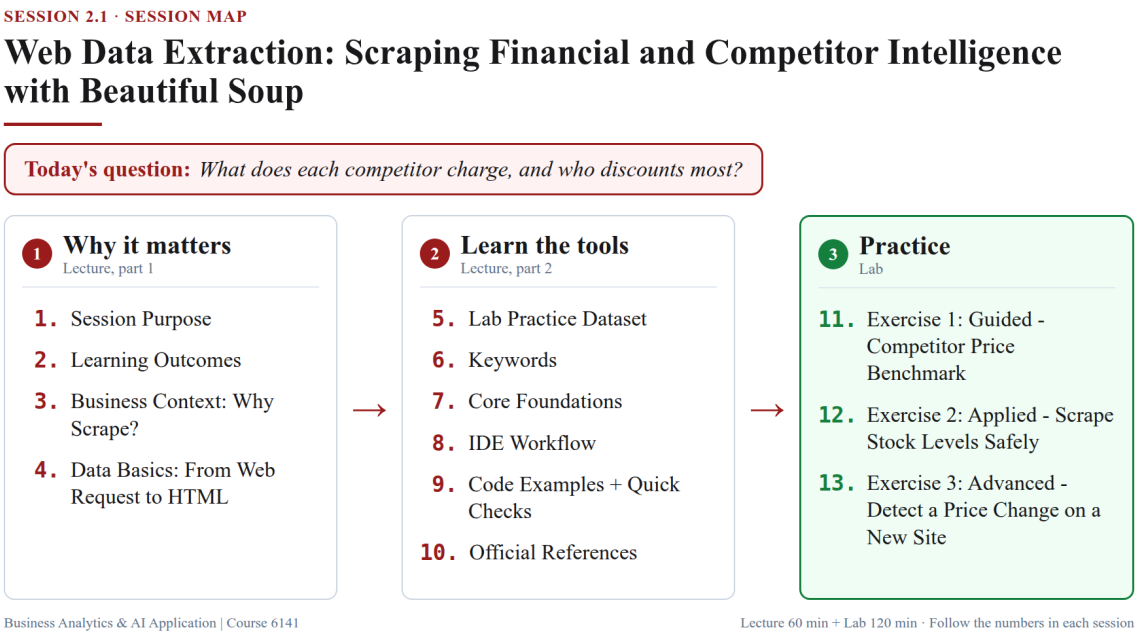
Session 2.1: Web Data Extraction: Scraping Financial and Competitor Intelligence with Beautiful Soup

Course 6141: Business Analytics & AI Application | Reading for Week 5



Schedule: Week 5 | In-person classes (1:2 Delivery Ratio: 60 Min Lecture + 120 Min Lab)

1. Session Purpose

- Our pricing team watches 3 competitor websites, where prices sit on web pages, not in CSV files
- Copying hundreds of prices by hand is slow and error-prone
- The pricing manager asks: what does each competitor charge, and who discounts most? Today, Python reads a web page and turns it into a table

2. Expected Learning Outcomes (By the end of this session, you will be able to)

1. "How does a web page reach my computer?": explain an HTTP (HyperText Transfer Protocol) GET request and read HTML (HyperText Markup Language: the text with tags that describes a web page)
2. "What is BetaMart charging?": extract prices with Beautiful Soup (a Python library that reads HTML) and CSS (Cascading Style Sheets) selectors (an address for tags, such as `.price`)

3. “What if a product has no stock tag?”: handle a missing tag (the search returns `None`, meaning nothing found) and fix an `AttributeError`

3. Business Context: Why Scrape?

- By hand: open each competitor page, copy each price into Excel, and do it all again tomorrow. 3 sites x 100 products = 300 copy-pastes a day, with typos.
- With a Python script: read every product on the page in seconds, get the same result every run, and run it again every morning. The output is a clean table, ready for pandas.
- Rules of the road (scrape responsibly):
  - Public pages only, never behind a login
  - Check the site’s `robots.txt` and terms of use. `robots.txt` is a short public file on a website (for example, `betamart.example/robots.txt`) that lists which pages automated scripts may and may not read.
  - Pause between requests, for example `time.sleep(2)` (wait 2 seconds), so the site is not overloaded
  - Name your script in the request. The User-Agent is a short label sent with every request that tells the site who is asking.

