AF3214 Final Exam Revision Checklist (Marked based on FINAL Revised Summaries & Extractions)

Exam Format & Rules: (Not covered in content summaries/extractions)

- **Part One:** Multiple Choices (10 Qs, 10 pts)
- **Part Two:** True/False (10 Qs, 10 pts - *Must explain 'False'*)
- **Part Three:** Short Answer Questions (5 Qs, 80 pts - *Coding: Only key blocks needed*)
- **Conditions:** Open Book, Open Computer, Open Internet
- **CRITICAL RULE: NO** Web-based Generative AI (GenAI) tools allowed.
- **CRITICAL RULE:** No phones, iPads, smart watches, earphones, P2P comms, E-mails.
- **CRITICAL RULE:** No discussion or sharing info about the exam.
- **Requirement:** Bring Student ID or HKID.
- **Requirement:** Ensure well-functioning laptop & power adaptor.

Content Scope (Weeks 1-12 Slides & Jupyter Notebooks):

Weeks 1-4: Python & Pandas Fundamentals

- [✓] **Python Basics:**
  - [✓] Arithmetic & Logical Operators
  - [✓] Data Types (int, float, str, bool, list, dict) & Type Conversion
  - [✓] Variables (Local vs. Global Scope, Naming Rules)
  - [✓] String Manipulation (Creation, Indexing, Slicing, Methods, Immutability, Comparison, advanced slicing)
  - [✓] Functions (Definition, Calling, Parameters/Arguments, Return, Scope, Docstrings, Syntax precision)
  - [✓] Lists & Dictionaries (Creation, Indexing/Access, Methods (pop, remove, del, append vs extend), Looping, Mutability, Aliasing vs. Cloning, List Operators, Dict creation syntax, deletion, Insertion Order (Dict 3.7+))
  - [✓] Error Handling (`try/except`, specific errors, else, finally, raising explicitly concept)
  - [✓] Loops (`for`, `while`, `break`/`continue`, accumulator pattern, `range()`)
  - [✓] Conditionals (`if/elif/else`, Indentation consistency)
  - [✓] Importing Packages/Modules (Syntax, Access, Namespaces, avoiding `import *`)
  - [✓] File I/O (Reading/Writing text files, `with open`, modes, methods (`.read`, `.readlines`, iterate), line processing loop (`.strip`, `.split`), relative vs absolute paths concept)
- [✓] **Pandas:**
  - [✓] Series & DataFrames (Creation, Fundamentals, Index (default/explicit), Values (`.values` attribute), Attributes (`.index`, `.columns`, `.shape`, `.size`, `.T`), NumPy base, Series data types (object dtype))
  - [✓] Data Indexing & Selection (`.loc`, `.iloc`, boolean masking, label vs position, slice endpoints, syntax (brackets), assignment via indexers, fancy indexing, View vs Copy distinction)
  - [✓] Grouping (`.groupby()`, Split-Apply-Combine, Syntax, Aggregation (`.mean`, `.sum`, etc., `.describe`), Multi-column/Aggregation (`.agg`), Index result (MultiIndex), `.idxmax()`, `.size()` vs `.count()`, aggregating ranges, accessing described results)
  - [✓] Concatenating (`pd.concat`, axis, `ignore_index`, index result precision) & Merging (`pd.merge`, keys (`on`, `left_on`/`right_on`), `how`, index merge (`left_index`/`right_index`))
  - [✓] Reading/Writing Files (CSV (`pd.read_csv`), Excel (`pd.read_excel`)) from/into DataFrames (incl. args: `delimiter`, `usecols`, `sheet_name`, `index` for `to_csv`)
  - [✓] Basic Plotting from DataFrames (`.plot()`, Seaborn mentioned, aggregation need, plot interpretation)
- [✓] **Statistics (Intro):** Covered via Pandas methods (`.mean`, `.describe`, `.median`, `.sum`, `.min`, `.max`, `.count`, `.size`, `.unique`, `.nunique`, `.info`, `.shape`)
- [✓] **Note:** Database knowledge is *not required*. (Acknowledged)

Week 6: Data Handling, Stats & Web Interaction

- [✓] **Scientific Computing Concepts:** Reproducibility & Replication (Validity distinction)
- [✓] **Data Types in Finance:** Cross-sectional, Time series, Panel/Longitudinal
- [✓] **Data Classification:** Quantitative vs. Qualitative (Nominal, Ordinal subtypes) (Raw text as unstructured)
- [✓] **Financial Returns:**
  - [✓] Log Returns (Calculation, Additivity benefit)
  - [✓] Simple Returns (Calculation, Normalization benefit)
- [✓] **Descriptive Statistics (Concepts & Calculation):**
  - [✓] Mean, Median, Mode (Central tendency)

    - [✓] Variance, Standard Deviation (Volatility) (Dispersion)
    - [✓] Correlation Matrices
- [✓] **HTTP Concepts:**
    - [✓] Standard protocol purpose (Client-Server Model, WWW vs Internet, TCP/IP role)
    - [✓] HTTP Request (Request Line, Headers (Host, User-Agent, Accept), Body, Method/URL/Version)
    - [✓] Request Methods (GET vs. POST - contrast, HTTPS)
    - [✓] HTTP Response (Status Line, Headers (Content-Type/Length), Body, Version/Code/Reason)
    - [✓] Status Codes (Categories: 2xx, 4xx, 5xx, specific checks, DELETE success code, state management via Cookies, Host header purpose, User-Agent role)
- [✓] **Python Web Tools (Conceptual Understanding & Basic Usage):**
    - [✓] `requests` library (import, get/post, headers/params, status_code, content access (`.text`, `.content`, `.json`), error handling (specific exceptions), rate limiting (`time.sleep`), `.json()` method vs `json.loads`, `data`/`json` args for POST, API key passing)
    - [✓] `BeautifulSoup` library (Parsing HTML (`html.parser`), Navigating (tags), `find()`/`find_all()` (by tag/attribute, `class_`), Extracting text/attributes (`.text`, `.get_text`, `['attr']`, `.get()`), handling `None` result, nested search logic, tag vs content)
    - [✓] API (Definition, Reliability vs Scraping, JSON/XML format, Keys, Rate Limiting, Wrappers)

Week 7: Market Microstructure (Take Home Lecture Notes)

- [✓] **Assets:** Stocks, Bonds, Options, Forex, ETF, Cryptocurrency (Definitions provided)
- [✓] **Market Players:** Exchanges, Broker/dealer, Buy-side/Sell-side
- [✓] **Execution Methods:**
    - [✓] Algorithm (Definition, Generations: TWAP/VWAP, TCA, Liquidity Search/ECN; Gen 1 weakness, Info Leakage risk)
    - [✓] Direct Market Access (DMA) (Definition, Broker infrastructure, Pre-trade Risk controls)
    - [✓] Sponsored Access (Definition, Client infrastructure via Broker ID, Pre-trade Risk controls, HFT use case, Naked Access risk)
- [✓] **Markets:** Primary vs. Secondary
- [✓] **Liquidity:** Depth, Tightness (Bid-Ask Spread), Resilience
- [✓] **Market Types:** Quote-driven vs. Order-driven (Trading Mechanisms)
- [✓] **Trading Frequency:** Continuous, Periodic, Request-driven (Definitions provided)
- [✓] **Order Types:** Market Order vs. Limit Order (Definitions)
- [✓] **Fill Instructions:** Immediate-or-Cancel (IOC), Fill or Kill (FOK), All-or-None (AON) (Definitions provided in supplementary content)
- [✓] **Order Concepts:** Preference & Directed Orders (Definitions provided in supplementary content)
- [✓] **Routing Instructions:** Do-not-route, Directed-routing, Intermarket sweep (Definitions provided in supplementary content)
- [✓] **Transaction Costs:** (Covered via TCA discussion - Timing risk, Opportunity cost, Market Impact; Investment-related mentioned conceptually)
- [✓] **Trade Analysis:** Pre-trade vs. Post-trade Analysis (Pre-trade mentioned with SA risk; Post-trade implied via TCA)

Weeks 7-9: Risk, Return & Portfolio Management

- [✓] **Risk & Return Basics:**
    - [✓] Realized Return (Definition)
    - [✓] Expected Return (Definition, Weighted Average, Historical Proxy, Calculation, Portfolio calc)
    - [✓] Excess Return (Definition, Formula)
    - [✓] Risk Premium (Definition, Formula, Market Risk Premium)
    - [✓] Mean, Variance, Standard Deviation (as measures, calculation (Pandas/Numpy), interpretation, formulas (population/sample), annualized volatility)
    - [✓] Correlation (as measure, calculation (`.corr()`), interpretation (-1 to +1, strength/direction), formula, role in diversification, Covariance vs Corr definition)
    - [✓] Difference between Variance & Standard Deviation (Interpretation, units, weighting outliers)
- [✓] **Market Properties:** Efficient Markets (Concept, Properties)
- [✓] **Risk Categories:** Systematic vs. Unsystematic Risk (Definitions, Market/Undiversifiable vs Diversifiable/Firm-Specific, relation to diversification/hedging)
- [✓] **Common Risk Measures:** Standard Deviation, Alpha (α), Beta (β) (Definitions, Formulas, Calculation via Regression (`statsmodels`), Interpretation)
- [✓] **Concept:** Risk-Return Tradeoff (Definition, Application)
- [✓] **Portfolio Basics:**
    - [✓] Value of portfolio (Definition)
    - [✓] Weights calculation (Definition, Sum=1, Negative/Leverage/Dollar-neutral, Interpretation)
    - [✓] Types of portfolio (Conceptual: Income, Growth, Value)
    - [✓] Portfolio Returns (Weighted average formula, matrix notation)

- [✓] **Matrices:** Covariance Matrix & Formula (`.cov()`), Correlation Matrix & Formula (`.corr()`) (Concept, Calculation, role in portfolio risk, annualization, extracting specific values)
- [✓] **Modern Portfolio Theory (MPT):**
    - [✓] Expected return of portfolio (Calculation, matrix (`np.dot`))
    - [✓] Variance and Volatility of portfolio (Formula application - matrix form (`np.dot`) crucial, role of covariance, *not* simple weighted average, non-linear graph, annualization) (**Critical input error noted in quiz**)
    - [✓] Correlation of portfolio (Role in MPT, impact on risk, varying correlation effects)
    - [✓] Efficient Frontier (Concept, Minimum-Variance Boundary, Shape, Tangency Portfolio, Straight line with risk-free asset, graphical representation)
    - [✓] Sharpe Ratio (Formula, Interpretation, Maximization via Tangency Portfolio, relation to slope) (Advanced concepts: constraints, limitations, optimization formulation)

Week 10: Machine Learning Concepts

- [✓] **Fundamental Concepts:**
    - [✓] Difference between AI, Machine Learning, Deep Learning (Hierarchy)
    - [✓] Unstructured vs. Structured Data (Definitions, examples)
    - [✓] Supervised (Classification, Regression) vs. Unsupervised (Clustering, Dimensionality Reduction) Learning (Definitions, examples, when to use, paradigm)
    - [✓] Features (Definition, input variables, engineering, SNR)
    - [✓] Overfitting (Definition, relation to training data/noise, poor generalization, ethical issues, avoiding via simpler models/tradeoffs)
- [✓] **Evaluation:**
    - [✓] Confusion Matrices (TP, FP, TN, FN - Definitions, calculation from examples)
    - [✓] Metrics: Sensitivity (Recall), Specificity, Positive Predictive Value (PPV)/Precision (Definitions/Formulas, Interpretation, Trade-offs, goal alignment)
- [✓] **Note:** Machine Learning Python demo is *not required*. (Acknowledged)

