Python Coding and Comment Style, Pydoc (PEP 8 and pydoc style):

#Summary from PEP 8:

- Use 4-space indentation, and no tabs.
  4 spaces are a good compromise between small indentation (allows greater nesting depth) and large indentation (easier to read). Tabs introduce confusion, and are best left out.

- Wrap lines so that they don't exceed 79 characters.
  This helps users with small displays and makes it possible to have several code files side-by-side on larger displays.

- Use blank lines to separate functions and classes, and larger blocks of code inside functions.

- When possible, put comments on a line of their own.

- Use docstrings.

- Use spaces around operators and after commas, but not directly inside bracketing constructs: a = f(1, 2) + g(3, 4).

- Name your classes and functions consistently; the convention is to use CamelCase for classes and lower_case_with_underscores for functions and methods. Always useself as the name for the first method argument.

- Don't use fancy encodings if your code is meant to be used in international environments. Plain ASCII works best in any case.

#Functions and Methods:

A function must have a docstring, unless it meets all of the following criteria:

- not externally visible
- very short
- obvious

A docstring should give enough information to write a call to the function without reading the function's code. A docstring should describe the function's calling syntax and its semantics, not its implementation. For tricky code, comments alongside the code are more appropriate than using docstrings.

Certain aspects of a function should be documented in special sections, listed below. Each section begins with a heading line, which ends with a colon. Sections should be indented two spaces, except for the heading.

Args:
: List each parameter by name. A description should follow the name, and be separated by a colon and a space. If the description is too long to fit on a single 80-character line, use a hanging indent of 2 or 4 spaces (be consistent with the rest of the file).

  The description should mention required type(s) and the meaning of the argument.

  If a function accepts *foo (variable length argument lists) and/or **bar (arbitrary keyword arguments), they should be listed as *foo and **bar.

Returns: (or Yields: for generators)
: Describe the type and semantics of the return value. If the function only returns None, this section is not required.

Raises:
: List all exceptions that are relevant to the interface.

```
def fetch_bigtable_rows(big_table, keys, other_silly_variable=None):
        """Fetches rows from a Bigtable.

        Retrieves rows pertaining to the given keys from the Table
        instance     represented by big_table.  Silly things may happen
        if     other_silly_variable is not None.

        Args:
                big_table: An open Bigtable Table instance.
                keys: A sequence of strings representing the key of each
                        table r              to fetch.
                other_silly_variable: Another optional variable, that
                        has a much
                        longer name than the other args, and which does
                        nothing.

        Returns:
                A dict mapping keys to the corresponding table row data
                fetched. Each row is represented as a tuple of strings.
                Raises:
        IOError: An error occurred accessing the bigtable.Table object.
        """
        pass
```

#Classes

Classes should have a doc string below the class definition describing the class. If your class has public attributes, they should be documented here in an Attributes section and follow the same formatting as a function's Args section.

```
class SampleClass(object):
    """Summary of class here.

    Longer class information....
    Longer class information....

    Attributes:
        likes_spam: A boolean indicating if we like SPAM or not.
        eggs: An integer count of the eggs we have laid.
    """

    def __init__(self, likes_spam=False):
        """Inits SampleClass with blah."""
        self.likes_spam = likes_spam
        self.eggs = 0

    def public_method(self):
        """Performs operation blah."""
```

#Block and Inline Comments

The final place to have comments is in tricky parts of the code. If you're going to have to explain it at the next code review, you should comment it now. Complicated operations get a few lines of comments before the operations commence. Non-obvious ones get comments at the end of the line.

```
# We use a weighted dictionary search to find out where i is in
# the array.  We extrapolate position based on the largest num
# in the array and the array size and then do binary search to
# get the exact number.

if i & (i-1) == 0:        # true iff i is a power of 2
```

To improve legibility, these comments should be at least 2 spaces away from the code.

On the other hand, never describe the code. Assume the person reading the code knows Python (though not what you're trying to do) better than you do.

#TODO Comment

Use `TODO` comments for code that is temporary, a short-term solution, or good-enough but not perfect.

`TODO`s should include the string `TODO` in all caps, followed by the name, e-mail address, or other identifier of the person who can best provide context about the problem referenced by the `TODO`, in parentheses. A colon is optional. A comment explaining what there is to do is required. The main purpose is to have a consistent `TODO` format that can be searched to find the person who can provide more details upon request. A `TODO` is not a commitment that the person referenced will fix the problem. Thus when you create a `TODO`, it is almost always your name that is given.

```
# TODO(kl@gmail.com): Use a "*" here for string repetition.
# TODO(Zeke) Change this to use relations.
```

If your `TODO` is of the form "At a future date do something" make sure that you either include a very specific date ("Fix by November 2009") or a very specific event ("Remove this code when all clients can handle XML responses.").

#Python Pydoc module

The **pydoc** module automatically generates documentation from Python modules. The documentation can be presented as pages of text on the console, served to a Web browser, or saved to HTML files.
For modules, classes, functions and methods, the displayed documentation is derived from the docstring (i.e. the **__doc__** attribute) of the object, and recursively of its documentable members. If there is no docstring, **pydoc** tries to obtain a description from the block of comment lines just above the definition of the class, function or method in the source file, or at the top of the module.

pydoc <name>…

The argument to **pydoc** can be the name of a function, module, or package, or a dotted reference to a class, method, or function within a module or module in a package.

pydoc -w <name>…

Specifying a `-w` flag before the argument will cause HTML documentation to be written out to a file in the current directory, instead of displaying text on the console.

Python Unit test (PyUnit)

PyUnit template (Also in my github/Misc):

```
import unittest
from class_practice import Employee # code from module you're testing


class SimpleTestCase(unittest.TestCase):

        def setUp(self):
                """Call before every test case."""
                self.emp = Employee('Lin',10000)
                self.emp2 = Employee('Jun',20000)

        def tearDown(self):
                """Call after every test case."""
                del self.emp
                del self.emp2

        def testGetName(self):
                """Test case A. note that all test method names must begin with 'test.'"""
                self.assertEqual(self.emp.getName(),'Lin')  # test getName() whether return correct answer"
                self.assertNotEqual(self.emp2.getName(),'Lin')

        def testGetSalary(self):
                """test case B"""
                self.assertEqual(self.emp2.getSalary(),20000)   # test getSalary() whether return correct answer
                self.assertNotEqual(self.emp.getSalary(),20000)

class OtherTestCase(unittest.TestCase):

        def setUp(self):
                self.emp3 = Employee('jiang',30000)

        def tearDown(self):
                del self.emp3

        def testGetEmpCount(self):
                """ getEmpCount() is a static method """
                self.assertEqual(Employee.getEmpCount(),1)    # test getEmpCount() whether return correct answer

        def testNothing(self):
                pass


if __name__ == "__main__":
        unittest.main() # run all tests
```

 # List of assert methods

Method	Checks that	New in
`assertEqual(a, b)`	`a == b`	
`assertNotEqual(a, b)`	`a != b`	
`assertTrue(x)`	`bool(x) is True`	
`assertFalse(x)`	`bool(x) is False`	
`assertIs(a, b)`	`a is b`	2.7
`assertIsNot(a, b)`	`a is not b`	2.7
`assertIsNone(x)`	`x is None`	2.7
`assertIsNotNone(x)`	`x is not None`	2.7
`assertIn(a, b)`	`a in b`	2.7
`assertNotIn(a, b)`	`a not in b`	2.7
`assertIsInstance(a, b)`	`isinstance(a, b)`	2.7
`assertNotIsInstance(a, b)`	`not isinstance(a, b)`	2.7

Python Data Structures

1. list

Internally, a list is represented as a **dynamic array** with iterator design pattern. If you need to add/remove at both ends, consider using a collections.deque instead.

Advantage: constant time random access O (1)

Disadvantage: Insertion or Deletion in the beginning or middle takes O(n), because everything must move. Also when the array grows beyond the current allocation size (because everything must move).

Operation	Average Case	Amortized Worst Case
Copy	O(n)	O(n)
Append[1]	O(1)	O(1)
Insert	O(n)	O(n)
Get Item	O(1)	O(1)
Set Item	O(1)	O(1)
Delete Item	O(n)	O(n)
Iteration	O(n)	O(n)
Get Slice	O(k)	O(k)
Del Slice	O(n)	O(n)
Set Slice	O(k+n)	O(k+n)
Extend[1]	O(k)	O(k)
Sort	O(n log n)	O(n log n)
Multiply	O(nk)	O(nk)
x in s	O(n)	
min(s), max(s)	O(n)	
Get Length	O(1)	O(1)

When to Use Lists

As shown in the examples above, lists are best used in the following situations:

- When you need a mixed collection of data all in one place.

- When the data needs to be ordered.
- When your data requires the ability to be changed or extended. Remember, lists are mutable.
- When you don't require data to be indexed by a custom value. Lists are numerically indexed and to retrieve an element, you must know its numeric position in the list.
- When you need a stack or a queue. Lists can be easily manipulated by appending/removing elements from the beginning/end of the list.
- When your data doesn't have to be unique. For that, you would use sets.

#List frequently throws IndexError and ValueError, which is helpful for Exception handling

#Common functions for list:

`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.`**`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.`**`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`**(*cmp=None*, *key=None*, *reverse=False*)
Sort the items of the list in place (the arguments can be used for sort customization, see **`sorted()`** for their explanation).

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

#Comparing sequence type

Sequence objects may be compared to other objects with the same sequence type. The comparison uses ***lexicographical*** ordering: first the first two items are compared, and if they differ this determines the outcome of the comparison; if they are equal, the next two items are compared, and so on, until either sequence is exhausted. If two items to be compared are themselves sequences of the same type, the lexicographical comparison is carried out recursively. If all items of two sequences compare equal, the sequences are considered equal. If one sequence is an initial sub-sequence of the other, the shorter sequence is the smaller (lesser) one. Lexicographical ordering for strings uses the ASCII ordering for individual characters. Some examples of comparisons between sequences of the same type:

```
(1, 2, 3)              < (1, 2, 4)
[1, 2, 3]              < [1, 2, 4]
'ABC' < 'C' < 'Pascal' < 'Python'
(1, 2, 3, 4)           < (1, 2, 4)
(1, 2)                 < (1, 2, -1)
(1, 2, 3)             == (1.0, 2.0, 3.0)
(1, 2, ('aa', 'ab'))   < (1, 2, ('abc', 'a'), 4)
```

Note that comparing objects of different types is legal. The outcome is deterministic but arbitrary: the types are ordered by their name. Thus, a list is always smaller than a string, a string is always smaller than a tuple, etc.

