Dictionary Data Structure

Anab Batool Kazmi

Data Structures

- Data structures are organized formats or **containers** used to **store, manage, and manipulate data efficiently** within computer programs, facilitating tasks like **insertion**, **retrieval**, and **modification** of information.
- Python offers a variety of data structures, including **lists, tuples, sets, dictionaries, linked lists, trees, graphs, and more**, facilitating efficient data manipulation and storage.

Data Type Mutability in Python

```
x=10
print(id(x))
x=11
print(id(x))
```

```
1402367328
1402367360
```

Mutable Data Types in Python

- Mutable data types in Python are those whose ***values can be changed in place*** after they have been created.
- **List, Dictionary, Set**

Immutable Data Types in Python

- Immutable data types in Python are those ***whose values, once assigned, cannot be changed***, and any operation that appears to ***modify them actually creates a new object*** with the modified value.
- **Tuple, String, Numeric, Boolean**

Dictionary Data Structure

- A dictionary in Python is a data structure that stores data in **key-value pairs**.
- Each **key is unique immutable type**, and each **value can be of any type (both mutable and immutable) and can be duplicated.**
- Dictionary keys are **case sensitive**
- As of Python **version 3.7, dictionaries are ordered**. In Python 3.6 and earlier, dictionaries are unordered.

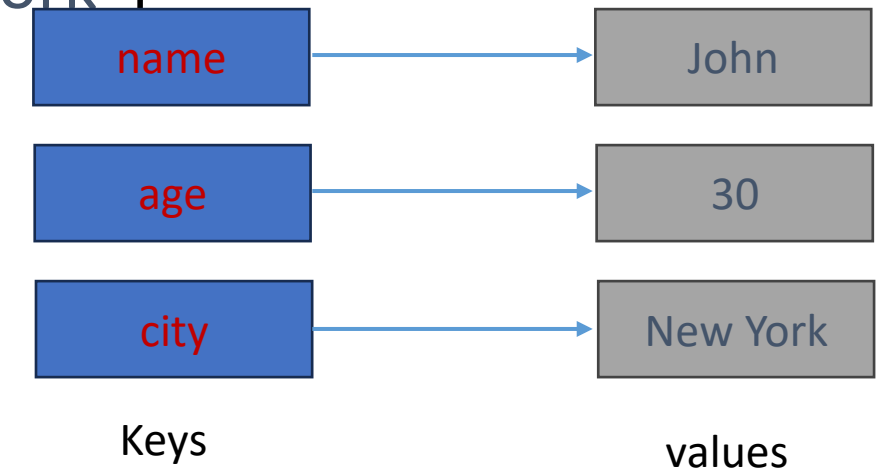
Dictionary Data Structure

- Dictionaries can be created using curly braces ({}).
- The keys are listed on the left side of the colon (:), and the values are listed on the right side of the colon.
- All the items(key value pairs) are separated by commas .
- e.g)

my_dict = {"name": "John", "age": 30, "city": "New York"}

Keys= name, age, city

Values= John,30, New York



Dictionary Items - Data Types

- The values in dictionary items can be of any data type

```
thisdict = {
  "brand": "Ford",
  "electric": False,
  "year": 1964,
  "colors": ["red", "white", "blue"]
}
```

How to fetch a particular value

In Python, you can access the values of a dictionary by using the keys as indexes.

- Using Square Brackets [ ]

```
dictionary_value = dictionaryName["key"]
```

- Using the get() Method

```
dictionary_value = dictionaryName.get("key“,
“optional default value if key doesn’t exist”)
```

How to fetch a particular value

```
my_dictionary = {
  "name": "Ayesha Khan",
  "age": 20,
  "occupation": "Student"
}

# Access the value for the key "name" using the key
name = my_dictionary["name"]

# Access the value for the key "occupation" using the get() method
occupation = my_dictionary.get("occupation")

# Print the values
print(name)
print(occupation)
```

```
Ayesha Khan
Student
```

How to fetch a particular value

```python
#How to fetch a particular value
my_dictionary = {
  "name": "Ayesha Khan",
  "age": 20,
  "occupation": "Student"
}
# Access the value for the key "occupation" using the get() method
occupation = my_dictionary.get("occupation")
# Access the value for the key "rollNo" using the get() method
var = my_dictionary.get("rollNo")
# Access the value for the key "rollNo" using the get() method and default value
var1 = my_dictionary.get("rollNo","Roll no is not available")

# Print the values
print(occupation)
print(var)
print(var1)
```

```
Student
None
Roll no is not available
```

