# Python Guide

## Table of Contents

1. Basic Python Operations
2. Pandas
3. NumPy
4. Machine Learning with Scikit-learn
5. Bioinformatics Operations
6. Plotting with Matplotlib & Seaborn
7. Complete Example Bioinformatics Disease Prediction Workflow

---

## Basic Python Operations

### Strings

```python
# Creating and manipulating strings
text = "Hello, Python"
print(text.lower())           # "hello, python"
print(text.upper())           # "HELLO, PYTHON"
print(text.replace("Python", "World"))  # "Hello, World"
print(text.split(", "))       # ['Hello', 'Python']

# String formatting
name = "Alice"
age = 30
print(f"{name} is {age} years old")  # f-strings (recommended)

# Treating strings as lists (indexing and slicing)
text = "Python"
print(text[0])                # 'P' - first character
print(text[-1])               # 'n' - last character
print(text[2])                # 't' - third character
print(text[1:4])              # 'yth' - slice from index 1 to 3
print(text[:3])               # 'Pyt' - first 3 characters
print(text[3:])               # 'hon' - from index 3 to end
print(text[-3:])              # 'hon' - last 3 characters
print(text[::2])              # 'Pto' - every 2nd character
print(text[::-1])             # 'nohtyP' - reverse string

# Iterating through strings
for char in text:
    print(char, end=' ')      # P y t h o n
print()

# String length and membership
print(len(text))              # 6
print('t' in text)            # True
print('x' in text)            # False

# Converting string to list and back
```

```
char_list = list(text)        # ['P', 'y', 't', 'h', 'o', 'n']
back_to_string = ''.join(char_list)  # 'Python'
word_list = "Hello World".split()    # ['Hello', 'World']
sentence = ' '.join(word_list)       # 'Hello World'
```

## Lists

```python
# Creating and modifying lists
fruits = ["apple", "banana", "cherry"]
fruits.append("date")          # Add to end
fruits.insert(1, "blueberry") # Insert at index
fruits.remove("banana")       # Remove by value
popped = fruits.pop()         # Remove and return last item

# List operations
numbers = [1, 2, 3, 4, 5]
print(numbers[0])             # First element: 1
print(numbers[-1])            # Last element: 5
print(numbers[1:3])           # Slicing: [2, 3]
print(len(numbers))           # Length: 5

# List comprehension
squares = [x**2 for x in numbers]  # [1, 4, 9, 16, 25]
```

## Tuples

```python
# Immutable sequences
coordinates = (10, 20)
x, y = coordinates            # Unpacking
print(x, y)                   # 10 20

# Tuples are useful for fixed data
person = ("Alice", 30, "Engineer")
name, age, job = person
```

## Dictionaries

```python
# Key-value pairs
student = {
    "name": "Bob",
    "age": 22,
    "major": "Biology"
}

# Accessing and modifying
print(student["name"])        # "Bob"
student["age"] = 23           # Update value
student["gpa"] = 3.8          # Add new key-value
del student["major"]          # Remove key

# Dictionary methods
print(student.keys())         # dict_keys(['name', 'age', 'gpa'])
print(student.values())       # dict_values(['Bob', 23, 3.8])
print(student.items())        # Key-value pairs
```

```python
# Safe access
print(student.get("email", "Not provided"))  # Returns default if key missing
```

### Random Package

```python
import random

# Generate random numbers
random_int = random.randint(1, 10)          # Random integer between 1 and 10
random_float = random.random()               # Random float between 0 and 1
random_range = random.uniform(5.0, 10.0)   # Random float between 5 and 10

# Random choice from a list
colors = ["red", "blue", "green", "yellow"]
chosen_color = random.choice(colors)

# Random sample (multiple items without replacement)
sample = random.sample(colors, 2)           # Pick 2 random colors

# Shuffle a list (modifies in place)
numbers = [1, 2, 3, 4, 5]
random.shuffle(numbers)
print(numbers)                              # Randomly shuffled

# Set seed for reproducibility
random.seed(42)
print(random.randint(1, 100))              # Always gives same result with seed 42

# Generate random list
random_list = [random.randint(1, 100) for _ in range(10)]
```

---

## Pandas

### Installation

```python
# pip install pandas
import pandas as pd
```

### Basic Usage

#### Creating DataFrames

```python
# From dictionary
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David'],
    'Age': [25, 30, 35, 28],
    'City': ['New York', 'Paris', 'London', 'Tokyo'],
    'Salary': [70000, 80000, 90000, 75000]
}
df = pd.DataFrame(data)
print(df)

# From lists
df2 = pd.DataFrame([
    ['Alice', 25, 'New York'],
    ['Bob', 30, 'Paris']
], columns=['Name', 'Age', 'City'])

# From CSV (we'll cover this more below)
df3 = pd.DataFrame({'A': [1, 2], 'B': [3, 4]})
```

## Reading Files

```
# Read CSV
df = pd.read_csv('data.csv')
df = pd.read_csv('data.csv', sep=';')  # Custom separator
df = pd.read_csv('data.csv', index_col=0)  # Use first column as index

# Read Excel
df = pd.read_excel('data.xlsx', sheet_name='Sheet1')

# Read TSV (tab-separated)
df = pd.read_csv('data.tsv', sep='\t')

# Read with specific columns
df = pd.read_csv('data.csv', usecols=['Name', 'Age'])
```

