15. 浮点算术：争议和限制

浮点数在计算机硬件中表示为以 2 为基数（二进制）的小数。举例而言，十进制的小数

```
0.125
```

等于 1/10 + 2/100 + 5/1000 ，同理，二进制的小数

```
0.001
```

等于0/2 + 0/4 + 1/8。这两个小数具有相同的值，唯一真正的区别是第一个是以 10 为基数的小数表示法，第二个则是 2 为基数。

不幸的是，大多数的十进制小数都不能精确地表示为二进制小数。这导致在大多数情况下，你输入的十进制浮点数都只能近似地以二进制浮点数形式储存在计算机中。

用十进制来理解这个问题显得更加容易一些。考虑分数 1/3 。我们可以得到它在十进制下的一个近似值

```
0.3
```

或者，更近似的，：

```
0.33
```

或者，更近似的，：

```
0.333
```

以此类推。结果是无论你写下多少的数字，它都永远不会等于 1/3 ，只是更加更加地接近 1/3 。

同样的道理，无论你使用多少位以 2 为基数的数码，十进制的 0.1 都无法精确地表示为一个以 2 为基数的小数。 在以 2 为基数的情况下，1/10 是一个无限循环小数

```
0.0001100110011001100110011001100110011001100110011...
```

在任何一个位置停下，你都只能得到一个近似值。因此，在今天的大部分架构上，浮点数都只能近似地使用二进制小数表示，对应分数的分子使用每 8 字节的前 53 位表示，分母则表示为 2 的幂次。在 1/10 这个例子中，相应的二进制分数是 `3602879701896397 / 2 ** 55`，它很接近 1/10 ，但并不是 1/10 。

大部分用户都不会意识到这个差异的存在，因为 Python 只会打印计算机中存储的二进制值的十进制近似值。在大部分计算机中，如果 Python 想把 0.1 的二进制对应的精确十进制打印出来，将会变成这样

```
>>> 0.1
0.1000000000000000055511151231257827021181583404541015625
```

这比大多数人认为有用的数字更多，因此Python通过显示舍入值来保持可管理的位数

```
>>> 1 / 10
0.1
```

牢记，即使输出的结果看起来好像就是 1/10 的精确值，实际储存的值只是最接近 1/10 的计算机可表示的二进制分数。

有趣的是，有许多不同的十进制数共享相同的最接近的近似二进制小数。例如， `0.1` 、 `0.10000000000000001` 、`0.1000000000000000055511151231257827021181583404541015625` 全都近似于 `3602879701896397 / 2 ** 55` 。由于所有这些十进制值都具有相同的近似值，因此可以显示其中任何一个，同时仍然保留不变的 `eval(repr(x)) == x` 。

在历史上，Python 提示符和内置的 `repr()` 函数会选择具有 17 位有效数字的来显示，即 `0.10000000000000001`。 从 Python 3.1 开始，Python（在大多数系统上）现在能够选择这些表示中最短的并简单地显示 `0.1` 。

请注意这种情况是二进制浮点数的本质特性：它不是 Python 的错误，也不是你代码中的错误。 你会在所有支持你的硬件中的浮点运算的语言中发现同样的情况（虽然某些语言在默认状态或所有输出模块下都不会 *显示* 这种差异）。

想要更美观的输出，你可能会希望使用字符串格式化来产生限定长度的有效位数:

```
>>> format(math.pi, '.12g')  # give 12 significant digits
'3.14159265359'

>>> format(math.pi, '.2f')   # give 2 digits after the point
'3.14'

>>> repr(math.pi)
'3.141592653589793'
```

必须重点了解的是，这在实际上只是一个假象：你只是将真正的机器码值进行了舍入操作再 *显示* 而已。

一个假象还可能导致另一个假象。 例如，由于这个 0.1 并非真正的 1/10，将三个 0.1 的值相加也不一定能恰好得到 0.3:

```
>>> .1 + .1 + .1 == .3
False
```

而且，由于这个 0.1 无法精确表示 1/10 的值而这个 0.3 也无法精确表示 3/10 的值，使用 `round()` 函数进行预先舍入也是没用的:

```
>>> round(.1, 1) + round(.1, 1) + round(.1, 1) == round(.3, 1)
False
```

虽然这些小数无法精确表示其所要代表的实际值， `round()` 函数还是可以用来“事后舍入”，使得实际的结果值可以做相互比较:

```
>>> round(.1 + .1 + .1, 10) == round(.3, 10)
True
```

二进制浮点运算会造成许多这样的意外。 有关 "0.1" 的问题会在下面的 "表示性错误" 一节中更精确详细地描述。 请参阅 浮点数的危险性 了解有关其他常见意外现象的更完整介绍。

正如那篇文章的结尾所言，“对此问题并无简单的答案。” 但是也不必过于担心浮点数的问题！ Python 浮点运算中的错误是从浮点运算硬件继承而来，而在大多数机器上每次浮点运算得到的 2**53 数码位都会被作为 1 个整体来处理。 这对大多数任务来说都已足够，但你确实需要记住它并非十进制算术，且每次浮点运算都可能会导致新的舍入错误。

虽然病态的情况确实存在，但对于大多数正常的浮点运算使用来说，你只需简单地将最终显示的结果舍入为你期望的十进制数值即可得到你期望的结果。 `str()` 通常已足够，对于更精度的控制可参看 格式字符串语法 中 `str.format()` 方法的格式描述符。