Weeks 11-12: Textual Analysis & LLM Concepts

- [✓] **Textual Analysis:**
    - [✓] Definition of Textual Analysis (vs. NLP, Goals: insights from text itself, simple examples)
    - [✓] Definition of NLP (Natural Language Processing) (Goals: understand linguistic use/context, diverse field examples)
    - [✓] Differences between Textual Analysis and NLP (Focus on semantics/structure in NLP)
    - [✓] Text Data Readability (Concept, Gunning Fog Index measure, calculation using `textstat`, processing PDFs (`PyPDF2`, `pdfminer.six`) for text)
- [✓] **Large Language Models (LLM):**
    - [✓] Definition of LLM (Scale: Parameters, Data; relation to GenAI, specific examples like Llama-2)
    - [✓] Parameters (Concept: weights storing knowledge, scale, how obtained via training, conceptual explanation)
    - [✓] Neural Network (Basic Concept: structure (layers, nodes), function (next word prediction), parameters dispersed, training linked to compression, Transformer architecture mentioned)
    - [✓] Steps to train a GenAI (Conceptual Overview: Pre-training/Base Model -> Fine-tuning/Assistant Model, data needs, compute needs)

Weeks 1-4: Python & Pandas Fundamentals

- **[✓] Python Basics:**
    - **[✓] Arithmetic & Logical Operators**
        - **Content:** Reviewed arithmetic operators for calculations (`+`, `-`, `*`, `/` division, `//` floor division, `%` modulus, `**` exponentiation), comparison operators for evaluating conditions (`>`, `<`, `>=`, `<=`, `==` equality, `!=` inequality), and logical operators for combining boolean expressions (`and`, `or`, `not`). Highlighted that integer and float equality (e.g., `10 == 10.0`) evaluates to True.
        - **Python Code/Syntax:**
            - Arithmetic: `+`, `-`, `*`, `/`, `//`, `%`, `**`
            - Comparison: `>`, `<`, `>=`, `<=`, `==`, `!=`
            - Logical: `and`, `or`, `not`
            - Example: `10 == 10.0` evaluates to `True`
    - **[✓] Data Types (int, float, str, bool, list, dict) & Type Conversion**

- **Content:** Covered fundamental data types: integers (`int`), floating-point numbers (`float`), strings (`str`), and booleans (`bool`). Also introduced collections: ordered, mutable lists (`list`), and key-value pair dictionaries (`dict`). Learned how to check the type of a variable using the `type()` function and practiced converting between basic types using `int()`, `float()`, and `str()`.
- **Python Code/Syntax:**
  - Types: `int`, `float`, `str`, `bool`, `list`, `dict`
  - Checking Type: `type(variable)`
  - Conversion: `int(value)`, `float(value)`, `str(value)`

- **[✓] Variables (Local vs. Global Scope, Naming Rules)**

  - **Content:** Covered variable assignment using `=`. Reviewed naming rules: cannot start with a number, no spaces, not a reserved keyword, starting with `_` is allowed. Explained variable scope: `local` (defined inside a function) vs. `global`. Function calls create local scopes. Assignment inside a function creates a local variable unless the `global` keyword is used. Highlighted the importance of tracking scope when predicting output.
  - **Python Code/Syntax:**
    - Assignment: `variable_name = value`
    - Scope Keyword: `global variable_name` (used inside function to modify global variable)
    - Naming Rules: (Descriptive rules provided in content)

- **[✓] String Manipulation (Creation, Indexing, Slicing, Methods, Immutability, Comparison, advanced slicing)**

  - **Content:** Covered string creation, use of special characters (`\\n`, `\\t`), and operators (`+` for concatenation, `*` for repetition). Explained indexing (positive/negative) and slicing (`[start:stop:step]`) including negative steps and reversing. Reviewed common methods like `len()`, `.upper()`, `.lower()`, `.strip()`, `.split()`, `.replace()`. Emphasized string immutability (content cannot be changed in-place, variable can be reassigned). Noted that string comparison (`>`, `<`) is case-sensitive ('Z' < 'a'). Practiced advanced slicing with negative indices/steps and ensuring method outputs (like `.replace()`) are handled correctly.
  - **Python Code/Syntax:**
    - Creation: `my_string = "hello"` or `'hello'`
    - Special Chars: `\\n` (newline), `\\t` (tab)
    - Operators: `+`, `*`
    - Indexing: `my_string[index]`
    - Slicing: `my_string[start:stop:step]`
    - Methods: `len(my_string)`, `my_string.upper()`, `my_string.lower()`, `my_string.strip()`, `my_string.split(delimiter)`, `my_string.replace(old, new)`
    - Comparison: `>`, `<` (case-sensitive)

- **[✓] Functions (Definition, Calling, Parameters/Arguments, Return, Scope, Docstrings, Syntax precision)**

  - **Content:** Covered defining functions using `def`, calling them, and understanding the difference between parameters (in definition) and arguments (in call). Explained using `return` to send values back (default is `None`). Discussed variable scope (local/global) in function context. Introduced docstrings (`"""Docstring"""`) for explanation. Highlighted the need for precise call syntax (correct name, matching parentheses) and paying attention to scope for output prediction.
  - **Python Code/Syntax:**
    - Definition: `def function_name(parameter1, parameter2): ... return value`
    - Calling: `function_name(argument1, argument2)`
    - Return: `return value`
    - Docstring: `"""This function does..."""`

- **[✓] Lists & Dictionaries (Creation, Indexing/Access, Methods (pop, remove, del, append vs extend), Looping, Mutability, Aliasing vs. Cloning, List Operators, Dict creation syntax, deletion, Insertion Order (Dict 3.7+))**

  - **Content:**
    - **Lists:** Creation (`[]`), ordered, mutable. Indexing/slicing same as strings. Methods like `.append()`, `.sort()` (mutates), `len()`, `sum()`, `min()`, `max()`. Looping (`for item in my_list`). Highlighted aliasing (`new = old`) vs. cloning (`new = old[:]`) side effects. List operators `+` (concatenates) and `*` (repeats) do not create nested lists. Practiced methods: `.pop(index)` (returns and removes), `.remove(value)` (removes first match), `del my_list[index]` (removes by index). Difference between `.append()` (adds single element) and `.extend()` (adds elements from iterable).

        - **Dictionaries:** Creation (`{}`), storing key-value pairs. Keys must be unique and immutable, values can be anything. Accessing values (`my_dict[key]` or safer `.get(key)`). Adding/updating (`my_dict[key] = value`). Looping through `.keys()`, `.values()`, `.items()`. Practiced creation syntax (colons `:`). Correct deletion is typically `del my_dict[key]`. Noted that standard Python dictionaries (3.7+) maintain insertion order (but are not sorted by key automatically).
    - **Python Code/Syntax:**
        - List Creation: `my_list = [1, 2, 3]`
        - List Methods: `.append(item)`, `.sort()`, `len(my_list)`, `sum(my_list)`, `min(my_list)`, `max(my_list)`, `.pop(index)`, `.remove(value)`, `del my_list[index]`, `.extend(iterable)`
        - List Cloning: `new_list = old_list[:]`
        - List Operators: `+`, `*`
        - Dict Creation: `my_dict = {'key1': value1, 'key2': value2}`
        - Dict Access: `my_dict['key']`, `my_dict.get('key')`
        - Dict Update: `my_dict['key'] = new_value`
        - Dict Looping: `.keys()`, `.values()`, `.items()`
        - Dict Deletion: `del my_dict['key']`

- **[✓] Error Handling (`try/except`, specific errors, else, finally, raising explicitly concept)**
    - **Content:** Covered using `try/except` blocks to catch runtime errors. Discussed catching specific errors (e.g., `ValueError`, `TypeError`, `IndexError`, `KeyError`, `ZeroDivisionError`) versus a bare `except`. Explained the use of `else` (runs if `try` succeeds) and `finally` (always runs). Refined the conceptual understanding of why one might `raise` an exception explicitly (to signal an unrecoverable state).
    - **Python Code/Syntax:**

```
try:
    # Code that might raise an error
    ...
except SpecificErrorType:
    # Code to handle that specific error
    ...
except AnotherErrorType as e:
    # Handle another error, potentially using the error object e
    ...
else:
    # Code to run if no exception occurred in the try block
    ...
finally:
    # Code that will always run, regardless of exceptions
    ...
```

        - Raising: `raise ValueError("Informative message")`

- **[✓] Loops (`for`, `while`, `break`/`continue`, accumulator pattern, `range()`)**
    - **Content:** Covered definite iteration using `for` loops (especially with `range()`) and indefinite iteration using `while` loops (requiring initialization and updates). Explained loop control statements `break` (exits loop immediately) and `continue` (skips to next iteration). Introduced the accumulator pattern for summing or counting within loops.
    - **Python Code/Syntax:**
        - For loop: `for item in iterable: ...`
        - For loop with range: `for i in range(start, stop, step): ...`
        - While loop: `while condition: ... # Must update condition variable`
        - Control: `break`, `continue`
        - Accumulator: `total = 0; for x in data: total += x`

- **[✓] Conditionals (`if/elif/else`, Indentation consistency)**

        - **Content:** Covered conditional logic using `if`, `elif` (else if), and `else`. Emphasized Python's strict indentation rules for defining code blocks and the need for consistency.
        - **Python Code/Syntax:**

```
if condition1:
    # code block 1
elif condition2:
    # code block 2
else:
    # code block 3
```

        (Indentation is crucial)

    - **[✓] Importing Packages/Modules (Syntax, Access, Namespaces, avoiding `import *`)**

        - **Content:** Covered syntax for importing modules (`import module`, `import module as alias`, `from module import specific_object`). Explained accessing functions/variables based on import type (e.g., `module.function()` vs `function()` after `from ... import`). Highlighted the importance of exact syntax for imports and calls. Explained namespace pollution as the reason to avoid `import *`.
        - **Python Code/Syntax:**
            - `import math` -> `math.pi`
            - `import numpy as np` -> `np.array(...)`
            - `from math import pi, pow` -> `pi`, `pow(...)`
            - `from math import *` (discouraged)

    - **[✓] File I/O (Reading/Writing text files, `with open`, modes, methods (`.read`, `.readlines`, iterate), line processing loop (`.strip`, `.split`), relative vs absolute paths concept)**

        - **Content:** Covered using `with open(...) as f:` for safe file handling (auto-close). Explained file modes (`'r'` read, `'w'` write, `'a'` append). Reviewed reading methods (`.read()`, `.readlines()`, iterating `for line in f:`). Reviewed writing methods (`.write()`, `.writelines()`). Demonstrated processing lines using `.strip()`/`.rstrip('\\n')` for cleaning and `.split(delimiter)` for parsing. Mastered the looping pattern for reading and processing files line-by-line within the loop. Discussed the conceptual difference between relative and absolute paths.
        - **Python Code/Syntax:**

```
# Reading line by line
with open('myfile.txt', 'r') as f:
    for line in f:
        cleaned_line = line.strip()
        parts = cleaned_line.split(',')
        # Process parts...

# Writing
lines_to_write = ["line1\\n", "line2\\n"]
with open('output.txt', 'w') as f:
    f.write("A single line\\n")
    f.writelines(lines_to_write)

# Reading entire content
with open('myfile.txt', 'r') as f:
    content = f.read() # Reads everything into one string
    # or
    lines = f.readlines() # Reads all lines into a list of strings
```

