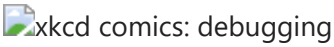
Problem 16: Debug-a-thon

Version 1.0

One skill we hope you have been developing is **debugging.** Errors are bound to happen, but good programmers can spot and rectify them. This notebook is all about pinpointing errors.

In the exercises below, we were careless coders and every program fragment has errors. Your task is to understand each problem, identify the error, and fix them. Good luck!

Exercise 0 (0 points). Here's a cartoon with some encouragement that you are not alone (source: `xkcd`):



Exercise 1 (2 points). Suppose we wish to count how many times different words occur in a string.

The cell below defines a function, `count_words(text)`, which has the following *intended* behavior.

1. The input argument is a string of text named `text`.
2. The input contains a sequence of zero or more words. A word is a sequence of alphabetic characters, and two words will be separated by one or more contiguous whitespace characters.
3. The letters within a word may be uppercase or lowercase.
4. The function should return a dictionary. Each key should be a word from `text` **in all lowercase letters**. The corresponding value is the number of times that word occurred. For example,

```
answer('Amazing amazing IS this amazing beautiful   world')
```
should return

```
{'amazing': 3, 'is': 1, 'this': 1, 'beautiful': 1, 'world': 1}
```
Unfortunately, the implementation of `count_words()` given below is buggy. It sometimes causes errors and may produce the incorrect output. Try to understand what is wrong and rectify the problem by modifying the code accordingly. If it would be helpful, we've provided an extra cell you can use for debugging.

In [ ]:
```
# Modify the function as you deem fit!

def count_words(text):
    list_words =  text.split(' ')
    word_dictionary = {}
    for word in list_words:
```

```
        word = word.lower()
        if word:
            word_dictionary[word] = word_dictionary.get(word, 0) + 1
    return(word_dictionary)
```

In [ ]:
```
## Additional cell for debugging

debug_text = 'Amazing amazing IS this amazing beautiful   world'
count_words(debug_text)
```

Out[ ]:
```
{'amazing': 3, 'is': 1, 'this': 1, 'beautiful': 1, 'world': 1}
```

In [ ]:
```
# Test cell: `exercise_1`

def gen_text():
    from random import randrange, choice, choices
    from collections import defaultdict
    alphabet = ''.join([chr(k) for k in range(ord('a'), ord('z')+1)])
    def gen_word(max_len=10):
        return ''.join(choices(alphabet, k=randrange(1, max_len+1)))
    def gen_vocab(max_len=5):
        return {gen_word(): 0 for _ in range(randrange(1, max_len+1))}
    def randomize_case(w):
        if len(w) == 1:
            return choice(w + w.upper())
        return ''.join([randomize_case(c) for c in w])
    def gen_sep(max_len=5, min_len=1):
        return ' ' * randrange(min_len, max_len+1)
    def gen_sentence(vocab, max_len=20):
        sentence = gen_sep(min_len=0)
        for i in range(randrange(0, max_len)):
            word_i = choice(list(vocab.keys()))
            vocab[word_i] += 1
            sentence += randomize_case(word_i) + gen_sep(randrange(1, 5))
        if randrange(0, 2):
            sentence.rstrip()
        return sentence
    def remove_zero_counts(vocab):
        return {w: c for w, c in vocab.items() if c > 0}

    vocab = gen_vocab()
    sentence = gen_sentence(vocab)
    vocab_soln = remove_zero_counts(vocab)
    return sentence, vocab_soln

for _ in range(10):
    sentence, answer = gen_text()
    your_answer = count_words(sentence)
    assert your_answer == answer, f"Your solution did not work on the following inp

assert len(count_words('')) == 0
assert len(count_words('     ')) == 0

print("\n(Passed!)")
```

```
(Passed!)
```

Exercise 2 (3 points: 1 point for an "exposed" test, 2 points for a hidden test).

Consider a *modified* Fibonacci sequence, which is defined by the recurrence,

$$t_i = \begin{cases} a & i = 1 \\ b & i = 2 \\ t_{i-2} + (t_{i-1})^2 & i > 2 \end{cases},$$

where a and b are given constants.

Suppose you want to write a function, `generate_numbers(a, b)`, that calculates the first seven (7) values of this sequence and returns them in order as a list, i.e., $[t_1 = a, t_2 = b, t_3, \ldots, t_7]$. For example, if `a=0` and `b=1`, then you want

```
generate_numbers(0, 1) == [0, 1, 1, 2, 5, 27, 734]
```

Your friend has tried to implement `generate_numbers()` in the code cell below. However, it's not working. Your task in this exercise is to correct it.

