Warmup (L6)

- Write a function `validTrangle` with arguments `(a, b, c)`, for the side lengths of a triangle, that returns `True` if the three side lengths can form a triangle, or `False` otherwise.
    - Triangle reminder: The length of every side of a triangle must be strictly less than the sum of the lengths of the other two sides.

CS114

Module 3: Loops

Repetition

CS114 M3

Repetition

- We're nearly doing real computing:
  - computation = ~~calculation~~ + **repetition** + ~~decision-making~~
- Repetition of calculations is necessary for computation!

Repetition

- Why did we do decision-making first?
- An old joke: A computer programmer buys a new bottle of shampoo. After several hours in the shower, his wife asks what's wrong. "The bottle says lather, rinse, repeat!"
- You have to be able to decide when to stop repeating, so decision-making comes first!

Repetition for laziness

- We will eventually need repetition to do interesting computing
- Let's start simpler: if we want to `print` a countdown from, say, 100, it sure would be annoying to write 100 `print`s!

The countdown

```
c = 100
while c >= 1:
    print(c)
    c = c - 1
print("Ignition")
```

The countdown

while: Like **if**, but instead of “do this if this condition is true”, “do this while this condition is true”. I.e., do an if, then when you’re done, come back and check again, repeatedly, until the condition is false.

```
while c >= 1:
    print(c)
    c = c - 1
print("Ignition")
```

The countdown

Remember that >= is ≥

```
c = 100
while c >= 1:
    print(c)
    c = c - 1
print("Ignition")
```

The countdown

This happened before the repetition, so `c` was 100 the first time we printed

```
c = 100
while c >= 1:
    print(c)
    c = c - 1
print("Ignition")
```

The countdown

We print every time we repeat

```
c = 100
while c >= 1:
    print(c)
    c = c - 1
print("Ignition")
```

The countdown

We reduce `c` by 1 every time we repeat, so the next time we'll print a lower number

```
c = 100
while c >= 1:
    print(c)
    c = c - 1
print("Ignition")
```

The countdown

This is unindented, so it's after (outside) the repetition

```
c = 100
while c >= 1:
    print(c)
    c = c - 1
print("Ignition")
```

The countdown

If we printed `c` here, we'd see that it's now 0

```
c = 100
while c >= 1:
    print(c)
    c = c - 1
print("Ignition at", c)
```

The countdown

Once `c` was 0, this condition stopped being true, so we didn't repeat this again

```
c = 100
while c >= 1:
    print(c)
    c = c - 1
print("Ignition")
```

The `while` loop

- New syntax: this is a *while loop*
- It repeats (loops) while a condition is true
- The most basic and powerful form of loop: we'll see other forms, but we can mimic any of them with a **`while`** loop
- Syntax is identical to **`if`**, except
  - **`while`** instead of **`if`**, and
  - there's no equivalent of **`elif`** or **`else`**

Infinite loops!

- A while loop will continue looping so long as its condition is true
- It's up to *you* to make sure its condition eventually becomes false!
- Infinite loops are usually undesirable…
  - … but they're often necessary to make something interactive. We won't be doing that much in this course ☹

Keeping track of state

- Imperative language: Commands are run in the given order
- Add loops: Commands are run repeatedly in the given order
- Consequence: We can no longer simply say “at this location in code, c has this value” (the location is repeated, and c has different values!)

Keeping track of state

- This is another dimension where surprising behavior can arise (and thus bugs)
- The winding path you take through code is called your *thread of execution*
- `print` is your friend!
- When a program is done, its printout is a record of what happened

Early `return`

- Remember how **`return`** can end a function early?
- That's even true in a loop!

```
def firstSquareGreaterThan(x: int) -> int:
    r = x + 1
    while True:
        sr = sqrt(r)
        if int(sr) == sr:
            return r
        r = r + 1
```

In-lecture quiz (L6)

- https://student.cs.uwaterloo.ca/~cs114/F25/quiz/
- Q1: What will this code print?

```
x = 5
while x != 0:
    print(x)
    x = x - 2
```