4. Data Basics: From Web Request to HTML

- 1. Your browser or Python script sends an HTTP GET request: “send me this page”.
- 2. The web server answers with the page as HTML text.
- The browser draws that text as a page for people; Python reads the text itself. In the example below, `data-sku` stores the product’s SKU (Stock Keeping Unit), the store’s product code:

```
<div class="product" data-sku="SKU-801">
  <h3 class="name">Office Desk Chair</h3>
  <span class="price">$49.50</span>
  <span class="stock">120 units</span>
</div>
```

5. Lab Practice Dataset

- Saved web page: `data/session_02_01_catalog.html`, a fictional BetaMart catalog page with 4 product cards (SKU-807 has no stock tag).
- File Link: `data/session_02_01_dataset.csv`
- Audited 12-row competitor pricing dataset compiled from automated extraction (competitor names are fictional):
  - `Product_SKU`: Stock Keeping Unit, the product code (e.g., `SKU-801`).
  - `Competitor_Name`: Competitor (AlphaRetail, BetaMart, GammaGlobal).
  - `Category_Name`: Merchandise category (Electronics, Office Supplies, Hardware, Appliances).
  - `List_Price_USD`: Extracted competitor catalog price in U.S. Dollars.
  - `Promotion_Discount_Pct`: Advertised promotional markdown percentage.
  - `Stock_Availability_Units`: Available warehouse stock units.

6. Keywords (Core Programming Concepts)

- `HTTP GET`:
  - What: a request, using the HyperText Transfer Protocol (HTTP), that asks a web server for a page
  - Why: every page your browser or script reads starts with one
- `HTML Tag`:
  - What: a label around content, such as `<span class="price">`
  - Why: tags tell you where each price or name sits
- `DOM Tree`:
  - What: Document Object Model: tags nested inside tags, like folders
  - Why: one product card holds its name, price, and stock
- `Beautiful Soup (bs4)`:
  - What: a Python library that reads HTML into a searchable tree
  - Why: find tags by name or class in one line
- `CSS Selector`:
  - What: a short address for tags: `.price` = class "price"
  - Why: `.select()` / `.select_one()` use it to pick tags
- `Ethical Scraping`:
  - What: follow `robots.txt`, pause between requests, stay on public pages
  - Why: protects the website and your company

7. Core Programming & Computational Foundations

- **7.1 A web page is text with tags, not a table**
  - *The idea:* When you open a web page, your browser receives a long piece of text written in HTML and draws it as a page. The prices you see are inside that text, wrapped in tags such as `<span class="price">$49.50</span>`. A tag has three parts: an opening tag (`<span ...>`), the content (`$49.50`), and a closing tag (`</span>`). The `class="price"` part is a label that the website's designers put on the tag, and it is what we use to find it.
  - *See it on real data:* one product card from our saved BetaMart page.

```
<div class="product" data-sku="SKU-801">
  <h3 class="name">Office Desk Chair</h3>
  <span class="price">$49.50</span>
  <span class="stock">120 units</span>
</div>
```

What we want	Where it is in the HTML
Product code	the `data-sku` setting of the `div` tag: `SKU-801`

What we want	Where it is in the HTML
Name	inside the `h3` tag with class `name`: `Office Desk Chair`
Price	inside the `span` tag with class `price`: `$49.50`
Stock	inside the `span` tag with class `stock`: `120 units`

  - *More examples:*

Code	Result	What it tells you
`<h3 class="name">Office Desk Chair</h3>`	tag `h3`, class `name`, content `Office Desk Chair`	the product name
`<span class="stock">120 units</span>`	tag `span`, class `stock`, content `120 units`	the stock level, as text
`<div class="product" data-sku="SKU-801">`	tag `div`, class `product`, setting `data-sku`	the card that holds one product; its code is a setting, not content
`<h1>BetaMart Weekly Prices</h1>`	tag `h1`, no class, content `BetaMart Weekly Prices`	the page heading

  - *Remember:* scraping means finding the right tags and taking the text inside them.

