15 محاسبات ممیز شناور(Floating-point) : **موضوعات و محدودیت ها:** اعداد اعشاری ممیز شناور در سخت افزار کامپیوتر بر اساس کسر مبنای ۲ (binary) نمایش داده می شوند. برای مثال کسر اعشاری 0.125، دارای مقادیر 1/10 + 2/100 + 5/1000 است. به همین ترتیب کسر دو دویی 0.001 ، مقادیر 1/8 + 0/4 + 0/2 را دارد. این دو کسر مقادیر یکسانی دارند، تنها تفاوت آن ها در این است که اولی در نمایش کسری مبنای ۱۰، و دومی در مبنای ۲ نوشته شده است.

متاسفانه بیشتر کسر های اعشاری را نمی توان دقیقا به صورت کسر دو دویی نمایش داد. به طور کلی یکی از عواقب آن، این است که اعداد اعشاری ممیز شناور که وارد می کنید، تنها توسط اعداد دودویی ممیز شناور که واقعا در ماشین ذخیره شده اند، تقریب زده می شوند.

درک مسئله ابتدا در مبنای ۱۰ ساده تر است. کسر 1/3 را در نظر بگیرید. می توانید آن را به عنوان کسر مبنای ۱۰ به صورت 0.3 در نظر بگیرید. یا بهتر 0.33 ، یا بهتر 0.333 و غیره. صرف نظر از اینکه چند رقم را بنویسید، هرگز دقیقا 1/3 نمی شود، اما تقریب بهتری از 1/3 می شود.

به همین ترتیب، صرف نظر از هر تعداد رقم مبنای ۲ که استفاده کنید، مقدار اعشاری 0.1 را نمی توان در کسر مبنای ۲ دقیقا نمایش داد. در مبنای ۲ ، 1/10 یک کسر تکرار شونده تا بی نهایت است

```
0.0001100110011001100110011001100110011001100110011...
```

اگر در هر تعداد محدودی از بیت ها توقف کنید، باز یک تقریب بدست می آورید. امروزه در بیشتر ماشین ها، اعداد float با استفاده از یک کسر دو دویی که صورت کسر با استفاده از ۵۳ بیت اول که از پر ارزش ترین بیت شروع می شود، و مخرج کسر به عنوان یک توان ۲ است، تقریب زده می شود. در مثال 1/10 ، کسر دودویی 55 ** 2 / 3602879701896397 است که نزدیک اما نه دقیقا برابر با مقدار واقعی 1/10 است.

بسیاری از کاربران به دلیل شیوه نمایش مقادیر، از تقریب زدن بی اطلاع هستند. پایتون تنها یک تقریب اعشاری از مقدار واقعی اعشار مربوط به تقریب دودویی ذخیره شده در ماشین را چاپ می کند. در بیشتر ماشین ها، اگر پایتون میخواست مقدار واقعی اعشار مربوط به تقریب دودویی ذخیره شده برای 0.1 را چاپ کند، باید خروجی زیر را نمایش دهد.

```
>>> 0.1
0.1000000000000000055511151231257827021181583404541015625
```

این تعداد ارقام بیشتر از چیزی است که اکثر مردم مفید می پندارند، بنابراین، پایتون با نمایش یک مقدار گرد شده(به جای عدد اصلی)، تعداد ارقام را قابل مدیریت می کند.

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

به یاد داشته باشید، حتی با وجود اینکه نتیجه چاپ شده شبیه مقدار دقیق 1/10 به نظر می رسد، مقدار واقعی ذخیره شده، نزدیک ترین کسر دودویی قابل نمایش است.

با کمال تعجب، اعداد اعشاری مختلف بسیاری وجود دارند که نزدیک ترین کسر دودویی تقریبی را به صورت مشترک دارند. برای مثال، اعداد 0.1 و 0.10000000000000001 و

0.1000000000000000055511151231257827021181583404541015625 ، همه به

55 ** 2 / 3602879701896397 تقریب زده می شوند. از آنجایی که تمامی این مقادیر اعشاری، تقریب یکسانی را به اشتراک می گذارند، هر یک از آنها، در زمانی که هنوز ثابت x == eval(repr(x)) را نگه می دارد، می تواند نمایش داده شود.

بر اساس تاریخچه، prompt پایتون و تابع داخلی repr() ، یک ۱ به همراه ۱۷ رقم پر ارزش را انتخاب خواهد کرد: 0.10000000000000001 . شروع با پایتون 3.1، امروزه پایتون روی اکثر سیستم ها قادر به انتخاب کوتاهترین گزینه است و 0.1 را نمایش می دهد.

