**Dictionaries**

martes, 8 de noviembre de 2022

6:03 p. m.

Dictionaries Fundamentals

Source: [Python Tutorial for Beginners 5: Dictionaries - Working with Key-Value Pairs](https://www.youtube.com/watch?v=daefaLgNkw0&list=PL-osiE80TeTskrapNbzXhwoFUiLCjGgY7&index=5)

Documentation: <https://docs.python.org/3/tutorial/datastructures.html?highlight=Dict> (5.5)

<https://docs.python.org/3/library/stdtypes.html?highlight=Mapping%20Types> - Mapping types - Dict (Near to the 3rd third)

A Dictionary could be understood as an array of a Key : Value pair array, and in other languages could be found as "Associative Memories", "Associative Arrays" or "Hash Maps".

In dictionaries the ancor value is always the Key meaning that many keys could have the same Value but never different values could have the same Key.

The syntax to make a dictionary is by assigning to a variable empty curly braces ( { } ) or by the dictionary constructor dict().

For the Keys they could take any immutable name to be identified, meaning that strings and integers will always be valid key names. Tuples could also be keys only if mutables are not within the tuple directly or indirectly. *This makes sense since if a object is not the same, it could bring troubles when being searched within the dictionary for a value.*

student = {'name': 'John', 'age': 25, 'courses': ['Math', 'CompSci']}

print(student['age']) = 25

If a Key is accessed but is not present within the dictionary, a KeyValueError will appear, and this could be important to control input and exepctions in some cases, but in the cases we don't need to keep track of them, using the dictionary method .**get**( *key*, *default return* ) we receive a None if the key doesn't exist or any other message passed in the second (optional) argument of the method.

print(student['phone']) = KeyValueError!

print(student.get('phone')) = None

print(student.get('phone', "Not Found!")) = Not Found!

To add a new Key to the dictionary is easy as declare it with the following syntax:

student['phone'] = '555-5555'

print(student) = {'name': 'John', 'age': 25, 'courses': ['Math', 'CompSci'], 'phone': '555-5555'}

print(student.get('phone')) = 555-5555

And if the same is done but with an existing Key, it get overwritten withing the dictionary, losing the former value of that Key.

student['name'] = 'Jane'

print(student) = {'name': 'Jane', 'age': 25, 'courses': ['Math', 'CompSci'], 'phone': '555-5555'}

Now, to update multiple values, the .**update**(*[other]*) method is useful. It could take any iterable arranged in pairs and it overwrites the value of the keys passed. As argument it could receive new dictionaries or tuples as well. It returns None, so it should not be assign to anything but just called instead.

student.update({'name': 'Gerardo', 'age': 29, 'courses': ['Arts', 'Socials'], 'gender' : "men"})

print(student) = {'name': 'Gerardo', 'age': 29, 'courses': ['Arts', 'Socials'], 'gender': 'men'}

To remove items from the dictionary, the del statement work as it does with lists.

del student['gender']

print(student) = {'name': 'Gerardo', 'age': 29, 'courses': ['Arts', 'Socials']}

When loop though a dictionary, we are actually going through the Keys and this is why when **len**( ) applied to a dictionary it will return the number only the keys within it.

print(len(student)) = 3

for elem in student:

    print (elem) =

name

age

courses

Now, if the would like loop though the dictionary including keys and value, then the **item**( ) method is needed, it will return a list of tuples of 2 element ordered Key, Value like:

for key, value in student.items():

    print(key, value) =

name John

age 25

courses ['Math', 'CompSci']

if **list**( )is casted into a dictionary, a list with the keys will be returned:

print(list(student)) = ['name', 'age', 'courses']

Some other dict methods

*Dict*.**clear**( *i* ): Removes all items from the dictionary

*Dict*.**copy**( *i* ): Returns a shallow copy of the dictionary

*Dict*.**fromkeys**( *iter, value* ): This Class method creates a new dictionary with the keys passed in the first argument, and assign the **same** value in the second (optional) argument. The risk with this method is that is the base value, as it could be mutable, changes, all the keys will be affected, since all the values would be pointing to the same memory location. If no *value* argument is passed, None will be assigned by default.

*If this is desired to be avoided, the recommendation is to use DictComprehension.*

li = ['a', 'b', 'c']

val = ['x']

new\_dict = dict.fromkeys(li, val)

print(new\_dict) = {'a': ['x'], 'b': ['x'], 'c': ['x']}

val.append('y')

print(new\_dict) = {'a': ['x', 'y'], 'b': ['x', 'y'], 'c': ['x', 'y']} Collateral effects

*Dict*.**pop**( *i* ): Works the same as *List*.**pop**( *i* ) but only returning the key and taking it put of the base dict, with the difference that it will raise a TypeError if no argument is passed.

