> created by Luis S. Flores for ITESM Contact: Luis_SFU@tec.mx
> August/2025

While Loops, Lists, and Charts in Python

This notebook teaches:

- While loops (repetition)
- Lists (store multiple values)
- Charts with Matplotlib
- (New) Quick NumPy intro

Keep it simple. Run cells top to bottom.

While Loops — the basics

A **while** loop repeats as long as a condition is `True`.

Structure:

```
#while <condition>:
#     <indented code>
# code continues here after loop ends
```

In [213…

```
x = 1
while x <= 5:
    print("x is:", x)
    x = x + 1   # update
print("Loop finished. x is now", x)
```

```
x is: 1
x is: 2
x is: 3
x is: 4
x is: 5
Loop finished. x is now 6
```

Indentation in while loops

- The **colon (:)** starts the loop.
- The **indented part** is the body (what repeats).
- After the loop ends, code **without indentation** runs once.

In [214…

```
n = 3
while n > 0:
```

```
    print("Inside loop. n =", n)  # repeated
    n = n - 1
print("This runs after the loop ends")  # not repeated
```

```
Inside loop. n = 3
Inside loop. n = 2
Inside loop. n = 1
This runs after the loop ends
```

Infinite loop with `True` and `break`

We can write `while True:` to loop forever, and use `break` to stop.

In [215…

```
count = 0
while True:
    print("count is", count)
    count = count + 1
    if count == 3:
        break   # leave the loop
print("Loop stopped with break")
```

```
count is 0
count is 1
count is 2
Loop stopped with break
```

Using `continue`

`continue` skips the rest of the body and goes to the next loop check.

In [216…

```
x = 0
while x < 6:
    x = x + 1
    if x % 2 == 0:
        continue     # skip even numbers
    print("Odd number:", x)
```

```
Odd number: 1
Odd number: 3
Odd number: 5
```

Counter

A **counter** is a variable that changes by a fixed step (often +1).

In [217…

```
count = 0
while count < 5:
    count = count + 1
    print("Counter now:", count)
```

```
Counter now: 1
Counter now: 2
Counter now: 3
Counter now: 4
Counter now: 5
```

Accumulator

An **accumulator** is a variable that adds different values each time. (No lists yet.)

In [218…

```
total = 0
total = total + 2
total = total + 5
total = total + 1
print("Final accumulator:", total)
```

```
Final accumulator: 8
```

Accumulator with `while` (still no lists)

We can also add a changing value each step.

In [219…

```
total = 0
k = 1
while k <= 4:
    total = total + (k * 2)   # adds 2, 4, 6, 8
    k = k + 1
print("Accumulator:", total)
```

```
Accumulator: 20
```

Lists — basics

A list stores multiple values in one variable.

- Written with square brackets `[ ]`
- First item is index **0**
- `len(list)` gives the number of items
- There is **no** `size()` for Python lists — use `len(...)`

In [220…

```
fruits = ["apple", "banana", "cherry"]

print("Index 0:", fruits[0])
print("Index 1:", fruits[1])
print("How many items:", len(fruits))

last_index = len(fruits) - 1
print("Last item:", fruits[last_index])
```

```
Index 0: apple
Index 1: banana
How many items: 3
Last item: cherry
```

Empty list, append, and change

- Start empty with `[]`
- `append(x)` puts `x` at the **end**
- Change by index with assignment

In [221…

```
grades = []            # empty list
grades.append(90)        # [90]
grades.append(85)        # [90, 85]
print("After appends:", grades)

grades = [90, 85, 78]
print("Before append:", grades)
grades.append(100)
print("After append:", grades)

grades[1] = 88           # change second item
print("After change:", grades)
```

```
After appends: [90, 85]
Before append: [90, 85, 78]
After append: [90, 85, 78, 100]
After change: [90, 88, 78, 100]
```

More list operations

- `insert(pos, value)` → add at position
- `extend([values])` → add many at once
- `remove(value)` → remove first match
- `pop()` → remove last item
- `len(list)` → how many items

In [222…

```
nums = [10, 20, 30]
nums.insert(1, 15)
nums.extend([40, 50])
print("After insert and extend:", nums)

nums.remove(20)
print("After remove:", nums)

last = nums.pop()
print("After pop:", nums, "popped:", last)

print("Length is", len(nums))
```

```
After insert and extend: [10, 15, 20, 30, 40, 50]
After remove: [10, 15, 30, 40, 50]
After pop: [10, 15, 30, 40] popped: 50
Length is 4
```

List Slices (read parts of a list)

Form: `list[start:end:step]`

- **start** → index to begin (default = 0)
- **end** → index to stop *before* (default = len(list))
- **step** → how much to jump each time (default = 1)

In [223…

```python
data = [10, 20, 30, 40, 50, 60]
# indexes:   0   1   2   3   4   5

print("data[:]     ->", data[:])      # copy whole list
print("data[1:4]   ->", data[1:4])    # 1,2,3
print("data[:3]    ->", data[:3])     # start..2
print("data[3:]    ->", data[3:])     # 3..end

print("data[::2]   ->", data[::2])    # step 2: [10, 30, 50]
print("data[1::2]  ->", data[1::2])   # start at 1, step 2: [20, 40, 60]
print("data[::3]   ->", data[::3])    # step 3: [10, 40]
```

```
data[:]     -> [10, 20, 30, 40, 50, 60]
data[1:4]   -> [20, 30, 40]
data[:3]    -> [10, 20, 30]
data[3:]    -> [40, 50, 60]
data[::2]   -> [10, 30, 50]
data[1::2]  -> [20, 40, 60]
data[::3]   -> [10, 40]
```

Negative indexes and negative step

- Negative indexes count from the end (`-1` = last).
- Negative step goes backwards.

In [224…

```python
data = [10, 20, 30, 40, 50, 60]

print("data[-1]    ->", data[-1])      # last item
print("data[-3:]   ->", data[-3:])     # last 3
print("data[::-1]  ->", data[::-1])    # reverse
print("data[5:2:-1]->", data[5:2:-1]) # 5 down to 3
```

```
data[-1]    -> 60
data[-3:]   -> [40, 50, 60]
data[::-1]  -> [60, 50, 40, 30, 20, 10]
data[5:2:-1]-> [60, 50, 40]
```

Slice Assignment (replace parts)

Because lists are mutable, slices can be replaced.

