Problem 23: How partisan is the US Congress?

Version 1.1b

This problem is about basic data processing using Python. It exercises your fundamental knowledge of Python data structures, such as lists, dictionaries, and strings. It has seven exercises, numbered 0-6.

Each exercise builds on the previous one. However, they may be completed independently. That is, if you can't complete an exercise, we provide some code that can run to load precomputed results for the next exercise. That way, you can keep moving even if you get stuck.

Pro-tips.

- If your program behavior seem strange, try resetting the kernel and rerunning everything.
- If you mess up this notebook or just want to start from scratch, save copies of all your partial responses and use `Actions` → `Reset Assignment` to get a fresh, original copy of this notebook. *(Resetting will wipe out any answers you've written so far, so be sure to stash those somewhere safe if you intend to keep or reuse them!)*
- If you generate excessive output (e.g., from an ill-placed `print` statement) that causes the notebook to load slowly or not at all, use `Actions` → `Clear Notebook Output` to get a clean copy. The clean copy will retain your code but remove any generated output. **However**, it will also **rename** the notebook to `clean.xxx.ipynb`. Since the autograder expects a notebook file with the original name, you'll need to rename the clean notebook accordingly.

Good luck!

Background

The United States Congress is the part of the US government that makes laws for the entire country. It is dominated by two rival political parties, the Democrats and the Republicans. You would expect that these parties oppose each other on most issues but occasionally agree.

Some have conjectured that, over time, the two parties agree less and less, which would reflect a perceived growing ideological or political divide in the US. But is that the real trend? In this problem, you'll explore this question using data collected by ProPublica, a nonprofit investigative media organization.

Setup and data loading

Run the code cells below to load the data. This code will hold the data in two variables, one named `votes` and another named `positions`.

In [ ]:
```
import sys
print(f"* Python version: {sys.version}")

from testing_tools import load_json, save_json, load_pickle, save_pickle

votes = load_json("votes.json")
vote_positions = [p for p in load_json("positions.json") if p['positions']]

print("\n==> Data loading complete.")

###
### AUTOGRADER TEST - DO NOT REMOVE
###
```

Part 0: Analyzing vote results

The Congress votes on various things, like new laws or political nominations. The results of these votes are stored in the `votes` variable loaded above. Let's look at it. First, note that `votes` is a list:

In [ ]:
```
print(type(votes))
print("Length:", len(votes))
```

Vote results. What is `votes` a list of? Each element is one **vote result.** Let's look at the first entry.

In [ ]:
```
from testing_tools import inspect_data

inspect_data(votes[0]) # View the first element of the list, `votes`
```

Observation 0. This first entry of the list is a dictionary, and the data structure is nested even more. For instance, the `"bill"` key has another dictionary as its value.

Remember that what you see above is `votes[0]`, the first entry of the `votes` list. For the rest of this problem, you may assume that all other entries of `votes` have the same keys and nesting structure.

Exercise 0 (2 points). Let's pick out a subset of the data to analyze. Complete the function below, `filter_votes(votes)`.

- The input, `votes`, is a list of vote results like the one above.
- Your function should return a copy of this list, keeping *only* vote results meeting the following criteria:
  1. The `'vote_type'` is one of the following: `"1/2"`, `"YEA-AND-NAY"`, `"RECORDED VOTE"`
  2. Notice that the `'total'` key is a dictionary. Retain only the vote results where this dictionary has *all* of the fields `'yes'`, `'no'`, `'present'`, and `'not_voting'`.

Your copy should include vote results in the same order as the input list.

> *Note 0:* The reason for Condition 2 above is that for some vote results, the `'total'` dictionary does not have those fields. For an example, see `votes[18319]`. You would not include that vote result in your output.
>
> *Note 1:* The test cell does not use real vote results, but randomly generated synthetic ones. Your solution should *not* depend on the presence of any specific keys other than the ones you need for filtering, namely, `'vote_type'` and `'total'`.

