Chapter 22: List slicing (selecting parts of lists)

Section 22.1: Using the third "step" argument

```
lst = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']

lst[::2]
# Output: ['a', 'c', 'e', 'g']

lst[::3]
# Output: ['a', 'd', 'g']
```

Section 22.2: Selecting a sublist from a list

```
lst = ['a', 'b', 'c', 'd', 'e']

lst[2:4]
# Output: ['c', 'd']

lst[2:]
# Output: ['c', 'd', 'e']

lst[:4]
# Output: ['a', 'b', 'c', 'd']
```

Section 22.3: Reversing a list with slicing

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

# steps through the list backwards (step=-1)
b = a[::-1]

# built-in list method to reverse 'a'
a.reverse()

if a = b:
    print(True)

print(b)

# Output:
# True
# [5, 4, 3, 2, 1]
```

Section 22.4: Shifting a list using slicing

```
def shift_list(array, s):
    """Shifts the elements of a list to the left or right.

    Args:
        array - the list to shift
        s - the amount to shift the list ('+': right-shift, '-': left-shift)

    Returns:
        shifted_array - the shifted list
```

```
    """
    # calculate actual shift amount (e.g., 11 --> 1 if length of the array is 5)
    s %= len(array)

    # reverse the shift direction to be more intuitive
    s *= -1

    # shift array with list slicing
    shifted_array = array[s:] + array[:s]

    return shifted_array

my_array = [1, 2, 3, 4, 5]

# negative numbers
shift_list(my_array, -7)
>>> [3, 4, 5, 1, 2]

# no shift on numbers equal to the size of the array
shift_list(my_array, 5)
>>> [1, 2, 3, 4, 5]

# works on positive numbers
shift_list(my_array, 3)
>>> [3, 4, 5, 1, 2]
```

Chapter 23: groupby()

Parameter	Details
iterable	Any python iterable
key	Function(criteria) on which to group the iterable

In Python, the `itertools.groupby()` method allows developers to group values of an iterable class based on a specified property into another iterable set of values.

Section 23.1: Example 4

In this example we see what happens when we use different types of iterable.

```
things = [("animal", "bear"), ("animal", "duck"), ("plant", "cactus"), ("vehicle", "harley"), \
          ("vehicle", "speed boat"), ("vehicle", "school bus")]
dic = {}
f = lambda x: x[0]
for key, group in groupby(sorted(things, key=f), f):
    dic[key] = list(group)
dic
```

Results in

```
{'animal': [('animal', 'bear'), ('animal', 'duck')],
 'plant': [('plant', 'cactus')],
 'vehicle': [('vehicle', 'harley'),
  ('vehicle', 'speed boat'),
  ('vehicle', 'school bus')]}
```

This example below is essentially the same as the one above it. The only difference is that I have changed all the tuples to lists.

```
things = [["animal", "bear"], ["animal", "duck"], ["vehicle", "harley"], ["plant", "cactus"], \
          ["vehicle", "speed boat"], ["vehicle", "school bus"]]
dic = {}
f = lambda x: x[0]
for key, group in groupby(sorted(things, key=f), f):
    dic[key] = list(group)
dic
```

Results

```
{'animal': [['animal', 'bear'], ['animal', 'duck']],
 'plant': [['plant', 'cactus']],
 'vehicle': [['vehicle', 'harley'],
  ['vehicle', 'speed boat'],
  ['vehicle', 'school bus']]}
```

Section 23.2: Example 2

This example illustrates how the default key is chosen if we do not specify any

```
c = groupby(['goat', 'dog', 'cow', 1, 1, 2, 3, 11, 10, ('persons', 'man', 'woman')])
dic = {}
for k, v in c:
    dic[k] = list(v)
```

```
dic
```

Results in

```
{1: [1, 1],
 2: [2],
 3: [3],
 ('persons', 'man', 'woman'): [('persons', 'man', 'woman')],
 'cow': ['cow'],
 'dog': ['dog'],
 10: [10],
 11: [11],
 'goat': ['goat']}
```

Notice here that the tuple as a whole counts as one key in this list

Section 23.3: Example 3

Notice in this example that mulato and camel don't show up in our result. Only the last element with the specified key shows up. The last result for c actually wipes out two previous results. But watch the new version where I have the data sorted first on same key.

```
list_things = ['goat', 'dog', 'donkey', 'mulato', 'cow', 'cat', ('persons', 'man', 'woman'), \
               'wombat', 'mongoose', 'malloo', 'camel']
c = groupby(list_things, key=lambda x: x[0])
dic = {}
for k, v in c:
    dic[k] = list(v)
dic
```

Results in

```
{'c': ['camel'],
 'd': ['dog', 'donkey'],
 'g': ['goat'],
 'm': ['mongoose', 'malloo'],
 'persons': [('persons', 'man', 'woman')],
 'w': ['wombat']}
```

Sorted Version

```
list_things = ['goat', 'dog', 'donkey', 'mulato', 'cow', 'cat', ('persons', 'man', 'woman'), \
               'wombat', 'mongoose', 'malloo', 'camel']
sorted_list = sorted(list_things, key = lambda x: x[0])
print(sorted_list)
print()
c = groupby(sorted_list, key=lambda x: x[0])
dic = {}
for k, v in c:
    dic[k] = list(v)
dic
```

Results in

```
['cow', 'cat', 'camel', 'dog', 'donkey', 'goat', 'mulato', 'mongoose', 'malloo', ('persons', 'man',
'woman'), 'wombat']

{'c': ['cow', 'cat', 'camel'],
```

```
'd': ['dog', 'donkey'],
'g': ['goat'],
'm': ['mulato', 'mongoose', 'malloo'],
'persons': [('persons', 'man', 'woman')],
'w': ['wombat']}
```

Chapter 24: Linked lists

A linked list is a collection of nodes, each made up of a reference and a value. Nodes are strung together into a sequence using their references. Linked lists can be used to implement more complex data structures like lists, stacks, queues, and associative arrays.

Section 24.1: Single linked list example

This example implements a linked list with many of the same methods as that of the built-in list object.

```
class Node:
    def __init__(self, val):
        self.data = val
        self.next = None

    def getData(self):
        return self.data

    def getNext(self):
        return self.next

    def setData(self, val):
        self.data = val

    def setNext(self, val):
        self.next = val

class LinkedList:
    def __init__(self):
        self.head = None

    def isEmpty(self):
        """Check if the list is empty"""
        return self.head is None

    def add(self, item):
        """Add the item to the list"""
        new_node = Node(item)
        new_node.setNext(self.head)
        self.head = new_node

    def size(self):
        """Return the length/size of the list"""
        count = 0
        current = self.head
        while current is not None:
            count += 1
            current = current.getNext()
        return count

    def search(self, item):
        """Search for item in list. If found, return True. If not found, return False"""
        current = self.head
        found = False
        while current is not None and not found:
            if current.getData() is item:
                found = True
            else:
                current = current.getNext()
```

```
        return found

    def remove(self, item):
        """Remove item from list. If item is not found in list, raise ValueError"""
        current = self.head
        previous = None
        found = False
        while current is not None and not found:
            if current.getData() is item:
                found = True
            else:
                previous = current
                current = current.getNext()
        if found:
            if previous is None:
                self.head = current.getNext()
            else:
                previous.setNext(current.getNext())
        else:
            raise ValueError
            print 'Value not found.'

    def insert(self, position, item):
        """
        Insert item at position specified. If position specified is
        out of bounds, raise IndexError
        """
        if position > self.size() - 1:
            raise IndexError
            print "Index out of bounds."
        current = self.head
        previous = None
        pos = 0
        if position is 0:
            self.add(item)
        else:
            new_node = Node(item)
            while pos < position:
                pos += 1
                previous = current
                current = current.getNext()
            previous.setNext(new_node)
            new_node.setNext(current)

    def index(self, item):
        """
        Return the index where item is found.
        If item is not found, return None.
        """
        current = self.head
        pos = 0
        found = False
        while current is not None and not found:
            if current.getData() is item:
                found = True
            else:
                current = current.getNext()
                pos += 1
        if found:
            pass
        else:
            pos = None
```

```
        return pos

    def pop(self, position = None):
        """
        If no argument is provided, return and remove the item at the head.
        If position is provided, return and remove the item at that position.
        If index is out of bounds, raise IndexError
        """
        if position > self.size():
            print 'Index out of bounds'
            raise IndexError

        current = self.head
        if position is None:
            ret = current.getData()
            self.head = current.getNext()
        else:
            pos = 0
            previous = None
            while pos < position:
                previous = current
                current = current.getNext()
                pos += 1
                ret = current.getData()
            previous.setNext(current.getNext())
        print ret
        return ret

    def append(self, item):
        """Append item to the end of the list"""
        current = self.head
        previous = None
        pos = 0
        length = self.size()
        while pos < length:
            previous = current
            current = current.getNext()
            pos += 1
        new_node = Node(item)
        if previous is None:
            new_node.setNext(current)
            self.head = new_node
        else:
            previous.setNext(new_node)

    def printList(self):
        """Print the list"""
        current = self.head
        while current is not None:
            print current.getData()
            current = current.getNext()
```

Usage functions much like that of the built-in list.

```
ll = LinkedList()
ll.add('l')
ll.add('H')
ll.insert(1,'e')
ll.append('l')
ll.append('o')
ll.printList()
```

```
H
e
l
l
o
```

Chapter 25: Linked List Node

Section 25.1: Write a simple Linked List Node in python

A linked list is either:

- the empty list, represented by None, or
- a node that contains a cargo object and a reference to a linked list.

```
#! /usr/bin/env python

class Node:
      def __init__(self, cargo=None, next=None):
          self.car = cargo
          self.cdr = next
      def __str__(self):
          return str(self.car)


      def display(lst):
          if lst:
             w("%s " % lst)
             display(lst.cdr)
          else:
             w("nil\n")
```

Chapter 26: Filter

Parameter	Details
function	*callable* that determines the condition or `None` then use the identity function for filtering (*positional-only*)
iterable	iterable that will be filtered (*positional-only*)

Section 26.1: Basic use of filter

To `filter` discards elements of a sequence based on some criteria:

```
names = ['Fred', 'Wilma', 'Barney']

def long_name(name):
    return len(name) > 5
```

Python 2.x `Version ≥ 2.0`

```
filter(long_name, names)
# Out: ['Barney']

[name for name in names if len(name) > 5] # equivalent list comprehension
# Out: ['Barney']


from itertools import ifilter
ifilter(long_name, names)       # as generator (similar to python 3.x filter builtin)
# Out: <itertools.ifilter at 0x4197e10>
list(ifilter(long_name, names)) # equivalent to filter with lists
# Out: ['Barney']

(name for name in names if len(name) > 5) # equivalent generator expression
# Out: <generator object <genexpr> at 0x0000000003FD5D38>
```

Python 2.x `Version ≥ 2.6`

```
# Besides the options for older python 2.x versions there is a future_builtin function:
from future_builtins import filter
filter(long_name, names)       # identical to itertools.ifilter
# Out: <itertools.ifilter at 0x3eb0ba8>
```

Python 3.x `Version ≥ 3.0`

```
filter(long_name, names)        # returns a generator
# Out: <filter at 0x1fc6e443470>
list(filter(long_name, names))  # cast to list
# Out: ['Barney']

(name for name in names if len(name) > 5) # equivalent generator expression
# Out: <generator object <genexpr> at 0x000001C6F49BF4C0>
```

Section 26.2: Filter without function

If the function parameter is `None`, then the identity function will be used:

```
list(filter(None, [1, 0, 2, [], '', 'a']))  # discards 0, [] and ''
# Out: [1, 2, 'a']
```

Python 2.x `Version ≥ 2.0.1`

```
[i for i in [1, 0, 2, [], '', 'a'] if i] # equivalent list comprehension
```

Python 3.x `Version ≥ 3.0.0`

```
(i for i in [1, 0, 2, [], '', 'a'] if i) # equivalent generator expression
```

Section 26.3: Filter as short-circuit check

`filter` (python 3.x) and `ifilter` (python 2.x) return a generator so they can be very handy when creating a short-circuit test like `or` or `and`:

Python 2.x `Version ≥ 2.0.1`

```
# not recommended in real use but keeps the example short:
from itertools import ifilter as filter
```

Python 2.x `Version ≥ 2.6.1`

```
from future_builtins import filter
```

To find the first element that is smaller than 100:

```
car_shop = [('Toyota', 1000), ('rectangular tire', 80), ('Porsche', 5000)]
def find_something_smaller_than(name_value_tuple):
    print('Check {0}, {1}$'.format(*name_value_tuple)
    return name_value_tuple[1] < 100
next(filter(find_something_smaller_than, car_shop))
# Print: Check Toyota, 1000$
#        Check rectangular tire, 80$
# Out: ('rectangular tire', 80)
```

The `next`-function gives the next (in this case first) element of and is therefore the reason why it's short-circuit.

Section 26.4: Complementary function: filterfalse, ifilterfalse

There is a complementary function for `filter` in the `itertools`-module:

Python 2.x `Version ≥ 2.0.1`

```
# not recommended in real use but keeps the example valid for python 2.x and python 3.x
from itertools import ifilterfalse as filterfalse
```

Python 3.x `Version ≥ 3.0.0`

```
from itertools import filterfalse
```

which works exactly like the *generator* `filter` but keeps only the elements that are `False`:

```
# Usage without function (None):
list(filterfalse(None, [1, 0, 2, [], '', 'a']))  # discards 1, 2, 'a'
# Out: [0, [], '']
```

```
# Usage with function
names = ['Fred', 'Wilma', 'Barney']

def long_name(name):
    return len(name) > 5

list(filterfalse(long_name, names))
# Out: ['Fred', 'Wilma']
```

```
# Short-circuit usage with next:
car_shop = [('Toyota', 1000), ('rectangular tire', 80), ('Porsche', 5000)]
def find_something_smaller_than(name_value_tuple):
    print('Check {0}, {1}$'.format(*name_value_tuple)
    return name_value_tuple[1] < 100
next(filterfalse(find_something_smaller_than, car_shop))
# Print: Check Toyota, 1000$
```

