**Lists, Tuples and Sets**

lunes, 24 de octubre de 2022

4:35 p. m.

Lists

Source: [Python Tutorial for Beginners 4: Lists, Tuples, and Sets](https://www.youtube.com/watch?v=W8KRzm-HUcc&list=PL-osiE80TeTskrapNbzXhwoFUiLCjGgY7&index=4)

Documentation: <https://docs.python.org/3/tutorial/datastructures.html?highlight=lists>

Some Lists methods

*List*.**append**( *i* ): adds the *i* element to the end of the list, making it the list last element.

*List*.**insert**( *index, i* ): adds the *i* element to a specific position (*index*) within the list.

*List*.**extend**( *iterable* ): works the same as append but with an iterable. It adds all the items within an iterable to the *list.*

*List*.**remove**( *i* ): looks up for the *i* within the list and take it out.

*List*.**pop**( *x* ): works similar to the remove method but the argument is optional. If no argument is passed to the method it will take off the last item within the list. The argument is supposed to be the position of the element popped out of the list.

This method return the item specified and affects the list object:

l = ['A', 'B', 'C', 'D']

print(l.pop()) = D

And, here's what happen if we just call the method and then print the list object:

l = ['A', 'B', 'C', 'D']

l.pop()

print(l) = ['A', 'B', 'C']

*List*.**reverse**( ): This method reverse the order of the list.

*List*.**sort***( key=None, reverse=False* ): Will automatically sort the order of the list in a ascending order alphabetically. The order could be shifted to descending by setting the keyworded argument *'reverse'* to **True***.* This method will alter the object, different from the **sorted**( ) built-in function.

list.**clear**( ): This method remove all items from the list.

list.**index**( *x[, start[, end]]* ): Works similar to str.index(), and return the index of *the x* argument within the list, from left to right. The optional arguments *start* and *end*, work as starting and ending point to execute the search for the argument *x.* A ValueError will be raised if there is no such item in the list or the slice. The end index, as is with the slices, is non-inclusive

l = ['A', 'B', 'C', 'D']

print(l.index('C')) = 2

with *start* & *end*:

l = ['A', 'B', 'C', 'D', 'F', 'R', 'C']

print(l.index('C', 3)) = 6

Or

print(l.index('C',l.index('C')+1)) = 6

list.**count**( *x* ): Same as the str.count(), return the number of repetitions of the argument *x*

list.**copy**( ): returns a shallow copy of the *list*.

5.1.3. List Comprehensions

To better understand ListComps: [Python Tutorial: Comprehensions - How they work and why you should be using them](https://www.youtube.com/watch?v=3dt4OGnU5sM)

Basic listcomp example:

squares = [x\*\*2 for x in range(10)]

print(squares) = [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

Another example:

listcomp = [ ( x, y ) for x in [1,2,3] for y in [3,1,4] if x != y ]

print(listcomp) = [(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]

Flatten a list using listcomp with two 'for's listcomp:

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

new\_list = [num for elem in vec for num in elem]

print(new\_list) = [1, 2, 3, 4, 5, 6, 7, 8, 9]

5.1.4. Nested List Comprehensions

Note: Matrices and Arrays are different. A Matrix is a special case of a two dimensional array that holds same sized elements. Whereas, Arrays can hold a fix number of items but could be multidimensional but strictly of the same size.

A Nested list is basically a list that hold others lists (of elements but by convention same sized lists). For this examples, a Matrix of 3 x 4, will serve as base:

matrix = [

    [1, 2, 3, 4],

    [5, 6, 7, 8],

    [9, 10, 11, 12],

]

With a listcomp a transposition of row and columns could be made with the following syntax

transposed = [[row[i] for row in matrix] for i in range(4)]

print(transposed) =

[ [1, 5, 9],

[2, 6, 10],

[3, 7, 11],

[4, 8, 12] ]

Which will be basically the same as to make a nested for loop

transposed = []

for i in range(4):

    transposed\_row = []

    for row in matrix:

        transposed\_row.append(row[i])

    transposed.append(transposed\_row)

print(transposed) =

[ [1, 5, 9],

[2, 6, 10],

[3, 7, 11],

[4, 8, 12] ]

Finally the python documentation recommends to use built-in functions to complex flow statements like the last one. For this use case the zip( ) function would do the job:

transposed = list(zip(\*matrix))

print(transposed) =

[ [1, 5, 9],

[2, 6, 10],

[3, 7, 11],

[4, 8, 12] ]

During learning 'List comprehension' I bumped with the idea of Arrays (Matrixes) and List comprehension (lists of lists).

