PDSP 2024, Lecture 07, 29 August 2024

In [1]:
```
matchlist = [
    ("Chennai","RCB","CSK","RCB","CSK",174),
    ("Mohali","DC","PK","PK","PK",175),
    ("Kolkata","KKR","SRH","SRH","KKR",209),
    ("Jaipur","RR","LSG","RR","RR",194),
    ("Ahmedabad","GT","MI","MI","GT",169),
    ("Bengaluru","PK","RCB","RCB","RCB",177),
    ("Chennai","CSK","GT","GT","CSK",207),
    ("Hyderabad","SRH","MI","MI","SRH",278),
    ("Jaipur","RR","DC","DC","RR",186),
    ("Bengaluru","RCB","KKR","KKR","KKR",183),
    ("Lucknow","LSG","PK","LSG","LSG",200),
    ("Ahmedabad","SRH","GT","SRH","GT",163),
    ("Visakhapatnam","DC","CSK","DC","DC",192),
    ("Mumbai","MI","RR","RR","RR",126),
    ("Bengaluru","LSG","RCB","RCB","LSG",182),
    ("Visakhapatnam","KKR","DC","KKR","KKR",273),
    ("Ahmedabad","GT","PK","PK","PK",200),
    ("Hyderabad","CSK","SRH","SRH","SRH",166),
    ("Jaipur","RCB","RR","RR","RR",184),
    ("Mumbai","MI","DC","DC","MI",235),
    ("Lucknow","LSG","GT","LSG","LSG",164),
    ("Chennai","KKR","CSK","CSK","CSK",138),
    ("Mohali","SRH","PK","PK","SRH",183),
    ("Jaipur","RR","GT","GT","GT",197),
    ("Mumbai","RCB","MI","MI","MI",197),
    ("Lucknow","LSG","DC","LSG","DC",168),
    ("Mohali","PK","RR","RR","RR",148),
    ("Kolkata","LSG","KKR","KKR","KKR",162),
    ("Mumbai","CSK","MI","MI","CSK",207),
    ("Bengaluru","SRH","RCB","RCB","SRH",288),
    ("Kolkata","KKR","RR","RR","RR",224),
    ("Ahmedabad","GT","DC","DC","DC",90),
    ("Mohali","MI","PK","PK","MI",193),
    ("Lucknow","CSK","LSG","LSG","LSG",177),
    ("Delhi","SRH","DC","DC","SRH",267),
    ("Kolkata","KKR","RCB","RCB","KKR",223),
    ("Mohali","PK","GT","PK","GT",143),
    ("Jaipur","MI","RR","MI","RR",180),
    ("Chennai","CSK","LSG","LSG","LSG",211),
    ("Delhi","DC","GT","GT","DC",225),
    ("Hyderabad","RCB","SRH","RCB","RCB",207),
    ("Kolkata","KKR","PK","PK","PK",262),
    ("Delhi","DC","MI","MI","DC",258),
    ("Lucknow","LSG","RR","RR","RR",197),
    ("Ahmedabad","GT","RCB","RCB","RCB",201),
    ("Chennai","CSK","SRH","SRH","CSK",213),
    ("Kolkata","DC","KKR","DC","KKR",154),
    ("Lucknow","MI","LSG","LSG","LSG",145),
    ("Chennai","CSK","PK","PK","PK",163),
    ("Hyderabad","SRH","RR","SRH","SRH",202),
    ("Mumbai","KKR","MI","MI","KKR",170),
    ("Bengaluru","GT","RCB","RCB","RCB",148),
    ("Dharamsala","CSK","PK","PK","CSK",168),
    ("Lucknow","KKR","LSG","LSG","KKR",236),
    ("Mumbai","SRH","MI","MI","MI",174),
    ("Delhi","DC","RR","RR","DC",222),
    ("Hyderabad","LSG","SRH","LSG","SRH",166),
    ("Dharamsala","RCB","PK","PK","RCB",242),
    ("Ahmedabad","GT","CSK","CSK","GT",232),
    ("Kolkata","KKR","MI","MI","KKR",158),
    ("Chennai","RR","CSK","RR","CSK",142),
    ("Bengaluru","RCB","DC","DC","RCB",188),
    ("Delhi","DC","LSG","LSG","DC",209),
    ("Guwahati","RR","PK","RR","PK",145),
    ("Mumbai","LSG","MI","MI","LSG",215),
    ("Bengaluru","RCB","CSK","CSK","RCB",219),
    ("Hyderabad","PK","SRH","PK","SRH",215),
    ("Ahmedabad","SRH","KKR","SRH","KKR",160),
    ("Ahmedabad","RCB","RR","RR","RR",173),
    ("Chennai","SRH","RR","RR","SRH",176),
    ("Chennai","SRH","KKR","SRH","KKR",114)
]
```

