Sample solutions

Bookmark this page

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 (https://docs.python.org/3/tutorial/floatingpoint.html) 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

$$\llbracket \mathtt{16180339887} \rrbracket_{10} = (1 \times 10^{10}) + (6 \times 10^{9}) + (1 \times 10^{8}) + \cdots + (8 \times 10^{1}) + (7 \times 10^{0}) = 16{,}180{,}339{,}887.$$

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

$$\llbracket \mathtt{100111010} \rrbracket_{2} = (1 \times 2^{8}) + (1 \times 2^{5}) + \cdots + (1 \times 2^{1}).$$

(What is this value?)

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

$$\llbracket s_d s_{d-1} \cdots s_1 s_0 \rrbracket_b = \sum_{i=0}^{d} s_i \times b^i .$$

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 i 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]:

```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-
    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)
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
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 [2]:

```
def eval_strint(s, base=2):
    assert type(s) is str
    assert 2 <= base <= 36
    ### BEGIN SOLUTION
    return int(s, base)
    ### END SOLUTION
```

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!)
```

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")
having negative indices. For instance,

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

Or, in general,

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

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 th
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 exac
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 [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-{} number.".format(s, base)
    ### BEGIN SOLUTION
    s_parts = s.split('.')
    assert len(s_parts) <= 2

    value_int = eval_strint(s_parts[0], base)
    if len(s_parts) == 2:
        r = len(s_parts[1])
        value_frac = eval_strint(s_parts[1], base) * (float(base) ** (-r))
    else:
        value_frac = 0
    return float(value_int) + value_frac
    ### END SOLUTION
```

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_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)

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 = ''
    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!)")
```

```
[6f.6ed68]_{18} ~= 123.37883368050264: You computed 123.37883368050264, which differs by 0.0.
[2h7680.c78a]_{18} ~= 5606640.689738607: You computed 5606640.689738607, which differs by 0.0.
[a.982a]_{11} ~= 10.886483163718324: You computed 10.886483163718324, which differs by 0.0.
[33b.3]_{18} ~= 1037.1666666666667: You computed 1037.1666666666667, which differs by 0.0.
[c1.958]_{13} ~= 157.7255348202094: You computed 157.7255348202094, which differs by 0.0.
[110.2]_{4} ~= 20.5: You computed 20.5, which differs by 0.0.
[qvm6s.bn7t]_{33} ~= 31972177.354672868: You computed 31972177.354672868, which differs by 0.0.
[l298f.51l]_{23} ~= 5905955.221007643: You computed 5905955.221007643, which differs by 0.0.
[tdhc7u.lqd]_{35} ~= 1543386350.6215277: You computed 1543386350.6215277, which differs by 0.0.
[no]_{28} ~= 668.0: You computed 668.0, which differs by 0.0.

(Passed!)
```

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 sigr 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 ar 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 *IEE 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 repres 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 use 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 z

> 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` (th 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))
```

```
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 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 $[\![0]\!]_{16} == [\![0000]\!]_2$ and $[\![f]\!] = [\![1111]\!]$ and

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 (https://stackoverflow.com/questions/1425493/convert-hex-to-binary)

In [12]:
```
def fp_bin(v):
    assert type(v) is float
### BEGIN SOLUTION
    sign = '-' if v < 0 else '+'
    v = abs(v).hex()[2:]
    significand, exponent = v.split('p')
    exponent = int(exponent)
    signif_lead, signif_rem = significand.split('.')

    # replace hex character with 4 digit binary literal
    signif_rem = ''.join([hex2bin(x, 4) for x in signif_rem])
    signif = signif_lead + '.' + signif_rem
    signif += '0' * (54 - len(signif))
    return sign, signif, exponent

def hex2bin(num, width=4): # Following hint...
    return bin(int(num, base=16))[2:].zfill(width)
### END SOLUTION
```

In [13]:
```
# 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))
print("\n(Passed!)")
```

```
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 [14]:
```
# 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))

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

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

6.2831853072 [0x1.921fb544486e0p+2] ==
        ('+', '1.1001001000011111101101010100010001001000011011100000', 2)
vs. you: ('+', '1.1001001000011111101101010100010001001000011011100000', 2)

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


(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.

