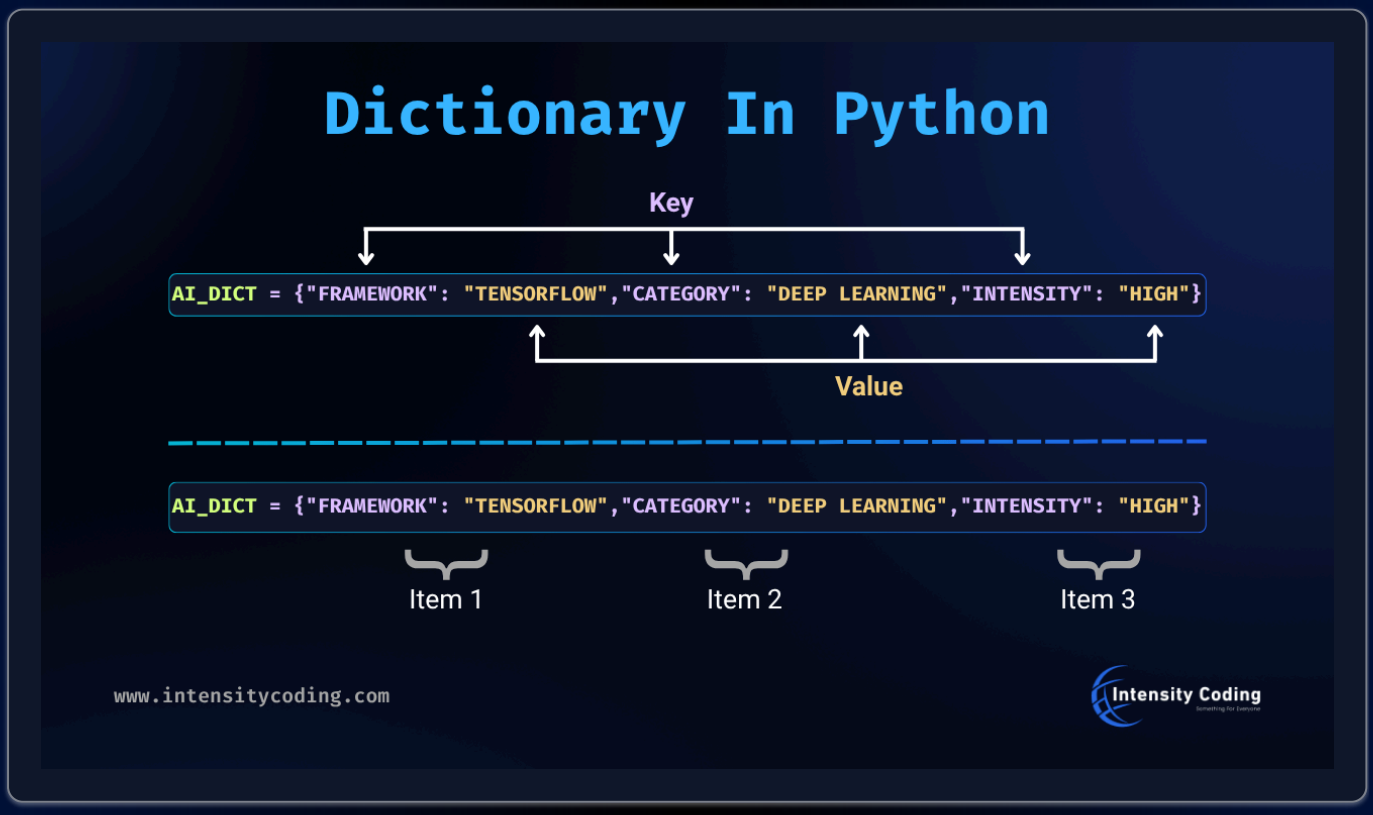
PYTHON

Python Dictionaries

Python

Introduction to Dictionary

- A **dictionary** is a collection that stores data in **key-value pairs**.
- It is **ordered** (as of Python 3.7), **changeable**, and **does not allow duplicate keys**.
- Dictionaries are defined using **curly brackets** `{}` or the `dict()` **constructor**.



Dictionary Items

- Dictionary items are **ordered**, **modifiable**, and **unique** in terms of keys.
- Values can be **of any data type**: strings, numbers, lists, booleans, or even other dictionaries.
- Items are accessed using **key names**.

```
# Example : Define a dictionary with AI-related fields
ai_dict = {
    "framework": "TensorFlow",
    "category": "deep learning",
    "intensity": "high"
}

# Print the dictionary
print(ai_dict)
# Output: {'framework': 'TensorFlow', 'category': 'deep learning',
'intensity': 'high'}

# Accessing a value using a key
print(ai_dict["framework"])  # Output: TensorFlow
```

```
{'framework': 'TensorFlow', 'category': 'deep learning', 'intensity':
'high'}
TensorFlow
```

Handling Duplicate Keys

- If a dictionary contains **duplicate keys**, the last occurrence **overwrites** the previous ones.

```
# Example: Define a dictionary with duplicate keys
ai_model = {
    "name": "GPT",
    "version": 3,
    "version": 4  # Overwrites the previous version key
```

```
}

# Print the dictionary
print(ai_model)
# Output: {'name': 'GPT', 'version': 4}
```

```
{'name': 'GPT', 'version': 4}
```

Dictionary Length

- Use the `len()` **function** to get the number of key-value pairs.

```
intensity_data = {"low": 10,"medium": 50,"high": 100}

# Get the length of the dictionary
print(len(intensity_data))  # Output: 3
```

```
3
```

