A not so short introduction to Python: part II

Luca Schenato

Research Institute for Hydrogeological Protection
Italian National Research Council
(CNR-IRPI)

03/25/2011

Lists

What is a list

It is a mutable container of iterable items.

Lists are the more versatile compound data types of Python, used to group together other values, not all of the same type. They are written as a list of comma-separated values (items) between square brackets.

```
>>> a = ['spam', 'eggs', 100, 1234]
>>> a
['spam', 'eggs', 100, 1234]
```

Like string indices, list indices start at 0, and lists can be sliced, concatenated and so on:

```
>>> len(a)
4
>>> a[0]
'spam'
>>> a[-2]
100
```

Lists

Return type of slice operations on lists are lists theirselves:

```
>>> a[:2] + ['bacon', 2*2]
['spam', 'eggs', 'bacon', 4]
>>> 3*a[:3] + ['Boo!']
['spam', 'eggs', 100, 'spam', 'eggs', 100, 'spam',
     'eggs', 100, 'Boo!']
>>> a[:]
['spam', 'eggs', 100, 1234]
```

Unlike strings, which are immutable, it is possible to change individual elements of a list:

```
>>> a
['spam', 'eggs', 100, 1234]
>>> a[2] = a[2] + 23
>>> a
['spam', 'eggs', 123, 1234]
```

Lists

Assignment to slices is also possible, and this can even change the size of the list or clear it entirely:

```
>>> # Replace some items:
... a[0:2] = [1, 12]
>>> a
[1, 12, 123, 1234]
>>> # Remove some:
... a[0:2] = []
>>> a
[123, 1234]
>>> # Insert some:
... a[1:1] = ['bletch', 'xyzzy']
>>> a
[123, 'bletch', 'xyzzy', 1234]
>>> # Insert (a copy of) itself at the beginning
>>> a[:0] = a
>>> a
[123, 'bletch', 'xyzzy', 1234, 123, 'bletch', 'xyzzy',
     1234]
>>> # Clear the list: replace all items with an empty
     list
>>> a[:] = []
>>> a
[]
```

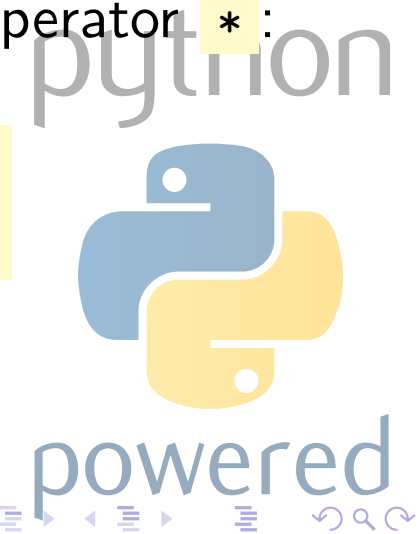
Lists

Concatenation of two or more lists can be done with the operator `+`:

```
>>> list1 = ['a','b','c']
>>> list2 = ['d','e','f']
>>> list3 = ['g','h','i']
>>> list_tot = list1 + list2 + list3
>>> list_tot
['a','b','c','d','e','f','g','h','i']
```

Iterative concatenation is also possible with the operator `*`:

```
>>> list1 = ['a','b','c']
>>> list_tot = list1 * 3
>>> list_tot
['a','b','c','a','b','c','a','b','c']
```



Lists

Another way to add an element at the end of a list is with the method `append()`:

```
>>> list1 = [1,2]
>>> list1.append(3)
>>> list1
[1,2,3]
```

Lists can be sorted with `sort()` and `reverse()`:

```
>>> list1 = ['f','b','r','k']
>>> list1.sort()
>>> list1
['b','f','k','r']
>>> list1.reverse()
>>> list1
['r','k','f','b']
```

. . . try to sort non-homogeous lists . . .

Lists

The index of an element can be retrieved with the `index()` method:

```
>>> list1 = ['f','b','r','k']
>>> list1.index('r')
2
```

. . . try to look for a non-existing element . . .

To delete a single element you can also use the statement `del`

```
>>> list1 = ['r','k','f','b']
>>> del list1[2]
>>> list1
['r','k','b']
```

List methods

The list data type has some more methods. Here are all of the methods of list objects:

`list.append(x)` Add an item to the end of the list; equivalent to `a[len(a):] = [x]`.

