How to Round Numbers in Python

by David Amos
Dec 07, 2024 52m 24 Comments intermediate best-practices python

Mark as Completed Share

Table of Contents

Help

Watch Now This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: **Rounding Numbers in Python**

Rounding numbers in Python is an essential task, especially when dealing with data precision. Python's built-in `round()` function uses the rounding half to even strategy, which rounds numbers like `2.5` to `2` and `3.5` to `4`. This method helps minimize rounding bias in datasets. To round numbers to specific decimal places, you can use the `round()` function with a second argument specifying the number of decimals.

For more advanced rounding strategies, you can explore Python's `decimal` module or use NumPy and pandas for data science applications. NumPy arrays and pandas DataFrames offer methods for rounding numbers efficiently. In NumPy, you can use functions like `np.round()`, `np.ceil()`, `np.floor()`, and `np.trunc()` to apply different rounding strategies. For pandas, the `df.round()` method allows rounding of entire DataFrames or specific columns.

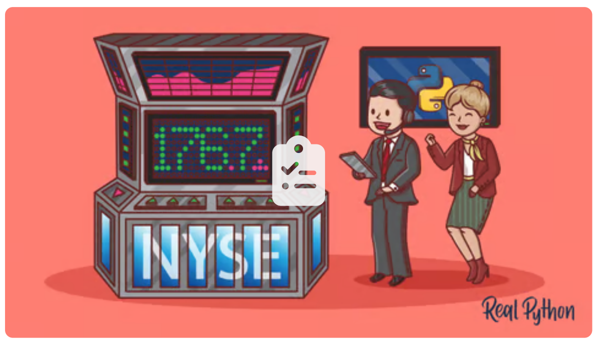
By the end of this tutorial, you'll understand that:

- Python uses the **rounding half to even** strategy, where ties round to the nearest even number.
- Python's default rounding strategy minimizes **rounding bias** in large datasets.
- You can round numbers to **specific decimal places** using Python's `round()` function with a second argument.
- Different **rounding strategies** can be applied using Python's `decimal` module or custom functions for precision control.
- **NumPy** and **pandas** provide methods for rounding numbers in **arrays** and **DataFrames**, offering flexibility in data manipulation.

You won't get a treatise on numeric precision in computing, although you'll touch briefly on the subject. Only a familiarity with the fundamentals of Python is necessary, and the math should feel familiar if you've had high school algebra.

You'll start by looking at Python's built-in rounding mechanism.

Take the Quiz: Test your knowledge with our interactive "Rounding Numbers in Python" quiz. You'll receive a score upon completion to help you track your learning progress:



Interactive Quiz

Rounding Numbers in Python

Test your knowledge of rounding numbers in Python.

Get Your Code: Click here to download the free sample code you'll use to learn about rounding numbers in Python.

Python's Built-in `round()` Function

Python has a built-in `round()` function that takes two numeric arguments, `n` and `ndigits`, and returns the number `n` rounded to `ndigits`. The `ndigits` argument defaults to zero, so leaving it out results in a number rounded to an integer. As you'll see, `round()` may not work quite as you expect.

The way most people are taught to round a number goes something like this:

- Round the number `n` to `p` decimal places by first shifting the decimal point in `n` by `p` places. To do that, multiply `n` by 10^p (10 raised to the `p` power) to get a new number, `m`.

- Then look at the digit `d` in the first decimal place of `m`. If `d` is less than 5, round `m` down to the nearest integer. Otherwise, round `m` up.

- Finally, shift the decimal point back `p` places by dividing `m` by 10^p.

It's an algorithm! For example, the number `2.5` rounded to the nearest whole number is `3`. The number `1.64` rounded to one decimal place is `1.6`.

Now open up an interpreter session and round `2.5` to the nearest whole number using Python's built-in `round()` function:

Python

```
>>> round(2.5)
2
```

Gasp!

Check out how `round()` handles the number `1.5`:

Python

```
>>> round(1.5)
2
```

So, `round()` rounds `1.5` up to `2`, and `2.5` down to `2`!

Before you go raising an issue on the Python bug tracker, rest assured you that `round(2.5)` is *supposed* to return `2`. There's a good reason why `round()` behaves the way it does.

In this tutorial, you'll learn that there are more ways to round a number than you might expect, each with unique advantages and disadvantages. `round()` behaves according to a particular rounding strategy—which may or may not be the one you need for a given situation.

You might be wondering, *Can the way I round numbers really have that much of an impact?* Next up, take a look at just how extreme the effects of rounding can be.

How Much Impact Can Rounding Have?

Suppose you have an incredibly lucky day and find $100 on the ground. Rather than spending all your money at once, you decide to play it smart and invest your money by buying some shares of different stocks.

The value of a stock depends on supply and demand. The more people there are who want to buy a stock, the more value that stock has, and vice versa. In high-volume stock markets, the value of a particular stock can fluctuate on a second-by-second basis.

Now run a little experiment. Pretend the overall value of the stocks that you purchased fluctuates by some small random number each second, say between $0.05 and -$0.05. This fluctuation may not necessarily be a nice value with only two decimal places. For example, the overall value may increase by $0.031286 one second and decrease the next second by $0.028476.

You don't want to keep track of your value to the fifth or sixth decimal place, so you decide to chop everything off after the third decimal place. In rounding jargon, this is called **truncating** the number to the third decimal place. There's some error to be expected here, but by keeping three decimal places, this error couldn't be substantial. Right?

To run your experiment using Python, you start by writing a `truncate()` function that truncates a number to three decimal places:

Python

```
>>> def truncate(n):
...     return int(n * 1000) / 1000
...
```

The `truncate()` function first shifts the decimal point in the number `n` three places to the right by multiplying `n` by `1000`. You get the integer part of this new number with `int()`. Finally, you shift the decimal point three places back to the left by dividing `n` by `1000`.

Next, you define the initial parameters of the simulation. You'll need two variables: one to keep track of the actual value of your stocks after the simulation is complete and one for the value of your stocks after you've been truncating to three decimal places at each step.

Start by initializing these variables to `100`:

Python

```
>>> actual_value, truncated_value = 100, 100
```

Now run the simulation for 1,000,000 seconds (approximately 11.5 days). For each second, generate a random value between `-0.05` and `0.05` with the `uniform()` function in the `random` module, and then update `actual` and `truncated`:

Python

```python
>>> import random
>>> random.seed(100)

>>> for _ in range(1_000_000):
...     delta = random.uniform(-0.05, 0.05)
...     actual_value = actual_value + delta
...     truncated_value = truncate(truncated_value + delta)
...

>>> actual_value
96.45273913513529

>>> truncated_value
0.239
```

The meat of the simulation takes place in the `for` loop, which loops over the range of numbers between 0 and 999,999. The value taken from `range()` at each step is stored in the variable _, which you use here because you don't actually need this value inside of the loop.

At each step of the loop, `random.uniform()` generates a new random number between `-0.05` and `0.05`, which you assign to the variable `delta`. You calculate the new value of your investment by adding `delta` to `actual_value`, and then you calculate the truncated total by adding `delta` to `truncated_value` and then truncating this value with `truncate()`.

As you can see by inspecting the `actual_value` variable after running the loop, you only lost about $3.55. However, if you'd been looking at `truncated_value`, you'd have thought that you'd lost almost all of your money!

Note: In the above example, you use the `random.seed()` function to seed the pseudo-random number generator so that you can reproduce the output shown here.

To learn more about randomness in Python, check out Generating Random Data in Python (Guide) and Using the NumPy Random Number Generator.

