Session 1.1: Business Analytics Architecture, Google Antigravity IDE Initialization, and Core Arithmetic

PEDAGOGICAL AGENDA

1:2 Delivery Rhythm

Session Learning Architecture

Structured 3-hour applied computing session designed for business students with zero prior coding experience.

- **Phase 1 (60 Min Lecture):** Computational foundations, business vocabulary, and syntactic mechanics.
- **Phase 2 (120 Min Lab):** Guided hands-on implementation, error diagnosis, and AI co-pilot resolution.
- **Zero Fluff Mandate:** Focus directly on executable Python code, tabular data wrangling, and commercial intelligence.

STRATEGIC OUTCOME

Operational Autonomy

Business Analytical Competency

Transforming passive spreadsheet users into autonomous programmatic decision makers.

- **Speed & Scale:** Processing datasets exceeding spreadsheet limits with instantaneous vectorized execution.
- **Repeatable Pipelines:** Replacing manual copy-paste workflows with fully automated Python scripts.
- **AI Debugging Literacy:** Leveraging modern IDEs to diagnose and resolve errors independently.

Core Computational Keywords & Business Meaning

CONCEPT 01 — Core Keyword

Python Interpreter

The computational engine that reads Python source code and executes instructions sequentially line-by-line in real time without requiring compilation.

CONCEPT 02 — Core Keyword

Variable Assignment

Storing data in computer memory using an assignment operator (`=`) and an intuitive descriptive label (such as `net_sales = 574785.0`), replacing fragile cell coordinates like `C13`.

CONCEPT 03 — Core Keyword

Primitive Data Types

The foundational categories of information processed by Python: whole numbers (`int`), decimal numbers (`float`), and text strings (`str`).

CONCEPT 04 — Core Keyword

Traceback & Exception

The automatic diagnostic navigation report generated by Python when an instruction violates a syntax or runtime rule, pinpointing the exact file and line number.

Computational Foundations & Business Operations

PILLAR 01 Architecture

**Core Programming & Computational Foundations

**

PILLAR 02 Architecture

Sequential Execution & Program State

Python executes code strictly line-by-line from top to bottom. Unlike spreadsheets where formulas across sheets can trigger circular reference loops, Python code maintains a clear, linear flow where each statement computes or alters memory state sequentially.

PILLAR 03 Architecture

Cell Coordinate Formulas vs. Descriptive Variables

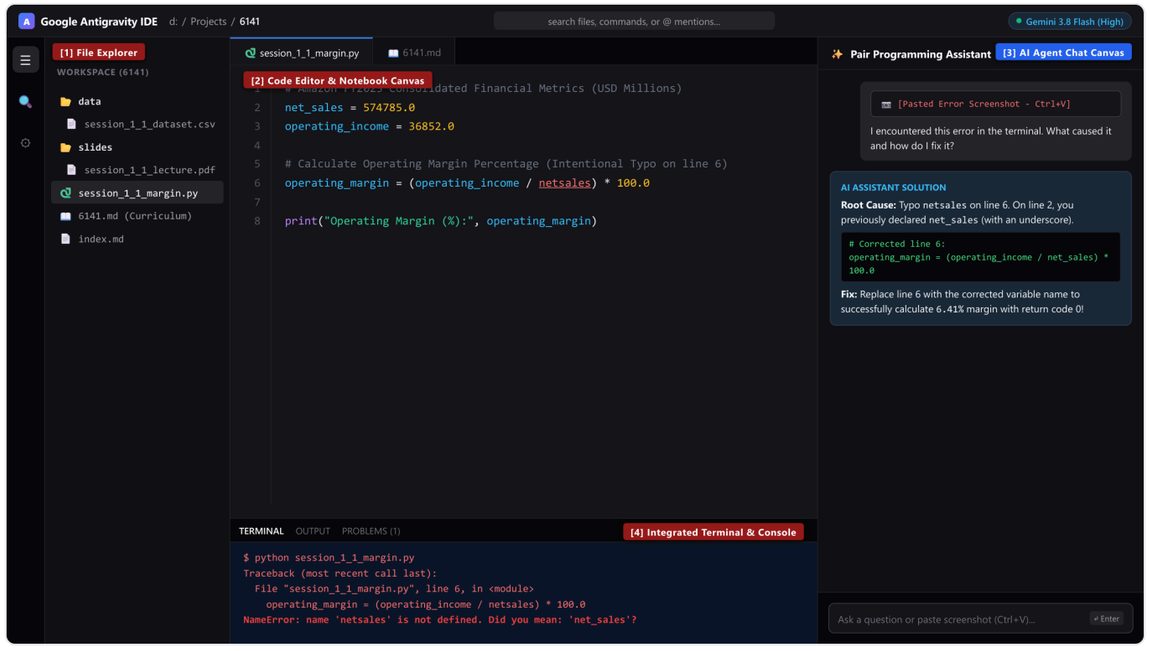
Transitioning from Excel spatial grid thinking (such as `=C13-E13`) to self-documenting computational recipes (`operating_expense = net_sales - operating_income`). Code variables retain clear business meaning and eliminate silent formula drift.

