Warmup (L5)

Write a function with arguments `(a, b, c, d)`, all `float`s, that returns the greatest value among its four arguments.

What does this look like with nested **`if`**s? With **`and`**/**`or`**? Is there a cleaner way?

(Note: there's a built-in `max` function, but don't use that; that's too easy!)

Booleans are *values*

- Here's a new version of `multiOp`:

```
def multiOp(x: float, y: float, op: int) -> float:
    """

    …
    """

    if opInRange(op):
        if op == 0:
            return x + y
        elif op == 1:
            return x - y
        return x * y
    else:
        return -1
```

- Let's write `opInRange` to work with it.

Booleans are *values*

- Here's a new version of `multiOp`:

```
def multiOp(x: float, y: float, op: int) -> float:
    """
    ...
    """
    if opInRange(op):
        if op == 0:
            return x
        elif op == 1:
            return x
        return x * y
    else:
        return -1
```

There's no comparison here (at least, not directly!). We got a `bool` because that's what `opInRange` returned!

- Let's write `opInRange` to work with it.

Booleans are *values*

- Here's a new version of `multiOp`:

```
def multiOp(x: float, y: float, op: int) -> float:
    """
    ...
    """
    if opInRange(op):
        if op == 0:
            return x
        elif op == 1:
            return x
        return x * y
    else:
        return -1
```

Making the `op == 2` condition totally implicit is usually poor style, because it's unclear. I did it here just to show an **`elif`** without an **`else`**.

- Let's write `opInRange` to work with it.

The power of abstraction

CS114 M2

Nonobvious conditions

- Let's write a function `isEven` to check if an integer is even.
- None of our comparators look like "is even" or "divisible by"...
- New operator! `%`
  - Remainder after division, e.g., `5%2==1`
  - Called "modulo"

Modulo and quotient

- In math, remainder after division is usually paired with *quotient* to keep division in integers
- We can do the same to keep division in `int`s.
- Quotient is `//` (two slashes)

Modulo

- Wait, remainder after division still isn't "is even" or "divisible by"…
- A number is even if it's divisible by 2…
- A number is divisible by *y* if the remainder after division by *y* is 0…
- So, we can use ==: `x%y==0`

isEven

- With modulo in mind, let's write our `isEven` function.

isEven

- With modulo in mind, let's write our `isEven` function.

```
def isEven(v: int) -> bool:
    """
    Returns True if v is even,
    False otherwise.
    """
    return v%2==0
```

Using a function call as a condition

- `isEven` returns a `bool`, so that it can be used as a condition
- Let's use `isEven` to tell the user whether a number is even

Using a function call as a condition

- `isEven` returns a `bool`, so that it can be used as a condition
- Let's use `isEven` to tell the user whether a number is even

```
def printEvenness(v: int) -> None:
    if isEven(v):
        print(v, "is even")
    else:
        print(v, "is odd")
```

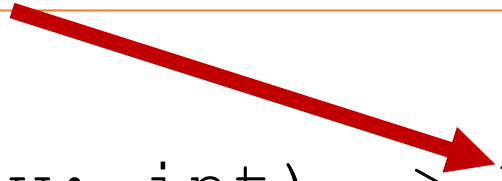
Using a function call as a condition

On slides I often won't show the docstring, just to make things shorter. You should still write docstrings for every function. Do as I say, not as I do!

```
def printEvenness(v: int) -> None:
    if isEven(v):
        print(v, "is even")
    else:
        print(v, "is odd")
```

Using a function call as a condition

Remember, if a function doesn't explicitly return, it returns `None`. You can use `None` as your return type to indicate that you meant to do that.



```
def printEvenness(v: int) -> None:
    if isEven(v):
        print(v, "is even")
    else:
        print(v, "is odd")
```

Using a function call as a condition

There's no comparison here, so where'd our boolean come from? A boolean is a *value!* `isEven` returned it, and we used that return directly.

```
def printEvenness(v: int) -> None:
    if isEven(v):
        print(v, "is even")
    else:
        print(v, "is odd")
```

Using a function call as a condition

This would also work, but it's considered poor style, because `== True` doesn't do anything useful here.

```
def printEvenness(v: int) -> None:
    if isEven(v) == True:
        print(v, "is even")
    else:
        print(v, "is odd")
```