2. collections.deque (In interview, only use when you need the bounded property)

A deque (double-ended queue) is represented internally as a doubly linked list. (Well, a list of arrays rather than objects, for greater efficiency.) Both ends are accessible, but even looking at the middle is slow, and adding to or removing from the middle is slower still.

Operation	Average Case	Amortized Worst Case
Copy	O(n)	O(n)
append	O(1)	O(1)
appendleft	O(1)	O(1)
pop	O(1)	O(1)
popleft	O(1)	O(1)
extend	O(k)	O(k)
extendleft	O(k)	O(k)
rotate	O(k)	O(k)
remove	O(n)	O(n)

#Common methods and constructor for deque:

class `collections.`**`deque`**([*iterable*[, *maxlen*]])

> Returns a new deque object initialized left-to-right (using **`append()`**) with data from *iterable*. If *iterable* is not specified, the new deque is empty. If *maxlen* is not specified or is *None*, deques may grow to an arbitrary length. Otherwise, the deque is bounded to the specified maximum length. Once a bounded length deque is full, when new items are added, a corresponding number of items are discarded from the opposite end.

`append`(*x*)

> Add *x* to the right side of the deque.

`appendleft`(*x*)

> Add *x* to the left side of the deque.

`clear`()

> Remove all elements from the deque leaving it with length 0.

count(*x*)
Count the number of deque elements equal to *x*.

New in version 2.7.

extend(*iterable*)
Extend the right side of the deque by appending elements from the iterable argument.

extendleft(*iterable*)
Extend the left side of the deque by appending elements from *iterable*. Note, the series of left appends results in reversing the order of elements in the iterable argument.

pop()
Remove and return an element from the right side of the deque. If no elements are present, raises an **IndexError**.

popleft()
Remove and return an element from the left side of the deque. If no elements are present, raises an **IndexError**.

remove(*value*)
Removed the first occurrence of *value*. If not found, raises a **ValueError**.

reverse()
Reverse the elements of the deque in-place and then return `None`.

rotate(*n*)
Rotate the deque *n* steps to the right. If *n* is negative, rotate to the left. Rotating one step to the right is equivalent to:`d.appendleft(d.pop())`.

3. dict

Internally, dict is implemented as hash table. dict uses open addressing(random probing in pseudo random order) to resolve hash collision. The Average Case times listed for dict objects assume that the hash function for the objects is sufficiently robust to make collisions uncommon. The Average Case assumes the keys used in parameters are selected uniformly at random from the set of all keys.

Operation	Average Case	Amortized Worst Case
Copy[2]	O(n)	O(n)
Get Item	O(1)	O(n)
Set Item[1]	O(1)	O(n)
Delete Item	O(1)	O(n)
Iteration[2]	O(n)	O(n)

When to Use a Dictionary

- When you need a logical association between a `key:value` pair.
- When you need fast lookup for your data, based on a custom key.
- When your data is being constantly modified. Remember, dictionaries are mutable.

#Quick defaultdict using dict comprehension

Here's an example:

```
>>> {x:x*x for x in (1, 2, 3)}
{1: 1, 2: 4, 3: 9}
```

#KeyError, can be used in Exception handling

#dict object constructor(noted the use of zip())

To illustrate, the following examples all return a dictionary equal to `{"one": 1, "two": 2, "three": 3}`:

```
>>> a = dict(one=1, two=2, three=3)
>>> b = {'one': 1, 'two': 2, 'three': 3}
>>> c = dict(zip(['one', 'two', 'three'], [1, 2, 3]))
>>> d = dict([('two', 2), ('one', 1), ('three', 3)])
>>> e = dict({'three': 3, 'one': 1, 'two': 2})
>>> a == b == c == d == e
True
```

#Common functions for dict

`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`.

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

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

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

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**.

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`.

4. set

Internally implemented as a hashtable. This is similar to dict, therefore the membership checking is O(1).

Operation	Average case	Worst Case
x in s	O(1)	O(n)
Union s|t	O(len(s)+len(t))	
Intersection s&t	O(min(len(s), len(t))	O(len(s) * len(t))
Multiple intersection s1&s2&..&sn		(n-1)*O(l) where l is max(len(s1),..,len(sn))
Difference s-t	O(len(s))	
s.difference_update(t)	O(len(t))	
Symmetric Difference s^t	O(len(s))	O(len(s) * len(t))
s.symmetric_difference_update(t)	O(len(t))	O(len(t) * len(s))

A set is an unordered collection with no duplicate values. A set can be created by using the keyword `set` or by using curly braces `{}`. However, to create an empty set you can only use the `set` construct, curly braces alone will create an empty dictionary.

#When to Use Sets

- When you need a unique set of data: Sets check the unicity of elements based on hashes.
- When your data constantly changes: Sets, just like lists, are mutable.
- When you need a collection that can be manipulated mathematically: With sets it's easy to do operations like difference, union, intersection, etc.
- When you don't need to store nested lists, sets, or dictionaries in a data structure: Sets don't support unhashable types.

#set supports set comprehension(noted the difference with dict comprehension)

```
>>> vowels = ['a', 'e', 'i', 'o', 'u']
>>> {x for x in 'maintenance' if x not in vowels }
set(['c', 'm', 't', 'n'])
```

#set operations

Operation	Equivalent	Result
`len(s)`		cardinality of set *s*
`x in s`		test *x* for membership in *s*
`x not in s`		test *x* for non-membership in *s*
`s.issubset(t)`	`s <= t`	test whether every element in *s* is in *t*
`s.issuperset(t)`	`s >= t`	test whether every element in *t* is in *s*
`s.union(t)`	`s | t`	new set with elements from both *s* and *t*
`s.intersection(t)`	`s & t`	new set with elements common to *s* and *t*
`s.difference(t)`	`s - t`	new set with elements in *s* but not in *t*
`s.symmetric_difference(t)`	`s ^ t`	new set with elements in either *s* or *t* but not both
`s.copy()`		new set with a shallow copy of *s*

#set common methods

`len(s)`

Return the cardinality of set *s*.

`x in s`

Test *x* for membership in *s*.

`x not in s`

Test *x* for non-membership in *s*.

`isdisjoint`(*other*)

Return `True` if the set has no elements in common with *other*. Sets are disjoint if and only if their intersection is the empty set.

`issubset`(*other*)
`set <= other`

Test whether every element in the set is in *other*.

`set < other`

Test whether the set is a proper subset of *other*, that is, `set <= other and set != other`.

`issuperset`(*other*)
`set >= other`

Test whether every element in *other* is in the set.

`set > other`

Test whether the set is a proper superset of *other*, that is, `set >= other and set != other`.

union(*other*, *...*)
set | other | ...
Return a new set with elements from the set and all others.

intersection(*other*, *...*)
set & other & ...
Return a new set with elements common to the set and all others.

difference(*other*, *...*)
set - other - ...
Return a new set with elements in the set that are not in the others.

symmetric_difference(*other*)
set ^ other
Return a new set with elements in either the set or *other* but not both.

add(*elem*)
Add element *elem* to the set.

remove(*elem*)
Remove element *elem* from the set. Raises **KeyError** if *elem* is not contained in the set.

discard(*elem*)
Remove element *elem* from the set if it is present.

5. File object

#Constructor: open()
File object can be created by open() function

open(*name*[, *mode*[, *buffering*]])

Open a file, returning an object of the **file** type described in section *File Objects*. If the file cannot be opened, **IOError** is raised. The most commonly-used values of *mode* are 'r' for reading, 'w' for writing (truncating the file if it already exists), and 'a' for appending (which on *some* Unix systems means that *all* writes append to the end of the file regardless of the current seek position). If*mode* is omitted, it defaults to 'r'.

A typical open file workflow(only for reading data from file):

```
f = open("hello.txt")
try:
    for line in f:
        print line,
finally:
    f.close()
```

or

```
with open("hello.txt") as f:
    for line in f:
        print line,
```

file.**close**()

Close the file. A closed file cannot be read or written any more. Any operation which requires that the file be open will raise a **ValueError**after the file has been closed. Calling **close()** more than once is allowed.

file.**read**([*size*])

Read at most *size* bytes from the file (less if the read hits EOF before obtaining *size* bytes). If the *size* argument is negative or omitted, read all data until EOF is reached. The bytes are returned as a string object. An empty string is returned when EOF is encountered immediately.

file.**readline**([*size*])

Read one entire line from the file. A trailing newline character is kept in the string (but may be absent when a file ends with an incomplete line). [6] If the *size* argument is present and non-negative, it is a maximum byte count (including the trailing newline) and an incomplete line may be returned. When *size* is not 0, an empty string is returned *only* when EOF is encountered immediately.

`file.`**`seek`**(*offset*[, *whence*])

Set the file's current position, like `stdio`'s **`fseek()`**. The *whence* argument is optional and defaults to `os.SEEK_SET` or `0` (absolute file positioning); other values are `os.SEEK_CUR` or `1` (seek relative to the current position) and `os.SEEK_END` or `2` (seek relative to the file's end). There is no return value. For example, `f.seek(2, os.SEEK_CUR)` advances the position by two and `f.seek(-3, os.SEEK_END)` sets the position to the third to last.

`file.`**`tell`**()

Return the file's current position, like `stdio`'s **`ftell()`**.

`file.`**`write`**(*str*)

Write a string to the file. There is no return value. Due to buffering, the string may not actually show up in the file until the **`flush()`** or**`close()`** method is called.

