Appendix E

Numbers in Python

This appendix provides an overview of numerical support in Python, mostly concentrating on the problems that arise from the implementation of reals as floating-point numbers. The code examples can be found at `http://coe.psu.ac.th/~ad/explore/code/E_Nums/`.

The python documentation includes an excellent section on this topic, "Floating Point Arithmetic: Issues and Limitations" (`https://docs.python.org/3/tutorial/floatingpoint.html`), and the standard academic reference is 'What Every Computer Scientist Should Know About Floating-Point Arithmetic' by David Goldberg [**Gol91**] (`https://people.cs.pitt.edu/~cho/cs1541/current/handouts/goldberg.pdf`).

E.1 The int and float Types

An integer is initially assigned a machine word (e.g. 64 or 32 bits), but if the number doesn't fit, or grows too big, then it's automatically promoted to a 'long' which can be as large as your RAM allows. For example:

```
>>> import math
>>> math.factorial(100)
9332621544394415268169923885626670049071596826438162146859296389521759
9999322991560894146397615651828625369792082722375825118521091686400000
0000000000000000000
```

Details on the integer type are available via the sys module:

```
>>> import sys
>>> sys.int_info
sys.int_info(bits_per_digit=30, sizeof_digit=4)
```

```
>>> sys.maxsize
9223372036854775807
```

sys.maxsize is the largest regular integer, before promotion kicks in.

Things aren't so rosy for Python floats. Almost all platforms map them to IEEE 754 binary64 doubles (`https://en.wikipedia.org/wiki/IEEE_754`). This supports very large and small numbers, but there are limits:

```
>>> sys.float_info
sys.float_info(max=1.7976931348623157e+308, max_exp=1024, max_10_exp=308,
min=2.2250738585072014e-308, min_exp=-1021, min_10_exp=-307, dig=15,
mant_dig=53, epsilon=2.220446049250313e-16, radix=2, rounds=1)
```

We'll describe some of the consequences of these limitations in what follows.

E.2 The math Module

Table E.1 (located at the end of this appendix) lists the functions in the math module, along with comments about some of the more obscure ones. They are grouped into six categories: number-theoretic, trigonometric, angular conversion, hyperbolic, special functions (e.g. gamma()), and constants. Full information can be found in Python's documentation at `https://docs.python.org/3/library/math.html`.

E.3 Numerical Errors

It's useful to distinguish between two types of errors in numerical calculations:

- **Approximation errors**. For example, you can generate the exponential, e^x, using a Taylor series:

  $$y = \sum_{n=0}^{n} \frac{x^n}{n!}$$

  but the result will always be approximate since the terms from $n + 1$ to ∞ aren't being evaluated.

- **Roundoff errors**. This kind of error appears every time a calculation is carried out using floating-point numbers. None of the IEEE 754 formats offers infinite precision, and so some information will inevitably be lost. For instance, we know that $\sqrt{2}^2 - 2 = 0$, but Python disagrees:

  ```
  >>> (math.sqrt(2))**2 - 2
  4.440892098500626e-16
  ```

E.4 Representing Reals

The general IEEE 754 standard represents a real number x in *normal* form using three values: s, f, and e, such that:

$$x = (-1)^s \times 1.f \times 2^{(e-bias)}$$

s is the sign bit, $1.f$ denotes that only the fractional part (i.e. the digits in f) is stored, without the leading 1 (which is sometimes called the hidden bit). e is stored, but the actual exponent is $e - bias$, where bias is a predetermined offset which means that e is always positive, and so can be represented by an unsigned integer.

In the IEEE double format (used by Python floats), a 64-bit word is divided up as follows: 1 bit for the sign s, 11 bits for the exponent e, and 52 bits for the fraction of the mantissa f (see Fig. E.1).

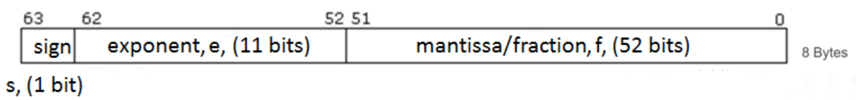

Figure E.1. The IEEE754 64-bit standard for floats

e is an 11-bit unsigned integer ranging from 0 to 2047, with the bias set to 1023 ($2^{10} - 1$) so the float's actual exponent can range between -1022 and +1023. (The exponents for -1023 (all 0s) and +1024 (all 1s) are reserved for special numbers.)

The fraction uses 52 bits: $m_{51}, m_{50}, , m_{49}, ..., m_1, m_0$, which are utilized in the float as:

$$1.f = 1 + m_{51} \times 2^{-1} + m_{50} \times 2^{-2} \cdots + m_0 \times 2^{-52}$$

For example, the binary version of 13256.625 is:

$$13256.625_{10} \equiv 11001111001000.101_2$$

Its normalized form is:

$$11001111001000.101_2 = 1.1001111001000101_2 \times 2^{13}.$$

The first digit of the normal form is always 1, so isn't stored in the fractional part of the machine word. Also, the stored exponent is offset by the bias, so $13 + 1023 = 1036_{10} = 10000001100_2$ is placed in the word.

floatUtils.py contains a printIEEE754() function that can perform these conversions on a supplied float:

```
>>> from floatUtils import *
>>> printIEEE754(13256.625)
IEEE754 of 13256.625 =
0 : 10000001100 : 1001111001000101000000000000000000000000000000000000
```

```
  Sign == +; Exponent == 13; mantissa == 1.6182403564453125
  Decimal value: 13256.6250000000000000
```

The final line of the output prints the binary string as a Python Decimal, which utilizes 28 digits of precision by default. This is useful for confirming whether the float really does have an exact binary value. If it does then the generated Decimal will be the same as the supplied float.

