Homework

Python-2

Sup'Biotech 3

Python

Pierre Parutto

December 3, 2016

Preamble

Document Property

Authors	Pierre Parutto
Version	1.0
Number of pages	10

Contact

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

Copyright

Contents

1 Introduction

In this homework we will apply the basic programming skills that you have just acquired to crack some biology-related questions.

1.1 File Architecture

You must respect the following architecture for your work:

```
login_l
├── AUTHORS
└── src
    └── files.py
```

You must have a folder named with your login, this folder must contain:

1. **A text file** named *AUTHORS* containing your first and last names.
2. **A folder** named *src* that contains your code files.

1.2 Submission

- Deadline: until **Wednesday November 9, 23h42**;
- Submission: **a zip file** named: *login_l.zip* to upload on the bioinformatics intranet.

A bad architecture of your submission may result in a 2 points penalty.

1.3 Cheating

Basically, **DO NOT CHEAT**. Your work will be automatically tested against cheating. If two people are detected as cheaters, they will receive the **grade 0** for the homework and the administration will be told of their attempt. All detected cheaters are treated equally, I do not care who wrote the code and who took it.

2 Example

Here is an example of how to answer an homework question.

2.1 Question

File: *toto.py*

Write the function called `my_sum(a: int, b:int) -> int` that returns the sum of `a` and `b`.

2.2 Answer

The content of the file *toto.py* is thus:

```python
def my_sum(a, b):
    return a + b
```

2.3 Testing your code

Although not mandatory in this first homework, **I strongly advise you to test your code**. You can do that by calling your function with some values and checking that the answer is what you expect. For example, your file *toto.py* becomes:

```python
def my_sum(a, b):
    return a + b

print(my_sum(5, 9) == 16)
print(my_sum(5, 0) == 5)
print(my_sum(5, -5) == 0)
```

And the Python output is:

```
True
True
True
```

You must remove all your tests before you submit your code. In your code files I want only function declaractions and nothing else.

3 Introduction

The primary sequence of a nucleic molecule is the succession of bases it contains. These sequences are conventionally represented from the 5' end to the 3' end.

Representing Bases Bases constituting nucleic molecules are represented in computers by a single symbol defined by the IUPAC:

Symbol	Description	Bases Represented
A	Adenine	A
T	Thymine	T
G	Guanine	G
C	Cytosine	C
U	Uracile	U
I	Inosine	I
W	Weak	AT
S	Strong	GC
M	Amino	A C
K	Keto	TG
R	Purine	A G
Y	Pyrimidine	T C
B	not A	TGC
V	not T	A GC
H	not G	AT C
D	not C	ATG
N	Any	ATGC

Primary Sequences The primary sequence of a nucleic molecule is traditionally represented as a string containing only the previous symbols. For example: `"ATTGT"` or `"AUCCG"`.

Ambiguous sequences The characters after the 6th row of the table are used to represent an ambiguity in the sequence. For example a fail in the sequencing or equivalent bases in a specific context. For example: `"AGR"` represents both the sequences `"AGA"` and `"AGG"`.

Alphabet We call alphabet an ensemble of characters. We say that a sequence is constructed over an alphabet if it contains only characters included in it. For example, considering the alphabet $\Omega =$ `ATGC`, a sequence constructed over Ω could be `"ATTTG"` or `"ACACT"` but **not** `"AECG"`.

4 Base Counting

File: *counting.py*

4.1 Bases And Distance (4 points)

Write the function `count_bases_dist(s: str, p: int, max_dist: int) -> dict` that given a sequence `s` over the alphabet `AUCG`, a valid position `p` in `s` and a maximal distance `max_dist`, returns the number of bases of each type appearing up to `max_dist` characters (included) away from `p` (including the character at `p`). The returned value is a dictionnary that associates a base (a string) to the number of times it appears (an int).

Note: If there are less than `max_dist` bases between `p` and any end of `s`, you have to stop at the border.

Example

```
>>> count_bases_dist("AUCGA", 2, 1)
{"A":0, "U":1, "C":1, "G":1}
>>> count_bases_dist("AUCCAUAUCG", 5, 3)
{"A":2, "U":2, "C":3, "G":0}
>>> count_bases_dist("CGCGAUGC", 6, 4)
{"A":1, "U": 1, "C":2, "G": 2}
```

Correction:

```
def count_bases_dist(s, p, max_dist):
    res = {"A": 0, "U": 0, "C": 0, "G": 0}

    for i in range(max(0, p - max_dist), min(len(s), p + max_dist + 1)):
        res[s[i]] = res[s[i]] + 1
    return res
```

4.2 Count Codons (3 points)

Write the function `count_codons(s: str) -> dict` that given a nucleic sequence over the alphabet `AUCG` returns a dictionnary associating each codon **appearing in** `s` to the number of times it appears.

Note: Codons not appearing in `s` must not appear in the dictionary.

Note: Codons start at the first character of `s` and do not overlap. For example the sequence `"AUCAGGCAC"` contains the three codons: `"AUC"`, `"AGG"` and `"CAC"`.

Note: When the last characters of s form an incomplete codon, for example if there are only one or two characters remaining in s, then just ignore them. For example the sequence "AUCAG" contains only one codon: "AUC".

Example

```
>>> count_codons("AUC")
{"AUC": 1}
>>> count_codons("AUCGG")
{"AUC": 1}
>>> count_codons("AGCCCGAUUAGC")
{"AGC": 2, "CCG": 1, "AUU": 1}
```

Correction:

```
def count_codons(s):
    res = {}

    for i in range(0, len(s)-2, 3):
        if s[i:i+3] in res:
            res[s[i:i+3]] = res[s[i:i+3]] + 1
        else:
            res[s[i:i+3]] = 1
    return res
```

5 Base Pairing

File: *pairing.py*

5.1 Pairing Types For RNA Sequences (2 points)

Each base from an RNA molecule car pair to another base, through the formation of hydrogen bonds, either from the same molecule or from another. Two types of pairing are to be distinguished:

- Watson-Crick (canonical) pairing:

Base	Pairs with
A	U
U	A
C	G
G	C

- Wobble (non-canonical) pairing:

Base	Pairs with
G	U
U	G

Bases pairing together are called complementary bases.