            - Modes: `'r'`, `'w'`, `'a'`

- **[✓] Pandas:**

- **[✓] Series & DataFrames (Creation, Fundamentals, Index (default/explicit), Values (`.values` attribute), Attributes (`.index`, `.columns`, `.shape`, `.size`, `.T`), NumPy base, Series data types (object dtype))**
    - **Content:** Introduced Pandas as the essential library for tabular data, built on NumPy. Explained the two primary structures: 1D `Series` and 2D `DataFrame`. Covered creation from lists and dictionaries. Explained the `index` (automatic 0-based, explicit labels, from dict keys). Accessing underlying data via `.values` (NumPy array). `DataFrame` is a table-like collection of Series sharing an index. `DataFrame` creation from dictionaries of lists/Series (aligns by index). Basic attributes: `.index`, `.columns`, `.values`, `.shape` (returns tuple `(rows, cols)`), `.size`, `.T` (transpose). Noted Series can hold mixed types (results in `object` dtype). Clarified `.values` is the standard attribute for the backing NumPy array.
    - **Python Code/Syntax:**

```
import pandas as pd
import numpy as np

# Series creation
s_from_list = pd.Series([10, 20, 30], index=['a', 'b', 'c'])
s_from_dict = pd.Series({'a': 10, 'b': 20, 'c': 30})

# DataFrame creation
data = {'col1': [1, 2], 'col2': [3, 4]}
df_from_dict = pd.DataFrame(data, index=['row1', 'row2'])

# Attributes
print(df_from_dict.index)
print(df_from_dict.columns)
print(df_from_dict.values) # NumPy array
print(df_from_dict.shape)  # (rows, cols) tuple
print(df_from_dict.size)
print(df_from_dict.T)
print(s_from_list.dtype)
```

- **[✓] Data Indexing & Selection (`.loc`, `.iloc`, boolean masking, label vs position, slice endpoints, syntax (brackets), assignment via indexers, fancy indexing, View vs Copy distinction)**
    - **Content:** Covered label-based indexing (`[]`, `.loc[]`) and position-based indexing (`.iloc[]`) for Series and DataFrames. `.loc` uses labels (slice includes endpoint); `.iloc` uses integer positions (slice excludes endpoint). Syntax requires square brackets `[]`. Boolean masking involves creating a boolean Series based on conditions (`df['col'] > value`) and using it to filter `df[mask]` or `df.loc[mask]`. Combining conditions requires `&` (and), `|` (or), `~` (not) with parentheses. Fancy indexing uses lists of labels/positions. Assignment can be done using indexers (`df.loc[...] = value`). Boolean masking generally creates a *copy*; modifying it might not affect the original DataFrame (View vs. Copy warning). Chained assignment (`df[][]=`) is discouraged; use `.loc[row_mask, col_label] = value` for conditional modification.
    - **Python Code/Syntax:**

```
# Series Indexing
print(s_from_list['a'])       # Label
print(s_from_list.loc['a':'c']) # Label slice (inclusive)
print(s_from_list.iloc[0])      # Position
print(s_from_list.iloc[0:2])    # Position slice (exclusive)

# DataFrame Indexing
print(df_from_dict['col1'])         # Select column (returns Series)
print(df_from_dict[['col1', 'col2']]) # Select multiple columns (returns DF)
print(df_from_dict.loc['row1'])       # Select row by label
print(df_from_dict.loc['row1', 'col1']) # Select scalar by label
print(df_from_dict.loc['row1':'row2', ['col1']]) # Slice rows, select cols by label
print(df_from_dict.iloc[0])          # Select row by position
print(df_from_dict.iloc[0, 0])         # Select scalar by position
print(df_from_dict.iloc[0:1, 0:1])     # Slice rows/cols by position

# Boolean Masking
mask = df_from_dict['col1'] > 1
print(df_from_dict[mask])
print(df_from_dict.loc[mask, 'col2'])

# Combining conditions
mask_combined = (df_from_dict['col1'] > 0) & (df_from_dict['col2'] < 4)
print(df_from_dict[mask_combined])

# Fancy Indexing
print(s_from_list[['a', 'c']])
print(df_from_dict.iloc[[0, 1], [0]]) # Rows 0,1, Col 0

# Assignment
df_from_dict.loc['row1', 'col1'] = 100
df_from_dict.loc[df_from_dict['col1'] > 50, 'col2'] = 999 # Conditional assignment
```

- **[✓] Grouping (`.groupby()`, Split-Apply-Combine, Syntax, Aggregation (`.mean`, `.sum`, etc., `.describe`), Multi-column/Aggregation (`.agg`), Index result (MultiIndex), `.idxmax()`, `.size()` vs `.count()`, accessing described results)**
  - **Content:** Explained the Split-Apply-Combine concept using `.groupby()`. Syntax is `df.groupby('col_or_list')`. Aggregation functions like `.mean()`, `.sum()`, `.min()`, `.max()`, `.size()` (counts all rows per group), `.count()` (counts non-nulls per column per group), `.describe()` can be applied after grouping. Multi-column grouping or aggregation is possible (`df.groupby(...)['col'].agg_func()`, `df.groupby(...)[['col1','col2']].agg_func()`, `df.groupby(...).agg({...})`). Grouping keys become the index, potentially a `MultiIndex`. `.idxmax()` finds the index label of the max value within a group. Highlighted accessing specific stats from `.describe()` on grouped data using tuple indexing for the MultiIndex columns (`.loc[row_label, (outer_col, inner_stat)]`). Noted to operate on specific columns *after* grouping for range calculations.
  - **Python Code/Syntax:**

```
# Example DataFrame
data = {'Category': ['A', 'B', 'A', 'B'], 'Value': [10, 20, 15, 25]}
df_group = pd.DataFrame(data)

# Grouping and aggregating
grouped_mean = df_group.groupby('Category').mean()
grouped_sum = df_group.groupby('Category')['Value'].sum()
grouped_desc = df_group.groupby('Category').describe()

# Multi-aggregation
grouped_agg = df_group.groupby('Category').agg(
    mean_val=('Value', 'mean'),
    sum_val=('Value', 'sum')
)

# Accessing describe output
# print(grouped_desc.loc['A', ('Value', 'mean')])

# Size vs Count
grouped_size = df_group.groupby('Category').size()
grouped_count = df_group.groupby('Category').count() # Counts non-nulls per column

print(grouped_mean)
print(grouped_sum)
print(grouped_agg)
print(grouped_size)
print(grouped_count)
```

- **[✓] Concatenating (`pd.concat`, axis, `ignore_index`, index result precision) & Merging (`pd.merge`, keys (`on`, `left_on`/`right_on`), `how`, index merge (`left_index`/`right_index`))**
  - **Content:**
    - **Concat:** `pd.concat()` stacks DataFrames row-wise (`axis=0`, default) or column-wise (`axis=1`). Aligns on the *other* axis (outer join by default). `ignore_index=True` creates a new default 0-based index. Need precision about index result with `axis=1` (union of indices).
    - **Merge:** `pd.merge()` performs database-style joins. Uses common columns (`on=...`) or different key names (`left_on=...`, `right_on=...`). Can also merge on index (`left_index=True`, `right_index=True`). `how` argument determines which keys to keep (`'inner'`, `'outer'`, `'left'`, `'right'`).
  - **Python Code/Syntax:**

```
df1 = pd.DataFrame({'A': ['A0', 'A1'], 'B': ['B0', 'B1']}, index=[0, 1])
df2 = pd.DataFrame({'A': ['A2', 'A3'], 'B': ['B2', 'B3']}, index=[2, 3])
df3 = pd.DataFrame({'C': ['C0', 'C1'], 'D': ['D0', 'D1']}, index=[0, 1])

# Concat Rows
row_concat = pd.concat([df1, df2]) # axis=0 default
row_concat_new_index = pd.concat([df1, df2], ignore_index=True)

# Concat Columns
col_concat = pd.concat([df1, df3], axis=1)

# Merge
left = pd.DataFrame({'key': ['K0', 'K1'], 'A': ['A0', 'A1']})
right = pd.DataFrame({'key': ['K0', 'K1'], 'B': ['B0', 'B1']})
inner_merge = pd.merge(left, right, on='key', how='inner') # Default how='inner'
outer_merge = pd.merge(left, right, on='key', how='outer')

print("Row Concat:\n", row_concat)
print("\\nCol Concat:\n", col_concat)
print("\\nInner Merge:\n", inner_merge)
```

- **[✓] Reading/Writing Files (CSV (`pd.read_csv`), Excel (`pd.read_excel`)) from/into DataFrames (incl. common arguments like `delimiter`, `usecols`, `sheet_name`, `index` for `to_csv`)**
  - **Content:** Covered reading data from CSV files using `pd.read_csv()` (common arguments: `delimiter`, `usecols`) and from Excel files using `pd.read_excel()` (common argument: `sheet_name`). Covered writing DataFrames to CSV using `.to_csv()` (common argument: `index=False` to prevent writing the DataFrame index as a column).
  - **Python Code/Syntax:**

```
# Reading CSV
df_csv = pd.read_csv('data.csv', usecols=['col1', 'col3'])

# Reading Excel
df_excel = pd.read_excel('data.xlsx', sheet_name='Sheet1')

# Writing CSV
df_excel.to_csv('output.csv', index=False)
```

(Actual file operations commented out as they require files)

- **[✓] Basic Plotting from DataFrames (`.plot()`, Seaborn mentioned, aggregation need, plot interpretation)**
  - **Content:** Introduced the basic `.plot()` method available on DataFrames/Series for simple visualizations (e.g., line plots by default). Mentioned Seaborn (`sns`) for enhanced plots (using `hue` argument to map color). Noted that data aggregation is often required before plotting grouped results. Emphasized that plotted values reflect the aggregated data used (sum, mean, etc.).
  - **Python Code/Syntax:**

```
# Assuming df_plot is a DataFrame with suitable data
df_plot['column_to_plot'].plot() # Basic line plot
df_plot.plot(kind='bar')        # Basic bar plot

# import seaborn as sns
grouped_data = df.groupby('category').mean()
sns.barplot(x=grouped_data.index, y='value_column', data=grouped_data)
```

(Requires matplotlib/seaborn installed and actual data)

- **[✓] Statistics (Intro):**
    - **Content:** Basic descriptive statistics were covered primarily through integrated Pandas methods applied to Series and DataFrames, such as `.mean()`, `.median()`, `.std()`, `.var()`, `.describe()`, `.count()`, `.sum()`, `.min()`, `.max()`, `.unique()`, `.nunique()`.
- **[✓] Note: Database knowledge is *not required*.**
    - **Content:** Explicitly stated in the Week 1-4 overview slide that database interactions are not required for the exam.

Week 6: Data Handling, Stats & Web Interaction

- **[✓] Scientific Computing Concepts: Reproducibility & Replication (Validity distinction)**
    - **Content:** Introduced the crucial concepts in scientific computing:
        - **Reproducibility:** Getting the same computational result using the original data and code. (Verifies the computation).
        - **Replication:** Achieving the same scientific finding, possibly using new data or methods. (Verifies the finding).
        - *Quiz/Assignment Feedback Highlight:* Emphasized that reproducibility does *not* guarantee scientific validity or generalizability. A flawed experiment can be reproducible.
    - **Python Code/Syntax:** Not applicable (Conceptual).
- **[✓] Data Types in Finance: Cross-sectional, Time series, Panel/Longitudinal**
    - **Content:** Differentiated structural data types based on how they handle subjects (e.g., firms, individuals) and time:
        - **Cross-sectional:** Data on multiple subjects at a single point in time.
        - **Time series:** Data for a single subject over multiple time periods.
        - **Panel (or Longitudinal):** Data for multiple subjects observed over multiple time periods.
    - **Python Code/Syntax:** Not applicable (Conceptual classification of data).
- **[✓] Data Classification: Quantitative vs. Qualitative (Nominal, Ordinal subtypes) (Raw text as unstructured)**
    - **Content:** Covered the nature of data:
        - **Quantitative:** Numerical, measurable/countable data (e.g., prices, volume, financial ratios).
        - **Qualitative:** Descriptive categories (e.g., text, names, sectors, ratings). Subtypes include:
            - **Nominal:** Categories with no inherent order (e.g., industry sectors).
            - **Ordinal:** Categories with a meaningful order (e.g., credit ratings).
        - *Quiz/Assignment Feedback Highlight:* Noted that raw text is typically considered **unstructured** data.
    - **Python Code/Syntax:** Not applicable (Conceptual classification of data variables).
- **[✓] Financial Returns:**
    - **[✓] Log Returns (Calculation, Additivity benefit)**
        - **Content:** Defined Log Returns (also known as continuously compounded returns). Formula: `Log_Return_t = Log(Price_t) - Log(Price_{t-1})` or equivalently `Log(Price_t / Price_{t-1})`. Key advantage noted is **additivity** over time (total log return = sum of periodic log returns), making them preferable for multi-period analysis. Calculation requires a time series of prices.
        - **Python Code/Syntax & Example (from Week 7 Notebook):**

```
import numpy as np
import pandas as pd

# Assuming stock_final_data is a DataFrame with 'date' index and price columns ('AAPL', 'AMZN')
# Method 1: Log difference
stock_log_ret_m1 = np.log(stock_final_data) - np.log(stock_final_data.shift(1))

# Method 2: Log of ratio
stock_log_ret_m2 = np.log(stock_final_data / stock_final_data.shift(1))

# Using pandas diff (for price changes) then log - Needs price series, not return series
# log_ret_m3 = np.log(prices_df).diff() # Equivalent to Method 1 after taking log of prices

print(stock_log_ret_m1.head())
```

Quiz/Assignment Feedback Highlight: Emphasized the correct calculation using `shift()` or `.diff()` on log prices, noting that subtracting prices *before* logging is incorrect.