> 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]:

```python
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, significanc
    assert type(exponent) is int

    ### BEGIN SOLUTION
    v_sign = 1.0 if sign == '+' else -1.0
    v_significand = eval_strfrac(significand, base)
    return v_sign * v_significand * (base ** exponent)
    ### END SOLUTION
```

In [16]:

```python
# 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, bas
exponent, v_true, v_you, delta_v)
    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.)
```

In [17]:

```python
# 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.)")
```

```
('+', ['1.0110000100111110111010001101100100100111010011100110']_{2}, 16) ~= 90430.90956350006: You computed 904
0.90956350006, which differs by 0.0.
('+', ['1.0111010011000000101011100110110111000000001011001111']_{2}, 16) ~= 95424.68136216256: You computed 954
4.68136216256, which differs by 0.0.
('+', ['1.0010110010101101111100101011111111011001011010001110']_{2}, 4) ~= 18.792467832003744: You computed 18.
2467832003744, which differs by 0.0.
('-', ['1.0011011000111001111001101111010110011010101010101001']_{2}, -8) ~= -0.004733675848509245: You compute
-0.004733675848509245, which differs by 0.0.
('+', ['1.1110011100110100011000101100111101010011001010111000']_{2}, -1) ~= 0.9515715482790119: You computed 0.
15715482790119, which differs by 0.0.

(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)` 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)`, s 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*: The test cell above 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 [18]:

```python
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)

    ### BEGIN SOLUTION
    u_value = eval_fp(u_sign, u_signif, u_exp, base=2)
    v_value = eval_fp(v_sign, v_signif, v_exp, base=2)
    w_value = u_value + v_value
    w_sign, w_signif, w_exp = fp_bin(w_value)
    w_signif = w_signif[:(signif_bits+1)] # May need to truncate
    return (w_sign, w_signif, w_exp)
    ### END SOLUTION
```

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!)")
```

```
('+', '1.010010', 0) + ('-', '1.000000', -2) == ('+', '1.000010', 0): You produced ('+', '1.000010', 0).
('+', '1.00000', 0) + ('+', '1.00000', -5) == ('+', '1.00001', 0): You produced ('+', '1.00001', 0).
('+', '1.00000', 0) + ('-', '1.00000', -5) == ('+', '1.11110', -1): You produced ('+', '1.11110', -1).
('+', '1.00000', 0) + ('+', '1.00000', -6) == ('+', '1.00000', 0): You produced ('+', '1.00000', 0).
('+', '1.00000', 0) + ('-', '1.00000', -6) == ('+', '1.11111', -1): You produced ('+', '1.11111', -1).

(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`.

Floating-point arithmetic

As a data analyst, you will be concerned primarily with *numerical programs* in which the bulk of the computational work involves floating-point computation. This notebook guides you through some of the most fundamental concepts in how computers store real numbers, so you can be smarter about your numbe crunching.

WYSInnWYG, or "what you see is not necessarily what you get."

One important consequence of a binary format is that when you print values in base ten, what you see may not be what you get! For instance, try running tl code below.

> This code invokes Python's `decimal` (https://docs.python.org/3/library/decimal.html) package, which implements base-10 floating-point arithmetic in software.

In [1]:

```
from decimal import Decimal
?Decimal # Asks for a help page on the Decimal() constructor
```

In [2]:

```
x = 1.0 + 2.0**(-52)

print(x)
print(Decimal(x)) # What does this do?

print(Decimal(0.1) - Decimal('0.1')) # Why does the output appear as it does?
```

```
1.0000000000000002
1.0000000000000002220446049250313080847263336181640625
5.551115123125782702118158340E-18
```

> Aside: If you ever need true decimal storage with no loss of precision (e.g., an accounting application), turn to the `decimal` package. Just be warned it might be slower. See the following experiment for a practical demonstration.

In [3]:

```
from random import random

NUM_TRIALS = 2500000

print("Native arithmetic:")
A_native = [random() for _ in range(NUM_TRIALS)]
B_native = [random() for _ in range(NUM_TRIALS)]
%timeit [a+b for a, b in zip(A_native, B_native)]

print("\nDecimal package:")
A_decimal = [Decimal(a) for a in A_native]
B_decimal = [Decimal(b) for b in B_native]
%timeit [a+b for a, b in zip(A_decimal, B_decimal)]
```

```
Native arithmetic:
212 ms ± 4.9 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

Decimal package:
583 ms ± 11.6 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

The same and not the same. Consider the following two program fragments:

Program 1:

```
s = a - b
t = s + b
```

Program 2:

```
s = a + b
t = s - b
```

Let $a = 1.0$ and $b = \epsilon_d/2 \approx 1.11 \times 10^{-16}$, i.e., machine epsilon for IEEE-754 double-precision. Recall that we do not expect these programs to return the sam value; let's run some Python code to see.

> Note: The IEEE standard guarantees that given two finite-precision floating-point values, the result of applying any binary operator to them is the same as if the operator were applied in infinite-precision and then rounded back to finite-precision. The precise nature of rounding can be controlled by so-called *rounding modes*; the default rounding mode is "round-half-to-even (http://en.wikipedia.org/wiki/Rounding)."

```
In [4]: a = 1.0
        b = 2.**(-53) # == $\epsilon_d$ / 2.0

        s1 =  a - b
        t1 = s1 + b

        s2 =  a + b
        t2 = s2 - b

        print("s1:", s1.hex())
        print("t1:", t1.hex())
        print("\n")
        print("s2:", s2.hex())
        print("t2:", t2.hex())

        print("")
        print(t1, "vs.", t2)
        print("(t1 == t2) == {}".format(t1 == t2))
```

```
s1: 0x1.fffffffffffffp-1
t1: 0x1.0000000000000p+0


s2: 0x1.0000000000000p+0
t2: 0x1.fffffffffffffp-1

1.0 vs. 0.9999999999999999
(t1 == t2) == False
```

By the way, the NumPy/SciPy package, which we will cover later in the semester, allows you to determine machine epsilon in a portable way. Just note this for now.

Here is an example of printing machine epsilon for both single-precision and double-precision values.

```
In [5]: import numpy as np

        EPS_S = np.finfo (np.float32).eps
        EPS_D = np.finfo (float).eps

        print("Single-precision machine epsilon:", float(EPS_S).hex(), "~", EPS_S)
        print("Double-precision machine epsilon:", float(EPS_D).hex(), "~", EPS_D)
```

```
Single-precision machine epsilon: 0x1.0000000000000p-23 ~ 1.1920929e-07
Double-precision machine epsilon: 0x1.0000000000000p-52 ~ 2.220446049250313e-16
```

Analyzing floating-point programs

Let's say someone devises an algorithm to compute $f(x)$. For a given value x, let's suppose this algorithm produces the value $\mathrm{alg}(x)$. One important question might be, is that output "good" or "bad?"

Forward stability. One way to show that the algorithm is good is to show that

$$|\mathrm{alg}(x) - f(x)|$$

is "small" for all x of interest to your application. What is small depends on context. In any case, if you can show it then you can claim that the algorithm is *forward stable*.

Backward stability. Sometimes it is not easy to show forward stability directly. In such cases, you can also try a different technique, which is to show that t algorithm is, instead, *backward stable*.

In particular, $\mathrm{alg}(x)$ is a *backward stable algorithm* to compute $f(x)$ if, for all x, there exists a "small" Δx such that

$$\mathrm{alg}(x) = f(x + \Delta x).$$

In other words, if you can show that the algorithm produces the exact answer to a slightly different input problem (meaning Δx is small, again in a context-dependent sense), then you can claim that the algorithm is backward stable.

