Python and Unix For Bioinformatics

You are a biologist who wants to learn Python. But you are a busy person, don’t have much time.

This worksheet is a good option for you! You can go through the chapters of this series in a couple of afternoon, and you’ll have a pretty good idea of programming in Bioinformatics.

What you have to do is manually type each commands in your Python terminal and get the output. Don’t just read or skim through the materials, reproduce them too!

And try to do the exercises. Learning to programming is like building muscle, you have to do exercise!

There are four chapters in this PDF:

Chapter 1: Python for Biologists

Chapter 2: Introduction to Functions in Python

Chapter 3: Data Science with Python

Chapter 4: Introduction to Object-Orientered Programming (OOP) in Python

Chapter 5: Introduction to Unix for Bioinformatics

I hope this will be helpful.

Yours Truely,

Arafat Rahman, PhD

http://arftrhmn.net

Chapter 1: Python for Biologists

This worksheet is an extended, hands-on journey through Python basics for biologists. Designed for 1 hours of interactive learning with examples and exercises. Each block is followed by a small task or thought experiment.

---

1. Python as a Calculator

```
5 + 2        # 7
9 - 4        # 5
3 * 4        # 12
10 / 3       # 3.33...
10 // 3      # 3 (integer division)
2 ** 4       # 16
17 % 5       # 2
```

Common Math Operations

- `+`: addition
- `-`: subtraction
- `*`: multiplication
- `/`: float division
- `//`: integer division
- `%`: modulo (remainder)
- `**`: exponentiation

Task: Calculate molecular weight of a DNA strand using `sum()` and dictionary lookup.

```
dna_base_masses = {'A': 313.2, 'T': 304.2, 'G': 329.2, 'C': 289.2}
sequence = "ATGCGT"
mass = sum(dna_base_masses[base] for base in sequence)
print("DNA mass:", mass)
```

---

2. math Module

```
import math
print(math.log10(2000000))  # Log10 of population
print(math.sqrt(36))        # Square root
print(math.exp(2))          # Exponential
```

Common Functions in `math`

- `math.sqrt(x)`: square root
- `math.exp(x)`: e^x
- `math.log(x, base)`: logarithm with custom base (base 10 = `math.log10()`)
- `math.pi`: π constant

** Task:** Use `math.log()` with different bases (e.g., natural log).

---

3. random Module

```
import random
print(random.choice(["A", "T", "G", "C"]))
print(''.join(random.choices(["A", "T", "G", "C"], k=10)))
```

Useful `random` Methods & Examples

- `random.random()`: random float [0.0, 1.0)
- `random.randint(a, b)`: random int between a and b
- `random.choice(list)`: pick one item
- `random.choices(list, k=n)`: pick n items with replacement
- `random.shuffle(list)`: shuffle in-place

 Task: Generate a 50-base random DNA sequence.

---

4. Data Types and Formatting

Python has several built-in data types. Here are the most commonly used ones:

Integers (`int`)

Whole numbers without a decimal point.

```
a = 42
print(type(a))  # <class 'int'>
```

Useful `int` methods/functions:

```
print(bin(10))  # '0b1010' (binary)
print(int("1010"))  # Convert string to integer
```

Floating-point numbers (`float`)

Numbers with decimal points.

```
b = 3.14
print(type(b))  # <class 'float'>
```

Methods/Functions:

```
print(round(3.14159, 2))  # 3.14
print(float("3.5"))  # Convert string to float
```

Strings (`str`)

Text or characters inside quotes.

```
name = "lacZ"
print(type(name))  # <class 'str'>
```

Useful `str` methods:

```
print(name.upper())   # 'LACZ'
print(name.lower())   # 'lacz'
print(name.startswith("la"))  # True
print(name.replace("Z", "Y"))  # 'lacY'
```

String Formatting Examples

```
gene = "trpE"
expr = 8.4567
print(f"{gene} expression: {expr:.2f}")
print("%s expression: %.2f" % (gene, expr))
```

Comparing Data Types

```
a = 3
b = 3.0
print(a == b)              # True (same value)
print(type(a), type(b))    # <class 'int'> <class 'float'>
```

Integer Division vs Float Division

```
print(10 / 3)   # 3.3333 (float division)
print(10 // 3)  # 3 (integer division)
```

Task: Format expression level to two decimal places using both f-string and %% formatting. Try converting a float string to a number.

```
name = "lacZ"
length = 1023
print(f"Gene {name} is {length} bp long")
print("Gene %s is %d bp long" % (name, length))
```

```
a = 3
b = 3.0
print(a == b)              # True in value
print(type(a), type(b))    # int vs float
```

String Formatting Examples

```
gene = "trpE"
expr = 8.4567
print(f"{gene} expression: {expr:.2f}")
print("%s expression: %.2f" % (gene, expr))
```

Task: Format expression level to two decimal places using f-string.

---

5. Operators and Comparisons

```
5 == 5       # True
5 != 3       # True
10 > 3       # True
10 >= 10     # True
2 < 5        # True
3 <= 3       # True
```

```
print(10 / 3)   # 3.3333 (float division)
print(10 // 3)  # 3 (integer division)
```

Operator Types & Examples

- Arithmetic: `+`, `-`, `*`, `/`, `//`, `%`, `**`
- Comparison: `==`, `!=`, `<`, `>`, `<=`, `>=`
- Logical: `and`, `or`, `not`

```
expr = 2.5
if expr >= 2 and expr <= 10:
    print("Valid expression range")
```

** Task:** Write a condition to check if a GC% is above 50.

---

6. Data Structures

Lists

```
genes = ["lacZ", "araC", "recA"]
genes.append("trpE")
print(genes[1])
```

Useful List Methods & Examples

```
genes.remove("recA")
print(genes)
print(len(genes))
print(sorted(genes))
```

Dictionaries

```
expression = {"lacZ": 5.2, "araC": 2.9}
print(expression["lacZ"])
expression["recA"] = 4.7
```

Useful Dictionary Methods

```
print(expression.keys())
print(expression.values())
print(expression.items())
```

Tuples

```
coords = (5, 10)
print(coords[0])
```

---

7. Loops

For Loops

```
dna = "ATGCGT"
for base in dna:
    print(base)
```

```
genes = ["lacZ", "araC", "recA"]
for gene in genes:
    print(gene.upper())
```

List Comprehension (One-liner loop)

A compact way to generate or transform lists.

```
# Get lengths of all sequences in a list
sequences = ["ATGC", "GGTACC", "T"]
lengths = [len(seq) for seq in sequences]
print(lengths)  # [4, 6, 1]
```

```
# Convert all gene names to lowercase
genes = ["LACZ", "ARAC", "RECA"]
lowercase_genes = [gene.lower() for gene in genes]
print(lowercase_genes)
```

While Loops

```
i = 0
while i < 5:
    print(i)
    i += 1
```

Task: Iterate through a list of DNA sequences and print their lengths.

---

8. Conditionals

```
expression = 4.5
if expression > 5:
    print("Highly expressed")
elif expression > 2:
    print("Moderately expressed")
else:
    print("Low expression")
```

---

9. File Handling

Writing

```
with open("genes.txt", "w") as file:
    file.write("lacZ\naraC\nrecA")
```

Reading

```
with open("genes.txt", "r") as file:
    for line in file:
        print(line.strip())
```

Task: Write a FASTA header and sequence to a file.

---

10. Frequently Used Keywords

Keywords to Recognize

- `if`, `else`, `elif`
- `for`, `while`, `break`, `continue`
- `def`, `return`
- `import`, `as`, `from`
- `with`, `open`, `in`, `not`, `and`, `or`, `is`
- `True`, `False`, `None`, `class`, `try`, `except`

Chapter 2: Introduction to Functions in Python

Part 1: Basic Concepts

1.1 What is a Function?

A function is a block of code that performs a specific task. It helps in making the code reusable and more organized.