## DataFrame Operations

### Viewing Data

```
df.head()           # First 5 rows
df.head(10)         # First 10 rows
df.tail()           # Last 5 rows
df.info()           # Data types and non-null counts
df.describe()       # Statistical summary
df.shape            # (rows, columns)
df.columns          # Column names
df.dtypes           # Data types of each column
```

### Adding Columns

```
# Add new column
df['Bonus'] = df['Salary'] * 0.1

# Add column with constant value
df['Country'] = 'USA'

# Add column based on condition
df['Senior'] = df['Age'] > 30

# Add multiple columns
df[['Tax', 'Net']] = df['Salary'] * 0.2, df['Salary'] * 0.8
```

### Removing Columns

```
# Drop single column
df = df.drop('Bonus', axis=1)
```

```
# or
df = df.drop(columns=['Bonus'])

# Drop multiple columns
df = df.drop(['Bonus', 'Country'], axis=1)

# Drop in place (modify original)
df.drop('Bonus', axis=1, inplace=True)
```

### Removing Rows

```
# Drop by index
df = df.drop(0)  # Drop first row
df = df.drop([0, 2])  # Drop multiple rows

# Drop by condition
df = df[df['Age'] > 25]  # Keep only rows where Age > 25

# Drop duplicates
df = df.drop_duplicates()
df = df.drop_duplicates(subset=['Name'])  # Based on specific column

# Drop rows with missing values
df = df.dropna()  # Drop any row with NaN
df = df.dropna(subset=['Age'])  # Drop rows where Age is NaN
```

### Updating Values

```
# Update single value
df.loc[0, 'Age'] = 26  # Update by label
df.iloc[0, 1] = 26     # Update by position

# Update entire column
df['Salary'] = df['Salary'] * 1.1  # 10% raise

# Update based on condition
df.loc[df['Age'] > 30, 'Salary'] = df['Salary'] * 1.2

# Replace values
df['City'] = df['City'].replace('New York', 'NYC')
df = df.replace({'Paris': 'Paris, France', 'London': 'London, UK'})
```

### Concatenating DataFrames

```
df1 = pd.DataFrame({'A': [1, 2], 'B': [3, 4]})
df2 = pd.DataFrame({'A': [5, 6], 'B': [7, 8]})

# Vertical concatenation (stack rows)
result = pd.concat([df1, df2], ignore_index=True)

# Horizontal concatenation (add columns)
result = pd.concat([df1, df2], axis=1)

# Merge (like SQL join)
df_left = pd.DataFrame({'key': ['A', 'B'], 'val1': [1, 2]})
df_right = pd.DataFrame({'key': ['A', 'B'], 'val2': [3, 4]})
result = pd.merge(df_left, df_right, on='key')
```

```
# Append rows (deprecated, use concat instead)
result = pd.concat([df1, df2], ignore_index=True)
```

## Extracting Rows and Columns

### Selecting Columns

```
# Single column (returns Series)
ages = df['Age']

# Multiple columns (returns DataFrame)
subset = df[['Name', 'Age']]

# Using loc
subset = df.loc[:, ['Name', 'Age']]
```

### Selecting Rows

```
# By index position (iloc)
first_row = df.iloc[0]          # First row
first_three = df.iloc[0:3]      # First 3 rows
last_row = df.iloc[-1]          # Last row

# By label (loc)
row = df.loc[0]                 # Row with index label 0
rows = df.loc[0:2]              # Rows 0 to 2 (inclusive)

# By condition
young_people = df[df['Age'] < 30]
high_earners = df[df['Salary'] > 75000]
combined = df[(df['Age'] > 25) & (df['Salary'] > 70000)]
```

### Selecting Rows and Columns

```
# loc[rows, columns]
value = df.loc[0, 'Name']               # Single value
subset = df.loc[0:2, ['Name', 'Age']]   # Multiple rows and columns

# iloc[rows, columns]
value = df.iloc[0, 1]                   # By position
subset = df.iloc[0:3, 0:2]              # Slice rows and columns
```

## Exporting to CSV

```
# Basic export
df.to_csv('output.csv')

# Without index
df.to_csv('output.csv', index=False)

# Custom separator
df.to_csv('output.tsv', sep='\t', index=False)
```

```python
# Export specific columns
df[['Name', 'Age']].to_csv('output.csv', index=False)

# Export to Excel
df.to_excel('output.xlsx', index=False, sheet_name='Data')
```

## Handling Missing Values

```python
# Check for missing values
print(df.isnull().sum())           # Count NaN values per column
print(df.isna().sum())             # Same as isnull()

# Visualize missing data
print(df.info())                   # Shows non-null counts

# Drop missing values
df_clean = df.dropna()             # Drop any row with NaN
df_clean = df.dropna(axis=1)       # Drop any column with NaN
df_clean = df.dropna(subset=['Age'])  # Drop rows where Age is NaN
df_clean = df.dropna(thresh=3)     # Keep rows with at least 3 non-NaN values

# Fill missing values
df['Age'].fillna(df['Age'].mean(), inplace=True)  # Fill with mean
df['City'].fillna('Unknown', inplace=True)        # Fill with constant
df.fillna(method='ffill', inplace=True)           # Forward fill
df.fillna(method='bfill', inplace=True)           # Backward fill

# Replace specific values
df.replace('N/A', np.nan, inplace=True)           # Replace 'N/A' with NaN
```

