Mathematics with Python and Ruby/Whole numbers in Ruby

A peculiarity of the integers is that any one of them possesses a *predecessor* and a *successor*. Although it is very easy to compute them (subtract or add one to an integer), *Ruby* has methods coined *pred* and *succ* to handle them!

1 From a string

This script

a=7 puts(a)

yields exactly the same result than

a="7" puts(a)

at least at first sight. But if one tries to add 2,

a=7 puts(a+2)

yields 9 as expected, but

a="7" puts(a+2)

yields an error message, as one can not add a number with a string.

So a string representing an integer number has to be converted into a number with the *to_i* method (*to integer*). Thus

a="7" b=a.to_i puts(b+2)

yields 9 this time.

An other way to get an integer number from a string is to count its letters. Which is made with its *length* property:

t="Supercalifragilisticexpialidocious" n=t.length puts(n)

2 From a real number

A real number too has a *to_i* method which converts it to an integer. This is sometimes useful because for *Ruby* $\sqrt{100}$ is not considered as an integer number:

a=Math.sqrt(100) puts(a.integer?)

Ruby answers *false* because the number has been computed as *10.0* and not *10*. So

a=Math.sqrt(100).to_i puts(a.integer?)

yields the expected *true*. But

a=3.9999999 b=a.to_i puts(b)

may not have the expected result: *to_i* uses a truncature instead of an approximation. Actually, *floor* has the same effect:

a=3.9999999 b=a.floor puts(b)

On the converse, a real number has also a *ceil* method:

a=3.9999999 b=a.ceil puts(b)

But now an other problem arouses:

a=3.0000001 b=a.ceil puts(b)

Maybe 4 was not the expected answer!

Finally the best method to get an integer approximation is *round*:

a=3.9999999 b=a.round puts(b)

3 From an other integer

So, to get the successor of an integer, one asks to it to tell its successor, with its method *succ*:

puts(7.succ)

tells that 7+1=8 (you bet!), whereas

puts(7.pred)

shows that 7-1=6. But contrary to Peano's axiom number 7, zero *has* a predecessor (which is -1) because for *Ruby* the integers are rational integers and not only natural ones. Then, any integer has an additive inverse, which one can obtain by just preceding its name by a *minus* sign:

a=−5 puts(-a)

This yields 5 because the additive inverse of -5 is 5.

4 Primality

To know if an integer is prime requires an other module called *mathn*, which offers a primality test *prime?* (a boolean):

require 'mathn' a=2**32+1 puts(a.prime?)

This shows at a glance that 4 294 967 297 is *not* prime , shame on Fermat (he has excuses, he didnt't own a *Ruby* console!)!

5 Addition, subtraction and multiplication

For *Ruby*, the arithmetic operations are denoted by +, - et * without surprise. These operations work also on negative integers:

a=5 b=−8 puts(a+b) puts(a-b) puts(a*b)

6 Division

6.1 Quotient

Unless otherwise specified, the division denoted by the *slash* operator is the integer one. To *specify otherwise* one needs to write one of the operands as a *float*, with a period:

num=3 den=2 q=num/den puts(q)

yields 1 instead of 1.5 because the integer quotient is 1 (with a remainder), while any if the following variants yields 1.5:

puts(3.0/2) puts(3/2.0) puts(3.0/2.0) puts(3.to_f/2)

To compute the exact value of a non integer quotient, one needs to use fractions.

6.2 Remainder

Sometimes (often?) the quotient is not as important as the remainder (RSA kryptography for example, or even the hours, that are counted modulo 12). The remainder is given by the modulo operator denoted as %:

a=13 b=8 r=a%b puts(r)

It is then quite possible to compute in modular arithmetic with *Ruby*.

6.3 Divisors

In *Ruby*, the gcd is an infix operation:

a=13572468 b=12345678 g=a.gcd(b) puts(g)

Of course, *a.gcd(b)* and *b.gcd(a)* yield the same result.

Likewise, the lcm of two integers *a* and *b* can be computed as *a.lcm(b)*.

An integer which has no divisor (except 1 and itself) is a prime number. *Ruby* can test the primality of an (not to big) integer:

require 'prime' n=2012 puts(n.prime?) puts(n.prime_division)

The above example shows that *Ruby* can even find the prime divisors of an integer!

7 Exponentiation

In *Ruby*, the exponentiation operator is denoted with the same asterisk than the multiplication, but written twice:

a=4 b=2 puts(a**b) puts(b**a)

This shows that $2^4 = 4^2$.

Remarks:

1. If the exponent is negative, the result of the exponentiation is a fraction.
2. If the exponent is a *float*, the result is a float even if it is actually an integer (like in *256**0.25*).

7.1 Priority

In *Ruby* like in algebra, the computings are made in this order:

1. First, the parentheses;
2. then the functions (like the exponentiation for example);

" then only the multiplications and divisions;

1. At last, the additions and subtractions.

Thus

```
puts(2+3*5)
```

displays 17 and not 25, because in this case the operations are not computed from left to right, but the multiplication first.

8 Text and image sources, contributors, and licenses

8.1 Text

- **Mathematics with Python and Ruby/Whole numbers in Ruby** *Source:* https://en.wikibooks.org/wiki/Mathematics_with_Python_and_Ruby/Whole_numbers_in_Ruby?oldid=2289955 *Contributors:* Recent Runes and Alain Busser

8.2 Images

- **File:Sheep_herding,_Arkansas.jpg** *Source:* https://upload.wikimedia.org/wikipedia/commons/e/e7/Sheep_herding%2C_Arkansas.jpg *License:* Public domain *Contributors:* USDA *Original artist:* Ken Hammond

8.3 Content license

- Creative Commons Attribution-Share Alike 3.0