Function Syntax:

```
def function_name(parameters):
    # Function body
    return result
```

** Example 1: Population Growth Rate**

The formula is: [\text{growth rate} = \frac{\text{population at time } t - \text{population at time } t_0}{\text{time difference}}]

```
def calculate_growth_rate(population_t, population_t0, time_diff):
    growth_rate = (population_t - population_t0) / time_diff
    return growth_rate
```

Test:

```
calculate_growth_rate(1500, 1000, 5)
```

1.2 Functions with Multiple Parameters

```
def calculate_growth_rate(initial_population, final_population, time_diff):
    growth_rate = (final_population - initial_population) / time_diff
    return growth_rate
```

Test:

```
calculate_growth_rate(1000, 5000, 10)
```

1.3 Using Return Values

Modify the function to interpret the result:

```
def calculate_growth_rate(initial_population, final_population, time_diff):
    growth_rate = (final_population - initial_population) / time_diff
    if growth_rate > 50:
        return "Rapid growth", growth_rate
    elif growth_rate > 0:
        return "Slow growth", growth_rate
```

```
    else:
        return "No growth or decline", growth_rate
```

Part 2: Intermediate Concepts

2.1 Default Arguments

```
def calculate_growth_rate(initial_population, final_population, time_diff=1):
    growth_rate = (final_population - initial_population) / time_diff
    return growth_rate
```

Test:

```
calculate_growth_rate(1000, 5000)
```

2.2 DNA Sequence GC Content

```
def calculate_gc_content(dna_sequence):
    gc_count = dna_sequence.count("G") + dna_sequence.count("C")
    gc_content = (gc_count / len(dna_sequence)) * 100
    return gc_content
```

Test:

```
calculate_gc_content("AGCTGCATG")
```

2.3 Multiple Return Values

```
def calculate_gc_content(dna_sequence):
    g_count = dna_sequence.count("G")
    c_count = dna_sequence.count("C")
    gc_content = ((g_count + c_count) / len(dna_sequence)) * 100
    return g_count, c_count, gc_content
```

Test:

```
calculate_gc_content("AGCTGCATG")
```

Part 3: Advanced Concepts

3.1 Lambda Functions, aka one-liner functions

```
gc_content = lambda dna: ((dna.count("G") + dna.count("C")) / len(dna)) * 100
```

Test:

```
gc_content("AGCTGCATG")
```

3.2 Recursion: Factorial Function

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

Test:

```
factorial(5)
```

3.3 Biochemical Reaction Rate

Using the Law of Mass Action: [\text{Rate} = k \times [A] \times [B]]

```
def rate_of_reaction(k, concentration_A, concentration_B):
    return k * concentration_A * concentration_B
```

Test:

```
rate_of_reaction(0.5, 2, 3)
```

Bonus Challenge

DNA Sequence Similarity

Write a function to calculate percent similarity between two DNA sequences based on matching characters at the same positions.

```
def sequence_similarity(seq1, seq2):
    min_len = min(len(seq1), len(seq2))
    match_count = 0
    for i in range(min_len):
        if seq1[i] == seq2[i]:
            match_count += 1
    return (match_count / min_len) * 100
```

Test:

```
sequence_similarity("AGCT", "AGGT")
```

Chapter 3: Data Science with Python

This worksheet introduces **NumPy**, **Pandas**, **Matplotlib**, and **Seaborn** using examples from biological data. It is designed for beginners in Python data science with an interest in life sciences.

Part 1: NumPy - Numerical Computing in Biology

Introduction to Arrays and Matrices

Biological data like gene expression, DNA microarray data, or mutation matrices can be efficiently represented with arrays.

NumPy provides several ways to generate ranges of numbers, which is especially useful for creating index vectors, time points, or synthetic biological data.

```
# np.arange: generates evenly spaced values within a given interval
print(np.arange(0, 10, 2))  # [0 2 4 6 8]

# np.linspace: generates evenly spaced numbers over a specified interval
print(np.linspace(0, 1, 5))  # [0.   0.25 0.5  0.75 1.  ]

# np.logspace: logarithmically spaced values (useful in plotting growth rates or
concentrations)
print(np.logspace(1, 3, num=4))  # [  10.  100. 1000.]

# Example: time points in a simulation
time_points = np.linspace(0, 24, num=25)  # hours from 0 to 24
print("Time points (hours):", time_points)
```

Matrix Manipulation and Multiplication

NumPy is widely used to do linear algebra, i.e. matrix manipulation. Matrix operations are central to many bioinformatics analyses, such as computing similarities, transformations, or modeling interactions.

```
import numpy as np

# Gene expression matrix (rows = genes, columns = samples)
gene_expression = np.array([
    [5.1, 3.5, 1.4, 0.2],
    [4.9, 3.0, 1.4, 0.2],
    [6.7, 3.1, 4.7, 1.5]
])
print("Shape:", gene_expression.shape)
print("Mean expression per gene:", np.mean(gene_expression, axis=1))
print("Max expression per sample:", np.max(gene_expression, axis=0))
```

```
# Matrix multiplication: sample correlation via dot product
# Let's assume we want to project gene expression using a transformation matrix
transformation = np.array([
```

```
    [1.0, 0.5, 0.2, 0.1],
    [0.3, 1.2, 0.4, 0.2]
])
projected = np.dot(transformation, gene_expression.T)
print("Projected gene expressions:
", projected)

# Element-wise multiplication
scaled_expression = gene_expression * 2
print("Scaled expression:
", scaled_expression)

# Matrix addition
offset = np.ones_like(gene_expression) * 0.5
adjusted_expression = gene_expression + offset
print("Adjusted expression:
", adjusted_expression)

# Identity matrix and its role in preserving matrix shape
identity = np.eye(gene_expression.shape[1])
identity_mult = np.dot(gene_expression, identity)
print("Multiplication with identity matrix:
", identity_mult)
```

Why NumPy is Faster

```
import time

# Using lists (Python)
a = list(range(1000000))
b = list(range(1000000))
start = time.time()
c = [x + y for x, y in zip(a, b)]
print("Python list time:", time.time() - start)

# Using NumPy
a_np = np.arange(1000000)
b_np = np.arange(1000000)
start = time.time()
c_np = a_np + b_np
print("NumPy time:", time.time() - start)
```

More Useful NumPy Operations

```
# Transpose of a matrix
print(gene_expression.T)

# Boolean masking
print(gene_expression[gene_expression > 5])

# Summarization
```

```
print("Standard deviation per gene:", np.std(gene_expression, axis=1))

# Linear algebra
cov_matrix = np.cov(gene_expression.T)
print("Covariance matrix:", cov_matrix)
```

Part 2: Matplotlib - Visualizing Biological Data

Basic Plots: Bar Chart

```
import matplotlib.pyplot as plt

genes = ['GeneA', 'GeneB', 'GeneC']
expression = [5.1, 4.9, 6.7]

plt.bar(genes, expression, color='teal')
plt.title('Expression Levels in Sample1')
plt.xlabel('Gene')
plt.ylabel('Expression')
plt.grid(True, linestyle='--', alpha=0.5)
plt.show()
```

Adding Layers to Plots

```
samples = ['Sample1', 'Sample2', 'Sample3']
geneA = [5.1, 5.3, 5.5]
geneB = [4.9, 5.0, 5.2]

plt.plot(samples, geneA, label='GeneA', marker='o', linestyle='-', color='green')
plt.plot(samples, geneB, label='GeneB', marker='s', linestyle='--', color='orange')
plt.title('Expression Over Samples')
plt.xlabel('Samples')
plt.ylabel('Expression')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

Other Plot Types

