2.3 ビット操作

指数関数の計算をしてみると分かることだが、計算結果は爆発的に大きくなってしまう。しかし **Python3** では心配はいらない。メモリが許す限り、いくらでも大きな値が扱えるからだ。いまは 2^x の小道を歩いているのだが、2^x の小道はもう一本ある。つまり、2^x の計算の仕方に別の方法があるということだ。

Python3 が内部で 2 進数計算をしていることは述べた。2 進数というのはコンピュータではよく使われる、1001011 や 11111011 といった数のことだ。われわれは普段 10 進数を使っているので、これらの数が 77 や 251 だということに気付かないものだ。コンピュータは 1 バイトという単位を使うが、1 バイトは 8 ビットである。え？　何を言ってるの分からないって？　つまりこういうことだ。1 ビットとは 0 か 1 だけが使える桁のことである。だから 8 ビットとは 0 か 1 が使える桁が 8 個あることを指している。そしてこれが 1 バイトになる。要するに 1 バイトで 00000000〜11111111 までの数が扱えるのだ。

説明を簡単にするため、ここでは数の下 4 ビットだけ調べておこう。本質的には何桁になっても同じなので、大きな数は各自で想像してほしい。さて少し、10 進数と 2 進数の対応を見てみよう。

10 進数	0	1	2	3	4	5	6	7	8	9	⋯
2 進数	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	⋯

これを見て、2 進数のもっとも特徴的な性質を見抜けたら立派である。その性質は、2 倍ごとに桁がずれていく、というものだ。つまり、$1 \to 2 \to 4 \to 8$ と倍々になると、2 進数は

$$00000001 \to 00000010 \to 00000100 \to 00001000$$

になる。$3 \to 6$ を見ても確かに 00000011 → 00000110 となっている。まあ、当たり前といえばそうである。10 倍ごとに桁がずれるのが 10 進数であるから、2 倍ごとに桁がずれればそれは 2 進数なのである。結局、2 倍することと桁がひとつずれることが同じなので、2^n 倍することと桁を左へ n だけずらすことが同じになる。

Python3 にはうまい具合に、桁をずらす（**シフトする**）演算子が用意されている。左へずらす演算子は “<<” である。`p << 4` と書けば `p` を 4 ビット左へシフト—すなわち 16 倍—することになる。もちろん、右へシフトする演算子 “>>” もある。

それではシフト操作による 2^x を求めるスクリプトを書いておこう。

[py script]

```
>>> def powof2(x):
...   p = 1
...   while x:
...     p <<= 1
...     x -= 1
...   return p
...
>>> powof2(10)
1024
```

早い話、`p *= 2` を `p <<= 1` に変えただけである。`p <<= 1` は `p = p << 1` のことだ。ビット操作のついでに、2 進数 と 10 進数に関わる誤差について話しておこう。**Python3** が小数の計算をする際は 2 進数でするので、10 進数で表示すると誤差がでる、と前に言ったことについての話だ。

10 進数で $abc.de$ なる数があれば、a は 10^2 の位、b は 10 の位、c は 1 の位、d は $\frac{1}{10} = 0.1$ の位、e は $\frac{1}{10^2} = 0.01$ の位である。これが 2 進数の $abc.de$ であれば、a は 2^2 の位、b は 2 の位、c は 1 の位、d は $\frac{1}{2} = 0.5$ の位、e は $\frac{1}{2^2} = 0.25$ の位となるのである。

その上で 10 進数の 0.1 を例にとろう。もちろん割り切れている小数で、0.1 の位に 1 がある数である。2 進数ではどう表せるだろうか。2 進数で $0.abcde\cdots$ で表される数は、a は 0.5 の位、b は 0.25 の位、c は 0.125 の位、d は 0.0625 の位、e は 0.03125 の位、... になっている。例に出した 10 進数は 0.1 だから、0.5 の位から 0.125 の位までは数字がないはずだ。これらの桁に 1 があれば、その数は 0.1 より大きくなってしまうからだ。0.0625 の位には 1 がある。0.03125 の位にも 1 がある。これで 0.09375 になった。次は 0.015625 の位だが、この桁には 1 はない。あれば 0.1 を超えてしまうからだ。

さて、このようなことを続けていけば分かるが、この操作でちょうど 0.1 にすることはできない。これ以上詳しく調べないけれど、10 進数の 0.1 は 2 進数では $0.000110011\cdots$ となる。つまり、無限小数なのだ。要するに小数を扱う限り、10 進数と 2 進数はぴったり同じ値を扱っているわけではないのである。そんな事情から、**Python3** で小数を扱うときは注意をしなくてはいけない。ちなみに、`print` 関数で値を表示する場合は上手に 10 進数に直してくれる。しかし、`return` 文で値を返す場合は 2 進数でそのまま返すので気をつけよう。

少し散歩道からそれてきたようだ。そのまま別の小道へ入ってみよう。

散歩のレベルが 3 になれば、数学ではもっとも有名な数の話題になると予想できるだろう。それにも関係するので、2^x 以外のべき乗の計算で伏線を張っておきたい。少し唐突だけれど $\left(1+\frac{1}{n}\right)^n$ の値を考えてみよう。$n=1$ のとき、この値は 2 である。$n=2$ なら 2.25 だが $n=3$ から先の計算

は厄介である。さあ、**Python3** の出番だ。

[py script]
```
>>> def npow(n):
...   for n in range(1, n):
...     print(pow(1 + 1/n, n))
...
>>> npow(10)
2.0
2.25
2.37037037037
2.44140625
2.4883199999999994
2.5216263717421135
2.546499697040712
2.565784513950348
2.5811747917131984
```

たいしたものである。わずか 3 行のスクリプトを書けば適当なところまでの計算ができてしまうのだ[1]。もっとも、この手の計算をわざわざスクリプトで書いて関数にすることはない。実際、

[py script]
```
>>> print(pow(1+1/1000000, 1000000))
2.7182804690957534
```

で即座に答が出る。スクリプトにしたのは、n が大きくなっても $\left(1+\frac{1}{n}\right)^n$ がべらぼうに大きな値にならない様子を見るためである。

ここの `for` 文がいままでと違うことに気づいただろうか。繰り返しをカウントする `n` が `for` ブロックの中でも使われている点である。すなわち、繰り返しの値 `n` が、`pow` 関数の `n` に連動する。一方で、`npow(n)` の引数 `n` は `range(1, n)` の `n` に引き渡されるので、スクリプトは正しく動作する。もし、見た目が紛らわしいと思えば、`for m in range(1, n):` および `pow(1 + 1/m, m)` とすればよいだろう。

関数に与える `n` の値を大きくして試してみれば分かるように、ある一定の値を表示するように感じるだろう。結論を言えば、この計算はある値—もちろん今回の散歩道にふさわしい値だ—に**収束**することが知られている。しかも数学では大変重要な値になっている。これが何なのかは、ずっと先まで散歩する必要がある。楽しみを先延ばしにして悪いが、もとの散歩道へ戻ることにしよう。

[1] もしここで結果に `1` ばかり表示されたら、version 2.x を使用している。`print(pow(1 + 1./n, n))` と直すか、version 3.x を起動し直そう。