アルゴリズム入門　＃2

地引　昌弘

2021.10.14

はじめに

今回は、次の二つを目標とします:

- コンピュータ上での数値の表現について理解し、誤差が生じる原因を説明できる。
- 整数 · 実数の使い分けや計算精度に留意したプログラムを書けるようになる。

注意: インデント (字下げ) について

本格的なアルゴリズム · プログラムを作成するに当たり、全体の見通しを良くするインデント (字下げ) について、少し触れておきます。例えば、文書を作成する場合は、一般に各行を左端に揃えて記述します。

```
def test
.....
.....
.....
.....
.....
end
```

しかし、プログラムでは、このような記述をすることはありません。下記のように、制御構造 (Python での複合文) に応じて適宜インデントを入れた記述をします。

```
def test
  .....
  .....
    .....
    .....
  .....
end
```

その理由は、次の通りです。文章を読む場合は、基本的に始めから終わりまで一方通行であり、逆戻りしたり飛び越したりして読むことは多くありません。しかし、プログラムでは、制御構造に合わせて制御の流れが逆戻りしたり飛び越したりすることが頻繁に発生し、また変数など様々な定義も、構造に合わせて有効範囲が決まります。そのため、「どこからどこまでの範囲が、どの構造に属しているか」を明示しておかないと、その読解が非常に難しくなるからです。

Python では、これを (強制的に?) 実践させるため、制御構造に応じたブロックの境界を、多くの言語で使われている “{” と “}” や “begin” と “end” ではなく、インデントの有無で示します[1]。

[1] プログラムの構造を示すのに、{ } や begin · end とインデントのどちらが分かり易いかは、基本的に各人の主観によりますが、Python 言語の設計 · 開発者は、インデントの方がプログラムを一目見た時にその構造を直感的に把握し易いと考えたのかも知れません。Python のインデントについては、これまでも様々な場所で賛否両論が繰り広げられて来ました。Python 言語の設計 · 開発者は、これに辟易したようで、インデントの話は聞きたくないと述べています。

1　前回の演習問題の解説

1.1　演習 1-3a — 四則演算を試す

演習 1-3a は、四則演算の関数を作るというものでした。まずは、和の計算です。

```
def add(x, y):
  return(x + y)
```

動かしているところを見てみましょう:

```
>>> add(3.5, 6.8)
10.3
>>>
```

後は、四則計算毎に上と同じものを作ればよいわけですが、和 · 差 · 商 · 積のための関数を四つ作る代わりに、一つで済ませる方法を考えてみましょう。まずは、先に説明したような関数の最後に一つだけ値を返す (計算する) のではなく、四つの計算を順次行ない、その都度結果を表示する方法をお見せします:

```
def shisoku0(x, y):
  print(x+y)
  print(x-y)
  print(x*y)
  print(x/y)
```

動かしているところは次の通り:

```
>>> shisoku0(3.3, 4.7)
8.0
-1.4000000000000004
15.51
0.702127659574468
>>>
```

shisoku0 関数では、print 関数を用いて各計算結果を順次表示した後、return 関数を記述しないで終了しています。この場合、shisoku0 関数からは、結果として返す値がないことを示す None (何もないことを示す値) が処理系の内部で返されています (処理系の内部なので、そのままでは外から見えません)。

上の方法だと、「最終的な計算結果が返る」わけではないのがちょっと、という気がするかも知れません。そこで次は、複数の数値を同時に返してくれる方法を見てみましょう。Python の return 関数は、複数の戻り値を返すことができます。return 関数に複数の引数 (各引数が戻り値になります) を書くだけです。これを利用した「四則演算」の関数を以下に示します。

```
def shisoku1(x, y):
  return(x+y, x-y, x*y, x/y)
```

実行しているところは次の通り:

```
>>> shisoku1(3.3, 4.7)
(8.0, -1.4000000000000004, 15.51, 0.702127659574468)
>>>
```

確かに、簡単に四つの数値が返されていますね。但し、これら複数の戻り値を利用する場合は、少し注意が必要です。上の実行例を見てみると、四つの戻り値が “(” と “)” で括られ、一つになっています。Python では、このように複数の値を “(” と “)” で括り、一つにしたものを**タプル** (Tuple) と呼びます。タプルは一見、値が複数個あるように見えますが、これらは一つにまとめられているため、各値を利用する際は、そのまとまりをほどく必要があります。例えば、次のような関数を考えてみましょう:

```
def plus_minus(x, y):
  return(x+y, x-y)

def multi(x, y):
  return(x*y)
```

これらを用いて $(5+3)(5-3)$ を計算する場合、以下のようにするとエラーになってしまいます:

```
>>> multi(plus_minus(5, 3))
Traceback (most recent call last):
  File "<pyshell#34>", line 1, in <module>
    multi(plus_minus(5, 3))
TypeError: multi() missing 1 required positional argument: 'y'
>>>
```