```
# Out: ('Toyota', 1000)
```

```
# Using an equivalent generator:
car_shop = [('Toyota', 1000), ('rectangular tire', 80), ('Porsche', 5000)]
generator = (car for car in car_shop if not car[1] < 100)
next(generator)
```

Chapter 27: Heapq

Section 27.1: Largest and smallest items in a collection

To find the largest items in a collection, `heapq` module has a function called `nlargest`, we pass it two arguments, the first one is the number of items that we want to retrieve, the second one is the collection name:

```
import heapq


numbers = [1, 4, 2, 100, 20, 50, 32, 200, 150, 8]
print(heapq.nlargest(4, numbers))  # [200, 150, 100, 50]
```

Similarly, to find the smallest items in a collection, we use `nsmallest` function:

```
print(heapq.nsmallest(4, numbers))  # [1, 2, 4, 8]
```

Both `nlargest` and `nsmallest` functions take an optional argument (key parameter) for complicated data structures. The following example shows the use of `age` property to retrieve the oldest and the youngest people from `people` dictionary:

```
people = [
    {'firstname': 'John', 'lastname': 'Doe', 'age': 30},
    {'firstname': 'Jane', 'lastname': 'Doe', 'age': 25},
    {'firstname': 'Janie', 'lastname': 'Doe', 'age': 10},
    {'firstname': 'Jane', 'lastname': 'Roe', 'age': 22},
    {'firstname': 'Johnny', 'lastname': 'Doe', 'age': 12},
    {'firstname': 'John', 'lastname': 'Roe', 'age': 45}
]

oldest = heapq.nlargest(2, people, key=lambda s: s['age'])
print(oldest)
# Output: [{'firstname': 'John', 'age': 45, 'lastname': 'Roe'}, {'firstname': 'John', 'age': 30,
'lastname': 'Doe'}]

youngest = heapq.nsmallest(2, people, key=lambda s: s['age'])
print(youngest)
# Output: [{'firstname': 'Janie', 'age': 10, 'lastname': 'Doe'}, {'firstname': 'Johnny', 'age': 12,
'lastname': 'Doe'}]
```

Section 27.2: Smallest item in a collection

The most interesting property of a `heap` is that its smallest element is always the first element: `heap[0]`

```
import heapq


numbers = [10, 4, 2, 100, 20, 50, 32, 200, 150, 8]

heapq.heapify(numbers)
print(numbers)
# Output: [2, 4, 10, 100, 8, 50, 32, 200, 150, 20]

heapq.heappop(numbers)  # 2
print(numbers)
# Output: [4, 8, 10, 100, 20, 50, 32, 200, 150]
```

```
heapq.heappop(numbers)  # 4
print(numbers)
# Output:  [8, 20, 10, 100, 150, 50, 32, 200]
```

Chapter 28: Tuple

A tuple is an immutable list of values. Tuples are one of Python's simplest and most common collection types, and can be created with the comma operator (`value = 1, 2, 3`).

Section 28.1: Tuple

Syntactically, a tuple is a comma-separated list of values:

```
t = 'a', 'b', 'c', 'd', 'e'
```

Although not necessary, it is common to enclose tuples in parentheses:

```
t = ('a', 'b', 'c', 'd', 'e')
```

Create an empty tuple with parentheses:

```
t0 = ()
type(t0)            # <type 'tuple'>
```

To create a tuple with a single element, you have to include a final comma:

```
t1 = 'a',
type(t1)            # <type 'tuple'>
```

Note that a single value in parentheses is not a tuple:

```
t2 = ('a')
type(t2)            # <type 'str'>
```

To create a singleton tuple it is necessary to have a trailing comma.

```
t2 = ('a',)
type(t2)            # <type 'tuple'>
```

Note that for singleton tuples it's recommended (see PEP8 on trailing commas) to use parentheses. Also, no white space after the trailing comma (see PEP8 on whitespaces)

```
t2 = ('a',)         # PEP8-compliant
t2 = 'a',           # this notation is not recommended by PEP8
t2 = ('a', )        # this notation is not recommended by PEP8
```

Another way to create a tuple is the built-in function `tuple`.

```
t = tuple('lupins')
print(t)            # ('l', 'u', 'p', 'i', 'n', 's')
t = tuple(range(3))
print(t)            # (0, 1, 2)
```

These examples are based on material from the book **Think Python** by Allen B. Downey.

Section 28.2: Tuples are immutable

One of the main differences between `list`s and `tuple`s in Python is that tuples are immutable, that is, one cannot add or modify items once the tuple is initialized. For example:

```
>>> t = (1, 4, 9)
>>> t[0] = 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Similarly, tuples don't have `.append` and `.extend` methods as `list` does. Using `+=` is possible, but it changes the binding of the variable, and not the tuple itself:

```
>>> t = (1, 2)
>>> q = t
>>> t += (3, 4)
>>> t
(1, 2, 3, 4)
>>> q
(1, 2)
```

Be careful when placing mutable objects, such as `lists`, inside tuples. This may lead to very confusing outcomes when changing them. For example:

```
>>> t = (1, 2, 3, [1, 2, 3])
(1, 2, 3, [1, 2, 3])
>>> t[3] += [4, 5]
```

Will ***both*** raise an error and change the contents of the list within the tuple:

```
TypeError: 'tuple' object does not support item assignment
>>> t
(1, 2, 3, [1, 2, 3, 4, 5])
```

You can use the `+=` operator to "append" to a tuple - this works by creating a new tuple with the new element you "appended" and assign it to its current variable; the old tuple is not changed, but replaced!

This avoids converting to and from a list, but this is slow and is a bad practice, especially if you're going to append multiple times.

Section 28.3: Packing and Unpacking Tuples

Tuples in Python are values separated by commas. Enclosing parentheses for inputting tuples are optional, so the two assignments

```
a = 1, 2, 3   # a is the tuple (1, 2, 3)
```

and

```
a = (1, 2, 3) # a is the tuple (1, 2, 3)
```

are equivalent. The assignment `a = 1, 2, 3` is also called *packing* because it packs values together in a tuple.

Note that a one-value tuple is also a tuple. To tell Python that a variable is a tuple and not a single value you can use

a trailing comma

```
a = 1  # a is the value 1
a = 1, # a is the tuple (1,)
```

A comma is needed also if you use parentheses

```
a = (1,) # a is the tuple (1,)
a = (1)  # a is the value 1 and not a tuple
```

To unpack values from a tuple and do multiple assignments use

```
# unpacking AKA multiple assignment
x, y, z = (1, 2, 3)
# x == 1
# y == 2
# z == 3
```

The symbol _ can be used as a disposable variable name if one only needs some elements of a tuple, acting as a placeholder:

```
a = 1, 2, 3, 4
_, x, y, _ = a
# x == 2
# y == 3
```

Single element tuples:

```
x, = 1,  # x is the value 1
x  = 1,  # x is the tuple (1,)
```

In Python 3 a target variable with a * prefix can be used as a *catch-all* variable (see Unpacking Iterables):

Python 3.x `Version ≥ 3.0`

```
first, *more, last = (1, 2, 3, 4, 5)
# first == 1
# more == [2, 3, 4]
# last == 5
```

Section 28.4: Built-in Tuple Functions

Tuples support the following build-in functions

Comparison

If elements are of the same type, python performs the comparison and returns the result. If elements are different types, it checks whether they are numbers.

- If numbers, perform comparison.
- If either element is a number, then the other element is returned.
- Otherwise, types are sorted alphabetically .

If we reached the end of one of the lists, the longer list is "larger." If both list are same it returns 0.

```
tuple1 = ('a', 'b', 'c', 'd', 'e')
tuple2 = ('1','2','3')
```

```
tuple3 = ('a', 'b', 'c', 'd', 'e')

cmp(tuple1, tuple2)
Out: 1

cmp(tuple2, tuple1)
Out: -1

cmp(tuple1, tuple3)
Out: 0
```

Tuple Length

The function `len` returns the total length of the tuple

```
len(tuple1)
Out: 5
```

Max of a tuple

The function `max` returns item from the tuple with the max value

```
max(tuple1)
Out: 'e'

max(tuple2)
Out: '3'
```

Min of a tuple

The function min returns the item from the tuple with the min value

```
min(tuple1)
Out: 'a'

min(tuple2)
Out: '1'
```

Convert a list into tuple

The built-in function `tuple` converts a list into a tuple.

```
list = [1,2,3,4,5]
tuple(list)
Out: (1, 2, 3, 4, 5)
```

Tuple concatenation

Use `+` to concatenate two tuples

```
tuple1 + tuple2
Out: ('a', 'b', 'c', 'd', 'e', '1', '2', '3')
```

Section 28.5: Tuple Are Element-wise Hashable and Equatable

```
hash( (1, 2) )  # ok
hash( ([], {"hello"})  # not ok, since lists and sets are not hashabe
```

Thus a tuple can be put inside a `set` or as a key in a `dict` only if each of its elements can.

```
{ (1, 2) } #  ok
```

```
{ ([], {"hello"}) ) # not ok
```

Section 28.6: Indexing Tuples

```
x = (1, 2, 3)
x[0]  # 1
x[1]  # 2
x[2]  # 3
x[3]  # IndexError: tuple index out of range
```

Indexing with negative numbers will start from the last element as -1:

```
x[-1] # 3
x[-2] # 2
x[-3] # 1
x[-4] # IndexError: tuple index out of range
```

Indexing a range of elements

```
print(x[:-1])   # (1, 2)
print(x[-1:])   # (3,)
print(x[1:3])   # (2, 3)
```

Section 28.7: Reversing Elements

Reverse elements within a tuple

```
colors = "red", "green", "blue"
rev = colors[::-1]
# rev: ("blue", "green", "red")
colors = rev
# colors: ("blue", "green", "red")
```

Or using reversed (reversed gives an iterable which is converted to a tuple):

```
rev = tuple(reversed(colors))
# rev: ("blue", "green", "red")
colors = rev
# colors: ("blue", "green", "red")
```

Chapter 29: Basic Input and Output

Section 29.1: Using the print function

Python 3.x `Version ≥ 3.0`

In Python 3, print functionality is in the form of a function:

```
print("This string will be displayed in the output")
# This string will be displayed in the output

print("You can print \n escape characters too.")
# You can print escape characters too.
```

Python 2.x `Version ≥ 2.3`

In Python 2, print was originally a statement, as shown below.

```
print "This string will be displayed in the output"
# This string will be displayed in the output

print "You can print \n escape characters too."
# You can print escape characters too.
```

Note: using `from __future__ import print_function` in Python 2 will allow users to use the `print()` function the same as Python 3 code. This is only available in Python 2.6 and above.

Section 29.2: Input from a File

Input can also be read from files. Files can be opened using the built-in function `open`. Using a `with <command> as <name>` syntax (called a 'Context Manager') makes using `open` and getting a handle for the file super easy:

```
with open('somefile.txt', 'r') as fileobj:
    # write code here using fileobj
```

This ensures that when code execution leaves the block the file is automatically closed.

Files can be opened in different modes. In the above example the file is opened as read-only. To open an existing file for reading only use `r`. If you want to read that file as bytes use `rb`. To append data to an existing file use `a`. Use `w` to create a file or overwrite any existing files of the same name. You can use `r+` to open a file for both reading and writing. The first argument of `open()` is the filename, the second is the mode. If mode is left blank, it will default to `r`.

```
# let's create an example file:
with open('shoppinglist.txt', 'w') as fileobj:
    fileobj.write('tomato\npasta\ngarlic')

with open('shoppinglist.txt', 'r') as fileobj:
    # this method makes a list where each line
    # of the file is an element in the list
    lines = fileobj.readlines()

print(lines)
# ['tomato\n', 'pasta\n', 'garlic']

with open('shoppinglist.txt', 'r') as fileobj:
```

```
    # here we read the whole content into one string:
    content = fileobj.read()
    # get a list of lines, just like int the previous example:
    lines = content.split('\n')

print(lines)
# ['tomato', 'pasta', 'garlic']
```

If the size of the file is tiny, it is safe to read the whole file contents into memory. If the file is very large it is often better to read line-by-line or by chunks, and process the input in the same loop. To do that:

```
with open('shoppinglist.txt', 'r') as fileobj:
    # this method reads line by line:
    lines = []
    for line in fileobj:
        lines.append(line.strip())
```

When reading files, be aware of the operating system-specific line-break characters. Although `for line in fileobj` automatically strips them off, it is always safe to call `strip()` on the lines read, as it is shown above.

Opened files (`fileobj` in the above examples) always point to a specific location in the file. When they are first opened the file handle points to the very beginning of the file, which is the position `0`. The file handle can display its current position with `tell`:

```
fileobj = open('shoppinglist.txt', 'r')
pos = fileobj.tell()
print('We are at %u.' % pos) # We are at 0.
```

Upon reading all the content, the file handler's position will be pointed at the end of the file:

```
content = fileobj.read()
end = fileobj.tell()
print('This file was %u characters long.' % end)
# This file was 22 characters long.
fileobj.close()
```

The file handler position can be set to whatever is needed:

```
fileobj = open('shoppinglist.txt', 'r')
fileobj.seek(7)
pos = fileobj.tell()
print('We are at character #%u.' % pos)
```

You can also read any length from the file content during a given call. To do this pass an argument for `read()`. When `read()` is called with no argument it will read until the end of the file. If you pass an argument it will read that number of bytes or characters, depending on the mode (`rb` and `r` respectively):

```
# reads the next 4 characters
# starting at the current position
next4 = fileobj.read(4)
# what we got?
print(next4) # 'cucu'
# where we are now?
pos = fileobj.tell()
print('We are at %u.' % pos) # We are at 11, as we was at 7, and read 4 chars.

fileobj.close()
```

To demonstrate the difference between characters and bytes:

```
with open('shoppinglist.txt', 'r') as fileobj:
    print(type(fileobj.read())) # <class 'str'>

with open('shoppinglist.txt', 'rb') as fileobj:
    print(type(fileobj.read())) # <class 'bytes'>
```

Section 29.3: Read from stdin

Python programs can read from unix pipelines. Here is a simple example how to read from `stdin`:

```
import sys

for line in sys.stdin:
    print(line)
```

Be aware that `sys.stdin` is a stream. It means that the for-loop will only terminate when the stream has ended.

You can now pipe the output of another program into your python program as follows:

```
$ cat myfile | python myprogram.py
```

In this example `cat myfile` can be any unix command that outputs to `stdout`.

