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

```
>>> '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:

```
>>> 'abc'[3]
IndexError: string index out of
range
```

Unlike most languages, you can use *negative* index values; these simply index from the *right*:

```
>>> '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

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

which selects the values in **`data`** that are **`>=i`** and **`< j`**. Thus

```
>>> 'abcde'[1:2]
'b'
>>> 'abcde'[0:3]
'abc'
```

```
>>> 'abcde'[2:2]
''
```

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):

```
>>> [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
>>> "abc"+"def"
'abcdef'
```

- `*` (Repetition):

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

- Membership (`in`, `not in`)

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

- `max` and `min`:

```
>>> 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):

```
>>> [1,2,3,3,21].count(3)
2
```

- **`index`** (Find first occurrence of an item in a list):

```
>>> [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):

```
>>> 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):

```
>>> 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]
>>> range(10,1,-2)
[10, 8, 6, 4, 2]
```

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':650, 'lexus':'LS 460',
    'mercedes':'S 550'}
```

You can use a dictionary much like an array, indexing it using keys, and updating it by assigning a new value to a key:

```
>>> c['bmw']
650
>>> c['bmw']='M6'
>>> c['honda']='accord'
```

You can delete a value using **`del`**:

```
>>> 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:

```
>>> c.has_key('edsel')
0
>>> c.keys()
['bmw', 'mercedes', 'lexus']
>>> c.values()
['M6', 'S 550', 'LS 460']
>>> c.items()
[('bmw', 'M6'), ('mercedes',
 'S 550'), ('lexus', 'LS 460')]
```

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,

```
>>> 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:

```
>>> 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,

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

Sets

Lists are often used to represent sets, and Python allows a list (or string or tuple) to be converted to a set using the **set** function:

```
>>> set([1,2,3,1])
set([1, 2, 3])
>>> set("abac")
set(['a', 'c', 'b'])
>>> set((1,2,3,2,1))
set([1, 2, 3])
```

Sets (of course) disallow duplicate elements. They are unordered (and thus can't be indexed), but they can be iterated through using a **for**:

```
>>> for v in set([1,1,2,2,3,4,2,1]):
...     print v,

1 2 3 4
```

The usual set operators are provided:

Union (|),

Intersection (&),

Difference (-)

and Symmetric Difference (^, select members in either but not both operands)

```
>>> set([1,2,3]) | set([3,4,5])
set([1, 2, 3, 4, 5])
>>> set([1,2,3]) & set([3,4,5])
set([3])
>>> set([1,2,3]) -  set([3,4,5])
set([1, 2])
>>> set([1,2,3]) ^ set([3,4,5])
set([1, 2, 4, 5])
```

List Comprehensions

Python provides an elegant mechanism for building a list by embedding a **`for`** within list brackets. This a termed a *List Comprehension.*

The general form is an expression, followed by a **`for`** to generate values, optionally followed by **`if`**s (to select or reject values) of additional **`for`**s.

In essence this is a procedural version of a **`map`**, without the need to actually provide a function to be mapped.

To begin with a simple example,

```
>>> [2*i for i in [1,2,3]]
[2, 4, 6]
```

This is the same as mapping the doubling function on the list **`[1,2,3]`**, but without an explicit function.

With an **`if`** to filter values, we might have:

```
>>> [2*i for i in [3,2,1,0,-1] if i != 0]
[6, 4, 2, -2]
```

We can also (in effect) nest **`for`**'s:

```
[(x,y) for x in [1,2,3] for y in [-1,0] ]

[(1, -1), (1, 0), (2, -1), (2, 0),
(3, -1), (3, 0)]
```

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)
```

```
2432902008176640000L
>>> fact(2.5)
3.75
>>> fact(2+1J)
(1+3j)
```

Scalar parameters are passed by value; mutable objects are allocated in the heap and hence are passed (in effect) by reference:

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

```
 >>> 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:

```
>>> 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:

```
>>> 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

```
>>> 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,

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

Other Operations on Functions

Since Python is interpreted, you can dynamically create and execute Python code.

The function **`eval(string)`** interprets **`string`** as a Python expression (in the current execution environment) and returns what is computed. For example,

```
>>> a=1;b=2
>>> eval('a+b')
3
```

`exec(string)` executes **`string`** as arbitrary Python code (in the current environment):

```
>>> a=1;b=2
>>> exec('for op in "+-*/":
print(eval("a"+op+"b"))')
3
-1
2
0
```

`execfile(string)` executes the contents of the file whose pathname is specified by **`string`**. This can be useful in loading an existing set of Python definitions.

The expression

```
lambda args: expression
```

creates an anonymous function with **args** as its argument list and **expression** as it body. For example,

```
>>> (lambda a:a+1)(2)
3
```

And there are definitions of **map**, **reduce** and **filter** to map a function to a list of values, to reduce a list (using a binary function) and to select values from a list (using a predicate):

```
>>> def double(a):
...    return 2*a;
...
>>> map(double,[1,2,3,4])
[2, 4, 6, 8]
```

```
>>> def sum(a,b):
...   return a+b
...
>>> reduce(sum,[1,2,3,4,5])
15
>>> def even(a):
...   return not(a%2)
...
>>> filter(even,[1,2,3,4,5])
[2, 4]
```

Generators

Many languages, including Java, C# and Python provide iterators to index through a collection of values. Typically, a `next` function is provided to generate the next value and `hasNext` is used to test for termination.

Python provides *generators*, a variety of function (in effect a co-routine) to easily and cleanly generate the sequence of values required of an iterator.

In any function a `yield` (rather than a `return`) can provide a value and suspend execution. When the next value is needed (by an invisible call to `next`) the function is resumed at the point of the `yield`. Further yields generate successive values. Normal termination indicates that **`hasNext`** is no longer true.

As a very simple example, the following function generates all the values in a list **`L`** except the initial value:

```
>>> def allButFirst(L):
...     for i in L[1:]:
...         yield i

>>> for j in allButFirst([1,2,3,4]):
...    print j,

2 3 4
```

The power of generators is their ability to create non-standard traversals of a data structure in a clean and compact manner.

As an example, assume we wish to visit the elements of a list not in left-to-right or right-to-left order, but in an order that visits even positions first, then odd positions. That is we will first see **`L[0]`**, then **`L[2]`**, then **`L[4]`**, ..., then **`L[1]`**, **`L[3]`**, ...

We just write a generator that takes a list and produces the correct visit order:

```
>>> def even_odd(L):
...    ind = range(0,len(L),2)
...    ind = ind + range(1,len(L),2)
...    for i in ind:
...      yield L[i]
```

Then we can use this generator wherever an iterator is needed:

```
>>> for j in even_odd([10,11,12,13,14]):
...    print j,
...
10 12 14 11 13
```

Generators work in list comprehensions too:

```
>>> [j for j in even_odd([11,12,13])]
[11, 13, 12]
```