

# From decimal expansions to reduced fractions

Rachid Hamadi, CSE, UNSW

COMP9021 Principles of Programming, Term 3, 2020

```
[1]: from math import gcd
```

A real number is rational if and only if a pattern eventually appears in its decimal expansion that repeats forever. So  $\pi$ , being irrational, is such that no finite sequence of consecutive digits in 3.14159265358979... eventually repeats forever. On the other hand,

- $\frac{25}{12} = 2.08333...3...$
- $\frac{97}{21} = 4.619047619047619047...619047...$
- $\frac{11941}{49950} = 0.23905905905...905...$

The decimal expansion is unique except for fractions that in reduced form, have a power of 10 as denominator: those fractions have two decimal expansions, one that ends in 0 repeating forever, another one that ends in 9 repeating forever. For instance,  $\frac{1234567}{1000} = 1234.567000...0... = 1234.566999...9....$

We want to, given two nonempty strings of digits  $\sigma$  and  $\tau$  (that we treat as strings or numbers depending on the context), find out the unique natural numbers  $p$  and  $q$  such that the decimal expansion of  $\frac{p}{q}$  reads as  $0.\sigma\tau\tau\tau...\tau...$  and

- either  $p = 0$  and  $q = 1$  (case where  $\sigma$  and  $\tau$  consist of nothing but 0's), or
- $p$  and  $q$  are coprime, so  $\frac{p}{q}$  is in reduced form (including the case where  $p = 1$  and  $q = 1$  because  $\sigma$  and  $\tau$  consist of nothing but 9's).

For instance, if  $\sigma = 23$  and  $\tau = 905$ , then  $p = 11941$  and  $q = 49950$ .

Writing  $|\sigma|$  for the length (number of digits) in a string of digits  $\sigma$ , we compute:

$$\begin{aligned} 0.\sigma\tau\tau\tau...\tau... &= \sigma 10^{-|\sigma|} + \tau(10^{-|\sigma|-|\tau|} + 10^{-|\sigma|-2|\tau|} + 10^{-|\sigma|-3|\tau|} + \dots) \\ &= \sigma 10^{-|\sigma|} + \frac{\tau 10^{-|\sigma|-|\tau|}}{(1 - 10^{-|\tau|})} \\ &= \sigma 10^{-|\sigma|} + \frac{\tau 10^{-|\sigma|}}{(10^{|\tau|} - 1)} \\ &= \frac{\sigma 10^{-|\sigma|}(10^{|\tau|} - 1) + \tau 10^{-|\sigma|}}{(10^{|\tau|} - 1)} \\ &= \frac{\sigma(10^{|\tau|} - 1) + \tau}{(10^{|\tau|} - 1)10^{|\sigma|}} \end{aligned}$$

Reducing the last fraction if needed provides the desired answer.

The result of the previous computation immediately translates to the function that follows:

```
[2]: def compute_fraction(sigma, tau):
    numerator = int(sigma) * (10 ** len(tau) - 1) + int(tau)
    denominator = (10 ** len(tau) - 1) * 10 ** len(sigma)
    return numerator, denominator

compute_fraction('23', '905')
compute_fraction('000', '97')
compute_fraction('97', '000')
compute_fraction('01234', '543210')
```

[2]: (23882, 99900)

[2]: (97, 99000)

[2]: (96903, 99900)

[2]: (1234541976, 99999900000)

To reduce a fraction, it suffices to divide its numerator and its denominator by their gcd (greatest common divisor). The math module has a gcd function:

```
[3]: gcd(1234541976, 99999900000)
```

[3]: 24

Let us implement the gcd function ourselves, following Euclid's algorithm, which is based on the following reasoning. Let  $a$  and  $b$  be two natural numbers with  $b > 0$ . Since  $a = \lfloor \frac{a}{b} \rfloor b + a \bmod b$ :

- if  $n$  divides both  $a$  and  $b$  then it divides both  $a$  and  $\lfloor \frac{a}{b} \rfloor b$ , hence it divides  $a - \lfloor \frac{a}{b} \rfloor b$ , hence it divides  $a \bmod b$ ;
- conversely, if  $n$  divides both  $b$  and  $a \bmod b$  then it divides  $\lfloor \frac{a}{b} \rfloor b + a \bmod b$ , hence it divides  $a$ .

Hence  $n$  divides both  $a$  and  $b$  iff  $n$  divides both  $b$  and  $a \bmod b$ . So  $\gcd(a, b) = \gcd(b, a \bmod b)$ .

