1. Compare and contrast the float and Decimal classes' benefits and drawbacks.

The main difference is Floats and Doubles are binary floating point types and a Decimal will store the value as a floating decimal point type. So Decimals have much higher precision and are usually used within monetary (financial) applications that require a high degree of accuracy.

Decimal vs Double vs Float

The Decimal, Double, and Float variable types are different in the way that they store the values. Precision is the main difference where float is a single precision (32 bit) floating point data type, double is a double precision (64 bit) floating point data type and decimal is a 128-bit floating point data type.

Float - 32 bit (7 digits)

Double - 64 bit (15-16 digits)

Decimal - 128 bit (28-29 significant digits)

Difference between Decimal, Float and Double

The main difference is Floats and Doubles are binary floating point types and a Decimal will store the value as a floating decimal point type. So Decimals have much higher precision and are usually used within monetary (financial) applications that require a high degree of accuracy. But in performance wise Decimals are slower than double and float types.

Decimal can 100% accurately represent any number within the precision of the decimal format, whereas Float and Double, cannot accurately represent all numbers, even numbers that are within their respective formats precision.

When To Use Decimal, Double, and Float

Decimal

In case of financial applications it is better to use Decimal types because it gives you a high level of accuracy and easy to avoid rounding errors

Double

Double Types are probably the most normally used data type for real values, except handling money.

Float

It is used mostly in graphic libraries because very high demands for processing powers, also used situations that can endure rounding errors.

Approximate Range

Accuracy

Float is less accurate than Double and Decimal.

Double is more accurate than Float but less accurate than Decimal.

Decimal is more accurate than Float and Double.

2. Decimal('1.200') and Decimal('1.2') are two objects to consider. In what sense are these the same object? Are these just two ways of representing the exact same value, or do they correspond to different internal states?

Many decimal numbers don’t have exact representations in binary [floating-point](https://www.pythontutorial.net/advanced-python/python-float/) such as 0.1. When using these numbers in arithmetic operations, you’ll get a result that you would not expect. For example:

x = 0.1

y = 0.1

z = 0.1

s = x + y + z

print(s)

Code language: PHP (php)

Output:

0.30000000000000004

Code language: CSS (css)

The result is 0.30000000000000004, not 0.3.

To solve this problem, you use the Decimal class from the decimal module as follows:

import decimal

from decimal import Decimal

x = Decimal('0.1')

y = Decimal('0.1')

z = Decimal('0.1')

s = x + y + z

print(s)

Code language: JavaScript (javascript)

Output:

0.3

Code language: CSS (css)

The output is as expected.

The Python decimal module supports arithmetic that works the same as the arithmetic you learn at school.

Unlike [floats](https://www.pythontutorial.net/advanced-python/python-float/), Python represents decimal numbers exactly. And the exactness carries over into arithmetic. For example, the following expression returns exactly 0.0:

Decimal('0.1') + Decimal('0.1') + Decimal('0.1') - Decimal('0.3')

Code language: JavaScript (javascript)

Decimal context

Decimal always associates with a [context](https://www.pythontutorial.net/advanced-python/python-context-managers/) that controls the following aspects:

Precision during an arithmetic operation

Rounding algorithm

By default, the context is global. The global context is the default context. Also, you can set a temporary context that will take effect locally without affecting the global context.

To get the default context, you call the getcontext() function from the decimal module:

decimal.getcontext()

Code language: CSS (css)

The getcontext() function returns the default context, which can be global or local.

To create a new context copied from another context, you use the localcontext() function:

decimal.localcontext(ctx=None)

The localcontext() returns a new context copied from the context ctx if specified.

Once getting the context object, you can access the precision and rouding via the prec and rounding property respectively:

ctx.pre: get or set the precision. The ctx.pre is an integer which defaults to 28

ctx.rounding: get or set the rounding mechanism. The rounding is a string. It defaults to 'ROUND\_HALF\_EVEN'. Note floats also use this rounding mechanism.

Python provides the following rounding mechanisms:

| Rounding | Description |
| --- | --- |
| ROUND\_UP | round away from zero |
| ROUND\_DOWN | round towards zero |
| ROUND\_CEILING | round to ceiling (towards positive infinity) |
| ROUND\_FLOOR | round to floor (towards negative infinity) |
| ROUND\_HALF\_UP | round to nearest, ties away from zero |
| ROUND\_HALF\_DOWN | round to nearest, ties towards zero |
| ROUND\_HALF\_EVEN | round to nearest, ties to even (least significant digit) |

This example illustrates how to get the default precision and rounding of the default context:

import decimal

ctx = decimal.getcontext()

print(ctx.prec)

print(ctx.rounding)

Code language: PHP (php)

Output:

28

ROUND\_HALF\_EVEN

3. What happens if the equality of Decimal('1.200') and Decimal('1.2') is checked?

I'd like to format following numbers into the numbers next to them with java:

1000 to 1k

5821 to 5.8k

10500 to 10k

101800 to 101k

2000000 to 2m

7800000 to 7.8m

92150000 to 92m

123200000 to 123m

The number on the right will be long or integer the number on the left will be string. How should I approach this. I already did little algorithm for this but I thought there might be already something invented out there that does nicer job at it and doesn't require additional testing if I start dealing with billions and trillions :)

Additional Requirements:

The format should have maximum of 4 characters

The above means 1.1k is OK 11.2k is not. Same for 7.8m is OK 19.1m is not. Only one digit before decimal point is allowed to have decimal point. Two digits before decimal point means not digits after decimal point.

No rounding is necessary. (Numbers being displayed with k and m appended are more of analog gauge indicating approximation not precise article of logic. Hence rounding is irrelevant mainly due to nature of variable than can increase or decrees several digits even while you are looking at the cached result.)

4. Why is it preferable to start a Decimal object with a string rather than a floating-point value?

floats are NOT usable for representing real world values like money - not reliably, anyways. e.g. 7.47 may actually be 7.4699999923423423423 when converted to float.

The [decimal](https://docs.python.org/3/library/decimal.html#module-decimal) module provides support for fast correctly rounded decimal floating point arithmetic. It offers several advantages over the [float](https://docs.python.org/3/library/functions.html#float) datatype:

Decimal “is based on a floating-point model which was designed with people in mind, and necessarily has a paramount guiding principle – computers must provide an arithmetic that works in the same way as the arithmetic that people learn at school.” – excerpt from the decimal arithmetic specification.

Decimal numbers can be represented exactly. In contrast, numbers like 1.1 and 2.2 do not have exact representations in binary floating point. End users typically would not expect 1.1 + 2.2 to display as 3.3000000000000003 as it does with binary floating point.

The exactness carries over into arithmetic. In decimal floating point, 0.1 + 0.1 + 0.1 - 0.3 is exactly equal to zero. In binary floating point, the result is 5.5511151231257827e-017. While near to zero, the differences prevent reliable equality testing and differences can accumulate. For this reason, decimal is preferred in accounting applications which have strict equality invariants.

The decimal module incorporates a notion of significant places so that 1.30 + 1.20 is 2.50. The trailing zero is kept to indicate significance. This is the customary presentation for monetary applications. For multiplication, the “schoolbook” approach uses all the figures in the multiplicands. For instance, 1.3 \* 1.2 gives 1.56 while 1.30 \* 1.20 gives 1.5600.