Alternatively, using the fileinput module can come in handy:

```
import fileinput
for line in fileinput.input():
    process(line)
```

Section 29.4: Using input() and raw_input()

Python 2.x `Version ≥ 2.3`

`raw_input` will wait for the user to enter text and then return the result as a string.

```
foo = raw_input("Put a message here that asks the user for input")
```

In the above example `foo` will store whatever input the user provides.

Python 3.x `Version ≥ 3.0`

`input` will wait for the user to enter text and then return the result as a string.

```
foo = input("Put a message here that asks the user for input")
```

In the above example `foo` will store whatever input the user provides.

Section 29.5: Function to prompt user for a number

```
def input_number(msg, err_msg=None):
    while True:
        try:
```

```
            return float(raw_input(msg))
        except ValueError:
            if err_msg is not None:
                print(err_msg)


def input_number(msg, err_msg=None):
    while True:
        try:
            return float(input(msg))
        except ValueError:
            if err_msg is not None:
                print(err_msg)
```

And to use it:

```
user_number = input_number("input a number: ", "that's not a number!")
```

Or, if you do not want an "error message":

```
user_number = input_number("input a number: ")
```

Section 29.6: Printing a string without a newline at the end

Python 2.x `Version ≥ 2.3`

In Python 2.x, to continue a line with `print`, end the `print` statement with a comma. It will automatically add a space.

```
print "Hello,",
print "World!"
# Hello, World!
```

Python 3.x `Version ≥ 3.0`

In Python 3.x, the `print` function has an optional `end` parameter that is what it prints at the end of the given string. By default it's a newline character, so equivalent to this:

```
print("Hello, ", end="\n")
print("World!")
# Hello,
# World!
```

But you could pass in other strings

```
print("Hello, ", end="")
print("World!")
# Hello, World!

print("Hello, ", end="<br>")
print("World!")
# Hello, <br>World!

print("Hello, ", end="BREAK")
print("World!")
# Hello, BREAKWorld!
```

If you want more control over the output, you can use `sys.stdout.write`:

```python
import sys

sys.stdout.write("Hello, ")
sys.stdout.write("World!")
# Hello, World!
```

Chapter 30: Files & Folders I/O

Parameter	Details
filename	the path to your file or, if the file is in the working directory, the filename of your file
access_mode	a string value that determines how the file is opened
buffering	an integer value used for optional line buffering

When it comes to storing, reading, or communicating data, working with the files of an operating system is both necessary and easy with Python. Unlike other languages where file input and output requires complex reading and writing objects, Python simplifies the process only needing commands to open, read/write and close the file. This topic explains how Python can interface with files on the operating system.

Section 30.1: File modes

There are different modes you can open a file with, specified by the `mode` parameter. These include:

- `'r'` - reading mode. The default. It allows you only to read the file, not to modify it. When using this mode the file must exist.
- `'w'` - writing mode. It will create a new file if it does not exist, otherwise will erase the file and allow you to write to it.
- `'a'` - append mode. It will write data to the end of the file. It does not erase the file, and the file must exist for this mode.
- `'rb'` - reading mode in binary. This is similar to `r` except that the reading is forced in binary mode. This is also a default choice.
- `'r+'` - reading mode plus writing mode at the same time. This allows you to read and write into files at the same time without having to use `r` and `w`.
- `'rb+'` - reading and writing mode in binary. The same as `r+` except the data is in binary
- `'wb'` - writing mode in binary. The same as `w` except the data is in binary.
- `'w+'` - writing and reading mode. The exact same as `r+` but if the file does not exist, a new one is made. Otherwise, the file is overwritten.
- `'wb+'` - writing and reading mode in binary mode. The same as `w+` but the data is in binary.
- `'ab'` - appending in binary mode. Similar to `a` except that the data is in binary.
- `'a+'` - appending and reading mode. Similar to `w+` as it will create a new file if the file does not exist. Otherwise, the file pointer is at the end of the file if it exists.
- `'ab+'` - appending and reading mode in binary. The same as `a+` except that the data is in binary.

```
with open(filename, 'r') as f:
    f.read()
with open(filename, 'w') as f:
    f.write(filedata)
with open(filename, 'a') as f:
    f.write('\\n' + newdata)
```

	r	r+	w	w+	a	a+
Read	✔	✔	✘	✔	✘	✔
Write	✘	✔	✔	✔	✔	✔

Creates file	✘	✘	✔	✔	✔	✔
Erases file	✘	✘	✔	✔	✘	✘
Initial position	Start	Start	Start	Start	End	End

Python 3 added a new mode for `exclusive creation` so that you will not accidentally truncate or overwrite and existing file.

- `'x'` - open for exclusive creation, will raise `FileExistsError` if the file already exists
- `'xb'` - open for exclusive creation writing mode in binary. The same as `x` except the data is in binary.
- `'x+'` - reading and writing mode. Similar to `w+` as it will create a new file if the file does not exist. Otherwise, will raise `FileExistsError`.
- `'xb+'` - writing and reading mode. The exact same as `x+` but the data is binary

	x	x+
Read	✘	✔
Write	✔	✔
Creates file	✔	✔
Erases file	✘	✘
Initial position	Start	Start

Allow one to write your file open code in a more pythonic manner:

Python 3.x `Version ≥ 3.3`

```
try:
    with open("fname", "r") as fout:
        # Work with your open file
except FileExistsError:
    # Your error handling goes here
```

In Python 2 you would have done something like

Python 2.x `Version ≥ 2.0`

```
import os.path
if os.path.isfile(fname):
    with open("fname", "w") as fout:
        # Work with your open file
else:
    # Your error handling goes here
```

Section 30.2: Reading a file line-by-line

The simplest way to iterate over a file line-by-line:

```
with open('myfile.txt', 'r') as fp:
    for line in fp:
        print(line)
```

`readline()` allows for more granular control over line-by-line iteration. The example below is equivalent to the one above:

```
with open('myfile.txt', 'r') as fp:
    while True:
        cur_line = fp.readline()
        # If the result is an empty string
        if cur_line == '':
            # We have reached the end of the file
```

```
            break
        print(cur_line)
```

Using the for loop iterator and readline() together is considered bad practice.

More commonly, the `readlines()` method is used to store an iterable collection of the file's lines:

```
with open("myfile.txt", "r") as fp:
    lines = fp.readlines()
for i in range(len(lines)):
    print("Line " + str(i) + ": " + line)
```

This would print the following:

> Line 0: hello
>
> Line 1: world

Section 30.3: Iterate files (recursively)

To iterate all files, including in sub directories, use os.walk:

```
import os
for root, folders, files in os.walk(root_dir):
    for filename in files:
        print root, filename
```

root_dir can be "." to start from current directory, or any other path to start from.

Python 3.x `Version ≥ 3.5`

If you also wish to get information about the file, you may use the more efficient method os.scandir like so:

```
for entry in os.scandir(path):
   if not entry.name.startswith('.') and entry.is_file():
       print(entry.name)
```

Section 30.4: Getting the full contents of a file

The preferred method of file i/o is to use the `with` keyword. This will ensure the file handle is closed once the reading or writing has been completed.

```
with open('myfile.txt') as in_file:
    content = in_file.read()

print(content)
```

or, to handle closing the file manually, you can forgo `with` and simply call `close` yourself:

```
in_file = open('myfile.txt', 'r')
content = in_file.read()
print(content)
in_file.close()
```

Keep in mind that without using a `with` statement, you might accidentally keep the file open in case an unexpected exception arises like so:

```
in_file = open('myfile.txt', 'r')
raise Exception("oops")
in_file.close()  # This will never be called
```

Section 30.5: Writing to a file

```
with open('myfile.txt', 'w') as f:
    f.write("Line 1")
    f.write("Line 2")
    f.write("Line 3")
    f.write("Line 4")
```

If you open `myfile.txt`, you will see that its contents are:

> Line 1Line 2Line 3Line 4

Python doesn't automatically add line breaks, you need to do that manually:

```
with open('myfile.txt', 'w') as f:
    f.write("Line 1\n")
    f.write("Line 2\n")
    f.write("Line 3\n")
    f.write("Line 4\n")
```

> Line 1
> Line 2
> Line 3
> Line 4

Do not use `os.linesep` as a line terminator when writing files opened in text mode (the default); use `\n` instead.

If you want to specify an encoding, you simply add the `encoding` parameter to the `open` function:

```
with open('my_file.txt', 'w', encoding='utf-8') as f:
    f.write('utf-8 text')
```

It is also possible to use the print statement to write to a file. The mechanics are different in Python 2 vs Python 3, but the concept is the same in that you can take the output that would have gone to the screen and send it to a file instead.

Python 3.x `Version ≥ 3.0`

```
with open('fred.txt', 'w') as outfile:
    s = "I'm Not Dead Yet!"
    print(s) # writes to stdout
    print(s, file = outfile) # writes to outfile

    #Note: it is possible to specify the file parameter AND write to the screen
    #by making sure file ends up with a None value either directly or via a variable
    myfile = None
    print(s, file = myfile) # writes to stdout
```

```
    print(s, file = None)   # writes to stdout
```

In Python 2 you would have done something like

Python 2.x Version ≥ 2.0

```
outfile = open('fred.txt', 'w')
s = "I'm Not Dead Yet!"
print s   # writes to stdout
print >> outfile, s   # writes to outfile
```

Unlike using the write function, the print function does automatically add line breaks.

Section 30.6: Check whether a file or path exists

Employ the EAFP coding style and `try` to open it.

```
import errno

try:
    with open(path) as f:
        # File exists
except IOError as e:
    # Raise the exception if it is not ENOENT (No such file or directory)
    if e.errno != errno.ENOENT:
        raise
    # No such file or directory
```

This will also avoid race-conditions if another process deleted the file between the check and when it is used. This race condition could happen in the following cases:

- Using the `os` module:

```
import os
os.path.isfile('/path/to/some/file.txt')
```

Python 3.x Version ≥ 3.4

- Using `pathlib`:

```
import pathlib
path = pathlib.Path('/path/to/some/file.txt')
if path.is_file():
    ...
```

To check whether a given path exists or not, you can follow the above EAFP procedure, or explicitly check the path:

```
import os
path = "/home/myFiles/directory1"

if os.path.exists(path):
    ## Do stuff
```

Section 30.7: Random File Access Using mmap

Using the `mmap` module allows the user to randomly access locations in a file by mapping the file into memory. This is an alternative to using normal file operations.

```
import mmap

with open('filename.ext', 'r') as fd:
    # 0: map the whole file
    mm = mmap.mmap(fd.fileno(), 0)

    # print characters at indices 5 through 10
    print mm[5:10]

    # print the line starting from mm's current position
    print mm.readline()

    # write a character to the 5th index
    mm[5] = 'a'

    # return mm's position to the beginning of the file
    mm.seek(0)

    # close the mmap object
    mm.close()
```

Section 30.8: Replacing text in a file

```
import fileinput

replacements = {'Search1': 'Replace1',
                'Search2': 'Replace2'}

for line in fileinput.input('filename.txt', inplace=True):
    for search_for in replacements:
        replace_with = replacements[search_for]
        line = line.replace(search_for, replace_with)
    print(line, end='')
```

Section 30.9: Checking if a file is empty

```
>>> import os
>>> os.stat(path_to_file).st_size == 0
```

or

```
>>> import os
>>> os.path.getsize(path_to_file) > 0
```

However, both will throw an exception if the file does not exist. To avoid having to catch such an error, do this:

```
import os
def is_empty_file(fpath):
    return os.path.isfile(fpath) and os.path.getsize(fpath) > 0
```

which will return a `bool` value.

Section 30.10: Read a file between a range of lines

So let's suppose you want to iterate only between some specific lines of a file

You can make use of `itertools` for that

```
import itertools

with open('myfile.txt', 'r') as f:
    for line in itertools.islice(f, 12, 30):
        # do something here
```

This will read through the lines 13 to 20 as in python indexing starts from 0. So line number 1 is indexed as 0

As can also read some extra lines by making use of the `next()` keyword here.

And when you are using the file object as an iterable, please don't use the `readline()` statement here as the two techniques of traversing a file are not to be mixed together

Section 30.11: Copy a directory tree

```
import shutil
source='//192.168.1.2/Daily Reports'
destination='D:\\Reports\\Today'
shutil.copytree(source, destination)
```

The destination directory **must not exist** already.

Section 30.12: Copying contents of one file to a different file

```
with open(input_file, 'r') as in_file, open(output_file, 'w') as out_file:
    for line in in_file:
        out_file.write(line)
```

- Using the `shutil` module:

```
import shutil
shutil.copyfile(src, dst)
```

Chapter 31: os.path

This module implements some useful functions on pathnames. The path parameters can be passed as either strings, or bytes. Applications are encouraged to represent file names as (Unicode) character strings.

Section 31.1: Join Paths

To join two or more path components together, firstly import os module of python and then use following:

```
import os
os.path.join('a', 'b', 'c')
```

The advantage of using os.path is that it allows code to remain compatible over all operating systems, as this uses the separator appropriate for the platform it's running on.

For example, the result of this command on Windows will be:

```
>>> os.path.join('a', 'b', 'c')
'a\b\c'
```

In an Unix OS:

```
>>> os.path.join('a', 'b', 'c')
'a/b/c'
```

Section 31.2: Path Component Manipulation

To split one component off of the path:

```
>>> p = os.path.join(os.getcwd(), 'foo.txt')
>>> p
'/Users/csaftoiu/tmp/foo.txt'
>>> os.path.dirname(p)
'/Users/csaftoiu/tmp'
>>> os.path.basename(p)
'foo.txt'
>>> os.path.split(os.getcwd())
('/Users/csaftoiu/tmp', 'foo.txt')
>>> os.path.splitext(os.path.basename(p))
('foo', '.txt')
```

Section 31.3: Get the parent directory

```
os.path.abspath(os.path.join(PATH_TO_GET_THE_PARENT, os.pardir))
```

Section 31.4: If the given path exists

to check if the given path exists

```
path = '/home/john/temp'
os.path.exists(path)
#this returns false if path doesn't exist or if the path is a broken symbolic link
```

Section 31.5: check if the given path is a directory, file, symbolic link, mount point etc

to check if the given path is a directory

```
dirname = '/home/john/python'
os.path.isdir(dirname)
```

to check if the given path is a file

```
filename = dirname + 'main.py'
os.path.isfile(filename)
```

to check if the given path is symbolic link

```
symlink = dirname + 'some_sym_link'
os.path.islink(symlink)
```

to check if the given path is a mount point

```
mount_path = '/home'
os.path.ismount(mount_path)
```

Section 31.6: Absolute Path from Relative Path

Use `os.path.abspath`:

```
>>> os.getcwd()
'/Users/csaftoiu/tmp'
>>> os.path.abspath('foo')
'/Users/csaftoiu/tmp/foo'
>>> os.path.abspath('../foo')
'/Users/csaftoiu/foo'
>>> os.path.abspath('/foo')
'/foo'
```

Chapter 32: Iterables and Iterators

Section 32.1: Iterator vs Iterable vs Generator

An **iterable** is an object that can return an **iterator**. Any object with state that has an `__iter__` method and returns an iterator is an iterable. It may also be an object *without* state that implements a `__getitem__` method. - The method can take indices (starting from zero) and raise an `IndexError` when the indices are no longer valid.