توجه داشته باشید، این در ذات ممیز شناور دودویی است: این نه یک اشکال در پایتون و نه یک اشکال در کد شما است. شما همین مسئله را در همه زبان هایی که از محاسبات ممیز شناور سخت افزار شما پشتیبانی می کند، پیدا خواهید کرد (اگر چه ممکن است برخی از زبان ها این تفاوت ها را به صورت پیش فرض (یا در تمامی حالات خروجی) نمایش ندهند.).

برای داشتن خروجی خوشایندتر، ممکن است بخواهید از قالب بندی رشته برای تولید تعداد محدودی از ارقام پر ارزش استفاده کنید.

```
>>> 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'
```

مهم است که توجه داشته باشید، در واقع این یک توهم است: شما دارید نمایش مقدار واقعی ماشین را گرد می کنید.

یک توهم ممکن است دیگری را به وجود آورد. برای مثال، از آنجایی که 0.1 (اعشار) دقیقا 1/10 (کسر) نیست، حاصل جمع سه مقدار 0.1 دقیقا 0.3 نمی شود، یا :

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

همچنین، از آنجایی که 0.1 نمی تواند بیشتر از این به مقدار دقیق 1/10 نزدیک شود، و نیز 0.3 نمی تواند بیشتر از این به مقدار دقیق 3/10 نزدیک شود، از قبل گرد کردن (pre-rounding) با استفاده از تابع `round()` نمی تواند کمکی کند.

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

با وجود اینکه اعداد به مقادیر دقیق مورد نظر نمی توانند نزدیک تر شوند، تابع `round()` برای بعد-گرد کردن (post-rounding) مناسب است، در نتیجه، نتایج با مقادیر دقیق، قابل مقایسه با یکدیگر خواهند بود.

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

محاسبات ممیز شناور دودویی شگفتی های بسیاری مانند این دارد. مشکل مربوط به "0.1"، با جزییات دقیق در ادامه در بخش نمایش خطا، توضیح داده شده است. بخش مخاطرات ممیز شناور (The Perils of Floating Point) را برای گزارش های کامل تری از شگفتی های متداول ببینید.

همانطور که در انتها گفته می شود، ' پاسخ های آسانی وجود ندارد'. با این وجود به طور نا عادلانه مراقب نقطه شناور نباشید. خطاهای موجود در عملیات float پایتون، از سخت افزار ممیز شناور به ارث رسیده است، و در اکثر ماشین ها از مرتبه کمتر از ۱ بخش در 2**53 ، به ازای هر عملیات است. این برای بیشتر وظایف، بیشتر از کافی است، اما باید به یاد داشته باشید که این محاسبات اعشاری نیست(decimal arithmetic) و هر عملیات float می تواند از یک خطای گرد کردن جدید متضرر شود.

تا زمانی که موارد آسیب شناسی وجود دارد، برای بیشتر کاربردهای معمول محاسبات ممیز شناور، اگر به سادگی نمایش نتایج نهایی خود را به تعداد ارقام اعشاری که انتظار دارید گرد کنید، نتایج مورد انتظارتان را در انتها خواهید دید. `str()` معمولا کافی است، و برای کنترل دقیق تر، قالب متد `str.format()` که در نحو قالب رشته (Format String Syntax) مشخص شده است را ببینید.

برای کاربردهایی که به نمایش دقیق اعشار نیاز دارند، از ماژول `decimal` استفاده کنید که محاسبات اعشاری را به طور مناسب برای برنامه های کاربردی حسابداری و برنامه های کاربردی با دقت بالا، پیاده سازی می کند.

نوع دیگری از محاسبات دقیق توسط ماژول `fractions` پشتیبانی می شود که محاسبات را بر اساس اعداد گویا (rational numbers) پیاده سازی می کند (بنابراین اعدادی مانند 1/3 می توانند به طور دقیق نمایش داده شوند).

اگر شما به شدت از عملیات ممیز شناور استفاده می کنید، باید به پکیج اعداد پایتون (Numerical Python package) و بسیاری از پکیج های دیگر که برای عملیات ریاضی و آماری، توسط پروژه SciPy آماده شده است، نگاهی بیاندازید. لینک <https://scipy.org> را ببینید.

پایتون ابزارهایی را برای کمک در آن مواقع نادری که شما می خواهید مقدار دقیق یک float را بدانید، فراهم کرده است.

متد `float.as_integer_ratio()` مقدار یک float را به صورت یک کسر بیان می کند.

