📘 What is a Dictionary?

- A dictionary is a built-in Python data structure that stores data in key-value pairs.
- Keys must be unique and immutable (e.g., strings, numbers, tuples).
- Values can be of any data type and may be duplicated.
- Dictionaries are defined using curly braces `{}` or the `dict()` constructor.
- They are unordered before Python 3.7, but ordered from Python 3.7 onward.
- Dictionaries are mutable, meaning they can be changed after creation.
- They allow fast access to data using keys.
- Syntax: `my_dict = {"name": "Alice", "age": 25}`
- Example:

```python
my_dict = {"name": "Alice", "age": 25}
print(my_dict)
```

- Output: `{'name': 'Alice', 'age': 25}` Programiz +1

🛠️ Creating a Dictionary

- You can create a dictionary using curly braces `{}` or the `dict()` function.
- Each item is a key-value pair separated by a colon `:`.
- Multiple pairs are separated by commas.
- Keys must be unique; duplicate keys overwrite previous values.
- Values can be any data type.
- Syntax: `my_dict = {"key1": "value1", "key2": "value2"}`
- Example:

```python
student = {"name": "John", "grade": "A"}
print(student)
```

- Output: `{'name': 'John', 'grade': 'A'}`
- You can also use `dict(name="John", grade="A")` Programiz +1

🔍 Accessing Values in a Dictionary

- Values are accessed using square brackets `[]` with the key.
- If the key doesn't exist, it raises a `KeyError`.
- You can also use the `get()` method to avoid errors.
- Syntax: `value = my_dict["key"]` or `value = my_dict.get("key")`
- Example:

```python
student = {"name": "John", "grade": "A"}
print(student["name"])
```

- Output: `John`

- Using `get()`:

```python
print(student.get("age", "Not Found"))
```

- Output: `Not Found`
- `get()` is safer for optional keys Programiz +1

🖊️ Updating Dictionary

- You can update values by assigning a new value to an existing key.
- Use `dict[key] = new_value` or `dict.update({key: value})`.
- If the key doesn't exist, it adds a new key-value pair.
- Syntax:

```python
my_dict["key"] = "new_value"
my_dict.update({"key": "new_value"})
```

- Example:

```python
student = {"name": "John", "grade": "A"}
student["grade"] = "B"
print(student)
```

- Output: `{'name': 'John', 'grade': 'B'}`
- Updating is useful for modifying data dynamically Programiz +1

❌ Deleting Elements from Dictionary

- Use `del dict[key]` to remove a specific key-value pair.
- Use `dict.pop(key)` to remove and return the value.
- Use `dict.clear()` to remove all items.
- Syntax:

```python
del my_dict["key"]
my_dict.pop("key")
my_dict.clear()
```

- Example:

```python
student = {"name": "John", "grade": "A"}
del student["grade"]
print(student)
```

- Output: `{'name': 'John'}`
- `pop()` is useful when you need the removed value Programiz +1

🧠 Dictionary Comprehension

- A concise way to create dictionaries using loops.
- Syntax: `{key_expr: value_expr for item in iterable}`
- Useful for transforming or filtering data.
- Example:

```python
squares = {x: x*x for x in range(5)}
print(squares)
```

- Output: `{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}`
- Can include conditions: `{x: x*x for x in range(5) if x % 2 == 0}`
- Output: `{0: 0, 2: 4, 4: 16}`
- Improves readability and performance Programiz +1

📌 Properties of Dictionary

- **Mutable**: Can be changed after creation.
- **Unordered (pre-3.7)**: No guaranteed order of items.
- **Ordered (3.7+)**: Maintains insertion order.
- **Keys are unique**: Duplicate keys overwrite previous values.
- **Keys must be immutable**: Strings, numbers, tuples.
- **Values can be any type**: Including lists, other dictionaries.
- **Fast lookup**: Uses hashing for quick access.
- **Dynamic**: Can grow or shrink as needed.
- **Nested dictionaries**: Dictionaries within dictionaries.
- **Versatile**: Used in many real-world applications GeeksForGeeks +1

Built-In Dictionary Methods

🧹 `clear()` Method

- Removes all items from the dictionary, leaving it empty.
- It does not delete the dictionary itself.
- Syntax: `dict.clear()`
- Example:

```python
student = {"name": "John", "grade": "A"}
student.clear()
print(student)
```

- Output: `{}`
- Useful when you want to reuse the dictionary structure.
- No return value; it modifies the dictionary in place.
- Safe to use without checking keys.
- Helps reset data during runtime.