Python's `str` class is an example of a `__getitem__` iterable.

An **Iterator** is an object that produces the next value in a sequence when you call `next(*object*)` on some object. Moreover, any object with a `__next__` method is an iterator. An iterator raises `StopIteration` after exhausting the iterator and *cannot* be re-used at this point.

Iterable classes:

Iterable classes define an `__iter__` and a `__next__` method. Example of an iterable class :

```
class MyIterable:

    def __iter__(self):

         return self

    def __next__(self):
         #code

#Classic iterable object in older versions of python, __getitem__ is still supported...
class MySequence:

    def __getitem__(self, index):

         if (condition):
             raise IndexError
         return (item)

 #Can produce a plain `iterator` instance by using iter(MySequence())
```

Trying to instantiate the abstract class from the `collections` module to better see this.

Example:

Python 2.x `Version ≥ 2.3`

```
import collections
>>> collections.Iterator()
>>> TypeError: Cant instantiate abstract class Iterator with abstract methods next
```

Python 3.x `Version ≥ 3.0`

```
>>> TypeError: Cant instantiate abstract class Iterator with abstract methods __next__
```

Handle Python 3 compatibility for iterable classes in Python 2 by doing the following:

Python 2.x `Version ≥ 2.3`

```
class MyIterable(object): #or collections.Iterator, which I'd recommend....

     ....
```

```
    def __iter__(self):

        return self

    def next(self): #code

    __next__ = next
```

Both of these are now iterators and can be looped through:

```
ex1 = MyIterableClass()
ex2 = MySequence()

for (item) in (ex1): #code
for (item) in (ex2): #code
```

Generators are simple ways to create iterators. A generator *is* an iterator and an iterator is an iterable.

Section 32.2: Extract values one by one

Start with `iter()` built-in to get **iterator** over iterable and use `next()` to get elements one by one until `StopIteration` is raised signifying the end:

```
s = {1, 2}   # or list or generator or even iterator
i = iter(s)  # get iterator
a = next(i)  # a = 1
b = next(i)  # b = 2
c = next(i)  # raises StopIteration
```

Section 32.3: Iterating over entire iterable

```
s = {1, 2, 3}

# get every element in s
for a in s:
    print a  # prints 1, then 2, then 3

# copy into list
l1 = list(s)  # l1 = [1, 2, 3]

# use list comprehension
l2 = [a * 2 for a in s if a > 2]  # l2 = [6]
```

Section 32.4: Verify only one element in iterable

Use unpacking to extract the first element and ensure it's the only one:

```
a, = iterable

def foo():
    yield 1

a, = foo()  # a = 1

nums = [1, 2, 3]
a, = nums  # ValueError: too many values to unpack
```

Section 32.5: What can be iterable

Iterable can be anything for which items are received *one by one, forward only*. Built-in Python collections are iterable:

```
[1, 2, 3]     # list, iterate over items
(1, 2, 3)     # tuple
{1, 2, 3}     # set
{1: 2, 3: 4}  # dict, iterate over keys
```

Generators return iterables:

```
def foo():  # foo isn't iterable yet...
    yield 1

res = foo()  # ...but res already is
```

Section 32.6: Iterator isn't reentrant!

```
def gen():
    yield 1

iterable = gen()
for a in iterable:
    print a

# What was the first item of iterable? No way to get it now.
# Only to get a new iterator
gen()
```

Chapter 33: Functions

Parameter	Details
arg1, ..., *argN*	Regular arguments
**args*	Unnamed positional arguments
kw1, ..., *kwN*	Keyword-only arguments
***kwargs*	The rest of keyword arguments

Functions in Python provide organized, reusable and modular code to perform a set of specific actions. Functions simplify the coding process, prevent redundant logic, and make the code easier to follow. This topic describes the declaration and utilization of functions in Python.

Python has many *built-in functions* like `print()`, `input()`, `len()`. Besides built-ins you can also create your own functions to do more specific jobs—these are called *user-defined functions*.

Section 33.1: Defining and calling simple functions

Using the `def` statement is the most common way to define a function in python. This statement is a so called *single clause compound statement* with the following syntax:

```
def function_name(parameters):
    statement(s)
```

`function_name` is known as the *identifier* of the function. Since a function definition is an executable statement its execution *binds* the function name to the function object which can be called later on using the identifier.

`parameters` is an optional list of identifiers that get bound to the values supplied as arguments when the function is called. A function may have an arbitrary number of arguments which are separated by commas.

`statement(s)` – also known as the *function body* – are a nonempty sequence of statements executed each time the function is called. This means a function body cannot be empty, just like any *indented block*.

Here's an example of a simple function definition which purpose is to print `Hello` each time it's called:

```
def greet():
    print("Hello")
```

Now let's call the defined `greet()` function:

```
greet()
# Out: Hello
```

That's another example of a function definition which takes one single argument and displays the passed in value each time the function is called:

```
def greet_two(greeting):
    print(greeting)
```

After that the `greet_two()` function must be called with an argument:

```
greet_two("Howdy")
# Out: Howdy
```

Also you can give a default value to that function argument:

```
def greet_two(greeting="Howdy"):
    print(greeting)
```

Now you can call the function without giving a value:

```
greet_two()
# Out: Howdy
```

You'll notice that unlike many other languages, you do not need to explicitly declare a return type of the function. Python functions can return values of any type via the `return` keyword. One function can return any number of different types!

```
def many_types(x):
    if x < 0:
        return "Hello!"
    else:
        return 0

print(many_types(1))
print(many_types(-1))

# Output:
0
Hello!
```

As long as this is handled correctly by the caller, this is perfectly valid Python code.

A function that reaches the end of execution without a return statement will always return `None`:

```
def do_nothing():
    pass

print(do_nothing())
# Out: None
```

As mentioned previously a function definition must have a function body, a nonempty sequence of statements. Therefore the `pass` statement is used as function body, which is a null operation – when it is executed, nothing happens. It does what it means, it skips. It is useful as a placeholder when a statement is required syntactically, but no code needs to be executed.

Section 33.2: Defining a function with an arbitrary number of arguments

Arbitrary number of positional arguments:

Defining a function capable of taking an arbitrary number of arguments can be done by prefixing one of the arguments with a `*`

```
def func(*args):
    # args will be a tuple containing all values that are passed in
    for i in args:
        print(i)

func(1, 2, 3)  # Calling it with 3 arguments
# Out: 1
#      2
#      3
```

```
list_of_arg_values = [1, 2, 3]
func(*list_of_arg_values)  # Calling it with list of values, * expands the list
# Out: 1
#      2
#      3

func()  # Calling it without arguments
# No Output
```

You **can't** provide a default for `args`, for example `func(*args=[1, 2, 3])` will raise a syntax error (won't even compile).

You **can't** provide these by name when calling the function, for example `func(*args=[1, 2, 3])` will raise a `TypeError`.

But if you already have your arguments in an array (or any other `Iterable`), you **can** invoke your function like this: `func(*my_stuff)`.

These arguments (`*args`) can be accessed by index, for example `args[0]` will return the first argument

Arbitrary number of keyword arguments

You can take an arbitrary number of arguments with a name by defining an argument in the definition with **two** `*` in front of it:

```
def func(**kwargs):
    # kwargs will be a dictionary containing the names as keys and the values as values
    for name, value in kwargs.items():
        print(name, value)

func(value1=1, value2=2, value3=3)   # Calling it with 3 arguments
# Out: value1 1
#      value2 2
#      value3 3

func()                               # Calling it without arguments
# No Out put

my_dict = {'foo': 1, 'bar': 2}
func(**my_dict)                      # Calling it with a dictionary
# Out: foo 1
#      bar 2
```

You **can't** provide these **without** names, for example `func(1, 2, 3)` will raise a `TypeError`.

`kwargs` is a plain native python dictionary. For example, `args['value1']` will give the value for argument `value1`. Be sure to check beforehand that there is such an argument or a `KeyError` will be raised.

Warning

You can mix these with other optional and required arguments but the order inside the definition matters.

The **positional/keyword** arguments come first. (Required arguments).
Then comes the **arbitrary** `*arg` arguments. (Optional).
Then **keyword-only** arguments come next. (Required).
Finally the **arbitrary keyword** `**kwargs` come. (Optional).

```
#       |-positional-|-optional-|---keyword-only--|-optional-|
def func(arg1, arg2=10 , *args, kwarg1, kwarg2=2, **kwargs):
    pass
```

- `arg1` must be given, otherwise a `TypeError` is raised. It can be given as positional (`func(10)`) or keyword argument (`func(arg1=10)`).
- `kwarg1` must also be given, but it can only be provided as keyword-argument: `func(kwarg1=10)`.
- `arg2` and `kwarg2` are optional. If the value is to be changed the same rules as for `arg1` (either positional or keyword) and `kwarg1` (only keyword) apply.
- `*args` catches additional positional parameters. But note, that `arg1` and `arg2` must be provided as positional arguments to pass arguments to `*args`: `func(1, 1, 1, 1)`.
- `**kwargs` catches all additional keyword parameters. In this case any parameter that is not `arg1`, `arg2`, `kwarg1` or `kwarg2`. For example: `func(kwarg3=10)`.
- In Python 3, you can use `*` alone to indicate that all subsequent arguments must be specified as keywords. For instance the `math.isclose` function in Python 3.5 and higher is defined using `def math.isclose (a, b, *, rel_tol=1e-09, abs_tol=0.0)`, which means the first two arguments can be supplied positionally but the optional third and fourth parameters can only be supplied as keyword arguments.

Python 2.x doesn't support keyword-only parameters. This behavior can be emulated with `kwargs`:

```
def func(arg1, arg2=10, **kwargs):
    try:
        kwarg1 = kwargs.pop("kwarg1")
    except KeyError:
        raise TypeError("missing required keyword-only argument: 'kwarg1'")

    kwarg2 = kwargs.pop("kwarg2", 2)
    # function body ...
```

Note on Naming

The convention of naming optional positional arguments `args` and optional keyword arguments `kwargs` is just a convention you **can** use any names you like **but** it is useful to follow the convention so that others know what you are doing, *or even yourself later* so please do.

Note on Uniqueness

Any function can be defined with **none or one** `*args` and **none or one** `**kwargs` but not with more than one of each. Also `*args` **must** be the last positional argument and `**kwargs` must be the last parameter. Attempting to use more than one of either **will** result in a Syntax Error exception.

Note on Nesting Functions with Optional Arguments

It is possible to nest such functions and the usual convention is to remove the items that the code has already handled **but** if you are passing down the parameters you need to pass optional positional args with a `*` prefix and optional keyword args with a `**` prefix, otherwise args with be passed as a list or tuple and kwargs as a single dictionary. e.g.:

```
def fn(**kwargs):
    print(kwargs)
    f1(**kwargs)

def f1(**kwargs):
    print(len(kwargs))

fn(a=1, b=2)
```

```
# Out:
# {'a': 1, 'b': 2}
# 2
```

Section 33.3: Lambda (Inline/Anonymous) Functions

The `lambda` keyword creates an inline function that contains a single expression. The value of this expression is what the function returns when invoked.

Consider the function:

```
def greeting():
    return "Hello"
```

which, when called as:

```
print(greeting())
```

prints:

```
Hello
```

This can be written as a lambda function as follows:

```
greet_me = lambda: "Hello"
```

> See note at the bottom of this section regarding the assignment of lambdas to variables. Generally, don't do it.

This creates an inline function with the name `greet_me` that returns `Hello`. Note that you don't write `return` when creating a function with lambda. The value after `:` is automatically returned.

Once assigned to a variable, it can be used just like a regular function:

```
print(greet_me())
```

prints:

```
Hello
```

`lambda`s can take arguments, too:

```
strip_and_upper_case = lambda s: s.strip().upper()

strip_and_upper_case("  Hello   ")
```

returns the string:

```
HELLO
```

They can also take arbitrary number of arguments / keyword arguments, like normal functions.

```
greeting = lambda x, *args, **kwargs: print(x, args, kwargs)
```

```
greeting('hello', 'world', world='world')
```

prints:

```
hello ('world',) {'world': 'world'}
```

`lambda`s are commonly used for short functions that are convenient to define at the point where they are called (typically with `sorted`, `filter` and `map`).

For example, this line sorts a list of strings ignoring their case and ignoring whitespace at the beginning and at the end:

```
sorted( [" foo ", "    bAR", "BaZ    "], key=lambda s: s.strip().upper())
# Out:
# ['    bAR', 'BaZ    ', ' foo ']
```

Sort list just ignoring whitespaces:

```
sorted( [" foo ", "    bAR", "BaZ    "], key=lambda s: s.strip())
# Out:
# ['BaZ    ', '    bAR', ' foo ']
```

Examples with `map`:

```
sorted( map( lambda s: s.strip().upper(), [" foo ", "    bAR", "BaZ    "]))
# Out:
# ['BAR', 'BAZ', 'FOO']

sorted( map( lambda s: s.strip(), [" foo ", "    bAR", "BaZ    "]))
# Out:
# ['BaZ', 'bAR', 'foo']
```

Examples with numerical lists:

```
my_list = [3, -4, -2, 5, 1, 7]
sorted( my_list, key=lambda x: abs(x))
# Out:
# [1, -2, 3, -4, 5, 7]

list( filter( lambda x: x>0, my_list))
# Out:
# [3, 5, 1, 7]

list( map( lambda x: abs(x), my_list))
# Out:
[3, 4, 2, 5, 1, 7]
```

One can call other functions (with/without arguments) from inside a lambda function.

```
def foo(msg):
    print(msg)

greet = lambda x = "hello world": foo(x)
greet()
```

prints:

```
hello world
```

This is useful because `lambda` may contain only one expression and by using a subsidiary function one can run multiple statements.

