Chapter 4 - Numeric Types

- Every data takes the form of ***objects*** – either built-in or user defined
- objects are the basis of every Python program you will ever write.

Numeric Type Basics

- integers and floating points
- A complete inventory of Python's numeric toolbox includes:
  - Integer and floating-point objects
  - Complex number objects
  - Decimal: fixed-precision objects
  - Fraction: rational number objects
  - Sets: collections with numeric operations
  - Booleans: true and false
  - Built-in functions and modules: round, math, random, etc.
  - Expressions; unlimited integer precision; bitwise operations; hex, octal, and binary formats
  - Third-party extensions: vectors, libraries, visualization, plotting, etc.

Numeric Literals

Literal	Interpretation
`1234, -24, 0, 99999999999999`	Integers (unlimited size)
`1.23, 1., 3.14e-10, 4E210, 4.0e+210`	Floating-point numbers
`0o177, 0x9ff, 0b101010`	Octal, hex, and binary literals in 3.X
`0177, 0o177, 0x9ff, 0b101010`	Octal, octal, hex, and binary literals in 2.X
`3+4j, 3.0+4.0j, 3J`	Complex number literals
`set('spam'), {1, 2, 3, 4}`	Sets: 2.X and 3.X construction forms
`Decimal('1.0'), Fraction(1, 3)`	Decimal and fraction extension types
`bool(X), True, False`	Boolean type and constants

Integer and floating-point literals

- *Python 2.X: normal and long:*
  - there are two integer types, **normal** (often 32 bits) and **long** (unlimited precision), and an integer may end in an **l or L** to force it to become a long integer.
- *Python 3.X: a single type:*
  - the normal and long integer types have been **merged**.
  - there is only integer, which automatically supports the unlimited precision of Python 2.X's separate long integer type.

Built-in Numeric Tools

Expression operators
`+, -, *, /, >>, **, &`, etc.

Built-in mathematical functions
`pow, abs, round, int, hex, bin`, etc.

Utility modules
`random, math`, etc.

Python Expression Operators

- Python expression operators and precedence

Operators	Description
`yield x`	Generator function send protocol
`lambda args: expression`	Anonymous function generation
`x if y else z`	Ternary selection (x is evaluated only if y is true)
`x or y`	Logical OR (y is evaluated only if x is false)
`x and y`	Logical AND (y is evaluated only if x is true)
`not x`	Logical negation
`x in y, x not in y`	Membership (iterables, sets)
`x is y, x is not y`	Object identity tests
`x < y, x <= y, x > y, x >= y`	Magnitude comparison, set subset and superset;
`x == y, x != y`	Value equality operators
`x | y`	Bitwise OR, set union
`x ^ y`	Bitwise XOR, set symmetric difference
`x & y`	Bitwise AND, set intersection
`x << y, x >> y`	Shift x left or right by y bits
`x + y`	Addition, concatenation;
`x - y`	Subtraction, set difference
`x * y`	Multiplication, repetition;
`x % y`	Remainder, format;
`x / y, x // y`	Division: true and floor
`-x, +x`	Negation, identity
`~x`	Bitwise NOT (inversion)
`x ** y`	Power (exponentiation)
`x[i]`	Indexing (sequence, mapping, others)
`x[i:j:k]`	Slicing
`x(...)`	Call (function, method, class, other callable)
`x.attr`	Attribute reference
`(...)`	Tuple, expression, generator expression
`[...]`	List, list comprehension
`{...}`	Dictionary, set, set and dictionary comprehensions

Version differences (Python 2.X vs 3.X)

- In Python 2.X, value inequality can be written as either X != Y or X <> Y. In Python 3.X, the latter of these options is removed because it is redundant. In either version, best practice is to use X != Y for all value inequality tests.
- In Python 2.X, a backquotes expression `X` works the same as repr(X) and converts objects to display strings. Due to its obscurity, this expression is removed in Python 3.X; use the more readable str and repr built-in functions, described in “Numeric Display Formats.”
- The X // Y floor division expression always truncates fractional remainders in both Python 2.X and 3.X. The X / Y expression performs true division in 3.X (retaining remainders) and classic division in 2.X (truncating for integers).
- The syntax [...] is used for both list literals and list comprehension expressions. The latter of these performs an implied loop and collects expression results in a new list.
- The syntax (...) is used for tuples and expression grouping, as well as generator expressions—a form of list comprehension that produces results on demand, instead of building a result list.
- The syntax {...} is used for dictionary literals, and in Python 3.X and 2.7 for set literals and both dictionary and set comprehensions.

- The yield and ternary if/else selection expressions are available in Python 2.5 and later. The former returns send(...) arguments in generators; the latter is shorthand for a multiline if statement. yield requires parentheses if not alone on the right side of an assignment statement.
- Comparison operators may be chained: X < Y < Z produces the same result as X < Y and Y < Z.
- In recent Pythons, the slice expression X[I:J:K] is equivalent to indexing with a slice object: X[slice(I, J, K)].
- In Python 2.X, magnitude comparisons of mixed types are allowed, and convert numbers to a common type, and order other mixed types according to type names. In Python 3.X, nonnumeric mixed-type magnitude comparisons are not allowed and raise exceptions; this includes sorts by proxy.
- Magnitude comparisons for dictionaries are also no longer supported in Python 3.X (though equality tests are); comparing sorted(aDict.items()) is one possible replacement.