6. heapq – (heap/priority queue)

*from heapq import **

To create a heap, use a list initialized to [], or you can transform a populated list into a min-heap via function heapify(). For max-heap, you need to negate the priority using a simple function lambda x: -x

Heap elements can tuples. This is useful for assigning comparison values(such as task priorities) alongside the main record being tracked:

```
>>> h = []
>>> heappush(h, (5, 'write code'))
>>> heappush(h, (7, 'release product'))
>>> heappush(h, (1, 'write spec'))
>>> heappush(h, (3, 'create tests'))
>>> heappop(h)
(1, 'write spec')
```

#Common functions about heapq:

The following functions are provided:

heapq.**heappush**(*heap*, *item*)

Push the value *item* onto the *heap*, maintaining the heap invariant.

heapq.**heappop**(*heap*)

Pop and return the smallest item from the *heap*, maintaining the heap invariant. If the heap is empty, **IndexError** is raised. To access the smallest item without popping it, use heap[0].

heapq.**heapify**(*x*)

Transform list *x* into a heap, in-place, in linear time.

heapq.**nlargest**(*n*, *iterable*[, *key*])

Return a list with the *n* largest elements from the dataset defined by *iterable*. *key*, if provided, specifies a function of one argument that is used to extract a comparison key from each element in the iterable: key=str.lower Equivalent to: sorted(iterable, key=key,reverse=True)[:n]

heapq.**nsmallest**(*n*, *iterable*[, *key*])

Return a list with the *n* smallest elements from the dataset defined by *iterable*. *key*, if provided, specifies a function of one argument that is used to extract a comparison key from each element in the iterable: `key=str.lower` Equivalent to: `sorted(iterable, key=key)[:n]`

Note: nlargest and nsmallest are generic function for all iterables.

Example: nlargest(2,list('AesfaW'),key=str.lower) => ['W','s']

7. Tuples

A typical pattern for entries is a tuple is in the form: (priority_number, data)

Tuples are immutable, but can hold mutable objects.

A tuple is represented by a number of values separated by commas. Unlike lists, tuples are immutable and the output is surrounded by parentheses so that nested tuples are processed correctly. Additionally, even though tuples are immutable, they can hold mutable data if needed.

When to Use Tuples

- When you need to store data that doesn't have to change.
- When the performance of the application is very important. In this situation you can use tuples whenever you have fixed data collections.
- When you want to store your data in logical immutable pairs, triples etc.
- Good for comparing immutable pairs.

Constructor and operation Example:

A tuple consists of a number of values separated by commas, for instance:

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Tuples may be nested:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
>>> # Tuples are immutable:
... t[0] = 88888
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> # but they can contain mutable objects:
... v = ([1, 2, 3], [3, 2, 1])
>>> v
([1, 2, 3], [3, 2, 1])
```

8. collections.Counter

A **Counter** is a **dict** subclass for counting hashable objects. It is an unordered collection where elements are stored as dictionary keys and their counts are stored as dictionary values. Counts are allowed to be any integer value including zero or negative counts. The**Counter** class is similar to bags or multisets in other languages.

 # A counter tool is provided to support convenient and rapid tallies.
For example:

```
>>> # Tally occurrences of words in a list
>>> cnt = Counter()
>>> for word in ['red', 'blue', 'red', 'green', 'blue', 'blue']:
...     cnt[word] += 1
>>> cnt
Counter({'blue': 3, 'red': 2, 'green': 1})

>>> # Find the ten most common words in Hamlet
>>> import re
>>> words = re.findall(r'\w+', open('hamlet.txt').read().lower())
>>> Counter(words).most_common(10)
[('the', 1143), ('and', 966), ('to', 762), ('of', 669), ('i', 631),
 ('you', 554),  ('a', 546), ('my', 514), ('hamlet', 471), ('in', 451)]
```

#Common Constructor

Elements are counted from an *iterable* or initialized from another *mapping* (or counter):

```
>>> c = Counter()                           # a new, empty counter
>>> c = Counter('gallahad')                 # a new counter from an iterable
>>> c = Counter({'red': 4, 'blue': 2})      # a new counter from a mapping
>>> c = Counter(cats=4, dogs=8)             # a new counter from keyword args
```

Counter objects have a dictionary interface except that they return a zero count for missing items instead of raising a `KeyError`:

```
>>> c = Counter(['eggs', 'ham'])
>>> c['bacon']                              # count of a missing element is zero
0
```

Setting a count to zero does not remove an element from a counter. Use `del` to remove it entirely:

```
>>> c['sausage'] = 0                        # counter entry with a zero count
>>> del c['sausage']                        # del actually removes the entry
```

#Common methods for Counter

`elements`()

Return an iterator over elements repeating each as many times as its count. Elements are returned in arbitrary order. If an element's count is less than one, **`elements()`** will ignore it.

```
>>> c = Counter(a=4, b=2, c=0, d=-2)
>>> list(c.elements())
['a', 'a', 'a', 'a', 'b', 'b']
```

`most_common`([*n*])

Return a list of the *n* most common elements and their counts from the most common to the least. If *n* is omitted or `None`,**`most_common()`** returns *all* elements in the counter. Elements with equal counts are ordered arbitrarily:

```
>>> Counter('abracadabra').most_common(3)
[('a', 5), ('r', 2), ('b', 2)]
```

`subtract`([*iterable-or-mapping*])

Elements are subtracted from an *iterable* or from another *mapping* (or counter). Like **`dict.update()`** but subtracts counts instead of replacing them. Both inputs and outputs may be zero or negative.

```
>>> c = Counter(a=4, b=2, c=0, d=-2)
>>> d = Counter(a=1, b=2, c=3, d=4)
>>> c.subtract(d)
>>> c Counter({'a': 3, 'b': 0, 'c': -3, 'd': -6})
```

#Common patterns for working with Counter object:

```
sum(c.values())                 # total of all counts
c.clear()                       # reset all counts
list(c)                         # list unique elements
set(c)                          # convert to a set
dict(c)                         # convert to a regular dictionary
c.items()                       # convert to a list of (elem, cnt) pairs
Counter(dict(list_of_pairs))    # convert from a list of (elem, cnt) pairs
c.most_common()[-1:-n-1:-1]       # n least common elements
c += Counter()                  # remove zero and negative counts
```

#Mathematical operations for Counter object:

Each operation can accept inputs with signed counts, but the output will exclude results with counts of zero or less.

```
>>> c = Counter(a=3, b=1)
>>> d = Counter(a=1, b=2)
>>> c + d                       # add two counters together:  c[x] + d[x]
Counter({'a': 4, 'b': 3})
>>> c - d                       # subtract (keeping only positive counts)
Counter({'a': 2})
>>> c & d                       # intersection:  min(c[x], d[x])
Counter({'a': 1, 'b': 1})
>>> c | d                       # union:  max(c[x], d[x])
Counter({'a': 3, 'b': 2})
```

9. collections.OrderedDict

Ordered dictionaries are just like regular dictionaries but they remember the order that items were inserted. When iterating over an ordered dictionary, the items are returned in the order their keys were first added.

#Common constructors and methods:

class `collections.`**`OrderedDict`**([*items*])

Return an instance of a dict subclass, supporting the usual **`dict`** methods. An *OrderedDict* is a dict that remembers the order that keys were first inserted. If a new entry overwrites an existing entry, the original insertion position is left unchanged. Deleting an entry and reinserting it will move it to the end.

`OrderedDict.`**`popitem`**(*last=True*)

The **`popitem()`** method for ordered dictionaries returns and removes a (key, value) pair. The pairs are returned in LIFO order if *last* is true or FIFO order if false.

#OrderDict recipes

Since an ordered dictionary remembers its insertion order, it can be used in conjuction with sorting to make a sorted dictionary:

```
>>> # regular unsorted dictionary
>>> d = {'banana': 3, 'apple':4, 'pear': 1, 'orange': 2}
>>> # dictionary sorted by key
>>> OrderedDict(sorted(d.items(), key=lambda t: t[0]))
OrderedDict([('apple', 4), ('banana', 3), ('orange', 2), ('pear', 1)])
>>> # dictionary sorted by value
>>> OrderedDict(sorted(d.items(), key=lambda t: t[1]))
OrderedDict([('pear', 1), ('orange', 2), ('banana', 3), ('apple', 4)])
>>> # dictionary sorted by length of the key string
>>> OrderedDict(sorted(d.items(), key=lambda t: len(t[0])))
OrderedDict([('pear', 1), ('apple', 4), ('orange', 2), ('banana', 3)])
```

An ordered dictionary can be combined with the **Counter** class so that the counter remembers the order elements are first encountered:

```
class OrderedCounter(Counter, OrderedDict):
     'Counter that remembers the order elements are first encountered'

     def __repr__(self):
         return '%s(%r)' % (self.__class__.__name__, OrderedDict(self))