Different Data Types in a Dictionary

- Dictionary values can be **strings, integers, lists, booleans, or even nested dictionaries**.

```
# Define a dictionary with mixed data types
ml_dict = {
    "algorithm": "Neural Network",
    "layers": 5,
    "active": True,
    "optimizers": ["SGD", "Adam", "RMSprop"]
}

# Print the dictionary
print(ml_dict)
```

```
# Output: {'algorithm': 'Neural Network', 'layers': 5, 'active': True,
'optimizers': ['SGD', 'Adam', 'RMSprop']}
```

```
{'algorithm': 'Neural Network', 'layers': 5, 'active': True,
'optimizers': ['SGD', 'Adam', 'RMSprop']}
```

Checking Dictionary Type

- Use the `type()` **function** to verify that an object is a dictionary.

```
# Define a dictionary
ai_params = {"learning_rate": 0.01,"batch_size": 32,"epochs": 10}

# Check the type
print(type(ai_params))  # Output: <class 'dict'>
```

```
<class 'dict'>
```

Converting a Dictionary to a String

- The `str()` **function** converts a dictionary into a printable string.

```
# Define a dictionary
dataset_info = {"name": "ImageNet", "classes": 1000}

# Convert dictionary to string
print("Dictionary as String:", str(dataset_info))
# Output: Dictionary as String: {'name': 'ImageNet', 'classes': 1000}
```

```
Dictionary as String: {'name': 'ImageNet', 'classes': 1000}
```

Creating a Dictionary Using the dict() Constructor

- The `dict()` **function** can be used to create a dictionary.

```
# Create a dictionary using dict() constructor
user_info = dict(name="John", age=30, domain="intensity coding")

# Print the dictionary
print(user_info)
# Output: {'name': 'John', 'age': 30, 'domain': 'intensity coding'}
```

```
{'name': 'John', 'age': 30, 'domain': 'intensity coding'}
```

Accessing Dictionary Items in Python

Accessing Values Using Keys

- Dictionary items can be accessed by referring to their **key names** inside square brackets `[]`.
- Alternatively, the `get()` **method** can be used to achieve the same result.
- If you attempt to access a key that doesn't exist, you'll get an error in case of `[]` while get() method returns None instead of throwing a KeyError.

```
# Define a dictionary with AI-related terms
ai_dict = {
    "framework": "TensorFlow",
    "category": "deep learning",
    "intensity": "high"
}

# Access value using key
x = ai_dict["framework"]
print(x)  # Output: TensorFlow
```

```
# Access value using get() method
y = ai_dict.get("category")
print(y)  # Output: deep learning
```

```
TensorFlow
deep learning
```

Retrieving All Keys

- The `keys()` **method** returns a view object containing all the dictionary's keys.
- This view is **dynamic**, meaning changes in the dictionary will be reflected in the keys list.

```
# Define a dictionary
ml_model = {
    "name": "GPT-4",
    "type": "Transformer",
    "trained_on": "large dataset"
}

# Get all keys
keys_list = ml_model.keys()
print(keys_list)
# Output: dict_keys(['name', 'type', 'trained_on'])

# Adding a new key-value pair updates the keys list dynamically
ml_model["accuracy"] = 98.5
print(keys_list)
# Output: dict_keys(['name', 'type', 'trained_on', 'accuracy'])
```

```
dict_keys(['name', 'type', 'trained_on'])
dict_keys(['name', 'type', 'trained_on', 'accuracy'])
```

Retrieving All Values

- The `values()` **method** returns a view object containing all the dictionary's values.
- Changes in the dictionary will be **automatically reflected** in the values list.

```
# Define a dictionary
dataset_info = {
    "name": "ImageNet",
    "classes": 1000,
    "size": "1.2M images"
}

# Get all values
values_list = dataset_info.values()
print(values_list)
# Output: dict_values(['ImageNet', 1000, '1.2M images'])

# Modifying an existing value updates the values list dynamically
dataset_info["size"] = "1.5M images"
print(values_list)
# Output: dict_values(['ImageNet', 1000, '1.5M images'])

# Adding a new key-value pair updates the values list dynamically
dataset_info["resolution"] = "256×256"
print(values_list)
# Output: dict_values(['ImageNet', 1000, '1.5M images', '256×256'])
```

```
dict_values(['ImageNet', 1000, '1.2M images'])
dict_values(['ImageNet', 1000, '1.5M images'])
dict_values(['ImageNet', 1000, '1.5M images', '256×256'])
```

Retrieving All Items (Key-Value Pairs)

- The `items()` **method** returns a view object containing **each dictionary entry as a tuple**.
- Changes in the dictionary will be reflected in the items list.

```
# Define a dictionary
hyperparams = {
    "learning_rate": 0.01,
    "batch_size": 32,
```

```
    "epochs": 10
}

# Get all key-value pairs
items_list = hyperparams.items()
print(items_list)
# Output: dict_items([('learning_rate', 0.01), ('batch_size', 32),
('epochs', 10)])

# Updating a value dynamically updates the items list
hyperparams["epochs"] = 20
print(items_list)
# Output: dict_items([('learning_rate', 0.01), ('batch_size', 32),
('epochs', 20)])

# Adding a new key-value pair updates the items list dynamically
hyperparams["optimizer"] = "Adam"
print(items_list)
# Output: dict_items([('learning_rate', 0.01), ('batch_size', 32),
('epochs', 20), ('optimizer', 'Adam')])
```

```
dict_items([('learning_rate', 0.01), ('batch_size', 32), ('epochs',
10)])
dict_items([('learning_rate', 0.01), ('batch_size', 32), ('epochs',
20)])
dict_items([('learning_rate', 0.01), ('batch_size', 32), ('epochs',
20), ('optimizer', 'Adam')])
```

