Warmup (L7)

This is our `firstSquareGreaterThan` function from last lecture:

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

Rewrite it such that it doesn't use an infinite loop or early return. That is, rewrite it so that the **`return`** is after the loop.

(If you're familiar with **`break`**, don't use it either.)

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
```

`for` loops

CS114 M3

More obvious loops

- The **`while`** loop is very powerful
- Most of the time we'll just have a grouping of values, and want to do something for every value in the group
- We'll see lots of groupings later, but focus on the simple `range` grouping for now

Looping over a range of numbers

- Remember our original `factorize` function? We just counted up to `n`.
- Python has a built-in facility to do these common counting loops that frees us from writing the obvious steps

```
def factorize(n: int) -> None:
    for f in range(1, n):
        if n%f == 0:
            print(f)
```

Two new concepts

- Our new factorize introduced two new concepts: the **`for`** loop and ranges
- ranges first: `range(1, n)` is a value that represents a grouping of all the values in the range from 1 to `n`
  - Lower-bound inclusive (1 is included)
  - Upper-bound exclusive (`n` is excluded)
  - Type is `range`

Two new concepts

- `for` is an easier but less powerful kind of loop than `while`
- It only lets us loop over a grouping
  - (Such as a range, but we'll see other groupings later)
- It's easier by saving us from typing the boilerplate (create a variable, update it each loop)
- But, it's less powerful because we can only loop over a grouping

for VS while

```python
def factorize(n: int) -> None:
    for f in range(1, n):
        if n%f == 0:
            print(f)
```

```python
def factorize(n: int) -> None:
    f = 1
    while f < n:
        if n%f == 0:
            print(f)
        f = f + 1
```

`for` VS `while`

Initial value comes from the range. No need to explicitly create the variable first.

```
def factorize(n: int) -> None:
    for f in range(1, n):
        if n%f == 0:
            print(f)
```

```
def factorize(n: int) -> None:
    f = 1
    while f < n:
        if n%f == 0:
            print(f)
        f = f + 1
```

`for` VS `while`

Update also comes from the range! No need to update the variable in the loop.

```
def factorize(n: int) -> None:
    for f in range(1, n):
        if n%f == 0:
            print(f)
```

```
def factorize(n: int) -> None:
    f = 1
    while f < n:
        if n%f == 0:
            print(f)
        f = f + 1
```

Don’t be afraid to `while`

- `for` is easy to use and applies in a lot of circumstances
- But it is strictly less powerful than `while`! Anything you can do with `for`, you can do with `while`, but `while` can do more!
- If you find yourself fighting a `for` loop that won’t do what you want, maybe you don’t want `for`

All the `ranges`

- `range(from, to)`
  - From `from` (inclusive) to `to` (exclusive)
- `range(to)`
  - From 0 to `to` (exclusive)
  - Same as `range(0, to)`
  - (Computer Scientists like to count from 0)
- `range(from, to, by)`
  - From `from` (inclusive) to `to` (exclusive), but skip by `by`. E.g. `range(0, 4, 2)` is {0, 2}
  - `by` can be negative to count backwards

`range` restrictions

- `range` only counts integers. No `float`s allowed!
- If the arguments don't make sense, there's no error, but there's no loop

It’s! Still! Imperative!

What will this print?

```
def countdown(from: int, to: int) -> None:
    for ct in range(from, to, -1):
        print(ct)
        to = to + 1
    print("Ignition")
countdown(5)
```

It’s! Still! Imperative!

The range is computed before the loop runs at all. Updating `to` does nothing.

```
def countdown(from: int, to: int) -> None:
    for ct in range(from, to, -1):
        print(ct)
        to = to + 1
    print("Ignition")
countdown(5)
```

It’s! Still! Imperative!

Perhaps more surprising, updating `ct` does nothing either. It steps through the range with no concern to how `ct` is changed.

```
def countdown(from: int, to: int) -> None:
    for ct in range(from, to, -1):
        ct = ct + 1
        print(ct)
    print("Ignition")
countdown(5)
```

In-lecture quiz (L7)

- https://student.cs.uwaterloo.ca/~cs114/F25/quiz/
- Q1: How many times does this print “x”?

```
for i in range(1, 4):
    while i < 4:
        print("x")
        i = i + 1
```

A. 0 (no times)
B. 3
C. 4
D. 6
E. 12

In-lecture quiz (L7)

https://student.cs.uwaterloo.ca/~cs114/F25/quiz/

- Q2: How many times does this print “x”?

```
i = 1
while i < 3:
    print("x")
    i = i + 1
    print("x")
    i = i + 1
    print("x")
    i = i + 1
    print("x")
    i = i + 1
```

A. 0 (no times)
B. 2
C. 3
D. 4
E. 8

Functions are values too

CS114 M3

Note

This section has nothing to do with loops. It’s here because of nowhere-else-to-stick-it-ism.

You can overwrite `print`

- When talking about variable names, I said nothing stops you from overwriting `print` (other than common sense)
- Implication: `print`, and any functions you make, are just variables!
- What's in the variable?

Functions are values too

```
q = abs
print(q(-3))
print(abs(-3))
print(q == abs)
w = print
w(q(-42))
```

Consider carefully

```
q = abs
```

```
q = abs(-3)
```

Consider carefully

```
q = abs
```

- `q` is now a function
- It’s the same function as `abs` (it *is* `abs`!)
- The code for `abs` never ran here

```
q = abs(-3)
```

- `q` is an `int`
- It’s 3
- The code for `abs` ran, and returned 3

Why???

- The power of abstraction!
- Previously we could work our way *out*, reusing the smaller abstractions to build bigger ones
- Now, we can abstract *big* things and fill in the inside later as we have other small things to do!

Typing functions

- The type for functions is in a module
  - This is the first type we've seen for which we need a module
- `import typing`
  `[…] f: typing.Callable […]`
- `from typing import Callable`
  `[…] f: Callable […]`

Why “callable”?

- Why is the type name “Callable” instead of “Function”?
- “Callable” just means “you can call it”, which is what we do with functions
- We’ll eventually see other kinds of things that can be called, and they’re just as good. So, we accept anything callable.

Prime factorization generalized

Let’s generalize our prime factorization function to do anything (rather than just `print`) for each factor

Prime factorization generalized

```
import typing

def primeFactors(n: int, cb: typing.Callable) -> 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
        cb(f)
        least = f
        n = n // f

def printAsFloat(x: int) -> None:
    print(float(x))

primeFactors(42, printAsFloat)
```

Prime factorization generalized

```
import typing

def primeFactors(n: int, cb: typing.Callable) -> None:
    assert n > 0, "Only positive integers
                     have factors"
    least = 2
```

“cb” (for “callback”) is a common name for a function argument when there’s no descriptive name for it

```
        cb(f)
        least = f
        n = n // f

def printAsFloat(x: int) -> None:
    print(float(x))

primeFactors(42, printAsFloat)
```