上のエラーは、multi 関数の実行に必要な引数 y が渡されていないことを意味しています。これを解決するには、plus_minus 関数の先頭に "$*$" を付けて戻り値のタプルをほどき、各値毎に分けてから multi 関数へ渡します:

```
>>> multi(*plus_minus(5, 3))
16
>>>
```

但し、関数の先頭に "$*$" を付けて戻り値のタプルをほどけるのは、戻り値を別の関数の引数に直接使う時だけなので注意して下さい (以下のような使い方はできません):

```
>>> *plus_minus(5, 3)
SyntaxError: can't use starred expression here
>>>
```

"$*$" を使わずに戻り値のタプルをほどく場合は、戻り値の個数に等しい変数を用意して、各値毎に代入します。こんな感じ:

```
>>> x, y = plus_minus(5, 3)
>>> multi(x, y)
16
>>>
```

以上、これからも様々な場面に出て来るので、戻り値が複数ある場合の扱いに慣れておいて下さい。

1.2 演習 1-3b — 剰余演算

演習 1-3b は剰余演算「%」を試すというものでした。

```
def jouyo(x, y):
  return(x % y)
```

実行してみましょう:

```
>>> jouyo(8, 5)
3
>>> jouyo(20, 5)
0
>>> jouyo(-8, 5)
2
>>> jouyo(-21, 5)
4
>>>
```

ところで、プログラムに渡す数値として、プラスの数だけではなくマイナスの数も試していただけたでしょうか? ここで「マイナスだとどうだろう」と考えるようになって頂きたいわけです (こんな感じに目が行き届くようになると、プログラムの誤り (バグ) も見通せるようになって来ます)。実行結果を見ると、割られる数 (被除数) がマイナスの時も剰余は負になりません。では、割る数 (除数) がマイナスだったらどうでしょうか?

```
>>> jouyo(8, -5)
-2
>>> jouyo(-8, -5)
-3
>>>
```

さて、この違いはどこから来るのでしょうか。剰余演算とは、割り算の関係式 $m \div n = q \cdots r$ ($m = q \cdot n + r$) より r (剰余) を求める演算です。r が満たすべき条件ついて考えてみると、まず頭に浮かぶのが、m と n が共に正の整数の場合に成り立つ $0 \leq r < n$ という関係でしょう。ここで、除数 n を負の整数に拡張した場合は、どうなるでしょうか。割り算の定義は、除数の逆数[2]を掛けることなので、答えは数を有理数に拡張すれば必ず求めることができます。では、答えを整数に限定した場合、余りはどうなるでしょうか。$n < 0$ なので、$0 \leq r < n < 0$ という関係は変ですね。除数 n が正整数の場合との整合性を考え、$0 \leq r < |n|$ としておくのが良さそうに見えます。

では、$m = 7, n = -3$ とした次の例はどうでしょうか (つまり、$7 \div (-3)$ の関係式です)。

```
7 = (－2)(－3) ＋ 1  ..... (1)
7 = (－3)(－3) － 2  ..... (2)
7 = (－4)(－3) － 5  ..... (3)
7 = (－5)(－3) － 8  ..... (4)
```

どの式も数学的には正しい式ですが、式 (3), (4) のような余りを認めてしまうと、一組の被除数 · 除数に対する余りの種類が無限になってしまうので、式 (3) 以降の余りは考えないことにしましょう。問題は式 (2) です。式 (2) の余りは、$0 \leq r < |n|$ を満たしませんが、これを $0 \leq |r| < |n|$ と拡張すれば満たします。そこで問題なのですが、余りが満たす条件を $0 \leq |r| < |n|$ と拡張してはならない合理的な理由は何かあるでしょうか。割り算の定義が、数の拡張に伴い素朴なものから先ほど述べた数学的なものへ拡張されたのと同様、余りについても $m = q \cdot n + r$ を満たす r (の範囲) を (必要に応じて) 限定できればよいのではないでしょうか。このような議論を背景に、余りは、それを利用する場面に応じた適当な制約を入れて使われているのが現状です。コンピュータによる計算では、計算速度や小数点表示の都合により、$-\frac{n}{2} \leq r < \frac{n}{2}$ を満たす r を剰余として計算する流儀もあります。負数を用いた剰余演算は、プログラミング言語によって結果が異なるため、注意が必要です (事前に言語仕様を調査する or 負数の使用を避けるなど)。

1.3 演習 1-3c — 円錐の体積

演習 1-3c は円錐の体積でした。底面の半径 r、高さ h として、まず円錐の底面の面積は πr^2。体積はこれに高さを掛けて 3 で割ればできます:

```
def cornvol(r, h):
  return((r**2*3.1416*h) / 3.0)
```

因みに、「`**`」は**べき乗** (Power) の演算子です。もちろん 2 乗は「`r*r`」と書いても構いません。

```
>>> cornvol(3.0, 4.0)
37.6992
>>>
```

