# merge3

June 14, 2023

## 0.1 Integers

Integers are objects - instances of the `int` class.

[1]:
```
print(type(100))
```

```
<class 'int'>
```

They are a variable length data type that can theoretically handle any integer magnitude. This will take up a variable amount of memory that depends on the particular size of the integer.

[2]:
```
import sys
```

Creating an integer object requires an overhead of 24 bytes:

[3]:
```
sys.getsizeof(0)
```

[3]: 24

Here we see that to store the number 1 required 4 bytes (32 bits) on top of the 24 byte overhead:

[4]:
```
sys.getsizeof(1)
```

[4]: 28

Larger numbers will require more storage space:

[5]:
```
sys.getsizeof(2**1000)
```

[5]: 160

Larger integers will also slow down calculations.

[6]:
```
import time
```

[7]:
```
def calc(a):
    for i in range(10000000):
        a * 2
```

We start with a small integer value for a (10):

[8]:
```
start = time.perf_counter()
calc(10)
end = time.perf_counter()
print(end - start)
```

```
0.3565246949777869
```

Now we set a to something larger (2100):

[9]:
```
start = time.perf_counter()
calc(2**100)
end = time.perf_counter()
print(end - start)
```

```
0.6125349575326144
```

Finally we set a to some really large value (210,000):

[10]:
```
start = time.perf_counter()
calc(2**10000)
end = time.perf_counter()
print(end - start)
```

```
5.023413039975091
```

## 0.2 Integers - Operations

Addition, subtraction, multiplication and exponentiation of integers always result in an integer. (In the case of exponentiation this holds only for positive integer exponents.)

[1]:
```
type(2 + 3)
```

[1]:
```
int
```

[2]:
```
type(3 - 10)
```

[2]:
```
int
```

[3]:
```
type(3 * 5)
```

[3]:
```
int
```

[4]:
```
type(3 ** 4)
```

[4]:
```
int
```

But the standard division operator **/ always** results in a float value.

[5]:
```
type(2 / 3)
```

[5]:
```
float
```

[6]:
```
type(10 / 2)
```

[6]:
```
float
```

The `math.floor()` method will return the floor of any number.

[7]:
```
import math
```

For non-negative values (>= 0), the floor of the value is the same as the integer portion of the value (truncation)

[8]:
```
math.floor(3.15)
```

[8]:
```
3
```

[9]:
```
math.floor(3.9999999)
```

[9]:
```
3
```

However, this is not the case for negative values:

[10]:
```
math.floor(-3.15)
```

[10]:
```
-4
```

[11]:
```
math.floor(-3.0000001)
```

[11]:
```
-4
```

**The Floor Division Operator** The floor division operator `a//b` is the floor of `a / b`

i.e. `a // b = math.floor(a / b)`

This is true whether `a` and `b` are positive or negative.

[12]:
```
a = 33
b = 16
print(a/b)
print(a//b)
print(math.floor(a/b))
```

```
2.0625
2
2
```

For positive numbers, `a//b` is basically the same as truncating (taking the integer portion) of `a / b`.

But this is **not** the case for negative numbers.

[13]:
```
a = -33
b = 16
print('{0}/{1} = {2}'.format(a, b, a/b))
print('trunc({0}/{1}) = {2}'.format(a, b, math.trunc(a/b)))
print('{0}//{1} = {2}'.format(a, b, a//b))
print('floor({0}//{1}) = {2}'.format(a, b, math.floor(a/b)))
```

```
-33/16 = -2.0625
trunc(-33/16) = -2
-33//16 = -3
floor(-33//16) = -3
```

[14]:
```
a = 33
b = -16
print('{0}/{1} = {2}'.format(a, b, a/b))
print('trunc({0}/{1}) = {2}'.format(a, b, math.trunc(a/b)))
print('{0}//{1} = {2}'.format(a, b, a//b))
print('floor({0}//{1}) = {2}'.format(a, b, math.floor(a/b)))
```

```
33/-16 = -2.0625
trunc(33/-16) = -2
33//-16 = -3
floor(33//-16) = -3
```

**The Modulo Operator** The modulo operator and the floor division operator will always satisfy the following equation:

```
a = b * (a // b) + a % b
```

[15]:
```
a = 13
b = 4
print('{0}/{1} = {2}'.format(a, b, a/b))
print('{0}//{1} = {2}'.format(a, b, a//b))
print('{0}%{1} = {2}'.format(a, b, a%b))
print(a == b * (a//b) + a%b)
```

```
13/4 = 3.25
13//4 = 3
13%4 = 1
True
```

[16]:
```
a = -13
b = 4
print('{0}/{1} = {2}'.format(a, b, a/b))
print('{0}//{1} = {2}'.format(a, b, a//b))
print('{0}%{1} = {2}'.format(a, b, a%b))
print(a == b * (a//b) + a%b)
```

```
-13/4 = -3.25
-13//4 = -4
-13%4 = 3
True
```

[17]:
```
a = 13
b = -4
print('{0}/{1} = {2}'.format(a, b, a/b))
print('{0}//{1} = {2}'.format(a, b, a//b))
print('{0}%{1} = {2}'.format(a, b, a%b))
print(a == b * (a//b) + a%b)
```

```
13/-4 = -3.25
13//-4 = -4
13%-4 = -3
True
```

[18]:
```
a = -13
b = -4
print('{0}/{1} = {2}'.format(a, b, a/b))
print('{0}//{1} = {2}'.format(a, b, a//b))
print('{0}%{1} = {2}'.format(a, b, a%b))
print(a == b * (a//b) + a%b)
```

```
-13/-4 = 3.25
-13//-4 = 3
-13%-4 = -1
True
```

### 0.3 Integers - Constructors and Bases

**Constructors** The `int` class has two constructors

[15]:
```
help(int)
```

```
Help on class int in module builtins:

class int(object)
 |  int(x=0) -> integer
 |  int(x, base=10) -> integer
 |
 |  Convert a number or string to an integer, or return 0 if no arguments
 |  are given.  If x is a number, return x.__int__().  For floating point
 |  numbers, this truncates towards zero.
 |
 |  If x is not a number or if base is given, then x must be a string,
 |  bytes, or bytearray instance representing an integer literal in the
 |  given base.  The literal can be preceded by '+' or '-' and be surrounded
 |  by whitespace.  The base defaults to 10.  Valid bases are 0 and 2-36.
```

```
|  Base 0 means to interpret the base from the string as an integer literal.
|  >>> int('0b100', base=0)
|  4
|
|  Methods defined here:
|
|  __abs__(self, /)
|      abs(self)
|
|  __add__(self, value, /)
|      Return self+value.
|
|  __and__(self, value, /)
|      Return self&value.
|
|  __bool__(self, /)
|      self != 0
|
|  __ceil__(…)
|      Ceiling of an Integral returns itself.
|
|  __divmod__(self, value, /)
|      Return divmod(self, value).
|
|  __eq__(self, value, /)
|      Return self==value.
|
|  __float__(self, /)
|      float(self)
|
|  __floor__(…)
|      Flooring an Integral returns itself.
|
|  __floordiv__(self, value, /)
|      Return self//value.
|
|  __format__(…)
|      default object formatter
|
|  __ge__(self, value, /)
|      Return self>=value.
|
|  __getattribute__(self, name, /)
|      Return getattr(self, name).
|
|  __getnewargs__(…)
|
|  __gt__(self, value, /)
```

```
 |      Return self>value.
 |
 |  __hash__(self, /)
 |      Return hash(self).
 |
 |  __index__(self, /)
 |      Return self converted to an integer, if self is suitable for use as an
index into a list.
 |
 |  __int__(self, /)
 |      int(self)
 |
 |  __invert__(self, /)
 |      ~self
 |
 |  __le__(self, value, /)
 |      Return self<=value.
 |
 |  __lshift__(self, value, /)
 |      Return self<<value.
 |
 |  __lt__(self, value, /)
 |      Return self<value.
 |
 |  __mod__(self, value, /)
 |      Return self%value.
 |
 |  __mul__(self, value, /)
 |      Return self*value.
 |
 |  __ne__(self, value, /)
 |      Return self!=value.
 |
 |  __neg__(self, /)
 |      -self
 |
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for accurate signature.
 |
 |  __or__(self, value, /)
 |      Return self|value.
 |
 |  __pos__(self, /)
 |      +self
 |
 |  __pow__(self, value, mod=None, /)
 |      Return pow(self, value, mod).
 |
```

```
|  __radd__(self, value, /)
|      Return value+self.
|
|  __rand__(self, value, /)
|      Return value&self.
|
|  __rdivmod__(self, value, /)
|      Return divmod(value, self).
|
|  __repr__(self, /)
|      Return repr(self).
|
|  __rfloordiv__(self, value, /)
|      Return value//self.
|
|  __rlshift__(self, value, /)
|      Return value<<self.
|
|  __rmod__(self, value, /)
|      Return value%self.
|
|  __rmul__(self, value, /)
|      Return value*self.
|
|  __ror__(self, value, /)
|      Return value|self.
|
|  __round__(…)
|      Rounding an Integral returns itself.
|      Rounding with an ndigits argument also returns an integer.
|
|  __rpow__(self, value, mod=None, /)
|      Return pow(value, self, mod).
|
|  __rrshift__(self, value, /)
|      Return value>>self.
|
|  __rshift__(self, value, /)
|      Return self>>value.
|
|  __rsub__(self, value, /)
|      Return value-self.
|
|  __rtruediv__(self, value, /)
|      Return value/self.
|
|  __rxor__(self, value, /)
|      Return value^self.
```

```
 |
 |  __sizeof__(…)
 |      Returns size in memory, in bytes
 |
 |  __str__(self, /)
 |      Return str(self).
 |
 |  __sub__(self, value, /)
 |      Return self-value.
 |
 |  __truediv__(self, value, /)
 |      Return self/value.
 |
 |  __trunc__(…)
 |      Truncating an Integral returns itself.
 |
 |  __xor__(self, value, /)
 |      Return self^value.
 |
 |  bit_length(…)
 |      int.bit_length() -> int
 |
 |      Number of bits necessary to represent self in binary.
 |      >>> bin(37)
 |      '0b100101'
 |      >>> (37).bit_length()
 |      6
 |
 |  conjugate(…)
 |      Returns self, the complex conjugate of any int.
 |
 |  from_bytes(…) from builtins.type
 |      int.from_bytes(bytes, byteorder, *, signed=False) -> int
 |
 |      Return the integer represented by the given array of bytes.
 |
 |      The bytes argument must be a bytes-like object (e.g. bytes or
bytearray).
 |
 |      The byteorder argument determines the byte order used to represent the
 |      integer.  If byteorder is 'big', the most significant byte is at the
 |      beginning of the byte array.  If byteorder is 'little', the most
 |      significant byte is at the end of the byte array.  To request the native
 |      byte order of the host system, use `sys.byteorder' as the byte order
value.
 |
 |      The signed keyword-only argument indicates whether two's complement is
 |      used to represent the integer.
```

```
 |
 |  to_bytes(…)
 |      int.to_bytes(length, byteorder, *, signed=False) -> bytes
 |
 |      Return an array of bytes representing an integer.
 |
 |      The integer is represented using length bytes.  An OverflowError is
 |      raised if the integer is not representable with the given number of
 |      bytes.
 |
 |      The byteorder argument determines the byte order used to represent the
 |      integer.  If byteorder is 'big', the most significant byte is at the
 |      beginning of the byte array.  If byteorder is 'little', the most
 |      significant byte is at the end of the byte array.  To request the native
 |      byte order of the host system, use `sys.byteorder' as the byte order
value.
 |
 |      The signed keyword-only argument determines whether two's complement is
 |      used to represent the integer.  If signed is False and a negative
integer
 |      is given, an OverflowError is raised.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  denominator
 |      the denominator of a rational number in lowest terms
 |
 |  imag
 |      the imaginary part of a complex number
 |
 |  numerator
 |      the numerator of a rational number in lowest terms
 |
 |  real
 |      the real part of a complex number
```