- **7.2 Tags inside tags: the DOM tree**

  - *The idea:* Tags are nested: the `div` card contains the name, price, and stock tags, just as a folder contains files. This nesting is called the DOM tree (Document Object Model). Beautiful Soup reads the HTML text and builds this tree, so you can ask for "the price tag inside this card" instead of searching through raw text.
  - *Why nesting matters:* our page has 4 cards, and each one has its own price. If you search the whole page for `.price`, you get all 4 prices. If you first take one card and then search inside it, you get exactly that card's price, already paired with the right name.
  - *See it on real data:*

Search	Result
`soup.select(".price")` (whole page)	4 tags: `$49.50`, `$320.00`, `$429.00`, `$195.00`
`card.select_one(".price")` (one card)	1 tag: `$49.50`, for `SKU-801` only

  - *More examples:*

Code	Result	What it tells you
`soup.select(".stock")`	3 tags: `120 units`, `38 units`, `15 units`	whole page: only 3, because one card has no stock tag
`soup.select(".product")[1].find("h3").text`	`Wireless Router`	the name inside the second card (position `1`)

Code	Result	What it tells you
`soup.select(".product")[3]["data-sku"]`	`SKU-810`	the code of the fourth card
`soup.select_one(".product .stock").text`	`120 units`	a space in a selector means "inside": a stock tag inside a product card

  - *Remember:* loop over the cards, then search inside each card; that keeps every value with its own product.

- **7.3 Four ways to find tags**

Method	Finds	Returns	Example
`.find("h3")`	the first tag with that name	one tag	`Office Desk Chair`
`.find_all("h3")`	every tag with that name	a list of tags	4 product names
`.select_one(".price")`	the first tag matching a CSS selector	one tag	`$49.50`
`.select(".price")`	every tag matching a CSS selector	a list of tags	4 prices

  - *The idea:* `find` methods search by tag name; `select` methods use CSS selectors (Cascading Style Sheets), the same short addresses web designers use: `.price` means "class `price`", and `div.product` means "a `div` tag with class `product`".
  - *Getting the content:* a tag is not yet text. `.text` takes the words inside it (`"$49.50"`), and `tag["data-sku"]` reads a setting written inside the opening tag (`"SKU-801"`). `.strip()` removes spaces and line breaks around the text.
  - *More examples:*

Code	Result	What it tells you
`soup.find("span").text`	`$49.50`	the first `span` on the page happens to be a price
`len(soup.find_all("span"))`	`7`	4 price tags + 3 stock tags
`len(soup.select("div.product"))`	`4`	the number of product cards
`soup.select_one(".discount")`	`None`	no tag on the page has the class `discount`
`soup.h1.text`	`BetaMart Weekly Prices`	a short way to get the first tag of a kind

  - *Remember:* one tag: `find` / `select_one`; all tags: `find_all` / `select`; then `.text` for the words.

- **7.4 Text is not a number yet**
  - *The idea:* Everything scraped is text, even prices. `"$49.50"` cannot be added or averaged. Two steps turn it into a number: remove the dollar sign with `.replace("$", "")`, which gives `"49.50"`, then convert it with `float()`, which gives `49.5`.
  - *Common mistake:* forgetting the conversion. Text "added" together is joined, not summed, and `float("$49.50")` with the `$` still there raises `ValueError: could not convert string to float`.
  - *More examples:*

Code	Result	What it tells you
`float("320.00")`	`320.0`	clean text converts directly
`float("$1,299.99".replace("$", "").replace(",", ""))`	`1299.99`	remove every symbol first: the `$` and the thousands comma
`int("120 units".split()[0])`	`120`	`.split()` cuts the text at spaces; `[0]` takes the first piece, `"120"`
`"49.5" + "10"`	`'49.510'`	text is joined, not added
`float("49.5") + 10`	`59.5`	after conversion, the math works

  - *Remember:* clean the text first, then `float()`.