Since  $a \bmod b < b$ , we get a sequence of equalities of the form:  $\gcd(a, b) = \gcd(a_1, b_1) = \gcd(a_2, b_2) = \dots = \gcd(a_{k-1}, b_{k-1}) = \gcd(a_k, 0)$  with  $k \geq 1$  and  $b > b_1 > b_2 > \dots > b_{k-1} > 0$ ; as  $\gcd(a_k, 0) = a_k$ ,  $a_k$  is the gcd of  $a$  and  $b$ .

To compute  $\lfloor \frac{a}{b} \rfloor$ , Python offers the `//` operator; to compute  $a \bmod b$ , the `%` operator:

```
[4]: # True division.
# The result is always a floating point number.
8 / 2, 8.0 / 2, 8 / 2.0, 8.0 / 2.0

# Integer division.
# The result is an integer iff both arguments are integers.
9 // 2, 9.0 // 2, 9 // 2.0, 9.0 // 2.0

# Remainder.
# The result is an integer iff both arguments are integers.
9 % 2, 9.0 % 2, 9 % 2.0, 9.0 % 2.0
```

[4]: (4.0, 4.0, 4.0, 4.0)

[4]: (4, 4.0, 4.0, 4.0)

[4]: (1, 1.0, 1.0, 1.0)

If  $a$  and  $b$  are arbitrary numbers (not necessarily integers) with  $b \neq 0$ , then the equality  $a = qb + r$  together with the conditions

- $q$  is an integer
- $|r| < |b|$
- $r \neq 0 \rightarrow (r > 0 \leftrightarrow b > 0)$

determine  $q$  and  $r$  uniquely; `//` and `%` operate accordingly:

```
[5]: 5 // 2, 5 % 2
      -5 // 2, -5 % 2
      5 // -2, 5 % -2
      -5 // -2, -5 % -2
      print()

      7.5 // 2, 7.5 % 2
      -7.5 // 2, -7.5 % 2
      7.5 // -2, 7.5 % -2
      -7.5 // -2, -7.5 % -2
```

[5]: (2, 1)

[5]: (-3, 1)

[5]: (-3, -1)

[5]: (2, -1)

[5]: (3.0, 1.5)

[5]: (-4.0, 0.5)

[5]: (-4.0, -0.5)

[5]: (3.0, -1.5)

The `divmod()` function offers an alternative to the previous combined use of `//` and `%`:

```
[6]: divmod(5, 2)
      divmod(-5, 2)
      divmod(5, -2)
      divmod(-5, -2)
      print()

      divmod(7.5, 2)
      divmod(-7.5, 2)
      divmod(7.5, -2)
      divmod(-7.5, -2)
```

[6]: (2, 1)

[6]: (-3, 1)

[6]: (-3, -1)

[6]: (2, -1)

[6]: (3.0, 1.5)

[6]: (-4.0, 0.5)

[6]: (-4.0, -0.5)

[6]: (3.0, -1.5)

Let us get back to Euclid's algorithm, so assume again that  $a$  and  $b$  are two natural numbers with  $b > 0$ . To implement the algorithm and compute  $\text{gcd}(a, b)$ , it suffices to have two variables, say  $a$  and  $b$ , initialised to  $a$  and  $b$ , and then change the value of  $a$  to  $b$  and change the value of  $b$  to  $a \bmod b$ , and do that again and again until  $b$  gets the value 0. To change the value of  $a$  to  $a \bmod b$  and change the value of  $b$  to  $b$ , it seems necessary to introduce a third variable:

```
[7]: a = 30
     b = 18
     c = a % b
     a = b
     b = c
     a, b
```

[7]: (18, 12)

But Python makes it easier:

```
[8]: a = 30
     b = 18
     # Evaluate the expression on the right hand side;
     # the result is the tuple (18, 12).
     # Then assign that result to the tuple on the left,
     # component by component.
     a, b = b, a % b
     a, b
```

[8]: (18, 12)

Note that when the value of  $a$  is strictly smaller than the value of  $b$ , then  $a, b = b, a \% b$  exchanges the values of  $a$  and  $b$ :

```
[9]: a = 12
     b = 18
     a, b = b, a % b
     a, b
```

[9]: (18, 12)

On the other hand, if the value of  $a$  is at least equal to the value of  $b$ , then this holds too after  $a, b = b, a \% b$  has been executed. Let us trace all stages in the execution of Euclid's algorithm.

