Inspiring Excellence

Course Title: Programming Language II

Course Code: CSE 111

Semester: Summer 2020

Topic: Introduction to Dictionary and Tuples

Table of Contents

Dictionary

A dictionary is an unordered collection of items that consists of **key-value** pairs. Dictionary is mutable which means it can be changed. It contains comma-separated collection of key-value pairs which are enclosed within curly braces (**{}**). A **key** is separated from its associated **value** by a colon (**:**). The **values** in dictionary are accessed by **keys**. An empty dictionary (without any items) can be defined with just curly braces (**{}**). The **values** in the dictionary can be of any data type and necessarily do not have to be unique, values can be duplicated, however **keys** must be immutable and they have to be unique. **Keys** in dictionary are case sensitive.

Creating python Dictionary

We can create a dictionary by placing the sequence of comma-separated elements within a curly braces (**{}**). The elements can be of different types such as string, float, integer etc. The element is basically a **key: value** pair.

```
# empty dictionary
my_dict ={}

# Dictionary of integer keys
my_dict = {101: ‘harry’, 202: ‘ron’, 303: ‘fred’}

# Dictionary of string keys
my_dict = {'one': 'une', 'two': 'deux', 'three': 'trois'}
```

```
# dictionary with mixed keys
my_dict = {'fruit': 'apple', 100: [101, 102, 103, 104]}

my_dict2 = {'name': ‘romeo', 23: ‘age’}
```

A dictionary can also be created by using **dict()** which is a built-in function.

```
# using dict()
my_dict = dict({101: ‘harry’, 202: ‘ron’, 303: ‘fred’})

my_dict2 = dict({'one': 'une', 'two': 'deux', 'three':
'trois'})

# from sequence having each item as a pair

my_dict = dict([('fruit1','apple'),
('fruit2','mango'),('fruit3','guava')])
```

Nested Dictionaries

Nested dictionary means a dictionary inside a dictionary, a collection of dictionaries in one single dictionary.

```
# Example 1
house = {'gryffindor': {'name': 'ginny', 'age': '11', 'sex':
'female'},
         'slytherin': {'name': 'draco', 'age': '10', 'sex':
'male'},
         'ravenclaw':{'name': 'luna', 'age': '12', 'sex':
'female'}}
```

```
# Example 2
book1 = {'genre': 'sci-fi', 'writer': 'isaac asimov'}
book2 = {'genre': 'horror', 'writer': 'stephen king'}
books = {'book1': book1, 'book2': book2}
print(books)
```

Accessing Elements from Dictionary

We can access the elements of a dictionary by addressing its **key** within square brackets (**[]**). Also, we can use **get()** method that generates the same output.

```
# get vs [] for retrieving elements
price = {'price1': 450, 'price2': 760}
print(price['price1'])
# Output: 450
print(price.get('price2'))
# Output: 760


country_book = {'country': 'USA', 'state': 'illinois'}
print(country_book['country'])
# Output: USA
print(country_book.get('state'))
# Output: illinois
```

However, if a key is not found in the dictionary, it will produce a KeyError when you using square brackets. In case of **get()** method, if the key if not found it will show None as output.

```
print(price.get('price4'))
# Output: None
print(price['price4'])
# Output: KeyError: 'price4'


print(country_book.['population'])
# Output: KeyError: 'population'
print(country_book.get('population'))
# Output: None
```

Dictionary Key Restrictions

You can use almost any type of data as a key in python dictionary. However, there are some restrictions you must follow while creating a key.

A key cannot be repeated or duplicated. You cannot give same name to more than one keys in a dictionary. When you define a key which already exists in dictionary, the second one will replace the first one which means it overrides the initial one.