As an example, suppose `V` is the following vote results list (only the salient keys are included):

```
V = [ {'vote_type': "1/2", 'total': {'yes': 5, 'no': 8, 'present': 0,
'not_voting': 2}, ...},
      {'vote_type': "RECORDED VOTE", 'total': {'yes': 12, 'present': 2,
'not_voting': 1}, ...},
      {'vote_type': "3/5", 'total': {'yes': 50, 'no': 14, 'present': 0,
'not_voting': 0}, ...},
      {'vote_type': "YEA-AND-NAY", 'total': {'yes': 25, 'no': 3,
'present': 3, 'not_voting': 0}, ...} ]
```

Then running `filter_votes(V)` would return the following new list:

```
[ {'vote_type': "1/2", 'total': {'yes': 5, 'no': 8, 'present': 0,
'not_voting': 2}, ...},
```

```
 {'vote_type': "YEA-AND-NAY", 'total': {'yes': 25, 'no': 3, 'present': 3,
'not_voting': 0}, ...} ]
```

In this case, `V[1]` is omitted because its `'total'` key is missing the `'no'` key; and `V[2]` is omitted because the `'vote_type'` is not one of `"1/2"`, `"YEA-AND-NAY"`, or `"RECORDED VOTE"`.

In [ ]:
```
def filter_votes(votes):
    assert isinstance(votes, list) and len(votes) >= 1
    assert isinstance(votes[0], dict)
    tlist = []
    for v in votes:
        # Check if 'vote_type' is one of the specified values
        valid_vote_types = {"1/2", "YEA-AND-NAY", "RECORDED VOTE"}
        if v['vote_type'] in valid_vote_types:
            # Corrected access to 'total' keys for the current vote dictionary (v)
            if {'yes', 'no', 'present', 'not_voting'} == set(v['total'].keys()):
                tlist.append(v)
    return tlist
```

In [ ]:
```
# Demo cell (feel free to use and edit for debugging)
V = [ {'vote_type': "1/2", 'total': {'yes': 5, 'no': 8, 'present': 0, 'not_voting':
      {'vote_type': "RECORDED VOTE", 'total': {'yes': 12, 'present': 2, 'not_voting
      {'vote_type': "3/5", 'total': {'yes': 50, 'no': 14, 'present': 0, 'not_voting
      {'vote_type': "YEA-AND-NAY", 'total': {'yes': 25, 'no': 3, 'present': 3, 'not
inspect_data(filter_votes(V))

print(len(filter_votes(votes)))
```

In [ ]:
```
# Test cell: ex0__filter_votes (2 points)

###
### AUTOGRADER TEST - DO NOT REMOVE
###

from testing_tools import ex0__check
print("Testing...")
for trial in range(10):
    ex0__check(filter_votes)

print("\n(Passed.)")
```

Precomputed filtered vote results. In case Exercise 0 does not pass, we've precomputed the filtered subset of `votes` we'll need in the remainder of the problem. Whether or not you passed, please run the following cell now to load this result, which will be stored in the variable, `votes_subset`.

In [ ]:
```
votes_subset = load_json("votes_subset.json")
print(len(votes_subset))
```

Observation 1-A: A *passing* vote. Recall the first vote result from above, which is present in `votes_subset` as `votes_subset[0]`. Here is how to interpret it.

In [ ]:
```
inspect_data(votes_subset[0])
```

It concerns a vote on November 19, 1999 (`"date": "1999-11-19"`). There were a total of 74 "yes" votes, 24 "no" votes, and 2 non-votes (abstentions or absences). Since there was a simple majority of "yes" votes---meaning strictly more "yes" votes than "no" votes---the result is considered to be *passing*.

Of the "yes" votes, 32 were cast by Democrats and 42 by Republicans; of the "no" votes, 12 were by Democrats and 12 by Republicans.

Observation 1-B: A *failing* vote. Let's take a look at a vote with a different result, `votes_subset[8]` :

In [ ]:
```
inspect_data(votes_subset[8])
```

This vote took place on November 18, 1999. There were a total of 207+2+0+3 = 212 votes by Democrats and 4+217+0+1 = 222 votes by Republicans. The measure did *not* pass: there were more "no" votes (219) than "yes" votes (212). Of the 219 "no" votes, 217 were cast by Republicans and 2 by Democrats.