In [225…

```
nums = [1, 2, 3, 4, 5, 6]
nums[2:5] = [99, 100]     # replace indices 2,3,4
print(nums)

nums[1:1] = [7, 8]        # insert at index 1
print(nums)

nums[0:3] = []            # delete a slice
print(nums)
```

```
[1, 2, 99, 100, 6]
[1, 7, 8, 2, 99, 100, 6]
[2, 99, 100, 6]
```

Operations on Lists

- Concatenate: `a + b`
- Repeat: `a * n`
- Membership: `x in a`, `x not in a`
- Min/Max/Sum: `min(a)`, `max(a)`, `sum(a)`
- Sorted copy: `sorted(a)` (new list)
- In-place sort: `a.sort()` (changes original)

In [226…

```
a = [3, 1, 2]
b = [9, 8]

print("a + b     ->", a + b)
print("a * 3     ->", a * 3)
print("2 in a    ->", 2 in a)
print("min/max   ->", min(a), max(a))
print("sum(a)    ->", sum(a))
print("sorted(a) ->", sorted(a), " ; original a:", a)

a.sort()
print("after a.sort():", a)
```

```
a + b     -> [3, 1, 2, 9, 8]
a * 3     -> [3, 1, 2, 3, 1, 2, 3, 1, 2]
2 in a    -> True
min/max   -> 1 3
sum(a)    -> 6
sorted(a) -> [1, 2, 3]  ; original a: [3, 1, 2]
after a.sort(): [1, 2, 3]
```

Proper Ways to Delete Items

- By position (and return value): `pop(index)` (default = last)
- By position (no return): `del lst[index]`
- By value: `remove(value)` (first match)

- By slice: `del lst[a:b]`
- Clear all: `clear()`

In [227…
```python
vals = [10, 20, 30, 40, 50]

last = vals.pop()          # removes 50, returns it
print("pop() ->", last, ";", vals)

item = vals.pop(1)         # removes index 1 (20)
print("pop(1) ->", item, ";", vals)

del vals[0]                # remove index 0 (10)
print("del[0] ->", vals)

vals.remove(40)            # remove value 40
print("remove(40) ->", vals)

vals.extend([60,70,80,90])
del vals[1:3]              # delete a slice
print("del slice ->", vals)

vals.clear()               # remove everything
print("clear() ->", vals)
```

```
pop() -> 50 ; [10, 20, 30, 40]
pop(1) -> 20 ; [10, 30, 40]
del[0] -> [30, 40]
remove(40) -> [30]
del slice -> [30, 80, 90]
clear() -> []
```

Safe Remove

`remove(value)` raises an error if value is not found. Check first.

In [228…
```python
items = [1, 2, 3]
target = 4

if target in items:
    items.remove(target)
else:
    print(target, "not in list, nothing removed.")
print(items)
```

```
4 not in list, nothing removed.
[1, 2, 3]
```

For loops

A **for** loop repeats over a sequence (like a list). Often simpler than `while` when you already have the items.

In [229…
```python
fruits = ["apple", "banana", "cherry"]

for f in fruits:
    print("Fruit:", f)
```
```
Fruit: apple
Fruit: banana
Fruit: cherry
```

For with `range()`

`range(n)` gives numbers from 0 up to n-1.

In [230…
```python
for i in range(5):
    print("i is:", i)
```
```
i is: 0
i is: 1
i is: 2
i is: 3
i is: 4
```

For with list indexes (using `len`)

In [231…
```python
grades = [90, 85, 78]

for i in range(len(grades)):   # indexes: 0,1,2
    print("Index", i, "has grade", grades[i])
```
```
Index 0 has grade 90
Index 1 has grade 85
Index 2 has grade 78
```

For with `enumerate()`

Gives both index and value at the same time.

In [232…
```python
grades = [90, 85, 78]

for index, value in enumerate(grades):
    print("Index", index, "has grade", value)
```
```
Index 0 has grade 90
Index 1 has grade 85
Index 2 has grade 78
```

While vs For — looping through a list

Both can loop a list; style differs.

In [233…

```
grades = [90, 85, 78]

# Using while
i = 0
while i < len(grades):
    print("While -> Index", i, "Grade", grades[i])
    i = i + 1

# Using for with range
for i in range(len(grades)):
    print("For -> Index", i, "Grade", grades[i])

# Using for directly
for grade in grades:
    print("For (direct) -> Grade", grade)
```

```
While -> Index 0 Grade 90
While -> Index 1 Grade 85
While -> Index 2 Grade 78
For -> Index 0 Grade 90
For -> Index 1 Grade 85
For -> Index 2 Grade 78
For (direct) -> Grade 90
For (direct) -> Grade 85
For (direct) -> Grade 78
```

Break and Continue note

Both `while` and `for` support:

- **`break`** → exit the loop early.
- **`continue`** → skip the rest of the body and jump to the next iteration.

Break and Continue inside a For loop

In [234…

```
grades = [90, 85, 78, 92, 88]

for g in grades:
    if g < 80:
        continue      # skip low grades
    if g > 90:
        break         # stop completely
    print("Grade:", g)
```

```
Grade: 90
Grade: 85
```

Important Note on Break

⚠ Do NOT make `break` your normal thought process.

- `break` is sometimes necessary (like stopping when something goes very wrong), but it should be an **exception**, not the default way to design loops.
- `continue` has more natural uses (like skipping unwanted items), but it should also be used carefully.
- The normal way: design the loop condition so it ends naturally, without `break`.

Charts with Matplotlib — examples

Matplotlib draws simple plots. In Colab it is already installed.

Matplotlib style short-codes (colors / markers / line styles)

Code	Color	Code	Marker	Code	Line style
`b`	blue	`.`	point	`-`	solid
`g`	green	`o`	circle	`:`	dotted
`r`	red	`x`	x-mark	`-.`	dashdot
`c`	cyan	`+`	plus	`--`	dashed
`m`	magenta	`*`	star		*(none)*
`y`	yellow	`s`	square		
`k`	black	`d`	diamond		
`w`	white	`v`	triangle down		
		`^`	triangle up		
		`<`	triangle left		
		`>`	triangle right		
		`p`	pentagon		
		`h`	hexagram		

> You can combine them in format strings like `'o-'` (circle markers with solid line).