E.4.1 Largest Number. The largest possible fractional part (f) of the machine word is all 1's, so $1.f$ will be 1.111...111, which we can approximate as 2. The biggest possible stored exponent is 2046, which means that the largest true exponent is 2046 - 1023 = 1023. Therefore the largest number is approximately:

$$+1 \times 2 \times 2^{1023} = 2^{1024} \approx 1.8 \times 10^{308}$$

The biggest float is stored in sys.float_info.max, which can be examined with printIEEE754():

```
>>> printIEEE754(sys.float_info.max)
IEEE754 of 1.7976931348623157e+308 =
0 : 11111111110 : 11111111111111111111111111111111111111111111111111
  Sign == +; Exponent == 1023; mantissa == 1.9999999999999998
  Decimal value: 1.797693134862315708145274237E+308
```

Since float has a maximum value, then it's possible for float calculations to overflow. The following code from floatTests.py shows how to catch an overflow as an exception:

```
v = 2
while True:
  print(f"exp({v}): ", end='')
  try:
    print(math.exp(v))
  except Exception as e:
    print(type(e).__name__, e.args)  # or just print(e)
    break
  v *= 2
```

The output:

```
exp(2): 7.38905609893065
exp(4): 54.598150033144236
  : # more lines, not shown
exp(256): 1.5114276650041035e+111
exp(512): 2.2844135865397565e+222
exp(1024): OverflowError ('math range error',)
```

E.4.2 Subnormals. When $e = 0$, IEEE 754 employs an alternative format to represent numbers less than the smallest normal number (2.23e-308). *Subnormals* are encoded as:

$$x = (-1)^s \times 0.f \times 2^{-1022}$$

The smallest possible fraction is 2^{-52} which when multiplied to the exponent (always 2^{-1022}) gives the smallest number, 2^{-1074}, which is $\approx 4.94 \times 10^{-324}$.

If we try to generate a number that's larger than 1.8×10^{308}, we get overflow. If we try to represent a number that's smaller than the subnormal 4.94×10^{-324}, then we get underflow. These two extremes are illustrated in Fig. E.2.

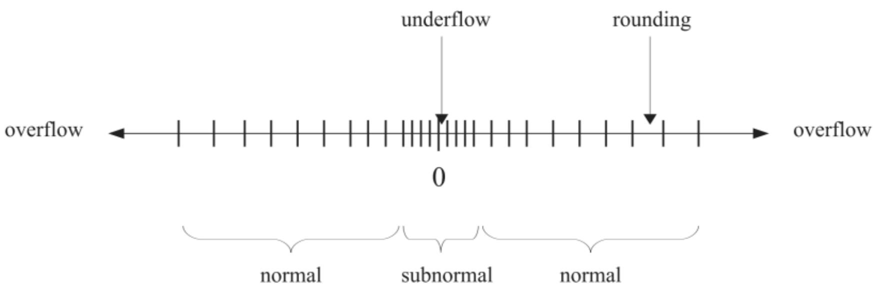

Figure E.2. Representing floating-point numbers

The following code snippet from floatTests.py starts with the smallest *normal* float, and keeps dividing it by 100 to show the switch to subnormals.

```
f = sys.float_info.min
print("float min: ", f)

while f != 0:
  f = f/100
  print("Divided by 100:", f)  # gradual underflow
```

The output:

```
float min:  2.2250738585072014e-308
Divided by 100: 2.2250738585072e-310
Divided by 100: 2.225073858507e-312
Divided by 100: 2.2250738583e-314
Divided by 100: 2.22507384e-316
Divided by 100: 2.225074e-318
Divided by 100: 2.2253e-320
Divided by 100: 2.2e-322
Divided by 100: 0.0
```

printIEEE754() is able to distinguish between the normal and subnormal formats when converting a float:

```
>>> printIEEE754(sys.float_info.min)
IEEE754 of 2.2250738585072014e-308 =
0 : 00000000001 : 0000000000000000000000000000000000000000000000000000
```

```
  Sign == +; Exponent == -1022; mantissa == 1.0
  Decimal value: 2.2250738585072013830902327170E-308

>>> printIEEE754(2.225074e-318)
IEEE754 of 2.225074e-318 =
0 : 00000000000 : 0000000000000000000000000000000001101101111100111000
Subnormal:  Sign == +; Exponent == -1022; mantissa == 1.000000082740371e-10
          Decimal value: 2.2250740426106379363535952150E-318
```

E.4.3 Zero, Infinity, NaN. There are several special values in the standard, shown in Fig. E.3.

