Newton School of Technology

17

Dictionaries in Python

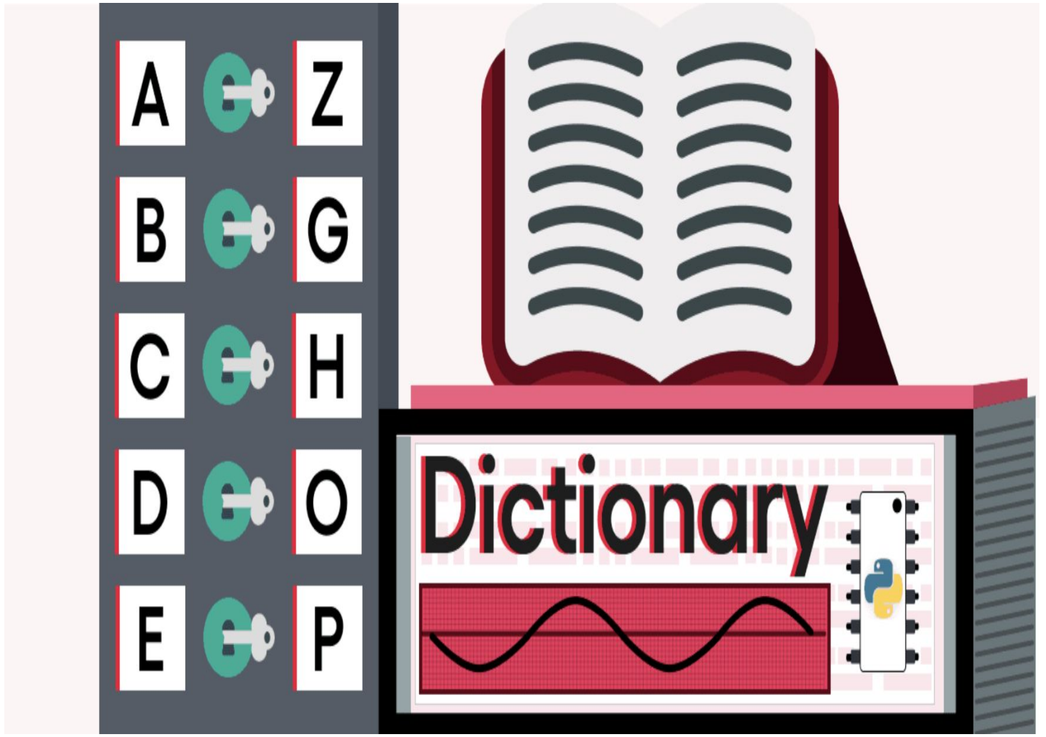
by Gladden Rumao

C14: Fundamentals of Programming

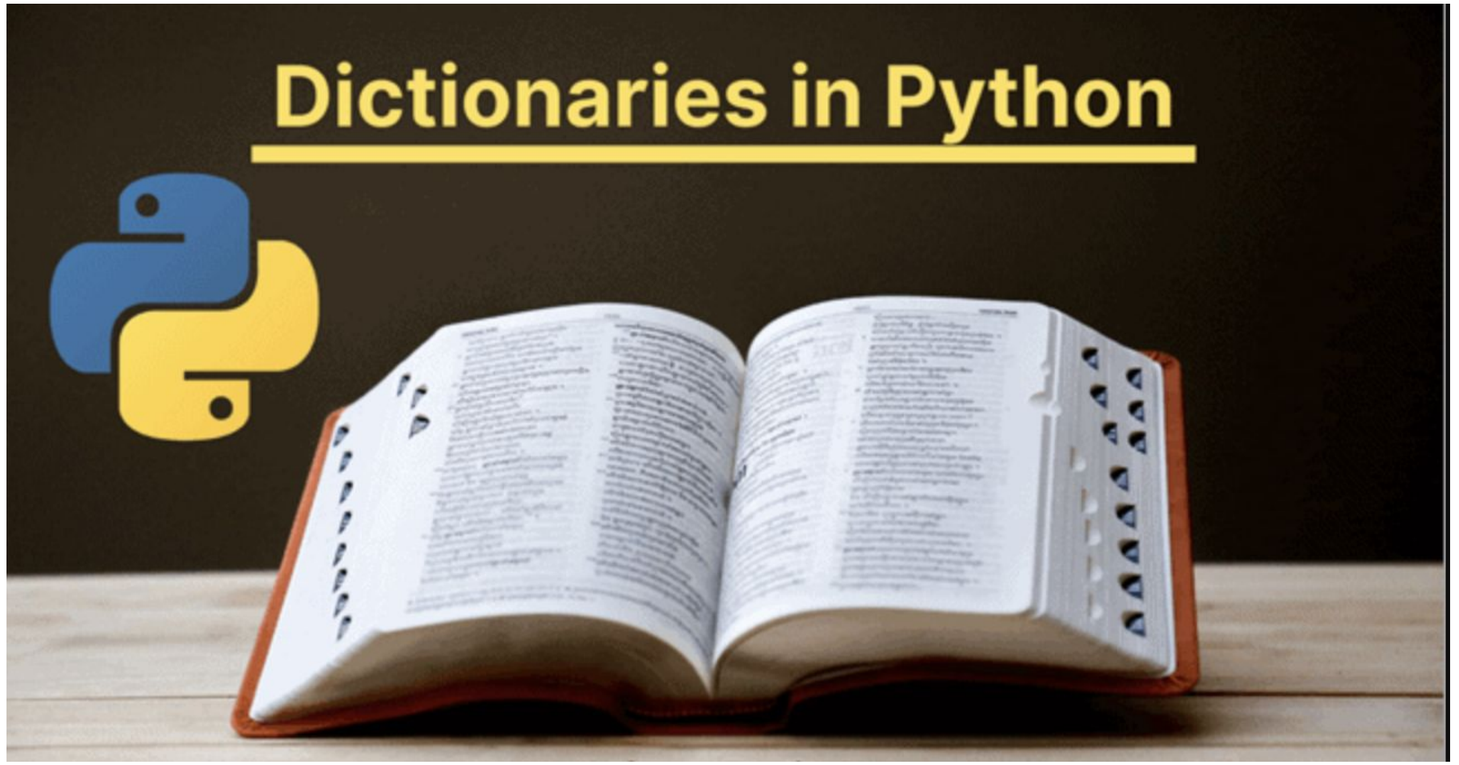
Newton School of Technology

Definition :

