

# Lesson 02: Data Structures

Data Science with Python – BSc Course

Data Science Program

BSc Course

45 Minutes

## Previously: Variables and Types



**L01:** You stored one value in one variable: `price = 185.50`

**Problem:** You have 500 stock prices. A single variable won't cut it.

**Solution:** Data structures – containers that hold multiple values at once.

---

From single values to organized collections of data

## Learning Objectives

After this lesson, you will be able to:

- **Create** lists and access elements by index
- **Slice** lists to extract sub-sequences
- **Build** dictionaries for key-value data storage
- **Choose** the right structure for each use case
- **Apply** list comprehensions for efficient processing

**Finance Application:** Store portfolios as dictionaries, price histories as lists.

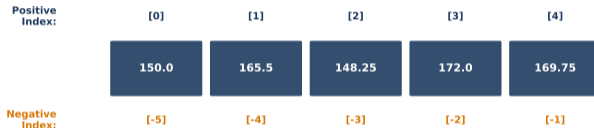
---

Data structures are containers for organizing information

## Lists: Ordered Sequences

A list is an ordered collection of items, accessed by position

### Python List Indexing



```
prices = [150.0, 165.5, 148.25, 172.0, 169.75]
```

```
prices[0] = 150.0 (first)
```

```
prices[-1] = 169.75 (last)
```

**Access:** `prices[0]` → first — `prices[-1]` → last — `prices[2]` → third

---

Indexing starts at 0, not 1. Negative indices count from the end.

## Creating and Modifying Lists

### Creating Lists:

```
# From literal values
prices = [150, 165, 148, 172]

# Empty list
signals = []

# Mixed types (allowed but rare)
info = ["AAPL", 185.5, True]

# From range
days = list(range(1, 6))

# [1, 2, 3, 4, 5]
```

### Modifying Lists:

```
prices = [150, 165, 148]

# Change element
prices[0] = 155

# Add to end
prices.append(172)

# Insert at position
prices.insert(1, 160)

# Remove by value
prices.remove(148)
```

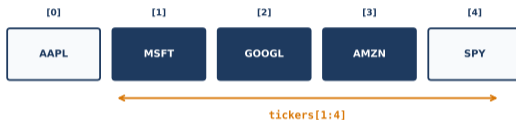
---

Lists are mutable – you can change, add, and remove elements

## Slicing: Extracting Sub-Sequences

**Syntax:** `list[start:end:step]` – end index is **exclusive**

### List Slicing: `list[start:end]`



Result: ["MSFT", "GOOGL", "AMZN"]

**start:** included

**end:** excluded

**step:** optional

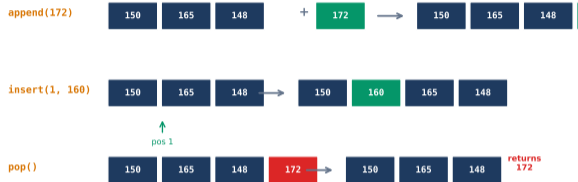
*Remember: End index is EXCLUSIVE*

**More:** `prices[0:3]` → first 3 — `prices[::2]` → every 2nd — `prices[::-1]` → reversed

---

Slicing creates a new list – the original is unchanged

## List Methods: Modifying Lists



*Methods modify the list in-place*

**Key distinction:** `sort()` modifies in place; `sorted()` returns a new list

---

Methods modify the list in-place (except `sorted()` which returns new list)

## Useful List Operations

### Aggregation:

```
prices = [150, 165, 148, 172]
len(prices) # 4
sum(prices) # 635
min(prices) # 148
max(prices) # 172
```

### Membership Test:

```
172 in prices # True
200 in prices # False
```

### Sorting:

```
prices.sort()
# [148, 150, 165, 172]
prices.sort(reverse=True)
# [172, 165, 150, 148]
```

### Counting:

```
grades = ["A", "B", "A", "C"]
grades.count("A") # 2
grades.index("B") # 1
```

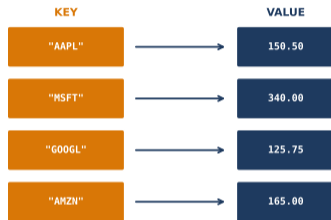
---

Built-in functions (len, sum, min, max) work on any sequence

## Dictionaries: Key-Value Pairs

A dictionary maps unique keys to values – like a phone book

### Dictionary: Key-Value Pairs



```
portfolio["AAPL"] = 150.50
```

*O(1) lookup time - very fast!*

**Methods:** `.keys()` – all keys — `.values()` – all values — `.items()` – pairs

---

Dictionaries provide  $O(1)$  lookup – instant access by key

## Working with Dictionaries

### Creating:

```
stock = {  
    "ticker": "AAPL",  
    "price": 185.50,  
    "shares": 100  
}
```

### Accessing:

```
stock["price"]      # 185.50  
stock.get("sector", "N/A")  
# "N/A" (safe access)
```

### Modifying:

```
# Update value  
stock["price"] = 190.00  
# Add new key  
stock["sector"] = "Tech"  
# Remove key  
del stock["shares"]
```

