Python for Optimization in Finance
Master 2 – MMEF & IRFA
Instructor: Karl El Kallab

📘 Session 1 – Python Basics for Finance

🔷 Course Objectives

At the end of this session, you will be able to:

- Understand Python variables and data types.
- Perform operations and comparisons.
- Use conditions (`if/else`) and loops (`for/while`).
- Write simple functions.
- Manipulate lists, tuples, and dictionaries.
- Apply all concepts in **financial contexts**.

Python in Finance – Introduction

- **What is Python?**
  - High-level programming language, created by **Guido van Rossum (1991)**.
  - Simple, readable, powerful.
  - Used in **finance** for analysis, modeling, and automation.
- **Why Learn Python in Finance?**
  - Automate calculations: P&L, risk metrics.
  - Data analysis & visualization: **pandas, matplotlib**.
  - Quantitative modeling: **Monte Carlo, Black-Scholes**.
  - Trading & portfolio management: algorithmic strategies.
- **Key Features**
  - Interpreted → test ideas quickly.
  - High-level → focus on financial logic.
  - Libraries: **NumPy, pandas, SciPy, matplotlib, QuantLib**.
  - Cross-platform & free.

What Does "Interpreted" Mean?

- An **interpreted language** runs **line by line**.
- No need to **compile** the whole program first.
- **Immediate results** → easy to test and debug.
- Python is **interpreted**, so you can quickly try ideas.

Example:

python Copy code

```python
print("Hello, world!")
print(2 + 3)
```

Output:

Copy code

```
Hello, world!
5
```

- Each line runs **one at a time**, results appear instantly.

Tip:

- Great for **experimentation, finance calculations, and prototyping**.

🟢 Part 1 – Variables & Data Types

Variables: What are they?

- A **variable** is like a “box” that stores a value.
- Created with `=` (assignment).
- Can be reused in calculations.

Python Data Types

Type	Example	Description
`int`	`1000`	Whole numbers
`float`	`3.14`	Decimal numbers
`str`	`"Finance"`	Text
`bool`	`True`, `False`	Logical values

⚠️ Python does **not** have `double`. All decimals = `float`.

Example (Finance)

python

```python
capital = 10000          # int
rate = 0.05              # float
bank = "BNP Paribas"     # str
is_profitable = True     # bool

print(capital, rate, bank, is_profitable)
```

Result

graphql

```
10000 0.05 BNP Paribas True
```

Exercise

Define variables:

- Initial investment = `20000`
- Annual rate = `0.03`
- Bank name = `"Société Générale"`
- Profitability = `False`

  Print all variables.

Correction

python

Copy Edit

```python
investment = 20000
rate = 0.03
bank = "Société Générale"
is_profitable = False

print(investment, rate, bank, is_profitable)
```

Result

graphql

Copy Edit

```graphql
20000 0.03 Société Générale False
```

🟢 Part 2 – Operations

Arithmetic Operators

python

```python
a = 10
b = 3

print(a + b)   # addition
print(a - b)   # subtraction
print(a * b)   # multiplication
print(a / b)   # division
print(a // b)  # integer division
print(a % b)   # remainder
print(a ** b)  # exponentiation
```

```
13
7
30
3.3333333333333335
3
1
1000
```

Comparison Operators

python

```python
x = 5
y = 10

print(x > y)
print(x < y)
print(x == y)
print(x != y)
print(x >= 5)
```

```
False
True
False
True
True
```

Example (Finance)

python

Copy Edit

```python
capital = 10000
rate = 0.05
years = 3

future_value = capital * (1 + rate * years)
print("Future value:", future_value)
```

Result

yaml

Copy Edit

```yaml
Future value: 11500.0
```

Exercise (Finance)

- Initial capital = `20000`, rate = `0.03`, years = `5`.
- Compute **compound interest**:

$$FV = capital \times (1 + rate)^{years}$$

- Check if result > 23000.

Correction

python

Copy Edit

```python
capital = 20000
rate = 0.03
years = 5

future_value = capital * (1 + rate) ** years
print("Future value:", future_value)
print("Future value > 23000 ?", future_value > 23000)
```

Result

yaml

Copy Edit

```yaml
Future value: 23185.44
Future value > 23000 ? True
```

🟢 Part 3 – Conditions & Loops

If Statement (Condition)

- Executes a block of code **only if a condition is True**.

Example

python Copy Edit

```python
profit = 200

if profit > 0:
    print("You made a profit!")
```

Result

css Copy Edit

```
You made a profit!
```

If / Else

- Adds an **alternative** if the condition is False.

