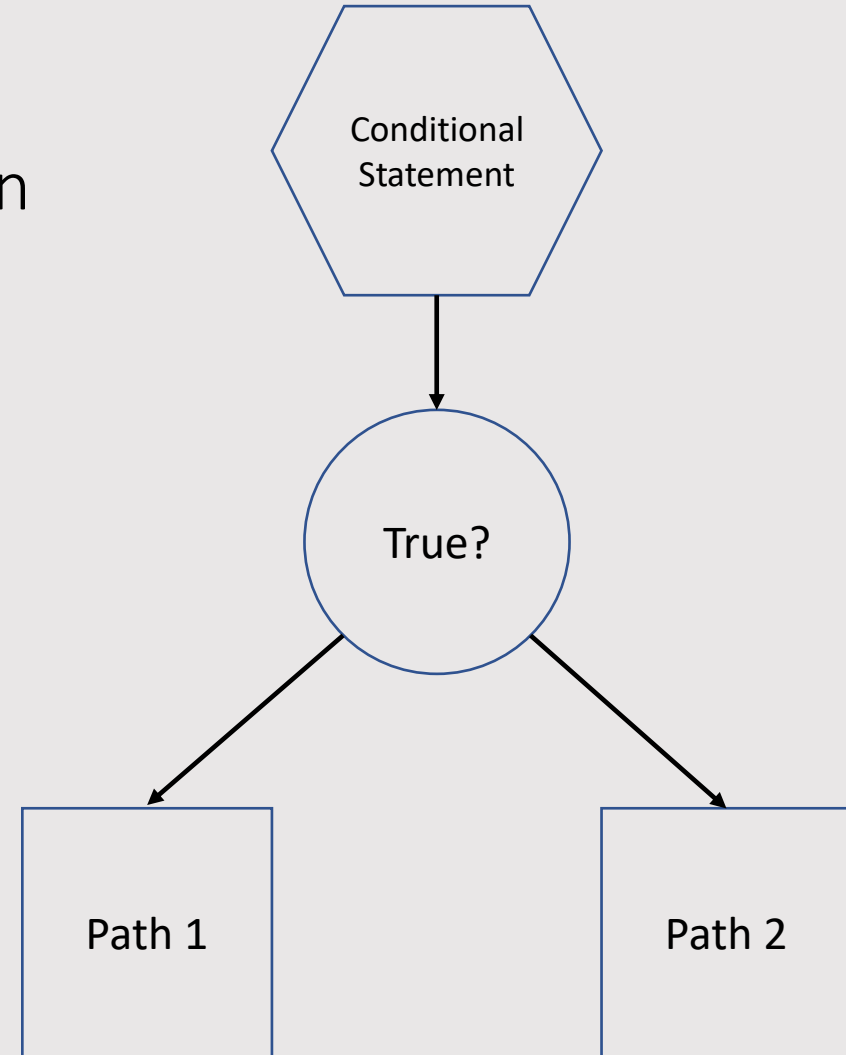
- Exercise 1 – 5

# CONDITIONAL STATEMENTS, FUNCTIONS, LOOPS

# CONDITIONAL STATEMENTS

Types of conditional statements in Python

- If
- If-else
- If-elif-else



# CONDITIONAL STATEMENTS: if

Implementation structure

If **condition**:

Code to be executed

Code example

```
In [11]: num1 = 10
          num2 = 20

          if num1 < num2: # equivalently, if (num1 < num2) == True
              print('num2 is larger than num1')

          num2 is larger than num1
```

```
In [2]: if type(num1) == int:
          print('num1 is integer')

          num1 is integer
```

# CONDITIONAL STATEMENTS: if-else

## Implementation structure

If **condition**:

Execute this code

else:

Execute this code  
instead

## Code example

```
In [5]: num1 = 20
        num2 = 10

        if num1 < num2:

            print('num2 is larger than num1')

        else:

            print('num2 is less or equal to num1')

num2 is less or equal to num1
```

# CONDITIONAL STATEMENTS: if-elif-else

## Implementation structure

If **condition 1**:

Execute this code

elif **condition 2**:

Execute this code  
instead

else:

Execute this code  
instead

## Code example

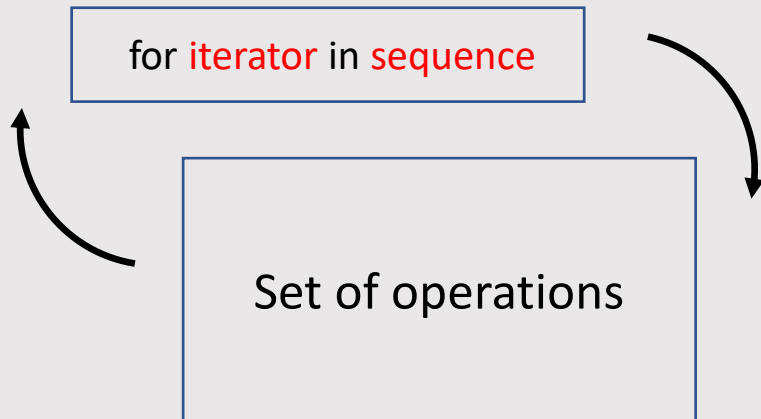
```
In [7]: num1 = 20

if type(num1) == float:
    print('num1 is float')
elif type(num1) == bool:
    print('num1 is boolean')
else:
    print('num1 is neither float nor boolean')

num1 is neither float nor boolean
```

Note: You can have multiple elif conditions between if and else

# LOOPS: for LOOP



```
for i in range(1, 11): # A sequence from 1 to 10

    if i % 2 == 0:
        print(i, " is even")
    else:
        print(i, " is odd")
```

```
1  is odd
2  is even
3  is odd
4  is even
5  is odd
6  is even
7  is odd
8  is even
9  is odd
10 is even
```

Iterate through sequence

```
# For Loop - Iterate through list elements
```

```
float_list = [2.5, 16.42, 10.77, 8.3, 34.21]
```

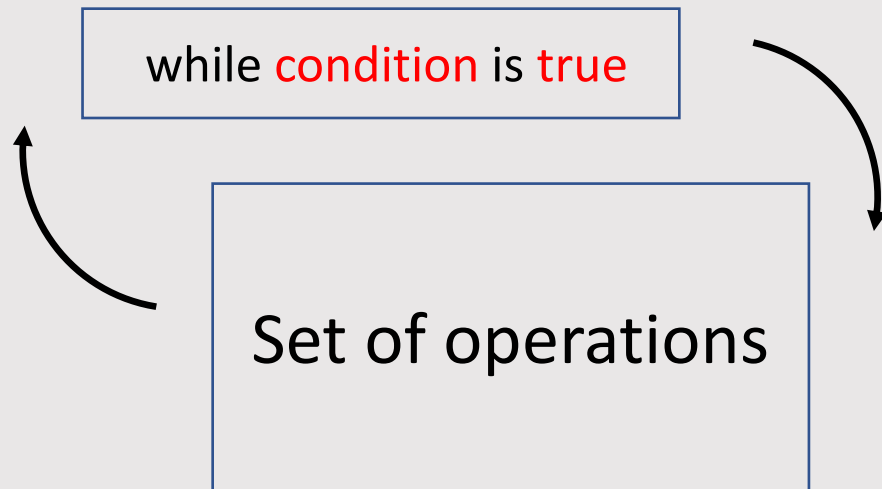
```
for num in float_list: # Iterator goes through each item in the list
    print([num, num * 2])
```

```
[2.5, 5.0]
[16.42, 32.84]
[10.77, 21.54]
[8.3, 16.6]
[34.21, 68.42]
```