1) Line chart (simple)

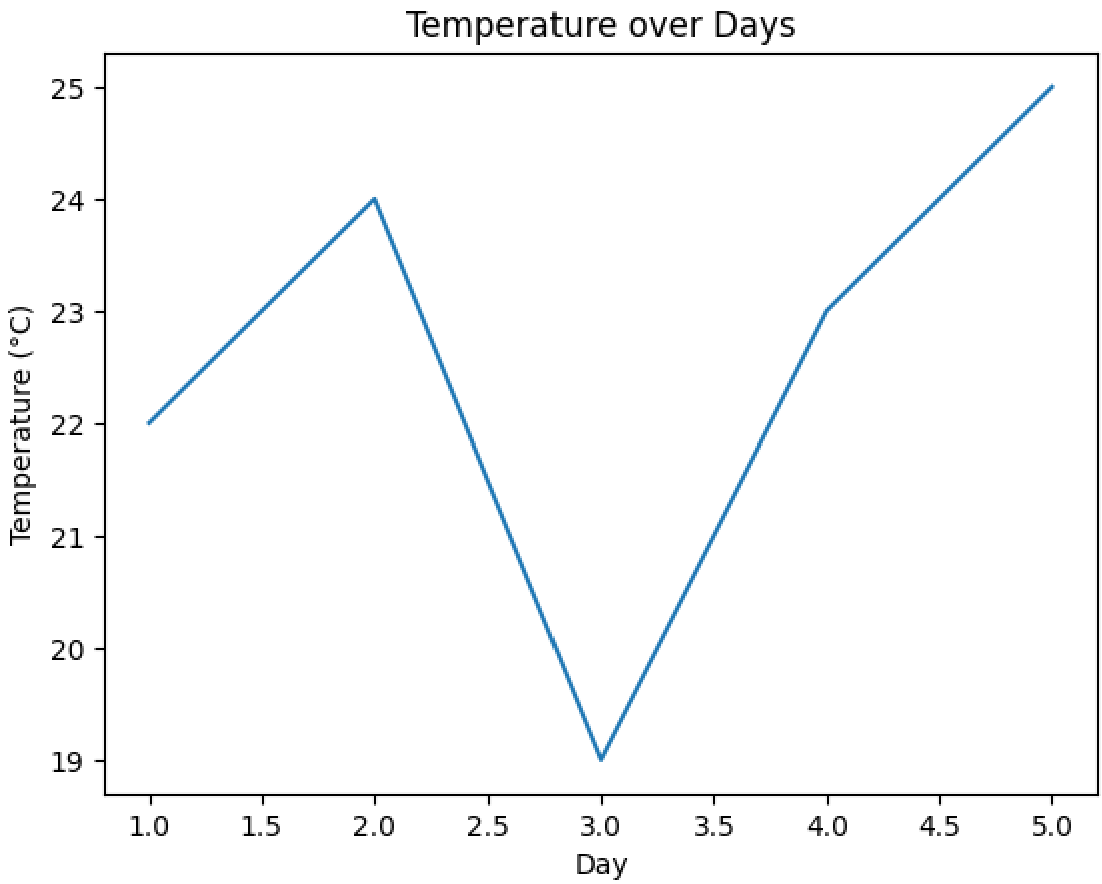
In [235…

```
import matplotlib.pyplot as plt

days = [1, 2, 3, 4, 5]
temperature = [22, 24, 19, 23, 25]

plt.plot(days, temperature)
plt.title("Temperature over Days")
```

```
plt.xlabel("Day")
plt.ylabel("Temperature (°C)")
plt.show()
```

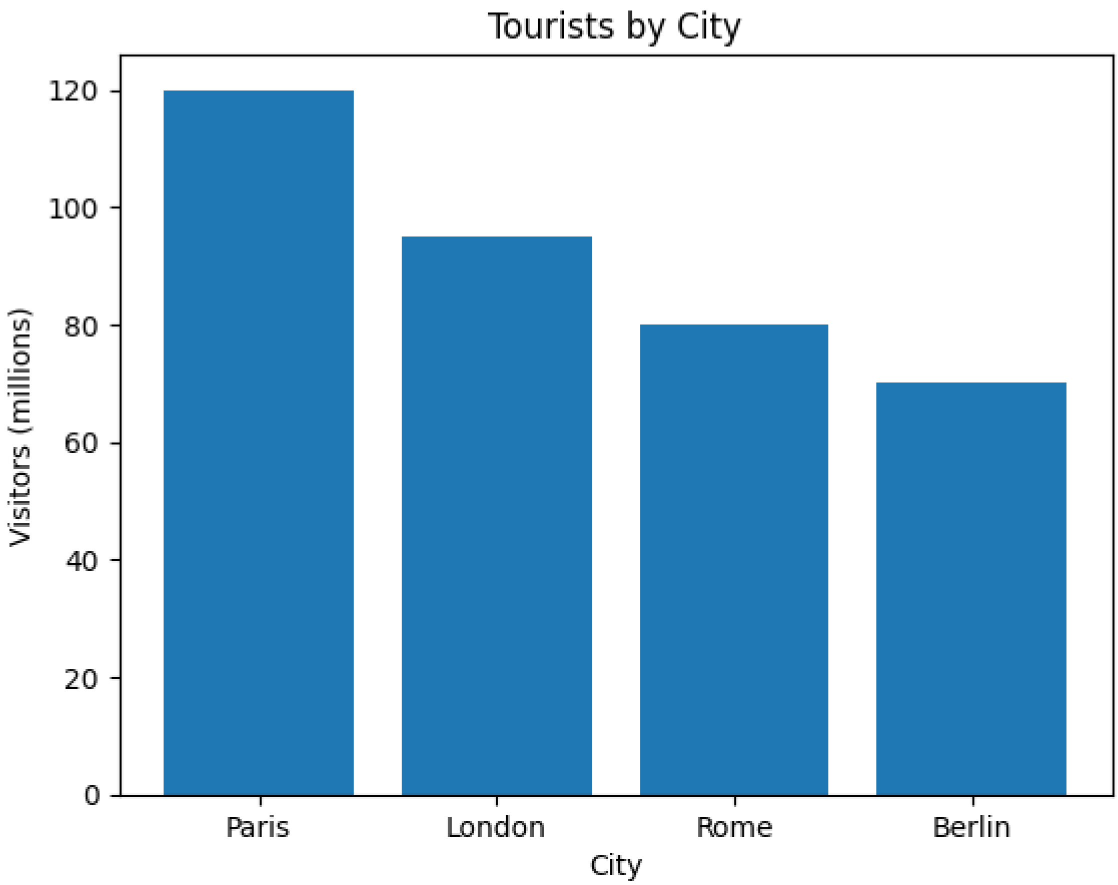


2) Bar chart (one series)

In [236…

```
cities = ["Paris", "London", "Rome", "Berlin"]
visitors = [120, 95, 80, 70]

plt.bar(cities, visitors)
plt.title("Tourists by City")
plt.xlabel("City")
plt.ylabel("Visitors (millions)")
plt.show()
```



3) Bar chart (two series side by side)

Call `plt.bar` once per dataset and offset the bars.

