Python Dictionary Methods

1. Why Dictionary Methods Matter

A Python dictionary stores data as **key-value pairs**:

```
student = {
    "name": "Hafsa",
    "age": 25,
    "course": "Python"
}
```

You can work with this data directly, but dictionary methods make common operations easier and safer.

For example:

```
student.get("email")
```

can safely retrieve a value even when `"email"` does not exist.

Dictionary methods help you:

- access data
- inspect keys and values
- add or update data
- remove data
- copy dictionaries
- safely retrieve missing values
- build cleaner programs

The main methods you should know are:

```
get()
keys()
values()
items()
update()
pop()
popitem()
setdefault()
clear()
copy()
fromkeys()
```

---

2. `get()`

The `get()` method retrieves the value associated with a key.

```
student = {
    "name": "Hafsa",
    "age": 25
}

print(student.get("name"))
```

Output:

```
Hafsa
```

The basic syntax is:

```
dictionary.get(key)
```

---

3. Why Use `get()`?

Suppose you access a missing key directly:

```
student = {
    "name": "Hafsa"
}

print(student["email"])
```

Python raises:

```
KeyError
```

But:

```
print(student.get("email"))
```

returns:

```
None
```

This makes `get()` useful when a key may not exist.

---

4. `get()` With a Default Value

You can provide a fallback value.

```
email = student.get("email", "Not provided")

print(email)
```

Output:

```
Not provided
```

Syntax:

```
dictionary.get(key, default_value)
```

For example:

```
user = {
    "name": "Hafsa"
}

country = user.get("country", "Unknown")

print(country)
```

Output:

```
Unknown
```

---

5. `get()` Does Not Add the Key

This is important.

```
student = {
    "name": "Hafsa"
}

student.get("age", 25)

print(student)
```

The dictionary remains:

```
{
    "name": "Hafsa"
}
```

`get()` only retrieves a value.

It does not create the missing key.

---

6. `keys()`

The `keys()` method returns the dictionary's keys.

```
student = {
    "name": "Hafsa",
    "age": 25,
    "course": "Python"
}

print(student.keys())
```

You will see a dictionary view object similar to:

```
dict_keys(['name', 'age', 'course'])
```

You can loop through it:

```
for key in student.keys():
    print(key)
```

Output:

```
name
age
course
```

---

7. Checking Keys With `keys()`

You can use:

```
if "name" in student.keys():
    print("Name exists")
```

However, you normally don't need `.keys()` for this.

This is simpler:

```
if "name" in student:
    print("Name exists")
```

Both work, but the second version is more common.

---

8. `values()`

The `values()` method returns the dictionary's values.

```
student = {
    "name": "Hafsa",
    "age": 25,
    "course": "Python"
}

print(student.values())
```

Output looks similar to:

```
dict_values(['Hafsa', 25, 'Python'])
```

You can loop through the values:

```
for value in student.values():
    print(value)
```

Output:

```
Hafsa
25
Python
```

---

9. When to Use `values()`

Use `values()` when the keys are not important.

For example:

```
scores = {
    "Hafsa": 92,
    "Asim": 85,
    "Ali": 78
}

for score in scores.values():
    print(score)
```

Output:

```
92
85
78
```

You only care about the scores, not the student names.

---

10. `items()`

The `items()` method returns the dictionary's key-value pairs.

```
student = {
    "name": "Hafsa",
    "age": 25,
    "course": "Python"
}

print(student.items())
```

Output looks similar to:

```
dict_items([
    ('name', 'Hafsa'),
    ('age', 25),
    ('course', 'Python')
])
```

---

11. Looping With `items()`

`items()` is especially useful when you need both the key and value.

```
for key, value in student.items():
    print(key, value)
```

Output:

```
name Hafsa
age 25
course Python
```

You can make it easier to read:

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

Output:

```
name: Hafsa
age: 25
course: Python
```

Mental model

```
keys()   → What are the names?
values() → What are the values?
items()  → What are the names + values together?
```

---

12. `update()`

The `update()` method adds new key-value pairs or changes existing ones.

```
student = {
    "name": "Hafsa",
    "age": 25
}

student.update({
    "course": "Python"
})
```

Now:

```
print(student)
```

Output:

```
{
    'name': 'Hafsa',
    'age': 25,
    'course': 'Python'
}
```

---

13. Updating Existing Values

If a key already exists, `update()` changes its value.

```
student = {
    "name": "Hafsa",
    "age": 25
}

student.update({
    "age": 26
})
```

Now:

```
print(student["age"])
```

Output:

```
26
```

---

14. Updating Multiple Values

You can update several items at once.

```
student.update({
    "age": 26,
    "course": "Python",
    "level": "Intermediate"
})
```

This is useful when multiple pieces of information need to change.

