Collections

https://docs.python.org/3/library/stdtypes.html#types-set (https://docs.python.org/3/library/stdtypes.html#types-set) In Python, there are several built-in collections, which can be used to group multiple objects together in different manners.

- Lists are used to store multiple objects in a sequence;
- Tuples are similar to lists but tuples are immutable;
- Sets are used to store unique elements and useful to frequently check the existence of an element;
- Dictionaries are used to store key-value pairs;
- Ranges are another types of immutable sequence and used to create ranges of integers. Ranges are generators.

Lists

- A list can be used to store multiple values, which can be accessed by their indices in the list.
- If a list x contains n elements, these n elements can be accessed by their indices as x[0], …, x[n-1].

In [1]:

```
# a list of strings
animals = ['cat', 'dog', 'fish', 'bison']

# a list of integers
numbers = [1, 7, 34, 20, 12]

# an empty list
an_empty_list = []
another_empty_list = list()

# a list of variables we defined somewhere else
things = [
animals,
numbers,
an_empty_list, # this trailing comma is legal in Python
]
```

In [2]:

```
print(animals[0]) # cat
print(numbers[1]) # 7

# This will give us an error, because the list only has four elements
try:
    print(animals[6])
except IndexError as e:
    print(e)

# access a list element by the index from the end of a list
print(animals[-1]) # the last element -- bison
print(numbers[-2]) # the second-last element – 20

# access a range of list elements animals[start:end:step]
print(animals[1:3]) # ['dog', 'fish']
print(animals[1:-1]) # ['dog', 'fish']
print(animals[2:]) # ['fish', 'bison']
print(animals[:2]) # ['cat', 'dog']
print(animals[:]) # a copy of the whole list
print(animals[::2]) # ['cat', 'fish']
```

```
cat
7
list index out of range
bison
20
['dog', 'fish']
['dog', 'fish']
['fish', 'bison']
['cat', 'dog']
['cat', 'dog', 'fish', 'bison']
['cat', 'fish']
```

In [3]:

```
print(things[0])
print(things[0][:2])
```

```
['cat', 'dog', 'fish', 'bison']
['cat', 'dog']
```

In [4]:

```
# Assign a new value to an existing element
animals[3] = "hamster"
```

In [5]:

```
# Add a new element to the end of the list
animals.append("squirrel")

# Remove an element by its index
del animals[2]
print(animals)
```

```
['cat', 'dog', 'hamster', 'squirrel']
```

Lists are mutable.

In [6]:

```
# Copy a list
animals_clone = animals.copy() # animals_clone = animals[:]; animals_clone = list(anim
print(animals_clone == animals) # True
print(animals_clone is animals) # False
animals_clone[0] = 'human'
print(animals_clone)
print(animals)
```

```
True
False
['human', 'dog', 'hamster', 'squirrel']
['cat', 'dog', 'hamster', 'squirrel']
```

In [7]:

```
# In Python, an assignment only creates bindings between a target and an object.
animals2 = animals;
print(animals2 == animals) # True
print(animals2 is animals) # True
animals2[0] = 'human'
print(animals)
print(animals2)
```

```
True
True
['human', 'dog', 'hamster', 'squirrel']
['human', 'dog', 'hamster', 'squirrel']
```

In [16]:

```
# A list can contain objects of different types
my_list = ['cat', 12, 35.8]

# Use the membership operators in and not in to check whether or not a list contains a
data = [1,2,3,4]
x = 10
if x in data:
    print("{} is in the list".format(x))
if x not in data:
    print("{} is not in the list".format(x))
```

```
10 is not in the list
```

List Methods and Functions

Built-in functions

- the length of a list

```
len(animals)
```

- the sum of a list of numbers

```
sum(numbers)
```

- max/min of a list of numbers

```
max(numbers) # min(numbers)
```

- are any of these values true?

```
any([1,0,1,0,1])
if any(numbers):
    ...
```