Round-off errors. You already know that numerical values can only be represented finitely, which introduces round-off error. Thus, at the very least we sho hope that a scheme to compute $f(x)$ is as insensitive to round-off errors as possible. In other words, given that there will be round-off errors, if you can prove that $\mathrm{alg}(x)$ is either forward or backward stable, then that will give you some measure of confidence that your algorithm is good.

Here is the "standard model" of round-off error. Start by assuming that every scalar floating-point operation incurs some bounded error. That is, let $a \odot b$ be the exact mathematical result of some operation on the inputs, a and b, and let $\mathrm{fl}(a \odot b)$ be the *computed* value, after rounding in finite-precision. The stanc model says that

$$\mathrm{fl}(a \odot b) \equiv (a \odot b)(1 + \delta),$$

where $|\delta| \leq \epsilon$, machine epsilon.

Let's apply these concepts on an example.

Example: Computing a sum

Let $x \equiv (x_0, \ldots, x_{n-1})$ be a collection of input data values. Suppose we wish to compute their sum.

The exact mathematical result is

$$f(x) \equiv \sum_{i=0}^{n-1} x_i.$$

Given x, let's also denote its exact sum by the synonym $s_{n-1} \equiv f(x)$.

Now consider the following Python program to compute its sum:

```
In [6]: def alg_sum(x): # x == x[:n]
            s = 0.
            for x_i in x: # x_0, x_1, \ldots, x_{n-1}
                s += x_i
            return s
```

In exact arithmetic, meaning without any rounding errors, this program would compute the exact sum. (See also the note below.) However, you know that fi arithmetic means there will be some rounding error after each addition.

Let δ_i denote the (unknown) error at iteration i. Then, assuming the collection x represents the input values exactly, you can show that `alg_sum(x)` comput $\hat{s}_{n-1}$ where

$$\hat{s}_{n-1} \approx s_{n-1} + \sum_{i=0}^{n-1} s_i \delta_i,$$

that is, the exact sum *plus* a perturbation, which is the second term (the sum). The question, then, is under what conditions will this sum will be small?

Using a *backward error analysis*, you can show that

$$\hat{s}_{n-1} \approx \sum_{i=0}^{n-1} x_i (1 + \Delta_i) = f(x + \Delta),$$

where $\Delta \equiv (\Delta_0, \Delta_1, \ldots, \Delta_{n-1})$. In other words, the computed sum is the exact solution to a slightly different problem, $x + \Delta$.

To complete the analysis, you can at last show that

$$|\Delta_i| \leq (n - i)\epsilon,$$

where ϵ is machine precision. Thus, as long as $n\epsilon \ll 1$, then the algorithm is backward stable and you should expect the computed result to be close to the result. Interpreted differently, as long as you are summing $n \ll \frac{1}{\epsilon}$ values, then you needn't worry about the accuracy of the computed result compared to the true result:

```
In [7]: print("Single-precision: 1/epsilon_s ~= {:.1f} million".format(1e-6 / EPS_S))
        print("Double-precision: 1/epsilon_d ~= {:.1f} quadrillion".format(1e-15 / EPS_D))

        Single-precision: 1/epsilon_s ~= 8.4 million
        Double-precision: 1/epsilon_d ~= 4.5 quadrillion
```

Based on this result, you can probably surmise why double-precision is usually the default in many languages.

In the case of this summation, we can quantify not just the *backward error* (i.e., Δ_i) but also the *forward error*. In that case, it turns out that