- **[✓] Simple Returns (Calculation, Normalization benefit)**
  - **Content:** Defined Simple Returns (or arithmetic returns). Formula: `Simple_Return_t = (Price_t - Price_{t-1}) / Price_{t-1}`. Benefit mentioned is normalization (expressing change relative to the starting value). Can be calculated using pandas `.pct_change()`.
  - **Python Code/Syntax & Example:**

```
# Assuming stock_final_data is a DataFrame with price columns
# Method 1: Manual calculation
simple_ret_m1 = (stock_final_data - stock_final_data.shift(1)) / stock_final_data.shift(1)

# Method 2: Using pandas pct_change()
simple_ret_m2 = stock_final_data.pct_change()

print(simple_ret_m2.head())
```

- **[✓] Descriptive Statistics (Concepts & Calculation):**
  - **[✓] Mean, Median, Mode (Central tendency)**
    - **Content:** Reviewed measures of central tendency: Mean (average), Median (middle value), Mode (most frequent value). Expected Return often proxied by the historical mean of returns.
    - **Python Code/Syntax & Example (using log returns DataFrame `stock_log_ret`):**

```
# Mean (Daily Expected Return example)
daily_mean_ret = stock_log_ret.mean()
print("Daily Mean Log Return:\\n", daily_mean_ret)

# Median
daily_median_ret = stock_log_ret.median()
print("\\nDaily Median Log Return:\\n", daily_median_ret)

# Mode (less common for continuous returns, might need binning)
# daily_mode_ret = stock_log_ret['AAPL'].mode()
# print("\\nDaily Mode Log Return (AAPL):\\n", daily_mode_ret)
```

  - **[✓] Variance, Standard Deviation (Volatility) (Dispersion)**
    - **Content:** Reviewed measures of dispersion: Variance (average squared deviation from the mean, units are squared) and Standard Deviation (square root of variance, same units as the data, more interpretable measure of volatility/risk).

        - **Python Code/Syntax & Example:**

```
# Variance
daily_variance = stock_log_ret.var()
print("Daily Log Return Variance:\\n", daily_variance)

# Standard Deviation
daily_std_dev = stock_log_ret.std()
print("\\nDaily Log Return Std Dev:\\n", daily_std_dev)

# Alternative for Std Dev
daily_std_dev_alt = np.sqrt(daily_variance)
print("\\nDaily Log Return Std Dev (Alt):\\n", daily_std_dev_alt)
```

    - **[✓] Correlation Matrices**
        - **Content:** Briefly touched upon Correlation Matrices as a way to view the co-movement between multiple assets. Correlation measures the strength and direction of linear relationship (-1 to +1). Crucial for understanding diversification.
        - **Python Code/Syntax & Example:**

```
# Correlation Matrix for log returns
correlation_matrix = stock_log_ret.corr()
print("Log Return Correlation Matrix:\\n", correlation_matrix)

# Covariance Matrix (related, used in portfolio variance)
covariance_matrix = stock_log_ret.cov()
print("\\nLog Return Covariance Matrix:\\n", covariance_matrix)
```

- **[✓] HTTP Concepts:**

    - **[✓] Standard protocol purpose (Client-Server Model, WWW vs Internet, TCP/IP role)**
        - **Content:** HTTP (Hypertext Transfer Protocol) is the application-layer protocol for the World Wide Web, used for requesting and transmitting web pages/resources. Operates within a Client-Server model (client requests, server responds). Differentiated the Web (information system using HTTP/HTML) from the Internet (underlying global TCP/IP infrastructure). HTTPS is HTTP secured via SSL/TLS, which runs over TCP/IP.
    - **[✓] HTTP Request (Request Line, Headers (Host, User-Agent, Accept), Body, Method/URL/Version)**
        - **Content:** Detailed the structure of an HTTP Request:
            - **Request Line:** Contains the Method (e.g., GET, POST), the URL (resource path), and the HTTP Version.
            - **Headers:** Provide additional information (e.g., `Host`: domain name, `User-Agent`: client software, `Accept`: acceptable content types). `Host` header is crucial for virtual hosting. `User-Agent` is important for politeness, content negotiation (mobile vs. desktop), and avoiding blocks.
            - **Body:** Optional section containing data sent *to* the server, typically used with POST requests (e.g., form data).
    - **[✓] Request Methods (GET vs. POST - contrast, HTTPS)**
        - **Content:** Distinguished key methods:
            - **GET:** Requests data from a specified resource. Can be cached, bookmarked, parameters appended to URL. Should not be used for sensitive data due to visibility. Length restrictions apply.
            - **POST:** Submits data to be processed to a specified resource (e.g., create/update). Not cached, not bookmarked, data sent in request body. No restrictions on data length. Preferred for sensitive data.
            - **HTTPS:** Mentioned as HTTP secured via SSL/TLS encryption and authentication.
    - **[✓] HTTP Response (Status Line, Headers (Content-Type/Length), Body, Version/Code/Reason)**
        - **Content:** Detailed the structure of an HTTP Response:
            - **Status Line:** Contains the HTTP Version, Status Code (e.g., 200), and Reason Phrase (e.g., OK).
            - **Headers:** Provide additional information about the response (e.g., `Content-Type`: MIME type of the body, `Content-Length`: size of the body).
            - **Body:** Contains the requested resource/content (e.g., HTML page, JSON data).
    - **[✓] Status Codes (Categories: 2xx, 4xx, 5xx, specific checks, DELETE success code, state management via Cookies, Host header purpose, User-Agent role)**

        - **Content:** Explained common Status Code categories: `2xx` (Success), `4xx` (Client Error, e.g., 404 Not Found), `5xx` (Server Error). Noted specific checks should use explicit comparisons (e.g., `response.status_code == 200` or `200 <= response.status_code < 300`). Highlighted that a successful DELETE often returns `200 OK` or `204 No Content`. Also mentioned Cookies as a mechanism for state management (sessions, logins) across stateless HTTP requests.
    - **Python Code/Syntax:** Primarily related to the `requests` library below.

- **[✓] Python Web Tools (Conceptual Understanding & Basic Usage):**

    - **[✓] `requests` library (import, get/post, headers/params, status_code, content access (`.text`, `.content`, `.json`), error handling (specific exceptions), rate limiting (`time.sleep`), `.json()` method vs `json.loads`, `data`/`json` args for POST, API key passing)**

        - **Content:** Practiced basic usage for making HTTP requests. Covered importing, making GET (`requests.get()`) and POST (`requests.post()`) requests, passing headers (`headers=...`) and URL parameters (`params=...`), checking the response status code (`response.status_code`), and accessing response content (`response.text` for text, `response.content` for bytes, `response.json()` for JSON decoding). Included basic `try...except` error handling (specifically mentioning `requests.exceptions.Timeout`, `ConnectionError`, `JSONDecodeError`) and using `time.sleep()` for rate limiting.
        - *Quiz/Assignment Feedback Highlight:* Noted using `.text` attribute vs `.text()`, using `.json()` method vs `json.loads()`, using `data=` for form data or `json=` for JSON payload in POST, and passing API keys usually via headers or params per API documentation.
        - **Python Code/Syntax & Example:**

```
import requests
import time
import json # Import json module for potential direct use, though response.json() is common

url = 'https://httpbin.org/get'
params = {'key1': 'value1'}
headers = {'User-Agent': 'MyPythonScript/1.0'}

try:
    response = requests.get(url, params=params, headers=headers, timeout=5)
    response.raise_for_status() # Raise an exception for bad status codes (4xx or 5xx)

    print("Status Code:", response.status_code)

    # Access content
    print("Text Content:", response.text[:100]) # First 100 chars
    # print("Byte Content:", response.content[:100])

    # Access JSON content
    json_data = response.json()
    print("JSON Args:", json_data.get('args'))

except requests.exceptions.Timeout:
    print("Request timed out")
except requests.exceptions.RequestException as e:
    print(f"An error occurred: {e}")
except json.JSONDecodeError: # Catching potential error if response.json() fails
     print("Failed to decode JSON response")

# Example POST
post_url = 'https://httpbin.org/post'
payload_form = {'key': 'value'}
payload_json = {'json_key': 'json_value'}
# response_post = requests.post(post_url, data=payload_form) # Form data
# response_post_json = requests.post(post_url, json=payload_json) # JSON data

# Rate limiting
# time.sleep(1) # Pause for 1 second
```

- **[✓] `BeautifulSoup` library (Parsing HTML (`html.parser`), Navigating (tags), `find()`/`find_all()` (by tag/attribute, `class_`), Extracting text/attributes (`.text`, `.get_text`, `['attr']`, `.get()`), handling `None` result, nested search logic, tag vs content)**
  - **Content:** Covered using BeautifulSoup for parsing HTML content obtained via `requests`. Basic usage involves creating a `BeautifulSoup` object (`BeautifulSoup(html_content, 'html.parser')`). Explained navigating/searching using direct tag access (`soup.tag`), finding the first element (`.find()`) or all elements (`.find_all()`) by tag name and/or attributes (like `id` or `class_`). Note the use of `class_` (with underscore) because `class` is a Python keyword. Covered extracting text content (`.text` or `.get_text()`) and attribute values (`tag['attr']` or `tag.get('attr')`).
  - *Quiz/Assignment Feedback Highlight:* Noted `.find()` returns `None` on failure while `.find_all()` returns an empty list `[]`. Emphasized checking for `None` before accessing attributes/text when looping through results. Mastered the pattern of finding a container element then finding items *within* that container (nested searching). Differentiated getting the tag object itself vs. its inner text content.
  - **Python Code/Syntax & Example:**

```
from bs4 import BeautifulSoup

# Assume html_doc contains HTML content from a requests response (e.g., response.text)
html_doc = """
<html><head><title>My Title</title></head>
<body><p class="content" id="para1">Some text.</p>
<p class="content">More text.</p>
<a href="http://example.com">Link</a>
<div id="container"> <p>Nested text</p> </div>
</body></html>
"""

soup = BeautifulSoup(html_doc, 'html.parser')

# Find by tag
title_tag = soup.find('title')
print("Title Tag:", title_tag)
print("Title Text:", title_tag.text if title_tag else "No title found")

# Find first paragraph by class
first_p = soup.find('p', class_='content')
print("First Paragraph:", first_p.get_text() if first_p else "No paragraph found")

# Find all paragraphs by class
all_p_content = soup.find_all('p', class_='content')
print("All content paragraphs:", [p.text for p in all_p_content])

# Find by ID
para1 = soup.find(id='para1')
print("Para 1 text:", para1.text if para1 else "No para1 found")

# Get attribute
link_tag = soup.find('a')
print("Link Href:", link_tag['href'] if link_tag else "No link found")
# Safer way: print("Link Href:", link_tag.get('href') if link_tag else "No link found")

# Nested Search
container = soup.find('div', id='container')
if container:
    nested_p = container.find('p')
    print("Nested Text:", nested_p.text if nested_p else "No nested p found")
```

