1. What are some key differences between lists, tuples, and dictionaries in Python, and when would you choose each?

   “A list in Python is ordered and mutable, so I can add, remove, or modify elements in place. A tuple is similar to a list but is immutable—it cannot be changed after creation. It’s often used for fixed collections of data or as a key in a dictionary because it’s hashable. A dictionary, on the other hand, stores key-value pairs for fast lookups.

   I would use a list if I need to maintain order and possibly alter the collection; a tuple if I need a fixed, unchangeable sequence; and a dictionary if I need to associate a unique key with a value for quick lookups.”

2. In Python, a list can store items of different types (e.g., integers, strings, objects) in the same list. Given that C#’s `List<T>` is strongly typed, how would you store heterogeneous data in a single list?

   Answer:
   `List<T>` in C# requires all elements to be of the same type `T`.
   If you need to store different types (heterogeneous data), you can declare a list of a common base type—often `List<object>` or a shared interface/base class if all objects derive from the same hierarchy.

   e.g.
   // A list that can hold any type because it uses 'object'

   List<object> heterogeneousList = new List<object>

```
{
42, // int
"Hello World", // string
3.14159, // double
new Person("Alice") // a custom class
};
```

3. I have 2 two input boxes on a screen, and a submit button. When I click the button, text entered in the box1, will be used to query a db for some row of data. The returned data would fill box 2. We do not want box 2 to allow user manual input:
    A. How would we prevent manual edits of box 2?
    B. We do not want the user to press the button more than once, to prevent multiple queries from a client before a response is received. How would you implement this (higher level design)

4. In dash, Input triggers callbacks, while States are only used when callback is called.

    import dash from dash import html, dcc, Input, Output, State

    app = dash.Dash(__name__)
    app.layout = html.Div([dcc.Input(id='name-input', type='text', placeholder='Enter your name'),
    html.Button('Submit', id='submit-button'),
    html.Div(id='greeting-output')])

```
@app.callback(
    Output('greeting-output', 'children'),
    Input('submit-button', 'n_clicks'),
    State('name-input', 'value')
)
def update_greeting(n_clicks, name):
    if n_clicks is None:
        # Before the button is ever clicked
        return "No greeting yet!"
    else:
        # Update greeting after the button is clicked
        return f"Hello, {name}!"
```

```
@app.callback(
    Output('greeting-output', 'children'),       # We want to update this output
    Input('submit-button', 'n_clicks'),          # The callback fires on button clicks
    Input('name-input', 'value')                 # The current text is read only when button is clicked
)
def update_greeting(n_clicks, name):
    if n_clicks is None:
        # Before the button is ever clicked
        return "No greeting yet!"
    else:
        # Update greeting after the button is clicked
        return f"Hello, {name}!"
```

Which of the two would have unexpected results, why?

5.

You're building a web application (using Plotly Dash) that displays two input boxes and a **Submit** button.

- **Box 1**: Accepts user input (enabled for typing).
- **Box 2**: Displays data fetched from the database (read-only).
- **Submit button**: When clicked, it triggers a Python callback that:
    A. Read the value from **Box 1**.
    B. Fetches corresponding data from an SQL database.
    C. Returns the data to populate **Box 2**.

Because Dash executes callbacks on the server side and does not allow partial/intermediate UI updates within a single callback, you need to ensure that:

The user **cannot click Submit again** while the system is still fetching data.

```
from dash import html, dcc, Input, Output, State, no_update

app.layout = html.Div([ dcc.Input(id='box1', type='text', placeholder='Enter Text'),
dcc.Input(id='box2', type='text', placeholder='Enter Text'),
html.Button('Submit', id='submit-button'),
@app.callback(
    Output('box2', 'value'),        # We want to update this output
    Input('submit-button', 'n_clicks'),          # The callback fires on button clicks
    State('box1', 'value')                  # The current text is read only when button is clicked
)
def update_ui(n_clicks, box1_val):
    if n_clicks is None:
        # Before the button is ever clicked
        return no_update
    else:
        data = get_info(box1_value)
        return data
```

```
dcc.Loading(id="ls-loading-2",children=[html.Div([html.Div(id="
ls-loading-output-2")])],type="circle")
```

6. In python, what is the difference between a list comprehension and a generator expression?

   Answer:
   List Comprehension: Creates a **list** in memory immediately.
   Generator Expression: Creates a **generator** that yields items **on-the-fly**, consuming less memory when dealing with large or infinite sequences.

   B. What is the main syntax difference?

my_list = [x * 2 for x in range(5)]

my_gen = (x * 2 for x in range(5))

C. When would you use one over the other?

7. What is the difference between `git fetch` and `git pull`, and when would you use each command?

   “`git fetch` simply downloads commits, files, and references from the remote repository but doesn’t integrate them into my local branches. It lets me see what changes are available without affecting my working copy. `git pull` does a `fetch` and then merges (or rebases) those changes into my current local branch. I typically use `git fetch` when I want to review the latest commits before deciding if or how to merge, and `git pull` once I’m ready to bring those changes into my local branch.”

