

## interest.py

### Scenario 1: Interest Accrued Monthly

```
Principal: 12450
Rate: 0.0505
Term: 2
Compound: 12
Investing $12450 in a CD with an 5.05000000000001% interest rate for a term of 2 year(s) will earn
$1320.225605400694 in interest for a total payout of $13770.225605400694
```

### Scenario 2: Interest Accrued Daily

```
Principal: 12450
Rate: 0.0505
Term: 2
Compound: 365
Investing $12450 in a CD with an 5.05000000000001% interest rate for a term of 2 year(s) will earn
$1323.0479663958195 in interest for a total payout of $13773.04796639582
```

## rate.py

```
Principal: 63741.40
Total: 66969.18
Term: 1
Compound: 12
The interest rate on a $63741.4 CD that pays out $66969.18 over a 1 year term is 4.95005466333174%
```

## Questions

- 1) As described in the lectures, these rounding errors (or floating point errors) are caused because computers represent numbers with binary values. More specifically, decimals are represented by binary fractions which are summations of fractional powers of two. For example,  $1/8 + 1/16$  can be represented in binary fractional notation as 0.0011, which is exactly equal to 0.1875 in decimal notation. Other decimals, such as 0.3, do not have an exact representation using binary fractions, and so must be rounded to give an approximate value.
  - a. [Representing Rational Numbers With Python Fractions – Real Python](#)
  - b. [15. Floating-Point Arithmetic: Issues and Limitations — Python 3.13.7 documentation](#)
- 2) For decimal numbers that cannot be represented exactly with binary fractions, we must apply some cutoff length to the infinitely repeating sequence approximation if we are ever going to obtain an output. In Python, this cutoff happens once we have

53 bits of precision according to IEEE 754 standards, and then that truncated value is rounded. The value that is displayed when we print the output is the shortest truncated version of that decimal that would still result in the same value were we to repeat this approximation process. Hence, we see the value of 63741.4 rather than 63741.40 because they both result in the same approximation but 63741.4 is shorter. If we were using a Python version prior to 2.7, then we instead would have seen the value returned with 17 significant digits as 63741.4000000000001.

- a. [14. Floating Point Arithmetic: Issues and Limitations — Python 2.7.18 documentation](#)
- b. [15. Floating-Point Arithmetic: Issues and Limitations — Python 3.13.7 documentation](#)