Iterate through list elements



# LOOPS: while LOOP



Note: while loop has a potential to run infinitely if not set correctly

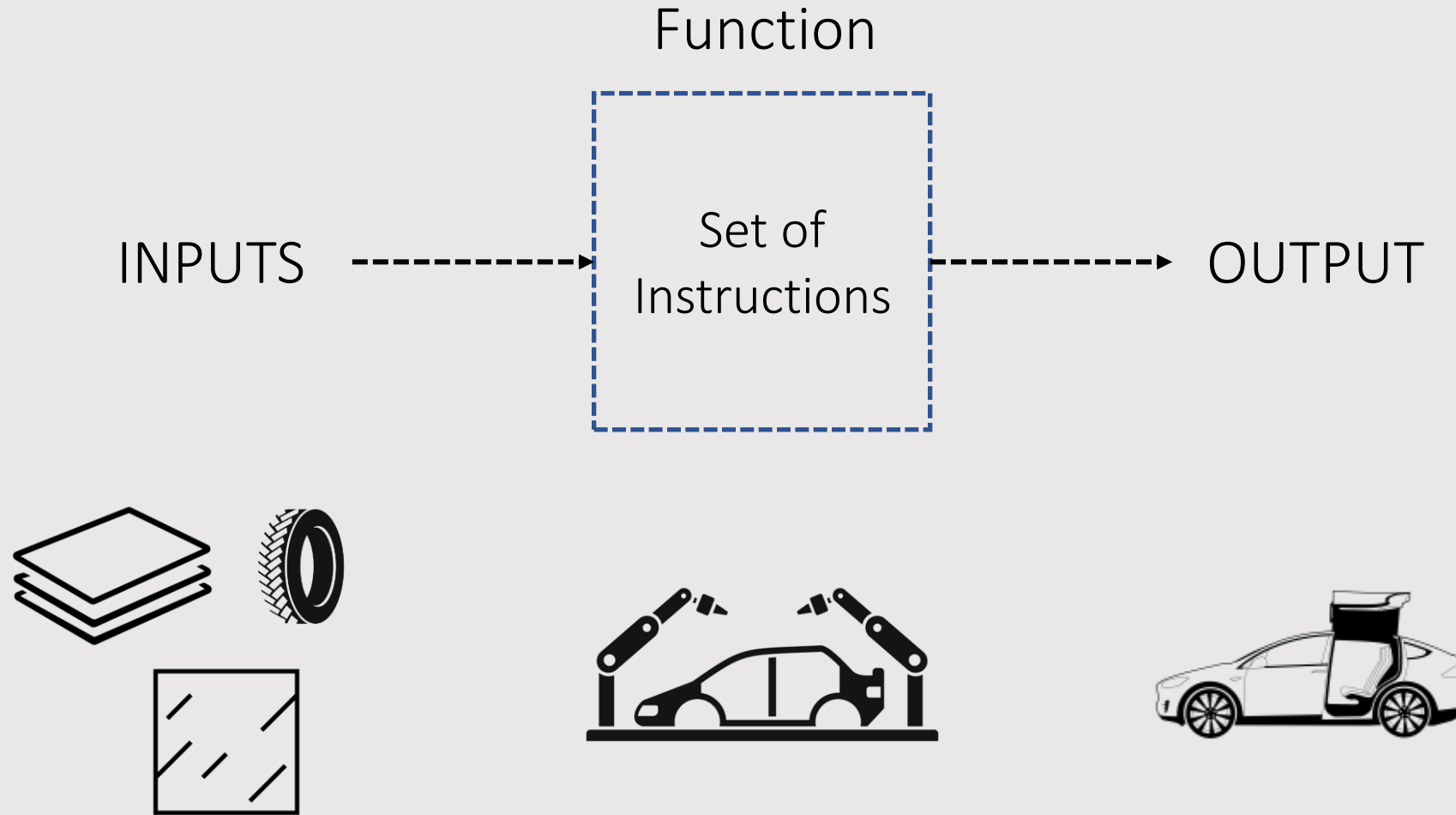
```
In [43]: number_list = [1,2,3,4,5,6,7,8,9,10]
k = 0
while number_list[k] < 5:
    powered = number_list[k] ** 2
    print(powered)
    k += 1
```

```
1
4
9
16
```

```
In [1]: x = 1
while(x > 0):
    print("This loop will never end!!")
```

```
This loop will never end!!
This loop will never end!!
This loop will never end!!
```

# FUNCTIONS



# DEFINING A FUNCTION

Define function name    Input parameters

In [16]: `def find_smaller_number(num1, num2):`

Set of instructions {

```
    if num1 < num2:
        minimum = num1

    elif num1 == num2:
        minimum = 'two numbers are equal'

    else:
        minimum = num2

    return minimum
```

Return output

# WHY USE FUNCTIONS?

## Without a function

```
num1 = 40
num2 = 40

if num1 < num2:
    minimum = num1
```

```
elif num1 == num2:
    minimum = 'two numbers are equal'
```

```
else:
    minimum = num2
```

```
print(minimum)
```

two numbers are equal

```
num1 = 34
num2 = 12

if num1 < num2:
    minimum = num1
```

```
elif num1 == num2:
    minimum = 'two numbers are equal'
```

```
else:
    minimum = num2
```

```
print(minimum)
```

12

```
num1 = 23
num2 = 23

if num1 < num2:
    minimum = num1
```

```
elif num1 == num2:
    minimum = 'two numbers are equal'
```

```
else:
    minimum = num2
```

```
print(minimum)
```

two numbers are equal

```
num1 = 5
num2 = 17
```

```
if num1 < num2:
    minimum = num1
```

```
elif num1 == num2:
    minimum = 'two numbers are equal'
```

```
else:
    minimum = num2
```

```
print(minimum)
```

5

## With a function

```
def find_smaller_number(num1, num2):
```

```
    if num1 < num2:
        minimum = num1
```

```
    elif num1 == num2:
        minimum = 'two numbers are equal'
```

```
    else:
        minimum = num2
```

```
    return minimum
```

```
In [23]: find_smaller_number(34, 12)
```

```
Out[23]: 12
```

```
In [24]: find_smaller_number(23, 23)
```

```
Out[24]: 'two numbers are equal'
```

```
In [25]: find_smaller_number(5, 17)
```

```
Out[25]: 5
```

Functions make complex set of operations reusable

Any variable defined within a function is 'Local'.

# FUNCTIONS: EXAMPLES

Find values of even indices

```
def find_even_indices_vals(vector):  
    values_in_even_indices = vector[::2]  
    return values_in_even_indices
```

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

```
[1, 3, 5, 7, 9]
```

Find values > set threshold

```
def find_outliers(vector, threshold):  
    above_threshold = []  
    for val in vector:  
        if val > threshold:  
            above_threshold.append(val)  
    print(above_threshold)
```

```
vector = [3,6,4,8,3,2,7,8,3.4,5,100,123,5083]  
find_outliers(vector, 100)
```

```
[123, 5083]
```

Find sine values with a given frequency

```
import math  
  
def compute_sin_amp(frequency, t_vec):  
    amplitudes = []  
    for t_val in t_vec:  
        amplitudes.append(math.sin(2 * math.pi * frequency * t_val))  
    return amplitudes
```

```
t_vector = [1/8, 1/6, 1/4]  
compute_sin_amp(1, t_vector)
```

```
[0.7071067811865476, 0.8660254037844386, 1.0]
```

Note: You can call defined functions within a function

# INTRODUCTION TO NUMPY

# WHAT IS NUMPY?

Fundamental package for scientific computing in Python

- Supports multi-dimensional array object
- Provides assortment of mathematical routines for arrays
- Fast array operations through pre-compiled C
- Support array-wide broadcasting for operations
- Included in Anaconda 3



# CONSTRUCTING NUMPY ARRAYS

## From Python lists

```
import numpy as np

# 1D array
arr = np.array([1,2,3,4,5])

# 2D array
arr_2d = np.array([[1,2,3,4,5],
                   [6,7,8,9,10],
                   [11,12,13,14,15]])

print("Array dimensions: ", arr.shape)
print("Array dimensions: ", arr_2d.shape)
print("Array type: ", type(arr))
```

```
Array dimensions: (5,)
Array dimensions: (3, 5)
Array type: <class 'numpy.ndarray'>
```

