ChatGPT

Python Data Analysis Cheat Sheet for Economics Students

1. Basic Python Syntax

- **Variables & Data Types:** In Python, you can assign values to variables without declaring a type. Common data types include integers, floats (decimals), strings (text), and booleans (True/False). For example:

```
# Assigning different types to variables
country = "France"        # str (text)
year = 2025               # int (integer)
gdp = 2.78                # float (in trillions)
is_developed = True       # bool (True/False)
```

You can use the `type()` function to check a variable's type. Python is dynamically typed, meaning it infers types at runtime (no need to specify types like in Stata or Excel).

- **Lists (Ordered Collections):** A list holds multiple items in a sequence (similar to a column of values or a series of entries in Excel). Lists are **0-indexed** (the first element is at index 0). Example:

```
# A list of values (e.g., GDP growth rates for 4 quarters)
quarterly_growth = [0.5, 0.8, 0.4, 0.6]
first_q = quarterly_growth[0]      # 0.5 (first item)
quarterly_growth.append(0.7)       # add a new element to the list
```

Here `quarterly_growth` is a list of floats. We accessed the first element with index 0, and added another entry. (In Stata, a list is analogous to a vector of values; in Excel, think of a list as a single column of cells with data.)

- **Dictionaries (Key-Value Pairs):** A dictionary stores mappings between keys and values (like an Excel lookup table or a mapping of IDs to values). For example:

```
# Dictionary of country capitals
capital_city = {"UK": "London", "France": "Paris", "USA": "Washington,
D.C."}
print(capital_city["France"])   # Output: Paris
```

Here the keys are country codes and the values are capitals. We retrieved the capital of France by using its key. (Analogy: a Python dict is like an Excel VLOOKUP table or a Stata associative array, mapping one set of values to another.)

2. Data Manipulation with Pandas

Pandas is Python's library for data analysis, offering a DataFrame object (like an Excel sheet or Stata dataset with rows and columns). First, import pandas and read data:

- **Reading Data (Excel/CSV):** Use `pandas` to load data from files into a DataFrame (table). For example, to read an Excel file:

```
import pandas as pd
df = pd.read_excel("gdpdata.xlsx")  # Load Excel data into DataFrame
print(df.head())  # show first 5 rows
```

This reads the file `gdpdata.xlsx` into `df`. You can also use `pd.read_csv("file.csv")` for CSV files. `df.head()` shows the first few rows, and `df.info()` provides data types and non-null counts. `df.describe()` gives summary stats for numeric columns (like Stata's `summarize`).

- **Selecting Columns:** To select one column, use `df['col_name']`. For multiple columns, provide a list of names: `df[['country', 'rgdpo']]`. For example:

```
countries = df[["country", "region"]]
# DataFrame with only country and region columns
```

This is like choosing specific variables in Stata or filtering to certain columns in Excel.

- **Filtering Rows (Querying Data):** You can filter rows by conditions, similar to using an "if" in Stata or filters in Excel. For example, to get all countries in Africa:

```
african_countries = df[df["region"] == "Africa"]
```

Or to filter by a numeric condition, say GDP per capita between 5,000 and 10,000:

```
mid_income = df[(df["gdppc"] > 5000) & (df["gdppc"] < 10000)]
```

(We use `&` for AND, `|` for OR, and each condition in parentheses.) Pandas also provides a `query()` method to filter with a SQL/Stata-like syntax, e.g. `df.query('gdppc > 5000 and gdppc < 10000')`. Both methods achieve the same result.

- **Creating New Variables:** Add new columns to the DataFrame by assigning to `df[new_column_name]`. You can base these on existing columns (analogous to using `generate` in Stata or creating a new column formula in Excel). For example:

```
# Assuming df has total GDP (rgdpo) and population (pop) columns
df["gdppc"] = df["rgdpo"] / df["pop"]  # GDP per capita
```

```
df["lab_prod"] = df["rgdpo"] / df["emp"]  # labor productivity (GDP per
worker)
```

This creates GDP per capita and labor productivity columns from existing data. We can also create conditional variables using NumPy or `where` clauses if needed.

