Analyzing a DNA Sequence

This code works with a DNA sequence and performs a few basic biological analyses. First, it counts how many times each base (A, T, C, G) appears in the sequence. Then, it calculates the GC content: the percentage of the sequence made up of G (guanine) and C (cytosine), which helps scientists understand the stability of DNA. Lastly, it finds the reverse complement of the DNA: which is like reading the DNA backward and flipping each base to its matching partner (A↔T, C↔G), something important for understanding how DNA is read in cells.

```
from Bio.Seq import Seq

# Define a DNA sequence
dna_sequence = Seq("ATGCGATAGCTAGCTAGCTAGCATCGATCG")

# Nucleotide count
print("Nucleotide counts:", {base: dna_sequence.count(base) for base in "ATCG"})

# GC content
gc_content = (dna_sequence.count("G") + dna_sequence.count("C")) / len(dna_sequence) * 100
print(f"GC Content: {gc_content:.2f}%")

# Reverse complement
reverse_comp = dna_sequence.reverse_complement()
print("Reverse complement:", reverse_comp)
```

```
Nucleotide counts: {'A': 8, 'T': 7, 'C': 7, 'G': 8}
GC Content: 50.00%
Reverse complement: CGATCGATGCTAGCTAGCTAGCTATCGCAT
```

Analyzing a Protein Sequence

This code analyzes a protein sequence, which is just a chain of amino acids (the building blocks of proteins). First, it calculates the amino acid composition, showing what percentage of the protein is made up of each type of amino acid. Next, it estimates the protein molecular weight, which tells us how heavy the molecule is. Finally, it calculates the net electrical charge of the protein at a typical biological pH (7.4), helping scientists understand how the protein might behave in the body (like how it interacts with other molecules).

```
from Bio.SeqUtils.ProtParam import ProteinAnalysis

protein_sequence = "MAEGEITTFTALTEKFNLPPGNYKKPKLLYCSNGGHFLRILPDGTVDGTRDRSDQHIQLQLSAESVGEVYIKSTETGQYLAMDTSGLLYGSQTPSEECLFLERLEENHYNTYTSKI

# Amino acid composition
protein = ProteinAnalysis(protein_sequence)
aa_composition = protein.get_amino_acids_percent()
print("Amino acid composition:", aa_composition)

# Molecular weight
mw = protein.molecular_weight()
print(f"Molecular weight: {mw:.2f} Da")

# Charge at pH 7.4
charge = protein.charge_at_pH(7.4)
print(f"Net charge at pH 7.4: {charge:.2f}")
```

```
Amino acid composition: {'A': 0.039473684210526314, 'C': 0.019736842105263157, 'D': 0.03289473684210526, 'E': 0.07894736842105263,
Molecular weight: 17103.16 Da
Net charge at pH 7.4: 0.36
C:\Users\LENOVO\AppData\Local\Programs\Python\Python313\Lib\site-packages\Bio\SeqUtils\ProtParam.py:106: BiopythonDeprecationWarning
  warnings.warn(
```