ところで「**円周率** (Circle Ratio) が 3.1416 というのは不正確だ」と考える人もいそうですね。しかし、(この後で詳しく説明しますが) コンピュータ上の計算は「電卓での計算」と同様、有限の桁数でしか行なえないので、計算をする際は、自分で必要と考える適当な桁数を決めてその範囲でやるしかないわけです。ここでは、有効数字 5 桁を選択しています[3]。

[2]数学的に正式な呼び名は逆元と言います

[3]実際には、3.141592653589793 程度まで扱える精度があるので、この定数をその都度書くのは嫌だという人のために `math.pi` と記号で表わせるようになっています。同様に、**自然対数の底** (Base of Natural Logarithm) e は `math.e` で表わせます。但し、これらの記号を使う場合は、プログラム内でこれらを参照している箇所より前に “import math” と記述し、事前に math ライブラリを読み込んでおく必要があります。

1.4 演習 1-3d — 四則演算の精度を調べる

演習 1-3d は、数学的には同じ計算でも、コンピュータ上での計算はどうなるのか、表示される計算結果の桁数を変えながら比較してみるというものでした。以下のプログラムを作成し、その結果を見てみます。まずは 10 桁:

```
def calc1(x):
  print("%.10g" % (x / 10.0))
  print("%.10g" % (x * 0.1))
```

実行結果は以下の通り:

```
>>> calc1(7)
0.7
0.7
>>>
```

次に、表示される計算結果の桁数を 20 桁にしてみます:

```
def calc2(x):
  print("%.20g" % (x / 10.0))
  print("%.20g" % (x * 0.1))
```

これの実行結果は以下になります:

```
>>> calc2(7)
0.69999999999999995559
0.70000000000000006661
>>>
```

あれ? 少し変ですね。今度は 30 桁にして様子を見てみましょう:

```
def calc3(x):
  print("%.30g", x / 10.0)
  print("%.30g", x * 0.1)
```

計算の精度 (正しさ) が良くなったようには見えません。

```
>>> calc3(7)
0.699999999999999955591079014994
0.700000000000000066613381477509
>>>
```

数学的には、10 で割ることと 0.1 を掛けることは同じ計算ですが、コンピュータの計算では、その結果に差が見られます。また、表示する桁数だけ増やしても計算の精度 (正しさ) は変わらないようです。では、次の計算はどうでしょう:

```
>>> print("%.20g" % (1.0/3.0))
0.33333333333333331483
>>>
```

本来ならば、「0.33333…」と 3 が無限に続くわけですが、どうもコンピュータは無限を扱えず、ある決まった桁数 (精度) までしか計算できないようです。その結果、ある決まった精度以降の値は、このように無効な値 (誤った値) になっています。

2　コンピュータ上での数値の表現

2.1　十進表現と二進表現

コンピュータが作られた当時の主要な目的は、人間に代わって文字通り「計算」を高速に/大量に/正確に行なうことでした。このため、コンピュータで最初に扱われたデータの種類は**数値** (Numerical Value) でした。数を表現する方法としては、**アラビア数字** (Arabic numerals — 0〜9 の数字) を用いた**位取り記法** (Positional Notation) が圧倒的に多く使われています。私達が使う**十進表現** (Decimal Representation) ないし**十進法** (Decimal System) の位取り記法では、数字として 0〜9 までの 10 種類で全ての数を書き表わし、その値は桁が 1 増えるごとに 10 倍になります。

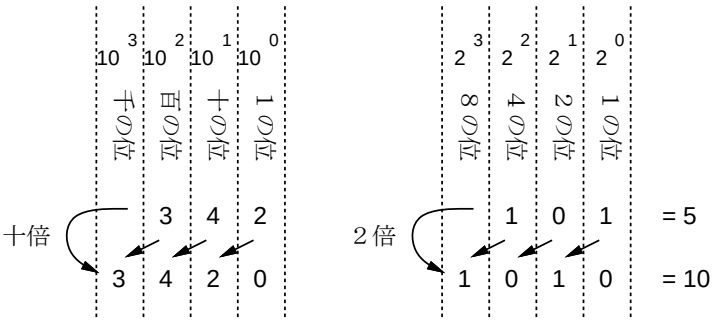


図 1: 十進法と二進法

例えば、図 1 左にある十進法の「342」は「一が 2 個、十が 4 個、百が 3 個」という意味であり、下にゼロをつけるとそれが「十が 2 個、百が 4 個、千が 3 個」となるので、全体として 10 倍になるわけです。一般に、4 桁の十進法で表記した数 *abcd* は次の式で表わせます。