Ignoring for the moment that `round()` doesn't behave quite as you expect, go ahead and try rerunning the simulation. You'll use `round()` this time to round to three decimal places at each step, and you'll seed the simulation again to get the same results as before:

Python

```python
>>> random.seed(100)
>>> actual_value, rounded_value = 100, 100

>>> for _ in range(1_000_000):
...     delta = random.uniform(-0.05, 0.05)
...     actual_value = actual_value + delta
...     rounded_value = round(rounded_value + delta, 3)
...

>>> actual_value
96.45273913513529

>>> rounded_value
96.258
```

What a difference!

Shocking as it may seem, this exact error caused quite a stir in the early 1980s when the system designed for recording the value of the Vancouver Stock Exchange truncated the overall index value to three decimal places instead of rounding. Rounding errors have swayed elections and even resulted in the loss of life.

How you round numbers is important, and as a responsible developer and software designer, you need to know what the common issues are and how to deal with them. So it's time to dive in and investigate what the different rounding methods are and how you can implement each one in pure Python.

Basic but Biased Rounding Strategies

There's a plethora of rounding strategies, each with advantages and disadvantages. In this section, you'll learn about some of the simplest techniques and see how they can influence your data. Later, you'll learn about better methods.

Truncating

The simplest, albeit crudest, method for rounding a number is to **truncate** the number to a given number of digits. When you truncate a number, you replace each digit after a given position with 0. Here are some examples:

Value	Truncated To	Result
12.345	Tens place	10
12.345	Ones place	12
12.345	Tenths place	12.3
12.345	Hundredths place	12.34

You've already seen one way to implement this in the `truncate()` function from the How Much Impact Can Rounding Have? section. In that function, the input number was truncated to three decimal places by:

- Multiplying the number by `1000` to shift the decimal point three places to the right
- Taking the integer part of that new number with `int()`
- Shifting the decimal place three places back to the left by dividing by `1000`

You can generalize this process by replacing `1000` with the number 10^p. That's `10` raised to the *p* power, where *p* is the number of decimal places to truncate to:

Python rounding.py

```python
def truncate(n, decimals=0):
    multiplier = 10**decimals
    return int(n * multiplier) / multiplier
```

In this version of `truncate()`, the second argument defaults to `0` so that if you don't pass a second argument to the function, then `truncate()` returns the integer part of whatever number you passed to it.

The `truncate()` function works well for both positive and negative numbers:

Python

```python
>>> from rounding import truncate

>>> truncate(12.5)
12.0

>>> truncate(-5.963, 1)
-5.9

>>> truncate(1.625, 2)
1.62
```

You can even pass a negative number to `decimals` to truncate digits to the left of the decimal point:

Python

```
>>> truncate(125.6, -1)
120.0

>>> truncate(-1374.25, -3)
-1000.0
```

When you truncate a positive number, you're rounding it down. Likewise, truncating a negative number rounds that number up. In a sense, truncation is a combination of rounding methods depending on the sign of the number that you're rounding.

Next up, you'll take a look at each of these rounding methods individually, starting with rounding up.

Rounding Up

The second rounding strategy that you'll look at is called **rounding up**. This strategy always rounds a number up to a specified number of digits. The following table summarizes this strategy:

Value	Round Up To	Result
12.345	Tens place	20
12.345	Ones place	13
12.345	Tenths place	12.4
12.345	Hundredths place	12.35

To implement the rounding up strategy in Python, you'll use the `ceil()` function from the `math` module.

The `ceil()` function gets its name from the mathematical term **ceiling**, which describes the nearest integer that's greater than or equal to a given number.

Every number that's not an integer lies between two consecutive integers. For example, the number `3.7` lies in the interval between `3` and `4`. The ceiling is the greater of the two endpoints of the interval. The lesser of the two endpoints is the **floor**. Thus, the ceiling of `3.7` is `4`, and the floor of `3.7` is `3`.

In mathematics, a special function called the **ceiling function** maps every number to its ceiling. To allow the ceiling function to accept integers, the ceiling of an integer is defined to be the integer itself. So the ceiling of the number `2` is `2`.

In Python, `math.ceil()` implements the ceiling function and always returns the nearest integer that's greater than or equal to its input:

Python

```
>>> import math

>>> math.ceil(3.7)
4

>>> math.ceil(2)
2

>>> math.ceil(-0.5)
0
```

Notice that the ceiling of `-0.5` is `0`, not `-1`. This makes sense because `0` is the nearest integer to `-0.5` that's greater than or equal to `-0.5`.

Now write a function called `round_up()` that implements the rounding up strategy:

Python `rounding.py`

```python
import math

# ...

def round_up(n, decimals=0):
    multiplier = 10**decimals
    return math.ceil(n * multiplier) / multiplier
```

You may notice that `round_up()` looks a lot like `truncate()`. First, you shift the decimal point in `n` the correct number of places to the right by multiplying `n` by `10**decimals`. You round this new value up to the nearest integer using `math.ceil()`, and then you shift the decimal point back to the left by dividing by `10**decimals`.

This pattern of shifting the decimal point, applying some rounding method to round to an integer, and then shifting the decimal point back will come up over and over again as you investigate more rounding methods. This is, after all, the mental algorithm that humans use to round numbers by hand.

Take a look at how well `round_up()` works for different inputs:

Python

```python
>>> from rounding import round_up

>>> round_up(1.1)
2.0

>>> round_up(1.23, 1)
1.3

>>> round_up(1.543, 2)
1.55
```

Just like with `truncate()`, you can pass a negative value to `decimals`:

Python

```python
>>> round_up(22.45, -1)
30.0

>>> round_up(1352, -2)
1400.0
```

When you pass a negative number to `decimals`, the number in the first argument of `round_up()` is rounded to the correct number of digits to the left of the decimal point.

Take a guess at what `round_up(-1.5)` returns:

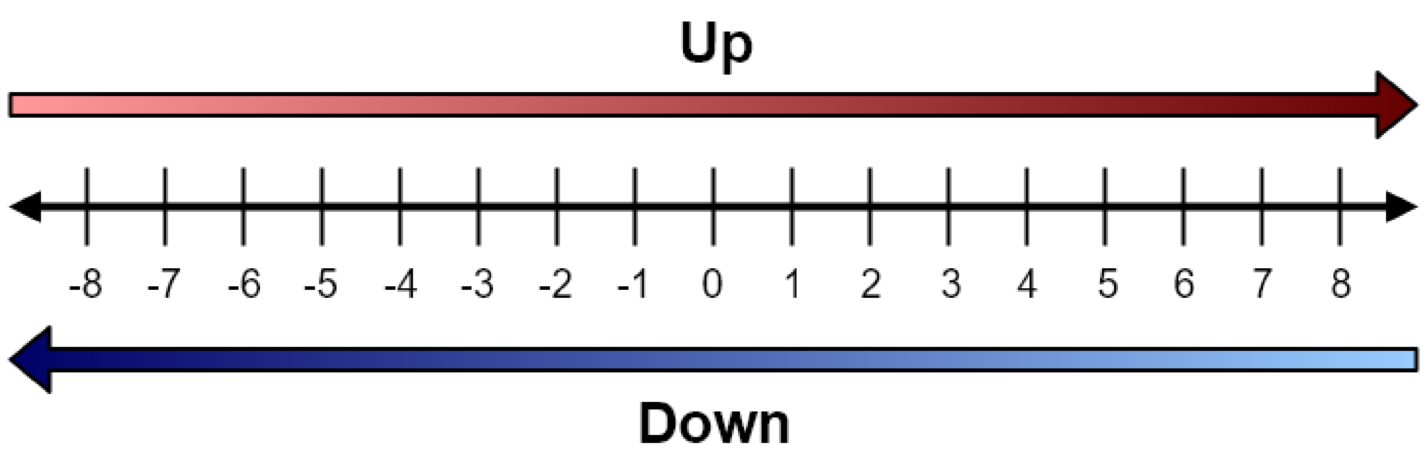
Python

```python
>>> round_up(-1.5)
-1.0
```

Is `-1.0` what you expected?

