Warmup (L19)

- Write a function `tiles` to find the *n*-tiles of a sequence (quartiles, quintiles, etc)
  - ```
    tiles(
        seq: typing.Sequence[float],
        n: int
    ) -> list[tuple[float, float]]
    ```
  - Return tuples are lowerbound and upperbound of each range
    ```

Classes

CS114 M8

Putting things in other things

CS114 M8

Putting things in other things

- Modules have things in them (e.g., `math.pi`, `np.sum`)
- Values have things in them too (e.g., `str.split`, `arr.shape`)
- This is organizationally helpful, because related things are together
- But it's also helpful for types: you can't accidentally try to split anything but a string, because split is *in* the string!

Putting things in other things

- Putting things in other things isn't just for built-in types: we can make our own types!
- Let's build a type that stores things like a list (in fact, we'll store things *in* a list), but keeps track of the minimum, maximum, sum, and mean as you go
  - We'll also throw in the product and geometric mean, for kicks

Classes

- Types are defined by classes
- A *class* boxes up functions to make them methods, and values to make them fields
  - Fields are like `arr.shape`: accessible on the values, but not functions, so not methods
- Collectively, methods and fields are the "*attributes*" of the type
- We can make values from the class, and they will have these attributes

Starting our class

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

- Docstrings are like in functions: if I call help(StatsList), I'll get that string
- Fields are just listed with their types

Classes

- When a class is called as a function, it makes a value that has these attributes
  - We usually call such values objects, but…
  - … all the values in Python are defined by classes, so they're all objects! That's not true in other programming languages.
- Let's use our StatsList and see what happens

Where are my fields?

```
x = StatsList()
print(x.lst)  # CRASH! StatsList
              # has no attribute
              # lst!
```

- But I gave it a field `lst`! What's going on?

Initializing classes

CS114 M8

Where are my fields?

- Something to remember about type annotations from waaaaaay back: *they're just documentation*
- To have a field there, we have to put it there
- We can add fields just like we add keys to dictionaries. That is, just set them!

There are my fields!

```
x = StatsList()
x.lst = []
print(x.lst)
```

- This works now, but it's a bit unsatisfying
- After all, when we make a NumPy array, we didn't have to make the array, *then* put something in it

The initializer

- Classes can contain methods
- One method is used as the *initializer*
  - (It's called when you create a value in the class)
- The initializer is named `__init__`
  - That's two underscores, then "init", then two underscores

Add an initializer

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

    def __init__(self) -> None:
        self.lst = []

x = StatsList()
print(x.lst) # []
```

Add an initializer

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

    def __init__(self) -> None:
        self.lst = []

x = StatsList()
```

If we just said lst = [], that would create a *local variable* called lst, not a field of the object

Add an initializer

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

    def __init__(self) -> None:
        self.lst = []

x = StatsList()
```

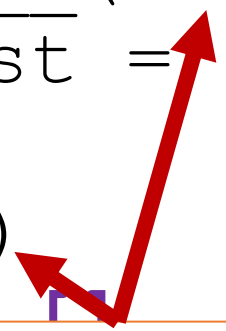
When you call a method, the object you're calling it *on* (the thing before the .) becomes the first argument

Add an initializer

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

    def __init__(self) -> None:
        self.lst = []

x = StatsList()
print(x.lst) # []
```



`__init__` is automatically called on new `StatsList` objects, so `self` here is the `StatsList` object we're building

Add an initializer

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

def initStatsList(self) -> None:
    self.lst = []

x = StatsList()
initStatsList(x)
```

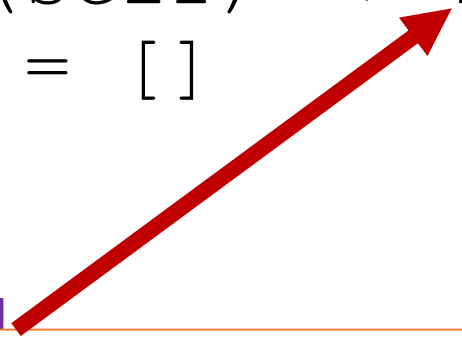
This does the same thing, but is more annoying since we have to initialize by hand

Add an initializer

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

    def __init__(self) -> None:
        self.lst = []

x = StatsList()
print(x.lst) # []
```



Note that `__init__` doesn't return anything. When we call the class *itself*, it returns the new value. `__init__` just initializes.

Add an initializer

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

    def __init__(self) -> None:
        self.lst = []

x = StatsList()
print(x.lst) # []
```

`self` doesn't require a type annotation, because it's clear from context (the first argument of a method is always typed as the class)

The mystery of `self`

- Whenever you call a method (that is, do `x.y(...)`), the object you're calling the method on (in this case `x`) becomes the first argument
- By convention, it's named `self`
- If you forget to put the `self` argument, the target will still become the first argument, and you'll get very weird error messages!

