PS1 (1)

February 19, 2024

Name: Collin Real

abc123: yhi267

Remember, it is okay to talk to your classmates about the homework, but do not share solutions. You should complete the homework on your own. Below, please list all of your peers ("collaborators") that you discussed the homework with.

Colaborators: - Joaquin Ramirez - Seth Harris - Leo Salazar

1 Problem Set 1 (PS 1)

Complete the 4 functions as directed to:

- validate the format of a string
- compute probabilities and store them in a lookup table (dictionary)
- generate a string deterministically based on a given string

Each function is worth 1 to 4 points, for a total of 10 points. Other assignments will be more complicated, and worth more points, for this I just want to make sure you are in the correct course. I care mainly about correctness of the implementation, but reserve the right to deduct points for code that is difficult to read.

This assignment will test your skills at implementing algorithms in Python from scratch. If this is your first experience with traditional programming assignments, then it may take some time to figure everything out. Please schedule time to meet with meet with me if you need help. I recommend you start early. For others with significant programming experience, it will not take long.

1.1 Submission Instructions

After completing the exercises below, generate a PDF *or* HTML of the code **with** outputs. Then, create a zip file containing both the completed exercise and the generated PDF/HTML. You are **required** to check the PDF/HTML to make sure all the code **and** outputs are clearly visible and easy to read. If your code goes off the page, you should reduce the line size. I generally recommend not going over 80 characters.

Finally, name the zip file using a combination of the assigment ID and your name, e.g., ps1_rios.zip. Submit your homework on Blackboard.

1.2 Preliminary Information

For the homework exercises, you will need to write code in the specified functions. The variable specified in the open and close parenthesis "my_function(**variable_name**)" of the function should **NOT** be reassigned. They will hold important information—such as the string you should process—which is required to complete each task. Your job is to write code that returns the required value for each exercise.

Please refer to the example below to understand how functions work in python:

```
def my_function(my_variable):
    # Your code goes here
    print("The parameter is {}".format(my_variable))
    new_variable = my_variable + 100
    return new_variable

# my_variable in my_function() will be assigned 42.
return_value = my_function(42) # return value will be asigned the value given by the "return"
print("The function returned the value {}".format(return_value))
```

If you run the code above, you will get the following output:

```
The parameter is 42
The function returned the value 142
```

Please note that you do **NOT** need to create new functions. The functions have already been specified. You simply need to fill them out with code.

1.3 Exercise 1 (4 points)

Checks whether the string is a valid employee ID. An employee ID is valid if and only if it consists only of 6-10 alphabetic characters (letters), followed by 2 numeric digits. Implement the validation **without** any external libraries (i.e., do not use the "re" package). You may use the .isalpha() and .isdigit() methods, variables, standard data structures such as lists, if statements, loops, etc.

[1]:
```
# Example use of .isalpha() and .isdigit(). This code is just to help, you can␣
↪delete this cell if you want.
print("32".isdigit()) # if the string is a digit, .isdigit() will return True,␣
↪otherwise it returns false
print("lkjsdaf".isdigit()) # Here is an example of .isdigit() returning False
print("lskdjf".isalpha()) # In this case we can check if all characters are␣
↪"alphabetic" by using .isalpha() it returns True here.
print("ab1".isalpha()) # Here is an example of .isalpha() resulting in a False
print(len("ALKJDSLKFJDS")) # You can check how many characters are in a string␣
↪by using the len() function.
```

```
True
False
True
False
12
```

[3]:
```python
# # Write code here. Do NOT delete this cell.
def validate(s):
    letters = s[:len(s)-2]
    # print(letters)
    numbers = s[len(s)-2:]
    # print(numbers)
    combined = letters.isalpha() and numbers.isdigit()

    if len(s) in range(8,13):
        return combined
    else:
        return False  # The code should return a boolean True or boolean False,␣
 ↪depending on the input.
```

The lines below give example inputs and correct outputs using asserts, and can be run to test the code. Passing these tests is necessary, but **NOT** sufficient to guarantee your implementation is correct. You may add additional test cases, but do not remove any tests.

