🔹 Step 1: Why dictionaries exist

In **lists**, data is accessed by **index numbers** (0, 1, 2 …).
Example:

```
fruits = ["Ginger", "Lemon", "Mint"]
print(fruits[1])  # Lemon
```

- The problem? You have to **remember positions**. If the order changes, `"Lemon"` may no longer be at index `1`.

Dictionaries solve this problem with **named indexing**.
Instead of numbers, you use **keys** (like labels).
Example:

```
chai_order = {
    "type": "Masala Chai",
    "size": "Large",
    "sugar": "2 spoons"
}
print(chai_order["type"])  # Masala Chai
```

-

So dictionaries are basically like **real dictionaries**: a word (`key`) maps to its meaning (`value`).

🔹 Step 2: Creating a dictionary

Two ways:

Using `dict()` function

```
chai_order = dict(type="Masala Chai", size="Large", sugar="2 spoons")
```

1.

Using **curly braces `{}`** (more common):

```
chai_order = {
```

```
    "type": "Masala Chai",
    "size": "Large",
    "sugar": "2 spoons"
}
```

2.

Both mean the same thing.

🔹 Step 3: Accessing dictionary values

You don't use numbers like in lists. You use **keys**.

```
print(chai_order["type"])   # Masala Chai
print(chai_order["size"])   # Large
```

-
- Important: **Order doesn't matter** in a dictionary.
    - The items may print in different orders, but since you use keys, you'll always get the right value.

🔹 Step 4: Adding new data

If you want to add a new property:

```
chai_order["extra"] = "Ginger"
print(chai_order)
# {'type': 'Masala Chai', 'size': 'Large', 'sugar': '2 spoons',
'extra': 'Ginger'}
```

It's just like updating an object in JavaScript if you've seen that:

```
chai_order["extra"] = "Ginger";
```

🔹 Step 5: Deleting data

If you don't want `"liquid": "milk"` anymore:

```
del chai_order["liquid"]
```

Now that key-value pair is gone.

🔹 Step 6: Membership testing (`in`)

You can check if a **key** exists:

```
print("sugar" in chai_order)   # True
print("price" in chai_order)   # False
```

This is like asking: *Does this word exist in the dictionary?*

🔹 Step 7: Getting keys, values, items

Python gives you helper methods:

- `dict.keys()` → all keys
- `dict.values()` → all values
- `dict.items()` → all pairs (key + value)

Example:

```
print(chai_order.keys())   # dict_keys(['type', 'size', 'sugar'])
print(chai_order.values()) # dict_values(['Masala Chai', 'Large', '2
spoons'])
print(chai_order.items())  # dict_items([('type', 'Masala Chai'),
('size', 'Large'), ('sugar', '2 spoons')])
```

🔹 Step 8: Real-world analogy

Think of a **hotel menu** stored in a dictionary:

```
menu = {
    "Burger": 120,
    "Pizza": 250,
    "Pasta": 150
}
```

- You don't order by saying "Give me item at position 0".
- You say "Give me a Pizza". That's how dictionaries work: **direct lookup by name**.

🔹 Instructor's example summary

- First compared **lists (indexing with numbers)** vs. **dictionaries (indexing with names)**.
- Showed how to:
    - Create (`dict` or `{}`)
    - Access (using `keys`)
    - Add values (`dict[key] = value`)
    - Delete (`del dict[key]`)
    - Test membership (`key in dict`)
    - Get all keys/values/items (`.keys()`, `.values()`, `.items()`)

✅ **In short:**
A Python dictionary is like a **real-world dictionary** or a **JSON object**.

- Keys = unique identifiers
- Values = associated data
- Advantage = fast lookup, no need to remember positions