

# Explain Python code



```

import pandas as pd
from Bio.Seq import Seq
from Bio.SeqUtils import gc_fraction

# Use the full path to your file
df = pd.read_csv(r"C:\Users\srush\Documents\Desktop\DS\bi\human.txt", sep="\t")

# Loop through each DNA sequence
for idx, row in df.iterrows():
    sequence = Seq(row["sequence"])
    print(f"\n== Sequence {idx+1} (Class {row['class']}) ===")

    # 1. Finding motifs
    motif = "ATG"
    positions = [i+1 for i in range(len(sequence)-len(motif)+1)
                 if sequence[i:i+len(motif)] == motif]
    print("Motif positions:", positions)

    # 2. Calculating GC content
    GC_content = gc_fraction(sequence)
    print("GC Content: {:.2f}%".format(GC_content * 100))

    # 3. Identifying coding region (translate to protein)
    start = sequence.find("ATG")
    stop_codons = ["TAA", "TAG", "TGA"]
    coding_region = ""
    if start != -1:
        for i in range(start + 3, len(sequence), 3):
            if str(sequence[i:i+3]) in stop_codons:
                coding_region = sequence[start:i+3]
                break
    print("Coding Region:", coding_region if coding_region else "Not found")

    # 4. Translating coding region into protein sequence
    if coding_region:
        protein = Seq(coding_region).translate(to_stop=True)
        print("Protein Sequence:", protein)
    else:
        print("Protein Sequence: Not available")      explain this code line by line

```

Got it Let's go through your code **step by step, line by line**, so you fully understand what each part does.

## Imports

```
python

import pandas as pd
from Bio.Seq import Seq
from Bio.SeqUtils import gc_fraction
```

- `pandas as pd`: Imports the pandas library, used for handling tabular data (like CSV or TSV files).
- `from Bio.Seq import Seq`: Imports `Seq` from Biopython, which represents biological sequences (DNA, RNA, Protein).
- `from Bio.SeqUtils import gc_fraction`: Imports the function to calculate GC content in a DNA sequence.

## Reading the dataset

```
python

df = pd.read_csv(r"C:\Users\srush\Documents\Desktop\DS\bi\human.txt", sep="\t")
```

- Reads a **tab-separated text file** (`human.txt`) into a pandas DataFrame called `df`.
- Each row in `df` contains a DNA sequence and its associated class (based on your later usage: `row["sequence"]` and `row["class"]`).
- `sep="\t"` tells pandas the file uses **tab characters** instead of commas.

## Iterating over sequences

```
python

for idx, row in df.iterrows():
```

- Loops over each row in the DataFrame (`df`).
- `idx` is the row index (0, 1, 2, ...).
- `row` is the actual row, accessed like a dictionary (`row["sequence"]`).

## Creating a Biopython sequence

```
python

sequence = Seq(row["sequence"])
```

- Takes the string from the "sequence" column and converts it into a **Biopython Seq object**.
- `Seq` objects behave like strings but include biological methods (like `translate`).

## Printing sequence header

```
python
print(f"\n==== Sequence {idx+1} (Class {row['class']}) ===")
```

- Prints a header for each sequence, showing its number (`idx+1` because Python indexes from 0) and its class.
- Example output:

```
pgsql
==== Sequence 1 (Class promoter) ===
```

## 1. Finding motifs

```
python
motif = "ATG"
positions = [i+1 for i in range(len(sequence)-len(motif)+1)
             if sequence[i:i+len(motif)] == motif]
print("Motif positions:", positions)
```

- Defines a **motif** ("ATG"), often the start codon in DNA.
- Uses a **list comprehension** to check every substring of length 3 in the sequence.
- If the substring matches "ATG", it stores its **position (1-based index)**.
- Example: if "ATG" occurs at index 0 and 10 → `positions = [1, 11]`.

## 2. Calculating GC content

```
python
GC_content = gc_fraction(sequence)
print("GC Content: {:.2f}%".format(GC_content * 100))
```

- Calls Biopython's `gc_fraction()` to calculate fraction of G and C bases in the sequence.
- Multiplies by 100 to get percentage.
- `"{:.2f}%".format(...)` formats it to 2 decimal places.
- Example: **GC Content: 47.65%**.

## 3. Identifying coding region

```
python
start = sequence.find("ATG")
stop_codons = ["TAA", "TAG", "TGA"]
```

```
coding_region = ""
```

- `start = sequence.find("ATG")`: Finds the first occurrence of "ATG" (start codon). Returns -1 if not found.
- Defines `stop_codons`: standard stop codons.
- `coding_region = ""`: Initializes as empty (in case no coding region is found).

python

```
if start != -1:
    for i in range(start + 3, len(sequence), 3):
        if str(sequence[i:i+3]) in stop_codons:
            coding_region = sequence[start:i+3]
            break
```

- If "ATG" was found (`start != -1`):
  - Loop over the sequence **in steps of 3** (codons), starting after the start codon.
  - Extract a codon: `sequence[i:i+3]`.
  - Convert to string (since `Seq` slices are `Seq` objects).
  - If codon matches one of the stop codons → this defines the **end of the coding region**.
  - `coding_region = sequence[start:i+3]`: store DNA subsequence from start codon to stop codon (inclusive).
  - `break`: stop once the first valid coding region is found.

python

```
print("Coding Region:", coding_region if coding_region else "Not found")
```

- Prints the coding region DNA if found, otherwise "Not found".