Unlike hardware based binary floating point, the decimal module has a user alterable precision (defaulting to 28 places) which can be as large as needed for a given problem:

>>>

>>> from decimal import \*

>>> getcontext().prec = 6

>>> Decimal(1) / Decimal(7)

Decimal('0.142857')

>>> getcontext().prec = 28

>>> Decimal(1) / Decimal(7)

Decimal('0.1428571428571428571428571429')

Both binary and decimal floating point are implemented in terms of published standards. While the built-in float type exposes only a modest portion of its capabilities, the decimal module exposes all required parts of the standard. When needed, the programmer has full control over rounding and signal handling. This includes an option to enforce exact arithmetic by using exceptions to block any inexact operations.

The decimal module was designed to support “without prejudice, both exact unrounded decimal arithmetic (sometimes called fixed-point arithmetic) and rounded floating-point arithmetic.” – excerpt from the decimal arithmetic specification.

The module design is centered around three concepts: the decimal number, the context for arithmetic, and signals.

A decimal number is immutable. It has a sign, coefficient digits, and an exponent. To preserve significance, the coefficient digits do not truncate trailing zeros. Decimals also include special values such as Infinity, -Infinity, and NaN. The standard also differentiates -0 from +0.

The context for arithmetic is an environment specifying precision, rounding rules, limits on exponents, flags indicating the results of operations, and trap enablers which determine whether signals are treated as exceptions. Rounding options include [ROUND\_CEILING](https://docs.python.org/3/library/decimal.html#decimal.ROUND_CEILING), [ROUND\_DOWN](https://docs.python.org/3/library/decimal.html#decimal.ROUND_DOWN), [ROUND\_FLOOR](https://docs.python.org/3/library/decimal.html#decimal.ROUND_FLOOR), [ROUND\_HALF\_DOWN](https://docs.python.org/3/library/decimal.html#decimal.ROUND_HALF_DOWN), [ROUND\_HALF\_EVEN](https://docs.python.org/3/library/decimal.html#decimal.ROUND_HALF_EVEN), [ROUND\_HALF\_UP](https://docs.python.org/3/library/decimal.html#decimal.ROUND_HALF_UP), [ROUND\_UP](https://docs.python.org/3/library/decimal.html#decimal.ROUND_UP), and [ROUND\_05UP](https://docs.python.org/3/library/decimal.html#decimal.ROUND_05UP).

Signals are groups of exceptional conditions arising during the course of computation. Depending on the needs of the application, signals may be ignored, considered as informational, or treated as exceptions. The signals in the decimal module are: [Clamped](https://docs.python.org/3/library/decimal.html#decimal.Clamped), [InvalidOperation](https://docs.python.org/3/library/decimal.html#decimal.InvalidOperation), [DivisionByZero](https://docs.python.org/3/library/decimal.html#decimal.DivisionByZero), [Inexact](https://docs.python.org/3/library/decimal.html#decimal.Inexact), [Rounded](https://docs.python.org/3/library/decimal.html#decimal.Rounded), [Subnormal](https://docs.python.org/3/library/decimal.html#decimal.Subnormal), [Overflow](https://docs.python.org/3/library/decimal.html#decimal.Overflow), [Underflow](https://docs.python.org/3/library/decimal.html#decimal.Underflow) and [FloatOperation](https://docs.python.org/3/library/decimal.html#decimal.FloatOperation).

For each signal there is a flag and a trap enabler. When a signal is encountered, its flag is set to one, then, if the trap enabler is set to one, an exception is raised. Flags are sticky, so the user needs to reset them before monitoring a calculation.

5. In an arithmetic phrase, how simple is it to combine Decimal objects with integers?

The [decimal](https://docs.python.org/3/library/decimal.html#module-decimal) module provides support for fast correctly rounded decimal floating point arithmetic. It offers several advantages over the [float](https://docs.python.org/3/library/functions.html#float) datatype:

Decimal “is based on a floating-point model which was designed with people in mind, and necessarily has a paramount guiding principle – computers must provide an arithmetic that works in the same way as the arithmetic that people learn at school.” – excerpt from the decimal arithmetic specification.

Decimal numbers can be represented exactly. In contrast, numbers like 1.1 and 2.2 do not have exact representations in binary floating point. End users typically would not expect 1.1 + 2.2 to display as 3.3000000000000003 as it does with binary floating point.

The exactness carries over into arithmetic. In decimal floating point, 0.1 + 0.1 + 0.1 - 0.3 is exactly equal to zero. In binary floating point, the result is 5.5511151231257827e-017. While near to zero, the differences prevent reliable equality testing and differences can accumulate. For this reason, decimal is preferred in accounting applications which have strict equality invariants.

The decimal module incorporates a notion of significant places so that 1.30 + 1.20 is 2.50. The trailing zero is kept to indicate significance. This is the customary presentation for monetary applications. For multiplication, the “schoolbook” approach uses all the figures in the multiplicands. For instance, 1.3 \* 1.2 gives 1.56 while 1.30 \* 1.20 gives 1.5600.

Unlike hardware based binary floating point, the decimal module has a user alterable precision (defaulting to 28 places) which can be as large as needed for a given problem:

>>>

>>> from decimal import \*

>>> getcontext().prec = 6

>>> Decimal(1) / Decimal(7)

Decimal('0.142857')

>>> getcontext().prec = 28

>>> Decimal(1) / Decimal(7)

Decimal('0.1428571428571428571428571429')

Both binary and decimal floating point are implemented in terms of published standards. While the built-in float type exposes only a modest portion of its capabilities, the decimal module exposes all required parts of the standard. When needed, the programmer has full control over rounding and signal handling. This includes an option to enforce exact arithmetic by using exceptions to block any inexact operations.

The decimal module was designed to support “without prejudice, both exact unrounded decimal arithmetic (sometimes called fixed-point arithmetic) and rounded floating-point arithmetic.” – excerpt from the decimal arithmetic specification.

The module design is centered around three concepts: the decimal number, the context for arithmetic, and signals.

A decimal number is immutable. It has a sign, coefficient digits, and an exponent. To preserve significance, the coefficient digits do not truncate trailing zeros. Decimals also include special values such as Infinity, -Infinity, and NaN. The standard also differentiates -0 from +0.

The context for arithmetic is an environment specifying precision, rounding rules, limits on exponents, flags indicating the results of operations, and trap enablers which determine whether signals are treated as exceptions. Rounding options include [ROUND\_CEILING](https://docs.python.org/3/library/decimal.html#decimal.ROUND_CEILING), [ROUND\_DOWN](https://docs.python.org/3/library/decimal.html#decimal.ROUND_DOWN), [ROUND\_FLOOR](https://docs.python.org/3/library/decimal.html#decimal.ROUND_FLOOR), [ROUND\_HALF\_DOWN](https://docs.python.org/3/library/decimal.html#decimal.ROUND_HALF_DOWN), [ROUND\_HALF\_EVEN](https://docs.python.org/3/library/decimal.html#decimal.ROUND_HALF_EVEN), [ROUND\_HALF\_UP](https://docs.python.org/3/library/decimal.html#decimal.ROUND_HALF_UP), [ROUND\_UP](https://docs.python.org/3/library/decimal.html#decimal.ROUND_UP), and [ROUND\_05UP](https://docs.python.org/3/library/decimal.html#decimal.ROUND_05UP).