Exercise 1 (2 points). Suppose you are given a *single* voting result, `v` (e.g., `v == votes_subset[0]` or `v == votes_subset[8]`). Complete the function `is_passing(v)` so that it returns `True` if the vote "passed" and `False` otherwise.

To determine if a vote is passing or not, check whether the number of `"yes"` votes associated with the `"total"` key is **strictly greater than** the number of `"no"` votes.

> *Note:* The test cell does not use real vote results but rather randomly generated synthetic ones. Your implementation should not depend on the presence of any specific keys other than the ones mentioned in the statement of this exercise.

In [ ]:
```
def is_passing(v):
    return v['total']['yes']>v['total']['no']
```

In [ ]:
```
# Demo cell
print(is_passing(votes_subset[0])) # Should return `True`
print(is_passing(votes_subset[8])) # Should return `False`
```

In [ ]:
```
# Another demo cell
num_passing = sum([is_passing(v) for v in votes_subset])
print(f"Of {len(votes_subset)} vote results, {num_passing} passed.")
```

In [ ]:
```
# Test cell: ex1__is_passing (2 points)

###
### AUTOGRADER TEST - DO NOT REMOVE
```

```
###

from testing_tools import ex1__check
print("Testing...")
for trial in range(1000):
    ex1__check(is_passing)

print("\n(Passed.)")
```

Passing and failing votes. In case your code for Exercise 1 does not pass the test cell, we've precomputed a list of vote results annotated with the result. Whether or not you passed, please run the following code cell, which produces a list of vote results named `votes_pf`, where `votes_pf["passed"]` is `True` if the outcome is a "pass," and `False` otherwise.

In [ ]:
```
votes_pf = load_json('votes_pf.json')
num_passed = sum([1 for v in votes_pf if v["passed"]])
print(f"{num_passed} vote results were passing, {len(votes_pf) - num_passed} were f
```

Definition: The partisan "vote" gap. Given a voting result, let's define a measure of how well the Democrats and Republicans agree.

Suppose a bill has some outcome, either "pass" or "fail." Let d be the proportion of Democrats who voted for that outcome, and let r be the proportion of Republicans who voted for that outcome. Then the *partisan vote gap* for that bill is the absolute difference between d and r, or $|d - r|$. The more Democrats and Republicans agree, the closer this value is to zero. But when they disagree strongly, this value could be 1.

For example, recall that in the first example, `votes_subset[0]`, the bill passed with 74 "yes" votes, 32 from Democrats and 42 from Republicans. Since there were 45 Democrats (32 yes, 12 no, and 1 non-voting), then $d = \frac{32}{45} \approx 0.711$. And since there were 55 Republicans (42 yes, 12 no, and 1 non-voting), then $r = \frac{42}{55} \approx 0.764$. Thus, the partisan vote gap is $|d - r| = \left|\frac{32}{45} - \frac{42}{55}\right| \approx 0.0525$.

In the second example, `votes_subset[8]`, recall that the vote failed with 219 "no" votes, 217 by Republicans out of 222 total, and 2 by Democrats out of 212 total. Thus, $|d - r| = \left|\frac{2}{212} - \frac{217}{222}\right| \approx 0.968$.

Comparing the two cases, the first is an example of reasonable agreement, whereas the second shows strong disagreement.

Exercise 2 (2 points). Given one voting result, `v`, complete the function `calc_partisan_vote_gap(v)` so that it returns the partisan voting gap as defined above. Assume that `v["passed"]` is `True` if the vote was a passing vote (majority "yes"), or `False` otherwise (majority "no").