If you examine the logic used in defining `round_up()`—in particular, the way the `math.ceil()` function works—then it makes sense that `round_up(-1.5)` returns `-1.0`. However, some people naturally expect symmetry around zero when rounding numbers, so that if `1.5` gets rounded up to `2`, then `-1.5` should get rounded up to `-2`.

It's useful to establish some terminology. For this tutorial, you'll use the terms **round up** and **round down** according to the following diagram:



Round up to the right and down to the left. (Image: David Amos)

Rounding up always rounds a number to the right on the number line, and rounding down always rounds a number to the left on the number line.

Rounding Down

The counterpart to rounding up is the **rounding down** strategy, which always rounds a number down to a specified number of digits. Here are some examples illustrating this strategy:

Value	Rounded Down To	Result
12.345	Tens place	10
12.345	Ones place	12
12.345	Tenths place	12.3
12.345	Hundredths place	12.34

To implement the rounding down strategy in Python, you can follow the same algorithm that you used for both `truncate()` and `round_up()`. First shift the decimal point, then round to an integer, and finally shift the decimal point back.

In `round_up()`, you used `math.ceil()` to round up to the ceiling of the number after shifting the decimal point. For the rounding down strategy, though, you need to round to the floor of the number after shifting the decimal point.

Luckily, the `math` module has a `floor()` function that returns the floor of its input:

Python

```
>>> import math

>>> math.floor(1.2)
1

>>> math.floor(-0.5)
-1
```

Here's the definition of `round_down()`:

Python rounding.py

```
# ...

def round_down(n, decimals=0):
    multiplier = 10**decimals
    return math.floor(n * multiplier) / multiplier
```

That looks just like `round_up()`, except you've replaced `math.ceil()` with `math.floor()`.

You can test `round_down()` on a few different values:

Python

```
>>> from rounding import round_down

>>> round_down(1.5)
1.0

>>> round_down(1.37, 1)
1.3

>>> round_down(-0.5)
-1.0
```

The effects of `round_up()` and `round_down()` can be pretty extreme. By rounding the numbers in a large dataset up or down, you could potentially remove a ton of precision and drastically alter computations made from the data.

Before you discuss any more rounding strategies, take a moment to consider how rounding can make your data biased.

Interlude: Rounding Bias

You've now seen three rounding methods: `truncate()`, `round_up()`, and `round_down()`. All three of these techniques are rather crude when it comes to preserving a reasonable amount of precision for a given number.

There's one important difference in `truncate()` vs `round_up()` and `round_down()` that highlights an important aspect of rounding: symmetry around zero.

Recall that `round_up()` isn't symmetric around zero. In mathematical terms, a function *f(x)* is symmetric around zero if, for all values of *x*, *f(x)* + *f(-x)* = 0. For example, `round_up(1.5)` returns `2`, but `round_up(-1.5)` returns `-1`. The `round_down()` function isn't symmetric around 0, either.

On the other hand, the `truncate()` function *is* symmetric around zero. This is because, after shifting the decimal point to the right, `truncate()` chops off the remaining digits. When the initial value is positive, this amounts to rounding the number down. Negative numbers are rounded up. So, `truncate(1.5)` returns `1`, and `truncate(-1.5)` returns `-1`.

The concept of symmetry introduces the notion of **rounding bias**, which describes how rounding affects numeric data in a dataset.

The rounding up strategy has a **round toward positive infinity bias**, because it'll always round the value up in the direction of positive infinity. Likewise, the rounding down strategy has a **round toward negative infinity bias**.

The truncation strategy exhibits a round toward negative infinity bias on positive values and a round toward positive infinity for negative values. Rounding functions with this behavior are said to have a **round toward zero bias**, in general.

To see how this works in practice, consider the following list of floats:

Python

```
>>> numbers = [1.25, -2.67, 0.43, -1.79, 8.19, -4.32]
```

You can compute the mean of the values in `numbers` using the `statistics.mean()` function:

Python

```
>>> import statistics

>>> statistics.mean(numbers)
0.18166666666666653
```

Now apply each of `truncate()`, `round_up()`, and `round_down()` in a generator expression to round each number in `numbers` to one decimal place and calculate the new mean:

Python

```
>>> from rounding import truncate, round_up, round_down

>>> [truncate(n, 1) for n in numbers]
[1.2, -2.6, 0.4, -1.7, 8.1, -4.3]

>>> statistics.mean(truncate(n, 1) for n in numbers)
0.18333333333333333

>>> statistics.mean(round_up(n, 1) for n in numbers)
0.23333333333333325

>>> statistics.mean(round_down(n, 1) for n in numbers)
0.13333333333333316
```

After every number in `numbers` is rounded up, the new mean is about `0.233`, which is greater than the actual mean of about `0.182`. Rounding down shifts the mean downward to about `0.133`. The mean of the truncated values is about `0.183` and is the closest to the actual mean.

This example does *not* imply that you should always truncate when you need to round individual values while preserving a mean value as closely as possible. The `numbers` list contains an equal number of positive and negative values. The `truncate()` function would behave just like `round_up()` on a list of all positive values, and just like `round_down()` on a list of all negative values.

What this example does illustrate is the effect that rounding bias has on values computed from data that's been rounded. You'll need to keep these effects in mind when drawing conclusions from data that's been rounded.

Typically, when rounding, you're interested in rounding to the nearest number with some specified precision, instead of just rounding everything up or down.

For example, if someone asks you to round the numbers `1.23` and `1.28` to one decimal place, then you would probably respond quickly with `1.2` and `1.3`. The `truncate()`, `round_up()`, and `round_down()` functions don't do anything like this.

What about the number `1.25`? You probably immediately think to round this to `1.3`, but in reality, `1.25` is equidistant from `1.2` and `1.3`. In a sense, `1.2` and `1.3` are both the nearest numbers to `1.25` with precision of a single decimal place. The number `1.25` is called a **tie** with respect to `1.2` and `1.3`. In cases like this, you must assign a tiebreaker.

The way that most people learn to break ties is by rounding to the greater of the two possible numbers.

Better Rounding Strategies in Python

In this section, you'll continue your exploration of rounding strategies. You've seen some weaknesses in the methods that you've worked with so far. The upcoming techniques are better at dealing with biases, but they're also a bit more complex to consider and to implement.

Rounding Half Up

The **rounding half up** strategy rounds every number to the nearest number with the specified precision and breaks ties by rounding up. Here are some examples:

Value	Round Half Up To	Result
13.825	Tens place	10
13.825	Ones place	14
13.825	Tenths place	13.8
13.825	Hundredths place	13.83

To implement the rounding half up strategy in Python, you start as usual by shifting the decimal point to the right by the desired number of places. At this point, though, you need a way to determine if the digit just after the shifted decimal point is less than or greater than or equal to `5`.