[16]:
```
int(10)
```

[16]: 10

[17]:
```
int(10.9)
```

[17]: 10

[18]:
```
int(-10.9)
```

[18]:
```
-10
```

[19]:
```
from fractions import Fraction
```

[20]:
```
a = Fraction(22, 7)
```

[21]:
```
a
```

[21]:
```
Fraction(22, 7)
```

[22]:
```
int(a)
```

[22]:
```
3
```

We can use the second constructor to generate integers (base 10) from strings in any base.

[23]:
```
int("10")
```

[23]:
```
10
```

[24]:
```
int("101", 2)
```

[24]:
```
5
```

[25]:
```
int("101", base=2)
```

[25]:
```
5
```

Python uses `a-z` for bases from 11 to 36.

Note that the letters are not case sensitive.

[26]:
```
int("F1A", base=16)
```

[26]:
```
3866
```

[27]:
```
int("f1a", base=16)
```

[27]:
```
3866
```

Of course, the string must be a valid number in whatever base you specify.

[28]:
```
int('B1A', base=11)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-28-d0cf8657e90a> in <module>()
----> 1 int('B1A', base=11)
```

```
ValueError: invalid literal for int() with base 11: 'B1A'
```

[ ]:
```
int('B1A', 12)
```

### Base Representations

#### Built-ins

[ ]:
```
bin(10)
```

[ ]:
```
oct(10)
```

[ ]:
```
hex(10)
```

Note the `0b`, `0o` and `0x` prefixes

You can use these in your own strings as well, and they correspond to prefixes used in integer literals as well.

[ ]:
```
a = int('1010', 2)
b = int('0b1010', 2)
c = 0b1010
```

[ ]:
```
print(a, b, c)
```

[ ]:
```
a = int('f1a', 16)
b = int('0xf1a', 16)
c = 0xf1a
```

[ ]:
```
print(a, b, c)
```

For literals, the `a-z` characters are not case-sensitive either

[ ]:
```
a = 0xf1a
b = 0xF1a
c = 0xF1A
```

[ ]:
```
print(a, b, c)
```

**Custom Rebasing** Python only provides built-in function to rebase to base 2, 8 and 16.

For other bases, you have to provide your own algorithm (or leverage some 3rd party library of your choice)

[ ]:
```
def from_base10(n, b):
    if b < 2:
        raise ValueError('Base b must be >= 2')
```

```
    if n < 0:
        raise ValueError('Number n must be >= 0')
    if n == 0:
        return [0]
    digits = []
    while n > 0:
        # m = n % b
        # n = n // b
        # which is the same as:
        n, m = divmod(n, b)
        digits.insert(0, m)
    return digits
```

[ ]:
```
from_base10(10, 2)
```

[ ]:
```
from_base10(255, 16)
```

Next we may want to encode the digits into strings using different characters for each digit in the base

[ ]:
```
def encode(digits, digit_map):
    # we require that digit_map has at least as many
    # characters as the max number in digits
    if max(digits) >= len(digit_map):
        raise ValueError("digit_map is not long enough to encode digits")

    # we'll see this later, but the following would be better:
    encoding = ''.join([digit_map[d] for d in digits])
    return encoding
```

Now we can encode any list of digits:

[ ]:
```
encode([1, 0, 1], "FT")
```

[ ]:
```
encode([1, 10, 11], '0123456789AB')
```

And we can combine both functions into a single one for easier use:

[ ]:
```
def rebase_from10(number, base):
    digit_map = '0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ'
    if base < 2 or base > 36:
        raise ValueError('Invalid base: 2 <= base <= 36')
    # we store the sign of number and make it positive
    # we'll re-insert the sign at the end
    sign = -1 if number < 0 else 1
    number *= sign

    digits = from_base10(number, base)
```

```
    encoding = encode(digits, digit_map)
    if sign == -1:
        encoding = '-' + encoding
    return encoding
```

[ ]:
```
e = rebase_from10(10, 2)
print(e)
print(int(e, 2))
```

[ ]:
```
e = rebase_from10(-10, 2)
print(e)
print(int(e, 2))
```

[ ]:
```
rebase_from10(131, 11)
```

[ ]:
```
rebase_from10(4095, 16)
```

[ ]:
```
rebase_from10(-4095, 16)
```

### 0.4 Rational Numbers

[1]:
```
from fractions import Fraction
```

We can get some info on the Fraction class:

[2]:
```
help(Fraction)
```

```
Help on class Fraction in module fractions:

class Fraction(numbers.Rational)
 |  This class implements rational numbers.
 |
 |  In the two-argument form of the constructor, Fraction(8, 6) will
 |  produce a rational number equivalent to 4/3. Both arguments must
 |  be Rational. The numerator defaults to 0 and the denominator
 |  defaults to 1 so that Fraction(3) == 3 and Fraction() == 0.
 |
 |  Fractions can also be constructed from:
 |
 |    - numeric strings similar to those accepted by the
 |      float constructor (for example, '-2.3' or '1e10')
 |
 |    - strings of the form '123/456'
 |
 |    - float and Decimal instances
 |
 |    - other Rational instances (including integers)
```

```
|
|  Method resolution order:
|      Fraction
|      numbers.Rational
|      numbers.Real
|      numbers.Complex
|      numbers.Number
|      builtins.object
|
|  Methods defined here:
|
|  __abs__(a)
|      abs(a)
|
|  __add__(a, b)
|      a + b
|
|  __bool__(a)
|      a != 0
|
|  __ceil__(a)
|      Will be math.ceil(a) in 3.0.
|
|  __copy__(self)
|
|  __deepcopy__(self, memo)
|
|  __eq__(a, b)
|      a == b
|
|  __floor__(a)
|      Will be math.floor(a) in 3.0.
|
|  __floordiv__(a, b)
|      a // b
|
|  __ge__(a, b)
|      a >= b
|
|  __gt__(a, b)
|      a > b
|
|  __hash__(self)
|      hash(self)
|
|  __le__(a, b)
|      a <= b
|
```

```
|  __lt__(a, b)
|      a < b
|
|  __mod__(a, b)
|      a % b
|
|  __mul__(a, b)
|      a * b
|
|  __neg__(a)
|      -a
|
|  __pos__(a)
|      +a: Coerces a subclass instance to Fraction
|
|  __pow__(a, b)
|      a ** b
|
|      If b is not an integer, the result will be a float or complex
|      since roots are generally irrational. If b is an integer, the
|      result will be rational.
|
|  __radd__(b, a)
|      a + b
|
|  __reduce__(self)
|      helper for pickle
|
|  __repr__(self)
|      repr(self)
|
|  __rfloordiv__(b, a)
|      a // b
|
|  __rmod__(b, a)
|      a % b
|
|  __rmul__(b, a)
|      a * b
|
|  __round__(self, ndigits=None)
|      Will be round(self, ndigits) in 3.0.
|
|      Rounds half toward even.
|
|  __rpow__(b, a)
|      a ** b
|
```

```
|  __rsub__(b, a)
|      a - b
|
|  __rtruediv__(b, a)
|      a / b
|
|  __str__(self)
|      str(self)
|
|  __sub__(a, b)
|      a - b
|
|  __truediv__(a, b)
|      a / b
|
|  __trunc__(a)
|      trunc(a)
|
|  limit_denominator(self, max_denominator=1000000)
|      Closest Fraction to self with denominator at most max_denominator.
|
|      >>> Fraction('3.141592653589793').limit_denominator(10)
|      Fraction(22, 7)
|      >>> Fraction('3.141592653589793').limit_denominator(100)
|      Fraction(311, 99)
|      >>> Fraction(4321, 8765).limit_denominator(10000)
|      Fraction(4321, 8765)
|
|  ----------------------------------------------------------------------
|  Class methods defined here:
|
|  from_decimal(dec) from abc.ABCMeta
|      Converts a finite Decimal instance to a rational number, exactly.
|
|  from_float(f) from abc.ABCMeta
|      Converts a finite float to a rational number, exactly.
|
|      Beware that Fraction.from_float(0.3) != Fraction(3, 10).
|
|  ----------------------------------------------------------------------
|  Static methods defined here:
|
|  __new__(cls, numerator=0, denominator=None, *, _normalize=True)
|      Constructs a Rational.
|
|      Takes a string like '3/2' or '1.5', another Rational instance, a
|      numerator/denominator pair, or a float.
|
```

```
|      Examples
|      --------
|
|      >>> Fraction(10, -8)
|      Fraction(-5, 4)
|      >>> Fraction(Fraction(1, 7), 5)
|      Fraction(1, 35)
|      >>> Fraction(Fraction(1, 7), Fraction(2, 3))
|      Fraction(3, 14)
|      >>> Fraction('314')
|      Fraction(314, 1)
|      >>> Fraction('-35/4')
|      Fraction(-35, 4)
|      >>> Fraction('3.1415') # conversion from numeric string
|      Fraction(6283, 2000)
|      >>> Fraction('-47e-2') # string may include a decimal exponent
|      Fraction(-47, 100)
|      >>> Fraction(1.47)  # direct construction from float (exact conversion)
|      Fraction(6620291452234629, 4503599627370496)
|      >>> Fraction(2.25)
|      Fraction(9, 4)
|      >>> Fraction(Decimal('1.47'))
|      Fraction(147, 100)
|
|  ----------------------------------------------------------------------
|  Data descriptors defined here:
|
|  denominator
|
|  numerator
|
|  ----------------------------------------------------------------------
|  Data and other attributes defined here:
|
|  __abstractmethods__ = frozenset()
|
|  ----------------------------------------------------------------------
|  Methods inherited from numbers.Rational:
|
|  __float__(self)
|      float(self) = self.numerator / self.denominator
|
|      It's important that this conversion use the integer's "true"
|      division rather than casting one side to float before dividing
|      so that ratios of huge integers convert without overflowing.
|
|  ----------------------------------------------------------------------
|  Methods inherited from numbers.Real:
```

```
|
|  __complex__(self)
|      complex(self) == complex(float(self), 0)
|
|  __divmod__(self, other)
|      divmod(self, other): The pair (self // other, self % other).
|
|      Sometimes this can be computed faster than the pair of
|      operations.
|
|  __rdivmod__(self, other)
|      divmod(other, self): The pair (self // other, self % other).
|
|      Sometimes this can be computed faster than the pair of
|      operations.
|
|  conjugate(self)
|      Conjugate is a no-op for Reals.
|
|  ----------------------------------------------------------------------
|  Data descriptors inherited from numbers.Real:
|
|  imag
|      Real numbers have no imaginary component.
|
|  real
|      Real numbers are their real component.
```

