Practical No. S 01

Aim: Sequence Manipulation

- Read and parse sequence data from files
- Perform basic sequence manipulations (e.g., reverse complement, translation)

Input:

```
def translate(seq):
    table = {
        'ATA':'I', 'ATC':'I', 'ATT':'I', 'ATG':'M',
        'ACA':'T', 'ACC':'T', 'ACG':'T', 'ACT':'T',
        'AAC':'N', 'AAT':'N', 'AAA':'K', 'AAG':'K',
        'AGC':'S', 'AGT':'S', 'AGA':'R', 'AGG':'R',
        'CTA':'L', 'CTC':'L', 'CTG':'L', 'CTT':'L',
        'CCA':'P', 'CCC':'P', 'CCG':'P', 'CCT':'P',
        'CAC':'H', 'CAT':'H', 'CAA':'Q', 'CAG':'Q',
        'CGA':'R', 'CGC':'R', 'CGG':'R', 'CGT':'R',
        'GTA':'V', 'GTC':'V', 'GTG':'V', 'GTT':'V',
        'GCA':'A', 'GCC':'A', 'GCG':'A', 'GCT':'A',
        'GAC':'D', 'GAT':'D', 'GAA':'E', 'GAG':'E',
        'GGA':'G', 'GGC':'G', 'GGG':'G', 'GGT':'G',
        'TCA':'S', 'TCC':'S', 'TCG':'S', 'TCT':'S',
        'TTC':'F', 'TTT':'F', 'TTA':'L', 'TTG':'L',
        'TAC':'Y', 'TAT':'Y', 'TAA':'', 'TAG':'',
        'TGC':'C', 'TGT':'C', 'TGA':'_', 'TGG':'W',
    }
    protein = ""
    if len(seq) % 3 == 0:
        for i in range(0, len(seq), 3):
            codon = seq[i:i + 3]
            protein += table.get(codon, '?')  # Add default for unknown codons
    return protein
```

```
def read_seq(inputfile):
    with open(inputfile, "r") as f:
        seq = f.read()
    seq = seq.replace("\n", "")
    seq = seq.replace("\r", "")
    return seq

def reverse_complement(seq):
    complement = {'A': 'T', 'T': 'A', 'C': 'G', 'G': 'C'}
    return ''.join(complement[base] for base in reversed(seq))

prt = read_seq("amino_acid_sequence_original.txt")
dna = read_seq("DNA_sequence_original.txt")

# Translate the original DNA sequence
p_original = translate(dna[20:935])

# Obtain the reverse complement of the DNA sequence
dna_reverse_complemented = reverse_complement(dna)

# Translate the reverse complemented DNA sequence
p_reverse_complemented = translate(dna_reverse_complemented[20:935])

print("Does the translated protein match the given protein sequence?", p_original == prt)
print("Does the translated protein from reverse complemented DNA match the given protein
sequence?", p_reverse_complemented == prt)

x = "-" * 150
print(x)
print("Translated protein sequence from original DNA:\n", p_original)
```

```
print(x)

print("Translated protein sequence from reverse complemented DNA:\n",
p_reverse_complemented)

print(x)

print("Given protein sequence:\n", prt)
```

