part0

September 4, 2019

1 Part 0: Representing numbers as strings

The following exercises are designed to reinforce your understanding of how we can view the encoding of a number as string of digits in a given base.

If you are interested in exploring this topic in more depth, see the "Floating-Point Arithmetic" section of the Python documentation.

1.1 Integers as strings

Consider the string of digits:

```
'16180339887'
```

If you are told this string is for a decimal number, meaning the base of its digits is ten (10), then its value is given by

$$[\![16180339887]\!]_{10} = (1 \times 10^{10}) + (6 \times 10^9) + (1 \times 10^8) + \cdots + (8 \times 10^1) + (7 \times 10^0) = 16{,}180{,}339{,}887.$$

Similarly, consider the following string of digits:

```
'100111010'
```

If you are told this string is for a binary number, meaning its base is two (2), then its value is

$$[\![100111010]\!]_2 = (1 \times 2^8) + (1 \times 2^5) + \cdots + (1 \times 2^1).$$

(What is this value?)
And in general, the value of a string of $d + 1$ digits in base b is,

$$[\![s_d s_{d-1} \cdots s_1 s_0]\!]_b = \sum_{i=0}^{d} s_i \times b^i.$$

Bases greater than ten (10). Observe that when the base at most ten, the digits are the usual decimal digits, 0, 1, 2, ..., 9. What happens when the base is greater than ten? For this notebook, suppose we are interested in bases that are at most 36; then, we will adopt the convention of using lowercase Roman letters, a, b, c, ..., z for "digits" whose values correspond to 10, 11, 12, ..., 35.

Before moving on to the next exercise, run the following code cell. It has three functions, which are used in some of the testing code. Given a base, one of these functions checks whether a single-character input string is a valid digit; and the other returns a list of all valid string digits. (The third one simply prints the valid digit list, given a base.) If you want some additional practice reading code, you might inspect these functions.

```
In [1]: def is_valid_strdigit(c, base=2):
            if type (c) is not str: return False # Reject non-string digits
            if (type (base) is not int) or (base < 2) or (base > 36): return False # Reject no
            if base < 2 or base > 36: return False # Reject bases outside 2-36
            if len (c) != 1: return False # Reject anything that is not a single character
            if '0' <= c <= str (min (base-1, 9)): return True # Numerical digits for bases up
            if base > 10 and 0 <= ord (c) - ord ('a') < base-10: return True # Letter digits f
            return False # Reject everything else

        def valid_strdigits(base=2):
            POSSIBLE_DIGITS = '0123456789abcdefghijklmnopqrstuvwxyz'
            return [c for c in POSSIBLE_DIGITS if is_valid_strdigit(c, base)]

        def print_valid_strdigits(base=2):
            valid_list = valid_strdigits(base)
            if not valid_list:
                msg = '(none)'
            else:
                msg = ', '.join([c for c in valid_list])
            print('The valid base ' + str(base) + ' digits: ' + msg)

        # Quick demo:
        print_valid_strdigits(6)
        print_valid_strdigits(16)
        print_valid_strdigits(23)

The valid base 6 digits: 0, 1, 2, 3, 4, 5
The valid base 16 digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f
The valid base 23 digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f, g, h, i, j, k, l, m
```

Exercise 0 (3 points). Write a function, `eval_strint(s, base)`. It takes a string of digits `s` in the base given by `base`. It returns its value as an integer.

That is, this function implements the mathematical object, $[\![s]\!]_b$, which would convert a string s to its numerical value, assuming its digits are given in base b. For example:

```
eval_strint('100111010', base=2) == 314
```

Hint: Python makes this exercise very easy. Search Python's online documentation for information about the `int()` constructor to see how you can apply it to solve this problem. (You have encountered this constructor already, in Notebook/Assignment 2.)