How to fetch a particular value

```
thisdict = {
  "brand": "Ford",
  "electric": False,
  "year": 1964,
  "colors": ["red", "white", "blue"]
}

print(thisdict["colors"][0])
```

```
red
```

Looping through a dictionary in Python

Loop Through Keys:

- You can use a for loop to iterate through the keys of a dictionary.

```
#Loop Through Keys
my_dict = {"name": "John", "age": 30, "city": "New York"}

for key in my_dict:
    print(key)
```

```
name
age
city
```

Looping through a dictionary in Python

Loop Through Values:

- You can also use a for loop to iterate through the values of a dictionary **by accessing them using the keys**.

```
#Loop Through Values:
my_dict = {"name": "John", "age": 30, "city": "New York"}

for key in my_dict:
    value = my_dict[key]
    print(value)
```

```
John
30
New York
```

Looping through a dictionary in Python

Loop Through Key-Value Pairs

- To iterate through both the keys and values of a dictionary simultaneously, you can use the **items()** method.

```
#Loop Through Key-Value Pairs
my_dict = {"name": "John", "age": 30, "city": "New York"}

for key, value in my_dict.items():
    print(f"Key: {key}, Value: {value}")
```

```
Key: name, Value: John
Key: age, Value: 30
Key: city, Value: New York
```

Built in methods

Function	Description	syntax
keys()	The keys() method will return a list of all the keys in the dictionary.	dictionaryName.keys()
values()	The values() method will return a list of all the values in the dictionary.	dictionaryName. values()
items()	The items() method will return each item in a dictionary, as tuples in a list.	dictionaryName. items()

Built in methods

```
my_dict = {"name": "John", "age": 30, "city": "New York"}
print(my_dict)
print(my_dict.keys())
print(my_dict.values())
print(my_dict.items())
```

```
{'name': 'John', 'age': 30, 'city': 'New York'}
dict_keys(['name', 'age', 'city'])
dict_values(['John', 30, 'New York'])
dict_items([('name', 'John'), ('age', 30), ('city', 'New York')])
```

Check if Key Exists

- To determine if a specified key or value is present in a dictionary use the **in** keyword

```
#To determine if a specified key or vallue is present in a dictionary use the in keyword:

thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
if "model" in thisdict:
  print("Yes, 'model' is one of the keys in the thisdict dictionary")

if "brand" in thisdict.keys():
  print("Yes, 'brand' is one of the keys in the thisdict dictionary")

if "Ford" in thisdict.values():
  print("Yes, 'Ford' is one of the values in the thisdict dictionary")

if ("brand","Ford") in thisdict.items():
  print("Yes, (brand,Ford) is one of the key,value pair in the thisdict dictionary")
```

```
Yes, 'model' is one of the keys in the thisdict dictionary
Yes, 'brand' is one of the keys in the thisdict dictionary
Yes, 'Ford' is one of the values in the thisdict dictionary
Yes, (brand,Ford) is one of the key,value pair in the thisdict dictionary
```

Change/Update Dictionary Items

There are two ways to update dictionary items in Python

- Using the [] operator
  - to update the value associated with a key in a dictionary
- Using the update() method
  - The update() method takes a dictionary as an argument and updates the existing dictionary with the key-value pairs from the argument dictionary.

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict["brand"]="BMW"
thisdict.update({"year": 1950})

print(thisdict)
```

```
{'brand': 'BMW', 'model': 'Mustang', 'year': 1950}
```

Create an empty dictionary

You can create an empty dictionary in Python using one of the following methods:

- Using Curly Braces {}

```
empty_dict = {}
```

- Using the dict() Constructor

```
empty_dict = dict()
```

Create an empty dictionary

Using Curly Braces {}

```
#CREATE EMPTY DICTIONARY Using Curly Braces {}:
import sys

my_dictionary = {}
print("Type of my_list data structure is :",type(my_dictionary))
print("List Elements are :",my_dictionary)
print("No. of Elements in my_list are :",len(my_dictionary))
print(f"Memory reserved by my_dictionary is {sys.getsizeof(my_dictionary)} bytes")
print(f"Memory address of my_dictionary is {id(my_dictionary)}")
```

```
Type of my_list data structure is : <class 'dict'>
List Elements are : {}
No. of Elements in my_list are : 0
Memory reserved by my_dictionary is 240 bytes
Memory address of my_dictionary is 2238672885296
```

Using the dict() Constructor

