```
import json
def load_file(filename):
    with open(filename,'r') as f:
        data=json.load(f)
    return data
def find_people_you_may_know(user_id,data):
    user_friends={}
    for user in data['users']:
        #for given user id find friends
        user_friends[user['id']]=set(user['friends'])

    #if user for which we are finding friends doesnot exist
    if user_id not in user_friends:
        return []

    #direct friend of user id
    direct_friends=user_friends[user_id]

    suggestions={}
    #take direct  friends  suppose 1 is friend with 2 and 3
    for friend in direct_friends:
        # now loop on 2 and 3 friends
        for mutual in user_friends[friend]:
            # suppose 2 is friend with 1 and 4 so it should
ignore 1 as well as if 2 is friend with 3 as well and 1 is
already friend with 3 so it should also ignore that
            if mutual !=user_id and mutual not in
direct_friends:
                #count mutual friends

suggestions[mutual]=suggestions.get(mutual,0)+1

sorted_suggestions=sorted(suggestions.items(),key=lambd
a x:x[1],reverse=True)
    return [user_id for user_id,mutual_count in
sorted_suggestions]
```

```
#load the data
data= load_file("massive_data.json")
user_id=1
recc=find_people_you_may_know(user_id,data)
print(recc)
```

i want comments in this code especially in find_people you_may_know in each line and also explain me with a example in

Sure! Here's your code with **clear comments on every important line**, especially inside `find_people_you_may_know()`, and **an example explanation at the end**.

✅ Fully Commented Code (Line-by-Line Explanation)

python

Copy code

```python
import json

# Function to load JSON file
def load_file(filename):
    with open(filename, 'r') as f:
        data = json.load(f)
    return data



def find_people_you_may_know(user_id, data):
    # Dictionary to store each user's friends as a set for fast lookup
    user_friends = {}

    # Loop over all users in the dataset
    for user in data['users']:
        # Store: user_id -> set of their friends
        user_friends[user['id']] = set(user['friends'])

    # If the given user_id does not exist in dataset, return empty list
    if user_id not in user_friends:
        return []

    # Get the direct friends of the given user
    direct_friends = user_friends[user_id]

    # Dictionary to count suggestions based on mutual friends
    suggestions = {}

    # Loop through each direct friend of the user
    for friend in direct_friends:

        # Loop through the friends of this friend → potential mutual connections
        for mutual in user_friends[friend]:

            # Ignore:
            # 1. The user itself
            # 2. People who are already direct friends
            if mutual != user_id and mutual not in direct_friends:

                # Count how many mutual connections each suggestion has
                suggestions[mutual] = suggestions.get(mutual, 0) + 1

    # Sort suggestions by highest mutual friend count
    sorted_suggestions = sorted(suggestions.items(), key=lambda x: x[1], reverse=

    # Return only the user IDs in sorted order
    return [user_id for user_id, mutual_count in sorted_suggestions]


# Load the data
data = load_file("massive_data.json")
user_id = 1
recc = find_people_you_may_know(user_id, data)
print(recc)
```

✅ Example to Understand the Logic

Suppose your dataset looks like this:

```json
{
    "users": [
        {"id": 1, "friends": [2, 3]},
        {"id": 2, "friends": [1, 4]},
        {"id": 3, "friends": [1, 4, 5]},
        {"id": 4, "friends": [2, 3]},
        {"id": 5, "friends": [3]}
    ]
}
```

🎯 Goal: find_people_you_may_know(1)

User **1** has friends:

→ `{2, 3}`

✔️ Step 1: Look at friends of friends

Friend 2's friends: `{1,4}`

- Ignore 1 (user itself)
- 4 is not friend of 1 → candidate suggestion

➡️ Mutual count for 4 → **1**

Friend 3's friends: `{1,4,5}`

- Ignore 1
- 4 (again) → increase count
- 5 is new → add with count 1

➡️ 4 → **2 mutual friends**

➡️ 5 → **1 mutual friend**

✔ Step 2: Sort by count (descending)

- 4 → 2
- 5 → 1

🎉 Final Recommendation Output

python Copy code

```python
[4, 5]
```

This means:

- **User 4** is the strongest recommendation → 2 mutual friends
- **User 5** is the next → 1 mutual friend

If you want, I can also:

✅ Convert this into a class
✅ Add docstrings
✅ Improve performance for huge datasets
✅ Make it return names instead of IDs

Just tell me!