     def __reduce__(self):
         return self.__class__, (OrderedDict(self),)
```

Python Built-in Functions

#Only list some built-in functions that maybe useful in some situations

range(*stop*)

range(*start*, *stop*[, *step*])

This is a versatile function to create lists containing arithmetic progressions. It is most often used in **for** loops. The arguments must be plain integers. If the *step* argument is omitted, it defaults to `1`. If the *start* argument is omitted, it defaults to `0`.

len(*s*)

Return the length (the number of items) of an object. The argument may be a sequence (such as a string, bytes, tuple, list, or range) or a collection (such as a dictionary, set, or frozen set).

abs(*x*)

Return the absolute value of a number. The argument may be a plain or long integer or a floating point number. If the argument is a complex number, its magnitude is returned.

max(*iterable*[, *key*])

max(*arg1*, *arg2*, **args*[, *key*])

Return the largest item in an iterable or the largest of two or more arguments.

The optional *key* argument specifies a one-argument ordering function like that used for **list.sort()**. The *key* argument, if supplied, must be in keyword form (for example, `max(a,b,c,key=func)`).

min(*iterable*[, *key*])

min(*arg1*, *arg2*, **args*[, *key*])

Return the smallest item in an iterable or the smallest of two or more arguments.

sum(*iterable*[, *start*])

Sums *start* and the items of an *iterable* from left to right and returns the total. *start* defaults to `0`. The *iterable*'s items are normally numbers, and the start value is not allowed to be a string.

For some use cases, there are good alternatives to **`sum()`**. The preferred, fast way to concatenate a sequence of strings is by calling `''.join(sequence)`. To add floating point values with extended precision, see **`math.fsum()`**.

pow(*x*, *y*[, *z*])

Return *x* to the power *y*; if *z* is present, return *x* to the power *y*, modulo *z* (computed more efficiently than `pow(x, y) % z`). The two-argument form `pow(x, y)` is equivalent to using the power operator: `x**y`.

round(*number*[, *ndigits*])

Return the floating point value *number* rounded to *ndigits* digits after the decimal point. If *ndigits* is omitted, it defaults to zero. The result is a floating point number. Values are rounded to the closest multiple of 10 to the power minus *ndigits*; if two multiples are equally close, rounding is done away from 0 (so, for example, `round(0.5)` is `1.0` and `round(-0.5)` is `-1.0`).

map(*function*, *iterable*, ...)

Apply *function* to every item of *iterable* and return a list of the results. If additional *iterable* arguments are passed, *function* must take that many arguments and is applied to the items from all iterables in parallel. If one iterable is shorter than another it is assumed to be extended with `None` items. If *function* is `None`, the identity function is assumed; if there are multiple arguments, **map()** returns a list consisting of tuples containing the corresponding items from all iterables (a kind of transpose operation). The *iterable* arguments may be a sequence or any iterable object; the result is always a list.

all(*iterable*)

Return `True` if all elements of the *iterable* are true (or if the iterable is empty).

any(*iterable*)

Return `True` if any element of the *iterable* is true. If the iterable is empty, return `False`.

bin(*x*)

Convert an integer number to a binary string. The result is a valid Python expression. If *x* is not a Python **int** object, it has to define an**__index__()** method that returns an integer.

class **int**(*x=0*)
class **int**(*x*, *base=10*)

Return an integer object constructed from a number or string *x*, or return `0` if no arguments are given. If *x* is a number, it can be a plain integer, a long integer, or a floating point number. If *x* is floating point, the conversion truncates towards zero. If the argument is outside the integer range, the function returns a long object instead.

If *x* is not a number or if *base* is given, then *x* must be a string or Unicode object representing an *integer literal* in radix *base*. Optionally, the literal can be preceded by `+` or `-` (with no space in between) and surrounded by whitespace. A base-n literal consists of the digits 0 to n-1, with `a` to `z` (or `A` to `Z`) having values 10 to 35. The default *base* is 10. The allowed values are 0 and 2-36. Base-2, -8, and -16 literals can be optionally prefixed with `0b`/`0B`, `0o`/`0O`/`0`, or `0x`/`0X`, as with integer literals in code. Base 0 means to interpret the string exactly as an integer literal, so that the actual base is 2, 8, 10, or 16.

class **float**([*x*])

Return a floating point number constructed from a number or string *x*.

If the argument is a string, it must contain a possibly signed decimal or floating point number, possibly embedded in whitespace. The argument may also be [+|-]nan or [+|-]inf. Otherwise, the argument may be a plain or long integer or a floating point number, and a floating point number with the same value (within Python's floating point precision) is returned. If no argument is given, returns `0.0`.

sorted(*iterable*[, *cmp*[, *key*[, *reverse*]]])

Return a new sorted list from the items in *iterable*.

The optional arguments *cmp*, *key*, and *reverse* have the same meaning as those for the **`list.sort()`** method

cmp specifies a custom comparison function of two arguments (iterable elements) which should return a negative, zero or positive number depending on whether the first argument is considered smaller than, equal

to, or larger than the second argument: `cmp=lambdax,y: cmp(x.lower(), y.lower())`. The default value is `None`.

key specifies a function of one argument that is used to extract a comparison key from each list element: `key=str.lower`. The default value is `None` (compare the elements directly).

reverse is a boolean value. If set to `True`, then the list elements are sorted as if each comparison were reversed.

reversed(*seq*)

Return a reverse *iterator*. *seq* must be an object which has a **__reversed__()** method or supports the sequence protocol (the **__len__()**method and the **__getitem__()** method with integer arguments starting at `0`).

cmp(*x*, *y*)

Compare the two objects *x* and *y* and return an integer according to the outcome. The return value is negative if `x < y`, zero if `x == y`and strictly positive if `x > y`.

chr(*i*)

Return a string of one character whose ASCII code is the integer *i*. For example, `chr(97)` returns the string `'a'`. This is the inverse of**ord()**. The argument must be in the range [0..255], inclusive; **ValueError** will be raised if *i* is outside that range.

ord(*c*)

Given a string of length one, return an integer representing the Unicode code point of the character when the argument is a unicode object, or the value of the byte when the argument is an 8-bit string. For example, `ord('a')` returns the integer `97`, `ord(u'\u2020')`returns `8224`. This is the inverse of **chr()** for 8-bit strings and of **unichr()** for unicode objects.

divmod(*a*, *b*)

Take two (non complex) numbers as arguments and return a pair of numbers consisting of their quotient and remainder when using long division. With mixed operand types, the rules for binary arithmetic operators apply. For plain and long integers, the result is the same as `(a // b, a % b)`. For floating point numbers the result is `(q, a % b)`,

where *q* is usually `math.floor(a / b)` but may be 1 less than that. In any case `q * b + a % b` is very close to *a*, if `a % b` is non-zero it has the same sign as *b*, and `0 <= abs(a % b) < abs(b)`.

enumerate(*sequence*, *start=0*)

Return an enumerate object. *sequence* must be a sequence, an *iterator*, or some other object which supports iteration. The **next()**method of the iterator returned by **enumerate()** returns a tuple containing a count (from *start* which defaults to 0) and the values obtained from iterating over *sequence*:

```
>>> seasons = ['Spring', 'Summer', 'Fall', 'Winter']
>>> list(enumerate(seasons))
[(0, 'Spring'), (1, 'Summer'), (2, 'Fall'), (3, 'Winter')]
>>> list(enumerate(seasons, start=1))
[(1, 'Spring'), (2, 'Summer'), (3, 'Fall'), (4, 'Winter')]
```

filter(*function*, *iterable*)

Construct a list from those elements of *iterable* for which *function* returns true. *iterable* may be either a sequence, a container which supports iteration, or an iterator. If *iterable* is a string or a tuple, the result also has that type; otherwise it is always a list. If *function* is`None`, the identity function is assumed, that is, all elements of *iterable* that are false are removed.

Note that `filter(function, iterable)` is equivalent to `[item for item in iterable if function(item)]` if function is not `None`.

reduce(*function*, *iterable*[, *initializer*])

Apply *function* of two arguments cumulatively to the items of *iterable*, from left to right, so as to reduce the iterable to a single value. For example, `reduce(lambda x, y: x+y, [1, 2, 3, 4, 5])` calculates `((((1+2)+3)+4)+5)`. The left argument, *x*, is the accumulated value and the right argument, *y*, is the update value from the *iterable*. If the optional *initializer* is present, it is placed before the items of the iterable in the calculation, and serves as a default when the iterable is empty.
If *initializer* is not given and *iterable* contains only one item, the first item is returned.

format(*value*[, *format_spec*])¶

Convert a *value* to a "formatted" representation, as controlled by *format_spec(which will be seen in the Python String section)*.

hash(*object*)

Return the hash value of the object (if it has one). Hash values are integers. They are used to quickly compare dictionary keys during a dictionary lookup. Numeric values that compare equal have the same hash value (even if they are of different types, as is the case for 1 and 1.0).

raw_input([*prompt*])

If the *prompt* argument is present, it is written to standard output without a trailing newline. The function then reads a line from input, converts it to a string (stripping a trailing newline), and returns that. When EOF is read, **EOFError** is raised. Example:

```
>>> s = raw_input('--> ')
--> Monty Python's Flying Circus
>>> s
"Monty Python's Flying Circus"
```

input([*prompt*])

Equivalent to `eval(raw_input(prompt))`.

isinstance(*object*, *classinfo*)

Return true if the *object* argument is an instance of the *classinfo* argument, or of a (direct, indirect or *virtual*) subclass thereof. Also return true if *classinfo* is a type object (new-style class) and *object* is an object of that type or of a (direct, indirect or *virtual*) subclass thereof.

issubclass(*class*, *classinfo*)

Return true if *class* is a subclass (direct, indirect or *virtual*) of *classinfo*. A class is considered a subclass of itself. *classinfo* may be a tuple of class objects, in which case every entry in *classinfo* will be checked. In any other case, a **TypeError** exception is raised.

getattr(*object*, *name*[, *default*])

Return the value of the named attribute of *object*. *name* must be a string. If the string is the name of one of the object's attributes, the result is the value of that attribute. For example, `getattr(x, 'foobar')` is

equivalent to `x.foobar`. If the named attribute does not exist, *default* is returned if provided, otherwise **`AttributeError`** is raised.

`hasattr`(*object*, *name*)

The arguments are an object and a string. The result is `True` if the string is the name of one of the object's attributes,`False` if not. (This is implemented by calling `getattr(object, name)` and seeing whether it raises an exception or not.)

`dir`([*object*])

Without arguments, return the list of names in the current local scope. With an argument, attempt to return a list of valid attributes and member functions for that object.

The default **`dir()`** mechanism behaves differently with different types of objects, as it attempts to produce the most relevant, rather than complete, information:

- If the object is a module object, the list contains the names of the module's attributes.
- If the object is a type or class object, the list contains the names of its attributes, and recursively of the attributes of its bases.
- Otherwise, the list contains the object's attributes' names, the names of its class's attributes, and recursively of the attributes of its class's base classes.

`zip`([*iterable*, *...*])

This function 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, **`zip()`** is similar to **`map()`** with an initial argument of `None`. With a single sequence argument, it returns a list of 1-tuples. 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
```

zip is also powerful to transpose a list matrix

```
>>> theArray = [['a','b','c'],['d','e','f'],['g','h','i']]
>>> zip(*theArray)
[('a', 'd', 'g'), ('b', 'e', 'h'), ('c', 'f', 'i')]
```

Python String and regular expression

#str.format(), string formatting syntax

Format strings contain “replacement fields” surrounded by curly braces `{}`. Anything that is not contained in braces is considered literal text, which is copied unchanged to the output. If you need to include a brace character in the literal text, it can be escaped by doubling: `{{` and`}}`.