Dictionaries in Python are **unordered** collections of **key-value** pairs, where each **unique** key is mapped to a corresponding value.



Dictionaries Analogy :

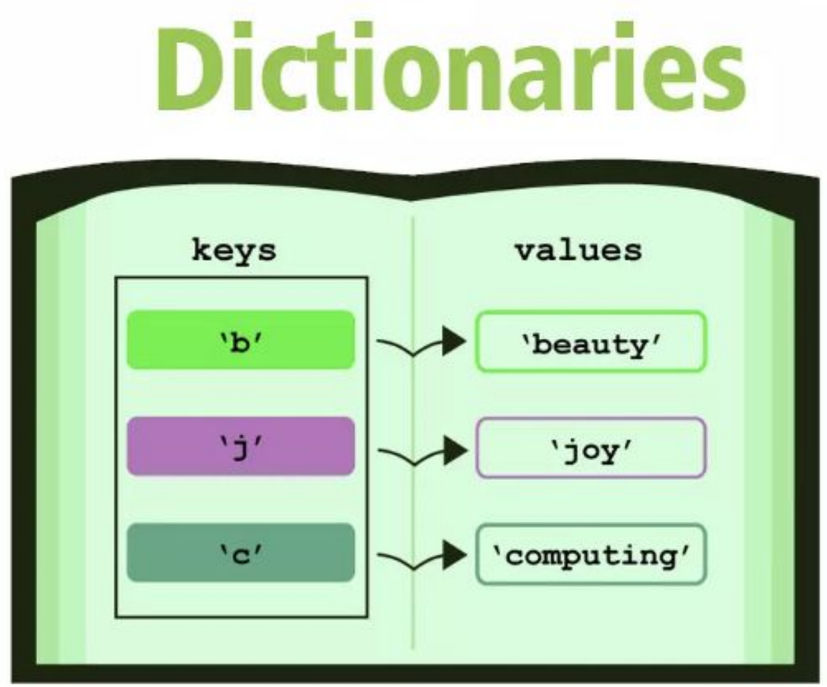


Python **dictionaries** are akin to **real-world** dictionaries found on bookshelves.

