Warmup (L13)

- You have a `dict[str, float]`
- Sort the *keys*, sorting by the *values* the keys are associated with in the dictionary, in decreasing order

```
x = {"A": 42, "B": -3, "C": math.pi}
# We want ["A", "C", "B"]
```

Conversions

CS114 M5

Converting to a dictionary

- Technically, `dict` can be used to convert a sequence to a dictionary…
- but, it wants a sequence of key-value pairs, with each pair as a tuple:

```
x = dict([(0, 0), (1, 1), (2, 4), (3, 9)])
```

- That's a pretty unlikely type to find unless you specifically intended to make a dictionary with it (and if you did, why didn't you just put it in a dictionary in the first place?)

Converting to a dictionary

- More generally, it usually doesn't make sense to convert to a dictionary. Here's how you might:

```
def toDictionary(
    seq: typing.Sequence
) -> dict:
    r = {}
    for idx in range(len(seq)):
        r[idx] = seq[idx]
    return r
```

Fun with dictionaries

CS114 M5

Memoization

- *Memoization* is remembering the result of a computation so that if the same computation is requested again, we can reuse the previous result
- Dictionaries are great for memoization!
- Let's memoize our `divisors` function

Memoized divisors

- Original for reference

```
def divisors(x: int, y: int) -> list[int]:
    r = []
    i = 1
    while i <= x and i <= y:
        if x%i == 0 and y%i == 0:
            r.append(i)
        i = i + 1
    return r
```

Memoized divisors

```
memo: dict[tuple[int, int], int] = {}

def divisors(x: int, y: int) -> list[int]:
    if (x, y) in memo:
        return memo[(x, y)]
    r = []
    i = 1
    while i <= x and i <= y:
        if x%i == 0 and y%i == 0:
            r.append(i)
        i = i + 1
    memo[(x, y)] = r
    return r
```

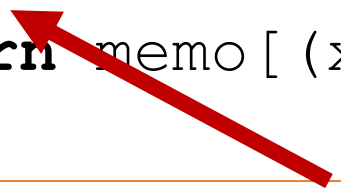
Memoized divisors

```
memo: dict[tuple[int, int], int] = {}

def divisors(x: int, y: int) -> list[int]:
    r = []
    i = 1
    while i <= x and i <= y:
        if x%i == 0 and y%i == 0:
            r.append(i)
        i = i + 1
    memo[(x, y)] = r
    return r
```

Python will usually guess the type if you don't tell it, but it doesn't like mystery dictionaries, so we had to put a type annotation here.

Memoized divisors

```
memo: dict[tuple[int, int], int] = {}

def divisors(x: int, y: int) -> list[int]:
    if (x, y) in memo:
        return memo[(x, y)]
    r = []
            r.append(i)
        i = i + 1
    memo[(x, y)] = r
    return r
```



Yes, even tuples can be the key!
(Fits really well here, since we have two arguments)

Memoized divisors

```
memo: dict[tuple[int, int], int] = {}

def divisors(x: int, y: int) -> list[int]:
    if (x, y) in memo:
        return memo[(x, y)]
    r = []
            r.append(i)
        i = i + 1
    memo[(x, y)] = r
    return r
```

memo changes every time we call this, so the next time, we'll see the changes made from the last time

Memoized divisors

- Big red flag on that example: lists are mutable!

```
lst = divisors(2, 4)
lst.append("a bag full of squirrels")
print(divisors(2, 4)) # [1, 2,
                      #  "a bag full of
                      #   squirrels"]
```

Memoized divisors (fixed)

```
memo: dict[tuple[int, int], int] = {}

def divisors(x: int, y: int) -> list[int]:
    if (x, y) in memo:
        return memo[(x, y)][:]
    r = []
    i = 1
    while i <= x and i <= y:
        if x%i == 0 and y%i == 0:
            r.append(i)
        i = i + 1
    memo[(x, y)] = r
    return r[:]
```

In-lecture quiz (L13)

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

```
a = {"q": [1, 2]}
b = a
c = a["q"]
c.append(3)
b["r"] = c
print(a)
```

A. `['q']`

B. `['q', 'r']`

C. `{'q': [1, 2, 3]}`

D. `{'q': [1, 2], 'r': [1, 2, 3]}`

E. `{'q': [1, 2, 3], 'r': [1, 2, 3]}`

Real histogram

- We made a distribution function, but real histograms divide things into *bins*
- Let's make a binning histogram
- Stage one: make bins
- Given a minimum and maximum value, divide that range into a given number of bins
- (Note: We're going to do this in a more complex way than is needed to demonstrate lists and sorting and dictionaries.)

