University of Zurich[UZH]

Grid Computing Competence Center

Sequences and iteration in Python

GC3: Grid Computing Competence Center,
University of Zurich

Sep. 28, 2012

Sequences

Python provides a few built-in *sequence* classes:

list	*mutable*, possibly inhomogeneous
tuple	*immutable*, possibly inhomogeneous
str	*immutable*, only holds characters

An additional class is provided by the NumPy package, and is commonly used in scientific Python codes:

array	*mutable*, homogeneous

Lists

Lists are by far the most common and used sequence type in Python.

Lists are created and initialized by enclosing values into '[' and ']':

```
>>> L = [ 'G', 'C' ]
```

You can append and remove items from a list:

```
>>> L.append('3')
>>> print (L)
['G', 'C', '3']
```

Reference: http://docs.python.org/tutorial/datastructures.html#more-on-lists

Sequences, II

You can access individual items in a sequence using the postfix `[]` operator.

Sequence indices start at 0.

```
>>> L = ['G', 'C', '3']
>>> print L[0], L[1], L[2]
'G' 'C' '3'
>>> S = 'GC3'
>>> print S[0], S[1], S[2]
'G' 'C' '3'
```

The `len()` function returns the number of elements in a sequence.

```
>>> len(L)
3
```

Slices

The notation `[`n`:`m`]` is used for accessing a *slice* of sequence (the items at positions n, $n+1$, ..., $m-1$).

```
>>> # list numbers from 0 to 9
>>> R = range(0,10)
>>> R[1:4]
[1, 2, 3]
```

If n is omitted it defaults to 0, if m is omitted it defaults to the length of the sequence.

List mutation

You can replace items in a *mutable* sequence by assigning them a new value:

```
>>> L[2] = 'Z'
>>> print L
['G', 'C', 'Z']
```

You can also replace an entire slice of a mutable sequence:

```
>>> L[1:3] = 'GG'
>>> print L
['G', 'G', 'G']
```

The new slice does not need to have the same length:

```
>>> L[1:] = 'Zurich'
>>> print L
['G', 'Z', 'u', 'r', 'i', 'c', 'h']
```

Dictionaries

The `dict` type implements a key/value mapping:

```
>>> D = { }
>>> D['a'] = 1
>>> D[2] = 'b'
>>> D
{'a': 1, 2: 'b'}
```

Equivalently, dictionaries can be created and initialized using two different syntaxes:

```
>>> D1 = { 'a':1, 'b':2 }
>>> D2 = dict(a=1, b=2)
>>> D1 == D2
True
```

Sets

The `set` type implements an unordered container that holds exactly one object per equivalence class:

```
>>> S = set()
>>> S.add(1)
>>> S.add(2)
>>> S.add(1)
>>> S
set([1, 2])
```

You can create a set and add elements to it in one go:

```
>>> S2 = set([1, 2])
>>> S2 == S
True
```

Mutable vs Immutable

Some objects (e.g., `tuple`, `int`, `str`) cannot be modified.

```
>>> T = ('U', 'Z', 'H')
>>> T[2] = 'h'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

```
>>> S = 'GC3'
>>> S[2] = '2'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignmen
```

`list`, `dict`, `set` and user-defined objects are mutable and can be modified in-place.

The 'in' operator

Use the **in** operator to test for presence of an item in a collection.

x in S
Evaluates to `True` if `x` is equal to a *value* contained in the `S` sequence (list, tuple, set).

x in D
Evaluates to `True` if `x` is equal to a *key* in the `D` dictionary.

x in T
Evaluates to `True` if `x` is a substring of string `T`.

All variables are references

In Python, **all objects are ever passed by reference**.

In particular, **variables always store a reference to an object**, never a copy!

Hence, you have to be careful when modifying objects:

```
>>> a = [1,2,3]
>>> b = a
>>> b.remove(2)
>>> a
[1, 3]
```

This applies particularly for variables that capture the arguments to a function call!

All variables are references, II

However:

```
>>> a = 1
>>> b = a
>>> b += 1
>>> a
1
>>> b
2
```

Question: *How can you explain this?*

All variables are references, II

However:

```
>>> a = 1
>>> b = a
>>> b += 1
>>> a
1
>>> b
2
```

Question: *How can you explain this?*
The `b += 1` *operator could be replaced by* `b = b + 1`*, and the* `b+1` *expression yields a new value.*

`for`-loops

With the `for` statement, you can loop over the values in a list:

```
for i in range(0, 4):
  # loop block
  print (i*i)
```

To break out of a `for` loop, use the `break` statement.

To jump to the next iteration of a `for` loop, use the `continue` statement.

The `for` statement can be used to loop over elements in *any sequence*.