Output:

```
IDLE Shell 3.12.2
File  Edit  Shell  Debug  Options  Window  Help
Python 3.12.2 (tags/v3.12.2:6abddd9, Feb  6 2024, 21:26:36) [MSC v.1937 64 bit (
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: C:\Users\junai\Downloads\msc cs sem-2\Bioinformatics\practical_1.py
Does the translated protein match the given protein sequence? True
Does the translated protein from reverse complemented DNA match the given protei
n sequence? False
--------------------------------------------------------------------------------
--------------------------------------------------------------------
Translated protein sequence from original DNA:
 MSTHDTSLKTTEEVAFQIILLCQFGVGTFANVFLFVYNFSPISTGSKQRPRQVILRHMAVANALTLFLTIFPNNMMTFA
PIIPQTDLKCKLEFFTRLVARSTNLCSTCVLSIHQFVTLVPVNSGKGILRASVTNMASYSCYSCWFFSVLNNIYIPIKVT
GPQLTDNNNNSKSKLFCSTSDFSVGIVFLRFAHDATFMSIMVWTSVSMVLLLHRHCQRMQYIFTLNQDPRGQAETTATHT
ILMLVVTFVGFYLLSLICIIFYTYFIYSHHSLRHCNDILVSGFPTISPLLLTFRDPKGPCSVFFNC
--------------------------------------------------------------------------------
--------------------------------------------------------------------
Translated protein sequence from reverse complemented DNA:
 _LTVSIRFQPEIFSYQQVIITLLFYKYSSEHHTLFIQAFKNIFME_SI_WY_QSCLLCLAFLVTLAFNS_RTQNKDP_G
L_RSTVKEKL_GNPKPKCHCNASGNDENI_NRCKR_YK_DLEDKSQQM_LPASG_YGLLWSLLGPWGPD_E_ICTAFSDN
VYGGEVPWRHWSRP_CS_MWHHGQTSRRQCLH_NQKWNRTTCF_SYCYCLLTVDQ_P_LECRCYLRH_RTNNCNKNNLPC
L_HLL_VFLYLN_QEQV_QTDGYSEHKLNTSLCFLPRGE_RILIYI_GQFEE_LEQKSSCCLEI__
--------------------------------------------------------------------------------
--------------------------------------------------------------------
Given protein sequence:
 MSTHDTSLKTTEEVAFQIILLCQFGVGTFANVFLFVYNFSPISTGSKQRPRQVILRHMAVANALTLFLTIFPNNMMTFA
PIIPQTDLKCKLEFFTRLVARSTNLCSTCVLSIHQFVTLVPVNSGKGILRASVTNMASYSCYSCWFFSVLNNIYIPIKVT
GPQLTDNNNNSKSKLFCSTSDFSVGIVFLRFAHDATFMSIMVWTSVSMVLLLHRHCQRMQYIFTLNQDPRGQAETTATHT
ILMLVVTFVGFYLLSLICIIFYTYFIYSHHSLRHCNDILVSGFPTISPLLLTFRDPKGPCSVFFNC
```

Practical No. 02

Aim: Sequence Alignment

● Perform pairwise sequence alignment using algorithms like NeedlemanWunsch or Smith-Waterman

Input:

```
import numpy as np

import pandas as pd

import os

for dirname, _, filenames in os.walk('/kaggle/input'):

   for filename in filenames:

      print(os.path.join(dirname, filename))

      from sklearn.datasets import load_iris

      data=load_iris()

dir(data)

from sklearn.model_selection import train_test_split

x_train,x_test,y_train,y_test=train_test_split(data.data,data.target,test_size=0.2)

len(x_train),len(x_test)

from sklearn.linear_model import LogisticRegression

reg=LogisticRegression()

reg.fit(x_train,y_train)

reg.predict(x_test),y_test

reg.score(x_test,y_test)
```

Output:

```
Best alignment:
ATCG-
|| |
AT-GC
  Score=5

>>>
```

*Smith-Waterman:

Input:

```
from Bio import pairwise2

# Define the two sequences to align

seq1 = "ATCG"

seq2 = "ATGC"


# Perform the alignment using the Smith-Waterman algorithm

alignments = pairwise2.align.localms(seq1, seq2, 2, -1, -.5, -.1)


# Print the best alignment

best_alignment = max(alignments, key=lambda x: x.score)

print("Best alignment:")

print(pairwise2.format_alignment(*best_alignment))
```

Output:

```
Best alignment:
1 ATCG
  || |
1 AT-G
  Score=5.5

>>>
```

Practical No.-03

Aim: Database Searching

● Perform sequence searches against databases (e.g., BLAST or FASTA)