RNA-Seq Data Analysis with DESeq2 (via rpy2): sample data generation

This code creates a simulated dataset that mimics(synthetic RNA molecules) RNA-Seq gene expression data. It includes 1,000 genes measured across 12 samples: 6 from a control group and 6 from a treatment group. The gene activity levels are generated randomly to resemble real biological data, and for 100 of the genes, the expression is artificially increased or decreased in the treatment group to simulate meaningful biological changes. The result is organized into two tables: one showing gene expression counts and another with sample details, such as group labels and experimental batch. This setup is commonly used in teaching or testing bioinformatics workflows that look for genes behaving differently between conditions.

```
import numpy as np
import pandas as pd
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt
```

```
import seaborn as sns
from scipy import stats
from statsmodels.stats.multitest import multipletests

# Set random seed for reproducibility
np.random.seed(42)

# Generate synthetic RNA-Seq data
n_genes = 1000
n_samples = 12  # 6 control, 6 treatment

# Create gene names
genes = [f'Gene_{i}' for i in range(n_genes)]

# Create sample names (6 control, 6 treatment)
samples = [f'Control_{i}' for i in range(6)] + [f'Treatment_{i}' for i in range(6)]

# Generate baseline expression (negative binomial distribution)
base_expression = np.random.negative_binomial(5, 0.1, size=(n_genes, n_samples))

# Add differential expression for 100 genes (50 up, 50 down)
de_genes_up = np.random.choice(n_genes, 50, replace=False)
de_genes_down = np.random.choice(np.setdiff1d(range(n_genes), de_genes_up), 50, replace=False)

# Add fold change to treatment samples (columns 6-11)
fold_change_up = np.random.uniform(1.5, 5, 50)
fold_change_down = np.random.uniform(0.2, 0.67, 50)

for i, gene in enumerate(de_genes_up):
    base_expression[gene, 6:] = (base_expression[gene, 6:] * fold_change_up[i]).astype(int)

for i, gene in enumerate(de_genes_down):
    base_expression[gene, 6:] = (base_expression[gene, 6:] * fold_change_down[i]).astype(int)

# Create DataFrame
count_data = pd.DataFrame(base_expression, index=genes, columns=samples)

# Add metadata
metadata = pd.DataFrame({
    'sample': samples,
    'condition': ['Control']*6 + ['Treatment']*6,
    'batch': ['A', 'B', 'C', 'A', 'B', 'C']*2
})

print("Sample count data:")
print(count_data.iloc[:5, :5])
print("\nMetadata:")
print(metadata)
```

```
Sample count data:
         Control_0  Control_1  Control_2  Control_3  Control_4
Gene_0         54         41         27         52         23
Gene_1         35         32         42         39         30
Gene_2         63         45         42         18         32
Gene_3         38         36         36         42         20
Gene_4         43         50         52         24         35

Metadata:
          sample  condition batch
0     Control_0    Control     A
1     Control_1    Control     B
2     Control_2    Control     C
3     Control_3    Control     A
4     Control_4    Control     B
5     Control_5    Control     C
6   Treatment_0  Treatment     A
7   Treatment_1  Treatment     B
8   Treatment_2  Treatment     C
9   Treatment_3  Treatment     A
10  Treatment_4  Treatment     B
11  Treatment_5  Treatment     C
```

```
Differential Expression Analysis (DESeq2/edgeR-like)
```

