🔢 Rounding in Python: `round()` vs Custom Rounding Functions

Python's built-in `round()` function typically works well for general use. However, it may not always behave as expected due to its use of **banker's rounding** (also called **round half to even**). This can surprise developers in certain use cases — especially in finance or precise measurement.

🧠 Default Behavior of `round()`

```
round(2.5)  # Output: 2
round(3.5)  # Output: 4
```

Why this happens:

- Python uses **round half to even**:
  - If a number is exactly halfway between two integers (e.g., 2.5), it rounds to the **nearest even number**.
  - This reduces **rounding bias** over large datasets, which is useful in **statistics** and **finance**.

❓ When Should You Define Your Own Rounding Function?

You should write your own rounding function when:

- ✅ You need consistent behavior for `.5` values (e.g., **always round up** or **always round down**).
- ✅ Your application is sensitive to rounding errors:
  - **Financial calculations**
  - **Precise engineering measurements**
- ✅ You want to match **common rounding expectations** ("round half up" like on paper).

✅ Custom "Round Half Up" Function

```
import math


def round_half_up(n):
    if n >= 0:
        return math.floor(n + 0.5)
    else:
        return math.ceil(n - 0.5)
```

🔍 How It Works (Step-by-Step)

🔸 **For Positive Numbers**

- `math.floor(n + 0.5)`
- Pushes numbers with fractional parts ≥ 0.5 just past the next integer:
  - `2.5 + 0.5 = 3.0 → floor(3.0) = 3`
  - `2.4 + 0.5 = 2.9 → floor(2.9) = 2`

🔸 **For Negative Numbers**

- `math.ceil(n - 0.5)`
- Subtracting 0.5 pushes the number further into the negative:
  - `-1.5 - 0.5 = -2.0 → ceil(-2.0) = -2`
  - `-2.5 - 0.5 = -3.0 → ceil(-3.0) = -3`

✅ Output Examples

```
print(round_half_up(2.5))   # 3
print(round_half_up(3.5))   # 4
print(round_half_up(1.4))   # 1
print(round_half_up(-1.5))  # -2
print(round_half_up(-2.5))  # -3
```

⚠️ Notes About Floor Division (`//`)

- `//` performs **floor division**, which always rounds **towards negative infinity**:

  ```
  5 // 2    # 2
  -5 // 2   # -3 (not -2)
  ```

- It's **not suitable** for general-purpose rounding — especially with **negative numbers**, due to asymmetric behavior.

📝 TL;DR

Use Case	Recommended Approach
General-purpose rounding	Use Python's built-in `round()`
Consistent rounding of `.5` up	Use a custom `round_half_up()`
Financial/precise applications	Use a custom approach to avoid bias
Working with negative numbers	Handle them explicitly (e.g., `ceil(n - 0.5)`)