We can create Fraction objects in a variety of ways:

Using integers:

[3]:
```
Fraction(1)
```

[3]:
```
Fraction(1, 1)
```

[4]:
```
Fraction(1, 3)
```

[4]:
```
Fraction(1, 3)
```

Using rational numbers:

[5]:
```
x = Fraction(2, 3)
y = Fraction(3, 4)
# 2/3 / 3/4 --> 2/3 * 4/3 --> 8/9
Fraction(x, y)
```

[5]:
```
Fraction(8, 9)
```

Using floats:

[6]:
```
Fraction(0.125)
```

[6]:
```
Fraction(1, 8)
```

[7]:
```
Fraction(0.5)
```

[7]:
```
Fraction(1, 2)
```

Using strings:

[8]:
```
Fraction('10.5')
```

[8]:
```
Fraction(21, 2)
```

[9]:
```
Fraction('22/7')
```

[9]:
```
Fraction(22, 7)
```

Fractions are automatically reduced:

[10]:
```
Fraction(8, 16)
```

[10]:
```
Fraction(1, 2)
```

Negative sign is attached to the numerator:

[11]:
```
Fraction(1, -4)
```

[11]:
```
Fraction(-1, 4)
```

Standard arithmetic operators are supported:

[12]:
```
Fraction(1, 3) + Fraction(1, 3) + Fraction(1, 3)
```

[12]:
```
Fraction(1, 1)
```

[13]:
```
Fraction(1, 2) * Fraction(1, 4)
```

[13]:
```
Fraction(1, 8)
```

[14]:
```
Fraction(1, 2) / Fraction(1, 3)
```

[14]:
```
Fraction(3, 2)
```

We can recover the numerator and denominator (integers):

[15]:
```
x = Fraction(22, 7)
print(x.numerator)
print(x.denominator)
```

```
22
7
```

Since floats have **finite** precision, any float can be converted to a rational number:

[16]:
```
import math
x = Fraction(math.pi)
print(x)
print(float(x))
```

```
884279719003555/281474976710656
3.141592653589793
```

[17]:
```
x = Fraction(math.sqrt(2))
print(x)
```

```
6369051672525773/4503599627370496
```

Note that these rational values are approximations to the irrational numbers $\pi$ and $\sqrt{2}$

**Beware!!**

Float number representations (as we will examine in future lessons) do not always have an exact representation.

The number 0.125 (1/8) **has** an exact representation:

[18]:
```
Fraction(0.125)
```

[18]:
```
Fraction(1, 8)
```

and so we see the expected equivalent fraction.

But, 0.3 (3/10) does **not** have an exact representation:

[19]:
```
Fraction(3, 10)
```

[19]:
```
Fraction(3, 10)
```

but

[20]:
```
Fraction(0.3)
```

[20]:
```
Fraction(5404319552844595, 18014398509481984)
```

We will study this in upcoming lessons.

But for now, let's just see a quick explanation:

[21]:
```
x = 0.3
```

[22]:
```
print(x)
```

```
0.3
```

Everything looks ok here - why am I saying 0.3 (float) is just an approximation?

Python is trying to format the displayed value for readability - so it rounds the number for a better display format!

We can instead choose to display the value using a certain number of digits:

[23]:
```
format(x, '.5f')
```

[23]:
```
'0.30000'
```

At 5 digits after the decimal, we might still think 0.3 is an exact representation.

But let's display a few more digits:

[24]:
```
format(x, '.15f')
```

[24]:
```
'0.300000000000000'
```

Hmm… 15 digits and still looking good!

How about 25 digits…

[25]:
```
format(x, '.25f')
```

[25]:
```
'0.2999999999999999888977698'
```

Now we see that **x** is not quite 0.3…

In fact, we can quantify the delta this way:

[26]:
```
delta = Fraction(0.3) - Fraction(3, 10)
```

Theoretically, delta should be 0, but it's not:

[27]:
```
delta == 0
```

[27]:
```
False
```

[28]:
```
delta
```

[28]:
```
Fraction(-1, 90071992547409920)
```

**delta** is a very small number, the above fraction…

As a float:

[29]:
```
float(delta)
```

[29]:
```
-1.1102230246251566e-17
```

**Constraining the denominator**

[30]:
```
x = Fraction(math.pi)
print(x)
print(format(float(x), '.25f'))
```

```
884279719003555/281474976710656
3.1415926535897931159979635
```

[31]:
```
y = x.limit_denominator(10)
print(y)
print(format(float(y), '.25f'))
```

```
22/7
3.1428571428571427937015414
```

[32]:
```
y = x.limit_denominator(100)
print(y)
print(format(float(y), '.25f'))
```

```
311/99
3.1414141414141414365701621
```

[33]:
```
y = x.limit_denominator(500)
print(y)
print(format(float(y), '.25f'))
```

```
355/113
3.1415929203539825209645642
```

## 0.5 Floats - Internal Representation

The `float` class can be used to represent real numbers.

[1]:
```
help(float)
```

```
Help on class float in module builtins:

class float(object)
 |  float(x) -> floating point number
 |
 |  Convert a string or number to a floating point number, if possible.
 |
 |  Methods defined here:
 |
```

```
 |  __abs__(self, /)
 |      abs(self)
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __bool__(self, /)
 |      self != 0
 |
 |  __divmod__(self, value, /)
 |      Return divmod(self, value).
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
 |  __float__(self, /)
 |      float(self)
 |
 |  __floordiv__(self, value, /)
 |      Return self//value.
 |
 |  __format__(…)
 |      float.__format__(format_spec) -> string
 |
 |      Formats the float according to format_spec.
 |
 |  __ge__(self, value, /)
 |      Return self>=value.
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __getformat__(…) from builtins.type
 |      float.__getformat__(typestr) -> string
 |
 |      You probably don't want to use this function.  It exists mainly to be
 |      used in Python's test suite.
 |
 |      typestr must be 'double' or 'float'.  This function returns whichever of
 |      'unknown', 'IEEE, big-endian' or 'IEEE, little-endian' best describes
the
 |      format of floating point numbers used by the C type named by typestr.
 |
 |  __getnewargs__(…)
 |
 |  __gt__(self, value, /)
 |      Return self>value.
 |
```

```
|  __hash__(self, /)
|      Return hash(self).
|
|  __int__(self, /)
|      int(self)
|
|  __le__(self, value, /)
|      Return self<=value.
|
|  __lt__(self, value, /)
|      Return self<value.
|
|  __mod__(self, value, /)
|      Return self%value.
|
|  __mul__(self, value, /)
|      Return self*value.
|
|  __ne__(self, value, /)
|      Return self!=value.
|
|  __neg__(self, /)
|      -self
|
|  __new__(*args, **kwargs) from builtins.type
|      Create and return a new object.  See help(type) for accurate signature.
|
|  __pos__(self, /)
|      +self
|
|  __pow__(self, value, mod=None, /)
|      Return pow(self, value, mod).
|
|  __radd__(self, value, /)
|      Return value+self.
|
|  __rdivmod__(self, value, /)
|      Return divmod(value, self).
|
|  __repr__(self, /)
|      Return repr(self).
|
|  __rfloordiv__(self, value, /)
|      Return value//self.
|
|  __rmod__(self, value, /)
|      Return value%self.
|
```

```
|  __rmul__(self, value, /)
|      Return value*self.
|
|  __round__(…)
|      Return the Integral closest to x, rounding half toward even.
|      When an argument is passed, work like built-in round(x, ndigits).
|
|  __rpow__(self, value, mod=None, /)
|      Return pow(value, self, mod).
|
|  __rsub__(self, value, /)
|      Return value-self.
|
|  __rtruediv__(self, value, /)
|      Return value/self.
|
|  __setformat__(…) from builtins.type
|      float.__setformat__(typestr, fmt) -> None
|
|      You probably don't want to use this function.  It exists mainly to be
|      used in Python's test suite.
|
|      typestr must be 'double' or 'float'.  fmt must be one of 'unknown',
|      'IEEE, big-endian' or 'IEEE, little-endian', and in addition can only be
|      one of the latter two if it appears to match the underlying C reality.
|
|      Override the automatic determination of C-level floating point type.
|      This affects how floats are converted to and from binary strings.
|
|  __str__(self, /)
|      Return str(self).
|
|  __sub__(self, value, /)
|      Return self-value.
|
|  __truediv__(self, value, /)
|      Return self/value.
|
|  __trunc__(…)
|      Return the Integral closest to x between 0 and x.
|
|  as_integer_ratio(…)
|      float.as_integer_ratio() -> (int, int)
|
|      Return a pair of integers, whose ratio is exactly equal to the original
|      float and with a positive denominator.
|      Raise OverflowError on infinities and a ValueError on NaNs.
|
```

```
|      >>> (10.0).as_integer_ratio()
|      (10, 1)
|      >>> (0.0).as_integer_ratio()
|      (0, 1)
|      >>> (-.25).as_integer_ratio()
|      (-1, 4)
|
|  conjugate(…)
|      Return self, the complex conjugate of any float.
|
|  fromhex(…) from builtins.type
|      float.fromhex(string) -> float
|
|      Create a floating-point number from a hexadecimal string.
|      >>> float.fromhex('0x1.ffffp10')
|      2047.984375
|      >>> float.fromhex('-0x1p-1074')
|      -5e-324
|
|  hex(…)
|      float.hex() -> string
|
|      Return a hexadecimal representation of a floating-point number.
|      >>> (-0.1).hex()
|      '-0x1.999999999999ap-4'
|      >>> 3.14159.hex()
|      '0x1.921f9f01b866ep+1'
|
|  is_integer(…)
|      Return True if the float is an integer.
|
|  ----------------------------------------------------------------------
|  Data descriptors defined here:
|
|  imag
|      the imaginary part of a complex number
|
|  real
|      the real part of a complex number
```

