Lesson 03: Control Flow

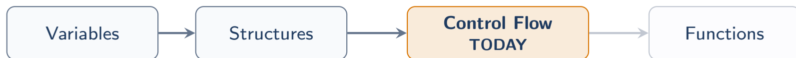
Data Science with Python – BSc Course

Data Science Program

BSc Course

45 Minutes

Previously: Data Structures



L02: You can store data in lists and dictionaries.

Problem: Your code runs top-to-bottom. What if you need it to **choose** or **repeat**?

Solution: Control flow – if/else for decisions, loops for repetition.

Control flow gives your code the ability to think and iterate

Learning Objectives

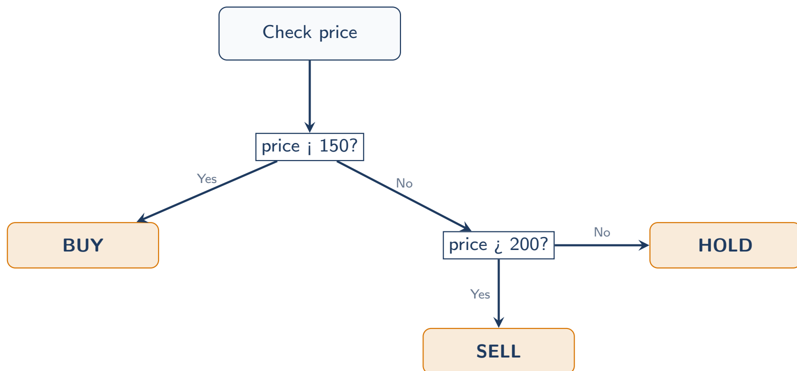
After this lesson, you will be able to:

- **Write** conditional statements with if/elif/else
- **Create** for loops to iterate over sequences
- **Implement** while loops for condition-based repetition
- **Control** loop execution with break and continue
- **Apply** patterns: accumulator, filter, search

Finance Application: Implement trading signals and position sizing rules.

Control flow determines which code executes based on conditions

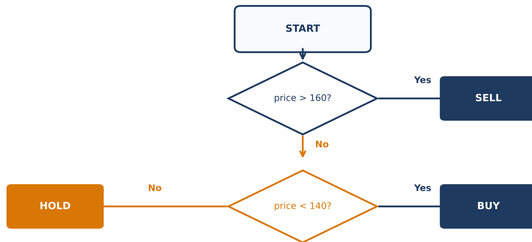
If-Else: Making Decisions



Key syntax: Colon after condition — 4-space indent — elif is optional

Indentation (4 spaces) defines code blocks in Python – not braces

If-Else Statement Structure



Pattern: `if` (required) → `elif` (optional, many) → `else` (optional, one)

Python evaluates conditions top-down – first True branch executes

Writing Conditional Logic

Simple If:

```
price = 145.00
if price < 150:
    print("Below threshold")
```

If-Else:

```
if price < 150:
    signal = "BUY"
else:
    signal = "HOLD"
```

If-Elif-Else:

```
if price < 150:
    signal = "BUY"
elif price > 200:
    signal = "SELL"
else:
    signal = "HOLD"
```

Common Mistake:

```
# Wrong:  = is assignment
if price = 150:  # Error!
# Correct:  == is comparison
if price == 150:  # OK!
```

Use == for comparison, = for assignment – never mix them

Nested Conditionals: Multi-Factor Decisions

Trading Rule with Volume:

```
price = 145.00
volume = 1200000
buy_th = 150.00
min_vol = 1000000
if price < buy_th:
    if volume > min_vol:
        action = "BUY"
    else:
        action = "WAIT"
elif price > 200:
    action = "SELL"
else:
    action = "HOLD"
```

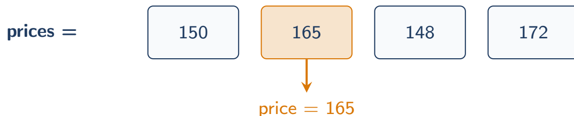
Equivalent with `and`:

```
if price < buy_th and
   volume > min_vol:
    action = "BUY"
elif price < buy_th:
    action = "WAIT"
elif price > 200:
    action = "SELL"
else:
    action = "HOLD"
```

Tip: Use `and`/`or` to avoid deep nesting. Flat is better than nested.

Trading rules are nested conditionals – keep them readable

For Loops: Iterating Over Sequences




```
for price in prices:
    print(price)
```

How it works: The variable `price` takes each value from the list, one at a time. The indented body runs once for each element.

