Numeric Types

There are four numeric types:

1. Integers, represented as a 32 bit (or longer) quantity. Digits sequences (possibly) signed are integer literals:
   1   -123   +456

2. Long integers, of unlimited precision. An integer literal followed by an \texttt{l} or \texttt{L} is a long integer literal:
   12345678900000000000000L

3. Floating point values, represented as a 64 bit floating point number. Literals are of fixed decimal or exponential form:
   123.456   1e10   6.0231023

4. Complex numbers, represented as a pair of floating point numbers. In complex literals \texttt{j} or \texttt{J} is used to
denote the imaginary part of the complex value:

1.0+2.0j  -22.1j  10e10J+20.0

There is no character type. A literal like 'a' or "c" denotes a string of length one.

There is no boolean type. A zero numeric value (any form), or None (the equivalent of void) or an empty string, list, tuple or dictionary is treated as false; other values are treated as true.

Hence

"abc" and "def"

is treated as true in an if, since both strings are non-empty.
## Arithmetic Operators

<table>
<thead>
<tr>
<th>Op</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>**</td>
<td>Exponentiation</td>
</tr>
<tr>
<td>−</td>
<td>Unary plus</td>
</tr>
<tr>
<td>+</td>
<td>Unary minus</td>
</tr>
<tr>
<td>~</td>
<td>Bit-wise complement (int or long only)</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
</tr>
<tr>
<td>%</td>
<td>Remainder</td>
</tr>
<tr>
<td>−</td>
<td>Binary plus</td>
</tr>
<tr>
<td>+</td>
<td>Binary minus</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>Bit-wise left shift (int or long only)</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>Bit-wise right shift (int or long only)</td>
</tr>
<tr>
<td>&amp;</td>
<td>Bit-wise and (int or long only)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>Bit-wise Xor (int or long only)</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
</tr>
</tbody>
</table>
\[ \begin{align*}
\geq & \quad \text{Greater than or equal} \\
\leq & \quad \text{Less than or equal} \\
== & \quad \text{Equal} \\
!= & \quad \text{Not equal} \\
\text{and} & \quad \text{Boolean and} \\
\text{or} & \quad \text{Boolean or} \\
\text{not} & \quad \text{Boolean not}
\end{align*} \]
Operator Precedence Levels

Listed from lowest to highest:

- `or` Boolean OR
- `and` Boolean AND
- `not` Boolean NOT
- `<, <=, >, >=, <>`, `!=`, `==` Comparisons
- `|` Bitwise OR
- `^` Bitwise XOR
- `&` Bitwise AND
- `<<, >>` Shifts
- `+`, `-` Addition and subtraction
- `*`, `/`, `%` Multiplication, division, remainder
- `**` Exponentiation
- `+`, `-` Positive, negative (unary)
- `~` Bitwise not
Arithmetic Operator Use

Arithmetic operators may be used with any arithmetic type, with conversions automatically applied. Bit-wise operations are restricted to integers and long integers. The result type is determined by the “generality” of the operands. (Long is more general than int, float is more general than both int and long, complex is the most general numeric type). Thus

```python
>>> 1 + 2
3
>>> 1 + 111L
112L
>>> 1 + 1.1
2.1
>>> 1 + 2.0j
(1+2j)
```
Unlike almost all other programming languages, relational operators may be “chained” (as in standard mathematics).

Therefore
\[ a > b > c \]

means \((a > b) \) and \((b > c)\)
Assignment Statements

In Python assignment is a statement not an expression.

Thus

\[ a + (b = 2) \]

is illegal.

Chained assignments are allowed:

\[ a = b = 3 \]

Since Python is dynamically typed, the type (and value) associated with an identifier can change because of an assignment:

```python
>>> a = 0
>>> print a
0
>>> a = a + 0L
>>> print a
```

573
>>> a = a + 0.0
>>> print a
0.0

>>> a = a + 0.0J
>>> print a
0j
If and While Statements

Python contains if and while statements that are fairly similar to those found in C and Java. There are some significant differences though.