## Encoding Categorical Variables

```python
# Label Encoding (convert categories to numbers)
from sklearn.preprocessing import LabelEncoder

df = pd.DataFrame({
    'Color': ['Red', 'Blue', 'Green', 'Red', 'Blue'],
    'Size': ['Small', 'Large', 'Medium', 'Small', 'Large']
})

le = LabelEncoder()
df['Color_Encoded'] = le.fit_transform(df['Color'])
# Red=2, Blue=0, Green=1 (alphabetical order)

# One-Hot Encoding (create binary columns)
df_encoded = pd.get_dummies(df, columns=['Color', 'Size'])
print(df_encoded)
# Creates: Color_Blue, Color_Green, Color_Red, Size_Large, Size_Medium, Size_Small

# Manual mapping
gender_map = {'Male': 0, 'Female': 1}
df['Gender_Encoded'] = df['Gender'].map(gender_map)

# One-hot encoding with prefix
df_encoded = pd.get_dummies(df, columns=['Color'], prefix='Col')
# Creates: Col_Blue, Col_Green, Col_Red
```

## NumPy

### Installation

```
# pip install numpy
import numpy as np
```

### Basic Usage

```
# Creating arrays
arr = np.array([1, 2, 3, 4, 5])
arr2d = np.array([[1, 2, 3], [4, 5, 6]])

# Common arrays
zeros = np.zeros((3, 4))        # 3x4 array of zeros
ones = np.ones((2, 3))          # 2x3 array of ones
arange = np.arange(0, 10, 2)    # [0, 2, 4, 6, 8]
linspace = np.linspace(0, 1, 5) # 5 evenly spaced values

# Array operations
arr = np.array([1, 2, 3, 4])
print(arr + 10)                 # [11, 12, 13, 14]
print(arr * 2)                  # [2, 4, 6, 8]
print(arr ** 2)                 # [1, 4, 9, 16]

# Statistical operations
print(arr.mean())               # 2.5
print(arr.std())                # Standard deviation
print(arr.sum())                # 10
print(arr.min(), arr.max())     # 1, 4
```

### Log2 for Fold Change

```
# Log2 is commonly used in genomics for fold change calculation
# Fold change = treatment / control

# Example: Gene expression data
control = np.array([100, 200, 150, 300])
treatment = np.array([200, 400, 150, 150])

# Calculate fold change
fold_change = treatment / control
print(f"Fold change: {fold_change}")
# Output: [2.0, 2.0, 1.0, 0.5]

# Calculate log2 fold change
log2_fc = np.log2(fold_change)
print(f"Log2 fold change: {log2_fc}")
# Output: [1.0, 1.0, 0.0, -1.0]

# Interpretation:
# log2(FC) = 1  means 2x upregulated
# log2(FC) = 0  means no change
# log2(FC) = -1 means 2x downregulated

# Avoid log of zero by adding pseudocount
control_pseudo = control + 1
```

```
treatment_pseudo = treatment + 1
log2_fc_safe = np.log2(treatment_pseudo / control_pseudo)

# Using with Pandas
import pandas as pd
df = pd.DataFrame({
    'Gene': ['GeneA', 'GeneB', 'GeneC', 'GeneD'],
    'Control': control,
    'Treatment': treatment
})
df['FoldChange'] = df['Treatment'] / df['Control']
df['Log2FC'] = np.log2(df['FoldChange'])
print(df)
```

---

## Machine Learning with Scikit-learn

### Installation

```
# pip install scikit-learn
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
from sklearn.preprocessing import LabelEncoder
```

### Train-Test Split

```
import pandas as pd
from sklearn.model_selection import train_test_split

# Sample data
df = pd.DataFrame({
    'Feature1': [1, 2, 3, 4, 5, 6, 7, 8],
    'Feature2': [2, 4, 6, 8, 10, 12, 14, 16],
    'Target': [0, 0, 1, 1, 0, 1, 1, 0]
})

# Separate features (X) and target (y)
X = df[['Feature1', 'Feature2']]
y = df['Target']

# 80-20 split (80% training, 20% testing)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

print(f"Training set size: {len(X_train)}")
print(f"Testing set size: {len(X_test)}")

# 70-30 split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42
)

# Stratified split (maintains class distribution)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)
```

## Support Vector Machine (SVM)

```
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, classification_report

# Create and train SVM model
svm_model = SVC(kernel='linear', random_state=42)
svm_model.fit(X_train, y_train)

# Make predictions
y_pred = svm_model.predict(X_test)

# Evaluate accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy * 100:.2f}%")

# Detailed classification report
print(classification_report(y_test, y_pred))

# Different SVM kernels
svm_rbf = SVC(kernel='rbf', random_state=42)      # Radial basis function
svm_poly = SVC(kernel='poly', degree=3, random_state=42)  # Polynomial
svm_sigmoid = SVC(kernel='sigmoid', random_state=42)      # Sigmoid

# Predict single sample
new_sample = [[5, 10]]
prediction = svm_model.predict(new_sample)
print(f"Prediction: {prediction[0]}")
```

## Model Evaluation

```
from sklearn.metrics import confusion_matrix, classification_report
import seaborn as sns
import matplotlib.pyplot as plt

# Confusion matrix
cm = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:")
print(cm)

# Visualize confusion matrix
plt.figure(figsize=(6, 4))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix')
plt.show()

# Classification report (precision, recall, f1-score)
print(classification_report(y_test, y_pred))

# Individual metrics
from sklearn.metrics import precision_score, recall_score, f1_score

precision = precision_score(y_test, y_pred, average='weighted')
recall = recall_score(y_test, y_pred, average='weighted')
```

```
f1 = f1_score(y_test, y_pred, average='weighted')

print(f"Precision: {precision:.2f}")
print(f"Recall: {recall:.2f}")
print(f"F1-Score: {f1:.2f}")
```