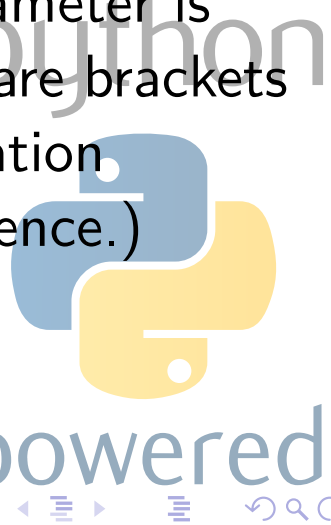
`list.extend(L)` Extend the list by appending all the items in the given list; equivalent to a[len(a):] = L.

`list.insert(i, x)` Insert an item at a given position. The first argument is the index of the element before which to insert, so `a.insert(0, x)` inserts at the front of the list, and `a.insert(len(a), x)` is equivalent to `a.append(x)`.

List methods

`list.remove(x)` Remove the first item from the list whose value is x. It is an error if there is no such item.

`list.pop([i])` Remove the item at the given position in the list, and return it. If no index is specified, `a.pop()` removes and returns the last item in the list. (The square brackets around the i in the method signature denote that the parameter is optional, not that you should type square brackets at that position. You will see this notation frequently in the Python Library Reference.)



List methods

`list.index(x)` Return the index in the list of the first item whose value is x. It is an error if there is no such item.

`list.count(x)` Return the number of times x appears in the list.

`list.sort()` Sort the items of the list, in place.

`list.reverse()` Reverse the elements of the list, in place.

Tuples

What is a tuple

It is a non-mutable container of iterable items, this means that cannot be dynamically changed once created (but can be redefined).

They are used when the content have not to be changed. They are very similar to lists, with the only difference that they are enclosed by round bracket.

```
>>> t1 = (1, 'a')
>>> t2 = (2, 'b')
>>> print t1[0]
1
>>> print t1*2
(1, 'a', 1, 'a')
>>> len(t1)
2
>>> t3 = t1 + t2
>>> print t3[1:3]
('a', 2)
```

Tuples: constructor

Tuples can be created by placing a comma-separated list of elememt pairs within round braces or by the `tuple` constructor.

class tuple([iterable])

Return a tuple whose items are the same and in the same order as iterable's items. iterable may be a sequence, a container that supports iteration, or an iterator object. If iterable is already a tuple, it is returned unchanged. For instance, `tuple('abc')` returns `('a', 'b', 'c')` and `tuple([1, 2, 3])` returns `(1, 2, 3)`. If no argument is given, returns a new empty tuple, `()`.

Tuples support the same operators of the list, apart those that would modify them.

Tuples: the zip function

`zip([iterable, ...])` returns a list of tuples, where the i-th tuple contains the i-th element from each of the argument sequences or iterables. The returned list is truncated in length to the length of the shortest argument sequence. When there are multiple arguments which are all of the same length. With no arguments, it returns an empty list. The left-to-right evaluation order of the iterables is guaranteed. This makes possible an idiom for clustering a data series into n-length groups using `zip(*[iter(s)]*n)` . `zip()` in conjunction with the `*` operator can be used to unzip a list:

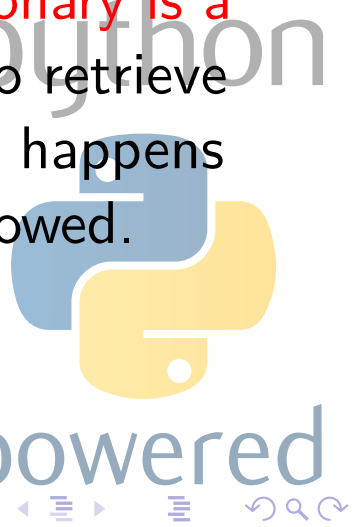
```
>>> x = [1, 2, 3]
>>> y = [4, 5, 6]
>>> zipped = zip(x, y)
>>> zipped
[(1, 4), (2, 5), (3, 6)]
>>> x2, y2 = zip(*zipped)
>>> x == list(x2) and y == list(y2)
True
```

Dictionaries

What is a dictionary

A dictionary is an unsorted collection of elements: it is an associative array, where arbitrary unique-defined keys are mapped to values.

A value is identified by a key (usually a string) instead of a numerical index, like in list. Each element of a dictionary is a couple (key : value), in which the key is necessary to retrieve the value (like in a "vocabulary"). Similarly to what happens for lists, non-homogeous element dictionaries are allowed.