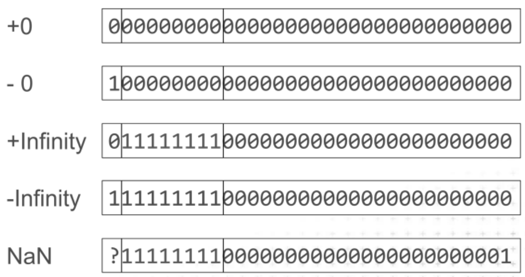

Figure E.3. Floating-point special values

Most arithmetic operations that produce 0 are given the value +0, with -0 reserved for very tiny negative numbers that are rounded to 0 due to precision limits.

NaN (Not a Number) is the result of indeterminate operations such as 0/0, $\infty - \infty$, or x + NaN.

E.4.4 Number Precision. Fig. E.2 also depicts the unequal effects of precision on number spacing. Even when the exponent is large, there is still only 52 bits to assign to the fraction, and so the distance between exactly representable number will be relatively large. But when the exponent is smaller, the fraction is able to denote more closely packed numbers.

This can be observed by using math.nextafter(x,y), introduced in Python 3.9, which returns the next float value after x when moving towards y. This can be employed alongside math.ulp(x) which returns the value of the least significant bit of x. ULP stands for 'unit of least precision', or 'unit in the last place'.

The following code reports next float values and ULPs for values between 1e-5 and 1e10:

```
for i in range(-5, 10):
  nxt = math.nextafter(10**i, float('inf'))
```

```
  print(f"next after {10**i} = {nxt}")
  print(" ulp =", math.ulp(nxt))
```

The output:

```
next after 1e-05 = 1.0000000000000003e-05
 ulp = 1.6940658945086007e-21
next after 0.0001 = 0.00010000000000000002
 ulp = 1.3552527156068805e-20
next after 0.001 = 0.0010000000000000002
 ulp = 2.168404344971009e-19
    :  # more lines; not shown
next after 10000000 = 10000000.000000002
 ulp = 1.862645149230957e-09
next after 100000000 = 100000000.00000001
 ulp = 1.4901161193847656e-08
next after 1000000000 = 1000000000.0000001
 ulp = 1.1920928955078125e-07
```

The general point is that the precision gap (the ULP) between representable numbers gets wider as larger numbers are considered. This is due to the number of bits in the fractional part of the number remaining constant.

Assume that a number x is stored as $\pm a \times 2^b$, where $\frac{1}{2} \leq a < 1$. Since the fractional part of a occupies 52 bits, there's a possible error in a in the interval $\pm\frac{1}{2} \times 2^{-52} = \pm 2^{-53}$. Thus the stored value is actually $a + E$, where $|E| \leq 2^{-53}$. It follows that the value of x is:

$$\pm(a + E) \times 2^b = \pm a \times 2^b \cdot (1 \pm \frac{E}{a}) = x(1 + \epsilon),$$

where

$$\epsilon = \frac{E}{a}, \quad \text{and} \quad |\epsilon| \leq \frac{1}{a} \times 2^{-53} \leq 2^{-52},$$

So a number x is actually stored as $x(1 + \epsilon)$, where $|\epsilon| \leq p = 2^{-52}$. p is the precision, which in terms of decimal digits, is approximately:

```
>>> import math
>>> math.floor( math.log10(2**52) )
15
```

Only the first 15 or 16 digits of a float can be accurate in the best case. Unfortunately, values may become a lot less accurate when they result from math operations involving other numbers.

Suppose we have x_1 and x_2, which are stored as $x_1(1 + \epsilon_1)$ and $x_2(1 + \epsilon_2)$, where $|\epsilon_1| \leq p$ and $|\epsilon_2| \leq p$ (recall that $p = 2^{-52}$). We multiply these numbers together to obtain $y = x_1x_2$. The actual value for y will be

$$\begin{aligned} x_1(1 + \epsilon_1)x_2(1 + \epsilon_2) &= x_1x_2(1 + \epsilon_1 + \epsilon_2 + \epsilon_1\epsilon_2) \\ &\simeq y(1 + \epsilon_1 + \epsilon_2) \end{aligned}$$

since $\epsilon_1\epsilon_2$ is small compared with the other terms involved.

Thus y is $y(1+\delta)$, where $\delta \simeq \epsilon_1+\epsilon_2$, and so $|\delta| \leq 2p$. In other words, the possible error for y may be double that of x_1 and x_2. However, this is an upper limit, and in many cases it may be much smaller. For instance if ϵ_1 and ϵ_2 have opposite signs then they will partially cancel out one another.

