Your code should not only be correct, but also follow the CSC120 Style Rules posted with Assignment 1. Go over the following snippets of code and identify the formatting and programming style errors that violate these rules. There are 4 to 5 errors in each function definition.

(a) `def repeat_word(word, RepeatNum):
    """ (string, integer) -> string
    Precondition: RepeatNum >= 0
    Return word repeated the specified number of times.
    >>> repeat_word('Marcia ', 3)
    'Marcia Marcia Marcia '
    >>> repeat_word('Buffalo ', 5)
    'Buffalo Buffalo Buffalo Buffalo Buffalo '
    """
    return word*RepeatNum

(b) `def function_name(first_name, last_name):
    """ (str, str) -> bool
    Returns True iff last_name is longer than first_name by comparing the values produced after calling built-in function len on first_name and last_name.
    >>> function_name("Myrto", "Papadopoulou")
    True
    >>> function_name("Jules", "Verne")
    False
    """
    if len(last_name) > len(first_name):
        return True
    else:
        return False

(c) `def passing_grade(g):
    """ (int) -> bool
    Precondition: g must be of type int.
    Return True iff g is greater than or equal to 50.
    """
    return (g>=50)==True

def passed_two_courses(grade1, grade2):
    """ (int, int) -> bool
    Returns True iff both grade1 and grade2 are greater than or equal to 50.
    >>> passed_two_courses(60, 80)
    True
    >>> passed_two_courses(99, 45)
    False
    """
    return (grade1 >= 50) and (grade2 >= 50)
CSC120 Style Rules (a slightly more detailed list is referenced in Assignment’s 1 handout)

Formatting Style
1. Use Python style conventions for your function and variable names. This means that you should use pothole case: lowercase letters with words separated by underscores (_) to improve readability. For example, the variable name dog_name is good but dogName is not.

2. Choose meaningful names for your functions and variables. For example, num_bright_spots is more helpful and readable than nbs.

3. Do not use the tab character to indent code. Instead, use four spaces for each indentation level.

4. Put a blank space before and after every operator. For example, the first line below is good but the second line is not. :

   b = 2 > x and x < 20  # Alternatively written as: b = 2 < x < 20
   b= 2>x and x<20

5. For each function, write a docstring according to our design recipe (Sept. 22 lecture). (See below for guidelines on the content of your docstrings.) Put the docstring’s closing quotation marks on their own line.

6. Each line must be less than 80 characters long including spaces. In Wing, the red vertical line indicates 80 characters. See Style Rules on the website for advice on how to break up long lines.

7. Within a docstring, put a blank line between the type contract and description, and between the description and the examples.

8. Put a blank line between the docstring and the function body.

Programming Style
1. Do not compare a Boolean expression to True or to False. For example, the first is good, the second is not:

   if a and b:  
       return 100

   if (a and b) == True:  
       return 100

2. Replace if statements of this form:

   if x > 100:  
       return True
   else:  
       return False

   with a single-line statement like this:

   return x > 100

3. Avoid duplicate code by calling on helper functions.

Docstring:
Follow these rules when writing the description portion of the docstring:

1. Describe precisely what the function does.

2. Do not reveal how the function does it.

3. Make the purpose of every parameter clear.

4. Use the name of every parameter.

5. For functions that return values, be clear on what the return value represents.

6. Explain any conditions that the function assumes are true; these are called Preconditions. These conditions should not refer to type requirements, because those are already covered by the type contract. However, sometimes there are conditions not covered by types, and you should include those. For example, if a function requires a parameters x to be a non-negative number, write: x \geq 0.

7. Be concise and grammatically correct.

8. Write the description as a command (e.g., Return the first ...) rather than a statement (e.g., Returns the first ...).
Testing Code - Finding Bugs

Before you start writing any code for any function, you need to think about how you will make sure your code is correct. You have to know in your own mind, for every function, what the expected outputs (or return values) are for all possible values of the parameter. That way, after you’ve written your functions, you can check whether they work as you expect.

Q. Would it be possible to test for every possible argument value?

A.

Selecting Testcases

We need to limit our testing to a relatively small set of values. The best way to do this is to pick some situations that are representative of others, and also to introduce both “usual” and “unusual” situations.

For example, if your function takes a string as a parameter, then predicting what your function will do with the empty string '' as a parameter, is an unusual test case. Unusual test cases are often referred to as corner cases. A representative test case might be the string 'r' (or 'K') standing for all one-character strings.

As another example, if your code has a series of if-statements, you will want to have at least one test case for each of the possible conditions. If you had a function that took a temperature as input and then printed whether water was a solid, a liquid, or a gas at that temperature, you should test at least one case from each of those categories, as well as on all the boundaries.

Practice: Think up several test cases for the following functions and write them down as docstring examples. For every example you write, think about what it is you are trying to test for.

```python
def valid_index(L, list_index):
    """ (list of int, int) -> bool

    Return True iff list_index is a valid index for L.

    # <put your examples/test cases here>
    """
    return list_index <= len(L)
```

There is an error (bug) in the code – did your tests catch it?
def first_positive_number(my_list):
    """ (list of int) -> int
    Return the first positive number in my_list. Return -1 if there is no such number.
    # <put your examples/test cases here>
    """

    # This code is BUGGY!
    result = -1
    for item in my_list:
        if item >= 0:
            result = item
        else:
            return -1
    return result

There are various bugs in the code above – did your tests catch them?