Checking if a Key Exists

- Use the `in` **keyword** to check whether a key exists in a dictionary.

```
# Define a dictionary
model_specs = {
    "name": "ResNet",
    "layers": 50,
    "pretrained": True
}

# Check if a key exists
```

```
if "layers" in model_specs:
    print("Yes, 'layers' is one of the keys in the model_specs
dictionary")
# Output: Yes, 'layers' is one of the keys in the model_specs dictionary
```

```
Yes, 'layers' is one of the keys in the model_specs dictionary
```

Changing Dictionary Items in Python

Modifying Values in a Dictionary

- You can update the value of a specific key by referring to it directly.
- This allows modifying existing data dynamically.

```
# Define a dictionary with AI-related terms
ai_project = {
    "name": "Intensity Coding",
    "language": "Python",
    "year": 2023
}

# Adding new key-value pairs
ai_project["version"] = 1.1

# Modify the value of an existing key
ai_project["year"] = 2025

print(ai_project)
# Output: {'name': 'Intensity Coding', 'language': 'Python', 'year':
2025, 'version': 1.1}
```

```
{'name': 'Intensity Coding', 'language': 'Python', 'year': 2025,
'version': 1.1}
```

Updating a Dictionary Using update()

- The `update()` **method** allows updating a dictionary with values from another dictionary or an iterable object containing key-value pairs.
- This is useful when updating multiple values at once.

```
# Define a dictionary
ml_framework = {
    "framework": "TensorFlow",
    "version": "2.10",
    "status": "stable"
}

# Update dictionary using another dictionary
ml_framework.update({"version": "2.12", "status": "latest"})

print(ml_framework)
# Output: {'framework': 'TensorFlow', 'version': '2.12', 'status':
'latest'}
```

```
{'framework': 'TensorFlow', 'version': '2.12', 'status': 'latest'}
```

Updating Using an Iterable (Tuple of Key-Value Pairs)

```
# Define a dictionary
dataset = {
    "name": "ImageNet",
    "classes": 1000,
    "size": "1.2M images"
}

# Update dictionary using an iterable (tuple inside a list)
dataset.update([("size", "1.5M images"), ("source", "Intensity Coding")])

print(dataset)
```

```
# Output: {'name': 'ImageNet', 'classes': 1000, 'size': '1.5M images',
'source': 'Intensity Coding'}
```

```
{'name': 'ImageNet', 'classes': 1000, 'size': '1.5M images',
'source': 'Intensity Coding'}
```

```

```

Adding Items to a Dictionary in Python

Adding a New Key-Value Pair

- You can add new items to a dictionary by assigning a value to a **new key**.
- If the key already exists, the value will be updated.

```
# Define a dictionary
ai_project = {
    "name": "Intensity Coding",
    "language": "Python",
    "year": 2023
}

# Adding a new key-value pair
ai_project["domain"] = "Machine Learning"

print(ai_project)
# Output: {'name': 'Intensity Coding', 'language': 'Python', 'year':
2023, 'domain': 'Machine Learning'}
```

```
{'name': 'Intensity Coding', 'language': 'Python', 'year': 2023,
'domain': 'Machine Learning'}
```

Updating a Dictionary Using update()

- The `update()` **method** updates the dictionary with values from another dictionary or an iterable object containing key-value pairs.
- If the key already exists, its value is updated; otherwise, the key is added.

```
# Define a dictionary
dataset_info = {
    "dataset": "ImageNet",
    "size": "1.2M images"
}

# Adding a new key-value pair using update()
dataset_info.update({"source": "Intensity Coding", "year": 2025})

print(dataset_info)
# Output: {'dataset': 'ImageNet', 'size': '1.2M images', 'source':
'Intensity Coding', 'year': 2025}
```

```
{'dataset': 'ImageNet', 'size': '1.2M images', 'source': 'Intensity
Coding', 'year': 2025}
```

Updating Using an Iterable (Tuple of Key-Value Pairs)

```
# Define a dictionary
ml_framework = {
    "framework": "PyTorch",
    "version": "1.12"
}

# Updating with an iterable (list of tuples)
ml_framework.update([("version", "2.0"), ("developer", "Meta")])
```

```
print(ml_framework)
# Output: {'framework': 'PyTorch', 'version': '2.0', 'developer': 'Meta'}
```

```
{'framework': 'PyTorch', 'version': '2.0', 'developer': 'Meta'}
```

Removing Items from a Dictionary in Python

- Python provides multiple methods to remove items from a dictionary:
  - `pop(key)` → Removes a specified key and returns its value.
  - `popitem()` → Removes the last inserted item
  - `del` → Removes a specific key or deletes the dictionary entirely.
  - `clear()` → Empties the dictionary.