## From Numpy commands

```
# Define number of each dimension |
n1 = 3
n2 = 4

# Zeros array
zeros_1d = np.zeros(n1)
zeros_2d = np.zeros((n1,n2))

# Ones array
ones_1d = np.ones(n1)
ones_2d = np.ones((n1,n2))

# Creating array using np.arange
arr_arange = np.arange(0, 10, 1) # (start, stop, stepsize)

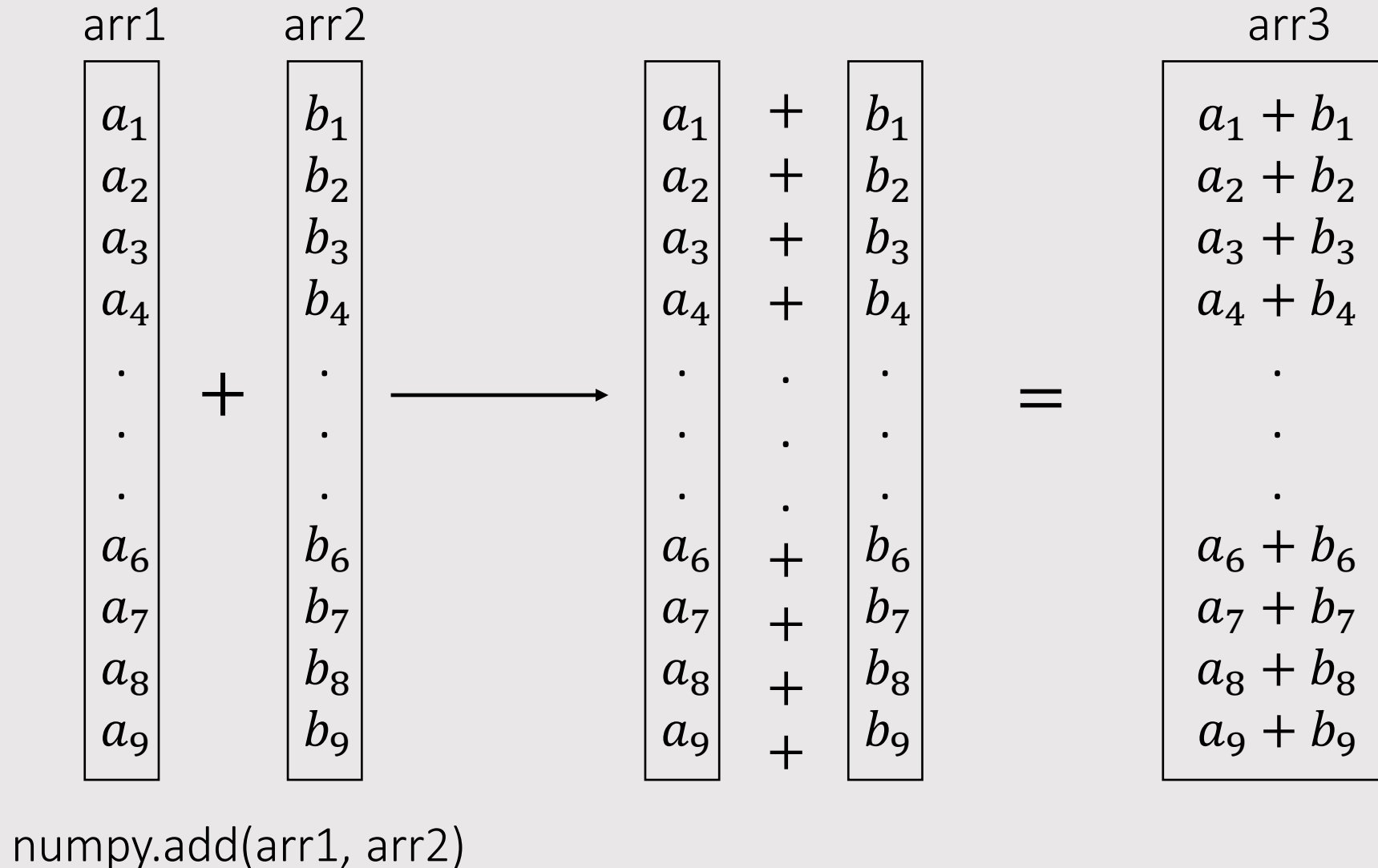
# Creating an array using np.linspace
arr_linspace = np.linspace(0, 9, 10) # (start, stop, # of bins)

print("1D zeros array: ", zeros_1d)
print("1D ones array: ", ones_1d)
print("Number sequence from 0 to 9 using arange: ", arr_arange)
print("Number sequence from 0 to 9 using linspace: ", arr_linspace)
```

```
1D zeros array: [0. 0. 0.]
1D ones array: [1. 1. 1.]
Number sequence from 0 to 9 using arange: [0 1 2 3 4 5 6 7 8 9]
Number sequence from 0 to 9 using linspace: [0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]
```



# ARRAY-WIDE OPERATIONS IN NUMPY



# NUMPY ARITHMETIC OPERATORS

## Operator

## Example

Addition

`np.add()`

```
arr_1 = np.arange(0, 10, 1) # 0 to 9
arr_2 = np.arange(10, 20, 1) # 10 to 19

print("arr_1 + arr_2:", np.add(arr_1, arr_2))
```

```
arr_1 + arr_2: [10 12 14 16 18 20 22 24 26 28]
```

Subtraction

`np.subtract()`

```
print("arr_1 - arr_2:", np.subtract(arr_1, arr_2))
```

```
arr_1 - arr_2: [-10 -10 -10 -10 -10 -10 -10 -10 -10 -10]
```

Multiplication

`np.multiply()`

```
print("arr_1 * arr_2:", np.multiply(arr_1, arr_2))
```

```
arr_1 * arr_2: [ 0 11 24 39 56 75 96 119 144 171]
```

Note: The syntax assumes "import numpy as np"

# NUMPY ARITHMETIC OPERATORS

## Operator

## Example

Exponent

`np.exp()`

```
print("exp(arr_1):", np.exp(arr_1)[:5]) # Print first 5
```

```
exp(arr_1): [ 1.          2.71828183  7.3890561  20.08553692 54.59815003]
```

Division

`np.divide()`

```
print("arr_1 / arr_2:", np.divide(arr_1, arr_2)[:5]) # Print first 5
```

```
arr_1 / arr_2: [0.          0.09090909 0.16666667 0.23076923 0.28571429]
```

Modulo

`np.mod()`

```
print("10 % 3:", np.mod(10, 3))
```

```
10 % 3: 1
```

# USEFUL NUMPY OPERATIONS: COMBINING ARRAYS

Operator

Example

Concatenation

`np.concatenate()`

```
print(np.concatenate([arr_1, arr_2]))
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19]
```

Stack Dimensions

`np.stack()`

```
print(np.stack([arr_1, arr_2]))
```

```
[[ 0  1  2  3  4  5  6  7  8  9]
 [10 11 12 13 14 15 16 17 18 19]]
```

Horizontal Stack

`np.hstack()`

```
print(np.hstack([arr_1, arr_2]))
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19]
```

Vertical Stack

`np.vstack()`

```
print(np.vstack([arr_1, arr_2]))
```

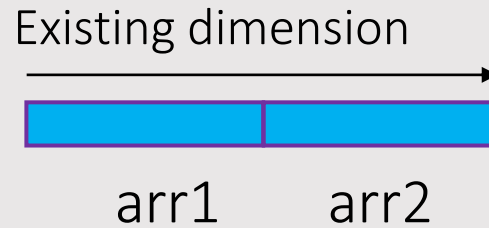
```
[[ 0  1  2  3  4  5  6  7  8  9]
 [10 11 12 13 14 15 16 17 18 19]]
```

# USEFUL NUMPY OPERATIONS: COMBINING ARRAYS

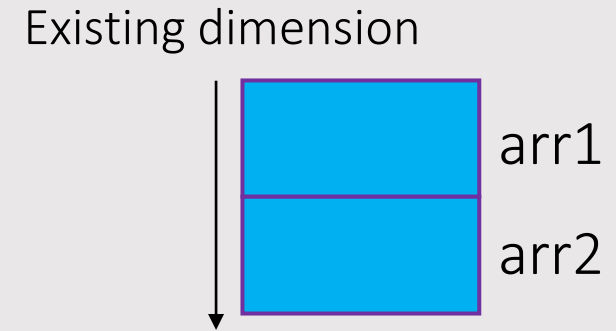
Operator

`np.concatenate()`

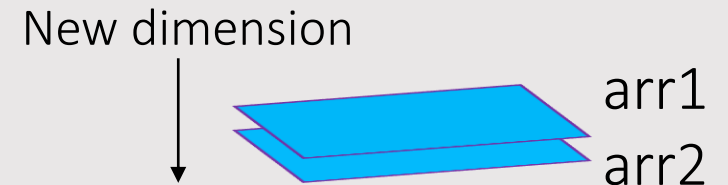
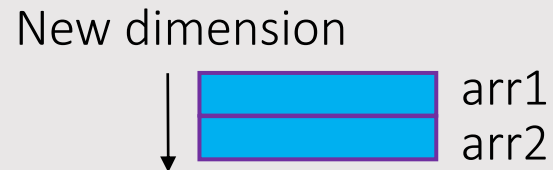
1D