Extract unique elements from a list

- Standard loop builds a new list of unique elements
- Check if each element in the original list is already in the new list before adding

In [2]:
```
def uniq(l):
    newl = []
    for x in l:
        if not (x in newl):
            newl.append(x)
    return(newl)
```

Complexity

- Worst case is when original list has no duplicates
- `l[k]` will be compared to `k` elements in `newl` before being added to `newl`
- Takes $1 + 2 + \cdots n - 1$ steps, which is $\frac{n(n-1)}{2}$
  - Proportional to n^2

Using a dictionary

- Cannot have duplicate keys in a dictionary
- Create a dictionary whose keys are values in the original list
  - Value associated with key is not important
  - If we see the same value twice, the key will be updated, not duplicated
- In the end, return the list of keys

In [3]:
```
def uniqd(l):
    newd = {}
    for x in l:
        newd[x] = 1
    return(list(newd))
```

Complexity

- Creating/updating a key in a dictionary takes a fixed amount of time, independent of the size of the dictionary
  - Assuming the hash function works well and there are no (or very few) collisions
- This works effectively in time proportional to n, the length of the list

- We can experimentally verify this by applying both functions to a large list without duplicates
  - In the examples below, we have asked for the length of the list rather than the list itself to avoid large outputs cluttering the page

In [4]:
```
len(uniq(list(range(100000)))) # Takes a long time
```

Out[4]:
```
100000
```

In [5]:
```
len(uniqd(list(range(100000))))  # Almost instantaneous
```

Out[5]:
```
100000
```

Nested collections

- List of lists, list of tuples, dictionary whose values are lists ...
- `matchlist` is a list of tuples
- Use two indices to extract a value
  - `matchlist[3]` is `("Jaipur","RR","LSG","RR","RR",194)`
  - `matchlist[3][1]` is `"RR"`

- List of teams who played IPL 2024

In [6]:
```
def get_teams(l):
    teamlist = []
    for m in l:
        teamlist.append(m[1]) # Add team 1 for match m
        teamlist.append(m[2]) # Add team 2 for match m
    return(uniqd(teamlist))   # Remove duplicates
```

In [7]:
```
get_teams(matchlist)
```

Out[7]:
```
['RCB', 'CSK', 'DC', 'PK', 'KKR', 'SRH', 'RR', 'LSG', 'GT', 'MI']
```

Map

- Apply a function $f()$ to each element in a list

- Convert $[x_0, x_1, \ldots, x_{n-1}]$ to $[f(x_0), f(x_1), \ldots, f(x_{n-1})]$
- In Python, `map(f,l)` applies `f` to each element of `l`

Example

- List full names of teams that played in IPL 2024
- First, a function to map team abbreviations to full names, using a dictionary

In [8]:
```
def expand(s):
    teamdict = {'CSK':'Chennai Super Kings',
                'PK':'Punjab Kings',
                'SRH':'Sunrisers Hyderabad',
                'LSG':'Lucknow Super Giants',
                'MI':'Mumbai Indians',
                'RCB':'Royal Challengers Bengaluru',
                'GT':'Gujarat Titans',
                'DC':'Delhi Capitals',
                'KKR':'Kolata Knight Riders',
                'RR':'Rajasthan Royals'}
    if (s in teamdict):
        return(teamdict[s])
    else:
        return('No info')
```