- are all of these values true?

```
all([1,0,1,0,1])
if all(numbers):
    ...
```

- output a sorted list:

```
sorted(numbers)
sorted(numbers, reverse=True)
```

- output a reversed list: reversed returns a generator, which is not a copy of the reversed list, so we have to covert it to a list before modifying it

```
list(reversed(numbers))
```

Useful arithmetic operators on lists

- Concatenate two lists

```
[1,2,3,4]+['a','b','c'] # [1, 2, 3, 4, 'a', 'b', 'c']
```

- Concatenate a list with itself multiple times

```
[1,2,3]*3 # [1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Instance functions (numbers = [1, 2, 3, 4, 5])

- add an element to the end:

```
numbers.append(5)
```

- count how many times a value appears in the list:

```
numbers.count(5)
```

- append several values at once to the end:

```
numbers.extend([56, 2, 12])
```

- find the index of a value:

```
numbers.index(3)
```

  - if the value appears more than once, we will get the index of the first one
  - if the value is not in the list, we will get a ValueError

- insert a value at a particular index:

```
numbers.insert(0, 45) # insert 45 at the beginning of the list
```

- remove an element by its index and assign it to a variable:

```
my_number = numbers.pop(0)
```

- remove an element by its value:

```
numbers.remove(12)
```

  - the first occurrence of this element will be removed.
- sort the elements in a list (in-place and stable sort):

```
numbers.sort()
numbers.sort(reverse=True)
```

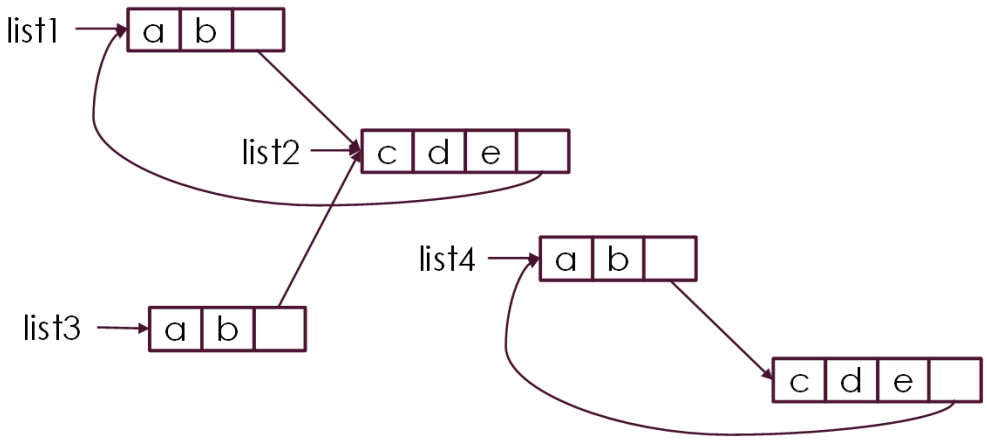
- reverse the elements in a list (in-place):

```
numbers.reverse()
```

Shallow Copy and Deep Copy

- Deep copy and shallow copy are different when copying a compound object (objects contain other objects)
  - A shallow copy constructs a new compound object and inserts references into it to the objects found in the original.
  - A deep copy constructs a new compound object and then, recursively, inserts copies into it of the objects found in the original. Deep copy can handle recursive objects.
  - To use deepcopy, import the module copy first.

Example



In [22]:

```
list1=['a','b']
list2=['c','d','e']
list1.append(list2)
list2.append(list1)
#list1 ['a', 'b', ['c', 'd', 'e', [...]]]
#list2 ['c', 'd', 'e', ['a', 'b', [...]]]

import copy
list3= list1.copy() # copy.copy(list1)
list4= copy.deepcopy(list1)

print(list1[2] is list2)
print(list2[3] is list1)
print(list3[2] is list2)
print(list4[2][3] is list1)
print(list4[2][3] is list4)
```

```
True
True
True
False
True
```

Tuples

- Tuples are similar to lists but tuples are immutable. Tuples are suitable for creating a sequence of values, which will not be modified.