2D



`np.stack()`

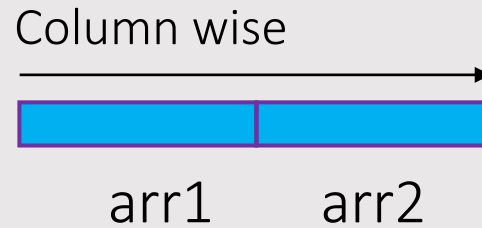


# USEFUL NUMPY OPERATIONS: COMBINING ARRAYS

Operator

`np.hstack()`

1D



2D

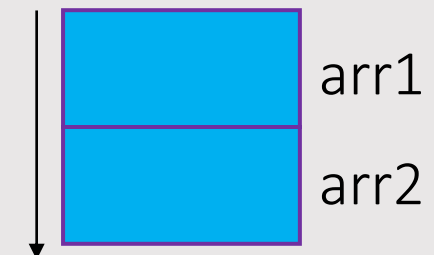


`np.vstack()`

Row wise



Row wise



# USEFUL NUMPY OPERATIONS: CHARACTERISTIC VALUES OF ARRAYS

	Operator	Example
Minimum Value	<code>np.min()</code>	<pre>print(np.min(arr_1))</pre> 0
Maximum Value	<code>np.max()</code>	<pre>print(np.max(arr_1))</pre> 9
Mean Value	<code>np.mean()</code>	<pre>print(np.mean(arr_1))</pre> 4.5
Summed Value	<code>np.sum()</code>	<pre>print(np.sum(arr_1))</pre> 45

Note: axis parameter allows you to compute characteristic value alongside specific axis - e.g. `np.sum(arr_1, axis =0)`: summation along row axis.

# USEFUL ARRAY OPERATIONS: INDEXING ARRAYS

	Operator	Example
Minimum Value Index	<code>np.argmin()</code>	<pre>arr_3 = np.array([4,2,6,7,8,9,3]) print(np.argmin(arr_3))</pre> <p>1</p>
Maximum Value Index	<code>np.argmax()</code>	<pre>print(np.argmax(arr_3))</pre> <p>5</p>
Sort Indices (low to high)	<code>np.argsort()</code>	<pre>print(np.argsort(arr_3))</pre> <p>[1 6 0 2 3 4 5]</p>
Find Indices satisfying a Condition	<code>np.where()</code>	<pre>print(np.where(arr_3 &lt; 7))</pre> <p>(array([0, 1, 2, 6], dtype=int64),)</p>



# MATH OPERATORS WITH NUMPY

Operator

Example

Sine

`np.sin(x)`

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

```
print(np.sin(x_arr))
```

```
[0.84147098 0.90929743 0.14112001]
```

Cosine

`np.cos(x)`

```
print(np.cos(x_arr))
```

```
[ 0.54030231 -0.41614684 -0.9899925 ]
```

Tangent

`np.tan(x)`

```
print(np.tan(x_arr))
```

```
[ 1.55740772 -2.18503986 -0.14254654]
```

Note: Trigonometric functions require radians as inputs

# MATH OPERATORS WITH NUMPY

Operator

Example

Pi

`np.pi`

```
print(np.pi)
```

```
3.141592653589793
```

Square Root

`np.sqrt(x)`

```
print(np.sqrt(x_arr))
```

```
[1.          1.41421356  1.73205081]
```

More functions: <https://numpy.org/doc/stable/reference/routines.math.html>

# PLOTTING WITH MATPLOTLIB

# BASIC PLOTTING WITH MATPLOTLIB

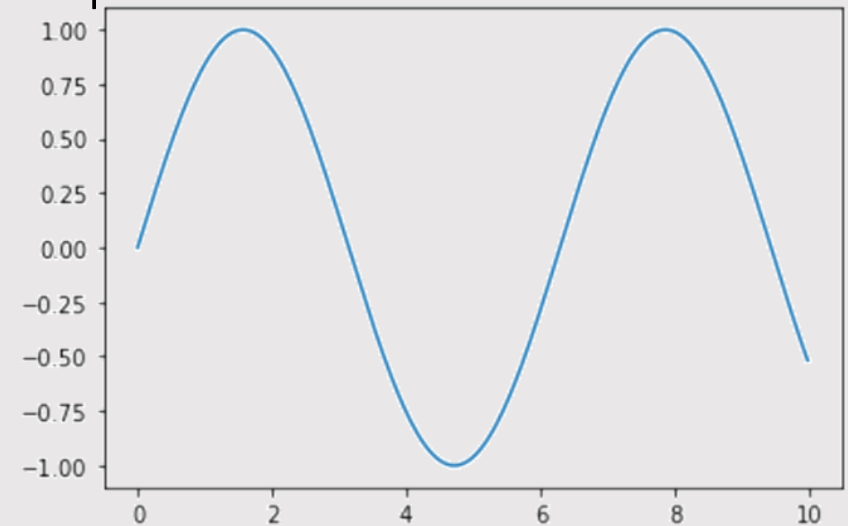
## Import Matplotlib

```
#!/matplotlib inline # If using local notebook runtime, allows you to display the plot inside the jupyter notebook  
#!/matplotlib notebook # Alternatively, you can use this line instead for interactive plots  
  
import matplotlib.pyplot as plt
```

## Code

```
x = np.arange(0, 10, 1/32) # x axis data  
y = np.sin(x)             # y axis data  
plt.plot(x, y)            # plot the data
```

## Output

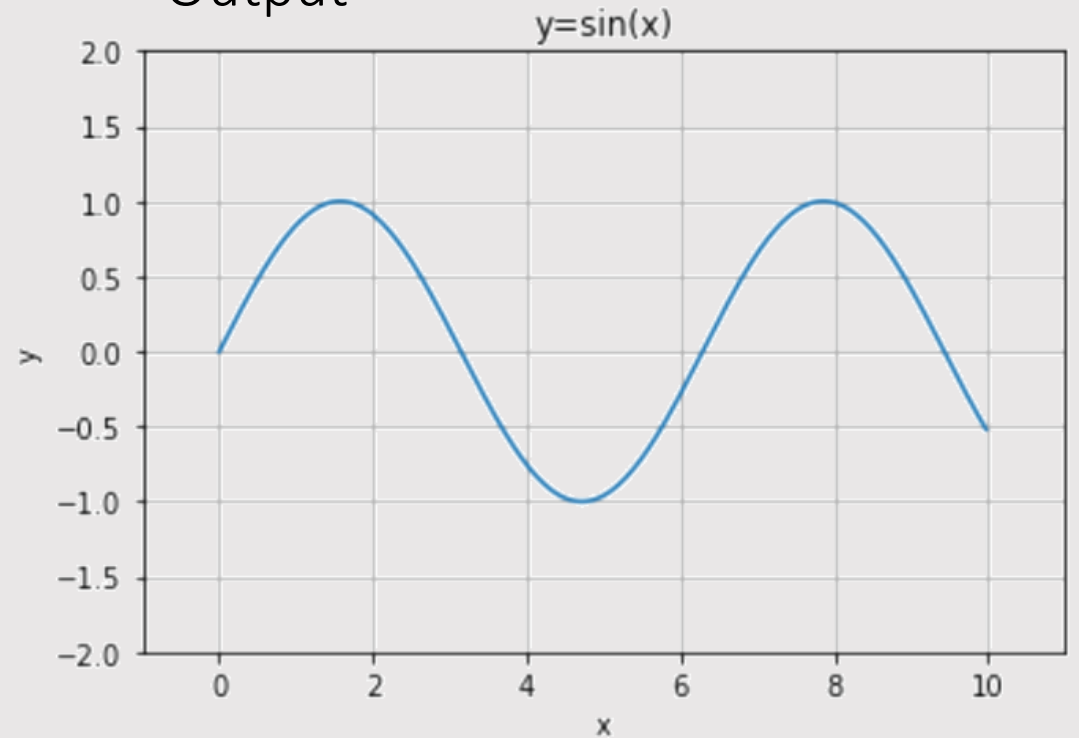


# LABELING YOUR PLOTS

## Code

```
plt.plot(x, y)
plt.title('y=sin(x)') # set the title
plt.xlabel('x')       # set the x axis label
plt.ylabel('y')       # set the y axis label
plt.xlim(-1, 11)     # set the x axis range
plt.ylim(-2, 2)      # set the y axis range
plt.grid()            # enable the grid
```