Wobble pairing consists in an ill-formed pairing where only 2 hydrogen bonds are formed between the molecules and are thus more fragile.

Write the function `pair_type(b1: str, b2: str) -> str` that given two bases `b1` and `b2` from the alphabet `AUGC`, returns `"NO"` if `b1` and `b2` cannot pair, `"WC"` if they can form a Watson-Crick pairing and `"WO"` if they can from a wobble pairing.

Example

```
>>> pair_type("A", "U")
"WC"
>>> pair_type("A", "A")
"NO"
>>> pair_type("U", "G")
"WO"
```

Correction:

```
def pair_type(b1, b2):
    if b1 + b2 == "AU" or b1 + b2 == "UA" or b1 + b2 == "GC" or b1 + b2 == "CG":
        return "WC"
    elif b1 + b2 == "GU" or b1 + b2 == "UG":
        return "WO"
    else:
        return "NO"
```

5.2 Is Recognized (3 points)

Transfer RNA (tRNA) are small RNA sequences, involved in the association of codons with their associated amino acids, acting during the transcription of RNA sequences into proteins. tRNAs can bind on one end to an amino acid molecule and on the other end exhibit a triplet of bases complementary to the codon corresponding to the amino acid on the other side. Some tRNAs exhibit a 6th type of base called isosine that can form the following wobble pairings:

Base	Pairs with
I	A
I	C
I	U

This is an efficient way of implementing the redundancy observed in the genetic code.

Write the function `is_recognized(trna: str, rna: str) -> str` that given the triplet (string of size 3) of bases `trna` from the alphabet `AUGCI` exhibited from a tRNA molecule and the triplet of bases `rna` from the alphabet `AUCG`, returns `True` if `trna` recognizes `rna`, that is all three bases from the tRNA can pair (both Watson-Crick and wobble) to the corresponding bases on the RNA, and `False` otherwise.

Example

```
>>> is_recognized("AUG", "UAC")
True
>>> is_recognized("AUG", "UUC")
False
>>> is_recognized("IAI", "AUU")
True
>>> is_recognized("IAI", "ACA")
False
```

Correction:

```
def is_recognized(trna, rna):
    reco = {"A": "UI", "U": "AI", "C": "GI", "G": "C"}

    res = True

    i = 0
    while i < len(rna) and res == True:
        tmp = False
        for b in reco[rna[i]]:
            if b == trna[i]:
                tmp = True
        res = tmp
        i = i + 1
    return res
```

5.3 Recognized Bases (4 points)

This question is based on the descriptions of the mechanisms presented on the previous question.

Write the function `recognizes(trna: str) -> list` that given the triplet (string of size 3) of bases `trna` from the alphabet `AUGCI` exhibited by a tRNA molecule, returns the list of RNA triplets (string of size 3) from the alphabet `AUCG` that are recognized by `trna`.

Example

```
>>> recognizes("AUG")
['UAU', 'UAC', 'UGU', 'UGC']
>>> recognizes("IAU")
['AUA', 'AUG', 'UUA', 'UUG', 'CUA', 'CUG']
>>> recognizes("IAI")
["AUA", "UUA", "CUA", "AUU", "UUU", "CUU", "AUC", "UUC", "CUC"]
```

Correction:

```
def recognizes(trna):
    reco = {"U": "AG", "A": "U", "G": "UC", "C": "G", "I": "AUC"}
    res = [""]
    for b in trna:
        new_res = []
        for r in res:
            for c in reco[b]:
                new_res.append(r + c)
        res = new_res
    return res
```

6 Distance Between Nucleic Sequences

File: *distance.py*

6.1 Hamming Distance Between Multiple Sequences (4 points)

The Hamming distance (that we studied in lab4) between the sequences s_1 and s_2 is defined as follows:

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

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.

This formulation can be extended to define the distance between n sequences $s_1, \ldots, s_n$ of the same size N as follows:

$$\text{hamming}(s_1, \ldots, s_n) = \sum_{i=0}^{N} \left(\prod_{k=1,l=2,k\neq l}^{n} \mathbb{1}_{s_k[i], s_l[i]} \right)$$

Meaning that the distance between the ensemble of sequence is +1 each time there is **at least** differing character in a column.

Write the function `hamming_multiple(seqs: list) -> int` that given a list `seqs` of sequences of identical size, returns their associated hamming distance.

Example

- Consider the following sequences:

x_1	AAAAAA
x_2	AAAAAA
x_3	AAAAAA

```
>>> hamming_multiple(["AAAAAA", "AAAAAA", "AAAAAA"])
0
```

- Consider the following sequences:

x_1	AAAAAA
x_2	AAAAAA
x_3	AAAAAC

```
>>> hamming_multiple(["AAAAAA", "AAAAAA", "AAAAAC"])
1
```

- Consider the following sequences:

x_1	ATTAGC
x_2	ATTACC
x_3	ATTACG
x_4	TTTAGG

```
>>> ss = ["ATTAGC", "ATTACC", "ATTACG", "TTTAGG"]
>>> hamming_multiple(ss)
3
```

Correction:

```
def hamming_multiple(seqs):
    res = 0

    for i in range(len(seqs[0])):
        d = 0
        for k in range(len(seqs)):
            if seqs[k][i] != seqs[0][i]:
                d = 1
        res = res + d
    return res
```