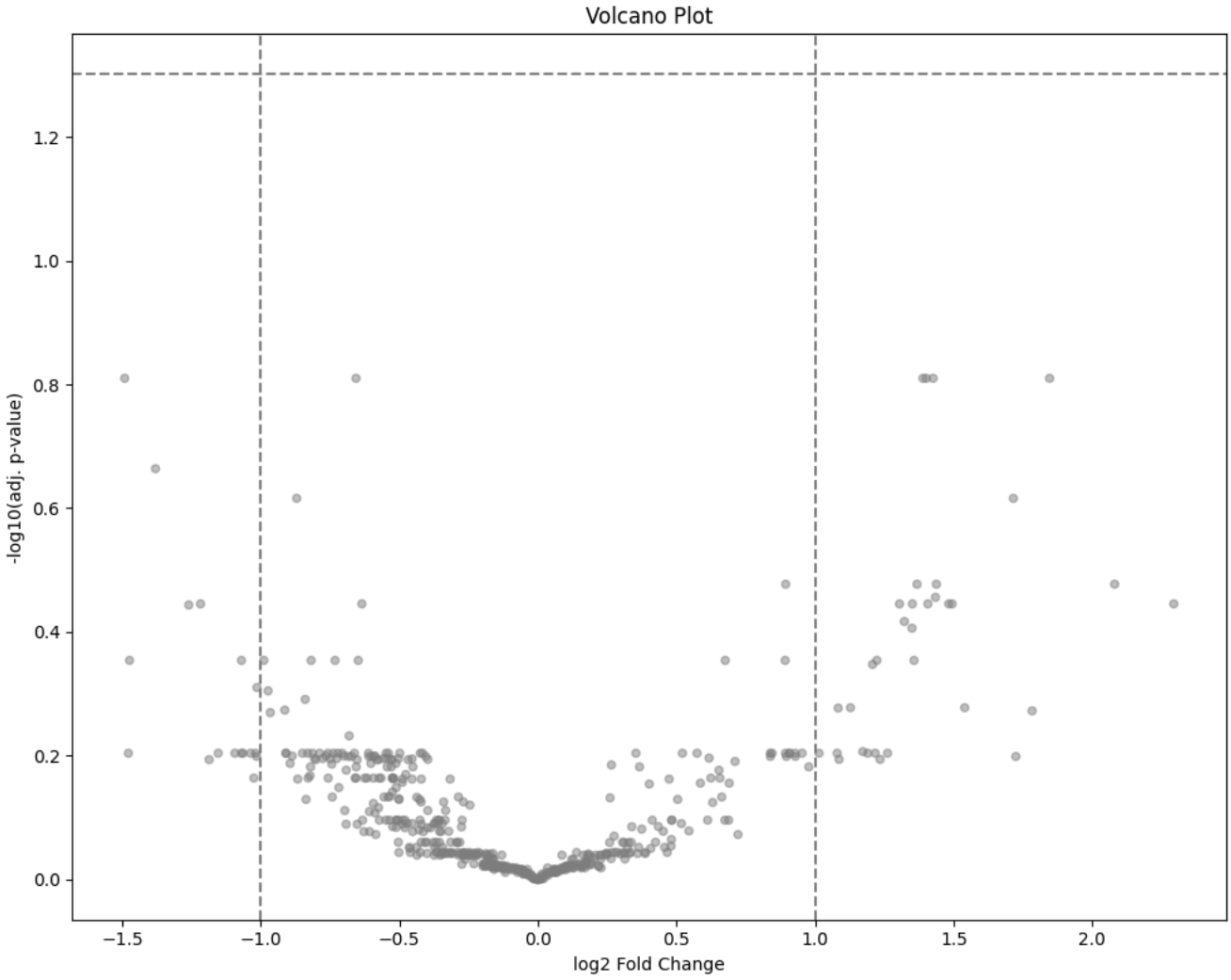
This script simulates a full RNA-Seq analysis workflow from start to finish using synthetic data. It first creates fake gene expression data for 500 genes across 10 samples (5 control and 5 treatment), including some differentially expressed genes to mimic real biological changes. It then performs differential expression analysis to find genes that change between groups and saves the results. A volcano plot visually highlights which genes are most significantly different. Next, the script runs PCA (Principal Component Analysis) to check how samples group based on gene expression and creates a colorful heatmap of the most variable genes to explore patterns across samples. Throughout, it saves all data, plots, and session info to an output folder, ensuring that the analysis is reproducible and well-documented.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from scipy import stats
from statsmodels.stats.multitest import multipletests
import seaborn as sns
import os

# Create output directory if it doesn't exist
output_dir = "D:\\LTU\\PU\\output"
os.makedirs(output_dir, exist_ok=True)

# Set random seed for reproducibility
np.random.seed(42)

# 1. Generate and save synthetic RNA-Seq data
print("1. Generating synthetic RNA-Seq data...")
n_genes = 500
n_samples = 10  # 5 control, 5 treatment

# Create data with some differentially expressed genes
data = np.random.negative_binomial(5, 0.1, size=(n_genes, n_samples))
de_genes = np.random.choice(n_genes, 50, replace=False)
data[de_genes, 5:] = data[de_genes, 5:] * np.random.uniform(1.5, 4, size=(50, 5))

genes = [f'Gene_{i}' for i in range(n_genes)]
samples = [f'Control_{i}' for i in range(5)] + [f'Treatment_{i}' for i in range(5)]
counts = pd.DataFrame(data, index=genes, columns=samples)

# Metadata with sample names matching count matrix columns
metadata = pd.DataFrame({
    'sample': samples,
    'condition': ['Control']*5 + ['Treatment']*5
}).set_index('sample')

# Save raw data
counts.to_csv(f"{output_dir}/raw_counts.csv")
metadata.to_csv(f"{output_dir}/metadata.csv")

print(f"\nCount matrix and metadata saved to {output_dir}/")

