Discussion 4

DSC 20, Spring 2023

Midterm 1 Prep + Review

Logistics

- Midterm 1 takes place 1pm, Friday April 28th (MANDE-B210 // normal lecture hall)
- No questions asked during exam
- Closed notes. Make sure to bring writing tools and your student ID

Topics

- Basic Data Types, Loops, Conditionals, slicing
- Lists, List Comprehension, Dictionaries, Mutability
- Files, Doctests, Asserts, return vs print

Basic Operations

+ - Numerical addition / concatenation operator

- - Numerical Subtraction operator

/ - Classic Division operator

// - Floor Division operator

***** - Numerical Multiplication / repetition operator

****** - Numerical Exponential operator

% - Numeric remainder operator

Basic Data Types

String - Data type for text

Int - Data type for whole numbers

Float - Data type for non-integers

Bool - True or False

Conditions

Logical Operators (in order priority):

- **not** - reverses the outcome of the following expression
- **and** - all expressions compared with “and” must be True to evaluate True
- **or** - at least one expression compared with “or” must be True to evaluate True

Comparison Operators (generates booleans):

- **==** - equality check
- **!=** - inequality check
- **>, >=, <, <=** directional check

Checkpoint

What do the following expressions evaluate to?

assume x, y = True, False and a, b, c = 1, 1.0, 2

1. not x and y

2. x or not y and x

3. a + b <= c

4. a + b != c

5. bool(-1)

Checkpoint Answers

What do the following expressions evaluate to?

assume x, y = True, False and a, b, c = 1, 1.0, 2

1. not x and y = **False**

not x = False => and condition to evaluate False is already met, therefore **False**

2. x ornot y and x = **True**

since x = True, the condition for or to evaluate True is already met, therefore **True**

3. a + b <= c = **True**

a + b = 2, 2 <= 2, therefore evaluate **True**

4. a + b != c = **False**

a + b = 2, 2 == 2, therefore evaluate **False**

5. bool(-1) = **True**

🤔

Conditional Statements

```
if (boolean expression):

    //Do stuff

elif (other boolean expression):
```

note: elif is optional, can have as many elifs as necessary

```
    //Do other stuff

else:
```

note: else is optional, will execute only if the conditions in the “if” and “elif” statements are not true

```
    //Do other other stuff
```

Loops

Loops are used to **repeat computations** many times.

- Two types of loops:
  - While loop: Uses logical conditions, do not know the number of iterations (as long as a condition is true, code will run)
  - For loop : Usually for when the number repetition is known.

Functions

- delineated by keyworddef
- usually contains areturnstatement (though not required for all functions)

Format :

def **function name**(formal parameters):

(function body)

return#optional

Semantics, but: arguments (in function calls) vs Parameters (in definition)

Print

print:generally to see what is happening inside the code. Represented value is None

In [1]:

```
def ex_print(value):
    print(value)
```

In [3]:

```
print(ex_print("test"))
```

```
test
None
```

"test" is the value being printed out by the function, None is the result of ex_print("test").

Using our class vocab, print is a "void function".

Return

return: how a function gives back a value

note : return is a "short-circuit", meaning that once a return statement executes, the function stops running.

In [4]:

```
def ex_return(value):
    return value
```

In [5]:

```
print(ex_return("test"))
```

```
test
```

"test" is the result of ex_return("test"). Using our class vocab, print is **not** a "void function".

Lists

- Mutable vector of values
- Can store any data type, multiple types at a time
- Elements are acccessed via indexing

List Slicing

- Format is similar to range ~ lst[start: stop: step] - [inclusive: exclusive]
- lst[::-1] reverses the list

List Comprehension

- Fancy, shorthand method of writing for loops
- Syntax changes depending on use case
- can be nested, just like lists

Syntax

  - [x for x in iterable]
  - [x for x in iterable if (condition)]
  - [x if (condition) else y for x in iterable]
  - [x if (condition) else y if (condition) else z for x in iterable]

Tuples

- Immutable vector of values
- Can store any data type, multiple types at a time
- Elements are acccessed via indexing

It's basically a list that can't be changed, though data inside it can still change.

Checkpoint

part 1

Assume the following code has been ran:

```
In [15]: lst = [(1,2), (3,'a'), ([4],5)]
```

What does the following line of code represent?

```
In [57]: x = lst[2][0]
```

```
In [58]: x
```

```
Out[58]: [4]
```

```
In [50]: x+=[6]
```

What will be shown if I run 'print(lst)'?

```
In [46]: print(lst)
```

```
[(1, 2), (3, 'a'), ([4, 6], 5)]
```

Explanation

In this code, x is set to a list, which means that it's an actual object (there's a reference to the original value). This means that changes that happens to x will be maintained within the list in the original data structure. Though the list sits in a tuple which may lead to the assumption that even the list inside cannot change as tuples are immutable, this is not the case. A tuple itself is unchangeable (we cannot change, add or remove items), but the elements inside of it can still change based on the situation.

Checkpoint

part 2

Assume the following code has been ran:

```
In [51]: lst = [(1,2), (3,'a'), ([4],5)]
```

What does the following line of code represent?

```
In [52]: x = lst[0][0]
```

```
In [53]: x
```

```
Out[53]: 1
```

```
In [54]: x+=1
```

What will be shown if I run 'print(lst)'?

```
In [55]: print(lst)
```

```
[(1, 2), (3, 'a'), ([4], 5)]
```

Explanation

In this code, x is set to a basic data type (int = 1), which means that it's not an object (there's no reference to the original value). This means that when you modify x, the change is not reflected in the original data structure.

