

### Identifier classes

<code>__*</code>	System-defined names
<code>_*</code>	Module private name, not imported via <code>import *</code>
<code>__*</code>	Class-private name

### Tuple

<code>( obj1, obj2, ... )</code>	tuple literal (parenthesis are optional, at least one comma is required)
<code>( obj1 , )</code>	singleton, tuple with only one object (parenthesis are optional)
<code>( )</code>	empty tuple

A tuple is an immutable sequence of objects

### List

<code>[ obj1, obj2, ... ]</code>	list literal
<code>[ obj1 ]</code>	single item list
<code>[ ]</code>	empty list

A list is a mutable sequence of objects

### Slices

<code>s[i]</code>	Item at index <i>i</i> of sequence <i>s</i> (not a slice)
<code>s[i:j]</code>	Items <i>i</i> to <i>j</i> -1 of <i>s</i>
<code>s[i:j:k]</code>	every <i>k</i> <sup>th</sup> item starting at index <i>i</i> and not including index <i>j</i> or higher
<code>s[:]</code>	" <code>s[:]</code> " is a copy of sequence <i>s</i> , while " <i>s</i> " is a reference to the same sequence

A Slice is a part of a sequence

Indexes start at 0 for the first item

Negative indexes start at the end of the sequence (eg. -1 is last element, -2 second last,...)

### Dictionaries

<code>{ key1: itm1, key2: itm2, ... }</code>	Dictionary of items referenced by key
<code>d[key]</code>	return item referenced by given key
<code>d[key] = x</code>	Add (or replaces) given key with attached item <i>x</i>
<code>del d[key]</code>	Remove <code>d[key]</code> from <i>d</i> . Raises <code>KeyError</code> if key not present in <i>d</i>
<code>key in d</code>	returns <code>True</code> if key is in dictionary
<code>key not in d</code>	returns <code>False</code> if key is in dictionary
<code>d.items()</code>	returns a list of key,value pairs

### Dictionaries (cont)

<code>d.keys()</code>	returns list of keys in dictionary
<code>d.values()</code>	returns a list of the dictionaries values
<code>d.copy()</code>	shallow copy of the dictionary (only top level objects are copies, any object remains is a reference to the "old" object)
<code>d.deepcopy()</code>	deep copy of the dictionary
<code>d.get(key[, default])</code>	returns <code>d[key]</code> if the key is present, or default if not. If default is not passed, <code>None</code> is returned
<code>d.pop(key[, default])</code>	same as <code>get()</code> except that the key,value pair is also removed from the dictionary
<code>d.popitem()</code>	pops an arbitrary (key, value) pair from the dictionary.
<code>d.setdefault(key[, default])</code>	returns <code>d[key]</code> if it exists, otherwise does <code>d[key]=default</code> and returns <code>default</code> as well. <code>default</code> defaults to <code>None</code>
<code>viewitems()</code>	returns dictionary view object containing either (key,value)
<code>viewkeys()</code>	pairs, only keys or only values
<code>viewvalues()</code>	

1) Keys are unique

2) If the key is a string it needs to be quoted

3) for loop "for k in dict" returns the keys, not the item

### (Im)mutable sequences

<code>x in s</code>	<code>True</code> if an item of <i>s</i> is equal to <i>x</i>
<code>x not in s</code>	<code>False</code> if an item of <i>s</i> is equal to <i>x</i>
<code>len(s)</code>	number of items in <i>s</i>
<code>min(s)</code>	smallest item of <i>s</i>
<code>max(s)</code>	largest item if <i>s</i>
<code>s.index(x)</code>	first index of item equal to <i>x</i> in <i>s</i>
<code>s.count(x)</code>	number of occurrences of <i>x</i> in <i>s</i>

These operations apply to mutable and immutable sequences

Sequence indexes start at 0 (first element), negative indexes start at the end (index -1 is last element, -2 second last,...)

*s*: sequence

*x*: object/item



### Mutable sequences

<code>s[i] = x</code>	replace item <i>i</i> of sequence <i>s</i> by <i>x</i>
<code>s[i:j] = x</code>	replace items <i>i</i> to <i>j</i> -1 (slice) by <i>x</i>
<code>s[i:j:k] = x</code>	replace each <i>k</i> <sup>th</sup> item of <i>s</i> starting at <i>i</i> and not including <i>j</i> or higher by <i>x</i>
<code>del &lt;slice&gt;</code>	delete the specified slice from the sequence
<code>s.append(x)</code>	appends object <i>x</i> to the end of the sequence <i>s</i>
<code>s.extend(s2)</code>	appends items of sequence <i>s2</i> to <i>s</i>
<code>s.insert(i, x)</code>	inserts object <i>x</i> before the current item at index <i>i</i>
<code>s.pop(i)</code>	Removes item <i>i</i> from the sequence and returns it. If <i>i</i> is omitted <i>i</i> =-1 is assumed
<code>s.remove(x)</code>	same as <code>del s[s.index(x)]</code>
<code>s.reverse()</code>	reverse the order of the items in <i>s</i> in place
<code>s.sort([ cmp [, key [, reverse]])</code>	sort elements in place (1)

- (1) `cmp` is optional comparison function with 2 arguments (the items to be compared) (i.e. `func(a,b)`) and returning -1 if `a<b`, 0 if `a==b` and 1 if `a>b`
- (2) `key` is an optional function taking one argument (item from the sequence) and used to extract the actual information from that item to be compared
- (3) `reverse`, if `True`, causes the sort result to have reverse order
- (4) in general specifying `key` and `reverse` (if possible) is much faster than the equivalent `cmp` function

### Classes

```
class <classname>:(1)
  <static-var>=<value> (2)
  def __init__(self,<args>...): (3)
    <class constructor>
    self.<attribute>=<value> (4)
  def method(self,<args>,...): (5)
    <method body>
  def __privateMethod(self, <args>...): (6)
    <method body>
```

- (1) Start definition of class
- (2) Static variable, same value for all instances and referenced via `<classname>.<varname>` rather than via `self`
- (3) Class constructor called via `<classname>()` with optional arguments
- (4) Attribute local to the instance
- (5) Class method definition. Each and every method in a class has at least one argument, "self" referring the class instance
- (6) private methods are not intended to be used from outside the class and are marked by two leading underscores

### Special class methods

<code>__init__(self[, ...])</code>	Called for any new instance creation in order to initialize it. Derived classes must call their base class <code>__init__</code> explicitly via <code>&lt;baseclass&gt;.__init__(self,...)</code> where <code>self</code> needs to be explicitly passed
<code>__del__(self)</code>	Called when the instance is to be destroyed
<code>__repr__(self)</code>	Called by the <code>repr()</code> function to return a string containing the classes representation. Usually used for debugging. If this method is defined and <code>__str__</code> is not, this method is also used for string conversions
<code>__str__(self)</code>	called by the <code>str()</code> function and by "print"
<code>__lt__(self,other)</code>	Rich comparison methods returning <code>True</code> if <code>self&lt;other</code> , <code>self&lt;=other</code> , <code>self==other</code> , <code>self!=other</code> .
<code>__le__(self,other)</code>	There is no defined relationship, i.e. <code>x==y</code> <code>True</code> does not imply that <code>x!=y</code> is <code>False</code>
<code>__eq__(self,other)</code>	
<code>__ne__(self,other)</code>	
<code>__gt__(self,other)</code>	
<code>__ge__(self,other)</code>	