2

```
In [2]: def get_val(s):
            if s.isdigit():
                return int(s)
            else:
                return ord(s.lower())-87

        def eval_strint(s, base=2):
            assert type(s) is str
            assert 2 <= base <= 36
            ###
            ### YOUR CODE HERE
            ###
            val=0
            s=s[::-1]
            for i in range(0,len(s)):
                if s[i].isdigit:
                    val = val + (base**i)*get_val(s[i])
            return val
```

```
In [3]: # Test: `eval_strint_test0` (1 point)

        def check_eval_strint(s, v, base=2):
            v_s = eval_strint(s, base)
            msg = "'{}' -> {}".format (s, v_s)
            print(msg)
            assert v_s == v, "Results do not match expected solution."

        # Test 0: From the videos
        check_eval_strint('16180339887', 16180339887, base=10)
        print ("\n(Passed!)")

'16180339887' -> 16180339887

(Passed!)
```

```
In [4]: # Test: `eval_strint_test1` (1 point)
        check_eval_strint('100111010', 314, base=2)
        print ("\n(Passed!)")

'100111010' -> 314

(Passed!)
```

```
In [5]: # Test: `eval_strint_test2` (1 point)
        check_eval_strint('a205b064', 2718281828, base=16)
        print ("\n(Passed!)")
```

```
'a205b064' -> 2718281828
```

(Passed!)

1.2 Fractional values

Recall that we can extend the basic string representation to include a fractional part by interpreting digits to the right of the "fractional point" (i.e., "the dot") as having negative indices. For instance,

$$[\![3.14]\!]_{10} = (3 \times 10^0) + (1 \times 10^{-1}) + (4 \times 10^{-2}).$$

Or, in general,

$$[\![s_d s_{d-1} \cdots s_1 s_0 \underset{\uparrow}{.} s_{-1} s_{-2} \cdots s_{-r}]\!]_b = \sum_{i=-r}^{d} s_i \times b^i.$$

Exercise 1 (4 points). Suppose a string of digits s in base `base` contains up to one fractional point. Complete the function, `eval_strfrac(s, base)`, so that it returns its corresponding floating-point value.

Your function should *always* return a value of type `float`, even if the input happens to correspond to an exact integer.

Examples:

```
eval_strfrac('3.14', base=10)  ~= 3.14
eval_strfrac('100.101', base=2)  == 4.625
eval_strfrac('2c', base=16)  ~= 44.0    # Note: Must be a float even with an integer input!
```

Comment. Because of potential floating-point roundoff errors, as explained in the videos, conversions based on the general polynomial formula given previously will not be exact. The testing code will include a built-in tolerance to account for such errors.

Hint. You should be able to construct an elegant solution that reuses the function, `eval_strint()`, from Exercise 0.

```
In [6]: def is_valid_strfrac(s, base=2):
            return all([is_valid_strdigit(c, base) for c in s if c != '.']) \
                and (len([c for c in s if c == '.']) <= 1)

        def eval_strfrac(s, base=2):
            assert is_valid_strfrac(s, base), "'{}' contains invalid digits for a base-{} numbe

            ###
            ### YOUR CODE HERE
            ###
            q=s.split('.')
            f=eval_strint(q[0], base)
            if len(q) > 1:
                h=eval_strint(q[1], base)
```

```
                h=int(h)/base**len(q[1])
                return f+h
            return float(f)
```

In [7]: *# Test 0: `eval_strfrac_test0` (1 point)*

```
        def check_eval_strfrac(s, v_true, base=2, tol=1e-7):
            v_you = eval_strfrac(s, base)
            assert type(v_you) is float, "Your function did not return a `float` as instructed
            delta_v = v_you - v_true
            msg = "[{}]_{{{}}} ~= {}: You computed {}, which differs by {}.".format(s, base, v_
                                                                            v_you, delt

            print(msg)
            assert abs(delta_v) <= tol, "Difference exceeds expected tolerance."

            # Test cases from the video
            check_eval_strfrac('3.14', 3.14, base=10)
            check_eval_strfrac('100.101', 4.625, base=2)
            check_eval_strfrac('11.0010001111', 3.1396484375, base=2)

            # A hex test case
            check_eval_strfrac('f.a', 15.625, base=16)

            print("\n(Passed!)")
```

```
[3.14]_{10} ~= 3.14: You computed 3.14, which differs by 0.0.
[100.101]_{2} ~= 4.625: You computed 4.625, which differs by 0.0.
[11.0010001111]_{2} ~= 3.1396484375: You computed 3.1396484375, which differs by 0.0.
[f.a]_{16} ~= 15.625: You computed 15.625, which differs by 0.0.

(Passed!)
```