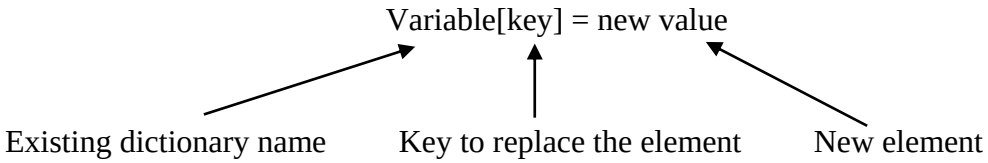
A dictionary key must be of immutable type such as integer, float, string or Boolean. But a list or another dictionary itself cannot serve as a key because we know lists and dictionaries are mutable.

```
house= {'gryffindor': 'ron'}
house['gryffindor'] = 'nevil' #It will replace existing value
house = {'gryffindor': 'ron','gryffindor': 'nevil'}
#it is replacing the previous value, not adding both of the keys


foods = {['apple', 'mango']: 'fruit', ['coke', 'pepsi']:
'drink', ['pizza', 'pasta']: 'italian food', ['cake', 'donut']:
'bakery'}
print(foods)
TypeError: unhashable type: 'list'
```

Dictionary Mutability

Like lists, dictionaries are mutable. We can easily change the value of existing items, add new items and delete or remove items. The structure for updating existing element is:



Example:

```
kindergarten = {'class': 'kg', 'section': 5, 'students': 100}
kindergarten['section'] = 7          # update existing entry
kindergarten['teacher'] = 'neilson'  # Add new entry
del kindergarten ['students']        # remove entry with key
kindergarten.clear();                # empties the dictionary
del kindergarten;                    # delete entire dictionary
kindergarten.pop('class')            # delete entry with key
```

Update items of a Dictionary

If we want to add a new element to the dictionary, we can do it by using a new key and providing value to it. You can also change an existing value with the corresponding key.

```
movie = {'Name': 'inception', 'director': 'nolan', 'release':
2012}
movie['release'] = 2010;           # update existing entry
print(movie)
# output:
{'Name': 'inception', 'director': 'nolan', 'release': 2010}

movie['genre'] = 'thriller';       # Add new entry
print(movie)
# output:
{'Name': 'inception', 'director': 'nolan', 'release': 2010,
'genre': 'thriller'}
```

Remove items from a Dictionary

The **pop()** method deletes an item with a specific key.

```
quiz={'quiz1': 20, 'quiz2': 18,'quiz3': 12}
quiz.pop('quiz3')
print(quiz)
# output:
{'quiz1': 20, 'quiz2': 18}
```

The **popitem()** method removes a random item.

```
students = {'harry': 101, 'ron': 209, 'tom': 267}
print(students.popitem())
# output:
('tom', 267)
print(students)
# output:
{'harry': 101, 'ron': 209}
```

You can use **del** keyword to remove an element with a specific key. Also, **clear()** method empties the dictionary.

```
closet = {'shirt': 3,'pant': 4,'scarf': 2}
del closet['scarf']
print(closet)
# output:
{'shirt': 3, 'pant': 4}

del closet   # delete entire dictionary and generate error
print(closet)
NameError: name 'closet' is not defined

closet.clear()
print(thisdict)
# output:
{}
```

Copy a Dictionary

In dictionary you cannot copy a dictionary to another one by writing **my_dict1 = my_dict2**.

One way to make a copy is to use the built in method **copy().**

```
novel = {'name': 'To Kill a Mockingbird', 'author': 'Harper Lee',
'published': 1960}
checklist = novel.copy()
print(checklist)
# output:
{'name': 'To Kill a Mockingbird', 'author': 'Harper Lee',
'published': 1960}
```

Another way is using built-in function **dict().**

```
fictions = {'sci-fi': 25,'mystery': 17,'mythology': 12}
library = dict(fictions)
print(library )
# output:
{'sci-fi': 25, 'mystery': 17, 'mythology': 12}
```