The `float` class has a single constructor, which can take a number or a string and will attempt to convert it to a float.

[2]:
```
float(10)
```

[2]:
```
10.0
```

[3]:
```
float(3.14)
```

[3]:
```
3.14
```

[4]:
```
float('0.1')
```

[4]:
```
0.1
```

However, strings that represent fractions cannot be converted to floats, unlike the Fraction class we saw earlier.

[5]:
```
float('22/7')
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-5-32cff4369993> in <module>()
----> 1 float('22/7')

ValueError: could not convert string to float: '22/7'
```

If you really want to get a float from a string such as `'22/7'`, you could first create a `Fraction`, then create a `float` from that:

[ ]:
```
from fractions import Fraction
```

[ ]:
```
float(Fraction('22/7'))
```

Floats do not always have an exact representation:

[ ]:
```
print(0.1)
```

Although this looks like `0.1` exactly, we need to reveal more digits after the decimal point to see what's going on:

[ ]:
```
format(0.1, '.25f')
```

However, certain numbers can be represented exactly in a binary fraction expansion:

[ ]:
```
format(0.125, '.25f')
```

This is because 0.125 is precisely 1/8, or 1/(2^3)

### 0.6 Floats - Equality Testing

Because not all real numbers have an exact `float` representation, equality testing can be tricky.

[1]:
```
x = 0.1 + 0.1 + 0.1
y = 0.3
x == y
```

[1]:
```
False
```

This is because `0.1` and `0.3` do not have exact representations:

[2]:
```
print('0.1 --> {0:.25f}'.format(0.1))
print('x --> {0:.25f}'.format(x))
print('y --> {0:.25f}'.format(y))
```

```
0.1 --> 0.1000000000000000055511151
x --> 0.3000000000000000444089210
y --> 0.2999999999999999888977698
```

However, in some (limited) cases where all the numbers involved do have an exact representation, it will work:

[3]:
```
x = 0.125 + 0.125 + 0.125
y = 0.375
x == y
```

[3]:
```
True
```

[4]:
```
print('0.125 --> {0:.25f}'.format(0.125))
print('x --> {0:.25f}'.format(x))
print('y --> {0:.25f}'.format(y))
```

```
0.125 --> 0.1250000000000000000000000
x --> 0.3750000000000000000000000
y --> 0.3750000000000000000000000
```

One simple way to get around this is to round to a specific number of digits and then compare

[5]:
```
x = 0.1 + 0.1 + 0.1
y = 0.3
round(x, 5) == round(y, 5)
```

[5]:
```
True
```

We can also use a more flexible technique implemented by the `isclose` method in the `math` module

[6]:
```
from math import isclose
```

[7]:
```
help(isclose)
```

```
Help on built-in function isclose in module math:

isclose(…)
    isclose(a, b, *, rel_tol=1e-09, abs_tol=0.0) -> bool

    Determine whether two floating point numbers are close in value.
```

```
    rel_tol
        maximum difference for being considered "close", relative to the
        magnitude of the input values
     abs_tol
        maximum difference for being considered "close", regardless of the
        magnitude of the input values

Return True if a is close in value to b, and False otherwise.

For the values to be considered close, the difference between them
must be smaller than at least one of the tolerances.

-inf, inf and NaN behave similarly to the IEEE 754 Standard.  That
is, NaN is not close to anything, even itself.  inf and -inf are
only close to themselves.
```

[8]:
```
x = 0.1 + 0.1 + 0.1
y = 0.3
isclose(x, y)
```

[8]:
```
True
```

The `isclose` method takes two optional parameters, `rel_tol` and `abs_tol`.

`rel_tol` is a relative tolerance that will be relative to the magnitude of the largest of the two numbers being compared. Useful when we want to see if two numbers are close to each other as a percentage of their magnitudes.

`abs_tol` is an absolute tolerance that is independent of the magnitude of the numbers we are comparing - this is useful for numbers that are close to zero.

In this situation we might consider x and y to be close to each other:

[9]:
```
x = 123456789.01
y = 123456789.02
```

but not in this case:

[10]:
```
x = 0.01
y = 0.02
```

In both these cases the difference between the two numbers was `0.01`, yet in one case we considered the numbers "equal" and in the other, not "equal". Relative tolerances are useful to handle these scenarios.

[11]:
```
isclose(123456789.01, 123456789.02, rel_tol=0.01)
```

[11]:
```
True
```

[12]:
```
isclose(0.01, 0.02, rel_tol=0.01)
```

[12]:
```
False
```

On the other hand, we have to be careful with relative tolerances when working with values that are close to zero:

[13]:
```
x = 0.0000001
y = 0.0000002
isclose(x, y, rel_tol=0.01)
```

[13]:
```
False
```

So, we could use an absolute tolerance here:

[14]:
```
isclose(x, y, abs_tol=0.0001, rel_tol=0)
```

[14]:
```
True
```

In general, we can combine the use of both relative and absolute tolerances in this way:

[15]:
```
x = 0.0000001
y = 0.0000002

a = 123456789.01
b = 123456789.02

print('x = y:', isclose(x, y, abs_tol=0.0001, rel_tol=0.01))
print('a = b:', isclose(a, b, abs_tol=0.0001, rel_tol=0.01))
```

```
x = y: True
a = b: True
```

### 0.6.1 Coercing Floats to Integers

**Truncation**

[1]:
```
from math import trunc
```

[2]:
```
trunc(10.3), trunc(10.5), trunc(10.6)
```

[2]:
```
(10, 10, 10)
```

[3]:
```
trunc(-10.6), trunc(-10.5), trunc(-10.3)
```

[3]:
```
(-10, -10, -10)
```

The **int** constructor uses truncation when a float is passed in:

[4]:
```
int(10.3), int(10.5), int(10.6)
```

[4]: (10, 10, 10)

```
[5]: int(-10.5), int(-10.5), int(-10.4)
```

[5]: (-10, -10, -10)

**Floor**

```
[6]: from math import floor
```

```
[7]: floor(10.4), floor(10.5), floor(10.6)
```

[7]: (10, 10, 10)

```
[8]: floor(-10.4), floor(-10.5), floor(-10.6)
```

[8]: (-11, -11, -11)

**Ceiling**

```
[9]: from math import ceil
```

```
[10]: ceil(10.4), ceil(10.5), ceil(10.6)
```

[10]: (11, 11, 11)

```
[11]: ceil(-10.4), ceil(-10.5), ceil(-10.6)
```

[11]: (-10, -10, -10)

```
[ ]:
```

### 0.6.2 Rounding

```
[4]: help(round)
```

```
Help on built-in function round in module builtins:

round(…)
    round(number[, ndigits]) -> number

    Round a number to a given precision in decimal digits (default 0 digits).
    This returns an int when called with one argument, otherwise the
    same type as the number. ndigits may be negative.
```

**n = 0**

[5]:
```
a = round(1.5)
a, type(a)
```

[5]:
```
(2, int)
```

[6]:
```
a = round(1.5, 0)
a, type(b)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-6-356ec769b541> in <module>()
      1 a = round(1.5, 0)
----> 2 a, type(b)

NameError: name 'b' is not defined
```

**n > 0**

[ ]:
```
round(1.8888, 3), round(1.8888, 2), round(1.8888, 1), round(1.8888, 0)
```

**n < 0**

[ ]:
```
round(888.88, 1), round(888.88, 0), \
round(888.88, -1), round(888.88, -2), \
round(888.88, -3)
```

**Ties**

[ ]:
```
round(1.25, 1)
```

[ ]:
```
round(1.35, 1)
```

This is rounding to nearest, with ties to nearest number with even least significant digit, aka Banker's Rounding.

Works similarly with **n** negative.

[ ]:
```
round(15, -1)
```

[ ]:
```
round(25, -1)
```

**Rounding to closest, ties away from zero** This is traditionally the type of rounding taught in school, which is different from the Banker's Rounding implemented in Python (and in many other programming languages)

1.5 –> 2 2.5 –> 3

-1.5 –> -2 -2.5 –> -3

To do this type of rounding (to nearest 1) we can add (for positive numbers) or subtract (for negative numbers) 0.5 and then truncate the resulting number.