$$a \times 10^3 + b \times 10^2 + c \times 10^1 + d \times 10^0$$

ところで、「10」という値は特別ではなく、別の数を用いることもできます。この、位取りの基準となる数を**基数** (Radix) と呼びます。我々が基数として「10」を使っている (十進表現を使っている) のは、単なる偶然 (両手の指を合わせると 10 本あるから) と言われています。コンピュータでは、主に**二進表現** (Binary Representation) ないし**二進法** (Binary System) が使われます[4]。これは、コンピュータの実装に使う電子回路において、「電流が流れている/いない」「電圧がある/ない」など二つの状態を持たせる回路が作り易いためです。二進表現では、数値として「0、1」の 2 種類を用い、1 桁右に行く毎に 2 倍の数を表わすことになります。例えば、図 1 右の「101」は「一が 1 個、二が 0 個、四が 1 個」を表わすため、その値は 5 です。これの右に 0 を付けて 1 桁ずらすことは 2 倍することを意味するので、その値は 10 になるわけです。一般に、4 桁の二進法で表記した数 *abcd* は次のように解釈できます。

$$a \times 2^3 + b \times 2^2 + c \times 2^1 + d \times 2^0$$

二進法をもっとイメージし易くするには、図 2 (次ページ) のように「1」「2」「4」「8」「16」…個の○が描かれたカードが並んでいて、その中から 1 に対応するカードのみ拾って○の数を合計する、と考えると良いかも知れません。

2.2　負数の表現と二の補数

上で説明した二進表現では、N ビットの場合、0〜$2^N - 1$ までの範囲の数を表わせます。これを (負の数が含まれないという意味で) **符号なし二進表現** (Unsigned Binary Representation) と呼びます。しかし、コンピュータでの計算では、負の数も当然必要です。そのため、1 ビットを**符号ビット** (Sign Bit) として用い、正負の数をともに扱うような表現方法が複数作られました。ここではその中から、現在の大半のコンピュータで採用されている**二の補数表現** (Two's Complement Representation) について説明します。

[4] 二進表現 · 十進表現された数のことを**二進数** (Binary Numbers) · **十進数** (Decimal Numbers) と呼ぶ流儀もありますが、数そのものはどのように表記しても同じ数なはずなので、これは厳密に言えばおかしい言葉遣いだと言えます。また、数学では素数 p に対する「p **進数** (p-adic Number)」という用語を全く別の意味で用いています。

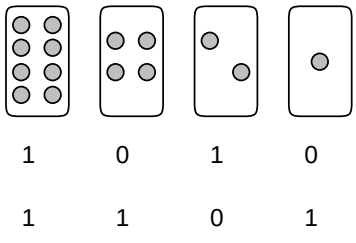


図 2: 1、2、4、8…個の○が描かれたカード

二の補数表現とは、二進表現を応用して負数を表現する記法です。例えば、3 ビットで表現できる数字を考えましょう。この場合、符号なし二進表現では 0〜7 の値が表現され、二の補数では −4〜3 の値が表現されます。3 ビットの二の補数表現を用いて −3 を表わすには、以下のような手順に従います。

a. −3 の絶対値である 3 を二進表現 011 で表わす。

b. 3 ビットより 1 桁大きい 4 ビットの最小数 1000 を考え、二進表現の引き算 1000 − 011 を計算する (桁の繰り下がりは十進表現の引き算と同じ)。

c. 具体的には、4 桁目の 1 を繰り下げ、3 桁目:1, 2 桁目:1, 1 桁目:10 (十進表現では 2) として、1 桁目から引き算を行なう。

3 ビットの符号なし二進表現と二の補数の対応は、図 3 のようになっています。

値	二進	二の補数
7	111	
6	110	
5	101	
4	100	
3	011	011
2	010	010
1	001	001
0	000	000
-1		111
-2		110
-3		101
-4		100

図 3: 3 ビットの二の補数表現

二の補数表現の特徴として、符号なし二進表現の計算と同じ回路で (単に最上位からの桁上がりを無視するだけで) 負の数を含んだ計算がそのまま行なえる、という点が挙げられます。例えば、「$-2+3=1$」は「$110+011=(1)001$」となり、確かに最上位の桁上がりを無視する点以外は符号なし二進表現と同じ計算で行なえています。また、**符号反転** (Negation — マイナス 1 を掛けること) の操作は、「各ビットの 0·1 を反転してから 1 を足す」操作で行なえます。例えば、3 は「011」なので、その 0·1 を反転して「100」、さらに 1 を足すと「101」となり、これも確かに −3 を二の補数表現で表わしたものになっています。逆も一応示しておくと、「101」→「010」→「011」で確かに元の 3 に戻ります。

符号なしの整数についても、二の補数表現の整数についても、整数という本来は無限個あるものの中から、与えられたビット数で表わせる有限の範囲を「切り取って」表現しているため、演算の結果が表わせる範囲を超えてしまうと正しくない結果が得られることになります。具体的には、「正の数と正の数を足したのに負の数になった」などの誤りが起こります。このような、扱える範囲を越える演算を行なったために結果が不正になることを、一般に**溢れ** (Overflow) と呼びます。また、二の補数では負数を 0 以上の数より 1 個多く表わせるため (図 3 を確認)、「符号を反転したのにまた元の数に戻ってしまう」数が存在することになります (図 3 にある二の補数のうち、どの数か分かりますか)。この場合も