Loop through a Dictionary

We can loop through a dictionary by using for loop.

```
chocolates = {'cadbury': 8,'hersheys': 9,'bueno': 12}
# print all the keys
for c in chocolates:
    print(c, end = " ")
# output:
cadbury hersheys bueno

# print all the values
for c in chocolates:
    print(chocolates[c] , end = " ")
# output:
8 9 12

# print all values and keys for a, b in
chocolates.items():
    print(a,":", b, end = " ")
# output:
cadbury : 8 hersheys : 9 bueno : 12
```

```
# assigning value with for loop
timetable = {}
for x in range(8):
    timetable[x] = x*12
print(timetable)
# output:
{0: 0, 1: 12, 2: 24, 3: 36, 4: 48, 5: 60, 6: 72, 7: 84}

# assigning values with if conditional
even_ttable = {x: x*15 for x in range(11) if x % 2 == 0}
print(even_ttable)
# output:
{0: 0, 2: 30, 4: 60, 6: 90, 8: 120, 10: 150}
```

Dictionary Membership Test

For checking whether a key is in the dictionary or not, we use the membership operator, **in** and **not in**. Remember, membership test checks only key, not value.

```
item = {'harry':'owl','ron':'mouse','hermione':'cat'}
print('harry' in item)
# output:
True

print('ginny' in item)
# output:
False
```

Dictionary methods

There are some built- in methods in python dictionary which we can use.

Method	Description	Example

items()	Returns a list containing a tuple for each key value pair	`music = {`
		`  'name': 'hey jude',`
		`  'band': 'beatles',`
		`  'year': 1968`
		`}`
		`x = music.items()`
		`print(x)`
		`# output:`
		`dict_items([('name', 'hey jude'), ('band', 'beatles'), ('year', 1968)])`

keys()	Returns a list containing the dictionary's keys	`music = {`
		`  'name': 'hey jude',`
		`  'band': 'beatles',`
		`  'year': 1968`
		`}`
		`x = music.keys()`
		`print(x)`
		`# output:`
		`dict_keys(['name', 'band', 'year'])`

setdefault()	Returns the value of the specified key. If the key does not exist: insert the key, with the specified value	`music = {`
		`  'name': 'hey jude',`
		`  'band': 'beatles',`
		`  'year': 1968`
		`}`
		`x= music.setdefault('album', 'hey jude')`
		`print(x)  # output: hey jude`

update()	Updates the dictionary with given key-value pairs	`music = {`
		`  'name': 'hey jude',`
		`  'band': 'beatles',`
		`  'year': 1968`
		`}`
		`music.update({'album': 'hey jude'})`
		`print(music)`
		`# output:`
		`{'name': 'hey jude', 'band': 'beatles', 'year': 1968, 'album': 'hey jude'}`

values()	Returns a list of all the values	`music = {`
		`  'name': 'hey jude',`
		`  'band': 'beatles',`
		`  'year': 1968`
		`}`
		`x = music.values()`
		`print(x)`
		`# output:`
		`dict_values(['hey jude', 'beatles', 1968])`

len(dict)	Provides the length of the dictionary which means the number of how many items are there	`music = {`
		`  'name': 'hey jude',`
		`  'band': 'beatles',`
		`  'year': 1968`
		`}`
		`print(len(music))`
		`# output: 3`

str(dict)	Generates a string representation of the dictionary	`music = {`
		`  'name': 'hey jude',`
		`  'band': 'beatles',`
		`  'year': 1968`
		`}`
		`print ("String :" ,str(music))`
		`# output:`

		String : {'name': 'hey jude', 'band': 'beatles', 'year': 1968}
fromkeys()	Returns a dictionary with the specific key and value	**x = ('num1', 'num2', 'num3')** **y = 100** **mydict = dict.fromkeys(x, y)** **print(mydict)** # output: **{'num1': 100, 'num2': 100, 'num3': 100}**
sorted()	Sorts the elements of the dictionary by keys and returns	**music = {** **'name': 'hey jude',** **'band': 'beatles',** **'year': 1968** **}** **print(sorted(music))** # output: **['band', 'name', 'year']**

Tuples

A tuple is a sequence of elements which is ordered and immutable. It is similar to lists. In Python we write tuples inside round brackets. The difference between list and python is that we cannot change the items of a tuple if it is assigned once whereas we can change the items of a list.