If instead we consider $y = x_1/x_2$, the result is:

$$\frac{x_1(1+\epsilon_1)}{x_2(1+\epsilon_2)} = \frac{x_1}{x_2}(1+\epsilon_1)(1-\epsilon_2+\epsilon_2^2-\epsilon_2^3+\cdots),$$

using the binomial expansion of $(1+\epsilon_2)^{-1}$. Ignoring terms involving products of two or more ϵ's, this gives

$$y(1+\epsilon_1-\epsilon_2).$$

Thus y is $y(1+\delta)$, where $\delta \simeq \epsilon_1-\epsilon_2$. However, since ϵ_1 and ϵ_2 can be either positive or negative, we again have the worst case that $\delta \leq 2p$.

Addition and subtraction are a little more difficult to analyze. If $y = x_1 + x_2$ then y is

$$x_1(1+\epsilon_1) + x_2(1+\epsilon_2) = x_1 + x_2 + x_1\epsilon_1 + x_2\epsilon_2 = y(1+\delta),$$

with

$$\delta = \frac{x_1\epsilon_1 + x_2\epsilon_2}{x_1+x_2}. \tag{E.1}$$

We know that $-p \leq \epsilon_1 \leq p$ and $-p \leq \epsilon_2 \leq p$; so if x_1 and x_2 are both positive, we can write

$$-\frac{x_1 p + x_2 p}{x_1+x_2} \leq \delta \leq \frac{x_1 p + x_2 p}{x_1+x_2}$$

or $|\delta| \leq p$. Since the limit for $|\delta|$ is the same as the limit for $|\epsilon|$ there is no additional loss of accuracy when two positive numbers are added together.

We can extend this argument to the case when x_1 and x_2 are both negative, but *not* to the situation when x_1 and x_2 are opposite signs, i.e. when we are doing subtraction. If x_2 is close to $-x_1$, the denominator $x_1 + x_2$ in Equ. (E.1) is nearly zero, so δ could be *very* large. It follows that we may suffer a substantial loss of accuracy when subtracting two nearly equal numbers. This situation is probably the most common precision problem in computational work.

E.4.5 Floating-point Arithmetic. The preceding discussion doesn't include errors caused by how floating-point arithmetic is implemented. IEEE 754 addition and subtraction require that the numbers have the same exponent size, which is achieved by shifting the bits of the fractional part of one of the numbers to the right.

We'll illustrate this process in a simple way by adding the decimals 123456.7 and 0.009876543 (without converting them to binary), while assuming that a number can only hold 7 significant digits. First, the numbers are converted to exponent form as 1.234567×10^5 and 9.876543×10^{-3}. The smaller second number is shifted right by 8 places, so its exponent matches that of the first number, resulting in 0.0000000987643 $\times 10^5$. The numbers are added, producing 1.23456709876543 $\times 10^5$. But, since only seven digits can be stored, the result is rounded to 1.234567 $\times 10^5$, which causes the contribution of the second number to be lost completely.

The loss of digits may be worse when two similarly-sized numbers are subtracted. As an example, consider 123456.787654 - 123456.712345. Once again let's assume that only seven significant digits can be stored, so the subtraction will involve 1.234568 $\times 10^5$ (note the rounding) and 1.234567 $\times 10^5$. The result is 0.000001 $\times 10^5$ which is normalized to 1 $\times 10^{-1}$, or just 0.1. Of course, the actual result should be 0.075309. Substituting these values into Equ. (E.1) gives us a δ of 0.1/0.075309 $\simeq$ 1.328, or 133%. Essentially, the computed answer contains no correct digits.

The problems become harder to detect when we remember that 'simple' floating-point numbers, such as 0.1, 0.2, and 0.3, may have rounding issues. Consider:

```
>>> 0.1 + 0.2
0.30000000000000004
>>> 0.1 + 0.2 == 0.3
False
```

There are two problems here: the rounding of the numbers when stored, and the loss of accuracy when they are added.

Calls to printIEEE754() reveal the details:

```
IEEE754 of 0.1 =
0 : 01111111011 : 1001100110011001100110011001100110011001100110011010
  Sign == +; Exponent == -4; mantissa == 1.6
  Decimal value: 0.1000000000000000055511151231

IEEE754 of 0.2 =
0 : 01111111100 : 1001100110011001100110011001100110011001100110011010
  Sign == +; Exponent == -3; mantissa == 1.6
  Decimal value: 0.2000000000000000111022302462

IEEE754 of 0.3 =
0 : 01111111101 : 0011001100110011001100110011001100110011001100110011
  Sign == +; Exponent == -2; mantissa == 1.2
  Decimal value: 0.2999999999999999888977697538
```

None of the numbers can be represented exactly in binary, so lose some accuracy through rounding. This is usually not apparent when the floats are printed since only at most 15 digits are reported:

```
>>> 0.1
0.1
>>> 0.2
0.2
```

```
>>> 0.3
0.3
```

However, the floating-point binary versions of 0.1 and 0.2 (which we'll call a and b) are:

$$a = 1.1001100110011001100110011001100110011001100110011010 \times 10^{-4}$$
$$b = 1.1001100110011001100110011001100110011001100110011010 \times 10^{-3}$$