NOTE

Bear in mind that PEP-8 (the official Python style guide) does not recommend assigning lambdas to variables (as we did in the first two examples):

> Always use a def statement instead of an assignment statement that binds a lambda expression directly to an identifier.
>
> Yes:
>
> ```
> def f(x): return 2*x
> ```
>
> No:
>
> ```
> f = lambda x: 2*x
> ```
>
> The first form means that the name of the resulting function object is specifically `f` instead of the generic `<lambda>`. This is more useful for tracebacks and string representations in general. The use of the assignment statement eliminates the sole benefit a lambda expression can offer over an explicit `def` statement (i.e. that it can be embedded inside a larger expression).

Section 33.4: Defining a function with optional arguments

Optional arguments can be defined by assigning (using `=`) a default value to the argument-name:

```
def make(action='nothing'):
    return action
```

Calling this function is possible in 3 different ways:

```
make("fun")
# Out: fun

make(action="sleep")
# Out: sleep

# The argument is optional so the function will use the default value if the argument is
# not passed in.
make()
# Out: nothing
```

> **Warning**
>
> Mutable types (`list`, `dict`, `set`, etc.) should be treated with care when given as **default** attribute. Any mutation of the default argument will change it permanently. See Defining a function with optional mutable arguments.

Section 33.5: Defining a function with optional mutable arguments

There is a problem when using **optional arguments** with a **mutable default type** (described in Defining a function with optional arguments), which can potentially lead to unexpected behaviour.

Explanation

This problem arises because a function's default arguments are initialised **once**, at the point when the function is *defined*, and **not** (like many other languages) when the function is *called*. The default values are stored inside the function object's `__defaults__` member variable.

```
def f(a, b=42, c=[]):
    pass

print(f.__defaults__)
# Out: (42, [])
```

For **immutable** types (see Argument passing and mutability) this is not a problem because there is no way to mutate the variable; it can only ever be reassigned, leaving the original value unchanged. Hence, subsequent are guaranteed to have the same default value. However, for a **mutable** type, the original value can mutate, by making calls to its various member functions. Therefore, successive calls to the function are not guaranteed to have the initial default value.

```
def append(elem, to=[]):
    to.append(elem)      # This call to append() mutates the default variable "to"
    return to

append(1)
# Out: [1]

append(2)  # Appends it to the internally stored list
# Out: [1, 2]

append(3, [])  # Using a new created list gives the expected result
# Out: [3]

# Calling it again without argument will append to the internally stored list again
append(4)
# Out: [1, 2, 4]
```

> **Note:** Some IDEs like PyCharm will issue a warning when a mutable type is specified as a default attribute.

Solution

If you want to ensure that the default argument is always the one you specify in the function definition, then the solution is to **always** use an immutable type as your default argument.

A common idiom to achieve this when a mutable type is needed as the default, is to use `None` (immutable) as the default argument and then assign the actual default value to the argument variable if it is equal to `None`.

```
def append(elem, to=None):
    if to is None:
        to = []
```

```
    to.append(elem)
    return to
```

Section 33.6: Argument passing and mutability

First, some terminology:

- **argument (*actual* parameter):** the actual variable being passed to a function;
- **parameter (*formal* parameter):** the receiving variable that is used in a function.

In Python, arguments are passed by *assignment* (as opposed to other languages, where arguments can be passed by value/reference/pointer).

- Mutating a parameter will mutate the argument (if the argument's type is mutable).

```
def foo(x):         # here x is the parameter
    x[0] = 9        # This mutates the list labelled by both x and y
    print(x)

y = [4, 5, 6]
foo(y)              # call foo with y as argument
# Out: [9, 5, 6]   # list labelled by x has been mutated
print(y)
# Out: [9, 5, 6]   # list labelled by y has been mutated too
```

- Reassigning the parameter won't reassign the argument.

```
def foo(x):         # here x is the parameter, when we call foo(y) we assign y to x
    x[0] = 9        # This mutates the list labelled by both x and y
    x = [1, 2, 3]   # x is now labeling a different list (y is unaffected)
    x[2] = 8        # This mutates x's list, not y's list

y = [4, 5, 6]       # y is the argument, x is the parameter
foo(y)              # Pretend that we wrote "x = y", then go to line 1
y
# Out: [9, 5, 6]
```

In Python, we don't really assign values to variables, instead we *bind* (i.e. assign, attach) variables (considered as *names*) to objects.

- **Immutable:** Integers, strings, tuples, and so on. All operations make copies.
- **Mutable:** Lists, dictionaries, sets, and so on. Operations may or may not mutate.

```
x = [3, 1, 9]
y = x
x.append(5)     # Mutates the list labelled by x and y, both x and y are bound to [3, 1, 9]
x.sort()        # Mutates the list labelled by x and y (in-place sorting)
x = x + [4]     # Does not mutate the list (makes a copy for x only, not y)
z = x           # z is x ([1, 3, 9, 4])
x += [6]        # Mutates the list labelled by both x and z (uses the extend function).
x = sorted(x)   # Does not mutate the list (makes a copy for x only).
x
# Out: [1, 3, 4, 5, 6, 9]
y
# Out: [1, 3, 5, 9]
z
```

```
# Out: [1, 3, 5, 9, 4, 6]
```

Section 33.7: Returning values from functions

Functions can `return` a value that you can use directly:

```
def give_me_five():
    return 5

print(give_me_five())  # Print the returned value
# Out: 5
```

or save the value for later use:

```
num = give_me_five()
print(num)             # Print the saved returned value
# Out: 5
```

or use the value for any operations:

```
print(give_me_five() + 10)
# Out: 15
```

If `return` is encountered in the function the function will be exited immediately and subsequent operations will not be evaluated:

```
def give_me_another_five():
    return 5
    print('This statement will not be printed. Ever.')

print(give_me_another_five())
# Out: 5
```

You can also `return` multiple values (in the form of a tuple):

```
def give_me_two_fives():
    return 5, 5  # Returns two 5

first, second = give_me_two_fives()
print(first)
# Out: 5
print(second)
# Out: 5
```

A function with *no* `return` statement implicitly returns `None`. Similarly a function with a `return` statement, but no return value or variable returns `None`.

Section 33.8: Closure

Closures in Python are created by function calls. Here, the call to `makeInc` creates a binding for `x` that is referenced inside the function `inc`. Each call to `makeInc` creates a new instance of this function, but each instance has a link to a different binding of `x`.

```
def makeInc(x):
  def inc(y):
     # x is "attached" in the definition of inc
```

```
    return y + x

  return inc

incOne = makeInc(1)
incFive = makeInc(5)

incOne(5) # returns 6
incFive(5) # returns 10
```

Notice that while in a regular closure the enclosed function fully inherits all variables from its enclosing environment, in this construct the enclosed function has only read access to the inherited variables but cannot make assignments to them

```
def makeInc(x):
  def inc(y):
    # incrementing x is not allowed
    x += y
    return x

  return inc

incOne = makeInc(1)
incOne(5) # UnboundLocalError: local variable 'x' referenced before assignment
```

Python 3 offers the `nonlocal` statement (Nonlocal Variables) for realizing a full closure with nested functions.

Python 3.x `Version ≥ 3.0`

```
def makeInc(x):
  def inc(y):
    nonlocal x
    # now assigning a value to x is allowed
    x += y
    return x

  return inc

incOne = makeInc(1)
incOne(5) # returns 6
```

Section 33.9: Forcing the use of named parameters

All parameters specified after the first asterisk in the function signature are keyword-only.

```
def f(*a, b):
    pass

f(1, 2, 3)
# TypeError: f() missing 1 required keyword-only argument: 'b'
```

In Python 3 it's possible to put a single asterisk in the function signature to ensure that the remaining arguments may only be passed using keyword arguments.

```
def f(a, b, *, c):
    pass

f(1, 2, 3)
# TypeError: f() takes 2 positional arguments but 3 were given
```

```
f(1, 2, c=3)
# No error
```

Section 33.10: Nested functions

Functions in python are first-class objects. They can be defined in any scope

```
def fibonacci(n):
    def step(a,b):
        return b, a+b
    a, b = 0, 1
    for i in range(n):
        a, b = step(a, b)
    return a
```

Functions capture their enclosing scope can be passed around like any other sort of object

```
def make_adder(n):
    def adder(x):
        return n + x
    return adder
add5 = make_adder(5)
add6 = make_adder(6)
add5(10)
#Out: 15
add6(10)
#Out: 16

def repeatedly_apply(func, n, x):
    for i in range(n):
        x = func(x)
    return x

repeatedly_apply(add5, 5, 1)
#Out: 26
```

Section 33.11: Recursion limit

There is a limit to the depth of possible recursion, which depends on the Python implementation. When the limit is reached, a RuntimeError exception is raised:

```
def cursing(depth):
  try:
    cursing(depth + 1) # actually, re-cursing
  except RuntimeError as RE:
    print('I recursed {} times!'.format(depth))

cursing(0)
# Out: I recursed 1083 times!
```

It is possible to change the recursion depth limit by using `sys.setrecursionlimit(limit)` and check this limit by `sys.getrecursionlimit()`.

```
sys.setrecursionlimit(2000)
cursing(0)
# Out: I recursed 1997 times!
```

From Python 3.5, the exception is a `RecursionError`, which is derived from `RuntimeError`.

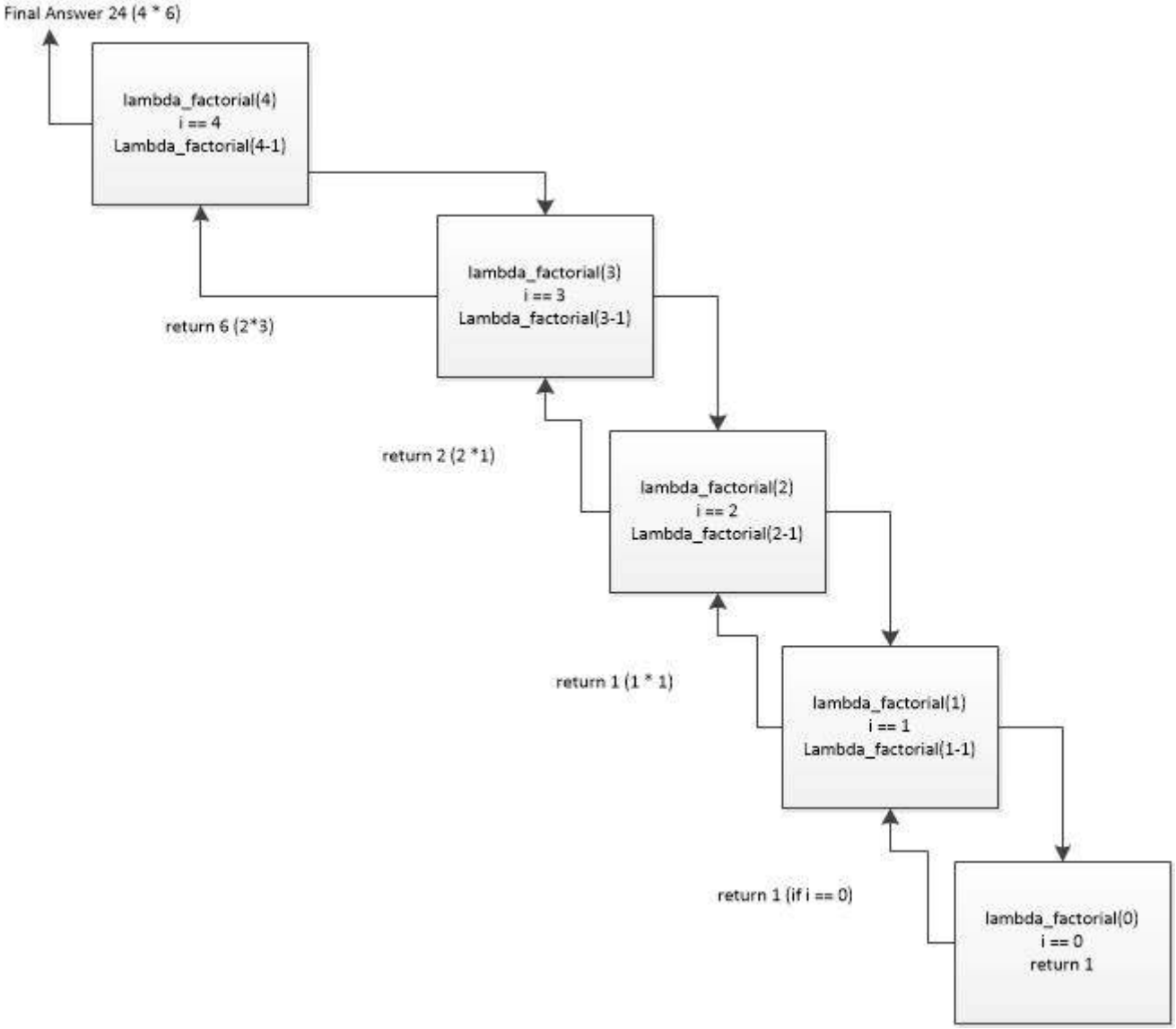
Section 33.12: Recursive Lambda using assigned variable

One method for creating recursive lambda functions involves assigning the function to a variable and then referencing that variable within the function itself. A common example of this is the recursive calculation of the factorial of a number - such as shown in the following code:

```
lambda_factorial = lambda i:1 if i==0 else i*lambda_factorial(i-1)
print(lambda_factorial(4)) # 4 * 3 * 2 * 1 = 12 * 2 = 24
```

Description of code

The lambda function, through its variable assignment, is passed a value (4) which it evaluates and returns 1 if it is 0 or else it returns the current value (`i`) * another calculation by the lambda function of the value - 1 (`i-1`). This continues until the passed value is decremented to 0 (`return 1`). A process which can be visualized as:



Section 33.13: Recursive functions

A recursive function is a function that calls itself in its definition. For example the mathematical function, factorial, defined by `factorial(n) = n*(n-1)*(n-2)*...*3*2*1`. can be programmed as

```
def factorial(n):
```

```
    #n here should be an integer
    if n == 0:
        return 1
    else:
        return n*factorial(n-1)
```

the outputs here are:

```
factorial(0)
#out 1
factorial(1)
#out 1
factorial(2)
#out 2
factorial(3)
#out 6
```

as expected. Notice that this function is recursive because the second `return factorial(n-1)`, where the function calls itself in its definition.