The difference is that in a List Comprehension, either built with a for loop or the list comprehension statement, are just simply a big list containing sublists of n-elements each one.

But an Array or Matrix is something similar but with a difference:

When an array is stated, either if its for an 2 or 3 dimensions, the algorithm is pretty similar, a temporary list is created within the for each loop of the first dimension, and it'll contain in order the n-dimensions right at the end of the n-th element, to later be appended to the resulting list, at the end of the loop of the first element.

Meaning that this will be a list containing 1st Element Number of list that contains list with the dimension ordered from higher to lower. To simplify: A list of lists of lists with data

Is way different to populate an Array vs populate like a list comprehension, even with for loop statements:

    #Populating an Array:

    array=[]

    for row in array:

        temp=[]

        for col in row:

            temp.append([row, col])

        array.append([temp])

    #populating like List Comprehension:

    array=[]

    for row in array:

        for col in row:

            array.append([row, col])

5.2. The del statement

There is a way to take out an element of a list without knowing the content but with the index instead of *the list*.remove() and *list*.pop() methods.

The del statement work as follows:

a = [-1, 1, 66.25, 333, 333, 1234.5]

del a[0]

print(a) = [1, 66.25, 333, 333, 1234.5]

del a[2:4]

print(a) = [1, 66.25, 1234.5]

del a[:]

print(a) = []

List built-in methods

**print**(**dir**(list)) = [

'\_\_add\_\_', '\_\_class\_\_', '\_\_class\_getitem\_\_', '\_\_contains\_\_', '\_\_delattr\_\_',

'\_\_delitem\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_getitem\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_iadd\_\_', '\_\_imul\_\_', '\_\_init\_\_', '\_\_init\_subclass\_\_', '\_\_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'

]

Tuples

Source: [Python Tutorial for Beginners 4: Lists, Tuples, and Sets](https://www.youtube.com/watch?v=W8KRzm-HUcc&list=PL-osiE80TeTskrapNbzXhwoFUiLCjGgY7&index=4)

Documentation: <https://docs.python.org/3/tutorial/datastructures.html?highlight=lists>

Tuples are datatypes pretty similar to lists but the difference is that are **immutable**.

Immutable datatypes are very useful when no side effects are desired. Let's compare cases:

l = ['A', 'B', 'C', 'D', 'F', 'R', 'C']

l2 = l

print(f'{l}\n{l2}') =

['A', 'B', 'C', 'D', 'F', 'R', 'C']

['A', 'B', 'C', 'D', 'F', 'R', 'C']

l[0] = '7'

print(f'{l}\n{l2}')

=

['7', 'B', 'C', 'D', 'F', 'R', 'C']

['7', 'B', 'C', 'D', 'F', 'R', 'C']

Here, as one item is modified in the original list "l", the second depending list "l2" get modified as well and then it could be problematic if later one or the other are needed to be consulted and any variation on the first would affect the second.

Then, a tuple would carry the same functionalities that of a list excepting mutating the object. So for storing and accessing data are a good fit.

l2 = tuple(l)

l[0] = '7'

print(f'{l}\n{l2}') =

['7', 'B', 'C', 'D', 'F', 'R', 'C']

('A', 'B', 'C', 'D', 'F', 'R', 'C')

Tuples can be created either by calling a sequence enclosed with a parenthesis or actually without them.

my\_tuple = 'word', 114, True

my\_tuple = ('word', 114, True)

A multiple assignment with tuples could be made if there are the same amount of variables matching the number of items within a tuple. Otherwise a ValueError would be raised. This is called *"sequence unpacking".*

\*Note: this multiple assignment works on **any sequence.**

a, b, c = my\_tuple

print(a) = word

print(b) = 114

Print(c) = True

Since tuples are immutable objects, is not possible to modify them, but is possible to add or remove items in a different way, merging and slicing tuples:

# Merging tuples

tuple1 = (1, 2, 3, 4)

tuple1 = tuple1 + (9,)

print(tuple1) = (1, 2, 3, 4, 9)

Technically this is not a modification of the tuple, is a rewritting of the tuple. Is basically create a tuple from another one, and the same goes to the subtraction of the tuple:

# Subtracting tuples

tuple1 = (1, 2, 3, 4)

tuple1 = tuple1[:-1]

print(tuple1) = (1, 2, 3)

# Adding items in the middle of the tuple

tuple1 = tuple1[:2] + ('A', 'B', 'C') + tuple1[2:]

print(tuple1) = (1, 2, 'A', 'B', 'C', 3, 4)