`copy()` Method

- Returns a shallow copy of the dictionary.
- Changes to the copy don't affect the original.
- Syntax: `new_dict = dict.copy()`
- Example:

```python
student = {"name": "John", "grade": "A"}
clone = student.copy()
print(clone)
```

- Output: `{'name': 'John', 'grade': 'A'}`
- Shallow copy means nested objects are shared.
- Use `copy()` to duplicate data safely.
- Ideal for backup before modification.
- Doesn't copy metadata or methods.

- **Only the outer dictionary is duplicated**, not the nested objects inside it.
- If the dictionary contains **mutable objects** (like lists or other dictionaries), both the original and the copy will **reference the same inner objects**.
- Changes to those inner objects will affect both dictionaries.

Example:

```python
original = {"name": "John", "scores": [90, 85]}
shallow = original.copy()

shallow["scores"].append(95)
print("Original:", original)
print("Shallow:", shallow)
```

Output:

```
Original: {'name': 'John', 'scores': [90, 85, 95]}
Shallow: {'name': 'John', 'scores': [90, 85, 95]}
```

As you can see, modifying the list inside `shallow` also changes it in `original`. That's the essence of a **shallow copy** — it doesn't create deep, independent copies of nested structures.

🔍 `get()` Method

- Retrieves the value for a given key.
- Returns `None` or a default value if the key is missing.
- Syntax: `dict.get(key, default)`
- Example:

```python
student = {"name": "John"}
print(student.get("grade", "Not Found"))
```

- Output: `Not Found`
- Prevents `KeyError` exceptions.
- Useful for optional keys.
- Can be used in conditional logic.
- Enhances code safety and readability.

🎹 `items()` Method

- Returns a view object of key-value pairs.
- Can be converted to a list or used in loops.
- Syntax: `dict.items()`
- Example:

```python
student = {"name": "John", "grade": "A"}
for key, value in student.items():
    print(key, value)
```

- Output:

```
name John
grade A
```

- Useful for iterating over dictionaries.
- View updates with dictionary changes.
- Efficient for inspection and debugging.

`keys()` Method

- Returns a view object of all keys.
- Syntax: `dict.keys()`
- Example:

```python
student = {"name": "John", "grade": "A"}
print(list(student.keys()))
```

- Output: `['name', 'grade']`
- Can be used to check key existence.
- View reflects live changes.
- Ideal for looping through keys.
- Helps in filtering or mapping operations.

`pop()` Method

- Removes a key and returns its value.
- Raises `KeyError` if key is missing.
- Syntax: `dict.pop(key, default)`
- Example:

```python
student = {"name": "John", "grade": "A"}
grade = student.pop("grade")
print(grade)
print(student)
```

- Output:

```
A
{'name': 'John'}
```

- Use default to avoid errors.
- Useful for extracting and deleting data.

`update()` Method

- Adds or modifies key-value pairs from another dictionary or iterable.
- Syntax: `dict.update(other_dict)`
- Example:

```python
student = {"name": "John"}
student.update({"grade": "A", "age": 20})
print(student)
```

- Output: `{'name': 'John', 'grade': 'A', 'age': 20}`
- Overwrites existing keys.
- Accepts keyword arguments too.
- Efficient for merging dictionaries.

`values()` Method

- Returns a view object of all values.
- Syntax: `dict.values()`
- Example:

```python
student = {"name": "John", "grade": "A"}
print(list(student.values()))
```

- Output: `['John', 'A']`
- Useful for value-based operations.
- View updates with dictionary changes.
- Can be used to check for duplicates.
- Ideal for data extraction.