8. How do you decide when to create a function versus a class?
   Simplicity / Single Responsibility

- If the behavior you need is straightforward and stateless—i.e., it doesn’t require maintaining or tracking data across multiple method calls—then a *function* is usually enough.
- For example, a short utility to validate an email string or calculate a factorial is best expressed as a single function because it focuses on *one job* without complex data structures or multiple interactions.

Stateful Behavior or Multiple Related Behaviors

- If you need to maintain state over time (e.g., a user session, a cache of intermediate results, or configuration settings) or if you have several related methods that operate on the same data, then a *class* is more appropriate.
- Classes let you store state in instance variables and group methods logically around that data, improving readability and maintainability as your codebase grows.

Extensibility & Reusability

- Classes are often easier to extend; you can subclass to modify or add behavior without altering the original implementation. If you foresee needing flexible or evolving functionality, a class can future-proof your design.

- Conversely, a simple function is often more reusable if it just takes input, does a computation, and returns output (e.g., a standalone library function).

Readability & Organization

- Well-structured classes can improve readability by acting as a “conceptual” container for closely related operations.
- However, if you find yourself writing a class with only one method (and no meaningful state), that’s often an indication a function might be simpler and clearer.

Performance Considerations

- While performance differences between functions and classes are minimal in many cases, functions can have slightly less overhead if you’re calling them repeatedly in tight loops.
- But maintainability and clarity typically matter more than micro-optimizations.

In short:

I. Use a **function** for discrete, stateless tasks.
II. Use a **class** if you need to model stateful behavior, group related methods, or expect to extend functionality in the future.

9. What is a pandas series?
A **Pandas Series** is a one-dimensional labeled array that can hold any data type (integers, floats, strings, Python objects, etc.). You can think of it like a single column in a spreadsheet with an attached **index** that labels each element in that column.

10. Explain the concepts of inheritance and polymorphism in OOP.

“Inheritance allows a class (child) to inherit attributes and methods from another class (parent). This lets us reuse code and extend functionality instead of rewriting everything. Polymorphism means we can use the same function or method name in different classes, but each class can implement it differently—promoting flexibility. For example, you might have a base `Shape` class with a `draw()` method, and both `Circle` and `Square` override `draw()` to implement their own drawing logic.”

11.

What are the key differences between localStorage and sessionStorage in HTML5, and when would you use each?

Answer:

- **Scope & Lifetime**:
  - **localStorage** stores data with no expiration date. It persists even when the browser is closed and reopened.
  - **sessionStorage** is limited to the current browser tab/session. Once the tab is closed, the data is cleared.
- **Accessibility**:
  - **localStorage** data is shared across all tabs/windows from the same origin (domain, protocol, and port).
  - **sessionStorage** is unique to each tab/window.
- **Use Cases**:
  - Use **localStorage** for user preferences, theme settings, or data that should persist across sessions.
  - Use **sessionStorage** for transient data that is only relevant for a single browsing session (e.g., form data that you don't want to persist).

12. .

Can you explain how Promises work in JavaScript and how they differ from callbacks?

Answer:

Promises are objects that represent a value that may be available now, in the future, or never. They have three states: *pending*, *fulfilled*, and *rejected*. You use `.then()` for fulfilled resolutions and `.catch()` for rejections, helping write more readable asynchronous code.

Callbacks are functions passed as an argument to be executed once an asynchronous operation is completed. Callbacks can become messy (often leading to "callback hell") if multiple asynchronous operations depend on each other.

Benefits of Promises:

a. More structured and chained syntax (`.then().then().catch()`), leading to better readability.

b. Easier error handling with `.catch()`.
c. Built-in methods like `Promise.all()` and `Promise.race()` facilitate handling multiple promises simultaneously.

13. _

What are semantic elements in HTML5, and why are they important?

Answer:

Semantic elements are HTML5 tags that provide meaningful structure to the content, such as `<header>`, `<footer>`, `<main>`, `<article>`, `<section>`, `<nav>`, and `<aside>`.

Importance:

a. **Improved Accessibility**: Screen readers and other assistive technologies can use semantic tags to better understand and navigate the page structure.
b. **SEO Benefits**: Search engines can more effectively parse the content, potentially improving ranking and discoverability.
c. **Maintainable Code**: Semantic markup makes the HTML more readable and maintainable for other developers.

14. __

What is the difference between the `==` operator and the `===` operator in JavaScript, and why is this distinction important?

Answer:

`==` (loose equality): Performs type coercion before comparing two values. For instance, `2 == "2"` is `true` because `"2"` is coerced to a number before comparison.

`===` (strict equality): Compares both type and value without performing type coercion. For example, `2 === "2"` is `false` because one is a number and the other is a string.

Importance:

a. Prevents unintended equality matches or type conversions that can lead to subtle bugs.
b. Encourages explicit handling of type conversions (e.g., using `parseInt()` or `parseFloat()` when dealing with numeric strings).