- **[✓] API (Definition, Reliability vs Scraping, JSON/XML format, Keys, Rate Limiting, Wrappers)**
  - **Content:** Defined APIs (Application Programming Interfaces) as structured ways for programs to interact, contrasting their reliability and structured data return types (usually JSON/XML) with the often less stable HTML scraping approach. Understood the role of API keys (for authentication/authorization) and rate limiting (to prevent abuse). Recognized that wrapper libraries (like `alpha_vantage` used in the notebook) simplify API calls by handling request formatting and response parsing.
  - **Python Code/Syntax:** The Week 7 notebook used the `alpha_vantage` wrapper library, demonstrating API interaction indirectly.

```
# Example using the wrapper library (conceptual, details in notebook)
from alpha_vantage.timeseries import TimeSeries

# API Key is passed during initialization or specific calls
ts = TimeSeries(key='YOUR_API_KEY', output_format='pandas')

# Wrapper handles the actual HTTP request and JSON parsing
# data, meta_data = ts.get_daily(symbol='AAPL', outputsize='full')
```

Week 7: Market Microstructure (Take Home Lecture Notes)

- **[✓] Assets: Stocks, Bonds, Options, Forex, ETF, Cryptocurrency**
    - **Content:** Overview of various financial instruments traded in markets.
        - **Stocks (Shares/Equity):** Security signifying proportionate ownership in a corporation, entitling holders to a share of assets and earnings (Slide 5).
        - **Bonds:** Fixed income instrument representing a loan from an investor to a borrower (Slide 6).
        - **Options:** Agreement giving the buyer the right (not obligation) to buy or sell an asset at a specific price on or before a certain date (Slide 7).
        - **FOREX (Foreign Exchange):** Decentralized global market for trading world currencies; highly liquid, $5+ trillion daily volume, active 24/7 (Slide 8).
        - **Cryptocurrencies:** Digital assets designed as a medium of exchange, generally decentralized and not regulated by a single authority (Slide 9).
        - **ETF (Exchange-Traded Fund):** Basket of securities (like stocks) tracking an underlying index, traded on an exchange like a single stock (Slide 19).
        - *(Funds also mentioned as an asset type but not detailed separately).*
    - **Python Code/Syntax:** Not Applicable (Conceptual definitions).
- **[✓] Market Players: Exchanges, Broker/dealer, Buy-side/Sell-side**
    - **Content:** Identified the main participants involved in trading.
        - **Exchanges:** Institutions hosting markets for trading securities (stocks, bonds, etc.), imposing rules and regulations (e.g., NYSE, NASDAQ) (Slide 10).
        - **Broker/Dealer:** Person or firm buying/selling securities. Acts as a **Broker** (agent) executing orders for clients. Acts as a **Dealer** (principal) trading for its own account (Slide 11).
        - **Buy-Side:** Institutional/individual investors purchasing assets for their company or clients (e.g., mutual funds, pension funds, individuals) (Slide 12).
        - **Sell-Side:** Brokers and dealers facilitating the issuance and selling of assets (e.g., investment banks, market makers) (Slide 12).
    - **Python Code/Syntax:** Not Applicable (Conceptual definitions).
- **[✓] Execution Methods:**
    - **[✓] Algorithm (Definition, Generations: TWAP/VWAP, TCA, Liquidity Search/ECN; Gen 1 weakness, Info Leakage risk)**
        - **Content:** Defined as an automated trading approach using computer algorithms to create and send orders based on predefined criteria (Slide 3, 18, 25). Algorithms evolved from simple slicing (breaking large orders) to targeting best execution.
            - *1st Gen:* Focused on benchmarks like TWAP (Time Weighted Average Price) and VWAP (Volume Weighted Average Price); used static schedules, vulnerable to pattern detection (Slide 27).
            - *2nd Gen:* Incorporated Transaction Cost Analysis (TCA), recognizing costs beyond market impact like timing risk and opportunity cost; became price/risk sensitive (Slide 28).
            - *3rd Gen:* Focused on searching for liquidity across proliferating electronic venues (ECNs, ATS, dark pools); complex liquidity-based routing (Slide 29-30). Advanced algos risk information leakage ("signaling risk").
    - **[✓] Direct Market Access (DMA) (Definition, Broker infrastructure, Pre-trade Risk controls)**
        - **Content:** Brokers allow clients access to their order routing infrastructure, enabling buy-side clients to send electronic orders almost directly to exchanges, giving them control similar to sell-side traders (Slide 15, 32, 33). Key concern is information leakage; services often run as separate entities. Broker typically provides pre-trade risk controls (Slide 34, Day 4 Summary).
    - **[✓] Sponsored Access (Definition, Client infrastructure via Broker ID, Pre-trade Risk controls, HFT use case, Naked Access risk)**

        - **Content:** Takes DMA further, allowing clients (often HFTs needing ultra-low latency) to connect directly to the market using the *broker's trading identifier* but often using their *own infrastructure* (Slide 15, 34). Pre-trade risk monitoring *can* be done by the broker (adds latency) or responsibility may shift more to the client. **Naked Access** is a form where monitoring is only post-trade, posing risks as the broker cannot prevent erroneous trades hitting the market under their name (Slide 34).
    - **Python Code/Syntax:** Not Applicable (Conceptual descriptions of execution methods).

- **[✓] Markets: Primary vs. Secondary**
    - **Content:** Distinguished between the two main market categories:
        - **Primary Market:** Deals with the issuance of new assets/securities (e.g., IPOs, government bond auctions) (Slide 42, 43).
        - **Secondary Market:** Involves the subsequent trading of previously issued assets. Important for providing liquidity, allowing investors to readily trade assets (Slide 42, 44). Market microstructure primarily focuses on secondary market efficiency.
    - **Python Code/Syntax:** Not Applicable (Conceptual definitions).

- **[✓] Liquidity: Depth, Tightness (Bid-Ask Spread), Resilience**
    - **Content:** Defined liquidity as a vital aspect of trading, associated with lower trading costs and higher volumes. Characterized by:
        - **Depth:** Total quantity of buy/sell orders available around the equilibrium price. Deep markets allow large trades without sizable price impact (Slide 45, 46).
        - **Tightness:** The bid-ask (or bid-offer) spread – the difference between the best price to sell and the best price to buy (Slide 45, 46).
        - **Resilience:** How quickly the market price recovers from a temporary shock or large trade (Slide 45, 46).
    - **Python Code/Syntax:** Not Applicable (Conceptual definitions).

- **[✓] Market Types: Quote-driven vs. Order-driven (Trading Mechanisms)**
    - **Content:** Explained the main trading mechanisms:
        - **Quote-Driven:** Traders transact with dealers/market makers who quote bid and ask prices (e.g., Bonds, Currencies) (Slide 50, 51).
        - **Order-Driven:** All traders can submit orders to a central order book; trades occur when buy/sell orders match based on set rules (e.g., most stock exchanges). Prices are set by orders, not dealer quotes. Offers visible liquidity and potentially more control (Slide 50, 52, 53).
    - **Python Code/Syntax:** Not Applicable (Conceptual definitions).

- **[✓] Trading Frequency: Continuous, Periodic, Request-driven**
    - **Content:** Described how often trades can occur:
        - **Continuous:** Trading can happen at any point during market hours (can lead to volatility) (Slide 54).
        - **Periodic:** Trading occurs only at scheduled times (e.g., opening/closing auctions) (Slide 54).
        - **Request-driven:** Trading occurs after requesting a quote from a market maker (less efficient on price) (Slide 54).
    - **Python Code/Syntax:** Not Applicable (Conceptual definitions).

- **[✓] Order Types: Market Order vs. Limit Order**
    - **Content:** Defined the two main order types:
        - **Market Order:** Instruction to trade immediately at the best available price. Execution is prioritized, but the price is uncertain (Slide 56). Demands liquidity.
        - **Limit Order:** Instruction to trade only at a specified price or better (max buy price, min sell price). Price is controlled, but execution is not guaranteed (Slide 56). Provides liquidity.
    - **Python Code/Syntax:** Not Applicable (Conceptual definitions).

- **[✓] Fill Instructions: Immediate-or-Cancel (IOC), Fill or Kill (FOK), All-or-None (AON)**
    - **Content:** (From supplementary extraction) Additional conditions for order execution, typically with limit orders.
        - **IOC:** Execute immediately, all or part; cancel any unfilled portion. Partial fills allowed.
        - **FOK:** Execute entire quantity immediately or cancel the whole order. No partial fills.
        - **AON:** Execute entire quantity, but not necessarily immediately; rests until full quantity available. No partial fills.
    - **Python Code/Syntax:** Not Applicable (Conceptual definitions).

- **[✓] Order Concepts: Preference & Directed Orders**
    - **Content:** (From supplementary extraction) Instructions specifying initial order destination.
        - **Preference Order:** Instruction to route to a specific dealer/market maker (historical concept).
        - **Directed Order:** Instruction to route to a specific exchange or trading venue first.

    - **Python Code/Syntax:** Not Applicable (Conceptual definitions).

- **[✓] Routing Instructions: Do-not-route (DNR / DNT), Directed-routing, Intermarket sweep (ISO)**

    - **Content:** (From supplementary extraction) Govern interaction with multiple venues.
        - **Do-not-route (DNR/DNT):** Instruction telling the venue *not* to route the order elsewhere, even for a better price. Used to add liquidity to a specific venue.
        - **Directed-routing:** Largely overlaps with Directed Orders concept.
        - **Intermarket Sweep Order (ISO):** Specialized limit order(s) to execute rapidly against liquidity across *several venues simultaneously*, complying with trade-through rules. Used to take liquidity quickly.
    - **Python Code/Syntax:** Not Applicable (Conceptual definitions).

- **[✓] Transaction Costs: Investment-related & Trading-related (TCA context)**

    - **Content:** Transaction Cost Analysis (TCA) breaks down trading costs. Highlighted that costs go beyond simple market impact. Key *trading-related* components discussed in the context of algorithm evolution include:
        - **Market Impact:** The effect the order itself has on the asset's price.
        - **Timing Risk:** Risk associated with executing over time vs. immediately.
        - **Opportunity Cost:** Cost of missed favorable price movements while working an order.
        - *(Note: Explicit list distinguishing Investment-related (Taxes, Delay) wasn't found in the provided text, but the TCA context covers major trading cost components.)* (Slide 28)
    - **Python Code/Syntax:** Not Applicable (Conceptual definitions).

- **[✓] Trade Analysis: Pre-trade vs. Post-trade Analysis**

    - **Content:**
        - **Pre-trade Analysis:** Analysis done *before* executing a trade. Mentioned in the context of broker monitoring for sponsored access to ensure no excessive risks are taken (Slide 34). Implies assessing potential costs, risks, and market conditions beforehand.
        - **Post-trade Analysis:** Analysis done *after* a trade is completed. Implied through the discussion of Transaction Cost Analysis (TCA), which measures the actual costs incurred (market impact, timing risk, opportunity cost) to evaluate execution quality (Slide 28).
    - **Python Code/Syntax:** Not Applicable (Conceptual definitions).

Weeks 7-9: Risk, Return & Portfolio Management

- **[✓] Risk & Return Basics:**

    - **[✓] Realized Return (Definition via formula)**

        - **Content:** The actual return achieved over a period. The Week 7 Slides (Slide 3) show a formula including dividends and capital gains: `Rit = (Dit + Pit - P(it-1)) / P(it-1) = Dit/P(it-1) + (Pit - P(it-1))/P(it-1)`. In practice, especially with daily data, often simplified to price changes (Simple or Log returns).
        - **Python Code/Syntax:** Calculated using Simple (`.pct_change()`) or Log Returns (see below) on price data. Realized returns are the *outputs* of these calculations.