$$\left|\hat{s}_{n-1} - s_{n-1}\right| \lessapprox n\epsilon\|x\|_1.$$

> **Note: Analysis in exact arithmetic.** We claimed above that `alg_sum()` is correct *in exact arithmetic*, i.e., in the absence of round-off error. You probably have a good sense of that just reading the code.
>
> However, if you wanted to argue about its correctness more formally, you might do so as follows using the technique of proof by induction (https://en.wikipedia.org/wiki/Mathematical_induction). When your loops are more complicated and you want to prove that they are correct, you can often adapt this technique to your problem.
>
> First, assume that the `for` loop enumerates each element `p[i]` in order from `i=0` to `n-1`, where `n=len(p)`. That is, assume `p_i` is `p[i]`.
>
> Let $p_k \equiv \mathrm{p}[k]$ be the k-th element of `p[:]`. Let $s_i \equiv \sum_{k=0}^{i} p_k$; in other words, s_i is the *exact* mathematical sum of `p[:i+1]`. Thus, s_{n-1} is the exact sum of `p[:]`.
>
> Let $\hat{s}_{-1}$ denote the initial value of the variable `s`, which is 0. For any $i \geq 0$, let $\hat{s}_i$ denote the *computed* value of the variable `s` immediately after

> the execution of line 4, where i = i. When i = i = 0, $\hat{s}_0 = \hat{s}_{-1} + p_0 = p_0$, which is the exact sum of p[:1]. Thus, $\hat{s}_0 = s_0$.
>
> Now suppose that $\hat{s}_{i-1} = s_{i-1}$. When i = i, we want to show that $\hat{s}_i = s_i$. After line 4 executes, $\hat{s}_i = \hat{s}_{i-1} + p_i = s_{i-1} + p_i = s_i$. Thus, the computed value $\hat{s}_i$ is the exact sum s_i.
>
> If $i = n$, then, at line 5, the value s = $\hat{s}_{n-1} = s_{n-1}$, and thus the program must in line 5 return the exact sum.

A numerical experiment: Summation

Let's do an experiment to verify that these bounds hold.

Exercise 0 (2 points). In the code cell below, we've defined a list,

```
N = [10, 100, 1000, 10000, 100000, 1000000, 10000000]
```

- Take each entry N[i] to be a problem size.
- Let t[:len(N)] be a list, which will hold computed sums.
- For each N[i], run an experiment where you sum a list of values x[:N[i]] using alg_sum(). You should initialize x[:] so that all elements have the value **0.1**. Store the computed sum in t[i].

In [8]:
```
N = [10, 100, 1000, 10000, 100000, 1000000, 10000000]

# Initialize an array t of size len(N) to all zeroes.
t = [0.0] * len(N)

# Your code should do the experiment described above for
# each problem size N[i], and store the computed sum in t[i].

### BEGIN SOLUTION
x = [0.1] * max(N)
t = [alg_sum(x[0:n]) for n in N]
### END SOLUTION

print(t)
```

```
[0.9999999999999999, 9.99999999999998, 99.9999999999986, 1000.0000000001588, 10000.000000018848, 100000.00000133
8, 999999.9998389754]
```

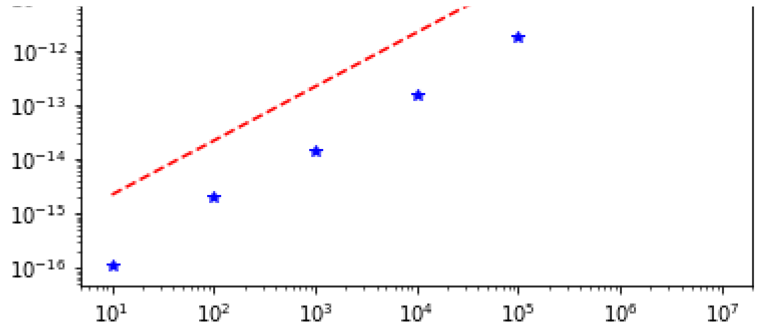
In [9]:
```
# Test: `experiment_results`
import pandas as pd
from IPython.display import display

import matplotlib.pyplot as plt
%matplotlib inline

s = [1., 10., 100., 1000., 10000., 100000., 1000000.] # exact sums
t_minus_s_rel = [(t_i - s_i) / s_i for s_i, t_i in zip (s, t)]
rel_err_computed = [abs(r) for r in t_minus_s_rel]
rel_err_bound = [ni*EPS_D for ni in N]

# Plot of the relative error bound
plt.loglog (N, rel_err_computed, 'b*', N, rel_err_bound, 'r--')

print("Relative errors in the computed result:")
display (pd.DataFrame ({'n': N, 'rel_err': rel_err_computed, 'rel_err_bound': [n*EPS_D for n in N]}))

assert all([abs(r) <= n*EPS_D for r, n in zip(t_minus_s_rel, N)])

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