## Output

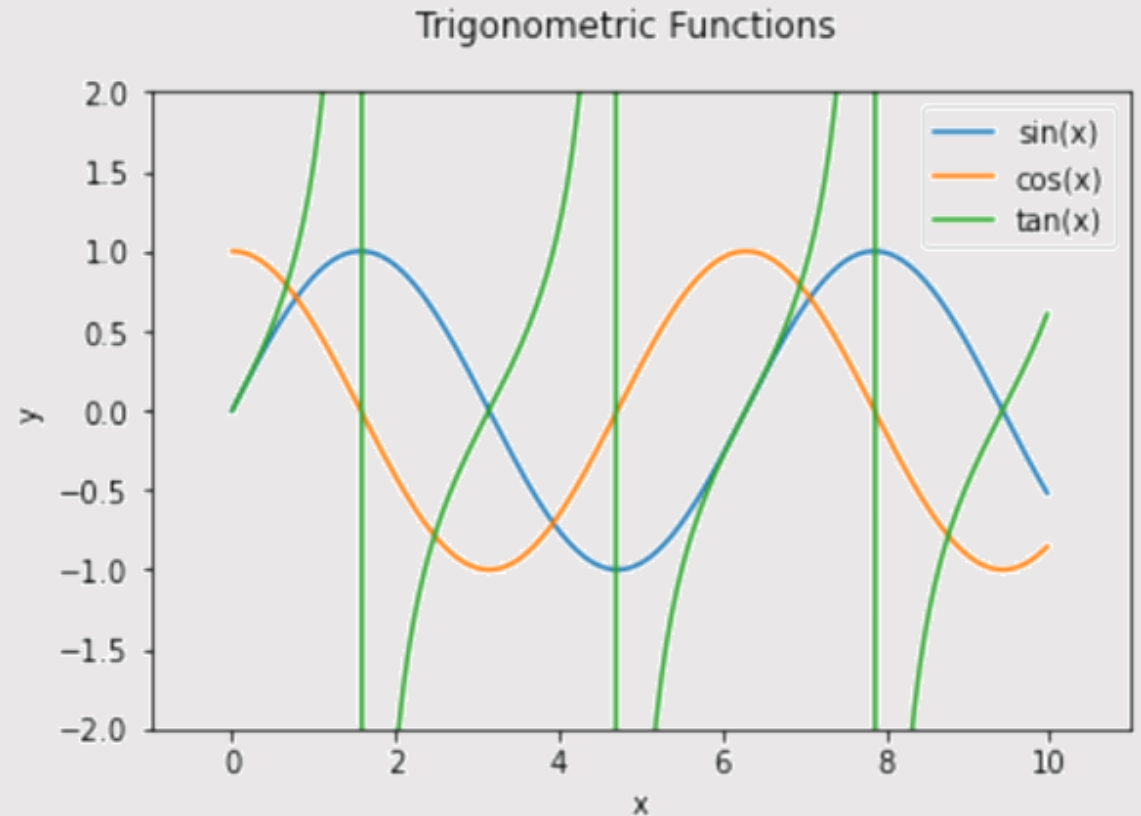


# MULTIPLE PLOTS

## Code

```
# Multiple Plots
# On same figure
x = np.arange(0, 10, 1/32) # x axis data
y1 = np.sin(x)             # y axis data 1
y2 = np.cos(x)             # y axis data 2
y3 = np.tan(x)             # y axis data 3
plt.figure(1)              # create figure 1
plt.plot(x, y1, label='sin(x)')
plt.plot(x, y2, label='cos(x)')
plt.plot(x, y3, label='tan(x)')
plt.xlabel('x')
plt.ylabel('y')
plt.xlim(-1, 11)
plt.ylim(-2, 2)
plt.suptitle('Trigonometric Functions')
plt.legend()
plt.show()
```

## Output



# CREATING SUBPLOTS

## Code

```
# Multiple Subplots
x = np.arange(0, 10, 1/32) # x axis data
y1 = np.sin(x)             # y axis data for subplot 1
y2 = np.cos(x)             # y axis data for subplot 2
y3 = np.tan(x)             # y axis data for subplot 3

fig = plt.figure(2,figsize=(8,8)) # create figure 2

plt.subplot(311)            # (number of rows, number of columns, current plot)
plt.plot(x, y1)
plt.title('sin(x)')
plt.xlabel('x')
plt.ylabel('y')

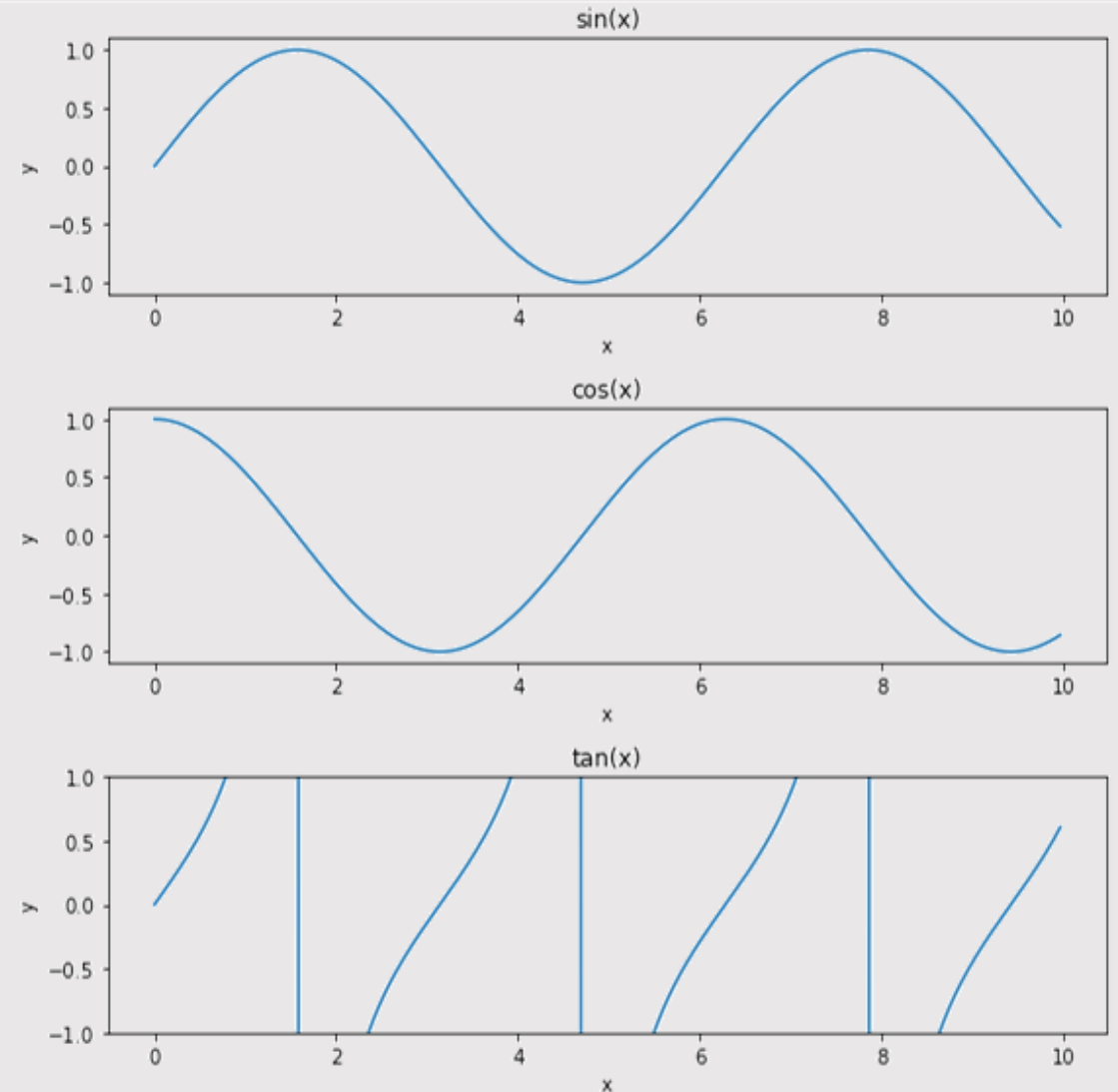
plt.subplot(312)
plt.plot(x, y2)
plt.title('cos(x)')
plt.xlabel('x')
plt.ylabel('y')

plt.subplot(313)
plt.plot(x, y3)
plt.title('tan(x)')
plt.xlabel('x')
plt.ylabel('y')
plt.ylim(-1, 1)

fig.tight_layout()
```