A line that contains an if, else or while ends in a “:” Thus we might write:

```python
if a > 0:
    b = 1
```

Moreover the indentation of the then part is significant! You don’t need { and } in Python because all statements indented at the same level are assumed to be part of the same block.
In the following Python statements
if a>0:
    b=1
c=2
d=3
the assignments to \texttt{b} and \texttt{c} constitute then part; the assignment to \texttt{d} follows the if statement, and is independent of it. In interactive mode a blank line is needed to allow the interpreter to determine where the if statement ends; this blank line is not needed in batch mode.
The if Statement

The full form of the `if` statement is

```python
if expression:
    statement(s)
elif expression:
    statement(s)
...
else:
    statement(s)
```

Note those pesky `:`'s at the end of the `if`, `elif` and `else` lines. The expressions following the `if` and optional `elif` lines are evaluated until one evaluates to true. Then the following statement(s), delimited by indentation, are executed. If no expression evaluates to true, the statements following the `else` are executed.
Use of `else` and `elif` are optional; a “bare” `if` may be used.

If any of the lists of statements is to be null, use `pass` to indicate that nothing is to be done.

For example

```python
if a > 0:
    b = 1
elif a < 0:
    pass
else:
    b = 0
```

This `if` sets `b` to 1 if `a` is > 0; it sets `b` to 0 if `a` == 0, and does nothing if `a` < 0.
While Loops

Python contains a fairly conventional while loop:

    while expression:
        body

Note the “:” that ends the header line. Also, indentation delimits the body of the loop; no braces are needed. For example,

```python
>>> a=0; b=0
>>> while a < 5:
    ... b = b+a**2
    ... a= a+1
    ...

>>> print a,b
5 30
```
Break, Continue and Else in Loops

Like C, C++ and Java, Python allows use of `break` within a loop to force loop termination. For example,

```python
>>> a = 1
>>> while a < 10:
...   if a + a == a**2:
...     break
...   else:
...     a = a + 1
...  
>>> print a
2
```
A continue may be used to force the next loop iteration:

```python
>>> a=1
>>> while a < 100:
...     a=a+1
...     if a%2==0:
...         continue
...     a=3*a
...

>>> print a
105```

Python also allows you to add an else clause to a while (or for) loop. The syntax is

```
while expression:
    body
else:
    statement(s)
```

The else statements are executed when the termination condition becomes false, but not when the loop is terminated with a break. As a result, you can readily program “search loops” that need to handle the special case of search failure:
>>> a=1
>>> while a < 1000:
...     if a**2 == 3*a-1:
...         print "winner: ",a
...         break
...     a=a+1
... else:
...     print "No match"
... 
No match
Sequence Types

Python includes three sequence types: strings, tuples and lists. All sequence types may be indexed, using a very general indexing system.

Strings are sequences of characters; tuples and lists may contain any type or combination of types (like Scheme lists).

Strings and tuples are immutable (their components may not be changed). Lists are mutable, and be updated, much like arrays.

Strings may be delimited by either a single quote (' ') or a double quote (" ") or even a triple quote (''' or """""""). A given string must start and stop with the same delimiter. Triply quoted strings may span multiple lines. There
is no character type or value; characters are simply strings of length 1. Legal strings include
'abc' "xyz" "'It's OK!'"
Lists are delimited by “[“ and “]”. Empty (or null lists) are allowed. Valid list literals include

[1,2,3] ["one",1]
[['a'],['b'],['c']] []

Tuples are a sequence of values separated by commas. A tuple may be enclosed within parentheses, but this isn’t required. A empty tuple is (). A singleton tuple ends with a comma (to distinguish it from a simple scalar value).

Thus (1,) or just 1, is a valid tuple of length one.
Indexing Sequence Types

Python provides a very general and powerful indexing mechanism. An index is enclosed in brackets, just like a subscript in C or Java. Indexing starts at 0.

Thus we may have

```python
>>> 'abcde'[2]
'c'
>>> [1,2,3,4,5][1]
2
>>> (1.1,2.2,3.3)[0]
1.1
```

Using an index that’s too big raises an IndexError exception:

```python
>>> 'abc'[3]
IndexError: string index out of range
```
Unlike most languages, you can use negative index values; these simply index from the right:

```python
>>> 'abc' [-1]
'c'
>>> [5, 4, 3, 2, 1] [-2]
2
>>> (1, 2, 3, 4) [-4]
1
```

You may also access a slice of a sequence value by supplying a range of index values. The notation is