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

از آنجایی که این نسبت دقیق است، می توان از آن برای بازگرداندن بی ضرر مقدار اصلی استفاده کرد:

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

متد `float.hex()` یک float را در مبنای ۱۶ بیان می کند، و مقدار دقیق ذخیره شده در کامپیوتر شما رو می دهد:

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

از این نمایش دقیق مبنای ۱۶ می توان برای بازسازی مقدار دقیق float استفاده کرد.

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

از آنجایی که نمایش دقیق است، برای انتقال قابل اطمینان مقادیر بین نسخه های مختلف پایتون(مستقل از پلتفرم)، و تبادل داده با سایر زبان هایی که از همین قالب مشابه پشتیبانی می کنند(مانند Java و C99) مفید است.

یک ابزار مفید دیگر، تابع `math.fsum()` است که به کاهش دقت در زمان حاصلجمع کمک می کند. این تابع، ارقام گم شده ("lost digits") را به عنوان مقادیری که به حاصلجمع در حال اجرا اضافه می شوند، پیگیری می کند. این می تواند در دقت کلی، تفاوت ایجاد کند، بنابراین خطاها، تا جایی که روی حاصلجمع نهایی اثر گذارند، رو هم انباشته نمی شوند.

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

15.1 نمایش خطا: این بخش مثال "0.1" را با جزییات توضیح می دهد و نشان می دهد چگونه می توانید یک ارزیابی دقیق از مواردی مانند این داشته باشید. فرض شده است با نمایش ممیز شناور دودویی آشنایی اولیه دارید.

نمایش خطا به این حقیقت اشاره دارد که برخی(در واقع اکثر) از کسر های اعشاری را نمی توان دقیقا به عنوان کسر دودویی (مبنای ۲) نمایش داد. این اصلی ترین دلیل برای این است که چرا پایتون (یا Perl, C ,C++,Java, Fortran و بسیاری زبان های دیگر) اغلب عدد اعشاری دقیقی که شما انتظار دارید را نمایش نمی دهد.

چرا این طور است؟؟؟ 1/10 دقیقا توسط یک کسر دودویی قابل نمایش نیست. امروزه (November 2000) تقریبا تمامی ماشین ها از محاسبات ممیز شناور IEEE-754 استفاده می کنند، و تقریبا همه پلتفرم ها، float های پایتون را به دقت دوبرابر (IEEE-754("double precision" نگاشت می کنند. 754 doubles شامل ۵۳ بیت دقت است، بنابراین در ورودی، کامپیوتر تلاش می کند تا 0.1 را به نزدیک ترین کسری که می تواند در قالب *J/2**N* که J یک عدد صحیح(integer) شامل دقیقا ۵۳ بیت است، تبدیل کند. این را:

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

به صورت

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

باز نویسی می کنیم، و یاد آوری می کنیم که J دقیقا ۵۳ بیت دارد (is >= 2**52 but < 2**53)، و بهترین مقدار برای N ، ۵۶ است.

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

۵۶ تنها مقدار برای N است که باعث می شود J دقیقا ۵۳ بیت باشد. پس بهترین مقدار ممکن برای J خارج قسمت گرد شده است:

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

از آنجایی که باقی مانده بیشتر از نصف ۱۰ است، بهترین تقریب با گرد کردن رو به بالا به دست می آید:

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

بنابراین، بهترین تقریب ممکن برای 1/10 در دقت دو برابر ۷۵۴ (754 double precision) این است:

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

تقسیم صورت و مخرج کسر به ۲ کسر را به مقدار زیر کاهش می دهد:

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

توجه داشته باشید، از آنجایی که رو به بالا گرد کرده ایم، این در واقع کمی بزرگتر از 1/10 است. اگر رو به بالا گرد نکرده بودیم، خارج قسمت کمی کوچکتر از 1/10 می شد. اما در هیچ شرایطی دقیقا 1/10 نمی شود.

بنابراین کامپیوتر هرگز 1/10 را نمی بیند: چیزی که می بیند، کسر دقیقی است که در بالا داده شده است، بهترین تقریب 754 double که می تواند داشته باشد.

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

اگر کسر را در 10**55 ضرب کنیم، می توانیم مقدار خروجی را در ۵۵ رقم اعشار ببینیم:

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

به این معنی که عددی که دقیقا در کامپیوتر ذخیره شده است، برابر است با مقدار اعشاری

0.1000000000000000055511151231257827021181583404541015625 . به جای نمایش کامل این مقدار

اعشاری، بسیاری از زبان ها(از جمله نسخه های قدیمی تر پایتون) نتیجه را تا ۱۷ رقم پر ارزش گرد می کنند.

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

ماژول های fractions و decimal انجام این محاسبات را ساده کرده اند.

```
>>> 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'
```