---

## Bioinformatics Operations

### DNA Sequence Analysis

```
# DNA complement and reverse complement
def complement(seq):
    """Return the complement of a DNA sequence"""
    comp = {'A': 'T', 'T': 'A', 'G': 'C', 'C': 'G', '-': '-'}
    return ''.join([comp[base] for base in seq])

def reverse_complement(seq):
    """Return the reverse complement of a DNA sequence"""
    return complement(seq)[::-1]

dna = "AGCTTTCGAATTCGA"
print(f"Original: {dna}")
print(f"Complement: {complement(dna)}")
print(f"Reverse complement: {reverse_complement(dna)}")

# GC Content calculation
def gc_content(seq):
    """Calculate GC content percentage"""
    seq = seq.upper()
    gc_count = seq.count('G') + seq.count('C')
    total = len(seq)
    return (gc_count / total) * 100 if total > 0 else 0

sequence = "AGCTTTCGAATTCGA"
print(f"GC Content: {gc_content(sequence):.2f}%")
```

### Finding Palindromic Sequences

```
def is_palindrome(seq):
    """Check if a sequence is palindromic"""
    return seq == seq[::-1]

def find_palindromes(sequence):
    """Find all unique palindromic substrings"""
    palindromes = set()
    n = len(sequence)

    # Check all possible substrings
    for i in range(n):
        for j in range(i + 1, n + 1):
            substring = sequence[i:j]
            if len(substring) >= 2 and is_palindrome(substring):
                palindromes.add(substring)

    return sorted(palindromes, key=len, reverse=True)

# Example usage
```

```python
dna = "AGCTTTCGAATTCGA"
palindromes = find_palindromes(dna)

print("Palindromic sequences found:")
for pal in palindromes:
    gc = gc_content(pal)
    print(f"{pal} - Length: {len(pal)}, GC Content: {gc:.2f}%")

# Find longest palindrome
if palindromes:
    longest = palindromes[0]
    print(f"\nLongest palindrome: {longest}")
```

## Pairwise Sequence Alignment

```python
# Simple pairwise alignment similarity score
def pairwise_similarity(seq1, seq2):
    """Calculate similarity between two sequences (0-100%)"""
    if len(seq1) ≠ len(seq2):
        # Align to same length (simple padding)
        max_len = max(len(seq1), len(seq2))
        seq1 = seq1.ljust(max_len, '-')
        seq2 = seq2.ljust(max_len, '-')

    matches = sum(1 for a, b in zip(seq1, seq2) if a == b)
    return (matches / len(seq1)) * 100

# Calculate pairwise similarities for multiple sequences
sequences = ["AGCTTTCGAA", "AGCTTTCGAT", "AGCTAACGAA"]

print("Pairwise Alignment Similarities:")
for i in range(len(sequences)):
    for j in range(i + 1, len(sequences)):
        similarity = pairwise_similarity(sequences[i], sequences[j])
        print(f"Seq{i+1} vs Seq{j+1}: {similarity:.2f}%")

# Using Biopython for more advanced alignment (if available)
# from Bio import pairwise2
# from Bio.pairwise2 import format_alignment
#
# alignments = pairwise2.align.globalxx(seq1, seq2)
# print(format_alignment(*alignments[0]))
```

## Distance Matrix for Phylogenetic Analysis

```python
import numpy as np
import pandas as pd

def create_distance_matrix(sequences):
    """Create distance matrix from sequences"""
    n = len(sequences)
    distances = np.zeros((n, n))

    for i in range(n):
        for j in range(n):
            if i ≠ j:
                similarity = pairwise_similarity(sequences[i], sequences[j])
                distances[i][j] = 100 - similarity  # Convert to distance
    return distances

sequences = ["AGCTTTCGAA", "AGCTTTCGAT", "AGCTAACGAA", "TGCTAACGAA"]
distances = create_distance_matrix(sequences)

# Convert to DataFrame for better visualization
df_distances = pd.DataFrame(
    distances,
    columns=[f"Seq{i+1}" for i in range(len(sequences))],
    index=[f"Seq{i+1}" for i in range(len(sequences))]
)

print("Distance Matrix:")
print(df_distances)

# Visualize as heatmap
import matplotlib.pyplot as plt
import seaborn as sns

plt.figure(figsize=(8, 6))
sns.heatmap(df_distances, annot=True, fmt='.1f', cmap='YlOrRd')
plt.title('Sequence Distance Matrix')
plt.show()
```

---

## Plotting with Matplotlib & Seaborn

### Installation

```
# pip install matplotlib seaborn
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
import numpy as np
```

### Sample Data for Plotting

```
# Create sample dataset
np.random.seed(42)
data = pd.DataFrame({
    'Category': ['A', 'B', 'C', 'D', 'E'] * 20,
    'Value': np.random.randn(100) * 10 + 50,
    'Score': np.random.randn(100) * 15 + 75,
    'Group': np.random.choice(['Group1', 'Group2', 'Group3'], 100)
})
```