For loops iterate a known number of times – once per element

For Loop Variations

Iterate Over List:

```
prices = [150, 165, 148, 172]
for price in prices:
    print(price)
# 150, 165, 148, 172
```

With Index (enumerate):

```
for i, p in enumerate(prices):
    print(f"Day {i}: {p}")
# Day 0: 150
# Day 1: 165 ...
```

Range Function:

`range(5)` → 0, 1, 2, 3, 4
`range(1, 5)` → 1, 2, 3, 4
`range(0, 10, 2)` → 0, 2, 4, 6, 8

Iterate Over Dict:

```
portfolio = {"AAPL": 100,
             "MSFT": 50}
for ticker in portfolio:
    print(ticker, portfolio[ticker])
```

enumerate() gives both index and value – very useful for data analysis

Loop Types Comparison

Loop Types: For vs While

FOR Loop

Syntax: `for x in items:`

Iterations: Known

Use case: Iterate over list

Increment: Automatic

```
for p in prices:
    total += p
```

WHILE Loop

Syntax: `while cond:`

Iterations: Unknown

Use case: Until condition

Increment: Manual

```
while price < 150:
    price *= 1.05
```

Choose FOR when you know iterations, WHILE when waiting for a condition

For: known iterations — **While:** unknown iterations, condition-based

Choose loop type based on whether iterations are known in advance

While Loops: Repeat Until Condition

Basic While:

```
balance = 10000
years = 0
while balance > 5000:
    balance *= 0.95  # -5%/yr
    years += 1
print(f"Broke in {years} years")
# Broke in 14 years
```

Warning: Ensure the condition eventually becomes False!

When to Use While:

- Unknown number of iterations
- Waiting for a condition
- Simulation until target reached
- User input validation

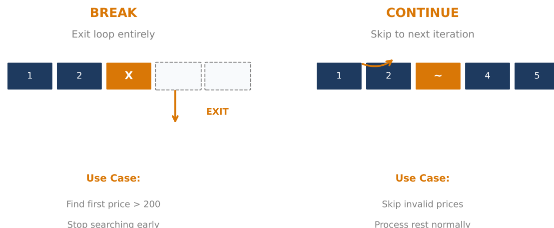
Infinite Loop (Danger):

```
# NEVER do this!
while True:
    print("stuck!")
# Fix:  add break condition
while True:
    if done:  break
```

While loops continue until condition becomes False – always ensure termination

Break and Continue

Control loop execution: exit early or skip iterations



break = stop the entire loop — **continue** = skip to next iteration

---

Break exits the loop entirely; continue skips the current iteration only

Break and Continue in Practice

Break: Find First Match

```
prices = [150, 165, 210, 148]
for price in prices:
    if price > 200:
        print(f"Alert: {price}")
        break
# Stops at 210, skips 148
```

Use case:
Stop searching once you find what you need.

Continue: Skip Invalid Data

```
prices = [150, -1, 165, 0, 172]
valid = []
for p in prices:
    if p <= 0:
        continue  # Skip bad data
    valid.append(p)
# valid = [150, 165, 172]
```

Use case:
Clean data by skipping invalid entries.

Break for early exit, continue for data cleaning – both very common

Checkpoint: Predict the Output

Q1: What does `break` do inside a loop?
(a) Skips one iteration (b) **Exits the loop** (c) Pauses execution

Q2: How many times does `for i in range(5)` run?
(a) 4 times (b) **5 times** (c) 6 times

Q3: What happens if a while condition never becomes False?
(a) Error (b) Returns None (c) **Infinite loop**

Q4: `for x in [1,2,3]: if x==2: continue` – what happens?
(a) Prints 1,3 (b) **Skips 2, processes 1 and 3** (c) Error

Understanding loop control is essential for writing efficient data processing code

Nested Loops: Multi-Dimensional Iteration

Multi-Stock Analysis:

```
tickers = ["AAPL", "MSFT"]
days = range(3)
for ticker in tickers:
    for day in days:
        print(f"{ticker} day {day}")
```

Output:

AAPL day 0, AAPL day 1, AAPL day 2,
MSFT day 0, MSFT day 1, MSFT day 2

Iteration Count:

Outer $\times$ Inner $=$ Total
2 tickers $\times$ 3 days $=$ 6 iterations

Performance Warning:

- $100 \times 100 = 10{,}000$
- $1000 \times 1000 = 1{,}000{,}000$
- Consider vectorization for large data (pandas, NumPy)

Rule of thumb:
Nested loops OK for small data.
Use pandas for large datasets.

Nested loops multiply iterations – watch performance on large datasets

List Comprehensions as Concise Loops

Loop Version:

```
prices = [150, 165, 148, 172]
returns = []
for i in range(1, len(prices)):
    r = (prices[i] - prices[i-1])
        / prices[i-1]
    returns.append(r)
```

4 lines of code.

Comprehension Version:

```
returns = [
    (prices[i] - prices[i-1])
    / prices[i-1]
    for i in range(1, len(prices))
]
```

Compact, Pythonic, often faster.