```
>>> for val in 1,2,3:
...   print val
1
2
3
```

Loop over tuples

The `for` statement can be used to loop over elements in *any sequence*.

```
>>> for val in 'GC3':
...   print val
'G'
'C'
'3'
```

Loop over strings

The `for` statement can be used to loop over elements in *any sequence*.

```
>>> D = dict(a=1, b=2)
>>> for val in D.keys():
...     print val
'a'
'b'
```

Loop over dictionary *keys*. *The* `.keys()` *part can be omitted, as it's the default!*

If you want to loop over dictionary *values*, you have to explicitly request it.

```
>>> D = dict(a=1, b=2)
>>> for val in D.values():
...   print val
1
2
```

Loop over dictionary *values*
The `.values()` *cannot be omitted!*

Multiple assignment can be used in `for` statements as well.

```
>>> L = [(1,'a'), (2,'b'), (3, 'c')]
>>> for num, char in L:
...     print ("num is " + str(num)
...            + ' and char is ' + char)
```

This is particularly useful with functions that return a tuple. For instance the `enumerate()` function (look it up with `help()`!).

Exercise A: Write a function `invert(D)` that takes a dictionary `D` and returns a dictionary `Dinv` with keys and values swapped. (We assume that `D` is 1-1.)

Exercise B: Implement a `zip2` function, that takes a list of 2-tuples and returns *two* lists: a list of all the first items in the pairs, and a list of all the second items in the pairs.

Default values

Function arguments can have default values.

```
>>> def hello(name='world'):
...     print ("Hello, " + name)
...
>>> hello()
'Hello, world'
```

Variadic functions, I

Functions like `sum`, `max`, `min` take a variable number of arguments.

That is possible with user-defined functions too.

If the last argument is prefixed with a `*` character, Python will make that argument into a *tuple* of arguments passed to the function. (Except for the ones that have already been assigned names.)

Variadic functions, II

```
>>> def varfn1(*args):
...   print args
>>> varfn1(1,2,3)
(1, 2, 3)

>>> def varfn2(a, b, *rest):
...   print rest
>>> varfn1(1,2,3)
(3,)
```

Variadic functions, III

You can call a function with an argument list that is only determined at run time.

The unary * operator takes any sequence and makes it into a function argument list:

```
>>> L = [1, 2, 3]
>>> max(*L)
3
```

Named arguments

Python allows calling a function with named arguments:

```
hello(name="Alice")
```

When passing arguments by name, they can be passed in any order:

```
>>> from fractions import Fraction
>>> Fraction(numerator=1, denominator=2)
Fraction(1, 2)
>>> Fraction(denominator=2, numerator=1)
Fraction(1, 2)
```

Keyword arguments, I

Python lets you catch arbitrary named arguments.

Prefix the very last argument name with `**`, and Python will make it into a dictionary: keys are actual argument names and dictionary values are actual argument values.

```
>>> def kwfn1(**kwargs):
...   print kwargs
>>> kwfn1(a=1, b=2)
{'a':1, 'b':2}
```

Keyword arguments, II

Keyword argument can be combined with positional arguments:

```
>>> def kwfn1(x, y, **kwargs):
...   print "x=%s␣y=%s␣kwargs=%s" % (x, y, kwargs)
>>> kwfn1(0, 42, a=1, b=2)
x=0 y=42 kwargs={'a':1, 'b':2}
```

. . . and also with variadic arguments:

```
>>> def kwfn2(x, *rest, **kwargs):
...   print ("x=%s␣rest=%s␣kwargs=%s"
...          % (x, rest, kwargs))
>>> kwfn2(0, 1, 2, 3, a=1, b=2)
x=0 rest=(1, 2, 3) kwargs={'a':1, 'b':2}
```

Keyword arguments, III

You can pass to a function a set of keyword arguments that is determined only at run time.
The `**` operator takes any *dictionary* and turns it into the bundle of keyword arguments for a function:

```
>>> D = { 'c':4, 'd':2 }
>>> kwfn2(x=1, **D)
x=1 rest=() kwargs={ 'c':4, 'd':2 }
```

Exercise C: Write a `maxarg` function, that takes arbitrary keyword arguments (but assume the values are all numbers), and returns the name of the argument with the largest value.

Example:

```
>>> maxarg(a=1, b=2)
'b'
```