- **Grouping & Aggregating:** To summarize data by categories (like a pivot table in Excel or Stata's `collapse` by group), use `groupby`. For example, to get average GDP per capita by region:

```
avg_gdppc_by_region = df.groupby("region")["gdppc"].mean()
print(avg_gdppc_by_region)
```

This returns the mean GDP per capita for each region (the index will be region names). Pandas `groupby` is analogous to Stata's `collapse` (by) operation [1] – it lets you compute aggregates (mean, sum, count, etc.) for each group of a key variable. You can aggregate multiple statistics at once using `.agg()`. For example:

```
df.groupby("incgroup")["gdppc"].agg(["mean", "min", "max"])
```

gives the mean, min, and max GDP per capita for each income group category. Use `.reset_index()` if you want to turn the group index back into a column (useful for viewing or merging results). *(Analogy:* `groupby` *is like summarizing data by group in Stata or using a PivotTable in Excel to get aggregate stats per category.)*

3. Plotting with Pandas/Matplotlib

Pandas integrates with Matplotlib (the main Python plotting library) to create charts quickly. First, make sure to import Matplotlib for any advanced customization: `import matplotlib.pyplot as plt`. Plotting basics:

- **Bar Chart:** Good for comparing category values. For example, to visualize average GDP per capita by region (from the groupby result above):

```
avg_gdppc_by_region.plot(kind="bar")
plt.title("Average GDP per Capita by Region")
plt.ylabel("GDP per Capita")
plt.show()
```

This produces a bar chart with regions on the x-axis and average GDP per capita on the y-axis. (This is similar to an Excel bar chart of grouped averages.)

- **Box Plot:** Useful for showing distribution of a variable (and comparing distributions). For instance, to plot the distribution of GDP per capita by income group:

```
df.boxplot(column="gdppc", by="incgroup")
plt.title("GDP per Capita by Income Group")
```

```
plt.suptitle("")  # Remove automatic suptitle
plt.ylabel("GDP per Capita")
plt.show()
```

This generates box plots of GDP per capita for each income group category, allowing visual comparison of medians and variability. (In Excel, this would be akin to creating multiple box plots for different categories.)

- **Scatter Plot:** Shows relationship between two numeric variables. For example, to visualize the relation between GDP per capita and labor productivity:

```
df.plot(kind="scatter", x="gdppc", y="lab_prod")
plt.title("Labor Productivity vs GDP per Capita")
plt.xlabel("GDP per Capita")
plt.ylabel("GDP per Worker")
plt.show()
```

Each point is a country (from our dataset) with its GDP per capita on the x-axis and GDP per worker on the y-axis. We might expect a positive correlation. You can add a trendline or regression later if needed (not shown here). *(Analogy: similar to creating an X-Y scatter in Excel to see correlation between two columns.)*

Note: Pandas plotting is built on Matplotlib, so you can use `plt.xlabel`, `plt.ylabel`, `plt.title`, etc., to customize labels and titles after calling `df.plot()`. If you need more control (multiple subplots, custom styles), you can use Matplotlib directly (`plt.plot(), plt.bar(), plt.scatter()`, etc.), but for quick charts, Pandas is very handy.

4. Retrieving Data from Online APIs

Instead of manually downloading data (e.g., from FRED or World Bank websites), Python can fetch data via their APIs. Two common sources for economics data are FRED (Federal Reserve Economic Data) and the World Bank's World Development Indicators (WDI). We use specialized libraries to access them:

- **FRED Data with `fredapi`:** First, obtain a free API key from the FRED website. Then:

```
from fredapi import Fred
fred = Fred(api_key="YOUR_FRED_API_KEY")
# Example: Get U.S. unemployment rate (UNRATE) time series
unemp_rate = fred.get_series("UNRATE")
print(unemp_rate.tail())  # print last few observations
```

This returns a Pandas Series indexed by date (here monthly unemployment rate). You can specify date ranges or frequency if needed, e.g. `fred.get_series("GDP", observation_start="1990-01-01", frequency='q')` to get quarterly GDP since 1990 [2]. The data comes directly from FRED's database, so it's always up-to-date.