## 4. Translating coding region into protein

python

```
if coding_region:
    protein = Seq(coding_region).translate(to_stop=True)
    print("Protein Sequence:", protein)
else:
    print("Protein Sequence: Not available")
```

- If a coding region exists:
  - Convert it into a `Seq` object and call `.translate(to_stop=True)`.
  - `.translate()` converts DNA → Protein sequence using genetic code.
  - `to_stop=True` means translation stops at the first stop codon instead of including it as \*.
- If no coding region found → print "Not available".

 In short, your code does this for each sequence:

1. Prints sequence number and class.
2. Finds all "ATG" motifs.
3. Calculates GC content.
4. Identifies the coding region between "ATG" and the first stop codon.
5. Translates the coding region into a protein sequence.

Do you want me to also **run through an example sequence** (like "ATGCGTAAATGA") and show the **actual step-by-step output** your code would generate?



```
import pandas as pd
import numpy as np
import statsmodels.api as sm
from statsmodels.stats.multitest import multipletests
import matplotlib.pyplot as plt
import seaborn as sns
from pydeseq2.utils import load_example_data

# =====
# Data Initialization
# =====

# Load example counts dataset
counts_df = load_example_data()

# Recreate metadata (conditions: half Control, half Treatment)
n_samples = counts_df.shape[1]
conditions = ["Control"] * (n_samples // 2) + ["Treatment"] * (n_samples // 2)
metadata_df = pd.DataFrame({"condition": conditions}, index=counts_df.columns)

print("Initial Data (first 5 rows):")
print(counts_df.head())

# =====
# Differential Expression Analysis
# =====

# Convert conditions to numeric design matrix
X = pd.get_dummies(metadata_df["condition"], drop_first=True).astype(float)
X = sm.add_constant(X) # add intercept

# Log2 transform to stabilize variance
df_log = np.log2(counts_df + 1)

# Collect results in list of dicts
```

```

results = []

for idx, gene in enumerate(df_log.index, 1):
    y = df_log.loc[gene].values.astype(float) # response variable

    model = sm.OLS(y, X).fit()

    log2_fc = model.params.iloc[1]
    fold_change = 2 ** log2_fc
    p_value = model.pvalues.iloc[1]

    results.append({
        "Gene": gene,
        "Log2_FC": log2_fc,
        "Fold_Change": fold_change,
        "P-value": p_value
    })

# Convert to DataFrame
results_df = pd.DataFrame(results)

# Adjust p-values using Benjamini-Hochberg (FDR)
results_df["Adj_P-value"] = multipletests(results_df["P-value"], method="fdr_bh")[1]

print("\nDifferential Expression Results (first 10 rows):")
print(results_df.head(10))

# =====
# Data Visualization
# =====

# Volcano plot
plt.figure(figsize=(10, 6))
sns.scatterplot(
    data=results_df,
    x="Log2_FC",
    y=-np.log10(results_df["P-value"]),
    hue=results_df["Adj_P-value"] < 0.05,
    palette={True: "red", False: "grey"},
    alpha=0.7
)

# Significance threshold line
plt.axhline(y=-np.log10(0.05), color="blue", linestyle="--", label="P-value = 0.05")

# Labels & styling
plt.xlabel("Log2 Fold Change")
plt.ylabel("-log10(P-value)")
plt.title("Volcano Plot of Differential Gene Expression")
plt.legend()

```

```
sns.despine()
plt.show()  explain this code line by line
```

Perfect, let's carefully walk through your **differential expression analysis code** line by line. This combines data handling, statistical modeling, multiple testing correction, and visualization — so I'll break it into sections for clarity.

## 📌 Imports

```
python

import pandas as pd
import numpy as np
import statsmodels.api as sm
from statsmodels.stats.multitest import multipletests
import matplotlib.pyplot as plt
import seaborn as sns
from pydeseq2.utils import load_example_data
```

- **pandas**: For working with tabular gene expression data.
- **numpy**: For numerical operations (log transform, arrays).
- **statsmodels.api as sm**: For statistical modeling (OLS regression).
- **multipletests**: For multiple hypothesis testing correction (FDR).
- **matplotlib.pyplot + seaborn**: For visualization (volcano plot).
- **load\_example\_data**: Loads a built-in **gene counts dataset** (like RNA-seq).

