

Lesson 02: Data Structures

Data Science with Python – BSc Course

Data Science Program

45 Minutes

After this lesson, you will be able to:

- Create and manipulate Python lists
- Access elements using indexing and slicing
- Build dictionaries for key-value data storage
- Apply list comprehensions for efficient data processing

Finance Application: Store portfolio holdings as lists and dictionaries.

Data structures are containers for organizing information

List Indexing in Python



Access Examples

prices[0]	→	150.0	# First element
prices[2]	→	148.25	# Third element
prices[-1]	→	169.75	# Last element
prices[-2]	→	172.0	# Second from end

Creating Lists:

```
prices = [185, 190, 188, 195]
```

Accessing Elements:

```
prices[0] → 185 (first)
```

```
prices[-1] → 195 (last)
```

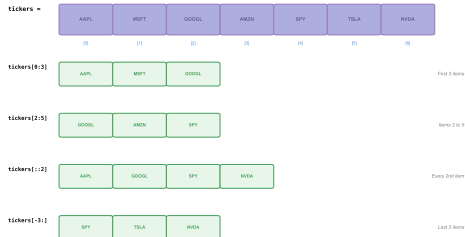
```
prices[1] → 190 (second)
```

Remember:

Python indexing starts at 0!

Negative indices count from the end: -1 is last element

List Slicing: [start:stop:step]



Slice Syntax: `list[start:end:step]`

`prices = [185, 190, 188, 195, 182]`

`prices[1:4] → [190, 188, 195]`

`prices[:3] → [185, 190, 188]`

`prices[2:] → [188, 195, 182]`

`prices[::2] → [185, 188, 182]`

Key: End index is exclusive

Slicing creates a new list – original unchanged

Dictionary Structure

Dictionary: Key-Value Pairs

KEY		VALUE
"AAPL"	:	150.5
"MSFT"	:	340.0
"GOOGL"	:	125.75
"AMZN"	:	165.0

Dictionary Operations

```
portfolio["AAPL"]           → 150.5      # Access value by key
portfolio["AAPL"] = 155.00   # Update value
portfolio["TSLA"] = 250.00   # Add new key-value pair
"MSFT" in portfolio         → True       # Check if key exists
```

Key-Value Pairs:

```
portfolio = {
    "AAPL": 50,
    "MSFT": 30,
    "GOOGL": 20
}
```

Access by Key:

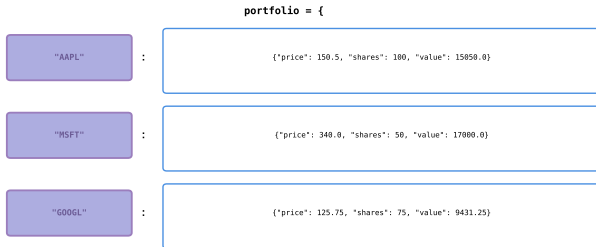
`portfolio["AAPL"]` → 50

`portfolio.keys()`

`portfolio.values()`

Dictionaries provide $O(1)$ lookup – very fast access

Nested Data Structures



Accessing Nested Data

<code>portfolio["AAPL"]</code>	→	<code>{ "price": 150.5, "shares": 100, ... }</code>	# Full nested dict
<code>portfolio["AAPL"]["price"]</code>	→	<code>150.5</code>	# Specific value
<code>portfolio["MSFT"]["shares"]</code>	→	<code>50</code>	# Shares for MSFT
<code>portfolio["GOOGL"]["value"]</code>	→	<code>9431.25</code>	# Total value