Some recursive functions can be implemented using lambda, the factorial function using lambda would be something like this:

```
factorial = lambda n: 1 if n == 0 else n*factorial(n-1)
```

The function outputs the same as above.

Section 33.14: Defining a function with arguments

Arguments are defined in parentheses after the function name:

```
def divide(dividend, divisor):  # The names of the function and its arguments
    # The arguments are available by name in the body of the function
    print(dividend / divisor)
```

The function name and its list of arguments are called the *signature* of the function. Each named argument is effectively a local variable of the function.

When calling the function, give values for the arguments by listing them in order

```
divide(10, 2)
# output: 5
```

or specify them in any order using the names from the function definition:

```
divide(divisor=2, dividend=10)
# output: 5
```

Section 33.15: Iterable and dictionary unpacking

Functions allow you to specify these types of parameters: positional, named, variable positional, Keyword args (kwargs). Here is a clear and concise use of each type.

```
def unpacking(a, b, c=45, d=60, *args, **kwargs):
    print(a, b, c, d, args, kwargs)
```

```
>>> unpacking(1, 2)
1 2 45 60 () {}
>>> unpacking(1, 2, 3, 4)
1 2 3 4 () {}
>>> unpacking(1, 2, c=3, d=4)
1 2 3 4 () {}
>>> unpacking(1, 2, d=4, c=3)
1 2 3 4 () {}


>>> pair = (3,)
>>> unpacking(1, 2, *pair, d=4)
1 2 3 4 () {}
>>> unpacking(1, 2, d=4, *pair)
1 2 3 4 () {}
>>> unpacking(1, 2, *pair, c=3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'c'
>>> unpacking(1, 2, c=3, *pair)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'c'

>>> args_list = [3]
>>> unpacking(1, 2, *args_list, d=4)
1 2 3 4 () {}
>>> unpacking(1, 2, d=4, *args_list)
1 2 3 4 () {}
>>> unpacking(1, 2, c=3, *args_list)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'c'
>>> unpacking(1, 2, *args_list, c=3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'c'


>>> pair = (3, 4)
>>> unpacking(1, 2, *pair)
1 2 3 4 () {}
>>> unpacking(1, 2, 3, 4, *pair)
1 2 3 4 (3, 4) {}
>>> unpacking(1, 2, d=4, *pair)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'd'
>>> unpacking(1, 2, *pair, d=4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'd'



>>> args_list = [3, 4]
>>> unpacking(1, 2, *args_list)
1 2 3 4 () {}
>>> unpacking(1, 2, 3, 4, *args_list)
1 2 3 4 (3, 4) {}
>>> unpacking(1, 2, d=4, *args_list)
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'd'
>>> unpacking(1, 2, *args_list, d=4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'd'


>>> arg_dict = {'c':3, 'd':4}
>>> unpacking(1, 2, **arg_dict)
1 2 3 4 () {}
>>> arg_dict = {'d':4, 'c':3}
>>> unpacking(1, 2, **arg_dict)
1 2 3 4 () {}
>>> arg_dict = {'c':3, 'd':4, 'not_a_parameter': 75}
>>> unpacking(1, 2, **arg_dict)
1 2 3 4 () {'not_a_parameter': 75}


>>> unpacking(1, 2, *pair, **arg_dict)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'd'
>>> unpacking(1, 2, 3, 4, **arg_dict)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'd'

# Positional arguments take priority over any other form of argument passing
>>> unpacking(1, 2, **arg_dict, c=3)
1 2 3 4 () {'not_a_parameter': 75}
>>> unpacking(1, 2, 3, **arg_dict, c=3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'c'
```

Section 33.16: Defining a function with multiple arguments

One can give a function as many arguments as one wants, the only fixed rules are that each argument name must be unique and that optional arguments must be after the not-optional ones:

```
def func(value1, value2, optionalvalue=10):
    return '{0} {1} {2}'.format(value1, value2, optionalvalue1)
```

When calling the function you can either give each keyword without the name but then the order matters:

```
print(func(1, 'a', 100))
# Out: 1 a 100

print(func('abc', 14))
# abc 14 10
```

Or combine giving the arguments with name and without. Then the ones with name must follow those without but the order of the ones with name doesn't matter:

```
print(func('This', optionalvalue='StackOverflow Documentation', value2='is'))
# Out: This is StackOverflow Documentation
```

Chapter 34: Defining functions with list arguments

Section 34.1: Function and Call

Lists as arguments are just another variable:

```
def func(myList):
    for item in myList:
        print(item)
```

and can be passed in the function call itself:

```
func([1,2,3,5,7])

1
2
3
5
7
```

Or as a variable:

```
aList = ['a','b','c','d']
func(aList)

a
b
c
d
```

Chapter 35: Functional Programming in Python

Functional programming decomposes a problem into a set of functions. Ideally, functions only take inputs and produce outputs, and don't have any internal state that affects the output produced for a given input.below are functional techniques common to many languages: such as lambda, map, reduce.

Section 35.1: Lambda Function

An anonymous, inlined function defined with lambda. The parameters of the lambda are defined to the left of the colon. The function body is defined to the right of the colon. The result of running the function body is (implicitly) returned.

```
s=lambda x:x*x
s(2)    =>4
```

Section 35.2: Map Function

Map takes a function and a collection of items. It makes a new, empty collection, runs the function on each item in the original collection and inserts each return value into the new collection. It returns the new collection.

This is a simple map that takes a list of names and returns a list of the lengths of those names:

```
name_lengths = map(len, ["Mary", "Isla", "Sam"])
print(name_lengths)    =>[4, 4, 3]
```

Section 35.3: Reduce Function

Reduce takes a function and a collection of items. It returns a value that is created by combining the items.

This is a simple reduce. It returns the sum of all the items in the collection.

```
total = reduce(lambda a, x: a + x, [0, 1, 2, 3, 4])
print(total)    =>10
```

Section 35.4: Filter Function

Filter takes a function and a collection. It returns a collection of every item for which the function returned True.

```
arr=[1,2,3,4,5,6]
[i for i in filter(lambda x:x>4,arr)]    # outputs[5,6]
```

Chapter 36: Partial functions

Param	details
x	the number to be raised
y	the exponent
raise	the function to be specialized

As you probably know if you came from OOP school, specializing an abstract class and use it is a practice you should keep in mind when writing your code.

What if you could define an abstract function and specialize it in order to create different versions of it? Thinks it as a sort of *function Inheritance* where you bind specific params to make them reliable for a specific scenario.

Section 36.1: Raise the power

Let's suppose we want raise *x* to a number *y*.

You'd write this as:

```
def raise_power(x, y):
    return x**y
```

What if your *y* value can assume a finite set of values?

Let's suppose *y* can be one of *[3,4,5]* and let's say you don't want offer end user the possibility to use such function since it is very computationally intensive. In fact you would check if provided *y* assumes a valid value and rewrite your function as:

```
def raise(x, y):
    if y in (3,4,5):
        return x**y
    raise NumberNotInRangeException("You should provide a valid exponent")
```

Messy? Let's use the abstract form and specialize it to all three cases: let's implement them **partially**.

```
from functors import partial
raise_to_three = partial(raise, y=3)
raise_to_four = partial(raise, y=4)
raise_to_five = partial(raise, y=5)
```

What happens here? We fixed the *y* params and we defined three different functions.

No need to use the abstract function defined above (you could make it *private*) but you could use **partial applied** functions to deal with raising a number to a fixed value.

Chapter 37: Decorators

Parameter	Details
f	The function to be decorated (wrapped)

Decorator functions are software design patterns. They dynamically alter the functionality of a function, method, or class without having to directly use subclasses or change the source code of the decorated function. When used correctly, decorators can become powerful tools in the development process. This topic covers implementation and applications of decorator functions in Python.

Section 37.1: Decorator function

Decorators augment the behavior of other functions or methods. Any function that takes a function as a parameter and returns an augmented function can be used as a **decorator**.

```
# This simplest decorator does nothing to the function being decorated. Such
# minimal decorators can occasionally be used as a kind of code markers.
def super_secret_function(f):
    return f

@super_secret_function
def my_function():
    print("This is my secret function.")
```

The `@`-notation is syntactic sugar that is equivalent to the following:

```
my_function = super_secret_function(my_function)
```

It is important to bear this in mind in order to understand how the decorators work. This "unsugared" syntax makes it clear why the decorator function takes a function as an argument, and why it should return another function. It also demonstrates what would happen if you *don't* return a function:

```
def disabled(f):
    """
    This function returns nothing, and hence removes the decorated function
    from the local scope.
    """
    pass

@disabled
def my_function():
    print("This function can no longer be called...")

my_function()
# TypeError: 'NoneType' object is not callable
```

Thus, we usually define a *new function* inside the decorator and return it. This new function would first do something that it needs to do, then call the original function, and finally process the return value. Consider this simple decorator function that prints the arguments that the original function receives, then calls it.

```
#This is the decorator
def print_args(func):
    def inner_func(*args, **kwargs):
        print(args)
        print(kwargs)
        return func(*args, **kwargs) #Call the original function with its arguments.
```

```
    return inner_func

@print_args
def multiply(num_a, num_b):
    return num_a * num_b

print(multiply(3, 5))
#Output:
# (3,5) - This is actually the 'args' that the function receives.
# {} - This is the 'kwargs', empty because we didn't specify keyword arguments.
# 15 - The result of the function.
```

Section 37.2: Decorator class

As mentioned in the introduction, a decorator is a function that can be applied to another function to augment its behavior. The syntactic sugar is equivalent to the following: `my_func = decorator(my_func)`. But what if the `decorator` was instead a class? The syntax would still work, except that now `my_func` gets replaced with an instance of the `decorator` class. If this class implements the `__call__()` magic method, then it would still be possible to use `my_func` as if it was a function:

```
class Decorator(object):
    """Simple decorator class."""

    def __init__(self, func):
        self.func = func

    def __call__(self, *args, **kwargs):
        print('Before the function call.')
        res = self.func(*args, **kwargs)
        print('After the function call.')
        return res

@Decorator
def testfunc():
    print('Inside the function.')

testfunc()
# Before the function call.
# Inside the function.
# After the function call.
```

Note that a function decorated with a class decorator will no longer be considered a "function" from type-checking perspective:

```
import types
isinstance(testfunc, types.FunctionType)
# False
type(testfunc)
# <class '__main__.Decorator'>
```

Decorating Methods

For decorating methods you need to define an additional `__get__`-method:

```
from types import MethodType

class Decorator(object):
    def __init__(self, func):
        self.func = func
```

```
    def __call__(self, *args, **kwargs):
        print('Inside the decorator.')
        return self.func(*args, **kwargs)

    def __get__(self, instance, cls):
        # Return a Method if it is called on an instance
        return self if instance is None else MethodType(self, instance)

class Test(object):
    @Decorator
    def __init__(self):
        pass

a = Test()
```

> Inside the decorator.

Warning!

Class Decorators only produce one instance for a specific function so decorating a method with a class decorator will share the same decorator between all instances of that class:

```
from types import MethodType

class CountCallsDecorator(object):
    def __init__(self, func):
        self.func = func
        self.ncalls = 0    # Number of calls of this method

    def __call__(self, *args, **kwargs):
        self.ncalls += 1   # Increment the calls counter
        return self.func(*args, **kwargs)

    def __get__(self, instance, cls):
        return self if instance is None else MethodType(self, instance)

class Test(object):
    def __init__(self):
        pass

    @CountCallsDecorator
    def do_something(self):
        return 'something was done'

a = Test()
a.do_something()
a.do_something.ncalls   # 1
b = Test()
b.do_something()
b.do_something.ncalls   # 2
```

Section 37.3: Decorator with arguments (decorator factory)

A decorator takes just one argument: the function to be decorated. There is no way to pass other arguments.

But additional arguments are often desired. The trick is then to make a function which takes arbitrary arguments and returns a decorator.

Decorator functions

```
def decoratorfactory(message):
    def decorator(func):
        def wrapped_func(*args, **kwargs):
            print('The decorator wants to tell you: {}'.format(message))
            return func(*args, **kwargs)
        return wrapped_func
    return decorator

@decoratorfactory('Hello World')
def test():
    pass

test()
```

> The decorator wants to tell you: Hello World

Important Note:

With such decorator factories you **must** call the decorator with a pair of parentheses:

```
@decoratorfactory # Without parentheses
def test():
    pass

test()
```

> TypeError: decorator() missing 1 required positional argument: 'func'

Decorator classes

```
def decoratorfactory(*decorator_args, **decorator_kwargs):

    class Decorator(object):
        def __init__(self, func):
            self.func = func

        def __call__(self, *args, **kwargs):
            print('Inside the decorator with arguments {}'.format(decorator_args))
            return self.func(*args, **kwargs)

    return Decorator

@decoratorfactory(10)
def test():
    pass

test()
```

> Inside the decorator with arguments (10,)

Section 37.4: Making a decorator look like the decorated

function

Decorators normally strip function metadata as they aren't the same. This can cause problems when using meta-programming to dynamically access function metadata. Metadata also includes function's docstrings and its name. `functools.wraps` makes the decorated function look like the original function by copying several attributes to the wrapper function.

```
from functools import wraps
```

The two methods of wrapping a decorator are achieving the same thing in hiding that the original function has been decorated. There is no reason to prefer the function version to the class version unless you're already using one over the other.

As a function

```
def decorator(func):
    # Copies the docstring, name, annotations and module to the decorator
    @wraps(func)
    def wrapped_func(*args, **kwargs):
        return func(*args, **kwargs)
    return wrapped_func

@decorator
def test():
    pass

test.__name__
```

> 'test'

As a class

```
class Decorator(object):
    def __init__(self, func):
        # Copies name, module, annotations and docstring to the instance.
        self._wrapped = wraps(func)(self)

    def __call__(self, *args, **kwargs):
        return self._wrapped(*args, **kwargs)

@Decorator
def test():
    """Docstring of test."""
    pass

test.__doc__
```

> 'Docstring of test.'