> *Note 0:* To determine the total number of Democrats or Republicans, add together all of their `"yes"`, `"no"`, `"present"`, and `"not_voting"`

> values using the appropriate party's object in `v`.
>
> *Note 1:* If a vote result has *no* Democrats or Republicans, then use $d = 0$ or $r = 0$, respectively. This scenario can happen if the sum of Democratic or Republican `"yes"`, `"no"`, `"present"`, and `"not_voting"` values is 0.
>
> *Note 2:* You do not need to round your results. The test cell accounts for possible rounding errors when comparing your computed result against the expected one.
>
> *Note 3:* The test cell does not use real vote results, but rather randomly generated synthetic ones. Your code should only depend on the presence of the relevant keys, namely, the party votes (`"democratic"` and `"republican"`) and `"passed"`.

In [ ]:
```
def calc_partisan_vote_gap(v):
    assert isinstance(v, dict)
    assert "passed" in v and "democratic" in v and "republican" in v
    R = sum([num for num in v['republican'].values() if type(num) is int])
    D = sum([num for num in v['democratic'].values() if type(num) is int])
    y_n = 'yes' if v['passed'] else 'no'
    r = v['republican'][y_n]/R if R > 0 else 0
    d = v['democratic'][y_n]/D if D > 0 else 0
    return abs(r-d)
```

In [ ]:
```
# Demo cell to help you debug
print(calc_partisan_vote_gap(votes_pf[0])) # should be about 0.0525
print(calc_partisan_vote_gap(votes_pf[8])) # ~ 0.968
```

In [ ]:
```
# Test cell: ex2__calc_partisan_vote_gap (2 points)

###
### AUTOGRADER TEST - DO NOT REMOVE
###

from testing_tools import ex2__check
print("Testing...")
for trial in range(2500):
    ex2__check(calc_partisan_vote_gap)

print("\n(Passed.)")
```

Precomputed partisan vote gaps. In case your Exercise 2 did not pass the test cell, we've precomputed a list of vote results with their partisan vote gaps. Whether or not you passed, please run the following cell to load this result, which will be stored in the variable, `votes_gap`. Each entry `v` of `votes_gap` will have a key, `v["gap"]`, that holds the vote gap.

In [ ]:
```
votes_gap = load_json("votes_gap.json")
from statistics import mean
```

```
overall_gap = mean([v["gap"] for v in votes_gap])
print(f"Average overall vote gap: {overall_gap}")
```

In [ ]:
```
type(votes_gap)
```

Exercise 3 (2 points): We are now ready to calculate the voting gap over time. Complete the function, `tally_gaps(votes_gap)`, below, where:

- the input `votes_gap` is a list of vote results augmented with the `"gap"` key as defined in Exercise 2;
- the function returns a list of tuples holding the year-by-year average vote gaps, as follows.

For example, suppose you run `gaps_over_time = tally_gaps(votes_gap)` is the output. Then,

- `gaps_over_time` is a list.
- Each element is a pair, `(yyyy, g)`, where `yyyy` is the year represented as an *integer* (`int`) and `g` is the *average* vote gap across all votes that took place in that year.

To determine the year of a vote, recall that each vote result has a `"date"` field. You may assume each vote result `v` in `votes_gap` has a key, `v["gap"]` holding its vote gap.

> *Note:* The test cell does not use real vote results, but rather randomly generated synthetic ones. Your solution should not depend on any particular keys in a vote result other than `"date"` and `"gap"`.

In [ ]:
```
def tally_gaps(votes_gap):
    from collections import defaultdict
    from statistics import mean
    year_gaps = defaultdict(list)
    for v in votes_gap:
        year = int(v['date'][:4])
        year_gaps[year].append(v['gap'])
    return [(y, mean(g)) for y,g in year_gaps.items()]
```

In [ ]:
```
# Demo cell you can use for debugging
gaps_over_time = tally_gaps(votes_gap)

for yyyy, g_avg in sorted(gaps_over_time, key=lambda x: x[0]):
    print(f"{yyyy}: {g_avg:.3f}")
```

In [ ]:
```
# Test cell: ex3__tally_gaps (2 points)

from testing_tools import ex3__check
print("Testing...")
for trial in range(100):
    ex3__check(tally_gaps)

print("\n(Passed.)")
```

Gaps over time. If your demo worked correctly, you should have seen a steady trend in which the vote gap increases over time, starting at around 0.41 in 1991 and increasing to 0.76 in 2020. That is one quantitative indicator of growing partisanship in the US Congress.