Removing a Specific Item Using pop()

- Removes the key-value pair of the specified key.
- If the key does not exist, it raises a `KeyError`.

```
# Define a dictionary
ai_project = {
    "name": "Intensity Coding",
    "language": "Python",
    "year": 2023
}

# Remove the "language" key
ai_project.pop("language")

print(ai_project)
# Output: {'name': 'Intensity Coding', 'year': 2023}
```

```
{'name': 'Intensity Coding', 'year': 2023}
```

Removing the Last Inserted Item Using popitem()

- Removes and returns the last added key-value pair.

```
# Define a dictionary
dataset_info = {
    "dataset": "ImageNet",
    "size": "1.2M images",
    "source": "Intensity Coding"
}

# Remove the last inserted item
dataset_info.popitem()

print(dataset_info)
# Output: {'dataset': 'ImageNet', 'size': '1.2M images'}
```

```
{'dataset': 'ImageNet', 'size': '1.2M images'}
```

Removing a Specific Item Using del

- The `del` keyword removes an item based on its key.

```
# Define a dictionary
ml_framework = {
    "framework": "TensorFlow",
    "version": "2.8",
    "developer": "Google"
}

# Delete the "developer" key
del ml_framework["developer"]

print(ml_framework)
# Output: {'framework': 'TensorFlow', 'version': '2.8'}
```

```
{'framework': 'TensorFlow', 'version': '2.8'}
```

Using `del` to Delete the Entire Dictionary

```
# Define a dictionary
model_params = {
    "layers": 5,
    "activation": "ReLU",
    "optimizer": "Adam"
}

# Delete the entire dictionary
del model_params

# print(model_params)  # This will raise a NameError since the dictionary
no longer exists.
```

Clearing a Dictionary Using clear()

- The `clear()` method removes all key-value pairs, leaving an empty dictionary.

```
# Define a dictionary
hyperparameters = {
    "batch_size": 32,
    "learning_rate": 0.001,
    "epochs": 50
}

# Clear the dictionary
hyperparameters.clear()

print(hyperparameters)
# Output: {}
```

```
{}
```

Looping Through a Dictionary in Python

- When iterating through a dictionary, you can access:
  - **Keys**
  - **Values**
  - **Key-Value pairs**

Looping Through Keys

- By default, iterating over a dictionary returns its keys.

```
# Define a dictionary
ai_project = {
    "name": "Intensity Coding",
    "language": "Python",
    "year": 2023
}

# Loop through keys
for key in ai_project:
    print(key)

# Output:
# name
# language
# year
```

```
name
language
year
```

Looping Through Values

- To iterate over values, use the `.values()` method.

```python
# Define a dictionary
ml_framework = {
    "framework": "TensorFlow",
    "version": "2.8",
    "developer": "Google"
}

# Loop through values
for value in ml_framework.values():
    print(value)

# Output:
# TensorFlow
# 2.8
# Google
```

```
TensorFlow
2.8
Google
```

Looping Through Keys and Accessing Values

- You can retrieve values inside the loop by accessing `dict[key]`.

```python
# Define a dictionary
dataset_info = {
    "dataset": "ImageNet",
    "size": "1.2M images",
    "source": "Intensity Coding"
}

# Loop through dictionary and access values
for key in dataset_info:
    print(dataset_info[key])

# Output:
# ImageNet
```

```
# 1.2M images
# Intensity Coding
```

```
ImageNet
1.2M images
Intensity Coding
```

Looping Through Keys Using .keys()

- The `.keys()` method explicitly returns dictionary keys.

```
# Define a dictionary
hyperparameters = {
    "batch_size": 32,
    "learning_rate": 0.001,
    "epochs": 50
}

# Loop through keys
for key in hyperparameters.keys():
    print(key)

# Output:
# batch_size
# learning_rate
# epochs
```

```
batch_size
learning_rate
epochs
```

Looping Through Key-Value Pairs Using .items()

- The `.items()` method returns both **keys and values** as tuples.

```python
# Define a dictionary
model_params = {
    "layers": 5,
    "activation": "ReLU",
    "optimizer": "Adam"
}

# Loop through key-value pairs
for key, value in model_params.items():
    print(key, ":", value)

# Output:
# layers : 5
# activation : ReLU
# optimizer : Adam
```

```
layers : 5
activation : ReLU
optimizer : Adam
```

Copying Dictionaries in Python

- When copying a dictionary, it is important to **avoid direct assignment** (`dict2 = dict1`), as it creates a **reference** instead of a true copy.

The Problem with Direct Assignment

- If you assign one dictionary to another using `=`, both variables **point to the same object**. Changes made to one will reflect in the other.

```python
# Define a dictionary
ai_project = {
    "name": "Intensity Coding",
    "language": "Python",
    "year": 2023
}
```

```
# Assign directly (creates reference)
new_project = ai_project

# Modify the new dictionary
new_project["year"] = 2025

# Print both dictionaries
print(ai_project)
print(new_project)

# Output:
# {'name': 'Intensity Coding', 'language': 'Python', 'year': 2025}
# {'name': 'Intensity Coding', 'language': 'Python', 'year': 2025}
# Both dictionaries are affected because they refer to the same object.
```

```
{'name': 'Intensity Coding', 'language': 'Python', 'year': 2025}
{'name': 'Intensity Coding', 'language': 'Python', 'year': 2025}
```

Using copy() Method

- The `.copy()` method creates a **true copy** of the dictionary.

```
# Define a dictionary
dataset_info = {
    "dataset": "ImageNet",
    "size": "1.2M images",
    "source": "Intensity Coding"
}

# Create a copy using copy()
dataset_copy = dataset_info.copy()

# Modify the copied dictionary
dataset_copy["size"] = "2.0M images"

# Print both dictionaries
print(dataset_info)
print(dataset_copy)
```

```
# Output:
# {'dataset': 'ImageNet', 'size': '1.2M images', 'source': 'Intensity
Coding'}
# {'dataset': 'ImageNet', 'size': '2.0M images', 'source': 'Intensity
Coding'}
# Original dictionary remains unchanged.
```

```
{'dataset': 'ImageNet', 'size': '1.2M images', 'source': 'Intensity
Coding'}
{'dataset': 'ImageNet', 'size': '2.0M images', 'source': 'Intensity
Coding'}
```