PILLAR 04 Architecture

Interpreter Diagnostics & Traceback Mechanics

Python acts as an interactive assistant. When a syntax or naming rule is violated, Python pauses and outputs a Traceback pointing to the exact line number and cause, turning errors into immediate learning signals rather than system failures.

Google Antigravity IDE Architecture & AI Debugging Protocol



[1] File Explorer

Project Hierarchy

Manage datasets in `data/`, scripts in root, and exported artifacts.

[2] Code Editor

Syntax Workspace

Modern syntax highlighting, lint diagnostics, and line-by-line script authoring.

[3] AI Agent Chat

AI Pair Programmer

Paste error screenshots directly to inspect root causes and get verified fixes.

[4] Terminal

Execution Engine

Run Python scripts (`python script.py`) and view console logs and tracebacks.

Zero-Friction AI Error Resolution Protocol

1. **Capture:** Press `Win + Shift + S` to clip terminal error traceback.
2. **Paste:** Press `Ctrl + V` into AI Chat [3] with "Explain root cause & fix".
3. **Apply:** Review the explanation and apply verified fix into Editor [2].

Script Demonstration 1: Script Demonstration 1

session_1_1_demo_1.py — Python 3.10

```python
# Amazon Consolidated Financial Metrics "kw">class="kw">for FY2023 ("kw">class
    net_sales = 574785.0
    operating_income = 36852.0

    # Calculate Operating Expenses
    operating_expense = net_sales - operating_income

    # Calculate Operating Margin Percentage
    operating_margin_pct = (operating_income / net_sales) * 100.0

    "fn">print("Operating Expense (USD M):", operating_expense)
    "fn">print("Operating Margin (%):", "fn">round(operating_margin_pct, 2))
```

```
$ python session_1_1_demo_1.py [Exit 0]
```

1 Syntactic Structure

Importing production libraries and defining clean analytical data structures.

2 Vectorized Transformation

Applying deterministic operations across the full dataset without fragile cell references.

3 Console Verification

Emitting formatted business KPIs to the terminal for auditability and compliance.

Script Demonstration 2: Script Demonstration 2

session_1_1_demo_2.py — Python 3.10

```python
aws_sales = 90757.0
    aws_income = 24631.0
    aws_margin_pct = (aws_income / aws_sales) * 100.0
    "fn">print("AWS Operating Margin (%):", "fn">round(aws_margin_pct, 2))
```

```
$ python session_1_1_demo_2.py [Exit 0]
```

1 Syntactic Structure

Importing production libraries and defining clean analytical data structures.

2 Vectorized Transformation

Applying deterministic operations across the full dataset without fragile cell references.

3 Console Verification

Emitting formatted business KPIs to the terminal for auditability and compliance.

Script Demonstration 3: Code Pipeline Step 3

session_1_1_demo_3.py — Python 3.10

```python
# Standard verification script
"kw">class="kw">import pandas "kw">class="kw">as pd
"fn">print('Execution verified')
```

```
$ python session_1_1_demo_3.py [Exit 0]
```

1 Syntactic Structure

Importing production libraries and defining clean analytical data structures.

2 Vectorized Transformation

Applying deterministic operations across the full dataset without fragile cell references.

3 Console Verification

Emitting formatted business KPIs to the terminal for auditability and compliance.

Interactive Two-Tier Lab Protocol & AI Diagnosis

EXERCISE 1

Guided Lab

Foundational Implementation

Step-by-step guided practice establishing baseline syntax mastery in Antigravity IDE.

- **Step 1:** Ingest dataset from `data/session_1_1_dataset.csv`.
- **Step 2:** Implement core analytical functions and compute primary business metrics.
- **Step 3:** Format console outputs and verify calculations against baseline ledger totals.
- **Step 4:** Run in Terminal [4] and confirm clean execution with exit code 0.

EXERCISE 2+

Coding Challenge

Advanced Scripting & AI Debugging

Applied programming challenge expanding pipeline logic and resolving intentional exceptions.

- **Task 1 (Pipeline Expansion):** Add secondary business unit logic and multi-tier calculations.
- **Task 2 (Deliberate Error & AI Capture):** Introduce intentional syntax or type error, capture traceback with `Win + Shift + S`, and diagnose via AI Chat [3].
- **Task 3 (Verified Production Run):** Apply verified fix, export audit output, and confirm zero errors with return code 0.

Interactive Lab Protocol & Delivery Rhythm

Hands-On Code Execution

The Pedagogical Contract

Computational mastery develops through active problem-solving. By creating intentional errors and using AI co-pilots to resolve them, students build lifelong debugging autonomy.