Creating a Tuple

We can create a tuple by placing all the comma- separated elements within a parenthesis (**()**). A tuple can consist of any number of elements and they also can be of different types (integer, float, list, string, etc.).

If you want create a tuple with only one element, then you have to add a comma after the element, or else python will not identify it as a tuple.

```
# Empty tuple
my_tuple = ()
print(my_tuple)

# Tuple having floats
my_tuple = (250.50, 350.60, 450.70)
print(my_tuple)

# Tuple having strings
students = ('frodo', 'bilbo', 'gollum')
print(students)

# tuple with mixed datatypes
mixed = (540, 'bonjour', 255.85)
print(mixed)

# nested tuple
my_tuple = ('hola', ['edward', 'lucy', 'caspian'], (10, 20, 30))
print(my_tuple)
```

You can also create a tuple without using parentheses. This is called tuple packing.

```
my_tuple = 300, 406.57, 'chocolate'
print(my_tuple)  # output: (300, 406.57, 'chocolate')

x,y,z=my_tuple   # tuple unpacking
print(x)         # output: 300
print(y)         # output: 406.57
print(z)         # output: chocolate
```

Accessing Elements of a Tuple

The elements of a tuple can be accessed by mentioning the index number, inside square brackets (**[]**). Like lists, the index can be positive or negative both. However, the index should be an integer, you cannot use float or other types. This will generate TypeError.

```
books=('scifi','mystery','mythology','biography','history','horror',
'thriller')
print(books[0])      # output: sci-fi
print(books[4])      # output: history
print(books[6])      # output: thriller
print(books[-1])     # output: thriller
print(books[-3])     # output: history
```

Nested tuple is accessed by nested indexing.

```
nested = ('hola', ['edward', 'lucy', 'caspian'], (10, 20, 30))
print(nested[0][2])        # output: l
print(nested[2][2])        # output: 30
print(nested[-1][-3])      # output: 10
print(nested[-2][-1])      # output: caspian
```

Mutability of Tuples

Unlike lists, tuples are immutable. Once you create a tuple, then you cannot change its elements. However, if the tuple is a nested tuple and one of its element is immutable type such as lists, in that case the nested element can be changed.

```
instruments = ('flute','piano','violin','ukelele')
instruments[1] = 'flute'
print(instruments)
TypeError: 'tuple' object does not support item assignment
```

Two tuples can be combined by using **+** operator which is called concatenation. Also, with ***** operator the items in a tuple can be repeated for a given number of times. Both the operations result in a new tuple.

```
snacks=('pizza','pasta','mac n cheese') + ('cake', 'brownie',
'coffee')
print(snacks)
# output
('pizza', 'pasta', 'mac n cheese', 'cake', 'brownie', 'coffee')

print(('bonne nuit',) * 4)
# output
('bonne nuit', 'bonne nuit', 'bonne nuit', 'bonne nuit')
```

As tuples cannot be changed, so you cannot remove elements from it, but you can delete the entire tuple.

```
fruits = ('apple', 'cherry', 'mango')
del fruits[1]
TypeError: 'tuple' object doesn't support item deletion

del fruits   # deletes entire table
```

Advantages of Tuple over List

As tuples and lists are similar, both of them are used in similar cases. Yet, there are some advantages of tuple over list.

- As tuple is immutable, it requires less memory space than that of list.
- A tuple is faster than list as they are immutable, it takes lesser time to iterate through a tuple.
- Tuple is generally used for heterogeneous (different) data types and lists for homogeneous (similar) data types.
- Tuples have immutable elements so they can be used as keys for a dictionary whereas with lists, that cannot not possible.
- In case of the data that do not change, implementing it as tuple will make sure that it remains write-protected.