Dictionaries

- Mutable storage of key, value pairs
- Can store any data type, multiple at a time
- Elements are acccessed via keys
- keys must be **hashable** and **unique**

note: hashablility correlates to the stability of the data - essentially, **data that can't change is hashable** (int, str, tuple, etc.) while **data that can change is not hashable (list, dictionary)**

Mutability

- Object is mutable if it can be changed after it is created
- If it can't, it is immutable
- List and dictionaries are **mutable**
- strings, tuples, and numbers are **immutable**

note: As mentioned before, mutability ~ hashability -> only immutable objects can be used as dictionary keys

Checkpoint

Given that the following strings have been declared:

In [32]:

```
str1 = 'DSC20'
str2 = 'Midterm 1'
```

What do the following expressions result in?

In [ ]:

```
# q1
str1[0]

#q2
str1[4] = "8"

#q3
str1 + str2

#q4
str3 = str1[:3]
str3*3
```

Checkpoint Answers

```
In [33]: str1[0]

Out[33]: 'D'

In [34]: str1[4] = '8'

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[34], line 1
----> 1 str1[4] = '8'

TypeError: 'str' object does not support item assignment

In [35]: str1 + str2

Out[35]: 'DSC20Midterm 1'

In [36]: str3 = str1[:3]
         str3 * 3

Out[36]: 'DSCDSCDSC'
```

Files

- storage for data (think csv's from DSC10, txt's from assignments, etc.)
- unique methods to access within code
- Access modes: write, append, read

Opening Files

In [ ]:

```
# method 1
file_object = open('file_name', 'access_mode')
# do stuff
file_object.close() # required for this method

# method 2
with open('file_name', 'access_mode') as file_object:
    # do stuff
```

note: once file is open, utilize file methods such as .read(), .readline(), .readlines(), .write()

Asserts + Doctests

- Used to evaluate written code
- **asserts** -> input validation (are the arguments the correct types)
- **doctests** -> code validation (does the function give the correct output)

note: asserts can be used in tandem with functions that return booleans (ex. all())

note: doctests are denoted by '>>> ' (space included).

Practice Questions

Question 1

Given a list of strings ***lst*** and a string ***comparer***, return the number of elements in the given list that are equal to ***comparer***. Your solution must be 1 line.
Also write assert statements to check the validity of the input.

In [3]:

```python
def test(lst, comparer):
    '''
    Function that counts the number of equivalent strings in a list.

    Args:
        lst (list): list of strings to be considered
        comparer (str): string to be considered
    Returns:
        The number of elements in 'lst' that is equal to 'comparer'
    Throws:
        AssertionError: if lst is not a list
        AssertionError: if comparer is not a string
        AssertionError: if there are non-strings in comparer

    >>> test(['good', 'luck', 'on', 'mideterm', 'luck', 'luck'], 'luck'
    3
    >>> test([], 'lol')
    0
    '''
    # Write your implementation here
    return
```