There is not tuple comprehension as naturally happens with Listcomp, Setcomp or Dictcomp, if one would try to use the same syntax a generator would be created, which is a totally different topic:

l = [ x for x in range(5) ]

print(l) = [0, 1, 2, 3, 4]

t = ( x for x in l ) # Not a tuple, but a generator object instead

print(t) = <generator object <genexpr> at 0x000002B21FBF0BA0>

But with adding the preceding " tuple " type before the syntax, the same result as Listcomp, Setcomp or Dictcomp is achieved.

input\_string = 'python 3.0'

result = tuple( elem for elem in input\_string )

print(result) = ('p', 'y', 't', 'h', 'o', 'n', ' ', '3', '.', '0')

print(type(result)) = <class 'tuple'>

Some tuple methods

*Tuple*.**count**( *i* ): works the same as *List*.**count**( *i* ) .

*Tuple*.**index**( *x[, start[, end]]* ): works the same as *List*.**index**( *x[, start[, end]]* )

Namedtuple( )

Source: <https://www.geeksforgeeks.org/namedtuple-in-python/>

[Python Tutorial: Namedtuple - When and why should you use namedtuples?](https://www.youtube.com/watch?v=GfxJYp9_nJA)

*"Python’s namedtuple was created to improve code readability by providing a way to access values using descriptive field names instead of integer indices, which most of the time don’t provide any context on what the values are. This feature also makes the code cleaner and more maintainable."*

Source: <https://realpython.com/python-namedtuple/#:~:text=Python's%20namedtuple%20was%20created%20to,code%20cleaner%20and%20more%20maintainable>.

Python's namedtuple() is a factory function available in collections module. It allows you to create tuple subclasses with named fields. You can access the values in a given named tuple using indices (which dictionaries don't allow to) and the field names, like a dictionary.

Namedtuples are the halfway between a tuple and a dictionary.

Let's say:

from collections import namedtuple

# Declaring the namedtuple()

Student = namedtuple('Student', ['name', 'age', 'ID'])

# Adding values

S = Student('Gerardo', 29, '3934')

# Accessing using index

print(f"The Student age using index is : {S[1]}") = The Student age using index is : 29

# Accessing using an attribute

print(f"The Student name using keyname is : {S.name}") = The Student name using keyname is : Gerardo

# It also supports the getattr()

print(f"The Student ID using getattr() is : {getattr(S, 'ID')}") = The Student ID using getattr() is : 3934

This namedtuples also support the creation of an instance with *object*.\_make( *iterable* ).

Namedtuples could return a OrderedDict( ) with the mapped values from a namedtuple instance with the *namedtuple*.\_asdict( ).

And with the double star ' \*\* ' operator an instance of a namedtuple could be created if takes a dictionary as argument:

# # Using the \_make(), \_asdict() and \*\* operator

# initializing an iterable

li = ['Juan', 30, '7825']

# using \_make() to return namedtuple()

print(f'The namedtuple instance using iterable is : {Student.\_make(li)}') =

The namedtuple instance using iterable is : Student(name='Juan', age=30, ID='7825')

# initializing dictionary

di = {'name':'Juan', 'age': 30, 'ID': '7825'}

# using \_asdict() to return an OrderedDict()

print(f'The OrderedDict instance using namedtuple is : {S.\_asdict()}') =

The OrderedDict instance using namedtuple is : {'name': 'Gerardo', 'age': 29, 'ID': '3934'}

# using \*\* operator to return namedtuple from dictionary

print(f'The namedtuple instance from dict is : {Student(\*\*di)}') =

he namedtuple instance from dict is : Student(name='Juan', age=30, ID='7825')

Namedtuples also has a \_fields attribute that display all the attributes within the instance and the Object.

\_replace( *attribute = 'new value'* ) would create a new instance with the replacement but leaving the original unaffected.

# # Additional operations: \_fields and \_replace()

# Using \_field attribute to display all the keynames of the namedtuple()

print(f'All the field of students are : {S.\_fields}') =

All the field of students are : ('name', 'age', 'ID')

# Using \_replace() to it would create a new namedtuple and will leave the original intact

print('returns a new namedtuple :')

print(S.\_replace( name = 'Juan'))

=

returns a new namedtuple : Student(name='Juan', age=29, ID='3934')

# Original namedtuple instance

print(S) = Student(name='Gerardo', age=29, ID='3934')

Tuple built-in methods

**print**(**dir**(tuple)) = [

'\_\_add\_\_', '\_\_class\_\_', '\_\_class\_getitem\_\_', '\_\_contains\_\_', '\_\_delattr\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_getitem\_\_', '\_\_getnewargs\_\_', '\_\_getstate\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_init\_subclass\_\_', '\_\_iter\_\_', '\_\_le\_\_', '\_\_len\_\_', '\_\_lt\_\_',

'\_\_mul\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_',

'\_\_rmul\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_',

'count', 'index'

]

Sets

Source: [Python Tutorial for Beginners 4: Lists, Tuples, and Sets](https://www.youtube.com/watch?v=W8KRzm-HUcc&list=PL-osiE80TeTskrapNbzXhwoFUiLCjGgY7&index=4)

Documentation: <https://docs.python.org/3/tutorial/datastructures.html?highlight=lists>

Sets are values grouped together with no specific order and have no duplicates. The syntax to create a set is to wrap the elements up within curly bracers "{ }".