```
# Pie chart
sizes = [40, 35, 25]
labels = ['Upregulated', 'Downregulated', 'Unchanged']
plt.pie(sizes, labels=labels, autopct='%1.1f%%')
plt.title('Gene Expression Categories')
plt.axis('equal')
plt.show()

# Histogram
plt.hist(gene_expression.flatten(), bins=5, color='purple', edgecolor='black')
```

```
plt.title('Histogram of All Expression Values')
plt.xlabel('Expression Level')
plt.ylabel('Frequency')
plt.show()
```

Part 3: Pandas - Managing Biological Data

Most of the time your biological sequence data will be accompanied by some kind of meta data, living in a spreadsheet. Pandas Dataframe is perfect for handling those data.

Creating DataFrames

```
import pandas as pd

data = {
    'Gene': ['GeneA', 'GeneB', 'GeneC'],
    'Sample1': [5.1, 4.9, 6.7],
    'Sample2': [5.3, 5.0, 6.9]
}
df = pd.DataFrame(data)
print(df)
```

However, it's unlikely that you have to manually type in the data. You should read the meta data from a spreadsheet/csv/tsv file.

```
# Reading a DataFrame from a CSV file
# Suppose you have a file 'gene_expression.csv' with gene expression values
# Format:
# Gene,Sample1,Sample2
# GeneA,5.1,5.3
# GeneB,4.9,5.0
# GeneC,6.7,6.9

df_from_csv = pd.read_csv('gene_expression.csv')
print(df_from_csv)

# To read a file with tab-delimited format (common in biological data), use:
df_tab = pd.read_csv('expression_data.tsv', sep='    ')

# For large CSVs, read only specific columns or rows:
df_subset = pd.read_csv('gene_expression.csv', usecols=['Gene', 'Sample1'], nrows=10)

# Setting the first column as index
df_indexed = pd.read_csv('gene_expression.csv', index_col=0)
```

Indexing and Subsetting

Often we need to find specific samples, subset the dataframe by particular condition, etc.

```
# Access column
print(df['Sample1'])

# Conditional subset
print(df[df['Sample1'] > 5.0])

# Using loc and iloc
print(df.loc[0, 'Gene'])
print(df.iloc[1, 2])

# Setting index
df.set_index('Gene', inplace=True)
print(df)

# Resetting index
df.reset_index(inplace=True)
```

Reshaping Data: Melt and Pivot

```
melted = pd.melt(df, id_vars=['Gene'], var_name='Sample', value_name='Expression')
print(melted)

pivoted = melted.pivot(index='Gene', columns='Sample', values='Expression')
print(pivoted)
```

Writing Data to CSV

Once you have transformed or cleaned your DataFrame, you may want to save it:

```
# Save the melted DataFrame to a CSV file
melted.to_csv('melted_expression.csv', index=False)

# Save the pivoted DataFrame, keeping the index
pivoted.to_csv('pivoted_expression.csv')

# Save only selected columns
df[['Gene', 'Sample1']].to_csv('subset_expression.csv', index=False)
```

More Useful Pandas Methods

```
# Describe summary
print(df.describe())

# Correlation
print(df.corr())

# Group by example
grouped = melted.groupby('Sample').mean()
print(grouped)
```

```
# Sorting
print(df.sort_values(by='Sample1', ascending=False))
```

Part 4: Seaborn - Advanced Visualization

Why Seaborn?

- Integrates well with Pandas
- Prettier and more informative plots
- Statistical functionality built-in
- Themes and aesthetics for scientific publication

```
import seaborn as sns
sns.set(style='whitegrid')
```

Creating Plots from Pandas DataFrame

```
sns.barplot(data=melted, x='Gene', y='Expression', hue='Sample', palette='muted')
plt.title('Gene Expression Comparison')
plt.show()
```

Other Plot Types

```
# Boxplot
sns.boxplot(data=melted, x='Gene', y='Expression', palette='pastel')
plt.title('Expression Distribution')
plt.show()

# Violin plot
sns.violinplot(data=melted, x='Gene', y='Expression', inner='point', palette='Set2')
plt.show()

# Scatterplot (without regression line)
sns.lmplot(data=melted, x='Gene', y='Expression', fit_reg=False)

# Heatmap
heatmap_data = pivoted.fillna(0)
sns.heatmap(heatmap_data, annot=True, cmap='YlGnBu')
plt.title('Expression Heatmap')
plt.show()
```

Using Seaborn with Matplotlib

```
plt.figure()
ax = sns.boxplot(data=melted, x='Gene', y='Expression')
ax.set_title('Gene Expression Boxplot')
ax.set_ylabel('Expression Level')
plt.grid(True)
plt.show()
```

Exercises

1. Create a NumPy array for mutation frequencies in 10 genes across 3 populations. Use `np.random.randint()`.
2. Visualize this using a grouped bar chart. Label each group properly.
3. Convert this to a Pandas DataFrame, melt it, and use Seaborn to make a boxplot.
4. Apply filters and slicing on the DataFrame to show only high-frequency mutations (e.g., >10).
5. Combine Matplotlib and Seaborn to make a final styled plot with custom titles, labels, and colors.

This worksheet is designed to get you started with real biological data analysis in Python. Explore more by using real-world datasets like those from GEO, TCGA, or Ensembl!

Chapter 4: Introduction to Object-Orientered Programming (OOP) in Python - A Bioinformatics Perspective

Welcome back! We're going to explore Object-Oriented Programming (OOP) in Python, this time through the lens of Bioinformatics. Bioinformatics deals with vast amounts of biological data, and OOP provides a fantastic way to manage and work with this data effectively.

Think about the fundamental biological molecules we study: DNA, RNA, and Proteins. Each of these has specific characteristics (like their sequence) and specific actions they can perform or undergo (like transcription, translation, or sequence alignment). OOP helps us model these biological entities and their interactions in our code.

What is OOP (Bioinformatics Edition)?

As before, OOP is about organizing your code around **objects**. In Bioinformatics, these objects can represent biological concepts or data structures.

- **Class:** Still the **blueprint**. In Bioinformatics, a class could represent a general type of biological data, like a `DNASequence`, `ProteinSequence`, or a `Gene`. It defines what kind of information (attributes) a `DNASequence` object will hold and what actions (methods) it can perform.
- **Object (or Instance):** A specific **instance** created from a class. A specific DNA sequence you're working with, like `"ATGCGTACGT"`, would be an **object** of the `DNASequence` class. It has the structure defined by the class, but its specific sequence data is unique.
- **Attribute:** The **data** or **properties** of an object. For a `DNASequence` object, attributes would be the sequence string itself (e.g., `"ATGCGTACGT"`), maybe an identifier (e.g., `"seq1"`), or its length. These are stored *within* the object.
- **Method:** The **functions** or **operations** that an object can perform. For a `DNASequence` object, methods could be `get_complement()`, `transcribe()`, `calculate_gc_content()`, or `get_length()`. These methods typically act on the object's own data (its attributes).

The `self` Keyword

You'll see `self` appear frequently in class methods, especially in the `__init__` method. `self` is a convention (you could technically call it something else, but don't!) that refers to the *instance of the class* (the object) that the method is being called on.

When you write `some_object.method()`, Python automatically passes `some_object` as the first argument to the `method`, and inside the method, this object is referred to as `self`. This allows methods to access and modify the object's specific attributes.

Why Use OOP in Bioinformatics?

Bioinformatics often involves handling many different biological sequences, each with associated data and operations.

Consider working with a set of DNA sequences.

Procedural Approach:

You might store sequences in a list of strings and have separate functions to operate on them:

```
sequences = ["ATGCGT", "GCAT", "AGCTAG"]
ids = ["seq1", "seq2", "seq3"]

def get_dna_length(dna_sequence):
  return len(dna_sequence)

def transcribe_dna_to_rna(dna_sequence):
  # simple transcription (T to U)
  return dna_sequence.replace('T', 'U')

# To transcribe the first sequence:
rna_seq1 = transcribe_dna_to_rna(sequences[0])
print(f"Length of seq1: {get_dna_length(sequences[0])}")
print(f"RNA of seq1: {rna_seq1}")
```

This works, but:

- The sequence data (`sequences`) and its associated information (`ids`) are in separate lists. It's easy for them to get out of sync (e.g., accidentally deleting a sequence but not its ID).
- The functions operate on raw strings. There's nothing inherently tying `get_dna_length` or `transcribe_dna_to_rna` specifically to DNA sequence *objects*. You could accidentally pass a protein sequence string to `transcribe_dna_to_rna`, and the function might run but produce a meaningless result.
- If you wanted to add another piece of data related to a sequence (like its source organism), you'd need another list. If you wanted to add a new operation (like calculating melting temperature), you'd need another function.

Object-Oriented Approach:

We can create a `DNASequence` class that bundles the sequence data and relevant operations together.

```
class DNASequence:
    def __init__(self, sequence, identifier=None):
        # Attributes: data specific to this DNA sequence object
        self.sequence = sequence.upper() # Store as uppercase
        self.identifier = identifier
        self.length = len(self.sequence)

    # Method: get the length (already stored, but demonstrates accessing attributes)
    def get_length(self):
```

```
        return self.length

    # Method: calculate GC content
    def calculate_gc_content(self):
        gc_count = self.sequence.count('G') + self.sequence.count('C')
        return (gc_count / self.length) * 100 if self.length > 0 else 0

    # Method: transcribe to RNA
    def transcribe(self):
        rna_sequence = self.sequence.replace('T', 'U')
        # We could return a string or even an RNASequence object if we had one!
        return rna_sequence

    # A nice way to represent the object when printed
    def __str__(self):
        return f"> {self.identifier if self.identifier else
'Untitled'}\n{self.sequence}"


# Create DNASequence objects:
seq1 = DNASequence("ATGCGT", identifier="GeneA")
seq2 = DNASequence("GCATAG", identifier="GeneB")

# Use methods on the objects:
print(seq1) # Uses the __str__ method
print(f"Length of {seq1.identifier}: {seq1.get_length()}")
print(f"GC content of {seq2.identifier}: {seq2.calculate_gc_content():.2f}%")
print(f"RNA transcribed from {seq1.identifier}: {seq1.transcribe()}")
```

Advantages in this context:

1. **Data Encapsulation:** The sequence data (`self.sequence`), identifier (`self.identifier`), and length (`self.length`) are bundled *within* the `DNASequence` object. They belong together.
2. **Organized Operations:** Methods like `calculate_gc_content()` and `transcribe()` are part of the `DNASequence` class, clearly indicating that these operations apply to DNA sequences. You call them directly on the object (`seq1.calculate_gc_content()`), making the code more readable and intuitive.
3. **Reusability:** You can easily create many `DNASequence` objects, each managing its own data and capable of performing the defined methods.
4. **Maintainability:** If you need to change how GC content is calculated, you only modify the `calculate_gc_content` method within the `DNASequence` class. This change automatically applies to all `DNASequence` objects.

Building a Basic Bioinformatics Tool (Conceptual)

Imagine building a tool that processes a file of DNA sequences. Using OOP, you could read each sequence from the file and create a `DNASequence` object for it. Then, you could store these objects in a list. This list of `DNASequence` objects is much more powerful than a simple list of strings, as each item in the list "knows" how to calculate its own GC content, transcribe itself, etc.

```
# Conceptual code (doesn't read from a real file)
# Imagine we read these from a file
sequence_data = [
    ("Seq1", "ATGCGTACGT"),
    ("Seq2", "CGATCGATCG"),
    ("Seq3", "AAAAATTTTT")
]

dna_sequences = []
for identifier, sequence_str in sequence_data:
    # Create a DNASequence object for each entry
    dna_obj = DNASequence(sequence_str, identifier=identifier)
    dna_sequences.append(dna_obj)

# Now we have a list of DNASequence objects
print("\nProcessing sequences:")
for dna_seq_obj in dna_sequences:
    print(dna_seq_obj) # Print the object using its __str__ method
    print(f"  Length: {dna_seq_obj.get_length()}")
    print(f"  GC Content: {dna_seq_obj.calculate_gc_content():.2f}%")
    print(f"  RNA: {dna_seq_obj.transcribe()}")
    print("-" * 20)
```

This structured approach makes it much easier to add more features (e.g., searching for motifs, performing alignments – which could also be represented as objects!) later on.

Exercise 1: Enhance the DNASequence Class

Add a new method to the `DNASequence` class called `get_complement()`. This method should return the reverse complement sequence. Remember the base pairing rules: A-T and G-C.

```
class DNASequence:
    def __init__(self, sequence, identifier=None):
        self.sequence = sequence.upper()
        self.identifier = identifier
        self.length = len(self.sequence)

    def get_length(self):
        return self.length

    def calculate_gc_content(self):
        gc_count = self.sequence.count('G') + self.sequence.count('C')
        return (gc_count / self.length) * 100 if self.length > 0 else 0

    def transcribe(self):
        rna_sequence = self.sequence.replace('T', 'U')
        return rna_sequence

    def __str__(self):
        return f"> {self.identifier if self.identifier else
'Untitled'}\n{self.sequence}"
```

```
    # Add your get_complement method here:
    def get_complement(self):
        # Hint: You might want to create a dictionary to map bases
        complement_dict = {'A': 'T', 'T': 'A', 'C': 'G', 'G': 'C'}
        # Build the complement sequence base by base
        complement = ""
        for base in self.sequence:
            complement += complement_dict.get(base, base) # Use .get for safety

        # Reverse the complement sequence
        return complement[::-1]


# Test your new method
test_seq = DNASequence("ATGCGT", identifier="TestSeq")
print(f"Original: {test_seq.sequence}")
print(f"Complement: {test_seq.get_complement()}")