```
#CREATE EMPTY DICTIONARY UsingUsing the dict() Constructor
import sys

my_dictionary = dict()
print("Type of my_list data structure is :",type(my_dictionary))
print("List Elements are :",my_dictionary)
print("No. of Elements in my_list are :",len(my_dictionary))
print(f"Memory reserved by my_dictionary is {sys.getsizeof(my_dictionary)} bytes")
print(f"Memory address of my_dictionary is {id(my_dictionary)}")
```

```
Type of my_list data structure is : <class 'dict'>
List Elements are : {}
No. of Elements in my_list are : 0
Memory reserved by my_dictionary is 240 bytes
Memory address of my_dictionary is 2238672887096
```

How to initialize a dictionary

Initializing with Key-Value Pairs:

- To create a dictionary with initial key-value pairs, you can use curly braces {} and provide the key-value pairs separated by colons :

```
my_dict = {"name": "John", "age": 30, "city": "New York"}
```

How to initialize a dictionary

Using a List of Tuples:

- You can initialize a dictionary using a list of tuples where each tuple represents a key-value pair
- This method is useful when you have the data in a list and want to convert it into a dictionary.

```
my_dict = dict([("name", "John"), ("age", 30), ("city", "New York")])
```

How to initialize a dictionary

Using Dictionary Comprehension:

- You can use dictionary comprehension to create a dictionary based on an iterable (e.g., a list of tuples or key-value pairs)
- This method allows you to create a dictionary with specific key-value pairs and can be useful for data transformation.

Python Dictionary Comprehension

- Dictionary comprehensions are **a concise way to create** dictionaries in Python.
- They work similarly to list comprehensions, but instead of returning a list, they **return a dictionary**.

Syntax

{key: value for key, value **in** iterable **if** condition}

Python Dictionary Comprehension

{key: value **for** key, value **in** iterable **if** condition}

Where:

- **key** is the key of the dictionary entry.
- **value** is the value of the dictionary entry.
- **iterable** is an iterable object, such as a list, tuple, or set.
- **condition** is an **optional expression** that must evaluate to True for the dictionary entry to be included in the dictionary.

Code snippet to create a dictionary from list of tuple

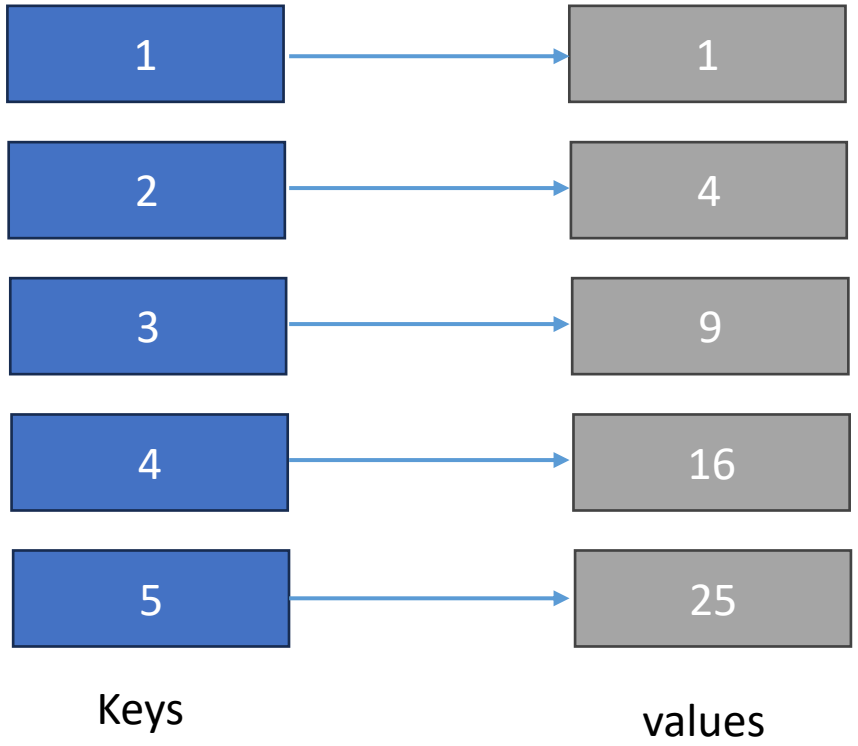
```
my_dict = {key: value for key, value in [("name", "John"), ("age", 30), ("city", "New York")]}
print(my_dict)
print(my_dict.keys())
print(my_dict.values())
print(my_dict.items())
```

