

PYTHONPATH - It has a role similar to PATH. This variable tells the Python interpreter where to locate the module files imported into a program. It should include the Python source library directory and the directories containing Python source code. PYTHONPATH is sometimes preset by the Python installer.

PYTHONSTARTUP - It contains the path of an initialization file containing Python source code. It is executed every time you start the interpreter. It is named as .pythonrc.py in UNIX and it contains commands that load utilities or modify PYTHONPATH.

PYTHONCASEOK – It is used in Windows to instruct Python to find the first case-insensitive match in an import statement. Set this variable to any value to activate it

PYTHONHOME – It is an alternative module search path. It is usually embedded in the PYTHONSTARTUP or PYTHONPATH directories to make switching module libraries easy.

Object Identity and Type

```
# Compare two objects
def compare(a,b):
    print 'The identity of a is ', id(a)
    print 'The identity of b is ', id(b)
    if a is b:
        print 'a and b are the same object'
    if a == b:
        print 'a and b have the same value'
    if type(a) is type(b):
        print 'a and b have the same type'
```

list Ex:

```
print([(x,y,z) for x in range(1,30) for y in range(x,30) for z in range(y,30) if x**2 + y**2 == z**2])
```

O/P:

```
[(3, 4, 5), (5, 12, 13), (6, 8, 10), (7, 24, 25), (8, 15, 17), (9, 12, 15), (10, 24, 26), (12, 16, 20), (15, 20, 25), (20, 21, 29)]
```

```
if type(s) is list:
    print 'Is a list'
if type(f) is file:
    print 'Is a file'
```

However, some type names are only available in the types module. For example:

```
import types
```

```
if type(s) is types.NoneType:
```

```
    print "is None"
```

Because types can be specialized by defining classes, a better way to check types is to use the built-in `isinstance(object, type)` function. For example:

```
if isinstance(s,list):
```

```
    print 'Is a list'
```

```
if isinstance(f,file):
```

```
    print 'Is a file'
```

```
if isinstance(n,types.NoneType):
```

```
    print "is None"
```

Ex:

```
# -*- coding: utf-8 -*-
```

```
a=[1,3,'raj','d',432]
```

```
b=(1,3,'raj','d',432)
```

```
print('ID of a is: ',id(a))
```

```
    ('ID of a is: ', 35821128)
```

```
print('ID of b is: ',id(b))
```

```
    ('ID of b is: ', 31260320)
```

```
print('Type of a is: ',type(a))
```

```
    ('Type of a is: ', <type 'list'>)
```

```
print('Type of b is: ',type(b))
```

```
    ('Type of b is: ', <type 'tuple'>)
```

```
c=4
```

```
d=4
```

```
if isinstance(c, int):
```

```
    print('c is Integer')
```

```
O/P: c is Integer
```

```
print('ID of a is: ',id(c))
```

```
    ('ID of a is: ', 31501532)
```

```
print('ID of b is: ',id(d))  
('ID of b is: ', 31501532)
```

```
print('Type of a is: ',type(c))  
('Type of a is: ', <type 'int'>)
```

```
print('Type of b is: ',type(d))  
('Type of b is: ', <type 'int'>)
```

```
if c is d:  
    print(c,'and ',d,' are same objects')  
if c==d:  
    print(c,'and ',d,' are same same')  
if type(c) is type(d):  
    print(c,'and ',d,' are same types')
```

```
if type(a) is list:  
    print('list');  
if type(b) is tuple:  
    print('tuple');
```

O/P:

```
(4, 'and ', 4, ' are same objects')  
(4, 'and ', 4, ' are same same')  
(4, 'and ', 4, ' are same types')  
list  
tuple
```

Reference Counting and Garbage Collection

An object's reference count is increased whenever it's assigned to a new name or placed in a container such as a list, tuple, or dictionary, as shown here:

```
a = 3.4 # Creates an object '3.4'  
b = a # Increases reference count on '3.4'  
c = []  
c.append(b) # Increases reference count on '3.4'
```

In this case object a is created with value 3.4. Later lines are just creates references to those objects. So, in this example the reference count for the value 3.4 is 3.

An object's reference count is decreased by the del statement or whenever a reference goes out of scope (or is reassigned). For example:

```
del a # Decrease reference count of 3.4
```

```
b = 7.8 # Decrease reference count of 3.4
```

```
c[0]=2.0 # Decrease reference count of 3.4
```

When an object's reference count reaches zero, it is garbage-collected.

However, in some cases a circular dependency may exist among a collection of objects that are no longer in use. For example:

```
a = { }
```

```
b = { }
```

```
a['b'] = b # a contains reference to b
```

```
b['a'] = a # b contains reference to a
```

```
del a
```

```
del b
```

In this example, the del statements decrease the reference count of a and b and destroy the names used to refer to the underlying objects. However, because each object contains a reference to the other, the reference count doesn't drop to zero and the objects remain allocated (resulting in a memory leak). To address this problem, the interpreter periodically executes a cycle-detector that searches for cycles of inaccessible objects and deletes them.

References and Copies

Reference:

```
b = [1,2,3,4]
```

```
a = b # a is a reference to b
```

```
a[2] = -100 # Change an element in 'a'
```

```
print b # Produces '[1, 2, -100, 4]'
```

Copy:

shallow copy

```
b = [ 1, 2, [3,4] ]
```

```
a = b[:] # Create a shallow copy of b.
```

```
a.append(100) # Append element to a.
```

```
print b # Produces '[1,2, [3,4]]'. b unchanged.  
a[2][0] = -100 # Modify an element of a.  
print b # Produces '[1,2, [-100,4]]'.
```

deep copy: creates a new object and recursively copies all the objects it contains

```
import copy  
b = [1, 2, [3, 4] ]  
a = copy.deepcopy(b)  
a[2] = -100  
print a # produces [1,2, -100, 4]  
print b # produces [1,2,3,4]
```

Built-in Types:

The None Type:

The None type denotes a null object (an object with no value). None has no attributes and evaluates to False in Boolean expressions.

Ex:

```
a=None  
print a
```

if None:

```
    print True;  
else:  
    print False;
```

O/p:

```
None  
False
```

Numeric Types:

Python uses five numeric types: Booleans, integers, long integers, floating-point numbers, and complex numbers. Except for Booleans, all numeric objects are signed. All numeric types are immutable.

Complex number Example:

```
a=4+6j  
print a.real  
print a.imag
```

o/p:

```
4.0  
6.0
```

Sequence Types

Sequences represent ordered sets of objects indexed by nonnegative integers and include strings, Unicode strings, lists, and tuples. Strings are sequences of characters, and lists and tuples are sequences of arbitrary Python objects. Strings and tuples are immutable; lists allow insertion, deletion, and substitution of elements. All sequences support iteration.

Table 3.5 String Methods

Method	Description
<code>s.capitalize()</code>	Capitalizes the first character.
<code>s.center(width [, pad])</code>	Centers the string in a field of length <i>width</i> . <i>pad</i> is a padding character.
<code>s.count(sub [, start [, end]])</code>	Counts occurrences of the specified substring <i>sub</i> .
<code>s.decode([encoding [, errors]])</code>	Decodes a string and returns a Unicode string.
<code>s.encode([encoding [, errors]])</code>	Returns an encoded version of the string.
<code>s.endswith(suffix [, start [, end]])</code>	Checks the end of the string for a suffix.
<code>s.expandtabs([tabsize])</code>	Replaces tabs with spaces.
<code>s.find(sub [, start [, end]])</code>	Finds the first occurrence of the specified substring <i>sub</i> .
<code>s.index(sub [, start [, end]])</code>	Finds the first occurrence or error in the specified substring <i>sub</i> .
<code>s.isalnum()</code>	Checks whether all characters are alphanumeric.
<code>s.isalpha()</code>	Checks whether all characters are alphabetic.
<code>s.isdigit()</code>	Checks whether all characters are digits.
<code>s.islower()</code>	Checks whether all characters are lowercase.
<code>s.isspace()</code>	Checks whether all characters are whitespace.
<code>s.istitle()</code>	Checks whether the string is a title-cased string (first letter of each word capitalized).
<code>s.isupper()</code>	Checks whether all characters are uppercase.
<code>s.join(t)</code>	Joins the strings <i>s</i> and <i>t</i> .
<code>s.ljust(width [, fill])</code>	Left-aligns <i>s</i> in a string of size <i>width</i> .

```
'rajesh paleru rajesh'
>>> s.find('raj')
0
>>> s.index('raj')
0
>>> s.find('raja')
-1
>>> s.index('raja')
Traceback (most recent call last):
  File "<pyshell#36>", line 1, in <module>
    s.index('raja')
ValueError: substring not found
>>> |
```

<code>s.lower()</code>	Converts to lowercase.
<code>s.lstrip([chars])</code>	Removes leading whitespace or characters supplied in <i>chars</i> .
<code>s.replace(old, new [,maxreplace])</code>	Replaces the substring.
<code>s.rfind(sub [,start [,end]])</code>	Finds the last occurrence of a substring.
<code>s.rindex(sub [,start [,end]])</code>	Finds the last occurrence or raises an error.
<code>s.rjust(width [, fill])</code>	Right-aligns <i>s</i> in a string of length <i>width</i> .
<code>s.rsplit([sep [,maxsplit]])</code>	Splits a string from the end of the string using <i>sep</i> as a delimiter. <i>maxsplit</i> is the maximum number of splits to perform. If <i>maxsplit</i> is omitted, the result is identical to the <code>split()</code> method.
<code>s.rstrip([chars])</code>	Removes trailing whitespace or characters supplied in <i>chars</i> .
<code>s.split([sep [,maxsplit]])</code>	Splits a string using <i>sep</i> as a delimiter. <i>maxsplit</i> is the maximum number of splits to perform.
<code>s.splitlines([keepends])</code>	Splits a string into a list of lines. If <i>keepends</i> is 1, trailing newlines are preserved.
<code>s.startswith(prefix [,start [,end]])</code>	Checks whether a string starts with <i>prefix</i> .
<code>s.strip([chars])</code>	Removes leading and trailing whitespace or characters supplied in <i>chars</i> .
<code>s.swapcase()</code>	Converts uppercase to lowercase, and vice versa.
<code>s.title()</code>	Returns a title-cased version of the string.
<code>s.translate(table [,delechars])</code>	Translates a string using a character translation table <i>table</i> , removing characters in <i>delechars</i> .
<code>s.upper()</code>	Converts a string to uppercase.
<code>s.zfill(width)</code>	Pads a string with zeros on the left up to the specified <i>width</i> .

Mapping Types

Dictionaries are the only built-in mapping type and are Python's version of a hash table or associative array. You can use any immutable object as a dictionary key value (strings, numbers, tuples, and so on). Lists, dictionaries, and tuples containing mutable objects cannot be used as keys (the dictionary type requires key values to remain constant). To select an item in a mapping object, use the key index operator `m[k]`, where *k* is a key value. If the key is not found, a `KeyError` exception is raised.

Table 3.6 Methods and Operations for Dictionaries

Item	Description
<code>len(m)</code>	Returns the number of items in <i>m</i> .
<code>m[k]</code>	Returns the item of <i>m</i> with key <i>k</i> .
<code>m[k]=x</code>	Sets <i>m[k]</i> to <i>x</i> .
<code>del m[k]</code>	Removes <i>m[k]</i> from <i>m</i> .
<code>m.clear()</code>	Removes all items from <i>m</i> .
<code>m.copy()</code>	Makes a shallow copy of <i>m</i> .
<code>m.has_key(k)</code>	Returns <code>True</code> if <i>m</i> has key <i>k</i> ; otherwise, returns <code>False</code> .
<code>m.items()</code>	Returns a list of (<i>key</i> , <i>value</i>) pairs.
<code>m.iteritems()</code>	Returns an iterator that produces (<i>key</i> , <i>value</i>) pairs.
<code>m.iterkeys()</code>	Returns an iterator that produces dictionary keys.
<code>m.itervalues()</code>	Returns an iterator that produces dictionary values.
<code>m.keys()</code>	Returns a list of key values.
<code>m.update(b)</code>	Adds all objects from <i>b</i> to <i>m</i> .
<code>m.values()</code>	Returns a list of all values in <i>m</i> .
<code>m.get(k [, v])</code>	Returns <i>m[k]</i> if found; otherwise, returns <i>v</i> .
<code>m.setdefault(k [, v])</code>	Returns <i>m[k]</i> if found; otherwise, returns <i>v</i> and sets <i>m[k] = v</i> .
<code>m.pop(k [, default])</code>	Returns <i>m[k]</i> if found and removes it from <i>m</i> ; otherwise, returns <i>default</i> if supplied or raises <code>KeyError</code> if not.
<code>m.popitem()</code>	Removes a random (<i>key</i> , <i>value</i>) pair from <i>m</i> and returns it as a tuple.

Set Types

'set' object does not support indexing

set type is **mutable** -- the contents can be changed using methods like `add()` and `remove()`. Since it is mutable, it has no hash value and cannot be used as either a dictionary key or as an element of another set. **frozenset** type is **immutable** and hashable -- its contents cannot be altered after it is created; however, it can be used as a dictionary key or as an element of another set.

```
>>> s
{1, 2, 3, 4}
>>> t
[4, 5, 3]
```

```
>>> t={3,4,5,s}
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    t={3,4,5,s}
TypeError: unhashable type: 'set'
>>> t={3,4,5,{5,4}}
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
```


t={3,4,5,{5,4}}

TypeError: unhashable type: 'set'

Instances of **set** and **frozenset** provide the following 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)	$s \leq t$	test whether every element in s is in t
s.issuperset(t)	$s \geq t$	test whether every element in t is in s
s.union(t)	$s \mid t$	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)	$s \wedge t$	new set with elements in either s or t but not both
s.copy()		new set with a shallow copy of s

A *set* is an unordered collection of unique items. Unlike sequences, sets provide no indexing or slicing operations. They are also unlike dictionaries in that there are no key values associated with the objects. In addition, the items placed into a set must be immutable. Two different set types are available:

`set` is a mutable set, and `frozenset` is an immutable set. Both kinds of sets are created using a pair of built-in functions:

```
s = set([1,5,10,15])
```

```
f = frozenset(['a',37,'hello'])
```

Table 3.8 Methods for Mutable Set Types

Item	Description
<code>s.add(item)</code>	Adds <i>item</i> to <i>s</i> . Has no effect if <i>item</i> is already in <i>s</i> .
<code>s.clear()</code>	Removes all items from <i>s</i> .
<code>s.difference_update(t)</code>	Removes all the items from <i>s</i> that are also in <i>t</i> .
<code>s.discard(item)</code>	Removes <i>item</i> from <i>s</i> . If <i>item</i> is not a member of <i>s</i> , nothing happens.
<code>s.intersection_update(t)</code>	Computes the intersection of <i>s</i> and <i>t</i> and leaves the result in <i>s</i> .
<code>s.pop()</code>	Returns an arbitrary set element and removes it from <i>s</i> .
<code>s.remove(item)</code>	Removes <i>item</i> from <i>s</i> . If <i>item</i> is not a member, <code>KeyError</code> is raised.
<code>s.symmetric_difference_update(t)</code>	Computes the symmetric difference of <i>s</i> and <i>t</i> and leaves the result in <i>s</i> .
<code>s.update(t)</code>	Adds all the items in <i>t</i> to <i>s</i> . <i>t</i> may be another set, a sequence, or any object that supports iteration.

Callable Types: Callable types represent objects that support the function call operation

Ex: user-defined functions, built-in functions, instance methods, and classes.

User-defined functions are callable objects created at the module level by using the `def` statement, at the class level by defining a static method, or with the `lambda` operator.

Here's an example:

```
def foo(x,y):  
    return x+y
```

```
class A(object):
    @staticmethod
    def foo(x,y):
    return x+y
```

Attribute(s)	Description
<code>f.__doc__</code> or <code>f.func_doc</code>	Documentation string
<code>f.__name__</code> or <code>f.func_name</code>	Function name
<code>f.__dict__</code> or <code>f.func_dict</code>	Dictionary containing function attributes
<code>f.func_code</code>	Byte-compiled code
<code>f.func_defaults</code>	Tuple containing the default arguments
<code>f.func_globals</code>	Dictionary defining the global name-space
<code>f.func_closure</code>	Tuple containing data related to nested scopes

Operators and Expressions:

Operations on Numbers

The following operations can be applied to all numeric types:

Operation	Description
<code>x + y</code>	Addition
<code>x - y</code>	Subtraction
<code>x * y</code>	Multiplication
<code>x / y</code>	Division
<code>x // y</code>	Truncating division
<code>x ** y</code>	Power (x^y)
<code>x % y</code>	Modulo ($x \bmod y$)
<code>-x</code>	Unary minus
<code>+x</code>	Unary plus
<code>x << y</code>	Left shift
<code>x >> y</code>	Right shift
<code>x & y</code>	Bitwise AND
<code>x y</code>	Bitwise OR
<code>x ^ y</code>	Bitwise XOR (exclusive OR)
<code>~x</code>	Bitwise negation

Operations on Sequences

The following operators can be applied to sequence types, including strings, lists, and tuples:

Operation	Description
<code>s + r</code>	Concatenation
<code>s * n, n * s</code>	Makes <i>n</i> copies of <i>s</i> , where <i>n</i> is an integer
<code>s % d</code>	String formatting (strings only)
<code>s[i]</code>	Indexing
<code>s[i:j]</code>	Slicing
<code>s[i:j:stride]</code>	Extended slicing
<code>x in s, x not in s</code>	Membership
<code>for x in s:</code>	Iteration
<code>len(s)</code>	Length
<code>min(s)</code>	Minimum item
<code>max(s)</code>	Maximum item

The slicing operator may be given an optional stride, **`s[i:j:stride]`**, that causes the slice to skip elements.

If the **starting index i** is omitted, it is set to the beginning of the sequence if stride is positive or the end of the sequence if stride is negative.

If the **ending index j** is omitted, it is set to the end of the sequence if stride is positive or the beginning of the sequence if stride is negative.

examples:

```
a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
b = a[::2] # b = [0, 2, 4, 6, 8]
```

```
c = a[::-2] # c = [9, 7, 5, 3, 1]
```

```
d = a[0:5:2] # d = [0,2,4]
```

```
e = a[5:0:-2] # e = [5,3,1]
```

```
f = a[:5:1] # f = [0,1,2,3,4]
```

```
g = a[:5:-1] # g = [9,8,7,6]
```

```
h = a[5::1] # h = [5,6,7,8,9]
```

```
i = a[5::-1] # i = [5,4,3,2,1,0]
```

```
j = a[5:0:-1] # j = [5,4,3,2,1]
```

```
a = [1,2,3,4,5]
```

```
a[1] = 6 # a = [1,6,3,4,5]
```

```
a[2:4] = [10,11] # a = [1,6,10,11,5]
```

```
a[3:4] = [-1,-2,-3] # a = [1,6,10,-1,-2,-3,5]
```

```
a[2:] = [0] # a = [1,6,0]
```

```
a = [1,2,3,4,5]
```

```
a[1::2] = [10,11] # a = [1,10,3,11,5]
```

```
a[1::2] = [30,40,50] # ValueError. Only two elements in slice on left
```

The slicing assignment operator `s[i:j] = r` replaces elements `k`, where `i <= k < j`, with elements from sequence `r`.

```
a = [1,2,3,4,5]
```

```
a[1] = 6 # a = [1,6,3,4,5]
```

```
a[2:4] = [10,11] # a = [1,6,10,11,5]
```

```
a[3:4] = [-1,-2,-3] # a = [1,6,10,-1,-2,-3,5]
```

```
a[2:] = [0] # a = [1,6,0]
```

Operations on Dictionaries

Dictionaries provide a mapping between names and objects. You can apply the following operations to dictionaries:

Operation Description

`x = d[k]` Indexing by key

`d[k] = x` Assignment by key

`del d[k]` Deletes an item by key

`len(d)` Number of items in the dictionary

Key values can be any immutable object, such as strings, numbers, and tuples. In addition, dictionary keys can be specified as a comma-separated list of values, like this:

```
d = { }
```

```
d[1,2,3] = "foo"
```

```
d[1,0,3] = "bar"
```

In this case, the key values represent a tuple, making the preceding assignments identical to the following:

```
d[(1,2,3)] = "foo"
```

```
d[(1,0,3)] = "bar"
```

Program:

```
d = {(1,'r'),(2,'a'),(3,'j'),(4,'e'),(5,'s'),(6,'h')}
```

```
print dict(d)
```

```
d = [(1,'r'),(2,'a'),(3,'j'),(4,'e'),(5,'s'),(6,'h')]
```

```
print dict(d)
```

```
d=((1,'r'),(2,'a'),(3,'j'),(4,'e'),(5,'s'),(6,'h'))
print dict(d)
```

```
d=(1,'r'),(2,'a'),(3,'j'),(4,'e'),(5,'s'),(6,'h')
print dict(d)
```

O/p:

```
{1: 'r', 2: 'a', 3: 'j', 4: 'e', 5: 's', 6: 'h'}
```

```
{1: 'r', 2: 'a', 3: 'j', 4: 'e', 5: 's', 6: 'h'}
```

```
{1: 'r', 2: 'a', 3: 'j', 4: 'e', 5: 's', 6: 'h'}
```

```
{1: 'r', 2: 'a', 3: 'j', 4: 'e', 5: 's', 6: 'h'}
```

Object Equality and Identity

The equality operator ($x == y$) tests the values of x and y for equality. In the case of lists and tuples, all the elements are compared and evaluated as true if they're of equal value. For dictionaries, a true value is returned only if x and y have the same set of keys and all the objects with the same key have equal values. Two sets are equal if they have the same elements, which are compared using equality ($==$).

The identity operators (x is y and x is not y) test two objects to see whether they refer to the same object in memory.