# 2. Differential Expression Analysis
print("\n2. Performing differential expression analysis...")
def simple_de(counts, metadata):
    # Normalize (logCPM)
    cpm = counts / counts.sum(axis=0) * 1e6
    logcpm = np.log2(cpm + 1)
    logcpm.to_csv(f"{output_dir}/normalized_counts_logcpm.csv")

    # Get sample groups
    ctrl_samples = metadata[metadata['condition']=='Control'].index
    treat_samples = metadata[metadata['condition']=='Treatment'].index

    # Calculate mean expression and logFC
    logFC = logcpm[treat_samples].mean(axis=1) - logcpm[ctrl_samples].mean(axis=1)

    # T-test
    pvals = [stats.ttest_ind(logcpm.loc[g, treat_samples],
                             logcpm.loc[g, ctrl_samples],
                             equal_var=False)[1]
             for g in logcpm.index]
    padj = multipletests(pvals, method='fdr_bh')[1]

    return pd.DataFrame({
        'log2FC': logFC,
        'pvalue': pvals,
        'padj': padj
    }, index=logcpm.index)

de_results = simple_de(counts, metadata)
de_results.to_csv(f"{output_dir}/differential_expression_results.csv")
print(f"\nDE results saved to {output_dir}/differential_expression_results.csv")

# 3. Volcano Plot
print("\n3. Generating volcano plot...")
def simple_volcano(de_results, fc_thresh=1, p_thresh=0.05):
    plt.figure(figsize=(10,8))

    # Color significant points
    sig = (de_results['padj'] < p_thresh) & (de_results['log2FC'].abs() > fc_thresh)
```

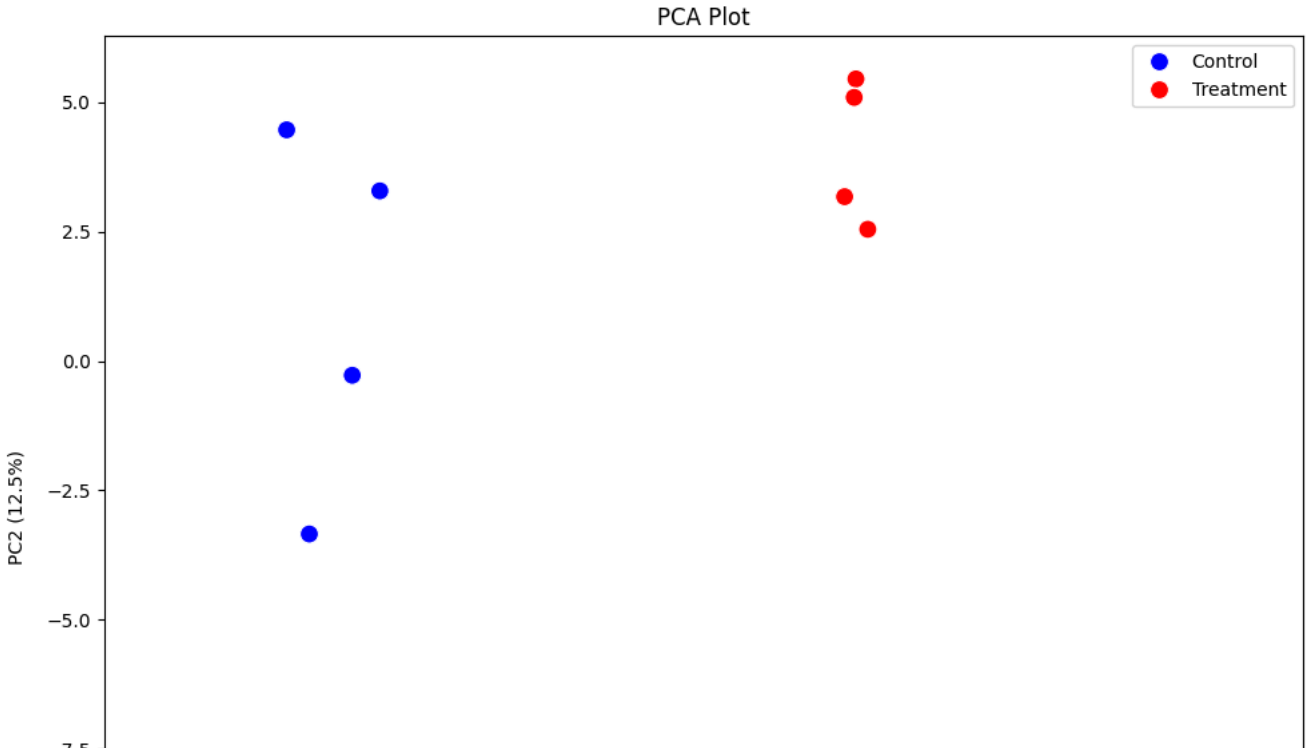
```
    plt.scatter(de_results['log2FC'], -np.log10(de_results['padj']),
                c=sig.map({True: 'red', False: 'gray'}), alpha=0.5, s=20)