符号反転時に溢れが起きていると言えます。コンピュータで数値を扱う時は、このようなことを常に意識しておく必要があります。

さて、以上の説明は多くのプログラミング言語 (C、C++、Java など) に当ててはまるのですが (これらの言語では主に 32 ビットの二の補数表現が使われています)、Python ではちょっと事情が違います。上で示した問題や限界は、あくまでも「ビット数の上限が決まっている」ことに起因するものでした。これを克服するため、**Python では整数値の演算結果がある標準のビット数以内で表わせなくなった場合、適宜ビット数を増やして表わせる範囲を自動的に広げる**仕様になっています。このため、Python では、ビット数の限界に伴う整数計算の不正などに困ることがなくなりますが、その代わり「数が大きくなるにつれて計算に要する時間も増える」といった副作用も生じるので、やはり「数学の数とは違う」と意識しておくことは必要です。

2.3 実数の表現と浮動小数点

ここまでは「正負の整数」を扱ってきましたが、数にはもちろん小数点付きの数もあります。数学の世界では**整数** (Integral Number) は**実数** (Real Number) の特別な場合として含まれるわけですが、コンピュータ上で数を表現する場合は、整数と実数では全く違った性質を持っていて、プログラムの上でもはっきりと区別されます。

整数と実数の違いが目立つ例の一つとして、整数同士の割り算があります。一般にコンピュータの世界は、数学の世界と少し異なり、全ての有理数をそのまま扱うことができません。分数で表現される数は、余りを無視した整数の商として扱われる (例えば、$1/4 = 0$)、あるいは実数として扱われる (同じく、$1/4 = 0.25$) のどちらかになります (ここでは割り算を取り上げましたが、根号を開く場合など、同様な状況は数多くあります)。例えば、Python に似た Ruby と呼ばれる言語では、計算に用いる数値の種類に応じて、自動的に切り替えてくれます。「10 を 3 で割る」例を見てみましょう (以下は、Ruby の処理系である irb による計算結果です):

```
irb> printf "%.30g", 10/3          ← 両方とも整数だと
3                                  ← 切捨ての割り算
=> nil
irb> printf "%.30g", 10.0/3        ← 片方が実数なら
3.33333333333333348136306995002    ← 実数の割り算
=> nil
```

Python は、Ruby と異なり、計算結果が整数以外になる場合は全て実数として扱われます。

```
>>> print("%.30g" % (10/3))        ← 両方とも整数だけど
3.33333333333333348136306995002    ← 実数の割り算として計算される
>>> print("%.30g" % (10.0/3))
3.33333333333333348136306995002    ← 上の結果を実数の割り算と比較してみましょう
```

但し、状況によっては、余りを無視した整数の商として解を得たい場合もあるため、実数を整数に変換する int 関数が用意されています (これとは逆に、整数を実数に変換する関数として float 関数があります)。int 関数は、小数点以下を全て切り捨てます。

```
>>> print("%.30g" % int(10/3))
3
>>> int(-3.2)
-3
```

上の例では、int 関数が 10/3 の整数商として 3 を返しています[5]。

では次に、実数を具体的に有限のビット数で実数を表わす方法について、考えてみましょう。例えば、8 桁を用いて数を表わす場合、下 4 桁で小数点以下を表わし、上 4 桁で小数点以上を表わすと決めることで、小数点付きの数が表わせるという考え方があります:

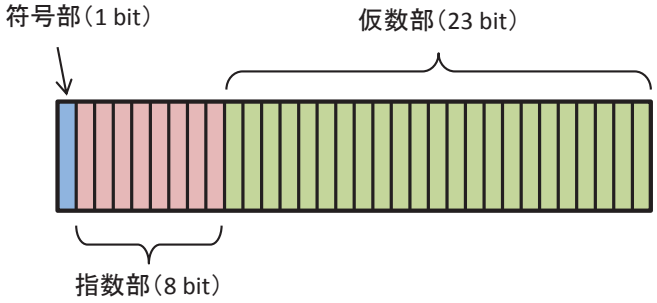
□□□□. □□□□

[5]参考ですが、print 関数の書式として例えば%f (浮動小数点数の 10 進表記) を指定すると、小数点以下も 30 桁が表示されます (つまり、3.00000000000000000000000000000 が表示されます)。

このような考え方を、小数点が決まった位置に固定されていることから**固定小数点** (Fixed Point) による実数表現と呼びます。しかし実際には、この方法はあまりうまく行きません。なぜならば、科学技術計算では頻繁に「30,000,000」や「0.0000001」といった数値が出て来るため、この方法では扱える数の範囲が狭過ぎるからです[6]。