```
animals = ('cat', 'dog', 'fish')
#empty tuple
empty_tuple = tuple()
```

- The elements in a tuple can be accessed by the same way of accessing the elements in a list.

```
animals[0]
animals[-1]
animals[0:]
```

- Use tuples to insert multiple values into a formatted string: "%d,%d,%d"%(1,2,3)
- Use (x,) to create a tuple of a single element x. Note that there is a trailing comma.

In [25]:

```
tuple_a = (1)
tuple_b = (1,)
print(tuple_a,tuple_b)
```

```
1 (1,)
```

Sets

- A set is a collection of unique hashable elements.
  - An object is hashable if it has a hash value not changed during its lifetime (it needs a __hash__() method), and can be compared to other objects (it needs an __eq__() or __cmp__() method). Hashable objects which compare equal must have the same hash value.
  - Hashability makes an object usable as a dictionary key and a set member, because these data structures use the hash value internally.
  - All of Python's immutable built-in objects are hashable, while no mutable containers are. Objects which are instances of user-defined classes are hashable by default; they all compare unequal, and their hash value is their id().
- The set type is mutable.
- The **frozenset** type is immutable. The **frozenset** type is also hashable.
- Unlike lists and tuples, the elements in sets are not ordered.
- To process the element in a set in increasing/decreasing order, convert this set to a list or a tuple and sort it.

Set creation

In [27]:

```
# Create a set from a list/tuple. Duplicate elements in this list/tuple will be elimin
x=[1,1,2,2,3,3,3]
s=set(x)
print(x)
print(s)

# Explicitly create a set
animals = {'cat', 'dog', 'goldfish', 'canary', 'cat'}

# Create an empty set
empty_set = {} # empty_set = set()
```

```
[1, 1, 2, 2, 3, 3, 3]
{1, 2, 3}
```

Set operations

- Set difference: s1 – s2
- Set union: s1 | s2
- Set intersection: s1 & s2
- s1^s2=(s1 | s2) – (s1&s2):

In [28]:

```
even_numbers = {2, 4, 6, 8, 10}
big_numbers = {6, 7, 8, 9, 10}
#set difference: s1 - s2
print(big_numbers - even_numbers) # big numbers which are not even

#set union: s1 | s2
print(big_numbers | even_numbers) # numbers which are big or even

#set intersection: s1 & s2
print(big_numbers & even_numbers) # numbers which are big and even

# s1^s2=(s1 | s2) - (s1&s2):
# numbers which are big or even but not both
print(big_numbers ^ even_numbers)

```

```
{9, 7}
{2, 4, 6, 7, 8, 9, 10}
{8, 10, 6}
{2, 4, 7, 9}
```

frozenset

Sets cannot be used as keys in dictionaries but frozensets can. ¶

In [30]:

```
d = {frozenset({1,2,3,4}): 10, frozenset([4,5]):30}
print(d[frozenset([1,2,3,4])])
```

```
10
```

Ranges

- Ranges are immutable. Ranges are generators and often used for creating a sequence of integers.
- range(f, t, s), where f is the lower, bound t is the upper bound, and s is the step size, will create a sequence of integer numbers: f, f + s, …, f + s*(math.ceil((t-f)/s)-1). That is, range(f, t, s) includes f and excludes t.
  - range(t) is equivalent to range(0,t,1)
  - range(f,t) is equivalent to range(f,t,1)

In [31]:

```
# print the integers from 0 to 9
print(list(range(10)))

# print the integers from 1 to 10
print(list(range(1, 11)))

# print the odd integers from 1 to 10
print(list(range(1, 11, 2)))

```

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

Dictionaries

A dictionary is used to store key-value pairs.

- create an empty dictionary

