# Homework 04

Re-submit Assignment

**Due** Feb 19 by 10pm **Points** 100 **Submitting** a file upload

# Requirements of Homework

This five part assignment offers practice iterating over lists, ranges, and strings using for and while loops as well as using a generator of random integers. It's also a good exercise in writing unittest test cases.

## Part 1: `count_vowels(seq)`

Write a function `count_vowels(s)` that takes a string as an argument and returns the number of vowels `('a', 'e', 'i' 'o', 'u')` in the string. Should you use a ***for*** or ***while*** loop? (**Implement this as a function, not as a class with a method**.)

Be sure to include unittest test cases to demonstrate that your code works properly, e.g.

```
def count_vowels(s: str) -> int:
    return 0  # Your definition goes here

class CountVowelsTest(unitttest.TestCase):
    def test_count_vowels(self) -> None:
        self.assertEqual(count_vowels('hello world'), 3)

Hint: Python offers an 'in' operator that evaluates to True or False, e.g.
```

```
"a" in ['a', 'e', 'i', 'o', 'u']
```

evaluates to `True`.

**Hint**: Strings can easily be converted to all lower case with the *string.lower()* method, e.g. "HeLlO wOrLd".lower() == "hello world"

## Part 2: `last_occurrence(target, sequence)`

Write a function `last_occurrence(target: Any, sequence: Sequence[Any]) -> Optional[int]` that takes two arguments:

1. `target`: A target item to find
2. `sequence`: A sequence of values, e.g. a list is a sequence as is str.

Your function should return the **index** (offset from 0) of the ***last*** occurrence of the target item or `None` if the target is not found. E.g. the last occurrence of 33 is at offset 3 in the list `[ 42, 33, 21, 33 ]` because 42 is offset 0, the first 33 is at offset 1, 21 is offset 2, and the last 33 is offset 3.

**Hints:**

- How do we specify the type hint for a function that returns either an integer or None?

```
from typing import Optional

def foo() -> Optional[int]:  # return an int or None
    return None
```

- How can we specify the type hint for a variable/parameter/return type of an arbitrary sequence that matches list or str or other sequences?

```
from typing import Sequence, Any

my_seq: Sequence[Any] # my_seq is a sequence of values of Any type
```

Use unittest to test your function with several different target values and lists.

Next, test your function with a character as the target, and a string as the list, e.g. `find('p', 'apple')`. What should happen?

Be sure to use unittest to demonstrate that your code works properly.

# Part 3: `Fraction.simplify()`

The fractions in Homework03 are correct, but not simplified, e.g. 2/4 can be simplified to 1/2, 14/21 can be simplified to 2/3, and 63/9 can be simplified to 7/1. Recall from elementary school that you can simplify fractions by finding the Greatest Common Factor (GCF) and then dividing the number and denominator. Here's pseudocode for one solution:

```
start = the min of abs(numerator) and abs(denominator)
          # the absolute value is important for negative values

for each integer gcf from start down to 2
    if numerator mod gcf == 0 and denominator mod gcf == 0 then
        gcf is the greatest common factor that evenly divides both the
                numerator and denominator

        return a new Fraction with numerator / gcf and denominator / gcf

 if you don't find a gcf, then return a copy of the Fraction because it can't be simplified
```

Extend your Fractions with a new method, `Fraction.simplify()` class from HW03 and add a new `simplify(self)` method that returns a new Fraction that is simplified or just returns a copy of self if self can't be simplified. E.g.

```
str(Fraction(9, 27).simplify()) == str(Fraction(1, 3))
```

**Hint**: Note that testing

```
Fraction(9, 27).simplify() == Fraction(1, 3)
```

is not sufficient because

```
Fraction(9, 27) == Fraction(1, 3)
```

Add unittest cases to test your new method.

**Hint**: what happens if the fraction has a negative numerator or denominator?

**Hint**: Your `Fraction.simplify()` method should **NOT** modify self, but should return a new instance of class Fraction with the appropriate numerator and denominator.

**NOTE**: You **MAY NOT** use Python's gcd() function. Implement your own.

# Part 4: `my_enumerate(seq)`

Recall that Python's built-in `enumerate(seq)` function is a generator that returns two values on each call to `next()`: the offset of the value and the value. E.g.

```
for offset, value in enumerate("hi!"):
    print(offset, value)
```

generates the output:

```
0 h
1 i
2 !
```

Write a generator, my_enumerate(seq: Sequence[Any]) -> Iterator[Any] that provides the same functionality **WITHOUT** calling the built-in `enumerate()`. Be sure to include an automated test to validate your solution.