# What if the sequence is empty?
empty_seq = DNASequence("")
print(f"Empty seq complement: {empty_seq.get_complement()}")
```

Exercise 2: Design a Protein Sequence Class

Think about what attributes and methods a `ProteinSequence` class might have.

1. What are some key **attributes** for a protein sequence? (e.g., sequence string, identifier, maybe a list of amino acids?)
2. What are some key **methods** for a protein sequence? (e.g., calculate length, count specific amino acids, potentially methods related to molecular weight or hydrophobicity - keep it simple!)

Write down your ideas!

ProteinSequence Class:

- **Attributes:**
    - ...
    - ...
- **Methods:**
    - ...
    - ...

Exercise 3: Why OOP for Biological Data?

Based on the `DNASequence` example and your ideas for the `ProteinSequence` class, explain in your own words why using OOP is a good approach for handling and analyzing biological data like sequences, compared to just using strings and separate functions.

Focus on how it helps keep things organized and makes your code potentially easier to scale up for larger projects.

...

Chapter 5: Introduction to Unix/Bash for Bioinformatics

Objective: To familiarize yourself with the Unix/Bash command-line interface (CLI) and learn basic commands essential for bioinformatics analysis.

Why Unix/Bash in Bioinformatics? Bioinformatics often involves working with very large datasets (genome sequences, gene expression data, protein structures, etc.). Graphical user interfaces (GUIs) can become slow, inefficient, or lack the specific tools needed for these complex tasks. The command line provides a powerful, efficient, scriptable, and reproducible way to:

- Navigate complex file systems containing sequencing data, reference genomes, etc.
- Manage and manipulate large text files (like FASTA, FASTQ, VCF, GFF).
- Automate repetitive analysis steps using scripts.
- Run specialized bioinformatics software (most are designed primarily for command-line use).
- Connect different tools together in analysis pipelines.

Part 1: The Command Line Interface (CLI) & Basic Navigation

The command line is a text-based way to interact with your computer. You type commands, press Enter, and the system responds. You'll see a *prompt*, often ending in `$` or `#`, indicating it's ready for your input.

Key Commands:

1. `pwd` (Print Working Directory)
    - Shows the full path of the directory you are currently located in.

      ```
      pwd
      ```

    - Example Output:* `/home/student/bio_project`
2. `ls` (List)
    - Lists the files and directories in the current directory.
    - `ls` : Basic, simple listing.
    - `ls -l` : **L**ong listing format (shows permissions, owner, size, modification date).
    - `ls -lh` : Long listing with **h**uman-readable file sizes (e.g., KB, MB, GB). Very useful!
    - `ls -lt` : Files ordered based on the time they were last changed.
    - `ls -a` : Lists **a**ll files, including hidden files (names starting with a dot `.`, like configuration files).

      ```
      ls
      ls -lh
      ls -a
      ```

3. `cd` (Change Directory)
    - Used to move between directories (folders).
    - `cd <directory_name>`: Moves into a subdirectory named `<directory_name>`. (e.g., `cd raw_data`)
    - `cd ..`: Moves one level **up** (to the parent directory).
    - `cd ~` or just `cd`: Moves to your **home** directory (your personal starting point).
    - `cd /`: Moves to the **root** directory (the top-level directory of the entire system).
    - *Tip:* Use the `Tab` key to auto-complete directory and file names!

```
# Assume you are in /home/student and have a directory 'bio_project'
pwd
cd bio_project
pwd # Check your new location
cd .. # Go back up to /home/student
pwd # Check again
cd ~ # Go to your home directory
pwd
```

Exercise 1: a. Use `pwd` to find out where you are right now. b. Use `ls -lh` to see what's in your current directory. Note the file sizes. c. If there's a directory listed, use `cd` to move into it. Use `pwd` again. d. Use `cd ..` to move back up. e. Use `cd ~` to ensure you are in your home directory.

Part 2: Working with Files and Directories

Key Commands:

1. `mkdir <directory_name>` (Make Directory)
    - Creates a new, empty directory.

```
# Create directories for a typical project structure
mkdir project_alpha
cd project_alpha
mkdir raw_data results scripts reference_genomes
ls # You should see the new directories
```

2. `touch <filename>`
    - Creates an empty file if it doesn't exist.
    - If the file *does* exist, it updates the file's last modification time (useful for some tools).

```
cd scripts
touch analysis_pipeline.sh
touch notes.txt
ls -l # See the new empty files with size 0
```

3. `cp <source_file> <destination>` (Copy)

- Copies a file.
- `cp file1 file2`: Copies `file1` to `file2` in the current directory.
- `cp file1 directory/`: Copies `file1` into the `directory`.
- `cp file1 directory/new_name`: Copies `file1` into `directory` and gives it a `new_name`.
- `cp -r <source_directory> <destination_directory>`: Copies a directory **r**ecursively (including all its contents).

```
# Assuming you're in project_alpha/scripts
cp notes.txt notes_backup.txt
ls
cd .. # Go up to project_alpha
cp scripts/notes.txt results/ # Copy notes into the results directory
ls results/
```

4. `mv <source> <destination>` (Move / Rename)

- Moves a file or directory to a new location OR renames it if the destination is in the same directory.

```
# Rename notes_backup.txt to old_notes.txt (in scripts directory)
cd scripts
mv notes_backup.txt old_notes.txt
ls
```

Move old_notes.txt to the results directory (go up first)

cd .. mv scripts/old_notes.txt results/ ls scripts/ # old_notes.txt should be gone ls results/ # old_notes.txt should be here

```
```

5. `rm <filename>` (Remove)

- Deletes a file. **USE WITH EXTREME CAUTION! There is usually NO UNDO or trash bin.** Double-check before pressing Enter.
- `rm -f <file_name>`: Removes a file without confirming to the user. **CAUTION**
- `rm -r <directory_name>`: Removes a directory and all its contents **r**ecursively. **EVEN MORE CAUTION!**
- `rm -i <filename>`: Interactive mode, prompts for confirmation before deleting each file. Safer option.

```
# Be careful! Let's remove the backup file in results
rm results/old_notes.txt
# Or safer:
# rm -i results/old_notes.txt # Asks 'remove regular empty file
results/old_notes.txt?'
```

6. `cat <filename>` (Concatenate)

- Displays the entire content of one or more files to the screen. Good for small files.

```
# First, put some text in notes.txt (we'll use a simple way for now)
echo "Project Alpha Notes" > results/notes.txt # '>' overwrites the file
echo "Analysis started: $(date)" >> results/notes.txt # '>>' appends to
the file
cat results/notes.txt
```

7. `less <filename>`

- Views file content page by page. Essential for large bioinformatics files (like FASTQ or VCF) that won't fit on the screen.
- **Navigation:** Use arrow keys (Up/Down), PageUp/PageDown, Spacebar (next page), `b` (back one page).
- **Search:** Type `/` followed by your search term, press Enter. `n` finds the next match, `N` finds the previous.
- **Quit:** Press `q`.

```
# Imagine notes.txt was very long
less results/notes.txt
# (Press 'q' to exit less)
```

8. `head <filename>`

- Displays the first few lines of a file (default is 10).
- `head -n 5 <filename>`: Shows the first 5 lines. Useful for checking file headers (e.g., in FASTA or VCF files).

```
# Let's create a sample FASTA file in raw_data
echo ">Seq1_OrganismA_GeneX" > raw_data/sample.fasta
echo "ATGCGTAGCTAGCTAGCATGCGTAGCTAGCTAGCATG" >> raw_data/sample.fasta
echo "CGTAGCTAGCTAGCTAGCTACGTACGTACGTACGTAC" >> raw_data/sample.fasta
echo ">Seq2_OrganismB_GeneY" >> raw_data/sample.fasta
echo "TTTTACGTAGCATGCATGCATTTTACGTAGCATGCAT" >> raw_data/sample.fasta
echo "AAAAACGTAGCATGCATGCAAAAAACGTAGCATGCAT" >> raw_data/sample.fasta
echo "GGGGACGTAGCATGCATGCAGGGGACGTAGCATGCAT" >> raw_data/sample.fasta
echo "CCCCACGTAGCATGCATGCACCCCACGTAGCATGCAT" >> raw_data/sample.fasta
echo "NNNNACGTAGCATGCATGCANNNNACGTAGCATGCAT" >> raw_data/sample.fasta
echo "ATATACGTAGCATGCATGCAATATACGTAGCATGCAT" >> raw_data/sample.fasta
echo "GCGCACGTAGCATGCATGCAGCGCACGTAGCATGCAT" >> raw_data/sample.fasta
echo ">Seq3_OrganismA_GeneZ" >> raw_data/sample.fasta
echo "GATTACAGATTACAGATTACAGATTACAGATTACAGA" >> raw_data/sample.fasta
```

head raw_data/sample.fasta head -n 4 raw_data/sample.fasta # Show first 4 lines (2 headers, 2 sequence lines)

9. `tail <filename>`

- Displays the last few lines of a file (default is 10).
- `tail -n 3 <filename>`: Shows the last 3 lines. Useful for checking if a long process finished correctly or seeing the end of large files.

