

**Scientific Programming in Python**  
**Notes**  
**22 September 2009**

## Simple Arithmetic

**Summary** The operators for numbers, in order of precedence, are

1. power (\*\*),
2. multiply (\*), divide (/), integer divide (//), remainder (%),
3. add (+), subtract (-).

A large part of Python is dedicated to manipulating numbers. Python has libraries for doing the most mundane arithmetic to applying the most advanced numerical methods. We are going to start by using Python as a calculator.

```
>>> 3+4
7
```

The above example shows how Python does addition. The other basic mathematical operations are also available.

```
>>> 3-4
-1
>>> 3*4
12
>>> 3/4
0.75
>>> 3**4
81
```

The last example above is of raising to a power,  $3^4 = 81$ . There are two more operators specifically for integer division. The // operator gives the quotient and the % the remainder.

```
>>> 13//4
3
>>> 13%4
1
```

We get these results since  $\frac{13}{4} = 3 + \frac{1}{4}$ . Python also supports the use of parentheses to clarify the order in which operations are to be performed.

```
>>> (1+2)*3-4//5
9
```

Without parentheses, operations are performed in the following order.

1. power (\*\*),
2. multiplication and division (\*, /, //, %),
3. addition and subtraction (+, -).

## Types of numbers

**Summary** The basic numerical types are integer (`int`), long integer (`long`), floating point (`float`), complex (`complex`).

Note that in the calculations above we have encountered two types of numbers—integers and real numbers. In Python real numbers are known as *floating point* numbers and are always displayed with a decimal point (`.`). We can ask Python what the type of a number is.

```
>>> type(3)
<type 'int'>
>>> type(3.4)
<type 'float'>
```

Here `int` is short for *integer* and `float` for *floating point number*. Imaginary numbers are formed using the suffix `j` (or `J`). All of the normal mathematical operators are defined for complex numbers.

```
>>> 3*(1+1j) - (1.8+0.5j)
(1.2+2.5j)
>>> 1j ** 2
(-1+0j)
>>> type(2 + 1j)
<type 'complex'>
```

One last note on floating point numbers—because of the way in which these numbers are represented on a computer, they are limited in that they do not have perfect precision.

```
>>> 1/3
0.33333333333333331
>>> 1/10
0.10000000000000001
>>> (3+4j) - (1.8+0.6j)
(1.2+3.3999999999999999j)
```

Note that for each of these commands the result is almost but not exactly what you would expect. Because of limited precision floating point numbers are prone to numerical errors. Normally these errors are very small ( $\approx 10^{-16}$  in the examples above) but you should be aware of them as they can grow larger if you are not careful.

Even though Python doesn't really tell you, the integers as described in the previous section are limited.

```
>>> 2**31
2147483648L
```

Did you notice the `L` at the end of the result? This shows that Python is representing this number as a *long integer* (`long`). Normal integers are in the range  $[-2^{31}, 2^{31} - 1]$ . Operations on long integers take more time to execute but have no restrictions on their range. To prove this to yourself, try to generate some big numbers.

```
>>> 2**1000
1071508607186267320948425049060001810561404811705533607443750388370351
0511249361224931983788156958581275946729175531468251871452856923140435
9845775746985748039345677748242309854210746050623711418779541821530464
7498358194126739876755916554394607706291457119647768654216766042983165
2624386837205668069376L
```

In practice you never have to worry about when to use long integers and when not to. Python will automatically convert integer results outside the  $[-2^{31}, 2^{31} - 1]$  range to long integers.

## Assignment

We use the assignment of values to variables to store the results of our calculations for later use.

```
>>> x = (3+4)*8
>>> y = 9*5
>>> x+y
101
```

Here we assign values to the variables `x` and `y` and then use those values in a calculation. Note that the `=` symbol means something different from the one used in mathematics. In mathematics it refers to equality; in Python to assignment. This code

```
>>> x = 3
>>> x = 5
```

makes perfect sense. It simply means that the value 3 is assigned to `x` in the first line and overwritten with the value 5 in the second line. If we regarded `=` as an equality operator, the above would have been a contradiction. There are some other assignment operators that are simply shorthand notation. These are

- add to (`+=`);
- subtract from (`-=`);
- multiply with (`*=`);
- divide by (`/=`);
- integer divide by (`//=`);
- remainder when divided by (`%=`); and
- raised to power (`**=`).

The first of these operators, `+=`, adds whatever is on its right hand side to the current value in the variable on the left hand side and stores that in the variable.

```
>>> x = 5
>>> x
5
>>> x += 1.3
>>> x
6.3
```

Similarly the `-=`, `*=`, `/=`, `//=`, `%=` and `**=` operators perform their respective arithmetic operations with the variable and assign the new value to the variable.

```
>>> x = 1
>>> y = 2
>>> z = 3.0

>>> x += 5           # x is now 6           equivalent to x = x + 5
>>> y -= x           # y is now -4          equivalent to y = y - x
>>> z *= 9           # z is now 27.0        equivalent to z = z * 9
>>> x //= -y // 2    # x is now 3           equivalent to x = x // (-y // 2)
>>> z /= y           # z is now -6.75       equivalent to z = z / y
>>> x %= 2           # x is now 1           equivalent to x = x % 2
```

Note that the `#` code starts a single line comment. Everything after the `#` until the end of the line is ignored by the Python interpreter. This is useful for adding any comments or other text to your code.

## Variables

Python places restrictions on what variable names may look like. As you have already seen, `x` and `y` are valid names. The rule is as follows.

- A variable name may start with any alphabetical character or an underscore (`_`)
- After the first character you may use any number of alphabetical characters, underscores, and digits.

The following are valid variable names.

```
var1
_this_is_a_variable_
student_name123
```

The following are not

```
3_blind_mice # may not start with a digit
ready?      # may not contain the ? character
```

Finally, note that Python is case-sensitive—it distinguishes between capital and lower case letters. The variable names `value`, `Value` and `ValuE` are all different.

## Mathematical Functions

We will discuss functions in depth later on but, for now, the code below shows two well-known mathematical functions available in Python—sine (`sin`) and square root (`sqrt`). You can also see  $\pi$  (`pi`) being used.

```
>>> sin(pi/4)
0.70710678118654746
>>> 1/sqrt(2)
0.70710678118654746
```

Some of the functions and constants available are listed in the table below. The hyperbolic variants of the trigonometric functions are also defined, for example `cosh` and `arcsinh`.

<code>arccos(x)</code>	$\cos^{-1} x$	<code>cos(x)</code>	$\cos x$	<code>ceil(x)</code>	$\lceil x \rceil$
<code>arcsin(x)</code>	$\sin^{-1} x$	<code>sin(x)</code>	$\sin x$	<code>floor(x)</code>	$\lfloor x \rfloor$
<code>arctan(x)</code>	$\tan^{-1} x$	<code>tan(x)</code>	$\tan x$	<code>abs(x)</code>	$ x $
<code>arctan2(y,x)</code>	$\tan^{-1} \frac{y}{x}$	<code>exp(x)</code>	$e^x$	<code>conj(z)</code>	$\bar{z}$
<code>sqrt(x)</code>	$\sqrt{x}$	<code>log(x)</code>	$\ln x$		
<code>hypot(x,y)</code>	$\sqrt{x^2 + y^2}$	<code>log10(x)</code>	$\log_{10} x$		
<code>e</code>	$e \approx 2.718$	<code>pi</code>	$\pi \approx 3.142$	<code>inf</code>	$\infty$