In [237…

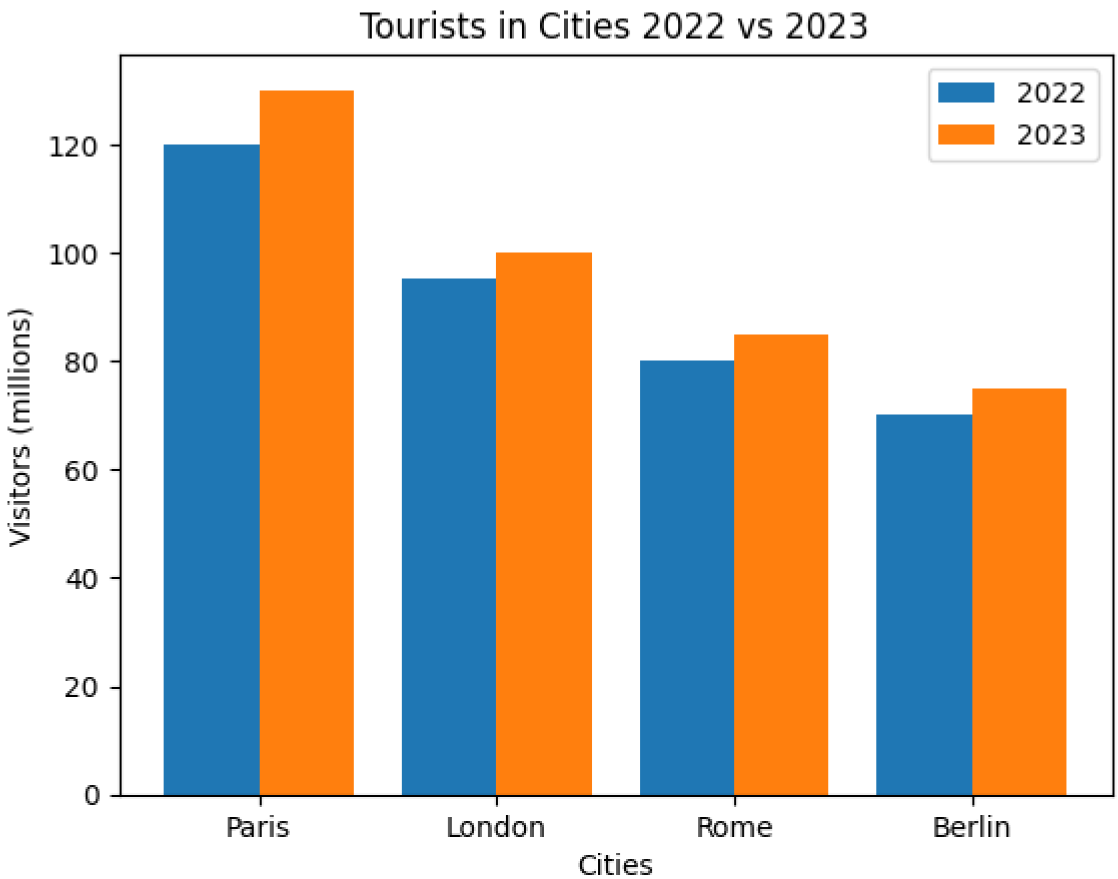
```
cities = ["Paris", "London", "Rome", "Berlin"]
visitors_2022 = [120, 95, 80, 70]
visitors_2023 = [130, 100, 85, 75]

x = list(range(len(cities)))
width = 0.4

x_left  = [xi - width/2 for xi in x]
x_right = [xi + width/2 for xi in x]

plt.bar(x_left, visitors_2022, width, label="2022")
plt.bar(x_right, visitors_2023, width, label="2023")

plt.xticks(x, cities)
plt.title("Tourists in Cities 2022 vs 2023")
plt.xlabel("Cities")
plt.ylabel("Visitors (millions)")
plt.legend()
plt.show()
```



4) Scatter plot with triangle and pentagon markers

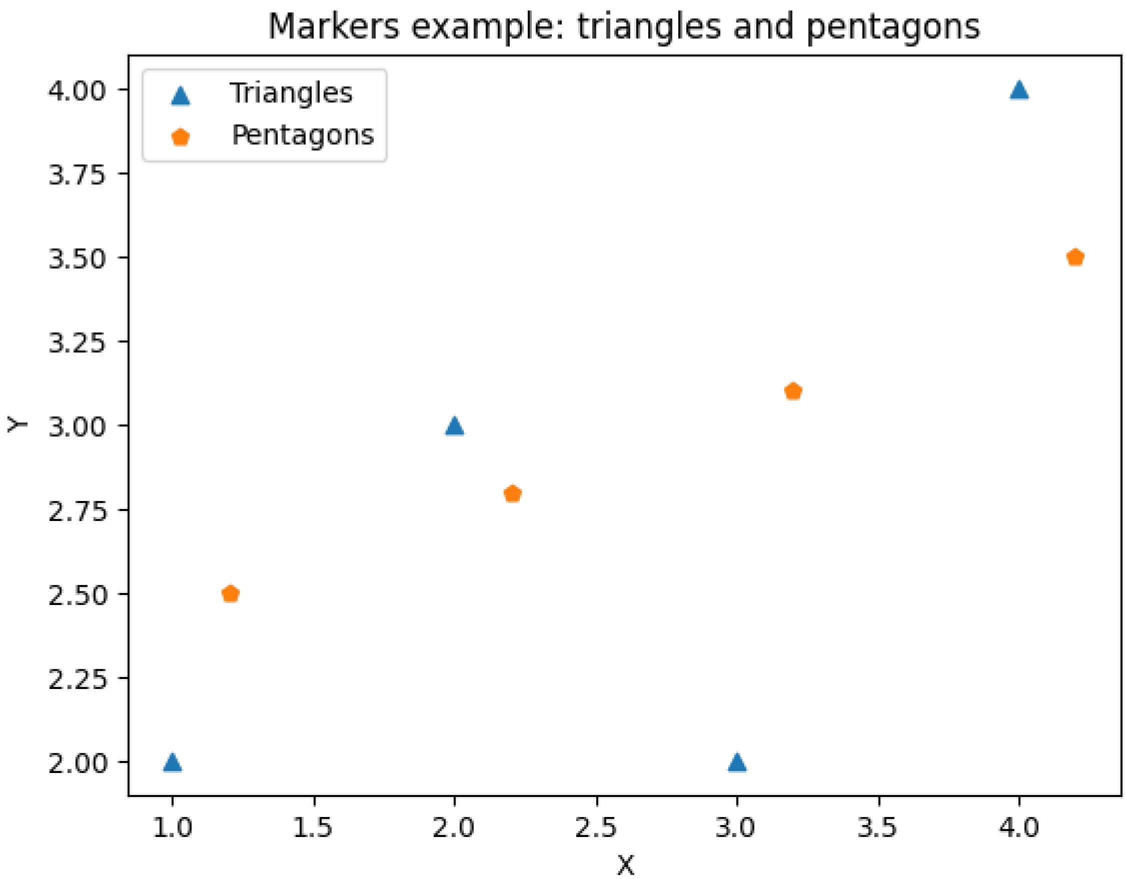
In [238…

```
x1 = [1, 2, 3, 4]
y1 = [2, 3, 2, 4]

x2 = [1.2, 2.2, 3.2, 4.2]
y2 = [2.5, 2.8, 3.1, 3.5]

plt.scatter(x1, y1, marker='^', label='Triangles')  # triangle up
plt.scatter(x2, y2, marker='p', label='Pentagons')  # pentagon

plt.title("Markers example: triangles and pentagons")
plt.xlabel("X")
plt.ylabel("Y")
plt.legend()
plt.show()
```