    - **[✓] Expected Return (Definition, Weighted Average, Historical Proxy, Calculation, Portfolio calc)**

        - **Content:** The anticipated return on an investment. Often proxied by the historical average (mean) of realized returns, assuming past performance is indicative (Week 7 Slides, Slide 3, 4; Week 7 Notebook). The expected return of a portfolio is the weighted average of the expected returns of its constituent assets (Week 7 Slides, Slide 16; Week 8 Slides, Slide 14, 15, 18). Formula: `E[Rp] = w1*E[R1] + w2*E[R2] + ... + wn*E[Rn]`. Represented in matrix form as `E[Rp] = w.T @ E[R]` (Day 5 Summary).
        - **Python Code/Syntax & Example (Individual Asset & Portfolio):**

```
import numpy as np
import pandas as pd
# Assuming stock_log_ret is a DataFrame of daily log returns for AAPL, AMZN

# Individual Asset Daily Expected Return (using historical mean)
aapl_daily_er = stock_log_ret['AAPL'].mean()
amzn_daily_er = stock_log_ret['AMZN'].mean()
print(f"AAPL Daily ER: {aapl_daily_er:.6f}")
print(f"AMZN Daily ER: {amzn_daily_er:.6f}")

# Individual Asset Annualized Expected Return (for log returns)
aapl_ann_ret = aapl_daily_er * 252
amzn_ann_ret = amzn_daily_er * 252
print(f"AAPL Annualized ER: {aapl_ann_ret:.4f}")
print(f"AMZN Annualized ER: {amzn_ann_ret:.4f}")

# Portfolio Expected Return (Daily)
portfolio_weights = np.array([0.5, 0.5]) # Equal weights for AAPL, AMZN
individual_ers = stock_log_ret.mean() # Series: AAPL ER, AMZN ER
portfolio_daily_er = np.sum(individual_ers * portfolio_weights)
# Alternative using dot product for >2 assets
# portfolio_daily_er = np.dot(portfolio_weights, individual_ers)
print(f"Portfolio Daily ER: {portfolio_daily_er:.6f}")

# Portfolio Expected Return (Annualized)
portfolio_ann_ret = portfolio_daily_er * 252
print(f"Portfolio Annualized ER: {portfolio_ann_ret:.4f}")
```

- **[✓] Excess Return (Definition, Formula, relation to Risk Premium)**
  - **Content:** The return earned above the risk-free rate (`Rf`). Formula: `Excess Return = Rit - Rf`. It represents the extra reward for taking on risk compared to a risk-free investment. Can be seen as a single period's realization of the risk premium. (Week 7 Slides, Slide 3; Week 7 Slides, Slide 11 - Alpha context).
  - **Python Code/Syntax:** `excess_return_series = return_series - risk_free_rate` (where `risk_free_rate` is the periodic risk-free rate).
- **[✓] Risk Premium (Definition, Formula, Market Risk Premium)**
  - **Content:** The *expected* return on a risky asset minus the risk-free rate. Formula: `Risk Premium = E[Rit] - Rf`. Represents the expected compensation for bearing the risk of a particular asset. The Market Risk Premium is specifically `E[Rm] - Rf`, the expected return of the overall market portfolio above the risk-free rate. (Week 7 Slides, Slide 3; Week 7 Notebook - CAPM context).
  - **Python Code/Syntax:** Calculated using the expected return (e.g., `aapl_ann_ret`) and an assumed risk-free rate: `aapl_risk_premium = aapl_ann_ret - annual_risk_free_rate`.
- **[✓] Mean, Variance, Standard Deviation (as measures, calculation (Pandas/Numpy), interpretation, formulas (population/sample), annualized volatility)**
  - **Content:** Fundamental statistical measures used to quantify return (Mean) and risk (Variance/Std Dev).
    - **Mean:** Average return, used as proxy for Expected Return (Week 7 Slides, Slide 4).
    - **Variance:** Measures dispersion around the mean (average squared deviation). Higher variance indicates greater fluctuation/volatility. Formula involves `E[(Rit - μi)^2]` (population) or sum of squared deviations divided by `n-1` (sample). It weighs outliers more heavily due to squaring (Week 7 Slides, Slide 4, 22, 24; Week 8 Slides, Slide 10).
    - **Standard Deviation:** Square root of variance. Measures dispersion in the *same units* as the return, making it more interpretable than variance as a measure of volatility/risk. Smaller std dev means less volatile (Week 7 Slides, Slide 4, 11, 23, 24; Week 8 Slides, Slide 10).
    - **Annualization:** For log returns, daily variance is multiplied by the number of trading days (e.g., 252), and daily standard deviation is multiplied by the square root of trading days (e.g., `sqrt(252)`) (Day 4 Summary; Week 8 Slides, Slide 22).

        - **Python Code/Syntax & Example:**

```
# Using stock_log_ret DataFrame

# Mean (Daily)
daily_means = stock_log_ret.mean()
print("Daily Means:\\n", daily_means)

# Variance (Daily, Sample Var assuming pandas default ddof=1)
daily_vars = stock_log_ret.var()
print("\\nDaily Variances:\\n", daily_vars)

# Standard Deviation (Daily, Sample Std Dev)
daily_stds = stock_log_ret.std()
print("\\nDaily Std Devs:\\n", daily_stds)

# Annualized Volatility (Standard Deviation)
annualized_stds = daily_stds * np.sqrt(252)
print("\\nAnnualized Volatility:\\n", annualized_stds)

# Annualized Variance
annualized_vars = daily_vars * 252
print("\\nAnnualized Variance:\\n", annualized_vars)
```

    - **[✓] Correlation (as measure, calculation (`.corr()`), interpretation (-1 to +1, strength/direction), formula, role in diversification, Covariance vs Corr definition)**
        - **Content:** Measures the degree and direction of *linear* association between two variables (e.g., asset returns). Range is -1 (perfect negative correlation) to +1 (perfect positive correlation). 0 indicates no linear correlation. Formula standardizes covariance: `Corr(Ri, Rj) = Cov(Ri, Rj) / (σi * σj)`. Crucial for diversification: combining assets with low or negative correlation reduces portfolio risk (Week 7 Slides, Slide 5, 6, 28; Day 4 Summary; Week 8 Slides, Slide 38). Covariance measures joint variability but its scale is hard to interpret directly, unlike correlation.
        - **Python Code/Syntax & Example:**

```
# Correlation matrix for log returns
corr_matrix = stock_log_ret.corr()
print("Correlation Matrix:\\n", corr_matrix)

# Accessing specific correlation
aapl_amzn_corr = corr_matrix.loc['AAPL', 'AMZN']
print(f"\\nAAPL-AMZN Correlation: {aapl_amzn_corr:.4f}")
```

    - **[✓] Difference between Variance & Standard Deviation (Interpretation, units, weighting outliers)**
        - **Content:** Both measure dispersion/volatility. Variance uses squared units, making it harder to interpret directly but emphasizing outliers. Standard Deviation is the square root of variance, returning the measure to the original units of the data (e.g., % return), making it more intuitive for assessing risk ranges (Week 7 Slides, Slide 24, 26).
        - **Python Code/Syntax:** Demonstrated in the Mean/Var/Std Dev section above.

- **[✓] Market Properties: Efficient Markets (Concept, Properties)**
    - **Content:** A well-functioning, highly competitive market where prices reflect all available information. Properties include: Returns are random/unpredictable; prices react quickly and correctly to new information; investors cannot earn *abnormal*, risk-adjusted returns (alpha ≈ 0) consistently. (Week 7 Slides, Slide 7).
    - **Python Code/Syntax:** Not Applicable (Conceptual).
- **[✓] Risk Categories: Systematic vs. Unsystematic Risk (Definitions, Market/Undiversifiable vs Diversifiable/Firm-Specific, relation to diversification/hedging)**

    - **Content:** Risk is broadly categorized into two types:
        - **Systematic Risk:** Affects the overall market (Market Risk, Undiversifiable Risk, Volatility Risk). Unpredictable and cannot be eliminated through diversification. Examples include changes in interest rates, economic recessions, political events. Can potentially be mitigated through hedging (e.g., options) (Week 7 Slides, Slide 10; Week 8 Slides, Slide 29, 36). Measured by Beta.
        - **Unsystematic Risk:** Specific to a company or sector (Diversifiable Risk, Firm-Specific Risk). Can be reduced or eliminated through diversification by combining assets that are not perfectly correlated (Week 7 Slides, Slide 10; Week 8 Slides, Slide 36; Day 5 Summary).
    - **Python Code/Syntax:** Not Applicable (Conceptual).

- **[✓] Common Risk Measures: Standard Deviation, Alpha (α), Beta (β) (Definitions, Formulas, Calculation via Regression (`statsmodels`), Interpretation)**
    - **Content:**
        - **Standard Deviation:** Measures total risk (systematic + unsystematic) or dispersion of returns around the mean (Week 7 Slides, Slide 11). (Calculation shown previously).
        - **Beta (β):** Measures systematic risk relative to the market. `β = Cov(Rs, Rm) / Var(Rm)`. Market beta = 1. Beta > 1 is more volatile than market, Beta < 1 is less volatile (Week 7 Slides, Slide 11, 13). Calculated via regression `Rp = α + βp * Rm + ε` (Week 7 Notebook).
        - **Alpha (α):** Measures risk-adjusted excess return above benchmark expectations (often based on CAPM/Beta). `α = Rp - (Rf + βp * (Rm - Rf))`. Positive alpha suggests outperformance not explained by systematic risk (Week 7 Slides, Slide 11). Calculated as the intercept in the regression `Rp = α + βp * Rm + ε` (Week 7 Notebook).
    - **Python Code/Syntax & Example (Beta & Alpha from Regression):**

```
import statsmodels.api as sm
# Assuming stock_port_ret contains 'Portfolio' returns
# and 'Mkt' contains corresponding market returns (e.g., SPY log returns)
# Ensure 'Portfolio' and 'Mkt' are aligned by date and NaN rows are dropped

# reg_data = stock_port_ret[['Portfolio', 'Mkt']].dropna()
# Y = reg_data['Portfolio']
# X = reg_data['Mkt']
# X = sm.add_constant(X) # Add intercept term for Alpha

# model = sm.OLS(Y, X)
# results = model.fit()

# print(results.summary())

# alpha = results.params['const'] # Or results.params[0]
# beta = results.params['Mkt']    # Or results.params[1]

# print(f"Alpha: {alpha:.6f}")
# print(f"Beta: {beta:.4f}")
```

    (Code requires market data and proper alignment/cleaning)

- **[✓] Concept: Risk-Return Tradeoff (Definition, Application)**
    - **Content:** The fundamental principle that higher potential returns are associated with higher levels of risk (uncertainty/volatility). Investors use this concept when making investment decisions and assessing portfolios (Week 7 Slides, Slide 14). Visualized in Mean-Variance space (Week 8 Slides).
    - **Python Code/Syntax:** Not Applicable (Conceptual).

- **[✓] Portfolio Basics:**
    - **[✓] Value of portfolio (Definition, Calculation)**
        - **Content:** The total market value of all assets held within the portfolio. For `n` assets with `Ni` shares and price `Pi`: `V = N1*P1 + N2*P2 + ... + Nn*Pn` (Week 7 Slides, Slide 16).