In [8]: *# Test 1: `eval_strfrac_test1` (1 point)*

```
        check_eval_strfrac('1101', 13, base=2)
```

```
[1101]_{2} ~= 13: You computed 13.0, which differs by 0.0.
```

In [9]: *# Test 2: `eval_strfrac_test2` (2 point)*

```
        def check_random_strfrac():
            from random import randint
            b = randint(2, 36) # base
            d = randint(0, 5) # leading digits
            r = randint(0, 5) # trailing digits
            v_true = 0.0
            s = ''
```

```python
        possible_digits = valid_strdigits(b)
        for i in range(-r, d+1):
            v_i = randint(0, b-1)
            s_i = possible_digits[v_i]

            v_true += v_i * (b**i)
            s = s_i + s
            if i == -1:
                s = '.' + s
        check_eval_strfrac(s, v_true, base=b)

    for _ in range(10):
        check_random_strfrac()

    print("\n(Passed!)")
```

```
[133.11]_{4} ~= 31.3125: You computed 31.3125, which differs by 0.0.
[1c]_{17} ~= 29.0: You computed 29.0, which differs by 0.0.
[gk.8f8]_{21} ~= 356.4158298239931: You computed 356.4158298239931, which differs by 0.0.
[16fo.19l]_{26} ~= 373566.05296995904: You computed 373566.05296995904, which differs by 0.0.
[u.q0alc]_{36} ~= 30.722449258243152: You computed 30.722449258243152, which differs by 0.0.
[238.j]_{20} ~= 868.95: You computed 868.95, which differs by 0.0.
[a]_{35} ~= 10.0: You computed 10.0, which differs by 0.0.
[818e.46bg4]_{19} ~= 55399.228874935936: You computed 55399.228874935936, which differs by 0.0
[0021.13]_{4} ~= 9.4375: You computed 9.4375, which differs by 0.0.
[8ef3fc.ibefr]_{29} ~= 174359931.6343659: You computed 174359931.6343659, which differs by 0.0

(Passed!)
```

1.3 Floating-point encodings

Recall that a floating-point encoding or format is a normalized scientific notation consisting of a *base*, a *sign*, a fractional *significand* or *mantissa*, and a signed integer *exponent*. Conceptually, think of it as a tuple of the form, $(\pm, [\![s]\!]_b, x)$, where b is the digit base (e.g., decimal, binary); $\pm$ is the sign bit; s is the significand encoded as a base b string; and x is the exponent. For simplicity, let's assume that only the significand s is encoded in base b and treat x as an integer value. Mathematically, the value of this tuple is $\pm [\![s]\!]_b \times b^x$.

IEEE double-precision. For instance, Python, R, and MATLAB, by default, store their floating-point values in a standard tuple representation known as *IEEE double-precision format*. It's a 64-bit binary encoding having the following components:

- The most significant bit indicates the sign of the value.
- The significand is a 53-bit string with an *implicit* leading one. That is, if the bit string representation of s is $s_0.s_1s_2\cdots s_d$, then $s_0 = 1$ always and is never stored explicitly. That also means $d = 52$.
- The exponent is an 11-bit string and is treated as a signed integer in the range $[-1022, 1023]$.

Thus, the smallest positive value in this format $2^{-1022} \approx 2.23 \times 10^{-308}$, and the smallest positive value greater than 1 is $1 + \epsilon$, where $\epsilon = 2^{-52} \approx 2.22 \times 10^{-16}$ is known as *machine epsilon* (in this case, for double-precision).

Special values. You might have noticed that the exponent is slightly asymmetric. Part of the reason is that the IEEE floating-point encoding can also represent several kinds of special values, such as infinities and an odd bird called "not-a-number" or `NaN`. This latter value, which you may have seen if you have used any standard statistical packages, can be used to encode certain kinds of floating-point exceptions that result when, for instance, you try to divide zero by zero.