```
tail raw_data/sample.fasta
tail -n 3 raw_data/sample.fasta # Show the last header and sequence
lines
```

Exercise 2: a. Navigate into the `project_alpha/reference_genomes` directory. b. Create an empty file named `human_genome.fasta` (it's empty for now, we'll download later). c. Copy the `sample.fasta` file from `raw_data` into the current directory (`reference_genomes`). d. Rename the copied `sample.fasta` in this directory to `test_sequences.fasta`. e. View the first 2 lines of `test_sequences.fasta`. f. View the last 2 lines of `test_sequences.fasta`. g. Delete the empty `human_genome.fasta` file using `rm`.

Part 3: Downloading Files (`wget`)

`wget` is a command-line utility to download files from web or FTP servers. Essential for getting reference genomes, annotations, software, etc.

Syntax: `wget <URL>`

```
# Let's download a small sample protein FASTA file from UniProt
# We'll put it in the reference_genomes directory
cd ~/project_alpha/reference_genomes # Make sure you're in the right place

# Download the Insulin sequence for Human (P01308)
# Note: URLs can change over time. This is for demonstration.
wget [https://rest.uniprot.org/uniprotkb/P01308.fasta]
(https://rest.uniprot.org/uniprotkb/P01308.fasta)

# Check if the file was downloaded
ls -lh P01308.fasta

# View the downloaded file
less P01308.fasta # (Press 'q' to quit)

# Rename it for clarity
mv P01308.fasta human_insulin.fasta
ls -lh
```

Exercise 3: a. Navigate to the `project_alpha/raw_data` directory. b. Use `wget` to download the FASTA sequence for Yeast Mbp1 transcription factor (UniProt ID P39678): `https://rest.uniprot.org/uniprotkb/P39678.fasta` c. Rename the downloaded file `P39678.fasta` to `yeast_mbp1.fasta`. d. Use `head` to look at the first 5 lines of the downloaded file.

Part 4: The Unix Philosophy & Piping (`|`)

The Philosophy (Simplified):

1. Write programs that do **one thing** and do it well. (`ls` lists, `grep` searches, `wc` counts).
2. Write programs to **work together**.
3. Write programs to handle **text streams**, because that is a universal interface.

Piping (`|`): The pipe symbol `|` is one of the most powerful concepts. It connects the *standard output* (stdout - the normal text output) of one command directly to the *standard input* (stdin) of the next command. This lets you chain simple tools to perform complex tasks without creating temporary intermediate files.

Example: Count the number of sequences in a FASTA file. FASTA sequence headers usually start with `>`. We can find these lines (`grep`) and then count how many lines were found (`wc -l`).

```
# Using the sample.fasta file from Part 2 (in raw_data)
cd ~/project_alpha/raw_data

# 1. Find lines starting with '>' using grep
grep "^>" sample.fasta # '^' means 'start of line'

# 2. Count the number of lines found using wc (word count)
#    wc -l counts lines
grep "^>" sample.fasta | wc -l
```

Here, the output of `grep "^>" sample.fasta` (which is the list of header lines) is *piped* directly as input to `wc -l`, which counts them. The result `3` appears on the screen.

Exercise 4: a. Use `cat`, `head`, and `tail` with pipes to display only lines 3-5 of your `sample.fasta` file. (Hint: `head -n 5 | tail -n 3`) b. Count how many lines in the `yeast_mbp1.fasta` file contain the sequence motif 'CGC' (case-sensitive). (Hint: Use `grep` and `wc -l`). Remember `grep` finds the pattern anywhere on the line.

Part 5: Searching and Filtering Text (`grep`)

`grep` (Global Regular Expression Print) searches for patterns (text strings or regular expressions) in files or input streams.

Key Options:

- `grep <pattern> <filename>` : Find pattern in file.
- `grep -i <pattern> <filename>` : **I**gnore case during search.
- `grep -v <pattern> <filename>` : In**v**ert match (show lines that *don't* match the pattern).
- `grep -c <pattern> <filename>` : **C**ount matching lines (shortcut for `grep ... | wc -l`).
- `grep -n <pattern> <filename>` : Show line **n**umbers along with matching lines.
- `grep -E <regex_pattern> <filename>` : Use **E**xtended regular expressions (more powerful patterns).
- `grep '^pattern'` : Find lines *starting* with the pattern (`^`).
- `grep 'pattern$'` : Find lines *ending* with the pattern (`$`).

```
# In raw_data/sample.fasta:
# Find sequences containing "GATTACA"
grep "GATTACA" sample.fasta

# Find sequence headers (lines starting with '>')
grep "^>" sample.fasta
# Count them directly
```

```
grep -c "^>" sample.fasta

# Find lines that are NOT sequence headers (i.e., the sequence lines)
grep -v "^>" sample.fasta

# Find headers containing "OrganismA"
grep "^>" sample.fasta | grep "OrganismA"

# Find headers for GeneY or GeneZ (using Extended Regex)
grep -E 'GeneY|GeneZ' sample.fasta | grep "^>"
```

Exercise 5: a. In `reference_genomes/human_insulin.fasta`, find the line containing the organism name (Hint: Search for `OS=`). Use the `-n` option to see the line number. b. Count how many sequence lines (lines *not* starting with `>`) are in `raw_data/yeast_mbp1.fasta`. Use `grep -v` and `grep -c`. c. Display the header line(s) in `raw_data/sample.fasta` that contain `GeneX`.

Part 6: Text Manipulation (`sed` and `awk`)

These are powerful stream editors for modifying text data. They are often used in bioinformatics pipelines to reformat files.

1. **`sed` (Stream Editor):** Good for simple substitutions, deletions, or printing specific lines based on patterns. Works line by line.
    - **Basic Substitution:** `sed 's/<find_pattern>/<replace_pattern>/g' <filename>`
        - `s`: Substitute command.
        - `/`: Delimiter (can be other characters like `#` or `|`).
        - `g`: **G**lobal replacement (replace all occurrences on a line, not just the first). Omit `g` to replace only the first match per line.
    - **Deleting lines:** `sed '/<pattern_to_match>/d' <filename>`
    - **Acting on specific lines:** `sed '/<pattern_to_match>/ s/<find>/<replace>/g' <filename>` (only performs substitution on lines matching the initial pattern).

```
cd ~/project_alpha/raw_data

# Display sample.fasta, replacing all 'A' with 'T' (doesn't change the file)
sed 's/A/T/g' sample.fasta

# Replace spaces in sequence headers with underscores ONLY on header lines
# We use a different delimiter '#' because the pattern contains '/'
sed '/^>/ s#_Organism#|Organism#' sample.fasta # Replace first _ with | on
headers

# Delete header lines (just display sequence)
sed '/^>/d' sample.fasta

# To save the changes, redirect output to a new file:
sed '/^>/ s/_/|/' sample.fasta > sample_header_mod.fasta
cat sample_header_mod.fasta
```

2. **`awk`:** A versatile pattern-scanning and processing language. Excellent for field-based operations (columns). By default, `awk` splits lines into fields based on whitespace (spaces or tabs). Fields are accessed using `$1`, `$2`, `$3`, etc. `$0` represents the entire line.
    - **Syntax:** `awk '<condition> { <action> }' <filename>`
    - The `<condition>` is optional (if omitted, action applies to all lines).
    - The `{ <action> }` block contains commands, often `print`.
    - `-F<delimiter>`: Specify a field separator (e.g., `-F','` for CSV, `-F'\t'` for TSV).