In particular, your corrected `generate_numbers(a, b)` should take as input:

1. `a` = the value of the first number in the sequence
2. `b` = the value of the second number in the sequence

And it should return:

1. A **list** of the first 7 numbers of that specific sequence. If `a` and `b` happen to be floating-point values, you should **round** each element of the returned list to 3 decimal places.

Note: Depending on the starting values `a` and `b`, it's possible some computed t_i may exceed the numerical range of floating-point values and Python might produce an overflow error. You do **NOT** need to handle that possibility in this function.

In [ ]:
```
# You can modify any or all of the code in this cell,
# but to pass, you must define a function
# generate_numbers(a, b) that meets the specifications.

length = 7  # these many modified Fibonacci numbers.
def fibonacci_gt(n, t1, t2):
    if n<0:
        print("Incorrect input")
    elif n == 1:
        return t1
    elif n == 2:
        return t2
    else:
        return fibonacci_gt(n-1,t1,t2)**2 + fibonacci_gt(n-2,t1,t2)

def generate_numbers(a, b):
```

```
    numbers = []
    for i in range(1, 8):
        numbers.append(fibonacci_gt(i,a,b))
    numbers = [round(ele, 3) for ele in numbers]
    print(type(numbers))
    return numbers

generate_numbers(0, 1)
print(generate_numbers(0.3, 0.7))
```

```
<class 'list'>
<class 'list'>
[0.3, 0.7, 0.79, 1.324, 2.543, 7.792, 63.261]
```

In [ ]:
```
# Exposed test cell: `exercise_2`
# assert type(generate_numbers(0, 1)) is list, 'please check the type of your outpu

# assert generate_numbers(0, 1) == [0, 1, 1, 2, 5, 27, 734]
# assert generate_numbers(0.3, 0.7) == [0.3, 0.7, 0.79, 1.324, 2.543, 7.792, 63.261
# assert generate_numbers(5, 1) == [5, 1, 6, 37, 1375, 1890662, 3574602799619]
# assert generate_numbers(1, 2) == [1, 2, 5, 27, 734, 538783, 290287121823]

# print("\n(Passed!)")
```

In [ ]:
```
# Hidden test cell: exercise_2_hidden

print("""
In addition to the tests above, this cell will include some hidden tests.
You will only know the result when you submit your solution to the
autograder.
""")

###
### AUTOGRADER TEST - DO NOT REMOVE
###
```

```
In addition to the tests above, this cell will include some hidden tests.
You will only know the result when you submit your solution to the
autograder.
```

Exercise 3 (2 points): The Smart Trader.

One of the CSE-6040 TAs wants to analyze what would have been the best time to buy and sell a bitcoin, given a historical list of daily prices. (Prices are positive integers and you can ignore the units.)

For example, suppose the daily prices on five consecutive days were `[10, 7, 2, 5, 1]`, and buying or selling must follow these rules:

1. Over this time period, only one "buy" and one "sell" transaction can occur.
2. If a person buys on day i, then she can only sell on a day $j \geq i$.
3. The profit is the selling price on day i minus the buying price on day j.

In this example, by buying on day $i = 2$, when the price was 2 units, and selling on day $j = 3$, when the prices was 5 units, one could have earned a profit of 5-2 = 3 units.

The TA has sketched a program to calculate the maximum possible profit. Her solution appears in the function `make_money(prices)`, defined in the code cell below. It takes a list of prices (`prices`) over several days and returns the maximum profit. It is correct **except** there are some missing values, or "holes," shown as **?**'s. Your task is to determine what goes in these holes.

You can use the debugging cell provided for your own analysis.

In [ ]:
```
# You can modify the function as you deem fit!
from itertools import accumulate
def make_money(prices):
    min_price = accumulate(prices, min)
    gains = [p-m for p,m in zip(prices, min_price)]
    max_gain = max(gains)
    return max_gain
```

In [ ]:
```
## Debugging cell
print(make_money([10, 7, 2, 5, 1]))
# assert make_money([10, 7, 2, 5, 1]) == 3, 'Check your solution'
```

```
3
```