Question 1 Solution

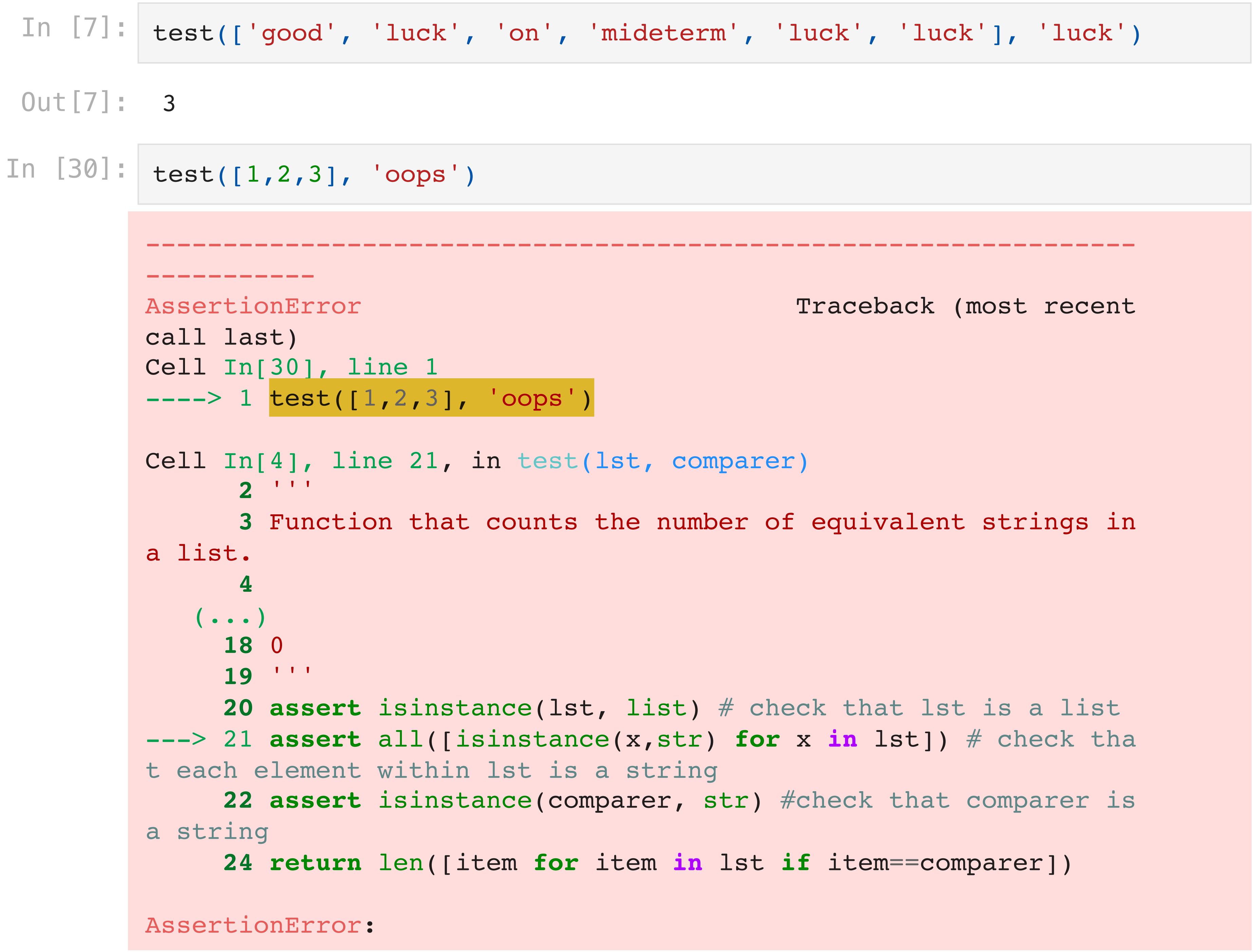
In [4]:

```
def test(lst, comparer):
    '''
    Function that counts the number of equivalent strings in a list.

    Args:
        lst (list): list of strings to be considered
        comparer (str): string to be considered
    Returns:
        The number of elements in 'lst' that is equal to 'comparer'
    Throws:
        AssertionError: if lst is not a list
        AssertionError: if comparer is not a string
        AssertionError: if there are non-strings in comparer

    >>> test(['good', 'luck', 'on', 'mideterm', 'luck', 'luck'], 'luck'
    3
    >>> test([], 'lol')
    0
    '''
    assert isinstance(lst, list) # check that lst is a list
    assert all([isinstance(x,str) for x in lst])
    # check that each element within lst is a string
    assert isinstance(comparer, str) #check that comparer is a string

    return len([item for item in lst if item==comparer])
```



```
In [7]: test(['good', 'luck', 'on', 'mideterm', 'luck', 'luck'], 'luck')

Out[7]: 3

In [30]: test([1,2,3], 'oops')
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
Cell In[30], line 1
----> 1 test([1,2,3], 'oops')

Cell In[4], line 21, in test(lst, comparer)
      2 '''
      3 Function that counts the number of equivalent strings in a list.
      4 
   (...)
     18 0
     19 '''
     20 assert isinstance(lst, list) # check that lst is a list
---> 21 assert all([isinstance(x,str) for x in lst]) # check that each element within lst is a string
     22 assert isinstance(comparer, str) #check that comparer is a string
     24 return len([item for item in lst if item==comparer])

AssertionError: 
```

Question 2

Given a dictionary with integers as keys and lists as values, return a new dictionary where the key and value are flipped. The new key would be the length of the value. If the key already exists, add the value to the pre-existing value.

In [20]:

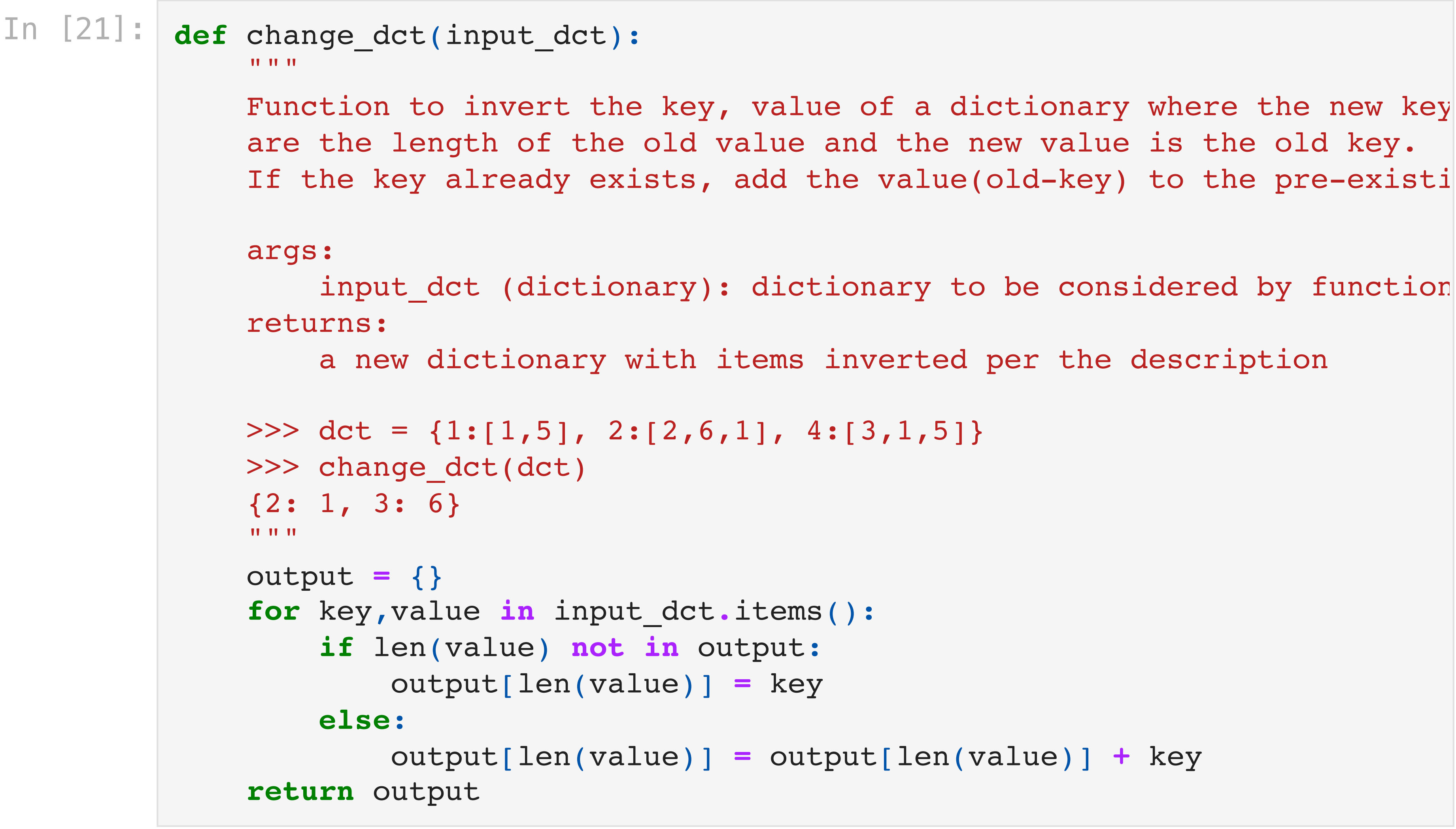
```
def change_dct(input_dct):
    """
    Function to invert the key, value of a dictionary where the new key
    are the length of the old value and the new value is the old key.
    If the key already exists, add the value to the pre-existing value.

    args:
        input_dct (dictionary): dictionary to be considered by function

    returns:
        a new dictionary with items inverted per the description

    >>> dct = {1:[1,5], 2:[2,6,1], 4:[3,1,5]}
    >>> change_dct(dct)
    {2: 1, 3: 6}
    """
    # Write your implementation here
    return
```

Question 2 Solution

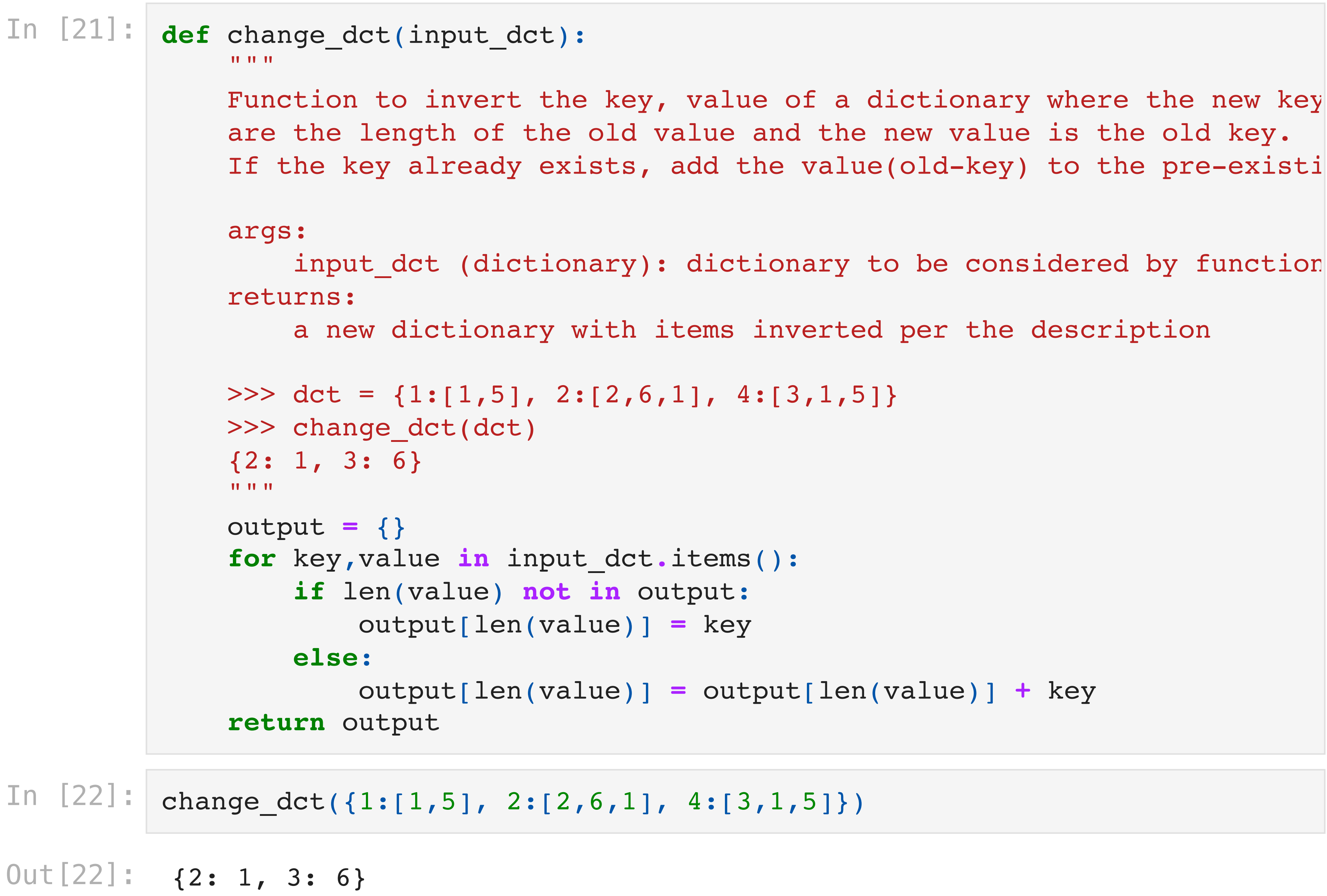


```
In [21]: def change_dct(input_dct):
             """
             Function to invert the key, value of a dictionary where the new key
             are the length of the old value and the new value is the old key.
             If the key already exists, add the value(old-key) to the pre-existi

             args:
                 input_dct (dictionary): dictionary to be considered by function
             returns:
                 a new dictionary with items inverted per the description

             >>> dct = {1:[1,5], 2:[2,6,1], 4:[3,1,5]}
             >>> change_dct(dct)
             {2: 1, 3: 6}
             """
             output = {}
             for key,value in input_dct.items():
                 if len(value) not in output:
                     output[len(value)] = key
                 else:
                     output[len(value)] = output[len(value)] + key
             return output
```