- **World Bank Data (WDI):** There are a few ways to retrieve WDI data. In the workshop, we used the `wbgapi` library; another convenient method is using Pandas DataReader's World Bank interface. For example, using **pandas_datareader**:

```
import pandas_datareader.data as web
# Get World Bank data: GDP per capita (indicator NY.GDP.PCAP.CD) for
USA, 2000-2020
wb_df = web.DataReader("NY.GDP.PCAP.CD", "wb", start=2000, end=2020,
country="USA")
print(wb_df.head())
```

This fetches the GDP per capita series (current US$) for the United States from 2000 to 2020 [3]. The result is a DataFrame with a hierarchical index (country and year); you might call `.reset_index()` to turn year into a column. In our workshop, we used the `wbgapi` package, which allows access to multiple series and countries easily. For example, using **wbgapi** one could do:

```
import wbgapi as wb
# Fetch GDP growth (%), inflation (%), and unemployment (%) for USA
(2010-2020)
data = wb.data.DataFrame(['NY.GDP.MKTP.KD.ZG', 'FP.CPI.TOTL.ZG',
'SL.UEM.TOTL.ZS'],
                         economy='USA', time=range(2010, 2021))
```

This returns a DataFrame with years as index and the three indicators as columns. You could then analyze or merge this data similar to any other DataFrame. *(Remember: you can always refer to World Bank indicator codes on their website or via search functions. For instance,* `wb.series.info` *or* `wb.search` *can help find indicator codes by keywords.)*

Tip: The structure of data from APIs can sometimes be complex (multi-index, different frequency). Use methods like `df.head()`, `df.columns`, and `df.index` to understand the format, and `df.reset_index()` or `df.rename()` to tidy up if necessary. These skills are handy for quickly pulling real-world data for assignments or research.

5. Portfolio Optimization (Simplified Overview)

Portfolio optimization is about allocating wealth across assets to balance risk and return (e.g., finding the Markowitz efficient portfolio). In Python, we can use libraries to handle the math. In the workshop, we followed these key steps:

- **Libraries for Portfolio Analysis:** We used `pandas` and `numpy` for data manipulation and math, `yfinance` (Yahoo Finance API) to download historical price data for assets, and **SciPy** for optimization. (In practice, specialized libraries like **PyPortfolioOpt** can perform mean-variance optimization in just a few lines [4], but we went through the process manually to understand the steps.)

- **Data Collection:** First, gather price data for the assets in your portfolio. For example:

```
import yfinance as yf
tickers = ["AAPL", "GOOGL", "TSLA", "MSFT"]  # assets we want to include
price_data = yf.download(tickers, start="2020-01-01", end="2023-01-01")
["Adj Close"]
```

This downloads daily adjusted closing prices for Apple, Google, Tesla, and Microsoft over 2020–2022. The result `price_data` is a DataFrame with dates as index and columns for each ticker's price.

- **Compute Returns & Stats:** Calculate daily returns from prices and then annualize statistics. For example:

```
daily_returns = price_data.pct_change().dropna()
exp_returns = daily_returns.mean() * 252
# expected annual return for each asset
cov_matrix = daily_returns.cov() * 252          # annualized covariance
matrix
```

Here 252 is the approximate number of trading days in a year, so we scale daily mean and covariance to annual values. `exp_returns` is a Pandas Series of expected returns, and `cov_matrix` is a DataFrame of variances/covariances.