Dictionaries Analogy :

Newton School of Technology

Just as a **traditional dictionary** pairs **words** with their **definitions**, Python dictionaries pair **keys** with their associated **values**.



Why Dictionaries ?

Python dictionaries pair **keys** with their associated **values**.

This allows efficient **lookup** and **retrieval** of information.

Also allows us to **add and delete** such data **efficiently**.

How to Create Dictionaries ?

Newton School of Technology

Empty Dictionaries

We can create empty dictionaries with curled braces {}.

```python
my_dict = {}
```

Dictionaries with Initial Values

To create a dictionary with initial values, specify key-value pairs within {}:

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

Using dict() constructor

Newton School of Technology

Python's dict() constructor allows you to create dictionaries in a slightly different format

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

- Keys are passed as arguments without quotes (name, age, city).
- Values are assigned to these keys ("John", 30, "New York").

Newton School of Technology

Accessing Values in Dictionary

Using keys

Newton School of Technology

The most direct way to access a value in a dictionary is by using its key within square brackets. This method is simple and commonly used.

Syntax:

my_dict[key]

```python
my_dict = {'name': 'Alice', 'age': 30, 'city': 'New York'}
value = my_dict['name']
print(value)  # Output: Alice
```

Using get()

Newton School of Technology

The **get()** method in Python dictionaries provides a more flexible way to access values, especially when you're not certain whether a key exists in the dictionary:

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

name_value = my_dict.get("name")
print(name_value)  # Output: John
```

Advantages of using get()

Newton School of Technology

- **Avoids KeyError**: If the key is not present in the dictionary, using get() will not raise a KeyError. Instead, it will return None (or a specified default value, if provided).

```python
job_value = my_dict.get("job")
print(job_value)  # Output: None
```

- **Provides a Default Value**: You can specify a default value as the second argument to get(), which will be returned if the key is not found in the dictionary:

```python
job_value = my_dict.get("job", "Not specified")
print(job_value)  # Output: Not specified
```

Newton School of Technology

Modifying Dictionaries

Adding new key-value pairs

Newton School of Technology

To add a new key-value pair to a dictionary, you can simply assign a value to a new key within square brackets:

```python
my_dict = {"name": "John", "age": 30}
my_dict["email"] = "john@example.com"
```

Adding new key-value pairs

Newton School of Technology

After executing this line, my_dict will contain:

```python
{'name': 'John', 'age': 30, 'email': 'john@example.com'}
```

Updating existing value

Newton School of Technology

If you want to update the value of an existing key in the dictionary, you can directly assign a new value to that key.

```python
my_dict["age"] = 31
```

The update() method

Newton School of Technology

The **update()** method allows you to modify multiple key-value pairs at once or add new ones. Here's how you can use it:

```python
my_dict = {"name": "John", "age": 30}
my_dict.update({"age": 32, "city": "San Francisco"})
```

The update() method

Newton School of Technology

After executing **my_dict.update({"age": 32, "city": "San Francisco"})**, my_dict will be updated to:

```python
{'name': 'John', 'age': 32, 'city': 'San Francisco'}
```

- The existing key "**age**" is updated to **32.**
- The new key-value pair "**city**": "**San Francisco**" is added to my_dict.

Newton School of Technology

Removing Key-Value Pair

‘del’ statement

Newton School of Technology

- The `del` statement is a straightforward way to remove a specific key-value pair from a dictionary.
- If you know the key of the pair you want to remove, you can use this method. If the key does not exist, it will raise a `KeyError`.

Syntax:

```python
del dictionary[key]
```

Example using ‘del’

Newton School of Technology

```python
my_dict = {"name": "Alice", "age": 30, "city": "New York"}
del my_dict["age"]
print(my_dict)
# Output: {'name': 'Alice', 'city': 'New York'}
```

‘pop()’ method

Newton School of Technology

- The pop() method removes a specific key and returns its associated value.
- If the key does not exist, it raises a KeyError unless a default value is provided.

Syntax:

```python
value = dictionary.pop(key, default)
```

Example using ‘pop()’

Newton School of Technology

```python
my_dict = {"name": "Alice", "age": 30, "city": "New York"}
name = my_dict.pop("name")
print(name)          # Output: Alice
print(my_dict)       # Output: {'age': 30, 'city': 'New York'}
```

You can also provide a **default** value to return if the key does not exist:

```python
age = my_dict.pop("age", "Not Found")
print(age)           # Output: 30
```

‘popitem()’ method

Newton School of Technology

- The `popitem()` method removes and returns the last key-value pair added to the dictionary. This is useful for LIFO (Last In, First Out) operations. It will raise a `KeyError` if the dictionary is empty.

Syntax:

```python
key, value = dictionary.popitem()
```

Example using ‘popitem()’

Newton School of Technology

```python
my_dict = {"name": "Alice", "age": 30, "city": "New York"}
key, value = my_dict.popitem()
print(key, value)  # Output: ('city', 'New York')
print(my_dict)     # Output: {'name': 'Alice', 'age': 30}
```

‘clear()’ method

Newton School of Technology

If you want to remove all key-value pairs from a dictionary, you can use the `clear()` method. This effectively empties the dictionary but keeps the dictionary object itself intact.

Syntax:
dictionary.clear()

Example:

```python
my_dict = {"name": "Alice", "age": 30, "city": "New York"}
my_dict.clear()
print(my_dict)  # Output: {}
```

Summary

Newton School of Technology

Feature	`del` Statement	`pop()` Method	`popitem()` Method	`clear()` Method
Description	Removes a specific key-value pair from the dictionary.	Removes a specific key-value pair and returns the value.	Removes and returns the last key-value pair from the dictionary.	Removes all key-value pairs from the dictionary.
Advantages	- Simple and direct. No return value needed.	- Returns the value of the removed key. Allows specifying a default value.	- Useful for LIFO operations. Does not require specifying a key.	- Completely empties the dictionary. Retains the dictionary object.
Disadvantages	- Raises `KeyError` if the key is not found.	- Raises `KeyError` if the key is not found (unless default is provided).	- Always removes the last item, which may not be desirable. Raises `KeyError` if dictionary is empty.	- Irreversible for the current dictionary object. May be excessive if only one item needs removal.
Example Usage	`del my_dict['key']`	`value = my_dict.pop('key')` `value = my_dict.pop('key', default)`	`key, value = my_dict.popitem()`	`my_dict.clear()`

Newton School of Technology

Dictionary Methods

keys()

Newton School of Technology

Description: Returns a view object that displays a list of all the keys in the dictionary.
Usage: Useful for iterating over or accessing the keys of the dictionary.

Example:

```python
my_dict = {'a': 1, 'b': 2, 'c': 3}
keys = my_dict.keys()
print(keys)  # Output: dict_keys(['a', 'b', 'c'])
```

values()

Newton School of Technology

Description: Returns a view object that displays a list of all the values in the dictionary.
Usage: Useful for iterating over or accessing the values of the dictionary.

Example:

```python
my_dict = {'a': 1, 'b': 2, 'c': 3}
values = my_dict.values()
print(values)  # Output: dict_values([1, 2, 3])
```

items()

Newton School of Technology

Description: Returns a view object that displays a list of key-value pairs (tuples) in the dictionary.
Usage: Useful for iterating over both keys and values together.

Example:

```python
my_dict = {'a': 1, 'b': 2, 'c': 3}
items = my_dict.items()
print(items)  # Output: dict_items([('a', 1), ('b', 2), ('c', 3)])
```

Example Usage

```
# Creating a dictionary
my_dict = {'name': 'Alice', 'age': 30, 'city': 'New York'}

# Getting keys
keys = my_dict.keys()
print("Keys:", keys)  # Output: Keys: dict_keys(['name', 'age', 'city'])

# Getting values
values = my_dict.values()
print("Values:", values)  # Output: Values: dict_values(['Alice', 30, 'New York'])

# Getting items
items = my_dict.items()
print("Items:", items)  # Output: Items: dict_items([('name', 'Alice'), ('age', 30

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

Thank You!