The grammar for a replacement field is as follows:

```
replacement_field ::=  "{" [field_name] ["!" conversion] [":" format_spec] "}"
field_name        ::=  arg_name ("." attribute_name | "[" element_index "]")*
arg_name          ::=  [identifier | integer]
attribute_name    ::=  identifier
element_index     ::=  integer | index_string
index_string      ::=  <any source character except "]"> +
conversion        ::=  "r" | "s"
format_spec       ::=  <described in the next section>
```

//field_name

The *field_name* is optionally followed by a *conversion* field, which is preceded by an exclamation point `'!'`, and a *format_spec*, which is preceded by a colon `':'`. These specify a non-default format for the replacement value.

The *field_name* itself begins with an *arg_name* that is either a number or a keyword. If it’s a number, it refers to a positional argument, and if it’s a keyword, it refers to a named keyword argument. If the numerical arg_names in a format string are 0, 1, 2, ... in sequence, they can all be omitted (not just some) and the numbers 0, 1, 2, ... will be automatically inserted in that order. The *arg_name* can be followed by any number of index or attribute expressions. An expression of the form `'.name'` selects the named attribute using **`getattr()`**, while an expression of the form `'[index]'` does an index lookup using **`__getitem__()`**.

Some simple format string examples:

```
"First, thou shalt count to {0}" # References first positional argument
"Bring me a {}"        # Implicitly references the first positional argument
"From {} to {}"                  # Same as "From {0} to {1}"
"My quest is {name}"             # References keyword argument 'name'
"Weight in tons {0.weight}"      # 'weight' attribute of first positional arg
"Units destroyed: {players[0]}"  # First element of keyword argument 'players'.
```

//Conversion

Two conversion flags are currently supported: `'!s'` which calls `str()` on the value, and `'!r'` which calls `repr()`.

Some examples:

```
"Harold's a clever {0!s}"        # Calls str() on the argument first
"Bring out the holy {name!r}"    # Calls repr() on the argument first
```

Difference between str() and repr():

In short __repr__ goal is to be unambigous and __str__ is to be readable. The official Python documentation says __repr__ is used to compute the "official" string representation of an object and __str__ is used to compute the "informal" string representation of an object. The print statement and str() built-in function uses __str__ to display the string representation of the object while the repr()built-in function uses __repr__ to display the object.

Lets create a datetime object:

```
>>> import datetime
 >>> today = datetime.datetime.now()
>>> str(today)
'2012-03-14 09:21:58.130922'
>>> repr(today)
 'datetime.datetime(2012, 3, 14, 9, 21, 58, 130922)'
>>> eval('datetime.datetime(2012, 3, 14, 9, 21, 58, 130922)')
datetime.datetime(2012, 3, 14, 9, 21, 58, 130922)
```

Only str() and print will show the string representation, all others will use repr representation. If __str__() is missing, __repr__ will be used. Thus in a general every class you code must have a `__repr__` and if you think it would be useful to have a string version of the object, as in the case of `datetime`, create a `__str__` function.

//format_spec

The general form of a *standard format specifier* is:

```
format_spec ::=  [[fill]align][sign][#][0][width][,][.precision][type]
fill        ::=  <any character>
align       ::=  "<" | ">" | "=" | "^"
sign        ::=  "+" | "-" | " "
width       ::=  integer
precision   ::=  integer
```

```
type          ::=  "b" | "c" | "d" | "e" | "E" | "f" | "F" | "g" | "G" |
"n" | "o" | "s" | "x" | "X" | "%"
```

Instead of talking all details about specifer here, we list some examples, details can be found in 'https://docs.python.org/2/library/string.html#formatspec'

//Formatting Examples

Accessing arguments by position:

```
>>> '{0}, {1}, {2}'.format('a', 'b', 'c')
'a, b, c'
>>> '{}, {}, {}'.format('a', 'b', 'c')  # 2.7+ only
'a, b, c'
>>> '{2}, {1}, {0}'.format('a', 'b', 'c')
'c, b, a'
>>> '{2}, {1}, {0}'.format(*'abc')      # unpacking argument sequence
'c, b, a'
>>> '{0}{1}{0}'.format('abra', 'cad')   # arguments' indices can be repeated
'abracadabra'
```

Accessing arguments by name:

```
>>> 'Coordinates: {latitude}, {longitude}'.format(latitude='37.24N', longitude='-115.81W')
'Coordinates: 37.24N, -115.81W'
>>> coord = {'latitude': '37.24N', 'longitude': '-115.81W'}
>>> 'Coordinates: {latitude}, {longitude}'.format(**coord)
'Coordinates: 37.24N, -115.81W'
```

Accessing arguments' attributes:

```
>>> c = 3-5j
>>> ('The complex number {0} is formed from the real part {0.real} '
...  'and the imaginary part {0.imag}.').format(c)
'The complex number (3-5j) is formed from the real part 3.0 and the imaginary part -5.0.'
>>> class Point(object):
...     def __init__(self, x, y):
...         self.x, self.y = x, y
...     def __str__(self):
...         return 'Point({self.x}, {self.y})'.format(self=self)
...
>>> str(Point(4, 2))
'Point(4, 2)'
```

Accessing arguments' items:

```
>>> coord = (3, 5)
>>> 'X: {0[0]};  Y: {0[1]}'.format(coord)
'X: 3;  Y: 5'
```

Replacing `%s` and `%r`:

```
>>> "repr() shows quotes: {!r}; str() doesn't: {!s}".format('test1', 'test2')
"repr() shows quotes: 'test1'; str() doesn't: test2"
```

Aligning the text and specifying a width:

```
>>> '{:<30}'.format('left aligned')
'left aligned                  '
>>> '{:>30}'.format('right aligned')
'                 right aligned'
>>> '{:^30}'.format('centered')
'           centered           '
>>> '{:*^30}'.format('centered')  # use '*' as a fill char
'***********centered***********'
```

Replacing `%+f`, `%-f`, and `% f` and specifying a sign:

```
>>> '{:+f}; {:+f}'.format(3.14, -3.14)  # show it always
'+3.140000; -3.140000'
>>> '{: f}; {: f}'.format(3.14, -3.14)  # show a space for positive numbers
' 3.140000; -3.140000'
>>> '{:-f}; {:-f}'.format(3.14, -3.14)  # show only the minus -- same as '{:f}; {:f}'
'3.140000; -3.140000'
```

Using the comma as a thousands separator:

```
>>> '{:,}'.format(1234567890)
'1,234,567,890'
```

Expressing a percentage:

```
>>> points = 19.5
>>> total = 22
>>> 'Correct answers: {:.2%}'.format(points/total)
'Correct answers: 88.64%'
```

String common methods:

`str.`**`split`**([*sep*[, *maxsplit*]])

Return a list of the words in the string, using *sep* as the delimiter string. If *maxsplit* is given, at most*maxsplit* splits are done (thus, the list will have at most `maxsplit+1` elements). If *maxsplit* is not specified or `-1`, then there is no limit on the number of splits (all possible splits are made).

If *sep* is given, consecutive delimiters are not grouped together and are deemed to delimit empty strings (for example, `'1,,2'.split(',')` returns `['1', '', '2']`). The *sep* argument may consist of multiple characters (for example, `'1<>2<>3'.split('<>')` returns `['1', '2', '3']`). Splitting an empty string with a specified separator returns `['']`.

If *sep* is not specified or is `None`, a different splitting algorithm is applied: runs of consecutive whitespace are regarded as a single separator, and the result will contain no empty strings at the start or end if the string has

leading or trailing whitespace. Consequently, splitting an empty string or a string consisting of just whitespace with a `None` separator returns `[]`.

`str.`**`join`**(*iterable*)

Return a string which is the concatenation of the strings in the *iterable* *iterable*. The separator between elements is the string providing this method.

`str.`**`lower`**()/`str.`**`upper`**()/

Return a copy of the string with all the cased characters converted to lowercase/uppercase.

`str.`**`isalnum`**()/`str.`**`isalpha`**()/`str.`**`isdigit`**()

Return true if all characters in the string are alphanumeric/alphabetic/digits and there is at least one character, false otherwise.

`str.`**`islower`**()/`str.`**`isupper`**()

Return true if all cased characters in the string are lowercase/uppercase and there is at least one cased character, false otherwise.

`str.`**`isspace`**()

Return true if there are only whitespace characters in the string and there is at least one character, false otherwise.

`str.`**`count`**(*sub*[, *start*[, *end*]])

Return the number of non-overlapping occurrences of substring *sub* in the range [*start*, *end*]. Optional arguments *start* and *end* are interpreted as in slice notation.

`str.`**`startswith`**(*prefix*[, *start*[, *end*]])

`str.`**`endswith`**(*suffix*[, *start*[, *end*]])

Return `True` if the string starts/ends with the specified *prefix*/*suffix*, otherwise return `False`. *prefix*/*suffix* can also be a tuple of suffixes to look for. With optional *start*, test beginning at that position. With optional *end*, stop comparing at that position.

`str.`**`index`**(*sub*[, *start*[, *end*]])

Like **`find()`**, but raise **`ValueError`** when the substring is not found.

`str.`**`find`**(*sub*[, *start*[, *end*]])

Return the lowest index in the string where substring *sub* is found, such that *sub* is contained in the slice `s[start:end]`. Optional arguments *start* and *end* are interpreted as in slice notation. Return `-1` if *sub* is not found.

`str.`**`replace`**(*old*, *new*[, *count*])

Return a copy of the string with all occurrences of substring *old* replaced by *new*. If the optional argument *count* is given, only the first *count* occurrences are replaced.

`str.`**`format`**(**args*, ***kwargs*)

Perform a string formatting operation. Details can be found in the formatting string section.

`str.`**`ljust`**(*width*[, *fillchar*])/ `str.`**`rjust`**(*width*[, *fillchar*])

Return the string left/right justified in a string of length *width*. Padding is done using the specified *fillchar*(default is a space). The original string is returned if *width* is less than or equal to `len(s)`. These two functions are equivalent to '<' and '>' option in format string.

`str.`**`strip`**([*chars*])/ `str.`**`lstrip`**([*chars*])/ `str.`**`rstrip`**([*chars*])

Return a copy of the string with both/leading/trailing characters removed. The *chars* argument is a string specifying the set of characters to be removed. If omitted or `None`, the *chars* argument defaults to removing whitespace. The *chars* argument is not a prefix; rather, all combinations of its values are stripped.