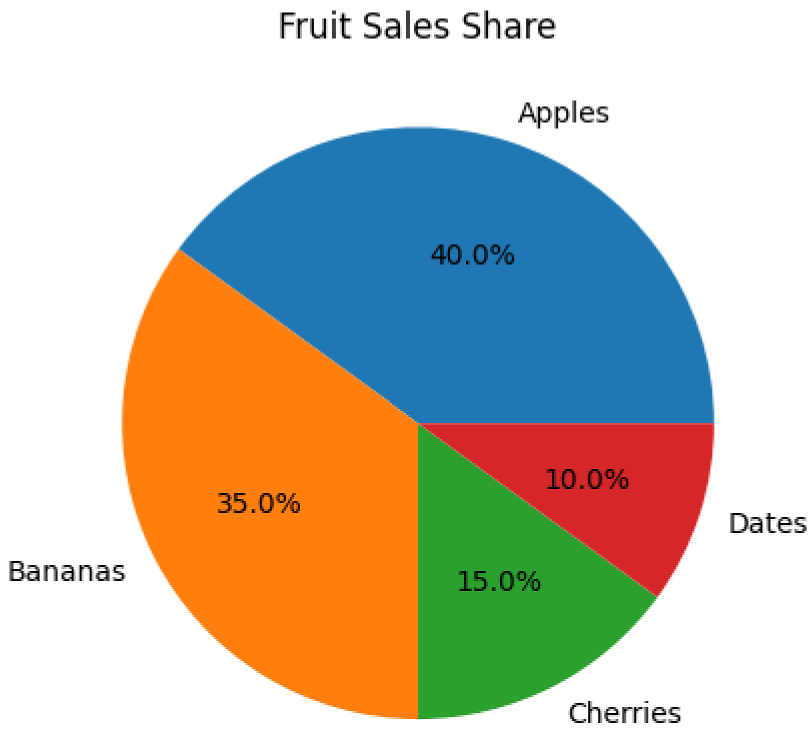


5) Pie chart

In [239…

```
sizes = [40, 35, 15, 10]
labels = ["Apples", "Bananas", "Cherries", "Dates"]

plt.pie(sizes, labels=labels, autopct='%1.1f%%')
plt.title("Fruit Sales Share")
plt.show()
```

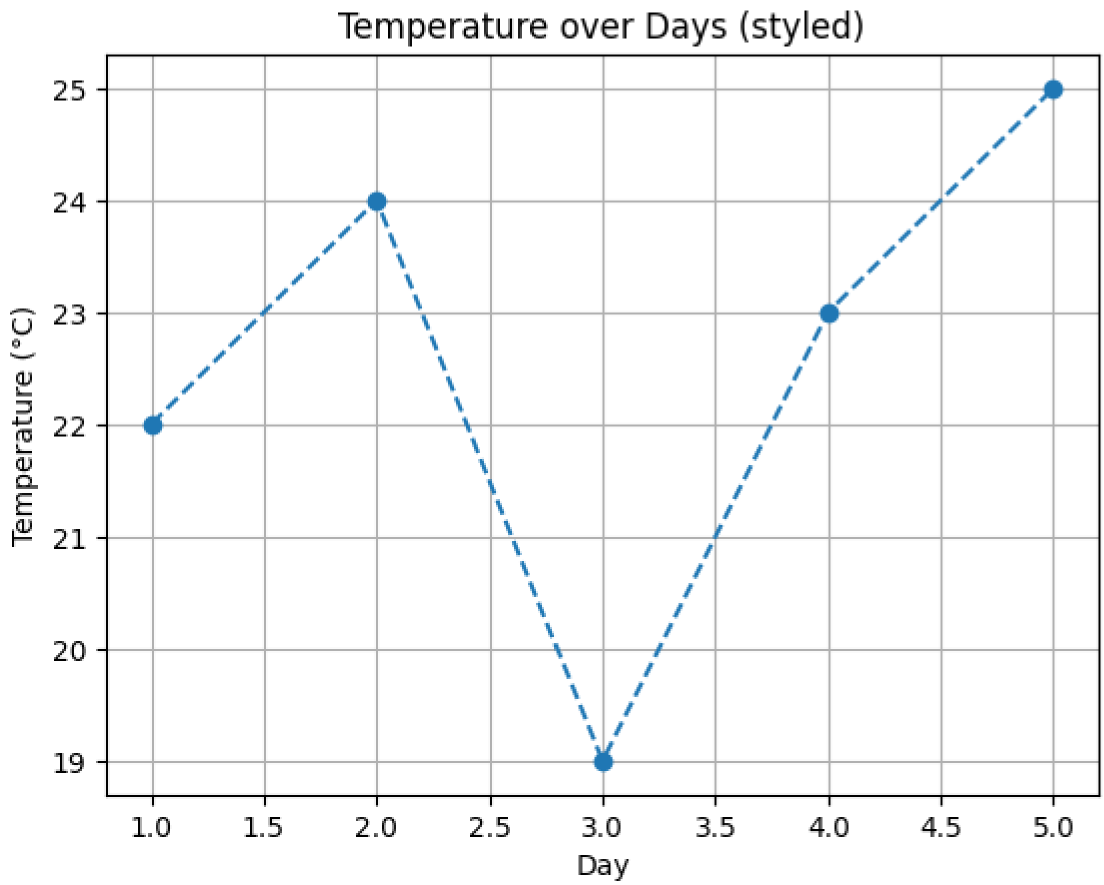


6) Customizing charts

In [240…

```
days = [1, 2, 3, 4, 5]
temperature = [22, 24, 19, 23, 25]

# Example of markers and line style together; color left to default.
plt.plot(days, temperature, marker='o', linestyle='--')
plt.title("Temperature over Days (styled)")
plt.xlabel("Day")
plt.ylabel("Temperature (°C)")
plt.grid(True)
plt.show()
```