科学の世界では、このような大きい数値や小さい数値を扱う場合、上のような表現ではなく、「3×10^8」や「1×10^{-6}」といった記法を用います。つまり、一つの数値を**指数** (Exponent — 桁取り/上記の 8 や -6) と**仮数** (Mantissa — 有効数字/上記の 3 や 1) に分けて扱うことで、広い範囲の数値を柔軟に扱うわけです。この方法は、指数によって小数点の位置を動かすものと考えて、**浮動小数点** (Floating Point) と呼ばれています。例えば、上と同じ 8 桁で十進法の数を表わす場合、6 桁の有効数字と 2 桁の指数に分けた浮動小数点表現を用いると、表わせる絶対値の最も大きい数は「$\pm\ 9.99999 \times 10^{99}$」、0 でない絶対値の最も小さい数は「$0.00001 \times 10^{-99}$」となり、ずっと広い範囲の数を扱えます。

注意! コンピュータでは「小さい字」が使えないので、伝統的に指数部分を「e ±指数」で表わします (e は exponent の e)。例えば、「3.0×10^{22}」であれば「3.0e+22」です。このような表示は「エラー」などではないので、注意して下さい。



43.75 を IEEE 754 規格(単精度)で表してみると‥

まずは、仮数部が二進表現で 1.xxxx となるように調整する。
43.75 = 43 + 0.75 と分解し、整数部/小数部をそれぞれ二進表現すると 101011.11
これを 1.xxxx という形にすると、1.0101111 × 2^5

次に、仮数部は先頭が必ず 1 なので、これを省略して左詰めで表現
指数部は、補数表示にすると大小比較に一手間必要となるので、+127 しておく。
符号部は、正数 = 0/負数 = 1

以上より、43.75 の IEEE 754 表現は、
符号 = 0, 指数部 = 10000100, 仮数部 = 01011110000000000000000

図 4: 浮動小数点表現の例 (IEEE 754 規格/単精度)

実際には、コンピュータでは二進法を利用するため、これを十進表現ではなく二進表現で行なっています (図 4 / 2.4 節の説明も適宜置き換えて下さい)。多くのプログラミング言語における実数データ型では、符号 1 ビット、仮数部 52 ビット、指数部 (符号含む) 11 ビット、合計 64 ビットの浮動小数点表現が使われています (このビットの割り当ては、**IEEE 754** と呼ばれる標準に従ったものです)[7]。

注意! 仮数部の取り扱いは、誤差を説明する際の鍵になるので、仮数部の範囲について少し補足しておきます。N 進法の浮動小数点表現を用いる場合、まずは、仮数部が 1 以上 N 未満 (例えば、二進法だと 1.xxx..、十進法だと 9.xxx..) になるよう補正されます。次に、整数部分は無視し、小数部分.xxx.. だけが実際に格納されます。

[6] これに対する一つの解として、小数点以上・以下をそれぞれ整数として扱い、2.2 節の最後で述べたように、適宜ビット数 (桁数) を増やして両者が表わせる範囲を自動的に広げることで、一見対応できそうに見えます。しかしながら、例えば二つの実数 x.12345 と x.123 を考えてみると、両者の小数点以下 12345 と 123 を、そのまま整数として扱うわけには行きません。これ以外にも、扱いの難しい例として x.00123 と x.123 などが考えられます。もう少し粘り、まずは標準として小数点以下を N 桁で表わすと決めておき、必要に応じて各桁に 0 を入れるという規則を作ることはできます (例えば、標準を 6 桁と決めれば、x.00123 の小数点以下は 001230 と表わされます)。そして、小数点以下が N 桁を超える場合は、通常の整数と同様に適宜桁数を増やせばよいというわけです。確かに、これならば規則上の破綻はなさそうですが、一つの実数を桁数が可変な二つの整数で表わすとなると、今度は計算時間が増えるという副作用が深刻になって来ます。つまり、実用性の観点からは、有効数字の桁数と計算時間とのバランスが重要というわけです。もちろん、絶対的に厳密な計算が必要な場合は存在しますが、そのような事例は多くないので、その場合は必要に応じて上で述べたような規則を作れば (or 適用すれば) 済むわけです。ここでの議論は、あらゆる計算に適用する規則として必要かどうか、ということが主題です。

[7] 図 4 の例は、全体を 32 ビットで表現する単精度規格です。全体を 64 ビットで表現したものを倍精度規格と呼びます。また、図 4 で指数部を+127 する理由ですが、もう少し詳しく説明すると、補数表示は正数・負数の両方を表現できることから補数表示自体が正負の符号を持っており、これは符号部とは別の符号が存在することを意味するため、そのままでは大小比較ができないからです。

2.4 浮動小数点と誤差

浮動小数点を用いた実数表現には、整数の表現とはまた違った注意点があります。まず、有効数字は当然ながら有限なので、その範囲で表わせない結果の細かい部分は**丸め** (Rouding — 十進表現で言えば四捨五入) が行なわれ、**丸め誤差** (Round-off Error) となります。言い替えれば、コンピュータによる実数計算は基本的に近似値による計算を行なっていることになります。例えば、図 5 の右側にある計算結果は、十進表現で考えると $1 \div 5$ の計算をしているので 0.2 です。これを 0.1×2^1 と変形し (図 5 右端にある "$1 \div 10 = 0.1$" は、この 0.1 を作るという意味です)、二進表現を考えてみると、仮数部の十進表現 0.1 は二進法だと無限小数になってしまうため、丸め誤差が発生するというわけです。