Using dict() Constructor

- Another way to copy a dictionary is by using the `dict()` function.

```
# Define a dictionary
ml_framework = {
    "framework": "TensorFlow",
    "version": "2.8",
    "developer": "Google"
}

# Create a copy using dict()
framework_copy = dict(ml_framework)

# Modify the copied dictionary
framework_copy["version"] = "3.0"

# Print both dictionaries
print(ml_framework)
print(framework_copy)

# Output:
# {'framework': 'TensorFlow', 'version': '2.8', 'developer': 'Google'}
# {'framework': 'TensorFlow', 'version': '3.0', 'developer': 'Google'}
# Original dictionary remains unchanged.
```

```
{'framework': 'TensorFlow', 'version': '2.8', 'developer': 'Google'}
{'framework': 'TensorFlow', 'version': '3.0', 'developer': 'Google'}
```

Deep Copy vs. Shallow Copy

- The `copy()` and `dict()` methods create a **shallow copy**.
- If the dictionary contains **nested dictionaries**, changes in the nested dictionaries **affect both copies**.
- To perform a **deep copy**, use the `copy.deepcopy()` method from the `copy` module.

```
import copy

# Define a nested dictionary
model_config = {
    "model": "ResNet",
    "parameters": {
        "layers": 50,
        "activation": "ReLU"
    }
}

# Create a deep copy
model_copy = copy.deepcopy(model_config)

# Modify the copied dictionary
model_copy["parameters"]["layers"] = 101

# Print both dictionaries
print(model_config)
print(model_copy)

# Output:
# {'model': 'ResNet', 'parameters': {'layers': 50, 'activation': 'ReLU'}}
# {'model': 'ResNet', 'parameters': {'layers': 101, 'activation':
'ReLU'}}
# The original dictionary remains unchanged.
```

```
{'model': 'ResNet', 'parameters': {'layers': 50, 'activation':
'ReLU'}}
{'model': 'ResNet', 'parameters': {'layers': 101, 'activation':
'ReLU'}}
```

Nested Dictionaries in Python

- A **nested dictionary** is a dictionary inside another dictionary. It is useful for organizing **complex data structures** such as configurations, datasets, and hierarchies.

```python
# Nested dictionary containing AI models
ai_models = {
    "model1": {
        "name": "GPT-4",
        "parameters": 175e9,
        "developer": "OpenAI"
    },
    "model2": {
        "name": "BERT",
        "parameters": 340e6,
        "developer": "Google"
    },
    "model3": {
        "name": "Stable Diffusion",
        "parameters": 890e6,
        "developer": "Stability AI"
    }
}

# Print the dictionary
print(ai_models)
```

```
{'model1': {'name': 'GPT-4', 'parameters': 175000000000.0,
'developer': 'OpenAI'}, 'model2': {'name': 'BERT', 'parameters':
340000000.0, 'developer': 'Google'}, 'model3': {'name': 'Stable
Diffusion', 'parameters': 890000000.0, 'developer': 'Stability AI'}}
```

Creating a Nested Dictionary Using Separate Dictionaries

- You can also define **individual dictionaries** first and then **combine** them.

```
# Individual dictionaries for datasets
dataset1 = {
    "name": "ImageNet",
    "size": "1.2M images",
    "source": "Intensity Coding"
}

dataset2 = {
    "name": "COCO",
    "size": "330K images",
    "source": "Microsoft"
}

dataset3 = {
    "name": "MNIST",
    "size": "60K images",
    "source": "Yann LeCun"
}

# Combine into a nested dictionary
datasets = {
    "dataset1": dataset1,
    "dataset2": dataset2,
    "dataset3": dataset3
}

# Print the nested dictionary
print(datasets)
```

```
{'dataset1': {'name': 'ImageNet', 'size': '1.2M images', 'source':
'Intensity Coding'}, 'dataset2': {'name': 'COCO', 'size': '330K
images', 'source': 'Microsoft'}, 'dataset3': {'name': 'MNIST',
'size': '60K images', 'source': 'Yann LeCun'}}
```

Accessing Items in Nested Dictionaries

- To access values inside a **nested dictionary**, use **multiple square brackets (`[]`)**.

```
# Access the name of model2
print(ai_models["model2"]["name"])
# Output: BERT

# Access the parameter count of model1
print(ai_models["model1"]["parameters"])
# Output: 175000000000.0
```

```
BERT
175000000000.0
```

Modifying Nested Dictionary Values

- You can update nested dictionaries just like normal dictionaries.

```
# Updating ImageNet dataset size
datasets["dataset1"]["size"] = "1.5M images"

# Print updated dataset info
print(datasets["dataset1"])
```

```
{'name': 'ImageNet', 'size': '1.5M images', 'source': 'Intensity
```

```
Coding'}
```

Looping Through a Nested Dictionary

- To iterate over a **nested dictionary**, use **nested loops**.

```python
for key, model in ai_models.items():
    print(f"Model ID: {key}")
    for attribute, value in model.items():
        print(f"  {attribute}: {value}")
    print("-" * 30)

# Output:
#   Model ID: model1
#   name: GPT-4
#   parameters: 175000000000.0
#   developer: OpenAI
# ------------------------------
# Model ID: model2
#   name: BERT
#   parameters: 340000000.0
#   developer: Google
# ------------------------------
# Model ID: model3
#   name: Stable Diffusion
#   parameters: 890000000.0
#   developer: Stability AI
# ------------------------------
```

```
Model ID: model1
  name: GPT-4
  parameters: 175000000000.0
  developer: OpenAI
------------------------------
Model ID: model2
  name: BERT
  parameters: 340000000.0
  developer: Google
------------------------------
```

```
Model ID: model3
  name: Stable Diffusion
  parameters: 890000000.0
  developer: Stability AI
_____________________________
```

