Lab 4

Sequences

Sup'Biotech 3

Python

Pierre Parutto

October 19, 2016

Preamble

Document Property

Authors	Pierre Parutto
Version	1.0
Number of pages	9

Contact

Contact the assistant team at: supbiotech-bioinfo-bt3@googlegroups.com

Copyright

Contents

1 Introduction **3**

2 DNA Sequence Characteristics **3**
2.1 Adenine Frequency . . . 3
Example . . . 3
2.2 Nucleotide Frequency . . . 3
Example . . . 3

3 Modification Of Sequences **4**
3.1 Complementary Strand . . . 4
Example . . . 4
3.2 Reverse Complementary Sequence . . . 5
Example . . . 5
3.3 Translation Of An Open Reading Frame . . . 6
Example . . . 6

4 Distance Between DNA Sequences **7**
4.1 Hamming Distance . . . 7
Example . . . 7
4.2 Weighted Distance . . . 8
Example . . . 9

1 Introduction

In this fifth lab, we will manipulate strings, lists and dictionaries.

2 DNA Sequence Characteristics

2.1 Adenine Frequency

Write a function `freq_A(s: str) -> float` that returns the frequency of the adenine nucleotide in the DNA sequence `s`.

Example

```
>>> freq_A("AAATGT")
0.5
>>> freq_A("TGTGTCG")
0.0
```

Correction:
First version with a `while` loop:

```
def freq_A(s):
    i = 0
    fA = 0.0
    while i < len(s):
        if s[i] == "A":
            fA = fA + 1
        i = i + 1
    return fA / len(s)
```

Second version with a `for` loop:

```
def freq_A(s):
    fA = 0.0
    for e in s:
        if e == "A":
            fA = fA + 1
    return fA / len(s)
```

You can see that this version is shorter than the first one, only the logic of the function remains, all the technical details have been abstracted away.

2.2 Nucleotide Frequency

Write a function `freq_nuc(s: str) -> dict` that returns the frequency of each nucleotide on the DNA sequence `s`. The returned value is a dictionary of the form: `nucleotide:float`.

Example

```
>>> freq_nuc("ATGC")
{"A":0.25,"T":0.25,"G":0.25,"C":0.25}
>>> freq_nuc("AAAGCG")
```

```
{"A":0.5,"T":0.0,"G":0.33333,"C": 0.16666}
```

Correction:
A first version with a `while` loop:

```
def freq_nuc(s):
        d = {"A":0.0, "T":0.0, "G":0.0, "C":0.0}

        if len(s) == 0:
                return d

        cpt = 0
        while cpt < len(s):
                d[s[cpt]] = d[s[cpt]] + 1.0
                cpt = cpt + 1

        tmp = "ATGC"
        i = 0
        while i < len(tmp):
                d[tmp[i]] = d[tmp[i]] / len(s)
                i = i + 1

        return d
```

The second version with a `for` loop:

```
def freq_nuc(s):
        d = {"A":0.0, "T":0.0, "G":0.0, "C":0.0}

        if len(s) == 0:
                return d

        for c in s:
                d[c] = d[c] + 1.0

       for n in "ATGC":
                d[n] = d[n] / len(s)

        return d
```

3 Modification Of Sequences

3.1 Complementary Strand

Write a function `complementary_DNA(s: str) -> str` that returns the complementary sequence corresponding to the DNA sequence `s`.

Example

```
>>> complementary_DNA("ATTTGC")
"TAAACG"
```

```
>>> complementary_DNA("CCGTA")
"GGCAT"
```

Correction:
A first version with a `while` loop:

```
def complementary_DNA(s):
    res = ""
    i = 0
    comp = {"A":"T", "G":"C", "C":"G", "T":"A"}
    while i < len(s):
        res = res + comp[s[i]]
        i = i + 1
    return res
```

And the second (shorter) version with a `for` loop:

```
def complementary_DNA(s):
    res = ""
    comp = {"A":"T", "G":"C", "C":"G", "T":"A"}
    for e in s:
        res = res + comp[e]
    return res
```

3.2 Reverse Complementary Sequence

Write a function `rev_complementary_DNA(s: str) -> str` that returns the reverse complementary sequence corresponding to the DNA sequence `s`.

Example

```
>>> rev_complementary_DNA("ATTTGC")
"GCAAAT"
>>> rev_complementary_DNA("CCGTA")
"TACGG"
```

Correction:
A first version with a `while` loop:

```
def rev_complementary_DNA(s):
    res = ""
    i = 0
    comp = {"A":"T", "G":"C", "C":"G", "T":"A"}
    while i < len(s):
        res = comp[s[i]] + res
        i = i + 1
    return res
```

And the second (shorter) version with a `for` loop:

```
def rev_complementary_DNA(s):
    res = ""
    comp = {"A":"T", "G":"C", "C":"G", "T":"A"}
    for e in s:
        res = comp[e] + res
    return res
```

3.3 Translation Of An Open Reading Frame

Write a function `translate(s: str) -> str` that returns the amino acid sequence corresponding to the RNA sequence `s`.