Part 1: Finding "compatible" lawmakers from opposing parties

Are there any pairs of lawmakers from opposing parties---that is, one Democrat and one Republican---who tend to vote similarly to one another? Perhaps such pairs can help bridge the divide between the two parties.

To help find such a pair, here is one final dataset, stored in the list, `vote_positions`. It consists of vote results from the last two years (2019-2020) along with *how each lawmaker voted* (yes or no). There are 278 such vote results available, which we can confirm by printing the length of the `vote_positions` list:

In [ ]:
```
print(len(vote_positions))
```

Let's inspect one of these vote results, `vote_positions[0]`:

In [ ]:
```
inspect_data(vote_positions[0])
```

It is a vote result with some more keys. The most important new key is `'positions'`. Its value is a list of everyone and what vote they cast. For example, for the vote result stored in `vote_positions[0]`, above, you can verify that

- `"Lamar Alexander"` is a Republican who voted yes (`'name'`: `"Lamar Alexander"`, `'party': "R"`, `'vote_position': "Yes"`); and
- `"Tammy Baldwin"` is a Democrat who voted no (`'name'`: `"Tammy Baldwin"`, `'party': "D"`, `'vote_position': "No"`).

> *Note:* The `'vote_position'` key can take on a third value, which is `"Not Voting"`. In this example, that is the case for `"Cindy Hyde-Smith"`.

Exercise 4 (2 points). Complete the function, `get_parties(vps)`, below. It should take as input a vote results list `vps` similar to `vote_positions` above. It should return a *dictionary,* where each key is the name of a lawmaker and the corresponding value is that person's party, taken from the `'party'` key.

For example, suppose you run `parties = get_parties(vote_positions)`. Then,

- `parties["Lamar Alexander"] == "R"`
- `parties["Tammy Baldwin"] == "D"`
- `parties["Angus King"] == "ID"`

> *Note 0:* You may assume that the party of a member does not change and that member names are unique.
>
> *Note 1:* You may assume there are only three possible party values in the data: `"R"` for Republican, `"D"` for Democrat, and `"ID"` for Independent.
>
> *Note 2:* Do *not* assume that every name appears in every vote result. Therefore, to ensure you get all the names, your solution needs to sweep all vote results. The test code may check whether you've done so.
>
> *Note 3:* The test cell does not use real data; rather, it uses randomly generated synthetic data. Your solution should not depend on the existence of any keys other than `'positions'` and, for each element of positions, `'name'` and `'party'`.

In [ ]:
```
def get_parties(vps):
    return {d['name']: d['party'] for vp in vps for d in vp['positions']}
```

In [ ]:
```
# Demo cell you can use for debugging
parties_demo = get_parties(vote_positions)
print(parties_demo["Lamar Alexander"]) # "R"
print(parties_demo["Tammy Baldwin"]) # "D"
print(parties_demo["Angus King"]) # "ID"
```

In [ ]:
```
# Test cell: ex4__get_parties (2 points)

###
### AUTOGRADER TEST - DO NOT REMOVE
###

from testing_tools import ex4__check
print("Testing...")
for trial in range(1000):
    ex4__check(get_parties)

print("\n(Passed.)")
```

Precomputed parties. Whether your solution to the previous exercise passes or not, we have precomputed the party affiliations of the members for you. Run the following code cell, which defines a dictionary named `parties` holding these affiliations. You'll need `parties` in a subsequent exercise.

In [ ]:
```
parties = load_pickle('parties.pickle')
for member in ["Lamar Alexander", "Tammy Baldwin", "Angus King"]:
    print(member, "=>", parties[member])
```

Voting vectors. To help us compare voting records between lawmakers, let's construct their *voting vectors*.

Each lawmaker will be represented by one voting vector. A voting vector is a Python *list*. Each element of the list is for a voting result and holds one of three values: `True` for a `"Yes"` vote, `False` for a `"No"` vote, and `None` for `"Not Voting"`.