Signals are groups of exceptional conditions arising during the course of computation. Depending on the needs of the application, signals may be ignored, considered as informational, or treated as exceptions. The signals in the decimal module are: [Clamped](https://docs.python.org/3/library/decimal.html#decimal.Clamped), [InvalidOperation](https://docs.python.org/3/library/decimal.html#decimal.InvalidOperation), [DivisionByZero](https://docs.python.org/3/library/decimal.html#decimal.DivisionByZero), [Inexact](https://docs.python.org/3/library/decimal.html#decimal.Inexact), [Rounded](https://docs.python.org/3/library/decimal.html#decimal.Rounded), [Subnormal](https://docs.python.org/3/library/decimal.html#decimal.Subnormal), [Overflow](https://docs.python.org/3/library/decimal.html#decimal.Overflow), [Underflow](https://docs.python.org/3/library/decimal.html#decimal.Underflow) and [FloatOperation](https://docs.python.org/3/library/decimal.html#decimal.FloatOperation).

For each signal there is a flag and a trap enabler. When a signal is encountered, its flag is set to one, then, if the trap enabler is set to one, an exception is raised. Flags are sticky, so the user needs to reset them before monitoring a calculation.

6. Can Decimal objects and floating-point values be combined easily?

Decimal objects

class decimal.Decimal(value='0', context=None)

Construct a new [Decimal](https://docs.python.org/3/library/decimal.html#decimal.Decimal) object based from value.

value can be an integer, string, tuple, [float](https://docs.python.org/3/library/functions.html#float), or another [Decimal](https://docs.python.org/3/library/decimal.html#decimal.Decimal) object. If no value is given, returns Decimal('0'). If value is a string, it should conform to the decimal numeric string syntax after leading and trailing whitespace characters, as well as underscores throughout, are removed:

sign ::= '+' | '-'

digit ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

indicator ::= 'e' | 'E'

digits ::= digit [digit]...

decimal-part ::= digits '.' [digits] | ['.'] digits

exponent-part ::= indicator [sign] digits

infinity ::= 'Infinity' | 'Inf'

nan ::= 'NaN' [digits] | 'sNaN' [digits]

numeric-value ::= decimal-part [exponent-part] | infinity

numeric-string ::= [sign] numeric-value | [sign] nan

Other Unicode decimal digits are also permitted where digit appears above. These include decimal digits from various other alphabets (for example, Arabic-Indic and Devanāgarī digits) along with the fullwidth digits '\uff10' through '\uff19'.

If value is a [tuple](https://docs.python.org/3/library/stdtypes.html#tuple), it should have three components, a sign (0 for positive or 1 for negative), a [tuple](https://docs.python.org/3/library/stdtypes.html#tuple) of digits, and an integer exponent. For example, Decimal((0, (1, 4, 1, 4), -3)) returns Decimal('1.414').

If value is a [float](https://docs.python.org/3/library/functions.html#float), the binary floating point value is losslessly converted to its exact decimal equivalent. This conversion can often require 53 or more digits of precision. For example, Decimal(float('1.1')) converts to Decimal('1.100000000000000088817841970012523233890533447265625').

The context precision does not affect how many digits are stored. That is determined exclusively by the number of digits in value. For example, Decimal('3.00000') records all five zeros even if the context precision is only three.

The purpose of the context argument is determining what to do if value is a malformed string. If the context traps [InvalidOperation](https://docs.python.org/3/library/decimal.html" \l "decimal.InvalidOperation" \o "decimal.InvalidOperation), an exception is raised; otherwise, the constructor returns a new Decimal with the value of NaN.

Once constructed, [Decimal](https://docs.python.org/3/library/decimal.html#decimal.Decimal) objects are immutable.

Changed in version 3.2: The argument to the constructor is now permitted to be a [float](https://docs.python.org/3/library/functions.html#float) instance.

Changed in version 3.3: [float](https://docs.python.org/3/library/functions.html#float) arguments raise an exception if the [FloatOperation](https://docs.python.org/3/library/decimal.html" \l "decimal.FloatOperation" \o "decimal.FloatOperation) trap is set. By default the trap is off.

Changed in version 3.6: Underscores are allowed for grouping, as with integral and floating-point literals in code.

Decimal floating point objects share many properties with the other built-in numeric types such as [float](https://docs.python.org/3/library/functions.html#float) and [int](https://docs.python.org/3/library/functions.html#int). All of the usual math operations and special methods apply. Likewise, decimal objects can be copied, pickled, printed, used as dictionary keys, used as set elements, compared, sorted, and coerced to another type (such as [float](https://docs.python.org/3/library/functions.html#float) or [int](https://docs.python.org/3/library/functions.html#int)).

There are some small differences between arithmetic on Decimal objects and arithmetic on integers and floats. When the remainder operator % is applied to Decimal objects, the sign of the result is the sign of the dividend rather than the sign of the divisor:

>>>

>>> (-7) % 4

1

>>> Decimal(-7) % Decimal(4)

Decimal('-3')

The integer division operator // behaves analogously, returning the integer part of the true quotient (truncating towards zero) rather than its floor, so as to preserve the usual identity x == (x // y) \* y + x % y:

>>>

>>> -7 // 4

-2

>>> Decimal(-7) // Decimal(4)

Decimal('-1')

The % and // operators implement the remainder and divide-integer operations (respectively) as described in the specification.

Decimal objects cannot generally be combined with floats or instances of [fractions.Fraction](https://docs.python.org/3/library/fractions.html" \l "fractions.Fraction" \o "fractions.Fraction) in arithmetic operations: an attempt to add a [Decimal](https://docs.python.org/3/library/decimal.html#decimal.Decimal) to a [float](https://docs.python.org/3/library/functions.html#float), for example, will raise a [TypeError](https://docs.python.org/3/library/exceptions.html" \l "TypeError" \o "TypeError). However, it is possible to use Python’s comparison operators to compare a [Decimal](https://docs.python.org/3/library/decimal.html#decimal.Decimal) instance x with another number y. This avoids confusing results when doing equality comparisons between numbers of different types.

Changed in version 3.2: Mixed-type comparisons between [Decimal](https://docs.python.org/3/library/decimal.html#decimal.Decimal) instances and other numeric types are now fully supported.

In addition to the standard numeric properties, decimal floating point objects also have a number of specialized methods:

adjusted()

Return the adjusted exponent after shifting out the coefficient’s rightmost digits until only the lead digit remains: Decimal('321e+5').adjusted() returns seven. Used for determining the position of the most significant digit with respect to the decimal point.

as\_integer\_ratio()

Return a pair (n, d) of integers that represent the given [Decimal](https://docs.python.org/3/library/decimal.html#decimal.Decimal) instance as a fraction, in lowest terms and with a positive denominator:

>>>

>>> Decimal('-3.14').as\_integer\_ratio()

(-157, 50)

The conversion is exact. Raise OverflowError on infinities and ValueError on NaNs.