```
empty_dict = dict()
empty_dict = {}
```

- create a dictionary

```
my_dict = {key1:value1, key2:value2, key3:value3}
```

- use my_dict[key1] to get value1.

If there are multiple copies of a key, this key is associated with the value of the last key-value pair of this key.

In [34]:

```
my_dict = {1:10, 2:30, 4:40, 9:-10, 1:20}
my_dict[1]+=30
print(my_dict[1])

```

```
50
```

Examples of dictionary manipulation

In [38]:

```
marbles = {"red": 34, "green": 30, "brown": 31, "yellow": 29 }
print(marbles["green"])

# This will raise a KeyError exception because there is no such a key in the dictionary
try:
    print(marbles["blue"])
except KeyError as erroneous_key:
    print('KeyError:',erroneous_key)

# Modify a value
marbles["green"] =70 # overwrite the old value
marbles["red"] += 3
marbles["purple"] = 40 # add a new key-value pair

#Check if a key is in the dictionary: key in marbles
print("yellow" in marbles)  # key not in marbles

#Check if a value is in the dictionary
print(30 in marbles.values())
```

```
30
KeyError: 'blue'
True
False
```

The types of keys can be different, and the types of values can also be different.

In [40]:

```
my_dict = {'a':[1,2,3,4], 1:(5,6,7), -10:lambda x: x+1}
print(my_dict['a'])
print(my_dict[1])
print(my_dict[-10](5))
```

```
[1, 2, 3, 4]
(5, 6, 7)
6
```

Commonly used instance methods

marbles = {"red": 34, "green": 30, "brown": 31, "yellow": 29 }

- Get a value by its key, or None if it doesn't exist

```
marbles.get("orange")
```

- We can specify a different default

```
marbles.get("orange", 0)
```

- Add several items to the dictionary at once

```
marbles.update({"orange": 34, "blue": 23, "purple": 36})
```

- All the keys in the dictionary

```
marbles.keys()
```

- All the values in the dictionary

```
marbles.values()
```

- All the items in the dictionary (key,value)

```
marbles.items()
```

- Delete an item from the dictionary

```
del marbles["red"]
```

- Delete an item from the dictionary and get the value of the item

```
value = marbles.pop("red", default)
```

- Delete all items in the dictionary

```
marbles.clear()
```

- Shallow copy of a dictionary

```
marbles.copy()
```

Conversion Between Collection Types

Iterate over a collection in a for loop to extract every item in this collection

```
for key, value in marbles.items():
    print(key, value)
```

Use the type function to cast a collection of some type to a collection of another type

Lists, tuples, and sets can convert to each other.

In [64]:

```
animals = ['cat', 'dog', 'goldfish', 'canary', 'cat']
print(animals)

animals_set = set(animals)
print(animals_set)

animals_unique_list = list(animals_set)
print(animals_unique_list)

animals_unique_tuple = tuple(animals_unique_list)
print(animals_unique_tuple)
```

```
['cat', 'dog', 'goldfish', 'canary', 'cat']
{'dog', 'goldfish', 'canary', 'cat'}
['dog', 'goldfish', 'canary', 'cat']
('dog', 'goldfish', 'canary', 'cat')
```

The keys, values, and key-value pairs in a dictionary can convert to a set, a tuple, and a list.

In [63]:

```
marbles = {"red": 34, "green": 30, "brown": 31, "yellow": 29 }
colours = list(marbles) # the keys will be used by default
print(colours)

counts = tuple(marbles.values()) # use a view to get the values
print(counts)

marbles_set = set(marbles.items()) # or the key-value pairs
print(marbles_set)
```

```
['brown', 'green', 'red', 'yellow']
(31, 30, 34, 29)
{('yellow', 29), ('red', 34), ('green', 30), ('brown', 31)}
```

Another Look at Strings

A string is a sequence of characters.

In [54]:

```
s = "abracadabra"
```

The length of a string

In [55]:

```
print(len(s))
```

```
11
```

The index of the first occurrence of a substring in a string

In [56]:

```
print(s.index("a"))
```

```
0
```

Accessing a range of characters in a string is like accessing a range of elements in a list

In [57]:

```
print(s[0])
print(s[3:5])
```

```
a
ac
```

Strings are immutable.

In [59]:

```
try:
    s[0]='a' # raise TypeError
except TypeError as type_error:
    print(type_error)
```

```
'str' object does not support item assignment
```

Membership operators can be applied to strings

In [60]:

```
print("rac" in s)
print("abc" not in s)
```

```
True
True
```

A string can be converted to a list, a tuple, or a set.

In [61]:

```
print(list(s))
print(set(s))
print(tuple(s))
```

```
['a', 'b', 'r', 'a', 'c', 'a', 'd', 'a', 'b', 'r', 'a']
{'d', 'a', 'r', 'b', 'c'}
('a', 'b', 'r', 'a', 'c', 'a', 'd', 'a', 'b', 'r', 'a')
```

A sequence of string can be joined by the join instance method.

In [53]:

```
s=['hello','Python']
print(",".join(s))
```

```
hello,Python
```

A string can be split into a list of strings by using the split method

In [52]:

```
print("cat dog fish\n".split())
print("cat|dog|fish".split("|"))
print("cat, dog, fish".split(", "))
print("cat, dog, fish".split(", ", 1))
```

```
['cat', 'dog', 'fish']
['cat', 'dog', 'fish']
['cat', 'dog', 'fish']
['cat', 'dog, fish']
```

Two-Dimensional Sequences

- A list of lists can be created and accessed in the following manner.

```
my_table = [[1,2,3,4],[5,6,7,8],[9,10,11,12]]
```

  - my_table[0] refers to the list [1,2,3,4] and print(my_table[0][0]) outputs 1.
- The inner sequences can be different in length.

```
my_2d_list=[[1,2,3,4],[5,6],[7,8,9]]
```

- A three-dimensional sequence can also be created by making a list of lists of lists.

```
my_3d_list[ [[1,2],[3,4]],
            [[5,6,7],[8,9,10]]]
```

- Create a two-dimensional table of m rows and n columns

```
my_table = [ [0]*n for _ in range(m)]
```

In [50]:

```
n, m = 4, 3
my_table = [[0]*n]*m
print(my_table)
my_table[0][1] = 1
print(my_table)
```

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

In [51]:

```
my_table = [ [0]*n for _ in range(m)]
print(my_table)
my_table[0][1] = 1
print(my_table)
```

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

The collections Module

https://docs.python.org/3/library/collections.html (https://docs.python.org/3/library/collections.html)

This module provides alternatives to Python's built-in containers, dict, list, set, and tuple.

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

---

collections.deque

A stack can be implemented by a list:

- Create an empty stack

```
stack = list()
```

- Push an element x into a stack

```
stack.append(x)
```

- Pop the top from a stack

```
x = stack.pop()
```

- Get the top of a stack

```
stack[-1]
```

In [44]:

```
stack=list([1,2,3,4])
stack.append(5) # push [1,2,3,4,5]
print(stack)
stack.pop() # pop [1,2,3,4]
print(stack)
```

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

Queues and dequeues can be implemented by collections.deque:

- Create an empty queue/dequeue

```
dq = collections.deque()
```

- Push an element x into a deque

```
dq.append(x)
dq.appendleft(x)
```

- Pop the top from a deque

```
x = dq.popleft()
x = dp.pop()
```

- Rotate a queue

```
dq.rotate(elm) # elm > 0 => rotate right, elm < 0 rotate left
```

In [46]:

```
from collections import deque

dq = deque([1,2,3,4,5])
print(dq)

dq.append(6) #[1,2,3,4,5,6]
print(dq)

dq.appendleft(7) #[7,1,2,3,4,5,6]
print(dq)

dq.pop() #[7,1,2,3,4,5]
print(dq)

dq.popleft() #[1,2,3,4,5]
print(dq)

dq.rotate(2) #[4,5,1,2,3]
print(dq)

dq.rotate(-2) #[1,2,3,4,5]
print(dq)

```

```
deque([1, 2, 3, 4, 5])
deque([1, 2, 3, 4, 5, 6])
deque([7, 1, 2, 3, 4, 5, 6])
deque([7, 1, 2, 3, 4, 5])
deque([1, 2, 3, 4, 5])
deque([4, 5, 1, 2, 3])
deque([1, 2, 3, 4, 5])
```

---

collections.namedtuple

In [42]:

```
from collections import namedtuple
# create a namedtuple

Color = namedtuple('Color', ['r', 'g', 'b'])
blue = Color(0,0,255) # Color(r=0,g=0,b=255)
print(blue.r,blue.g,blue.b)
print(blue[0],blue[1],blue[2])


# create an instance of named tuple from an iterable
it=[255,0,0]
red = Color(*it) # red = Color._make(it)

# convert to an OrderedDict
red._asdict() # OrderedDict([('r', 255), ('g', 0), ('b', 0)])

#
print(Color._fields) #('r', 'g', 'b')
print(red._fields) #('r', 'g', 'b')

```

```
0 0 255
0 0 255
('r', 'g', 'b')
('r', 'g', 'b')
```

---

collections.defaultdict

There are three possible methods for handling missing values when using dict.

In [9]:

```
data = ['red', 'blue', 'red', 'green', 'blue', 'blue']
# Method 1
d = dict()
for s in data:
    if s not in d: # check if s is in d
        d[s] = 0
    d[s] += 1

```

In [10]:

```
d = dict()
# Method 2
for s in data:
    try:
        d[s] += 1 # if s is not in d, this statement can raise an exception
    except:
        d[s] = 1
```

In [11]:

```
d = dict()
# Method 3
for s in data:
    c = d.get(s,0) # get the value associated with s
    d[s] = c+1
```

defaultdict is a dict subclass that calls a factory function to supply missing values.

In [12]:

```
# Using defaultdict

from collections import defaultdict

#d = defaultdict(int)
d = defaultdict(lambda : 0) # this argument must be callable or None

for s in data:
    d[s] += 1
```

---

collections.Counter

collections.Counter is a dict subclass for counting hashable objects.

In [13]:

```
from collections import Counter
# Tally occurrences of words in a list
data = ['red', 'blue', 'red', 'green', 'blue', 'blue']

#Method 1:
cnt1 = Counter()
for word in data:
    cnt1[word] += 1

#Method 2:
cnt2 = Counter(data)

print(cnt1,cnt2)

top1 = cnt1.most_common(1)
top3 = cnt1.most_common(3)
print(top1,'key:{}, value:{}'.format(top1[0][0],top1[0][1]))
print(top3)
```

```
Counter({'blue': 3, 'red': 2, 'green': 1}) Counter({'blue': 3, 'red': 2, 'gr
een': 1})
[('blue', 3)] key:blue, value:3
[('blue', 3), ('red', 2), ('green', 1)]
```

collections.ChainMap

collections.ChainMap is a dict-like class for creating a single view of multiple mappings.

In [14]:

```
# use the update instance method to combine multiple dicts into one dict
d1={'a':1,'b':2,'c':3}
d2={'e':2,'a':4,'d':5}
d={}
d.update(d1)
d.update(d2)
```

In [15]:

```
# use the ChainMap to combine multiple dicts into one dict

from collections import ChainMap

f = ChainMap(d1,d2,d)

# gets 1 # search d1 for 'a' if not found, then search d2 for 'a', …
print(f['a'])

# because 'g' is not in d1, d2, and d, 'g' will be inserted into d1, and d1['g'] is eq
f['g'] = 99
print(f['g'],d1['g'])
```

```
1
99 99
```

In [ ]:

```

```