Note: To create an empty set, just calling a variable with an empty curly bracers would create an empty dictionary instead. The correct way to do it would be to use the set function generator set()

All element within a set must be hashables, that implies that a set cannot be created with another set() inside

set1 = {} #this creates an empty dictionary

print(type(set1)) = <class 'dict'>

set2 = set()

print(type(set2)) = <class 'set'>

Sets also support comprehensions:

import random

set3 = {random.randint(1,20) for i in range(20)}

print(set3) = {2, 3, 4, 5, 7, 8, 9, 10, 11, 13, 14, 17, 20}

When printed, set may show its elements in a different order to what the elements were input, and this is because this datatype is meant to test 'Membership', meaning if an element is whether within a set.

Sets do this whether an element is within a group better compared to list or tuples, since sets rule out duplicates and therefore will have to check less items in the test.

And even with identical elements the interpreter would run twice as fast through a set compared with a list.

If an element is input twice or more times in the set, when called the set would only hold once the element.

Some set methods

*Set*.**intersection**( *set* ): this method would return a set with the shared elements between the two sets.

cs\_courses = {'History', 'Math', 'Physics', 'CompSci'}

art\_courses = {'History', 'Math', 'Chemistry', 'Art'}

print(cs\_courses.intersection(art\_courses)) = {'Math', 'History'}

*Set*.**intersection\_update**( ): this method does not return a set, returns None. But instead updates the base set with the actual intersection between it and the argument set.

Is equivalent to the syntax cs\_courses = cs\_courses.intersection(art\_courses).

cs\_courses.intersection\_update(art\_courses)

print(cs\_courses) = {'History', 'Math'}

---

new\_set = cs\_courses.intersection\_update(art\_courses)

print(new\_set) = None

*Set*.**difference**( *set* ): this method returns a set with the elements present in the base set but not in the argument.

print( cs\_courses.difference(art\_courses) ) = {'Physics', 'CompSci'}

*Set*.**difference\_update**( *set* ): this method does not return a set, returns None. but instead updates the base set with the actual difference between it and the argument set.

Is equivalent to the syntax cs\_courses = cs\_courses.difference(art\_courses).

cs\_courses.difference\_update(art\_courses)

print(cs\_courses) = {'Physics', 'CompSci'}

---

new\_set = cs\_courses.difference\_update(art\_courses)

print(new\_set) = None

*Set*.**union**( *set* ): this method would merge the two set into a new set.

print(cs\_courses.union(art\_courses)) = {'Art', 'History', 'Chemistry', 'Math', 'CompSci', 'Physics'}

*Set*.**add***( elem* ): This method works similar to *list*.append(), it just adds a new element to the set, but if the element already exist within the set it doesn't alter the set. The method does not return a new set, if a new set is created and something added to it, it will return None.

cs\_courses.add('Literature')

print(cs\_courses) = {'Math', 'Physics', 'CompSci', 'History', 'Literature'}

---

new\_set = set().add('element')

print(new\_set) = None

Or

cs\_courses = cs\_courses.add('Literature')

print(cs\_courses) = None

*Set*.**clear**( ): This method does not take any arguments and simply clears out the whole set.

cs\_courses.clear()

print(cs\_courses) = set()

*Set*.**copy**( ): This method does not take any arguments and simply creates a copy of the whole set.

*Set*.**discard***( elem* ): This method takes one argument, the one wanted to be removed from the set and returns a None. If the element is not present in the set, the method leaves it as it is.

*Set*.**isdisjoint***( iter* ): This method is a validator between a set and any other iterable to check if there are any element present in both the base set and the iterable. If it does, then False is returned, otherwise True is returned.