A. 5, 4, 3, 2, 1, 0
B. 5, 3, 1
C. 5, 3, 1, -1
D. 5, 4, 3, 2, 1
E. Endless output (it prints forever)

In-lecture quiz (L6)

- https://student.cs.uwaterloo.ca/~cs114/F25/quiz/
- Q2: What will this code print?

```
x = 5
while x > 0:
    print(x)
    x = x - 2
```

A. 5, 4, 3, 2, 1, 0
B. 5, 3, 1
C. 5, 3, 1, -1
D. 5, 4, 3, 2, 1
E. Endless output (it prints forever)

Computing with loops

CS114 M3

Computing something

- Let's use loops for something more useful:
  - `factorial` (n! = n * (n-1) * (n-2) … * 1)
  - `sumTo` (sumTo(n) = n + (n-1) + (n-2) … + 1)
- Let's add `print`s to factorial to understand how the values change at different steps around the loop

Types aside

- Let's run factorial with a large `int` and the same `float`
- In Python, `int`s have unlimited range, but `float`s don't!
- `float`s are still good enough for most uses; this is just a distinction to bear in mind

No need for counting

- So far we've just been counting up or down
- Your conditions and steps can be anything
- Let's write a function `printFloat` to print a floating-point value, only calling `print` on `int`s
  - Yes, I know that `print` will do this, but `print` isn't magic, it's just code!

No need for counting

```
def printFloat(f: float) -> None:
    i = int(f)
    print(i, ".")
    rem = f - i
    while rem > 0:
        rem = rem * 10
        i = int(rem)
        print(i)
        rem = rem - i
```

Computation!

- Let's do something that really feels like computing: factorization
- We'll write a function `factorize` that prints the factors of a whole number
- Then, a function `gcd` to compute the greatest common divisor of two numbers

Computation!

```
def factorize(n: int) -> None:
    assert n > 0, "Only positive
                   integers have
                   factors"

    f = 1
    while f <= n:
        if n%f == 0:
            print(f)
        f = f + 1
```

Computation!

```
def gcd(a: int, b: int) -> int:
    assert a > 0 and b > 0, "Only positive
                             integers have
                             factors"

    gcd = 1

    candidate = 2

    while candidate <= a and candidate <= b:

        if a%candidate == 0 and b%candidate == 0:

            gcd = candidate

        candidate = candidate + 1

    return gcd
```

Computation!

Note: The `gcd` we just wrote could have been done with an early return, and probably more clearly, by counting down instead of up. I'm just trying to show interesting loops here, not necessarily the best way to do it ☺

Loops within loops

- You can put anything[1] in your loops that you could put outside your loops
- You can even put loops in your loops!
- Let's complete our factorization computations by performing *prime* factorization

[1] You can even put function definitions and imports in loops, but this is usually considered very confusing.

Loops within loops

```
def primeFactors(n: int) -> None:
    assert n > 0, "Only positive integers
                   have factors"

    least = 2

    while n > 1:

        f = least

        while f < n and n%f != 0:

            f = f + 1

        print(f)

        least = f

        n = n // f
```

Loops within loops

Sometimes the condition is more about when you want the loop to *stop* than when you want the loop to *go*. Here, we want to *stop* at the first factor.

```
while n > 1:
    f = least
    while f < n and n%f != 0:
        f = f + 1
    print(f)
    least = f
    n = n // f
```

Aside on primes

- Where did I check that the factor was prime?
- Because I keep dividing out the primes I find, the least factor I find will *always* be a prime: every smaller value has already been divided out

Less obvious loops

- Let's write our own ~~terrible~~ version of `math.sqrt`
- We'll do this by approximating, then narrowing in until we find the value we want
- The math:
  - $r^2 = n$, so $r = \frac{n}{r}$
  - Guess an r.
  - If it's too small, $\frac{n}{r}$ is too big and vice-versa
  - In either case, choose a value between r and $\frac{n}{r}$ until we're close enough (within tolerance)

Less obvious loops

```
def sqrtButTerrible(n: float) -> float:
    assert n >= 0, "Imaginary numbers
                     unsupported"
    g = n/2
    while abs(g*g - n) > 0.0001:
        # print(g)
        g = (g + n/g) / 2
    return g
```