Program:

```
a={1:'r',2:'a'}
b={1:'r',2:'a'}
```

```
if a==b:
    print a," and ",b," are same"
else:
    print a," and ",b," are not same"
```

```
a=set((1,2,3))
b=set((1,2,3,3,3))
if a==b:
    print a," and ",b," are same"
else:
    print a," and ",b," are not same"
```

o/P:

```
{1: 'r', 2: 'a'} and {1: 'r', 2: 'a'} are same
```

```
set([1, 2, 3]) and set([1, 2, 3]) are same
```

Exceptions:

Exceptions indicate errors and break out of the normal control flow of a program. An exception is raised using the raise statement.

```
raise RuntimeError, "Unrecoverable Error"
```

To catch an exception, use the try and except statements, as shown here:

```
try:
```

```
    f = open('foo')
```

```
except IOError, e:
```

```
    print "Unable to open 'foo': ", e
```

Multiple exception-handling blocks are specified using multiple except clauses, as in the following example:

```
try:
```

```
    do something
```

```
except IOError, e:
```

```
    # Handle I/O error
```

```
...
```

```
except TypeError, e:
```

```
    # Handle Type error
```

```
...
```

```
except NameError, e:
```

```
    # Handle Name error
```

```
...
```

A single handler can catch multiple exception types like this:

```
try:
```

```
    do something
```

```
except (IOError, TypeError, NameError), e:
```

```
    # Handle I/O, Type, or Name errors
```

```
...
```

To ignore an exception, use the pass statement as follows:

```
try:
```

```
    do something
```

```
except IOError:
```

```
    pass # Do nothing (oh well).
```

To catch all exceptions, omit the exception name and value:


```
try:
do something
except:
print 'An error occurred'
```

The try statement also supports an else clause, which must follow the last except clause. This code is executed if the code in the try block doesn't raise an exception.

Here's an example:

```
try:
f = open('foo', 'r')
except IOError:
print 'Unable to open foo'
else:
data = f.read()
f.close()
```

The finally statement defines a cleanup action for code contained in a try block.

For example:

```
f = open('foo', 'r')
try:
# Do some stuff
...
finally:
f.close()
print "File closed regardless of what happened."
```

The finally clause isn't used to catch errors. Rather, it's used to provide code that must always be executed, regardless of whether an error occurs. If no exception is raised, the code in the finally clause is executed immediately after the code in the try block. If an exception occurs, control is first passed to the first statement of the finally clause. After this code has executed, the exception is re-raised to be caught by another exception handler. The finally and except statements cannot appear together within a single try statement.

Defining New Exceptions

All the built-in exceptions are defined in terms of classes. To create a new exception, create a new class definition that inherits from exceptions.Exception, such as the following:

```
import exceptions
```

Exception class

```
class NetworkError(Exception):
```

```
def __init__(self, args=None):
```

```
self.args = args
```

The name args should be used as shown. This allows the value used in the raise statement to be properly printed in tracebacks and other diagnostics. In other words, raise NetworkError, "Cannot find host."

creates an instance of NetworkError using the following call:

```
NetworkError("Cannot find host.")
```

The object that is created will print itself as "NetworkError: Cannot find host." If you use a name other than the self.args name or don't store the argument, this feature won't work correctly.

Range():

```
range(5, 10)
```

5 through 9

```
range(0, 10, 3)
```

0, 3, 6, 9

```
range(-10, -100, -30)
```

-10, -40, -70

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

```
>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
```

```
>>> for i in range(len(a)):
```

```
...     print(i, a[i])
```

```
...
```

0 Mary

1 had

2 a

3 little

4 lamb

In many ways the object returned by range() behaves as if it is a list, but in fact it isn't. It is an object which returns the successive items of the desired sequence when you iterate over it. We say such an object is iterable.

break and continue Statements, and else Clauses on Loops

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

This is exemplified by the following loop, which searches for prime numbers:

```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print(n, 'equals', x, '*', n//x)
            break
    else:
        print(n, 'is a prime number')
```

(Yes, this is the correct code. Look closely: the else clause belongs to the for loop, not the if statement.)

pass Statements

The pass statement does nothing. It can be used when a statement is required syntactically but the program requires no action. For example:

```
>>> while True:
...     pass # Busy-wait for keyboard interrupt (Ctrl+C)
...
```

This is commonly used for creating minimal classes:

```
>>> class MyEmptyClass:
...     pass
```

Defining Functions

The value of the function name has a type that is recognized by the interpreter as a user-defined function. This value can be assigned to another name which can then also be used as a function.

```
>>> fib
<function fib at 10042ed0>
>>> f = fib
>>> f(100)
0 1 1 2 3 5 8 13 21 34 55 89
```

In fact, even functions without a return statement do return a value, albeit a rather boring one. This value is called None (it's a built-in name).

Example:

<pre>def get_value(): n=input('Enter a Value:') return n print(get_value())</pre>	<pre>def get_value(): n=input('Enter a Value:') print(get_value())</pre>
Enter a Value:234 234	Enter a Value:234 None

It is simple to write a function that returns a list of the numbers of the Fibonacci series, instead of printing it:

```
>>> def fib2(n): # return Fibonacci series up to n
...     """Return a list containing the Fibonacci series up to n."""
...     result = []
...     a, b = 0, 1
...     while a < n:
...         result.append(a) # see below
...         a, b = b, a+b
...     return result
...

>>> f100 = fib2(100) # call it

>>> f100 # write the result
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Define functions with a variable number of arguments

1. Default Argument Values

The most useful form is to specify a default value for one or more arguments. This creates a function that can be called with fewer arguments than it is defined to allow. For example:

```
def ask_ok(prompt, retries=4, complaint='Yes or no, please!'):
    while True:
        ok = input(prompt)
        if ok in ('y', 'ye', 'yes'):
            return True
        if ok in ('n', 'no', 'nop', 'nope'):
            return False
```

```

retries = retries - 1
if retries < 0:
    try:
        raise IOError('refusenik user')
    except:
        print('IOError Exception')
    finally:
        print('Completed')
        exit()
print(complaint)

print(ask_ok('Enter a value:'))

```

Execution:

```

Enter a value:5
Yes or no, please!
Enter a value:3
Yes or no, please!
Enter a value:6
Yes or no, please!
Enter a value:yes
True

```

The default values are evaluated at the point of function definition in the defining scope, so that

```
i = 5
```

```

def f(arg=i):
    print(arg)

```

```

i = 6
f()

```

will print 5.

The default value is evaluated only once. This makes a difference when the default is a mutable object such as a list, dictionary, or instances of most classes.

```
def f(a, L=[]):  
    L.append(a)  
    return L
```

```
print(f(1))  
print(f(2))  
print(f(3))
```

This will print

```
[1]  
[1, 2]  
[1, 2, 3]
```

If you don't want the default to be shared between subsequent calls, you can write the function like this instead:

```
def f(a, L=None):  
    if L is None:  
        L = []  
    L.append(a)  
    return L
```

```
print(f(1))  
print(f(2))  
print(f(3))
```

This will print:

```
[1]  
[2]  
[3]
```

1. Keyword Arguments

Functions can also be called using **keyword arguments** of the form `kwarg=value`. For instance, the following function:

```
def parrot(voltage, state='a stiff', action='voom', type='Norwegian Blue'):  
    print("-- This parrot wouldn't", action, end=' ')  
    print("if you put", voltage, "volts through it.")  
    print("-- Lovely plumage, the", type)  
    print("-- It's", state, "!")
```


accepts one required argument (voltage) and three optional arguments (state, action, and type). This function can be called in any of the following ways:

```
parrot(1000) # 1 positional argument
parrot(voltage=1000) # 1 keyword argument
parrot(voltage=1000000, action='VOOOOOM') # 2 keyword arguments
parrot(action='VOOOOOM', voltage=1000000) # 2 keyword arguments
parrot('a million', 'bereft of life', 'jump') # 3 positional arguments
parrot('a thousand', state='pushing up the daisies') # 1 positional, 1 keyword
```

But all the following calls would be invalid:

```
parrot() # required argument missing
parrot(voltage=5.0, 'dead') # non-keyword argument after a keyword argument
parrot(110, voltage=220) # duplicate value for the same argument
parrot(actor='John Cleese') # unknown keyword argument
```

In a function call, keyword arguments must follow positional arguments.

Valid:	Invalid:
parrot(5.0, 'dead')	parrot(voltage=5.0, 'dead')
parrot(5.0, state='dead')	
parrot(voltage=5.0, state='dead')	

All the keyword arguments passed must match one of the arguments accepted by the function, and their order is not important.

```
parrot(actor='John Cleese') # unknown keyword argument
parrot(action='VOOOOOM', voltage=1000000) - Valid
```

No argument may receive a value more than once. Here's an example that fails due to this restriction:

```
>>> def function(a):
...     pass
...
>>> function(0, a=0)
Traceback (most recent call last):
File "<stdin>", line 1, in ?
TypeError: function() got multiple values for keyword argument 'a'
```

When a final formal parameter of the form `**name` is present, it receives a dictionary (see typesmapping) containing all keyword arguments except for those corresponding to a formal parameter. This may be combined with a formal parameter of the form `*name` (described in the next subsection) which receives a tuple containing the positional arguments beyond the formal parameter list. (`*name` must occur before `**name`.) For example, if we define a function like this:

```
def cheeseshop(kind, *arguments, **keywords):
    print("-- Do you have any", kind, "?")
    print("-- I'm sorry, we're all out of", kind)
    for arg in arguments:
        print(arg)
    print("-" * 40)
    keys = sorted(keywords.keys())
    for kw in keys:
        print(kw, ":", keywords[kw])
```

It could be called like this:

```
cheeseshop("Limburger", "It's very runny, sir.",
           "It's really very, VERY runny, sir.",
           shopkeeper="Michael Palin",
           client="John Cleese",
           sketch="Cheese Shop Sketch")
```

and of course it would print:
-- Do you have any Limburger ?
-- I'm sorry, we're all out of Limburger
It's very runny, sir.
It's really very, VERY runny, sir.

client : John Cleese
shopkeeper : Michael Palin
sketch : Cheese Shop Sketch

1. Arbitrary Argument Lists

Finally, the least frequently used option is to specify that a function can be called with an arbitrary number of arguments. These arguments will be wrapped up in a tuple (see [Tuples and Sequences](#)). 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))
```

Correct:

```
def concat(*args, sep="/"):  
    return sep.join(args)  
  
print(concat("earth", "mars", "venus"))  
print(concat("earth", "mars", "venus", sep="."))
```

Wrong1:

```
def concat(sep="/", *args):  
    return sep.join(args)  
  
print(concat("earth", "mars", "venus", sep="."))  
TypeError: concat() got multiple values for argument 'sep'
```

Wrong2:

```
def concat(sep="/", *args):  
    return sep.join(args)  
  
print(concat(sep=".", "earth", "mars", "venus"))
```

SyntaxError: non-keyword arg after keyword arg

1. Unpacking Argument Lists

```
>>> list(range(3, 6)) # normal call with separate arguments
[3, 4, 5]
>>> args = [3, 6]
>>> list(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='vroom'):
...     print("-- This parrot wouldn't", action, end=' ')
...     print("if you put", voltage, "volts through it.", end=' ')
...     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.
```

1. Lambda Forms

With the `lambda` keyword, small anonymous functions can be created. Note that the `lambda` definition does not include a `"return"` statement -- it always contains an expression which is returned.

Here's a function that returns the sum of its two arguments: `lambda a, b: a+b`.

Ex1: Normal usage

```
>>> def f(x): return x**2
...
>>> print f(8)
64
>>>
>>> g = lambda x: x**2
>>> print g(8)
64
```

Ex2: Nested scope

```
def make_incrementor(n):  
    return lambda x: x + n
```

```
f = make_incrementor(42)  
print(f(0))  
print(f(1))  
print(make_incrementor(110)(230))
```

O/P:

```
42  
43  
340
```

we don't even have to assign the function anywhere -- you can just use it instantly and forget it when it's not needed anymore.

Ex:

```
>>> foo = [2, 18, 9, 22, 17, 24, 8, 12, 27]  
>>>  
>>> print filter(lambda x: x % 3 == 0, foo)  
[18, 9, 24, 12, 27]  
>>>  
>>> print map(lambda x: x * 2 + 10, foo)  
[14, 46, 28, 54, 44, 58, 26, 34, 64]  
>>>  
>>> print reduce(lambda x, y: x + y, foo)  
139
```

We have more usages with lambda, like: filter(), map(), reduce(). These take **function & list** as parameters.

Ex:

```
>>> foo = [2, 18, 9, 22, 17, 24, 8, 12, 27]  
>>>  
>>> print filter(lambda x: x % 3 == 0, foo) -> used to filter the data  
[18, 9, 24, 12, 27]  
>>>  
>>> print map(lambda x: x * 2 + 10, foo) -> used to convert our list  
[14, 46, 28, 54, 44, 58, 26, 34, 64]  
>>>  
>>> print reduce(lambda x, y: x + y, foo)  
139
```

The `reduce()` function is called with the first two elements from the list, then with the result of that call and the third element, and so on, until all of the list elements have been handled.

Ex:

<pre>>>> sentence = 'It is raining cats and dogs' >>> words = sentence.split() >>> print words ['It', 'is', 'raining', 'cats', 'and', 'dogs'] >>> >>> lengths = map(lambda word: len(word), words) >>> print lengths [2, 2, 7, 4, 3, 4]</pre>	<pre>>>> print map(lambda w: len(w), 'It is raining cats and dogs'.split()) [2, 2, 7, 4, 3, 4]</pre>
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------

Ex:

```
str='my name is rajesh';
```

```
words=str.split();
```

```
print(words);
```

```
print(list(map(lambda x:x,words)))
```

```
print(list(map(lambda x:len(x),words)))
```

O/P:

```
['my', 'name', 'is', 'rajesh']
```

```
['my', 'name', 'is', 'rajesh']
```

```
[2, 4, 2, 6]
```

To find all mount points in our file system:

```
>>> import commands
```

```
>>>
```

```
>>> mount = commands.getoutput('mount -v')
```

```
>>> lines = mount.splitlines()
```

```
>>> points = map(lambda line: line.split()[2], lines)
```

```
>>>
```

```
>>> print points
```

```
['/', '/var', '/usr', '/usr/local', '/tmp', '/proc']
```

We could write all of above in one single statement, which increases compactness but reduces readability:

```
print map(lambda x: x.split()[2], commands.getoutput('mount -v').splitlines())  
['/', '/var', '/usr', '/usr/local', '/tmp', '/proc']
```


DATA STRUCTURES:

Lists:

1. `list.append(x) == a[len(a):] = [x]`

Ex:

`a=[1,2,3]`

`b=['a','b']`

`a.append(b)`

`print(a)`

O/P: `[1, 2, 3, ['a', 'b']]`

1. `list.extend(L) == a[len(a):] = L`

Ex:

`a=[1,2,3]`

`b=['a','b']`

`a.extend(b)`

`print(a)`

O/P:

`[1, 2, 3, 'a', 'b']`

`import re`

re.sub()

```
>>> help(re.sub)
```

```
Help on function sub in module re:
```

```
sub(pattern, repl, string, count=0, flags=0)
```

```
Return the string obtained by replacing the leftmost
non-overlapping occurrences of the pattern in string by the
replacement repl.  repl can be either a string or a callable;
if a string, backslash escapes in it are processed.  If it is
a callable, it's passed the match object and must return
a replacement string to be used.
```

```
>>> re.sub(r'([a-z]+) \1', r'\1', 'cat in the the hat')
'cat in the hat'
```

```
>>> re.sub(r'[a-z]*',r'the','rajesh is in the verizon')
'the the the the the'
>>> re.sub(r'raj',r'ram','rajesh is in the verizon')
'ramesh is in the verizon'
```

re.findall()

```
rajesh is in the verizon
>>> re.findall(r'raj', 'rajesh nickname is raj and rajesh works for verizon')
['raj', 'raj', 'raj']
>>> re.findall(r'\br[a-z]*', 'rajesh raj is raj raj and for raj')
['rajesh', 'raj', 'raj', 'raj', 'raj']
```

```
>>> 'tea for too'.replace('too', 'two')
'tea for two'
```

import random

```
>>> random.choice(['rajesh','anjaneyulu','vijaya','kalyani','ram','ani','shiv','shad','nithya'])
'shad'
>>> random.choice(['rajesh','anjaneyulu','vijaya','kalyani','ram','ani','shiv','shad','nithya'])
'ram'
```

```
>>> random.sample(range(10),4)
[8, 4, 1, 9]
>>> random.sample(range(10),3)
[5, 7, 2]
```

```
>>> random.random()
0.5915703039141714
>>> random.randrange(10)
7
```

List all files from current Directory and its subdirectory files:

def print_directory_contents(sPath):

import os

for sChild in os.listdir(sPath):

sChildPath = os.path.join(sPath,sChild)

```
if os.path.isdir(sChildPath):
    print_directory_contents(sChildPath)
else:
    print(sChildPath)
```

Decorators:

```
@my_decorator
def my_func(stuff):
    do_things
```

Is equivalent to

```
def my_func(stuff):
    do_things

my_func = my_decorator(my_func)
```

Sending mail in Python:

```
>>> import smtplib
>>> server=smtplib.SMTP('smtp.gmail.com', 587)
>>> server.starttls()
(220, b'2.0.0 Ready to start TLS')
>>> server.login('pythontestuser@gmail.com', 'pythonisamazing')
(235, b'2.7.0 Accepted')
>>> server.sendmail('pythontestuser@gmail.com', 'madhur.bhatia@hotmail.com', 'Hello World!')
{}
.
```

[A guide to Python's function decorators](#)

Python is rich with powerful features and expressive syntax. In the context of design patterns, decorators dynamically alter the functionality of a function, method or class without having to directly use subclasses. This is ideal when you need to extend the functionality of functions that you don't want to modify. We can implement the decorator pattern anywhere, but Python facilitates the implementation by providing much more expressive features and syntax for that.

In this post I will be discussing Python's function decorators in depth, accompanied by a bunch of examples on the way to clear up the concepts. All examples are in Python 2.7 but the same concepts should apply to Python 3 with some change in the syntax.

Essentially, decorators work as wrappers, modifying the behavior of the code before and after a target function execution, without the need to modify the function itself, augmenting the original functionality, thus decorating it.

What you need to know about functions

Before diving in, there are some prerequisites that should be clear. In Python, functions are first class citizens, they are objects and that means we can do a lot of useful stuff with them.

Assign functions to variables

```
def greet(name):  
    return "hello "+name
```

```
greet_someone = greet  
print greet_someone("John")
```

```
# Outputs: hello John
```

Define functions inside other functions

```
def greet(name):  
    def get_message():  
        return "Hello "
```

```
    result = get_message()+name  
    return result
```

```
print greet("John")
```

```
# Outputs: Hello John
```

Functions can be passed as parameters to other functions

```
def greet(name):  
    return "Hello " + name
```

```
def call_func(func):  
    other_name = "John"  
    return func(other_name)
```

```
print call_func(greet)
```

```
# Outputs: Hello John
```

Functions can return other functions

In other words, functions generating other functions.

```
def compose_greet_func():  
    def get_message():  
        return "Hello there!"
```

```
    return get_message
```

```
greet = compose_greet_func()  
print greet()
```

```
# Outputs: Hello there!
```

Inner functions have access to the enclosing scope

More commonly known as a **closure**. A very powerful pattern that we will come across while building decorators. Another thing to note, Python only allows **read access to the outer scope** and not assignment. Notice how we modified the example above to read a "name" argument from the enclosing scope of the inner function and return the new function.

```
def compose_greet_func(name):  
    def get_message():  
        return "Hello there "+name+"!"
```

```
    return get_message
```

```
greet = compose_greet_func("John")
print greet()
```

```
# Outputs: Hello there John!
```

Composition of Decorators::

Function decorators are simply wrappers to existing functions. Putting the ideas mentioned above together, we can build a decorator. In this example let's consider a function that wraps the string output of another function by **p** tags.

```
def get_text(name):
    return "lorem ipsum, {0} dolor sit amet".format(name)
```

```
def p_decorate(func):
    def func_wrapper(name):
        return "<p>{0}</p>".format(func(name))
    return func_wrapper
```

```
my_get_text = p_decorate(get_text)
```

```
print my_get_text("John")
```

```
# <p>Outputs lorem ipsum, John dolor sit amet</p>
```

That was our first decorator. A function that takes another function as an argument generates a new function, augmenting the work of the original function, and returning the generated function so we can use it anywhere. To have `get_text` itself be decorated by `p_decorate`, we just have to assign `get_text` to the result of `p_decorate`.

```
get_text = p_decorate(get_text)
print get_text("John")
```

```
# Outputs lorem ipsum, John dolor sit amet
```

Another thing to notice is that our decorated function takes a name argument. All what we had to do in the decorator is to let the wrapper of `get_text` pass that argument.

Python's Decorator Syntax

Python makes creating and using decorators a bit cleaner and nicer for the programmer through some **syntactic sugar**. To decorate `get_text` we don't have to `get_text = p_decorator(get_text)`. There is a neat shortcut for that, which is to mention the name of the decorating function before the function to be decorated. The name of the decorator should be prepended with an `@` symbol.

```
def p_decorate(func):
    def func_wrapper(name):
        return "<p>{0}</p>".format(func(name))
    return func_wrapper

@p_decorate
def get_text(name):
    return "lorem ipsum, {0} dolor sit amet".format(name)

print get_text("John")

# Outputs <p>lorem ipsum, John dolor sit amet</p>
```

Now let's consider we wanted to decorate our `get_text` function by 2 other functions to wrap a `div` and `strong` tag around the string output.

```
def p_decorate(func):
    def func_wrapper(name):
        return "<p>{0}</p>".format(func(name))
    return func_wrapper

def strong_decorate(func):
    def func_wrapper(name):
        return "<strong>{0}</strong>".format(func(name))
    return func_wrapper

def div_decorate(func):
    def func_wrapper(name):
        return "<div>{0}</div>".format(func(name))
    return func_wrapper
```

With the basic approach, decorating `get_text` would be along the lines of


```
get_text = div_decorate(p_decorate(strong_decorate(get_text)))
```

With Python's decorator syntax, same thing can be achieved with much more expressive power.

```
@div_decorate
@p_decorate
@strong_decorate
def get_text(name):
    return "lorem ipsum, {0} dolor sit amet".format(name)
```

```
print get_text("John")
```

```
# Outputs <div><p><strong>lorem ipsum, John dolor sit amet</strong></p></div>
```

One important thing to notice here is that the order of setting our decorators matters. If the order was different in the example above, the output would have been different.