One way to do this is to add `0.5` to the shifted value and then round down with `math.floor()`. This works because:

- If the digit in the first decimal place of the shifted value is less than `5`, then adding `0.5` won't change the integer part of the shifted value, so the floor is equal to the integer part.
- If the first digit after the decimal place is greater than or equal to `5`, then adding `0.5` will increase the integer part of the shifted value by `1`, so the floor is equal to this larger integer.

Here's what this looks like in Python:

Python rounding.py

```python
# ...

def round_half_up(n, decimals=0):
    multiplier = 10**decimals
    return math.floor(n * multiplier + 0.5) / multiplier
```

Notice that `round_half_up()` looks a lot like `round_down()`. This might be somewhat counterintuitive, but internally `round_half_up()` only rounds down. The trick is to add the `0.5` after shifting the decimal point so that the result of rounding down matches the expected value.

Now you can test `round_half_up()` on a few values to see that it works:

Python

```
>>> from rounding import round_half_up

>>> round_half_up(1.23, 1)
1.2

>>> round_half_up(1.28, 1)
1.3

>>> round_half_up(1.25, 1)
1.3
```

Since `round_half_up()` always breaks ties by rounding to the greater of the two possible values, negative values like `-1.5` round to `-1`, not to `-2`:

Python

```
>>> round_half_up(-1.5)
-1.0

>>> round_half_up(-1.25, 1)
-1.2
```

Great! You can now finally get that result that the built-in `round()` function denied you:

Python

```
>>> round_half_up(2.5)
3.0
```

Before you get too excited though, check out what happens when you try and round `-1.225` to `2` decimal places:

Python

```
>>> round_half_up(-1.225, 2)
-1.23
```

Wait. You just noted how ties get rounded to the greater of the two possible values. `-1.225` is smack in the middle of `-1.22` and `-1.23`. Since `-1.22` is the greater of these two, `round_half_up(-1.225, 2)` should return `-1.22`. But instead, you got `-1.23`.

Is there a bug in the `round_half_up()` function?

When `round_half_up()` rounds `-1.225` to two decimal places, the first thing it does is multiply `-1.225` by `100`. Make sure this works as expected:

Python

```
>>> -1.225 * 100
-122.50000000000001
```

Well … that's wrong! But it does explain why `round_half_up(-1.225, 2)` returns -1.23. Now continue the `round_half_up()` algorithm step-by-step, utilizing _ in the REPL to recall the last value output at each step:

Python

```
>>> _ + 0.5
-122.00000000000001

>>> math.floor(_)
-123

>>> _ / 100
-1.23
```

Even though `-122.00000000000001` is really close to `-122`, the nearest integer that's less than or equal to it is `-123`. When you shift the decimal point back to the left, the final value is `-1.23`.

Well, now you know how `round_half_up(-1.225, 2)` returns `-1.23` even though there's no logical error, but why does Python say that `-1.225 * 100` is `-122.50000000000001`? Is there a bug in Python?

Note*: For a hint about what's happening, type the following in a Python interpreter session:

Python

```
>>> 0.1 + 0.1 + 0.1
0.30000000000000004
```

Seeing this for the first time can be pretty shocking, but this is a classic example of the **floating-point representation error**. It has nothing to do with Python. The error has to do with how machines store floating-point numbers in memory.

Most modern computers store floating-point numbers as binary decimals with 53-bit precision. Only numbers that have finite binary decimal representations that can be expressed in 53 bits are stored as an exact value. Not every number has a finite binary decimal representation.

You can learn more about how Python treats floating point numbers in Numbers in Python.

For a more in-depth treatise on floating-point arithmetic, check out David Goldberg's article What Every Computer Scientist Should Know About Floating-Point Arithmetic, originally published in the journal *ACM Computing Surveys*, Vol. 23, No. 1, March 1991.

The fact that Python says that `-1.225 * 100` is `-122.50000000000001` is an artifact of the floating-point representation error. You might be asking yourself, *Okay, but is there a way to fix this?* A better question to ask yourself is *Do I* need *to fix this?*

Floating-point numbers don't have exact precision, and therefore you should *not* use them in situations where precision is paramount. For applications where the exact precision is necessary, you can use the `Decimal` class from Python's `decimal` module. You'll learn more about the `Decimal` class below.

If you've determined that Python's standard `float` class is sufficient for your application, then some occasional errors in `round_half_up()` due to the floating-point representation error shouldn't be a concern.

Now that you've gotten a taste of how machines round numbers in memory, you can continue your journey into rounding strategies by looking at another way to break a tie.

Rounding Half Down

The rounding half down strategy rounds to the nearest number with the desired precision, just like the rounding half up method, except that it breaks ties by rounding to the lesser of the two numbers. Here are some examples:

Value	Round Half Down To	Result
13.825	Tens place	10
13.825	Ones place	14
13.825	Tenths place	13.8
13.825	Hundredths place	13.82

You can implement the **rounding half down** strategy in Python by replacing `math.floor()` in the `round_half_up()` function with `math.ceil()` and subtracting `0.5` instead of adding:

Python rounding.py

```python
# ...

def round_half_down(n, decimals=0):
    multiplier = 10**decimals
    return math.ceil(n * multiplier - 0.5) / multiplier
```

Now check `round_half_down()` against a few test cases:

Python

```python
>>> from rounding import round_half_down

>>> round_half_down(1.5)
1.0

>>> round_half_down(-1.5)
-2.0

>>> round_half_down(2.25, 1)
2.2
```

Both `round_half_up()` and `round_half_down()` have no bias *in general*. However, rounding data with lots of ties does introduce a bias. For an extreme example, consider the following list of numbers:

Python

```python
>>> numbers = [-2.15, 1.45, -4.35, 12.75]
```

Compute the mean of these numbers:

Python

```python
>>> import statistics

>>> statistics.mean(numbers)
1.925
```

Next, compute the mean on the data after rounding to one decimal place with `round_half_up()` and `round_half_down()`:

Python

```
>>> statistics.mean(round_half_up(n, 1) for n in numbers)
1.975

>>> statistics.mean(round_half_down(n, 1) for n in numbers)
1.8749999999999996
```

Every number in `numbers` is a tie with respect to rounding to one decimal place. The `round_half_up()` function introduces a round toward positive infinity bias, and `round_half_down()` introduces a round toward negative infinity bias.

The remaining rounding strategies all attempt to mitigate these biases in different ways.

Rounding Half Away From Zero

If you examine `round_half_up()` and `round_half_down()` closely, then you'll notice that neither of these functions is symmetric around zero:

Python

```
>>> from rounding import round_half_down, round_half_up

>>> round_half_up(1.5)
2.0

>>> round_half_up(-1.5)
-1.0

>>> round_half_down(1.5)
1.0

>>> round_half_down(-1.5)
-2.0
```

One way to introduce symmetry is to always round a tie away from zero. The following table illustrates how this works:

Value	Round Half Away From Zero To	Result
15.25	Tens place	20
15.25	Ones place	15
15.25	Tenths place	15.3
-15.25	Tens place	-20
-15.25	Ones place	-15
-15.25	Tenths place	-15.3

To implement the **rounding half away from zero** strategy on a number `n`, you start as usual by shifting the decimal point to the right a given number of places. Then you look at the digit `d` immediately to the right of the decimal place in this new number. At this point, there are four cases to consider:

1. If `n` is positive and `d >= 5`, round up.
2. If `n` is positive and `d < 5`, round down.
3. If `n` is negative and `d >= 5`, round down.
4. If `n` is negative and `d < 5`, round up.

After rounding according to one of the above four rules, you then shift the decimal place back to the left.