↳ ✅ **Example to Understand the Logic** ==================================== Suppose your dataset looks like this: json Copy code `{ "users": [{"id": 1, "friends": [2, 3]}, {"id": 2, "friends": [1, 4]}, {"id": 3, "friends": [1…

explain this explanination code wise

Below is the **same explanation but written purely in code-wise thinking**, showing **exactly what happens at each step in the program**.

I will walk through the function **line by line with the sample data**, so you can see how values change inside the code.

✅ Sample Data

```python
data = {
    "users": [
        {"id": 1, "friends": [2, 3]},
        {"id": 2, "friends": [1, 4]},
        {"id": 3, "friends": [1, 4, 5]},
        {"id": 4, "friends": [2, 3]},
        {"id": 5, "friends": [3]}
    ]
}
```

✅ Now Run: `find_people_you_may_know(1, data)`

✔️ Step 1: Build `user_friends` dictionary

Code:

```python
for user in data['users']:
    user_friends[user['id']] = set(user['friends'])
```

What is happening?

Iteration	user	user_friends after
1	{"id":1, "friends":[2,3]}	`{1: {2,3}}`
2	{"id":2, "friends":[1,4]}	`{1:{2,3}, 2:{1,4}}`

Iteration	user	user_friends after
3	{"id":3, "friends":[1,4,5]}	`{1:{2,3}, 2:{1,4}, 3:{1,4,5}}`
4	{"id":4, "friends":[2,3]}	`{1:{2,3},2:{1,4},3:{1,4,5},4:{2,3}}`
5	{"id":5, "friends":[3]}	`{1:{2,3},2:{1,4},3:{1,4,5},4:{2,3},5:{3}}`

✔️ Step 2: Get direct friends of user 1

Code:

```python
direct_friends = user_friends[user_id]
```

Result:

```ini
direct_friends = {2, 3}
```

✔️ Step 3: Start scanning friends of your friends

```python
for friend in direct_friends:
```

This means we loop over:

```ini
friend = 2
friend = 3
```

💡 WHEN friend = 2

Code:

```python
for mutual in user_friends[2]:   # user_friends[2] → {1,4}
```

So the loop runs on:

```ini
mutual = 1
mutual = 4
```

1️⃣ mutual = 1

```python
if mutual != 1 and mutual not in {2,3}:
```

- mutual == 1 → SAME AS USER → ❌ skip

2️⃣ mutual = 4

```python
if 4 != 1 and 4 not in {2,3}:
```

✔️ valid → add suggestion

```python
suggestions[4] = suggestions.get(4,0) + 1
```

Now:

```ini
suggestions = {4: 1}
```

💡 WHEN friend = 3

Code:

```python
for mutual in user_friends[3]:   # {1, 4, 5}
```

Loop values:

```ini
mutual = 1
mutual = 4
mutual = 5
```

1️⃣ mutual = 1

Skip again (same user)

2️⃣ mutual = 4

Check:

```python
if 4 != 1 and 4 not in {2,3}:
```

✔️ valid again → increase count

```python
suggestions[4] = suggestions.get(4,0) + 1
```

Now:

```ini
suggestions = {4: 2}
```

3️⃣ mutual = 5

Check:

```python
if 5 != 1 and 5 not in {2,3}:
```

✔️ valid → add it

```python
suggestions[5] = 1
```

Now:

```ini
suggestions = {4: 2, 5: 1}
```

✔️ Step 4: Sort suggestions by mutual friend count

Code:

```python
sorted_suggestions = sorted(suggestions.items(), key=lambda x: x[1], reverse=True
```

Suggestions dictionary:

```css
{4:2, 5:1}
```

Sorted list:

```css
[(4, 2), (5, 1)]
```

✔️ Step 5: Return only user IDs

Code:

```python
return [user_id for user_id, mutual_count in sorted_suggestions]
```

Gives:

```csharp
[4, 5]
```

🎉 FINAL OUTPUT

```csharp
[4, 5]
```

👍 Summary in plain code logic

- User 1 has friends 2 and 3
- User 2 brings friend 4
- User 3 brings friends 4 and 5
- Count mutuals → 4 has 2, 5 has 1
- Sort → `[4, 5]`

If you want, I can also show this with a diagram or flowchart.

↳ ### Code: python Copy code `sorted_suggestions = sorted(suggestions.items(), key=lambda x: x[1], r`

explain this line of code

Here is a **very clear and simple explanation** of this line of code:

python Copy code

```python
sorted_suggestions = sorted(suggestions.items(), key=lambda x: x[1], reverse=True
```

I will break it into **small parts** so you understand exactly what happens.