**Hint**: Generators *yield* results while functions *return* results.

**Hint**: Your automated test may compare `list(my_enumerate(your_sequence))` to the expected output. E.g.

```
list(my_enumerate(your_sequence)) == list(enumerate(your_sequence))
```

> **Note from Benji**: I used 2 lines of code (not including docstring) to implement the function - I got a one-liner solution but two-line solution is actually more readable ;-)

**Hints**:

The type hint for generators is

```
   from typing import Iterator

    def my_generator() -> Iterator[int]:  # my_generator is a generator that yields a sequence of in
t values
```

The type hint for arbitrary sequences is

```
    from typing import Sequence, List

    any_seq_any_values: Sequence[Any]  # any sequence, e.g. list, str, tuple, etc of any values
    any_seq_int: Sequence[int] # any sequence, e.g. list, str, tuple, etc of ints
    list_of_ints : List[int]   # a list of int values
    list_of_strings : List[str] # a list of str values
```

# Part 5: (OPTIONAL - NOT REQUIRED but interesting challenge)

(a) `random_integer_generator(minimum: int, maximum: int) -> Iterator[int]`

Write a generator `random_integer_generator(minimum: int, maximum: int) -> Iterator[int]` that returns a potentially infinite sequence of random integers between a min and max value. E.g. say that `min = 0` and `max = 10`, then the generator returns a sequence of random integers between 0 and 10 inclusive, one on each call to the generator function.

**Hint**: The following code generates a random number between 0 and 10 inclusive.

```
import random

r = random.randint(0, 10) # return a random integer between 0 and 10 inclusive.
```

(b) find_target(target: int, min_val: int, max_val: int, max_attempts: int) -> Optional[int]

Use the `random_integer_generator` from (a) in a function, `find_target(target, min_value, max_value, max_attempts)` where `find_target()` passes `min_value` and `max_value` to the random integer generator and then loops, reading random values from the random integer generator until the specified target is

found, then returns how many random integers were read before finding the target or `None` if the target isn't found in `max_attempts` tries.

E.g. consider the call `find_target(3, 0, 10, 100)` where we're asking how many random integers between 0 and 10 are generated when we find the first occurrence of 3. Say the random integer generator generated the sequence [9, 4, 1, 3, 2, …] . The generator returns a single value at a time and `find_target()` continues asking for more values until the target, 3, is found and then returns 4 because 4 random integers were generated when the target was found.

Note that the random generator may randomly choose values that never match the target value. The `max_attempts` parameter to `find_target()` is an escape hatch to force `find_target()` to return if the target value is not found to avoid a potential infinite loop.

**Note**: while you should pass `min_value` and `max_value` from `find_target()` when creating the generator, **DO NOT** include the `max_attempts` logic in the generator. The generator should return a potentially infinite number of random values.

One of the challenges of this assignment is how to test `find_target()` automatically since it relies on a random number generator. One technique is to create a generator that can generate only a single random value which then must be returned on the first call to the generator. E.g. `find_target(3, 3, 3, 1)` must return 1 because the random generator must generate a random number n, where 3 <= n <= 3, i.e. n == 3. Since the random number generator must return a sequence of the value 3, then `find_target(3, 3, 3, 1)` will find the target value of 3 on the first attempt.

You should also raise `ValueError` exceptions if there are any problems with the relationship between `target`, `min_value`, and `max_value`, e.g. min_value <= target <= max_value

You might find that writing your code, and then stepping through your solution with the debugger and variable explorer in VS Code may help you to understand what your program is doing correctly (or not).

As always, please email questions to me.

# Deliverable

# File Structure

In this assignment, you will need to **separate your code into two files** - one for code logic and one for unit test. It's always a good practice to separate the code and the test.

You are going to have **four .py files** for submission:

1. `HW04_FirstName_LastName.py`
2. `HW04_Test_FirstName_LastName.py`
3. `HW03_FirstName_LastName.py - your HW03 solution with the new Fraction.simplify() method`

4. `HW03_Test_FirstName_LastName.py - your HW03 test solution with tests for the new Fraction.simplify() method`

In `HW04_FirstName_LastName.py` you should have:

1. `count_vowels()` - function
2. `last_occurrence(target, sequence)` - function
3. `my_enumerate(seq)` - function
4. Optional:
    1. `random_integer_generator(minimum, maximum)`
    2. `find_target(target, min_value, max_value, max_attempts)`

In `HW04_Test_FirstName_LastName.py` you will have three or four test classes:

1. `CountVowelsTest`
2. `FindLastTest`
3. `EnumerateTest`
4. `FindTargetTest (Optional)`

You'll find optional framework files for **HW04.py** and **HW04_Test.py** in Canvas.

If your have any concern, please do not hesitate to reach out.