Given a number `n` and a value for `decimals`, you could implement this in Python by using `round_half_up()` and `round_half_down()`:

Python

```
if n >= 0:
    rounded = round_half_up(n, decimals)
else:
    rounded = round_half_down(n, decimals)
```

That looks pretty good, but there's actually a simpler way!

If you first take the absolute value of `n` using Python's built-in `abs()` function, then you can just use `round_half_up()` to round the number. Then all you need to do is give the rounded number the same sign as `n`. One way to do this is using the `math.copysign()` function.

`math.copysign()` takes two numbers `a` and `b` and returns `a` with the sign of `b`:

Python

```
>>> import math

>>> math.copysign(1, -2)
-1.0
```

Notice that `math.copysign()` returns a `float`, even though both of its arguments were integers.

Using `abs()`, `round_half_up()`, and `math.copysign()`, you can implement the rounding half away from zero strategy in just two lines of Python:

Python rounding.py

```
# ...

def round_half_away_from_zero(n, decimals=0):
    rounded_abs = round_half_up(abs(n), decimals)
    return math.copysign(rounded_abs, n)
```

In `round_half_away_from_zero()`, you round the absolute value of `n` to `decimals` decimal places using `round_half_up()`, and you assign this result to the variable `rounded_abs`. Then you apply the original sign of `n` to `rounded_abs` using `math.copysign()`, and your function returns this final value with the correct sign.

Checking `round_half_away_from_zero()` on a few different values shows that the function behaves as expected:

Python

```
>>> from rounding import round_half_away_from_zero

>>> round_half_away_from_zero(1.5)
2.0

>>> round_half_away_from_zero(-1.5)
-2.0

>>> round_half_away_from_zero(-12.75, 1)
-12.8
```

The `round_half_away_from_zero()` function rounds numbers the way most people tend to round numbers in everyday life. Besides being the most familiar rounding function that you've seen so far, `round_half_away_from_zero()` also eliminates rounding bias well in datasets that have an equal number of positive and negative ties.

Check out how well `round_half_away_from_zero()` mitigates rounding bias in the example from the previous section:

Python

```
>>> numbers = [-2.15, 1.45, -4.35, 12.75]

>>> import statistics
>>> statistics.mean(numbers)
1.925

>>> statistics.mean(round_half_away_from_zero(n, 1) for n in numbers)
1.925
```

The mean value of the numbers in `numbers` is preserved exactly when you round each number in `numbers` to one decimal place with `round_half_away_from_zero()`!

However, `round_half_away_from_zero()` will exhibit a rounding bias when you round every number in datasets with only positive ties, only negative ties, or more ties of one sign than the other. It only mitigates bias well if there's a similar number of positive and negative ties in the dataset.

How do you handle situations where you have a drastically different number of positive and negative ties? The answer to this question brings you full circle to that deceptive function from the beginning of this tutorial: Python's built-in `round()` function.

Rounding Half to Even

One way to mitigate rounding bias when rounding values in a dataset is to round ties to the nearest even number at the desired precision. Here are some examples of how to do that:

Value	Round Half To Even To	Result
15.255	Tens place	20
15.255	Ones place	15
15.255	Tenths place	15.3
15.255	Hundredths place	15.26

The **rounding half to even strategy** is the strategy that Python's built-in `round()` function uses, and it's the default rounding rule in the IEEE-754 standard. This strategy works under the assumption that there's an equal probability of rounding a tie in a dataset down or up. In practice, this is usually the case.

Now you know why `round(2.5)` returns `2`. It's not a mistake. It is a conscious design decision based on solid recommendations.

To prove to yourself that `round()` really does round to even, try it on a few different values:

Python

```
>>> round(4.5)
4

>>> round(3.5)
4

>>> round(1.75, 1)
1.8

>>> round(1.65, 1)
1.6
```

The `round()` function is nearly free from bias, but it isn't perfect. For example, rounding bias can still creep in if the majority of the ties in your dataset round up to even instead of rounding down. Strategies that mitigate bias even better than rounding half to even do exist, but they're somewhat obscure and only necessary in extreme circumstances.

Finally, `round()` suffers from the same hiccups that you saw in `round_half_up()`, thanks to floating-point representation error:

Python

```
>>> # Expected value: 2.68
>>> round(2.675, 2)
2.67
```

You shouldn't be concerned with these occasional errors if floating-point precision is sufficient for your application.

When precision *is* paramount, you should use Python's `Decimal` class.

The `Decimal` Class

Python's `decimal` module is one of those batteries-included features of the language that you might not be aware of if you're new to Python. You can find the guiding principle of the `decimal` module in the documentation:

> Decimal "is based on a floating-point model which was designed with people in mind, and necessarily has a paramount guiding principle – computers must provide an arithmetic that works in the same way as the arithmetic that people learn at school." – excerpt from the decimal arithmetic specification. (Source)

The benefits of the `decimal` module include:

- **Exact decimal representation**: `0.1` is *actually* `0.1`, and `0.1 + 0.1 + 0.1 - 0.3` returns `0`, as you'd expect.
- **Preservation of significant digits**: When you add `1.20` and `2.50`, the result is `3.70`, with the trailing zero maintained to indicate significance.
- **User-alterable precision**: The default precision of the `decimal` module is twenty-eight digits, but you can alter this value to match the problem at hand.

But how does rounding work in the `decimal` module? Start by typing the following into a Python REPL:

Python

```
>>> import decimal
>>> decimal.getcontext()
Context(
    prec=28,
    rounding=ROUND_HALF_EVEN,
    Emin=-999999,
    Emax=999999,
    capitals=1,
    clamp=0,
    flags=[],
    traps=[InvalidOperation, DivisionByZero, Overflow]
)
```

`decimal.getcontext()` returns a `Context` object representing the default context of the `decimal` module. The context includes the default precision and the default rounding strategy, among other things.

As you can see in the example above, the default rounding strategy for the `decimal` module is `ROUND_HALF_EVEN`. This aligns with the built-in `round()` function and should be the preferred rounding strategy for most purposes.

Now declare a number using the `decimal` module's `Decimal` class. To do so, create a new `Decimal` instance by passing a string containing the desired value:

Python

```
>>> from decimal import Decimal
>>> Decimal("0.1")
Decimal('0.1')
```

You've created a `Decimal` object. You'll now see how to make calculations with it.

> **Note:** It's possible to create a `Decimal` instance from a floating-point number, but doing so introduces the floating-point representation error right off the bat. For example, check out what happens when you create a `Decimal` instance from the floating-point number `0.1`:

Python

```
>>> Decimal(0.1)
Decimal('0.1000000000000000055511151231257827021181583404541015625')
```

In order to maintain exact precision, you must create `Decimal` instances from strings containing the decimal numbers you need.

Just for fun, test the assertion that `Decimal` maintains exact decimal representation:

Python

```
>>> Decimal("0.1") + Decimal("0.1") + Decimal("0.1")
Decimal('0.3')
```

Ahhh. That's satisfying, isn't it?

You can round a `Decimal` with the `.quantize()` method or with the `round()` built-in function:

Python

```
>>> Decimal("1.65").quantize(Decimal("1.0"))
Decimal('1.6')

>>> round(Decimal("1.65"), 1)
Decimal('1.6')
```

The `.quantize()` method probably looks a little funky, so you'll break it down. The `Decimal("1.0")` argument in `.quantize()` determines the number of decimal places to round the number. Since `1.0` has one decimal place, the number `1.65` rounds to a single decimal place. The default rounding strategy is rounding half to even, so the result is `1.6`.