Question 2 Solution



In [21]:

```
def change_dct(input_dct):
    """
    Function to invert the key, value of a dictionary where the new key
    are the length of the old value and the new value is the old key.
    If the key already exists, add the value(old-key) to the pre-existi

    args:
        input_dct (dictionary): dictionary to be considered by function
    returns:
        a new dictionary with items inverted per the description

    >>> dct = {1:[1,5], 2:[2,6,1], 4:[3,1,5]}
    >>> change_dct(dct)
    {2: 1, 3: 6}
    """
    output = {}
    for key,value in input_dct.items():
        if len(value) not in output:
            output[len(value)] = key
        else:
            output[len(value)] = output[len(value)] + key
    return output
```

In [22]:

```
change_dct({1:[1,5], 2:[2,6,1], 4:[3,1,5]})
```

Out[22]:

```
{2: 1, 3: 6}
```

Question 3

Given a nested list that contains positive and negative integers, return a nested list that changes negative integers to positive and multiplies positive numbers by 2. **You are only allowed to use list comprehension**.

Also write assert statements to check the validity of the input.

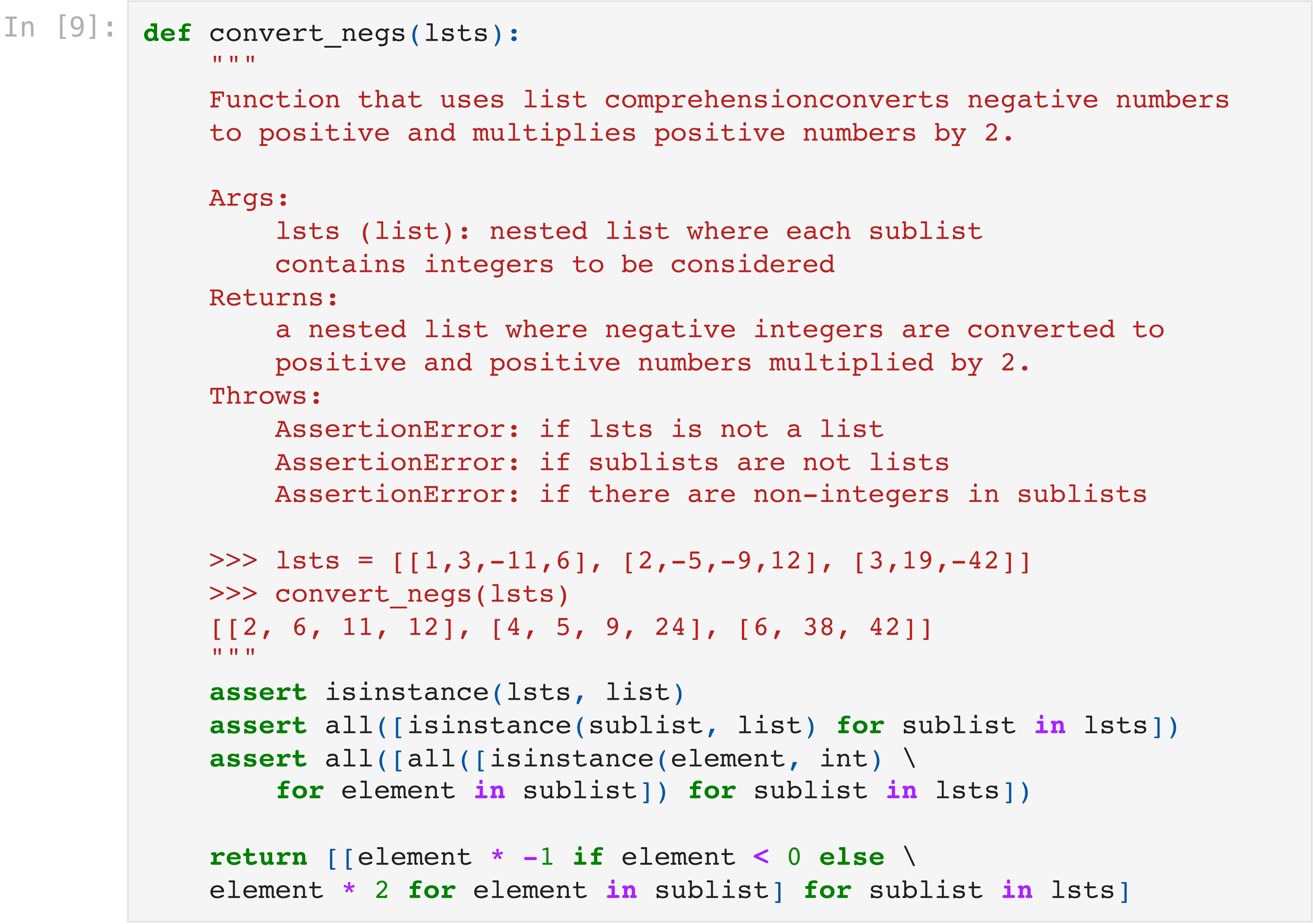
In [23]:

```
def convert_negs(lsts):
    """
    Function that uses list comprehensionconverts negative numbers
    to positive and multiplies positive numbers by 2.

    Args:
        lsts (list): nested list where each sublist contains
        integers to be considered
    Returns:
        a nested list where negative integers are converted to
        positive and positive numbers multiplied by 2.
    Throws:
        AssertionError: if lsts is not a list
        AssertionError: if sublists are not lists
        AssertionError: if there are non-integers in sublists

    >>> lsts = [[1,3,-11,6], [2,-5,-9,12], [3,19,-42]]
    >>> convert_negs(lsts)
    [[2, 6, 11, 12], [4, 5, 9, 24], [6, 38, 42]]
    """
    # Write your implementation here
    return
```