> If you are familiar with languages like C, C++, or Java, then IEEE double-precision format is the same as the `double` primitive type. The other common format is single-precision, which is `float` in those same languages.

Inspecting a floating-point number in Python. Python provides support for looking at floating-point values directly! Given any floating-point variable, `v` (that is, `type(v) is float`), the method `v.hex()` returns a string representation of its encoding. It's easiest to see by example, so run the following code cell:

```
In [10]: def print_fp_hex(v):
             assert type(v) is float
             print("v = {} ==> v.hex() == '{}'".format(v, v.hex()))

         print_fp_hex(0.0)
         print_fp_hex(1.0)
         print_fp_hex(16.0625)
         print_fp_hex(-0.1)

v = 0.0 ==> v.hex() == '0x0.0p+0'
v = 1.0 ==> v.hex() == '0x1.0000000000000p+0'
v = 16.0625 ==> v.hex() == '0x1.0100000000000p+4'
v = -0.1 ==> v.hex() == '-0x1.999999999999ap-4'
```

Observe that the format has these properties: * If `v` is negative, the first character of the string is `'-'`. * The next two characters are always `'0x'`. * Following that, the next characters up to but excluding the character `'p'` is a fractional string of hexadecimal (base-16) digits. In other words, this substring corresponds to the significand encoded in base-16. * The `'p'` character separates the significand from the exponent. The exponent follows, as a signed integer (`'+'` or `'-'` prefix). Its implied base is two (2)---**not** base-16, even though the significand is.

Thus, to convert this string back into the floating-point value, you could do the following: * Record the sign as a value, `v_sign`, which is either +1 or -1. * Convert the significand into a fractional value, `v_signif`, assuming base-16 digits. * Extract the exponent as a signed integer value, `v_exp`. * Compute the final value as `v_sign * v_signif * (2.0**v_exp)`.

For example, here is how you can get 16.025 back from its `hex()` representation, `'0x1.0100000000000p+4'`:

```
In [11]: # Recall: v = 16.0625 ==> v.hex() == '0x1.0100000000000p+4'
         print((+1.0) * eval_strfrac('1.0100000000000', base=16) * (2**4))
```

7

16.0625

Exercise 2 (4 points). Write a function, `fp_bin(v)`, that determines the IEEE-754 tuple representation of any double-precision floating-point value, v. That is, given the variable v such that `type(v) is float`, it should return a tuple with three components, `(s_sign, s_bin, v_exp)` such that

- `s_sign` is a string representing the sign bit, encoded as either a `'+'` or `'-'` character;
- `s_signif` is the significand, which should be a string of 54 bits having the form, x.xxx...x, where there are (at most) 53 x bits (0 or 1 values);
- `v_exp` is the value of the exponent and should be an *integer*.

For example:

```
v = -1280.03125
assert v.hex() == '-0x1.4002000000000p+10'
assert fp_bin(v) == ('-', '1.0100000000000001000000000000000000000000000000000000', 10)
```

There are many ways to approach this problem. One we came up exploits the observation that $[\![0]\!]_{16} == [\![0000]\!]_2$ and $[\![f]\!]_{16} = [\![1111]\!]$ and applies an idea in this Stackoverflow post: https://stackoverflow.com/questions/1425493/convert-hex-to-binary

```
In [12]: from math import log, floor

         def fp_bin(v):
             assert type(v) is float
             print(v)
         ###
         ### YOUR CODE HERE
         ###
             hex_value=v.hex()
             print(hex_value)
             s_sign='+'
             if(hex_value.startswith('-')):
                 s_sign='-'
                 hex_value=hex_value[1:]
             hex_value=hex_value[2:]
             exponent=int(hex_value.split('p')[1])
             hex_num=hex_value.split('p')[0]
             padded_with_zeros=list(hex_num[2:])+['0']*(13-len(hex_num[2:]))
             s_signif=hex_num[:2]+''.join([bin(int(i,16))[2:].zfill(4) for i in padded_with_ze:
             return tuple([s_sign,s_signif,exponent])

In [13]: # Test: `fp_bin_test0` (2 points)

         def assert_fp_bin_props(v_you_tuple):
             assert len(v_you_tuple) == 3, "Your floating point value should be a tuple with tl
             v_you_sign, v_you_signif, v_you_exp = v_you_tuple
```