Decorating Methods::

In Python, methods are functions that expect their first parameter to be a reference to the current object. We can build decorators for methods the same way, while taking **self** into consideration in the wrapper function.

```
def p_decorate(func):
    def func_wrapper(self):
        return "<p>{0}</p>".format(func(self))
    return func_wrapper
```

```
class Person(object):
    def __init__(self):
        self.name = "John"
        self.family = "Doe"
```

```
@p_decorate
def get_fullname(self):
    return self.name+" "+self.family
```

```
my_person = Person()
print my_person.get_fullname()
```

A much better approach would be to make our decorator useful for functions and methods alike. This can be done by putting ***args and **kwargs** as parameters for the wrapper, then it can accept any arbitrary number of arguments and keyword arguments.

```
def p_decorate(func):  
    def func_wrapper(*args, **kwargs):  
        return "<p>{0}</p>".format(func(*args, **kwargs))  
    return func_wrapper
```

```
class Person(object):  
    def __init__(self):  
        self.name = "John"  
        self.family = "Doe"
```

```
@p_decorate  
def get_fullname(self):  
    return self.name + " " + self.family
```

```
my_person = Person()
```

```
print my_person.get_fullname()
```

Passing arguments to decorators

Looking back at the example before the one above, you can notice how redundant the decorators in the example are. 3 decorators (div_decorate, p_decorate, strong_decorate) each with the same functionality but wrapping the string with different tags. We can definitely do much better than that. Why not have a more general implementation for one that takes the tag to wrap with as a string? Yes please!

```
def tags(tag_name):  
    def tags_decorator(func):  
        def func_wrapper(name):  
            return "<{0}>{1}</{0}>".format(tag_name, func(name))  
        return func_wrapper  
    return tags_decorator
```

```
@tags("p")  
def get_text(name):
```

```
return "Hello "+name
```

```
print get_text("John")
```

```
# Outputs <p>Hello John</p>
```

It took a bit more work in this case. Decorators expect to receive a function as an argument, that is why we will have to build a function that takes those extra arguments and generate our decorator on the fly. In the example above **tags**, is our decorator generator.

Debugging decorated functions

At the end of the day decorators are just wrapping our functions, in case of debugging that can be problematic since the wrapper function does not carry the name, module and docstring of the original function. Based on the example above if we do:

```
print get_text.__name__  
# Outputs func_wrapper
```

The output was expected to be **get_text** yet, the attributes `__name__`, `__doc__`, and `__module__` of **get_text** got overridden by those of the wrapper(func_wrapper). Obviously we can re-set them within func_wrapper but Python provides a much nicer way.

Functools to the rescue

Fortunately Python (as of version 2.5) includes the **functools** module which contains **functools.wraps**. Wraps is a decorator for updating the attributes of the wrapping function(func_wrapper) to those of the original function(get_text). This is as simple as decorating func_wrapper by @wraps(func). Here is the updated example:

```
from functools import wraps
```

```
def tags(tag_name):  
    def tags_decorator(func):  
        @wraps(func)  
        def func_wrapper(name):  
            return "<{0}>{1}</{0}>".format(tag_name, func(name))  
        return func_wrapper  
    return tags_decorator
```

```

@tags("p")
def get_text(name):
    """returns some text"""
    return "Hello "+name

print get_text.__name__ # get_text
print get_text.__doc__ # returns some text
print get_text.__module__ # __main__

```

You can notice from the output that the attributes of `get_text` are the correct ones now.

Where to use decorators

The examples in this post are pretty simple relative to how much you can do with decorators. They can give so much power and elegance to your program. In general, decorators are ideal for extending the behavior of functions that we don't want to modify. For a great list of useful decorators I suggest you check out the [Python Decorator Library](#)

`object.__repr__(self)`: called by the `repr()` built-in function and by string conversions (reverse quotes) to compute the "official" string representation of an object.

`object.__str__(self)`: called by the `str()` build-in function and by the `print` statement to compute the "informal" string representation of an object.

For int:

```

>>> str(1)==repr(1)
True
-----
>>> str(1)==eval(repr(1))
False

```

For Strings:

```

-----
>>> a='raj'
>>> str(a)==repr(a)
False
>>> str(a)==eval(repr(a))
True

```

```

>>> str(a)
'raj'
>>> eval(str(a))
Traceback (most recent call last):
  File "<pyshell#38>", line 1, in <module>
    eval(str(a))
  File "<string>", line 1, in <module>
NameError: name 'raj' is not defined
>>> eval(repr(a))
'raj'

```

Therefore, a "formal" representation of an object should be callable by **eval()** and return the same object, if possible.

```

>>> class A(object):
    def method1():
        print('Class A: Method1')

```

```

>>> class B(A):
    def method1():
        print('Class B: Method 1')

```

```

>>> class C(A,B):
    def method1():
        print('class C: method 1')

```

```

Traceback (most recent call last):
  File "<pyshell#698>", line 1, in <module>
    class C(A,B):
TypeError: Cannot create a consistent method resolution
order (MRO) for bases B, A

```

```

>>> class C(B,A):
    def method1():
        print('class C: method 1')

```

```

>>> C.__mro__

```

```
(<class '__main__.C'>, <class '__main__.B'>, <class '__main__.A'>, <class 'object'>)
```

```
>>> class C(B):
```

```
    def method1():
```

```
        print('class C: method 1')
```

```
>>> C.__mro__
```

```
(<class '__main__.C'>, <class '__main__.B'>, <class '__main__.A'>, <class 'object'>)
```

```
>>> class C(A):
```

```
    def method1():
```

```
        print('class C: method 1')
```

```
>>> C.__mro__
```

```
(<class '__main__.C'>, <class '__main__.A'>, <class 'object'>)
```

To get attribute of a class:

```
>>> A().__getattr__('name')
```

```
'rajesh'
```

To get IP address:

```
>>> a
```

```
'date:09:07:2016 Time: 06:46 PM. we have logged into the IP 10.73.124.62 and while  
logging to the IP 10.73.124.73, you got error'
```

```
>>> re.findall(r'\d+[\.]\d+[\.]\d+[\.]\d+',a)
```

```
['10.73.124.62', '10.73.124.73']
```

```
>>> re.findall(r'\d+\.\d+\.\d+\.\d+',a)
```

```
['10.73.124.62', '10.73.124.73']
```

To validate IP:

```
>>> import socket
```

```
>>> for ip in re.findall(r'\d+\.\d+\.\d+\.\d+',a):
```

```
    print(socket.inet_aton(ip))
```

```
b'\n|>'
```

```
b'\n| |'
```

```
>>> socket.inet_aton('999.10.20.30')
Traceback (most recent call last):
  File "<pyshell#648>", line 1, in <module>
    socket.inet_aton('999.10.20.30')
OSError: illegal IP address string passed to inet_aton
```

collections.Counter():

```
>>> a='rrrraaaajjjjeeessssshhhh'
>>> a
'rrrraaaajjjjeeessssshhhh'
```

```
>>> collections.Counter(a)
Counter({'s': 6, 'r': 4, 'h': 4, 'j': 4, 'a': 4, 'e': 4})
>>> collections.Counter(a).most_common()
[('s', 6), ('r', 4), ('h', 4), ('j', 4), ('a', 4), ('e', 4)]
>>> collections.Counter(a).most_common(2)
[('s', 6), ('r', 4)]
>>> a='rrrraaaajjjjeeessssshhhh'
>>> a='rrrraaaajjjjeeessssshhhh'
>>> collections.Counter(a).most_common(2)
[('s', 6), ('h', 5)]
```

itertools.combinations()

```
>>> itertools.combinations('ABCD', 2)
<itertools.combinations object at 0x02FF5C00>
>>> list(itertools.combinations('ABCD', 2))
[('A', 'B'), ('A', 'C'), ('A', 'D'), ('B', 'C'), ('B', 'D'), ('C', 'D')]
>>> itertools.combinations('ABCD', 3)
<itertools.combinations object at 0x02FF5C60>
>>> list(itertools.combinations('ABCD', 3))
[('A', 'B', 'C'), ('A', 'B', 'D'), ('A', 'C', 'D'), ('B', 'C', 'D')]
>>> list(itertools.combinations('ABCD', 1))
[('A',), ('B',), ('C',), ('D',)]
>>> list(itertools.combinations('ABCD', 4))
[('A', 'B', 'C', 'D')]
```

```
>>> list(itertools.combinations('ABCD', 2))
[('A', 'B'), ('A', 'C'), ('A', 'D'), ('B', 'C'), ('B', 'D'), ('C', 'D')]
>>> list(itertools.combinations('ABCD', 5))
[]
>>> list(itertools.combinations('ABCD', 0))
[()]
>>> list(itertools.combinations('ABCD', 2))
[('A', 'B'), ('A', 'C'), ('A', 'D'), ('B', 'C'), ('B', 'D'), ('C', 'D')]
```

```
>>> a=[[1, 2], [3, 4], [5, 6]]
>>> a
[[1, 2], [3, 4], [5, 6]]
>>> itertools.chain(*a)
<itertools.chain object at 0x02F4BDB0>
```

```
>>> list(itertools.chain(a))
[[1, 2], [3, 4], [5, 6]]
```

<pre>>>> list(itertools.chain(*a)) [1, 2, 3, 4, 5, 6]</pre>
<pre>>>> list(functools.reduce(lambda x,y: x+y, a)) [1, 2, 3, 4, 5, 6]</pre>
<pre>>>> list(itertools.chain.from_iterable(a)) [1, 2, 3, 4, 5, 6]</pre>


```
>>> a=[[1,2],[3,4],[5,6],[7,8]]
>>> list(functools.reduce(lambda x,y: x+y, a))
[1, 2, 3, 4, 5, 6, [7, 8]]
>>> list(itertools.chain(*a))
[1, 2, 3, 4, 5, 6, [7, 8]]
>>> list(itertools.chain.from_iterable(a))
[1, 2, 3, 4, 5, 6, [7, 8]]
```

```
>>> import os
>>> os.__file__
'C:\\Python34\\lib\\os.py'
```

Inheritance

If a derived class defines an attribute with the same name as an attribute in a base class, instances of the derived class use the attributes in the derived class. If it's ever necessary to access the original attribute, a fully qualified name can be used as follows:

```
class D(A):
    def method1(self):
        print "Class D : method1"
        A.method1(self) # Invoke base class method
```

One of the most common applications of this is in the initialization of class instances. When an instance is created, the `__init__()` methods of base classes are not invoked. Therefore, it's up to a derived class to perform the proper initialization of its base classes, if necessary. For example:

```
class D(A):
    def __init__(self, args1):
        # Initialize the base class
        A.__init__(self)
        # Initialize myself
    ...
```

Python provides a function, `super(class,obj)`, that can be used to call methods in a superclass. This function is most useful if you want to invoke a method in one of the parent classes without having to reimplement Python's method resolution algorithm.

For example:

```
class D(A,B):
    def method1(self):
        print "Class D : method1"
        super(D,self).method1() # Invoke appropriate base class method
```

Polymorphism (*dynamic binding*): refers to the ability of an object to adapt the code to the type of the data it is processing.

Ex:

```
>>> def sum(a,b):
    try:
        return a+b
    except:
        print('Incompatible parameters passed')
```

```
>>> sum(2,3)
5
>>> sum('rajesh',' anusha')
'rajesh anusha'
>>> sum('rajesh',5)
Incompatible parameters passed
```

```
>>> num='123'
>>> len(num)
3
>>> num.__len__
<method-wrapper '__len__' of str object at 0x03007BA0>
>>> num.__len__()
3
```

Open XML File:

```

from xml.dom import minidom
Test_file = open('C:\\Users\\v528250\\Desktop\\test_file.xml','r')
xmldoc = minidom.parse(Test_file)

```

```

print('=====')
print(xmldoc.documentElement)
print(xmldoc.documentElement.childNodes)
print('=====')

```

```
Test_file.close()
```

```

def printNode(node):
    print (node)
    for child in node.childNodes:
        printNode(child)

```

```
printNode(xmldoc.documentElement)
```

When executed above code on below data of xml, o/p is:

Sample xml file content:

```

<a>
  <b>testing 1</b>
  <c>testing 2</c>
</a>

```

```

C:\Users\v528250\Desktop>python 1.py
=====
<DOM Element: a at 0x28ed620>
[<DOM Text node "'\n  '">, <DOM Element: b at 0x28ed580>, <DOM Text node "'\n  '">, <DOM Element: c at 0x28edd00>, <DOM Text node "'\n'">]
=====
<DOM Element: a at 0x28ed620>
<DOM Text node "'\n  '">
<DOM Element: b at 0x28ed580>
<DOM Text node "'testing 1'">
<DOM Text node "'\n  '">
<DOM Element: c at 0x28edd00>
<DOM Text node "'testing 2'">
<DOM Text node "'\n'">

```

By default, all attributes are “public.” This means that all attributes of a class instance are accessible without any restrictions. It also implies that everything defined in a base class is inherited and accessible within a derived class. This behavior is often undesirable in object-oriented applications because it exposes the internal implementation of an object and can lead to namespace conflicts between objects defined in a derived class and those defined in a base class.

To fix this problem, all names in a class that start with a double underscore, such as `__Foo`, are automatically mangled to form a new name of the form `_Classname__Foo`. This effectively provides a way for a class to have private attributes, because private names used in a derived class won’t collide with the same private names used in a base class. For example:

```
class A(object):
    def __init__(self):
        self.__X = 3 # Mangled to self._A__X
class B(A):
    def __init__(self):
        A.__init__(self)
        self.__X = 37 # Mangled to self._B__X
```

```
class A(object): pass
class B(A): pass
class C(object): pass
```

```
a = A() # Instance of 'A'
b = B() # Instance of 'B'
c = C() # Instance of 'C'
```

```
type(a) # Returns the class object A
```

```
isinstance(a,A) # Returns True
```

```
isinstance(b,A) # Returns True, B derives from A
```

```
isinstance(b,C) # Returns False, C not derived from A
```

Similarly, the built-in function `issubclass(A,B)` returns True if the class A is a subclass of class B. For example:

```
issubclass(B,A) # Returns True
issubclass(C,A) # Returns False
```

Metaclasses

When you define a class in Python, the class definition itself becomes an object. For example:

```
class Foo(object): pass
instance(Foo,object) # Returns True
```

If you think about this long enough, you will realize that something had to create the Foo object. This creation of the class object is controlled by a special kind of object called a *metaclass*.

collections Library:

```
>>> dir(collections)
```

```
['ByteString', 'Callable', 'ChainMap', 'Container', 'Counter', 'Hashable', 'ItemsView', 'Iterable',
'Iterator', 'KeysView', 'Mapping', 'MappingView', 'MutableMapping', 'MutableSequence',
'MutableSet', 'OrderedDict', 'Sequence', 'Set', 'Sized', 'UserDict', 'UserList', 'UserString',
'ValuesView', '_Link', '__all__', '__builtins__', '__cached__', '__doc__', '__file__',
'__loader__', '__name__', '__package__', '__path__', '__spec__', '_chain', '_class_template',
'_collections_abc', '_count_elements', '_eq', '_field_template', '_heapq', '_iskeyword',
'_itemgetter', '_proxy', '_recursive_repr', '_repeat', '_repr_template', '_starmap', '_sys',
'abc', 'defaultdict', 'deque', 'namedtuple']
```

<u>namedtuple()</u>	factory function for creating tuple subclasses with named fields
<u>deque</u>	list-like container with fast appends and pops on either end
<u>ChainMap</u>	dict-like class for creating a single view of multiple mappings
<u>Counter</u>	dict subclass for counting hashable objects
<u>OrderedDict</u>	dict subclass that remembers the order entries were added
<u>defaultdict</u>	dict subclass that calls a factory function to supply missing values
<u>UserDict</u>	wrapper around dictionary objects for easier dict subclassing
<u>UserList</u>	wrapper around list objects for easier list subclassing
<u>UserString</u>	wrapper around string objects for easier string subclassing

collections.namedtuple(*typename*, *field_names*, *verbose=False*, *rename=False*)[1](#): Returns a new tuple subclass named *typename*. The new subclass is used to create tuple-like objects that have fields accessible by attribute lookup as well as being indexable and iterable.

The *field_names* are a single string with each fieldname separated by whitespace and/or commas, for example 'x y' or 'x, y'. Alternatively, *field_names* can be a sequence of strings such as ['x', 'y'].

```
>>> collections.namedtuple('mytuple', 'x y')
<class '__main__.mytuple'>
>>> collections.namedtuple('mytuple', 'x,y')
<class '__main__.mytuple'>
>>> collections.namedtuple('mytuple', '_x,y')
Traceback (most recent call last):
  File "<pyshell#717>", line 1, in <module>
    collections.namedtuple('mytuple', '_x,y')
  File "C:\Python34\lib\collections\__init__.py", line 348, in namedtuple
    '%r' % name)
ValueError: Field names cannot start with an underscore: '_x'
```

```
>>> Point = collections.namedtuple('mytuple', ['x', 'y'])
>>> p=Point(11,y=23)
>>> p[0]
11
>>> p[1]
23
>>> p=mytuple(11,y=23)
Traceback (most recent call last):
  File "<pyshell#729>", line 1, in <module>
    p=mytuple(11,y=23)
NameError: name 'mytuple' is not defined
>>> p
mytuple(x=11, y=23)
```

unpack like a regular tuple

```
>>> q,r=p
>>> q
11
>>> r
23
```

```
>>> t=collections.namedtuple('person','name age')
>>> t
<class '__main__.person'>
>>> t.name='Rajesh'
>>> t.age=29
>>> t
<class '__main__.person'>
>>> t.name
'Rajesh'
>>> t.age
29
```

```
EmployeeRecord = namedtuple('EmployeeRecord', 'name, age, title, department, paygrade')

import csv
for emp in map(EmployeeRecord._make, csv.reader(open("employees.csv", "rb"))):
    print(emp.name, emp.title)

import sqlite3
conn = sqlite3.connect('/companydata')
cursor = conn.cursor()
cursor.execute('SELECT name, age, title, department, paygrade FROM employees')
for emp in map(EmployeeRecord._make, cursor.fetchall()):
    print(emp.name, emp.title)
```

```
>>> p=collections.namedtuple('mytuple','x,y,z')
>>> d={'x':100,'y':300,'z':213}
>>> p(**d)
mytuple(x=100, y=300, z=213)
```

```
>>> p
<class '__main__.mytuple'>
>>> p._fields
('x', 'y', 'z')
```

```
>>> p._source
"from builtins import property as _property, tuple as _tuple\nfrom operator import i
tter as _itemgetter\nfrom collections import OrderedDict\n\nclass mytuple(tuple):\n
    ytuple(x, y, z)\n\n    __slots__ = ()\n\n    __fields__ = ('x', 'y', 'z')\n\n    def _
_(_cls, x, y, z):\n        'Create new instance of mytuple(x, y, z)'\n        return
le._new__(_cls, (x, y, z))\n\n    @classmethod\n    def _make(cls, iterable, new=tu
ple._new_, len=len):\n        'Make a new mytuple object from a sequence or iterable'\n
        result = new(cls, iterable)\n        if len(result) != 3:\n            raise TypeE
rror\n        'Expected 3 arguments, got %d' % len(result)\n        return result\n\n    def _rep
_(_self, **kwargs):\n        'Return a new mytuple object replacing specified fields with
values'\n        result = _self._make(map(kwds.pop, ('x', 'y', 'z'), _self))\n        k
wds:\n            raise ValueError('Got unexpected field names: %r' % list(kwds))\n"
```

class collections.deque([iterable[, maxlen]]): Double Ended Queue {deck}

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.

Deques are a generalization of stacks and queues (the name is pronounced “deck” and is short for “double-ended queue”). Deques support thread-safe, memory efficient appends and pops from either side of the deque with approximately the same $O(1)$ performance in either direction.

Though list objects support similar operations, they are optimized for fast fixed-length operations and incur $O(n)$ memory movement costs for `pop(0)` and `insert(0, v)` operations which change both the size and position of the underlying data representation.

Deque objects support the following methods:

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.

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.

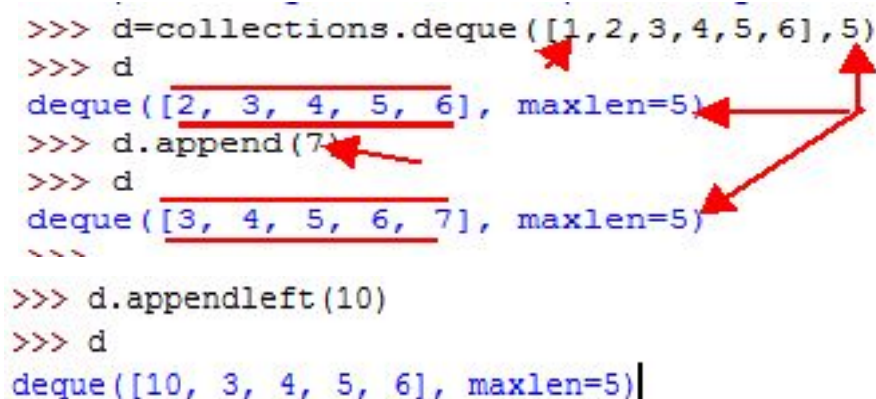
popleft(): Remove and return an element from the left side of the deque. If no elements are present, raises an IndexError.