[ ]:
```
def _round(x):
    from math import copysign
    return int(x + 0.5 * copysign(1, x))
```

[ ]:
```
round(1.5), _round(1.5)
```

[ ]:
```
round(2.5), _round(2.5)
```

### 0.6.3 Decimals

[1]:
```
import decimal
```

[2]:
```
from decimal import Decimal
```

Decimals have context, that can be used to specify rounding and precision (amongst other things)

Contexts can be local (temporary contexts) or global (default)

**Global Context**

[3]:
```
g_ctx  = decimal.getcontext()
```

[4]:
```
g_ctx.prec
```

[4]:
```
28
```

[5]:
```
g_ctx.rounding
```

[5]:
```
'ROUND_HALF_EVEN'
```

We can change settings in the global context:

[6]:
```
g_ctx.prec = 6
```

[7]:
```
g_ctx.rounding = decimal.ROUND_HALF_UP
```

And if we read this back directly from the global context:

[8]:
```
decimal.getcontext().prec
```

[8]:
```
6
```

[9]:
```
decimal.getcontext().rounding
```

[9]:
```
'ROUND_HALF_UP'
```

we see that the global context was indeed changed.

**Local Context** The `localcontext()` function will return a context manager that we can use with a `with` statement:

[10]:
```
with decimal.localcontext() as ctx:
    print(ctx.prec)
    print(ctx.rounding)
```

```
6
ROUND_HALF_UP
```

Since no argument was specified in the `localcontext()` call, it provides us a context manager that uses a copy of the global context.

Modifying the local context has no effect on the global context

[11]:
```
with decimal.localcontext() as ctx:
    ctx.prec = 10
    print('local prec = {0}, global prec = {1}'.format(ctx.prec, g_ctx.prec))
```

```
local prec = 10, global prec = 6
```

**Rounding**

[12]:
```
decimal.getcontext().rounding
```

[12]:
```
'ROUND_HALF_UP'
```

The rounding mechanism is ROUND_HALF_UP because we set the global context to that earlier in this notebook. Note that normally the default is ROUND_HALF_EVEN.

So we first reset our global context rounding to that:

[13]:
```
decimal.getcontext().rounding = decimal.ROUND_HALF_EVEN
```

[14]:
```
x = Decimal('1.25')
y = Decimal('1.35')
print(round(x, 1))
print(round(y, 1))
```

```
1.2
1.4
```

Let's change the rounding mechanism in the global context to ROUND_HALF_UP:

[15]:
```
decimal.getcontext().rounding = decimal.ROUND_HALF_UP
```

[16]:
```
x = Decimal('1.25')
y = Decimal('1.35')
print(round(x, 1))
print(round(y, 1))
```

```
1.3
1.4
```

As you may have realized, changing the global context is a pain if you need to constantly switch between different precisions and rounding algorithms. Also, it could introduce bugs if you forget that you changed the global context somewhere further up in your module.

For this reason, it is usually better to use a local context manager instead:

First we reset our global context rounding to the default:

[17]:
```
decimal.getcontext().rounding = decimal.ROUND_HALF_EVEN
```

[18]:
```
x = Decimal('1.25')
y = Decimal('1.35')
print(round(x, 1), round(y, 1))
with decimal.localcontext() as ctx:
    ctx.rounding = decimal.ROUND_HALF_UP
    print(round(x, 1), round(y, 1))
print(round(x, 1), round(y, 1))
```

```
1.2 1.4
1.3 1.4
1.2 1.4
```

### 0.6.4 Decimals: Constructors and Contexts

The **Decimal** constructor can handle a variety of data types

[1]:
```
import decimal
from decimal import Decimal
```

**Integers**

[2]:
```
Decimal(10)
```

[2]:
```
Decimal('10')
```

[3]:
```
Decimal(-10)
```

[3]:
```
Decimal('-10')
```

**Strings**

[4]:
```
Decimal('0.1')
```

[4]:
```
Decimal('0.1')
```

[5]:
```
Decimal('-3.1415')
```

[5]:
```
Decimal('-3.1415')
```

**Tuples**

[6]:
```
Decimal ((0, (3,1,4,1,5), -4))
```

[6]:
```
Decimal('3.1415')
```

[7]:
```
Decimal((1, (1,2,3,4), -3))
```

[7]:
```
Decimal('-1.234')
```

[8]:
```
Decimal((0, (1,2,3), 3))
```

[8]:
```
Decimal('1.23E+5')
```

**But don't use Floats**

[9]:
```
format(0.1, '.25f')
```

[9]:
```
'0.1000000000000000055511151'
```

[10]:
```
Decimal(0.1)
```

[10]:
```
Decimal('0.1000000000000000055511151231257827021181583404541015625')
```

As you can see, since we passed an approximate binary float to the Decimal constructor it did it's best to represent that binary float **exactly**!!

So, instead, use strings or tuples in the Decimal constructor.

**Context Precision and the Constructor** The context precision does nto affect the precision used when creating a Decimal object - those are independent of each other.

Let's set our global (default) context to a precision of 2

[11]:
```
decimal.getcontext().prec = 2
```

Now we can create decimal numbers of higher precision than that:

[12]:
```
a = Decimal('0.12345')
b = Decimal('0.12345')
```

[13]:
```
a
```

[13]:
```
Decimal('0.12345')
```

[14]:
```
b
```

[14]:
```
Decimal('0.12345')
```

But when we add those two numbers up, the context precision will matter:

[15]:
```
a+b
```

[15]:
```
Decimal('0.25')
```

As you can see, we ended up with a sum that was rounded to 2 digits after the decimal point (precision = 2)

**Local and Global Contexts are Independent**

[16]:
```
decimal.getcontext().prec = 6
```

[17]:
```
decimal.getcontext().rounding
```

[17]:
```
'ROUND_HALF_EVEN'
```

[18]:
```
a = Decimal('0.12345')
b = Decimal('0.12345')
print(a + b)
with decimal.localcontext() as ctx:
    ctx.prec = 2
    c = a + b
    print('c within local context: {0}'.format(c))
print('c within global context: {0}'.format(c))
```

```
0.24690
c within local context: 0.25
c within global context: 0.25
```

Since **c** was created within the local context by adding **a** and **b**, and the local context had a precision of 2, **c** was rounded to 2 digits after the decimal point.

Once the local context is destroyed (after the **with** block), the variable **c** still exists, and its precision is **still** just 2 - it doesn't magically suddenly get the global context's precision of 6.

### 0.6.5 Decimals - Math Operations

**Div and Mod** The // and % operators (and consequently, the divmod() function) behave differently for integers and Decimals.

This is because integer division for Decimals is performed differently, and results in a truncated division, whereas integers use a floored division.

These differences are only when negative numbers are involved. If all numbers involved are positive, then integer and Decimal div and mod operations are equal.

But in both cases the // and % operators satisfy the equation:

```
n = d * (n // d) + (n % d)
```

[1]:
```
import decimal
from decimal import Decimal
```

[2]:
```
x = 10
y = 3
print(x//y, x%y)
print(divmod(x, y))
print( x == y * (x//y) + x % y)
```

```
3 1
(3, 1)
True
```

[3]:
```
a = Decimal('10')
b = Decimal('3')
print(a//b, a%b)
print(divmod(a, b))
print( a == b * (a//b) + a % b)
```

```
3 1
(Decimal('3'), Decimal('1'))
True
```

As we can see, the // and % operators had the same result when both numbers were positive.

[4]:
```
x = -10
y = 3
print(x//y, x%y)
print(divmod(x, y))
print( x == y * (x//y) + x % y)
```

```
-4 2
(-4, 2)
True
```

[5]:
```
a = Decimal('-10')
b = Decimal('3')
print(a//b, a%b)
print(divmod(a, b))
print( a == b * (a//b) + a % b)
```

```
-3 -1
(Decimal('-3'), Decimal('-1'))
True
```

On the other hand, we see that in this case the // and % operators did not result in the same values, although the equation was satisfied in both instances.

**Other Mathematical Functions** The Decimal class implements a variety of mathematical functions.

[6]:
```
a = Decimal('1.5')
print(a.log10())  # base 10 logarithm
print(a.ln())     # natural logarithm (base e)
print(a.exp())    # e**a
print(a.sqrt())   # square root
```

```
0.17609125905568124208128900 85
0.40546510810816438197801311 55
4.48168907033806482260205546 0
1.22474487139158904909864203 7
```

Although you can use the math function of the math module, be aware that the math module functions will cast the Decimal numbers to floats when it performs the various operations. So, if the precision is important (which it probably is if you decided to use Decimal numbers in the first place), choose the math functions of the Decimal class over those of the math module.

[7]:
```
x = 2
x_dec = Decimal(2)
```

[8]:
```
import math
```

[9]:
```
root_float = math.sqrt(x)
root_mixed = math.sqrt(x_dec)
root_dec = x_dec.sqrt()
```

[10]:
```
print(format(root_float, '1.27f'))
print(format(root_mixed, '1.27f'))
print(root_dec)
```

```
1.41421356237309514547462185 9
1.41421356237309514547462185 9
1.41421356237309504880168872 4
```

[11]:
```
print(format(root_float * root_float, '1.27f'))
print(format(root_mixed * root_mixed, '1.27f'))
print(root_dec * root_dec)
```

```
2.00000000000000044408920985 0
2.00000000000000044408920985 0
1.99999999999999999999999999
```

[12]:
```
x = 0.01
x_dec = Decimal('0.01')

root_float = math.sqrt(x)
root_mixed = math.sqrt(x_dec)
```

```
root_dec = x_dec.sqrt()

print(format(root_float, '1.27f'))
print(format(root_mixed, '1.27f'))
print(root_dec)
```

```
0.100000000000000005551115123
0.100000000000000005551115123
0.1
```

[13]:
```
print(format(root_float * root_float, '1.27f'))
print(format(root_mixed * root_mixed, '1.27f'))
print(root_dec * root_dec)
```

```
0.010000000000000001942890293
0.010000000000000001942890293
0.01
```

### 0.6.6 Decimals: Performance Considerations

**Memory Footprint** Decimals take up a lot more memory than floats.

[1]:
```
import sys
from decimal import Decimal
```

[2]:
```
a = 3.1415
b = Decimal('3.1415')
```

[3]:
```
sys.getsizeof(a)
```

[3]: 24

24 bytes are used to store the float 3.1415

[4]:
```
sys.getsizeof(b)
```

[4]: 104

104 bytes are used to store the Decimal 3.1415

**Computational Performance** Decimal arithmetic is also much slower than float arithmetic (on a CPU, and even more so if using a GPU)

We can do some rough timings to illustrate this.