8

```python
        assert v_you_sign in ['+', '-'], "Your sign is '{}' instead of either '+' or '-'"
        assert type(v_you_signif) is str, "Your significand is a '{}', rather than a strin
        assert len(v_you_signif) == 54, "Your significand has {} chars instead of 54.".for
        assert v_you_signif[1] == '.', "Your significand should have a point (period) char
        assert v_you_signif[0] in ['0', '1'], "Your significand have a leading 0 or 1, not
        assert all([c in ['0', '1'] for c in v_you_signif[2:]]), "Your significand should
        assert type(v_you_exp) is int, "Your exponent should be an integer value, but isn
        assert -1022 <= v_you_exp <= 1023, "The exponent of {} lies outside the interval,

    def check_fp_bin(v, x_true):
        x_you = fp_bin(v)
        print("""{} [{}] ==
            {}
    vs. you: {}
    """.format(v, v.hex(), x_true, x_you))
        assert_fp_bin_props(x_you)
        assert x_you == x_true, "Results do not match!"

    check_fp_bin(0.0, ('+', '0.0000000000000000000000000000000000000000000000000000', 0))
    check_fp_bin(-0.1, ('-', '1.1001100110011001100110011001100110011001100110011010', -4)
    check_fp_bin(1.0 + (2**(-52)), ('+', '1.00000000000000000000000000000000000000000000000000

    check_fp_bin(-1280.03125, ('-', '1.010000000000001000000000000000000000000000000000000
    check_fp_bin(6.2831853072, ('+', '1.1001001000011111101101010100010001001000011011100
    check_fp_bin(-0.7614972118393695, ('-', '1.1000101111000101111011001101001101100001100

    print("\n(Passed!)")
```

```
0.0
0x0.0p+0
0.0 [0x0.0p+0] ==
        ('+', '0.0000000000000000000000000000000000000000000000000000', 0)
vs. you: ('+', '0.0000000000000000000000000000000000000000000000000000', 0)


-0.1
-0x1.999999999999ap-4
-0.1 [-0x1.999999999999ap-4] ==
        ('-', '1.1001100110011001100110011001100110011001100110011010', -4)
vs. you: ('-', '1.1001100110011001100110011001100110011001100110011010', -4)


1.0000000000000002
0x1.0000000000001p+0
1.0000000000000002 [0x1.0000000000001p+0] ==
        ('+', '1.0000000000000000000000000000000000000000000000000001', 0)
vs. you: ('+', '1.0000000000000000000000000000000000000000000000000001', 0)


-1280.03125
-0x1.4002000000000p+10
```

```
-1280.03125 [-0x1.4002000000000p+10] ==
        ('-', '1.0100000000000010000000000000000000000000000000000000', 10)
vs. you: ('-', '1.0100000000000010000000000000000000000000000000000000', 10)


6.2831853072
0x1.921fb544486e0p+2
6.2831853072 [0x1.921fb544486e0p+2] ==
        ('+', '1.1001001000011111101101010100010001001001000011011100000', 2)
vs. you: ('+', '1.1001001000011111101101010100010001001001000011011100000', 2)


-0.7614972118393695
-0x1.85e2f669b0c80p-1
-0.7614972118393695 [-0x1.85e2f669b0c80p-1] ==
        ('-', '1.1000010111100010111101100110100110110000110010000000', -1)
vs. you: ('-', '1.1000010111100010111101100110100110110000110010000000', -1)



(Passed!)
```