Recall that the `round()` function, which also uses the rounding half to even strategy, failed to round `2.675` to two decimal places correctly. Instead of giving you `2.68`, `round(2.675, 2)` returns `2.67`. Thanks to the `decimal` module's exact decimal representation, you won't have this issue with the `Decimal` class:

Python

```
>>> round(Decimal("2.675"), 2)
Decimal('2.68')
```

Another benefit of the `decimal` module is that it automatically takes care of rounding after performing arithmetic, and it preserves significant digits. To see this in action, change the default precision from twenty-eight digits to two, and then add the numbers `1.23` and `2.32`:

Python

```
>>> decimal.getcontext().prec = 2
>>> Decimal("1.23") + Decimal("2.32")
Decimal('3.6')
```

To change the precision, you call `decimal.getcontext()` and set the `.prec` attribute. If setting the attribute on a function call looks odd to you, you can do this because `.getcontext()` returns a special `Context` object that represents the current internal context containing the default parameters that the `decimal` module uses.

The exact value of `1.23` plus `2.32` is `3.55`. Since the precision is now two digits, and the rounding strategy is set to the default of rounding half to even, the value `3.55` is automatically rounded to `3.6`.

To change the default rounding strategy, you can set the `decimal.getcontext().rounding` property to any one of several flags. The following table summarizes these flags and which rounding strategy they implement:

Flag	Rounding Strategy
`decimal.ROUND_DOWN`	Truncation

Flag	Rounding Strategy
`decimal.ROUND_CEILING`	Rounding up
`decimal.ROUND_FLOOR`	Rounding down
`decimal.ROUND_UP`	Rounding away from zero
`decimal.ROUND_HALF_UP`	Rounding half away from zero
`decimal.ROUND_HALF_DOWN`	Rounding half toward zero
`decimal.ROUND_HALF_EVEN`	Rounding half to even
`decimal.ROUND_05UP`	Rounding up and rounding toward zero

The first point to notice is that the `decimal` module's naming scheme differs from what you were using earlier in the tutorial. For example, `decimal.ROUND_UP` implements the rounding away from zero strategy, which actually rounds negative numbers down.

Secondly, some of the rounding strategies mentioned in the table may look unfamiliar since you haven't explored them here. You've already seen how `decimal.ROUND_HALF_EVEN` works, so you'll take a look at each of the others in action.

The `decimal.ROUND_CEILING` strategy works just like the `round_up()` function that you defined earlier:

Python

```python
>>> decimal.getcontext().rounding = decimal.ROUND_CEILING

>>> round(Decimal("1.32"), 1)
Decimal('1.4')

>>> round(Decimal("-1.32"), 1)
Decimal('-1.3')
```

Notice that the results of `decimal.ROUND_CEILING` aren't symmetric around zero.

The `decimal.ROUND_FLOOR` strategy works just like your `round_down()` function:

Python

```python
>>> decimal.getcontext().rounding = decimal.ROUND_FLOOR

>>> round(Decimal("1.32"), 1)
Decimal('1.3')

>>> round(Decimal("-1.32"), 1)
Decimal('-1.4')
```

Like `decimal.ROUND_CEILING`, the `decimal.ROUND_FLOOR` strategy isn't symmetric around zero.

The `decimal.ROUND_DOWN` and `decimal.ROUND_UP` strategies have somewhat deceptive names. Both `ROUND_DOWN` and `ROUND_UP` are symmetric around zero:

Python

```
>>> decimal.getcontext().rounding = decimal.ROUND_DOWN

>>> round(Decimal("1.32"), 1)
Decimal('1.3')

>>> round(Decimal("-1.32"), 1)
Decimal('-1.3')

>>> decimal.getcontext().rounding = decimal.ROUND_UP

>>> round(Decimal("1.32"), 1)
Decimal('1.4')

>>> round(Decimal("-1.32"), 1)
Decimal('-1.4')
```

The `decimal.ROUND_DOWN` strategy rounds numbers toward zero, just like the `truncate()` function. On the other hand, `decimal.ROUND_UP` rounds everything away from zero. This is a clear break from the terminology that you were using earlier in the article, so keep that in mind when you're working with the `decimal` module.

There are three strategies in the `decimal` module that allow for more nuanced rounding. The `decimal.ROUND_HALF_UP` method rounds everything to the nearest number and breaks ties by rounding away from zero:

Python

```
>>> decimal.getcontext().rounding = decimal.ROUND_HALF_UP

>>> round(Decimal("1.35"), 1)
Decimal('1.4')

>>> round(Decimal("-1.35"), 1)
Decimal('-1.4')
```

Notice that `decimal.ROUND_HALF_UP` works just like your `round_half_away_from_zero()` and not like `round_half_up()`.

There's also a `decimal.ROUND_HALF_DOWN` strategy that breaks ties by rounding toward zero:

Python

```
>>> decimal.getcontext().rounding = decimal.ROUND_HALF_DOWN

>>> round(Decimal("1.35"), 1)
Decimal('1.3')

>>> round(Decimal("-1.35"), 1)
Decimal('-1.3')
```

The final rounding strategy available in the `decimal` module is very different from anything that you've seen so far:

Python

```
>>> decimal.getcontext().rounding = decimal.ROUND_05UP

>>> round(Decimal("1.38"), 1)
Decimal('1.3')

>>> round(Decimal("1.35"), 1)
Decimal('1.3')

>>> round(Decimal("-1.35"), 1)
Decimal('-1.3')
```

In the above examples, it looks as if `decimal.ROUND_05UP` rounds everything toward zero. In fact, this is exactly how `decimal.ROUND_05UP` works, unless the result of rounding ends in a `0` or `5`. In that case, the number gets rounded away from zero:

Python

```
>>> round(Decimal("1.49"), 1)
Decimal('1.4')

>>> round(Decimal("1.51"), 1)
Decimal('1.6')
```

In the first example, you first round the number `1.49` toward zero in the second decimal place, producing `1.4`. Since `1.4` doesn't end in a `0` or a `5`, you leave it as is. On the other hand, you round `1.51` toward zero in the second decimal place, resulting in the number `1.5`. This ends in a `5`, so you then round the first decimal place away from zero to `1.6`.

In this section, you've only focused on the rounding aspects of the `decimal` module. There are numerous other features that make `decimal` an excellent choice for applications where the standard floating-point precision is inadequate, such as banking and some problems in scientific computing.

For more information on `Decimal`, check out the Quick-start Tutorial in the Python docs.

Next, turn your attention to two staples of Python's scientific computing and data science stacks: NumPy and pandas.

Rounding NumPy Arrays

In the domains of data science and scientific computing, you often store your data as a NumPy `array`. One of NumPy's most powerful features is its use of vectorization and broadcasting to apply operations to an entire array at once instead of one element at a time.

You'll generate some data by creating a 3×4 NumPy array of pseudo-random numbers:

Python

```
>>> import numpy as np
>>> rng = np.random.default_rng(seed=444)

>>> data = rng.normal(size=(3, 4))
>>> data
array([[-0.69011449, -2.10455459, -0.78789037,  0.93417409],
       [ 0.58204206,  0.7022188 , -0.23768089,  0.1183704 ],
       [-0.41049553, -0.55583871,  0.00684936, -0.50000095]])
```

First, you create a random number generator and add a seed so that you can easily reproduce the output. Then you create a 3×4 NumPy array of floating-point numbers with `.normal()`, which produces normally distributed random numbers.