- **7.5 Select by class name, not by position**
  - *The idea:* You could tell a script "take the 2nd tag in each card". That works until the website adds something, such as a red "SALE" badge before the price; then the 2nd tag is the badge and the script quietly collects the word "SALE". Selecting by class name (`.price`) follows the label, not the position, so it keeps working when the layout changes.
  - *More examples:* the same card before and after the website adds a "SALE" badge in front of the price.

Code	Before the badge	After the badge	What it tells you
`card.find_all("span")[0].text`	`$49.50`	`SALE`	"the first `span`" now points at the badge
`card.select_one(".price").text`	`$49.50`	`$44.50`	the class name still finds the price

  - *Remember:* use class names; still recheck the script whenever the site is redesigned, because class names can change too.

- **7.6 When a tag is missing: `None`**

– *The idea:* Real pages are irregular. On our page, the card for `SKU-807` has no stock tag. When `.select_one(".stock")` finds nothing, it returns `None`, Python's word for "nothing". Asking for the text of nothing, `None.text`, stops the script with `AttributeError: 'NoneType' object has no attribute 'text'`.

– *The fix:* store the result first, check it, then read it:

```
stock_tag = card.select_one(".stock")
if stock_tag is not None:
    stock = stock_tag.text.strip()
else:
    stock = "not shown"
```

– *More examples:*

Code	Result	What it tells you
`soup.select(".product")[2].select_one(".stock")`	`None`	the third card (`SKU-807`) has no stock tag
`[c["data-sku"] for c in soup.select(".product") if c.select_one(".stock") is None]`	`['SKU-807']`	list every card with a missing stock tag
`soup.select(".product")[0].select_one(".stock").text`	`120 units`	the first card has one, so `.text` works

– *Remember:* any tag might be missing; check for `None` before you read `.text`.

8. Development Workflow in Google Antigravity IDE

- `[1]` File Explorer: create your script with your name and the session code (for example, `Reacher_S21_demo_1.py`), and check the files are in `data/`.
- `[2]` Code Editor: write a few lines, run, repeat. Small steps keep errors small.
- `[3]` AI Agent Chat: on an error, capture the traceback (Python's error report) with `Win + Shift + S` (macOS: `Cmd + Shift + 4`), paste it with `Ctrl + V`, and ask why it failed.
- `[4]` Terminal: run `python` followed by your script name. Return code `0` means the script finished with no errors. First time only: `pip install beautifulsoup4`.

9. Hands-On Coding Micro-Examples

- Every example has three parts: **1. Input**, **2. Process**, and **3. Output**. Name each file with your own name and the session code, for example `Reacher_S21_demo_1.py` (pattern: `YourName_S21_demo_1.py`).
- First time only: install Beautiful Soup in the Terminal with `pip install beautifulsoup4`.

- The examples read a saved copy of a fictional competitor page, `data/session_02_01_catalog.html`, so they work the same in class without internet access. On a live site, `requests.get(url).text` returns the same kind of HTML text.
- *Micro-Example 1 (Read a Web Page with Beautiful Soup):* `YourName_S21_demo_1.py`

```
# 1. Input: read the saved web page (HTML text)
from bs4 import BeautifulSoup
html_text = open("data/session_02_01_catalog.html").read()

# 2. Process: build the tree, then find the first product name
soup = BeautifulSoup(html_text, "html.parser")
first_name = soup.find("h3").text

# 3. Output: show what we found
print("Page title:", soup.title.text)
print("First product:", first_name)
```

  - *How it works, line by line:*
    - `from bs4 import BeautifulSoup` loads Beautiful Soup. The library is installed as `beautifulsoup4` but imported as `bs4`.
    - `open("data/...html").read()` opens the saved page and reads the whole file as one long piece of text, stored in `html_text`. On a live website, `requests.get(url).text` would give you the same kind of text.
    - `BeautifulSoup(html_text, "html.parser")` reads that text and builds the DOM tree (section 7.2), stored in `soup`. `"html.parser"` is the HTML reader that comes with Python.
    - `soup.find("h3")` returns the first `h3` tag on the page, and `.text` takes the words inside it.
    - `soup.title.text` reads the page's title, the text shown on the browser tab.
  - *Read the output:*

```
Page title: BetaMart Online Catalog (fictional)
First product: Office Desk Chair
```

    - The title confirms that the script read the right page.
    - Only one name is printed, because `.find()` stops at the first match; Quick Check 1 counts all of them.

