Warmup (L20)

- Create a class that:
  - Stores a list
  - Lets the user append to the list with an `append` method
  - Gives the user a NumPy array with an `array` method
  - Gives them the same NumPy array (does not re-convert) if they ask for it twice without `append`ing in between

Class invariants

CS114 M8

Invariant

invariant (adjective):
constant, unchanging
specifically: unchanged by specified mathematical or physical operations or transformations
— Merriam Webster Dictionary

- What are the invariant properties of our `StatsList`s?

Invariants

- Invariant properties of `StatsList`:
  - The `sum` field contains the sum of the values in the `lst` field
  - The `product` field contains the product of the values in the `lst` field
  - The `min` and `max` fields contain the least and greatest values in the `lst` field

("Invariant properties" is usually abbreviated as "invariants")

Maintaining invariants

- In a properly written class, all methods maintain the invariants
- In the case of `StatsList`'s invariants, every method that modifies `lst` should consummately modify `sum` and `product`
- Fields can't protect invariants, so it's the job of the user not to break them
  - Don't do `sl.lst.append(...)`, or that'll break its invariants!

Maintaining invariants is hard

- Let's add a `pop` (remove) method to our StatsList
- Maintaining the `sum` invariant is easy: just subtract it
- Maintaining the `product` invariant is not
- Let's find out why

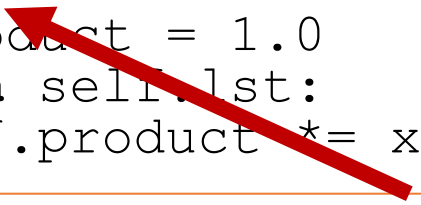
```
class StatsList:
    """Stores a list of numbers
       and provides some simple
       statistics."""
    lst: list[float]
    sum: float
    product: float
    …

    def append(self, val: float) -> None:
        self.lst.append(val)
        self.sum += val
        self.product *= val # Watch for zeroes!
        …

    def pop(self, idx: int) -> float:
        val = self.lst.pop(idx)
        self.sum -= val
        self.product /= val # I said watch for
                            # zeroes!!!
        # How to deal with min and max?
        return val

    …
```

Maintaining invariants

- It's up to you to anticipate corner cases like this one
- It's also up to you to deal with them!
- Let's look at how to deal with the `product` invariant (threatened by zeroes) and the `min`/`max` invariants

```
class StatsList:
    """Stores a list of numbers
       and provides some simple
       statistics."""
    lst: list[float]
    sum: float
    product: float
    …

    def append(self, val: float) -> None:
        self.lst.append(val)
        self.sum += val
        self.product *= val
        …

    def pop(self, idx: int) -> float:
        val = self.lst.pop(idx)
        self.sum -= val
        if val == 0:
            self.product = 1.0
            for x in self.lst:
                self.product *= x
        else:
            self.product /= val
        return val

    …
```

```
class StatsList:
    """Stores a list of numbers
       and provides some simple
       statistics."""
    lst: list[float]
    sum: float
    product: float
    …

    def append(self, val: float) -> None:
        self.lst.append(val)
        self.sum += val
        self.product *= val
        …

    def pop(self, idx: int) -> float:
        val = self.lst.pop(idx)
        self.sum -= val
        if val == 0:
            self.product = 1.0
            for x in self.lst:
                self.product *= x
        else:
    …
```



If the value was zero, then it already broke the product. We just have to recalculate.

```
class StatsList:
    """Stores a list of numbers
       and provides some simple
       statistics."""
    lst: list[float]
    …
    min: float
    max: float
    …

    def pop(self, idx: int) -> float:
        val = self.lst.pop(idx)
        …
        if len(self.lst) == 0:
            # No longer a min or max
            self.min = 0.0
            self.max = 0.0
        else:
            if val == self.min: # == is fine here! Why?
                self.min = min(self.lst)
            if val == self.max:
                self.max = max(self.lst)
        return val

    …
```

```
class StatsList:
    """Stores a list of numbers
       and provides some simple
       statistics."""
    lst: list[float]
    …
    min: float
    max: float
    …

    def pop(self, idx: int) -> float:
        val = self.lst.pop(idx)
        …
        if len(self.lst) == 0:
            # No longer a min or max
            self.min = 0.0
            self.max = 0.0
        else:
            if val == self.min: # == is fine here! Why?
```