Note: You'll need to run `python -m pip install numpy` before typing the above code into your REPL if you don't already have NumPy in your environment. If you installed Python with Anaconda, then you're already set!

If you haven't used NumPy before, you can get a quick introduction in NumPy Tutorial: Your First Steps Into Data Science in Python and dive into how to write fast NumPy code in Look Ma, No `for` Loops: Array Programming With NumPy here at Real Python.

For more information on NumPy's `random` module, check out Using the NumPy Random Number Generator and Generating Random Data in Python (Guide).

To round all of the values in the `data` array, you can pass `data` as the argument to the `np.round()` function. You set the desired number of decimal places with the `decimals` keyword argument. The NumPy function uses the round half to even strategy, just like Python's built-in `round()` function.

For example, the following code rounds all of the values in `data` to three decimal places:

Python

```
>>> np.round(data, decimals=3)
array([[-0.69 , -2.105, -0.788,  0.934],
       [ 0.582,  0.702, -0.238,  0.118],
       [-0.41 , -0.556,  0.007, -0.5  ]])
```

Alternatively, you can also use the `.round()` method on the array:

Python

```
>>> data.round(decimals=3)
array([[-0.69 , -2.105, -0.788,  0.934],
       [ 0.582,  0.702, -0.238,  0.118],
       [-0.41 , -0.556,  0.007, -0.5  ]])
```

Typically, the `.round()` method call will be slightly faster than calling the `np.round()` function, as the latter delegates to the method under the hood.

If you need to round the data in your array to integers, then NumPy offers several options:

- `numpy.ceil()`
- `numpy.floor()`
- `numpy.trunc()`
- `numpy.rint()`

The `np.ceil()` function rounds every value in the array to the nearest integer greater than or equal to the original value:

Python

```
>>> np.ceil(data)
array([[-0., -2., -0.,  1.],
       [ 1.,  1., -0.,  1.],
       [-0., -0.,  1., -0.]])
```

Hey, that's a new number! Negative zero!

Actually, the IEEE-754 standard requires the implementation of both a positive and negative zero. What possible use is there for something like this? Wikipedia knows the answer:

> Informally, one may use the notation "`-0`" for a negative value that was rounded to zero. This notation may be useful when a negative sign is significant; for example, when tabulating Celsius temperatures, where a negative sign means below freezing. (Source)

To round every value down to the nearest integer, use `np.floor()`:

Python

```
>>> np.floor(data)
array([[-1., -3., -1.,  0.],
       [ 0.,  0., -1.,  0.],
       [-1., -1.,  0., -1.]])
```

You can also truncate each value to its integer component with `np.trunc()`:

Python

```
>>> np.trunc(data)
array([[-0., -2., -0.,  0.],
       [ 0.,  0., -0.,  0.],
       [-0., -0.,  0., -0.]])
```

Finally, to round to the nearest integer using the rounding half to even strategy, use `np.rint()`:

Python

```
>>> np.rint(data)
array([[-1., -2., -1.,  1.],
       [ 1.,  1., -0.,  0.],
       [-0., -1.,  0., -1.]])
```

You might have noticed that a lot of the rounding strategies that you studied earlier are missing here. For the vast majority of situations, the `.round()` method is all you need. If you need to implement another strategy, such as `round_half_up()`, you can do so with a small modification to your earlier code:

Python rounding.py

```python
import numpy as np

# ...

def round_half_up(n, decimals=0):
    multiplier = 10**decimals
    return np.floor(n * multiplier + 0.5) / multiplier
```

Here, you've replaced `math.floor()` with `np.floor()`. You can do similar modifications to your other rounding functions.

Thanks to NumPy's vectorized operations, the updated function works just as you expect:

Python

```
>>> from rounding import round_half_up

>>> round_half_up(data, decimals=2)
array([[-0.69, -2.1 , -0.79,  0.93],
       [ 0.58,  0.7 , -0.24,  0.12],
       [-0.41, -0.56,  0.01, -0.5 ]])
```

Now that you're a NumPy rounding master, take a look at Python's other data science heavyweight: the pandas library.

Rounding pandas `Series` and `DataFrame`

The pandas library has become a staple for data scientists and data analysts who work in Python. In the words of Real Python's own Joe Wyndham:

> pandas is a game changer for data science and analytics, particularly if you came to Python because you were searching for something more powerful than Excel and VBA. (Source)

Note: Before you continue, you'll need to run `python -m pip install pandas` if you don't already have it in your environment. As was the case for NumPy, if you installed Python with Anaconda, you should be ready to go!

The two main pandas data structures are the `DataFrame`, which in very loose terms works sort of like an Excel spreadsheet, and the `Series`, which you can think of as a column in a spreadsheet. You can round both `Series` and `DataFrame` objects efficiently using the `Series.round()` and `DataFrame.round()` methods:

Python

```
>>> import numpy as np
>>> import pandas as pd

>>> rng = np.random.default_rng(seed=444)

>>> series = pd.Series(rng.normal(size=4))
>>> series
0   -0.690114
1   -2.104555
2   -0.787890
3    0.934174
dtype: float64

>>> series.round(2)
0   -0.69
1   -2.10
2   -0.79
3    0.93
dtype: float64

>>> df = pd.DataFrame(rng.normal(size=(3, 3)), columns=["A", "B", "C"])
>>> df
          A         B         C
0  0.582042  0.702219 -0.237681
1  0.118370 -0.410496 -0.555839
2  0.006849 -0.500001 -0.141762

>>> df.round(3)
       A      B      C
0  0.582  0.702 -0.238
1  0.118 -0.410 -0.556
2  0.007 -0.500 -0.142
```

The `DataFrame.round()` method can also accept a dictionary or a `Series` to specify a different precision for each column. In the following example, you round the first column of `df` to one decimal place, the second to two, and the third to three decimal places:

Python

```
>>> df.round({"A": 1, "B": 2, "C": 3})
     A     B      C
0  0.6  0.70 -0.238
1  0.1 -0.41 -0.556
2  0.0 -0.50 -0.142
```

If you need more rounding flexibility, you can apply NumPy's `floor()`, `ceil()`, `trunc()`, and `rint()` functions to pandas `Series` and `DataFrame` objects:

Python

```
>>> np.floor(df)
     A    B    C
0  0.0  0.0 -1.0
1  0.0 -1.0 -1.0
2  0.0 -1.0 -1.0

>>> np.ceil(df)
     A    B    C
0  1.0  1.0 -0.0
1  1.0 -0.0 -0.0
2  1.0 -0.0 -0.0

>>> np.trunc(df)
     A    B    C
0  0.0  0.0 -0.0
1  0.0 -0.0 -0.0
2  0.0 -0.0 -0.0

>>> np.rint(df)
     A    B    C
0  1.0  1.0 -0.0
1  0.0 -0.0 -1.0
2  0.0 -1.0 -0.0
```

The modified `round_half_up()` function from the previous section will also work here:

Python

```
>>> from rounding import round_half_up

>>> round_half_up(df, decimals=2)
      A     B     C
0  0.58  0.70 -0.24
1  0.12 -0.41 -0.56
2  0.01 -0.50 -0.14
```

Congratulations, you're well on your way to rounding mastery! You now know that there are more ways to round a number than there are taco combinations. (Well … maybe not!) You can implement numerous rounding strategies in pure Python, and you've sharpened your skills on rounding NumPy arrays and pandas `Series` and `DataFrame` objects.

There's just one more step: knowing when to apply the right strategy.

