NUMBERS - A

Numeric Type Basics

In Python, numbers are not really a single object type, but a **category** of similar types.

Python supports different numerical types:

- **int** (signed integers): They are often called just integers or ints. They are positive or negative whole numbers with no decimal point. Integers in Python 3 are of unlimited size. Python 2 has two integer types - int and long. There is no 'long integer' in Python 3 anymore.
- **float** (floating point real values) : Also called floats, they represent real numbers and are written with a decimal point dividing the integer and the fractional parts. Floats may also be in scientific notation, with E or e indicating the power of 10 (2.5e2 = 2.5 x 102 = 250).
- **complex** (complex numbers) : are of the form a + bJ, where a and b are floats and J (or j) represents the square root of -1 (which is an imaginary number). The real part of the number is a, and the imaginary part is b.

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

Among its basic types, Python provides ***integers***, which are positive and negative whole numbers, and ***floating-point*** numbers, which are numbers with a fractional part (sometimes called "floats" for verbal economy).

Python also allows us to write integers using hexadecimal, octal, and binary literals; offers a complex number type; and allows integers to have unlimited *precision*—they can grow to have as many digits as your memory space allows.

The Table shows what Python's numeric types look like when written out in a program as literals or constructor function calls.

1234, −24, 0, 99999999999999	Signed 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
3+4j, 3.0+4.0j, 3J	Complex number literals
Decimal('1.0'), Fraction(1, 3)	Decimal and fraction extension types

Integer and floating-point literals
Integers are written as strings of decimal digits.
Floating-point numbers have a decimal point and/or an optional signed exponent introduced by an e or E and followed by an optional sign. If you write a number with a decimal point or exponent, Python makes it a floating-point object and uses floating-point (not integer) math when the object is used in an expression.

Hexadecimal, octal, and binary literals
Integers may be coded in decimal (base 10), hexadecimal (base 16), octal (base 8), or binary (base 2), the last three of which are common in some programming domains. Hexadecimals start with a leading 0x or 0X, followed by a string of hexadecimal digits (0–9 and A–F). Hex digits may be coded in lower- or uppercase. Octal literals start with a leading 0o or 0O (zero and lower- or uppercase letter *o*), followed by a

string of digits (0–7). Binary literals, new as of 2.6 and 3.0, begin with a leading 0b or 0B, followed by binary digits (0–1).

Note that all of these literals produce integer objects in program code; they are just alternative syntaxes for specifying values. The built-in calls hex(*I*), oct(*I*), and bin(*I*) convert an integer to its representation string in these three bases, and int(*str*, *base*) converts a runtime string to an integer per a given base.

Complex numbers

Python complex literals are written as *realpart*+*imaginarypart*, where the *imaginarypart* is terminated with a j or J. The *realpart* is technically optional, so the *imaginarypart* may appear on its own. Internally, complex numbers are implemented as pairs of floating-point numbers, but all numeric operations perform complex math when applied to complex numbers. Complex numbers may also be created with the complex(*real*, *imag*) built-in call.

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

PYTHON NUMERIC OPERATORS

Python's basic arithmetic operations are listed in the Table. In this table, the symbols A and B can be either numbers or expressions containing numbers and operators.

Operation	What It Does	Example	Value
A + B	Returns the sum of A and B	5+ 2	7
A – B	Returns the result of subtracting B from A	5– 2	3
A * B	Returns the product of A and B	5* 2	10
A / B	Returns the exact result of dividing A by B	5/2	2.5
A // B	Returns the integer quotient from dividing A by B	5 // 2	2
A % B	Returns the integer remainder from dividing A by B	5 % 2	1
A ** B	Returns AB	5 ** 2	25
– A	Returns the arithmetic negation of A	– (5 * 2)	–10

Note the following points about the arithmetic operations:

- The / operator produces the exact result of division, as a floating-point number.
- The // operator produces an integer quotient.
- When two integers are used with the other operators, the result is an integer.
- When at least one floating-point number is used with the other operators, the result is a floating-point number. Thus, 5 * 2 is 10, whereas 5 * 2.3 is 11.5.

As in mathematics, the arithmetic operators are governed by **precedence rules**. If operators of the same precedence appear in consecutive positions, they are evaluated

in left-to-right order. For example, the expression 3 + 4 – 2 + 5 is evaluated from left to right, producing 10.

When the operators do not have the same precedence, ** is evaluated first, then multiplication (*, /, //, or %), and finally addition (+ or –). For example, the expression 4 + 3 * 2 ** 3 first evaluates 2 ** 3, then 3 * 8, and finally 4 + 24, to produce 32.

You can use parentheses to override these rules. What are the results of evaluating these two expressions? Open a shell and check! For example, (3 + 4) * 2 begins evaluation with the addition, whereas 3 + 4 * 2 begins evaluation with the multiplication.