Official documentation:  
<https://matplotlib.org/stable/tutorials/introductory/usage.html#sphx-glr-tutorials-introductory-usage-py>

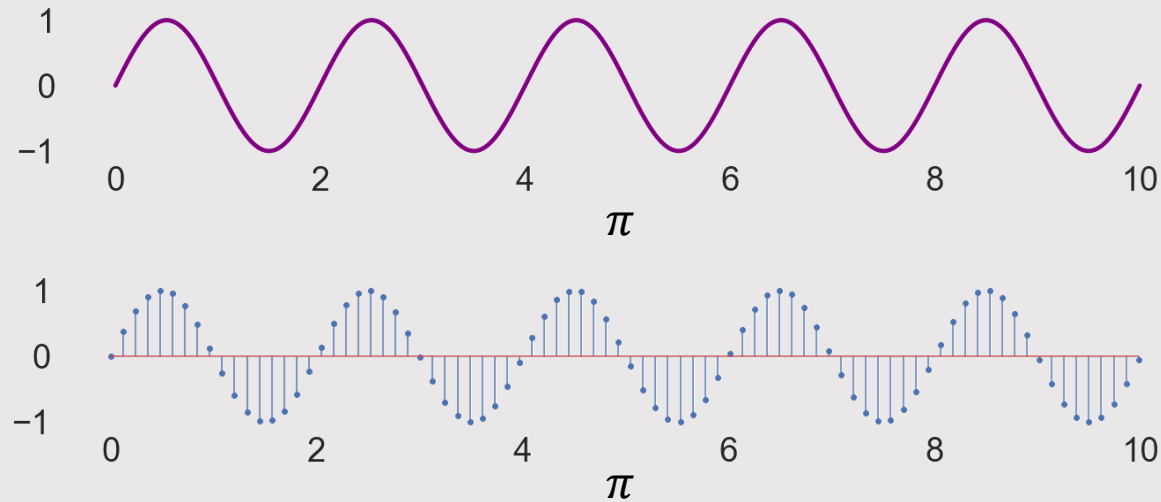
## Output



READ/WRITE/PLAY AUDIO DATA



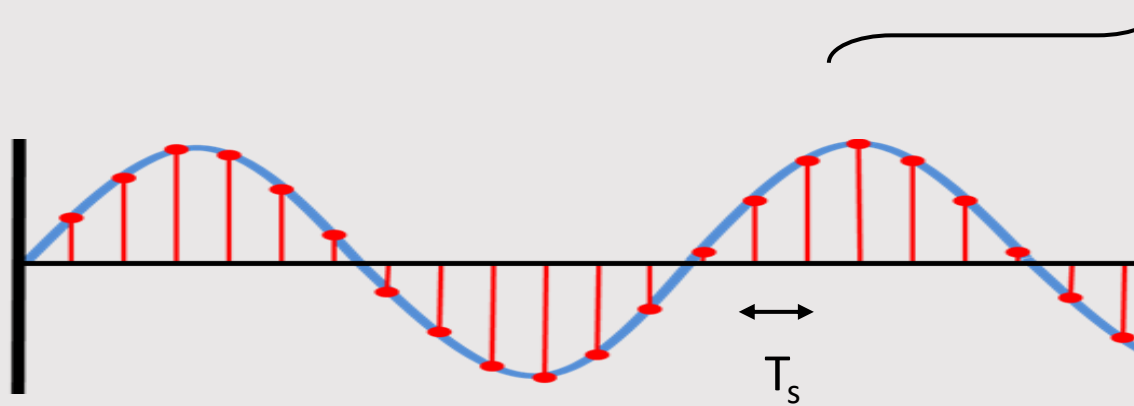
# DIGITAL AUDIO DATA



Analog sound waveform



Digital representation of a sound (array)



Sampling frequency (Hz):  $F_s = 1/T_s$

 Mono (1 channel)

Time

 Stereo (2 channels)

Time

Channels

# READING AND PLAYING AUDIO FILES: simpleaudio

```
(base) C:\Users\Jimin>pip install simpleaudio
Collecting simpleaudio
  Downloading simpleaudio-1.0.4-cp38-cp38-win_amd64.whl (2.0 MB)
    |████████████████████| 2.0 MB 3.3 MB/s
Installing collected packages: simpleaudio
Successfully installed simpleaudio-1.0.4
```

Installing simpleaudio package  
(Anaconda Prompt for Windows)

```
In [82]: import simpleaudio as sa
```

Import simpleaudio

```
In [85]: wav_obj = sa.WaveObject.from_wave_file('train32.wav')
fs = wav_obj.sample_rate
channels = wav_obj.num_channels

print('Sampling rate: ' + str(fs) + 'Hz')
print('Channels: ' + str(channels))
```

Load audio and extract information

```
Sampling rate: 32000Hz
Channels: 1
```

```
In [88]: play_obj = wav_obj.play()
         play_obj.wait_done()
```

Play the audio

Necessary to create delays between audios when playing multiple sounds

# READING AND PLAYING AUDIO FILES: scipy.io

```
from scipy.io import wavfile as wav

fs1, data1 = wav.read('train32.wav')
print('Sampling rate: ' + str(fs1) + 'Hz')
print('Channels: ' + str(len(data1.shape))) # 1D has shape of (n1, ), 2D has shape of (n1, n2)
```

Sampling rate: 32000Hz  
Channels: 1

```
fs2, data2 = wav.read('tuba11.wav')
print('Sampling rate: ' + str(fs2) + 'Hz')
print('Channels: ' + str(len(data2.shape)))
```

Sampling rate: 11025Hz  
Channels: 2

```
play_obj_1 = sa.play_buffer(data1, num_channels = 1, bytes_per_sample = 2, sample_rate = fs1)
play_obj_1.wait_done()
```

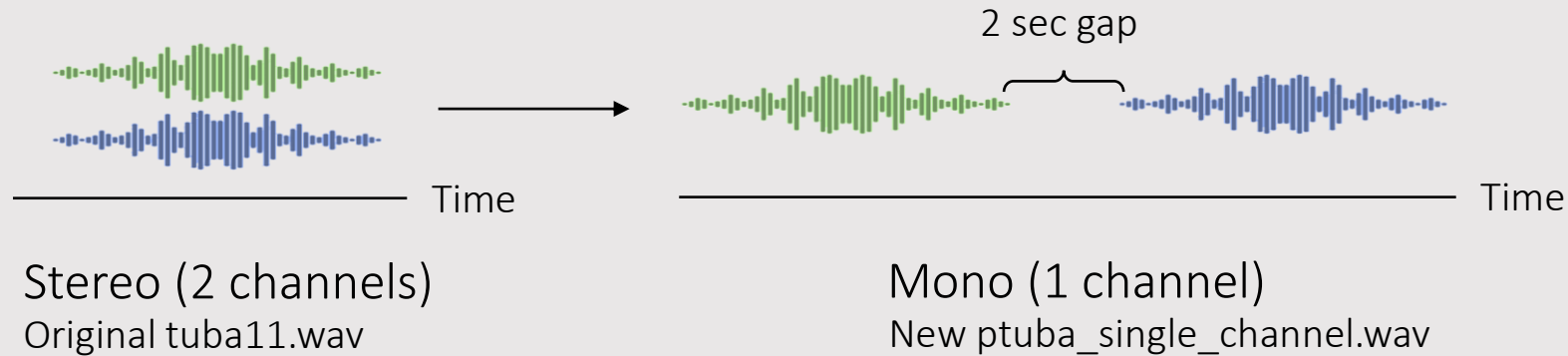
```
play_obj_2 = sa.play_buffer(data2, num_channels = 2, bytes_per_sample = 2, sample_rate = fs2)
play_obj_2.wait_done()
```