Before adding them, the mantissa of the smaller number a (0.1) is shifted 1 bit to the right so that both number's exponents are the same:

$$a = 0.1100110011001100110011001100110011001100110011001101 \times 10^{-3}$$

$a + b$ produces c (the value approximating 0.3):

$$c = 10.0110011001100110011001100110011001100110011001100111 \times 10^{-3}$$

This is normalized by shifting it 1 bit to the right:

$$c = 1.00110011001100110011001100110011001100110011001100111 \times 10^{-2}$$

Internally, the leading 1 isn't stored, but more importantly the fractional part now has 53 bits, so the result must be rounded up to:

$$c = 1.0011001100110011001100110011001100110011001100110\underline{1000} \times 10^{-2}$$

The changed digits have been underlined. The last bit is now discarded, leaving the fraction as:

```
0011001100110011001100110011001100110011001100110100
```

This is *not* the same as the printIEEE754(0.3) output shown above, but is the floating-point result of $0.1 + 0.2$:

```
>>> printIEEE754(0.1+0.2)
IEEE754 of 0.30000000000000004 =
0 : 01111111101 : 0011001100110011001100110011001100110011001100110100
  Sign == +; Exponent == -2; mantissa == 1.2000000000000002
  Decimal value: 0.3000000000000000444089209850
```

One way of avoiding this problem is by comparing floats using less decimal places (i.e. less than 15 significant digits). For instance:

```
>>> round(0.1 + 0.2, 5) == round(0.3, 5)
True
```

E.4.6 The Dangers of Equality. One common place where floats cause problems is within tests for equality, often inside if-branches and while loops. For instance:

```
print("\nCount down to 0:")
a = 2
while a != 0:
  print(a, end= ' ')
  a -= 0.1
  if a < -2:
    print("\nReached -2!")
    break
```

The lack of explicit typing in Python contributes to the confusion since a starts as an integer but is cast to a float when it's incremented by 0.1. As we saw above, 0.1 is not represented exactly in binary, and the small error will gradually increase as the loop carries out more additions, making it very unlikely that a will land exactly on 0. This is shown in the output:

```
Count down to 0:
2 1.9 1.7999999999999998 1.6999999999999997 1.5999999999999996
1.4999999999999996 1.3999999999999995 1.2999999999999994 ...
0.09999999999999937 -6.38378239159465e-16 -0.10000000000000064
-0.20000000000000065 -0.30000000000000066 -0.4000000000000007
... -1.500000000000001 -1.6000000000000012 1.7000000000000013
-1.8000000000000014 -1.9000000000000015
Reached -2!
```

E.4.7 Ordering Matters. The result of operations involving floating-point numbers may depend on the order in which those operations are carried out. An example:

```
>>> (0.7 + 0.1) + 0.3
1.0999999999999999
>>> 0.7 + (0.1 + 0.3)
1.1
```

A more dramatic difference:

```
>>> xt = 1e20
>>> yt = -1e20
>>> zt = 1
>>> (xt + yt) + zt
1.0
>>> xt + (yt + zt)
0.0
```

This second example is really performing *subtraction* of similar numbers, and so it's to be expected that the rounding error may increase significantly. In the first bracketed calculation, the two large numbers, `xt` and `yt`, cancel each other and `zt` becomes the answer. In the second version, adding 1 to the negative large number `yt` rounds to `yt`. Then, `xt` and `yt` cancel each other out.

One fix for summing problems is to use math.fsum(), which sums a list while avoiding loss of precision by tracking the intermediate partial sums. The following contrasts the results of sum() and fsum():

```
>>> xt = 1e20
>>> yt = -1e20
>>> zt = 1
>>> sum([xt, yt, zt])
1.0
>>> sum([yt, zt, xt])
0.0
>>> import math
>>> math.fsum([xt, yt, zt])
1.0
>>> math.fsum([yt, zt, xt])
1.0
```

E.4.8 Rearranging Expressions. Two versions of the same expression:

$$\text{poor}(x) = \frac{1}{\sqrt{x^2+1} - x} \quad \text{and} \quad \text{good}(x) = \sqrt{x^2+1} + x$$

encoded as two functions in exprEval.py:

```
def poor_eval(x):
    return 1/(math.sqrt(x**2 + 1) - x)

def good_eval(x):
    return (math.sqrt(x**2 + 1) + x)


for v in [1000, 10000, 1000000, 10000000]:
  print(f"{v:8d} poor: {poor_eval(v):.8f}; good: {good_eval(v):.8f}")
```

Listing E.1. Evaluating two versions of the same expression

The loop is never completed since 100000000 evaluates to a zero denominator in poor_eval():

```
> python exprEval.py
    1000 poor: 2000.00049981; good: 2000.00050000
   10000 poor: 19999.99977765; good: 20000.00005000
 1000000 poor: 1999984.77112922; good: 2000000.00000050
10000000 poor: 19884107.85185185; good: 20000000.00000005
Traceback:
  File "exprEval.py", line 15, in <module>
```

```
  File "exprEval.py", line 8, in poor_eval
ZeroDivisionError: float division by zero
```

The earlier results also point to an obvious blooming of errors in poor_eval() compared to the more accurate good_eval(). The reason lies in poor_eval()'s use of subtraction. For large values of x, two almost identical values are being subtracted, producing a number close to 0 with a large relative error.