The code makes use of a `while` statement whose condition is not a boolean expression. Applying `bool()` to an expression reveals which one of `True` or `False` the expression evaluates to in contexts where one or the other is expected:

```
[10]: bool(None)
      bool(0), bool(5), bool(-3)
      bool(0.0), bool(0.1), bool(-3.14)
      bool([]), bool([0]), bool([[]])
      bool({}), bool({0: 0}), bool({0: None, 1: None})
      bool(''), bool(' '), bool('0000')
```

```
[10]: False
```

```
[10]: (False, True, True)
```

```
[10]: (False, True, True)
```

```
[10]: (False, True, True)
```

```
[10]: (False, True, True)
```

```
[10]: (False, True, True)
```

```
[11]: def trace_our_gcd(a, b):
      while b:
          a, b = b, a % b
          print(a, b)

      for a, b in (1233, 1233), (1233, 990), (990, 1233):
          print(f'\nTracing the computation of gcd of {a} and {b}:')
          trace_our_gcd(a, b)
```

```
Tracing the computation of gcd of 1233 and 1233:
1233 0
```

```
Tracing the computation of gcd of 1233 and 990:
990 243
243 18
18 9
9 0
```

```
Tracing the computation of gcd of 990 and 1233:
1233 990
990 243
243 18
18 9
9 0
```

The gcd is the value of `a` when exiting the `while` loop:

```
[12]: def our_gcd(a, b):
      while b:
```

```

    a, b = b, a % b
    return a

```

compute\_fraction() returns the numerator and denominator of a fraction that another function, say reduce(), can easily reduce thanks to our\_gcd(). It is natural to let reduce() take two arguments, the numerator and the denominator of the fraction to simplify, respectively. But compute\_fraction() returns those as the first and second elements of a tuple; a function always returns a single value. Between the parentheses that surround the arguments of a function f(), one can insert an expression that evaluates to a tuple and precede it with the \* symbol, which "unpacks" the members of the tuple and make them the arguments of f():

```

[13]: def f(a, b):
        return 2 * a, 2 * b

# Makes a equal to (1, 3), and provides no value to b.
f((1, 3))

```

```

↳-----

TypeError                                Traceback (most recent call↳
↳last)

```

```

<ipython-input-13-81290db4c2eb> in <module>
      3
      4 # Makes a equal to (1, 3), and provides no value to b.
----> 5 f((1, 3))

```

```

TypeError: f() missing 1 required positional argument: 'b'

```

```

[14]: f(1, 3)
# f(f(1, 3)) would be f((2, 6)); f(*f(1, 3)) is f(2, 6)
f(*f(1, 3))
f(*f(*f(1, 3)))

```

```

[14]: (2, 6)

```

```

[14]: (4, 12)

```

```

[14]: (8, 24)

```

The \* symbol can also be used in the definition of a function and precede the name of a parameter. It then has the opposite effect, namely, it makes a tuple out of all arguments that are provided to the function:

```

[15]: # x is the tuple of all arguments passed to f().
def f(*x):
    return x * 2

```

```
f()
f(0)
f(f(0))
f(*f(0))
f(f(f(0)))
f(f(*f(0)))
f(*f(*f(0)))
```

[15]: ()

[15]: (0, 0)

[15]: ((0, 0), (0, 0))

[15]: (0, 0, 0, 0)

[15]: (((0, 0), (0, 0)), ((0, 0), (0, 0)))

[15]: ((0, 0, 0, 0), (0, 0, 0, 0))

[15]: (0, 0, 0, 0, 0, 0, 0, 0)

Thanks to this syntax, it is possible to let `reduce()` as well as another function `output()` take two arguments `numerator` and `denominator`, and "pipe" `compute_fraction()`, `reduce()` and `output()` together so that the unpacked returned value of one function becomes the arguments of the function that follows:

```
[16]: def reduce(numerator, denominator):
        if numerator == 0:
            return 0, 1
        the_gcd = our_gcd(numerator, denominator)
        return numerator // the_gcd, denominator // the_gcd
```

```
[17]: def output(numerator, denominator):
        print(f'{numerator}/{denominator}')
```

```
[18]: output(*reduce(*compute_fraction('23', '905')))
        output(*reduce(*compute_fraction('000', '97')))
        output(*reduce(*compute_fraction('97', '000')))
        output(*reduce(*compute_fraction('01234', '543210')))
```

```
11941/49950
97/99000
97/100
51439249/4166662500
```