Import scipy.io

Load audio and extract information

Play audio using simpleaudio

# WRITING AUDIO WITH scipy.io



## Example Task

```
pause = np.zeros(int(2*fs2))

data0 = data2[:,0]
data1 = data2[:,1]

ptuba_data = np.concatenate([data0, pause, data1])

outfile = 'ptuba_single_channel.wav'

wav.write(outfile, fs2, ptuba_data.astype('int16'))
```

```
wav_obj = sa.WaveObject.from_wave_file('ptuba_single_channel.wav')
play_obj = wav_obj.play()
play_obj.wait_done()
```

Create an array corresponding to 2 sec gap

Extract both channels from audio data

Concatenate channel 1 data + gap + channel 2 data

Set the name of the new audio file

Write new audio file with correct sampling frequency

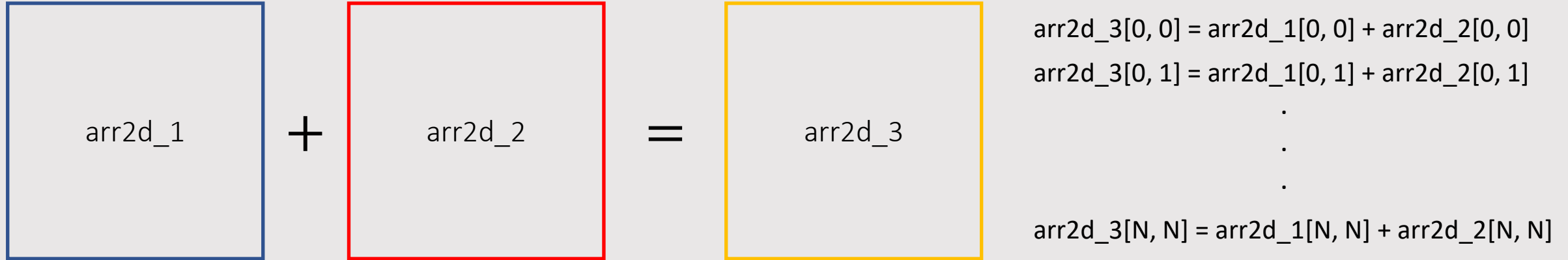
Validate the new audio file by playing it

# LAB ASSIGNMENTS

Download ipynb template in Canvas page:

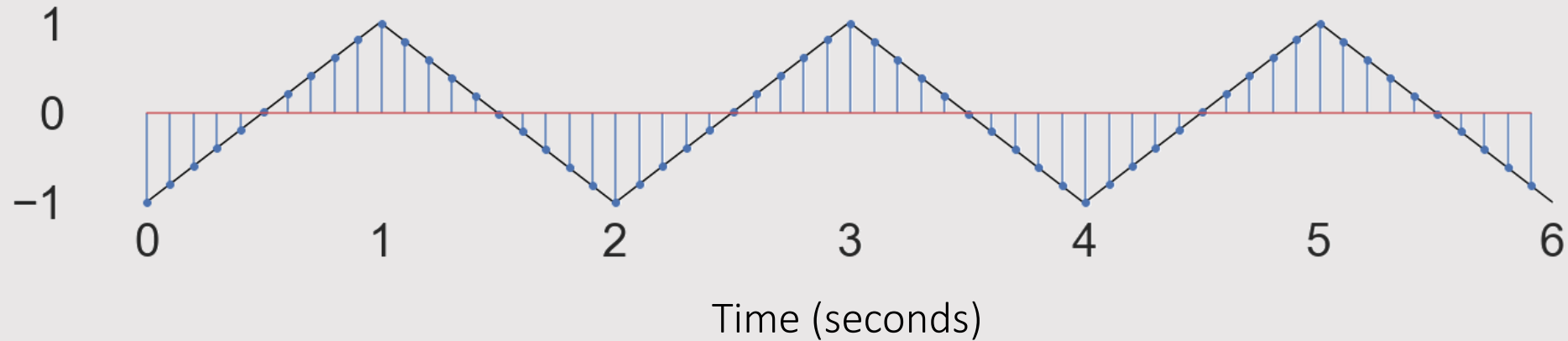
Assignments/Lab 2 report -> click “Lab 2 Report Templates”

# EXERCISE 1: Loops vs Numpy operations



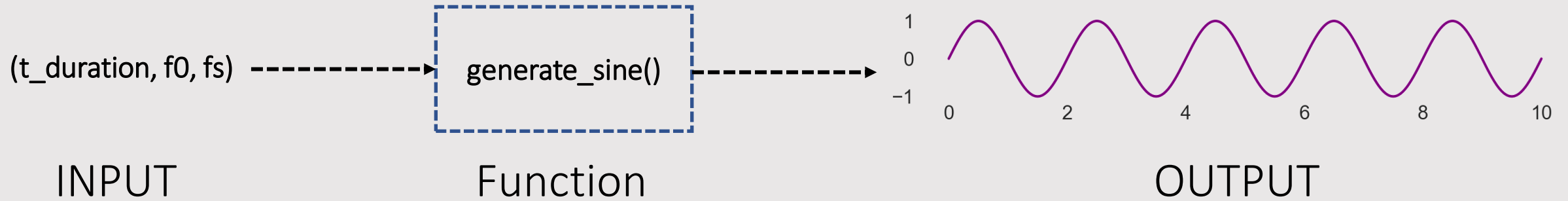
- In lab2\_report\_template.ipynb, we provided numpy array variables 'array2d\_1' and 'array2d\_2'. Both have dimensions (1000, 1000).
- We want to perform elementwise addition between two and create a new array called 'array2d\_3'.
- Your task is to implement this operation in two ways:
  - Using a loop without numpy (using two nested loop)
  - Using an appropriate numpy function
- Run pre-written code to ensure two outputs are equal.
- Measure and compare the computation time for each operation using the code in the template. Report which operation is faster and by how much.

# EXERCISE 2: Generate Triangular Waveform



- One of the popular waveform in signal analysis is a triangular wave.
- Your task is to implement a triangular wave shown above by using appropriate Numpy functions and Python commands.
- The waveform should have **amplitude** of 1 and **frequency** of 0.5Hz. Use the **sampling frequency** of 10Hz
- Validate your code by plotting the waveform in the time range of **0 – 6 seconds** as shown above. You can use `plt.stem()` function in matplotlib to generate the stem plot style.

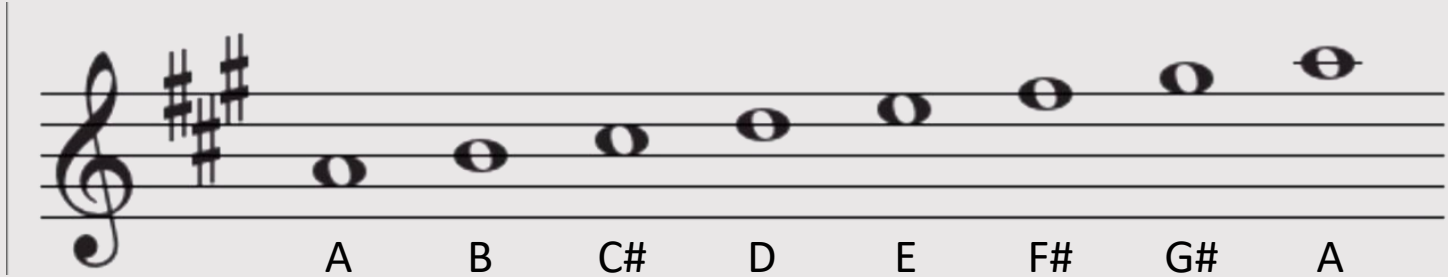
# EXERCISE 3: Sinusoidal Generator



- Construct a function `generate_sine()` which given time duration, wave frequency and sampling frequency, outputs two 1D numpy arrays each corresponding to time points and sine wave form.
- The function should accept following parameters
  - `t_duration` – Time duration in seconds which the sine wave is defined
  - `f0` – Wave frequency
  - `fs` – Sampling frequency
- Note that you need to convert Hz to angular frequency using  $\omega = 2\pi f_0$  for `np.sin()`.
- Test your function against with three given sets of `(t_duration, f0, fs)` in `Lab2_Report_Template.ipynb`.
- Plot three sine waveforms using 3 x 1 subplots. Include proper time axis, title and labels for each subplot.