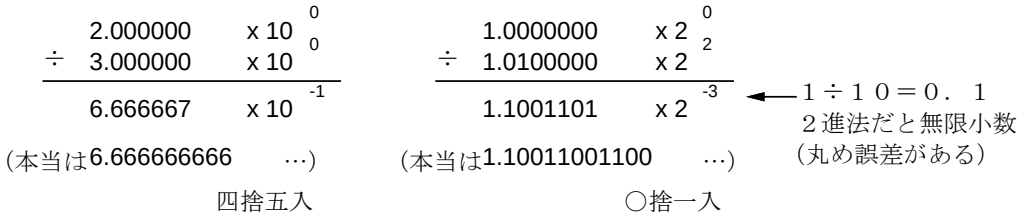


図 5: 丸め誤差

また、絶対値が大きく異なる二つの数を足したり引いたりすると、絶対値が小さい方の数値にある下の桁は (演算のために大きい数値の桁数に揃えられた結果) 捨てられてしまい、これも誤差の原因となります。これを**情報落ち** (Loss of Information) と言います。極端な例として、演算した結果が元の (絶対値が大きい方の) 数のまま、ということも起こります。これは、例えば図 6 左のような例を思い浮かべてみれば分かると思います。

逆に、非常に値が近い数値同士を引き算する場合も、誤差が大きくなります。コンピュータによる実数 (N 進法の浮動小数点数) 計算では、前にも述べた通り計算結果の仮数部が 1 以上 N 未満になるよう補正されます (例えば、十進表現であれば 1.0000.. ～ 9.9999..、二進表現であれば 1.0000.. ～ 1.1111..)。仮数部が 1 より小さい場合は、仮数部を不足の桁に応じて単純に N^n 倍することで補正します。非常に値が近い数値同士を引き算すると、仮数部が 1 より大幅に小さくなるため不足の桁が大きくなってしまいます。その結果、補正幅が大きくなる (つまり、単純に N^n 倍する n が増える) ために、誤差が大きくなるわけです (図 6 右)。これを**桁落ち** (Cancellation) と言います。

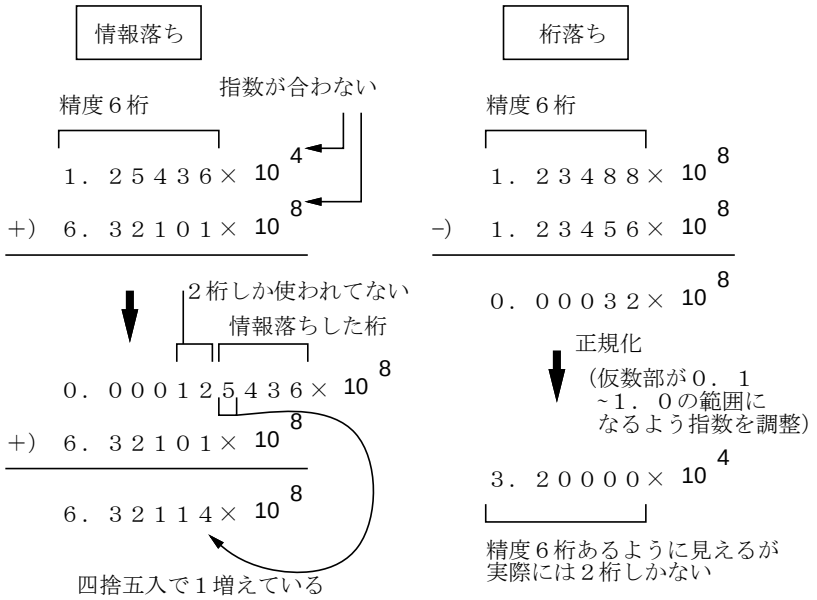


図 6: 情報落ちと桁落ち

素朴に計算すると桁落ちが問題になる例として、次の計算式を考えてみましょう。

$$\sqrt{x+1}-1$$

x が 0 に近いとき、$\sqrt{x+1}$ も 1 に近いので桁落ちが起きます。これを避けるためには、次のように変形します。

$$\sqrt{x+1}-1=\frac{(\sqrt{x+1}-1)(\sqrt{x+1}+1)}{\sqrt{x+1}+1}=\frac{x}{\sqrt{x+1}+1}$$

この変形により、引き算を消すことができるので、桁落ちから逃れられるわけです。ところで、このように変形してみると、$x \to 0$ の時にこの式はおよそ $\frac{x}{2}$ だと予想されますね。実際に両方の式で計算し、確認してみましょう[8]:

```
import math

def calc1(x):
  return(math.sqrt(x + 1.0) - 1.0)

def calc2(x):
  return(x / (math.sqrt(x + 1.0) + 1.0))
```