- *Quick Check 1 (`.find_all()`):*
  - New tool: `.find_all()` returns every matching tag as a list; `.find()` returns only the first. Why: count how many products the page shows.
  - Task: add one line to part **3. Output**: `print("Products on page:", len(soup._____("h3")))`
  - Check: the Terminal shows `Products on page: 4`.

- *Micro-Example 2 (Turn Product Cards into Rows):* `YourName_S21_demo_2.py`

```
# 1. Input: read and parse the saved web page
from bs4 import BeautifulSoup
html_text = open("data/session_02_01_catalog.html").read()
soup = BeautifulSoup(html_text, "html.parser")

# 2. Process: one dictionary per product card
products = []
for card in soup.select(".product"):
    price_text = card.select_one(".price").text
    products.append({
        "sku": card["data-sku"],
        "name": card.select_one(".name").text.strip(),
        "price_usd": float(price_text.replace("$", "")),
    })

# 3. Output: one line per product
for item in products:
    print(item)
```

  - *How it works, line by line:*
    - `products = []` starts an empty list that will collect one dictionary per product.
    - `for card in soup.select(".product"):` repeats the indented lines once for each product card; on each pass, `card` is one card (section 7.2).
    - `card.select_one(".price").text` finds the price tag **inside this card** and takes its text, such as `"$49.50"`.
    - `products.append({...})` adds one dictionary to the list. A dictionary pairs names with values: `"sku"` with `card["data-sku"]`, `"name"` with the cleaned name, and `"price_usd"` with the price as a number.
    - `float(price_text.replace("$", ""))` removes the dollar sign and converts the text to a number (section 7.4).
    - The last loop prints each dictionary on its own line.
  - *Read the output:*

```
{'sku': 'SKU-801', 'name': 'Office Desk Chair', 'price_usd': 49.5}
{'sku': 'SKU-804', 'name': 'Wireless Router', 'price_usd': 320.0}
{'sku': 'SKU-807', 'name': 'Compact Refrigerator', 'price_usd': 429.0}
{'sku': 'SKU-810', 'name': 'Cordless Drill', 'price_usd': 195.0}
```

    - 4 cards became 4 rows, each with its code, name, and price kept together.
    - `49.5` has no quotation marks: it is now a number. `pd.DataFrame(products)` would turn this list straight into a table with columns `sku`, `name`, and `price_usd`.

- *Quick Check 2 (`.get_text(strip=True)`):*
  - New tool: `.get_text(strip=True)` returns the words inside a tag, with spaces and line breaks removed from both ends. Why: one call does the work of `.text.strip()`.
  - Task: in part **2. Process**, change `"name": card.select_one(".name").text.strip(),` to `"name": card.select_one(".name").________(strip=True),`

  - Check: the output does not change; the first line is still `{'sku': 'SKU-801', 'name': 'Office Desk Chair', 'price_usd': 49.5}`.