```
{'name': 'John', 'age': 30, 'city': 'New York'}
dict_keys(['name', 'age', 'city'])
dict_values(['John', 30, 'New York'])
dict_items([('name', 'John'), ('age', 30), ('city', 'New York')])
```

Code snippet to create a dictionary of squares from 1 to 5

```
# Create a dictionary of squares from 1 to 5
squares = {x: x * x for x in range(1, 6)}
print(squares)
print(squares.keys())
print(squares.values())
print(squares.items())
```

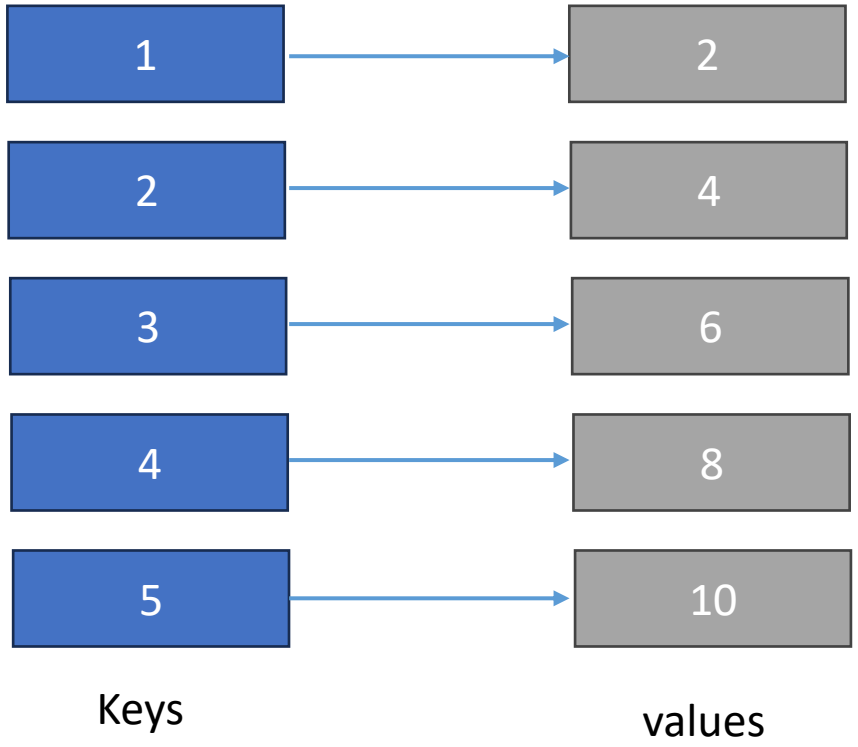
```
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
dict_keys([1, 2, 3, 4, 5])
dict_values([1, 4, 9, 16, 25])
dict_items([(1, 1), (2, 4), (3, 9), (4, 16), (5, 25)])
```



Code snippet to # Create a dictionary of all the even numbers from 1 to 5

```
# Create a dictionary of all the even numbers from 1 to 5
even_numbers  = {key: key * 2 for key in range(1, 6)}
print(even_numbers)
print(even_numbers.keys())
print(even_numbers.values())
print(even_numbers.items())
```

```
{1: 2, 2: 4, 3: 6, 4: 8, 5: 10}
dict_keys([1, 2, 3, 4, 5])
dict_values([2, 4, 6, 8, 10])
dict_items([(1, 2), (2, 4), (3, 6), (4, 8), (5, 10)])
```



Add Dictionary Items

```
#add dictionary item
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict["color"]="red"
print(thisdict)
thisdict.update({"type": "car"})
print(thisdict)
```

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964, 'color': 'red'}
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964, 'color': 'red', 'type': 'car'}
```

Adding an item to the dictionary is done by

Using the [] operator

- using a new index key and assigning a value to it
- Using the **update()** method
  - The update() method will update the dictionary with the items from a given argument. If the item does not exist, the item will be added.

Remove Dictionary Items

- The **pop()** method removes the item with the specified key name

```
#remove item
#The pop() method removes the item with the specified key name
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.pop("model")
print(thisdict)
```

```
{'brand': 'Ford', 'year': 1964}
```

Remove Dictionary Items

- The **popitem()** method removes the last inserted item (in versions before 3.7, a random item is removed instead

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.popitem()
print(thisdict)
```

```
{'brand': 'Ford', 'model': 'Mustang'}
```

Remove Dictionary Items

- The **del** keyword removes the item with the specified key name:

```
#The del keyword removes the iter
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
del thisdict["model"]
print(thisdict)
```