Brief aside

- The idea of classes is very common
- In all other programming languages (I'm aware of), the first argument ("`self`") isn't explicitly listed
- It's also usually called "`this`", but "`self`" is actually the older name
- Otherwise, methods work very similarly

In-lecture quiz (L19)

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

```
class RatherSilly:
    def useless(x):
        return x
a = RatherSilly()
print(a.useless(42))
```

A. Nothing or an error
B. <__main__.RatherSilly object ...>
C. 42
D. RatherSilly
E. useless

In-lecture quiz (L19)

- https://student.cs.uwaterloo.ca/~cs114/quiz/
- Q2: OK, *now* what will this print?

```
class RatherSilly:
    def useless(self, x):
        return x
a = RatherSilly()
print(a.useless(42))
```

A. Nothing or an error
B. <__main__.RatherSilly object ...>
C. 42
D. RatherSilly
E. useless

Adding methods

CS114 M8

Other methods

- So far, `StatsList` is just a pointless way of boxing up a list
- Our reason for it was to make it do stats automatically when we expand the list
- We can't actually change the methods of the list in its field: that's just a list
- So, we add our own methods and tell users to use those

Append

- Let’s add a method to append to the list

Append

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

    def __init__(self) -> None:
        self.lst = []

    def append(self, val: float) -> None:
        self.lst.append(val)
```

Stats

- Of course, the whole point of this class was that it should do stats
- So, when we append a value, let's also do some statistics
- We'll compute the sum and product, so that we can easily get the mean and geometric mean
  - Geometric mean = product to the count root

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

    def __init__(self) -> None:
        self.lst = []
        self.sum = 0.0
        self.product = 1.0

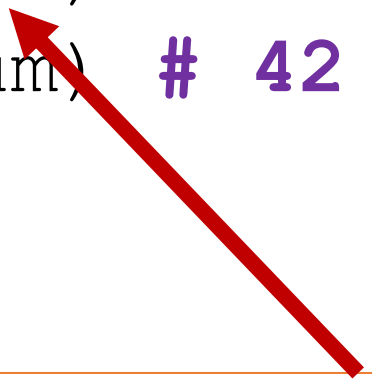
    def append(self, val: float) -> None:
        self.lst.append(val)
        self.sum += val
        self.product *= val # Watch for zeroes!
```

Stats

```
x = StatsList()
x.append(42)
print(x.sum) # 42
```

Stats

```
x = StatsList()
x.append(42)
print(x.sum) # 42
```



Note how `append` has two arguments, but we only write one here. That's because `x` is implicitly the first argument, `self`!

I said “stats”!

- Now that we have the sum and product, it’s easy to get the mean and geometric mean
- So far, the intention of our methods was to *do* things, but these methods just *calculate* things
- That’s fine, they’re still just methods!
- Just mind your return types

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

    def __init__(self) -> None:
        self.lst = []
        self.sum = 0.0
        self.product = 1.0

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

    def mean(self) -> float:
        return self.sum / len(self.lst)

    def geometricMean(self) -> float:
        return self.product ** (1/len(self.lst))
```

Last few stats

- Our goal was for our class to to record:
  - **Minimum**
  - **Maximum**
  - Sum
  - Mean
  - Product
  - Geometric mean
- Let’s handle the two we haven’t done yet

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

    def __init__(self) -> None:
        self.lst = []
        self.sum = 0.0
        self.product = 1.0
        # We don't know a minimum or maximum yet, so
        # these are just arbitrary values
        self.min = 0.0
        self.max = 0.0

    def append(self, val: float) -> None:
        self.lst.append(val)
        self.sum += val
        self.product *= val # Watch for zeroes!
        # Be careful! If this is the *first* element in
        # the list, then min and max haven't yet been set!
        if len(self.lst) == 1:
            self.min = val
            self.max = val
        else:
            if val < self.min:
                self.min = val
            if val > self.max:
                self.max = val

    …
```

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")