Section 37.5: Using a decorator to time a function

```
import time
def timer(func):
    def inner(*args, **kwargs):
        t1 = time.time()
```

```
        f = func(*args, **kwargs)
        t2 = time.time()
        print 'Runtime took {0} seconds'.format(t2-t1)
        return f
    return inner

@timer
def example_function():
    #do stuff


example_function()
```

Section 37.6: Create singleton class with a decorator

A singleton is a pattern that restricts the instantiation of a class to one instance/object. Using a decorator, we can define a class as a singleton by forcing the class to either return an existing instance of the class or create a new instance (if it doesn't exist).

```
def singleton(cls):
    instance = [None]
    def wrapper(*args, **kwargs):
        if instance[0] is None:
            instance[0] = cls(*args, **kwargs)
        return instance[0]

    return wrapper
```

This decorator can be added to any class declaration and will make sure that at most one instance of the class is created. Any subsequent calls will return the already existing class instance.

```
@singleton
class SomeSingletonClass:
    x = 2
    def __init__(self):
        print("Created!")

instance = SomeSingletonClass()  # prints: Created!
instance = SomeSingletonClass()  # doesn't print anything
print(instance.x)                # 2

instance.x = 3
print(SomeSingletonClass().x)    # 3
```

So it doesn't matter whether you refer to the class instance via your local variable or whether you create another "instance", you always get the same object.

Chapter 38: Classes

Python offers itself not only as a popular scripting language, but also supports the object-oriented programming paradigm. Classes describe data and provide methods to manipulate that data, all encompassed under a single object. Furthermore, classes allow for abstraction by separating concrete implementation details from abstract representations of data.

Code utilizing classes is generally easier to read, understand, and maintain.

Section 38.1: Introduction to classes

A class, functions as a template that defines the basic characteristics of a particular object. Here's an example:

```
class Person(object):
    """A simple class."""                           # docstring
    species = "Homo Sapiens"                        # class attribute

    def __init__(self, name):                       # special method
        """This is the initializer. It's a special
        method (see below).
        """
        self.name = name                            # instance attribute

    def __str__(self):                              # special method
        """This method is run when Python tries
        to cast the object to a string. Return
        this string when using print(), etc.
        """
        return self.name

    def rename(self, renamed):                      # regular method
        """Reassign and print the name attribute."""
        self.name = renamed
        print("Now my name is {}".format(self.name))
```

There are a few things to note when looking at the above example.

1. The class is made up of *attributes* (data) and *methods* (functions).
2. Attributes and methods are simply defined as normal variables and functions.
3. As noted in the corresponding docstring, the `__init__()` method is called the *initializer*. It's equivalent to the constructor in other object oriented languages, and is the method that is first run when you create a new object, or new instance of the class.
4. Attributes that apply to the whole class are defined first, and are called *class attributes*.
5. Attributes that apply to a specific instance of a class (an object) are called *instance attributes*. They are generally defined inside `__init__()`; this is not necessary, but it is recommended (since attributes defined outside of `__init__()` run the risk of being accessed before they are defined).
6. Every method, included in the class definition passes the object in question as its first parameter. The word `self` is used for this parameter (usage of `self` is actually by convention, as the word `self` has no inherent meaning in Python, but this is one of Python's most respected conventions, and you should always follow it).
7. Those used to object-oriented programming in other languages may be surprised by a few things. One is that Python has no real concept of `private` elements, so everything, by default, imitates the behavior of the C++/Java `public` keyword. For more information, see the "Private Class Members" example on this page.
8. Some of the class's methods have the following form: `__functionname__(self, other_stuff)`. All such methods are called "magic methods" and are an important part of classes in Python. For instance, operator overloading in Python is implemented with magic methods. For more information, see the relevant

documentation.

Now let's make a few instances of our `Person` class!

```
>>> # Instances
>>> kelly = Person("Kelly")
>>> joseph = Person("Joseph")
>>> john_doe = Person("John Doe")
```

We currently have three `Person` objects, `kelly`, `joseph`, and `john_doe`.

We can access the attributes of the class from each instance using the dot operator `.` Note again the difference between class and instance attributes:

```
>>> # Attributes
>>> kelly.species
'Homo Sapiens'
>>> john_doe.species
'Homo Sapiens'
>>> joseph.species
'Homo Sapiens'
>>> kelly.name
'Kelly'
>>> joseph.name
'Joseph'
```

We can execute the methods of the class using the same dot operator `.`:

```
>>> # Methods
>>> john_doe.__str__()
'John Doe'
>>>  print(john_doe)
'John Doe'
>>>  john_doe.rename("John")
'Now my name is John'
```

Section 38.2: Bound, unbound, and static methods

The idea of bound and unbound methods was removed in Python 3. In Python 3 when you declare a method within a class, you are using a `def` keyword, thus creating a function object. This is a regular function, and the surrounding class works as its namespace. In the following example we declare method `f` within class `A`, and it becomes a function `A.f`:

Python 3.x `Version ≥ 3.0`

```
class A(object):
    def f(self, x):
        return 2 * x
A.f
# <function A.f at ...>  (in Python 3.x)
```

In Python 2 the behavior was different: function objects within the class were implicitly replaced with objects of type `instancemethod`, which were called *unbound methods* because they were not bound to any particular class instance. It was possible to access the underlying function using `.__func__` property.

Python 2.x `Version ≥ 2.3`

```
A.f
```

```
# <unbound method A.f>   (in Python 2.x)
A.f.__class__
# <type 'instancemethod'>
A.f.__func__
# <function f at ...>
```

The latter behaviors are confirmed by inspection - methods are recognized as functions in Python 3, while the distinction is upheld in Python 2.

Python 3.x `Version ≥ 3.0`

```
import inspect

inspect.isfunction(A.f)
# True
inspect.ismethod(A.f)
# False
```

Python 2.x `Version ≥ 2.3`

```
import inspect

inspect.isfunction(A.f)
# False
inspect.ismethod(A.f)
# True
```

In both versions of Python function/method `A.f` can be called directly, provided that you pass an instance of class `A` as the first argument.

```
A.f(1, 7)
# Python 2: TypeError: unbound method f() must be called with
#                      A instance as first argument (got int instance instead)
# Python 3: 14
a = A()
A.f(a, 20)
# Python 2 & 3: 40
```

Now suppose `a` is an instance of class `A`, what is `a.f` then? Well, intuitively this should be the same method `f` of class `A`, only it should somehow "know" that it was applied to the object `a` – in Python this is called method *bound* to `a`.

The nitty-gritty details are as follows: writing `a.f` invokes the magic `__getattribute__` method of `a`, which first checks whether `a` has an attribute named `f` (it doesn't), then checks the class `A` whether it contains a method with such a name (it does), and creates a new object `m` of type `method` which has the reference to the original `A.f` in `m.__func__`, and a reference to the object `a` in `m.__self__`. When this object is called as a function, it simply does the following: `m(...) => m.__func__(m.__self__, ...)`. Thus this object is called a **bound method** because when invoked it knows to supply the object it was bound to as the first argument. (These things work same way in Python 2 and 3).

```
a = A()
a.f
# <bound method A.f of <__main__.A object at ...>>
a.f(2)
# 4

# Note: the bound method object a.f is recreated *every time* you call it:
a.f is a.f  # False
# As a performance optimization you can store the bound method in the object's
# __dict__, in which case the method object will remain fixed:
a.f = a.f
```

```
a.f is a.f  # True
```

Finally, Python has **class methods** and **static methods** – special kinds of methods. Class methods work the same way as regular methods, except that when invoked on an object they bind to the *class* of the object instead of to the object. Thus `m.__self__ = type(a)`. When you call such bound method, it passes the class of `a` as the first argument. Static methods are even simpler: they don't bind anything at all, and simply return the underlying function without any transformations.

```
class D(object):
    multiplier = 2

    @classmethod
    def f(cls, x):
        return cls.multiplier * x

    @staticmethod
    def g(name):
        print("Hello, %s" % name)

D.f
# <bound method type.f of <class '__main__.D'>>
D.f(12)
# 24
D.g
# <function D.g at ...>
D.g("world")
# Hello, world
```

Note that class methods are bound to the class even when accessed on the instance:

```
d = D()
d.multiplier = 1337
(D.multiplier, d.multiplier)
# (2, 1337)
d.f
# <bound method D.f of <class '__main__.D'>>
d.f(10)
# 20
```

It is worth noting that at the lowest level, functions, methods, staticmethods, etc. are actually descriptors that invoke `__get__`, `__set__` and optionally `__del__` special methods. For more details on classmethods and staticmethods:

- What is the difference between @staticmethod and @classmethod in Python?
- Meaning of @classmethod and @staticmethod for beginner?

Section 38.3: Basic inheritance

Inheritance in Python is based on similar ideas used in other object oriented languages like Java, C++ etc. A new class can be derived from an existing class as follows.

```
class BaseClass(object):
    pass

class DerivedClass(BaseClass):
    pass
```

The `BaseClass` is the already existing (*parent*) class, and the `DerivedClass` is the new (*child*) class that inherits (or *subclasses*) attributes from `BaseClass`. **Note**: As of Python 2.2, all classes implicitly inherit from the `object` class, which is the base class for all built-in types.

We define a parent `Rectangle` class in the example below, which implicitly inherits from `object`:

```
class Rectangle():
    def __init__(self, w, h):
        self.w = w
        self.h = h

    def area(self):
        return self.w * self.h

    def perimeter(self):
        return 2 * (self.w + self.h)
```

The `Rectangle` class can be used as a base class for defining a `Square` class, as a square is a special case of rectangle.

```
class Square(Rectangle):
    def __init__(self, s):
        # call parent constructor, w and h are both s
        super(Square, self).__init__(s, s)
        self.s = s
```

The `Square` class will automatically inherit all attributes of the `Rectangle` class as well as the object class. `super()` is used to call the `__init__()` method of `Rectangle` class, essentially calling any overridden method of the base class. **Note**: in Python 3, `super()` does not require arguments.

Derived class objects can access and modify the attributes of its base classes:

```
r.area()
# Output: 12
r.perimeter()
# Output: 14

s.area()
# Output: 4
s.perimeter()
# Output: 8
```

Built-in functions that work with inheritance

`issubclass(DerivedClass, BaseClass)`: returns `True` if `DerivedClass` is a subclass of the `BaseClass`

`isinstance(s, Class)`: returns `True` if s is an instance of `Class` or any of the derived classes of `Class`

```
# subclass check
issubclass(Square, Rectangle)
# Output: True

# instantiate
r = Rectangle(3, 4)
s = Square(2)

isinstance(r, Rectangle)
# Output: True
isinstance(r, Square)
```

```
# Output: False
# A rectangle is not a square

isinstance(s, Rectangle)
# Output: True
# A square is a rectangle
isinstance(s, Square)
# Output: True
```

Section 38.4: Monkey Patching

In this case, "monkey patching" means adding a new variable or method to a class after it's been defined. For instance, say we defined class `A` as

```
class A(object):
    def __init__(self, num):
        self.num = num

    def __add__(self, other):
        return A(self.num + other.num)
```

But now we want to add another function later in the code. Suppose this function is as follows.

```
def get_num(self):
    return self.num
```

But how do we add this as a method in `A`? That's simple we just essentially place that function into `A` with an assignment statement.

```
A.get_num = get_num
```

Why does this work? Because functions are objects just like any other object, and methods are functions that belong to the class.

The function `get_num` shall be available to all existing (already created) as well to the new instances of `A`

These additions are available on all instances of that class (or its subclasses) automatically. For example:

```
foo = A(42)

A.get_num = get_num

bar = A(6);

foo.get_num() # 42

bar.get_num() # 6
```

Note that, unlike some other languages, this technique does not work for certain built-in types, and it is not considered good style.

Section 38.5: New-style vs. old-style classes

Python 2.x `Version ≥ 2.2.0`

New-style classes were introduced in Python 2.2 to unify *classes* and *types*. They inherit from the top-level `object`

type. *A new-style class is a user-defined type*, and is very similar to built-in types.

```
# new-style class
class New(object):
    pass

# new-style instance
new = New()

new.__class__
# <class '__main__.New'>
type(new)
# <class '__main__.New'>
issubclass(New, object)
# True
```

Old-style classes do **not** inherit from `object`. Old-style instances are always implemented with a built-in `instance` type.

```
# old-style class
class Old:
    pass

# old-style instance
old = Old()

old.__class__
# <class __main__.Old at ...>
type(old)
# <type 'instance'>
issubclass(Old, object)
# False
```

Python 3.x `Version ≥ 3.0.0`

In Python 3, old-style classes were removed.

New-style classes in Python 3 implicitly inherit from `object`, so there is no need to specify `MyClass(object)` anymore.

```
class MyClass:
    pass

my_inst = MyClass()

type(my_inst)
# <class '__main__.MyClass'>
my_inst.__class__
# <class '__main__.MyClass'>
issubclass(MyClass, object)
# True
```

Section 38.6: Class methods: alternate initializers

Class methods present alternate ways to build instances of classes. To illustrate, let's look at an example.

Let's suppose we have a relatively simple `Person` class:

```
class Person(object):
```

```
    def __init__(self, first_name, last_name, age):
        self.first_name = first_name
        self.last_name = last_name
        self.age = age
        self.full_name = first_name + " " + last_name

    def greet(self):
        print("Hello, my name is " + self.full_name + ".")
```

It might be handy to have a way to build instances of this class specifying a full name instead of first and last name separately. One way to do this would be to have `last_name` be an optional parameter, and assuming that if it isn't given, we passed the full name in:

```
class Person(object):

    def __init__(self, first_name, age, last_name=None):
        if last_name is None:
            self.first_name, self.last_name = first_name.split(" ", 2)
        else:
            self.first_name = first_name
            self.last_name = last_name

        self.full_name = self.first_name + " " + self.last_name
        self.age = age

    def greet(self):
        print("Hello, my name is " + self.full_name + ".")
```

However, there are two main problems with this bit of code:

1. The parameters `first_name` and `last_name` are now misleading, since you can enter a full name for `first_name`. Also, if there are more cases and/or more parameters that have this kind of flexibility, the if/elif/else branching can get annoying fast.
2. Not quite as important, but still worth pointing out: what if `last_name` is `None`, but `first_name` doesn't split into two or more things via spaces? We have yet another layer of input validation and/or exception handling...