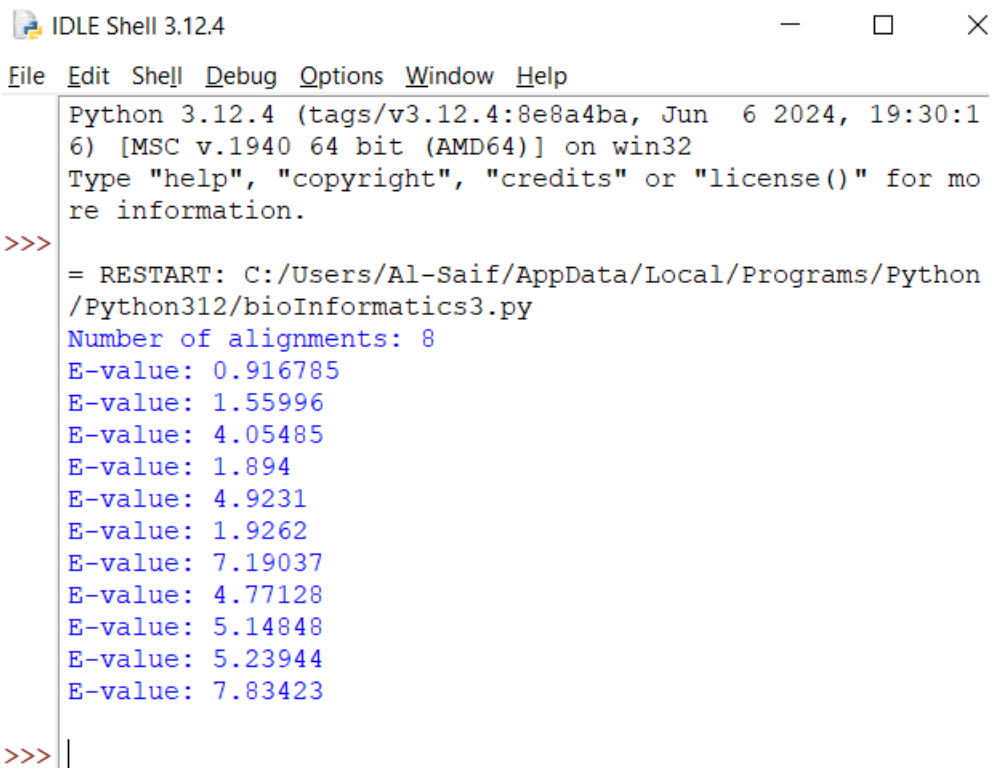
● Retrieve and analyze search results

Input:

```
from Bio.Blast import NCBIWWW, NCBIXML

# Define your protein sequence (example sequence provided)
sequence_data = """
>example
MENSDSLGKHGQSHQRHPSPLSPGVPQLQHRGVAPSPGGVHSQHRGVAPSPGSLSSQHRGVQ
"""
# Perform the BLAST search (note the change to 'blastp' and 'nr' database)
result_handle = NCBIWWW.qblast("blastp", "nr", sequence_data)

# Save the results to a file
with open("my_blast.xml", "w") as out_handle:
    out_handle.write(result_handle.read())
result_handle.close()  # Close the result handle after writing the results

# Read the BLAST results
with open("my_blast.xml") as result_handle:
    blast_record = NCBIXML.read(result_handle)

# Check if any alignments were found
if not blast_record.alignments:
    print("No alignments found.")
else:
    print(f"Number of alignments: {len(blast_record.alignments)}")

# Analyze the BLAST results
```

```
for alignment in blast_record.alignments:
    for hsp in alignment.hsps:
        print(f"E-value: {hsp.expect}")  # Print e-value for debugging
        if hsp.expect < 0.01:  # Adjust the threshold as needed
            print("Alignment")
            print(f"sequence: {alignment.title}")
            print(f"length: {alignment.length}")
            print(f"e value: {hsp.expect}")
            print(hsp.query)
            print(hsp.match)
            print(hsp.sbjct)
```