*Dict*.**popitem**( ): This method does not receive parameters and returns the pair Key : Values in LIFO order.

*Dict*.**setdefault**( *Key, default\_value* ): This method returns a value of the key passed in the first argument. If the Key is not in the dictionary, by default a new key is created in the dictionary to return the *default\_value* passed in the second (optional) argument.

If no *default\_value* is passed, None will be the default value to be passed.

person = {'name': 'Phill', 'age': 22}

print(person.setdefault('name')) = Phill

print(person.setdefault('Gender', 'Men'), person) = Men {'name': 'Phill', 'age': 22, 'Gender': 'Men'

print(person.setdefault('Sign'), person) = None {'name': 'Phill', 'age': 22, 'Gender': 'Men', 'Sign': None}

OrderedDict()

Source: <https://docs.python.org/3/library/collections.html?highlight=ordereddict#collections.OrderedDict>

Ordered dictionaries are a subclass of the dictionary class and can be found in the collections module.

The main concept of ordered dictionaries are dictionaries that cares for and remember the order in which the element are indexed. This objects can support all the methods from dictionaries and this are the three main differences with dictionaries:

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

Different from regular dictionaries, this method returns and remove the key, value pair according to FIFO (Not LIFO like their counterparts) if the arugment *last* is set to False.

from collections import OrderedDict

reg\_d = {'a': 1, 'b': 2, 'c': 3}

d = OrderedDict()

d['a'] = 1

d['b'] = 2

d['c'] = 3

print(reg\_d) = {'a': 1, 'b': 2, 'c': 3}

pop\_rd = reg\_d.popitem()

print(pop\_rd, reg\_d) = ('c', 3) {'a': 1, 'b': 2}

print(d) = OrderedDict([('a', 1), ('b', 2), ('c', 3)])

pop\_d = d.popitem(last=False)

print(pop\_d, d) = ('a', 1) OrderedDict([('b', 2), ('c', 3)])

* OrderedDict.move\_to\_end*( key*, *last = True* ):

This method would take the key passed in the first argument and move it to the rightmost of the collection if the second optional argument is set to True *(default*) and to the leftmost of the collection if False.

print(d) = OrderedDict([('a', 1), ('b', 2), ('c', 3)])

d.move\_to\_end('b', last=True)

print(d) = OrderedDict([('a', 1), ('c', 3), ('b', 2)])

d.move\_to\_end('b', last=False)

print(d) = OrderedDict([('b', 2), ('a', 1), ('c', 3)])

* Comparing dicts and OrderedDicts

Esentially to compare dicts order does not affect the truth value, while all the keys, values pairs are present in both variables, the comparison will return True. For OrderedDicts the comparison is order sensitive.

reg\_d = {'a': 1, 'b': 2, 'c': 3}

reg\_p = {'c': 3, 'a': 1, 'b': 2}

d = OrderedDict()

d['a'] = 1

d['b'] = 2

d['c'] = 3

p = OrderedDict()

p['c'] = 3

p['a'] = 1

p['b'] = 2

print(reg\_d == reg\_p) = True

print(d == p) = False

Dictionary Comprehensions

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

# Dictionary Comprehensions

names = ['Bruce', 'Clark', 'Peter', 'Logan', 'Wade']

heros = ['Batman', 'Superman', 'Spiderman', 'Wolverine', 'Deadpool']

dict1 = dict()

for i in range(len(names)):

    dict1[names[i]] = heros[i]

dict1 = { names[i]: heros[i] for i in range(len(names))  } # This is one way to do it

dict1 = { name : hero for name, hero in zip(names, heros)  } # This is another way to do it

print(dict1) = {'Bruce': 'Batman', 'Clark': 'Superman', 'Peter': 'Spiderman', 'Logan': 'Wolverine', 'Wade': 'Deadpool'}

Dictionary built-in methods

**print**(**dir**(dict)) = [

'\_\_class\_\_', '\_\_class\_getitem\_\_', '\_\_contains\_\_', '\_\_delattr\_\_', '\_\_delitem\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_getitem\_\_', '\_\_getstate\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_init\_subclass\_\_', '\_\_ior\_\_', '\_\_iter\_\_', '\_\_le\_\_', '\_\_len\_\_', '\_\_lt\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_or\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_reversed\_\_', '\_\_ror\_\_', '\_\_setattr\_\_', '\_\_setitem\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_',

'clear', 'copy', 'fromkeys', 'get', 'items', 'keys', 'pop', 'popitem', 'setdefault', 'update', 'values'

]