### Bar Chart

```
# Matplotlib bar chart
categories = ['A', 'B', 'C', 'D']
values = [23, 45, 56, 78]

plt.figure(figsize=(8, 5))
plt.bar(categories, values, color='skyblue', edgecolor='black')
plt.xlabel('Categories')
plt.ylabel('Values')
```

```python
plt.title('Simple Bar Chart')
plt.grid(axis='y', alpha=0.3)
plt.show()

# Seaborn bar chart (with error bars)
plt.figure(figsize=(8, 5))
sns.barplot(data=data, x='Category', y='Value', errorbar='sd')
plt.title('Bar Chart with Error Bars')
plt.show()

# Grouped bar chart
plt.figure(figsize=(10, 6))
sns.barplot(data=data, x='Category', y='Value', hue='Group')
plt.title('Grouped Bar Chart')
plt.legend(title='Group')
plt.show()
```

## Box Plot

```python
# Matplotlib box plot
plt.figure(figsize=(8, 5))
data_for_box = [data[data['Category'] == cat]['Value'].values
                for cat in ['A', 'B', 'C', 'D', 'E']]
plt.boxplot(data_for_box, labels=['A', 'B', 'C', 'D', 'E'])
plt.xlabel('Category')
plt.ylabel('Value')
plt.title('Box Plot')
plt.grid(axis='y', alpha=0.3)
plt.show()

# Seaborn box plot (recommended - easier and prettier)
plt.figure(figsize=(10, 6))
sns.boxplot(data=data, x='Category', y='Value', hue='Group')
plt.title('Box Plot by Category and Group')
plt.show()

# Horizontal box plot
plt.figure(figsize=(8, 6))
sns.boxplot(data=data, y='Category', x='Value')
plt.title('Horizontal Box Plot')
plt.show()
```

## Scatter Plot

```python
# Matplotlib scatter plot
plt.figure(figsize=(8, 6))
plt.scatter(data['Value'], data['Score'], alpha=0.6, c='blue', edgecolors='black')
plt.xlabel('Value')
plt.ylabel('Score')
plt.title('Scatter Plot')
plt.grid(alpha=0.3)
plt.show()

# Seaborn scatter plot with regression line
plt.figure(figsize=(8, 6))
sns.regplot(data=data, x='Value', y='Score', scatter_kws={'alpha':0.5})
plt.title('Scatter Plot with Regression Line')
plt.show()
```

```
# Scatter plot with color by category
plt.figure(figsize=(10, 6))
sns.scatterplot(data=data, x='Value', y='Score', hue='Group',
                style='Group', s=100, alpha=0.7)
plt.title('Scatter Plot Colored by Group')
plt.show()
```

## Violin Plot

```
# Seaborn violin plot
plt.figure(figsize=(10, 6))
sns.violinplot(data=data, x='Category', y='Value')
plt.title('Violin Plot')
plt.show()

# Violin plot with groups
plt.figure(figsize=(12, 6))
sns.violinplot(data=data, x='Category', y='Value', hue='Group', split=False)
plt.title('Violin Plot by Group')
plt.show()

# Split violin plot (compare two groups)
data_two_groups = data[data['Group'].isin(['Group1', 'Group2'])]
plt.figure(figsize=(10, 6))
sns.violinplot(data=data_two_groups, x='Category', y='Value',
               hue='Group', split=True, palette='Set2')
plt.title('Split Violin Plot')
plt.show()
```

## Heatmap

```
# Create correlation matrix
numeric_data = data[['Value', 'Score']].copy()
numeric_data['Random'] = np.random.randn(len(data)) * 5 + 25
correlation = numeric_data.corr()

# Seaborn heatmap
plt.figure(figsize=(8, 6))
sns.heatmap(correlation, annot=True, cmap='coolwarm', center=0,
            square=True, linewidths=1, cbar_kws={"shrink": 0.8})
plt.title('Correlation Heatmap')
plt.show()

# Heatmap with custom data
matrix_data = np.random.rand(5, 5)
plt.figure(figsize=(8, 6))
sns.heatmap(matrix_data, annot=True, fmt='.2f', cmap='viridis',
            xticklabels=['A', 'B', 'C', 'D', 'E'],
            yticklabels=['Gene1', 'Gene2', 'Gene3', 'Gene4', 'Gene5'])
plt.title('Gene Expression Heatmap')
plt.show()

# Clustered heatmap (hierarchical clustering)
plt.figure(figsize=(10, 8))
sns.clustermap(correlation, annot=True, cmap='coolwarm', center=0,
               linewidths=1, figsize=(10, 8))
plt.title('Clustered Heatmap')
plt.show()
```

### Customizing Plots

```python
# Setting style
sns.set_style("whitegrid")  # Options: white, dark, whitegrid, darkgrid, ticks

# Setting color palette
sns.set_palette("husl")     # Options: deep, muted, pastel, bright, dark, colorblind

# Multiple subplots
fig, axes = plt.subplots(2, 2, figsize=(12, 10))

axes[0, 0].bar(['A', 'B', 'C'], [1, 2, 3])
axes[0, 0].set_title('Bar Chart')

axes[0, 1].boxplot([data['Value'], data['Score']])
axes[0, 1].set_title('Box Plot')

axes[1, 0].scatter(data['Value'], data['Score'], alpha=0.5)
axes[1, 0].set_title('Scatter Plot')

sns.violinplot(data=data, x='Category', y='Value', ax=axes[1, 1])
axes[1, 1].set_title('Violin Plot')

plt.tight_layout()
plt.show()

# Saving figures
plt.savefig('my_plot.png', dpi=300, bbox_inches='tight')
plt.savefig('my_plot.pdf')  # Vector format for publications
```

---

## Complete Example: Bioinformatics Disease Prediction Workflow

This example demonstrates a complete machine learning pipeline for disease prediction using protein sequences and clinical data, inspired by bioinformatics research tasks.