```
Relative errors in the computed result:
```

	n	rel_err	rel_err_bound
0	10	1.110223e-16	2.220446e-15
1	100	1.953993e-15	2.220446e-14
2	1000	1.406875e-14	2.220446e-13
3	10000	1.588205e-13	2.220446e-12
4	100000	1.884837e-12	2.220446e-11
5	1000000	1.332883e-11	2.220446e-10
6	10000000	1.610246e-10	2.220446e-09

```
(Passed!)
```

10^{-9}
10^{-10}
10^{-11}



Computing dot products

Let x and y be two vectors of length n, and denote their dot product by $f(x, y) \equiv x^T y$.

Now suppose we store the values of x and y *exactly* in two Python arrays, `x[0:n]` and `y[0:n]`. Further suppose we compute their dot product by the progra `alg_dot()`.

In [10]:

```
def alg_dot (x, y):
    p = [xi*yi for (xi, yi) in zip (x, y)]
    s = alg_sum (p)
    return s
```

Exercise 1 (OPTIONAL -- 0 points, not graded or collected). Show under what conditions `alg_dot()` is backward stable.

> *Hint.* Let (x_k, y_k) denote the exact values of the corresponding inputs, $(\mathrm{x}[k], \mathrm{y}[k])$. Then the true dot product, $x^T y = \sum_{l=0}^{n-1} x_l y_l$. Next, let $\hat{p}_k$ denote the k-th computed product, i.e., $\hat{p}_k \equiv x_k y_k (1 + \gamma_k)$, where γ_k is the k-th round-off error and $|\gamma_k| \leq \epsilon$. Then apply the results for `alg_sum()` to analyze `alg_dot()`.

Answer. Following the hint, `alg_sum` will compute $\hat{s}_{n-1}$ on the *computed* inputs, $\{\hat{p}_k\}$. Thus,

$$\begin{aligned}\hat{s}_{n-1} &\approx \sum_{l=0}^{n-1} \hat{p}_l (1 + \Delta_l) \\ &= \sum_{l=0}^{n-1} x_l y_l (1 + \gamma_l)(1 + \Delta_l) \\ &= \sum_{l=0}^{n-1} x_l y_l (1 + \gamma_l + \Delta_l + \gamma_l \Delta_l).\end{aligned}$$

Mathematically, this appears to be the exact dot product to an input in which x is exact and y is perturbed (or vice-versa). To argue that `alg_dot` is backwar stable, we need to establish under what conditions the perturbation, $\left|\gamma_l + \Delta_l + \gamma_l \Delta_l\right|$, is "small." Since $|\gamma_l| \leq \epsilon$ and $|\Delta_l| \leq n\epsilon$,

$$\left|\gamma_l + \Delta_l + \gamma_l \Delta_l\right| \leq |\gamma_l| + |\Delta_l| + |\gamma_l| \cdot |\Delta_l| \leq (n+1)\epsilon + \mathcal{O}(n\epsilon^2) \approx (n+1)\epsilon.$$

More accurate summation

Suppose you wish to compute the sum, $s = x_0 + x_1 + x_2 + x_3$. Let's say you use the "standard algorithm," which accumulates the terms one-by-one from left-right, as done by `alg_sum()` above.

For the standard algorithm, let the i-th addition incur a roundoff error, δ_i. Then our usual error analysis would reveal that the absolute error in the computed sum, $\hat{s}$, is approximately:

$$\hat{s} - s \quad \approx \quad x_0(\delta_0 + \delta_1 + \delta_2 + \delta_3) + x_1(\delta_1 + \delta_2 + \delta_3) + x_2(\delta_2 + \delta_3) + x_3\delta_3.$$

And since $|\delta_i| \leq \epsilon$, you would bound the absolute value of the error by,

$$\left|\hat{s} - s\right| \quad \lesssim \quad (4|x_0| + 3|x_1| + 2|x_2| + 1|x_3|)\epsilon.$$

Notice that $|x_0|$ is multiplied by 4, $|x_1|$ by 3, and so on.

In general, if there are n values to sum, the $|x_i|$ term will be multiplied by $n - i$.

Exercise 2 (3 points). Based on the preceding observation, implement a new summation function, `alg_sum_accurate(x)` that computes a more accurate sum than `alg_sum()`.

> *Hint 1.* You do **not** need `Decimal()` in this problem. Some of you will try to use it, but it's not necessary.
>
> *Hint 2.* Some of you will try to "implement" the error formula to somehow compensate for the round-off error. But that shouldn't make sense to do. (Why not? Because the formula above is a *bound*, not an exact formula.) Instead, the intent of this problem is to see if you can look at the formula and understand how to interpret it. That is, what does the formula tell you?

In [11]:
```
def alg_sum_accurate(x):
    assert type(x) is list
    ### BEGIN SOLUTION
    # Idea: Each term of the error bound is $(n-i) |x_i| \epsilon$.
    # In other words, lower values of $i$ have a larger coefficient $n-i$.
    # Therefore, one idea is simply to _sort_ the data in increasing order
    # of **magnitude** (absolute value), so that the larger coefficients
    # are paired with the smaller values.
    x_sorted = sorted(x, key=abs)
    return sum(x_sorted)
    ### END SOLUTION
```

In [12]:
```
# Test: `alg_sum_accurate_test`
from math import exp
from numpy.random import lognormal