    plt.axhline(-np.log10(p_thresh), linestyle='--', color='gray')
    plt.axvline(fc_thresh, linestyle='--', color='gray')
    plt.axvline(-fc_thresh, linestyle='--', color='gray')

    plt.xlabel('log2 Fold Change')
    plt.ylabel('-log10(adj. p-value)')
    plt.title('Volcano Plot')
    plt.tight_layout()
    plt.savefig(f"{output_dir}/volcano_plot.png", dpi=300)
    plt.savefig(f"{output_dir}/volcano_plot.pdf")
    plt.show()

simple_volcano(de_results)
print(f"Volcano plot saved to {output_dir}/volcano_plot.[png/pdf]")

# 4. PCA Plot
print("\n4. Performing PCA analysis...")
def simple_pca(counts, metadata):
    # Normalize
    cpm = counts / counts.sum(axis=0) * 1e6
    logcpm = np.log2(cpm + 1)

    # Perform PCA
    pca = PCA(n_components=2)
    pc = pca.fit_transform(logcpm.T)

    # Create plot data
    plot_data = pd.DataFrame(pc, columns=['PC1', 'PC2'], index=logcpm.columns)
    plot_data = plot_data.join(metadata)

    # Save PCA coordinates
    plot_data.to_csv(f"{output_dir}/pca_coordinates.csv")

    # Plot
    plt.figure(figsize=(10,8))
    sns.scatterplot(data=plot_data, x='PC1', y='PC2', hue='condition',
                   palette={'Control':'blue', 'Treatment':'red'}, s=100)

    plt.xlabel(f'PC1 ({pca.explained_variance_ratio_[0]*100:.1f}%)')
    plt.ylabel(f'PC2 ({pca.explained_variance_ratio_[1]*100:.1f}%)')
    plt.title('PCA Plot')
    plt.legend()
    plt.tight_layout()
    plt.savefig(f"{output_dir}/pca_plot.png", dpi=300)
    plt.savefig(f"{output_dir}/pca_plot.pdf")
    plt.show()

    # Print and save variance explained
    var_exp = pd.DataFrame({
        'PC': ['PC1', 'PC2'],
        'Variance Explained': pca.explained_variance_ratio_[:2]
    })
    var_exp.to_csv(f"{output_dir}/pca_variance_explained.csv", index=False)
    print(f"\nPCA results saved to {output_dir}/pca_* files")

simple_pca(counts, metadata)

# 5. Clustering Heatmap
print("\n5. Generating clustered heatmap...")
def simple_heatmap(counts, metadata, n_genes=30):
    # Normalize and select top variable genes
    cpm = counts / counts.sum(axis=0) * 1e6
    logcpm = np.log2(cpm + 1)
    top_genes = logcpm.var(axis=1).sort_values(ascending=False).index[:n_genes]