Python Dictionary Methods

No	Method	Description
1	`clear()`	Deletes all key-value pairs, making the dictionary empty.
2	`copy()`	Creates and returns a duplicate of the dictionary.
3	`fromkeys()`	Creates a new dictionary with specified keys, assigning a common default value.
4	`get()`	Retrieves the value associated with a key; returns `None` if the key is absent.
5	`items()`	Returns all dictionary entries as a collection of key-value tuple pairs.
6	`keys()`	Provides a view containing all the keys in the dictionary.
7	`pop()`	Removes a key-value pair by key and returns its value.
8	`popitem()`	Removes and returns the most recently added key-value pair.
9	`setdefault()`	Returns the value of a key; if missing, inserts it with a specified default.
10	`update()`	Merges another dictionary or iterable into the existing dictionary, updating keys if they exist.
11	`values()`	Returns a view object containing all dictionary values.

1. clear()

Point	Description
Use Case	The `clear()` method removes all key-value pairs from a dictionary, leaving it empty.
Syntax	`dictionary.clear()`
Parameter Values	This method does not accept any parameters.

Useful Information

No.	Point	Description
1	Modifies Original Dictionary	The method empties the dictionary in place instead of returning a new one.
2	No Return Value	It does not return anything; it simply clears the dictionary.
3	Dictionary Still Exists	The dictionary remains but becomes empty after calling `clear()`.

```
# Dictionary storing AI model configurations
ai_model_config = {
    "model": "Transformer",
    "layers": 12,
    "learning_rate": 0.001,
    "optimizer": "Adam"
}

# Clearing all configurations
ai_model_config.clear()

# Printing the dictionary after clearing
print("AI Model Configurations:", ai_model_config)  # Output: AI Model
Configurations: {}
```

```
AI Model Configurations: {}
```

2. copy()

Point	Description
Use Case	The `copy()` method creates and returns a shallow copy of a dictionary.
Syntax	`dictionary.copy()`
Parameter Values	This method does not accept any parameters.

Useful Information

No.	Point	Description
1	Shallow Copy	The copied dictionary is independent but contains references to mutable objects.
2	Does Not Modify Original	The original dictionary remains unchanged after copying.
3	Deep Copy Required for Nested Dictionaries	If the dictionary contains nested structures, use `copy.deepcopy()` for a deep copy.

```python
import copy

# Original dictionary storing AI dataset metadata
dataset_info = {
    "name": "ImageNet",
    "size": "150GB",
    "categories": ["Animals", "Objects", "Scenes"]
}

# Creating a shallow copy
dataset_copy = dataset_info.copy()

# Modifying the copied dictionary
dataset_copy["name"] = "CIFAR-10"

# Printing original and copied dictionaries
print("Original Dataset Info:", dataset_info)
```

```
# Output: {'name': 'ImageNet', 'size': '150GB', 'categories': ['Animals',
'Objects', 'Scenes']}

print("Copied Dataset Info:", dataset_copy)
# Output: {'name': 'CIFAR-10', 'size': '150GB', 'categories': ['Animals',
'Objects', 'Scenes']}
```

```
Original Dataset Info: {'name': 'ImageNet', 'size': '150GB',
'categories': ['Animals', 'Objects', 'Scenes']}
Copied Dataset Info: {'name': 'CIFAR-10', 'size': '150GB',
'categories': ['Animals', 'Objects', 'Scenes']}
```

3. fromkeys()

Point	Description
Use Case	The `fromkeys()` method creates a new dictionary with the given keys, all assigned to the same specified value.
Syntax	`dict.fromkeys(keys, value)`
Parameter Values	`keys`: Required. An iterable specifying the dictionary keys.
	`value`: Optional. The value assigned to all keys (default is `None`).

Useful Information

No.	Point	Description
1	Same Value for All Keys	All keys receive the same value provided.
2	Default Value is `None`	If the value is not specified, all keys are assigned `None`.
3	Mutable Value Reference	If a mutable object (like a list) is used as the value, all keys share the same reference.

```python
# Defining a tuple of parameter names
parameters = ("learning_rate", "batch_size", "epochs")

# Assigning a default value of 0 to all parameters
default_values = 0
model_config = dict.fromkeys(parameters, default_values)

print(model_config)
# Output: {'learning_rate': 0, 'batch_size': 0, 'epochs': 0}

# Example without specifying a default value (assigns None)
default_config = dict.fromkeys(parameters)

print(default_config)
# Output: {'learning_rate': None, 'batch_size': None, 'epochs': None}
```

```
{'learning_rate': 0, 'batch_size': 0, 'epochs': 0}
{'learning_rate': None, 'batch_size': None, 'epochs': None}
```

4. get()

Point	Description
Use Case	The `get()` method retrieves the value associated with a specified key in a dictionary. If the key is not found, it returns a default value instead of raising an error.
Syntax	`dictionary.get(keyname, value)`
Parameter Values	`keyname`: Required. The key whose value needs to be fetched.
	`value`: Optional. A default value returned if the key does not exist (default is `None`).