[6]:
```python
# var = validate('AbcdEf00')
# print(var)
assert(validate('AbcdEf00') == True)
assert(validate('swAbcdEf00') == True)
assert(validate('swAbcdEfef00') == True)
assert(validate('AbcdEfAbcdEf00') == False)
assert(validate('Abcdef000000') == False)
assert(validate('Abcd00') == False)
assert(validate('00') == False)
assert(validate('AbcdEf119$') == False)
assert(validate('$0RQLpCHz49') == False)
print("Asserts Completed Successfully")
```

```
Asserts Completed Successfully
```

1.4 Exercise 2 (3 points)

Given a sequence of the DNA bases {A, C, G, T}, stored as a string, returns a probability table in a data structure such that one base can be looked up to get the probability p(base). More specifically, write code to return a dictionary such that the value for each key is the probably of that key occuring the the string. For example, give the string “abb”, p(a) is equal to $\frac{1}{3}$ and p(b) is $\frac{2}{3}$ Simply, divide the number of times each character appears by the total number of characters. So, the code should return the following dictionary:

```
{"a": 0.33333, "b": 0.6666}
```

You may use the collections module, but no other libraries.

[24]:
```python
def dna_prob1(seq):
    init_dict = {}
    for letter in seq:
        # print(letter)
        if letter in init_dict:
            init_dict[letter] += 1
            # print(init_dict)
        else:
            init_dict[letter] = 1
            # print(init_dict)

    total_letters = len(seq)
    # print(total_letters)
    print(init_dict.items())
    table_probs = {letter: letter_count/total_letters for letter, letter_count␣
↪in init_dict.items()}
    return table_probs # The code should return a dictionary
```

The lines below give example inputs and correct outputs using asserts, and can be run to test the code. Passing these tests is **NOT** sufficient to guarantee your implementation is correct. You may add additional test cases, but do not remove any tests.

[25]:
```
tbl = dna_prob1('ATCGATTGAGCTCTAGCG')
print(tbl)
assert(tbl['A'] == 4/18)
assert(tbl['T'] == 5/18)
assert(tbl['G'] == 5/18)
assert(tbl['C'] == 4/18)
print("Asserts Completed Successfully")
```

```
dict_items([('A', 4), ('T', 5), ('C', 4), ('G', 5)])
{'A': 0.2222222222222222, 'T': 0.2777777777777778, 'C': 0.2222222222222222, 'G':
0.2777777777777778}
Asserts Completed Successfully
```

1.5 Exercise 3 (3 points)

Given a string representing a sequence of DNA bases, returns the paired sequence, also as a string, where A is always paired with T and C with G, i.e., replace A with T, T with A, …

Do not use any libraries.

Hint: this can be done in one line. (More than one line is okay too.)

[28]:
```
def dna_bp(seq):
    mapping_dict = {'A': 'T', 'C': 'G', 'G': 'C', 'T': 'A'}
    updated_seq = ''.join([mapping_dict[i] for i in seq])
    return updated_seq # The code should return a string
```

The lines below give example inputs and correct outputs using asserts, and can be run to test the code. Passing these tests is **NOT** sufficient to guarantee your implementation is correct. You may add additional test cases, but do not remove any tests.

```
[30]: assert(dna_bp('ATCGATTGAGCTCTAGCG') == 'TAGCTAACTCGAGATCGC')
      seq = 'ATCGATTGAGCTCTAGCG'
      print(dna_bp(seq))
      print("Asserts Completed Successfully")
```

```
TAGCTAACTCGAGATCGC
Asserts Completed Successfully
```

1.6 Extra Credit (2 points)

Given a sequence of the DNA bases {A, C, G, T}, stored as a string, returns a conditional probability table in a data structure such that one base (b1) can be looked up, and then a second (b2), to get the probability p(b2 | b1) of the second base occurring immediately after the first. (Assumes the length of seq is >= 3, and that the probability of any b1 and b2 which have never been seen together is 0. Ignores the probability that b1 will be followed by the end of the string character.)