- Now, we can `map` this function to the outcome of our earlier function

In [9]:
```
teams = get_teams(matchlist)
```

In [10]:
```
map(expand,teams)
```

Out[10]:
```
<map at 0x7fbf3032f1c0>
```

- Output of `map` is a sequence, but not a list, like `range`
- Explicitly convert it to a list

In [11]:
```
list(map(expand,teams))
```

Out[11]:
```
['Royal Challengers Bengaluru',
 'Chennai Super Kings',
 'Delhi Capitals',
 'Punjab Kings',
 'Kolata Knight Riders',
 'Sunrisers Hyderabad',
 'Rajasthan Royals',
 'Lucknow Super Giants',
 'Gujarat Titans',
 'Mumbai Indians']
```

- Since `expand` returns `No info` for unknown keys, the following works
- Note that keys need not be of uniform type: `7` is merely an unknown key, not an invalid one because it is not a string

In [12]:
```
list(map(expand,['xxx','yyy',7]))
```

Out[12]:
```
['No info', 'No info', 'No info']
```

Filter

- Check if each item x in a list satisfies a property $p(x)$
- Retain only such elements
- Filter out elements that do not satisfy $p()$
- In Python, `filter(p,l)`

Example

- List matches where CSK won the toss
- First define the filter function -- returns `True` or `False`

In [13]:
```
def csktosswin(t): # t is expected to be one tuple from matchlist
    return(t[3] == 'CSK')
```

- Now, filter matchlist using this function

In [14]:
```
list(filter(csktosswin,matchlist))
```

Out[14]:
```
[('Chennai', 'KKR', 'CSK', 'CSK', 'CSK', 138),
 ('Ahmedabad', 'GT', 'CSK', 'CSK', 'GT', 232),
 ('Bengaluru', 'RCB', 'CSK', 'CSK', 'RCB', 219)]
```

List comprehension

- Combine map and filter to create a list
- *Set comprehension*: Squares of positive even integers = $\{x^2 \mid x \in \mathbb{Z}, x > 0\}$
- In Python: `[ f(x) for x in l if p(x) ]`

Example

- Full names of all teams in IPL 2024

In [15]:
```
[ expand(t) for t in get_teams(matchlist) ]
```

Out[15]:
```
['Royal Challengers Bengaluru',
 'Chennai Super Kings',
 'Delhi Capitals',
 'Punjab Kings',
 'Kolata Knight Riders',
 'Sunrisers Hyderabad',
 'Rajasthan Royals',
 'Lucknow Super Giants',
 'Gujarat Titans',
 'Mumbai Indians']
```

- List both teams in matches where CSK won the toss

In [16]:
```
[ (t[1],t[2]) for t in matchlist if t[3] == "CSK" ]
```

Out[16]:
```
[('KKR', 'CSK'), ('GT', 'CSK'), ('RCB', 'CSK')]
```

- Same, with full names

In [17]:
```
[ (expand(t[1]),expand(t[2])) for t in matchlist if t[3] == "CSK" ]
```

Out[17]:
```
[('Kolata Knight Riders', 'Chennai Super Kings'),
 ('Gujarat Titans', 'Chennai Super Kings'),
 ('Royal Challengers Bengaluru', 'Chennai Super Kings')]
```

- Similar notation works for dictionaries
- Create a dictionary matching team abbreviations to full names for teams in IPL 2024

In [18]:
```
{ t:expand(t) for t in get_teams(matchlist) }
```

Out[18]:
```
{'RCB': 'Royal Challengers Bengaluru',
 'CSK': 'Chennai Super Kings',
 'DC': 'Delhi Capitals',
 'PK': 'Punjab Kings',
 'KKR': 'Kolata Knight Riders',
 'SRH': 'Sunrisers Hyderabad',
 'RR': 'Rajasthan Royals',
 'LSG': 'Lucknow Super Giants',
 'GT': 'Gujarat Titans',
 'MI': 'Mumbai Indians'}
```