Negative numbers are represented with a minus sign. This sign is also used to negate more complex expressions, as in – (3 * 5). The precedence of the minus sign when used in this way is higher than that of any other arithmetic operator.

1. The / operator performs floating point division. It returns a **float** even if both the numerator and denominator are **ints**.

2. The // operator performs a quirky kind of integer division. When the result is positive, you can think of it as truncating (not rounding) to 0 decimal places, but be careful with that.

3. When integer-dividing negative numbers, the // operator rounds “up” to the nearest integer. Mathematically speaking, it’s rounding “down” since −6 is less than −5, but it could trip you up if you were expecting it to truncate to −5.

4. The // operator doesn’t always return an integer. If either the numerator or denominator is a float, it will still round to the nearest integer, but the actual return value will be a float.

5. The ** operator means “raised to the power of.”

6. The % operator gives the remainder after performing integer division. 11 divided by 2 is 5 with a remainder of 1, so the result here is 1.

NUMBER DISPLAY FORMATS

```
> > > print("%f" % (4023 - 22.4))
4000.600000
```

When you prepare strings to contain a number, you have a lot of flexibility.

```
> > > print("$%.02f" % 30.0)
$30.00
```

%f Format Specifier

Try out the following code and observe the different ways Python handles floating - point mathematics and then how you can manipulate the results with formatting:

```
> > > print("%f" % (5/3))
1.666667
> > > print("%.2f" % (5/3))
1.67
> > > print("%f" % (415 * 20.2))
8383.000000
> > > print("%0.f" % (415 * 20.2))
8383

> > > print("$%.03f" % 30.00123)
$30.001
> > > print("$%.03f" % 30.00163)
$30.002
> > > print("%.03f" % 30.1777)
30.178
> > > print("%.03f" % 30.1113)
30.111
```

As you can see, when you specify a format requiring more accuracy than you have asked Python to display, it will not just cut off the number. It will do the mathematically proper rounding for you as well.

OCTAL, HEX

It is possible to represent an integer in hexa-decimal or octal form.

```
>>> number = 0xA0F        #Hexa-decimal
>>> number
2575

>>> number=0o37     #Octal
>>> number
31

>>> print('Octal uses the letter "o" lowercase. %d =%o' % (10,10))
Octal uses the letter "o" lowercase. 10=12

>>> print('Octal of: %d %o' % (8,8))
Octal of: 8 10

>>> print('Hex uses the letter "x" or "X". %d %x %X' % (10, 10, 10))
Hex uses the letter "x" or "X". 10 a A

>>> print('Hex of: %d %x %X' % (25, 25, 25))
Hex of: 25 19 19

>>> print('Hex of: %d %x %X' % (33, 33, 33))
Hex of: 33 21 21

>>> print('Hex of: %d %x %X' % (45, 45, 45))
Hex of: 45 2d 2D
>>>
```

Built-in mathematical functions
max, min, abs, round.

BUILT-IN FUNCTIONS

The following example shows the usage of the **abs()** method.

```
print ("abs(-45) : ", abs(-45))
print ("abs(100.12) : ", abs(100.12))
```

When we run the above program, it produces the following result

```
abs(-45) : 45
abs(100.12) : 100.12
```

The following example shows the usage of **round()** method.

```
print ("round(70.23456) : ", round(70.23456))
print ("round(56.659,1) : ", round(56.659,1))
print ("round(80.264, 2) : ", round(80.264, 2))
print ("round(100.000056, 3) : ", round(100.000056, 3))
print ("round(-100.000056, 3) : ", round(-100.000056, 3))
```

When we run the above program, it produces the following result

```
round(70.23456) : 70
round(56.659,1) : 56.7
round(80.264, 2) : 80.26
round(100.000056, 3) : 100.0
round(-100.000056, 3) : -100.0
```

The following example shows the usage of the **max()** method.

```
print ("max(80, 100, 1000) : ", max(80, 100, 1000))
print ("max(-20, 100, 400) : ", max(-20, 100, 400))
print ("max(-80, -20, -10) : ", max(-80, -20, -10))
print ("max(0, 100, -400) : ", max(0, 100, -400))
```

When we run the above program, it produces the following result

```
max(80, 100, 1000) : 1000
max(-20, 100, 400) : 400
max(-80, -20, -10) : -10
max(0, 100, -400) : 100
```

The following example shows the usage of the **min()** method.

```
print ("min(80, 100, 1000) : ", min(80, 100, 1000))
print ("min(-20, 100, 400) : ", min(-20, 100, 400))
print ("min(-80, -20, -10) : ", min(-80, -20, -10))
print ("min(0, 100, -400) : ", min(0, 100, -400))
```

When we run the above program, it produces the following result

```
min(80, 100, 1000) : 80
min(-20, 100, 400) : -20
min(-80, -20, -10) : -80
min(0, 100, -400) : -400
```