For example, since there are 278 vote results in this `vote_positions` list, the length of a voting vector will be 278. Now suppose Lamar Alexander's voting vector is `x`, Tammy Baldwin's is `y`, and Cindy Hyde-Smith's is `z`. Then for the example voting position above, we would have `x[0] == True`, `y[0] == False`, and `z[0] == None`.

Exercise 5 (2 points). Complete the function, `build_voting_vectors(vps)`. It should take as input a vote results list `vps` like `vote_positions` above. It should return a *dictionary* where each key is the name of a lawmaker and the corresponding value is a voting vector. Recall that a voting vector is a Python list of `True` or `False` values.

For example, suppose you run `results = build_voting_vectors(vote_positions)`. Then, from the example of `vote_positions[0]`, we would expect the following:

- `results["Lamar Alexander"][0] == True`
- `results["Tammy Baldwin"][0] == False`
- `results["Cindy Hyde-Smith"][0] == None`

> *Note 0:* The test cell does not use real vote results, but rather randomly generated synthetic ones. Your code should not assume the existence of keys other than the ones directly relevant to this exercise's problem statement, namely, `'positions'`, and for each lawmaker's vote, `'name'` and `'vote_position'`.
>
> *Note 1:* Although the sample voting dataset has 278 vote results, each synthetic test cases might have a different number.
>
> *Note 2:* Your code should not assume that every lawmaker voted in every vote result. See `Demo cell 0`, below, where `"Anita Borg"` did not vote in the first vote result, and in one instance, `"Alan Turing"` is marked as `"Not Voting"`.

In [ ]:
```
def build_voting_vectors(vps):
    from collections import defaultdict
    result = defaultdict(lambda: [None]*len(vps))
    vote_map = {'Yes': True, 'No': False}
    for i, vote in enumerate(vps):
        for p in vote['positions']:
            name = p['name']
            v = p['vote_position']
            result[name][i] = vote_map.get(v,None)
    return result
```

In [ ]:
```
# Demo cell 0, which you can use for debugging

# Example with three vote results and three different lawmakers:
vps_example = [{'positions': [{'name': 'Grace Hopper', 'vote_position': 'Yes'},
                              {'name': 'Alan Turing', 'vote_position': 'No'}]},
               {'positions': [{'name': 'Anita Borg', 'vote_position': 'Yes'},
                              {'name': 'Alan Turing', 'vote_position': 'Not Voting'
                              {'name': 'Grace Hopper', 'vote_position': 'No'}]},
               {'positions': [{'name': 'Grace Hopper', 'vote_position': 'No'},
                              {'name': 'Anita Borg', 'vote_position': 'Yes'},
                              {'name': 'Alan Turing', 'vote_position': 'Yes'}]}
              ]

build_voting_vectors(vps_example)

# Should print something like
#     {'Grace Hopper': [True, False, False],
#      'Alan Turing': [False, None, True],
#      'Anita Borg': [None, True, True]}
```

In [ ]:
```
# Demo cell 1
results = build_voting_vectors(vote_positions)

# Lamar Alexander cast 236 "Yes" votes, 7 "No" votes, and did not vote 35 times.
# The collections.Counter object can help verify that fact from your results.
Lamars_votes = results["Lamar Alexander"]
from collections import Counter
Counter(Lamars_votes)
```

In [ ]:
```
# Test cell: ex5__build_voting_vectors (2 points)

###
### AUTOGRADER TEST - DO NOT REMOVE
###

from testing_tools import ex5__check
for trial in range(1000):
    ex5__check(build_voting_vectors)

print("\n(Passed.)")
```

Precomputed voting vectors. Whether or not your previous solution passed, we've precomputed the voting vectors for this dataset. The following code cell loads these results into the object `voting_vectors`, which you'll need in subsequent exercises.