    # Save top variable genes list
    pd.Series(top_genes).to_csv(f"{output_dir}/top_variable_genes.csv", index=False)

    # Create color mapping
    colors = metadata['condition'].map({'Control':'blue', 'Treatment':'red'})

    # Plot
    plt.figure(figsize=(12,10))
    g = sns.clustermap(
        logcpm.loc[top_genes],
        cmap='viridis',
        z_score=0,
        col_colors=colors,
        figsize=(12,10),
```

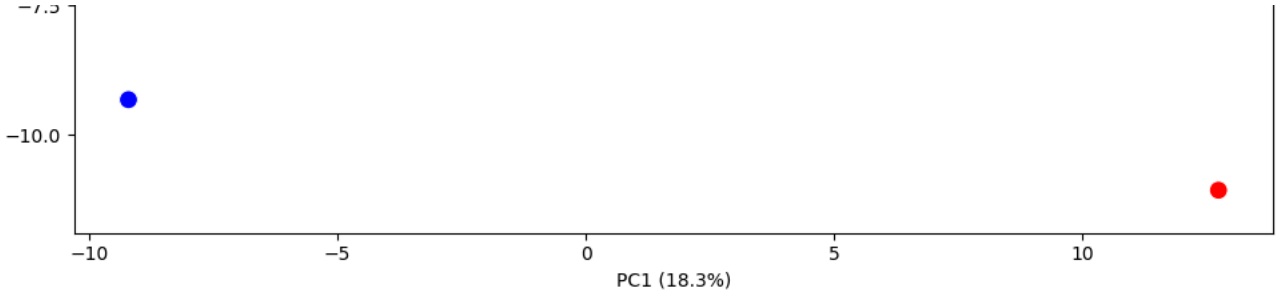
```
            yticklabels=True
        )
        plt.title(f'Top {n_genes} Variable Genes Heatmap')

        # Adjust layout to prevent cutting off labels
        g.savefig(f"{output_dir}/heatmap.png", dpi=300, bbox_inches='tight')
        g.savefig(f"{output_dir}/heatmap.pdf", bbox_inches='tight')
        plt.close()
        print(f"Heatmap saved to {output_dir}/heatmap.[png/pdf]")

    simple_heatmap(counts, metadata)

    # 6. Save session info for reproducibility
    with open(f"{output_dir}/analysis_session_info.txt", "w") as f:
        f.write("RNA-Seq Analysis Session Info\n")
        f.write(f"Date: {pd.Timestamp.now()}\n\n")
        f.write("Libraries and versions:\n")
        f.write(f"numpy: {np.__version__}\n")
        f.write(f"pandas: {pd.__version__}\n")
        f.write(f"scipy: {stats.__version__}\n")
        f.write(f"sklearn: {PCA.__module__.split('.')[0]}\n")
        f.write(f"seaborn: {sns.__version__}\n")
        f.write(f"matplotlib: {plt.__version__}\n")

    print("\nAnalysis complete! All results saved to:", os.path.abspath(output_dir))
```

```
1. Generating synthetic RNA-Seq data...

Count matrix and metadata saved to D:\LTU\PU\output/

2. Performing differential expression analysis...

DE results saved to D:\LTU\PU\output/differential_expression_results.csv

3. Generating volcano plot...
```




```
Volcano plot saved to D:\LTU\PU\output/volcano_plot.[png/pdf]

4. Performing PCA analysis...
```






```
PCA results saved to D:\LTU\PU\output/pca_* files

5. Generating clustered heatmap...
Heatmap saved to D:\LTU\PU\output/heatmap.[png/pdf]
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[16], line 184
    182 f.write(f"numpy: {np.__version__}\n")
    183 f.write(f"pandas: {pd.__version__}\n")
--> 184 f.write(f"scipy: {stats.__version__}\n")
    185 f.write(f"sklearn: {PCA.__module__.split('.')[0]}\n")
    186 f.write(f"seaborn: {sns.__version__}\n")

AttributeError: module 'scipy.stats' has no attribute '__version__'