Enter class methods. Rather than having a single initializer, we will create a separate initializer, called `from_full_name`, and decorate it with the (built-in) `classmethod` decorator.

```
class Person(object):

    def __init__(self, first_name, last_name, age):
        self.first_name = first_name
        self.last_name = last_name
        self.age = age
        self.full_name = first_name + " " + last_name

    @classmethod
    def from_full_name(cls, name, age):
        if " " not in name:
            raise ValueError
        first_name, last_name = name.split(" ", 2)
        return cls(first_name, last_name, age)

    def greet(self):
        print("Hello, my name is " + self.full_name + ".")
```

Notice `cls` instead of `self` as the first argument to `from_full_name`. Class methods are applied to the overall class, *not* an instance of a given class (which is what `self` usually denotes). So, if `cls` is our `Person` class, then the returned value from the `from_full_name` class method is `Person(first_name, last_name, age)`, which uses `Person`'s `__init__` to create an instance of the `Person` class. In particular, if we were to make a subclass `Employee` of `Person`, then `from_full_name` would work in the `Employee` class as well.

To show that this works as expected, let's create instances of `Person` in more than one way without the branching in `__init__`:

```
In [2]: bob = Person("Bob", "Bobberson", 42)

In [3]: alice = Person.from_full_name("Alice Henderson", 31)

In [4]: bob.greet()
Hello, my name is Bob Bobberson.

In [5]: alice.greet()
Hello, my name is Alice Henderson.
```

Other references:

- Python @classmethod and @staticmethod for beginner?
- https://docs.python.org/2/library/functions.html#classmethod
- https://docs.python.org/3.5/library/functions.html#classmethod

Section 38.7: Multiple Inheritance

Python uses the C3 linearization algorithm to determine the order in which to resolve class attributes, including methods. This is known as the Method Resolution Order (MRO).

Here's a simple example:

```
class Foo(object):
    foo = 'attr foo of Foo'


class Bar(object):
    foo = 'attr foo of Bar' # we won't see this.
    bar = 'attr bar of Bar'

class FooBar(Foo, Bar):
    foobar = 'attr foobar of FooBar'
```

Now if we instantiate FooBar, if we look up the foo attribute, we see that Foo's attribute is found first

```
fb = FooBar()
```

and

```
>>> fb.foo
'attr foo of Foo'
```

Here's the MRO of FooBar:

```
>>> FooBar.mro()
[<class '__main__.FooBar'>, <class '__main__.Foo'>, <class '__main__.Bar'>, <type 'object'>]
```

It can be simply stated that Python's MRO algorithm is

1. Depth first (e.g. `FooBar` then `Foo`) unless
2. a shared parent (`object`) is blocked by a child (`Bar`) and
3. no circular relationships allowed.

That is, for example, Bar cannot inherit from FooBar while FooBar inherits from Bar.

For a comprehensive example in Python, see the wikipedia entry.

Another powerful feature in inheritance is `super`. super can fetch parent classes features.

```
class Foo(object):
    def foo_method(self):
        print "foo Method"

class Bar(object):
    def bar_method(self):
        print "bar Method"

class FooBar(Foo, Bar):
    def foo_method(self):
        super(FooBar, self).foo_method()
```

Multiple inheritance with init method of class, when every class has own init method then we try for multiple inheritance then only init method get called of class which is inherit first.

for below example only Foo class **init** method getting called **Bar** class init not getting called

```
    class Foo(object):
        def __init__(self):
            print "foo init"

    class Bar(object):
        def __init__(self):
            print "bar init"

    class FooBar(Foo, Bar):
        def __init__(self):
            print "foobar init"
            super(FooBar, self).__init__()

    a = FooBar()
```

Output:

```
    foobar init
    foo init
```

But it doesn't mean that **Bar** class is not inherit. Instance of final **FooBar** class is also instance of **Bar** class and **Foo** class.

```
print isinstance(a,FooBar)
print isinstance(a,Foo)
```

```
print isinstance(a,Bar)
```

Output:

```
True
True
True
```

Section 38.8: Properties

Python classes support **properties**, which look like regular object variables, but with the possibility of attaching custom behavior and documentation.

```
class MyClass(object):

    def __init__(self):
       self._my_string = ""

    @property
    def string(self):
        """A profoundly important string."""
        return self._my_string

    @string.setter
    def string(self, new_value):
        assert isinstance(new_value, str), \
               "Give me a string, not a %r!" % type(new_value)
        self._my_string = new_value

    @string.deleter
    def x(self):
        self._my_string = None
```

The object's of class `MyClass` will *appear* to have a property `.string`, however it's behavior is now tightly controlled:

```
mc = MyClass()
mc.string = "String!"
print(mc.string)
del mc.string
```

As well as the useful syntax as above, the property syntax allows for validation, or other augmentations to be added to those attributes. This could be especially useful with public APIs - where a level of help should be given to the user.

Another common use of properties is to enable the class to present 'virtual attributes' - attributes which aren't actually stored but are computed only when requested.

```
class Character(object):
    def __init__(name, max_hp):
        self._name = name
        self._hp = max_hp
        self._max_hp = max_hp

    # Make hp read only by not providing a set method
    @property
    def hp(self):
        return self._hp
```

```
    # Make name read only by not providing a set method
    @property
    def name(self):
        return self.name

    def take_damage(self, damage):
        self.hp -= damage
        self.hp = 0 if self.hp <0 else self.hp

    @property
    def is_alive(self):
        return self.hp != 0

    @property
    def is_wounded(self):
        return self.hp < self.max_hp if self.hp > 0 else False

    @property
    def is_dead(self):
        return not self.is_alive

bilbo = Character('Bilbo Baggins', 100)
bilbo.hp
# out : 100
bilbo.hp = 200
# out : AttributeError: can't set attribute
# hp attribute is read only.

bilbo.is_alive
# out : True
bilbo.is_wounded
# out : False
bilbo.is_dead
# out : False

bilbo.take_damage( 50 )

bilbo.hp
# out : 50

bilbo.is_alive
# out : True
bilbo.is_wounded
# out : True
bilbo.is_dead
# out : False

bilbo.take_damage( 50 )
bilbo.hp
# out : 0

bilbo.is_alive
# out : False
bilbo.is_wounded
# out : False
bilbo.is_dead
# out : True
```

Section 38.9: Default values for instance variables

If the variable contains a value of an immutable type (e.g. a string) then it is okay to assign a default value like this

```
class Rectangle(object):
    def __init__(self, width, height, color='blue'):
        self.width = width
        self.height = height
        self.color = color

    def area(self):
        return self.width  * self.height

# Create some instances of the class
default_rectangle = Rectangle(2, 3)
print(default_rectangle.color) # blue

red_rectangle = Rectangle(2, 3, 'red')
print(red_rectangle.color) # red
```

One needs to be careful when initializing mutable objects such as lists in the constructor. Consider the following example:

```
class Rectangle2D(object):
    def __init__(self, width, height, pos=[0,0], color='blue'):
        self.width = width
        self.height = height
        self.pos = pos
        self.color = color

r1 = Rectangle2D(5,3)
r2 = Rectangle2D(7,8)
r1.pos[0] = 4
r1.pos # [4, 0]
r2.pos # [4, 0] r2's pos has changed as well
```

This behavior is caused by the fact that in Python default parameters are bound at function execution and not at function declaration. To get a default instance variable that's not shared among instances, one should use a construct like this:

```
class Rectangle2D(object):
    def __init__(self, width, height, pos=None, color='blue'):
        self.width = width
        self.height = height
        self.pos = pos or [0, 0] # default value is [0, 0]
        self.color = color

r1 = Rectangle2D(5,3)
r2 = Rectangle2D(7,8)
r1.pos[0] = 4
r1.pos # [4, 0]
r2.pos # [0, 0] r2's pos hasn't changed
```

See also Mutable Default Arguments and "Least Astonishment" and the Mutable Default Argument.

Section 38.10: Class and instance variables

Instance variables are unique for each instance, while class variables are shared by all instances.

```
class C:
    x = 2  # class variable
```

```
    def __init__(self, y):
        self.y = y  # instance variable

C.x
# 2
C.y
# AttributeError: type object 'C' has no attribute 'y'

c1 = C(3)
c1.x
# 2
c1.y
# 3

c2 = C(4)
c2.x
# 2
c2.y
# 4
```

Class variables can be accessed on instances of this class, but assigning to the class attribute will create an instance variable which shadows the class variable

```
c2.x = 4
c2.x
# 4
C.x
# 2
```

Note that *mutating* class variables from instances can lead to some unexpected consequences.

```
class D:
    x = []
    def __init__(self, item):
        self.x.append(item)  # note that this is not an assignment!

d1 = D(1)
d2 = D(2)

d1.x
# [1, 2]
d2.x
# [1, 2]
D.x
# [1, 2]
```

Section 38.11: Class composition

Class composition allows explicit relations between objects. In this example, people live in cities that belong to countries. Composition allows people to access the number of all people living in their country:

```
class Country(object):
    def __init__(self):
        self.cities=[]

    def addCity(self,city):
        self.cities.append(city)
```

```
class City(object):
    def __init__(self, numPeople):
        self.people = []
        self.numPeople = numPeople


    def addPerson(self, person):
        self.people.append(person)

    def join_country(self,country):
        self.country = country
        country.addCity(self)

        for i in range(self.numPeople):
                person(i).join_city(self)


class Person(object):
    def __init__(self, ID):
        self.ID=ID

    def join_city(self, city):
        self.city = city
        city.addPerson(self)

    def people_in_my_country(self):
        x= sum([len(c.people) for c in self.city.country.cities])
        return x

US=Country()
NYC=City(10).join_country(US)
SF=City(5).join_country(US)

print(US.cities[0].people[0].people_in_my_country())

# 15
```

Section 38.12: Listing All Class Members

The `dir()` function can be used to get a list of the members of a class:

```
dir(Class)
```

For example:

```
>>> dir(list)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__',
'__iadd__', '__imul__', '__init__', '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__', '__setattr__',
'__setitem__', '__sizeof__', '__str__', '__subclasshook__', 'append', 'clear', 'copy', 'count',
'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

It is common to look only for "non-magic" members. This can be done using a simple comprehension that lists members with names not starting with `__`:

```
>>> [m for m in dir(list) if not m.startswith('__')]
['append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse',
'sort']
```

Caveats:

Classes can define a `__dir__()` method. If that method exists calling `dir()` will call `__dir__()`, otherwise Python will try to create a list of members of the class. This means that the dir function can have unexpected results. Two quotes of importance from the official python documentation:

> If the object does not provide **dir**(), the function tries its best to gather information from the object's **dict** attribute, if defined, and from its type object. The resulting list is not necessarily complete, and may be inaccurate when the object has a custom **getattr**().

> **Note:** Because dir() is supplied primarily as a convenience for use at an interactive prompt, it tries to supply an interesting set of names more than it tries to supply a rigorously or consistently defined set of names, and its detailed behavior may change across releases. For example, metaclass attributes are not in the result list when the argument is a class.

Section 38.13: Singleton class

A singleton is a pattern that restricts the instantiation of a class to one instance/object. For more info on python singleton design patterns, see here.

```
class Singleton:
    def __new__(cls):
        try:
            it = cls.__it__
        except AttributeError:
            it = cls.__it__ = object.__new__(cls)
        return it

    def __repr__(self):
        return '<{}>'.format(self.__class__.__name__.upper())

    def __eq__(self, other):
         return other is self
```

Another method is to decorate your class. Following the example from this answer create a Singleton class:

```
class Singleton:
    """
    A non-thread-safe helper class to ease implementing singletons.
    This should be used as a decorator -- not a metaclass -- to the
    class that should be a singleton.

    The decorated class can define one `__init__` function that
    takes only the `self` argument. Other than that, there are
    no restrictions that apply to the decorated class.

    To get the singleton instance, use the `Instance` method. Trying
    to use `__call__` will result in a `TypeError` being raised.

    Limitations: The decorated class cannot be inherited from.

    """

    def __init__(self, decorated):
```

```
        self._decorated = decorated

    def Instance(self):
        """
        Returns the singleton instance. Upon its first call, it creates a
        new instance of the decorated class and calls its `__init__` method.
        On all subsequent calls, the already created instance is returned.

        """
        try:
            return self._instance
        except AttributeError:
            self._instance = self._decorated()
            return self._instance

    def __call__(self):
        raise TypeError('Singletons must be accessed through `Instance()`.')

    def __instancecheck__(self, inst):
        return isinstance(inst, self._decorated)
```

To use you can use the `Instance` method

```
@Singleton
class Single:
    def __init__(self):
        self.name=None
        self.val=0
    def getName(self):
        print(self.name)

x=Single.Instance()
y=Single.Instance()
x.name='I\'m single'
x.getName() # outputs I'm single
y.getName() # outputs I'm single
```

Section 38.14: Descriptors and Dotted Lookups

Descriptors are objects that are (usually) attributes of classes and that have any of `__get__`, `__set__`, or `__delete__` special methods.

Data Descriptors have any of `__set__`, or `__delete__`

These can control the dotted lookup on an instance, and are used to implement functions, `staticmethod`, `classmethod`, and `property`. A dotted lookup (e.g. instance `foo` of class `Foo` looking up attribute `bar` - i.e. `foo.bar`) uses the following algorithm:

1. `bar` is looked up in the class, `Foo`. If it is there and it is a **Data Descriptor**, then the data descriptor is used. That's how `property` is able to control access to data in an instance, and instances cannot override this. If a **Data Descriptor** is not there, then
2. `bar` is looked up in the instance `__dict__`. This is why we can override or block methods being called from an instance with a dotted lookup. If `bar` exists in the instance, it is used. If not, we then
3. look in the class `Foo` for `bar`. If it is a **Descriptor**, then the descriptor protocol is used. This is how functions (in this context, unbound methods), `classmethod`, and `staticmethod` are implemented. Else it simply returns the object there, or there is an `AttributeError`