### Check Membership:

```
"ticker" in stock # True  
"volume" in stock # False
```

---

Use `.get()` for safe access – avoids `KeyError` on missing keys

## Tuples: Immutable Sequences

### Creating Tuples:

```
# Parentheses (optional)
coords = (40.71, -74.01)

# Without parentheses
trade = "AAPL", 185.50, 100

# Single element (note comma)
singleton = (42,)
```

### Accessing:

```
coords[0] → 40.71
trade[-1] → 100
```

### Why Tuples?

- **Immutable** – cannot be changed after creation
- Safer: prevents accidental modification
- Faster than lists (slight performance edge)
- Can be dictionary keys (lists cannot)

### Unpacking:

```
ticker, price, shares = trade
# ticker = "AAPL"
# price = 185.50
# shares = 100
```

---

Use tuples for data that should not change (coordinates, dates, records)

## Sets: Unique Collections

### Creating Sets:

```
tickers = {"AAPL", "MSFT", "AAPL"}  
# {"AAPL", "MSFT"} -- no duplicates!  
  
# From list  
unique = set([1, 2, 2, 3, 3])  
# {1, 2, 3}
```

### Set Operations:

A | B – union  
A & B – intersection  
A - B – difference

### Finance Use Case:

```
portfolio = {"AAPL", "MSFT", "GOOGL"}  
sp500 = {"AAPL", "MSFT", "AMZN"}  
  
# Stocks in both  
overlap = portfolio & sp500  
# {"AAPL", "MSFT"}  
  
# Only in portfolio  
unique_to_me = portfolio - sp500  
# {"GOOGL"}
```

**Key property:** Unordered, no duplicates

---

Sets: fast membership testing and duplicate removal

## Checkpoint: Which Structure?

**Q1:** Store 252 daily closing prices in order?

- (a) dict      (b) **list**      (c) set      (d) tuple

**Q2:** Map ticker symbols to company names?

- (a) list      (b) **dict**      (c) tuple      (d) set

**Q3:** Find unique sectors in your portfolio?

- (a) list      (b) dict      (c) tuple      (d) **set**

**Q4:** Store an (x, y) coordinate that should never change?

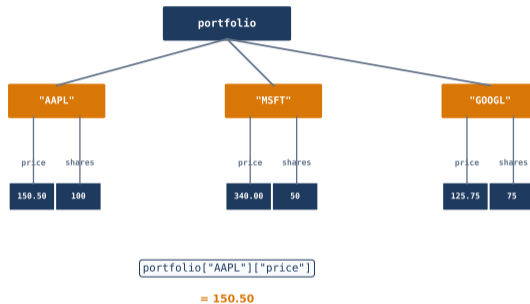
- (a) list      (b) dict      (c) **tuple**      (d) set

---

Choosing the right structure is a key skill – it affects performance and clarity

# Nested Data Structures

## Nested Dictionary Structure



**Real data is hierarchical:** portfolios contain stocks, stocks have attributes

---

Nested structures model complex financial data naturally

## Building Nested Structures

### Portfolio Structure:

```
portfolio = {  
  "AAPL": {"price": 185.5,  
          "shares": 100},  
  "MSFT": {"price": 340.0,  
          "shares": 50}  
}
```

### Accessing:

```
portfolio["AAPL"]["price"]  
→ 185.5
```

### List of Dicts:

```
trades = [  
  {"ticker": "AAPL",  
   "action": "BUY",  
   "shares": 50},  
  {"ticker": "MSFT",  
   "action": "SELL",  
   "shares": 30}  
]
```

### Accessing:

```
trades[0]["ticker"] → "AAPL"
```

---

Dict of dicts or list of dicts – choose based on primary access pattern

# Choosing the Right Structure

## Choosing: List vs Dictionary

### LIST

```
[1, 2, 3, 4]
```

Ordered sequence

Access by index

Duplicates OK

$O(n)$  search

*Use for: prices over time*

### DICTIONARY

```
{"a": 1, "b": 2}
```

Key-value pairs

Access by key

Unique keys

$O(1)$  lookup

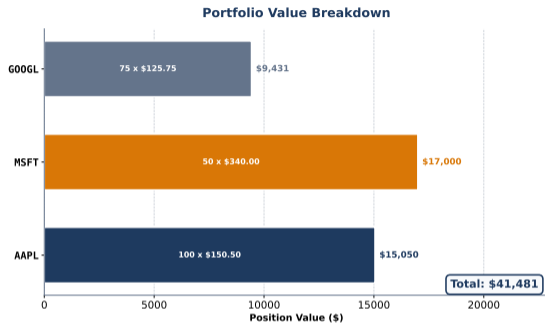
*Use for: ticker lookups*

**Lists for sequences, Dictionaries for mappings**

---

Lists for sequences, dicts for lookups, sets for uniqueness, tuples for immutability

## Finance: Portfolio Calculations



### Calculate Total Value:

```
portfolio = {  
    "AAPL": {"shares": 100,  
            "price": 185.5},  
    "MSFT": {"shares": 50,  
            "price": 340.0}  
}  
  
total = 0  
for t in portfolio:  
    s = portfolio[t]["shares"]  
    p = portfolio[t]["price"]  
    total += s * p  
  
# total = 35,550.00
```