---

15. `update()` With Another Dictionary

You can pass another dictionary:

```
student = {
    "name": "Hafsa",
    "age": 25
}

new_data = {
    "course": "Python",
    "level": "Beginner"
}

student.update(new_data)
```

Now:

```
print(student)
```

contains all four pieces of information.

---

16. `update()` and Conflicting Keys

Suppose:

```
student = {
    "name": "Hafsa",
    "age": 25
}

new_data = {
    "age": 26
}
```

Then:

```
student.update(new_data)
```

The new value replaces the old one:

```
25 → 26
```

The existing key is updated.

---

17. `pop()`

The `pop()` method removes a specific key and returns its value.

```
student = {
    "name": "Hafsa",
    "age": 25,
    "course": "Python"
}

age = student.pop("age")

print(age)
```

Output:

```
25
```

The dictionary is now:

```
{
    "name": "Hafsa",
    "course": "Python"
}
```

---

18. `pop()` With a Default

If the key might not exist, you can provide a default.

```
email = student.pop("email", "Not available")

print(email)
```

Output:

```
Not available
```

Without the default:

```
student.pop("email")
```

would raise a `KeyError` if `"email"` doesn't exist.

---

19. `popitem()`

`popitem()` removes and returns the **last inserted key-value pair**.

```
student = {
    "name": "Hafsa",
    "age": 25,
    "course": "Python"
}

item = student.popitem()

print(item)
```

Output:

```
('course', 'Python')
```

The remaining dictionary is:

```
{
    "name": "Hafsa",
    "age": 25
}
```

---

20. `pop()` vs `popitem()`

Method	Removes

`pop(key)`	A specific key
`popitem()`	The last inserted key-value pair

Example:

```
student.pop("age")
```

removes `"age"`.

While:

```
student.popitem()
```

removes the last inserted pair.

---

21. `clear()`

The `clear()` method removes every item from a dictionary.

```
student = {
    "name": "Hafsa",
    "age": 25,
    "course": "Python"
}

student.clear()

print(student)
```

Output:

```
{}
```

The dictionary still exists.

It is simply empty.

---

22. `clear()` vs `del`

These are different.

`clear()`

```
student.clear()
```

Removes everything but keeps the dictionary variable.

`del`

```
del student
```

Deletes the dictionary variable itself.

After:

```
del student
```

trying to use:

```
print(student)
```

causes a `NameError`.

---

23. `copy()`

The `copy()` method creates a **shallow copy** of a dictionary.

```
student = {
    "name": "Hafsa",
    "age": 25
}

student_copy = student.copy()
```

Now:

```
student_copy["age"] = 26
```

does not change the original dictionary's `"age"` value.

```
print(student["age"])
```

Output:

```
25
```

And:

```
print(student_copy["age"])
```

Output:

```
26
```

---

24. Why `copy()` Matters

Consider this:

```
student = {
    "name": "Hafsa"
}

student_copy = student
```

This does **not** create an independent dictionary.

Both variables refer to the same dictionary.

```
student_copy["age"] = 25

print(student)
```

Output:

```
{
    "name": "Hafsa",
    "age": 25
}
```

The original changed too.

Instead:

```
student_copy = student.copy()
```

creates a separate top-level dictionary.

---

25. `setdefault()`

`setdefault()` gets a value if the key exists.

If the key does not exist, it creates the key with a default value.

```
student = {
    "name": "Hafsa"
}

age = student.setdefault("age", 25)
```

```
print(age)
```

Output:

```
25
```

And the dictionary becomes:

```
{
    "name": "Hafsa",
    "age": 25
}
```

---

26. `setdefault()` When the Key Already Exists

Suppose:

```
student = {
    "name": "Hafsa",
    "age": 25
}
```

Now:

```
student.setdefault("age", 30)
```

The value remains:

```
25
```

It does not replace the existing value with `30`.

This is an important difference between `setdefault()` and direct assignment.

---

27. `setdefault()` vs Direct Assignment

Direct assignment:

```
student["age"] = 30
```

always sets the value to `30`.

`setdefault()`:

```
student.setdefault("age", 30)
```

only adds `30` if `"age"` does not already exist.