E.4.9 Computing the Exponential Function. A direct implementation of the exponential function using the Taylor/Maclaurin series will sum $x^n/n!$ for each term:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$$

This suffers from (at least) two problems: (a) calculating the power and the factorial is costly, and (b) both x^n and $n!$ can become very large (potentially overflowing) even though the required term ratio is small.

Instead, we take advantage of the fact that the n-th term in the expansion is related to the $(n-1)$-th term:

$$\frac{x^n}{n!} = \frac{x}{n}\frac{x^{n-1}}{(n-1)!}$$

There are two main ways to terminate a summation: (a) test for when a new term is "small enough", or (b) test for when the running total reaches a particular value. We'll employ (a) and terminate the loop when adding the n-th term to the total doesn't change it (due to the precision limits of floats). calcExp() is shown in Listing E.2 (compexp.py).

```
def calcExp(x):
  n = 0
  oldsum, tot, term = 0, 1, 1
  while tot != oldsum:
    oldsum = tot
    n += 1
    term *= x/n
    tot += term
    # tot += (x**n)/math.factorial(n)
  print(n, "iterations")
  return tot
```

Listing E.2. Calculating exp()

It's instructive to compare calcExp() and math.exp() for different x values, starting with $x = 2$.

```
>>> import math
>>> from compexp import *
>>> calcExp(2)
23 iterations
```

```
7.389056098930649
>>> math.exp(2)
7.38905609893065
```

There's agreement to about 15 decimal digits, which is to be expected when dealing with Python floats.

For x = 20, more iterations are needed, but the answer is again in agreement with math.exp() to 15 decimal digits.

```
>>> calcExp(20)
68 iterations
485165195.40979046
>>> math.exp(20)
485165195.4097903
```

The output for $x = -20$ tells a different story:

```
>>> calcExp(-20)
95 iterations
6.147561828914626e-09
>>> math.exp(-20)
2.061153622438558e-09
```

Even more iterations are required, but the real problem is that the function produces a sum that is totally wrong (although of the right order of magnitude). This is due to the way that the terms in the series have alternating signs, and so cancellation occurs between numbers of comparable magnitudes:

$$e^{-x} = 1 - x + \frac{x^2}{2!} - \frac{x^3}{3!} + \cdots$$

Fortunately, there's an easy fix for this particular problem: take advantage of $e^{-x} = 1/e^x$:

```
>>> math.exp(-20)
2.061153622438558e-09
>>> x = calcExp(20)
68 iterations
>>> 1/x
2.061153622438557e-09
```

The Taylor expansion for positive x doesn't suffer from cancellations, and supplies 15 correct digits. Dividing 1 by this large number leads to an answer also with 15 correct digits.

The main part of compexp.py (Listing E.3; compexp.py) graphs the relative differences in the outputs between calcExp(x) and math.exp(x) for x values between 0 and 30:

```
xs = list(frange(0, 30, 0.1))
ys = []
```

```
for x in xs:
  ys.append( (calcExp(x) - math.exp(x))/math.exp(x) )
  # ys.append( (calcExp(D(x)) - D(x).exp())/D(x).exp() )

plt.xlabel("x")
plt.ylabel("error")
plt.plot(xs, ys)

plt.title("Relative Diff Errors for exp()s")
plt.show()
```

Listing E.3. Graphing math.exp() and calcExp()

Fig. E.4 confirms that the relative error stays in the range $\pm 1e - 15$, the precision of floats.

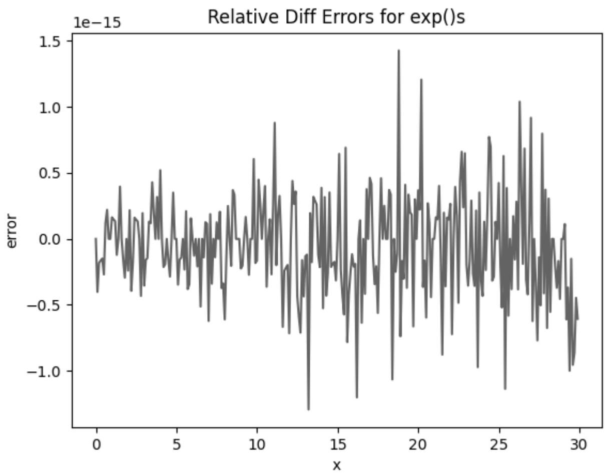

Figure E.4. Relative differences between math.exp() and calcExp()

E.5 Avoiding Float Problems

The simplest way to reduce the errors caused by the 'limited' precision of floats (i.e. ±1e-15) is to switch to Python's Decimal or Fraction types. Another solution is to employ a third-party library, such as mpmath.

E.5.1 Decimal. The Decimal module (`https://docs.python.org/3/library/decimal.html`) allows decimals to be represented exactly so that, for example, 0.1 + 0.1 + 0.1 == 0.3 is true. Unlike float, Decimal has no maximum, although RAM space and running time will constrain larger calculations. Also a Decimal's precision (which defaults to 28 places) can be adjusted, and the programmer has many more rounding options.