****

- Mixed operators follow operator precedence
- Parentheses group subexpressions
- Mixed types are converted up

  40 + 3.14 => 43.14
- You can force the issue by calling built-in functions to convert types manually:

  int(3.1415) => 3 # Truncates float to integer

  float(3) => 3.0 # Converts integer to float
- Python automatically converts up to the more complex type within an expression

Numbers in Action

- In Python:
  - Variables are created when they are first assigned values.
  - Variables are replaced with their values when used in expressions.
  - Variables must be assigned before they can be used in expressions.
  - Variables refer to objects and are never declared ahead of time.

```
>>> a + 1, a - 1              # Addition (3 + 1), subtraction (3 - 1)
(4, 2)
>>> b * 3, b / 2              # Multiplication (4 * 3), division (4 / 2)
(12, 2.0)
>>> a % 2, b ** 2             # Modulus (remainder), power (4 ** 2)
(1, 16)
>>> 2 + 4.0, 2.0 ** b         # Mixed-type conversions
(6.0, 16.0)
```

- **You don't need to predeclare variables in Python**, but they must have been assigned at least once before you can use them.

Numeric Display Formats

```
>>> num = 1 / 3.0
>>> num                       # Auto-echoes
0.3333333333333333
>>> print(num)                # Print explicitly
0.3333333333333333

>>> '%e' % num                # String formatting expression
'3.333333e-01'
>>> '%4.2f' % num             # Alternative floating-point format
'0.33'
>>> '{0:4.2f}'.format(num)    # String formatting method: Python 2.6, 3.0, and later
'0.33'
```

Comparisons: Normal and Chained

```
>>> 1 < 2                  # Less than
True
>>> 2.0 >= 1               # Greater than or equal: mixed-type 1 converted to 1.0
True
>>> 2.0 == 2.0             # Equal value
True
>>> 2.0 != 2.0             # Not equal value
False
```

- mixed types are allowed in numeric expressions (only).

1 == 2 < 3 # Same as: 1 == 2 and 2 < 3

- numeric comparisons are based on magnitudes, which are generally simple—though *floating-point* numbers may not always work as you'd expect

```
>>> 1.1 + 2.2 == 3.3              # Shouldn't this be True?...
False
>>> 1.1 + 2.2                     # Close to 3.3, but not exactly: limited precision
3.3000000000000003
>>> int(1.1 + 2.2) == int(3.3)    # OK if convert: see also round, floor, trunc ahead
True                              # Decimals and fractions (ahead) may help here too
```

Division: Classic, Floor, and True

- X / Y: *Classic* and *true* division.
  - In Python 2.X, this operator performs ***classic*** division, truncating results for integers, and keeping remainders (i.e., fractional parts) for floating-point numbers.
  - In Python 3.X, it performs ***true*** division, always keeping remainders in floating-point results, regardless of types.
  - Ref: https://www.informit.com/articles/article.aspx?p=1439189&seqNum=2
- X // Y: *Floor* division
  - This operator always truncates fractional remainders down to their floor, regardless of types. Its result type depends on the types of its operands.
- 2.x vs 3.x
  - *In 3.X*, the / now always performs *true* division, returning a float result that includes any remainder, **regardless of operand types**. The // performs *floor* division, which truncates the remainder and returns an integer for integer operands or a float if any operand is a float.
  - *In 2.X*, the / does *classic* division, performing truncating integer division if both operands are integers and float division (keeping remainders) otherwise. The // does *floor* division and works as it does in 3.X, performing truncating division for integers and floor division for floats.

```
C:\code> C:\Python33\python
>>>
>>> 10 / 4              # Differs in 3.X: keeps remainder
2.5
>>> 10 / 4.0            # Same in 3.X: keeps remainder
2.5
>>> 10 // 4             # Same in 3.X: truncates remainder
2
>>> 10 // 4.0           # Same in 3.X: truncates to floor
2.0

C:\code> C:\Python27\python
>>>
>>> 10 / 4              # This might break on porting to 3.X!
2
>>> 10 / 4.0
2.5
>>> 10 // 4             # Use this in 2.X if truncation needed
2
>>> 10 // 4.0
2.0
```

- the data type of the result for // is still dependent on the operand types in 3.X: if either is a float, the result is a float, otherwise, it is an integer.

Bitwise Operations

Code here

```
>>> X = 0b0001          # Binary literals
>>> X << 2              # Shift left
4
>>> bin(X << 2)         # Binary digits string
'0b100'

>>> bin(X | 0b010)      # Bitwise OR: either
'0b11'
>>> bin(X & 0b1)        # Bitwise AND: both
'0b1'
```

```
>>> X = 0xFF            # Hex literals
>>> bin(X)
'0b11111111'
>>> X ^ 0b10101010      # Bitwise XOR: either but not both
85
>>> bin(X ^ 0b10101010)
'0b1010101'

>>> int('01010101', 2)  # Digits=>number: string to int per base
85
>>> hex(85)             # Number=>digits: Hex digit string
'0x55'
```

Other Built-in Numeric Tools

- pow
- abs
- sum
- min/max
- round
- math module

Questions:

1. What is the value of the expression `2 * (3 + 4)` in Python?
2. What is the value of the expression `2 * 3 + 4` in Python?
3. What is the value of the expression `2 + 3 * 4` in Python?
4. What tools can you use to find a number's square root, as well as its square?
5. What is the type of the result of the expression `1 + 2.0 + 3`?
6. How can you truncate and round a floating-point number?
7. How can you convert an integer to a floating-point number?

8. How would you display an integer in octal, hexadecimal, or binary notation?
9. How might you convert an octal, hexadecimal, or binary string to a plain integer?