Output:



Practical No.-04

Aim: Genomic Data Analysis

A) Retrieve genomic data from databases (e.g., NCBI)

Input:

```
from Bio import Entrez, SeqIO

# Setting  email

Entrez.email = "mominhadi8080@gmail.com"

# Fetch a gene sequence from NCBI

def fetch_gene_sequence(gene_id):

    handle = Entrez.efetch(db="nucleotide", id=gene_id, rettype="fasta", retmode="text")

    record = SeqIO.read(handle, "fasta")

    handle.close()

    return record


# Example: Fetch BRCA1 gene sequence

gene_id = "NM_007294"  # BRCA1 gene

sequence_record = fetch_gene_sequence(gene_id)

print(sequence_record)
```

Output:

```
>>>
    ================= RESTART: C:/Users/Al-Saif/Bioinformatics.py =================
    ID: NM_007294.4
    Name: NM_007294.4
    Description: NM_007294.4 Homo sapiens BRCA1 DNA repair associated (BRCA1), transcript variant 1, mRNA
    Number of features: 0
    Seq('GCTGAGACTTCCTGGACGGGGGACAGGCTGTGGGGTTTCTCAGATAACTGGGCC...CCA')
>>>
```

B) Analyze gene annotations, promoter regions, or regulatory elements

Input:

```
import requests

import json

def fetch_gene_annotations(gene_id):

    url =f"https://rest.ensembl.org/lookup/id/{gene_id}?expand=1"

    response = requests.get(url, headers={"Content-Type": "application/json"})

    if not response.ok:

        response.raise_for_status()

    return response.json()


# Example: Fetch annotations for BRCA1 gene (Ensembl ID: ENSG00000012048)

gene_id = "ENSG00000012048"

annotations = fetch_gene_annotations(gene_id)

print(json.dumps(annotations, indent=2))
```

Output:

```
================================================================== RESTART: C:/Users
{
  "version": 26,
  "end": 43170245,
  "description": "BRCA1 DNA repair associated [Source:HGNC Symbol;Acc:HGNC:1100]",
  "object_type": "Gene",
  "biotype": "protein_coding",
  "source": "ensembl_havana",
  "seq_region_name": "17",
  "species": "homo_sapiens",
  "assembly_name": "GRCh38",
  "display_name": "BRCA1",
  "strand": -1,
  "canonical_transcript": "ENST00000357654.9",
  "id": "ENSG00000012048",
  "logic_name": "ensembl_havana_gene_homo_sapiens",
  "db_type": "core",
  "start": 43044295
}
```