✅ 1. `suggestions.items()`

If `suggestions` looks like:

python Copy code

```python
suggestions = {4: 2, 5: 1}
```

Then:

python Copy code

```python
suggestions.items()
```

becomes:

css Copy code

```css
[(4, 2), (5, 1)]
```

Each element is a **tuple**:

- `x[0]` → user ID
- `x[1]` → mutual friend count

✅ 2. `key=lambda x: x[1]`

This tells Python **how to sort** the items.

- `x` represents each tuple → e.g., `(4, 2)`
- `x[1]` means → sort by the **second value**, i.e., mutual friend count

So:

- For `(4,2)` → key = 2
- For `(5,1)` → key = 1

Python sorts using these numbers.

✅ 3. `reverse=True`

This means:

- Sort from **biggest to smallest**
  (Descending order)

So the result will be:

```css
[(4, 2), (5, 1)]
```

because **2** > **1**.

🎉 Final Meaning (Easy Version)

👉 **Sort the suggestions by highest mutual friends.**

That's all.

✔️ Visual Breakdown

Input dictionary:

```css
```

```
{4:2, 5:1}
```

Convert to list:

```css
[(4,2), (5,1)]
```

Sort by second value (`x[1]`):

- 2 (bigger) comes first
- 1 comes later

Final:

```css
[(4,2), (5,1)]
```

👍 One-Line Summary

This line sorts the suggestions so the user with the most mutual friends appears first.

If you want, I can also rewrite this line in a simpler way or without lambda.

↪ return \[user_id for user_id, mutual_count in sorted_suggestion

please explain this as well

Here is a **very simple explanation** of this line:

```python
return [user_id for user_id, mutual_count in sorted_suggestions]
```

We'll break it into **small, easy parts**.

✅ 1. What is inside `sorted_suggestions`?

From earlier, after sorting, you have something like:

```python
sorted_suggestions = [(4, 2), (5, 1)]
```

Each tuple means:

- `4` → suggested person
- `2` → number of mutual friends

✅ 2. What this loop is doing

```python
for user_id, mutual_count in sorted_suggestions
```

This means:

- First loop:

  `user_id = 4`

  `mutual_count = 2`
- Second loop:

  `user_id = 5`

  `mutual_count = 1`

It is **pulling out values** from each tuple.

✅ 3. What we keep and what we ignore

```python
[user_id for user_id, mutual_count in sorted_suggestions]
```

This means:

- Keep only `user_id`
- Ignore `mutual_count`

So from:

```scss
(4, 2) → keep 4
(5, 1) → keep 5
```

🎉 Final Output

After the list comprehension runs, the result is:

```python
[4, 5]
```

Which means:

➡️ These are the people you may know
➡️ Already sorted by mutual friends

📌 Simple explanation in one sentence

This line extracts only the user IDs from the sorted suggestions list and returns them as a clean list like `[4, 5]`.

If you want, I can also rewrite it in a longer (but simple) version without list comprehension.

return [(user_id),(mutual_count) for user_id, mutual_count in sorted_suggestions] what is thhe problem

Your code is incorrect because of **where you placed the parentheses**.

You wrote:

python Copy code

```python
return [(user_id), (mutual_count) for user_id, mutual_count in sorted_suggestions
```

This is **invalid Python syntax**.

❌ Why is it wrong?

In a list comprehension, the format must be:

css Copy code

```css
[ expression  for variables in iterable ]
```

But your code looks like this:

rust Copy code

```rust
[(user_id), (mutual_count)   for user_id, mutual_count in ...]
      ↑             ↑
  two expressions → Python doesn't know what to do
```

Python sees **two separate expressions** `(user_id)` AND `(mutual_count)` before the `for` loop, which is not allowed.

✅ Correct Version (You want a tuple)

If you want each element to be:

```scss
(user_id, mutual_count)
```

Then write:

```python
return [(user_id, mutual_count) for user_id, mutual_count in sorted_suggestions]
```

✔ This is valid

✔ This returns a list of tuples like:

```css
[(4, 2), (5, 1)]
```

✅ Correct Version (If you want a list inside a list)

```python
return [[user_id, mutual_count] for user_id, mutual_count in sorted_suggestions]
```

⭐ Summary

Your code	Problem	Fix
`[(user_id),(mutual_count) for ...]`	Two expressions before `for` → invalid	`[(user_id, mutual_count) for ...]`

If you want, I can explain WHY Python doesn't allow multiple expressions before `for`.