Question 3 Solution



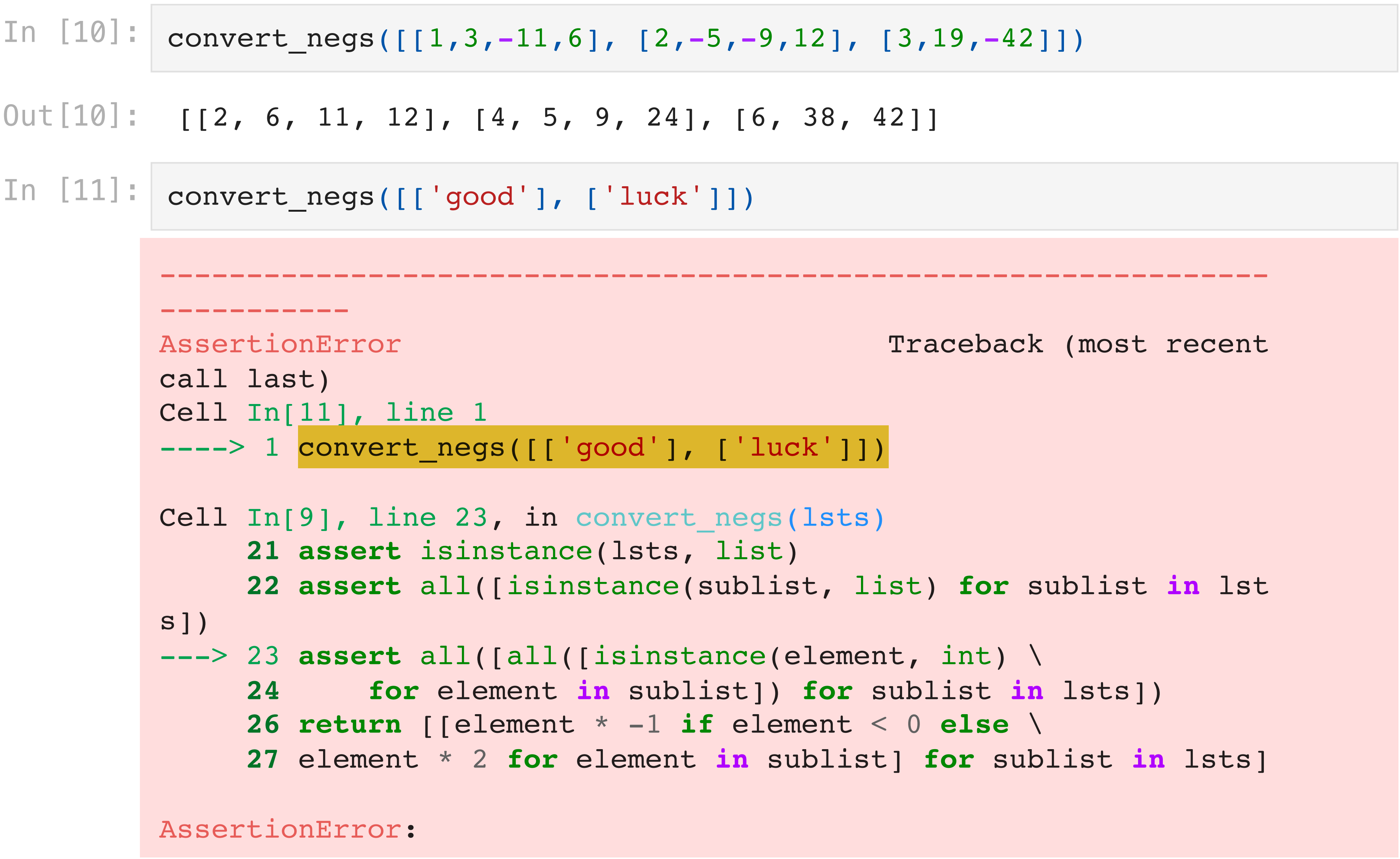
In [9]:

```python
def convert_negs(lsts):
    """
    Function that uses list comprehensionconverts negative numbers
    to positive and multiplies positive numbers by 2.

    Args:
        lsts (list): nested list where each sublist
        contains integers to be considered
    Returns:
        a nested list where negative integers are converted to
        positive and positive numbers multiplied by 2.
    Throws:
        AssertionError: if lsts is not a list
        AssertionError: if sublists are not lists
        AssertionError: if there are non-integers in sublists

    >>> lsts = [[1,3,-11,6], [2,-5,-9,12], [3,19,-42]]
    >>> convert_negs(lsts)
    [[2, 6, 11, 12], [4, 5, 9, 24], [6, 38, 42]]
    """
    assert isinstance(lsts, list)
    assert all([isinstance(sublist, list) for sublist in lsts])
    assert all([all([isinstance(element, int) \
        for element in sublist]) for sublist in lsts])

    return [[element * -1 if element < 0 else \
    element * 2 for element in sublist] for sublist in lsts]
```



```
In [10]: convert_negs([[1,3,-11,6], [2,-5,-9,12], [3,19,-42]])

Out[10]: [[2, 6, 11, 12], [4, 5, 9, 24], [6, 38, 42]]

In [11]: convert_negs([['good'], ['luck']])
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
Cell In[11], line 1
----> 1 convert_negs([['good'], ['luck']])

Cell In[9], line 23, in convert_negs(lsts)
     21 assert isinstance(lsts, list)
     22 assert all([isinstance(sublist, list) for sublist in lsts])
---> 23 assert all([all([isinstance(element, int) \
     24     for element in sublist]) for sublist in lsts])
     26 return [[element * -1 if element < 0 else \
     27 element * 2 for element in sublist] for sublist in lsts]

AssertionError: 
```