```
# Create a simple tab-separated file (e.g., gene counts)
echo -e "GeneID\tCount1\tCount2" > ../results/counts.tsv # -e enables
interpretation of \t (tab)
echo -e "GeneA\t100\t50" >> ../results/counts.tsv
echo -e "GeneB\t200\t75" >> ../results/counts.tsv
echo -e "GeneC\t50\t25" >> ../results/counts.tsv
cat ../results/counts.tsv

# Print only the first column (GeneID) from counts.tsv
awk '{ print $1 }' ../results/counts.tsv

# Print GeneID (col 1) and Count2 (col 3)
awk '{ print $1, $3 }' ../results/counts.tsv

# Print GeneID and sum of counts (field 2 + field 3) for lines where Count1 >
60
# NR > 1 skips the header row (Record Number > 1)
awk 'NR > 1 && $2 > 60 { print $1, $2 + $3 }' ../results/counts.tsv

# Extract just the sequence ID from sample.fasta headers (remove '>')
# Pipe grep output to awk. Set '>' as Field Separator, print 2nd field.
grep "^>" sample.fasta | awk -F'>' '{ print $2 }'
```

Exercise 6: a. Use `sed` to display the `reference_genomes/human_insulin.fasta` content, but remove the `(Homo sapiens)` part including the parentheses. (Hint: `sed 's/(Homo sapiens)//'`). b. Use `awk` on the `results/counts.tsv` file to print only the gene names (first column), skipping the header line. (Hint: Use `NR > 1`). c. Use `awk` on `results/counts.tsv` to print the GeneID and Count1, but only for genes where Count2 is less than or equal to 50 (skip the header).

Part 7: Scripting Basics (Loops, Conditionals, Reading Files)

You can combine commands into script files (usually ending in `.sh`) to automate tasks. A script typically starts with `#!/bin/bash` (called a "shebang") to specify that Bash should execute it.

1. **`for` loop:** Iterates over a list of items (like filenames).

```
# Create a script file in project_alpha/scripts
cd ~/project_alpha/scripts
# Use a text editor like 'nano' or 'vim' to create the file:
# nano process_fastas.sh
```

```
# --- Start of process_fastas.sh ---
#!/bin/bash

echo "Processing FASTA files in ../raw_data/"

# Loop through all files ending in .fasta in the raw_data directory
for fasta_file in ../raw_data/*.fasta
do
  echo "--- Processing $fasta_file ---"

  # Get the base filename without the path
  base_name=$(basename "$fasta_file")

  echo "Counting sequences in $base_name:"
  grep -c "^>" "$fasta_file"

  echo "Extracting headers from $base_name:"
  grep "^>" "$fasta_file" | sed 's/>//' # Remove '>'

  echo "" # Add a blank line for readability
done

echo "--- Processing complete. ---"
# --- End of process_fastas.sh ---

# Save and exit the editor (e.g., Ctrl+X, then Y, then Enter in nano)

# Make the script executable
chmod +x process_fastas.sh

# Run the script (use ./ to run from current directory)
./process_fastas.sh
```

- `"$fasta_file"`: Always quote variables containing filenames to handle spaces or special characters correctly.
- `$(command)`: Command substitution - runs the command and substitutes its output.

2. **`if-else` conditional:** Executes commands based on whether a condition is true or false.
   - **Syntax:**

     ```
     if [ <condition> ]; then
       # commands if condition is true
     elif [ <another_condition> ]; then
       # commands if another_condition is true
     else
       # commands if all conditions are false
     fi
     ```

   - **Common Conditions:**
     - `-f <file>`: True if `file` exists and is a regular file.

- `-d <dir>`: True if `dir` exists and is a directory.
- `-z "$var"`: True if variable `var` is empty (zero length).
- `-n "$var"`: True if variable `var` is not empty.
- `"$var1" == "$var2"`: True if strings are equal.
- `"$var1" != "$var2"`: True if strings are not equal.
- `N1 -eq N2` (equal), `-ne` (not equal), `-gt` (greater than), `-ge` (greater or equal), `-lt` (less than), `-le` (less or equal) for numerical comparisons.

```
# --- Example script: check_file.sh ---
#!/bin/bash

FILENAME="../reference_genomes/human_insulin.fasta"

echo "Checking for file: $FILENAME"

if [ -f "$FILENAME" ]; then
  echo "File exists."
  # Check if it's empty
  if [ -s "$FILENAME" ]; then # -s checks if file size is greater than zero
     echo "File is not empty. Header:"
     head -n 1 "$FILENAME"
  else
     echo "File exists BUT IS EMPTY."
  fi
else
  echo "ERROR: File not found."
fi
# --- End of check_file.sh ---

# chmod +x check_file.sh
# ./check_file.sh
```

3. **Reading a file line by line (`while IFS= read -r line`)**: The standard, safe way to process a file's content line by line within a script.

```
# Create a file with sample IDs: sample_list.txt in results dir
echo "Sample_A01" > ../results/sample_list.txt
echo "Sample_B05" >> ../results/sample_list.txt
echo "Sample_C12" >> ../results/sample_list.txt

# --- Example script: run_analysis_per_sample.sh ---
#!/bin/bash

INPUT_LIST="../results/sample_list.txt"

if [ ! -f "$INPUT_LIST" ]; then
  echo "Error: Sample list $INPUT_LIST not found."
  exit 1 # Exit script with an error status
fi

echo "Starting analysis based on $INPUT_LIST"
```

```
while IFS= read -r sample_id || [[ -n "$sample_id" ]]; do
  # Skip empty lines or lines starting with # (comments)
  if [[ -z "$sample_id" || "$sample_id" == \#* ]]; then
      continue # Skip to next iteration
  fi

  echo "  Processing Sample ID: $sample_id"
  # Imagine running a bioinformatics tool here, e.g.:
  # echo "  Running alignment for $sample_id..."
  # bwa mem reference.fasta ../raw_data/${sample_id}_R1.fastq.gz
../raw_data/${sample_id}_R2.fastq.gz > ../results/${sample_id}.sam
  sleep 0.5 # Simulate work
done < "$INPUT_LIST" # Redirect file content into the while loop's input

echo "Finished processing all samples."
# --- End of run_analysis_per_sample.sh ---

# chmod +x run_analysis_per_sample.sh
# ./run_analysis_per_sample.sh
```

- `IFS=` : Prevents trimming leading/trailing whitespace from lines.
- `read -r` : Prevents backslash interpretation (raw read).
- `|| [[ -n "$sample_id" ]]` : Handles the case where the last line in the file doesn't end with a newline character.
- `< "$INPUT_LIST"` : Redirects the content of the file to the standard input of the `while` loop.

Exercise 7: a. Create a script named `check_data.sh` in the `scripts` directory. b. Inside the script, use a `for` loop to iterate through the files `../raw_data/sample.fasta` and `../raw_data/yeast_mbp1.fasta`. c. Inside the loop, use an `if` statement (`if [ -f "$filename" ]`) to check if the file exists. d. If it exists, print a message like "`<filename>` exists, number of lines: `N`", where `N` is the line count obtained using `wc -l`. (Hint: Use command substitution `$(wc -l < "$filename" | awk '{print $1}')` to get just the number). e. If it doesn't exist, print an error message like "ERROR: `<filename>` not found." f. Make the script executable and run it.

Part 8: Environment Variables & Configuration (`.bashrc`)

Environment Variables: Variables that store information about your shell session and operating system environment. They are often used by programs to find necessary files or settings. Conventionally, they are named in uppercase.

- `echo $VARNAME` : Displays the value of a variable (e.g., `echo $HOME` , `echo $USER` , `echo $PATH`).
- `export VARNAME="value"` : Sets an environment variable for the current shell session and any programs or scripts started *from* that session.