In [19]:
```
{ t:expand(t) for t in get_teams(matchlist) }
```

Out[19]:
```
{'RCB': 'Royal Challengers Bengaluru',
 'CSK': 'Chennai Super Kings',
 'DC': 'Delhi Capitals',
 'PK': 'Punjab Kings',
 'KKR': 'Kolata Knight Riders',
 'SRH': 'Sunrisers Hyderabad',
 'RR': 'Rajasthan Royals',
 'LSG': 'Lucknow Super Giants',
 'GT': 'Gujarat Titans',
 'MI': 'Mumbai Indians'}
```

- Recall that `uniqd` created a list of unique items via keys of a dictionary
- Here is a short way to do this using list comprehension

In [20]:
```
list({ x[2]:1 for x in matchlist})
```

Out[20]:
```
['CSK', 'PK', 'SRH', 'LSG', 'MI', 'RCB', 'GT', 'DC', 'KKR', 'RR']
```

- Uses the fact that a dictionary `d` when interpreted as a sequence is implcitly `d.keys()`
- `list(d)` looks for a sequence `d`

- Can also do the equivalent of relational algebra selection and projection
- Project matchlist onto columns `team 1, team 2, target` (columns 1,2,5) where CSK won the toss
  - Filter by CSK winning the toss (select)
  - Project onto columns 1,2,5

In [21]:
```
[ (t[1],t[2],t[5]) for t in matchlist if t[3] == "CSK" ]
```

Out[21]:
```
[('KKR', 'CSK', 138), ('GT', 'CSK', 232), ('RCB', 'CSK', 219)]
```

Mutable and immutable values

- Lists and dictionaries can be updated in place
  - Can reassign `l[i]` or `d[k]`
  - These are *mutable* values
- Numbers (`int` , `float`), booleans, strings, tuples cannot be updated in place
  - Immutable values

Mutability and assignment

- Assiging a mutable value creates an *alias*
- Updating through either the old or the new name indirectly affects the other

In [22]:
```
l = [1,2,3]
newl = l
newl[0] = 4
```

In [23]:
```
l, newl
```

Out[23]:
```
([4, 2, 3], [4, 2, 3])
```

In [24]:
```
l[1] = 5
```

In [25]:
```
l, newl
```

Out[25]:
```
([4, 5, 3], [4, 5, 3])
```

- For immutable values, assignment behaves as we would expect
- The two names can be updated without affecting each other
  - It is as though assignment copies the value

In [26]:
```
x = 17
y = x
y = 19
```

In [27]:
```
x, y
```

Out[27]:
```
(17, 19)
```

In [28]:
```
x = 18
```

In [29]:
```
x, y
```

Out[29]:
```
(18, 19)
```

- We can update a mutable value inside a function
- However, we should be careful to use updates that do not reassign the name
- Use `l.append(v)` vs `l = l + [v]`

In [30]:
```
def bad(l,v):
    l = l + [v]
    print(l)
    return
```

In [31]:
```
def good(l,v):
    l.append(v)
    print(l)
    return
```

- `bad(l,v)` appends `v` within the function, but creates a new copy of `l` in the process, that is different from the `l` passed as an argument

In [32]:
```
l = [1,2,3]
```

In [33]:
```
bad(l,4)
```

```
[1, 2, 3, 4]
```

`good(l,v)` on the other hand updates `l` in place, so the effect is visible outside

In [34]:
```
l
```

Out[34]:
```
[1, 2, 3]
```

In [35]:
```
good(l,4)
```

```
[1, 2, 3, 4]
```

In [36]:
```
l
```

Out[36]:
```
[1, 2, 3, 4]
```

- We can update `bad(l,v)` to return the modified list, but then we have to reassign `l` to the returned value

In [37]:
```
def bad2(l,v):
    l = l + [v]
    print(l)
    return(l)
```

In [ ]:
```

```

In [38]:
```
l = [1,2,3]
returnlist = bad2(l,4)
```

```
[1, 2, 3, 4]
```

In [39]:
```
l,returnlist
```

Out[39]:
```
([1, 2, 3], [1, 2, 3, 4])
```

In [40]:
```
l = [1,2,3]
l = bad2(l,4)
```

```
[1, 2, 3, 4]
```

In [41]:
```
l
```

Out[41]:
```
[1, 2, 3, 4]
```