The following dictionary gives the mapping between codon and amino acid:

```
gencode = {
    "UUU":"Phe", "UCU":"Ser", "UAU":"Tyr", "UGU":"Cys",
    "UUC":"Phe", "UCC":"Ser", "UAC":"Tyr", "UGC":"Cys",
    "UUA":"Leu", "UCA":"Ser", "UAA":"STOP", "UGA":"STOP",
    "UUG":"Leu", "UCG":"Ser", "UAG":"STOP", "UGG":"Trp",
    "CUU":"Leu", "CCU":"Pro", "CAU":"His", "CGU":"Arg",
    "CUC":"Leu", "CCC":"Pro", "CAC":"His", "CGC":"Arg",
    "CUA":"Leu", "CCA":"Pro", "CAA":"Gln", "CGA":"Arg",
    "CUG":"Leu", "CCG":"Pro", "CAG":"Gln", "CGG":"Arg",
    "AUU":"Ile", "ACU":"Thr", "AAU":"Asn", "AGU":"Ser",
    "AUC":"Ile", "ACC":"Thr", "AAC":"Asn", "AGC":"Ser",
    "AUA":"Ile", "ACA":"Thr", "AAA":"Lys", "AGA":"Arg",
    "AUG":"Met", "ACG":"Thr", "AAG":"Lys", "AGG":"Arg",
    "GUU":"Val", "GCU":"Ala", "GAU":"Asp", "GGU":"Gly",
    "GUC":"Val", "GCC":"Ala", "GAC":"Asp", "GGC":"Gly",
    "GUA":"Val", "GCA":"Ala", "GAA":"Glu", "GGA":"Gly",
    "GUG":"Val", "GCG":"Ala", "GAG":"Glu", "GGG":"Gly",
}
```

Note: The file *gencode.py* on your intranet contains this dictionary.

Example

```
>>> translate("UUCUCACGU")
"PheSerArg"
>>> translate("UUCUCACGUUGAAGC")
"PheSerArg"
```

Correction:
A first version with a `while` loop:

```
def translate(arn):
        res = ""
        you = 0
        while you < len(arn) - 2:
                AA = gencode[arn[you:you+3]]
                if AA == "STOP":
                        return res
                res = res + AA
                you = you + 3
        return res
```

for

```
def translate(arn):
        res = ""
       for i in range(0, len(arn), 3):
                AA = gencode[arn[i:i+3]]
                if AA == "STOP":
                        return res
                res = res + AA
        return res
```

range

range

4 Distance Between DNA Sequences

4.1 Hamming Distance

The Hamming distance is the simplest method you can use to compare two sequences s_1 and s_2. It works as follows: we compare s_1 and s_2 character by character if they are different we add +1 to the distance and 0 if they are identical. Mathematically, it gives the following formula:

$$\text{hamming}(s_1, s_2) = \sum_{i=0}^{N} \mathbb{1}_{s_1[i],s_2[i]}$$

where:

- N is the length of the sequences;
- $\mathbb{1}_{n_1,n_2} = \begin{cases} 1 & \text{if } n_1 \neq n_2 \\ 0 & \text{otherwise} \end{cases}$ is a function that evaluates to 1 if n_1 and n_2 are different and 0 otherwise.

Write a function `hamming(s1: str, s2: str) -> int` that returns the Hamming distance between the two DNA sequences `s1` and `s2`. We consider that `s1` and `s2` have the same size.

Example

```
>>> hamming("ATT", "TTA")
2
>>> hamming("AAA", "GGG")
3
```

Correction:
First version with a while loop:

```
def hamming(s1, s2):
       res = 0
       i = 0
       while i < len(s1):
              if s1[i] != s2[i]:
                     res = res + 1
              i = i + 1
       return res
```

The second version with a for loop:

```
def hamming(s1, s2):
       res = 0
      for i in range(len(s1)):
              if s1[i] != s2[i]:
                     res = res + 1
       return res
```

In this version, as we have to access the values from the two sequences at the same time we cannot use a loop of the form `for c in s1`. Instead we use the `range` function to get access to the indices in the `for` loop.

4.2 Weighted Distance

The weighted distance is very similar to the Hamming distance except that we will now add different values to the distance depending of the type of mismatch. The new mathematical formula is the following:

$$\mathrm{d}_{\text{weighted}}(s_1, s_2) = \sum_{i=0}^{N} w_{s_1[i],s_2[i]} * \mathbb{1}_{s_1[i],s_2[j]}$$

Where:

- N is the size of the sequences;
- $\mathbb{1}_{s_1[i],s_2[j]}$ is the same function as for the Hamming distance;
- $w_{s_1[i],s_2[j]}$ is the weight (or coefficient) associated to the mismatch between characters $s_1[i]$ and $s_2[i]$. We will represent it in Python using a dictionary.

This dictionary is used in the following way:

```
>>> w = {"A":{"A":0,"T":0.5,"G":-0.5,"C":0.3},
        "T":{"A":0.5,"T":0,"G":1.2,"C":-5},
        "G":{"A":-10,"T":1.2,"G":0,"C":0.3},
        "C":{"A":0.3,"T":-5,"G":5.3,"C":0}}
>>> w["A"]["T"]
0.5
>>> w["C"]["A"]
0.3
```

Write a function `weighted_dist(s1: str, s2: str, w: dict) -> float` that returns the weighted distance between the two DNA sequences `s1` and `s2` using the weights in `w`.

Example

```
>>> w = {"A":{"A":0,"T":0.5,"G":-0.5,"C":0.3}, \
  "T":{"A":0.5,"T":0,"G":1.2,"C":-5}, \
  "G":{"A":-10,"T":1.2,"G":0,"C":0.3}, \
  "C":{"A":0.3,"T":-5,"G":5.3,"C":0}}
>>> weighted_dist("ATT", "TTA", w)
1.0
>>> weighted_dist("AAAG", "GGGA", w)
-11.5
```

Correction:
The first version with a `while` loop:

```
def weighted_dist(s1, s2, w):
        res = 0
        i = 0
        while i < len(s1):
                res = res + w[s1[i]][s2[i]]
                i = i + 1
        return res
```

The second version with a `for` loop:

```
def weighted_dist(s1, s2, w):
        res = 0
        for i in range(len(s1)):
                res = res + w[s1[i]][s2[i]]
        return res
```