Sorting and Dictionaries

M5

in order? put things Why

CS114 M5

Sorting

- Lists can be in any order (whatever order you put things in it)
- Some things would be easier if they were in order: What’s the least value? The most? The biggest gap?
- So, in some cases it’s useful to put things in order

Sorting a list

- Lists have a method to do this!

```
lst = [2, 4, 6, 0, 1]
lst.sort()
print(lst) # [0, 1, 2, 4, 6]
```

Sorting a list

- Lists have a method to do this!

```
lst = [2, 4, 6, 0, 1]
x = lst
lst.sort()
print(x) # [0, 1, 2, 4, 6]
```

This sorts *in place*, so all references will see the same sorting.

It’s just <

- Under the surface, it’s just using < to sort
- < works on numbers and strings, but not *between* numbers and strings

```
lst = [99, "bottles of beer on the wall"]
lst.sort() # Error!
```

Surprising sorts!

- < can also compare lists, so `.sort()` can sort a list of lists!

```
x = [[3], [2, 1]]
x[0] < x[1] # False
x.sort()
print(x) # [[2, 1], [3]]
```

Non-mutating sort

- `.sort()` mutates the list
- It’s a method of lists: doesn’t work on strings, ranges, tuples
- Also a built-in function to sort any sequence, by duplicating it into a list:

```
x = sorted("hello, world!")
print(x) # [" ", "!", ",", "d", "e",
         #  "h", "l", "l", "l", "o",
         #  "o", "r", "w"]
```

Least, greatest, gap

- Let’s solve exactly the question we started with: find the least, greatest, and greatest gap of a list

Least, greatest, gap

- Let’s solve exactly the question we started with: find the least, greatest, and greatest gap of a list

```
def lgg(lst: list[float]) -> tuple[float, float, float]:
    s = sorted(lst)
    greatestGap = 0.0
    for idx in range(0, len(lst)-1):
        gap = s[idx+1] - s[idx]
        if gap > greatestGap:
            greatestGap = gap
    return (s[0], s[-1], greatestGap)
```

Named parameters

CS114 M5

Read the documentation!

- Feeling a bit limited by `.sort()` and `sorted`? Remember that `help` can tell us how to use anything!

```
help([1, 2, 3].sort)
Help on built-in function sort:

sort(*, key=None, reverse=False) method of builtins.list
instance
    Sort the list in ascending order and return None.

    The sort is in-place (i.e. the list itself is modified) and stable
(i.e. the
    order of two equal elements is maintained).

    If a key function is given, apply it once to each list item and
sort them,
    ascending or descending, according to their function
values.

    The reverse flag can be set to sort in descending order.
```

```
Help on built-in function sort:

sort(*, key=None, reverse=False) method of builtins.list instance
    Sort the list in ascending order and return None.

    The sort is in-place (i.e. the list itself is modified) and stable (i.e. the
    order of two equal elements is maintained).

    If a key function is given, apply it once to each list item and sort them,
    ascending or descending, according to their function values.

    The reverse flag can be set to sort in descending order.
```

```
Help on built-in function sort:

sort(*, key=None, reverse=False) method of builtins.list instance
    Sort the list in ascending order and return None.

    The sort is in-place (i.e. the list itself is modified) and stable (i.e. the
```

An “instance” is one thing from a type of things.
`[1, 2, 3]` is an instance of a list,
1 is an instance of an int

```
    The reverse flag can be set to sort in descending order.
```

Help on built-in function sort:

sort(*, key=None, reverse=False) method of builtins.list instance

Sort the list in ascending order and return None.

The sort is in-place (i.e. the list itself is modified) and stable (i.e. the

The asterisk (somewhat confusingly) says that there are no normal parameters. We can't do `x.sort(42)`, because what would the 42 mean?

The reverse flag can be set to sort in descending order.

```
Help on built-in function sort:

sort(*, key=None, reverse=False) method of builtins.list instance
    Sort the list in ascending order and return None.

    The sort is in-place (i.e. the list itself is modified) and stable (i.e. the
```

But what are these things???

```
    If a key function is given, apply it once to each list item and sort them,
    ascending or descending, according to their function values.

    The reverse flag can be set to sort in descending order.
```

Reverse sort

- What if we want things backwards? `.sort` has a parameter for reversing, but it doesn't look like a normal parameter…
- It's a *named parameter* (or *keyword argument*). To pass it in, you have to name it:

Reverse sort

- What if we want things backwards? `.sort` has a parameter for reversing, but it doesn't look like a normal parameter…
- It's a *named parameter* (or *keyword argument*). To pass it in, you have to name it:

```
x = [2, 4, 6, 0, 1]
x.sort(reverse=True)
print(x) # [6, 4, 2, 1, 0]
```

```
Help on built-in function sort:

sort(*, key=None, reverse=False) method of builtins.list instance
    Sort the list in ascending order and return None.

    The sort is in-place (i.e. the list itself is modified) and stable (i.e. the
    order of two equal elements is maintained).

    If a key function is given, apply it once to each list item and sort them,
    ascending or descending, according to their function values.

    The reverse flag can be set to sort in descending order.
```

OK, we saw “reverse”, but what does this mean?

Key sort

- Sorting normally uses <
- This only works on things you can compare with <, but it's also pretty limited
- What if I wanted to sort strings by their *length*?
- `.sort()` provides a way to change the sorting criteria: a *key function*

Sort strings by length

```
x = ["an", "excellent", "list", "of", "strings"]
x.sort(key=len)
print(x) # ["an", "of", "list", "strings", "excellent"]
```

Sort ints even/odd

```
def parity(val: int) -> int:
    return val%2
lst = [8, 6, 7, 5, 3, 0, 9]
s = sorted(lst, key=parity)
print("After sorted, lst is", lst)
print("Sorted:", s)
lst.sort(key=parity)
print("After .sort, lst is", lst)

# [8, 6, 0, 7, 5, 3, 9]
```

Sort ints even/odd

```
def parity(val: int) -> int:
    return val%2
lst = [8, 6, 7, 5, 3, 0, 9]
s = sorted(lst, key=parity)
```

NOTE: Sorting is “stable”. This means that if two distinct values could go in either order (such as 8 and 0 here) it won’t swap them.

```
print("After sort, lst is", lst)

# [8, 6, 0, 7, 5, 3, 9]
```

There’s much more!

- Let’s use `help` to learn about
  - `list.reverse`
  - `list.index`
  - `str.join`
  - `str.split`
  - `str.find`

Dictionaries

CS114 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 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)
```

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]:
```

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.

```
    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]:
    if (x, y) in memo:
        return memo[(x, y)]
    r = []
```

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

```
            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]:
    if (x, y) in memo:
        return memo[(x, y)]
    r = []
```

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

```
            r.append(i)
        i = i + 1
    memo[(x, y)] = r
    return r
```

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

```python
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
```

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.

```
    return bins
```

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
```

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.

```
        if val >  bin[0] and val < bin[1]:
            return bin
    # Some default if the above somehow fails
    return bin[-1]
```

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

```python
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