Mental model

```
assignment
    ↓
"Make the value this."

setdefault()
    ↓
"If missing, use this value."
```

---

28. `setdefault()` for Grouping Data

One useful pattern is grouping values.

Suppose you have:

```
students = [
    ("Python", "Hafsa"),
    ("Python", "Asim"),
    ("JavaScript", "Ali")
]
```

You can create groups:

```
courses = {}

for course, student in students:
    courses.setdefault(course, []).append(student)

print(courses)
```

Result:

```
{
    "Python": ["Hafsa", "Asim"],
    "JavaScript": ["Ali"]
}
```

This is a practical use of `setdefault()`.

---

29. `fromkeys()`

`fromkeys()` creates a new dictionary using a collection of keys.

Example:

```
keys = ["name", "age", "course"]

student = dict.fromkeys(keys)

print(student)
```

Output:

```
{
    "name": None,
    "age": None,
    "course": None
}
```

The basic syntax is:

```
dict.fromkeys(keys)
```

---

30. `fromkeys()` With a Default Value

You can provide a default value:

```
keys = ["Python", "JavaScript", "React"]

skills = dict.fromkeys(keys, "Beginner")

print(skills)
```

Output:

```
{
    "Python": "Beginner",
    "JavaScript": "Beginner",
    "React": "Beginner"
}
```

This is useful when multiple keys should start with the same value.

---

31. `fromkeys()` Use Case

Imagine creating an empty status record:

```
features = ["login", "search", "profile"]

status = dict.fromkeys(features, False)
```

Result:

```
{
    "login": False,
    "search": False,
    "profile": False
}
```

You can then update individual values:

```
status["login"] = True
```

---

32. Important `fromkeys()` Warning

Be careful when using mutable values such as lists.

For example:

```
keys = ["a", "b", "c"]

data = dict.fromkeys(keys, [])
```

All keys refer to the same list object.

So changing one can affect the others:

```
data["a"].append(10)
```

You may then see:

```
{
    "a": [10],
    "b": [10],
    "c": [10]
}
```

For independent lists, create them separately instead.

---

33. Dictionary Methods and Return Values

Some methods modify the dictionary and return something.

Method	**Main effect**	**Returns**

`get()`	Reads	value/default
`keys()`	Views keys	key view
`values()`	Views values	value view
`items()`	Views pairs	item view
`update()`	Adds/updates	`None`
`pop()`	Removes key	removed value
`popitem()`	Removes last pair	removed pair
`setdefault()`	Reads/adds	value
`clear()`	Removes everything	`None`
`copy()`	Creates copy	new dictionary
`fromkeys()`	Creates dictionary	new dictionary

This matters because you should know whether a method **changes the dictionary**, **returns information**, or **creates a new dictionary**.

---

34. Methods That Return Dictionary Views

These methods:

```
keys()
values()
items()
```

return view objects.

For example:

```
student = {
    "name": "Hafsa",
    "age": 25
}

keys = student.keys()

print(keys)
```

Output:

```
dict_keys(['name', 'age'])
```

You can convert it into a list:

```
key_list = list(student.keys())
```

Now:

```
print(key_list)
```

Output:

```
['name', 'age']
```

---

35. Dictionary Views Can Reflect Changes

Consider:

```
student = {
    "name": "Hafsa"
}

keys = student.keys()

student["age"] = 25

print(keys)
```

The view reflects the updated dictionary:

```
dict_keys(['name', 'age'])
```

This is one reason dictionary views are useful.

---

36. Converting Dictionary Views to Lists

Sometimes you need an actual list.

```
student = {
    "name": "Hafsa",
    "age": 25
}

keys = list(student.keys())
values = list(student.values())
items = list(student.items())
```

Now:

```
keys
→ ['name', 'age']

values
→ ['Hafsa', 25]

items
→ [('name', 'Hafsa'), ('age', 25)]
```

---

37. Counting With `get()`

A common dictionary pattern is counting occurrences.

```
letters = ["a", "b", "a", "c", "a", "b"]

counts = {}

for letter in letters:
    counts[letter] = counts.get(letter, 0) + 1

print(counts)
```

Output:

```
{
    "a": 3,
    "b": 2,
    "c": 1
}
```

Let's understand:

```
counts.get(letter, 0)
```

means:

```
If the key exists → get its current count.
If it doesn't exist → use 0.
```

Then:

```
+ 1
```

increments the count.

This is one of the most useful dictionary patterns in Python.