First we look at the performance difference creating floats vs decimals:

[5]:
```
import time
from decimal import Decimal
```

```
def run_float(n=1):
    for i in range(n):
        a = 3.1415

def run_decimal(n=1):
    for i in range(n):
        a = Decimal('3.1415')
```

Timing float and Decimal operations:

[6]:
```
n = 10000000
```

[7]:
```
start = time.perf_counter()
run_float(n)
end = time.perf_counter()
print('float: ', end-start)

start = time.perf_counter()
run_decimal(n)
end = time.perf_counter()
print('decimal: ', end-start)
```

```
float:  0.21406484986433047
decimal:  2.1353148079910156
```

We make a slight variant here to see how addition compares between the two types:

[11]:
```
def run_float(n=1):
    a = 3.1415
    for i in range(n):
        a + a

def run_decimal(n=1):
    a = Decimal('3.1415')
    for i in range(n):
        a + a

start = time.perf_counter()
run_float(n)
end = time.perf_counter()
print('float: ', end-start)

start = time.perf_counter()
run_decimal(n)
end = time.perf_counter()
print('decimal: ', end-start)
```

```
float:  0.1875864573764936
decimal:  0.3911394302055555
```

How about square roots:

(We drop the n count a bit)

[10]:
```
n = 5000000

import math

def run_float(n=1):
    a = 3.1415
    for i in range(n):
        math.sqrt(a)

def run_decimal(n=1):
    a = Decimal('3.1415')
    for i in range(n):
        a.sqrt()

start = time.perf_counter()
run_float(n)
end = time.perf_counter()
print('float: ', end-start)

start = time.perf_counter()
run_decimal(n)
end = time.perf_counter()
print('decimal: ', end-start)
```

```
float:  0.673833850211659
decimal:  14.73112183459776
```

### 0.6.7 Complex Numbers

Python's built-in class provides support for complex numbers.

Complex numbers are defined in rectangular coordinates (real and imaginary parts) using either the constructor or a literal expression.

The complex number 1 + 2j can be defined in either of these ways:

[1]:
```
a = complex(1, 2)
b = 1 + 2j
```

[2]:
```
a == b
```

[2]: True

Note that the real and imaginary parts are defined as floats, and can be retrieved as follows:

[3]:
```
a.real, type(a.real)
```

[3]:
```
(1.0, float)
```

[4]:
```
a.imag, type(a.imag)
```

[4]:
```
(2.0, float)
```

The complex conjugate can be calculated as follows:

[5]:
```
a.conjugate()
```

[5]:
```
(1-2j)
```

The standard arithmetic operatots are polymorphic and defined for complex numbers

[6]:
```
a = 1 + 2j
b = 3 - 4j
c = 5j
d = 10
```

[7]:
```
a + b
```

[7]:
```
(4-2j)
```

[8]:
```
b * c
```

[8]:
```
(20+15j)
```

[9]:
```
c / d
```

[9]:
```
0.5j
```

[10]:
```
d - a
```

[10]:
```
(9-2j)
```

The // and % operators, although also polymorphic, are not defined for complex numbers:

[11]:
```
a // b
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-11-d3400ddfd09e> in <module>()
----> 1 a // b

TypeError: can't take floor of complex number.
```

[12]:
```
a % b
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-12-209a85c6b5ad> in <module>()
----> 1 a % b

TypeError: can't mod complex numbers.
```

The == and != operators support complex numbers - but since the real and imaginary parts of complex numbers are floats, the same problems comparing floats using == and != also apply to complex numbers.

[14]:
```
a = 0.1j
a + a + a == 0.3j
```

[14]:
```
False
```

In addition, the standard comparison operators (<, <=, >, >=) are not defined for complex numbers.

[15]:
```
a = 1 + 1j
b = 100 + 100j
a < b
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-15-a9748a0ff9df> in <module>()
      1 a = 1 + 1j
      2 b = 100 + 100j
----> 3 a < b

TypeError: '<' not supported between instances of 'complex' and 'complex'
```

**Math Functions** The **cmath** module provides complex alternatives to the standard **math** functions.

In addition, the **cmath** module provides the complex implementation of the **isclose()** method available for floats.

[16]:
```
import cmath

a = 1 + 5j
print(cmath.sqrt(a))
```

```
(1.7462845577958914+1.4316108957382214j)
```

The standard **math** module functions will not work with complex numbers:

[18]:
```
import math
print(math.sqrt(a))
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-18-77a8fdd31911> in <module>()
      1 import math
----> 2 print(math.sqrt(a))

TypeError: can't convert complex to float
```

**Polar / Rectangular Conversions** The **cmath.phase()** function can be used to return the phase (or argument) of any complex number.

The standard **abs()** function supports complex numbers and will return the magnitude (euclidean norm) of the complex number.

[19]:
```
a = 1 + 1j
```

[22]:
```
r = abs(a)
phi = cmath.phase(a)
print('{0} = ({1},{2})'.format(a, r, phi))
```

```
(1+1j) = (1.4142135623730951,0.7853981633974483)
```

Complex numbers in polar coordinates can be converted to rectangular coordinates using the **math.rect()** function:

[26]:
```
r = math.sqrt(2)
phi = cmath.pi/4
print(cmath.rect(r, phi))
```

```
(1.0000000000000002+1.0000000000000002j)
```

**Euler's Identity and the isclose() function** $ei + 1 = 0$

[28]:
```
RHS = cmath.exp(cmath.pi * 1j) + 1
print(RHS)
```

```
1.2246467991473532e-16j
```

Which, because of limited precision is not quite zero.

However, the result is very close to zero.

We can use the **isclose()** method of the **cmath** module, which behaves similarly to the **math.isclose()** method. Since we are testing for closeness of two numbers close to zero, we need to make sure an absolute tolerance is also specified:

[29]:
```
cmath.isclose(RHS, 0, abs_tol=0.00001)
```

[29]:
```
True
```

If we had not specified an absolute tolerance:

[30]:
```
cmath.isclose(RHS, 0)
```

[30]:
```
False
```

### 0.6.8 Booleans

The **bool** class is used to represent boolean values.

The **bool** class inherits from the **int** class.

[1]:
```
issubclass(bool, int)
```

[1]:
```
True
```

Two built-in constants, **True** and **False** are singleton instances of the bool class with underlying int values of 1 and 0 respectively.

[7]:
```
type(True), id(True), int(True)
```

[7]:
```
(bool, 1658060976, 1)
```

[8]:
```
type(False), id(False), int(False)
```

[8]:
```
(bool, 1658061008, 0)
```

These two values are instances of the **bool** class, and by inheritance are also **int** objects.

[5]:
```
isinstance(True, bool)
```

[5]:
```
True
```

[6]:
```
isinstance(True, int)
```

[6]:
```
True
```

Since **True** and **False** are singletons, we can use either the **is** operator, or the == operator to compare them to **any** boolean expression.

[9]:
```
id(True), id(1 < 2)
```

[9]:
```
(1658060976, 1658060976)
```

[10]:
```
id(False), id(1 == 3)
```

[10]:
```
(1658061008, 1658061008)
```

[12]:
```
(1 < 2) is True, (1 < 2) == True
```

[12]:
```
(True, True)
```

[13]:
```
(1 == 2) is False, (1 == 2) == False
```

[13]:
```
(True, True)
```

Be careful with that last comparison, the parentheses are necessary!

[15]:
```
1 == 2 == False
```

[15]:
```
False
```

[16]:
```
(1 == 2) == False
```

[16]:
```
True
```

We'll look into this in detail later, but, for now, this happens because a chained comparison such as **a == b == c** is actually evaluated as **a == b and b == c**

So **1 == 2 == False** is the same as **1 == 2 and 2 == False**

[17]:
```
1 == 2, 2 == False, 1==2 and 2==False
```

[17]:
```
(False, False, False)
```

But,

[18]:
```
(1 == 2)
```

[18]:
```
False
```

So **(1 == 2) == False** evaluates to True

But since **False** is also **0**, we get the following:

[36]:
```
(1 == 2) == 0
```

[36]:
```
True
```

The underlying integer values of True and False are:

[19]:
```
int(True), int(False)
```

[19]:
```
(1, 0)
```

So, using an equality comparison:

[20]:
```
1 == True, 0 == False
```

[20]:
```
(True, True)
```

But, from an object perspective 1 and True are not the same (similarly with 0 and False)

[21]:
```
1 == True, 1 is True
```

[21]:
```
(True, False)
```

[23]:
```
0 == False, 0 is False
```

[23]:
```
(True, False)
```

Any integer can be cast to a boolean, and follows the rule:

bool(x) = True for any x except for zero which returns False

[24]:
```
bool(0)
```

[24]:
```
False
```

[25]:
```
bool(1), bool(100), bool(-1)
```

[25]:
```
(True, True, True)
```

Since booleans are subclassed from integers, they can behave like integers, and because of polymorphism all the standard integer operators, properties and methods apply

[26]:
```
True > False
```

[26]:
```
True
```

[27]:
```
True + 2
```

[27]:
```
3
```

[29]:
```
False // 2
```

[29]:
```
0
```

[33]:
```
True + True + True
```

[33]:
```
3
```

[32]:
```
(True + True + True) % 2
```

[32]:
```
1
```

[34]:
```
-True
```

[34]:
```
-1
```

[35]:
```
100 * False
```

[35]:
```
0
```

I certainly **do not** recommend you write code like that shown above, but be aware that it does work.

### 0.6.9 Booleans: Truth Values

All objects in Python have an associated **truth value**, or **truthyness**

We saw in a previous lecture that integers have an inherent truth value:

[2]:
```
bool(0)
```

[2]:
```
False
```

[3]:
```
bool(1), bool(-1), bool(100)
```

[3]:
```
(True, True, True)
```

This truthyness has nothing to do with the fact that **bool** is a subclass of **int**.