Real histogram: bins

```
def bins(
    min: float, max: float,
    binCt: int
) -> list[tuple[float, float]]:
    bins = []
    span = max - min
    for binNum in range(binCt):
        bins.append((
            span/binCt*binNum + min,
            span/binCt*(binNum+1) + min
        ))
    return bins
```

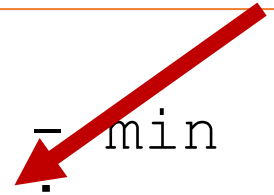
Real histogram: bins

```
def bins(
    min: float, max: float,
    binCt: int
) -> list[tuple[float, float]]:
    bins = []
    span = max - min
    return bins
```

What a complicated type! Well, a bin is a range (a minimum and maximum for that bin), so it's two numbers. Thus, our set of bins will be a list of those pairs.

Real histogram: bins

We know how many bins we want, but there's nothing else to loop over, so we simply loop over the bin number (0, 1, ..., `binCt`-1).



```
bins = []
span = max - min
for binNum in range(binCt):
    bins.append((
        span/binCt*binNum + min,
        span/binCt*(binNum+1) + min
    ))
return bins
```

Real histogram: bins

Each bin is a tuple (mind your parentheses!)

```
    min: float, max: float,
    binCt: int
) -> list[tuple[float, float]]:
    bins = []
    span = max - min
    for binNum in range(binCt):
        bins.append((
            span/binCt*binNum + min,
            span/binCt*(binNum+1) + min
        ))
    return bins
```

Real histogram: bins

What's this math? We split the range into `binCt` many bins (`span/binCt`), and this is bin #`binNum` (`*binNum`). But, that just split up the span, i.e., `max-min`. To get it back into range, we need to add on `min`.

```
span = max - min
for binNum in range(binCt):
    bins.append((
        span/binCt*binNum + min,
        span/binCt*(binNum+1) + min
    ))
return bins
```

Real histogram: bins

- There is a problem with the bins we just made: what part of the range is inclusive vs. exclusive?
- If we say it's lower-bound-inclusive, upper-bound-exclusive, then the max value won't actually go into any bin…
- We'll take that approach, and just fix the max value later.

Real histogram: find my bin

- Step two: Which bin does this value belong to?
- Given a list of bins and a value, choose the appropriate bin
- To solve the exclusivity problem, we'll also look for values that don't seem to be in any bin

Real histogram: find my bin

```
def findBin(
    val: float,
    bins: list[tuple[float, float]]
) -> tuple[float, float]:
    # Fix the exclusivity problem
    if val <= bins[0][0]: # min
        return bins[0]
    if val >= bins[-1][1]: # max
        return bins[-1]
    # Look for a matching bin
    for bin in bins:
        if val >= bin[0] and val < bin[1]:
            return bin
    # Some default if the above somehow fails
    return bins[-1]
```

Real histogram: find my bin

```
def findBin(
    val: float,
    bins: list[tuple[float, float]]
) -> tuple[float, float]:
    # Fix the exclusivity problem
    if val <= bins[0][0]: # min
        return bins[0]
    for bin in bins:
        if val > bin[0] and val < bin[1]:
            return bin
    # Some default if the above somehow fails
    return bin[-1]
```

Our squishy human brains can deduce that this return is unnecessary (we can never get here), but Python doesn't know that, so we put a default return to make the type checker happy.

Real histogram: make the histogram

- Finally, let’s put it together and make a histogram for a list!

Real histogram

```
def histogram(
    values: list[float],
    binCt: int
) -> dict[tuple[float, float], int]:
    s = sorted(values)
    vBins = bins(s[0], s[-1], binCt)
    r = {}
    # Each bin starts empty
    for bin in vBins:
        r[bin] = 0
    # Add the values
    for val in values:
        bin = findBin(val, vBins)
        r[bin] = r[bin] + 1
    return r
```

Real histogram

```
def histogramChart(
    lst: list[float], binCt: int
) -> None:
    hist = histogram(lst, binCt)
    for key in sorted(hist):
        print(key, "*" * hist[key])
```

Invert dictionary

- Let’s invert a dictionary (swap keys for values)

Invert dictionary

```
def invertDictionary(inDict: dict) -> dict:
    outDict = {}
    for key in inDict:
        outDict[inDict[key]] = key
    return outDict
```

Invert dictionary

- That inversion is imperfect, because of how keys work: multiple keys can have the same value
- Let’s make a version that inverts into a list (the list of all keys that had the same value)

Invert dictionary

```
def invertDictionaryList(
    inDict: dict
) -> dict[typing.Any, list]:
    outDict = {}
    for key in inDict:
        val = inDict[key]
        if not (val in outDict):
            outDict[val] = []
        outDict[val].append(key)
    return outDict
```

Module summary

CS114 M5

Module summary

- Sort with .sort or sorted
- Sort can reverse
- Sort can take a “key”
- Dictionaries associate keys (different kind of keys) with values
- Dictionaries are mutable
- Looping over dictionaries