---

38. Finding Missing Data

Suppose an application receives incomplete user data:

```
user = {
    "name": "Hafsa",
    "country": "Pakistan"
}
```

You can safely retrieve optional information:

```
phone = user.get("phone", "Not provided")
```

Then:

```
print(phone)
```

Output:

```
Not provided
```

This prevents your program from crashing just because optional information is missing.

---

39. Filtering Dictionary Data

Suppose:

```
scores = {
    "Hafsa": 92,
    "Asim": 65,
    "Ali": 88,
    "Sara": 55
}
```

You can use `.items()` with a condition:

```
for name, score in scores.items():
    if score >= 80:
        print(name, score)
```

Output:

```
Hafsa 92
Ali 88
```

Here:

```
items()
+
loop
+
condition
```

becomes a simple way to process dictionary data.

---

40. Checking Whether a Value Exists

The `in` operator normally checks dictionary **keys**, not values.

```
student = {
    "name": "Hafsa",
    "age": 25
}

print("name" in student)
```

Output:

```
True
```

But:

```
print("Hafsa" in student)
```

returns:

```
False
```

because `"Hafsa"` is a value, not a key.

To check values:

```
print("Hafsa" in student.values())
```

Output:

```
True
```

---

41. Checking Key-Value Pairs

You can also check whether a complete pair exists:

```
student = {
    "name": "Hafsa",
    "age": 25
}

print(("name", "Hafsa") in student.items())
```

Output:

```
True
```

This can be useful when working with structured records.

---

42. Copying and Updating Safely

Suppose you have a default configuration:

```
default_settings = {
    "theme": "light",
    "notifications": True,
    "language": "English"
}
```

You want a user's own settings without changing the defaults.

Use:

```
user_settings = default_settings.copy()

user_settings.update({
    "theme": "dark"
})
```

Now:

```
default_settings
→ theme: light

user_settings
→ theme: dark
```

This is a common real-world pattern.

---

43. Real World Example: haas.dev Resource Metadata

Imagine a resource record:

```
resource = {
    "title": "Python Dictionary Methods",
    "category": "Python",
    "level": "Beginner",
    "free": True
}
```

Get the title:

```
title = resource.get("title")
```

Add a tag:

```
resource.update({
    "tag": "data structures"
})
```

Check the available fields:

```
print(resource.keys())
```

Loop through the complete resource:

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

This shows how dictionary methods can work together inside an application.

---

44. Real World Example: User Preferences

```
settings = {
    "theme": "dark",
    "notifications": True
}
```

Safely read a setting:

```
language = settings.get("language", "English")
```

Add language:

```
settings["language"] = language
```

Or:

```
settings.setdefault("language", "English")
```

The second version only adds the value if the key is missing.

---

45. Real World Example: Permissions

```
permissions = {
    "read": True,
    "write": True,
    "delete": False
}
```

Check permissions:

```
if permissions.get("delete", False):
    print("Delete allowed")
else:
    print("Delete not allowed")
```

The default `False` makes the program safer when `"delete"` is missing.

---

46. Choosing the Right Method

Ask what you want to accomplish.

Need to read one value?

```
dictionary.get("key")
```

Need all keys?

```
dictionary.keys()
```

Need all values?

```
dictionary.values()
```

Need keys and values?

```
dictionary.items()
```

Need to add or update several values?

```
dictionary.update(...)
```

Need to remove a specific key?

```
dictionary.pop("key")
```

Need to remove the last inserted pair?

```
dictionary.popitem()
```

Need to add a key only if missing?

```
dictionary.setdefault("key", value)
```

Need to remove everything?

```
dictionary.clear()
```

Need an independent top-level copy?

```
dictionary.copy()
```

Need to create a dictionary from keys?

```
dict.fromkeys(...)
```

---

47. Common Mistakes

Mistake 1: Expecting `get()` to create a key

```
student.get("age", 25)
```

This does not add `"age"`.

Use:

```
student.setdefault("age", 25)
```

if you want to add it only when missing.

Mistake 2: Using `values()` to access a key

Incorrect:

```
student.values("name")
```

`values()` doesn't accept a key.

Use:

```
student.get("name")
```

Mistake 3: Expecting `pop()` to return the dictionary

```
result = student.pop("age")
```

`result` contains the removed **value**, not the dictionary.

Mistake 4: Forgetting that `update()` modifies the dictionary

```
result = student.update({
    "age": 26
})
```

`result` is:

```
None
```

The dictionary itself has been changed.