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())`.

```
>>> d=collections.deque([1,2,3,4,5,6],5)
>>> d
deque([2, 3, 4, 5, 6], maxlen=5)
>>> d.append(7)
>>> d
deque([3, 4, 5, 6, 7], maxlen=5)
...
>>> d.appendleft(10)
>>> d
deque([10, 3, 4, 5, 6], maxlen=5)
```




```

>>> d=collections.deque(
>>> d
deque([])
>>> l=[]
>>> l.append(3)
>>> d.append(3)
>>> l
[3]
>>> d
deque([3])
>>> d.appendleft(5)
>>> l.insert(0,5)
>>> l
[5, 3]
>>> d
deque([5, 3])
>>> l.count(3)
1
>>> d.count(3)
1
>>> print(*'rajesh')
r a j e s h

>>> d.extend([9,10])
>>> l.extend([9,10])
...


>>> d.extendleft([23,34,45])
>>> l
[5, 3, 9, 10]
>>> d
deque([45, 34, 23, 5, 3, 9, 10])
...
>>> d
deque([45, 34, 23, 5, 3, 9])
>>> d.pop()
9
>>> d
deque([45, 34, 23, 5, 3])
>>> d.popleft()
45
>>> d
deque([34, 23, 5, 3])
...

```


```
>>> l
[5, 3, 9, 10]
>>> d
deque([34, 23, 5, 3])
>>> l.remove(9)
>>> d.remove(23)
>>> l
[5, 3, 10]
>>> d
deque([34, 5, 3])

>>> d
deque([3, 5, 34])
>>> d.rotate()
>>> d
deque([34, 3, 5])
```

```
>>> d=collections.deque()
>>> d
deque([])
>>> d.append([4,6,3,67,3,74,3,54,7,3,])
>>> d
deque([[4, 6, 3, 67, 3, 74, 3, 54, 7, 3]])
>>>
```



```
>>> d=collections.deque()
>>> d.extend([4,9,34,76,43])
>>> d
deque([4, 9, 34, 76, 43])
>>> d.rotate()
>>> d
deque([43, 4, 9, 34, 76])
>>> d.appendleft(d.pop())
>>> d
deque([76, 43, 4, 9, 34])
>>>
```



```

>>> d
deque([76, 43, 4, 9, 34])
>>> print(d.maxlen)
None
>>> d[0:3]
Traceback (most recent call last):
  File "<pyshell#389>", line 1, in <module>
    d[0:3]
TypeError: sequence index must be integer, not 'slice'
>>> d[0]
76

>>> d
deque([76, 43, 4, 9, 34])
>>> d*3
Traceback (most recent call last):
  File "<pyshell#392>", line 1, in <module>
    d*3
TypeError: unsupported operand type(s) for *: 'collections.deque' and 'int'
>>> d+d
Traceback (most recent call last):
  File "<pyshell#393>", line 1, in <module>
    d+d
TypeError: unsupported operand type(s) for +: 'collections.deque' and 'collections.deque'
...

>>> d
deque([10, 3, 4, 5, 6], maxlen=5)
>>> d.clear()
>>> d
deque([], maxlen=5)
...

>>> d = collections.deque('ghi')
>>> d
deque(['g', 'h', 'i'])
>>> collections.Counter(d)
Counter({'g': 1, 'h': 1, 'i': 1})

>>> d
deque(['g', 'h', 'i'])
>>> reversed(d)
<_collections._deque_reverse_iterator object at 0x0300A420>
>>> list(reversed(d))
['i', 'h', 'g']

```

```
>>> d
deque(['g', 'h', 'i'])
>>> d.rotate(1)
>>> d
deque(['i', 'g', 'h'])
>>> d.rotate(-1)
>>> d
deque(['g', 'h', 'i'])
>>> d.rotate(-1)
>>> d
deque(['h', 'i', 'g'])
...

```

```
def tail(filename, n=10):
    'Return the last n lines of a file'
    with open(filename) as f:
        return deque(f, n)

```

Below is equivalent to `del d[n]`:

```
def delete_nth(d, n):
    d.rotate(-n)
    d.popleft()
    d.rotate(n)

```

```
>>> d
deque(['h', 'i', 'g'])
>>> del d[2]
>>> d
deque(['h', 'i'])

```

To check valid variable name:

```
>>> '5'.isidentifier()
False
>>> 'x'.isidentifier()
True

```

class collections.ChainMap(*maps)[¶](#): A [ChainMap](#) class is provided for quickly linking a number of mappings so they can be treated as a single unit. It is often much faster than creating a new dictionary and running multiple [update\(\)](#) calls.

A [ChainMap](#) groups multiple dicts or other mappings together to create a single, updateable view. If no *maps* are specified, a single empty dictionary is provided so that a new chain always has at least one mapping. All of the usual dictionary methods are supported.

maps: A user updateable list of mappings. The list is ordered from first-searched to last-searched. It is the only stored state and can be modified to change which mappings are searched. The list should always contain at least one mapping.

```
>>> d={'one':1,'two':2}
>>> d1={1:'one',2:'two'}
>>> d
{'two': 2, 'one': 1}
>>> d1
{1: 'one', 2: 'two'}
>>> pychain=collections.ChainMap(d,d1)
>>> pychain
ChainMap({'two': 2, 'one': 1}, {1: 'one', 2: 'two'})
```

```
>>> a = {'a': 'A', 'c': 'C'}
>>> b = {'b': 'B', 'c': 'D'}
>>> a
{'a': 'A', 'c': 'C'}
>>> b
{'b': 'B', 'c': 'D'}
>>> m = collections.ChainMap(a, b)
>>> m
ChainMap({'a': 'A', 'c': 'C'}, {'b': 'B', 'c': 'D'})
>>> print('Individual Values')
Individual Values
>>> print('a = {}'.format(m['a']))
a = A
>>> print('b = {}'.format(m['b']))
b = B
>>> print('Items:')
Items:
>>> for k, v in m.items():
    print('{} = {}'.format(k, v))
```

```
a = A
b = B
c = C
```

class `collections.Counter([iterable-or-mapping])`[1](#)

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.

```
>>> import collections
>>> collections.Counter('gallahad')
Counter({'a': 3, 'l': 2, 'g': 1, 'h': 1, 'd': 1})
>>> collections.Counter({'red': 4, 'blue': 2})
Counter({'red': 4, 'blue': 2})
>>> collections.Counter(cats=4, dogs=8)
Counter({'dogs': 8, 'cats': 4})

>>> l=['a','a','b','c','c','c',5]
>>> collections.Counter(l)
Counter({'c': 3, 'a': 2, 'b': 1, 5: 1})
>>> collections.Counter(l)[0]

>>> collections.Counter(l)+collections.Counter(l)
Counter({'c': 6, 'a': 4, 'b': 2, 5: 2})

>>> collections.Counter(l)-collections.Counter(l)
Counter()
```

elements()[1](#): 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 = collections.Counter(a=4, b=2, c=0, d=-2)
>>> c
Counter({'a': 4, 'b': 2, 'c': 0, 'd': -2})
>>> print(*c.elements())
b b a a a a

>>> l=['a','a','b','c','c','c',5]
>>> c=collections.Counter(l)
>>> c
Counter({'c': 3, 'a': 2, 'b': 1, 5: 1})
>>> print(c.elements())
<itertools.chain object at 0x02FD54D0>
>>> print(*c.elements())
a a b 5 c c c
>>>
```

```
>>> prime_factors = collections.Counter({2: 2, 3: 3, 17: 1})
>>> product=1
>>> for val in prime_factors.elements():
    product*=val

>>> product
1836
```


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:

collections.Counter() → Gives dict output


collections.most_common() → Gives tuples output in a list

```
if n is None:
    return sorted(self.items(), key=_itemgetter(1), reverse=True)
return _heapq.nlargest(n, self.items(), key=_itemgetter(1))
```

```
>>> l=[1, 2, 3, 4, 5, 2, 3, 4, 2, 3, 4]
>>> collections.Counter(l)
Counter({2: 3, 3: 3, 4: 3, 1: 1, 5: 1})
>>> collections.Counter(l).most_common()
[(2, 3), (3, 3), (4, 3), (1, 1), (5, 1)]
>>> from operator import itemgetter
>>> sorted(collections.Counter(l).items(),key=itemgetter(1),reverse=True)
[(2, 3), (3, 3), (4, 3), (1, 1), (5, 1)]
```



```
>>> l
[1, 2, 3, 4, 5, 2, 3, 4, 2, 3, 4]
>>> collections.Counter(l)
Counter({2: 3, 3: 3, 4: 3, 1: 1, 5: 1})
>>> collections.Counter(l).most_common(2)
[(2, 3), (3, 3)]
>>> sorted(collections.Counter(l).items(),key=itemgetter(1),reverse=True)[:2]
[(2, 3), (3, 3)]
```



```
>>> c.most_common()
[('c', 3), ('a', 2), ('b', 1), (5, 1)]
>>> c.most_common(2)
[('c', 3), ('a', 2)]
```



```
>>> c
Counter({'c': 3, 'a': 2, 'b': 1, 5: 1})
>>> sorted(c, key=c.get)
['b', 5, 'a', 'c']
```

```
>>> sorted(c.items(), key=lambda x: c.get(x[0]))
[('b', 1), (5, 1), ('a', 2), ('c', 3)]
```

```
>>> l=[1,2,3,4,5,2,3,4,2,3,4]
>>> print(collections.Counter(l))
Counter({2: 3, 3: 3, 4: 3, 1: 1, 5: 1})
>>> print(collections.Counter(l).most_common(2))
[(2, 3), (3, 3)]
>>> print(collections.Counter(l).most_common())
[(2, 3), (3, 3), (4, 3), (1, 1), (5, 1)]
```

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

subtract([iterable-or-mapping]) & **update([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.

```
>>> l
['a', 'a', 'b', 'c', 'c', 'c', 5]
>>> c
Counter({'c': 3, 'a': 2, 'b': 1, 5: 1})
>>> c1=collections.Counter()
>>> c1
Counter()
>>> c1.update(c)
>>> c
Counter({'c': 3, 'a': 2, 'b': 1, 5: 1})
>>> c1
Counter({'c': 3, 'a': 2, 'b': 1, 5: 1})

>>> c.clear()                                # empty the counter
```



```
>>> c.subtract(c1)
>>> c
Counter({'a': 0, 'b': 0, 'c': 0, 'd': 0})
>>> c1
Counter({'c': 3, 'a': 2, 'b': 1, 'd': 1})
... |
```

```
>>> c
Counter({'c': 3, 'a': 2, 'b': 1, 'd': 1})
>>> c1
Counter({'c': 3, 'a': 2, 'b': 1, 'd': 1})
>>> c-c1
Counter()
>>> c+c1
Counter({'c': 6, 'a': 4, 'b': 2, 'd': 2})
... |
```

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

Intersection & Union:

Sets:

```
>>> {1,2} & {2,3}
{2}
>>> {1,2} | {2,3}
{1, 2, 3}
```

Counters:

```

>>> l=['a', 'a', 'b', 'c', 'c', 'c', 5]
>>> l1=['a', 'b', 'c', 'c', 5]
>>> c=collections.Counter(l)
>>> c1=collections.Counter(l1)
>>> c
Counter({'c': 3, 'a': 2, 'b': 1, 5: 1})
>>> c1
Counter({'c': 2, 'a': 1, 'b': 1, 5: 1})
>>> c & c1
Counter({'c': 2, 'a': 1, 'b': 1, 5: 1})
>>> c | c1
Counter({'c': 3, 'a': 2, 'b': 1, 5: 1})

>>> c
Counter({'c': 3, 'a': 2, 'b': 1, 5: 1})
...

>>> c.update({'k':-10})
...

>>> c
Counter({'c': 3, 'a': 2, 'b': 1, 5: 1, 'k': -10})

>>> +c
Counter({'c': 3, 'a': 2, 'b': 1, 5: 1})
>>> -c
Counter({'k': 10})
...

>>> c
Counter({'c': 3, 'a': 2, 'b': 1, 5: 1, 'k': -10})
>>> print(*c.elements())
a a b 5 c c c

```

*Handwritten note: A red box highlights 'k': -10 in the Counter output, with an arrow pointing to the output of `print(*c.elements())` and the text "WHERE K?" written next to it.*

Counter objects have a dictionary interface except that they return a zero count for missing items instead of raising a [KeyError](#):

```

>>> l=['Rajesh', 'rajesh', 'Shanmukh', 'Shiv', 'Shadgun', 'Ram', 'nithya', 'anusha', 'rajesh']
>>> c=collections.Counter(l)
>>> c['rajesh']
2
>>> c['paleru']
0

```

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

```

>>> c
Counter({'rajesh': 2, 'anusha': 1, 'Shiv': 1, 'Rajesh': 1, 'Shadgun': 1, 'nithya': 1, 'Shanmukh': 1, 'Ram': 1})
>>> c['rajesh']
2
>>> c['rajesh']=0

```

```
>>> print(*c.items())
('anusha', 1) ('Shiv', 1) ('rajesh', 0) ('Rajesh', 1) ('Shadhgun', 1) ('nithya', 1) ('Shanmukh', 1) ('Ram', 1)
>>>
>>> del c['rajesh']
>>> print(*c.items())
('anusha', 1) ('Shiv', 1) ('Rajesh', 1) ('Shadhgun', 1) ('nithya', 1) ('Shanmukh', 1) ('Ram', 1)
```

fromkeys():

```
>>> l
[1, 2, 3, 4, 5, 2, 3, 4, 2, 3, 4]
>>> dict.fromkeys(l)
{1: None, 2: None, 3: None, 4: None, 5: None}
>>> dict.fromkeys(l,10)
{1: 10, 2: 10, 3: 10, 4: 10, 5: 10}
```

```
>>> collections.Counter(l).fromkeys(c)
Traceback (most recent call last):
  File "<pyshell#216>", line 1, in <module>
    collections.Counter(l).fromkeys(c)
  File "C:\Python34\lib\collections\__init__.py", line 513, in fromkeys
    'Counter.fromkeys() is undefined. Use Counter(iterable) instead.')
NotImplementedError: Counter.fromkeys() is undefined. Use Counter(iterable) instead.
```

```
>>> l
[1, 2, 3, 4, 5, 2, 3, 4, 2, 3, 4]
>>> collections.Counter(l)
Counter({2: 3, 3: 3, 4: 3, 1: 1, 5: 1})
>>> sorted(collections.Counter(l))
[1, 2, 3, 4, 5]
>>> set(l)
{1, 2, 3, 4, 5}
```

```
>>> d={'one':1,'two':2,'three':3}
>>> d
{'three': 3, 'two': 2, 'one': 1}
>>> collections.Counter(d)
Counter({'three': 3, 'two': 2, 'one': 1})
>>> collections.Counter(d).elements()
<itertools.chain object at 0x02FD5F10>
>>> print(*collections.Counter(d).elements())
two two three three three one
```

```
>>> d={1:'one',2:'two',3:'three'}
>>> print(*collections.Counter(d).elements())
Traceback (most recent call last):
  File "<pyshell#230>", line 1, in <module>
    print(*collections.Counter(d).elements())
TypeError: print() argument after * must be a sequence, not itertools.chain
>>> collections.Counter(d)
Counter({2: 'two', 3: 'three', 1: 'one'})
```

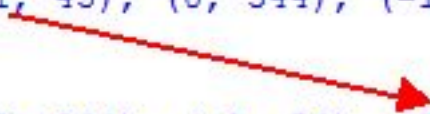
class collections.OrderedDict([items]): Return an instance of a dict subclass, supporting the usual [dict](#) methods. An *OrderedDict* is a dict that remembers the order that keys were first inserted. If a new entry overwrites an existing entry, the original insertion position is left unchanged. Deleting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

```
>>> o=collections.OrderedDict()
>>> o
OrderedDict()
>>> o[3]=23
>>> o[1]=45
>>> o[0]=344
>>> o[-1]=54
>>> o
OrderedDict([(3, 23), (1, 45), (0, 344), (-1, 54)])
```

Look by Key. Not by index:

```
>>> o[3]
23
>>> o[2]
Traceback (most recent call last):
  File "<pyshell#784>", line 1, in <module>
    o[2]
KeyError: 2
>>>
-----
>>> o.items()
ItemsView(OrderedDict([(3, 23), (1, 45), (0, 344), (-1, 54)]))
```

```
>>> o
OrderedDict([(3, 23), (1, 45), (0, 344), (-1, 54)])
>>> o.move_to_end(1)
>>> o
OrderedDict([(3, 23), (0, 344), (-1, 54), (1, 45)])
```




```
>>> d = OrderedDict.fromkeys('abcde')
>>> d.move_to_end('b')
>>> ''.join(d.keys())
'acdeb'
>>> d.move_to_end('b', last=False)
>>> ''.join(d.keys())
'bacde'
```

```
>>> o
OrderedDict([(3, 23), (0, 344), (-1, 54), (1, 45)])
>>> o.popitem()
(1, 45)
>>> o
OrderedDict([(3, 23), (0, 344), (-1, 54)])
>>> o.pop(3)
23
>>> o
OrderedDict([(0, 344), (-1, 54)])
>>> o[7]=87
>>> o
OrderedDict([(0, 344), (-1, 54), (7, 87), (6, 872)])
>>> o.popitem(last=False)
(0, 344)
>>> o
OrderedDict([(-1, 54), (7, 87), (6, 872)])
>>> o
OrderedDict([(-1, 54), (7, 87), (6, 872), (5, 23)])
>>> o.move_to_end(7,last=False)
>>> o
OrderedDict([(7, 87), (-1, 54), (6, 872), (5, 23)])
>>> o.move_to_end(7,last=True)
>>> o
OrderedDict([(-1, 54), (6, 872), (5, 23), (7, 87)])
```

```

>>> # 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)])

```

Regular Dictionaries Don't care about the order:

```

>>> o
OrderedDict([(3, 5), (1, 9), (8, 4), (0, 3)])
>>> o1
OrderedDict([(3, 5), (1, 9), (0, 3), (8, 4)])
>>> o == o1
False
>>> {1:2, 2:3} == {2:3, 1:2}
True

```

```

>>> o=collections.OrderedDict([(1, 54), (6, 872), (5, 23), (7, 87), (6,234)])
>>> o
OrderedDict([(1, 54), (6, 234), (5, 23), (7, 87)])

```

`class collections.defaultdict([default_factory[, ...]])`¹: Returns a new dictionary-like object. `defaultdict` is a subclass of the built-in `dict` class.

```

>>> d = {}
>>> for k, v in s:
...     d.setdefault(k, []).append(v)
...
>>> list(d.items())
[('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])]

```

Above one & Below one, Both are Same. But, Below one takes less time.

```

>>> s = [('yellow', 1), ('blue', 2), ('yellow', 3), ('blue', 4), ('red', 1)]
>>> d = defaultdict(list)
>>> for k, v in s:
...     d[k].append(v)
...
>>> list(d.items())
[('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])]

```

```

>>> collections.defaultdict
<class 'collections.defaultdict'>
>>> data='aazeeeehh'
>>> d={}
>>> for dat in data:
...     if dat in d:
...         d[dat]+=1
...     else:
...         d[dat]=1

```

```

>>> d
{'a': 2, 'h': 2, 'e': 4, 'z': 1}

```

```

>>> data='aazeeeehh'
>>> d={}
>>> for dat in data:
...     d[dat]=d.get(dat,0)+1

```

```

>>> d
{'a': 2, 'h': 2, 'e': 4, 'z': 1}

```

```

>>> data='aazeeeehh'
>>> d={}
>>> d=collections.defaultdict(int)
>>> for dat in data:
...     d[dat]+=1

```

```

>>> d
defaultdict(<class 'int'>, {'a': 2, 'h': 2, 'e': 4, 'z': 1})
>>>

```

```
>>> collections.Counter(data)
Counter({'e': 4, 'a': 2, 'h': 2, 'z': 1})
....|
```

```
>>> s = 'mississippi'
>>> d=collections.defaultdict(int)
>>> for k in s:
        d[k]+=1
```

```
>>> d
defaultdict(<class 'int'>, {'i': 4, 'p': 2, 'm': 1, 's': 4})
>>> collections.Counter(s)
Counter({'i': 4, 's': 4, 'p': 2, 'm': 1})
```

```
>>> test_results={}
>>> data=''Alice:76
Bob:87
Charlie:80
Alice:98
Bob:98
Charlie:87''
>>> for line in data.splitlines():
        name,score=line.split(':')
        score=int(score)
        if name in test_results:
            test_results[name].append(score)
        else:
            test_results[name]=[score]
```

```
>>> test_results
{'Bob': [87, 98], 'Alice': [76, 98], 'Charlie': [80, 87]}
```



```
>>> test_results=collections.defaultdict(list)
>>> data='''Alice:76
Bob:87
Charlie:80
Alice:98
Bob:98
Charlie:87'''
>>> for line in data.splitlines():
    name,score=line.split(':')
    score=int(score)
    test_results[name].append(score)

>>> test_results
defaultdict(<class 'list'>, {'Bob': [87, 98], 'Alice': [76, 98], 'Charlie': [80, 87]})
```

```
>>> collections.defaultdict(lambda: None)
defaultdict(<function <lambda> at 0x0300C6A8>, {})

>>> d={}
>>> d.setdefault(3, []).append(4)
>>> d
{3: [4]}
```

```
>>> def constant_factory(value):
    return lambda:value

>>> d = collections.defaultdict(constant_factory('<missing>'))
>>> d.update(name='John', action='ran')
>>> d
defaultdict(<function constant_factory.<locals>.<lambda> at 0x0300CCD8>, {'action': 'ran', 'name': 'John'})
>>> '%(name)s %(action)s to %(object)s' % d
'John ran to <missing>'

>>> s = [('red', 1), ('blue', 2), ('red', 3), ('blue', 4), ('red', 1), ('blue', 4)]
>>> d=collections.defaultdict(list)
>>> for k,v in s:
    d[k].append(v)

>>> d
defaultdict(<class 'list'>, {'blue': [2, 4, 4], 'red': [1, 3, 1]})
```

...

UserDict objects

...