Basic statistics on lists

In [241…

```
import statistics as stats

grades = [90, 85, 78, 92, 88, 76, 95, 89, 85]

print("mean     :", stats.mean(grades))
print("median   :", stats.median(grades))
print("mode     :", stats.mode(grades))       # may raise if no unique mode
print("multimode:", stats.multimode(grades))  # returns list of all modes
```

```
mean     : 86.44444444444444
median   : 88
mode     : 85
multimode: [85]
```

Manual average (good to understand)

In [242…

```
values = [2, 4, 6, 8]
avg = sum(values) / len(values)
print("manual average:", avg)
```

```
manual average: 5.0
```

List Comprehensions (optional)

In [243…

```python
# Squares of 0..5
squares = [x*x for x in range(6)]
print(squares)

# Keep only even numbers
evens = [x for x in range(10) if x % 2 == 0]
print(evens)
```

```
[0, 1, 4, 9, 16, 25]
[0, 2, 4, 6, 8]
```

Very quick NumPy intro

NumPy arrays are like lists but optimized for **fast math**.

- Create from a list: `np.array([1,2,3])`
- Vectorized math: add, multiply, etc. works **elementwise**
- Useful creators: `np.arange`, `np.linspace`
- Basic stats: `.mean()`, `.sum()`, `.min()`, `.max()`

In [244…

```python
import numpy as np

# Create arrays
a = np.array([1, 2, 3, 4])
b = np.array([10, 20, 30, 40])

print("a:", a)
print("b:", b)

# Elementwise operations
print("a + b     ->", a + b)
print("a * 2     ->", a * 2)      # broadcasting with a scalar
print("b - a     ->", b - a)
print("a ** 2    ->", a ** 2)     # power elementwise

# Quick creators
c = np.arange(0, 10, 2)           # 0,2,4,6,8
d = np.linspace(0, 1, 5)          # 5 points between 0 and 1
print("arange:", c)
print("linspace:", d)

# Basic stats
print("mean(a):", a.mean())
print("sum(b):", b.sum())
print("min(b):", b.min(), "max(b):", b.max())

# Shape and dtype
print("shape of a:", a.shape, "dtype:", a.dtype)
```

```
a: [1 2 3 4]
b: [10 20 30 40]
a + b      -> [11 22 33 44]
a * 2      -> [2 4 6 8]
b - a      -> [ 9 18 27 36]
a ** 2     -> [ 1  4  9 16]
arange: [0 2 4 6 8]
linspace: [0.   0.25 0.5  0.75 1.  ]
mean(a): 2.5
sum(b): 100
min(b): 10 max(b): 40
shape of a: (4,) dtype: int64
```

```
d = np.linspace(0, 1, 5)   # 5 points between 0 and 1
```

What does `linspace` do???

- **`np.linspace(start, stop, num)`** generates `num` evenly spaced numbers **between** `start` and `stop`, inclusive.

So here:

- `start = 0`
- `stop = 1`
- `num = 5`

Step calculation

It divides the interval `[0, 1]` into 4 equal steps (because 5 points create 4 gaps).

Step size =

$$\frac{stop - start}{num - 1} = \frac{1-0}{5-1} = \frac{1}{4} = 0.25$$

Actual values

So the points are:

```
[0.00, 0.25, 0.50, 0.75, 1.00]
```

Why use it?

- When you want **evenly spaced samples** from a range.
- Useful in plotting, interpolation, or numerical simulations.
- Unlike `np.arange`, `linspace` ensures that both endpoints are included (unless you specify `endpoint=False`).

Extra options

- `endpoint=False` → excludes the stop value. Example:

  ```
  np.linspace(0, 1, 5, endpoint=False)
  # [0.  0.2  0.4  0.6  0.8]
  ```
- `retstep=True` → also returns the step size:

  ```
  np.linspace(0, 1, 5, retstep=True)
  # (array([0.  , 0.25, 0.5 , 0.75, 1.  ]), 0.25)
  ```

Notice the difference between a list array and a numpy array!

In [245…
```
x = [1, 2, 3, 4]
z = [10, 20, 30, 40]
print("x + z     ->",x + z)
```

```
x + z     -> [1, 2, 3, 4, 10, 20, 30, 40]
```

In [246…
```
a = np.array([1, 2, 3, 4])
b = np.array([10, 20, 30, 40])
print("a + b     ->", a + b)
```

```
a + b     -> [11 22 33 44]
```

NumPy: Counting, Summing, and Frequencies

In [247…
```
import numpy as np # I'm only putting it here to show we need it, it doesn't

years = np.array([2010, 2015, 2020, 2018, 2020, 2012, 2019, 2021, 2015, 2020
```

Count matches (booleans: True = 1, False = 0)

In [248…
```
# Count years of
count_2020_plus = np.count_nonzero(years >= 2020)
print("Frequency of 2020+:", count_2020_plus)

# Count prices >= 2015 but below 2020
count_15_upto_20 = np.count_nonzero((years >= 2015) & (years < 2020))
print("Prices >=2015 (but not 2020+):", count_15_upto_20)
```

```
Frequency of 2020+: 4
Prices >=2015 (but not 2020+): 6
```

Note: `np.sum(grades)` adds the numeric values.

`np.sum(grades == 100)` adds booleans → a **count**.

Sum the values that match a condition (masking)

In [249…
```
prices = np.array([150, 200, 99, 300, 150, 220, 180, 99, 250, 150, 175, 99])

# Sum of prices >= 180 but not 300
sum_180_up = np.sum(prices[(prices >= 180) & (prices < 250)])
print("Sum of 180–249:", sum_180_up)
```

```
Sum of 180–249: 600
```

Masking rule: `array[condition]` keeps only the elements where `condition` is `True`.