对于需要精确十进制表示的使用场景，请尝试使用 `decimal` 模块，该模块实现了适合会计应用和高精度应用的十进制运算。

另一种形式的精确运算由 `fractions` 模块提供支持，该模块实现了基于有理数的算术运算（因此可以精确表示像 1/3 这样的数值）。

如果你是浮点运算的重度用户则你应当了解一下 NumPy 包以及由 SciPy 项目所提供的许多其他数字和统计运算包。 参见 <https://scipy.org>。

Python 也提供了一些工具，可以在你真的 *想要* 知道一个浮点数精确值的少数情况下提供帮助。 例如 `float.as_integer_ratio()` 方法会将浮点数表示为一个分数:

```
>>> x = 3.14159
>>> x.as_integer_ratio()
(3537115888337719, 1125899906842624)
```

由于这是一个精确的比值，它可以被用来无损地重建原始值:

```
>>> x == 3537115888337719 / 1125899906842624
True
```

`float.hex()` 方法会以十六进制（以 16 为基数）来表示浮点数，同样能给出保存在你的计算机中的精确值:

```
>>> x.hex()
'0x1.921f9f01b866ep+1'
```

这种精确的十六进制表示法可被用来精确地重建浮点值:

```
>>> x == float.fromhex('0x1.921f9f01b866ep+1')
True
```

由于这种表示法是精确的，它适用于跨越不同版本（平台无关）的 Python 移植数值，以及与支持相同格式的其他语言（例如 Java 和 C99）交换数据.

另一个有用的工具是 `math.fsum()` 函数，它有助于减少求和过程中的精度损失。 它会在数值被添加到总计值的时候跟踪“丢失的位”。 这可以很好地保持总计值的精确度， 使得错误不会积累到能影响结果总数的程度:

```
>>> sum([0.1] * 10) == 1.0
False
>>> math.fsum([0.1] * 10) == 1.0
True
```

15.1. 表示性错误

本小节将详细解释 "0.1" 的例子，并说明你可以怎样亲自对此类情况进行精确分析。 假定前提是已基本熟悉二进制浮点表示法。

表示性错误 是指某些（其实是大多数）十进制小数无法以二进制（以 2 为基数的计数制）精确表示这一事实造成的错误。 这就是为什么 Python（或者 Perl、C、C++、Java、Fortran 以及许多其他语言）经常不会显示你所期待的精确十进制数值的主要原因。

为什么会这样？ 1/10 是无法用二进制小数精确表示的。 目前（2000年11月）几乎所有使用 IEEE-754 浮点运算标准的机器以及几乎所有系统平台都会将 Python 浮点数映射为 IEEE-754 “双精度类型”。 754 双精度类型包含 53 位精度，因此在输入时，计算会尽量将 0.1 转换为以 *J*/2***N* 形式所能表示的最接近分数，其中 *J* 为恰好包含 53 个二进制位的整数。 重新将

```
1 / 10 ~= J / (2**N)
```

写为

```
J ~= 2**N / 10
```

并且由于 *J* 恰好有 53 位 (即 `>= 2**52` 但 `< 2**53`)，*N* 的最佳值为 56:

```
>>> 2**52 <=  2**56 // 10  < 2**53
True
```

也就是说，56 是唯一的 *N* 值能令 *J* 恰好有 53 位。 这样 *J* 的最佳可能值就是经过舍入的商:

```
>>> q, r = divmod(2**56, 10)
>>> r
6
```

由于余数超过 10 的一半，最佳近似值可通过四舍五入获得:

```
>>> q+1
7205759403792794
```

这样在 754 双精度下 1/10 的最佳近似值为:

```
7205759403792794 / 2 ** 56
```

分子和分母都除以二则结果小数为:

```
3602879701896397 / 2 ** 55
```

请注意由于我们做了向上舍入，这个结果实际上略大于 1/10；如果我们没有向上舍入，则商将会略小于 1/10。 但无论如何它都不会是 *精确的* 1/10！

因此计算永远不会“看到”1/10：它实际看到的就是上面所给出的小数，它所能达到的最佳 754 双精度近似值:

```
>>> 0.1 * 2 ** 55
3602879701896397.0
```

如果我们将该小数乘以 10**55，我们可以看到该值输出为 55 位的十进制数:

```
>>> 3602879701896397 * 10 ** 55 // 2 ** 55
1000000000000000055511151231257827021181583404541015625
```

这意味着存储在计算机中的确切数值等于十进制数值 0.1000000000000000055511151231257827021181583404541015625。 许多语言（包括较旧版本的 Python）都不会显示这个完整的十进制数值，而是将结果舍入为 17 位有效数字:

```
>>> format(0.1, '.17f')
'0.10000000000000001'
```

`fractions` 和 `decimal` 模块可令进行此类计算更加容易:

```
>>> from decimal import Decimal
>>> from fractions import Fraction

>>> Fraction.from_float(0.1)
Fraction(3602879701896397, 36028797018963968)

>>> (0.1).as_integer_ratio()
(3602879701896397, 36028797018963968)

>>> Decimal.from_float(0.1)
Decimal('0.1000000000000000055511151231257827021181583404541015625')

>>> format(Decimal.from_float(0.1), '.17')
'0.10000000000000001'
```