Note that the last validation throws x because even when 'a' is present in the dictionary, it does not as a key, making it disjoint to new\_set.

new\_set = {'a', 'b', 'c', 'd', 'e'}

new\_list = ['e', 'f', 'g', 'h']

dict1 = {'a' : 1, 'f' : 2, 'h' : 4}

dict2 = {1 :'a', 2 : 'f'}

print(f'{new\_set.isdisjoint(new\_list)}') = False

print(f'{new\_set.isdisjoint(dict1)}') = False

print(f'{new\_set.isdisjoint(dict2)}') = True

*Set*.**issubset**( iter ): This method is a validator between a set and any other iterable to check if all elements present in base set are present in the iterable. If they do, then True is returned, otherwise False is returned.

set1 = {'a', 'b', 'c', 'd', 'e'}

set2 = {'b', 'c'}

list1 = {'b', 'c'}

dict1 = {'b' : 1, 'c' : 2}

print(set2.issubset(set1)) = True

print(set2.issubset(list1)) = True

print(set2.issubset(dict1)) = True

*Set*.**issuperset**( ): This method is a validator between a set and any other iterable to check if all elements present in the argument are also included in the base set. If they do, then True is returned, otherwise False is returned.

print(set1.issuperset(set2)) = True

print(set1.issuperset(list1)) = True

print(set2.issuperset(set1)) = False

*Set*.**pop**( ): This method works the same *as list.*pop() and does not receive any parameters returning a random element of the set and modify the set by taking the element out. If the set to be popped is empty a KeyError will raise.

print(set1.pop()) = b

print(set1) = {'d', 'e', 'a', 'c'}

*Set*.**remove***( elem* ): This method works the same *as list.*remove(). remove() takes a parameter and it'll modify the set by taking the element out. The method returns None If the element to be removed isn't in the set KeyError will raise, and to not get this error is recommended to use *set*.discard( *elem* )

But why is there two methods that does the same?

Took from Stackoverflow:

"*Errors are raised to be caught and processed. They are not annoyances or hurdles. They are tools to identify conceptual errors, or to indicated unexpected behavior that one needs to pay attention to, or to deal with parts of the system that one does not have control over, or to use to control the flow of the code where the python doctrine says „fail rather than test“ i.e. let the code raise exceptions you expect rather than testing with if statements.*

*In the case of .discard() and .remove(): .discard() calls .remove() silently catch the exception in case the value was not there and silently returns. It’s a shortcut for a silent .remove(). It might be suitable for your special use-case. Other use-cases might require an exception to be raised when the value does not exist.*

*So .remove() is the general case that gives the developer control over the exception and .discard() is just a special use case where the developer does not need to catch that execration.*"

*Set*.**symmetric\_difference**( *iter* ): this method works the same as *Set*.**difference**( *iter* ) with the only difference that the returned set also includes the element not present both in the base and the argument.

print(cs\_courses.symmetric\_difference(art\_courses)) = {'Art', 'Physics', 'CompSci', 'Chemistry'}

*Set*.**symmetric\_difference\_update**( *iter* ): And this last one works the same as *Set*.**difference\_update**( *iter* ) with the same feature that *Set*.**symmetric\_difference**( *iter* )

*Set*.**update***( iterables* ): this method works sort of similar to *Set*.**add**( *elem* ) with the difference that update() can add multiple sequences or iterables to the set.

**Note**: in the case a dictionary is updated to a set, the method would include the keys of the dictionary as new set elements.

Frozensets( )

Documentation: <https://docs.python.org/3/library/stdtypes.html?highlight=frozenset#frozenset>

As sets are sequences or iterables that are mutable, the Frozensets are the immutable version of them.

They work pretty much as sets with the difference that can not be changed once created, therefore, methods like add, clear, discard, pop or remove are not supported by frozensets.

Frozenset as are immutable, are also hashable, from there that can contain other frozensets as well.

Set built-in methods

**print**(**dir**(set)) = [

'\_\_and\_\_', '\_\_class\_\_', '\_\_class\_getitem\_\_', '\_\_contains\_\_', '\_\_delattr\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_getstate\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_iand\_\_', '\_\_init\_\_', '\_\_init\_subclass\_\_', '\_\_ior\_\_', '\_\_isub\_\_', '\_\_iter\_\_', '\_\_ixor\_\_', '\_\_le\_\_', '\_\_len\_\_', '\_\_lt\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_or\_\_', '\_\_rand\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_ror\_\_', '\_\_rsub\_\_', '\_\_rxor\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_sub\_\_', '\_\_subclasshook\_\_', '\_\_xor\_\_',

'add', 'clear', 'copy', 'difference', 'difference\_update', 'discard', 'intersection', 'intersection\_update', 'isdisjoint', 'issubset', 'issuperset', 'pop', 'remove', 'symmetric\_difference', 'symmetric\_difference\_update', 'union', 'update'

]