Instead, it has to do with the fact that the **int** class implements a `__bool__()` method:

[4]:
```
help(bool)
```

```
Help on class bool in module builtins:

class bool(int)
 |  bool(x) -> bool
 |
 |  Returns True when the argument x is true, False otherwise.
 |  The builtins True and False are the only two instances of the class bool.
 |  The class bool is a subclass of the class int, and cannot be subclassed.
 |
 |  Method resolution order:
 |      bool
 |      int
 |      object
 |
 |  Methods defined here:
 |
 |  __and__(self, value, /)
 |      Return self&value.
 |
```

```
|  __new__(*args, **kwargs) from builtins.type
|      Create and return a new object.  See help(type) for accurate signature.
|
|  __or__(self, value, /)
|      Return self|value.
|
|  __rand__(self, value, /)
|      Return value&self.
|
|  __repr__(self, /)
|      Return repr(self).
|
|  __ror__(self, value, /)
|      Return value|self.
|
|  __rxor__(self, value, /)
|      Return value^self.
|
|  __str__(self, /)
|      Return str(self).
|
|  __xor__(self, value, /)
|      Return self^value.
|
|  ----------------------------------------------------------------------
|  Methods inherited from int:
|
|  __abs__(self, /)
|      abs(self)
|
|  __add__(self, value, /)
|      Return self+value.
|
|  __bool__(self, /)
|      self != 0
|
|  __ceil__(…)
|      Ceiling of an Integral returns itself.
|
|  __divmod__(self, value, /)
|      Return divmod(self, value).
|
|  __eq__(self, value, /)
|      Return self==value.
|
|  __float__(self, /)
|      float(self)
|
```

```
 |  __floor__(…)
 |      Flooring an Integral returns itself.
 |
 |  __floordiv__(self, value, /)
 |      Return self//value.
 |
 |  __format__(…)
 |      default object formatter
 |
 |  __ge__(self, value, /)
 |      Return self>=value.
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __getnewargs__(…)
 |
 |  __gt__(self, value, /)
 |      Return self>value.
 |
 |  __hash__(self, /)
 |      Return hash(self).
 |
 |  __index__(self, /)
 |      Return self converted to an integer, if self is suitable for use as an
index into a list.
 |
 |  __int__(self, /)
 |      int(self)
 |
 |  __invert__(self, /)
 |      ~self
 |
 |  __le__(self, value, /)
 |      Return self<=value.
 |
 |  __lshift__(self, value, /)
 |      Return self<<value.
 |
 |  __lt__(self, value, /)
 |      Return self<value.
 |
 |  __mod__(self, value, /)
 |      Return self%value.
 |
 |  __mul__(self, value, /)
 |      Return self*value.
 |
```

```
|  __ne__(self, value, /)
|      Return self!=value.
|
|  __neg__(self, /)
|      -self
|
|  __pos__(self, /)
|      +self
|
|  __pow__(self, value, mod=None, /)
|      Return pow(self, value, mod).
|
|  __radd__(self, value, /)
|      Return value+self.
|
|  __rdivmod__(self, value, /)
|      Return divmod(value, self).
|
|  __rfloordiv__(self, value, /)
|      Return value//self.
|
|  __rlshift__(self, value, /)
|      Return value<<self.
|
|  __rmod__(self, value, /)
|      Return value%self.
|
|  __rmul__(self, value, /)
|      Return value*self.
|
|  __round__(…)
|      Rounding an Integral returns itself.
|      Rounding with an ndigits argument also returns an integer.
|
|  __rpow__(self, value, mod=None, /)
|      Return pow(value, self, mod).
|
|  __rrshift__(self, value, /)
|      Return value>>self.
|
|  __rshift__(self, value, /)
|      Return self>>value.
|
|  __rsub__(self, value, /)
|      Return value-self.
|
|  __rtruediv__(self, value, /)
|      Return value/self.
```

```
 |
 |  __sizeof__(…)
 |      Returns size in memory, in bytes
 |
 |  __sub__(self, value, /)
 |      Return self-value.
 |
 |  __truediv__(self, value, /)
 |      Return self/value.
 |
 |  __trunc__(…)
 |      Truncating an Integral returns itself.
 |
 |  bit_length(…)
 |      int.bit_length() -> int
 |
 |      Number of bits necessary to represent self in binary.
 |      >>> bin(37)
 |      '0b100101'
 |      >>> (37).bit_length()
 |      6
 |
 |  conjugate(…)
 |      Returns self, the complex conjugate of any int.
 |
 |  from_bytes(…) from builtins.type
 |      int.from_bytes(bytes, byteorder, *, signed=False) -> int
 |
 |      Return the integer represented by the given array of bytes.
 |
 |      The bytes argument must be a bytes-like object (e.g. bytes or
bytearray).
 |
 |      The byteorder argument determines the byte order used to represent the
 |      integer.  If byteorder is 'big', the most significant byte is at the
 |      beginning of the byte array.  If byteorder is 'little', the most
 |      significant byte is at the end of the byte array.  To request the native
 |      byte order of the host system, use `sys.byteorder' as the byte order
value.
 |
 |      The signed keyword-only argument indicates whether two's complement is
 |      used to represent the integer.
 |
 |  to_bytes(…)
 |      int.to_bytes(length, byteorder, *, signed=False) -> bytes
 |
 |      Return an array of bytes representing an integer.
 |
```

```
 |      The integer is represented using length bytes.  An OverflowError is
 |      raised if the integer is not representable with the given number of
 |      bytes.
 |
 |      The byteorder argument determines the byte order used to represent the
 |      integer.  If byteorder is 'big', the most significant byte is at the
 |      beginning of the byte array.  If byteorder is 'little', the most
 |      significant byte is at the end of the byte array.  To request the native
 |      byte order of the host system, use `sys.byteorder' as the byte order
value.
 |
 |      The signed keyword-only argument determines whether two's complement is
 |      used to represent the integer.  If signed is False and a negative
integer
 |      is given, an OverflowError is raised.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from int:
 |
 |  denominator
 |      the denominator of a rational number in lowest terms
 |
 |  imag
 |      the imaginary part of a complex number
 |
 |  numerator
 |      the numerator of a rational number in lowest terms
 |
 |  real
 |      the real part of a complex number
```

If you scroll down in the documentation you shoudl reach a section that looks like this:

```
|  __bool__(self, /) |      self != 0
```

So, when we write:

[5]:
```
bool(100)
```

[5]:
```
True
```

Python is actually calling 100.___bool___() and returning that:

[7]:
```
(100).__bool__()
```

[7]:
```
True
```

[8]:
```
(0).__bool__()
```

[8]:
```
False
```

Most objects will implement either the `__bool__()` or `__len__()` methods. If they don't, then their associated value will be **True** always.

**Numeric Types** Any non-zero numeric value is truthy. Any zero numeric value is falsy:

[9]:
```
from fractions import Fraction
from decimal import Decimal
bool(10), bool(1.5), bool(Fraction(3, 4)), bool(Decimal('10.5'))
```

[9]:
```
(True, True, True, True)
```

[27]:
```
bool(0), bool(0.0), bool(Fraction(0,1)), bool(Decimal('0')), bool(0j)
```

[27]:
```
(False, False, False, False, False)
```

**Sequence Types** An empty sequence type object is Falsy, a non-empty one is truthy:

[28]:
```
bool([1, 2, 3]), bool((1, 2, 3)), bool('abc'), bool(1j)
```

[28]:
```
(True, True, True, True)
```

[14]:
```
bool([]), bool(()), bool('')
```

[14]:
```
(False, False, False)
```

**Mapping Types** Similarly, an empty mapping type will be falsy, a non-empty one truthy:

[16]:
```
bool({'a': 1}), bool({1, 2, 3})
```

[16]:
```
(True, True)
```

[17]:
```
bool({}), bool(set())
```

[17]:
```
(False, False)
```

**The None Object** The singleton **None** object is always falsy:

[18]:
```
bool(None)
```

[18]:
```
False
```

**One Application of Truth Values** Any conditional expression which involves objects other than **bool** types, will use the associated truth value as the result of the conditional expression.

[37]:
```
a = [1, 2, 3]
if a:
    print(a[0])
else:
    print('a is None, or a is empty')
```

```
1
```

[38]:
```
a = []
if a:
    print(a[0])
else:
    print('a is None, or a is empty')
```

```
a is None, or a is empty
```

[39]:
```
a = 'abc'
if a:
    print(a[0])
else:
    print('a is None, or a is empty')
```

```
a
```

[40]:
```
a = ''
if a:
    print(a[0])
else:
    print('a is None, or a is empty')
```

```
a is None, or a is empty
```

We could write this using a more lengthy expression:

[41]:
```
a = 'abc'
if a is not None and len(a) > 0:
    print(a[0])
else:
    print('a is None, or a is empty')
```

```
a
```

Doing the following would break our code in some instances:

[43]:
```
a = 'abc'
if a is not None:
    print(a[0])
```

```
a
```

works, but:

[44]:
```
a = ''
if a is not None:
    print(a[0])
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-44-47991d9c7397> in <module>()
      1 a = ''
      2 if a is not None:
----> 3     print(a[0])

IndexError: string index out of range
```

or even:

[45]:
```
a = None
if len(a) > 0:
    print(a[0])
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-45-92ec20435e20> in <module>()
      1 a = None
----> 2 if len(a) > 0:
      3     print(a[0])

TypeError: object of type 'NoneType' has no len()
```

To be torough we would need to write:

[46]:
```
a = None
if a is not None and len(a) > 0:
    print(a[0])
```

Also, the order of the boolean expressions matter here!

We'll discuss this and short-circuit evaluations in an upcoming video.