print("Generating non-uniform random values...")
N = [10, 10000, 10000000]
x = [lognormal(-10.0, 10.0) for _ in range(max(N))]
print("Range of input values: [{}, {}]".format(min(x), max(x)))

print("Computing the 'exact' sum. May be slow so please wait...")
x_exact = [Decimal(x_i) for x_i in x]
s_exact = [float(sum(x_exact[:n])) for n in N]
print("==>", s_exact)

print("Running alg_sum()...")
s_alg = [alg_sum(x[:n]) for n in N]
print("==>", s_alg)

print("Running alg_sum_accurate()...")
s_acc = [alg_sum_accurate(x[:n]) for n in N]
print("==>", s_acc)

print("Summary of relative errors:")
ds_alg = [abs(s_a - s_e) / s_e for s_a, s_e in zip(s_alg, s_exact)]
ds_acc = [abs(s_a - s_e) / s_e for s_a, s_e in zip(s_acc, s_exact)]
display (pd.DataFrame ({'n': N,
                        'rel_err(alg_sum)': ds_alg,
                        'rel_err(alg_sum_accurate)': ds_acc}))

assert all([r_acc < r_alg for r_acc, r_alg in zip(ds_acc[1:], ds_alg[1:])]), \
       "The 'accurate' algorithm appears to be less accurate than the conventional one!"

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

```
Generating non-uniform random values...
Range of input values: [2.4988313527996013e-28, 4.508417815653759e+17]
Computing the 'exact' sum. May be slow so please wait...
==> [0.34441634937208654, 59286881538197.42, 1.5616792932459144e+18]
Running alg_sum()...
==> [0.34441634937208654, 59286881538197.09, 1.5616792932375094e+18]
Running alg_sum_accurate()...
==> [0.34441634937208654, 59286881538197.42, 1.5616792932459144e+18]
Summary of relative errors:
```

‹ Previous

Next ›

Legal

Terms of Service & Honor Code
Privacy Policy
Accessibility Policy
Trademark Policy
Sitemap

Connect

Blog
Contact Us
Help Center
Media Kit
Donate

GET IT ON Google play
Download on the App Store