Warmup (L11)

- Write a function `sortByWordCount` to sort a list of strings in place by number of words (not string length).
  - Hint: `str.split()` splits a string into a list of strings using whitespace.

Dictionaries

CS114 L10 (M5)

The trouble with tuples

- Here's some info on me

```
info = (
     "Richards",
     "Gregor",
     1.76,
     "University of Waterloo",
     "Purdue University",
     2014,
     11
)
```

- … but, what means what?

Name your variables!

- Good variable naming is important to understandable code
- Tuples essentially prevent that: the values within the tuple just have indices
- If only we could group values together but still name them all!

The dictionary

- Dictionaries store various values (like tuples) but associate each value with a “key”
- The key can be anything, but let’s start with a string to demonstrate

Basic dictionary

```
info = {
    "surname": "Richards",
    "given name": "Gregor",
    "height": 1.76,
    "employer": "University of Waterloo",
    "alma mater": "Purdue University",
    "graduation year": 2014,
    "employment years": 11
}

print(
    info["given name"], info["surname"],
    "works at", info["employer"]
)
```

New syntax!

- Dictionaries are written in curly braces: `{` and `}`
- Dictionaries contain *key-value pairs*: if you use this key, you will find this value
- Key-value pair written with a colon *`key`*`: `*`value`* e.g. `"surname": "Richards"`
- The key is any Python value (confusingly), so strings can be used as names as done here

Dictionaries are mutable

- Dictionaries are mutable reference types
- Value can be changed by setting it

```
info["employment years"] = 12
print(info["employment years"]) # Now 12
```

Dictionary powers

CS114 L10 (M5)

Expanding and contracting

- Dictionaries can be expanded by setting new keys

```
print(info["citizenship"]) # ERROR!
info["citizenship"] = ["USA"]
print(info["surname"]) # Still there
print(info["citizenship"]) # Now also there
```

Expanding and contracting

- With dictionaries, "**`in`**" is *key* presence

```
if "age" in info:
    print("This person is", info["age"], "years old")
```

Expanding and contracting

- Remove a key (and its value) with `.pop`

```
info.pop("employer") # Fired for tormenting
                     # Science students
print(info["employer"]) # ERROR!
```

Typing dictionaries

- The type for a dictionary is `dict`
- If you know the key and value types, and they're consistent, `dict[key, value]`
- You can use `typing.Any` for either key or value if one is consistent but the other isn't
- This will become clearer when we write some code, so…

Distribution

- Let’s write a function to count the number of instances of each value in a sequence
  - e.g. in [8, 6, 7, 5, 3, 0, 9, 2, 4, 6, 0, 1], we want 8 associated with 1, 6 associated with 2, etc.

Distribution

```
import typing

def distribution(
    lst: typing.Sequence
) -> dict[typing.Any, int]:
    r = {}
    for val in lst:
        if not (val in r):
            r[val] = 0
        r[val] = r[val] + 1
    return r

print(distribution([
    8, 6, 7, 5, 3, 0, 9, 2, 4, 6, 0, 1
]))
```

Before incrementing the value in the dictionary, we need to make sure there's something there

Almost a chart

- Building on distribution, let's make a simple distribution chart by printing as many *s as there are instances of each value
  - We need one trick first:

```
"*" * 3 == "***"
```

Almost a chart (first try)

```
def distributionChart(
    lst: typing.Sequence
) -> None:
    dist = distribution(lst)
    for key in dist:
        print(key, "*" * dist[key])
```

- **`for`** with a dictionary loops over *keys*
- This version is a bit unsatisfying, because it's printed in whatever order they first appeared in the sequence

Ordered chart

- To loop in order, we're going to have to sort the keys
- To do that, we need to get the keys as a sequence (we can sort any sequence)
- But we could `for` over it: the keys were already a sequence!
- In short: when you treat a dictionary as a sequence, it's a sequence of keys.

Ordered chart

```
def distributionChart(
    lst: typing.Sequence
) -> None:
    dist = distribution(lst)
    for key in sorted(dist):
        print(key, "*" * dist[key])
```

- Bonus: This isn't specific to lists! Works with any sequence, even strings!

Careful with floats!

- Remember that floats lie

```
annoying = {}
annoying[0.3-0.2] = "Hello"
annoying[0.1] = "world"
print(annoying)
```

In-lecture quiz (L11)

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

```
def first(s: str) -> str:
    return s[0]
print(sorted(
    ["an", "aardvark", "ate", "ants"],
    key=first
))
```

A. Nothing or an error

B. `['an', 'aardvark', 'ate', 'ants']`

C. `['aardvark', 'an', 'ants', 'ate']`

D. `['an', 'ate', 'ants', 'aardvark']`

E. `aardvark an ants ate`

Conversions

CS114 L11 (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 L10 (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
```

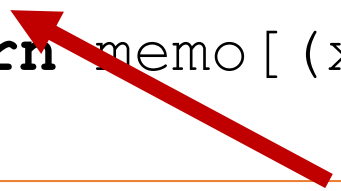
Memoized divisors

```
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[:]
```

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
```

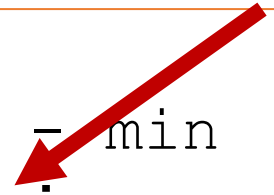
Real histogram: bins

```
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 L10 (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