🔹 What the Instructor is Saying

1. **Advanced Data Types**
    - Python has **basic built-in types** (int, float, string, list, dict, set, tuple).
    - But there are also **advanced data types** that you don't get by default — you need to **import modules (extra code)** to use them.
2. **Modules (third-party code)**
    - A *module* is simply **somebody else's code** that you bring into your program with `import`.
    - Example: `import datetime`, `import collections`, `import arrow`.
3. **Examples of Advanced Data Types**
    - **datetime** → work with dates & times.
    - **time** → only time values.
    - **calendar** → calendar operations.
    - **timedelta** → difference between two dates/times.
        - Example: time between *order placed* and *order delivered*.
    - **arrow** / **dateutil** → powerful third-party libraries for working with time zones, formatting, etc.

Example with `arrow`:

```
 import arrow

brewing_time = arrow.utcnow()      # current UTC time
print(brewing_time)

# Convert UTC to another timezone
rome_time = brewing_time.to('Europe/Rome')
```

```
print(rome_time)
```

4.

🔹 Collections Module (More Advanced Types)

The instructor then moves to the **`collections` module**, which gives extra data structures:

namedtuple → Tuples with names for fields.

```
from collections import namedtuple

ChaiProfile = namedtuple("ChaiProfile", ["flavor", "aroma", "color"])
masala_chai = ChaiProfile("spicy", "strong", "brown")

print(masala_chai.flavor)  # spicy
```

-
- **deque (deck)** → A double-ended queue (fast append/remove from both sides).
- **Counter** → Counts elements automatically.
- **OrderedDict** → Dictionary that remembers the order of items.
- **defaultdict** → Dictionary with default values.
- **ChainMap, UserDict**, etc.

These are **special versions** of lists/dictionaries/tuples that solve specific problems.

🔹 Instructor's Note

- You don't need to master these **right now** if you're a beginner.
- This is more of a **"bonus" preview** so you know they exist.

- Later, when you face real-world problems (like handling time zones, counting things, or storing structured data), you'll use them.

✅ Key Takeaway

- **Basic types**: int, float, string, list, dict, tuple, set → you already know.
- **Advanced types**: datetime, timedelta, calendar, arrow, dateutil, collections (namedtuple, deque, counter, etc.).
- They're not built-in in the same way, so you need to **import modules**.
- You'll use them **later in real projects**, not in the early learning stage.

🔹 What is a `namedtuple`?

Normally, a **tuple** in Python is just a collection of values:

```
chai = ("spicy", "strong", "brown")
print(chai[0])   # spicy
```

- Problem → You must remember that index `0` = flavor, `1` = aroma, `2` = color. That's hard to read.
- A `namedtuple` solves this by letting you give **names to tuple fields**.
  👉 It's like a **lightweight class**.

🔹 Example with `namedtuple`

```
from collections import namedtuple

# Define a blueprint (like a class)
ChaiProfile = namedtuple("ChaiProfile", ["flavor", "aroma", "color"])
```

```
# Create an object using that blueprint
masala_chai = ChaiProfile("spicy", "strong", "brown")

print(masala_chai)           # ChaiProfile(flavor='spicy',
aroma='strong', color='brown')
print(masala_chai.flavor)    # spicy
print(masala_chai.aroma)     # strong
print(masala_chai.color)     # brown
```

So instead of doing `chai[0]`, you do `chai.flavor`. Much easier to read. ✅

🔹 Why not just use a dictionary?

Good question! You *could* use a dictionary:

```
chai = {"flavor": "spicy", "aroma": "strong", "color": "brown"}
print(chai["flavor"])
```

But:

- Dicts are **bigger/slower** (more memory).
- Keys can be changed accidentally.
- Order wasn't guaranteed in old Python versions.

`namedtuple` is:

- **Fast & lightweight** like a tuple.
- **Readable** like a dictionary.
- **Immutable** (cannot change values after creation, safer).

🔹 Real-world analogy

Think of a **namedtuple** as a **fixed form**:

- Tuple = a list without names (`("Hari", 25, "India")`).
- Namedtuple = a form with **labels** (`Person(name="Hari", age=25, country="India")`).

🔹 Quick Example: Order Receipt

```
from collections import namedtuple

Order = namedtuple("Order", ["item", "price", "quantity"])

order1 = Order("Masala Chai", 20, 2)

print(order1.item)      # Masala Chai
print(order1.price)     # 20
print(order1.quantity)  # 2
```

This looks **much cleaner** than using `order[0]` or a plain tuple.

✅ **In short**:

- `namedtuple` = tuple + names for fields.
- Cleaner & safer than a normal tuple.
- Faster & lighter than a dictionary.
- Great for structured, read-only data.

1. Dicts are bigger/slower (more memory)

- A dictionary keeps **extra internal stuff** (hashing, references, dynamic resizing).
- A tuple (or namedtuple) is **much simpler**, so it takes less memory and is faster.
  👉 If you only need fixed fields, a `namedtuple` is better.

Example:

```
chai_dict = {"flavor": "spicy", "aroma": "strong", "color": "brown"}
chai_tuple = ("spicy", "strong", "brown")
```

The dict needs extra storage for keys (`flavor`, `aroma`, `color`), while the tuple just stores the values.

🔹 2. Keys can be changed accidentally

In a **dictionary**, the keys (labels) are just strings → you can **add, remove, or overwrite** them by mistake.

Example:

```
chai = {"flavor": "spicy", "aroma": "strong", "color": "brown"}

chai["flavor"] = "sweet"      # Key value changed accidentally
chai["extra"] = "lemon"       # New key added accidentally
del chai["aroma"]             # Key deleted accidentally

print(chai)
# {'flavor': 'sweet', 'color': 'brown', 'extra': 'lemon'}
```

👉 If this happens in a big program, you might **lose important data** or **mess up structure**.

But with `namedtuple`:

```
from collections import namedtuple

ChaiProfile = namedtuple("ChaiProfile", ["flavor", "aroma", "color"])
chai = ChaiProfile("spicy", "strong", "brown")
```

```
# chai.flavor = "sweet"  ❌ ERROR → immutable, you cannot change
# chai.extra = "lemon"   ❌ ERROR → cannot add new fields
```

✅ Structure is **locked and safe**.

🔹 3. Order wasn't guaranteed in old Python

- Before Python 3.7, dictionaries didn't always keep insertion order.

Example:

```
chai = {"flavor": "spicy", "aroma": "strong", "color": "brown"}
print(chai)
# Maybe: {'aroma': 'strong', 'flavor': 'spicy', 'color': 'brown'}
```

-
- With tuples or namedtuples, **order is always fixed**.

🔹 Why `namedtuple` is better in some cases

- ✅ **Fast & lightweight** like a tuple (low memory).
- ✅ **Readable** like a dict (`chai.flavor` instead of `chai[0]`).
- ✅ **Immutable** → cannot change accidentally.

So, when your data structure is **fixed** (like `flavor, aroma, color` for chai), use a **namedtuple**.
When you need **flexibility** (add/remove keys often), use a **dict**.