The module is based on the Decimal Arithmetic Specification developed by IBM (`https://speleotrove.com/decimal/decarith.html`).

It's best if you initialize Decimals using strings:

```
import decimal
from decimal import Decimal as D

x = D('0.1')
print("Decimal str 0.1:", x)
y = D('0.3')
res = x + x + x
print("Decimal str 0.1 + 0.1 + 0.1 == 0.3? ", (res == y), res)

x = D(0.1)
print("\nDecimal float 0.1:", x)
y = D(0.3)
res = x + x + x
print("Decimal float 0.1 + 0.1 + 0.1 == 0.3? ", (res == y), res)
```

The first block of code manipulates its decimals exactly, but the second part has the same problems as float, albeit with greater precision (28 places for Decimal vs. 15 for float).

```
Decimal str 0.1: 0.1
Decimal str 0.1 + 0.1 + 0.1 == 0.3?  True 0.3

Decimal float 0.1:
0.1000000000000000055511151231257827021181583404541015625
Decimal float 0.1 + 0.1 + 0.1 == 0.3?  False
0.3000000000000000166533453694
```

A useful feature for code portability is that integers are cast to decimals when they're used with Decimals. This means that many functions can be reused unchanged for both float and decimal calculations. Consider calcExp() (Listing E.2): no changes are needed to that function except for calling it with a Decimal argument.

```
>>> import math
>>> import decimal
>>> from decimal import Decimal as D
>>> from compexp import *

>>> calcExp(2)      # float call
23 iterations
7.389056098930649
>>> calcExp(D(2))   # Decimal call
```

```
34 iterations
Decimal('7.389056098930650227230427460')

>>> calcExp(20)     # float call
68 iterations
485165195.40979046
>>> calcExp(D(20))  # Decimal call
88 iterations
Decimal('485165195.4097902779691068300')
```

Since the Decimal precision is larger, more iterations are required before the loop inside calcExp() terminates.

The increased precision for Decimal *may* mean that the rounding errors will be small enough not to swamp the calculation, as in the case of exp(-20):

```
>>> math.exp(-20)
2.061153622438558e-09
>>> calcExp(-20)
95 iterations
6.147561828914626e-09   # wrong
>>> calcExp(D(-20))
112 iterations
Decimal('2.061153622431177804227533375E-9')   # correct-ish
```

Decimal precision can be changed:

```
>>> decimal.getcontext().prec = 100
>>> calcExp(D(-20))
194 iterations
Decimal('2.061153622438557827965940380155820976375807275599103692972 2
4466162916402378455935328307418415960603 5E-9')   # even more correct-ish
```

Wolfram Alpha (`https://www.wolframalpha.com`) reports e^{-20} to be

```
2.0611536224385578279659403801558209763758072755991036929722
4466162916402378455935 32 7991092790558136703638789079215 E-9
```

which suggests that calcExp() is correct to around 85 decimal digits when Decimal precision is set to 100.

decimal.quantize() rounds a number to a fixed exponent. This method is useful for monetary applications, especially when rounding flags are utilized as well.

```
d = D("10.005")
print("Value:", d)  # 10.005
print("2 dps rounded:", d.quantize(D("1.00")))   # 10.00
print("2 dps rounded up:", d.quantize(D("1.00"),
                       decimal.ROUND_HALF_UP))   # 10.01
```

`decimal.ROUND_HALF_UP` ensures that a number is rounded up if its neighboring digits are equidistant, a criteria used in most business math.

A drawback of the Decimal module is a lack of math operations, although it does have logarithm, exponent, and square root functions. However, the 'Recipes' section of the documentation (`https://docs.python.org/3/library/decimal.html#recipes`) has Taylor series approximations for calculating π, sin(), and cos(), which we've copied to a local file, decTrig.py. We've also added a function for calculating arctan(), which has proved useful (e.g. see Sec. 6.7.8).

E.5.2 Fraction. For number problems involving rationals, it may be simpler to encode them as Fractions in Python:

```
>>> from fractions import Fraction as F
>>> F(3,10)
Fraction(3, 10)
>>> F(3,7) - F(5,9)
Fraction(-8, 63)
>>> F(3,10) * F(10,3) - 1
Fraction(0, 1)
>>> F(3,10) * F(10,3) - 1 == 0
True
```

These examples show that $\frac{3}{10}$ is represented exactly (recall that float has problems with 0.3), and calculations such as $\frac{3}{10} \times \frac{10}{3}$ are exact. It's also possible to combine fractions and integers.

E.5.3 Other Numeric Modules in Python. Python's math module is built atop the `numbers` class (`https://docs.python.org/3/library/numeric.html`), which also underpins classes for complex numbers (`cmath`), pseudo-random numbers (`random`) and statistics (`statistics`).

E.5.4 Third-party Libraries. **Mpmath** (`https://mpmath.org/`) supports arbitrary-precision similar to Python's Decimal, but unlike Decimal also implements functions across a wide range of math topics, including numerical integration and differentiation, limits and summations of infinite series, root-finding, Chebyshev approximations, fourier and taylor series, ODEs, and linear algebra.