- *Micro-Example 3 (From Scraped Prices to a Comparison Table):* `YourName_S21_demo_3.py`

```
# 1. Input: prices already scraped from 3 competitor sites
import pandas as pd
df = pd.read_csv("data/session_02_01_dataset.csv")

# 2. Process: price after the advertised discount
df["Effective_Price_USD"] = (
    df["List_Price_USD"] * (1 - df["Promotion_Discount_Pct"] / 100)
).round(2)
electronics = df.loc[df["Category_Name"] == "Electronics"]

# 3. Output: compare competitors, then save the table
print(electronics[["Competitor_Name", "List_Price_USD",
                   "Effective_Price_USD"]])
electronics.to_csv("data/electronics_prices.csv", index=False)
```

  - *How it works, line by line:*
    - The CSV holds prices already scraped from 3 competitor sites, one row per product.
    - `df["List_Price_USD"] * (1 - df["Promotion_Discount_Pct"] / 100)` calculates the price a customer actually pays after the advertised discount, for all 12 rows at once. Example: `299.99 x (1 - 10 / 100) = 269.99`.
    - `.round(2)` keeps 2 decimals, as for money.
    - `df.loc[df["Category_Name"] == "Electronics"]` keeps the 3 Electronics rows, one per competitor.
    - `electronics.to_csv("data/electronics_prices.csv", index=False)` saves those rows as a new CSV file; `index=False` leaves out the row labels, which are not data.
  - *Read the output:*

```
  Competitor_Name  List_Price_USD  Effective_Price_USD
0     AlphaRetail          299.99               269.99
4        BetaMart          320.00               281.60
8     GammaGlobal          310.00               263.50
```

    - AlphaRetail has the lowest list price, but GammaGlobal is cheapest after its 15% discount (`263.50`).
    - A comparison on list prices alone would pick the wrong competitor: always compare the price the customer actually pays.

- *Quick Check 3 (`.min()`):*
  - New tool: `.min()` returns the smallest value in a column; `.max()` returns the largest. Why: answers "what is the lowest competitor price?" in one step.
  - Task: add two lines to part **3. Output**: `lowest = electronics["Effective_Price_USD"].__ _()` and `print("Lowest price:", lowest)`
  - Check: the Terminal shows `Lowest price: 263.5` (GammaGlobal).

10. Official Python Documentation & References

- Beautiful Soup 4 Official Documentation.
- Requests: HTTP for Humans Official Documentation.
- World Wide Web Consortium (W3C) HTML5 Standards.

11. Exercise 1: Guided Practice - Competitor Price Benchmark

- *Step 1: Open Script:* In Google Antigravity IDE, create a new script named `YourName_S21_benchmark.py`, replacing `YourName` with your own name (for example, `Reacher_S21_benchmark.py`).
- *Step 2: Benchmark Competitors by Category:* Type the following code:

```
# 1. Input: prices scraped from 3 competitor sites
import pandas as pd
df = pd.read_csv("data/session_02_01_dataset.csv")

# 2. Process: effective price, then average by category and competitor
df["Effective_Price_USD"] = (
    df["List_Price_USD"] * (1 - df["Promotion_Discount_Pct"] / 100)
).round(2)
benchmark = df.groupby(["Category_Name", "Competitor_Name"]).agg(
    Avg_Discount_Pct=("Promotion_Discount_Pct", "mean"),
    Avg_Effective_Price=("Effective_Price_USD", "mean"),
)

# 3. Output: one row per category and competitor
print("Competitor Category Pricing Benchmark:\n", benchmark)
```

- *Step 3: Analyze Business Insights:* In Electronics and Hardware, find the competitor with the deepest discount and the competitor with the lowest effective price. Are they the same?
- *Step 4: Execute in Terminal:* Run `python YourName_S21_benchmark.py` in Panel `[4]` (Terminal) and confirm return code `0`.
- *Success Check:* Electronics: GammaGlobal has both the deepest discount (15%) and the lowest price (`263.5`). Hardware: GammaGlobal has the deepest discount (15%), but AlphaRetail has the lowest price (`157.5`). The deepest discount is not always the lowest price.

12. Exercise 2: Applied Practice - Scrape Stock Levels Safely

- *Coding Challenge:* In `YourName_S21_practice.py`, extend Micro-Example 2 so each row also has the stock level.
- *Task 1 (Add the Stock Field):* Inside the loop, add `"stock": card.select_one(".stock").text.strip()` to the dictionary.