---

Dictionaries are the natural structure for portfolio data

## Finance: Price History Analysis

### Price History as List:

```
prices = [150.0, 152.5, 148.0,
          155.0, 160.0]

# Basic statistics
avg = sum(prices) / len(prices)
# 153.10

high = max(prices) # 160.0
low = min(prices) # 148.0

# Price range
spread = high - low # 12.0
```

### Computing Returns:

```
# Daily returns
returns = []
for i in range(1, len(prices)):
    r = (prices[i] - prices[i-1])
        / prices[i-1]
    returns.append(r)

# [0.0167, -0.0295,
#  0.0473, 0.0323]
```

### Key pattern:

Returns need consecutive prices – lists preserve order.

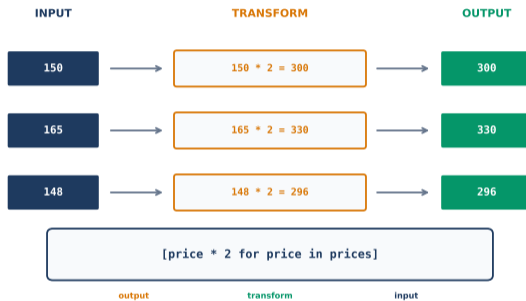
---

Lists maintain insertion order – essential for time series data

# List Comprehensions

A concise way to create lists from existing sequences

## List Comprehension: Transform Each Element



**Syntax:** `[expression for item in iterable if condition]`

---

Comprehensions are more Pythonic and often faster than loops

## Comprehension Patterns

### Transform:

```
prices = [150, 165, 148, 172]
# Double all prices
doubled = [p * 2 for p in prices]
# [300, 330, 296, 344]
# Convert to returns
base = prices[0]
rets = [(p - base) / base
        for p in prices]
```

### Traditional loop (equivalent but longer):

```
high = []
for p in prices:
    if p > 150:
        high.append(p)
```

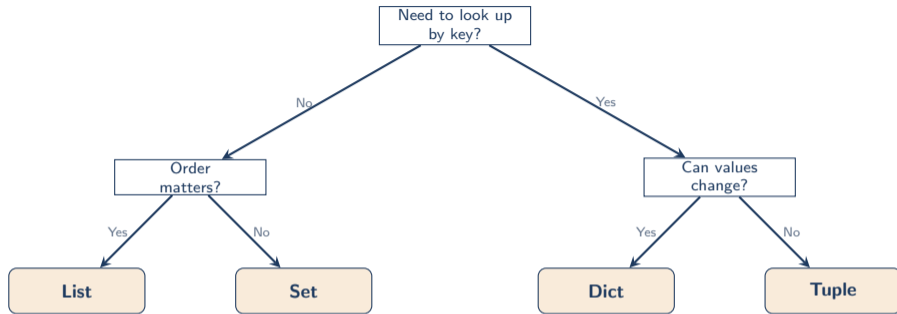
### Filter:

```
# Only prices above 150
high = [p for p in prices
        if p > 150]
# [165, 172]
# Uppercase tickers
tickers = ["aapl", "msft"]
upper = [t.upper()
         for t in tickers]
# ["AAPL", "MSFT"]
```

---

Use comprehensions when the logic fits one line; use loops for complex logic

# Data Structure Decision Guide



Structure	Ordered?	Mutable?	Duplicates?
List	Yes	Yes	Yes
Tuple	Yes	No	Yes
Dict	Yes (3.7+)	Yes	Keys: No
Set	No	Yes	No

Pick the simplest structure that meets your needs

## Hands-on Exercise (25 min)

### Build a portfolio tracker:

1. Create a list of tickers: `tickers = ["AAPL", "MSFT", "GOOGL", "AMZN"]`
2. Create a dict with shares owned: `shares = {"AAPL": 50, "MSFT": 30, ...}`
3. Create a dict with current prices
4. Calculate portfolio value using list comprehension: `values = [shares[t] * prices[t] for t in tickers]`
5. Find total portfolio value: `sum(values)`
6. Filter stocks worth more than \$5000

---

Save your work – we add control flow next lesson

## Organizing Data: Complete

### List

Ordered, mutable

prices, tickers

### Dict

Key-value pairs

portfolios, lookups

### Tuple

Immutable

coordinates, records

**Comprehensions:** `[x*2 for x in prices if x > 150]`

**Next: Make your code think and repeat with control flow**

---

You can now store and organize any data – next we process it

## Lesson Summary

### Key Takeaways:

- Lists store ordered sequences, accessed by index starting at 0
- Dictionaries map keys to values for instant lookups
- Tuples are immutable lists – use for data that shouldn't change
- Sets provide uniqueness and fast membership testing
- List comprehensions create lists concisely: `[expr for x in seq]`
- Choose structure based on access pattern and mutability needs

**Next Lesson:** Control Flow – if/else decisions and loops for repetition.

---

Data structures + control flow = the core of programming logic