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.

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

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

[[16180339887]]10=(1×1010)+(6×109)+(1×108)+⋯+(8×101)+(7×100)=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

$$[\![\texttt{100111010}]\!]_{2} = (1 \times 2^{8}) + (1 \times 2^{5}) + \cdots + (1 \times 2^{1}).$$

[[100111010]]2=(1×28)+(1×25)+⋯+(1×21).

(What is this value?)

And in general, the value of a string of $d+1$ d+1 digits in base b b is,

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

[[sdsd−1⋯s1s0]]b=∑i=0dsi×bi.

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 [ ]:
```python
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 non-integer bases outside 2-36
    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 to 10
    if base > 10 and 0 <= ord (c) - ord ('a') < base-10: return True # Letter digits for bases > 10
    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)
```

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, $[\![s]\!]_b$ [[s]]b, which would convert a string ss to its numerical value, assuming its digits are given in base bb. 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 Lab/Notebook 2.)

In [ ]:
```
def eval_strint(s, base=2):
    assert type(s) is str
    assert 2 <= base <= 36
    #
    # YOUR CODE HERE
    #
```

In [ ]:
```
# 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)
```

In [ ]:
```
# Test: `eval_strint_test1` (1 point)
check_eval_strint('100111010', 314, base=2)
```

In [ ]:
```
# Test: `eval_strint_test2` (1 point)
check_eval_strint('a205b064', 2718281828, base=16)
```

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,

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

[[3.14]]10=(3×100)+(1×10−1)+(4×10−2).

Or, in general,

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

[[sdsd−1⋯s1s0.↑s−1s−2⋯s−r]]b=∑i=−rdsi×bi.

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.

In [ ]:
```
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-{} number.".format(s, base)

    #
    # YOUR CODE HERE
    #
```

In [ ]:
```
# 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_true,
                                                                            v_you, delta_v)
    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)
```

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

In [ ]:
```
# 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 = ''
    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()
```

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 [ ]:
```
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)
```

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 [ ]:
```
# Recall: v = 16.0625 ==> v.hex() == '0x1.0100000000000p+4'
```

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) == ('-', '1.0100000000000010000000000000000000000000000000000000', 10)
```

> There are many ways to approach this problem. One we came up exploits the observation that $[\![0]\!]_{16} == [\![0000]\!]_2$[[0]]16==[[0000]]2 and $[\![f]\!]_{16} = [\![1111]\!]$[[f]]16=[[1111]] and applies an idea in this Stackoverflow post: https://stackoverflow.com/questions/1425493/convert-hex-to-binary

In [1]:
```
def fp_bin(v):
    assert type(v) is float
#
# YOUR CODE HERE
#
```

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

def check_fp_bin(v, x_true):
    x_you = fp_bin(v)
    print("""{} [{}] ==
        {}
vs. you: {}
""".format(v, v.hex(), x_true, 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)), ('+', '1.0000000000000000000000000000000000000000000000000001', 0))
```

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

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

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


(Passed!)
```

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

check_fp_bin(-1280.03125, ('-', '1.0100000000000010000000000000000000000000000000000000', 10))
check_fp_bin(6.2831853072, ('+', '1.1001001000011111101101010100010001001000011011100000', 2))
check_fp_bin(-0.7614972118393695, ('-', '1.1000010111100010111101100110100110110000110010000000', -1))
```

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.

> 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 [ ]:
```
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-{}: '{}'".format(base, significand)
    assert type(exponent) is int

    #
    # YOUR CODE HERE
    #
```

In [ ]:
```
# 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 {}.".format(sign, significand, base, exponent, v_tr
    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)
```

In [ ]:
```
# 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)
```

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)` and `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.

> *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.

In [ ]:
```
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
    #
```

In [ ]:
```
# 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)
```

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