Frequencies: unique values and counts

In [250…
```
values, counts = np.unique(years, return_counts=True) # This returns a tuple
print("values:", values)   # sorted unique grades
print("counts:", counts)   # frequency of each value

# Optional: dict form (value -> count)
freq = dict(zip(values, counts))
print(freq)
```

```
values: [2010 2012 2015 2017 2018 2019 2020 2021]
counts: [1 1 3 1 1 1 3 1]
{2010: 1, 2012: 1, 2015: 3, 2017: 1, 2018: 1, 2019: 1, 2020: 3, 2021: 1}
```

If you just `print(np.unique(grades, return_counts=True))`, you'll see a **tuple**: `(array_of_values, array_of_counts)`.

Optional: percentages of groups

In [251…
```
total = years.size

per_2020_plus   = count_2020_plus / total
per_15_upto_20  = count_15_upto_20 / total

print("Pct 2020 plus:", per_2020_plus)
print("Pct 15–19:", per_15_upto_20)
```

```
Pct 2020 plus: 0.3333333333333333
Pct 15–19: 0.5
```

Recap

- While loop: repeats while condition is True; indentation defines the body.
- Counter = fixed step; Accumulator = adds varying values.
- Lists: `[ ]`, index starts at 0; use `len()` for length; slicing uses `[start:end:step]`.

- Modify lists: `append`, index assign, `insert`, `extend`, `remove`, `pop`, `del`, `clear`.
- For loops: iterate items directly, or with `range(len(...))`, or `enumerate`.
- `break` / `continue` exist, but **avoid `break` by default**.
- Matplotlib: line, bar, scatter, pie; markers/linestyles table above.
- NumPy arrays: fast math, vectorized operations, simple creators/stats.
- **Count matches:** `np.count_nonzero(condition)` (clear), or `(condition).sum()`
- **Sum matches:** `np.sum(arr[condition])`
- **Unique + counts:** `np.unique(arr, return_counts=True)`
- **Combine conditions:** `&` (and), `|` (or), `~` (not)

Some interesting behaviours

In [252…

```python
valid = False
number = 1
while not valid:
    value = input("Enter a number: ")
    if value.replace(".", "", 1).isdigit():
        #if you don't even want floats just take out '.replace(".", "", 1)'
        number = int(value)   # or float(value) depending what you want
        valid = True
    else:
        print(f"Invalid input '{value}', please enter a valid number")

print("valid number: ", number)
```

```
valid number:  4
```

Step-by-step explanation of string operation - The more you know!

- `.replace(old, new, count)` creates a **new string** — it does not modify the original one, because strings are immutable in Python.

- For example:

  ```python
  s = "12.3"
  s.replace(".", "", 1)
  print(s)   # still "12.3"
  ```

- To actually change it, you must reassign:

  ```python
  s = s.replace(".", "", 1)
  print(s)   # now "123"
  ```

Why use `1` as the count?

- With `1` → only one dot is removed:

```python
"12.3.4".replace(".", "", 1)    # "123.4"
```

  This still has a dot, so `.isdigit()` returns `False`. The invalid input `"12.3.4"` is rejected.

- With `2` → two dots are removed:

```python
"12.3.4".replace(".", "", 2)    # "1234"
```

  `.isdigit()` returns `True`. `"12.3.4"` would be incorrectly accepted as valid.

Using `1` enforces the rule of **at most one decimal point**, which is how proper numbers are written.

Why negatives are rejected

- Input: `"-12.3"`
- After `.replace(".", "", 1)` → `"-123"`
- `.isdigit()` sees the `-` and returns `False`.
- Result: negative numbers are not accepted.

Allowing negatives with `lstrip("-")`

To allow a **leading minus**, we can combine `.lstrip("-")` with `.replace`:

```python
if value.lstrip("-").replace(".", "", 1).isdigit():
    number = float(value)
```

- `.lstrip("-")` only removes a minus sign **from the start of the string**.

```python
"-123".lstrip("-")    # "123"
"12-3".lstrip("-")    # "12-3"  (unchanged)
```

  This prevents invalid inputs like `"12-3"` from being accepted.

- Because `.lstrip` returns a new string, we can **chain** it directly with `.replace(".", "", 1)`.

- There's no need for a temporary variable, since both methods already produce new strings each time.

This way the validation works for:

- `"42"` → valid
- `"3.14"` → valid
- `"-42"` → valid with `lstrip`
- `"-3.14"` → valid with `lstrip`
- `"12.3.4"` → rejected
- `"12-3"` → rejected
- `"abc"` → rejected

`np.linspace(start, stop, num)` confusion vs arange

- Generates exactly `num` points evenly spaced between `start` and `stop`.
- By default, it **includes** the stop value.
- Example:

  ```
  np.linspace(0, 1, 5)
  ```
  Output:

  ```
  [0.   0.25 0.5  0.75 1.  ]
  ```
  → 5 points from 0 to 1, inclusive.

- If you set `endpoint=False`, the stop is excluded:

  ```
  np.linspace(0, 1, 5, endpoint=False)
  ```
  Output:

  ```
  [0.  0.2 0.4 0.6 0.8]
  ```
  → 5 points, but the last one is just before 1.

`np.arange(start, stop, step)`

- Generates numbers starting at `start`, increasing by `step`, stopping **just before `stop`** (like Python's built-in `range`).
- The number of points depends on step size, not a fixed count.
- Example:

  ```
  np.arange(0, 1.01, 0.25)
  ```
  Output:

  ```
  [0.   0.25 0.5  0.75 1.  ]
  ```
  → Same sequence as `linspace(0, 1, 5)`, but here you control the **step** (0.25), not the number of points.

- If step doesn't divide evenly, the stop might not be hit exactly:

  ```
  np.arange(0, 1, 0.3)
  ```
  Output:

  ```
  [0.  0.3  0.6  0.9]
  ```
  → Notice it stopped before 1.2 (would exceed 1).

When to use which?

- Use `linspace` when you want a specific number of evenly spaced samples (e.g., 100 points for plotting a curve).
- Use `arange` when you care about step size (e.g., count by 0.25 seconds, generate multiples of 2).