Dictionaries

A (university) student's record book can be represented as a dictionary:

```
>>> diz1 = {'Analisi 1' : 25,'Analisi 2' : 23,'Metodi e
     Modelli matematici' : 12}
>>> diz1
{'Metodi e Modelli matematici': 12, 'Analisi 2': 23,
     'Analisi 1': 25}
>>> len(diz1)
3
```

The empty dictionary is represented by `{}`:

```
>>> diz2 = {}
>>> len(diz2)
0
```

Dictionaries

Another useful fuction is `key in d` that returns true [false] if key x does [not] exist:

```
>>> 'Analisi 2' in diz1
True
>>> diz1.has_key('Analisi 2') #DEPRECATED
True
>>> 'Geometria' in diz1
False
```

Insertion can be easily achieved:

```
>>> diz1['Geometria'] = 18
>>> diz1
{'Geometria': 18, 'Metodi e Modelli matematici': 12,
     'Analisi 2': 23, 'Analisi 1': 25}
```

Dictionaries

This is one of the most powerful build-in type of Python: it is usable, efficient and flexible and can be very useful for DBMS, especially if combined with lists:

```
>>> db=[]
>>> surnames = {'0001':'Goose', '0002':'Mouse',
      '0003':'Duck'}
>>> names = {'0001':'Goofy', '0002':'Mickey',
      '0003':'Donald'}
>>> marks = {'0001':'A', '0002':'B--', '0003':'A+'}
>>> db.append(names)
>>> db.append(surnames)
>> db.append(marks)
>>> db
[{'0001': 'A', '0002': 'B--', '0003': 'A+'}, {'0001':
      'Goofy', '0002': 'Mickey', '0003': 'Donald'},
      {'0001': 'Goose', '0002': 'Mouse', '0003': 'Duck'}]
```

This list is made of 3 elements, each of which is a dictionary. Note that, internally, the dictionary are implemented as hash-table!!!.

Dictionaries

Note how easy is to build the table of scores:

```
>>> tabscore = {}
>>> for matricola in db[1].keys():
...     tabscore[db[1][matricola] + ' ' +
     db[2][matricola]]=db[0][matricola]
...     print db[1][matricola] + ' ' +
     db[2][matricola]+' score is '+ db[0][matricola]
...
Goofy Goose score is A
Mickey Mouse score is B--
Donald Duck score is A+
>>> tabscore
{'Goofy Goose': 'A', 'Mickey Mouse': 'B--', 'Donald
     Duck': 'A+'}
```

Dictionaries: constructor

Dictionaries can be created by placing a comma-separated list of `key: value` pairs within braces or by the `dict` constructor.

class dict([arg])

Return a new dictionary initialized from an optional positional argument or from a set of keyword arguments. If no arguments are given, return a new empty dictionary. If the positional argument arg is a mapping object, return a dictionary mapping the same keys to the same values as does the mapping object. Otherwise the positional argument must be a sequence, a container that supports iteration, or an iterator object.

Dictionary: constructor

The elements of the argument must each also be of one of those kinds, and each must in turn contain exactly two objects. The first is used as a key in the new dictionary, and the second as the key's value. If a given key is seen more than once, the last value associated with it is retained in the new dictionary. Example:

- `dict(ciro=1, luca=2)`
- `dict('ciro': 1, 'luca': 2)`
- `dict(zip(('ciro', 'luca'), (1, 2)))` (see later on)
- `dict([['luca', 2], ['ciro', 1]])`

these all return a dictionary equal to "ciro": 1, "luca": 2. The first example only works for keys that are valid Python identifiers; the others work with any valid keys.

Dictionary: operators

These are the operations that dictionaries support:

`len(d)`	Return the number of items in the dictionary d.
`d[key]`	Return the item of d with key key. Raises a `KeyError` if key is not in the map.
`d[key] = value`	Set d[key] to value.
`del d[key]`	Remove d[key] from d. Raises a `KeyError` if key is not in the map.
`key in d`	Return True if d has a key key, else False.
`key not in d`	Equivalent to not key in d.
`iter(d)`	Return an iterator over the keys of the dictionary. This is a shortcut for iterkeys().
`clear()`	Remove all items from the dictionary.
`copy()`	Return a shallow copy of the dictionary.