`str.`**`translate`**(*table*[, *deletechars*])

Return a copy of the string where all characters occurring in the optional argument *deletechars* are removed, and the remaining characters have been mapped through the given translation table.You can use the **`maketrans()`** helper function in the **`string`** module to create a translation table. For string objects, set the *table* argument to `None` for translations that only delete characters:

```
>>> 'read this short text'.translate(None, 'aeiou')
'rd ths shrt txt'
```

`string.`**`maketrans`**(*from*, *to*) #must import string module first

Return a translation table suitable for passing to **`translate()`**, that will map each character in *from* into the character at the same position in *to*; *from* and *to* must have the same length.

Python OO programming

 # __repr__(self) in every new class to have its repr string representation
For example,

```
def __repr__(self):
        return self.id
```

#getattr(self,attr) can be used to get the self.attr inside a instance method(one way to inspect), for example:

```
def upateAttr(self,attr,new):
        attr = getattr(self,attr)
        attr.append(new)
```

#__cmp__(self,other) in new class to define the comparison(by default, if other is None, it will be smallest). For example:

```
def __cmp__(self,other):
        return cmp(self.value,other.value)
```

#__eq__(self,other) in new class to define equal, the precedence is higher than __cmp__(self,other), sometimes equal can not be defined by __cmp__ when you use only one attribute to compare in __cmp__. For example,

```
def __eq__(self,other):
         if other != None:
                return self.id==other.id and \
                        self.length == other.length and \
                        self.value==other.value
         else:
                return False
```

#__getattr__(self,attr) and __setattr__(self,attr,value) in new class to delegate current object's behavior to another object. Used in many design patterns, including singleton and adapter design pattern.
For example,

```
class DogAdapter():
        def __init__(self,dog_obj):
                self.dog_obj = dog_obj
        def speak(self):
                self.dog_obj.bark()
        def __getattr__(self,attr):
                return getattr(self.dog_obj,attr)
```

Python Core, execution model

#Python Execution Model

"Everything is an object?"

When most people first hear that in Python, "everything is an object", it triggers flashbacks to languages like Java where everything the *user* writes is encapsulated in an object. Others assume this means that in the implementation of the Python interpreter, everything is implemented as objects. The first interpretation is wrong; the second is true but not particularly interesting (for our purposes). What the phrase actually refers to is the fact that all "things", be they values, classes, functions, object instances (obviously), and almost every other language construct is conceptually an object.

What does it mean for everything to be an object? It means all of the "things" mentioned above have all the properties we usually associate with objects (in the object oriented sense); types have member functions, functions have attributes, modules can be passed as arguments, etc. And it has important implications with regards to how assignment in Python works.

A feature of the Python interpreter that often confuses beginners is what happens when `print()` is called on a "variable" assigned to a user-defined object (I'll explain the quotes in a second). With built-in types, a proper value is usually printed, like when calling `print()` on `strings` and `ints`. For simple, user-defined classes, though, the interpreter spits out some odd looking string like:

```
>>> class Foo(): pass
>>> foo = Foo()
>>> print(foo)
<__main__.Foo object at 0xd3adb33f>
```

`print()` is supposed to print the value of a "variable", right? So why is it printing that garbage?

To answer that, we need to understand what `foo` actually represents in Python. Most other languages would call it a variable. Indeed, many Python articles would refer to `foo` as a variable, but really only as a shorthand notation.

In languages like C, `foo` represents storage for "stuff". If we wrote

```
int foo = 42;
```

it would be correct to say that the integer variable `foo` contained the value `42`. That is, *variables are a sort of container for values*.

And now for something completely different...

In Python, this isn't the case. When we say:

```
>>> foo = Foo()
```

it would be wrong to say that foo "contained" a Foo object. Rather, foo *is a name with a binding to the object created by* Foo(). The portion of the right hand side of the equals sign creates an object. Assigning foo to that object merely says "I want to be able to refer to this object as foo." **Instead of variables (in the classic sense), Python has names and bindings**.

So when we printed foo earlier, what the interpreter was showing us was the address in memory where the object that foo is bound to is stored. This isn't as useless as it sounds. If you're in the interpreter and want to see if two names are bound to the same object, you can do a quick-and-dirty check by printing them and comparing the addresses. If they match, they're bound to the same object; if not, their bound to different objects. Of course, the idiomatic way to check if two names are bound to the same object is to use is

If we continued our example and wrote

```
>>> baz = foo
```

we should read this as "Bind the name baz to the same object foo is bound to (whatever that may be)." It should be clear, then why the following happens

```
 >>> baz.some_attribute
Traceback (most recent call last): File "<stdin>", line 1, in <module>
AttributeError: 'Foo' object has no attribute 'some_attribute'
>>> foo.some_attribute = 'set from foo'
>>> baz.some_attribute
'set from foo'
```

Changing the object in some way using foo will also be reflected in baz: they are both bound to the same underlying object.

What's in a name...

names in Python are not unlike names in the real world. If my wife calls me "Jeff", my dad calls me "Jeffrey", and my boss calls me "Idiot", it doesn't fundamentally change *me*. If my boss decides to call me "Captain Programming," great, but it still hasn't changed anything about me. It does mean, however, that if my wife kills "Jeff" (and who could blame her), "Captain Programming" is also dead. Likewise, in Python binding a name to an object doesn't change it. Changing some property of the object, however, will be reflected in all other names bound to that object.

Everything really *is* an object. I swear.

Here, a questions arises: How do we know that the thing on the right hand side of the equals sign will always be an object we can bind a name to? What about

```
>>> foo = 10
```

or

```
>>> foo = "Hello World!"
```

Now is when "everything is an object" pays off. Anything you can (legally) place on the right hand side of the equals sign is (or creates) an object in Python. Both `10` and `Hello World` are objects. Don't believe me? Check for yourself

```
>>> foo = 10
>>> print(foo.__add__)
<method-wrapper '__add__' of int object at 0x8502c0>
```

If `10` was actually just the number '10', it probably wouldn't have an `__add__` attribute (or any attributes at all).

In fact, we can see all the attributes `10` has using the `dir()` function:

```
>>> dir(10)
#a bunch of attributes
```

With all those attributes and member functions, I think it's safe to say `10` is an object.

Two types of objects

It turns out Python has two flavors of objects: `mutable` and `immutable`. The value of mutable objects can be changed after they are created. The value of immutable objects cannot be. A `list` is a mutable object. You can create a list, append some values, and the list is updated in place.
A `string` is immutable. Once you create a string, you can't change its value.

I know what you're thinking: "Of course you can change the value of a string, I do it all the time in my code!" When you "change" a string, you're actually rebinding it to a newly created string object. The original object remains unchanged, even though its possible that nothing refers to it anymore.

See for yourself:

```
>>> a = 'foo'
>>> a
```

```
'foo'
>>> b = a
>>> a += 'bar'
>>> a
'foobar'
>>> b
'foo'
```

Even though we're using `+=` and it *seems* that we're modifying the string, we really just get a new one containing the result of the change. This is why you may hear people say, "string concatenation is slow.". It's because concatenating strings must allocate memory for a new string and copy the contents, while appending to a list (in most cases) requires no allocation. Immutable objects are fundamentally expensive to "change", because doing so involves creating a copy. Changing mutable objects is cheap.

Immutable object weirdness

When I said the value of immutable objects can't change after they're created, it wasn't the whole truth. A number of containers in Python, such as `tuple`, are immutable. The value of a `tuple` can't be changed after it is created. But the "value" of a tuple is conceptually just a sequence of names with unchangeable bindings to objects. The key thing to note is that the *bindings* are unchangeable, not the objects they are bound to.

This means the following is perfectly legal:

```
>>> class Foo():
...     def __init__(self):
...         self.value = 0
...     def __str__(self):
...         return str(self.value)
...     def __repr__(self):
...         return str(self.value)
...
>>> f = Foo()
>>> print(f)
0
>>> foo_tuple = (f, f)
>>> print(foo_tuple)
(0, 0)
>>> foo_tuple[0] = 100
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> f.value = 999
>>> print(f)
999
>>> print(foo_tuple)
(999, 999)
```

When we try to change an element of the tuple directly, we get a TypeError telling us that (once created), tuples can't be assigned to. But changing the underlying object has the effect of "changing" the value of the tuple. This is a subtle point, but nonetheless important: the "value" of an immutable object *can't* change, but it's constituent objects *can*.

Function calls

If variables are just names bound to objects, what happens when we pass them as arguments to a function? The truth is, we aren't really passing all that much. Take a look at this code:

```
def list_changer(input_list):
        input_list[0] = 10
        input_list = range(1, 10)
        print(input_list)
        input_list[0] = 10
        print(input_list)
>>> test_list = [5, 5, 5]
>>> list_changer(test_list)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
[10, 2, 3, 4, 5, 6, 7, 8, 9]
>>> print test_list
[10, 5, 5]
```

Our first statement *does* change the value of the underlying list (as we can see in the last line printed). However, once we rebind the name input_list by saying input_list = range(1, 10), **we're now referring to a completely different object**. We basically said "bind the name input_list to this new list." After that line, we have no way of referring to the original input_list parameter again.

By now, you should have a clear understanding of how binding a name works. There's just one more item to take care of.

Blocks and Scope

The concepts of names, bindings, and objects should be quite familiar at this point. What we haven't covered, though, is how the interpreter "finds" a name. To see what I mean, consider the following:

```
GLOBAL_CONSTANT = 42
def print_some_weird_calculation(value):
        number_of_digits = len(str(value))

        def print_formatted_calculation(result):
                print('{value} * {constant} = {result}'.format(value=value,
                print('{}   {}'.format('^' * number_of_digits, '++'))
                print('\nKey: ^ points to your number, + points to constant')
        print_formatted_calculation(value * GLOBAL_CONSTANT)
```

```
>>> print_some_weird_calculation(123)
123 * 42 = 5166
^^^  ++
Key: ^ points to your number, + points to constant
```

This is a contrived example, but a couple of things should jump out at you. First, how does the print_formatted_calculation function have access tovalue and number_of_digits even though they were never passed as arguments? Second, how do both functions seem to have access to GLOBAL_CONSTANT?