Here is an example:

`GAAAGG`

P(A | G) = probability of A given the letter G occured directly before it = 1/2 = 0.5. The denominator is 2 instead of 3 because the last G does not have a letter after it. The denominator should be the number of times a letter appears with annother letter directly after it.

P(G | G) = probability of G given the letter G occured directly before it = 1/2 = 0.5

P(G | A) = probability of G given the letter A occured directly before it = 1/3 = 0.33

P(A | A) = probability of A given the letter A occured directly before it = 2/3 = .66

The dictionary returned by dna_prob2 should be:

```
{'A': {'A': 2/3, 'G': 1/3, 'T': 0, 'C': 0}, 'G': {'A': 1/2, 'G': 1/2, 'T': 0, 'C': 0}, 'T': {'
```

Notice this is a "nested" dictionary, where the value of the outer dictionary is also a dictionary. All of the keys are strings.

Note: 2 extra points adds 20% to your final homework grade. This can improve your grade substantially. So, I recommend working on this, if you are worried about the quizzes and Midterm.

```
[45]: def dna_prob2(seq):
          dict_temp = {}

          for i in range(len(seq)-1):
              # print(i)
              dict_temp[seq[i:i+2]] = dict_temp.get(seq[i:i+2],0) + 1
              # print(dict_temp)

          dna_bases = 'ACGT'
          final_dict = {}

          for i in dna_bases:
              # print(i)
```

```
        final_dict[i] = {}
        # print(final_dict)
        total_count = 0

        for j in dna_bases:
            # print(dict_temp)
            if i+j not in dict_temp:
                final_dict[i][j] = 0
                # print(final_dict)
            else:
                final_dict[i][j] = dict_temp[i+j]
                # print(dict_temp)
                total_count += final_dict[i][j]
                # print(total_count)
                # print(final_dict)
        for j in dna_bases:
            # print(total_count)
            final_dict[i][j] /= total_count
            # print(final_dict)
    return final_dict # The code should return a dictionary
```

The lines below give example inputs and correct outputs using asserts, and can be run to test the code. Passing these tests is **NOT** sufficient to guarantee your implementation is correct. You may add additional test cases, but do not remove any tests.

[46]:
```
tbl = dna_prob2('ATCGATTGAGCTCTAGCG')
print(tbl)
assert(tbl['A']['A'] == 0.0)
assert(tbl['A']['C'] == 0.0)
assert(tbl['A']['G'] == 0.5)
assert(tbl['A']['T'] == 0.5)
assert(tbl['C']['A'] == 0.0)
assert(tbl['C']['C'] == 0.0)
assert(tbl['C']['G'] == 0.5)
assert(tbl['C']['T'] == 0.5)
assert(tbl['G']['A'] == 0.5)
assert(tbl['G']['C'] == 0.5)
assert(tbl['G']['G'] == 0.0)
assert(tbl['G']['T'] == 0.0)
assert(tbl['T']['A'] == 0.2)
assert(tbl['T']['C'] == 0.4)
assert(tbl['T']['G'] == 0.2)
assert(tbl['T']['T'] == 0.2)
print("Asserts Completed Successfully")
```

```
{'A': {'A': 0.0, 'C': 0.0, 'G': 0.5, 'T': 0.5}, 'C': {'A': 0.0, 'C': 0.0, 'G':
0.5, 'T': 0.5}, 'G': {'A': 0.5, 'C': 0.5, 'G': 0.0, 'T': 0.0}, 'T': {'A': 0.2,
'C': 0.4, 'G': 0.2, 'T': 0.2}}
```

```
Asserts Completed Successfully
```

[ ]:

```
```