List Methods

Common List Methods

Method	Description	Example	Result
<code>append()</code>	Add item to end	<code>prices.append(175.88)</code>	<code>[158, 165, 175]</code>
<code>insert()</code>	Add item at position	<code>prices.insert(1, 168)</code>	<code>[158, 168, 165]</code>
<code>remove()</code>	Remove first occurrence	<code>prices.remove(165)</code>	<code>[158]</code>
<code>pop()</code>	Remove and return item	<code>prices.pop()</code>	Returns: 165
<code>sort()</code>	Sort list in place	<code>prices.sort()</code>	<code>[158, 165, 175]</code>
<code>reverse()</code>	Reverse list order	<code>prices.reverse()</code>	<code>[185, 158]</code>
<code>count()</code>	Count occurrences	<code>prices.count(158)</code>	1

Adding Elements:

```
prices.append(200)
prices.insert(0, 180)
```

Removing:

```
prices.remove(188)
prices.pop() – removes last
```

Sorting:

```
prices.sort()
prices.reverse()
```

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

Portfolio Representation: Dictionary

"AAPL"	shares: 100, buy: 145.0, current: 158.5
"MSFT"	shares: 50, buy: 320.0, current: 340.0
"GOOGL"	shares: 75, buy: 120.0, current: 125.75

```
portfolio = {
```

Portfolio Calculations

```
# Calculate total value
total_value = 0
for ticker in portfolio:
    shares = portfolio[ticker]["shares"]
    price = portfolio[ticker]["current_price"]
    total_value += shares * price

print(f"Total portfolio value: ${total_value:.2f}")
# Output: Total portfolio value: $31,481.25
```

Portfolio Dictionary:

```
prices = {
    "AAPL": 185.50,
    "MSFT": 378.20,
    "GOOGL": 141.80
}
```

Calculate Total:

```
total = sum(prices.values())
```

Check Existence:

```
"AAPL" in prices → True
```

Dictionaries are ideal for ticker-to-data mappings

List Comprehension

List Comprehension: Concise List Creation

Traditional Loop

```
prices = [150, 165, 140, 172]
doubled = []
for price in prices:
    doubled.append(price * 2)
```

Result: [300, 330, 280, 344]

More Concise

List Comprehension

```
prices = [150, 165, 140, 172]
doubled = [price * 2
            for price in prices]
```

Result: [300, 330, 280, 344]

List Comprehension Examples

Basic transformation:

```
[x * 2 for x in prices]
# Double all prices
```

With condition (filter):

```
[x for x in prices if x > 150]
# Only prices > 150
```

String manipulation:

```
[t.lower() for t in tickers]
# Lowercase all tickers
```

Math operations:

```
[x**2 for x in [1,2,3,4]]
# Squares: [1,4,9,16]
```

With if-else:

```
[x if x > 150 else 0 for x in prices]
# Set low prices to 0
```

Traditional Loop:

```
returns = []
for p in prices:
    returns.append(p * 1.05)
```

List Comprehension:

```
returns = [p * 1.05 for p in prices]
```

With Condition:

```
high = [p for p in prices if p > 190]
```

Comprehensions are more Pythonic and often faster

Choosing the Right Data Structure

Need to store data?

Ordered sequence?

Use LIST

Examples:

- Stock prices over time
- List of tickers

Key-value mapping?

Use DICTIONARY

Examples:

- Stock ticker → price
- Portfolio holdings

Feature Comparison		
List	Feature	Dictionary
Ordered	Order	Unordered
prices[0]	Access	portfolio["AAPL"]
O(n) search	Speed	O(1) lookup
Duplicates OK	Duplicates	Unique keys
Integer indices	Keys	Any immutable

Build a portfolio tracker:

- 1 Create a list of stock tickers:
`tickers = ["AAPL", "MSFT", "GOOGL", "AMZN"]`
- 2 Create a dictionary with shares owned:
`shares = {"AAPL": 50, "MSFT": 30, ...}`
- 3 Create a dictionary 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'll add more features next lesson

Key Takeaways:

- Lists store ordered sequences (accessed by index)
- Dictionaries store key-value pairs (accessed by key)
- Slicing extracts portions: `list[start:end]`
- List comprehensions create lists efficiently
- Choose structure based on access pattern

Next Lesson: Control Flow (if/else, loops)

Data structures + control flow = programming logic