C) Perform genomic variant analysis

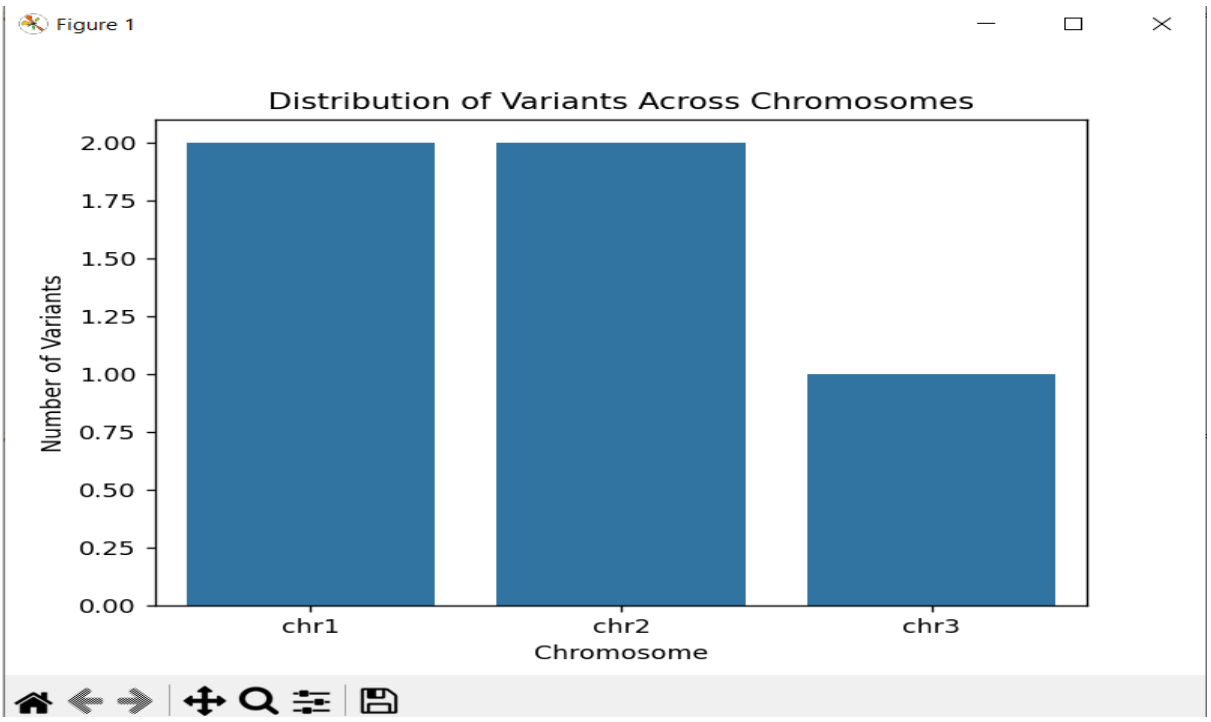
Input:

```
import pandas as pd
# Create a simplified VCF-like DataFrame
data = {
    'CHROM': ['chr1', 'chr1', 'chr2', 'chr2', 'chr3'],
    'POS': [101, 202, 303, 404, 505],
    'ID': ['rs1', 'rs2', 'rs3', 'rs4', 'rs5'],
    'REF': ['A', 'T', 'G', 'C', 'A'],
    'ALT': ['G', 'C', 'A', 'T', 'G'],
    'QUAL': [100, 99, 98, 97, 96],
    'FILTER': ['PASS', 'PASS', 'q10', 'PASS', 'q20'],
    'INFO': ['AF=0.1;DP=100', 'AF=0.2;DP=200', 'AF=0.3;DP=300', 'AF=0.4;DP=400', 'AF=0.5;DP=500']
}
vcf_data = pd.DataFrame(data)
print(vcf_data)

# Count variants per chromosome
print(vcf_data['CHROM'].value_counts())

# Identify common variants (those that passed the filter)
print(vcf_data[vcf_data['FILTER'] == 'PASS'])

# Extract allele frequency (AF) from the INFO column
vcf_data['AF'] = vcf_data['INFO'].str.extract(r'AF=([\d.]+)').astype(float)
print(vcf_data[['CHROM', 'POS', 'ID', 'REF', 'ALT', 'AF']])

import matplotlib.pyplot as plt
import seaborn as sns
```

```
# Plot the distribution of variants across chromosomes
sns.countplot(x='CHROM', data=vcf_data, order=vcf_data['CHROM'].value_counts().index)
plt.title('Distribution of Variants Across Chromosomes')
plt.xlabel('Chromosome')
plt.ylabel('Number of Variants')
plt.show()
```