- **[✓] Weights calculation (Definition, Sum=1, Negative/Leverage/Dollar-neutral, Interpretation)**
  - **Content:** The proportion of the total portfolio value invested in each asset `i`. Formula: `wi = (Ni * Pi) / V`. For standard long-only portfolios, weights sum to 1 (`Σwi = 1`). Weights can be > 1 (leverage), negative (short selling), or sum to 0 (dollar-neutral) (Week 7 Slides, Slide 15, 16; Week 8 Slides, Slide 3, 5, 6, 7; Day 5 Summary).
- **[✓] Types of portfolio (Conceptual: Income, Growth, Value)**
  - **Content:** Described based on investment strategy/goal:
    - **Income:** Focuses on steady income (dividends).
    - **Growth:** Invests in high-growth companies (potentially higher risk).
    - **Value:** Seeks undervalued assets/bargains. (Week 7 Slides, Slide 20).
- **[✓] Portfolio Returns (Weighted average formula, matrix notation)**
  - **Content:** The return of a portfolio is the weighted average of the returns of the individual assets within it. Formula: `Rp = w1*R1 + w2*R2 + ... + wn*Rn` (Week 8 Slides, Slide 14, 18).
  - **Python Code/Syntax & Example:**

```
# Assuming stock_log_ret has daily returns for AAPL, AMZN
# portfolio_weights = np.array([0.6, 0.4]) # Example weights 60% AAPL, 40% AMZN

# Calculate daily portfolio return series
portfolio_return_series = stock_log_ret.dot(portfolio_weights)
# Alternative: portfolio_return_series = (stock_log_ret * portfolio_weights).sum(axis=1)

print(portfolio_return_series.head())
```

- **[✓] Matrices: Covariance Matrix & Formula (`.cov()`), Correlation Matrix & Formula (`.corr()`) (Concept, Calculation, role in portfolio risk, annualization, extracting specific values)**
  - **Content:** Essential for calculating portfolio risk.
    - **Covariance:** Measures how two asset returns move together. Positive means they tend to move in the same direction, negative means opposite directions. Formula involves `E[(Rit - μi)(Rjt - μj)]`. The diagonal elements of a covariance matrix are the variances of the individual assets. (Week 7 Slides, Slide 5, 27; Week 8 Slides, Slide 13).
    - **Correlation:** Standardized covariance, ranging from -1 to +1, easier to interpret (Week 7 Slides, Slide 5, 28; Week 8 Slides, Slide 13).
    - **Annualization:** Daily covariance matrix is multiplied by the number of trading days (e.g., 252) (Day 5 Summary). Correlation is unitless and typically not annualized itself.
    - *Quiz/Assignment Feedback Highlight:* Emphasized needing `.loc[row, col]` or `series1.cov(series2)` to extract specific pairwise covariance values from the matrix.
  - **Python Code/Syntax & Example:**

```
# Daily Covariance Matrix
daily_cov_matrix = stock_log_ret.cov()
print("Daily Covariance Matrix:\\n", daily_cov_matrix)

# Daily Correlation Matrix
daily_corr_matrix = stock_log_ret.corr()
print("\\nDaily Correlation Matrix:\\n", daily_corr_matrix)

# Annualized Covariance Matrix
annual_cov_matrix = daily_cov_matrix * 252
print("\\nAnnualized Covariance Matrix:\\n", annual_cov_matrix)

# Extract specific covariance
aapl_amzn_cov = daily_cov_matrix.loc['AAPL', 'AMZN']
print(f"\\nAAPL-AMZN Daily Cov: {aapl_amzn_cov:.8f}")
```

- **[✓] Modern Portfolio Theory (MPT):**
  - **[✓] Expected return of portfolio (Calculation, matrix (`np.dot`))**

        - **Content:** As previously defined, the weighted average of individual asset expected returns. Matrix form: `E[Rp] = w.T @ E[R]` or `np.dot(weights, expected_returns)`. (Day 5 Summary; Week 8 Slides, Slide 18).
        - **Python Code/Syntax:** (See Expected Return section above).
    - **[✓] Variance and Volatility of portfolio (Formula application - matrix form (`np.dot`) crucial, role of covariance, *not* simple weighted average, non-linear graph, annualization) (Critical input error noted in quiz)**
        - **Content:** Portfolio variance depends *critically* on the weights, individual asset variances, *and* the pairwise covariances between all assets. It's NOT a simple weighted average of individual variances (unless correlation is +1). Formula for 2 assets: $\sigma_p^2 = w1^2\sigma1^2 + w2^2\sigma2^2 + 2*w1*w2*Cov12$ (Week 7 Slides, Slide 22; Week 8 Slides, Slide 15). Matrix formula for n assets: $\sigma_p^2 = w.T @ \Sigma @ w$ where Σ is the covariance matrix (Day 5 Summary). Volatility is the square root of variance (σ_p). The relationship between portfolio risk and return (weights) is generally non-linear (bullet shape) due to covariance effects (Week 8 Slides, Slide 23, 28).
        - *Quiz/Assignment Feedback Highlight:* **CRITICAL** error noted was using *mean returns* inside the variance calculation instead of the *covariance matrix*. Formula *must* use the covariance matrix.
        - **Python Code/Syntax & Example (Annualized):**

```
# Assume annual_cov_matrix and portfolio_weights are defined
# portfolio_weights = np.array([0.5, 0.5])
# annual_cov_matrix = stock_log_ret.cov() * 252

# Portfolio Variance (using dot product / matrix multiplication)
portfolio_variance = np.dot(portfolio_weights.T, np.dot(annual_cov_matrix, portfolio_weights))
# Alternative: portfolio_variance = portfolio_weights @ annual_cov_matrix @ portfolio_weights
print(f"Portfolio Annualized Variance: {portfolio_variance:.6f}")

# Portfolio Volatility (Standard Deviation)
portfolio_volatility = np.sqrt(portfolio_variance)
print(f"Portfolio Annualized Volatility: {portfolio_volatility:.4f}")
```

    - **[✓] Correlation of portfolio (Role in MPT, impact on risk, varying correlation effects)**
        - **Content:** Lower correlation between assets significantly reduces portfolio risk for a given expected return. Diversification benefits stem from combining assets that are not perfectly positively correlated (Week 8 Slides, Slide 24, 25, 28, 38).
    - **[✓] Efficient Frontier (Concept, Minimum-Variance Boundary, Shape, Tangency Portfolio, Straight line with risk-free asset, graphical representation)**
        - **Content:** The set of optimal portfolios offering the highest expected return for a defined level of risk (volatility) or the lowest risk for a given level of expected return. Graphically represented as the upper edge of the "bullet-shaped" feasible region (minimum-variance boundary) in mean-standard deviation space. Rational investors choose portfolios on the efficient frontier. When a risk-free asset exists, the efficient frontier becomes a straight line (Capital Market Line) starting from the risk-free rate and tangent to the original risky asset frontier. The point of tangency represents the optimal risky portfolio (Tangency Portfolio) that all investors should hold in combination with the risk-free asset. (Week 8 Slides, Slide 30, 32, 33, 34, 37).
    - **[✓] Sharpe Ratio (Formula, Interpretation, Maximization via Tangency Portfolio, relation to slope)**
        - **Content:** Measures risk-adjusted return (reward-to-volatility). Formula: `Sharpe Ratio = (Rp - Rf) / σp`. Higher is better. The Tangency Portfolio is the portfolio of risky assets with the highest possible Sharpe Ratio. The slope of the Capital Market Line (the line from Rf to the Tangency Portfolio) is equal to the Sharpe Ratio of the Tangency Portfolio. (Week 7 Notebook; Week 8 Slides, Slide 35; Day 5 Summary).
        - **Python Code/Syntax & Example:**

```
# Assume portfolio_ann_ret, portfolio_volatility are calculated
risk_free_rate = 0.01 # Example annual risk-free rate of 1%

sharpe_ratio = (portfolio_ann_ret - risk_free_rate) / portfolio_volatility
print(f"Sharpe Ratio: {sharpe_ratio:.4f}")
```

Week 10: Machine Learning Concepts

- **[✓] Fundamental Concepts:**
    - **[✓] Difference between AI, Machine Learning, Deep Learning (Hierarchy)**

        - **Content:** Explained the relationship between these terms. Artificial Intelligence (AI) is the broad science of building intelligent programs/machines. Machine Learning (ML) is a *subset* of AI that focuses on systems learning from data without explicit programming. Deep Learning (DL) is a *subset* of ML using complex, multi-layered neural networks (often called Neural Nets). (Week 9 Slides, Slide 11; Day 6 Summary, Section 1).
        - **Python Code/Syntax:** Not Applicable (Conceptual Hierarchy).
    - **[✓] Unstructured vs. Structured Data**
        - **Content:** Differentiated between data types based on organization.
            - **Structured Data:** Highly organized, typically in rows and columns (like relational databases or spreadsheets), easily searchable (e.g., numerical data like prices, dates). Estimated ~20% of enterprise data (Gartner). Requires less storage, easier to manage/protect.
            - **Unstructured Data:** Lacks a predefined format or organization (e.g., text documents, emails, images, audio, video, social media posts). Cannot be readily displayed in rows/columns. Estimated ~80% of enterprise data (Gartner). Requires more storage, more difficult to manage/protect/analyze. (Week 9 Slides, Slide 12; Day 6 Summary, Section 1).
            - *Quiz Feedback Highlight:* Raw text is considered unstructured data.
        - **Python Code/Syntax:** Not Applicable (Conceptual Data Types).
    - **[✓] Supervised (Classification, Regression) vs. Unsupervised (Clustering, Dimensionality Reduction) Learning (Definitions, examples, when to use, paradigm)**
        - **Content:** Explained the two main paradigms based on the type of input data:
            - **Supervised Learning:** Uses *labeled* training data (feature/label pairs). The goal is to learn a mapping function (a rule or model) that predicts the label for previously unseen input features. Examples include:
                - *Classification:* Predicts a discrete category label (e.g., spam/not spam, buy/sell/hold, different types of customers). K-nearest-neighbor mentioned as an example method. (Week 9 Slides, Slide 3, 16, 17, 24, 38, 51; Day 6 Summary, Section 1).
                - *Regression:* Predicts a continuous numerical value (e.g., predicting house prices, stock returns).
            - **Unsupervised Learning:** Uses *unlabeled* training data (feature vectors only). The goal is to find hidden patterns or intrinsic structures in the data. Examples include:
                - *Clustering:* Groups similar data points together into "natural clusters" based on some distance/similarity measure. K-means mentioned as an example method. (Week 9 Slides, Slide 3, 16, 17, 19, 38, 40, 51; Day 6 Summary, Section 1).
                - *Dimensionality Reduction:* Reduces the number of features while preserving important information (e.g., PCA - Principal Component Analysis, mentioned in Day 6 Summary, Section 2, though PCA details might belong more to a stats/econometrics context than pure ML intro).
        - **Python Code/Syntax:** Not Applicable (Conceptual Paradigms).
    - **[✓] Features (Definition, input variables, engineering, SNR)**
        - **Content:** Defined features as the input variables or attributes used by models to make predictions or find patterns (e.g., height and weight in the basketball example, GPA/Python experience for grade prediction). Feature representation (how examples are described by feature vectors) is crucial for generalization. "Feature engineering" is the process of selecting/creating useful features. Not all potential features are helpful; some might cause overfitting (e.g., birth date). The goal is to maximize the Signal-to-Noise Ratio (SNR) – the ratio of useful input to irrelevant input. (Week 9 Slides, Slide 3, 17, 30, 31, 39, 43; Day 6 Summary, Section 1).
        - **Python Code/Syntax:** Not Applicable (Conceptual Definition).
    - **[✓] Overfitting (Definition, consequence - poor generalization, relation to model complexity, ethical issues, avoiding via simpler models/tradeoffs)**
        - **Content:** Occurs when a statistical model learns the training data *too well*, fitting exactly or very closely to it, including noise. The model essentially "memorizes" the training set instead of learning the underlying pattern. Consequence: The model performs poorly on new, unseen data (test data) because it cannot generalize. Often related to overly complex models (e.g., complex classifier surfaces). Avoiding overfitting might require tolerating some training errors (trading off false positives/negatives) or choosing simpler models. (Week 9 Slides, Slide 24, 29, 31, 32, 39, 43; Day 6 Summary, Section 1).
        - *Quiz Feedback Highlight:* Overfitting can have ethical implications (e.g., in loan decisions) beyond just being inaccurate.
        - **Python Code/Syntax:** Not Applicable (Conceptual Problem).
- **[✓] Evaluation:**
    - **[✓] Confusion Matrices (TP, FP, TN, FN - Definitions, calculation from examples)**