```
# See your home directory path
echo $HOME

# See your username
```

```
echo $USER

# See the PATH: a colon-separated list of directories where the shell looks for
commands
echo $PATH

# Temporarily add your project's script directory to the PATH for this session
# (This means you could run your scripts from anywhere without typing ./ or the full
path)
export PATH="$PATH:$HOME/project_alpha/scripts"
echo $PATH # See the updated path

# Now try running a script from your home directory (cd ~)
# check_data.sh # It should run if the PATH was set correctly

# Note: This change is only for the current terminal session.
```

`.bashrc` file: A hidden configuration script in your home directory (`~/.bashrc`). It runs automatically every time you start a *new* interactive Bash shell (e.g., open a new terminal window). Use it to:

- Set environment variables permanently (like adding custom directories to `$PATH`).
- Create *aliases* (shortcuts for longer commands).
- Customize your shell prompt (`$PS1`).

```
# Edit the file (use nano or another editor)
# nano ~/.bashrc

# --- Add lines like these to the END of ~/.bashrc ---

# Add my project scripts directory to the PATH permanently
# export PATH="$PATH:$HOME/project_alpha/scripts"

# Create an alias 'll' for 'ls -lh' (a common useful alias)
# alias ll='ls -lh'

# Create a bioinformatics alias: count sequences in a fasta file
# alias countfasta='grep -c "^>"'

# --- Save and exit the editor ---

# To apply the changes to your CURRENT shell session, run:
source ~/.bashrc
# Alternatively, just open a new terminal window.

# Now test your aliases:
ll # Should run ls -lh
countfasta ../raw_data/sample.fasta # Should output 3
```

Exercise 8: a. Use `echo` to view the value of your `$SHELL` environment variable (shows your default shell). b. Edit your `~/.bashrc` file and add an alias: `alias projectdir='cd ~/project_alpha'` . c. Run `source ~/.bashrc` . d. Go to your root

directory (`cd /`). e. Type `projectdir` and press Enter. Use `pwd` to verify that the alias took you to the `~/project_alpha` directory.

Part 9: Unix Wildcards

Wildcards are special characters used in Unix-like operating systems to represent one or more other characters in a filename or path. They are incredibly useful for working with multiple files at once using commands like `ls` (list files), `cp` (copy files), `mv` (move/rename files), and `rm` (remove files).

The most common wildcards are:

- `*`: Matches any sequence of characters (including no characters).
- `?`: Matches any single character.
- `[]`: Matches any single character within the square brackets. Can specify a range (e.g., `[a-z]`, `[0-9]`).
- `[^]`: Matches any single character NOT within the square brackets.

Let's look at some examples:

Example 1: Using `*` (The Asterisk)

The asterisk is the most flexible wildcard. It matches zero or more characters.

- **To list all files in the current directory:**

  ```
  ls *
  ```

- **To list all files ending with `.txt`:**

  ```
  ls *.txt
  ```

- **To list all files starting with `data_`:**

  ```
  ls data_*
  ```

- **To list all files containing the word "report" in their name:**

  ```
  ls *report*
  ```

- **Bioinformatics Example:** You have several DNA sequence files in FASTA format, some ending in `.fasta` and some in `.fa`.

  - To list all FASTA files:

    ```
    ls *.fasta *.fa
    ```

    or

    ```
    ls *.fa*
    ```

  - To copy all `.fastq` files from the current directory to a new directory called `raw_reads`:

    ```
    cp *.fastq raw_reads/
    ```

Example 2: Using `?` (The Question Mark)

The question mark matches exactly one character.

- **To list files named `gene_1.txt`, `gene_2.txt`, but not `gene_10.txt`:**

  ```
  ls gene_?.txt
  ```

- **To list files like `sampleA.csv` and `sampleB.csv` but not `sampleABC.csv`:**

  ```
  ls sample?.csv
  ```

- **Bioinformatics Example:** You have a series of output files from a tool, named `align_output_1.sam`, `align_output_2.sam`, ..., `align_output_9.sam`, and `align_output_10.sam`.
  - To list only the output files from 1 to 9:

    ```
    ls align_output_?.sam
    ```

Example 3: Using `[]` (Square Brackets)

Square brackets match any single character within the brackets. You can specify a list of characters or a range.

- **To list files named `report_2023.csv` and `report_2024.csv`:**

  ```
  ls report_202[34].csv
  ```

- **To list files starting with a lowercase letter:**

  ```
  ls [a-z]*
  ```

- **To list files ending with a digit:**

  ```
  ls *[0-9]
  ```

- **Bioinformatics Example:** You have sequencing data files from different lanes, named `lane_1_read1.fastq`, `lane_2_read1.fastq`, `lane_A_read1.fastq`.
  - To list read 1 files only from lanes 1, 2, and 3 (if they exist):

    ```
    ls lane_[123]_read1.fastq
    ```

  - To list read 1 files from lanes with a single uppercase letter identifier:

    ```
    ls lane_[A-Z]_read1.fastq
    ```

Example 4: Using `[^]` (Negating Square Brackets)

This matches any single character NOT within the square brackets.

- **To list all files in the directory except those starting with a number:**

  ```
  ls [^0-9]*
  ```

- **To list files that do not end with `.fasta`:**

  ```
  ls *[^a].fasta # This is a bit tricky and less common for extensions.
  ```

  (A more typical way to exclude file types involves other commands or more complex patterns, but this demonstrates the `[^]` concept).

- **Bioinformatics Example:** You have various sequence files, but you want to list only those that are *not* `.fasta` or `.fastq`.

  - This is a more advanced use case and often better handled with other tools or combined patterns. A direct application of `[^]` for this specific scenario across the *entire* extension is not straightforward with just wildcards. However, you could use it to exclude specific characters *within* a filename pattern. For instance, to list files in a directory that do not have a digit immediately following an underscore:

    ```
    ls *_[^0-9]*
    ```

    This might help find files with different naming conventions.

Practice:

1. List all files in your current directory that end with `.fna` (a common extension for genome sequences).
2. You have alignment files named `sample_A.sam`, `sample_B.sam`, and `sample_C.sam`. List only `sample_A.sam` and `sample_C.sam` using `[]`.
3. You have many result files, some of which have a single digit after an underscore (e.g., `result_1.txt`, `result_5.txt`), and others have two digits (e.g., `result_10.txt`, `result_25.txt`). List only the files with a single digit after the underscore using `?`.
4. List all files in your directory that do NOT start with the letter 'a' or 'b'.

Conclusion

You've now covered the fundamentals of the Unix/Bash command line, including navigation, file management, downloading, the power of piping, text processing with `grep`, `sed`, and `awk`, basic scripting constructs, and environment configuration. These are essential skills for tackling bioinformatics analyses efficiently and reproducibly.

Key Takeaways:

- The command line is powerful for handling large data and automating tasks.
- Mastering `cd`, `ls`, `cp`, `mv`, `rm` is crucial for file management (be careful with `rm`!).
- `wget` is your tool for downloading data.
- Piping (`|`) lets you build complex workflows from simple tools.
- `grep`, `sed`, and `awk` are indispensable for searching and modifying text files (like FASTA, GFF, VCF).

- Shell scripting (`for` , `if` , `while`) automates your analyses.
- `.bashrc` helps customize your environment and create shortcuts (aliases).

Practice is key! Try applying these commands to other sample files or exploring the options (`man <command>` or `<command> --help`) for the tools you've learned. Good luck!