<Figure size 1200x1000 with 0 Axes>
```

Phylogenetic Tree Construction: Building a Tree with Biopython

This script creates a set of sample protein sequences (fragments of the cytochrome b protein) from different species, introducing small differences (mutations) to make each sequence unique. It then performs a multiple sequence alignment (MSA) to line up these sequences and compare them, simulating how real biological sequences would be analyzed. Using the aligned sequences, it calculates a distance matrix that shows how similar or different each pair of species is. Next, it builds two types of phylogenetic trees (Neighbor-Joining and UPGMA) to visualize evolutionary relationships based on those distances. The script saves all the sequences, alignments, distance data, and tree files, and also produces nice tree diagrams as images. Finally, it prints a simple text version of one tree for quick viewing—all helping researchers understand how these species relate to each other at the protein level.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from Bio import AlignIO, Phylo
from Bio.Phylo.TreeConstruction import DistanceCalculator, DistanceTreeConstructor
from Bio import SeqIO
from Bio.Seq import Seq
from Bio.SeqRecord import SeqRecord
from Bio.Align import MultipleSeqAlignment
from Bio.Seq import MutableSeq
import os

# Set output directory
output_dir = r"D:\LTU\PU\output"
os.makedirs(output_dir, exist_ok=True)

# 1. Generate realistic sample protein sequences
print(f"1. Creating sample protein sequences (Cytochrome b fragments)...\nOutput will be saved to: {output_dir}")

# Define sample sequences
sequences = [
    SeqRecord(Seq("MTPIRNKSLLSLKTLLLTLTSPVMAEGVLTWGSQMSDEWGIVQNINNYWGEVTSVLAYM"),
              id="Human", description="Homo sapiens"),
    SeqRecord(Seq("MTPIRNKSLLSLKTLLLTLTSPVMAEGVLTWGSQMSDEWGIVQNINNYWGEVTSVLAYM"),
              id="Chimp", description="Pan troglodytes"),
    SeqRecord(Seq("MTPIRNKSLLSLKTLLLTLTSPVMAEGVLTWGSQMSDEWGIVQNINNYWGEVTSVLAYM"),
              id="Gorilla", description="Gorilla gorilla"),
    SeqRecord(Seq("MTPIRNKSLLSLKTLLLTLTSPVMAEGVLTWGSQMSDEWGIVQNINNYWGEVTSVLAYM"),
              id="Orangutan", description="Pongo pygmaeus"),
    SeqRecord(Seq("MTPIRNKSLLSLKTLLLTLTSPVMAEGVLTWGSQMSDEWGIVQNINNYWGEVTSVLAYM"),
              id="Gibbon", description="Hylobates lar"),
    SeqRecord(Seq("MTPIRNKSLLSLKTLLLTLTSPVMAEGVLTWGSQMSDEWGIVQNINNYWGEVTSVLAYM"),
              id="Macaque", description="Macaca mulatta"),
    SeqRecord(Seq("MTPIRNKSLLSLKTLLLTLTSPVMAEGVLTWGSQMSDEWGIVQNINNYWGEVTSVLAYM"),
              id="Mouse", description="Mus musculus"),
    SeqRecord(Seq("MTPIRNKSLLSLKTLLLTLTSPVMAEGVLTWGSQMSDEWGIVQNINNYWGEVTSVLAYM"),
              id="Cow", description="Bos taurus"),
    SeqRecord(Seq("MTPIRNKSLLSLKTLLLTLTSPVMAEGVLTWGSQMSDEWGIVQNINNYWGEVTSVLAYM"),
              id="Whale", description="Balaenoptera musculus"),
```

```
    SeqRecord(Seq("MTPIRNKSLLSLKTLLLTLTSPVMAEGVLTWGSQMSDEWGIVQNINNYWGEVTSVLAYM"),
              id="Chicken", description="Gallus gallus")
]

# Introduce mutations to differentiate sequences
variations = {
    "Chimp": [(5, 'K'), (40, 'D')],
    "Gorilla": [(12, 'T'), (25, 'V')],
    "Orangutan": [(18, 'R'), (53, 'E')],
    "Gibbon": [(25, 'L'), (32, 'M')],
    "Macaque": [(5, 'N'), (47, 'S')],
    "Mouse": [(12, 'A'), (18, 'S'), (25, 'T'), (32, 'A'), (40, 'E'), (47, 'T'), (53, 'D')],
    "Cow": [(5, 'S'), (18, 'A'), (40, 'N')],
    "Whale": [(12, 'G'), (25, 'A'), (47, 'A')],