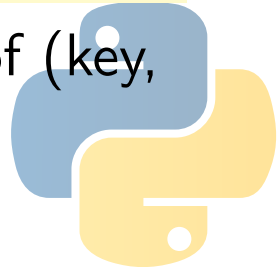
Dictionary: operators

`fromkeys(seq[, value])` Create a new dictionary with keys from seq and values set to value.

`get(key[, default])` Return the value for key if key is in the dictionary, else default. If default is not given, it defaults to None, so that this method never raises a `KeyError`.

`has_key(key)` Test for the presence of key in the dictionary. `has_key()` is deprecated in favor of `key in d`.

`items()` Return a copy of the dictionary's list of (key, value) pairs.



Dictionary: operators

`iteritems()` Return an iterator over the dictionary's (key, value) pairs.

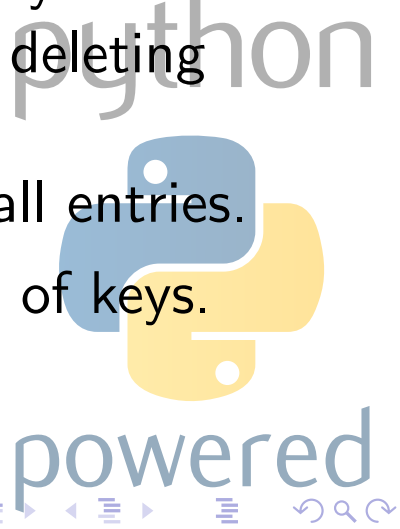
`iterkeys()` Return an iterator over the dictionary's keys.

Using `iterkeys()` while adding or deleting entries in the dictionary may raise a `RuntimeError` or fail to iterate over all entries.

`itervalues()` Return an iterator over the dictionary's values.

Using `itervalues()` while adding or deleting entries in the dictionary may raise a `RuntimeError` or fail to iterate over all entries.

`keys()` Return a copy of the dictionary's list of keys.

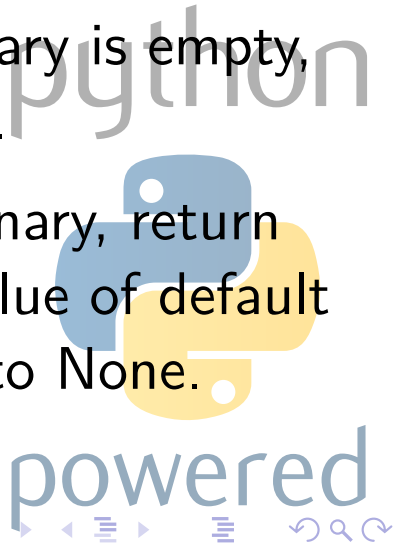


Dictionary: operators

`pop(key[, default])` If key is in the dictionary, remove it and return its value, else return default. If default is not given and key is not in the dictionary, a `KeyError` is raised.

`popitem()` Remove and return an arbitrary (key, value) pair from the dictionary. `popitem()` is useful to destructively iterate over a dictionary, as often used in set algorithms. If the dictionary is empty, calling `popitem()` raises a `KeyError`.

`setdefault(key[, default])` If key is in the dictionary, return its value. If not, insert key with a value of default and return default. default defaults to None.



Dictionary: operators

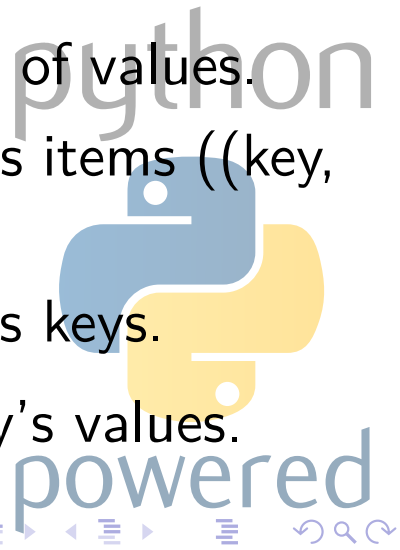
`update([other])` Update the dictionary with the key/value pairs from other, overwriting existing keys. Return None. `update()` accepts either another dictionary object or an iterable of key/value pairs (as tuples or other iterables of length two). If keyword arguments are specified, the dictionary is then updated with those key/value pairs: `d.update(red=1, blue=2)`.