Example

python Copy Edit

```python
profit = -50

if profit > 0:
    print("Profit")
else:
    print("Loss")
```

Result

nginx Copy Edit

```
Loss
```

If / Elif / Else

- Handles **multiple conditions**.

Example

python Copy Edit

```python
profit = 0

if profit > 0:
    print("Profit")
elif profit < 0:
    print("Loss")
else:
    print("Break-even")
```

Result

mathematica Copy Edit

```mathematica
Break-even
```

What is a Loop?

- A **loop** repeats a block of code multiple times automatically.
- Useful to avoid rewriting the same code repeatedly.
- Common in finance for simulating growth, iterating over returns, or processing datasets.

Two main types of loops in Python:

1. **For Loop** – iterates over each element in a sequence (list, string, dictionary).
2. **While Loop** – repeats a block **as long as a condition is true**.

Comparison: For vs While

Feature	For Loop	While Loop
Usage	Iterate over a known sequence	Repeat until condition is false
Stopping	Ends when all elements are processed	Ends when condition becomes false
Best for	Fixed number of iterations	Unknown number of iterations
Finance use	Go through a list of daily returns	Grow capital until a target is reached

For Loop Example (Finance)

python

Copy Edit

```python
returns = [0.01, -0.02, 0.03]

for r in returns:
    print("Return:", r)
```

Result

makefile

Copy Edit

```makefile
Return: 0.01
Return: -0.02
Return: 0.03
```

While Loop Example (Finance)

python

Copy Edit

```python
capital = 1000
rate = 0.1
year = 0

while capital < 2000:
    capital *= (1 + rate)
    year += 1

print("It takes", year, "years to double the capital.")
```

Result

vbnet

Copy Edit

```vbnet
It takes 8 years to double the capital.
```

Exercise – Loops in Finance

1. Returns = `[0.02, -0.01, 0.03, -0.02, 0.04]`
    - Count positive and negative days with a **for loop**.
2. Capital = `1000`, rate = `0.05`
    - Use a **while loop** to simulate growth until capital ≥ `1500`.

Correction

python

Copy Edit

```python
returns = [0.02, -0.01, 0.03, -0.02, 0.04]

gains = 0
losses = 0
for r in returns:
    if r > 0:
        gains += 1
    else:
        losses += 1

print("Positive days:", gains)
print("Negative days:", losses)

capital = 1000
rate = 0.05
years = 0
while capital < 1500:
    capital *= (1 + rate)
    years += 1

print("Years needed:", years)
```

Result

yaml

```yaml
Positive days: 3
Negative days: 2
Years needed: 9
```

🟢 Part 4 – Functions

What is a Function?

- A **function** is a reusable block of code.
- Defined using `def`.
- Can take **parameters** (inputs) and **return outputs**.

Example Function (Finance)

python Copy Edit

```python
def future_value(capital, rate, years):
    return capital * (1 + rate) ** years

print(future_value(10000, 0.05, 3))
```

Result

Copy Edit

```
11576.25
```

🟢 Built-in Functions in Python

What are Built-in Functions?

- Predefined functions in Python that perform common tasks.
- Save time and simplify code.
- Commonly used in finance for calculations, loops, and data analysis.

---

sum() – Sum of List Elements

- Adds all numbers in a list.

Example

python Copy Edit

```python
returns = [0.02, 0.03, -0.01, 0.04]
total_return = sum(returns)
print("Total return:", total_return)
```

Result

kotlin Copy Edit

```
Total return: 0.08
```

---

len() – Number of Elements

- Counts the number of elements in a list, tuple, string, etc.

Example

python

Copy Edit

```python
assets = ["AAPL", "TSLA", "GOOG"]
num_assets = len(assets)
print("Number of assets:", num_assets)
```

Result

javascript

Copy Edit

```
Number of assets: 3
```

---

range() – Generate a Sequence

- Generates a sequence of numbers for loops.

Example

python Copy Edit

```python
for year in range(1, 6):
    print("Year:", year)
```

Result

makefile Copy Edit

```makefile
Year: 1
Year: 2
Year: 3
Year: 4
Year: 5
```

---

min() and max() – Find Extremes

- `min()` → smallest value, `max()` → largest value.

Example

python

Copy Edit

```python
returns = [0.02, -0.01, 0.03, -0.02, 0.04]
print("Min return:", min(returns))
print("Max return:", max(returns))
```

Result

kotlin

Copy Edit

```kotlin
Min return: -0.02
Max return: 0.04
```

round() – Round Numbers

- Rounds a number to the specified number of decimals.