The following test cell tries some additional cases, which are random floating-point values. The cell uses your `eval_strfrac()` function, so that must be working properly in order for this test to work.

```
In [14]: # Test: `fp_bin_test1` (2 points)

         def check_rand_case():
             from random import random, expovariate
             sign_rand0 = 1.0 if random() < 0.5 else -1.0
             base_rand = random()
             factor_rand = 1.0 if random() < 0.5 else expovariate(1/10000 if random() < 0.5 el
             v_rand = sign_rand0 * base_rand * factor_rand
             print(f"==> Random test case: {v_rand} [{v_rand.hex()}]")
             v_you_tuple = fp_bin(v_rand)
             print(f"    You generated: {v_you_tuple}")
             assert_fp_bin_props(v_you_tuple)
             v_you_sign = 1.0 if v_you_tuple[0] == '+' else -1.0
             v_you_signif = eval_strfrac(v_you_tuple[1], base=2)
             v_you_exp = v_you_tuple[2]
             v_you = v_you_sign * v_you_signif * (2.0 ** v_you_exp)
             print(f"    The equivalent value is: {v_you}")
             assert v_you == v_rand, "Values do not match!"

         for _ in range(10):
             check_rand_case()

         print("\n(Passed.)")

==> Random test case: -0.4071138605844644 [-0x1.a0e274b3d5958p-2]
```

```
-0.4071138605844644
-0x1.a0e274b3d5958p-2
    You generated: ('-', '1.1010000111000100111010010110011110101011001011000', -2)
    The equivalent value is: -0.4071138605844644
==> Random test case: -0.37849845330605036 [-0x1.8395193a24ef6p-2]
-0.37849845330605036
-0x1.8395193a24ef6p-2
    You generated: ('-', '1.1000001110010101000110010011101000100100111011110110', -2)
    The equivalent value is: -0.37849845330605036
==> Random test case: 2522.060872394256 [0x1.3b41f2aaa9d1ep+11]
2522.060872394256
0x1.3b41f2aaa9d1ep+11
    You generated: ('+', '1.0011101101000001111100101010101010101001110100011110', 11)
    The equivalent value is: 2522.060872394256
==> Random test case: 0.032376870018012815 [0x1.093b37bbf9830p-5]
0.032376870018012815
0x1.093b37bbf9830p-5
    You generated: ('+', '1.0000100100111011001101111011101111111001100000110000', -5)
    The equivalent value is: 0.032376870018012815
==> Random test case: -8.780476919504205e-05 [-0x1.7047a72d10934p-14]
-8.780476919504205e-05
-0x1.7047a72d10934p-14
    You generated: ('-', '1.0111000001000111101001110010110100010000100100110100', -14)
    The equivalent value is: -8.780476919504205e-05
==> Random test case: -0.6187435212357904 [-0x1.3ccbf368a3a2cp-1]
-0.6187435212357904
-0x1.3ccbf368a3a2cp-1
    You generated: ('-', '1.0011110011001011111100110110100010100011101000101100', -1)
    The equivalent value is: -0.6187435212357904
==> Random test case: -3.955577677006718e-05 [-0x1.4bd1624090336p-15]
-3.955577677006718e-05
-0x1.4bd1624090336p-15
    You generated: ('-', '1.0100101111010001011000100100000001001000000011001100110', -15)
    The equivalent value is: -3.955577677006718e-05
==> Random test case: -278.09652501774036 [-0x1.1618b5dd1270dp+8]
-278.09652501774036
-0x1.1618b5dd1270dp+8
    You generated: ('-', '1.0001011000011000101101011101110100010010011100001101', 8)
    The equivalent value is: -278.09652501774036
==> Random test case: 3.193221508767435e-05 [0x1.0bdde8e50fb7cp-15]
3.193221508767435e-05
0x1.0bdde8e50fb7cp-15
    You generated: ('+', '1.0000101111011101111010001110010100001111101101111100', -15)
    The equivalent value is: 3.193221508767435e-05
==> Random test case: 1.8314821725646697e-06 [0x1.eba27f1de5e04p-20]
1.8314821725646697e-06
0x1.eba27f1de5e04p-20
    You generated: ('+', '1.1110101110100010011111110001101111001011110000000100', -20)
```

```
The equivalent value is: 1.8314821725646697e-06
```

(Passed.)

Exercise 3 (2 points). Suppose you are given a floating-point value in a base given by `base` and in the form of the tuple, `(sign, significand, exponent)`, where

- `sign` is either the character '+' if the value is positive and '-' otherwise;
- `significand` is a *string* representation in base-`base`;
- `exponent` is an *integer* representing the exponent value.