The hat trick

- We were printing mostly the same string in either case
- Why not use a function to make the string, then print the string it returns?
  - This also means we could use it for other purposes in the future, not just printing

The hat trick

- Let’s make a function to stringify evenness:

The hat trick

- Let's make a function to stringify evenness:

```
def describeEvenness(v: int) -> str:
    if isEven(v):
        return "even"
    else:
        return "odd"

def printEvenness(v: int) -> None:
    print(v, "is", describeEvenness(v))
```

The hat trick

- This is getting complicated! In Jupyter, let's add some `print`s to trace through exactly what's happening

In-lecture quiz (L5)

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

```
def checkRange(x: int) -> None:
    if x < 10:
        print("in range")
    elif x < 5:
        print("too small")
    elif x > 10:
        print("too big")
checkRange(1)
```

A. in range

B. too small

C. too big

D. Nothing

In-lecture quiz (L5)

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

```
def checkRange(x: int) -> None:
    if x < 10:
        print("in range")
    elif x < 5:
        print("too small")
    elif x > 10:
        print("too big")
checkRange(10)
```

A. in range

B. too small

C. too big

D. Nothing

The power of abstraction

(Please excuse the overt grandiosity of this analogy)

Humans achieved complex societies through *specialization*: When some people specialize as farmers, that frees up time and energy for others to specialize as, e.g., metalworkers. The metalworkers free up time and energy so others can specialize in, say, construction.

The power of abstraction

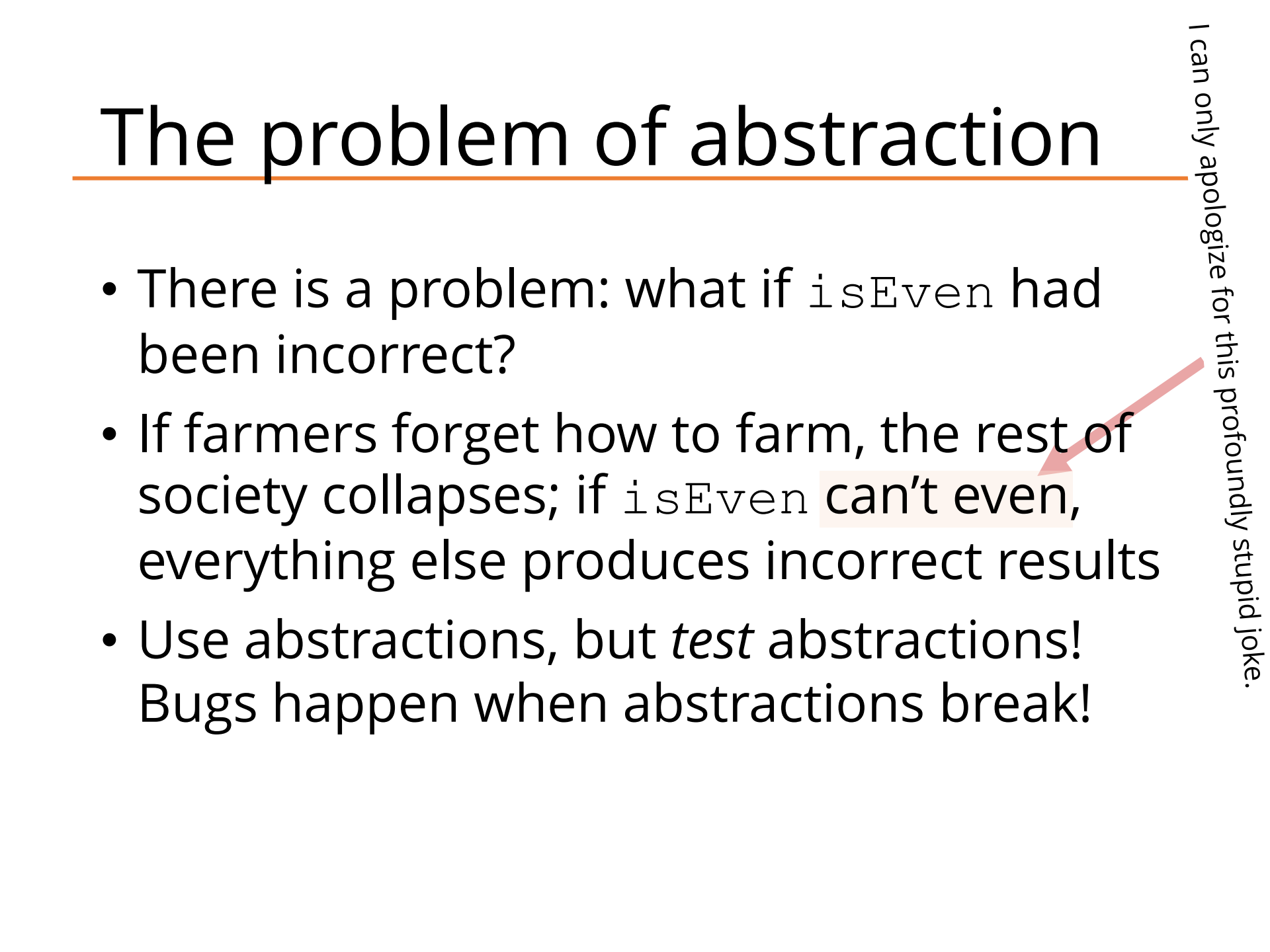
- Computer programs achieve similar power through *abstraction*
- Once you (or somebody else!) has written a function that does what you need, that frees you up from reinventing it
- As the abstractions keep building on other abstractions, our power to write specialized and more sophisticated programs increases