- *Task 2 (Read the AttributeError & AI Diagnosis):* Run the script. The SKU-807 card has no stock tag, so `.select_one(".stock")` returns `None` and Python stops with `AttributeError: 'NoneType' object has no attribute 'text'`. Capture the traceback with `Win + Shift + S`, paste it into Panel `[3]` (AI Agent Chat) with `Ctrl + V`, and review the AI's explanation.
- *Task 3 (Safe Extraction and Export):* Save the tag first with `stock_tag = card.select_one(".stock")`. If `stock_tag is not None`, use `stock_tag.text.strip()`; else use `"not shown"`. Save the table with `pd.DataFrame(products).to_csv("data/competitor_price_audit.csv", index=False)`.
- *Success Check:* the CSV file has 4 rows, SKU-807 shows `not shown`, and the script ends with return code `0`.

13. Exercise 3: Advanced Practice - Detect a Price Change on a New Site

- *Goal:* Scrape a second competitor page that uses **different tags**, compare it with last week's data in the CSV, and report which product's discount changed.
- *What you need:*
  - Saved web page: `data/session_02_01_gamma.html`, a fictional GammaGlobal page with 4 products.
  - Last week's data: `data/session_02_01_dataset.csv` (use only the rows where `Competitor_Name` is `"GammaGlobal"`).
  - Script name: `YourName_S21_challenge.py`.
  - Tools you already know: `BeautifulSoup`, `.select()`, `.select_one()`, `None` checks, `float()`, `pd.DataFrame()`, `pd.merge()`, and `.to_csv()`.
- *How the GammaGlobal page is built (open the file in the Code Editor to see it):*
  - Each product is an `li` tag with class `item`, and its SKU is stored in `data-sku`.
  - The product name is in a tag with class `title`.
  - Every product has a `list-price` tag, such as `$189.00`.
  - A product **on sale** also has a `sale-price` tag. A product without a sale has **no** `sale-price` tag.
- *Requirements (follow them in order):*
  1. For each product, collect `sku`, `name`, `List_Price_Web` (the list price as a number), and `Current_Price_Web`: the sale price if the `sale-price` tag exists, otherwise the list price.
  2. Add `Discount_Pct_Web` = (`List_Price_Web` - `Current_Price_Web`) / `List_Price_Web` x 100, rounded to 1 decimal place.
  3. Turn the list into a DataFrame. Merge it with the GammaGlobal rows of the CSV, matching `sku` on the web side with `Product_SKU` on the CSV side.
  4. Add `Discount_Changed` = `True` when `Discount_Pct_Web` is not equal to `Promotion_Discount_Pct`.
  5. Print `sku`, `name`, `Current_Price_Web`, `Discount_Pct_Web`, `Promotion_Discount_Pct`, and `Discount_Changed`, then save the table to `data/gamma_price_check.csv` without row labels.
  6. Print one summary line, for example `Products with a changed discount: 1`.
- *New tools for this exercise:*
  - `.select(".item")` works for any class name; only the name changes from the BetaMart page (`.product`) to this page (`.item`).

  - Merging when the key columns have **different names**: `pd.merge(web_df, gamma_df, left_on="sku", right_on="Product_SKU", how="left")`. `left_on` names the key in the first table and `right_on` the key in the second.
  - `!=` means "not equal". On two columns, it gives one `True` / `False` per row.

- *Hints:*
  - Read the sale tag first with `sale_tag = card.select_one(".sale-price")`, then use `if sale_tag is not None:` to choose the current price.
  - Filter the CSV before merging: `gamma_df = df.loc[df["Competitor_Name"] == "GammaGlobal"]`.

- *Self-Check (expected results):*
  - Current prices on the page: `SKU-802` `160.65`, `SKU-805` `45.0` (no sale tag), `SKU-808` `263.5`, `SKU-811` `391.0`.
  - Only `SKU-811` (Electric Oven) changed: the web shows `15.0`% off, while last week's CSV says `10.0`%.
  - Summary line: `Products with a changed discount: 1`. Business meaning: GammaGlobal deepened its oven discount this week, so our pricing team should review our appliance prices.