The answer is all about scope. In Python, when a name is bound to an object, that name is only usable within the name's scope. The scope of a name is determined by the block in which it was created. A block is just a "block" of Python code that is executed as a single unit. The three most common types of blocks are modules, class definitions, and the bodies of functions. So the scope of a name is the innermost block in which it's defined.

Let's now return to the original question: how does the interpreter "find" what a name is bound to (or if it's even a valid name at all)? It begins by checking thescope of the innermost block. Then it checks the scope that contained the innermost block, then the scope that contained that, and so on.

In the print_formatted_calculation function, we reference value. This is resolved by first checking the scope of the innermost block, which in this case is the body of the function itself. When it doesn't find value defined there, it checks the scope that print_formatted_calculation was defined in. In our case, that's the body of the print_some_weird_calculation function. Here it does find the name value, and so it uses that binding and stops looking. The same is true for GLOBAL_CONSTANT, it just needs to look an extra level higher: the module (or script) level. Anything defined at this level is considered aglobal name. These are accessible from anywhere.

A few quick things to note. A name's scope extends to any blocks contained in the block where the name was defined, *unless the name is rebound in one of those blocks*. If print_formatted_calculation had the line value = 3, then the scope of the name value in print_some_weird_calculationwould only be the body of that function. It's scope would not include print_formatted_calculation, since that block rebound the name.

Use this power wisely...

There are two keywords that can be used to tell the interpreter to **reuse a preexisting binding**. Every other time we bind a name, it binds that name to a new object, *but only in the current scope*. In the example above, if we rebound value in print_formatted_calculation, it would have no affect on the valuein print_some_weird_calculation, which is print_formatted_calculation's enclosing scope. With the following two keywords, we can actually affect the bindings outside our local scope.

global my_variable tells the interpreter to use the binding of the name my_variable in the top-most (or "global" scope). Putting global my_variable in a code block is a way of saying, "copy the

binding of this global variable, or if you don't find it, create the name my_variable in the global scope." Similarly, thenonlocal my_variable statement instructs the interpreter to use the binding of the name my_variable defined in the nearest *enclosing* scope. This is a way to rebind a name not defined in either the local or global scope. Without nonlocal, we would only be able to alter bindings in the local scope or the global scope. Unlike global my_variable however, if we use nonlocal my_variable then my_variable must already exist; it won't be created if it's not found.

To see this in action, let's write a quick example:

```
GLOBAL_CONSTANT = 42
print(GLOBAL_CONSTANT)
def outer_scope_function():
        some_value = hex(0x0)
        print(some_value)

        def inner_scope_function():
                nonlocal some_value
                some_value = hex(0xDEADBEEF)
        inner_scope_function()
        print(some_value)
        global GLOBAL_CONSTANT
        GLOBAL_CONSTANT = 31337
outer_scope_function()
print(GLOBAL_CONSTANT)
# Output:  # 42 # 0x0 # 0xdeadbeef # 31337
```

By making use of global and nonlocal, we're able to use and change the existing binding of a name rather than merely assigning the name a new binding and losing the old one.

#Python parameter passing: call by object or call by object reference

Immutable objects: String, float, integer, and tuple
Mutable: List, user-defined object and all others

In Python, (almost) everything is an object. What we commonly refer to as "variables" in Python are more properly called *names*. Likewise, "assignment" is really the *binding* of a name to an *object*. Each binding has a *scope* that defines its visibility, usually the *block* in which the name originates.

There are actually two kinds of objects in Python. A *mutable* object exhibits time-varying behavior. Changes to a mutable object are visible through all names bound to it. Python's lists are an example of mutable objects. An *immutable* object does not exhibit time-varying behavior. The value of immutable objects cannot be modified after they are created. They *can* be used to compute the values of **new** objects, which is how a function like string.join works. When you think about it, this dichotomy is necessary because, again, everything is an object in Python. If integers were not immutable I could change the meaning of the number '2' throughout my program.

Example:
On line 1, we create a *binding* between a *name*, some_guy, and a string *object* containing 'Fred'. In the context of program execution, the *environment* is altered; a binding of the name some_guy' to a string object is created in the *scope* of the *block* where the statement occurred. When we later say some_guy = 'George', the string object containing 'Fred' is unaffected. We've just changed the binding of the name some_guy. **We haven't, however, changed either the 'Fred' or 'George' string objects**. As far as we're concerned, they may live on indefinitely.

With only a single name binding, this may seem overly pedantic, but it becomes more important when bindings are shared and function calls are involved. Let's say we have the following bit of Python code:

```
some_guy = 'Fred'
first_names = []
first_names.append(some_guy)
another_list_of_names = first_names
another_list_of_names.append('George')
some_guy = 'Bill'
print (some_guy, first_names, another_list_of_names)
```

So what get's printed in the final line? Well, to start, the binding of some_guy to the string object containing 'Fred' is added to the block's *namespace*. The name first_names is bound to an empty list object. On line 4, a method is called on the list objectfirst_names is bound to, appending the object some_guy is bound to. At this point, there are still only two objects that exist: the string object and the list object. some_guy and first_names[0] *both refer to the same object* (Indeed, print(some_guy is first_names[0]) shows this).

Let's continue to break things down. On line 6, a new name is bound: another_list_of_names. Assignment between names does not create a new object. Rather, both names are simply bound to the same object. As a result, the string object and list object are still the only objects that have been created by the interpreter. On line 7, a member function is called on the objectanother_list_of_names is bound to and it is *mutated* to contain a reference to a new object: 'George'. So to answer our original question, the output of the code is

```
Bill ['Fred', 'George'] ['Fred', 'George']
```

By now, you should almost be able to intuit how function calls work in Python. If I call foo(bar), I'm merely creating a binding within the scope of foo to the object the argument bar is bound to when the function is called. If bar refers to a mutable object and foochanges its value, then these changes will be visible outside of the scope of the function.

```
def foo(bar):
        bar.append(42)
        print(bar)
```

```
        # >> [42]
answer_list = []
foo(answer_list)
print(answer_list) # >> [42]
```

On the other hand, if `bar` refers to an immutable object, the most that `foo` can do is create a name `bar` in its local namespace and bind it to some other object.

```
def foo(bar):
        bar = 'new value'
        print (bar)    # >> 'new value'
answer_list = 'old value'
foo(answer_list)
print(answer_list) # >> 'old value'
```

Hopefully by now it is clear why Python is neither "call-by-reference" nor "call-by-value". In Python a variable is not an alias for a location in memory. Rather, it is simply a binding to a Python object. While the notion of "everything is an object" is undoubtedly a cause of confusion for those new to the language, it allows for powerful and flexible language constructs, which I'll discuss in my next post.

#Python copy module and concepts of shallow copy and deep copy

Assignment statements in Python do not copy objects, they create bindings between a target and an object. For collections that are mutable or contain mutable items, a copy is sometimes needed so one can change one copy without changing the other. This module provides generic shallow and deep copy operations (explained below).

Interface summary:

`copy.`**`copy`**(*x*)
Return a shallow copy of *x*.

`copy.`**`deepcopy`**(*x*)
Return a deep copy of *x*.

The difference between shallow and deep copying is only relevant for compound objects (objects that contain other objects, like lists or class instances):

- A ***shallow copy*** constructs a new compound object and then (to the extent possible) inserts*references* into it to the objects found in the original. For example, ls[:] is a shallow copy, it creates new list object, but the elements inside the list are still reference to original element in original list.

```
>>a = [[1],[2]]
>>b = a[:]
>>a[0].append(1)
>>print a,b
[[1,1],[2]]   [[1,1],[2]]
```

- A ***deep copy*** constructs a new compound object and then, recursively, inserts *copies* into it of the objects found in the original.

Python odds and ends

#Bitwise operations

Bitwise operations only make sense for integers. Negative numbers are treated as their 2's complement value (this assumes a sufficiently large number of bits that no overflow occurs during the operation).

The priorities of the binary bitwise operations are all lower than the numeric operations and higher than the comparisons; the unary operation `~` has the same priority as the other unary numeric operations (`+` and `-`).

This table lists the bitwise operations sorted in ascending priority:

Operation	Result	Notes
`x | y`	bitwise *or* of *x* and *y*	
`x ^ y`	bitwise *exclusive or* of *x* and *y*	
`x & y`	bitwise *and* of *x* and *y*	
`x << n`	*x* shifted left by *n* bits	(1)(2)
`x >> n`	*x* shifted right by *n* bits	(1)(3)
`~x`	the bits of *x* inverted	

Notes:

1. Negative shift counts are illegal and cause a `ValueError` to be raised.
2. A left shift by *n* bits is equivalent to multiplication by `pow(2, n)`. A long integer is returned if the result exceeds the range of plain integers.
3. A right shift by *n* bits is equivalent to division by `pow(2, n)`.

#Defining functions with * and **

//Arbitrary Argument Lists(*) or key word argument Lists(**)

These arguments will be wrapped up in a **tuple/dict**. Before the variable number of arguments, zero or more normal arguments may occur.

```
def write_multiple_items(file, separator, *args):
    file.write(separator.join(args))

def foo(*args):
    print args
>>foo(1,2,"34",21,"65")
(1,2,"34",21,"65")
def foo2(**kwargs):
```

```
    print kwargs
>>foo2(a=1,b='2')
{'a':1,'b':'2'}
```

The order of arguments should be
def my_function(arg1, arg2=default, *args, **kwargs):

//Unpacking Argument Lists, *(list), **(dict)

The reverse situation occurs when the arguments are already in a list or tuple but need to be unpacked for a function call requiring separate positional arguments. For instance, the built-in `range()` function expects separate *start* and *stop* arguments. If they are not available separately, write the function call with the **`*`-operator** to unpack the arguments out of a list or tuple:

```
>>> range(3, 6)             # normal call with separate arguments
[3, 4, 5]
>>> args = [3, 6]
>>> range(*args)            # call with arguments unpacked from a list
[3, 4, 5]
```

In the same fashion, dictionaries can deliver keyword arguments with the **`**`-operator**:

```
>>> def parrot(voltage, state='a stiff', action='voom'):
...     print "-- This parrot wouldn't", action,
...     print "if you put", voltage, "volts through it.",
...     print "E's", state, "!"
...
>>> d = {"voltage": "four million", "state": "bleedin' demised",
"action": "VOOM"}
>>> parrot(**d)
-- This parrot wouldn't VOOM if you put four million volts through it.
E's bleedin' demised !
```

#Python time/timeit module

//time.time() to get current time, and calculate the running time of some code.
For example,

```
cur_time = time.time()
some_function()
print "running time is ", time.time() – cur_time
```

//Other methods or Timer class in timeit are specifically for testing statements

```
>>> # attribute is missing
>>> s = """\
... try:
...     str.__nonzero__
... except AttributeError:
...     pass
... """
>>> timeit.timeit(stmt=s, setup='import something', number=100000)
0.9138244460009446
>>> s = "if hasattr(str, '__bool__'): pass"
>>> timeit.timeit(stmt=s, setup = 'import something', number=100000)
0.5829014980008651
```

#Python random module

//Basic function random.seed(i) and random.random()

`random.`**`seed`**([*x*])

Initialize the basic random number generator. Optional argument *x* can be any *hashable* object. If *x*is omitted or `None`, current system time is used; current system time is also used to initialize the generator when the module is first imported.