`values()` Return a copy of the dictionary's list of values.

`viewitems()` Return a new view of the dictionary's items ((key, value) pairs).

`viewkeys()` Return a new view of the dictionary's keys.

`viewvalues()` Return a new view of the dictionary's values.



Control Flows

What is a control flow

Within an imperative programming language, a control flow statement is a statement whose execution results in a choice being made as to which of two or more paths should be followed (CC go to Wikipedia©).

In particular, Python knows the usual control flow statements known from other languages, with some twists plus others.

if statement

Perhaps the most well-known statement type is the if statement. For example:

```
>>> x = int(input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...     x = 0
...     print 'Negative changed to zero'
... elif x == 0:
...     print 'Zero'
... elif x == 1:
...     print 'Single'
... else:
...     print 'More'
...
More
```

There can be zero or more elif parts, and the else part is optional.

for statement

The for statement in Python is different from what you may be used to in C or in Pascal. Rather than always iterating over an arithmetic progression of numbers (like in Pascal), or giving the user the ability to define both the iteration step and halting condition (as C), Python's for statement iterates over the items of any sequence (a list or a string), in the order that they appear in the sequence. For example:

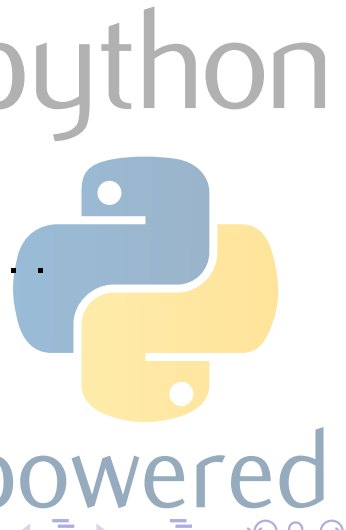
```
>>> # Measure some strings:
... a = ['pippo', 'pluto', 'paperino']
>>> for x in a:
...     print x, len(x)
...
pippo 5
pluto 5
paperino 8
```

for statement

It is not recommended to modify the mutable sequence being iterated within the loop. If you need to modify the list you are iterating over (for example, to duplicate selected items) you must iterate over a copy. The slice notation makes this particularly convenient:

```
>>> for x in a[:]: # make a slice copy of the entire list
...     if len(x) > 6: a.insert(0, x)
...
>>> a
['pippo', 'pluto', 'paperino', 'paperino']
```

Try to execute the code above over the original list. . .



while statement

The while loop executes as long as the condition remains true. In Python, like in C, any non-zero integer value is true; zero is false. The condition may also be a string or list value (any iterable type); anything with a non-zero length is true, empty sequences are false.

```
>>> # Fibonacci series:
... # the sum of two elements defines the next
... a, b = 0, 1
>>> while b < 10:
...     print b
...     a, b = b, a+b
...
1
1
2
3
5
8
```

A useful function: range()

If you do need to iterate over a sequence of numbers, the built-in function `range()` comes in handy. It generates lists containing arithmetic progressions:

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

The given end point is never part of the generated list; range(10) generates a list of 10 values, the legal indices for items of a sequence of length 10. It is possible to let the range start at another number, or to specify a different increment (even negative; sometimes this is called the âĂŸstepâĂŹ):

```
>>> range(5, 10)
[5, 6, 7, 8, 9]
>>> range(0, 10, 3)
[0, 3, 6, 9]
>>> range(-10, -100, -30)
[-10, -40, -70]
```

A useful function: range()

To iterate over the indices of a sequence, you can combine range() and len() as follows:

```
>>> a = ['Python', 'is', 'a', 'cool', 'tool']
>>> for i in range(len(a)):
...     print i, a[i]
...
0 Python
1 is
2 a
3 cool
4 tool
```

break and continue Statements, and else Clauses on Loops

break and continue

The break statement, like in C, breaks out of the smallest enclosing for or while loop. The continue statement, also borrowed from C, continues with the next iteration of the loop.

else on loops

Loop statements may have an else clause; it is executed when the loop terminates through exhaustion of the list (with for) or when the condition becomes false (with while), but not when the loop is terminated by a break statement.

break and continue Statements, and else Clauses on Loops

An example:

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print n, 'equals', x, '*', n/x
...             break
...     else:
...         # loop fell through without finding a factor
...         print n, 'is a prime number'
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

Credit

Credit goes to www.python.org and herein contents.