Applications and Best Practices

The last stretch on your road to rounding virtuosity is understanding when to apply your newfound knowledge. In this section, you'll learn some best practices to make sure you round your numbers the right way.

Store More and Round Late

When you deal with large sets of data, storage can be an issue. In most relational databases, each column in a table is designed to store a specific data type, and numeric data types are often assigned precision to help conserve memory.

For example, a temperature sensor may report the temperature in a long-running industrial oven every ten seconds, accurate to eight decimal places. You could use the readings from this to detect abnormal fluctuations in temperature that could indicate the failure of a heating element or some other component. So, there might be a Python script running that compares each incoming reading to the last to check for large fluctuations.

The readings from this sensor are also stored in a SQL database so that the daily average temperature inside the oven can be computed each day at midnight. The manufacturer of the heating element inside the oven recommends replacing the component whenever the daily average temperature drops `.05` degrees below normal.

For this calculation, you only need three decimal places of precision. But you know from the incident at the Vancouver Stock Exchange that removing too much precision can drastically affect your calculation.

If you have the space available, then you should store the data at full precision. If storage is an issue, then a good rule of thumb is to store at least two or three more decimal places of precision than you need for your calculation.

Finally, when you compute the daily average temperature, you should calculate it to the full precision available and round the final answer.

Obey Local Currency Regulations

When you order a cup of coffee for $2.40 at the coffee shop, the merchant typically adds the required tax. The amount of that tax depends a lot on where you are geographically, but for the sake of argument, say it's 6%. The tax to be added comes out to $0.144. Should you round this up to $0.15 or down to $0.14? The answer probably depends on the regulations set forth by the local government!

Situations like this can also arise when you're converting one currency to another. In 1999, the European Commission on Economical and Financial Affairs codified the use of the rounding half away from zero strategy when converting currencies to the Euro, but other currencies may have adopted different regulations.

Another scenario, Swedish rounding, occurs when the minimum unit of currency at the accounting level in a country is smaller than the lowest unit of physical currency. For example, if a cup of coffee costs $2.54 after tax, but there are no one-cent coins in circulation, what do you do? The buyer won't have the exact amount, and the merchant can't make exact change.

A country's government typically determines how to handle situations like this. You can find a list of rounding methods used by various countries on Wikipedia.

If you're designing software for calculating currencies, then you should always check the local laws and regulations in your users' locations.

When in Doubt, Round Ties to Even

When you're rounding numbers in large datasets that are used in complex computations, the primary concern is limiting the growth of the error due to rounding.

Of all the methods that you've explored in this article, the rounding half to even strategy minimizes rounding bias the best. Fortunately, Python, NumPy, and pandas all default to this strategy, so by using the built-in rounding functions, you're already well protected!

Conclusion

Whew! What a journey this has been!

In this article, you've learned that:

- There are **various rounding strategies**, which you now know how to **implement** in pure Python.
- Every rounding strategy inherently introduces a **rounding bias**, and the **rounding half to even** strategy mitigates this bias well, most of the time.
- The way in which computers store floating-point numbers in memory naturally introduces a subtle **rounding error**, but you learned how to work around this with the **`decimal` module** in Python's standard library.
- You can round **NumPy** arrays and **pandas** `Series` and `DataFrame` objects.
- There are **best practices** for rounding with real-world data.

Get Your Code: Click here to download the free sample code you'll use to learn about rounding numbers in Python.

If you're interested in learning more and digging into the nitty-gritty details of everything that you've covered, then the links below should keep you busy for quite a while.

At the very least, if you've enjoyed this tutorial and learned something new from it, pass it on to a friend or team member! Be sure to share your thoughts in the comments. Your fellow programmers would love to hear some of your own rounding-related battle stories!

Happy Pythoning!

Additional Resources

Rounding strategies and bias:

- Rounding, Wikipedia
- Rounding Numbers without Adding a Bias, from ZipCPU

Floating-point and decimal specifications:

- IEEE-754, Wikipedia
- IBM’s General Decimal Arithmetic Specification

Interesting Reads:

- What Every Computer Scientist Should Know About Floating-Point Arithmetic, David Goldberg, ACM Computing Surveys, March 1991
- Floating Point Arithmetic: Issues and Limitations, from python.org
- Why Python’s Integer Division Floors, by Guido van Rossum

Frequently Asked Questions

Now that you have some experience with rounding numbers in Python, you can use the questions and answers below to check your understanding and recap what you’ve learned.

These FAQs are related to the most important concepts you’ve covered in this tutorial. Click the *Show/Hide* toggle beside each question to reveal the answer.

How does Python round numbers by default? Show/Hide

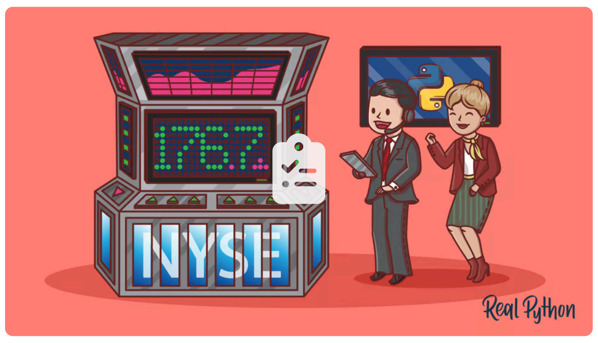
Why does Python round 2.5 to 2 instead of 3? Show/Hide

How can you round numbers to a specific number of decimal places in Python? Show/Hide

How do you apply different rounding strategies in Python? Show/Hide

How do you round numbers in NumPy arrays or pandas DataFrames? Show/Hide

Take the Quiz: Test your knowledge with our interactive “Rounding Numbers in Python” quiz. You’ll receive a score upon completion to help you track your learning progress:



Interactive Quiz

Rounding Numbers in Python

Test your knowledge of rounding numbers in Python.

Mark as Completed Share

Watch Now This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: **Rounding Numbers in Python**

About **David Amos**

David is a writer, programmer, and mathematician passionate about exploring mathematics through code.

» More about David

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

Adriana

Brenda

Bartosz

Geir Arne

Joanna

Kate

What Do You Think?

Rate this article:

LinkedIn Twitter Bluesky Facebook Email

What's your #1 takeaway or favorite thing you learned? How are you going to put your newfound skills to use? Leave a comment below and let us know.

Commenting Tips: The most useful comments are those written with the goal of learning from or helping out other students. Get tips for asking good questions and get answers to common questions in our support portal.

Looking for a real-time conversation? Visit the Real Python Community Chat or join the next "Office Hours" Live Q&A Session. Happy Pythoning!

Keep Learning

Related Topics: intermediate best-practices python

Recommended Video Course: Rounding Numbers in Python

Related Tutorials:

- Python's Built-in Functions: A Complete Exploration
- Sorting Algorithms in Python
- Lists vs Tuples in Python
- Python range(): Represent Numerical Ranges

Learn Python	Courses & Paths	Community	Membership	Company
Start Here	Learning Paths	Podcast	Plans & Pricing	About Us
Learning Resources	Quizzes & Exercises	Newsletter	Team Plans	Team
Code Mentor	Browse Topics	Community Chat	For Business	Sponsorships
Python Reference	Workshops	Office Hours	For Schools	Careers
Support Center	Books	Learner Stories	Reviews	Press Kit
				Merch

Privacy Policy · Terms of Use · Security · Contact

♥ Happy Pythoning!

© 2012–2025 DevCademy Media Inc. DBA Real Python. All rights reserved.
REALPYTHON™ is a trademark of DevCademy Media Inc.

Real Python
Online Python Training