New in version 3.6.

as\_tuple()

Return a [named tuple](https://docs.python.org/3/glossary.html#term-named-tuple) representation of the number: DecimalTuple(sign, digits, exponent).

canonical()

Return the canonical encoding of the argument. Currently, the encoding of a [Decimal](https://docs.python.org/3/library/decimal.html#decimal.Decimal) instance is always canonical, so this operation returns its argument unchanged.

compare(other, context=None)

Compare the values of two Decimal instances. [compare()](https://docs.python.org/3/library/decimal.html#decimal.Decimal.compare) returns a Decimal instance, and if either operand is a NaN then the result is a NaN:

a or b is a NaN ==> Decimal('NaN')

a < b ==> Decimal('-1')

a == b ==> Decimal('0')

a > b ==> Decimal('1')

compare\_signal(other, context=None)

This operation is identical to the [compare()](https://docs.python.org/3/library/decimal.html#decimal.Decimal.compare) method, except that all NaNs signal. That is, if neither operand is a signaling NaN then any quiet NaN operand is treated as though it were a signaling NaN.

compare\_total(other, context=None)

Compare two operands using their abstract representation rather than their numerical value. Similar to the [compare()](https://docs.python.org/3/library/decimal.html#decimal.Decimal.compare) method, but the result gives a total ordering on [Decimal](https://docs.python.org/3/library/decimal.html#decimal.Decimal) instances. Two [Decimal](https://docs.python.org/3/library/decimal.html#decimal.Decimal) instances with the same numeric value but different representations compare unequal in this ordering:

>>>

>>> Decimal('12.0').compare\_total(Decimal('12'))

Decimal('-1')

Quiet and signaling NaNs are also included in the total ordering. The result of this function is Decimal('0') if both operands have the same representation, Decimal('-1') if the first operand is lower in the total order than the second, and Decimal('1') if the first operand is higher in the total order than the second operand. See the specification for details of the total order.

This operation is unaffected by context and is quiet: no flags are changed and no rounding is performed. As an exception, the C version may raise InvalidOperation if the second operand cannot be converted exactly.

compare\_total\_mag(other, context=None)

Compare two operands using their abstract representation rather than their value as in [compare\_total()](https://docs.python.org/3/library/decimal.html" \l "decimal.Decimal.compare_total" \o "decimal.Decimal.compare_total), but ignoring the sign of each operand. x.compare\_total\_mag(y) is equivalent to x.copy\_abs().compare\_total(y.copy\_abs()).

This operation is unaffected by context and is quiet: no flags are changed and no rounding is performed. As an exception, the C version may raise InvalidOperation if the second operand cannot be converted exactly.

conjugate()

Just returns self, this method is only to comply with the Decimal Specification.

copy\_abs()

Return the absolute value of the argument. This operation is unaffected by the context and is quiet: no flags are changed and no rounding is performed.

copy\_negate()

Return the negation of the argument. This operation is unaffected by the context and is quiet: no flags are changed and no rounding is performed.

copy\_sign(other, context=None)

Return a copy of the first operand with the sign set to be the same as the sign of the second operand. For example:

>>>

>>> Decimal('2.3').copy\_sign(Decimal('-1.5'))

Decimal('-2.3')

This operation is unaffected by context and is quiet: no flags are changed and no rounding is performed. As an exception, the C version may raise InvalidOperation if the second operand cannot be converted exactly.

exp(context=None)

Return the value of the (natural) exponential function e\*\*x at the given number. The result is correctly rounded using the [ROUND\_HALF\_EVEN](https://docs.python.org/3/library/decimal.html#decimal.ROUND_HALF_EVEN) rounding mode.

>>>

>>> Decimal(1).exp()

Decimal('2.718281828459045235360287471')

>>> Decimal(321).exp()

Decimal('2.561702493119680037517373933E+139')

classmethod from\_float(f)

Alternative constructor that only accepts instances of [float](https://docs.python.org/3/library/functions.html#float) or [int](https://docs.python.org/3/library/functions.html#int).

Note Decimal.from\_float(0.1) is not the same as Decimal('0.1'). Since 0.1 is not exactly representable in binary floating point, the value is stored as the nearest representable value which is 0x1.999999999999ap-4. That equivalent value in decimal is 0.1000000000000000055511151231257827021181583404541015625.

Note

From Python 3.2 onwards, a [Decimal](https://docs.python.org/3/library/decimal.html#decimal.Decimal) instance can also be constructed directly from a [float](https://docs.python.org/3/library/functions.html#float).

>>>

>>> Decimal.from\_float(0.1)

Decimal('0.1000000000000000055511151231257827021181583404541015625')

>>> Decimal.from\_float(float('nan'))

Decimal('NaN')

>>> Decimal.from\_float(float('inf'))

Decimal('Infinity')

>>> Decimal.from\_float(float('-inf'))

Decimal('-Infinity')

New in version 3.1.

fma(other, third, context=None)

Fused multiply-add. Return self\*other+third with no rounding of the intermediate product self\*other.

>>>

>>> Decimal(2).fma(3, 5)

Decimal('11')

is\_canonical()

Return [True](https://docs.python.org/3/library/constants.html#True) if the argument is canonical and [False](https://docs.python.org/3/library/constants.html#False) otherwise. Currently, a [Decimal](https://docs.python.org/3/library/decimal.html#decimal.Decimal) instance is always canonical, so this operation always returns [True](https://docs.python.org/3/library/constants.html#True).

is\_finite()

Return [True](https://docs.python.org/3/library/constants.html#True) if the argument is a finite number, and [False](https://docs.python.org/3/library/constants.html#False) if the argument is an infinity or a NaN.

is\_infinite()

Return [True](https://docs.python.org/3/library/constants.html#True) if the argument is either positive or negative infinity and [False](https://docs.python.org/3/library/constants.html#False) otherwise.

is\_nan()

Return [True](https://docs.python.org/3/library/constants.html#True) if the argument is a (quiet or signaling) NaN and [False](https://docs.python.org/3/library/constants.html#False) otherwise.

is\_normal(context=None)

Return [True](https://docs.python.org/3/library/constants.html#True) if the argument is a normal finite number. Return [False](https://docs.python.org/3/library/constants.html#False) if the argument is zero, subnormal, infinite or a NaN.

is\_qnan()

Return [True](https://docs.python.org/3/library/constants.html#True) if the argument is a quiet NaN, and [False](https://docs.python.org/3/library/constants.html#False) otherwise.

is\_signed()

Return [True](https://docs.python.org/3/library/constants.html#True) if the argument has a negative sign and [False](https://docs.python.org/3/library/constants.html#False) otherwise. Note that zeros and NaNs can both carry signs.

is\_snan()

Return [True](https://docs.python.org/3/library/constants.html#True) if the argument is a signaling NaN and [False](https://docs.python.org/3/library/constants.html#False) otherwise.

is\_subnormal(context=None)

Return [True](https://docs.python.org/3/library/constants.html#True) if the argument is subnormal, and [False](https://docs.python.org/3/library/constants.html#False) otherwise.

is\_zero()

Return [True](https://docs.python.org/3/library/constants.html#True) if the argument is a (positive or negative) zero and [False](https://docs.python.org/3/library/constants.html#False) otherwise.

ln(context=None)

Return the natural (base e) logarithm of the operand. The result is correctly rounded using the [ROUND\_HALF\_EVEN](https://docs.python.org/3/library/decimal.html#decimal.ROUND_HALF_EVEN) rounding mode.