Example

python Copy Edit

```python
value = 1234.5678
print("Rounded to 2 decimals:", round(value, 2))
```

Result

yaml Copy Edit

```yaml
Rounded to 2 decimals: 1234.57
```

Python Data Structures: Lists, Tuples, Dictionaries

Lists – Explanation

- **Definition**: A **list** is an ordered, mutable (modifiable) collection of items.
- **Characteristics**:
  - Allows **duplicates**
  - **Indexable** (can access elements with `[ ]`)
  - Elements can be of **different data types**
  - Supports many built-in methods like `append()`, `remove()`, `sort()`

👉 Lists are widely used for storing **sequences of data** such as prices, returns, or asset names.

List – Example Code

```python
# Creating a list of stock prices
prices = [100, 105, 102, 108]

# Accessing elements
print(prices[0])   # first element
print(prices[-1])  # last element

# Modifying a list
prices.append(110)
prices.remove(102)

# Iterating over a list
for p in prices:
    print(p)
```

List – Result

```csharp
100
108
[100, 105, 108, 110]
```

Tuples – Explanation

- **Definition**: A **tuple** is an ordered, immutable collection of items.
- **Characteristics**:
  - Cannot be modified after creation
  - Faster than lists (performance advantage)
  - Useful for **fixed data** (e.g., coordinates, parameter sets, dates)
  - Syntax uses **parentheses ( )**

👉 Example in finance: storing a pair `(ticker, price)` that should not change.

Tuple – Example Code

python

Copy code

```python
# Creating a tuple
stock_info = ("AAPL", 175)

# Accessing elements
print(stock_info[0])  # ticker
print(stock_info[1])  # price

# Tuples cannot be changed
# stock_info[1] = 180  # ❌ ERROR
```

Tuple – Result

php

Copy code

```
AAPL
175
TypeError: 'tuple' object does not support item assignment
```

Dictionaries – Explanation

- **Definition**: A **dictionary** is a collection of **key-value pairs**.
- **Characteristics**:
  - Unordered (in Python 3.6+, ordered by insertion)
  - Access by **key** instead of index
  - Keys must be unique
  - Very efficient for **mapping data** (like ticker → price, country → currency)
  - Syntax uses **curly braces { }**

👉 In finance: useful for mapping **stock tickers to their last prices**.

Dictionary – Example Code

python

```python
# Creating a dictionary of stock prices
stock_prices = {
    "AAPL": 175,
    "MSFT": 310,
    "TSLA": 250
}

# Accessing values
print(stock_prices["AAPL"])

# Modifying dictionary
stock_prices["TSLA"] = 260
stock_prices["GOOG"] = 2800  # add new key-value

# Iterating over dictionary
for ticker, price in stock_prices.items():
    print(ticker, price)
```

Dictionary – Result

yaml

```yaml
175
AAPL 175
MSFT 310
TSLA 260
GOOG 2800
```

Comparison Table

Feature	List	Tuple	Dictionary
Syntax	`[ ]`	`( )`	`{ key: value }`
Order	Ordered	Ordered	Ordered (insertion order)
Mutable?	✅ Yes	❌ No	✅ Yes
Duplicates?	✅ Yes	✅ Yes	❌ Keys must be unique
Access by	Index (`[0]`)	Index (`[0]`)	Key (`dict["AAPL"]`)
Use Case	Price series, lists	Fixed sets, configs	Mapping tickers to prices, metadata

Exercise – Collections

- Prices = `[150, 152, 155]`
- Trade = `(TSLA, 50, 800.5)`
- Portfolio = `{"AAPL": 10000, "TSLA": 5000, "GOOG": 12000}`

Find:

1. Price of Day 2
2. Asset name in the trade
3. Total portfolio value

Correction

python

Copy Edit

```python
prices = [150, 152, 155]
trade = ("TSLA", 50, 800.5)
portfolio = {"AAPL": 10000, "TSLA": 5000, "GOOG": 12000}

print("Day 2 price:", prices[1])
print("Trade asset:", trade[0])
print("Portfolio total:", sum(portfolio.values()))
```

Result

yaml

Copy Edit

```yaml
Day 2 price: 152
Trade asset: TSLA
Portfolio total: 27000
```

🟢 Final Recap Challenge

Exercise – Full Challenge

Investor starts with `capital = 10000`.