In [253…

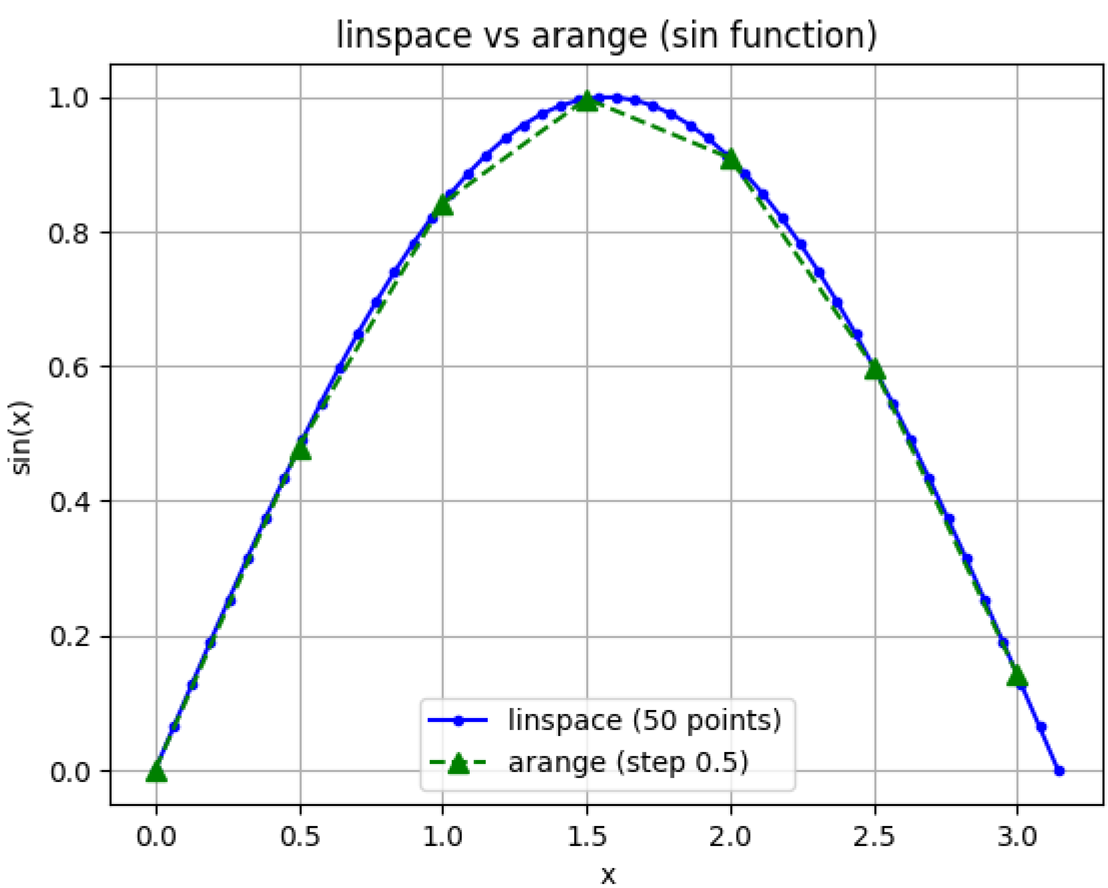
```
import numpy as np
import matplotlib.pyplot as plt

# Generate data with linspace (smooth curve)
x_lin = np.linspace(0, np.pi, 50)
y_lin = np.sin(x_lin)

# Generate data with arange (uneven step, 0.3 to make points less aligned)
x_ar = np.arange(0, np.pi , 0.5)
y_ar = np.sin(x_ar)

# Plot
plt.plot(x_lin, y_lin, label="linspace (50 points)", color="blue", linestyle
plt.plot(x_ar, y_ar, label="arange (step 0.5)", color="green", linestyle="--

plt.title("linspace vs arange (sin function)")
plt.xlabel("x")
plt.ylabel("sin(x)")
plt.legend()
plt.grid(True)
plt.show()
```



In [254…

```
#x_ar = np.arange(0, np.pi , 0.5)
#y_ar = np.sin(x_ar)
print("x arange values (0, np.pi , 0.5): ")
print(x_ar)
print("y arange values : ")
print(y_ar)
```

```
x arange values (0, np.pi , 0.5):
[0.  0.5 1.  1.5 2.  2.5 3. ]
y arange values :
[0.         0.47942554 0.84147098 0.99749499 0.90929743 0.59847214
 0.14112001]
```

In [255…

```
#x_lin = np.linspace(0, np.pi, 50)
#y_lin = np.sin(x_lin)
print("x linspace values np.linspace(0, np.pi, 50): ")
print(x_lin)
print("y linspace values : ")
print(y_lin)
```

```
x linspace values np.linspace(0, np.pi, 50):
[0.         0.06411414 0.12822827 0.19234241 0.25645654 0.32057068
 0.38468481 0.44879895 0.51291309 0.57702722 0.64114136 0.70525549
 0.76936963 0.83348377 0.8975979  0.96171204 1.02582617 1.08994031
 1.15405444 1.21816858 1.28228272 1.34639685 1.41051099 1.47462512
 1.53873926 1.60285339 1.66696753 1.73108167 1.7951958  1.85930994
 1.92342407 1.98753821 2.05165235 2.11576648 2.17988062 2.24399475
 2.30810889 2.37222302 2.43633716 2.5004513  2.56456543 2.62867957
 2.6927937  2.75690784 2.82102197 2.88513611 2.94925025 3.01336438
 3.07747852 3.14159265]
y linspace values :
[0.00000000e+00 6.40702200e-02 1.27877162e-01 1.91158629e-01
 2.53654584e-01 3.15108218e-01 3.75267005e-01 4.33883739e-01
 4.90717552e-01 5.45534901e-01 5.98110530e-01 6.48228395e-01
 6.95682551e-01 7.40277997e-01 7.81831482e-01 8.20172255e-01
 8.55142763e-01 8.86599306e-01 9.14412623e-01 9.38468422e-01
 9.58667853e-01 9.74927912e-01 9.87181783e-01 9.95379113e-01
 9.99486216e-01 9.99486216e-01 9.95379113e-01 9.87181783e-01
 9.74927912e-01 9.58667853e-01 9.38468422e-01 9.14412623e-01
 8.86599306e-01 8.55142763e-01 8.20172255e-01 7.81831482e-01
 7.40277997e-01 6.95682551e-01 6.48228395e-01 5.98110530e-01
 5.45534901e-01 4.90717552e-01 4.33883739e-01 3.75267005e-01
 3.15108218e-01 2.53654584e-01 1.91158629e-01 1.27877162e-01
 6.40702200e-02 1.22464680e-16]
```

In [255…