# Homework 1 Solutions **hw01.zip (hw01.zip)**

## Solution Files

You can find the solutions in hw01.py (hw01.py).

# Required Questions

## Q1: Welcome Forms

Please fill out both the Syllabus Quiz (https://go.cs61a.org/syllabus-quiz), which is based off of our policies found on the course syllabus (https://cs61a.org/articles/about/), as well as the optional Welcome Survey (https://go.cs61a.org/welcome-survey).

## Q2: A Plus Abs B

Fill in the blanks in the following function for adding `a` to the absolute value of `b`, without calling `abs`. You may **not** modify any of the provided code other than the two blanks.

```
def a_plus_abs_b(a, b):
    """Return a+abs(b), but without calling abs.

    >>> a_plus_abs_b(2, 3)
    5
    >>> a_plus_abs_b(2, -3)
    5
    """
    if b < 0:
        f = sub
    else:
        f = add
    return f(a, b)
```

Use Ok to test your code:

```
python3 ok -q a_plus_abs_b
```

If `b` is positive, we add the numbers together. If `b` is negative, we subtract the numbers. Therefore, we choose the operator `add` or `sub` based on the sign of `b`.

## Q3: Two of Three

Write a function that takes three *positive* numbers as arguments and returns the sum of the squares of the two smallest numbers. **Use only a single line for the body of the function.**

```
def two_of_three(x, y, z):
    """Return a*a + b*b, where a and b are the two smallest members of the
    positive numbers x, y, and z.

    >>> two_of_three(1, 2, 3)
    5
    >>> two_of_three(5, 3, 1)
    10
    >>> two_of_three(10, 2, 8)
    68
    >>> two_of_three(5, 5, 5)
    50
    """
    return min(x*x+y*y, x*x+z*z, y*y+z*z)
    # Alternate solution
def two_of_three_alternate(x, y, z):
    return x**2 + y**2 + z**2 - max(x, y, z)**2
```

**Hint:** Consider using the `max` or `min` function:

```
>>> max(1, 2, 3)
3
>>> min(-1, -2, -3)
-3
```

Use Ok to test your code:

```
python3 ok -q two_of_three
```

We use the fact that if `x>y` and `y>0`, then `square(x)>square(y)`. So, we can take the `min` of the sum of squares of all pairs. The `min` function can take an arbitrary number of arguments.

Alternatively, we can do the sum of squares of all the numbers. Then we pick the largest value, and subtract the square of that.

## Q4: Largest Factor

Write a function that takes an integer `n` that is **greater than 1** and returns the largest integer that is smaller than `n` and evenly divides `n`.

```
def largest_factor(n):
    """Return the largest factor of n that is smaller than n.

    >>> largest_factor(15) # factors are 1, 3, 5
    5
    >>> largest_factor(80) # factors are 1, 2, 4, 5, 8, 10, 16, 20, 40
    40
    >>> largest_factor(13) # factor is 1 since 13 is prime
    1
    """
    factor = n - 1
    while factor > 0:
        if n % factor == 0:
            return factor
        factor -= 1
```

> **Hint:** To check if `b` evenly divides `a`, you can use the expression `a % b == 0`, which can be read as, "the remainder of dividing `a` by `b` is 0."

Use Ok to test your code:

```
python3 ok -q largest_factor
```

Iterating from `n-1` to 1, we return the first integer that evenly divides `n`. This is guaranteed to be the largest factor of `n`.

## Q5: If Function Refactor

Here are two functions that have a similar structure. In both, `if` prevents a `ZeroDivisionError` when `x` is 0.

```
def invert(x, limit):
    """Return 1/x, but with a limit.

    >>> x = 0.2
    >>> 1/x
    5.0
    >>> invert(x, 100)
    5.0
    >>> invert(x, 2)    # 2 is smaller than 5
    2

    >>> x = 0
    >>> invert(x, 100)  # No error, even though 1/x divides by 0!
    100
    """
    if x != 0:
        return min(1/x, limit)
    else:
        return limit

def change(x, y, limit):
    """Return abs(y - x) as a fraction of x, but with a limit.

    >>> x, y = 2, 5
    >>> abs(y - x) / x
    1.5
    >>> change(x, y, 100)
    1.5
    >>> change(x, y, 1)    # 1 is smaller than 1.5
    1

    >>> x = 0
    >>> change(x, y, 100)  # No error, even though abs(y - x) / x divides by 0!
    100
    """
    if x != 0:
        return min(abs(y - x) / x, limit)
    else:
        return limit
```