```python
data[i:j]
```

which selects the values in `data` that are \( \geq i \) and \( < j \). Thus

```python
>>> 'abcde' [1:2]
'b'
>>> 'abcde' [0:3]
'abc'
```
You may omit a lower or upper bound on a range. A missing lower bound defaults to 0 and a missing upper bound defaults to the maximum legal index. For example,

>>> [1, 2, 3, 4, 5][2:]
[3, 4, 5]
>>> [1, 2, 3, 4, 5][:3]
[1, 2, 3]

An upper bound that’s too large in a range is interpreted as the maximum legal index:

>>> 'abcdef'[3:100]
'def'

You may use negative values in ranges too—they’re interpreted as being relative to the right end of the sequence:
>>> 'abcde'[0:-2]
'abc'
>>> 'abcdefg'[-5:-2]
'cde'
>>> 'abcde'[-3:]
'cde'
>>> 'abcde'[: -1]
'abcd'

Since arrays may be assigned to, you may assign a slice to change several values at once:

>>> a=[1,2,3,4]
>>> a[0:2]=[-1,-2]
>>> a
[-1, -2, 3, 4]
>>> a[2:]=[33,44]
>>> a
[-1, -2, 33, 44]
The length of the value assigned to a slice need not be the same size as the slice itself, so you can shrink or expand a list by assigning slices:

>>> a=[1,2,3,4,5]
>>> a[2:3]=[3.1,3.2]
>>> a
[1, 2, 3.1, 3.2, 4, 5]
>>> a[4:]=[]
>>> a
[1, 2, 3.1, 3.2]
>>> a[:0]=[-3,-2,-1]
>>> a
[-3, -2, -1, 1, 2, 3.1, 3.2]
Other Operations on Sequences

Besides indexing and slicing, a number of other useful operations are provided for sequence types (strings, lists and tuples).

These include:

+ (catenation):

```python
>>> [1,2,3]+[4,5,6]
[1, 2, 3, 4, 5, 6]
>>> (1,2,3)+(4,5)
(1, 2, 3, 4, 5)
>>> (1,2,3)+[4,5]
TypeError: illegal argument type for built-in operation
```
• * (Repetition):

```python
>>> 'abc'*2
'abcabc'
>>> [3,4,5]*3
[3, 4, 5, 3, 4, 5, 3, 4, 5]
```

• Membership (in, not in)

```python
>>> 3 in [1,2,3,4]
1
>>> 'c' in 'abcde'
1
```

• max and min:

```python
>>> max([3,8,-9,22,4])
22
>>> min('aa','bb','abc')
'aa'
```
Operations on Lists

As well as the operations available for all sequence types (including lists), there are many other useful operations available for lists. These include:

- **count** (Count occurrences of an item in a list):
  
  ```python
  >>> [1,2,3,3,21].count(3)
  2
  ```

- **index** (Find first occurrence of an item in a list):
  
  ```python
  >>> [1,2,3,3,21].index(3)
  2
  >>> [1,2,3,3,21].index(17)
  ValueError: list.index(x): x not in list
  ```
• **remove** (Find and remove an item from a list):

```python
>>> a=[1,2,3,4,5]
>>> a.remove(4)
>>> a
[1, 2, 3, 5]
>>> a.remove(17)
ValueError: list.remove(x): x not in list
```

• **pop** (Fetch and remove i-th element of a list):

```python
>>> a=[1,2,3,4,5]
>>> a.pop(3)
4
>>> a
[1, 2, 3, 5]
>>> a.pop()
5
>>> a
[1, 2, 3]
```
• reverse a list:
  >>> a=[1,2,3,4,5]
  >>> a.reverse()
  >>> a
  [5, 4, 3, 2, 1]

• sort a list:
  >>> a=[5,1,4,2,3]
  >>> a.sort()
  >>> a
  [1, 2, 3, 4, 5]

• Create a range of values:
  >>> range(1,5)
  [1, 2, 3, 4]
  >>> range(1,10,2)
  [1, 3, 5, 7, 9]
Dictionaries

Python also provides a dictionary type (sometimes called an associative array). In a dictionary you can use a number (including a float or complex), string or tuple as an index. In fact any immutable type can be an index (this excludes lists and dictionaries).

An empty dictionary is denoted \{\}. A non-empty dictionary may be written as

\{ key_1:value_1, key_2:value_2, ... \}

For example,

