Example 20. Python Explain the following floating-point rounding issue:

```
>>> 1 + 1 + 1 == 3
```

```
True
```

```
>>> 0.1 + 0.1 + 0.1 == 0.3
```

```
False
```

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

```
0.30000000000000004
```

Solution. As we saw in Example 6, 0.1 cannot be stored exactly as a floating-point number (when using base 2). Instead, it gets rounded up slightly. After adding three copies of this number, the error has increased to the point that it becomes visible as in the above output.

IMPORTANT. In the Python code above, we used the operator == (two equal signs) to compare two quantities. Note that we cannot use = (single equal sign) because that operator is used for assignment (`x = y` assigns the value of `y` to `x`, whereas `x == y` checks whether `x` and `y` are equal).

Comment. As the above issue shows, we should never test two floats x and y for equality. Instead, one typically tests whether the difference $|x-y|$ is less than a certain appropriate threshold. An alternative practical way is to round the floats before comparison (below, we round to 8 decimal digits):

```
>>> round(0.1 + 0.1 + 0.1, 8) == round(0.3, 8)
```

```
True
```

Example 21. (warmup) Using 64 bits, how many decimal digits can we store? With 53 bits?

Solution. Using 64 bits we can store 2^{64} different numbers. Since $\log_{10}(2^{64}) = 64\log_{10}(2) \approx 19.27$, we can store 19 decimal digits using 64 bits.

Likewise, since $\log_{10}(2^{53}) = 53\log_{10}(2) \approx 15.95$, we can store 15 decimal digits using 53 bits.

Example 22. To store 19 decimal digits, how many bits are needed?

Solution. There are 10^{19} many possibilities with 19 decimal digits. Since $\log_2(10^{19}) = 19\log_2(10) \approx 63.12$, we need 64 bits.

Comment. Note how this matches the conclusion of our computation in the previous example.

Example 23. Python Let us perform the previous calculation of $19\log_2(10)$ using Python. First of all, we need to get access to the $\log$ function because it is not available by default. Instead it resides in a module called `math`:

```
>>> from math import log
```

```
>>> 19*log(10, 2)
```

```
63.11663380285989
```

It might be unexpected that the 2 is the second argument of $\log$. If you want to learn more about how to use a function, you can enter the function name followed by a question mark:

```
>>> log?
```

This works in interactive mode (for instance, if you use Colab) but does not work in the textbox on our website.

Another floating-point issue

Example 24. Explain the following floating-point issue about mixing large and small numbers:

```
>>> 10.**9 + 10.**-9

    1000000000.0

>>> 10.**9 + 10.**-9 == 10.**9

    True

>>> 10.**9

    1000000000.0

>>> 10.**-9

    1e-09
```

Solution. Recall that double precision floats (which is what Python currently uses) use 52 bits for the significand which, together with the initial 1 (which is not stored), means that we are able to store numbers with 53 binary digits of precision. This translates to about $\log_{10}(2^{53}) = 53\log_{10}(2) \approx 15.95$ decimal digits. However, storing $10^9 + 10^{-9}$ exactly requires 19 decimal digits.

Coding in Python: binary digits of 0.1

In the next several examples, we will gradually get more professional in using Python for writing our first serious code.

Example 25. `Python` Recall that, to express, say, 0.1 in binary, we compute:

- $2 \cdot 0.1 = \boxed{0}.2$
- $2 \cdot 0.2 = \boxed{0}.4$
- $2 \cdot 0.4 = \boxed{0}.8$
- $2 \cdot 0.8 = \boxed{1}.6$
- $2 \cdot 0.6 = \boxed{1}.2$
- and so on...

The above 5 multiplications with 2 reveal 5 digits after the "decimal" point: $0.1 = (0.00011\cdots)_2$. (We can further see that the last four digits repeat; but we will ignore that fact here.)

Let us use Python to do this computation for us. We will start with very basic and naive code, and then upgrade it later.

```
>>> x = 0.1  # or any value < 1
```

Comment. Everything after the # symbol is considered a comment. This is useful for reminding ourselves of things related to the surrounding code. Comments are usually on a separate line but can be used as above (here, we remind ourselves that the code that follows is not going to handle a number like 2.1 correctly).

To have Python do the above computation for us, we plan to multiply x by 2 (call the result x again), collect a digit (we get that digit as the integer part of x), then subtract that digit from x and repeat. Python has a function called `trunc` which "truncates" a float to its integer part but we need to import it from a package called `math` to make it available.

```
>>> from math import trunc
```

Advanced comment. We can also use `*` in place of `trunc` to import all the functions from the `math` package. However, it is good practice to be explicit about what we need from a package. Note that the function `trunc` is very close to the function `floor` (which computes $\lfloor x \rfloor$, the floor of x, which is the closest integer when rounding down) which also seems appropriate here; however, `floor` returns a float rather than an integer, and we prefer the latter. Also note that we could use `int` (this is a general function that converts an input to an integer) instead of `trunc`. We chose `trunc` because it is more explicitly what we want, and because it gives us a chance to see how to import functions from a package.

We are now ready to compute the first digit:

```
>>> x = 2*x
    digit = trunc(x)
    print(digit)
    x = x-digit

   0
```

By copying-and-pasting these four lines four more times, we can produce the next four digits:

```
>>> x = 2*x
    digit = trunc(x)
    print(digit)
    x = x-digit
    x = 2*x
    digit = trunc(x)
    print(digit)
    x = x-digit
    x = 2*x
    digit = trunc(x)
    print(digit)
    x = x-digit
    x = 2*x
    digit = trunc(x)
    print(digit)
    x = x-digit

   0
   0
   1
   1
```

Clearly, we should not have to copy-and-paste repeated code like this. We will fix this issue in the next example, as well as discuss several other important improvements.