    "Chicken": [(5, 'P'), (12, 'S'), (18, 'G'), (25, 'S'), (32, 'G'), (40, 'K'), (47, 'R'), (53, 'K')]
}

# Apply mutations
for species, mutations in variations.items():
    for seq in sequences:
        if seq.id == species:
            mutable_seq = MutableSeq(str(seq.seq))
            for pos, aa in mutations:
                mutable_seq[pos] = aa
            seq.seq = Seq(str(mutable_seq))
            break

# Save sequences
fasta_path = os.path.join(output_dir, "cytochrome_b.fasta")
SeqIO.write(sequences, fasta_path, "fasta")
print(f"\nSample sequences saved to {fasta_path}")

# 2. Multiple Sequence Alignment (MSA) with enhanced display
print("\n2. Performing Multiple Sequence Alignment (MSA)...")

# Create alignment
aligned_seqs = []
for seq in sequences:
    seq_str = str(seq.seq)
    # Introduce gaps to simulate alignment
    if seq.id == "Chicken":
        seq_str = seq_str[:10] + "--" + seq_str[12:]
    if seq.id == "Mouse":
        seq_str = seq_str[:15] + "-" + seq_str[16:]
    aligned_seqs.append(SeqRecord(Seq(seq_str), id=seq.id, description=seq.description))

alignment = MultipleSeqAlignment(aligned_seqs)

# Display alignment preview
print("\nAlignment Preview (first 30 positions):")
print("ID        Sequence")
for seq in alignment[:5]:  # Show first 5 sequences
    print(f"{seq.id[:8]:<8} {str(seq.seq[:30])}...")

# Save full alignment
aln_path = os.path.join(output_dir, "alignment.aln")
with open(aln_path, "w") as f:
    AlignIO.write(alignment, f, "clustal")
print(f"\nFull alignment saved to {aln_path}")
print(f"Alignment length: {alignment.get_alignment_length()} positions")
print(f"Number of sequences: {len(alignment)}")

# 3. Distance Matrix Calculation with enhanced display
print("\n3. Calculating distance matrix...")

calculator = DistanceCalculator('identity')
dm = calculator.get_distance(alignment)

# Convert to DataFrame
dm_df = pd.DataFrame(dm.matrix, index=dm.names, columns=dm.names)

# Display distance matrix
print("\nDistance Matrix (showing first 5x5 entries):")
print(dm_df.iloc[:5, :5].round(3))

print("\nDistance Statistics:")
print(dm_df.stack().describe().round(3))

# Highlight interesting pairs
human_col = dm_df['Human'].sort_values()
print("\nDistances from Human:")
print(human_col.round(3))
```

```
# Save distance matrix
dm_path = os.path.join(output_dir, "distance_matrix.csv")
dm_df.to_csv(dm_path)
print(f"\nFull distance matrix saved to {dm_path}")

# 4. Phylogenetic Tree Construction
print("\n4. Building phylogenetic trees...")

constructor = DistanceTreeConstructor()
nj_tree = constructor.nj(dm)
upgma_tree = constructor.upgma(dm)

# Save trees
nj_path = os.path.join(output_dir, "nj_tree.newick")
upgma_path = os.path.join(output_dir, "upgma_tree.newick")
Phylo.write(nj_tree, nj_path, "newick")
Phylo.write(upgma_tree, upgma_path, "newick")
print(f"Trees saved to:\n- {nj_path}\n- {upgma_path}")

# Visualize trees
plt.figure(figsize=(15, 8))
plt.subplot(1, 2, 1)
Phylo.draw(nj_tree, do_show=False)
plt.title("Neighbor-Joining Tree")

plt.subplot(1, 2, 2)
Phylo.draw(upgma_tree, do_show=False)
plt.title("UPGMA Tree")

# Save visualizations
tree_img_path = os.path.join(output_dir, "phylogenetic_trees.png")
tree_pdf_path = os.path.join(output_dir, "phylogenetic_trees.pdf")
plt.savefig(tree_img_path, dpi=300)
plt.savefig(tree_pdf_path)
plt.show()
print(f"\nTree visualizations saved to:\n- {tree_img_path}\n- {tree_pdf_path}")

# Print ASCII representation
print("\nNeighbor-Joining Tree (ASCII representation):")
Phylo.draw_ascii(nj_tree)

print("\nAnalysis complete! All files saved in:", output_dir)
```