Useful Information

No.	Point	Description
1	Avoids Key Errors	Unlike `dictionary[key]`, `get()` does not raise a `KeyError` if the key is missing.
2	Allows Default Values	You can specify a default value to return if the key is not found.
3	Returns `None` by Default	If no default value is provided and the key does not exist, `None` is returned.

```python
# Dictionary representing AI model configurations
model_config = {
    "optimizer": "adam",
    "learning_rate": 0.001,
    "batch_size": 32
}

# Fetching the learning rate
lr = model_config.get("learning_rate")
print(lr)
# Output: 0.001

# Fetching a non-existent key with a default value
dropout = model_config.get("dropout_rate", 0.2)
print(dropout)
# Output: 0.2

# Fetching a key that does not exist without a default value
momentum = model_config.get("momentum")
print(momentum)
# Output: None
```

```
0.001
0.2
None
```

5. items()

Point	Description
Use Case	The `items()` method returns a view object containing all key-value pairs of a dictionary as tuples. This view object dynamically reflects changes to the dictionary.
Syntax	`dictionary.items()`
Parameter Values	No parameters.

No.	Point	Description
1	Returns a View Object	The returned view object reflects real-time changes made to the dictionary.
2	Contains Tuples	The key-value pairs are returned as a list of tuples.
3	Useful for Iteration	The `items()` method is commonly used for iterating over dictionary elements.

```
# Dictionary representing a machine learning model's parameters
model_params = {
    "optimizer": "adam",
    "learning_rate": 0.001,
    "batch_size": 32
}

# Retrieving all key-value pairs as tuples
params_view = model_params.items()
print(params_view)
# Output: dict_items([('optimizer', 'adam'), ('learning_rate', 0.001),
('batch_size', 32)])

# Modifying the dictionary after calling items()
model_params["learning_rate"] = 0.002
print(params_view)
# Output: dict_items([('optimizer', 'adam'), ('learning_rate', 0.002),
('batch_size', 32)])
```

```
dict_items([('optimizer', 'adam'), ('learning_rate', 0.001),
('batch_size', 32)])
```

```
dict_items([('optimizer', 'adam'), ('learning_rate', 0.002),
('batch_size', 32)])
```

6. keys()

Point	Description
Use Case	The `keys()` method returns a view object containing all dictionary keys. This view dynamically updates when the dictionary is modified.
Syntax	`dictionary.keys()`
Parameter Values	No parameters.

Useful Information

No.	Point	Description
1	Returns a View Object	The returned object reflects any real-time changes made to the dictionary.
2	Contains Only Keys	It provides access to dictionary keys without values.
3	Useful for Iteration	Commonly used for looping over dictionary keys.

```python
# Dictionary representing a dataset's features
dataset_features = {
    "feature_1": "age",
    "feature_2": "height",
    "feature_3": "weight"
}

# Retrieving dictionary keys
keys_view = dataset_features.keys()
print(keys_view)
# Output: dict_keys(['feature_1', 'feature_2', 'feature_3'])

# Modifying the dictionary after calling keys()
dataset_features["feature_4"] = "blood_pressure"
```

```
print(keys_view)
# Output: dict_keys(['feature_1', 'feature_2', 'feature_3', 'feature_4'])
```

```
dict_keys(['feature_1', 'feature_2', 'feature_3'])
dict_keys(['feature_1', 'feature_2', 'feature_3', 'feature_4'])
```

7. pop()

Point	Description
Use Case	The `pop()` method removes a specified key-value pair from a dictionary and returns the value of the removed item.
Syntax	`dictionary.pop(keyname, defaultvalue)`
Parameter Values	`keyname` : **(Required)** The key of the item to remove.
	`defaultvalue` : **(Optional)** Value to return if the key does not exist. If omitted and the key is missing, a `KeyError` is raised.

Useful Information

No.	Point	Description
1	Removes a Specific Item	Deletes the key-value pair and returns the value of the removed item.
2	Raises an Error if Key is Missing	If the key is not found and no default value is provided, a `KeyError` is raised.
3	Default Value Prevents Errors	Supplying a default value avoids errors when the key does not exist.

```
# Dictionary representing a dataset with model details
model_info = {
    "model_name": "NeuralNet",
    "accuracy": 92.5,
    "parameters": 150000
```

```
}

# Removing a key and displaying the updated dictionary
model_info.pop("accuracy")
print(model_info)
# Output: {'model_name': 'NeuralNet', 'parameters': 150000}

# Removing a key and storing the removed value
removed_value = model_info.pop("parameters")
print(removed_value)
# Output: 150000

# Attempting to remove a non-existent key with a default value
default_output = model_info.pop("layers", "Not Available")
print(default_output)
# Output: Not Available
```

```
{'model_name': 'NeuralNet', 'parameters': 150000}
150000
Not Available
```

8. popitem()

Point	Description
Use Case	The `popitem()` method removes and returns the last inserted key-value pair from a dictionary as a tuple.
Syntax	`dictionary.popitem()`
Parameter Values	No parameters.

Useful Information

No.	Point	Description
1	Removes the Last Inserted Item	From Python 3.7+, it removes the last added key-value pair. Before Python 3.7, it removed a random item.