Be careful of edge cases like this! Trying to find the min when the list is empty wouldn't work!

```
    …
```

```
class StatsList:
    """Stores a list of numbers
       and provides some simple
       statistics."""
    lst: list[float]
    …
    min: float
    max: float
    …

    def pop(self, idx: int) -> float:
        val = self.lst.pop(idx)
        …
        if len(self.lst) == 0:
            # No longer a min or max
            self.min = 0.0
            self.max = 0.0
        else:
            if val == self.min: # == is fine here! Why?
                self.min = min(self.lst)
                …
```

== is fine here, in spite of the fact that these are floats, because there was no math done. If this was the min, we stored the exact value in `self.min`.

In-lecture quiz (L20)

- https://student.cs.uwaterloo.ca/~cs114/quiz/
- Q1: I accidentally added a value with `sl.lst.append` instead of `sl.append` and broke my invariants. How do I fix them?

A. You can't
B. `sl.pop(-1)`
C. `sl.lst.pop(-1)`
D. `sl.pop(-1)`, then update `sum` and `product` manually

Special methods

CS114 M8

Special methods

- `__init__` is a “special method”
- “Special method” means what it says: a method that serves a special purpose to the language
- `__init__` is not the only special method
- Most are called for some operator. E.g., `x + y` becomes `x.__add__(y)`

Terminology

These are often called “dunder methods”

“Dunder” just stands for “double-underscore”

Special methods

- `str(x)` → `x.__repr__()`
  - I.e., “representation”
  - Also used for `f"{x}"` and `print(x)`
- Be careful about the `self` parameter! `__repr__` takes one argument, `(self)`, not zero!
- Most other special methods are for operators

Special methods

- `x == y` → `x.__eq__(y)`,
  `x != y` → `x.__ne__(y)`
- `x < y` → `x.__lt__(y)`
  - Similar for `__le__`, `__gt__`, `__ge__`
- `x + y` → `x.__add__(y)`
  - Similar for `__sub__`, `__mul__`, `__mod__`, `__pow__`
- `x / y` → `x.__truediv__(y)`,
  `x // y` → `x.__floordiv__(y)`

Special methods

- This is best demonstrated by doing
- Let's add every special method (that makes any sense) to our `StatsList` class

(This example is too big to duplicate in the slides. If you're reading along in the slides, open the notebook here.)

Advanced special methods

These are less frequently useful (and wouldn't appear on an exam), but you might want to know them for your own use:

- `if x:` → `if x.__bool__():`
- `x(a, b, c)` → `x.__call__(a, b, c)`
- `len(x)` → `x.__len__()`
- `x[y]` → `x.__getitem__(y)`
- `x[y] = z` → `x.__setitem__(y, z)`
- `for y in x:` →
  `for y in x.__iter__():`
- `y in x` → `x.__contains__(y)`

There are even more than these! Search for "Python special methods"

Advanced topics

- More we won't be covering ☹
- You can make *subclasses*, which expand on the behavior of an existing class
  - Idea is you could make multiple subclasses to serve different needs
- There are also ways of making your classes have interesting types, like `StatsList[float]`.

Objects all the way down

- Remember when I said that in Python, everything is an object?
- Every operator has always been a special method:
  - `(3).__add__(4) == 7`

Fun with classes

CS114 M8

Fun with classes

- Some extended exercises:
  - Our warmup appendable array, but give it array-like operators and the ability to initialize from an existing iterable
  - Insertion sorting: A list that always remains sorted because we insert things in their sorted position

Module summary

CS114 M8

Module summary

- Create your own types with classes
- List fields with their types
- `__init__` to initialize objects
- All methods take `self` argument
- Add any methods you want
- Class invariants
- Special methods