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix

# ============================================================================
# PART 1: DATA LOADING AND PREPROCESSING
# ============================================================================

# (a) Read the dataset
df = pd.read_csv('disease_data.csv')
print("Dataset loaded successfully!")
print(f"Shape: {df.shape}")
print(df.head())

# (b) Data preprocessing
print("\n=== Data Preprocessing ===")

# Check for missing values
```

```python
print("\nMissing values per column:")
print(df.isnull().sum())

# Handle missing values
# Option 1: Drop rows with missing values
# df = df.dropna()

# Option 2: Fill missing values
if 'age' in df.columns:
    df['age'].fillna(df['age'].mean(), inplace=True)
if 'sex' in df.columns:
    df['sex'].fillna(df['sex'].mode()[0], inplace=True)  # Fill with most frequent

# Encode categorical variables
le_sex = LabelEncoder()
le_disease = LabelEncoder()
le_severity = LabelEncoder()

if 'sex' in df.columns:
    df['sex_encoded'] = le_sex.fit_transform(df['sex'])

if 'disease' in df.columns:
    df['disease_encoded'] = le_disease.fit_transform(df['disease'])

if 'severity' in df.columns:
    df['severity_encoded'] = le_severity.fit_transform(df['severity'])

# Create binary target: diseased (1) or healthy (0)
if 'disease' in df.columns:
    df['is_diseased'] = (df['disease'] != 'Healthy').astype(int)

print("\nData after preprocessing:")
print(df.head())

# (c) Split dataset into training and testing sets (80-20 split)
print("\n=== Train-Test Split ===")

# Select features for training
feature_cols = ['age', 'sex_encoded', 'severity_encoded']
X = df[feature_cols]
y = df['is_diseased']

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)

print(f"Training set size: {len(X_train)} samples")
print(f"Testing set size: {len(X_test)} samples")
print(f"Class distribution in training: {y_train.value_counts().to_dict()}")

# ============================================================================
# PART 2: DATA VISUALIZATION
# ============================================================================

# (d) Create heatmap for correlation analysis
print("\n=== Correlation Analysis ===")

correlation_data = df[['age', 'sex_encoded', 'severity_encoded']].corr()

plt.figure(figsize=(8, 6))
sns.heatmap(correlation_data, annot=True, cmap='coolwarm', center=0,
            square=True, linewidths=1, cbar_kws={"shrink": 0.8})
plt.title('Correlation Heatmap: Age, Gender, and Disease Severity')
plt.tight_layout()
plt.savefig('correlation_heatmap.png', dpi=300)
plt.show()

# (e) Insights from visualization
print("\n=== Distribution Analysis ===")

# Disease distribution by age group
df['age_group'] = pd.cut(df['age'], bins=[0, 30, 50, 70, 100],
                          labels=['Young', 'Middle', 'Senior', 'Elderly'])

disease_by_age = pd.crosstab(df['age_group'], df['disease'])
print("\nDisease distribution by age group:")
print(disease_by_age)

# Disease distribution by gender
disease_by_gender = pd.crosstab(df['sex'], df['disease'])
print("\nDisease distribution by gender:")
print(disease_by_gender)

# Visualize distributions
fig, axes = plt.subplots(1, 2, figsize=(14, 5))

disease_by_age.plot(kind='bar', ax=axes[0], colormap='viridis')
axes[0].set_title('Disease Distribution by Age Group')
axes[0].set_xlabel('Age Group')
axes[0].set_ylabel('Count')
axes[0].legend(title='Disease', bbox_to_anchor=(1.05, 1))

disease_by_gender.plot(kind='bar', ax=axes[1], colormap='plasma')
axes[1].set_title('Disease Distribution by Gender')
axes[1].set_xlabel('Gender')
axes[1].set_ylabel('Count')
axes[1].legend(title='Disease', bbox_to_anchor=(1.05, 1))

plt.tight_layout()
plt.savefig('disease_distributions.png', dpi=300)
plt.show()

# ============================================================================
# PART 3: MACHINE LEARNING MODEL
# ============================================================================

# (f) Train SVM classification model
print("\n=== Training SVM Model ===")

svm_model = SVC(kernel='rbf', random_state=42, probability=True)
svm_model.fit(X_train, y_train)

print("Model training completed!")

# (g) Evaluate model accuracy
print("\n=== Model Evaluation ===")

y_pred = svm_model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)

print(f"Accuracy: {accuracy * 100:.2f}%")
print("\nClassification Report:")
print(classification_report(y_test, y_pred))

# Confusion matrix
cm = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(6, 4))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
```

```python
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title(f'Confusion Matrix (Accuracy: {accuracy*100:.2f}%)')
plt.tight_layout()
plt.savefig('confusion_matrix.png', dpi=300)
plt.show()

# (h) Discuss challenges and solutions
print("\n=== Challenges in Disease Prediction ===")
challenges = """
Challenges in predicting diseases from DNA sequences:

1. High Dimensionality: DNA sequences contain thousands of base pairs
   Solution: Use dimensionality reduction (PCA) or feature selection