Numpy (`https://numpy.org/`) supports large, multi-dimensional arrays and matrices, and a large assortment of routines for fast operations on those arrays, including sorting, selecting, I/O, shape manipulation, discrete Fourier transforms, linear algebra, and statistical operations.

As we mentioned in the preface, we've (mostly) avoided using numpy in this book since Python lists are almost as expressive in the 1D and 2D cases. Also, numpy programming with arrays employs element-by-element operations, which makes it difficult (in an untyped language) to know what an operation like $a * b$ means. The APL-style of programming that this encourages is very elegant, but also quite confusing for a beginner, especially when the concept of *nested* *arrays* is utilized. Its support in MATLAB and the Wolfram Language (the programming language of Mathematica) is one reason why the second author think these tools are too complex for beginners. Nevertheless, numpy is the clear winner for large data, with some measurements showing its arrays to be 50x faster than lists.

The numpy `ndarray` have become the de-facto tool for multi-dimensional data in Python, and so numpy is employed by many other libraries.

Sympy (`https://www.sympy.org/`) focuses on symbolic mathematics, including algebraic manipulation, calculus, and equation solving, using symbolic rather than numerical techniques.

Scipy (`https://scipy.org/`) aims to offer support for all types of scientific and engineering applications. It's built on top of numpy, so includes routines for manipulating arrays, matrices, and other kinds of multidimensional data, and adds a large number of algorithms, such as: Airy functions, elliptic functions and integrals, a variety of statistical functions and distributions, Fresnel integrals, Legendre, hypergeometric, and spheroidal functions.

Panda (`https://pandas.pydata.org/`) provides tools for analyzing tabular data, including functions for importing and cleaning, statistical analysis, and visualization. Its name derives from the term "panel data", otherwise known as longitudinal data.

Exercises

(1) Fermat's last theorem states that no three positive integers x, y and z can satisfy the equation $x^n + y^n - z^n = 0$ for any integer $n > 2$. Explain this apparent counter-example:

```
>>> 844487.0**5 + 1288439.0**5 - 1318202.0**5
0.0
```

Another version of this problem, proposed by noted mathematician Homer Simpson is $3987^{12} + 3987^{12} = 3987^{12}$ (`https://boingboing.net/2014/10/17/homers-last-theorem.html`). Sadly, even when this is converted to powers of floats, it still (correctly) reports false. Why the difference from the first equation?

(2) The functions $f(x) = (1 - \cos^2 x)/x^2$ and $g(x) = \sin^2 x/x^2$ are mathematically indistinguishable, but plotted in the region $-0.001 \leq x \leq 0.001$ show a marked difference. Explain why.

(3) A direct implementation of Heron's formula for the area of a triangle,

$$A = \sqrt{s(s-a)(s-b)(s-c)} \quad \text{where } s = \frac{1}{2}(a+b+c),$$

is inaccurate if one side is very much smaller than the other two (i.e. resulting in "needle-shaped" triangles). Why? Demonstrate that the following reformulation gives a more accurate result by considering a triangle with sides $(10^{-13}, 1, 1)$, which has area 5×10^{-14}:

$$A = \frac{1}{4}\sqrt{(a+(b+c))(c-(a-b))(c+(a-b))(a+(b-c))},$$

The sides are labeled so that $a \geq b \geq c$.

Functions	Comments
Number-theoretic	
exp(x)	
exp2(x)	2^x
log(x, base), log2(x), log10(x)	
log1p(x)	Natural log of $1 + x$
pow(x, y), sqrt(x), isqrt(n)	
cbrt(x)	$x^{\frac{1}{3}}$
ceil(x), floor(x), trunc(x), fabs(x), copysign(x, y)	
fmod(x, y), modf(x)	
factorial(n)	
gcd(integers), lcm(integers)	
comb(n, k)	Returns the number of ways to choose k items from n items without repetition and **without** order.
perm(n, k)	Returns the number of ways to choose k items from n items without repetition and **with** order.
prod(iterable), sumprod(p, q)	
fsum(iterable)	Avoids loss of precision by tracking multiple intermediate partial sums.
isclose(a, b), isfinite(x), isinf(x), isnan(x)	
frexp(x)	Returns the mantissa and exponent of the internal representation of a float.
ldexp(x, i)	Returns $x \times (2^i)$. The inverse of frexp().
nextafter(x, y, steps), ulp(x)	
remainder(x, y)	Returns IEEE 754-style remainder of x/y.
Trigonometric	
sin(x), cos(x), tan(x)	
asin(x), acos(x), atan(x), atan2(y, x)	
Angular Conversion	
degrees(x), radians(x)	
Hyperbolic	
sinh(x), cosh(x), tanh(x)	
asinh(x), acosh(x), atanh(x)	
Special Functions	
erf(x)	
erfc(x)	Complementary error function at x.
gamma(x)	
lgamma(x)	Natural log of Gamma function at x.
Constants	
pi, e, tau	
inf, nan	

Table E.1. Math module functions