Question 4

Given a file containing text that's been "grafitti'd", write a function that rewrites the text in the original form. You may not use .remove() (i.e. the removal must be using string slicing).

In [5]:
```
with open('files/grafitti.txt', 'r') as f: print(f.readlines())
```

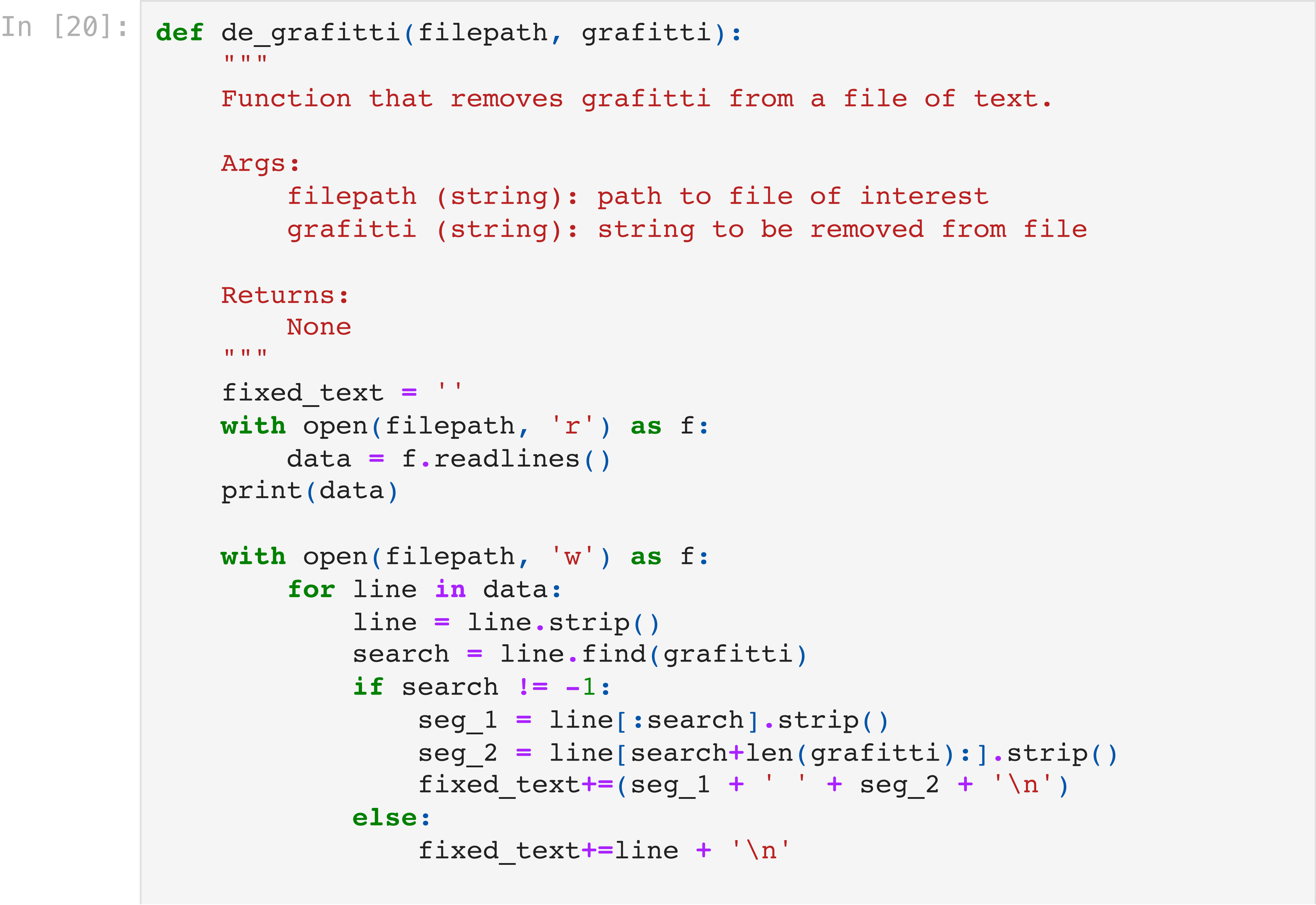
```
['Yar har, fiddle de dee \n', 'Being a stinky pirate is alright
to be\n', "Do what you want 'cause a stinky pirate is free\n",
'You are a stinky pirate!\n', 'Yar!']
```