```
>>> d=collections.UserDict()  
>>> d  
{}  
>>> d[0]=34  
>>> d  
{0: 34}  
>>> d.data  
{0: 34}  
>>> d[4]=324  
>>> d  
{0: 34, 4: 324}  
>>> d.data  
{0: 34, 4: 324}
```

UserList objects

```

>>> d=collections.UserList()
>>> d
[]
>>> d.data
[]
>>> d.data is d
False
>>> type(d)
<class 'collections.UserList'>
>>> type(d.data)
<class 'list'>

```

UserString objects

```

>>> d=collections.UserString
>>> d
<class 'collections.UserString'>
>>> type(d)
<class 'abc.ABCMeta'>
...

```

ITERTOOLS Library:

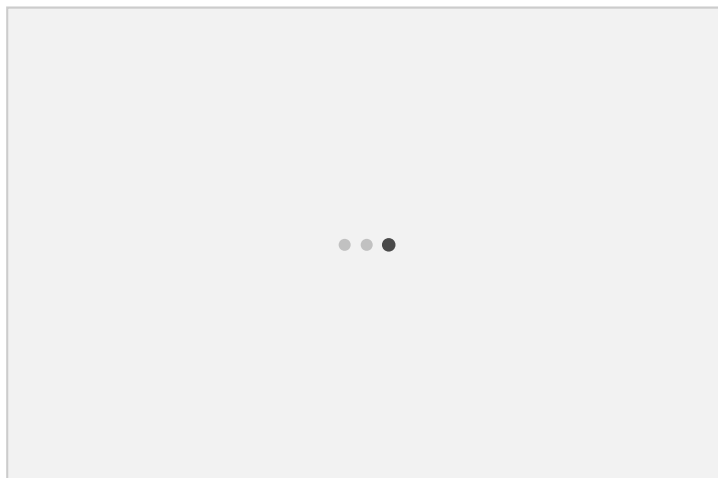
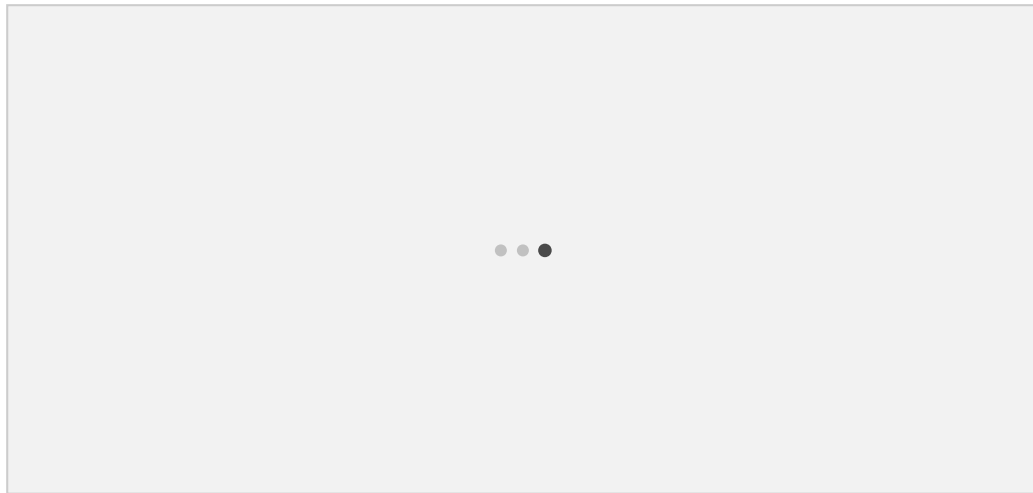
```

>>> import itertools
>>> dir(itertools)
['__doc__', '__loader__', '__name__', '__package__', '__spec__', '_grouper', '_tee',
'_tee_dataobject', 'accumulate', 'chain', 'combinations', 'combinations_with_replacement',
'compress', 'count', 'cycle', 'dropwhile', 'filterfalse', 'groupby', 'islice', 'permutations',
'product', 'repeat', 'starmap', 'takewhile', 'tee', 'zip_longest']

```

Infinite iterators:			
Iterator	Arguments	Results	Example
count()	start, [step]	start, start+step, start+2*step, ...	count(10) --> 10 11 12 13 14 ...
cycle()	p	p0, p1, ... plast, p0, p1, ...	cycle('ABCD') --> A B C D A B C D ...
repeat()	elem [,n]	elem, elem, elem, ... endlessly or up to n times	repeat(10, 3) --> 10 10 10
Iterators terminating on the shortest input sequence:			
Iterator	Arguments	Results	Example
accumulate()	p [,func]	p0, p0+p1, p0+p1+p2, ...	accumulate([1,2,3,4,5]) --> 1 3 6 10 15
chain()	p, q, ...	p0, p1, ... plast, q0, q1, ...	chain('ABC', 'DEF') --> A B C D E F
chain.from_iterable()	iterable	p0, p1, ... plast, q0, q1, ...	chain.from_iterable(['ABC', 'DEF']) --> A B C D E F
compress()	data, selectors	(d[0] if s[0]), (d[1] if s[1]), ...	compress('ABCDEF', [1,0,1,0,1,1]) --> A C E F
dropwhile()	pred, seq	seq[n], seq[n+1], starting when pred fails	dropwhile(lambda x: x<5, [1,4,6,4,1]) --> 6 4 1
filterfalse()	pred, seq	elements of seq where pred(elem) is false	filterfalse(lambda x: x%2, range(10)) --> 0 2 4 6 8
groupby()	iterable[, keyfunc]	sub-iterators grouped by value of keyfunc(v)	
islice()	seq, [start,] stop [, step]	elements from seq[start:stop:step]	islice('ABCDEFG', 2, None) --> C D E F G
starmap()	func, seq	func(*seq[0]), func(*seq[1]), ...	starmap(pow, [(2,5), (3,2), (10,3)]) --> 32 9 1000
takewhile()	pred, seq	seq[0], seq[1], until pred fails	takewhile(lambda x: x<5, [1,4,6,4,1]) --> 1 4
tee()	it, n	it1, it2, ... itn splits one iterator into n	
zip_longest()	p, q, ...	(p[0], q[0]), (p[1], q[1]), ...	zip_longest('ABCD', 'xy', fillvalue='-') --> Ax By C- D-
Combinatoric generators:			
Iterator	Arguments	Results	
product()	p, q, ... [repeat=1]	cartesian product, equivalent to a nested for-loop	
permutations()	p[, r]	r-length tuples, all possible orderings, no repeated elements	
combinations()	p, r	r-length tuples, in sorted order, no repeated elements	
combinations with replacement()	p, r	r-length tuples, in sorted order, with repeated elements	
product('ABCD', repeat=2)		AA AB AC AD BA BB BC BD CA CB CC CD DA DB DC DD	
permutations('ABCD', 2)		AB AC AD BA BC BD CA CB CD DA DB DC	
combinations('ABCD', 2)		AB AC AD BC BD CD	
combinations with replacement('ABCD', 2)		AA AB AC AD BB BC BD CC CD DD	

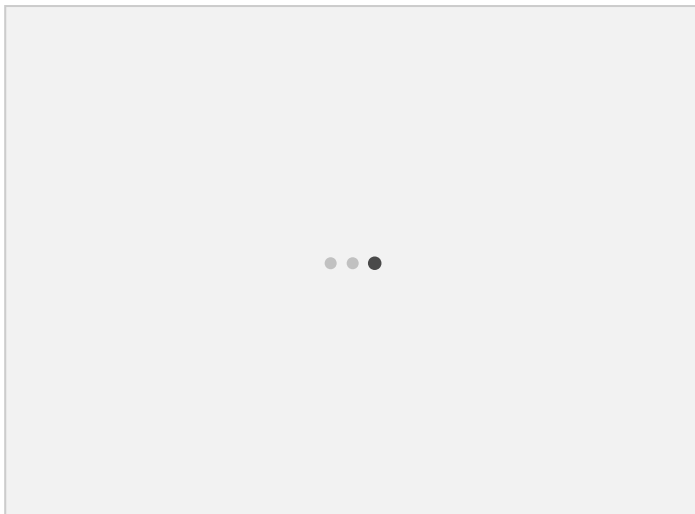
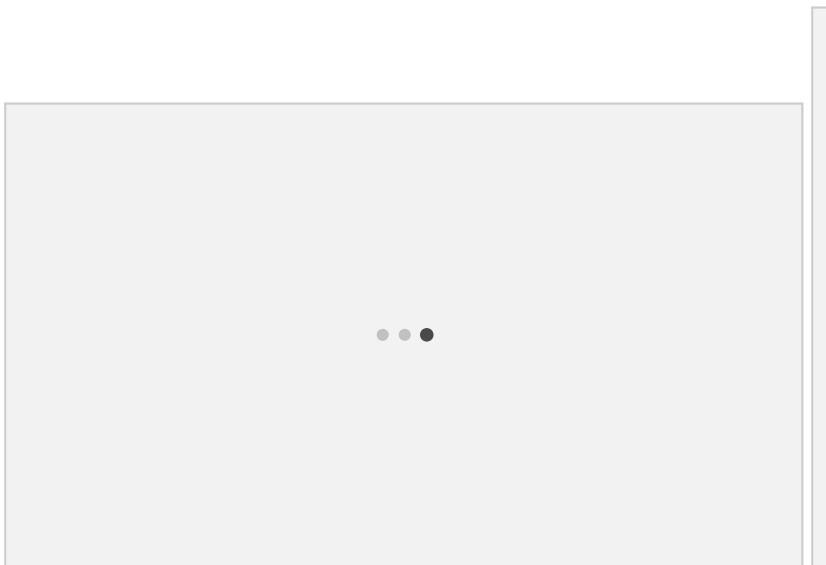
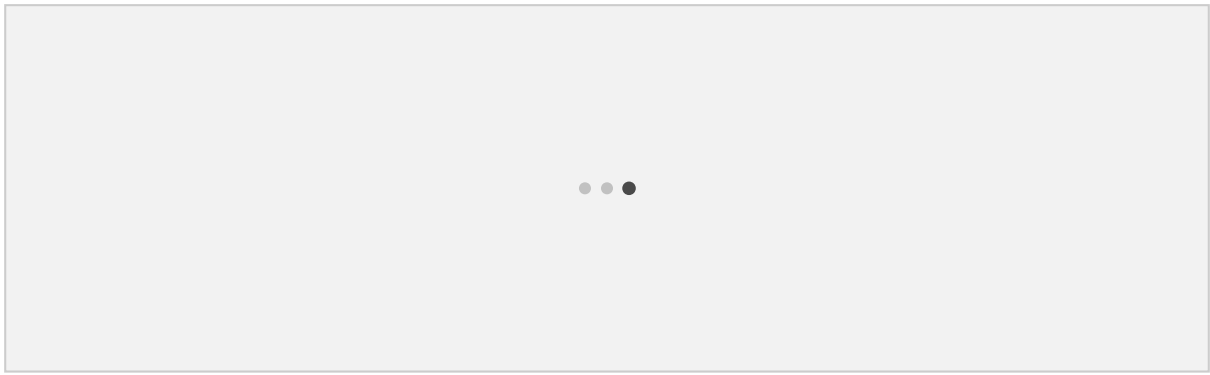
itertools.count(start=0, step=1):



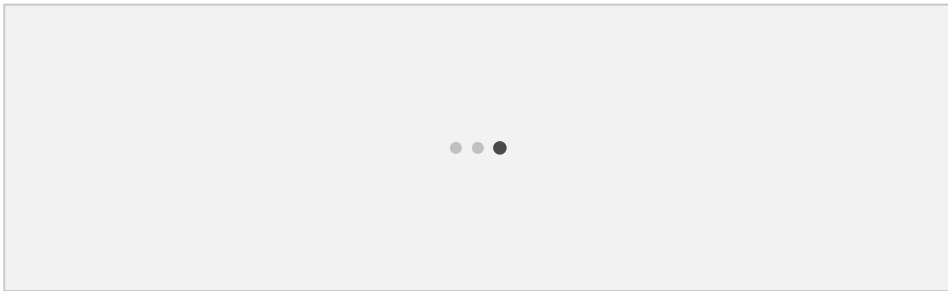
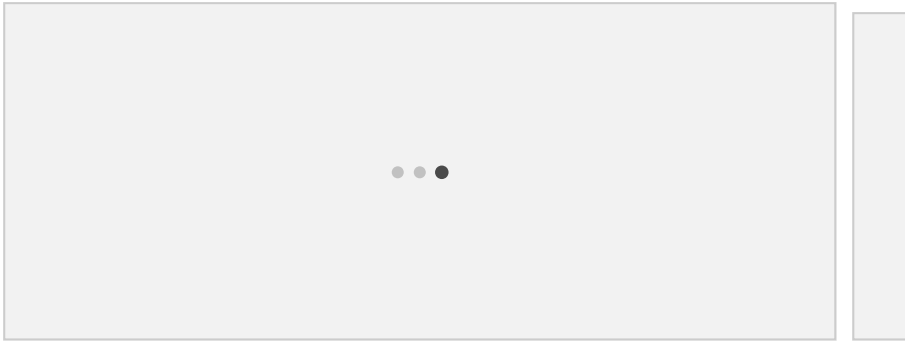
itertools.cycle(iterable)[1](#): Make an iterator returning elements from the iterable and saving a copy of each. When the iterable is exhausted, return elements from the saved copy. Repeats indefinitely. Equivalent to:

Original code of cycle() is:

```
def cycle(iterable):  
    # cycle('ABCD') --> A B C D A B C D A B C D ...  
    saved = []  
    for element in iterable:  
        yield element  
        saved.append(element)  
    while saved:  
        for element in saved:  
            yield element
```

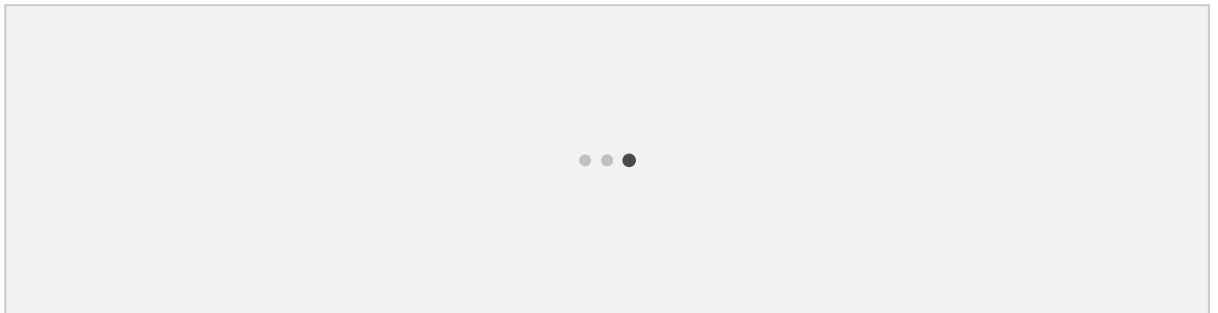


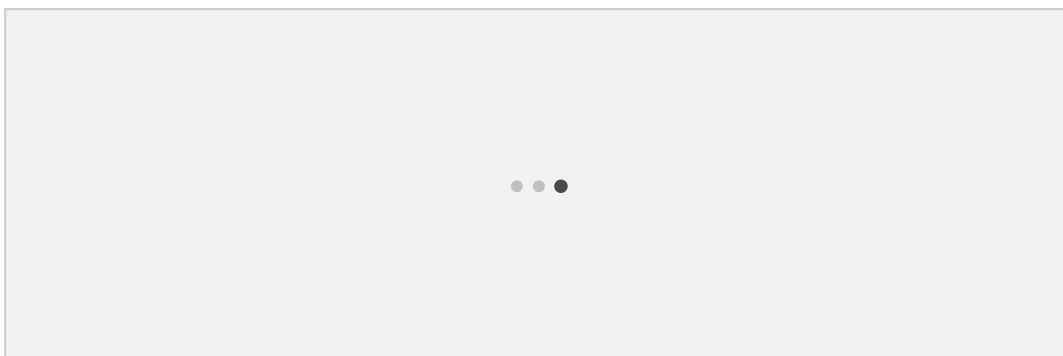
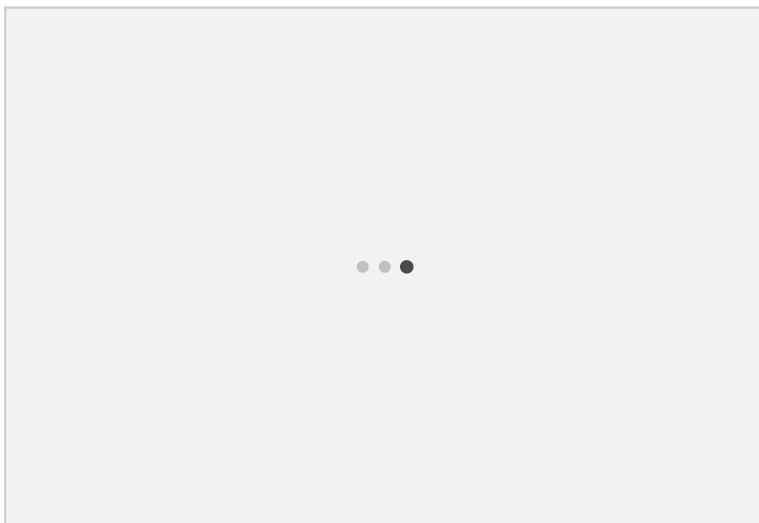
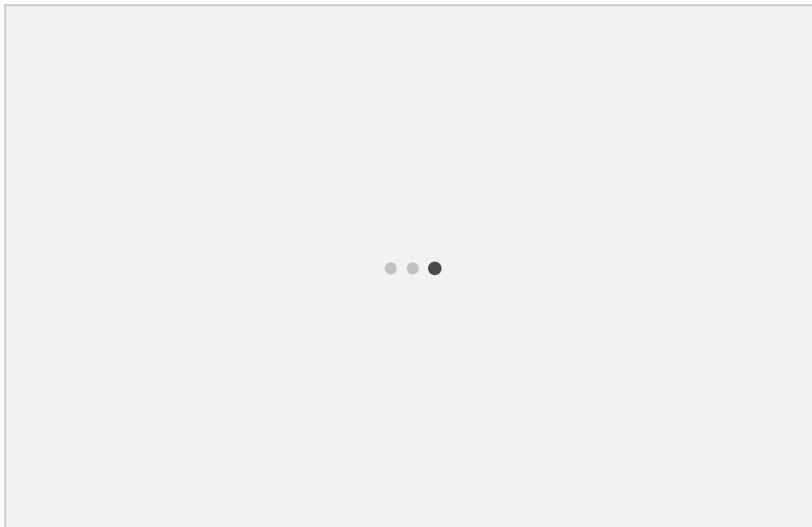
`itertools.repeat(object[, times])`¹: Make an iterator that returns *object* over and over again. Runs indefinitely unless the *times* argument is specified. Used as argument to [map\(\)](#) for invariant parameters to the called function. Also used with [zip\(\)](#) to create an invariant part of a tuple record. Equivalent to:



`itertools.accumulate(iterable[, func])`[¶](#): Make an iterator that returns accumulated sums. Elements may be any addable type including [Decimal](#) or [Fraction](#). If the optional *func* argument is supplied, it should be a function of two arguments and it will be used instead of addition.

Original code:





itertools.chain(*iterables)[1](#): Make an iterator that returns elements from the first iterable until it is exhausted, then proceeds to the next iterable, until all of the iterables are exhausted. Used for treating consecutive sequences as a single sequence. Equivalent to:

...

...

classmethod `chain.from_iterable(iterable)`¹: Alternate constructor for `chain()`. Gets chained inputs from a single iterable argument that is evaluated lazily. Roughly equivalent to:

...

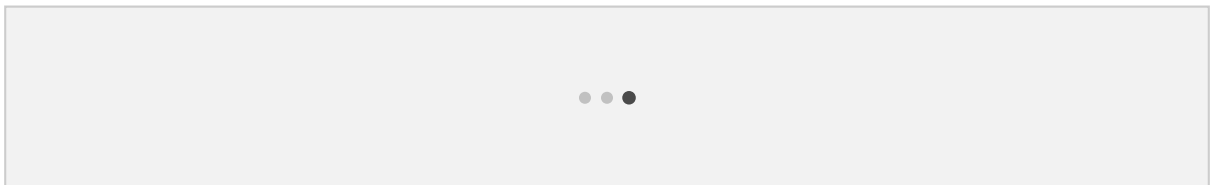
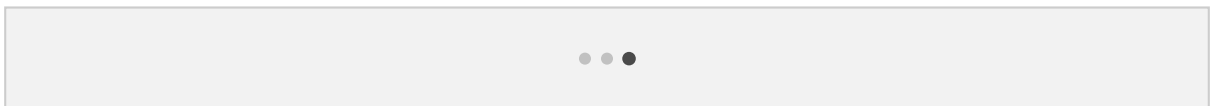
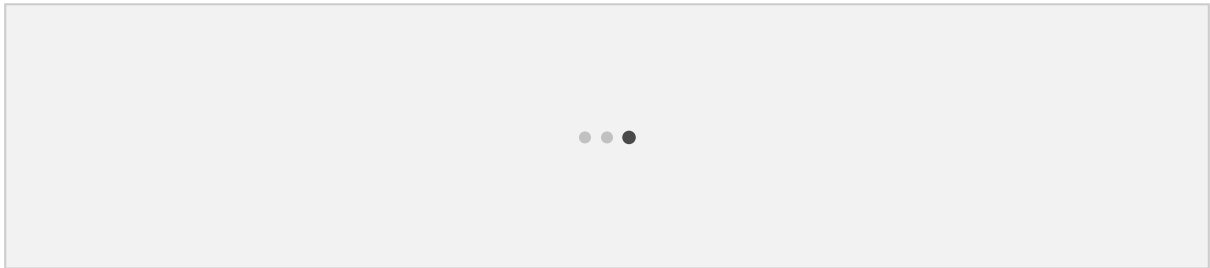
...

itertools.compress(*data*, *selectors*)¹: Make an iterator that filters elements from *data* returning only those that have a corresponding element in *selectors* that evaluates to `True`. Stops when either the *data* or *selectors* iterables has been exhausted. Equivalent to:

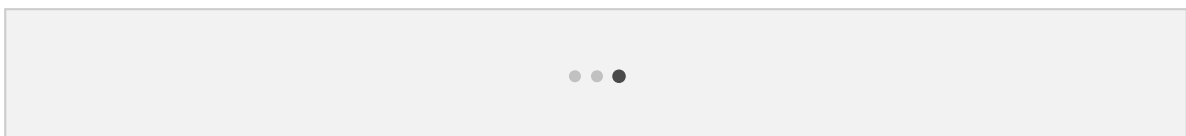
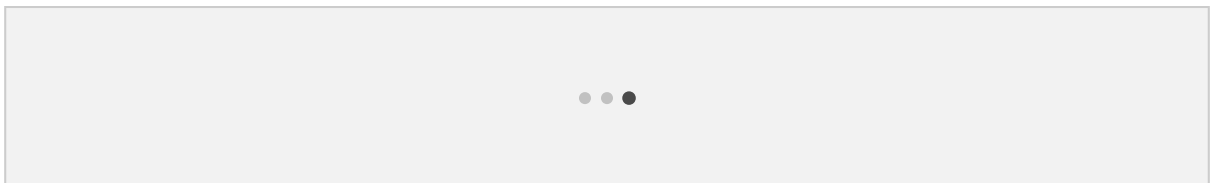
...

...

`itertools.dropwhile(predicate, iterable)`¹: Make an iterator that drops elements from the iterable as long as the predicate is true; afterwards, returns every element. Note, the iterator does not produce *any* output until the predicate first becomes false, so it may have a lengthy start-up time. Equivalent to:



`itertools.filterfalse(predicate, iterable)`¹: Make an iterator that filters elements from iterable returning only those for which the predicate is `False`. If `predicate` is `None`, return the items that are false. Equivalent to:



`itertools.groupby(iterable, key=None)`¹: Make an iterator that returns consecutive keys and groups from the *iterable*. The *key* is a function computing a key value for each element. If not specified or is `None`, *key* defaults to an identity function and returns the element unchanged. Generally, the iterable needs to already be sorted on the same key function. The operation of `groupby()` is similar to the `uniq` filter in Unix.

...

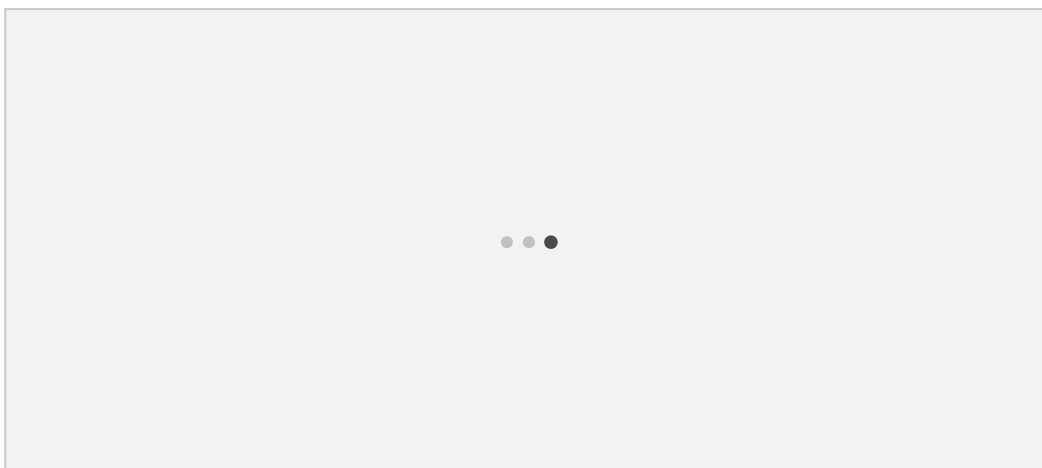
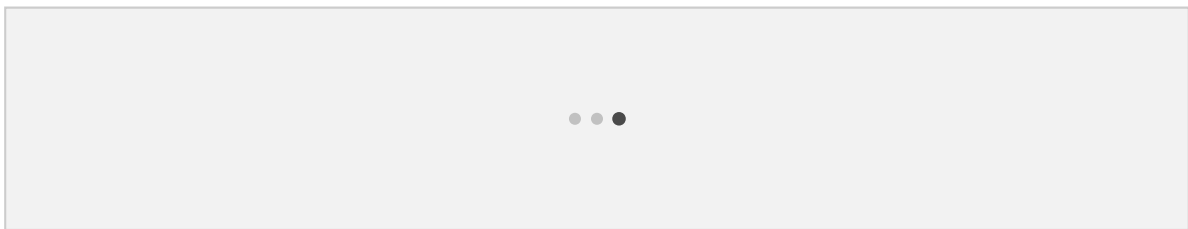
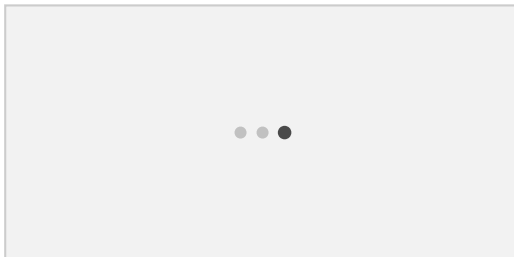
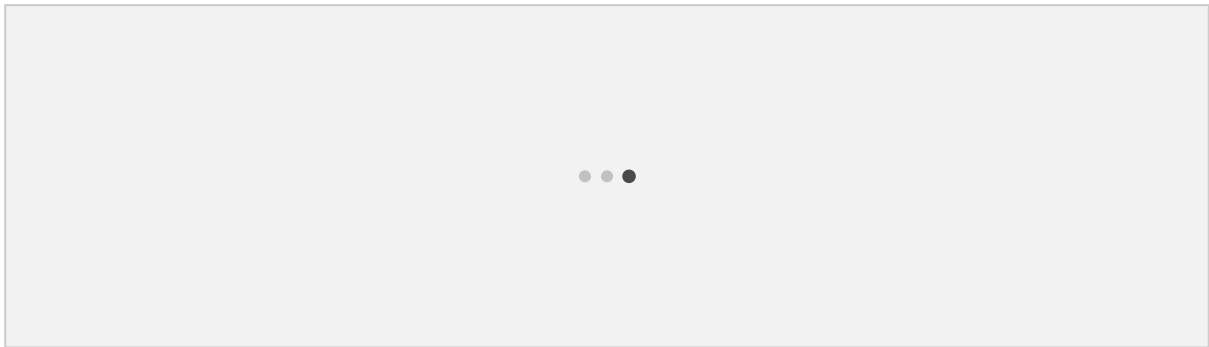
...

...

...

`itertools.islice(iterable, stop)`

`itertools.islice(iterable, start, stop[, step]):`



`itertools.starmap(function, iterable)`¹: Make an iterator that computes the function using arguments obtained from the iterable. Used instead of [map\(\)](#) when argument parameters are already grouped in tuples from a single iterable (the data has been “pre-zipped”). The difference between [map\(\)](#) and [starmap\(\)](#) parallels the distinction between `function(a,b)` and `function(*c)`. Equivalent to:

...

Its behavior is similar to below code:

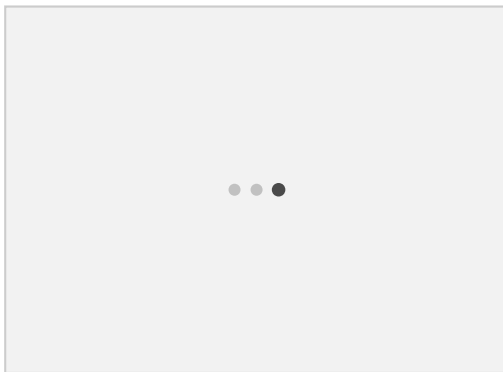
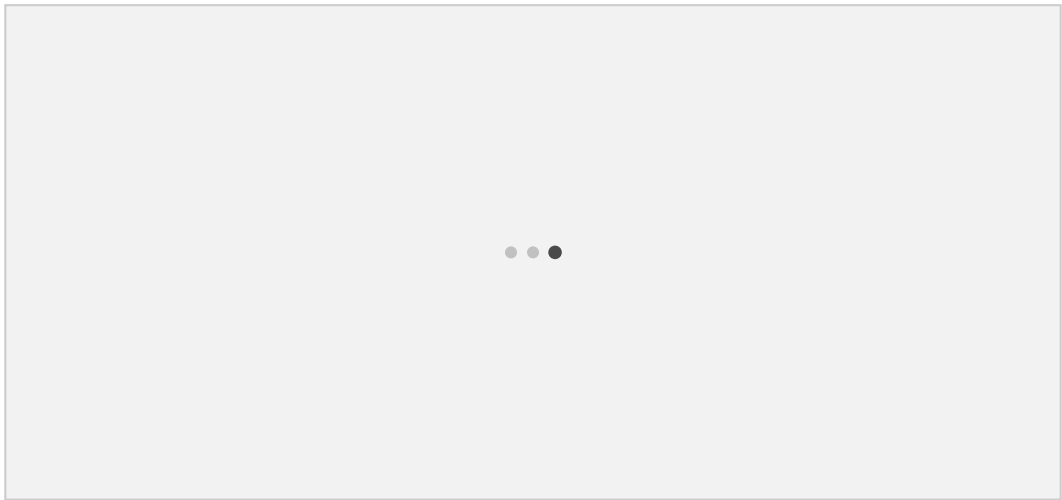
...

`itertools.takewhile(predicate, iterable)`[1](#): Make an iterator that returns elements from the iterable as long as the predicate is true. Equivalent to:

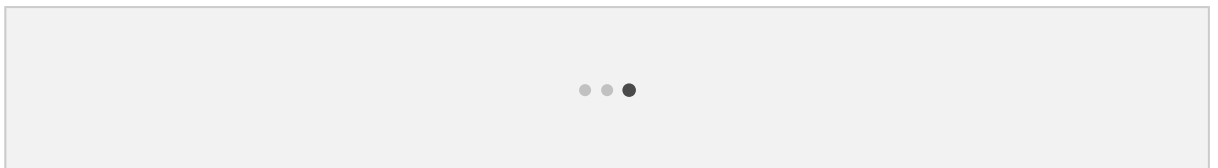
...

...

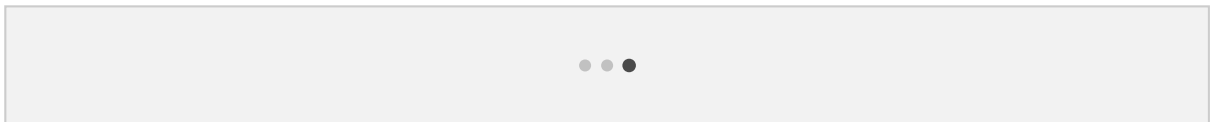
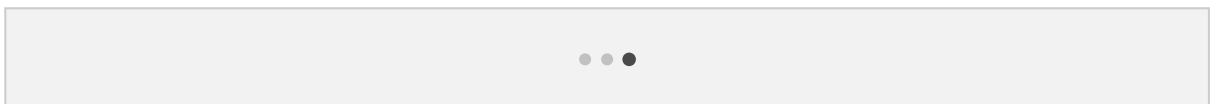
`itertools.tee(iterable, n=2)`[1](#): Return *n* independent iterators from a single iterable. Equivalent to:



`itertools.zip_longest(*iterables, fillvalue=None)`[1](#): Make an iterator that aggregates elements from each of the iterables. If the iterables are of uneven length, missing values are filled-in with *fillvalue*. Iteration continues until the longest iterable is exhausted.



`itertools.product(*iterables, repeat=1)`[1](#): Cartesian product of input iterables.



...

itertools.permutations(*iterable*, *r=None*)[1](#): Return successive *r* length permutations of elements in the *iterable*. If *r* is not specified or is `None`, then *r* defaults to the length of the *iterable* and all possible full-length permutations are generated.

...

...

...

...

itertools.combinations(*iterable*, *r*)[1](#): Return *r* length subsequences of elements from the input *iterable*. Elements are treated as unique based on their position, not on their value. So if the input elements are unique, there will be no repeat values in each combination.

...

...

`itertools.combinations_with_replacement(iterable, r)`[1](#):

Return r length subsequences of elements from the input *iterable* allowing individual elements to be repeated more than once.

...

...



The `commands` module has been removed in Python 3. Use the `subprocess` module instead.

Python 2: `raw_input()` takes exactly what the user typed and passes it back as a string. `input()` takes the `raw_input()` and performs an `eval()` on it as well. The main difference is that `input()` expects a syntactically correct python statement where `raw_input()` does not. So `input()=eval(raw_input())`.

Python 3: `raw_input()` was renamed to `input()` and the old `input()` was removed. If you want to use the old `input()`, you can do `eval(input())`.

Sets:

Create it by `{}` or `set()`. Ex: `{1,2,3,2,4,5,2,3,3}` or `set(1,2,3,4,2,3,4,2,3,2)`

Create Empty set as: `set()`, not `{}`

```
>>> string1, string2, string3 = '', 'Trondheim', 'Hammer Dance'
>>> non_null = string1 or string2 or string3
>>> non_null
'Trondheim'
```

Getting DOC of a function:

```
>>> a=dict([(1,2)])
```

```
>>> a
```

```
{1: 2}
```

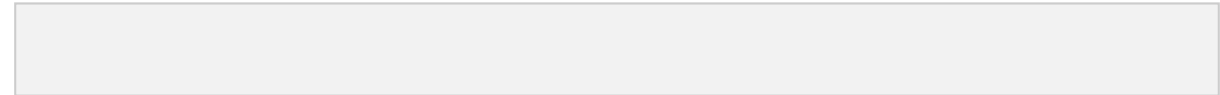
```
>>> a.__doc__
```

"dict() -> new empty dictionary\nndict(mapping) -> new dictionary initialized from a mapping object's\n (key, value) pairs\nndict(iterable) -> new dictionary initialized as if via:\n d = {}\n for k, v in iterable:\n d[k] = v\nndict(**kwargs) -> new dictionary initialized with the name=value pairs\n in the keyword argument list. For example: dict(one=1, two=2)"

External Libraries:

numpy:

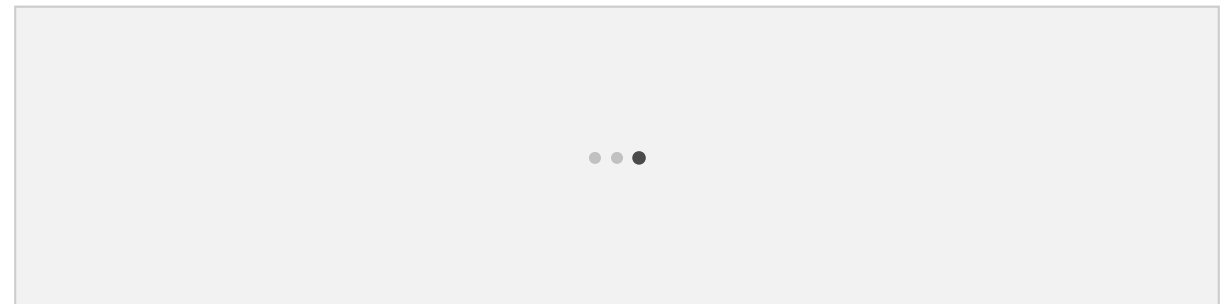
Process to install numpy package:



it is installed @:

```
>>> numpy.__file__
```

```
'C:\\Python34\\lib\\site-packages\\numpy\\__init__.py'
```



```
>>> len(dir(numpy))  
598
```

Arrays

The central feature of NumPy is the *array* object class. Arrays are similar to lists in Python, except that every element of an array must be of the same type, typically a numeric type like float or int. Arrays make operations with large amounts of numeric data very fast and are generally much more efficient than lists.

An array can be created from a list:

```
>>> a = np.array([1, 4, 5, 8], float)
```

```
>>> a
```

```
array([ 1.,  4.,  5.,  8.])
```

```
>>> type(a)
```

```
<type 'numpy.ndarray'>
```

Here, the function `array` takes two arguments: the list to be converted into the array and the type of each member of the list. Array elements are accessed, sliced, and manipulated just like lists:

```
>>> a[:2]
array([ 1., 4.])
>>> a[3]
8.0
>>> a[0] = 5.
>>> a
array([ 5., 4., 5., 8.])
```

Arrays can be multidimensional. Unlike lists, different axes are accessed using commas inside bracket notation. Here is an example with a two-dimensional array (e.g., a matrix):

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]], float)
>>> a
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]])
>>> a[0,0]
1.0
>>> a[0,1]
2.0
```

...

...

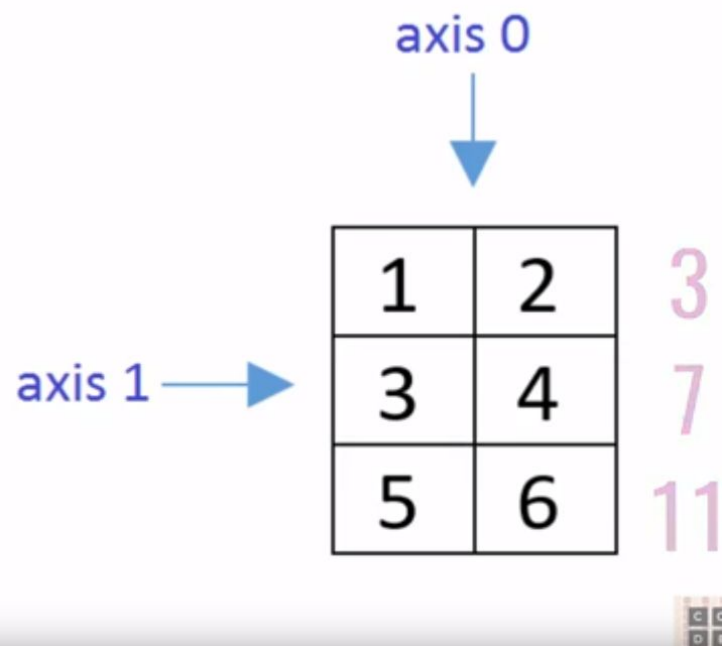
...

```
>>> np.arange(1,5)
array([1, 2, 3, 4])
>>> np.arange(1,5,2)
array([1, 3])
>>> np.linspace(1,5,10)
array([ 1.          ,  1.44444444,  1.88888889,  2.33333333,  2.77777778,
        3.22222222,  3.66666667,  4.11111111,  4.55555556,  5.          ])
>>> np.linspace(1,5,5)
array([ 1.,  2.,  3.,  4.,  5.])
```

...

...

```
>>>  
>>>  
>>> a  
array([[1, 2],  
       [3, 4],  
       [5, 6]])  
>>> a.min()  
1  
>>> a.max()  
6  
>>> a.sum()  
21  
>>> a.sum(axis=0)  
array([ 9, 12])  
>>> a.sum(axis=1)  
array([ 3,  7, 11])
```



```
>>> np.sqrt(a)
array([[ 1.          ,  1.41421356],
       [ 1.73205081,  2.          ],
       [ 2.23606798,  2.44948974]])
>>> a
array([[1, 2],
       [3, 4],
       [5, 6]])
```

...

```
>>> a=np.array([[6,7,8], [1,2,3],[9,3,2]])
>>> a
array([[6, 7, 8],
       [1, 2, 3],
       [9, 3, 2]])
>>> a[1,2]
3
>>> a[0:2,2]
array([8, 3])
>>>
```

$a[0:2, 2] = [8, 3]$

0	6	7	8
1	1	2	3
2	9	3	2

```
>>> a=np.array([[6,7,8], [1,2,3],[9,3,2]])
>>> a
array([[6, 7, 8],
       [1, 2, 3],
       [9, 3, 2]])
>>> for row in a:
        print(row)
```

```
[6 7 8]
[1 2 3]
[9 3 2]
```

```
>>> a = np.arange(12).reshape(3,4)
>>> a
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> b = a > 4
>>> b
array([[False, False, False, False],
       [False,  True,  True,  True],
       [ True,  True,  True,  True]], dtype=bool)
>>> a[b]
array([ 5,  6,  7,  8,  9, 10, 11])
>>> a[b]=-1
>>> a
array([[ 0,  1,  2,  3],
       [ 4, -1, -1, -1],
       [-1, -1, -1, -1]])
>>> |
```

```
>>> a=np.arange(6).reshape(3,2)
>>> b=np.arange(6,12).reshape(3,2)
>>> a
array([[0, 1],
       [2, 3],
       [4, 5]])
>>> b
array([[ 6,  7],
       [ 8,  9],
       [10, 11]])
>>> np.vstack(a,b)
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    np.vstack(a,b)
TypeError: vstack() takes 1 positional argument but 2 were given
>>> np.vstack((a,b))
array([[ 0,  1],
       [ 2,  3],
       [ 4,  5],
       [ 6,  7],
       [ 8,  9],
       [10, 11]])
>>> np.hstack((a,b))
array([[ 0,  1,  6,  7],
       [ 2,  3,  8,  9],
       [ 4,  5, 10, 11]])
```

...

```
In [26]: import numpy as np
         n=np.array([[7,8,9],[4,5,6],[1,2,3]])
         print('with flatten:')
         n.flatten()
```

with flatten:

```
Out[26]: array([7, 8, 9, 4, 5, 6, 1, 2, 3])
```

```
In [27]: import numpy as np
         n=np.array([[7,8,9],[4,5,6],[1,2,3]])
         print('with ravel:')
         np.ravel(n)
```

with ravel:

```
Out[27]: array([7, 8, 9, 4, 5, 6, 1, 2, 3])
```

with nditer:

Traceback (most recent call last):

File "<stdin>", line 4, in <module>

```
for x in np.nditer(n,order='R'):
```

ValueError: order must be one of 'C', 'F', 'A', or 'K'

```
In [38]: import numpy as np
         n=np.array([[7,8,9],[4,5,6],[1,2,3]])
         print('with nditer:')
         for x in np.nditer(n,order='C'):
```

```
             print(x)
```

with nditer:

7

8

9

4

5

6

1

2

3

```
In [39]: import numpy as np
         n=np.array([[7,8,9],[4,5,6],[1,2,3]])
         print('with nditer:')
         for x in np.nditer(n,order='F'):
```



```
        print(x)
with nditer:
7
4
1
8
5
2
9
6
3
```

```
In [40]: import numpy as np
        n=np.array([[7,8,9],[4,5,6],[1,2,3]])
        print('with nditer:')
        for x in np.nditer(n,order='A'):
            print(x)
with nditer:
7
8
9
4
5
6
1
2
3
```

```
In [41]: import numpy as np
        n=np.array([[7,8,9],[4,5,6],[1,2,3]])
        print('with nditer:')
        for x in np.nditer(n,order='K'):
            print(x)
with nditer:
7
8
9
4
5
6
```

1
2
3

```
In [2]: a = np.arange(12).reshape(3,4)
a
```

```
Out[2]: array([[ 0,  1,  2,  3],
               [ 4,  5,  6,  7],
               [ 8,  9, 10, 11]])
```

```
In [8]: for x in np.nditer(a,order='F',flags=['external_loop']):
        print(x)
```

```
[0 4 8]
[1 5 9]
[ 2  6 10]
[ 3  7 11]
```

```
In [10]: for x in np.nditer(a,op_flags=['readwrite']):
        x[...] = x*x
```

```
In [11]: a
```

```
Out[11]: array([[ 0,  1,  4,  9],
               [16, 25, 36, 49],
               [64, 81, 100, 121]])
```

```
In [11]: a
```

```
Out[11]: array([[ 0,  1,  4,  9],
               [16, 25, 36, 49],
               [64, 81, 100, 121]])
```

```
In [13]: b = np.arange(3,15,4).reshape(3,1)
b
```

```
Out[13]: array([[ 3],
               [ 7],
               [11]])
```

```
In [14]: for x,y in np.nditer([a,b]):
        print(x,y)
```

```
0 3
1 3
4 3
9 3
16 7
25 7
36 7
49 7
64 11
81 11
```

...

xlrd:

...

```
>>> import xlrd
>>> book=xlrd.open_workbook('1.xlsx')
>>> book.nsheets
3
```



```
>>> book.sheet_names()
['Sheet1', 'Sheet2', 'Sheet3']
```

```
>>> book=xlrd.open_workbook('1.xlsx')
```



```
>>> book.sheet_names()
['Rajesh', 'Sheet2', 'Sheet3']
```

```
>>> book.sheet_by_index(0)
```

<xlrd.sheet.Sheet object at 0x0000000002B8A9E8>

```
>>> book=xlrd.open_workbook('1.xlsx')
```

values1

```
>>> book.sheet_by_index(0).row_values(0)
['values1']
```

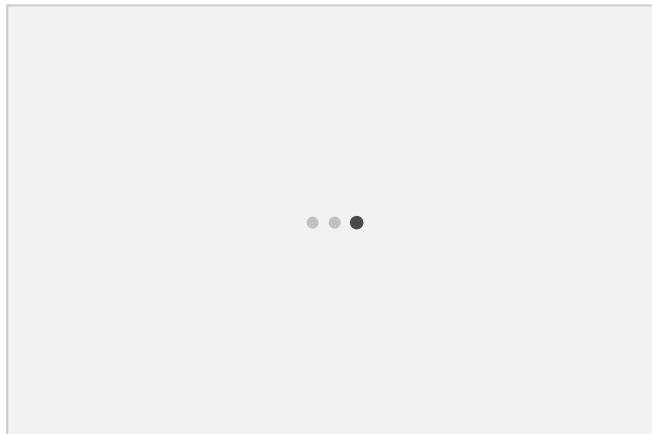
A	B	C	D	E	F
row1value1	row1value2	row1value3	row1value4	row1value5	row1value6
row2value1	row2value2	row2value3	row2value4	row2value5	row2value6
row3value1	row3value2	row3value3	row3value4	row3value5	row3value6
row4value1	row4value2	row4value3	row4value4	row4value5	row4value6

```
>>> book=xlrd.open_workbook('1.xlsx')
>>> book.sheet_by_index(0).row_values(0)
['row1value1', 'row1value2', 'row1value3', 'row1value4', 'row1value5', 'row1value6']
>>> book.sheet_by_index(0).row_values(1)
['row2value1', 'row2value2', 'row2value3', 'row2value4', 'row2value5', 'row2value6']
>>> book.sheet_by_index(0).cell(1,3)
text:'row2value4'
>>> book.sheet_by_index(0).cell(1,3).value
'row2value4'
>>> book.sheet_by_index(0).row_slice(rowx=1,start_colx=2,end_colx=4)
[text:'row2value3', text:'row2value4']
```

re Library:

```
>>> import re
>>> dir(re)
```

['A', 'ASCII', 'DEBUG', 'DOTALL', 'I', 'IGNORECASE', 'L', 'LOCALE', 'M', 'MULTILINE', 'S', 'Scanner', 'T', 'TEMPLATE', 'U', 'UNICODE', 'VERBOSE', 'X', '_MAXCACHE', '__all__', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__', '__version__', '_alphanum_bytes', '_alphanum_str', '_cache', '_cache_repl', '_compile', '_compile_repl', '_expand', '_pattern_type', '_pickle', '_subx', 'compile', 'copyreg', 'error', 'escape', 'findall', 'finditer', 'fullmatch', 'match', 'purge', 'search', 'split', 'sre_compile', 'sre_parse', 'sub', 'subn', 'sys', 'template']

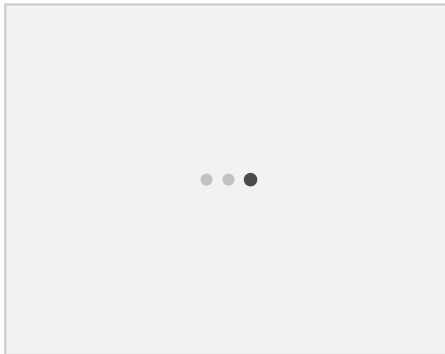


'.'	In the default mode, this matches any character except a newline. If the DOTALL flag has been specified, this matches any character including a newline.
-----	----------------------------------------------------------------------------------------------------------------------------------------------------------

```
>>> re.search(r'.+',text).group()
'Rajesh'
>>> re.search(r'.+',text,re.DOTALL).group()
'Rajesh\npaleru\nanusha\npaleru\nshiv\npaleru\nnanjaneyulu\nvijayalshmi\nkalyani\nram\nshad\nnithya'
>>> re.search(r'.+',text,re.MULTILINE).group()
'Rajesh'
```

'^'	Matches the start of the string, and in MULTILINE mode also matches immediately after each newline.
-----	-----------------------------------------------------------------------------------------------------

```
>>> re.search(r'^p',text).group()
Traceback (most recent call last):
  File "<pyshell#30>", line 1, in <module>
    re.search(r'^p',text).group()
AttributeError: 'NoneType' object has no attribute 'group'
>>> re.search(r'^p',text,re.MULTILINE).group()
'p'
.
```



Identifiers:

- \d = any number
- \D = anything but a number
- \s = space
- \S = anything but a space
- \w = any letter
- \W = anything but a letter
- . = any character, except for a new line
- \b = space around whole words
- \. = period. must use backslash, because . normally means any character.

Modifiers:

- {1,3} = for digits, u expect 1-3 counts of digits, or "places"
- + = match 1 or more
- ? = match 0 or 1 repetitions.
- * = match 0 or MORE repetitions
- \$ = matches at the end of string
- ^ = matches start of a string
- | = matches either/or. Example x|y = will match either x or y
- [] = range, or "variance"
- {x} = expect to see this amount of the preceding code.
- {x,y} = expect to see this x-y amounts of the precedng code

White Space Charts:

- \n = new line
- \s = space

- \t = tab
- \e = escape
- \f = form feed
- \r = carriage return

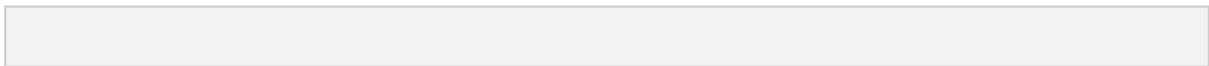
Characters to REMEMBER TO ESCAPE IF USED!

- . + * ? [] \$ ^ () { } | \

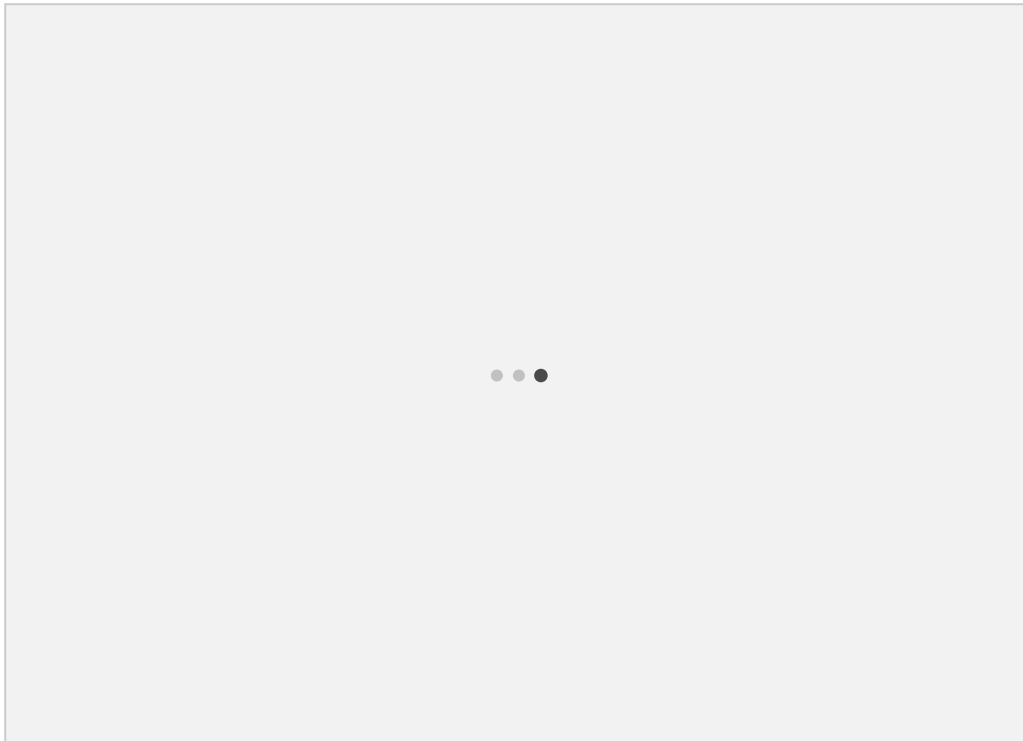
Brackets:

- [] = quant[ia]tative = will find either quantitative, or quantatative.
- [a-z] = return any lowercase letter a-z
- [1-5a-qA-Z] = return all numbers 1-5, lowercase letters a-q and uppercase A-Z

Getting paragraphs from a html source code:



'\$'	Matches the end of the string or just before the newline at the end of the string, and in MULTILINE mode also matches before a newline. foo matches both 'foo' and 'foobar', while the regular expression foo\$ matches only 'foo'. More interestingly, searching for foo.\$ in 'foo1\nfoo2\n' matches 'foo2' normally, but 'foo1' in MULTILINE mode; searching for a single \$ in 'foo\n' will find two (empty) matches: one just before the newline, and one at the end of the string.
------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------



'*'	Causes the resulting RE to match 0 or more repetitions of the preceding RE, as many repetitions as are possible. <code>ab*</code> will match 'a', 'ab', or 'a' followed by any number of 'b's'.
-----	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

```
>>> re.search(r'R*', 'RRajesh').group()
'RR'
```

'+'	Causes the resulting RE to match 1 or more repetitions of the preceding RE. <code>ab+</code> will match 'a' followed by any non-zero number of 'b's'; it will not match just 'a'.
-----	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

```
>>> re.search(r'R+', 'RRajesh').group()
'RR'
```

'?'	Causes the resulting RE to match 0 or 1 repetitions of the preceding RE. <code>ab?</code> will match either 'a' or 'ab'.
-----	--------------------------------------------------------------------------------------------------------------------------


```
>>> re.search(r'R?', 'RRajesh').group()
'R'
>>> re.search(r'R', 'RRajesh').group()
'R'
>>> re.search(r'(R+)?', 'RRajesh').group()
'RR'
>>> re.search(r'R+?', 'RRajesh').group()
'R'
```

?, +?, ??	The '', '+', and '?' qualifiers are all <i>greedy</i> ; they match as much text as possible. Sometimes this behaviour isn't desired; if the RE <code><.*></code> is matched against <code><H1>title</H1></code> , it will match the entire string, and not just <code><H1></code> . Adding '?' after the qualifier makes it perform the match in <i>non-greedy</i> or <i>minimal</i> fashion; as few characters as possible will be matched. Using <code>.*?</code> in the previous expression will match only <code><H1></code> .
------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

```
>>> re.search(r'<.*>', '<H1>title</H1>').group()
'<H1>title</H1>'
>>> re.search(r'<.+>', '<H1>title</H1>').group()
'<H1>title</H1>'
>>> re.search(r'<.*?>', '<H1>title</H1>').group()
'<H1>'
>>> re.search(r'<.+?>', '<H1>title</H1>').group()
'<H1>'
```

{m}	Specifies that exactly <i>m</i> copies of the previous RE should be matched; fewer matches cause the entire RE not to match. For example, <code>a{6}</code> will match exactly six 'a' characters, but not five.
-----	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------



{m,n}	Causes the resulting RE to match from <i>m</i> to <i>n</i> repetitions of the preceding RE, attempting to match as many repetitions as possible. For example, <code>a{3,5}</code> will match from 3 to 5 'a' characters. Omitting <i>m</i> specifies a lower bound of zero, and omitting <i>n</i> specifies an infinite upper bound. As an example, <code>a{4,}b</code> will match <code>aaaab</code> or a thousand 'a' characters followed by a <code>b</code> , but not <code>aaab</code> . The comma may not be omitted or the modifier would be confused with the previously described form.
-------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------



{m,n}?	Causes the resulting RE to match from <i>m</i> to <i>n</i> repetitions of the preceding RE, attempting to match as few repetitions as possible. This is the non-greedy version of the previous qualifier. For example, on the 6-character string <code>'aaaaaa'</code> , <code>a{3,5}</code> will match 5 'a' characters, while <code>a{3,5}?</code> will only match 3 characters.
--------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

```
>>> re.search(r'ab{3,4}?', 'aaabbb').group()
'abbb'
>>> re.search(r'ab{3,4}?', 'aaabbbbbbb').group()
'abbb'
>>> re.search(r'ab{3,}?', 'aaabbbbbbb').group()
'abbb'
```

\number	Matches the contents of the group of the same number. Groups are numbered starting from 1. For example, <code>(.+) \1</code> matches <code>'the the'</code> or <code>'55 55'</code> , but not <code>'thethe'</code> (note the space after the group).
---------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------



\A	Matches only at the start of the string.
----	------------------------------------------

<code>\b</code>	Matches the empty string, but only at the beginning or end of a word. A word is defined as a sequence of Unicode alphanumeric or underscore characters, so the end of a word is indicated by whitespace or a non-alphanumeric, non-underscore Unicode character. Note that formally, <code>\b</code> is defined as the boundary between a <code>\w</code> and a <code>\W</code> character (or vice versa), or between <code>\w</code> and the beginning/end of the string. This means that <code>r'\bfoo\b'</code> matches <code>'foo'</code> , <code>'foo.'</code> , <code>'(foo)'</code> , <code>'bar foo baz'</code> but not <code>'foobar'</code> or <code>'foo3'</code> .
<code>\B</code>	Matches the empty string, but only when it is not at the beginning or end of a word. This means that <code>r'py\B'</code> matches <code>'python'</code> , <code>'py3'</code> , <code>'py2'</code> , but not <code>'py'</code> , <code>'py.'</code> , or <code>'py!'</code> . <code>\B</code> is just the opposite of <code>\b</code> , so word characters are Unicode alphanumerics or the underscore, although this can be changed by using the ASCII flag.
<code>\d</code>	<p>For Unicode (str) patterns: Matches any Unicode decimal digit (that is, any character in Unicode character category <code>[Nd]</code>). This includes <code>[0-9]</code>, and also many other digit characters. If the ASCII flag is used only <code>[0-9]</code> is matched (but the flag affects the entire regular expression, so in such cases using an explicit <code>[0-9]</code> may be a better choice).</p> <p>For 8-bit (bytes) patterns: Matches any decimal digit; this is equivalent to <code>[0-9]</code>.</p>
<code>\D</code>	Matches any character which is not a Unicode decimal digit. This is the opposite of <code>\d</code> . If the ASCII flag is used this becomes the equivalent of <code>[^0-9]</code> (but the flag affects the entire regular expression, so in such cases using an explicit <code>[^0-9]</code> may be a better choice).
<code>\s</code>	<p>For Unicode (str) patterns: Matches Unicode whitespace characters (which includes <code>[\t\n\r\f\v]</code>, and also many other characters, for example the non-breaking spaces mandated by typography rules in many languages). If the ASCII flag is used, only <code>[\t\n\r\f\v]</code> is matched (but the flag affects the entire regular expression, so in such cases using an explicit <code>[\t\n\r\f\v]</code> may be a better choice).</p> <p>For 8-bit (bytes) patterns: Matches characters considered whitespace in the ASCII character set; this is equivalent to <code>[\t\n\r\f\v]</code>.</p>
<code>\S</code>	Matches any character which is not a Unicode whitespace character. This is the opposite of <code>\s</code> . If the ASCII flag is used this becomes the equivalent of <code>[^\t\n\r\f\v]</code> (but the flag affects the entire regular expression, so in such cases using an explicit <code>[^\t\n\r\f\v]</code> may be a better choice).

\w	<p>For Unicode (str) patterns:</p> <p>Matches Unicode word characters; this includes most characters that can be part of a word in any language, as well as numbers and the underscore. If the ASCII flag is used, only [a-zA-Z0-9_] is matched (but the flag affects the entire regular expression, so in such cases using an explicit [a-zA-Z0-9_] may be a better choice).</p> <p>For 8-bit (bytes) patterns:</p> <p>Matches characters considered alphanumeric in the ASCII character set; this is equivalent to [a-zA-Z0-9_].</p>
\W	<p>Matches any character which is not a Unicode word character. This is the opposite of \w. If the ASCII flag is used this becomes the equivalent of [^a-zA-Z0-9_] (but the flag affects the entire regular expression, so in such cases using an explicit [^a-zA-Z0-9_] may be a better choice).</p>
\Z	<p>Matches only at the end of the string.</p>

Most of the standard escapes supported by Python string literals are also accepted by the regular expression parser:

<code>\a</code>	<code>\b</code>	<code>\f</code>	<code>\n</code>
<code>\r</code>	<code>\t</code>	<code>\u</code>	<code>\U</code>
<code>\v</code>	<code>\x</code>	<code>\\</code>	

`re.compile(pattern, flags=0)`[¶]: Compile a regular expression pattern into a regular expression object, which can be used for matching using its `match()` and `search()` methods, described below.

...

Using `re.compile()` and saving the resulting regular expression object for reuse is more efficient when the expression will be used several times in a single program.

```
>>> text='in the the the 55 55 raj raj level'
>>> compile_obj=re.compile(r'(.+) \1')
>>> value=compile_obj.findall(text)
>>> value
['the', '55', 'raj']
```

```
a = re.compile(r"""\d + # the integral part
               \.  # the decimal point
               \d * # some fractional digits""", re.X)
b = re.compile(r"\d+\.\d*")
```

`re.search(pattern, string, flags=0)`[¶]: Scan through *string* looking for the first location where the regular expression *pattern* produces a match, and return a corresponding `match object`. Return `None` if no position in the string matches the pattern; note that this is different from finding a zero-length match at some point in the string.

```
>>> re.search(r'boy','a boy in the park').group()
'boy'
>>> re.search(r'girl','a boy in the park').group()
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    re.search(r'girl','a boy in the park').group()
AttributeError: 'NoneType' object has no attribute 'group'
```

`re.match(pattern, string, flags=0)`: If zero or more characters at the beginning of *string* match the regular expression *pattern*, return a corresponding `match object`. Return `None` if the string does not match the pattern; note that this is different from a zero-length match.

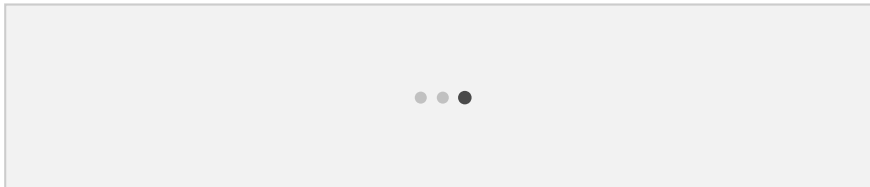
```
>>> re.match(r'^boy','a boy in the park')
>>> re.match(r'^a','a boy in the park')
<_sre.SRE_Match object; span=(0, 1), match='a'>
```

If you want to locate a match anywhere in *string*, use `search()` instead.

search() vs. match():

[re.match\(\)](#) checks for a match only at the beginning of the string

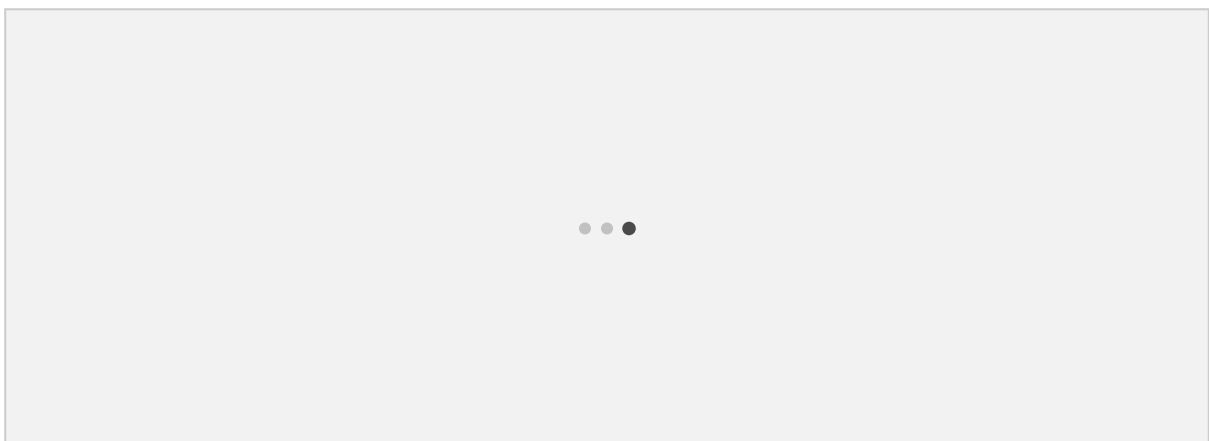
[re.search\(\)](#) checks for a match anywhere in the string



Note however that in MULTILINE mode [match\(\)](#) only matches at the beginning of the string, whereas using [search\(\)](#) with a regular expression beginning with '^' will match at the beginning of each line.



re.fullmatch(pattern, string, flags=0): If the whole *string* matches the regular expression *pattern*, return a corresponding [match object](#). Return None if the string does not match the pattern; note that this is different from a zero-length match.