Annual returns for 10 years: `[0.05, 0.02, -0.01, 0.04, 0.03, 0.06, -0.02, 0.05, 0.01, 0.07]`

Tasks:

1. Store returns in a list
2. Update capital year by year using a loop
3. Store each year's capital in a list
4. Create a dictionary mapping `year → capital`
5. Write a function `final_value(returns, capital)` to compute the final amount

Correction

python Copy Edit

```python
returns = [0.05, 0.02, -0.01, 0.04, 0.03, 0.06, -0.02, 0.05, 0.01, 0.07]

capital = 10000
capitals = []

for r in returns:
    capital *= (1 + r)
    capitals.append(capital)

portfolio_history = {year+1: capitals[year] for year in range(len(capitals))}

def final_value(returns, capital=10000):
    for r in returns:
        capital *= (1 + r)
    return capital

print("Yearly capitals:", capitals)
print("Portfolio history:", portfolio_history)
print("Final value:", final_value(returns))
```

Result

yaml

Copy Edit

```yaml
Yearly capitals: [10500.0, 10710.0, 10602.9, 11027.016, 11357.82648,
12039.2960688, 11798.510147424, 12388.4356547952, 12512.319011343152, 13388.18134
Portfolio history: {1: 10500.0, 2: 10710.0, 3: 10602.9, ..., 10: 13388.18}
Final value: 13388.18
```

Final Complex Exercise – Finance Simulation

Scenario:

An investor manages a **portfolio of 3 assets** over **5 years**.

- Initial capital per asset: `AAPL = 10000`, `TSLA = 8000`, `GOOG = 12000`
- Annual returns for each asset (as lists):

```python
AAPL_returns = [0.05, 0.02, -0.01, 0.04, 0.03]
TSLA_returns = [0.10, -0.05, 0.02, 0.06, 0.08]
GOOG_returns = [0.03, 0.04, 0.01, 0.05, 0.02]
```

Tasks:

1. Store all asset returns in a **dictionary**: `{asset_name: returns_list}`.
2. Write a **function** `simulate_growth(capital, returns)` that computes the yearly capital for an asset and returns a **list of yearly capitals**.
3. Using a **loop**, simulate the growth of each asset. Store the **final capital per asset in a dictionary**.
4. Calculate the **total portfolio value after 5 years**.
5. Determine **which asset performed the best** and **worst**.
6. Use built-in functions like `sum()`, `max()`, `min()`, and `len()` where appropriate.
7. Round all final capital values to **2 decimals**.

```python
# Initial capitals
portfolio = {"AAPL": 10000, "TSLA": 8000, "GOOG": 12000}

# Returns per asset
returns_dict = {
    "AAPL": [0.05, 0.02, -0.01, 0.04, 0.03],
    "TSLA": [0.10, -0.05, 0.02, 0.06, 0.08],
    "GOOG": [0.03, 0.04, 0.01, 0.05, 0.02]
}

# Function to simulate growth
def simulate_growth(capital, returns):
    capitals = []
    for r in returns:
        capital *= (1 + r)
        capitals.append(round(capital, 2))
    return capitals

# Simulate each asset
final_capitals = {}
portfolio_history = {}
```

```
for asset, initial_capital in portfolio.items():
    yearly_capitals = simulate_growth(initial_capital, returns_dict[asset])
    portfolio_history[asset] = yearly_capitals
    final_capitals[asset] = yearly_capitals[-1]

# Total portfolio value
total_portfolio = sum(final_capitals.values())

# Best and worst performing asset
best_asset = max(final_capitals, key=final_capitals.get)
worst_asset = min(final_capitals, key=final_capitals.get)

# Print results
print("Yearly Capitals per Asset:", portfolio_history)
print("Final Capitals:", final_capitals)
print("Total Portfolio Value:", total_portfolio)
print("Best Performing Asset:", best_asset)
print("Worst Performing Asset:", worst_asset)
```

Example Result

yaml

Copy Edit

```
Yearly Capitals per Asset:
{
 'AAPL': [10500.0, 10710.0, 10602.9, 11026.02, 11356.8],
 'TSLA': [8800.0, 8360.0, 8527.2, 9038.83, 9751.75],
 'GOOG': [12360.0, 12834.4, 12962.74, 13610.87, 13883.09]
}
Final Capitals: {'AAPL': 11356.8, 'TSLA': 9751.75, 'GOOG': 13883.09}
Total Portfolio Value: 34991.64
Best Performing Asset: GOOG
Worst Performing Asset: TSLA
```