In [6]:
```
def de_grafitti(filepath, grafitti):
    """
    Function that removes grafitti from a file of text.

    Args:
        filepath (string): path to file of interest
        grafitti (string): string to be removed from file

    Returns:
        None
    """
    # Write your implementation here
```

Question 4 Solution



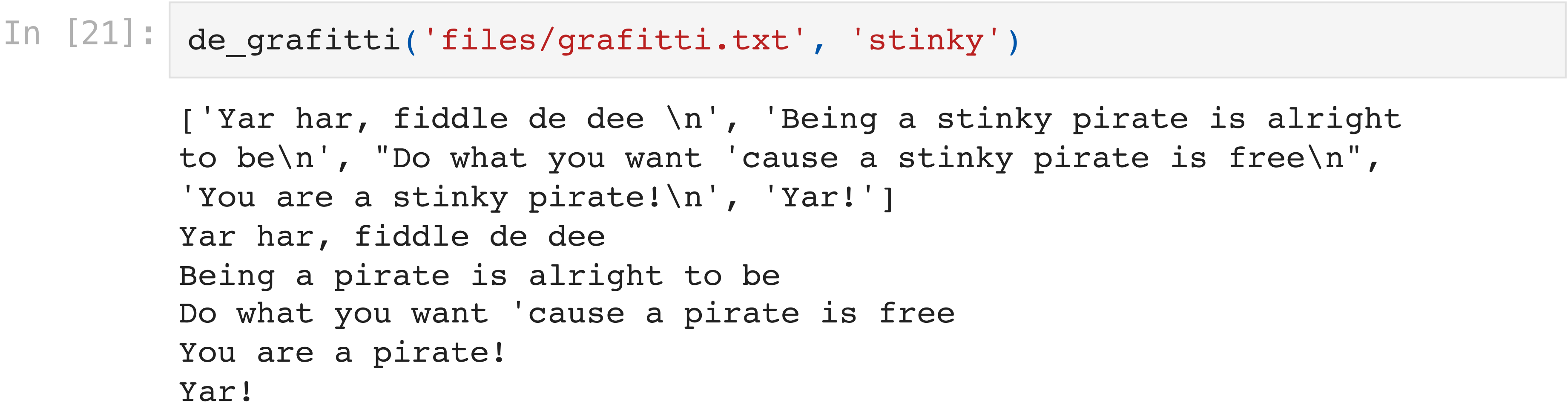
In [20]:

```python
def de_grafitti(filepath, grafitti):
    """
    Function that removes grafitti from a file of text.

    Args:
        filepath (string): path to file of interest
        grafitti (string): string to be removed from file

    Returns:
        None
    """
    fixed_text = ''
    with open(filepath, 'r') as f:
        data = f.readlines()
    print(data)

    with open(filepath, 'w') as f:
        for line in data:
            line = line.strip()
            search = line.find(grafitti)
            if search != -1:
                seg_1 = line[:search].strip()
                seg_2 = line[search+len(grafitti):].strip()
                fixed_text+=(seg_1 + ' ' + seg_2 + '\n')
            else:
                fixed_text+=line + '\n'
        print(fixed_text)
        f.write(fixed_text)
```



```
In [21]: de_grafitti('files/grafitti.txt', 'stinky')
```

```
['Yar har, fiddle de dee \n', 'Being a stinky pirate is alright
to be\n', "Do what you want 'cause a stinky pirate is free\n",
'You are a stinky pirate!\n', 'Yar!']
Yar har, fiddle de dee
Being a pirate is alright to be
Do what you want 'cause a pirate is free
You are a pirate!
Yar!
```

Thanks for Coming!

Good luck on the exam!