log10(context=None)

Return the base ten logarithm of the operand. The result is correctly rounded using the [ROUND\_HALF\_EVEN](https://docs.python.org/3/library/decimal.html#decimal.ROUND_HALF_EVEN) rounding mode.

logb(context=None)

For a nonzero number, return the adjusted exponent of its operand as a [Decimal](https://docs.python.org/3/library/decimal.html#decimal.Decimal) instance. If the operand is a zero then Decimal('-Infinity') is returned and the [DivisionByZero](https://docs.python.org/3/library/decimal.html" \l "decimal.DivisionByZero" \o "decimal.DivisionByZero) flag is raised. If the operand is an infinity then Decimal('Infinity') is returned.

logical\_and(other, context=None)

[logical\_and()](https://docs.python.org/3/library/decimal.html#decimal.Decimal.logical_and) is a logical operation which takes two logical operands (see [Logical operands](https://docs.python.org/3/library/decimal.html#logical-operands-label)). The result is the digit-wise and of the two operands.

logical\_invert(context=None)

[logical\_invert()](https://docs.python.org/3/library/decimal.html#decimal.Decimal.logical_invert) is a logical operation. The result is the digit-wise inversion of the operand.

logical\_or(other, context=None)

[logical\_or()](https://docs.python.org/3/library/decimal.html#decimal.Decimal.logical_or) is a logical operation which takes two logical operands (see [Logical operands](https://docs.python.org/3/library/decimal.html#logical-operands-label)). The result is the digit-wise or of the two operands.

logical\_xor(other, context=None)

[logical\_xor()](https://docs.python.org/3/library/decimal.html#decimal.Decimal.logical_xor) is a logical operation which takes two logical operands (see [Logical operands](https://docs.python.org/3/library/decimal.html#logical-operands-label)). The result is the digit-wise exclusive or of the two operands.

max(other, context=None)

Like max(self, other) except that the context rounding rule is applied before returning and that NaN values are either signaled or ignored (depending on the context and whether they are signaling or quiet).

max\_mag(other, context=None)

Similar to the [max()](https://docs.python.org/3/library/decimal.html#decimal.Decimal.max) method, but the comparison is done using the absolute values of the operands.

min(other, context=None)

Like min(self, other) except that the context rounding rule is applied before returning and that NaN values are either signaled or ignored (depending on the context and whether they are signaling or quiet).

min\_mag(other, context=None)

Similar to the [min()](https://docs.python.org/3/library/decimal.html#decimal.Decimal.min) method, but the comparison is done using the absolute values of the operands.

next\_minus(context=None)

Return the largest number representable in the given context (or in the current thread’s context if no context is given) that is smaller than the given operand.

next\_plus(context=None)

Return the smallest number representable in the given context (or in the current thread’s context if no context is given) that is larger than the given operand.

next\_toward(other, context=None)

If the two operands are unequal, return the number closest to the first operand in the direction of the second operand. If both operands are numerically equal, return a copy of the first operand with the sign set to be the same as the sign of the second operand.

normalize(context=None)

Normalize the number by stripping the rightmost trailing zeros and converting any result equal to Decimal('0') to Decimal('0e0'). Used for producing canonical values for attributes of an equivalence class. For example, Decimal('32.100') and Decimal('0.321000e+2') both normalize to the equivalent value Decimal('32.1').

number\_class(context=None)

Return a string describing the class of the operand. The returned value is one of the following ten strings.

"-Infinity", indicating that the operand is negative infinity.

"-Normal", indicating that the operand is a negative normal number.

"-Subnormal", indicating that the operand is negative and subnormal.

"-Zero", indicating that the operand is a negative zero.

"+Zero", indicating that the operand is a positive zero.

"+Subnormal", indicating that the operand is positive and subnormal.

"+Normal", indicating that the operand is a positive normal number.

"+Infinity", indicating that the operand is positive infinity.

"NaN", indicating that the operand is a quiet NaN (Not a Number).

"sNaN", indicating that the operand is a signaling NaN.

quantize(exp, rounding=None, context=None)

Return a value equal to the first operand after rounding and having the exponent of the second operand.

>>>

>>> Decimal('1.41421356').quantize(Decimal('1.000'))

Decimal('1.414')

Unlike other operations, if the length of the coefficient after the quantize operation would be greater than precision, then an [InvalidOperation](https://docs.python.org/3/library/decimal.html" \l "decimal.InvalidOperation" \o "decimal.InvalidOperation) is signaled. This guarantees that, unless there is an error condition, the quantized exponent is always equal to that of the right-hand operand.

Also unlike other operations, quantize never signals Underflow, even if the result is subnormal and inexact.

If the exponent of the second operand is larger than that of the first then rounding may be necessary. In this case, the rounding mode is determined by the rounding argument if given, else by the given context argument; if neither argument is given the rounding mode of the current thread’s context is used.

An error is returned whenever the resulting exponent is greater than Emax or less than Etiny.

radix()

Return Decimal(10), the radix (base) in which the [Decimal](https://docs.python.org/3/library/decimal.html#decimal.Decimal) class does all its arithmetic. Included for compatibility with the specification.

remainder\_near(other, context=None)

Return the remainder from dividing self by other. This differs from self % other in that the sign of the remainder is chosen so as to minimize its absolute value. More precisely, the return value is self - n \* other where n is the integer nearest to the exact value of self / other, and if two integers are equally near then the even one is chosen.

If the result is zero then its sign will be the sign of self.

>>>

>>> Decimal(18).remainder\_near(Decimal(10))

Decimal('-2')

>>> Decimal(25).remainder\_near(Decimal(10))

Decimal('5')

>>> Decimal(35).remainder\_near(Decimal(10))

Decimal('-5')

rotate(other, context=None)

Return the result of rotating the digits of the first operand by an amount specified by the second operand. The second operand must be an integer in the range -precision through precision. The absolute value of the second operand gives the number of places to rotate. If the second operand is positive then rotation is to the left; otherwise rotation is to the right. The coefficient of the first operand is padded on the left with zeros to length precision if necessary. The sign and exponent of the first operand are unchanged.

same\_quantum(other, context=None)

Test whether self and other have the same exponent or whether both are NaN.

This operation is unaffected by context and is quiet: no flags are changed and no rounding is performed. As an exception, the C version may raise InvalidOperation if the second operand cannot be converted exactly.

scaleb(other, context=None)

Return the first operand with exponent adjusted by the second. Equivalently, return the first operand multiplied by 10\*\*other. The second operand must be an integer.

shift(other, context=None)

Return the result of shifting the digits of the first operand by an amount specified by the second operand. The second operand must be an integer in the range -precision through precision. The absolute value of the second operand gives the number of places to shift. If the second operand is positive then the shift is to the left; otherwise the shift is to the right. Digits shifted into the coefficient are zeros. The sign and exponent of the first operand are unchanged.

sqrt(context=None)

Return the square root of the argument to full precision.

to\_eng\_string(context=None)

Convert to a string, using engineering notation if an exponent is needed.

Engineering notation has an exponent which is a multiple of 3. This can leave up to 3 digits to the left of the decimal place and may require the addition of either one or two trailing zeros.