Complete the function,

```
def eval_fp(sign, significand, exponent, base):
    ...
```

so that it converts the tuple into a numerical value (of type `float`) and returns it.
For example, `eval_fp('+', '1.25000', -1, base=10)` should return a value that is close to 0.125.

One of the two test cells below uses your implementation of `fp_bin()` from a previous exercise. If you are encountering errors you cannot figure out, it's possible that there is still an unresolved bug in `fp_bin()` that its test cell did *not* catch.

```
In [15]: def eval_fp(sign, significand, exponent, base=2):
             assert sign in ['+', '-'], "Sign bit must be '+' or '-', not '{}'.".format(sign)
             assert is_valid_strfrac(significand, base), "Invalid significand for base-{}: '{}
             assert type(exponent) is int

             ###
             ### YOUR CODE HERE
             ###
             flt=eval_strfrac(significand,base)
             if sign == '-':
                 flt=(-1)*flt
             flt=flt*(base**exponent)
             return flt

In [16]: # Test: `eval_fp_test0` (1 point)

         def check_eval_fp(sign, significand, exponent, v_true, base=2, tol=1e-7):
             v_you = eval_fp(sign, significand, exponent, base)
             delta_v = v_you - v_true
             msg = "('{}', ['{}']_{{{}}}, {}) ~= {}: You computed {}, which differs by {}.".fo
             print(msg)
             assert abs(delta_v) <= tol, "Difference exceeds expected tolerance."

         # Test 0: From the videos
```

```
        check_eval_fp('+', '1.25000', -1, 0.125, base=10)

        print("\n(Passed.)")
```

```
('+', ['1.25000']_{10}, -1) ~= 0.125: You computed 0.125, which differs by 0.0.

(Passed.)
```

Here is an additional test cell that you must pass. Note that it uses your `fp_bin()` function from a previous exercise, assuming that is correct. Therefore, if it has bugs, that may cause this test to fail or give strange errors.

```
In [17]: # Test: `eval_fp_test1` -- Random floating-point binary values (1 point)
         def gen_rand_fp_bin():
             from random import random, randint
             v_sign = 1.0 if (random() < 0.5) else -1.0
             v_mag = random() * (10**randint(-5, 5))
             v = v_sign * v_mag
             s_sign, s_bin, s_exp = fp_bin(v)
             return v, s_sign, s_bin, s_exp

         for _ in range(5):
             (v_true, sign, significand, exponent) = gen_rand_fp_bin()
             check_eval_fp(sign, significand, exponent, v_true, base=2)

         print("\n(Passed.)")
```

```
-1.6200279742315505e-05
-0x1.0fcbabf32ca37p-16
('-', ['1.0000111111001011101010111111001100101100101000110111']_{2}, -16) ~= -1.6200279742315!
54.03802241091837
0x1.b04ddeb19b46ap+5
('+', ['1.1011000001001101110111101011000110011011010001101010']_{2}, 5) ~= 54.03802241091837:
3.7311473360455817e-06
0x1.f4c93f0c23c02p-19
('+', ['1.1111010011001001001111110000110000100011110000000010']_{2}, -19) ~= 3.7311473360455!
8.191109922019734
0x1.061d928e20820p+3
('+', ['1.0000011000011101100100101000111000100000100000100000']_{2}, 3) ~= 8.191109922019734:
-0.0003625015944067008
-0x1.7c1c47e2333a8p-12
('-', ['1.0111110000011100010001111110001000110011001110101000']_{2}, -12) ~= -0.00036250159440

(Passed.)
```

Exercise 4 (2 points). Suppose you are given two binary floating-point values, u and v, in the tuple form given above. That is,

```
    u == (u_sign, u_signif, u_exp)
    v == (v_sign, v_signif, v_exp)
```

where the base for both `u` and `v` is two (2). Complete the function `add_fp_bin(u, v, signif_bits)`, so that it returns the sum of these two values with the resulting significand *truncated* to `signif_bits` digits.