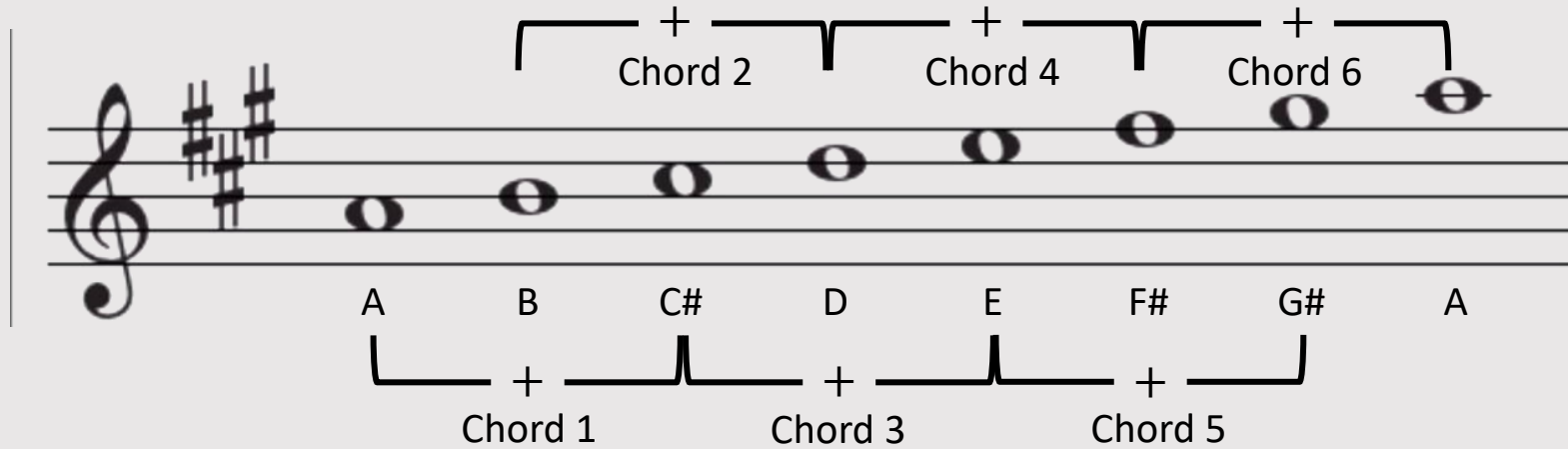
# EXERCISE 4: Notes Synthesis



- For this exercise we will synthesize 8 notes for A-Major scale shown above
- Using the `generate_sine()` function, synthesize 8 notes that make up A-Major scale. Each note should last 1 second.
- For each note, use sampling rate  $f_s = 8000$  and **amplitude = 1**
- Once all 8 notes are constructed, concatenate them into a single 1D array.
- Play the concatenated notes using `simpleaudio` and write it into an audio file with name “a\_major\_scale.wav”.

Note	Frequency (Hz)
A	220
B	$220 * 2^{\frac{2}{12}}$
C#	$220 * 2^{\frac{4}{12}}$
D	$220 * 2^{\frac{5}{12}}$
E	$220 * 2^{\frac{7}{12}}$
F#	$220 * 2^{\frac{9}{12}}$
G#	$220 * 2^{\frac{11}{12}}$
A	440

# EXERCISE 5: Chord Synthesis



Chord 1 = A + C#

Chord 2 = B + D

Chord 3 = C# + E

Chord 4 = D + F#

Chord 5 = E + G#

Chord 6 = F# + A

Note Frequency (Hz)

A	220
B	$220 * 2^{\frac{2}{12}}$
C#	$220 * 2^{\frac{4}{12}}$
D	$220 * 2^{\frac{5}{12}}$
E	$220 * 2^{\frac{7}{12}}$
F#	$220 * 2^{\frac{9}{12}}$
G#	$220 * 2^{\frac{11}{12}}$
A	440

- For this exercise, we will expand upon exercise 4 to synthesize music chords – i.e. set of pitches consisting of multiple notes
- For example, a chord could be A + B or B + C# + D, etc
- Generate 6 chords as shown above figure – each chord consists of **addition of two notes** and should last for 1s with sampling frequency of 8000Hz.
- Make sure you normalize the amplitude of the chord between -1 and 1 to be compatible with simpleaudio standard.
- Concatenate 6 chords into a single 1D array and write into audio file with a name “6\_chords.wav”

SUPPLEMENTARY:

FUNCTION CALL ERROR &  
LOOKING UP FUNCTION DOCUMENTATION

# FUNCTION CALL ERROR VIA INCORRECT ARGUMENT

```
1 import numpy as np
```

```
1 np.add([10, 20])
```

```
-----  
ValueError                                Traceback (most recent call last)  
<ipython-input-3-4b4371c24abd> in <module>  
----> 1 np.add([50, 10])
```

**ValueError:** invalid number of arguments

```
1 np.add(10, 20, 30)
```

```
-----  
TypeError                                Traceback (most recent call last)  
<ipython-input-9-02f3bf84d5c2> in <module>  
----> 1 np.add(10, 20, 30)
```


**TypeError:** return arrays must be of ArrayType

```
1 np.add(50, 10)
```

} Incorrect function arguments

Correct function arguments

# LOOKING UP FUNCTION DOCUMENTATION

 NumPy

User Guide **API reference** Development

Search the docs ...

Array objects

Constants

Universal functions ( **ufunc** )

**Routines**

Array creation routines

Array manipulation routines

Binary operations

String operations

C-Types Foreign Function Interface ( **numpy.ctypeslib** )

Datetime Support Functions

Data type routines

Optionally SciPy-accelerated routines ( **numpy.dual** )

Mathematical functions with automatic domain ( **numpy.emath** )

Floating point error handling

Discrete Fourier Transform ( **numpy.fft** )

Functional programming

NumPy-specific help functions

## numpy.add

```
numpy.add(x1, x2, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature, extobj]) = <ufunc 'add'>
```

Add arguments element-wise.

**Parameters:** **x1, x2** : *array\_like*

The arrays to be added. If **x1.shape** **!=** **x2.shape**, they must be broadcastable to a common shape (which becomes the shape of the output).

**out** : *ndarray, None, or tuple of ndarray and None, optional*

A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where** : *array\_like, optional*

This condition is broadcast over the input. At locations where the condition is True, the **out** array will be set to the ufunc result. Elsewhere, the **out** array will retain its original value. Note that if an uninitialized **out** array is created via the default **out=None**, locations within it where the condition is False will remain uninitialized.

**\*\*kwargs**

For other keyword-only arguments, see the [ufunc docs](#).

**Returns:** **add** : *ndarray or scalar*

The sum of **x1** and **x2**, element-wise. This is a scalar if both **x1** and **x2** are scalars.

User defined arguments

Optional arguments  
(already have default values)

Function Output

# FUNCTION CALL ERROR VIA INCORRECT ARGUMENT

```
1 import numpy as np
```

```
1 np.add([10, 20])
```

Missing x2 argument (x1 = [10, 20])

```
-----  
ValueError                                Traceback (most recent call last)  
<ipython-input-3-4b4371c24abd> in <module>  
----> 1 np.add([50, 10])
```

**ValueError:** invalid number of arguments

```
1 np.add(10, 20, 30)
```

30 recognized as “out” parameter  
which expects ArrayType

```
-----  
TypeError                                Traceback (most recent call last)  
<ipython-input-9-02f3bf84d5c2> in <module>  
----> 1 np.add(10, 20, 30)
```

**TypeError:** return arrays must be of ArrayType

```
1 np.add(50, 10)
```

Incorrect function arguments

Correct function arguments