For example, this converts Decimal('123E+1') to Decimal('1.23E+3').

to\_integral(rounding=None, context=None)

Identical to the [to\_integral\_value()](https://docs.python.org/3/library/decimal.html" \l "decimal.Decimal.to_integral_value" \o "decimal.Decimal.to_integral_value) method. The to\_integral name has been kept for compatibility with older versions.

to\_integral\_exact(rounding=None, context=None)

Round to the nearest integer, signaling [Inexact](https://docs.python.org/3/library/decimal.html#decimal.Inexact) or [Rounded](https://docs.python.org/3/library/decimal.html#decimal.Rounded) as appropriate if rounding occurs. The rounding mode is determined by the rounding parameter if given, else by the given context. If neither parameter is given then the rounding mode of the current context is used.

to\_integral\_value(rounding=None, context=None)

Round to the nearest integer without signaling [Inexact](https://docs.python.org/3/library/decimal.html#decimal.Inexact) or [Rounded](https://docs.python.org/3/library/decimal.html#decimal.Rounded). If given, applies rounding; otherwise, uses the rounding method in either the supplied context or the current context.

Logical operands

The logical\_and(), logical\_invert(), logical\_or(), and logical\_xor() methods expect their arguments to be logical operands. A logical operand is a [Decimal](https://docs.python.org/3/library/decimal.html#decimal.Decimal) instance whose exponent and sign are both zero, and whose digits are all either 0 or 1.

Context objects

Contexts are environments for arithmetic operations. They govern precision, set rules for rounding, determine which signals are treated as exceptions, and limit the range for exponents.

Each thread has its own current context which is accessed or changed using the [getcontext()](https://docs.python.org/3/library/decimal.html" \l "decimal.getcontext" \o "decimal.getcontext) and [setcontext()](https://docs.python.org/3/library/decimal.html" \l "decimal.setcontext" \o "decimal.setcontext) functions:

decimal.getcontext()

Return the current context for the active thread.

decimal.setcontext(c)

Set the current context for the active thread to c.

You can also use the [with](https://docs.python.org/3/reference/compound_stmts.html#with) statement and the [localcontext()](https://docs.python.org/3/library/decimal.html" \l "decimal.localcontext" \o "decimal.localcontext) function to temporarily change the active context.

decimal.localcontext(ctx=None, \\*\\*kwargs)

Return a context manager that will set the current context for the active thread to a copy of ctx on entry to the with-statement and restore the previous context when exiting the with-statement. If no context is specified, a copy of the current context is used. The kwargs argument is used to set the attributes of the new context.

For example, the following code sets the current decimal precision to 42 places, performs a calculation, and then automatically restores the previous context:

from decimal import localcontext

with localcontext() as ctx:

ctx.prec = 42 # Perform a high precision calculation

s = calculate\_something()

s = +s # Round the final result back to the default precision

Using keyword arguments, the code would be the following:

from decimal import localcontext

with localcontext(prec=42) as ctx:

s = calculate\_something()

s = +s

Raises [TypeError](https://docs.python.org/3/library/exceptions.html" \l "TypeError" \o "TypeError) if kwargs supplies an attribute that [Context](https://docs.python.org/3/library/decimal.html#decimal.Context) doesn’t support. Raises either [TypeError](https://docs.python.org/3/library/exceptions.html" \l "TypeError" \o "TypeError) or [ValueError](https://docs.python.org/3/library/exceptions.html" \l "ValueError" \o "ValueError) if kwargs supplies an invalid value for an attribute.

Changed in version 3.11: [localcontext()](https://docs.python.org/3/library/decimal.html" \l "decimal.localcontext" \o "decimal.localcontext) now supports setting context attributes through the use of keyword arguments.

New contexts can also be created using the [Context](https://docs.python.org/3/library/decimal.html#decimal.Context) constructor described below. In addition, the module provides three pre-made contexts:

class decimal.BasicContext

This is a standard context defined by the General Decimal Arithmetic Specification. Precision is set to nine. Rounding is set to [ROUND\_HALF\_UP](https://docs.python.org/3/library/decimal.html#decimal.ROUND_HALF_UP). All flags are cleared. All traps are enabled (treated as exceptions) except [Inexact](https://docs.python.org/3/library/decimal.html#decimal.Inexact), [Rounded](https://docs.python.org/3/library/decimal.html#decimal.Rounded), and [Subnormal](https://docs.python.org/3/library/decimal.html#decimal.Subnormal).

Because many of the traps are enabled, this context is useful for debugging.

7. Using the Fraction class but not the Decimal class, give an example of a quantity that can be expressed with absolute precision.

Abstract

Floating-point arithmetic is considered an esoteric subject by many people. This is rather surprising because floating-point is ubiquitous in computer systems. Almost every language has a floating-point datatype; computers from PCs to supercomputers have floating-point accelerators; most compilers will be called upon to compile floating-point algorithms from time to time; and virtually every operating system must respond to floating-point exceptions such as overflow. This paper presents a tutorial on those aspects of floating-point that have a direct impact on designers of computer systems. It begins with background on floating-point representation and rounding error, continues with a discussion of the IEEE floating-point standard, and concludes with numerous examples of how computer builders can better support floating-point.

Categories and Subject Descriptors: (Primary) C.0 [Computer Systems Organization]: General -- instruction set design; D.3.4 [Programming Languages]: Processors -- compilers, optimization; G.1.0 [Numerical Analysis]: General -- computer arithmetic, error analysis, numerical algorithms (Secondary)

D.2.1 [Software Engineering]: Requirements/Specifications -- languages; D.3.4 Programming Languages]: Formal Definitions and Theory -- semantics; D.4.1 Operating Systems]: Process Management -- synchronization.

General Terms: Algorithms, Design, Languages

Additional Key Words and Phrases: Denormalized number, exception, floating-point, floating-point standard, gradual underflow, guard digit, NaN, overflow, relative error, rounding error, rounding mode, ulp, underflow.

Introduction

Builders of computer systems often need information about floating-point arithmetic. There are, however, remarkably few sources of detailed information about it. One of the few books on the subject, Floating-Point Computation by Pat Sterbenz, is long out of print. This paper is a tutorial on those aspects of floating-point arithmetic (floating-point hereafter) that have a direct connection to systems building. It consists of three loosely connected parts. The first section, [Rounding Error](https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html#680), discusses the implications of using different rounding strategies for the basic operations of addition, subtraction, multiplication and division. It also contains background information on the two methods of measuring rounding error, ulps and relative error. The second part discusses the IEEE floating-point standard, which is becoming rapidly accepted by commercial hardware manufacturers. Included in the IEEE standard is the rounding method for basic operations. The discussion of the standard draws on the material in the section [Rounding Error](https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html#680). The third part discusses the connections between floating-point and the design of various aspects of computer systems. Topics include instruction set design, optimizing compilers and exception handling.

I have tried to avoid making statements about floating-point without also giving reasons why the statements are true, especially since the justifications involve nothing more complicated than elementary calculus. Those explanations that are not central to the main argument have been grouped into a section called "The Details," so that they can be skipped if desired. In particular, the proofs of many of the theorems appear in this section. The end of each proof is marked with the z symbol. When a proof is not included, the z appears immediately following the statement of the theorem.