For example:

[47]:
```
a = None
if len(a) > 0 and a is not None:
    print(a[0])
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
```

```
<ipython-input-47-0842480c6625> in <module>()
      1 a = None
----> 2 if len(a) > 0 and a is not None:
      3     print(a[0])

TypeError: object of type 'NoneType' has no len()
```

### 0.6.10 Booleans: Precedence and Short-Circuiting

[1]:
```
True or True and False
```

[1]: True

this is equivalent, because of `and` having higer precedence than `or`, to:

[3]:
```
True or (True and False)
```

[3]: True

This is not the same as:

[2]:
```
(True or True) and False
```

[2]: False

#### Short-Circuiting

[13]:
```
a = 10
b = 2

if a/b > 2:
    print('a is at least double b')
```

```
a is at least double b
```

[12]:
```
a = 10
b = 0

if a/b > 2:
    print('a is at least double b')
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-12-98ac73a1accd> in <module>()
      2 b = 0
      3
----> 4 if a/b > 2:
      5     print('a is at least double b')
```

```
ZeroDivisionError: division by zero
```

[11]:
```
a = 10
b = 0

if b and a/b > 2:
    print('a is at least double b')
```

Can also be useful to deal with null or empty strings in a database:

[14]:
```
import string
```

[15]:
```
help(string)
```

```
Help on module string:

NAME
    string - A collection of string constants.

DESCRIPTION
    Public module variables:

    whitespace -- a string containing all ASCII whitespace
    ascii_lowercase -- a string containing all ASCII lowercase letters
    ascii_uppercase -- a string containing all ASCII uppercase letters
    ascii_letters -- a string containing all ASCII letters
    digits -- a string containing all ASCII decimal digits
    hexdigits -- a string containing all ASCII hexadecimal digits
    octdigits -- a string containing all ASCII octal digits
    punctuation -- a string containing all ASCII punctuation characters
    printable -- a string containing all ASCII characters considered printable

CLASSES
    builtins.object
        Formatter
        Template

    class Formatter(builtins.object)
     |  Methods defined here:
     |
     |  check_unused_args(self, used_args, args, kwargs)
     |
     |  convert_field(self, value, conversion)
     |
     |  format(*args, **kwargs)
     |
```

```
 |  format_field(self, value, format_spec)
 |
 |  get_field(self, field_name, args, kwargs)
 |      # given a field_name, find the object it references.
 |      #  field_name:   the field being looked up, e.g. "0.name"
 |      #                 or "lookup[3]"
 |      #  used_args:    a set of which args have been used
 |      #  args, kwargs: as passed in to vformat
 |
 |  get_value(self, key, args, kwargs)
 |
 |  parse(self, format_string)
 |      # returns an iterable that contains tuples of the form:
 |      # (literal_text, field_name, format_spec, conversion)
 |      # literal_text can be zero length
 |      # field_name can be None, in which case there's no
 |      #  object to format and output
 |      # if field_name is not None, it is looked up, formatted
 |      #  with format_spec and conversion and then used
 |
 |  vformat(self, format_string, args, kwargs)
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)

class Template(builtins.object)
 |  A string class for supporting $-substitutions.
 |
 |  Methods defined here:
 |
 |  __init__(self, template)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  safe_substitute(*args, **kws)
 |
 |  substitute(*args, **kws)
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
```

```
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  delimiter = '$'
 |
 |  flags = <RegexFlag.IGNORECASE: 2>
 |
 |  idpattern = '[_a-z][_a-z0-9]*'
 |
 |  pattern = re.compile('\n    \\$(?:\n
(?P<escaped>\\$)…_a-z][_a-…

FUNCTIONS
    capwords(s, sep=None)
        capwords(s [,sep]) -> string

        Split the argument into words using split, capitalize each
        word using capitalize, and join the capitalized words using
        join.  If the optional second argument sep is absent or None,
        runs of whitespace characters are replaced by a single space
        and leading and trailing whitespace are removed, otherwise
        sep is used to split and join the words.

DATA
    __all__ = ['ascii_letters', 'ascii_lowercase', 'ascii_uppercase', 'cap…
    ascii_letters = 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
    ascii_lowercase = 'abcdefghijklmnopqrstuvwxyz'
    ascii_uppercase = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
    digits = '0123456789'
    hexdigits = '0123456789abcdefABCDEF'
    octdigits = '01234567'
    printable = '0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTU…
    punctuation = '!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
    whitespace = ' \t\n\r\x0b\x0c'

FILE
    c:\users\fbapt\anaconda3\envs\deepdive\lib\string.py
```

[16]:
```
string.digits
```

[16]:
```
'0123456789'
```

[17]:
```
string.ascii_letters
```

[17]:
```
'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

[19]:
```
name = ''
if name[0] in string.digits:
    print('Name cannot start with a digit!')
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-19-b6d1fe6f1f39> in <module>()
      1 name = ''
----> 2 if name[0] in string.digits:
      3     print('Name cannot start with a digit!')

IndexError: string index out of range
```

[20]:
```
name = ''
if name and name[0] in string.digits:
    print('Name cannot start with a digit!')
```

[21]:
```
name = None
if name and name[0] in string.digits:
    print('Name cannot start with a digit!')
```

[22]:
```
name = 'Bob'
if name and name[0] in string.digits:
    print('Name cannot start with a digit!')
```

[23]:
```
name = '1Bob'
if name and name[0] in string.digits:
    print('Name cannot start with a digit!')
```

```
Name cannot start with a digit!
```

[ ]:
```

```

### 0.6.11 Booleans: Boolean Operators

The way the Boolean operators `and`, `or` actually work is a littel different in Python:

**or** `X or Y`: If X is falsy, returns Y, otherwise evaluates and returns X

[1]:
```
'' or 'abc'
```

[1]:
```
'abc'
```

[3]:
```
0 or 100
```

[3]:
```
100
```

[4]:
```
[] or [1, 2, 3]
```

[4]:
```
[1, 2, 3]
```

[5]:
```
[1, 2] or [1, 2, 3]
```

[5]:
```
[1, 2]
```

You should note that the truth value of `Y` is never even considered when evaluating the `or` result!

Only the left operand matters.

Of course, Y will be evaluated if it is being returned - but its truth value does not affect how the `or` is being calculated.

You probably will notice that this means `Y` is not evaluated if `X` is returned - short-circuiting!!!

We could (almost!) write the `or` operator ourselves in this way:

[15]:
```
def _or(x, y):
    if x:
        return x
    else:
        return y
```

[25]:
```
print(_or(0, 100) == (0 or 100))
print(_or(None, 'n/a') == (None or 'n/a'))
print(_or('abc', 'n/a') == ('abc' or 'n/a'))
```

```
True
True
True
```

Why did I say almost?

Unlike the `or` operator, our `_or` function will always evaluate x and y (they are passed as arguments) - so we do not have short-circuiting!

[31]:
```
1 or 1/0
```

[31]:
```
1
```

[32]:
```
_or(1, 1/0)
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-32-7b66dcdf3d9c> in <module>()
```

```
----> 1 _or(1, 1/0)

ZeroDivisionError: division by zero
```

**and** `X and Y`: If X is falsy, returns X, otherwise evaluates and returns Y

Once again, note that the truth value of Y is never considered when evaluating **and**, and that `Y` is only evaluated if it needs to be returned (short-circuiting)

[33]:
```
s1 = None
s2 = ''
s3 = 'abc'
```

[35]:
```
print(s1 and s1[0])
print(s2 and s2[0])
print(s3 and s3[0])
```

```
None

a
```

[51]:
```
print((s1 and s1[0]) or '')
print((s2 and s2[0]) or '')
print((s3 and s3[0]) or '')
```

```


a
```

This technique will also work to return any default value if **s** is an empty string or None:

[54]:
```
print((s1 and s1[0]) or 'n/a')
print((s2 and s2[0]) or 'n/a')
print((s3 and s3[0]) or 'n/a')
```

```
n/a
n/a
a
```

The `not` function

[1]:
```
not 'abc'
```

[1]: False

[2]:
```
not []
```

[2]: True

[4]:
```
bool(None)
```

[4]:
```
False
```

[5]:
```
not None
```

[5]:
```
True
```

### 0.6.12 Comparison Operators

**Identity and Membership Operators** The **is** and **is not** operators will work with any data type since they are comparing the memory addresses of the objects (which are integers)

[3]:
```
0.1 is (3+4j)
```

[3]:
```
False
```

[4]:
```
'a' is [1, 2, 3]
```

[4]:
```
False
```

The **in** and **not in** operators are used with iterables and test membership:

[5]:
```
1 in [1, 2, 3]
```

[5]:
```
True
```

[6]:
```
[1, 2] in [1, 2, 3]
```

[6]:
```
False
```

[7]:
```
[1, 2] in [[1,2], [2,3], 'abc']
```

[7]:
```
True
```

[8]:
```
'key1' in {'key1': 1, 'key2': 2}
```

[8]:
```
True
```

[9]:
```
1 in {'key1': 1, 'key2': 2}
```

[9]:
```
False
```

We'll come back to these operators in later sections on iterables and mappings.

**Equality Operators** The == and **!=** operators are value comparison operators.

They will work with mixed types that are comparable in some sense.

For example, you can compare Fraction and Decimal objects, but it would not make sense to compare string and integer objects.

[10]:
```
1 == '1'
```

[10]:
```
False
```

[11]:
```
from decimal import Decimal
from fractions import Fraction
```

[12]:
```
Decimal('0.1') == Fraction(1, 10)
```

[12]:
```
True
```

[13]:
```
1 == 1 + 0j
```

[13]:
```
True
```

[14]:
```
True == Fraction(2, 2)
```

[14]:
```
True
```

[15]:
```
False == 0j
```

[15]:
```
True
```

**Ordering Comparisons** Many, but not all data types have an ordering defined.

For example, complex numbers do not.

[16]:
```
1 + 1j < 2 + 2j
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-16-82ffa8a7b757> in <module>()
----> 1 1 + 1j < 2 + 2j

TypeError: '<' not supported between instances of 'complex' and 'complex'
```

Mixed type ordering comparisons is supported, but again, it needs to make sense:

[17]:
```
1 < 'a'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-17-ca85dbce74b5> in <module>()
----> 1 1 < 'a'
```

```
TypeError: '<' not supported between instances of 'int' and 'str'
```

[18]:
```
Decimal('0.1') < Fraction(1, 2)
```

[18]:
```
True
```

**Chained Comparisons** It is possible to chain comparisons.

For example, in **a < b < c**, Python simply **ands** the pairwise comparisons: **a < b and b < c**

[19]:
```
1 < 2 < 3
```

[19]:
```
True
```

[20]:
```
1 < 2 > -5 < 50 > 4
```

[20]:
```
True
```

[29]:
```
1 < 2 == Decimal('2.0')
```

[29]:
```
True
```

[28]:
```
import string
'A' < 'a' < 'z' > 'Z' in string.ascii_letters
```

[28]:
```
True
```