In [ ]:
```
voting_vectors = load_pickle('voting_vectors.pickle')
from collections import Counter
for example in ["Lamar Alexander", "Tammy Baldwin"]:
    example_vv = voting_vectors[example]
    print(example, "==>", Counter(example_vv))
```

Exercise 6 (2 points). For this last exercise, complete the function `opposing_pairs(voting_vectors, parties)`. This function takes two inputs:

- `voting_vectors` is the dictionary that maps each member of Congress to their voting vector
- `parties` is the dictonary that holds the party affiliation of each member

It should then do the following:

- Consider every pair of opposing members. That is, let `(a, b)` be a pair of names, where `a` is some Republican and `b` is some Democrat. (`a` must be a Republican and `b` a Democrat; do not flip these.) Ignore any Independents.
- Use their voting vectors to count how many times `a` and `b` cast *identical* and *non-`None`* votes. That is, if they both voted `"Yes"` or they both voted `"No"`, those are identical votes. However, if either or both has `None` recorded for a vote, that should *not* be counted.
- Denote the count between `a` and `b` by `c`. Then, the function should return a list of all tuples, `(a, b, c)`.

For example, suppose you run the following:

```
P = {"alice": "R", "bob": "D", "carol": "D", "dale": "ID"}
V = {"alice": [True, True, False, True, None],
     "bob": [True, False, None, True, True],
     "carol": [False, False, False, None, None],
     "dale": [False, None, None, False, False]}
pairs = opposing_pairs(V, P)
```

Then `pairs` should be the following list:

```
[('alice', 'bob', 2), ('alice', 'carol', 1)]
```

For instance, the result contains `('alice', 'bob', 2)`, since `'alice'` is a Republican, `'bob'` is a Democrat, and they cast the same vote twice (entries 0 and 3 of their voting vectors). It also contains `('alice', 'carol', 1)`, since `'alice'` is a Republican and `'carol'` is a Democrat, and they cast the same non-`None` votes just once.

> *Note:* Your function does not have to return the list with entries in the same *order* as above; any permutation is fine, as long as it contains all valid pairs.

In [ ]:
```
def opposing_pairs(voting_vectors, parties):
    def count_common(x,y):
        return sum([(xi==yi) for xi,yi in zip(x,y) if (xi is not None) and (yi is n
    R = {name for name, party in parties.items() if party == 'R'}
    D = {name for name, party in parties.items() if party == 'D'}
    pairs = []
    for r in R:
        for d in D:
            r_vec = voting_vectors[r]
            d_vec = voting_vectors[d]
            c = count_common(r_vec, d_vec)
            pairs.append((r,d,c))
    return pairs
```

In [ ]:
```
# Demo cell
P = {"alice": "R", "bob": "D", "carol": "D", "dale": "ID"}
V = {"alice": [True, True, False, True, None],
     "bob": [True, False, None, True, True],
     "carol": [False, False, False, None, None],
     "dale": [False, None, None, False, False]}
pairs = opposing_pairs(V, P)
pairs
```

In [ ]:
```
# Test cell: ex6__opposing_pairs (2 points)

from testing_tools import ex6__check
for trial in range(1000):
    ex6__check(opposing_pairs)

print("\n(Passed.)")
```

Most and least "compatible" opposing pairs. If your `opposing_pair` implementation works, then the following code should print the "most compatible" opposing pairs. (If it doesn't work, that's okay -- the cell below will fail, but it shouldn't affect your final score.)

In [ ]:
```
congress_pairs = opposing_pairs(voting_vectors, parties)
print("Top opposing pairs:")
sorted(congress_pairs, key=lambda x: x[-1], reverse=True)[:11]
```

In [ ]:
```
print("Most polar (least compatible) opposing pairs:")
sorted(congress_pairs, key=lambda x: x[-1], reverse=True)[-11:]
```

Epilogue. The analysis contained herein is just the tip of the iceberg of what you could do with this data, as well as all the other data available through ProPublica's Congress APIs. If you live in the US and are civic-minded, consider mining it for even more interesting information!

Fin! You've reached the end of Midterm 1. Don't forget to restart and run all cells again to make sure it's all working when run in sequence; and make sure your work passes the submission process. Good luck!