Rounding Error

Squeezing infinitely many real numbers into a finite number of bits requires an approximate representation. Although there are infinitely many integers, in most programs the result of integer computations can be stored in 32 bits. In contrast, given any fixed number of bits, most calculations with real numbers will produce quantities that cannot be exactly represented using that many bits. Therefore the result of a floating-point calculation must often be rounded in order to fit back into its finite representation. This rounding error is the characteristic feature of floating-point computation. The section [Relative Error and Ulps](https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html#689) describes how it is measured.

Since most floating-point calculations have rounding error anyway, does it matter if the basic arithmetic operations introduce a little bit more rounding error than necessary? That question is a main theme throughout this section. The section [Guard Digits](https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html#693) discusses guard digits, a means of reducing the error when subtracting two nearby numbers. Guard digits were considered sufficiently important by IBM that in 1968 it added a guard digit to the double precision format in the System/360 architecture (single precision already had a guard digit), and retrofitted all existing machines in the field. Two examples are given to illustrate the utility of guard digits.

The IEEE standard goes further than just requiring the use of a guard digit. It gives an algorithm for addition, subtraction, multiplication, division and square root, and requires that implementations produce the same result as that algorithm. Thus, when a program is moved from one machine to another, the results of the basic operations will be the same in every bit if both machines support the IEEE standard. This greatly simplifies the porting of programs. Other uses of this precise specification are given in [Exactly Rounded Operations](https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html#704).

Floating-point Formats

Several different representations of real numbers have been proposed, but by far the most widely used is the floating-point representation.[1](https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html#1370) Floating-point representations have a base  (which is always assumed to be even) and a precision p. If  = 10 and p = 3, then the number 0.1 is represented as 1.00 × 10-1. If  = 2 and p = 24, then the decimal number 0.1 cannot be represented exactly, but is approximately 1.10011001100110011001101 × 2-4.

In general, a floating-point number will be represented as ± d.dd... d × e, where d.dd... d is called the significand[2](https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html#1377) and has p digits. More precisely ± d0 . d1 d2 ... dp-1 × e represents the num

The term floating-point number will be used to mean a real number that can be exactly represented in the format under discussion. Two other parameters associated with floating-point representations are the largest and smallest allowable exponents, emax and emin. Since there are p possible significands, and emax - emin + 1 possible exponents, a floating-point number can be encoded in

bits, where the final +1 is for the sign bit. The precise encoding is not important for now.

There are two reasons why a real number might not be exactly representable as a floating-point number. The most common situation is illustrated by the decimal number 0.1. Although it has a finite decimal representation, in binary it has an infinite repeating representation. Thus when  = 2, the number 0.1 lies strictly between two floating-point numbers and is exactly representable by neither of them. A less common situation is that a real number is out of range, that is, its absolute value is larger than  ×  or smaller than 1.0 ×  . Most of this paper discusses issues due to the first reason. However, numbers that are out of range will be discussed in the sections [Infinity](https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html#918) and [Denormalized Numbers](https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html#929).

Floating-point representations are not necessarily unique. For example, both 0.01 × 101 and 1.00 × 10-1 represent 0.1. If the leading digit is nonzero (d0  0 in equation [(1)](https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html#687) above), then the representation is said to be normalized. The floating-point number 1.00 × 10-1 is normalized, while 0.01 × 101 is not. When  = 2, p = 3, emin = -1 and emax = 2 there are 16 normalized floating-point numbers, as shown in [FIGURE D-1](https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html#1374). The bold hash marks correspond to numbers whose significand is 1.00. Requiring that a floating-point representation be normalized makes the representation unique. Unfortunately, this restriction makes it impossible to represent zero! A natural way to represent 0 is with 1.0 ×  , since this preserves the fact that the numerical ordering of nonnegative real numbers corresponds to the lexicographic ordering of their floating-point representations.[3](https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html#685) When the exponent is stored in a k bit field, that means that only 2k - 1 values are available for use as exponents, since one must be reserved to represent 0.

Note that the × in a floating-point number is part of the notation, and different from a floating-point multiply operation. The meaning of the × symbol should be clear from the context. For example, the expression (2.5 × 10-3) × (4.0 × 102) involves only a single floating-point multiplication.

8. Describe a quantity that can be accurately expressed by the Decimal or Fraction classes but not by a floating-point value.

The Python decimal module provides support for fast correctly-rounded decimal floating point arithmetic.

By default, Python interprets any number that includes a decimal point as a double precision floating point number. The Decimal is a floating decimal point type which more precision and a smaller range than the float. It is appropriate for financial and monetary calculations. It is also closer to the way how humans work with numbers.

Unlike hardware based binary floating point, the decimal module has a user alterable precision which can be as large as needed for a given problem. The default precision is 28 places.

Some values cannot be exactly represented in a float data type. For instance, storing the 0.1 value in float (which is a binary floating point value) variable we get only an approximation of the value. Similarly, the 1/3 value cannot be represented exactly in decimal floating point type.

Q9.Consider the following two fraction objects: Fraction(1, 2) and Fraction(1, 2). (5, 10). Is the internal state of these two objects the same? Why do you think that is?

Fractions are numbers that represent a part of the whole. When an object or a group of [objects](https://www.toppr.com/guides/science/sorting-materials-into-group/objects-around-us/) is divided into equal parts, then each individual part is a fraction. A fraction is usually written as 1/2 or 5/12 or 7/18 and so on. It is divided into a numerator and denominator where the denominator represents the total [number](https://www.toppr.com/guides/maths/knowing-our-numbers/operations-on-numbers/) of equal parts into which the whole is divided. The numerator is the number of equal parts that are taken out. For e.g. in the [fraction](https://www.toppr.com/guides/maths/fractions/introduction-to-fraction/) 3/4, 3 is the numerator and 4 is the denominator

Real Life Example of a Fraction

It’s your birthday and mom has ordered pizza for you and your friends. When the pizza arrives, you open the box and find that it is cut into slices. Let’s assume that there are 8 slices and you have 7 [friends](https://www.toppr.com/guides/essays/essay-on-friendship/). So, there are 8 people who are going to eat the 8 slices of the pizza.

How much does each person get? Well, if we divide the entire pizza into eight equal parts, then each person gets 1/8 or one-eighth of the pizza. The pizza can be cut into a different number of equal slices creating different fractions. (Like a 6-slice pizza or a 4-slice pizza or a 12-slice pizza)

Fractions on a Number Line

Fractions can also be shown on a number line just like whole numbers. Let’s try and plot 1/2 on a number line. Now, we know that 1/2 is greater than 0 but lesser than 1. Hence, it lies between 0 and 1. Further, since the denominator is 2, we divide the [distance](https://www.toppr.com/guides/quantitative-aptitude/number-series/heights-and-distances/) between 0 and 1 into two equal parts. Refer diagram below:

Let’s look at one more example. How can we show 2/3 on a number [line](https://www.toppr.com/guides/maths/basic-geometrical-ideas/lines/)? Again, we know that 2/3 is greater than 0 but less than 1 (since the numerator is smaller than the denominator). Next, since the denominator is 3, we divide the distance between 0 and 1 into three equal parts. Now, 2/3 is two parts out of these three parts as shown below:

Proper, Improper and Mixed Fractions

In a fraction there are two simple possibilities:

The numerator is smaller than the denominator

The numerator is bigger than the denominator

Proper Fraction

When the numerator is smaller than the denominator, it is a Proper Fraction. These fractions are less than 1 and none of them lies beyond 1 on the number line. The denominator represents the number of equal parts in which the whole is divided.