For example:

```
u = ('+', '1.010010', 0)
v = ('-', '1.000000', -2)
assert add_fp_bin(u, v, 7) == ('+', '1.000010', 0)  # Caller asks for a significand with 7 dig
```

and:

```
u = ('+', '1.00000', 0)
v = ('-', '1.00000', -6)
assert add_fp_bin(u, v, 6) == ('+', '1.11111', -1)  # Caller asks for a significand with 6 dig
```

(Check these examples by hand to make sure you understand the intended output.)

> *Note 0*: Assume that `signif_bits` *includes* the leading 1. For instance, suppose `signif_bits == 4`. Then the significand will have the form, `1.xxx`.

> *Note 1*: You may assume that `u_signif` and `v_signif` use `signif_bits` bits (including the leading 1). Furthermore, you may assume each uses far fewer bits than the underlying native floating-point type (`float`) does, so that you can use native floating-point to compute intermediate values.

> *Hint*: An earlier exercise defines a function, `fp_bin(v)`, which you can use to convert a Python native floating-point value (i.e., `type(v) is float`) into a binary tuple representation.

```python
In [18]: def add_fp_bin(u, v, signif_bits):
             u_sign, u_signif, u_exp = u
             v_sign, v_signif, v_exp = v

             # You may assume normalized inputs at the given precision, `signif_bits`.
             assert u_signif[:2] == '1.' and len(u_signif) == (signif_bits+1)
             assert v_signif[:2] == '1.' and len(v_signif) == (signif_bits+1)

             ###
             ### YOUR CODE HERE
             ###
             u=eval_strfrac(u_signif, base=2)*(2**u_exp)
             v=eval_strfrac(v_signif, base=2)*(2**v_exp)
             if (u_sign=='-'):
                 u=(-1)*u
             if (v_sign=='-'):
                 v=(-1)*v
             a,b,c=fp_bin(u+v)
             return (a,b[:signif_bits+1],c)
```

14

```
In [19]: # Test: `add_fp_bin_test`

         def check_add_fp_bin(u, v, signif_bits, w_true):
             w_you = add_fp_bin(u, v, signif_bits)
             msg = "{} + {} == {}: You produced {}.".format(u, v, w_true, w_you)
             print(msg)
             assert w_you == w_true, "Results do not match."

         u = ('+', '1.010010', 0)
         v = ('-', '1.000000', -2)
         w_true = ('+', '1.000010', 0)
         check_add_fp_bin(u, v, 7, w_true)

         u = ('+', '1.00000', 0)
         v = ('+', '1.00000', -5)
         w_true = ('+', '1.00001', 0)
         check_add_fp_bin(u, v, 6, w_true)

         u = ('+', '1.00000', 0)
         v = ('-', '1.00000', -5)
         w_true = ('+', '1.11110', -1)
         check_add_fp_bin(u, v, 6, w_true)

         u = ('+', '1.00000', 0)
         v = ('+', '1.00000', -6)
         w_true = ('+', '1.00000', 0)
         check_add_fp_bin(u, v, 6, w_true)

         u = ('+', '1.00000', 0)
         v = ('-', '1.00000', -6)
         w_true = ('+', '1.11111', -1)
         check_add_fp_bin(u, v, 6, w_true)

         print("\n(Passed!)")

1.03125
0x1.0800000000000p+0
('+', '1.010010', 0) + ('-', '1.000000', -2) == ('+', '1.000010', 0): You produced ('+', '1.000
1.03125
0x1.0800000000000p+0
('+', '1.00000', 0) + ('+', '1.00000', -5) == ('+', '1.00001', 0): You produced ('+', '1.00001
0.96875
0x1.f000000000000p-1
('+', '1.00000', 0) + ('-', '1.00000', -5) == ('+', '1.11110', -1): You produced ('+', '1.11110
1.015625
0x1.0400000000000p+0
('+', '1.00000', 0) + ('+', '1.00000', -6) == ('+', '1.00000', 0): You produced ('+', '1.00000
0.984375
```

15

```
0x1.f800000000000p-1
('+', '1.00000', 0) + ('-', '1.00000', -6) == ('+', '1.11111', -1): You produced ('+', '1.1111

(Passed!)
```

Done! You've reached the end of part0. Be sure to save and submit your work. Once you are satisfied, move on to part1.