No.	Point	Description
2	Returns a Tuple	The removed key-value pair is returned as a tuple `(key, value)`.
3	Raises an Error if Empty	If the dictionary is empty, calling `popitem()` results in a `KeyError`.

```python
# Dictionary representing dataset metadata
dataset_info = {
    "dataset_name": "ImageNet",
    "num_samples": 1000000,
    "num_classes": 1000
}

# Removing the last inserted item
dataset_info.popitem()
print(dataset_info)
# Output: {'dataset_name': 'ImageNet', 'num_samples': 1000000}

# Removing another item and storing the removed key-value pair
removed_entry = dataset_info.popitem()
print(removed_entry)
# Output: ('num_samples', 1000000)

# Attempting to pop from an empty dictionary raises an error
dataset_info.clear()  # Empty the dictionary
# dataset_info.popitem()  # Uncommenting this line will raise KeyError
```

```
{'dataset_name': 'ImageNet', 'num_samples': 1000000}
('num_samples', 1000000)
```

9. setdefault()

Point	Description
Use Case	The `setdefault()` method retrieves the value of a specified key. If the key does not exist, it inserts the key with a given default value and returns it.

Point	Description
Syntax	`dictionary.setdefault(keyname, value)`
Parameter Values	`keyname` : Required. The key whose value needs to be retrieved or inserted.
	`value` : Optional. If the key does not exist, this value is assigned. Default is `None`.

Useful Information

No.	Point	Description
1	No Effect on Existing Keys	If the key is present, `setdefault()` does not change its value.
2	Inserts Key if Missing	If the key does not exist, it is added with the specified default value.
3	Returns the Key's Value	Whether the key exists or not, `setdefault()` always returns the key’s value.

```
# Dictionary representing machine learning model settings
model_config = {
    "optimizer": "Adam",
    "learning_rate": 0.001,
    "epochs": 10
}

# Retrieving an existing key's value
optimizer = model_config.setdefault("optimizer", "SGD")
print(optimizer)
# Output: Adam (existing value remains unchanged)

# Setting a default value for a missing key
batch_size = model_config.setdefault("batch_size", 32)
print(batch_size)
# Output: 32 (since "batch_size" was missing, it is now added)

# Printing the updated dictionary
print(model_config)
# Output: {'optimizer': 'Adam', 'learning_rate': 0.001, 'epochs': 10,
'batch_size': 32}
```

```
Adam
32
{'optimizer': 'Adam', 'learning_rate': 0.001, 'epochs': 10,
'batch_size': 32}
```

10. update()

Point	Description
Use Case	The `update()` method adds key-value pairs from another dictionary or iterable object to an existing dictionary. If a key already exists, its value is updated.
Syntax	`dictionary.update(iterable)`
Parameter Values	`iterable` : A dictionary or an iterable object with key-value pairs to be inserted into the dictionary.

Useful Information

No.	Point	Description
1	Updates Existing Keys	If a key exists, `update()` changes its value.
2	Adds New Key-Value Pairs	If a key does not exist, it is added to the dictionary.
3	Accepts Different Data Types	The method accepts dictionaries, lists of tuples, and other iterables containing key-value pairs.
4	No Return Value	`update()` modifies the dictionary in place and does not return anything.

```python
# Dictionary representing model hyperparameters
model_params = {
    "optimizer": "Adam",
    "learning_rate": 0.001,
    "epochs": 10
}
```

```python
# Updating with new hyperparameters
model_params.update({"batch_size": 32, "learning_rate": 0.0005})

print(model_params)
# Output: {'optimizer': 'Adam', 'learning_rate': 0.0005, 'epochs': 10,
'batch_size': 32}

# Updating using an iterable (list of tuples)
new_params = [("dropout", 0.3), ("activation", "relu")]
model_params.update(new_params)

print(model_params)
# Output: {'optimizer': 'Adam', 'learning_rate': 0.0005, 'epochs': 10,
'batch_size': 32, 'dropout': 0.3, 'activation': 'relu'}
```

```
{'optimizer': 'Adam', 'learning_rate': 0.0005, 'epochs': 10,
'batch_size': 32}
{'optimizer': 'Adam', 'learning_rate': 0.0005, 'epochs': 10,
'batch_size': 32, 'dropout': 0.3, 'activation': 'relu'}
```

11. values()

Point	Description
Use Case	The `values()` method returns a view object containing all values in the dictionary. Any updates to the dictionary reflect in this view.
Syntax	`dictionary.values()`
Parameter Values	No parameters.

Useful Information

No.	Point	Description
1	Dynamic View	The returned object reflects dictionary updates in real time.

No.	Point	Description
2	Not a List	The result is a dictionary view, but can be converted to a list using `list(dictionary.values())`.
3	No Return Modification	The view only allows reading values, not modifying them directly.

```python
# Dictionary representing hyperparameters
model_params = {
    "optimizer": "Adam",
    "learning_rate": 0.001,
    "epochs": 10
}

# Getting values
param_values = model_params.values()
print(param_values)
# Output: dict_values(['Adam', 0.001, 10])

# Updating a dictionary value
model_params["learning_rate"] = 0.0005
print(param_values)
# Output: dict_values(['Adam', 0.0005, 10]) → Reflects the change
```

```
dict_values(['Adam', 0.001, 10])
dict_values(['Adam', 0.0005, 10])
```

Found this helpful ?

Follow on LinkedIn
Master AI/ML with Intensity Coding

@bhavdippatel2020

Like | Comment | Share | Save

Each Article Includes Everything You Need

THEORY MADE SIMPLE
Complex ideas explained in an easy way

PYTHON CODE
Hands-on coding for practice

VISUAL LEARNING
Visual diagrams for clarity

MATH BEHIND AI/ML
Step-by-step explanation of core concepts

Explore more tutorials at Intensity Coding

www.intensitycoding.com

Intensity Coding
Something For Everyone