To "refactor" a program means to rewrite it so that it has the same behavior but with some change to the design. Below is an attempt to refactor both functions to have short one-line definitions by defining a new function `limited` that contains their common structure.

```
def limited(x, z, limit):
    """Logic that is common to invert and change."""
    if x != 0:
        return min(z / x, limit)
    else:
        return limit

def invert_short(x, limit):
    """Return 1/x, but with a limit.

    >>> x = 0.2
    >>> 1/x
    5.0
    >>> invert_short(x, 100)
    5.0
    >>> invert_short(x, 2)    # 2 is smaller than 5
    2

    >>> x = 0
    >>> invert_short(x, 100)  # No error, even though 1/x divides by 0!
    100
    """
    return limited(x, 1, limit)

def change_short(x, y, limit):
    """Return abs(y - x) as a fraction of x, but with a limit.

    >>> x, y = 2, 5
    >>> abs(y - x) / x
    1.5
    >>> change_short(x, y, 100)
    1.5
    >>> change_short(x, y, 1)    # 1 is smaller than 1.5
    1

    >>> x = 0
    >>> change_short(x, y, 100)  # No error, even though abs(y - x) / x divides by 0!
    100
    """
    return limited(x, abs(y - x), limit)
```

There's a problem with this refactored code! Try `invert_short(0, 100)` and see. It causes a `ZeroDivisionError` while `invert(0, 100)` did not.

Your first job is to understand why the behavior changed. In `invert`, division by `x` only happens when `x` is not 0, but in `invert_short` it always happens. Read the rules of evaluation for `if` statements (http://composingprograms.com/pages/15-control.html#conditional-statements) and call expressions (http://composingprograms.com/pages/12-elements-of-programming.html#call-expressions) to see why.

Your second job is to edit `invert_short` and `change_short` so that they have the same behavior as `invert` and `change` but still have just one line each. You will also need to edit `limited`. You don't need to use `and` or `or` or `if` in `invert`; just pay attention to when the division takes place.

Use Ok to test your code:

```
python3 ok -q invert_short
python3 ok -q change_short
```

In the initial `invert_short` and `change_short`, we will first evaluate the arguments to the `limited` function call, which gives us a `ZeroDivisionError` before the call to `limited` was made.

Thus, we want to postpone the evaluation of the division so that it occurs inside the `if` statement in `limited`, observing that dividing by `x` is another common piece of logic between `invert_short` and `change_short`.

We can now update `invert_short` and `change_short` so they only pass in the numerator of the expression, as the denominator is the common `x` expression and now abstracted into `limited` instead. No more `ZeroDivisionError`!

## Q6: Hailstone

Douglas Hofstadter's Pulitzer-prize-winning book, *Gödel, Escher, Bach*, poses the following mathematical puzzle.

1. Pick a positive integer `n` as the start.
2. If `n` is even, divide it by 2.
3. If `n` is odd, multiply it by 3 and add 1.
4. Continue this process until `n` is 1.

The number `n` will travel up and down but eventually end at 1 (at least for all numbers that have ever been tried -- nobody has ever proved that the sequence will terminate). Analogously, a hailstone travels up and down in the atmosphere before eventually landing on earth.

This sequence of values of `n` is often called a Hailstone sequence. Write a function that takes a single argument with formal parameter name `n`, prints out the hailstone sequence starting at `n`, and returns the number of steps in the sequence:

```
def hailstone(n):
    """Print the hailstone sequence starting at n and return its
    length.

    >>> a = hailstone(10)
    10
    5
    16
    8
    4
    2
    1
    >>> a
    7
    """
    length = 1
    while n != 1:
        print(n)
        if n % 2 == 0:
            n = n // 2      # Integer division prevents "1.0" output
        else:
            n = 3 * n + 1
        length = length + 1
    print(n)                # n is now 1
    return length
```

Hailstone sequences can get quite long! Try 27. What's the longest you can find?

Use Ok to test your code:

```
python3 ok -q hailstone
```

**Curious about hailstones or hailstone sequences? Take a look at these articles:**

- Check out this article (https://www.nationalgeographic.org/encyclopedia/hail/) to learn more about how hailstones work!
- In 2019, there was a major development (https://www.quantamagazine.org/mathematician-terence-tao-and-the-collatz-conjecture-20191211/) in understanding how the hailstone conjecture works for most numbers!

We keep track of the current length of the hailstone sequence and the current value of the hailstone sequence. From there, we loop until we hit the end of the sequence, updating the length in each step.

> Note: we need to do floor division `//` to remove decimals.

# Just for fun Question

> This question is out of scope for 61A. You can try it if you want an extra challenge, but it's just a puzzle that has no practical value and is not required or recommended at all. Almost all students will skip it, and that's fine.

## Q7: Quine

Write a one-line program that prints itself, using only the following features of the Python language:

- Number literals
- Assignment statements
- String literals that can be expressed using single or double quotes
- The arithmetic operators `+`, `-`, `*`, and `/`
- The built-in `print` function
- The built-in `eval` function, which evaluates a string as a Python expression
- The built-in `repr` function, which returns an expression that evaluates to its argument

You can concatenate two strings by adding them together with `+` and repeat a string by multipying it by an integer. Semicolons can be used to separate multiple statements on the same line. E.g.,

```
>>> c='c';print('a');print('b' + c * 2)
a
bcc
```

> **Hint**: Explore the relationship between single quotes, double quotes, and the `repr` function applied to strings.

A program that prints itself is called a Quine. Place your solution in the multi-line string named `quine`.

Use Ok to test your code:

```
python3 ok -q quine_test
```