## 📌 Data Initialization

```
python

# Load example counts dataset
counts_df = load_example_data()
```

- Loads a **counts dataset** (rows = genes, columns = samples).
- Each cell = raw counts of reads mapped to that gene in that sample.

```
python

# Recreate metadata (conditions: half Control, half Treatment)
n_samples = counts_df.shape[1]
conditions = ["Control"] * (n_samples // 2) + ["Treatment"] * (n_samples // 2)
metadata_df = pd.DataFrame({"condition": conditions}, index=counts_df.columns)
```

- **n\_samples = counts\_df.shape[1]**: number of samples (columns).
- Creates conditions: first half "Control", second half "Treatment".
- Wraps this in a DataFrame (**metadata\_df**) with sample names as index.

```
python
```

```
print("Initial Data (first 5 rows):")
print(counts_df.head())
```

- Displays the first 5 genes (rows) to check the input.

## 📌 Differential Expression Analysis

```
python
```

```
# Convert conditions to numeric design matrix
X = pd.get_dummies(metadata_df["condition"], drop_first=True).astype(float)
X = sm.add_constant(X) # add intercept
```

- `pd.get_dummies`: Converts "Control" / "Treatment" into **dummy variables**.
  - If `drop_first=True`, "Control" becomes baseline (0), "Treatment" = 1.
- `sm.add_constant(X)`: Adds an intercept (so model =  $\beta_0 + \beta_1 \cdot \text{Treatment}$ ).

```
python
```

```
# Log2 transform to stabilize variance
df_log = np.log2(counts_df + 1)
```

- RNA-seq counts are **skewed** (some genes have very high counts).
- Log2 transform reduces variance and makes data more normal-like.
- `+1` avoids `log(0)`.

```
python
```

```
# Collect results in list of dicts
results = []
```

- Initialize empty list to store per-gene results.

## Loop through genes

```
python
```

```
for idx, gene in enumerate(df_log.index, 1):
    y = df_log.loc[gene].values.astype(float) # response variable

    model = sm.OLS(y, X).fit()

    log2_fc = model.params.iloc[1]
    fold_change = 2 ** log2_fc
    p_value = model.pvalues.iloc[1]

    results.append({
        "Gene": gene,
        "Log2_FC": log2_fc,
        "Fold_Change": fold_change,
```

```

    "P-value": p_value
})

```

For each gene:

1. `y = df_log.loc[gene].values`: Expression levels across samples.
2. Fit **OLS regression**: `y ~ intercept + Treatment`.
  - Treatment coefficient ( $\beta_1$ ) = log2 fold change.
3. Extract results:
  - `log2_fc = model.params.iloc[1]`: Log2 fold change.
  - `fold_change = 2 ** log2_fc`: Convert log2FC back to fold change scale.
  - `p_value = model.pvalues.iloc[1]`: p-value for treatment effect.
4. Append results into a dictionary.

## Combine results

```

python

results_df = pd.DataFrame(results)

```

- Converts list of dictionaries into a results DataFrame.

```

python

# Adjust p-values using Benjamini-Hochberg (FDR)
results_df["Adj_P-value"] = multipletests(results_df["P-value"], method="fdr_bh")[1]

```

- Adjusts p-values for **multiple testing** (thousands of genes).
- Uses Benjamini-Hochberg procedure (controls False Discovery Rate).
- [1] extracts the adjusted p-values array.

```

python

print("\nDifferential Expression Results (first 10 rows):")
print(results_df.head(10))

```

- Shows first 10 genes with log2FC, fold change, raw p-value, adjusted p-value.

## 📌 Data Visualization (Volcano Plot)

```

python

plt.figure(figsize=(10, 6))
sns.scatterplot(
    data=results_df,
    x="Log2_FC",
    y=-np.log10(results_df["P-value"]),

```

```

hue=results_df["Adj_P-value"] < 0.05,
palette={True: "red", False: "grey"},
alpha=0.7
)

```

- Creates a **volcano plot**:
  - x-axis = Log2 Fold Change (effect size).
  - y-axis = -log10(p-value) (statistical significance).
  - Color: red = significant (FDR < 0.05), grey = not significant.

python

```

# Significance threshold line
plt.axhline(y=-np.log10(0.05), color="blue", linestyle="--", label="P-value = 0.05")

```

- Adds a horizontal dashed line at p = 0.05.
- Helps visualize significance cutoff.

python

```

# Labels & styling
plt.xlabel("Log2 Fold Change")
plt.ylabel("-log10(P-value)")
plt.title("Volcano Plot of Differential Gene Expression")
plt.legend()
sns.despine()
plt.show()

```

- Labels axes, adds title and legend.
- `sns.despine()`: Removes top/right borders for cleaner look.
- `plt.show()`: Displays the plot.

### ✓ In summary, this code:

- Loads RNA-seq count data and sample conditions.
- Runs per-gene regression (`expression ~ condition`).
- Estimates **log2 fold changes** & p-values.
- Corrects p-values for multiple testing.
- Visualizes results in a **volcano plot** (fold change vs significance).

Do you want me to also show you **a mock output example** (like what the results table and volcano plot would look like for 1-2 genes)?