2. Class Imbalance: Some diseases are rare
   Solution: Use SMOTE, weighted loss functions, or stratified sampling

3. Sequence Variability: Mutations and variations in sequences
   Solution: Use alignment algorithms and consider mutation tolerance

4. Limited Training Data: Insufficient samples for rare diseases
   Solution: Data augmentation, transfer learning, or ensemble methods

5. Overfitting: Model learns noise instead of patterns
   Solution: Cross-validation, regularization, and early stopping

6. Interpretability: Understanding which features drive predictions
   Solution: Use feature importance analysis and SHAP values
"""
print(challenges)

# ============================================================================
# PART 4: SEQUENCE ANALYSIS
# ============================================================================

# (i) Predict disease for new sequences
print("\n=== Predicting for New Sequences ===")

# Read sequences from file
with open('sequences.txt', 'r') as f:
    sequences = [line.strip() for line in f.readlines() if line.strip()]

# Create features from sequences (example: length, GC content)
def extract_features(sequence):
    length = len(sequence)
    gc_content = (sequence.count('G') + sequence.count('C')) / len(sequence) * 100
    at_content = (sequence.count('A') + sequence.count('T')) / len(sequence) * 100
    return [length, gc_content, at_content]

sequence_features = [extract_features(seq) for seq in sequences]

# For this example, we'll use dummy features matching our model
# In real scenario, you'd extract relevant features from sequences
dummy_predictions = []
for i, seq in enumerate(sequences):
    # Create dummy features (age, sex, severity)
    dummy_features = [[35 + i*5, i % 2, i % 3]]
    pred = svm_model.predict(dummy_features)
    pred_proba = svm_model.predict_proba(dummy_features)

    dummy_predictions.append({
        'sequence': seq[:20] + '...',  # Show first 20 chars
        'predicted_disease': pred[0],
        'confidence': max(pred_proba[0]) * 100
    })
    print(f"Sequence {i+1}: Disease={pred[0]}, Confidence={max(pred_proba[0])*100:.1f}%")

# (j) Calculate pairwise alignment similarity
print("\n=== Pairwise Alignment Similarity ===")

def pairwise_similarity(seq1, seq2):
    """Calculate similarity percentage between two sequences"""
    max_len = max(len(seq1), len(seq2))
    seq1 = seq1.ljust(max_len, '-')
    seq2 = seq2.ljust(max_len, '-')
    matches = sum(1 for a, b in zip(seq1, seq2) if a == b)
    return (matches / max_len) * 100

# Group sequences by predicted disease
pred_diseases = [p['predicted_disease'] for p in dummy_predictions]
for disease in set(pred_diseases):
    disease_seqs = [sequences[i] for i, d in enumerate(pred_diseases) if d == disease]

    if len(disease_seqs) > 1:
        print(f"\nPairwise similarities for disease {disease}:")
        for i in range(len(disease_seqs)):
            for j in range(i + 1, len(disease_seqs)):
                similarity = pairwise_similarity(disease_seqs[i], disease_seqs[j])
                print(f"  Seq{i+1} vs Seq{j+1}: {similarity:.2f}%")

# (k) Construct distance matrix for phylogenetic tree
print("\n=== Distance Matrix for UPGMA ===")

def create_distance_matrix(sequences):
    n = len(sequences)
    distances = np.zeros((n, n))
    for i in range(n):
        for j in range(n):
            if i != j:
                similarity = pairwise_similarity(sequences[i], sequences[j])
                distances[i][j] = 100 - similarity
    return distances

distance_matrix = create_distance_matrix(sequences[:5])  # Use first 5 sequences

df_distances = pd.DataFrame(
    distance_matrix,
    columns=[f"Seq{i+1}" for i in range(len(distance_matrix))],
    index=[f"Seq{i+1}" for i in range(len(distance_matrix))]
)

print("\nDistance Matrix:")
print(df_distances)

# Visualize distance matrix
plt.figure(figsize=(8, 6))
sns.heatmap(df_distances, annot=True, fmt='.1f', cmap='YlOrRd', cbar_kws={'label': 'Distance'})
plt.title('Sequence Distance Matrix (for UPGMA Tree Construction)')
plt.tight_layout()
plt.savefig('distance_matrix.png', dpi=300)
plt.show()

print("\n=== Analysis Complete ===")
print("Observations from phylogenetic analysis:")
print("- Sequences with lower distance values are more similar")
print("- Clustered sequences likely share common ancestry")
print("- Distance matrix can be used for UPGMA tree construction")
```

```python
# ============================================================================
# PART 5: DIFFERENTIAL EXPRESSION ANALYSIS
# ============================================================================

print("\n=== Differential Expression Analysis ===")

# (a) Load RNA-Seq dataset
dea_df = pd.read_csv('dea.csv')
print("DEA dataset loaded!")
print(f"Shape: {dea_df.shape}")

# (b) Display summary description
print("\nDataset Summary:")
print(dea_df.describe())
print("\nDataset Info:")
print(dea_df.info())

# (c) Compute mean expression for treated and control
# Assuming columns: Gene, Control_1, Control_2, Control_3, Treated_1, Treated_2, Treated_3
control_cols = [col for col in dea_df.columns if 'Control' in col]
treated_cols = [col for col in dea_df.columns if 'Treated' in col]

dea_df['Control_Mean'] = dea_df[control_cols].mean(axis=1)
dea_df['Treated_Mean'] = dea_df[treated_cols].mean(axis=1)

print("\nMean expression calculated:")
print(dea_df[['Gene', 'Control_Mean', 'Treated_Mean']].head())