The power of abstraction

In our `isEven` example (in which every step is admittedly rather simple), we freed `describeEvenness` from the task of *determining* evenness, and `printEvenness` from *describing* evenness.

The problem of abstraction

- There is a problem: what if `isEven` had been incorrect?
- If farmers forget how to farm, the rest of society collapses; if `isEven` can't even, everything else produces incorrect results
- Use abstractions, but *test* abstractions! Bugs happen when abstractions break!



More examples

- Let's do some examples in Jupyter. We'll write tests first, then fill in the function.

We will categorize all numbers into one of four (rather silly) categories:

- Even integer (`"even"`),
- odd integer (`"odd"`),
- negative non-integer (`"neg"`),
- positive non-integer (`"pos"`).

```
def numberCategory(v: float) -> str:
```

More examples

- We wrote `numberCategory` with nested conditions. Let's try to write it without nested conditions.
- Ew! Ugly! If avoiding nesting requires rewriting conditions, nesting is preferable!

On testing

- When I introduced **`assert`**s, I suggested writing tests first. This is called *black-box testing*.

  black box
  noun
  2: A device which performs intricate functions but whose internal mechanism may not readily be inspected or understood; (hence) any component of a system specified only in terms of the relationship between inputs and outputs. Also *figurative*.
  (— Oxford English Dictionary)

- Knowing how the code works can bias your tests. When writing your own tests, *black-box tests* should be written *before* the code.
- *White-box* tests are written with knowledge of the code. Write these too!

On testing

- Of course, practically speaking, you should have some sense of how your code will work before you write it
- You can’t black-box your own mind
- Don’t overthink it. Just test!
- Black-box testing is good for avoiding biases; white-box testing is good for catching corner cases specific to the implementation.

Other comparisons

CS114 M2

Other comparisons

- Our comparators are for anything, not just numbers
- `==` and `!=` can be used to compare strings, or even bools
  - `"foo" == "foo"`
  - `"foo" != "bar"`
  - `False == False`
  - `5 != "5"`
  - `True == 1` ???
    This is a weird CS thing, you can ignore it ☺

Other comparisons

- You can also use <, <=, >, and >= with strings, and it mostly compares them how they would be put in a dictionary
  - `"aardvark" < "zebra"`
  - `"Richards" > "Gregor"`
- But not quite…
  - `"Z" < "a"`

```
# Upper-case letters are
# all “less than” lower-
# case letters
```

Final examples

- Some more examples to try:
  - A *safe division* function that avoids dividing by zero (look up `math.inf`)
  - Python doesn't provide `xor` (exclusive or), so we'll make our own
    - `xor3` too!
  - Three-party majority vote

Module summary

CS114 M2

Module summary

- Use **`if`**, **`elif`**, **`else`** to make decisions
- Decisions let you run things conditionally
- Early returns can confuse conditions
- Decisions are booleans: `True` or `False`
- Combine booleans with **`and`**, **`or`**, **`not`**
- Booleans are values and can be passed around
- Get accustomed to how conditional code works!