`random.`**`random`**()

Return the next random floating point number in the range [0.0, 1.0).

//Functions for integer

`random.`**`randrange`**(*stop*)

`random.`**`randrange`**(*start*, *stop*[, *step*])

Return a randomly selected element from `range(start, stop, step)`. This is equivalent to `choice(range(start, stop, step))`, but doesn't actually build a range object.

`random.`**`randint`**(*a*, *b*)

Return a random integer *N* such that `a <= N <= b`.

//Functions for sequence

`random.`**`choice`**(*seq*)

Return a random element from the non-empty sequence *seq*. If *seq* is empty, raises **`IndexError`**.

`random.`**`shuffle`**(*x*[, *random*])

Shuffle the sequence *x* in place. The optional argument *random* is a 0-argument function returning a random float in [0.0, 1.0); by default, this is the function **`random()`**.

`random.`**`sample`**(*population*, *k*)

Return a *k* length list of unique elements chosen from the population sequence. Used for random sampling without replacement. Returns a new list containing elements from the population while leaving the original population unchanged

#Define functions inside a function

To keep code isolated, the inside method does not need (self,) when defined inside an instance method. The life scope is within the outside function. Useful for some big functions, like ValidRowCol().

#Python datetime module

//timedelta Object

A **`timedelta`** object represents a duration, the difference between two dates or times. It is usually generated by operations on two datetimes/dates/times.

Instance attributes (read-only):

Attribute	Value
`days`	Between -999999999 and 999999999 inclusive
`seconds`	Between 0 and 86399 inclusive
`microseconds`	Between 0 and 999999 inclusive

Instance methods:

`timedelta.`**`total_seconds`**()

Return the total number of seconds contained in the duration. Equivalent to `(td.microseconds +(td.seconds + td.days * 24 * 3600) * 10**6) / 10**6` computed with true division enabled.

//datetime Object

A **`datetime`** object is a single object containing all the information from a **`date`** object and a **`time`** object. Like a **`date`** object, **`datetime`** assumes the current Gregorian calendar extended in both directions; like a time object, **`datetime`** assumes there are exactly 3600*24 seconds in every day.

Some useful constructors and class methods:

class `datetime.`**`datetime`**(*year*, *month*, *day*[, *hour*[, *minute*[, *second*[, *microsecond*[, *tzinfo*]]]]])

The year, month and day arguments are required.

classmethod `datetime.today()`

Return the current local datetime, with `tzinfo` `None`. This is equivalent to `datetime.fromtimestamp(time.time())`. Compared with date Object, date.today() only returns date info with time

classmethod `datetime.now([tz])`

Return the current local date and time.

datetime Object can generate date and time objects respectively.

`datetime.date()`

Return `date` object with same year, month and day.

`datetime.time()`

Return `time` object with same hour, minute, second and microsecond. `tzinfo` is `None`.

datetime Object Supported operations:

Operation	Result
`datetime2 = datetime1 + timedelta`	Moving forward(+) or backford(-)
`datetime2 = datetime1 - timedelta`	
`timedelta = datetime1 - datetime2`	Duration between two datetimes
`datetime1 < datetime2`	Compares `datetime` to `datetime`.

//global and nonlocal variables

global my_variable tells the interpreter to use the binding of the name my_variable in the top-most (or "global" scope). Putting global my_variable in a code block is a way of saying, "copy the binding of this global variable, or if you don't find it, create the name my_variable in the global scope." Similarly, the nonlocal my_variable statement instructs the interpreter to use the binding of the name my_variable defined in the nearest *enclosing* scope. This is a way to rebind a name not defined in either the local or global scope. Without nonlocal, we would only be able to alter bindings in the local scope or the global scope. Unlike global my_variable however, if we use nonlocal my_variable then my_variable must already exist; it won't be created if it's not found.

Python good programming practices and tips

The goal of this section is to write concise and clear code.

#Change value in an iterable through 'for loop' using enumerate (this is inappropriate if you want to manipulate the index, in that case, use while):

```
for idx, item in enumerate(ls):
        ls[idx] = item+1
```

or

```
for i in range(len(ls)):
        ls[i] = ls[i]+1
```

#Multiple variables assignment in one line

```
>>a,b,c = 1,2,3
>>print a,b,c
1 2 3
>>c1=c2=c3=4
>>print c1,c2,c3
4 4 4
```

#Swap the values of two variables in one line

```
>>a,b=1,2
>>a,b = b,a
>>print a,b
2 1
```

#List Comprehension (also works for string, deque, tuple/ set, dict)

//for

```
>>sum([x**2 for x in range(n+1)])        #sum of square
```

//for, if (equivalent to filter())

```
>>sum([x**2 for x in range(n+1) if x%2 == 0])
#add condition for filtering
```

//for-for, if

```
>> [(x,y) for x in [1,2,3] for y in [3,1,4] if x!=y]
#very helpful in 2d-list problem
[(1,3),(1,4),(2,3),(2,1),(2,4),(3,1),(3,4)]
```

// if-else, for

```
>>[x if x<5 else 10 for x in range(n)]        #add condition inside loop
```

//if-else-if-else, for

```
>>[x if x<3 else 5 if x<5 else 10 for x in range(n)]
```

//nested

```
>>[[row[i] for row in matrix] for i in range(len(matrix[0]))]
#transpose a matrix
```

#Exception handling (very useful for IndexError)

```
try:
        ls[i][j]….
except IndexError:
        pass
```

#print string in same line with multiple statements

```
>>import sys
>>sys.stdout.write("-1");sys.stdout.write("23")
-123
```

#In searching algorithm, quit nested when found

```
class Found(Exception):
        pass
try:
        while…
                while…
                        raise Found
        process_unfound()
except Found:
        process_found()
```

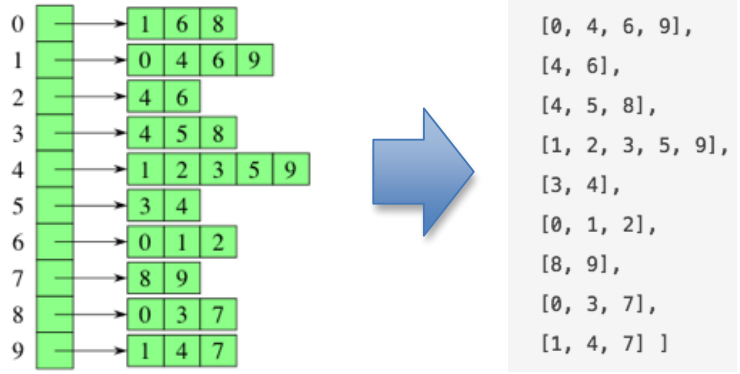
#Binary operation tips

// Get the rightmost 1s with two's complement

```
a &= -a     #1110 =>0010
```

#Store graph

//Adjacency list



Vertex numbers in an adjacency list are not required to appear in any particular order, though it is often convenient to list them in increasing order. If the graph is weighted, then each item in each adjacency list is either a tuple or an object, giving the vertex number and the edge weight.

To call a function on each vertex adjacent to vertex I, you could use the following code:

```
vertex = graph[i]
for node in vertex:
        doStuff(node)
```

//Adjacency matrix

Two-dimension list can store a matrix

```
[ [0, 1, 0, 0, 0, 0, 1, 0, 1, 0],
  [1, 0, 0, 0, 1, 0, 1, 0, 0, 1],
  [0, 0, 0, 0, 1, 0, 1, 0, 0, 0],
  [0, 0, 0, 0, 1, 1, 0, 0, 1, 0],
  [0, 1, 1, 1, 0, 1, 0, 0, 0, 1],
  [0, 0, 0, 1, 1, 0, 0, 0, 0, 0],
  [1, 1, 1, 0, 0, 0, 0, 0, 0, 0],
  [0, 0, 0, 0, 0, 0, 0, 0, 1, 1],
  [1, 0, 0, 1, 0, 0, 0, 1, 0, 0],
  [0, 1, 0, 0, 1, 0, 0, 1, 0, 0] ]
```

Edges can be accessed by graph[i][j]. If it is weighted graph, we can store weight in the matrix.

#Sorting or comparison for object or tuple/list

Use lambda as the cmp argument, for example, you want to compare by the second item in tuple ls=[(0,1),(-1,3)].

```
ls.sort(cmp=lambda x,y:cmp(x[1],y[1]))        #inplace sorting
sorted(ls,cmp=lambda x,y:cmp(x[1],y[1]))    #general sorting
```

#print number or str from list/sequence in a good format(fixed width, fixed decimal). For example:

```
>>print " ".join(map(format,range(20),[">2" for _ in range(20)]))
0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19
```

#List comprehension from two list using zip

```
>>sum([x*y for x,y in zip(l1,l2)])
#calculate the dot product between l1 and l2
```

Python implementation of Design Pattern
(See Concepts in OOAD_DesignPattern,
See implementations in github/Misc/DesignPatterns)

#MVC design pattern
#Singleton design pattern
#Facade design pattern
#Observer design pattern
#Factory design pattern
#Abstract Factory design pattern
#Adapter design pattern
#Iterator design pattern (built-in in python by __iter__)
#Strategy design pattern

Python 3 Main differences