最初の素朴版から見てみます。

```
>>> calc1(0.00000000001)
5.000000413701855e-12
>>> calc1(0.000000000001)
5.000444502911705e-13
>>> calc1(0.0000000000001)
4.9960036108132044e-14
>>> calc1(0.00000000000001)
4.884981308350689e-15
>>> calc1(0.000000000000001)
4.440892098500626e-16
```

x が小さくなると、どんどん $\frac{x}{2}$ から外れて行きます。では修正版ではどうでしょうか。

```
>>> calc2(0.00000000001)
4.9999999999875e-12
>>> calc2(0.000000000001)
4.99999999999875e-13
>>> calc2(0.0000000000001)
4.999999999999876e-14
>>> calc2(0.00000000000001)
4.999999999999988e-15
>>> calc2(0.000000000000001)
4.999999999999999e-16
```

確かにこちらは大丈夫です。

最後に一つ、浮動小数点表現そのものに関する注意をしておきます。整数では全てのビット パターンを数値の表現として使っていましたが、浮動小数点では指数部と仮数部の組み合わせ方に制約があるので (例えば、仮数部が 0 であれば値が 0 なので指数部には意味がなく、この時は指数部も 0 にしておく)、これを利用して正負の**無限大** (Infinity — $\pm\infty$) や**非数** (NaN — Not a Number) などの特別な値を用意しています。また、0 にも「+0」と「−0」があったりします。だから、演算の結果として、これらの変な値が表示されても驚かないようにして下さい。

[8] x の**平方根** (Square Root) は **math.sqrt**(x) で計算できます。但し、プログラム内で math.sqrt 関数を利用する箇所より前に “import math” と記述し、事前に math ライブラリを読み込んでおく必要があります。

演習 2-1 整数の計算と実数の計算において、除算以外で結果が違う計算の例を Python で作成せよ (`print` 関数の表示桁数を増やしてみることを薦めます)。どのような場合に違いが現れるか。

> ヒント: 「123451234512345 + 1」は「123451234512346」ですね。では、「123451234512345.0 + 1.0」はどうでしょうか。また「12345」をもう 1 回増やすとどうでしょうか。これらの変化が生じる原因を考えてみましょう。

演習 2-2 実数の演算では、既に見たように、「ある数を 10 で割る」[9]場合と「ある数に 0.1 を掛ける」場合とでは、結果の異なる例が存在する。しかし、割り算とその逆数の掛け算とで、常に結果が同じとなる場合も存在する。このような計算の例を Python で作成せよ。また、その理由について考察せよ。

> ヒント: 例えば、「8 で割る」場合と「0.125 を掛ける」場合はどうでしょうか。「10 で割る」「0.1 を掛ける」場合と「8 で割る」「0.125 を掛ける」場合とでは、何が異なっているのでしょうか。

演習 2-3 次の計算をしたところ、左式と右式の値は等しくならなかった。それぞれについて、丸め誤差 · 桁落ち誤差 · 情報落ち誤差のうち、どの誤差が生じているのかを説明しなさい。

a. `7 / 10 != 7.0 * 0.1`

b. `(100000000.0 + 1.0)**2 != 100000000.0**2 + 2*100000000.0 + 1.0`

c. `1234567890.12345 − 1234567890.0 != 0.12345`

演習 2-4 複素数 $x = a+bi$ を考える。複素数 x の絶対値を求める abs 関数および、二つの複素数 $x1 = a1+b1i$, $x2 = a2+b2i$ を加算する plus 関数を下記のように作成した。

```
import math

def abs(a, b):
  val = math.sqrt(a*a + b*b)
  return(val)

def plus(a1, b1, a2, b2):
  val_a = a1 + a2
  val_b = b1 + b2
  return(val_a, val_b)
```

plus 関数と abs 関数を利用し、複素数 10^8+1 を二つの複素数 $(10^8, 1)$ に分解して $|10^8+1|$ を abs(*plus(10.0**8, 0.0, 1.0, 0.0)) と計算した結果は[10]、(当然ながら) abs(10.0**8, 0.0) とは異なる値であった。これに対し、10^8+i を二つの複素数 $(10^8, i)$ に分解して $|10^8+i|$ を abs(*plus(10.0**8, 0.0, 0.0, 1.0)) と計算したところ、今度は abs(10.0**8, 0.0) と同じ値が得られた。まずは、plus 関数と abs 関数を作成し、様々な場合について (例えば、数値を変える/両関数を個別に動かしてみる等) 実際に確かめてみなさい。次に、この違いが生じる理由について考察しなさい。

[9]前にも述べたように、Python では整数同士の割り算による商が有理数になる場合は、実数として扱われます。それに対し Ruby などの言語では、余りを切り捨てた整数商として扱われます。Ruby のような言語で誤差を調べる場合は、10 ではなく 10.0 にする必要があります。

[10]複数の戻り値を別の関数の引数に直接使う場合の書き方を確認しておいて下さい。