```
c={ 'bmw':545, 'lexus':'sc 430', 'mercedes':'e 500'}
```
You can use a dictionary much like an array, indexing it using keys, and updating it by assigning a new value to a key:

```python
>>> c['bmw']
545
>>> c['bmw'] = 'm5'
>>> c['honda'] = 'accord'
```

You can delete a value using `del`:

```python
>>> del c['honda']
>>> c['honda']
KeyError: honda
```
You can also check to see if a given key is valid, and also list all keys, values, or key-value pairs in use:

```python
>>> c.has_key('edsel')
0

>>> c.keys()
['bmw', 'mercedes', 'lexus']

>>> c.values()
['m5', 'e 500', 'sc 430']

>>> c.items()
[('bmw', 'm5'), ('mercedes', 'e 500'), ('lexus', 'sc 430')]
```
For Loops

In Python’s `for` loops, you don’t explicitly control the steps of an iteration. Instead, you provide a sequence type (a string, list or sequence), and Python automatically steps through the values.

Like a `while` loop, you must end the for loop header with a “:” and the body is delimited using indentation. For example,

```python
>>> for c in 'abc':
    ...    print c
    ...
 a
 b
 c
```
The `range` function, which creates a list of values in a fixed range is useful in for loops:

```python
>>> a=[5,2,1,4]
>>> for i in range(0,len(a)):
...     a[i]=2*a[i]
...
>>> print a
[10, 4, 2, 8]
```
You can use an else with for loops too. Once the values in the specified sequence are exhausted, the else is executed unless the for is exited using a break. For example,

```python
for i in a:
    if i < 0:
        print 'Neg val:',i
        break
else:
    print 'No neg vals'
```
Function Definitions

Function definitions are of the form

```
def  name(args):
    body
```

The symbol `def` tells Python that a function is to be defined. The function is called `name` and `args` is a tuple defining the names of the function’s arguments. The body of the function is delimited using indentation. For example,

```
def fact(n):
    if n<=1:
        return 1
    else:
        return n*fact(n-1)

>>> fact(5)
120
>>> fact(20L)
```
Scalar parameters are passed by value; mutable objects are allocated in the heap and hence are passed (in effect) by reference:

```python
>>> def asg(ar):
    ... a[1]=0
    ... print ar
    ... print ar

>>> a=[1,2,3,4.5]
>>> asg(a)
[1, 0, 3, 4.5]```
Arguments may be given a default value, making them optional in a call. Optional parameters must follow required parameters in definitions. For example,

```python
>>> def expo(val, exp=2):
...     return val**exp
...
>>> expo(3, 3)
27
>>> expo(3)
9
>>> expo()
TypeError: not enough arguments; expected 1, got 0
```
A variable number of arguments is allowed; you prefix the last formal parameter with a \*; this parameter is bound to a tuple containing all the actual parameters provided by the caller:

```python
>>> def sum(*args):
...   sum=0
...   for i in args:
...     sum=sum+i
...   return sum
...

>>> sum(1,2,3)
6
>>> sum(2)
2
>>> sum()
0
```
You may also use the name of formal parameters in a call, making the order of parameters less important:

```python
>>> def cat(left="[", body="", right"]"):  
...     return left+body+right
...  
>>> cat(body='xyz');
' [xyz]'  
>>> cat(body='hi there!', left='--[')
'--[hi there!]'
```
Scoping Rules in Functions

Each function body has its own local namespace during execution. An identifier is resolved (if possible) in the local namespace, then (if necessary) in the global namespace.

Thus

```python
>>> def f():
...     a=11
...     return a+b
... 
>>> b=2; f()
13
>>> a=22; f()
13
>>> b=33; f()
44
```
Assignments are to local variables, even if a global exists. To force an assignment to refer to a global identifier, you use the declaration

```
global id
```

which tells Python that in this function `id` should be considered global rather than local. For example,

```python
>>> a=1;b=2
>>> def f():
...     global a
...     a=111;b=222
... 
>>> f();print a,b
111 2
```