```
{'brand': 'Ford', 'year': 1964}
```

Remove Dictionary Items

- The del keyword can also delete the dictionary completely

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
del thisdict
print(thisdict) #this will cause an error because "thisdict" no longer exists.
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-47-cf8f189c2310> in <module>()
      5 }
      6 del thisdict
----> 7 print(thisdict) #this will cause an error because "thisdict" no longer exists.

NameError: name 'thisdict' is not defined
```

Remove Dictionary Items

- The clear() method empties the dictionary:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.clear()
print(thisdict)
```

```
{}
```

Copy Dictionaries

- You cannot copy a dictionary simply by typing **dict2 = dict1**, because: **dict2 will only be a reference to dict1**, and changes made in dict1 will automatically also be made in dict2.
- There are ways to make a copy
  - one way is to use the built-in Dictionary method **copy().**
  - Another way to make a copy is to use the built-in function **dict().**

Copy Dictionaries

- Make a copy of a dictionary with the copy() method:

```
#copy dictionary items using copy item
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
mydict = thisdict.copy()
print(mydict)
```

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
```

Copy Dictionaries

- Make a copy of a dictionary with the dict() function:

```
#Make a copy of a dictionary with the dict() function:
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
mydict = dict(thisdict)
print(mydict)
```

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
```

Nested Dictionaries in python

- A nested dictionary in Python is a dictionary that contains another dictionary, or even multiple dictionaries. This allows you to store complex data structures in a concise and organized way.

- Nested dictionaries can be created in Python using the same syntax as regular dictionaries.

Nested Dictionaries in python

```python
#nested dictionary
nested_dict = {
    "child1": {
        "name": "Emil",
        "year": 2004
    },
    "child2": {
        "name": "Tobias",
        "year": 2007
    }
}

print(nested_dict)
print(nested_dict.keys())
print(nested_dict.values())
print(nested_dict.items())
```

```
{'child1': {'name': 'Emil', 'year': 2004}, 'child2': {'name': 'Tobias', 'year': 2007}}
dict_keys(['child1', 'child2'])
dict_values([{'name': 'Emil', 'year': 2004}, {'name': 'Tobias', 'year': 2007}])
dict_items([('child1', {'name': 'Emil', 'year': 2004}), ('child2', {'name': 'Tobias', 'year': 2007})])
```

Nested Dictionaries in python

- To access **a value** in a nested dictionary, you can use the **key path**. The **key path is a string that specifies the path to the value, starting from the root dictionary**. For example, to access the name of child 2, you would use the following code:

```
print(nested_dict["child2"])
print(nested_dict["child2"]["name"])
```

```
{'name': 'Tobias', 'year': 2007}
Tobias
```

Nested Dictionaries in python

- Using **key path to access specific record of dictionary.**

```
student_records = {
    "student1": {
        "name": "Alice",
        "age": 15,
        "courses": ["math", "science", "english"]
    },
    "student2": {
        "name": "Bob",
        "age": 16,
        "courses": ["math", "science", "history"]
    },
    "student3": {
        "name": "Carol",
        "age": 17,
        "courses": ["math", "art", "music"]
    }
}
print(student_records.keys())
print(student_records["student1"]["courses"][1])
```

```
dict_keys(['student1', 'student2', 'student3'])
science
```

- Nested dictionaries can be used to represent a variety of data structures, such as **objects, schemas, and databases**. For example, the following code creates a nested dictionary to represent a student's grades:

```
student_grades = {
    "name": "Alice",
    "courses": {
        "math": 90,
        "science": 85,
        "english": 95
    }
}
print(student_grades)
```

```
{'name': 'Alice', 'courses': {'math': 90, 'science': 85, 'english': 95}}
```

Quiz

```
library = {
    "978-0345378484": {
        "title": "The Hitchhiker's Guide to the Galaxy",
        "author": "Douglas Adams",
        "year": 1979,
        "genres": ["Science Fiction", "Comedy"]
    },
    "978-1451673319": {
        "title": "1984",
        "author": "George Orwell",
        "year": 1949,
        "genres": ["Dystopian", "Political Fiction"]
    },
    "978-0061120084": {
        "title": "To Kill a Mockingbird",
        "author": "Harper Lee",
        "year": 1960,
        "genres": ["Classic", "Legal Drama"]
    }
}
```

Write python code for following queries. Also give output

- Print all the keys of library dictionary.
- Print all the keys against the value assigned to 978-1451673319
- Update the second genres of 978-1451673319 to Science Fiction
- Delete genre of 978-1451673319