- **Set Up Optimization Problem:** We want to find weights (allocations) that optimize some objective (e.g., minimize portfolio variance for a given return, or maximize the Sharpe ratio). This involves constrained optimization: weights must sum to 1 (100% of capital) and often be non-negative (no short selling, unless allowed). For example, to minimize variance for a target return, we would define a function for portfolio variance given weights, set up a return constraint, and use SciPy's optimizer. In code, the setup might look like:

```
import numpy as np
from scipy.optimize import minimize

n = len(tickers)
init_guess = np.repeat(1/n, n)  # start with equal weights
bounds = [(0,1)] * n
# no short-selling (weights between 0 and 1)

# Constraint: weights sum to 1
constraints = ({'type': 'eq', 'fun': lambda w: np.sum(w) - 1})
# (Add another constraint for target return if needed: e.g. expected
return = R_target)

# Define objective: portfolio variance given weights w
def portfolio_variance(w):
    return float(np.dot(w.T, np.dot(cov_matrix, w)))  # w^T Σ w

result = minimize(portfolio_variance, init_guess, bounds=bounds,
```

```
constraints=constraints)
optimal_weights = result.x  # optimal weights array
```

We provided an initial guess and bounds, set the constraint that weights sum to 1, and minimized the portfolio variance. If we wanted to enforce a target return, we could include `{'type':'eq', 'fun': lambda w: np.dot(w, exp_returns) - R_target}` as another constraint. To **maximize Sharpe ratio**, one common approach is to maximize returns minus a penalty for risk (or equivalently minimize the negative Sharpe). In practice, this can be set up similarly by defining an objective function for negative Sharpe ratio.

- **Results – Optimal Weights:** The output `optimal_weights` is an array of numbers (one per asset) that give the optimal allocation. For clarity, you can pair these with tickers:

```
optimal_allocation = {ticker: f"{w*100:.1f}%" for ticker, w in
zip(tickers, optimal_weights)}
print("Optimal allocation:", optimal_allocation)
```

This might output something like:

```
Optimal allocation: {'AAPL': '10.0%', 'GOOGL': '25.0%', 'TSLA': '30.0%',
'MSFT': '35.0%'}
```

(Example numbers for illustration only.) These percentages sum to 100%. You can then calculate the **expected portfolio return** (`np.dot(optimal_weights, exp_returns)`) and **portfolio volatility** (`np.sqrt(optimal_weights @ cov_matrix @ optimal_weights.T)`) for these weights to evaluate the portfolio's performance. For instance, suppose the optimized portfolio has an expected annual return of 8.5% and a volatility of 5.4%, yielding a Sharpe ratio (assuming risk-free ~0) of about 1.57 – indicating a good risk-adjusted return.

- **Practical Notes:** This process is akin to using Excel's Solver for portfolio optimization (set an objective, adjust weights, apply constraints). Python's advantage is that it can handle many assets and scenarios programmatically. The example above uses SciPy's SLSQP solver to find the minimum variance portfolio. If you want to explore the **efficient frontier**, you can repeat the optimization for different target returns and plot the risk-return curve. In our workshop, we also tackled maximizing the Sharpe ratio directly (the so-called tangency portfolio). In real projects, one might use a library like **PyPortfolioOpt**, which can compute the optimal weights in a few lines and also provides functions to get the efficient frontier and performance metrics easily [4].

Summary: With these tools – Python basics, Pandas for data manipulation, plotting with Matplotlib, data retrieval from APIs, and optimization techniques – you have a powerful toolkit to analyze economic data. The skills parallel tasks you might do in Excel or Stata: for example, filtering and summarizing data (Pandas) is like data wrangling in Stata, and solving an optimization is like using Excel's Solver for portfolio selection. Practice these patterns, and you'll be able to apply them in coursework, research, or job tasks (like automating data reports or performing quantitative analysis in interviews). Happy coding!

---

1 Comparison with Stata — pandas 2.3.3 documentation
https://pandas.pydata.org/docs/getting_started/comparison/comparison_with_stata.html

2 GitHub - mortada/fredapi: Python API for FRED (Federal Reserve Economic Data) and ALFRED (Archival FRED)
https://github.com/mortada/fredapi

3 Remote Data Access — pandas-datareader 0.10.0 documentation
https://pandas-datareader.readthedocs.io/en/latest/remote_data.html

4 An Introduction to Portfolio Optimization in Python | Built In
https://builtin.com/data-science/portfolio-optimization-python