Output:

```
>>> ================================================================== REST
  CHROM  POS   ID REF ALT  QUAL FILTER           INFO
0  chr1  101  rs1   A   G   100   PASS  AF=0.1;DP=100
1  chr1  202  rs2   T   C    99   PASS  AF=0.2;DP=200
2  chr2  303  rs3   G   A    98    q10  AF=0.3;DP=300
3  chr2  404  rs4   C   T    97   PASS  AF=0.4;DP=400
4  chr3  505  rs5   A   G    96    q20  AF=0.5;DP=500
chr1    2
chr2    2
chr3    1
Name: CHROM, dtype: int64
  CHROM  POS   ID REF ALT  QUAL FILTER           INFO
0  chr1  101  rs1   A   G   100   PASS  AF=0.1;DP=100
1  chr1  202  rs2   T   C    99   PASS  AF=0.2;DP=200
3  chr2  404  rs4   C   T    97   PASS  AF=0.4;DP=400
  CHROM  POS   ID REF ALT   AF
0  chr1  101  rs1   A   G  0.1
1  chr1  202  rs2   T   C  0.2
2  chr2  303  rs3   G   A  0.3
3  chr2  404  rs4   C   T  0.4
4  chr3  505  rs5   A   G  0.5
```



Practical No.-05

Aim : Data Preprocessing

● Cleaning and preprocessing biological data (e.g., gene expression data, DNA sequences)

● Handling missing values, outliers, and normalization of data

● Feature selection and dimensionality reduction techniques

Input:

```
import numpy as np
import re

# Sample gene expression data
gene_data = np.array([
    [1.2, 3.4, 2.1, np.nan],
    [2.3, 4.5, np.nan, 3.2],
    [3.4, np.nan, 1.8, 5.6],
    [np.nan, 2.5, 4.3, 6.7]
])

# Cleaning and preprocessing biological data
def clean_biological_data(data):
    # Remove non-biological characters from gene names
    cleaned_gene_names = [re.sub(r'[^a-zA-Z0-9]', '', str(gene_name)) for gene_name in data[:, 0]]
    cleaned_data = np.copy(data)  # Create a copy of the original data
    cleaned_data[:, 0] = cleaned_gene_names
    return cleaned_data

gene_data_cleaned = clean_biological_data(gene_data)

print("Original Gene Data:")
print(gene_data)

print("\nCleaned Gene Data:")
```

```
print(gene_data_cleaned)

# Handling missing values: Replace NaN with mean of respective column
def handle_missing_values(data):
    col_means = np.nanmean(data, axis=0)
    inds = np.where(np.isnan(data))
    data[inds] = np.take(col_means, inds[1])
    return data

gene_data = handle_missing_values(gene_data)

# Handling outliers: Replace outliers with median of respective column
def handle_outliers(data):
    col_medians = np.nanmedian(data, axis=0)
    for i in range(data.shape[1]):
        col = data[:, i]
        median = col_medians[i]
        col[col < (median - 2 * np.std(col))] = median
        col[col > (median + 2 * np.std(col))] = median
    return data
gene_data = handle_outliers(gene_data)

# Normalization of data: Min-Max normalization
def min_max_normalization(data):
    min_vals = np.nanmin(data, axis=0)
    max_vals = np.nanmax(data, axis=0)
    normalized_data = (data - min_vals) / (max_vals - min_vals)
    return normalized_data

gene_data_normalized = min_max_normalization(gene_data)
# Feature selection: Selecting top features based on variance
```

```
def select_top_features(data, num_features):
    variances = np.nanvar(data, axis=0)
    top_feature_indices = np.argsort(variances)[-num_features:]
    selected_data = data[:, top_feature_indices]
    return selected_data

selected_gene_data = select_top_features(gene_data_normalized, num_features=2)

print("Preprocessed Data:")
print(selected_gene_data)
```

Output:

```
>>>
    ================================================ RESTART: C:/Users/Al-Saif/AppData/Local/Programs/
    Original Gene Data:
    [[1.2 3.4 2.1 nan]
     [2.3 4.5 nan 3.2]
     [3.4 nan 1.8 5.6]
     [nan 2.5 4.3 6.7]]

    Cleaned Gene Data:
    [[12.   3.4  2.1  nan]
     [23.   4.5  nan  3.2]
     [34.   nan  1.8  5.6]
     [ nan  2.5  4.3  6.7]]
    Preprocessed Data:
    [[0.56190476 0.12      ]
     [0.         0.37333333]
     [0.68571429 0.        ]
     [1.         1.        ]]
>>>
```