And, the numerator represents the number of these equal parts that are considered. The example of the pizza stated above with each person getting 1/8th of the pizza shows a proper fraction.

Improper Fraction

When the numerator is larger than the denominator, it is an Improper Fraction. These fractions are greater than 1 and lie beyond 1 on the number line. The come into play when more than one object is divided equally into certain parts. The denominator represents the number of equal parts required. The numerator is the number of objects available.

Q10. How do the Fraction class and the integer type (int) relate to each other? Containment or inheritance?

A very common example to show the details of implementing a user-defined class is to construct a class to implement the abstract data type Fraction. We have already seen that Python provides a number of numeric classes for our use. There are times, however, that it would be most appropriate to be able to create data objects that “look like” fractions.

A fraction such as 35 consists of two parts. The top value, known as the numerator, can be any integer. The bottom value, called the denominator, can be any integer greater than 0 (negative fractions have a negative numerator). Although it is possible to create a floating point approximation for any fraction, in this case we would like to represent the fraction as an exact value.

The operations for the Fraction type will allow a Fraction data object to behave like any other numeric value. We need to be able to add, subtract, multiply, and divide fractions. We also want to be able to show fractions using the standard “slash” form, for example 3/5. In addition, all fraction methods should return results in their lowest terms so that no matter what computation is performed, we always end up with the most common form.

In Python, we define a new class by providing a name and a set of method definitions that are syntactically similar to function definitions. For this example,

class Fraction:

#the methods go here

provides the framework for us to define the methods. The first method that all classes should provide is the constructor. The constructor defines the way in which data objects are created. To create a Fraction object, we will need to provide two pieces of data, the numerator and the denominator. In Python, the constructor method is always called \_\_init\_\_ (two underscores before and after init) and is shown in [Listing 2](https://runestone.academy/ns/books/published/pythonds/Introduction/ObjectOrientedProgramminginPythonDefiningClasses.html#lst-pyconstructor).

Listing 2

class Fraction:

def \_\_init\_\_(self,top,bottom):

self.num = top

self.den = bottom

Notice that the formal parameter list contains three items (self, top, bottom). self is a special parameter that will always be used as a reference back to the object itself. It must always be the first formal parameter; however, it will never be given an actual parameter value upon invocation. As described earlier, fractions require two pieces of state data, the numerator and the denominator. The notation self.num in the constructor defines the fraction object to have an internal data object called num as part of its state. Likewise, self.den creates the denominator. The values of the two formal parameters are initially assigned to the state, allowing the new fraction object to know its starting value.

To create an instance of the Fraction class, we must invoke the constructor. This happens by using the name of the class and passing actual values for the necessary state (note that we never directly invoke \_\_init\_\_). For example,

myfraction = Fraction(3,5)

creates an object called myfraction representing the fraction 35 (three-fifths). [Figure 5](https://runestone.academy/ns/books/published/pythonds/Introduction/ObjectOrientedProgramminginPythonDefiningClasses.html#fig-fraction1) shows this object as it is now implemented.

Figure 5: An Instance of the Fraction Class

The next thing we need to do is implement the behavior that the abstract data type requires. To begin, consider what happens when we try to print a Fraction object.

>>> myf = Fraction(3,5)

>>> print(myf)

<\_\_main\_\_.Fraction instance at 0x409b1acc>

The fraction object, myf, does not know how to respond to this request to print. The print function requires that the object convert itself into a string so that the string can be written to the output. The only choice myf has is to show the actual reference that is stored in the variable (the address itself). This is not what we want.

There are two ways we can solve this problem. One is to define a method called show that will allow the Fraction object to print itself as a string. We can implement this method as shown in [Listing 3](https://runestone.academy/ns/books/published/pythonds/Introduction/ObjectOrientedProgramminginPythonDefiningClasses.html#lst-showmethod). If we create a Fraction object as before, we can ask it to show itself, in other words, print itself in the proper format. Unfortunately, this does not work in general. In order to make printing work properly, we need to tell the Fraction class how to convert itself into a string. This is what the print function needs in order to do its job.

Listing 3

def show(self):

print(self.num,"/",self.den)

>>> myf = Fraction(3,5)

>>> myf.show()

3 / 5

>>> print(myf)

<\_\_main\_\_.Fraction instance at 0x40bce9ac>

>>>

In Python, all classes have a set of standard methods that are provided but may not work properly. One of these, \_\_str\_\_, is the method to convert an object into a string. The default implementation for this method is to return the instance address string as we have already seen. What we need to do is provide a “better” implementation for this method. We will say that this implementation overrides the previous one, or that it redefines the method’s behavior.

To do this, we simply define a method with the name \_\_str\_\_ and give it a new implementation as shown in [Listing 4](https://runestone.academy/ns/books/published/pythonds/Introduction/ObjectOrientedProgramminginPythonDefiningClasses.html#lst-str). This definition does not need any other information except the special parameter self. In turn, the method will build a string representation by converting each piece of internal state data to a string and then placing a / character in between the strings using string concatenation. The resulting string will be returned any time a Fraction object is asked to convert itself to a string. Notice the various ways that this function is used.

Listing 4

def \_\_str\_\_(self):

return str(self.num)+"/"+str(self.den)

>>> myf = Fraction(3,5)

>>> print(myf)

3/5

>>> print("I ate", myf, "of the pizza")

I ate 3/5 of the pizza

>>> myf.\_\_str\_\_()

'3/5'

>>> str(myf)

'3/5'

>>>

We can override many other methods for our new Fraction class. Some of the most important of these are the basic arithmetic operations. We would like to be able to create two Fraction objects and then add them together using the standard “+” notation. At this point, if we try to add two fractions, we get the following:

>>> f1 = Fraction(1,4)

>>> f2 = Fraction(1,2)

>>> f1+f2

Traceback (most recent call last):

File "<pyshell#173>", line 1, in -toplevel-

f1+f2

TypeError: unsupported operand type(s) for +:

'instance' and 'instance'

>>>

If you look closely at the error, you see that the problem is that the “+” operator does not understand the Fraction operands.

We can fix this by providing the Fraction class with a method that overrides the addition method. In Python, this method is called \_\_add\_\_ and it requires two parameters. The first, self, is always needed, and the second represents the other operand in the expression. For example,

f1.\_\_add\_\_(f2)

would ask the Fraction object f1 to add the Fraction object f2 to itself. This can be written in the standard notation, f1+f2.

Two fractions must have the same denominator to be added. The easiest way to make sure they have the same denominator is to simply use the product of the two denominators as a common denominator so that ab+cd=adbd+cbbd=ad+cbbd The implementation is shown in [Listing 5](https://runestone.academy/ns/books/published/pythonds/Introduction/ObjectOrientedProgramminginPythonDefiningClasses.html#lst-addmethod). The addition function returns a new Fraction object with the numerator and denominator of the sum. We can use this method by writing a standard arithmetic expression involving fractions, assigning the result of the addition, and then printing our result.