        - **Content:** A table used to evaluate the performance of a classification model. It summarizes the counts of:
            - **True Positives (TP):** Correctly predicted positive instances.
            - **False Positives (FP):** Incorrectly predicted positive instances (Type I error).
            - **True Negatives (TN):** Correctly predicted negative instances.
            - **False Negatives (FN):** Incorrectly predicted negative instances (Type II error).
            - Examples provided in Week 9 Slides (Slide 47, 48, 49) show how to populate the matrix based on actual vs. predicted classes (e.g., Democrat/Republican voters). Trading off FP vs FN mentioned (Slide 24, 29). (Day 6 Summary, Section 1).
        - **Python Code/Syntax:** Not Applicable (Conceptual Evaluation Tool).
    - **[✓] Metrics: Sensitivity (Recall), Specificity, Positive Predictive Value (PPV)/Precision (Definitions/Formulas, Interpretation, Trade-offs, goal alignment)**
        - **Content:** Key performance metrics derived from the Confusion Matrix:
            - **Sensitivity (Recall or True Positive Rate):** Measures how well the model identifies actual positives. Formula: `TP / (TP + FN)`. (Percentage correctly found).
            - **Specificity (True Negative Rate):** Measures how well the model identifies actual negatives. Formula: `TN / (TN + FP)`. (Percentage correctly rejected).
            - **Positive Predictive Value (PPV) or Precision:** Measures the accuracy of positive predictions. Formula: `TP / (TP + FP)`. (Out of all predicted positives, how many were actually positive?).
            - **Accuracy:** Overall correctness. Formula: `(TP + TN) / (TP + FP + TN + FN)`.
            - Explained the trade-off between Sensitivity and Specificity (increasing one might decrease the other). (Week 9 Slides, Slide 50; Day 6 Summary, Section 1).
            - *Quiz Feedback Highlight:* Emphasized the need for precise interpretation and understanding which metric aligns with specific goals (e.g., minimizing False Negatives requires high Sensitivity; minimizing False Positives relates to high Specificity or Precision depending on context).
        - **Python Code/Syntax:** Not Applicable (Conceptual Evaluation Metrics).
- **[✓] Note: Machine Learning Python demo is *not required.***
    - **Content:** Explicitly stated in the Week 10 introductory slide and reinforced by the lack of implementation code in the summaries/slides for Week 10 topics.

Weeks 11-12: Textual Analysis & LLM Concepts

- **[✓] Textual Analysis:**
    - **[✓] Definition of Textual Analysis (vs. NLP, Goals: insights from text itself, simple examples)**
        - **Content:** Defined as extracting meaningful information from text. Can be simple (extracting specific words/phrases) or complex (extracting latent/hidden patterns like sentiment, content themes, emotion, writer characteristics). Called "text mining" in computer science/engineering and "textual analysis" in accounting/finance. The goal is to derive quality insights *solely from the text or words themselves*, without necessarily considering deeper linguistic structure or semantics. Examples include calculating word frequency, sentence length, or presence/absence of specific keywords. (Week 11 Textual - Part 1, Slide 2, 4; Day 6 Summary, Section 4).
        - **Python Code/Syntax:** While specific analysis examples like sentiment weren't fully coded (Python tools listed conceptually on Wk11-P1 S8), basic text processing related to readability (counting words/sentences) is foundational.
    - **[✓] Definition of NLP (Natural Language Processing) (Goals: understand linguistic use/context, diverse field examples)**
        - **Content:** An emerging field devoted to understanding human language. It's a diverse field within computer science/engineering. The goal is to understand the *linguistic use and context* behind the text, considering grammatical structures and semantics. Examples of NLP tasks include grammar/linguistics analysis, conversation modeling, audio/image-to-text conversion, translation, dictation, text generation, semantic meaning analysis, sentiment analysis, automatic summarization. (Week 11 Textual - Part 1, Slide 3, 4; Day 6 Summary, Section 4).
        - **Python Code/Syntax:** Conceptual definition. Tools like NLTK, SpaCy mentioned for more advanced NLP tasks (Wk11-P1 S8).
    - **[✓] Differences between Textual Analysis and NLP**
        - **Content:** The key difference lies in the depth of analysis. Textual Analysis often focuses on surface-level features (word counts, frequency, presence/absence) to gain insights directly from the text itself, often ignoring semantics. NLP aims for a deeper understanding of linguistic use, context, grammar, and semantics. NLP goals are broader, encompassing tasks like translation,

generation, and semantic interpretation, which go beyond simple text feature extraction. (Week 11 Textual - Part 1, Slide 4; Day 6 Summary, Section 4).

- **Python Code/Syntax:** Not Applicable (Conceptual difference).

- **[✓] Text Data Readability (Concept, Gunning Fog Index measure, calculation using `textstat`, processing PDFs (`PyPDF2`, `pdfminer.six`) for text)**

  - **Content:** Readability is the ease with which a reader can understand a written text. The **Gunning Fog Index** was introduced as a specific measure. It estimates the years of formal education needed to understand the text on the first reading. A score of 6 is easily readable by sixth-graders, around 8 is aimed for the public, 17+ suggests graduate level. Formula involves average sentence length and percentage of "complex" words (three or more syllables). (Week 11 Textual - Part 1, Slide 7; Day 6 Summary, Section 4). Processing text from PDFs is necessary to apply readability measures.
  - **Python Code/Syntax & Example:**
    - **PDF Processing:** Demonstrated using two libraries: `PyPDF2` (page-by-page extraction) and `pdfminer.six` (extracting text from the whole file). Looping through pages is necessary with `PyPDF2` for full-document analysis. Joining text from multiple pages into a single string is required before calculating readability for the whole document.
    - **Readability Calculation:** Used the `textstat` library.

```
# 1. PDF Processing (Example using pdfminer.six - simpler for full text)
# !pip install pdfminer.six
from pdfminer.high_level import extract_text

try:
    # Assuming 'SSRN-id2625975.pdf' is in the same directory
    pdf_text_content = extract_text("SSRN-id2625975.pdf")
    # print(pdf_text_content[:500]) # Print first 500 chars
except FileNotFoundError:
    print("Error: PDF file not found.")
    pdf_text_content = "" # Set to empty string to avoid error later

# 2. Readability Calculation
# !pip install textstat
import textstat

if pdf_text_content: # Check if text was extracted successfully
    readability_score = textstat.gunning_fog(pdf_text_content)
    print(f"Gunning Fog Index: {readability_score:.2f}")
else:
    print("Cannot calculate readability: No text content extracted.")

# Example using PyPDF2 (for page looping - more complex for full doc)
# !pip install PyPDF2==2.12.1 # Specific version might be needed
from PyPDF2 import PdfReader

try:
    pdf_reader = PdfReader("SSRN-id3694637.pdf")
    number_of_pages = len(pdf_reader.pages)
    all_pages_text_list = []
    for i in range(number_of_pages):
        page = pdf_reader.pages[i]
        try:
            all_pages_text_list.append(page.extract_text())
        except Exception as e:
            print(f"Error extracting text from page {i}: {e}") # Handle potential errors per page

    full_doc_text = " ".join(filter(None, all_pages_text_list)) # Join non-empty page texts

    if full_doc_text:
         readability_pypdf2 = textstat.gunning_fog(full_doc_text)
         print(f"Gunning Fog Index (PyPDF2): {readability_pypdf2:.2f}")
    else:
         print("No text extracted using PyPDF2.")

except FileNotFoundError:
     print("Error: PyPDF2 target PDF not found.")
except Exception as e:
     print(f"An error occurred with PyPDF2: {e}")
```

(Code requires libraries installed and PDF files present). (Week 12 Notebook).

- **[✓] Large Language Models (LLM):**
    - **[✓] Definition of LLM (Scale: Parameters, Data; relation to GenAI, specific examples like Llama-2)**

- **Content:** Large Language Models are a type of Generative AI (GenAI). Defined by their large scale, particularly in terms of:
  - **Parameters:** Billions of weights in the underlying neural network (e.g., Llama-2 series: 7b, 13b, 34b, 70b parameters).
  - **Data:** Trained on massive amounts of text data (e.g., ~10TB for ChatGPT pre-training).
  - Examples mentioned: Llama-2 (Meta), ChatGPT series (OpenAI), Claude (Anthropic), Gemini (Google), Command R (Cohere). LLM Leaderboards track performance. (Week 11 Textual - Part 1, Slide 9, 10; Week 11 Textual - Part 2, Slide 2, 3, 16, 17, 18; Day 6 Summary, Section 4).
- **Python Code/Syntax:** Not Applicable (Conceptual Definition).

- **[✓] Parameters (Concept: weights storing knowledge, scale, how obtained via training, conceptual explanation)**

  - **Content:** Parameters are the adjustable weights within the LLM's neural network. Conceptually, these billions of parameters store the "knowledge" learned from the vast training data, enabling the model to predict sequences (like next word). They are obtained through a computationally intensive *model training* process (distinct from easier *model inference*). Training involves optimizing/adjusting these parameters iteratively (based on predicting text from the training data) using large GPU clusters over extended periods (e.g., 6000 GPUs, 12 days, ~$2M for Llama-2 70b pre-training). The exact function of individual parameters within the massive network is generally not well understood. (Week 11 Textual - Part 1, Slide 9, 10, 11, 13; Day 6 Summary, Section 4).
  - **Python Code/Syntax:** Not Applicable (Conceptual Explanation).

- **[✓] Neural Network (Basic Concept: structure (layers, nodes), function (next word prediction), parameters dispersed, training linked to compression, Transformer architecture mentioned)**

  - **Content:** The underlying computational structure of LLMs. Visually represented as interconnected nodes in layers (input, hidden, output). Key function in LLMs is **next-word prediction**. Billions of parameters are dispersed throughout the network and iteratively adjusted during training to improve this prediction. Training is conceptually linked to compressing the information from the training dataset (like the internet) into the network's parameters. The **Transformer NN architecture** is specifically mentioned as common for modern LLMs. (Week 11 Textual - Part 1, Slide 9, 11, 13; Week 11 Textual - Part 2, Slide 2; Supplementary Extraction).
  - **Python Code/Syntax:** Not Applicable (Conceptual Explanation).

- **[✓] Steps to train a GenAI (Conceptual Overview: Pre-training/Base Model -> Fine-tuning/Assistant Model)**

  - **Content:** Explained the typical two-stage training process for models like ChatGPT:
    - **Stage 1: Pre-training:** Train a **Base Model** on a massive, general dataset (e.g., ~10TB of internet text) primarily to learn language structure, facts, and reasoning ability through next-word prediction. This is computationally very expensive and time-consuming (e.g., weeks/months, millions of dollars).
    - **Stage 2: Fine-tuning:** Take the pre-trained Base Model and further train it on a smaller, high-quality dataset specific to the desired behavior (e.g., instruction following, conversation). Often involves supervised fine-tuning on curated Q&A pairs, comparisons, or feedback (like RLHF - Reinforcement Learning from Human Feedback) to create an **Assistant Model** aligned with user intent. This stage is much faster (e.g., ~1 day). The process involves steps like writing labeling instructions, collecting high-quality data, fine-tuning, evaluating, deploying, and monitoring/fixing misbehaviors. (Week 11 Textual - Part 2, Slide 2, 14, 15, 16; Day 6 Summary, Section 4).
  - **Python Code/Syntax:** Not Applicable (Conceptual Process).