In [ ]:
```
# Test cell: `exercise_3`

assert make_money([7,1,5,3,6,4]) == 5, 'Check your solution'
print('passed - test 1')

assert make_money([10,8,5,3,6,4]) == 3, 'Check your solution'
print('passed - test 2')

assert make_money([10,8,5,9,6,4]) == 4, 'Check your solution'
print('passed - test 3')

assert make_money([1,8,9,14,14]) == 13, 'Check your solution'
print('passed - test 4')

assert make_money([11,8,5,2,1]) == 0, 'Check your solution'
print('passed - test 5')

def gen_prices(max_price=100, max_run=10):
    from random import randrange, random
    def rand_run(b, s, n):
        return [randrange(b, s+1) for _ in range(randrange(n))]
    buy_price = randrange(max_price)
    sell_price = randrange(buy_price, max_price)
    if buy_price != sell_price and random() < 0.1:
        sell_price = buy_price
    X = rand_run(buy_price, sell_price, max_run)
    Y = rand_run(buy_price, sell_price, max_run)
    Z = rand_run(buy_price, sell_price, max_run)
    prices = X + [buy_price] + Y
```

```
    buy_day = len(X)
    if buy_price != sell_price or random() > 0.5:
        sell_day = len(prices)
        prices += [sell_price]
    else:
        sell_day = buy_day
    prices += Z
    return prices, sell_price - buy_price, (buy_day, buy_price), (sell_day, sell_pr

def test_max_profit_rand(verbose=False):
    prices, max_profit_true, (buy_day, buy_price), (sell_day, sell_price) = gen_pri
    if verbose: print(f"Buy at {buy_price} on day {buy_day}, sell at {sell_price} o
    your_max_profit = make_money(prices)
    assert your_max_profit == max_profit_true, \
           f"Your solution didn't work on the following prices:\n==> {prices}\nThe

print("Running a battery of longer tests...")
for _ in range(20):
    test_max_profit_rand()

print("\n(Passed!)")
```

```
passed - test 1
passed - test 2
passed - test 3
passed - test 4
passed - test 5
Running a battery of longer tests...

(Passed!)
```

Exercise 4 (3 points) Modified Tic-Tac-Toe Board Initialisation.

We are building a complex Tic-Tac-Toe bot and we need a function to initialise a given board of size (`size * size`) as a 2-d list with blanks (`''`) except for the location (defined by `position` which is a tuple `(row_num,column_num)`)).

Your task is to find what is causing the below function to fail and rectify the bug.

eg : `initialise_board(3,(1,1))` should output the following -

`[['', '', ''], ['', 'X', ''], ['', '', '']]`

Note 1: You have been given a snippet of code that has errors. You have to identify what is going wrong and rectify the errors in the same function.

In [ ]:
```
# You can modify the function as you deem fit!

def initialise_board(size, position):
    # Create a new list for each row to ensure they are independent
    board = [["" for _ in range(size)] for _ in range(size)]

    # Unpack the position tuple directly into row and column indices
    row_index, col_index = position
```

```
    # Place an "X" at the specified position
    board[row_index][col_index] = "X"

    return board

initialise_board(3,(1,1))
```

Out[ ]: `[['', '', ''], ['', 'X', ''], ['', '', '']]`

In [ ]:
```
## Feel free to use this cell for debugging
```

In [ ]:
```
# Test cell: `exercise_4`

assert initialise_board(4,(1,1)) == [['', '', '', ''], ['', 'X', '', ''], ['', '',
assert initialise_board(4,(1,2)) == [['', '', '', ''], ['', '', 'X', ''], ['', '',
assert initialise_board(3,(2,2)) == [['', '', ''], ['', '', ''], ['', '', 'X']]
assert initialise_board(6,(2,4)) == [['', '', '', '', '', ''],['', '', '', '', '',

def rand_test_board(max_n=10):
    from random import randrange
    n = randrange(1, max_n)
    xi, xj = randrange(n), randrange(n)
    print(f"Testing {n}x{n} board with an initial mark at position {(xi, xj)}...")
    your_board = initialise_board(n, (xi, xj))
    for i in range(n):
        for j in range(n):
            true_mark = 'X' if (xi, xj) == (i, j) else ''
            your_mark = your_board[i][j]
            assert your_mark == true_mark, f"Position {(i, j)} should have a '{true

for _ in range(10):
    rand_test_board()

print("\n(Passed!)")
```

```
Testing 9x9 board with an initial mark at position (3, 4)...
Testing 9x9 board with an initial mark at position (2, 4)...
Testing 7x7 board with an initial mark at position (2, 4)...
Testing 5x5 board with an initial mark at position (2, 3)...
Testing 7x7 board with an initial mark at position (4, 0)...
Testing 7x7 board with an initial mark at position (6, 0)...
Testing 1x1 board with an initial mark at position (0, 0)...
Testing 4x4 board with an initial mark at position (3, 3)...
Testing 6x6 board with an initial mark at position (5, 0)...
Testing 7x7 board with an initial mark at position (4, 1)...

(Passed!)
```

Fin! You've reached the end of this part. Don't forget to restart and run all cells again to make sure it's all working when run in sequence; and make sure your work passes the submission process. Good luck!