With Filter:

```
gains = [r for r in returns
         if r > 0]
```

Comprehensions combine for-loop and append into one expression

Essential Control Flow Patterns

Accumulator:

```
total = 0
for p in prices:
    total += p
```

```
# Sum all prices
```

Use: Sum, count, average

Filter:

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

```
# Select subset
```

Use: Screening stocks

Search:

```
for p in prices:
    if p > 200:
        print("Found!")
        break
```

```
# Find first match
```

Use: Find first trigger

Transformation:

```
returns = [(prices[i] - prices[i-1]) / prices[i-1] for i in range(1, len(prices))]
```

Master these 4 patterns and you can solve most data processing tasks

Finance: Screening Stocks with Conditions

Stock Screen:

```
stocks = {
  "AAPL":  {"price":  185, "pe":  28},
  "MSFT":  {"price":  340, "pe":  32},
  "INTC":  {"price":  45, "pe":  15},
  "NVDA":  {"price":  480, "pe":  60}
}
# Value stocks:  PE < 25
value = []
for t in stocks:
    if stocks[t]["pe"] < 25:
        value.append(t)
# value = ["INTC"]
```

Multi-Criteria Screen:

```
candidates = []
for t in stocks:
    s = stocks[t]
    if s["pe"] < 35 and
       s["price"] < 200:
        candidates.append(t)
# ["AAPL", "INTC"]
```

Real-world screening:

Hedge funds screen thousands of stocks using nested conditions and loops exactly like this.

Stock screening = filtering with conditions – the basis of quantitative investing

Finance: Portfolio Iteration

Calculate Position Values:

```
portfolio = {
  "AAPL": {"shares": 100,
           "price": 185.5},
  "MSFT": {"shares": 50,
           "price": 340.0}
}
total = 0
for ticker in portfolio:
    pos = portfolio[ticker]
    value = pos["shares"]
           * pos["price"]
    total += value
    print(f"{ticker}: ${value:,.0f}")
```

Output:

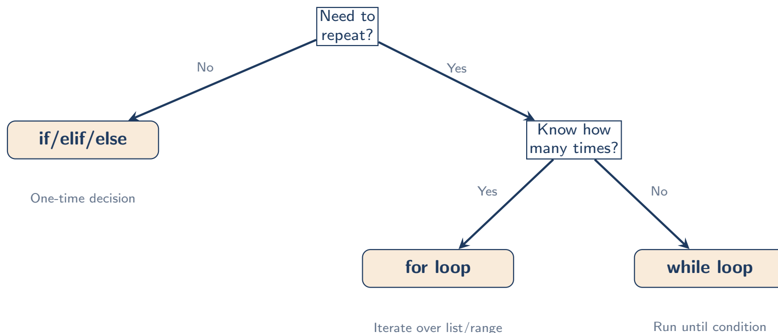
```
AAPL: $18,550
MSFT: $17,000
```

Find Largest Position:

```
max_val = 0
max_ticker = ""
for t in portfolio:
    v = portfolio[t]["shares"]
       * portfolio[t]["price"]
    if v > max_val:
        max_val = v
        max_ticker = t
# max_ticker = "AAPL"
```

Combining loops with conditionals = the core of portfolio analytics

Which Control Flow to Use?



Construct	Use When	Example
`if/else`	One-time choice	Buy or sell decision
`for`	Known iterations	Process each stock
`while`	Unknown iterations	Wait for price target

Most data analysis uses for loops – while loops are for simulations

Error Handling: try/except

Without Error Handling:

```
prices = [150, 0, 165]
for p in prices:
    ratio = 1000 / p  # Crashes!
# ZeroDivisionError at p=0
```

One bad value crashes the entire program.

With try/except:

```
for p in prices:
    try:
        ratio = 1000 / p
        print(f"{ratio:.2f}")
    except ZeroDivisionError:
        print("Skipping zero")
# 6.67
# Skipping zero
# 6.06
```

Key idea: Wrap risky code in `try`. If an error occurs, `except` handles it gracefully.

try/except prevents crashes when processing messy real-world data

Hands-on Exercise (25 min)

Implement a simple trading system:

1. Create price list: `prices = [180, 185, 195, 188, 205, 198]`
2. Implement trading rules:
   - If price ¿ 200: SELL
   - If price ¡ 185: BUY
   - Else: HOLD
3. Loop through prices and generate signals for each day
4. Count total BUY, SELL, HOLD signals (accumulator pattern)
5. Find first SELL trigger using `break`
6. **Bonus:** Calculate daily returns using a for loop

This exercise combines if/else, for loops, and all 4 patterns

Your Code Can Now Think and Repeat

if / elif / else	for loop	while loop
Decisions	Known iterations	Condition-based

break / continue	try / except
Loop control	Error handling

Next: Package logic into reusable functions

Variables + structures + control flow = you can solve real problems

Lesson Summary

Key Takeaways:

- `if/elif/else` for conditional execution (evaluated top-down)
- `for` loops iterate over sequences (lists, ranges, dicts)
- `while` loops repeat until a condition becomes False
- `break` exits a loop; `continue` skips one iteration
- Four patterns: accumulator, filter, search, transformation
- `try/except` prevents crashes from bad data

Next Lesson: Functions – encapsulate logic into reusable, testable blocks.

Control flow + functions = modular, maintainable trading systems