# (d) Calculate fold change
dea_df['Fold_Change'] = dea_df['Treated_Mean'] / dea_df['Control_Mean']
dea_df['Log2_FC'] = np.log2(dea_df['Fold_Change'])

print("\nFold change calculated:")
print(dea_df[['Gene', 'Fold_Change', 'Log2_FC']].head())

# (e) Identify up-regulated and down-regulated genes
up_regulated = dea_df[dea_df['Fold_Change'] > 1.6]
down_regulated = dea_df[dea_df['Fold_Change'] < 0.67]

print(f"\nUp-regulated genes (FC > 1.6): {len(up_regulated)}")
print(f"Down-regulated genes (FC < 0.67): {len(down_regulated)}")

print("\nTop 5 up-regulated genes:")
print(up_regulated.nlargest(5, 'Fold_Change')[['Gene', 'Fold_Change', 'Log2_FC']])

print("\nTop 5 down-regulated genes:")
print(down_regulated.nsmallest(5, 'Fold_Change')[['Gene', 'Fold_Change', 'Log2_FC']])

# (f) Plot fold changes in a bar chart
plt.figure(figsize=(12, 6))

# Show top 10 up and down regulated genes
top_up = up_regulated.nlargest(10, 'Fold_Change')
top_down = down_regulated.nsmallest(10, 'Fold_Change')
plot_data = pd.concat([top_down, top_up])

colors = ['red' if fc < 1 else 'green' for fc in plot_data['Fold_Change']]
plt.bar(range(len(plot_data)), plot_data['Log2_FC'], color=colors, alpha=0.7, edgecolor='black')
plt.axhline(y=np.log2(1.6), color='green', linestyle='--', label='Up-regulation threshold')
plt.axhline(y=np.log2(0.67), color='red', linestyle='--', label='Down-regulation threshold')
plt.axhline(y=0, color='black', linestyle='-', linewidth=0.5)
plt.xticks(range(len(plot_data)), plot_data['Gene'], rotation=45, ha='right')
plt.xlabel('Gene')
plt.ylabel('Log2 Fold Change')
plt.title('Differential Gene Expression (Top 10 Up and Down Regulated)')
plt.legend()
plt.tight_layout()
plt.savefig('fold_change_barplot.png', dpi=300)
plt.show()

# Volcano plot (if p-values available)
if 'p_value' in dea_df.columns:
    dea_df['neg_log10_p'] = -np.log10(dea_df['p_value'])

    plt.figure(figsize=(10, 6))
    plt.scatter(dea_df['Log2_FC'], dea_df['neg_log10_p'], alpha=0.5, s=10)
    plt.axvline(x=np.log2(1.6), color='green', linestyle='--')
    plt.axvline(x=np.log2(0.67), color='red', linestyle='--')
    plt.axhline(y=-np.log10(0.05), color='blue', linestyle='--')
    plt.xlabel('Log2 Fold Change')
    plt.ylabel('-Log10(p-value)')
    plt.title('Volcano Plot')
    plt.tight_layout()
    plt.savefig('volcano_plot.png', dpi=300)
    plt.show()

print("\n=== Complete Analysis Finished ===")
```

## DNA Palindrome Analysis Example

```
"""
Find palindromic substrings and calculate their GC content
"""

def find_palindromes_with_gc(sequence):
    """Find all palindromes and calculate GC content"""
    palindromes = {}
    n = len(sequence)

    # Find all substrings
    for i in range(n):
        for j in range(i + 1, n + 1):
            substring = sequence[i:j]

            # Check if palindrome
            if len(substring) >= 2 and substring == substring[::-1]:
                # Calculate GC content
                gc_count = substring.count('G') + substring.count('C')
                gc_percentage = (gc_count / len(substring)) * 100

                palindromes[substring] = {
                    'length': len(substring),
                    'gc_content': gc_percentage
                }

    return palindromes

# Analyze DNA sequence
dna_sequence = "AGCTTTCGAATTCGA"
print(f"Analyzing sequence: {dna_sequence}\n")

palindromes = find_palindromes_with_gc(dna_sequence)

# Sort by length (longest first)
```

```python
sorted_palindromes = sorted(palindromes.items(),
                            key=lambda x: x[1]['length'],
                            reverse=True)

print("Unique Palindromic Substrings:")
print("-" * 50)
for seq, info in sorted_palindromes:
    print(f"{seq:15} | Length: {info['length']:2} | GC: {info['gc_content']:.1f}%")

# Find longest palindrome
if sorted_palindromes:
    longest = sorted_palindromes[0]
    print(f"\nLongest palindrome: {longest[0]}")
    print(f"Length: {longest[1]['length']}")
    print(f"GC Content: {longest[1]['gc_content']:.1f}%")
```

---

## Tips and Best Practices

1. **Pandas**: Always use `inplace=False` (default) and reassign to avoid confusion. Use `df.copy()` when creating new DataFrames from subsets.
2. **NumPy**: Add pseudocounts before log transformations to avoid errors with zeros: `np.log2(values + 1)`
3. **Plotting**: Use Seaborn for statistical plots and Matplotlib for fine control. Always label axes and add titles.
4. **Data Analysis**: Always explore data with `head()`, `info()`, and `describe()` before analysis.
5. **File Handling**: Use `index=False` when exporting to CSV unless you need the index.
6. **Performance**: For large datasets, use vectorized operations instead of loops whenever possible.