`re.split(pattern, string, maxsplit=0, flags=0)`¹: Split string by the occurrences of pattern. If capturing parentheses are used in pattern, then the text of all groups in the pattern are also returned as part of the resulting list. If maxsplit is nonzero, at most maxsplit splits occur, and the remainder of the string is returned as the final element of the list.

```
>>> re.split(r'\.', 'rajesh is at home.now')
['rajesh is at home', 'now']
>>> re.split(r' ', 'rajesh is at home.now')
['rajesh', 'is', 'at', 'home.now']
>>> re.split(r' |\.', 'rajesh is at home.now')
['rajesh', 'is', 'at', 'home', 'now']

>>> re.split('\W+', 'Words, words, words.')
['Words', 'words', 'words', '']
>>> re.split('(\W+)', 'Words, words, words.')
['Words', ',', 'words', ',', 'words', '.', '']
>>> re.split('\W+', 'Words, words, words.', 1)
['Words', 'words, words.']
>>> re.split('\s+', 'Words, words, words.')
['Words,', 'words,', 'words.']

>>> re.split('[a-f]+', '0a3B9')
['0', '3B9']
>>> re.split('[a-f]+', '0a3B9', flags=re.IGNORECASE)
['0', '3', '9']
```

I/P: `l = "one two 3.4 5,6 seven.eight nine,ten"`

O/P: `["one", "two", "3.4", "5,6", "seven", "eight", "nine", "ten"]`

Command is:

```
>>> re.split('\s|(?<!\d)[.](?!\\d)', l)
['one', 'two', '3.4', '5,6', 'seven', 'eight', 'nine', 'ten']
```

`re.findall(pattern, string, flags=0)`: Return all non-overlapping matches of *pattern* in *string*, as a list of strings. The *string* is scanned left-to-right, and matches are returned in the order found. If one or more groups are present in the pattern, return a list of groups; this will be a list of tuples if the pattern has more than one group. Empty matches are included in the result unless they touch the beginning of another match.

`re.finditer(pattern, string, flags=0)`: Return an [iterator](#) yielding [match objects](#) over all non-overlapping matches for the RE *pattern* in *string*. The *string* is scanned left-to-right, and matches are returned in the order found. Empty matches are included in the result unless they touch the beginning of another match.

...

```
>>> m = re.match(r"(\w+) (\w+)", "Isaac Newton, physicist")
>>> m.group(0)
'Isaac Newton'
>>> m.group(1)
'Isaac'
>>> m.group(2)
'Newton'
>>> m.groups()
('Isaac', 'Newton')
>>> m.group(1,2)
('Isaac', 'Newton')
```

`re.purge()`: Clear the regular expression cache.

Python Data Structures:

2-3 Trees:

...

...

...

```
>>> f=open('C:\\Users\\v528250\\Desktop\\1.txt','w')
>>> l=['1\\n','2\\n','3\\n']
>>> f.writelines(l)
>>> f.close()
>>>
```

1	1
2	2
3	3

f.flush() - Writes the contents that sits on buffer(f in this case) so far, into the file.

in below code, greet function will be executed, only if we execute the script directly. If the script is imported as module, then it will not be executed.

...

```
Microsoft Windows [Version 10.0.10586]
(c) 2015 Microsoft Corporation. All rights reserved.

C:\Users\wrih>cd \Python34\myscripts

C:\Python34\myscripts>python argv_demo.py
Here is sys.argv: ['argv_demo.py']
Here is the current configuration:
Program: argv_demo.py
Source: default.src
Destination: default.dst

C:\Python34\myscripts>
C:\Python34\myscripts>python argv_demo.py one.src
Here is sys.argv: ['argv_demo.py', 'one.src']
Here is the current configuration:
Program: argv_demo.py
Source: one.src
Destination: default.dst

C:\Python34\myscripts>python argv_demo.py one.src two.dst
Here is sys.argv: ['argv_demo.py', 'one.src', 'two.dst']
Here is the current configuration:
Program: argv_demo.py
Source: one.src
Destination: two.dst

C:\Python34\myscripts>python argv_demo.py one.src two.dst
Here is sys.argv: ['C:\Python34\myscripts\argv_demo.py']
Here is the current configuration:
Program: C:\Python34\myscripts\argv_demo.py
Source: default.src
Destination: default.dst

C:\Python34\myscripts>_

File Edit Format Run Options Window Help

''' Accept command line arguments to the module '''

program = 'argv_demo.py'
source = 'default.src'
dest = 'default.dst'

def show_config():
    print('Here is the current configuration:')
    print('Program: %s' % program)
    print('Source: %s' % source)
    print('Destination: %s' % dest)

if __name__ == '__main__':
    import sys
    print('Here is sys.argv: %s' % sys.argv)
    if len(sys.argv) > 2: # Two or more arguments passed
        program, source, dest = sys.argv[:3]
    elif len(sys.argv) > 1: # Only one argument passed
        program, source = sys.argv[:2]
    else: # No arguments passed
        program = sys.argv[0]
    show_config()
```

```
>>> type(l)
<type 'list'>
>>> l.__class__
<type 'list'>
>>> type(l)==l.__class__
True
>>> isinstance(l,list)
True
```

What does `__str__(self)`, does in a class:

```
>>> class A:
...     pass
...
>>> a=A()
>>> print('%s' % a)
<__main__.A instance at 0x11e990>
>>> class A:
...     def __str__(self):
...         return('Value Returned')
...
>>> a=A()
>>> print('%s' % a)
Value Returned
```

[Reading from a CSV spreadsheet](#)

```
import csv

with open('example.csv') as csvfile:
    readCSV = csv.reader(csvfile, delimiter=',')
    print(readCSV)

    for row in readCSV:
        print(row)
```

```
< csv.reader object at 0x000000000322A9A0>
['1/2/2014', '5', '8', 'red']
['1/3/2014', '5', '2', 'green']
['1/4/2014', '9', '1', 'blue']
```

[urllib:](#)

```
import urllib.request
import urllib.parse

#x = urllib.request.urlopen('https://www.google.com')
#print(x.read())

url = 'http://pythonprogramming.net'
values = {'s': 'basic',
          'submit': 'search'}

data = urllib.parse.urlencode(values)
data = data.encode('utf-8')
req = urllib.request.Request(url, data)
resp = urllib.request.urlopen(req)
respData = resp.read()

print(respData)
```

[argparse:](#)

```

import argparse
import sys

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('--x', type=float, default=1.0,
                        help='What is the first number?')
    parser.add_argument('--y', type=float, default=1.0,
                        help='What is the second number?')
    parser.add_argument('--operation', type=str, default='add',
                        help='What operation? (add, sub, mul, or div)')
    args = parser.parse_args()
    sys.stdout.write(str(calc(args)))

def calc(args):
    if args.operation == 'add':
        return args.x + args.y
    elif args.operation == 'sub':
        return args.x - args.y
    elif args.operation == 'mul':
        return args.x * args.y
    elif args.operation == 'div':
        return args.x / args.y

if __name__ == '__main__':
    main()

```

...

[multiprocessing](#): join is used to wait for other process to get completed.

```
import multiprocessing

def spawn():
    print('Spawned!')

if __name__ == '__main__':
    for i in range(55):
        p = multiprocessing.Process(target=spawn)
        p.start()
        #p.join()
```

Module 1: `imp`

`imp`: The [imp](#) package is pending deprecation in favor of [importlib](#).

```
import imp
>>> imp.find_module('os')
(<_io.TextIOWrapper name='C:\\Program Files\\Python36\\lib\\os.py' mode='r'
encoding='utf-8'>, 'C:\\Program Files\\Python36\\lib\\os.py', ('.py', 'r', 1))
```

Module 2: `optparse`

`optparse` — Parser for command line options: Deprecated since version 3.2: The [optparse](#) module is deprecated and will not be developed further; development will continue with the [argparse](#) module.

Module 3: `logging`

Program 1:

```
import logging
logging.basicConfig(level=logging.DEBUG)
logger = logging.getLogger(__name__)

logger.info('Start reading database')
records = {'john': 55, 'tom': 66}
logger.debug('Records: %s', records)
logger.info('Updating records ...')
```

```
logger.info('Finish updating records')
```

Execution:

```
INFO:__main__:Start reading database
DEBUG:__main__:Records: {'john': 55, 'tom': 66}
INFO:__main__:Updating records ...
INFO:__main__:Finish updating records
```

Program 2: Writing into a file

```
import logging

logger = logging.getLogger(__name__)
logger.setLevel(logging.INFO)

# create a file handler
handler = logging.FileHandler('hello.log')
handler.setLevel(logging.INFO)

# create a logging format
formatter = logging.Formatter('%(asctime)s - %(name)s -
%(levelname)s - %(message)s')
handler.setFormatter(formatter)

# add the handlers to the logger
logger.addHandler(handler)

logger.info('Hello baby')
```

Execution: written into the file hello.log

```
2017-12-19 21:50:07,594 - __main__ - INFO - Hello baby
```

Module 4: [threading: Manage concurrent threads](https://pymotw.com/2/threading/#module-threading)

_thread: This module provides low-level primitives for working with multiple threads (also called *light-weight processes* or *tasks*) — multiple threads of control sharing their global data space. For synchronization, simple locks (also called *mutexes* or *binary semaphores*) are provided. The [threading](#) module provides an easier to use and higher-level threading API built on top of this module.

Daemon Threads: Some threads do background tasks, like sending keepalive packets, or performing periodic garbage collection, or whatever. These are only useful when the main program is running, and it's okay to kill them off once the other, non-daemon, threads have exited.

Without daemon threads, you'd have to keep track of them, and tell them to exit, before your program can completely quit. By setting them as daemon threads, you can let them run and forget about them, and when your program quits, any daemon threads are killed automatically.

Thread Objects:

Program 1:

```
import threading

def worker():
    """thread worker function"""
    print('Worker')

for i in range(5):
    threading.Thread(target=worker).start()
```

Execution:

Worker
Worker
Worker
Worker
Worker

Program 2:

```
import threading

def worker(name):
```

```

    """thread worker function"""
    print(name)

for i in range(5):
    name=input('Enter Name:')
    threading.Thread(target=worker(name)).start()

```

Execution:

```

Enter Name:Rajesh
Rajesh
Enter Name:Paleru
Paleru
Enter Name:Anusha
Anusha
Enter Name:Paleru
Paleru
Enter Name:Shanmukh Shiv
Shanmukh Shiv

```

Program 3:

```

import threading

def worker(num):
    """thread worker function"""
    print('Worker: {0}'.format(num))

for i in range(5):
    threading.Thread(target=worker(i)).start()

```

Execution:

```

Worker: 0
Worker: 1
Worker: 2
Worker: 3
Worker: 4

```


Determining the Current Thread:

Program 4:

```
import threading
import time

def fun1():
    print('inside fun1')
    time.sleep(1)
    print('fun1 started')
    time.sleep(1)
    print('fun1 ended')

def fun2():
    print('inside fun2')
    time.sleep(1)
    print('fun2 started')
    time.sleep(1)
    print('fun2 ended')

def fun3():
    print('inside fun3')
    time.sleep(1)
    print('fun3 started')
    time.sleep(1)
    print('fun3 ended')

threading.Thread(target=fun1).start()
threading.Thread(target=fun2).start()
threading.Thread(target=fun3).start()
```

Execution:

```
inside fun1
inside fun2
inside fun3
fun2 started
fun1 started
```

fun3 started

fun3 ended

fun1 ended

fun2 ended

Program 5:

```
import threading
import time

def worker():
    print(threading.currentThread().getName(), 'Starting')
    time.sleep(2)
    print(threading.currentThread().getName(), 'Exiting')

def my_service():
    print(threading.currentThread().getName(), 'Starting')
    time.sleep(3)
    print(threading.currentThread().getName(), 'Exiting')

t = threading.Thread(name='my_service', target=my_service)
w = threading.Thread(name='worker', target=worker)
w2 = threading.Thread(target=worker) # use default name

w.start()
w2.start()
t.start()
```

Execution:

worker Starting

Thread-1 Starting

my_service Starting

worker Exiting

Thread-1 Exiting

my_service Exiting

Program 6: threading & logging

```
import logging
```

```

import threading
import time

logging.basicConfig(level=logging.DEBUG,
                    format='[% (levelname)s - %(asctime)s]
(% (threadName)s -10s) %(message)s',
                    )

def worker():
    logging.debug('Starting')
    time.sleep(2)
    logging.debug('Exiting')

def my_service():
    logging.debug('Starting')
    time.sleep(3)
    logging.debug('Exiting')

t = threading.Thread(name='my_service', target=my_service)
w = threading.Thread(name='worker', target=worker)
w2 = threading.Thread(target=worker) # use default name

w.start()
w2.start()
t.start()

```

Execution:

```

[DEBUG - 2017-12-19 21:03:40,723] (worker ) Starting
[DEBUG - 2017-12-19 21:03:40,723] (Thread-1 ) Starting
[DEBUG - 2017-12-19 21:03:40,723] (my_service) Starting
[DEBUG - 2017-12-19 21:03:42,740] (Thread-1 ) Exiting
[DEBUG - 2017-12-19 21:03:42,740] (worker ) Exiting
[DEBUG - 2017-12-19 21:03:43,730] (my_service) Exiting

```

Daemon vs. Non-Daemon Threads

Program 1: If non-daemon completes before daemon thread

```
import threading
import time
import logging

logging.basicConfig(level=logging.DEBUG,
                    format='(%(threadName)-10s) %(message)s',
                    )

def daemon():
    logging.debug('Starting')
    time.sleep(2)
    logging.debug('Exiting')

d = threading.Thread(name='daemon', target=daemon)
d.setDaemon(True)

def non_daemon():
    logging.debug('Starting')
    logging.debug('Exiting')

t = threading.Thread(name='non-daemon', target=non_daemon)

d.start()
t.start()
```

Execution: Notice that the output does not include the "Exiting" message from the daemon thread, since all of the non-daemon threads (including the main thread) exit before the daemon thread wakes up from its two second sleep.

```
(daemon ) Starting
(non-daemon) Starting
(non-daemon) Exiting
```

Program 2: Manual: If non-daemon doesn't completes before daemon thread

```
import threading
import time
```

```
import logging

logging.basicConfig(level=logging.DEBUG,
                    format='%(threadName)-10s) %(message)s',
                    )

def daemon():
    logging.debug('Starting')
    time.sleep(2)
    logging.debug('Exiting')

d = threading.Thread(name='daemon', target=daemon)
d.setDaemon(True)

def non_daemon():
    logging.debug('Starting')
    time.sleep(3)
    logging.debug('Exiting')

t = threading.Thread(name='non-daemon', target=non_daemon)

d.start()
t.start()
```

Execution:

```
(daemon ) Starting
(non-daemon) Starting
(daemon ) Exiting
(non-daemon) Exiting
```

```

without join:
+---+---+-----
|   |   |
|   |   +.....
+.....
                                main-thread
                                child-thread(short)
                                child-thread(long)

with join
+---+---+-----*****+###
|   |   |
|   |   +.....join()
+.....join().....
                                main-thread
                                child-thread(short)
                                child-thread(long)

with join and demon thread
+---+---+-----*****+###
|   |   |
|   |   +.....join()
|   |   +.....join().....
+,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
                                parent-thread
                                child-thread(short)
                                child-thread(long)
                                child-thread(long+demonized)

'-' main-thread/parent-thread/main-program execution
'.' child-thread execution
'#' optional parent-thread execution after join()-blocked parent-thread could
    continue
'*' main-thread 'sleeping' in join-method, waiting for child-thread to finish
',' demonized thread - 'ignores' lifetime of other threads;
    terminates when main-programs exits; is normally meant for
    join-independent tasks

```

Program 3: Programmatic: To wait until a daemon thread has completed its work, use the `join()` method

```
import threading
```

```
import time
```

```
import logging
```

```
logging.basicConfig(level=logging.DEBUG,
                    format='(%(threadName)-10s) %(message)s',
                    )
```

```
def daemon():
```

```
    logging.debug('Starting')
```

```
    time.sleep(2)
```

```
    logging.debug('Exiting')
```

```
d = threading.Thread(name='daemon', target=daemon)
```

```
d.setDaemon(True)
```

```

def non_daemon():
    logging.debug('Starting')
    logging.debug('Exiting')

t = threading.Thread(name='non-daemon', target=non_daemon)

d.start()
t.start()

d.join()
t.join()

```

Execution: Waiting for the daemon thread to exit using `join()` means it has a chance to produce its "Exiting" message.

```

(daemon ) Starting
(non-daemon) Starting
(daemon ) Exiting
(non-daemon) Exiting

```

By default, `join()` blocks indefinitely. It is also possible to pass a timeout argument (a float representing the number of seconds to wait for the thread to become inactive). If the thread does not complete within the timeout period, `join()` returns anyway.

```

import threading
import time
import logging

logging.basicConfig(level=logging.DEBUG,
                    format='(%(threadName)-10s) %(message)s',
                    )

def daemon():
    logging.debug('Starting')
    time.sleep(2)
    logging.debug('Exiting')

d = threading.Thread(name='daemon', target=daemon)

```

```

d.setDaemon(True)

def non_daemon():
    logging.debug('Starting')
    logging.debug('Exiting')

t = threading.Thread(name='non-daemon', target=non_daemon)

d.start()
t.start()

d.join(1)
print('d.isAlive()', d.isAlive())
t.join()
if not t.isAlive():
    d.join()

```

Output: *Since the timeout passed is less than the amount of time the daemon thread sleeps, the thread is still "alive" after join() returns.*

```

(daemon ) Starting
(non-daemon) Starting
(non-daemon) Exiting
d.isAlive() True
(daemon ) Exiting

```

Enumerating All Threads

```

import random, time, threading, logging

logging.basicConfig(level=logging.DEBUG, format='%(threadName)s) %(message)s')

def worker():
    pause=random.randint(1,10)
    logging.debug('sleeping %s',pause)
    time.sleep(pause)
    logging.debug('ending')
    return

```


#below one starts all threads, which can be found in enumerate.

```
for i in range(3):
    t=threading.Thread(target=worker)
    t.setDaemon(True)
    t.start()

main_thread=threading.current_thread()
for thread in threading.enumerate():
    if thread is main_thread:
        continue
    logging.debug('joining %s', thread.getName())
    thread.join()
```

Output:

```
(Thread-1) sleeping 7
(Thread-2) sleeping 3
(Thread-3) sleeping 8
(MainThread) joining Thread-1
(Thread-2) ending
(Thread-1) ending
(MainThread) joining Thread-2
(MainThread) joining Thread-3
(Thread-3) ending
```

Subclassing Thread

At start-up, a Thread does some basic initialization and then calls its run() method, which calls the target function passed to the constructor. To create a subclass of Thread, override run() to do whatever is necessary.

```
import threading
import logging
```

```
logging.basicConfig(level=logging.DEBUG, format='%(threadName)
-10s) %(message)s',)
```

```

class MyThread(threading.Thread):
    def run(self):
        logging.debug('running')
        return

for i in range(5):
    t = MyThread()
    t.start()

```

Output:

```

(Thread-1 ) running
(Thread-2 ) running
(Thread-3 ) running
(Thread-4 ) running
(Thread-5 ) running

```

Because the *args* and *kwargs* values passed to the **Thread** constructor are saved in private variables, they are not easily accessed from a subclass. To pass arguments to a custom thread type, redefine the constructor to save the values in an instance attribute that can be seen in the subclass.

```

import threading
import logging

logging.basicConfig(level=logging.DEBUG,
                    format='(%(threadName)-10s) %(message)s',
                    )

```

```

class MyThreadWithArgs(threading.Thread):

    def __init__(self, group=None, target=None, name=None,
                 args=(), kwargs=None, verbose=None):
        threading.Thread.__init__(self, group=group,
target=target, name=name)
        self.args = args
        self.kwargs = kwargs
        return

    def run(self):

```

```
        logging.debug('running with %s and %s', self.args,
self.kwargs)
        return

for i in range(5):
    t = MyThreadWithArgs(args=(i,), kwargs={'a': 'A', 'b': 'B'})
    t.start()
```

Output:

```
(Thread-1 ) running with (0,) and {'a': 'A', 'b': 'B'}
(Thread-2 ) running with (1,) and {'a': 'A', 'b': 'B'}
(Thread-3 ) running with (2,) and {'a': 'A', 'b': 'B'}
(Thread-4 ) running with (3,) and {'a': 'A', 'b': 'B'}
(Thread-5 ) running with (4,) and {'a': 'A', 'b': 'B'}
```