Mistake 5: Confusing `copy()` with assignment

```
copy = original
```

does not create an independent dictionary.

Use:

```
copy = original.copy()
```

for a shallow copy.

---

48. Dictionary Methods vs Direct Syntax

Many dictionary operations can be performed in more than one way.

Task	Common approach
Read existing value	`data["key"]`
Safely read value	`data.get("key")`
Add/update one value	`data["key"] = value`
Add/update many values	`data.update(...)`
Delete specific key	`data.pop("key")`
Delete everything	`data.clear()`
Inspect keys	`data.keys()`
Inspect values	`data.values()`
Inspect pairs	`data.items()`

The important skill is not memorizing every method separately.

It is understanding **which operation solves which problem**.

---

49. Method Combination Example

Dictionary methods become more powerful when combined.

```
user = {
    "name": "Hafsa",
    "role": "developer"
}

user.setdefault("experience", 0)

user.update({
    "active": True
})

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

Possible output:

```
name: Hafsa
role: developer
experience: 0
active: True
```

Several methods worked together to build and process the record.

---

50. Mini Project: Developer Profile Manager

Create:

```
developer = {
    "name": "Hafsa",
    "role": "Software Engineer",
    "skills": ["Python", "JavaScript"]
}
```

Step 1: Safely get experience

```
experience = developer.get("experience", 0)
```

Step 2: Add experience

```
developer.setdefault("experience", 0)
```

Step 3: Update the role

```
developer.update({
    "role": "Full Stack Developer"
})
```

Step 4: Display all information

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

Step 5: Remove a field

```
developer.pop("experience")
```

This mini project practices:

```
get()
setdefault()
update()
items()
pop()
```

---

51. 5 Minute Challenge

Create this dictionary:

```
profile = {
    "name": "Hafsa",
    "role": "Developer",
    "country": "Pakistan"
}
```

Now:

1. Use `get()` to retrieve `"name"`.
2. Use `get()` to retrieve a missing `"email"` with `"Not provided"` as the default.
3. Use `keys()` to display all keys.
4. Use `values()` to display all values.
5. Use `items()` to display every key-value pair.

6. Use `update()` to add `"experience": 2`.
7. Use `setdefault()` to add `"language": "Python"`.
8. Use `pop()` to remove `"country"`.
9. Create a copy using `copy()`.
10. Clear the copied dictionary using `clear()`.

---

52. Practice Exercises

Exercise 1: Safe User Data

Create:

```
user = {
    "name": "Hafsa",
    "username": "hafsa_dev"
}
```

Use `get()` to retrieve:

```
email
phone
country
```

Give each one a useful default value.

Exercise 2: Dictionary Inspection

Create a product dictionary:

```
product = {
    "name": "Laptop",
    "price": 150000,
    "stock": 8
}
```

Use:

- `keys()`
- `values()`
- `items()`

to display its information.

Exercise 3: Update Product

Use `update()` to change:

```
price → 145000
stock → 10
```

and add:

```
category → Electronics
```

Exercise 4: Remove Data

Create:

```
account = {
    "username": "hafsa",
    "email": "user@example.com",
    "temporary_code": "12345"
}
```

Remove `"temporary_code"` using `pop()`.

Exercise 5: Default Values

Create:

```
settings = {
    "theme": "dark"
}
```

Use `setdefault()` to add:

```
notifications → True
language → English
```

Then try setting `"theme"` to `"light"` using `setdefault()`.

Observe what happens.

Exercise 6: Count Words

Given:

```
words = ["python", "javascript", "python", "react", "python", "react"]
```

Use a dictionary and `get()` to count how many times each word appears.

Expected:

```
{
    "python": 3,
    "javascript": 1,
    "react": 2
}
```

Exercise 7: Group Students

Given:

```
students = [
    ("Python", "Hafsa"),
    ("Python", "Asim"),
    ("JavaScript", "Ali"),
    ("JavaScript", "Sara")
]
```

Use `setdefault()` to create:

```
{
    "Python": ["Hafsa", "Asim"],
    "JavaScript": ["Ali", "Sara"]
}
```

---

53. Mini Quiz

Question 1

Which method safely retrieves a value when a key might not exist?

A. `items()`

B. `get()`

C. `popitem()`

D. `clear()`

Answer: B

Question 2

Which method returns key-value pairs?

A. `keys()`

B. `values()`

C. `items()`

D. `get()`

Answer: C

Question 3

Which method adds a value only if the key does not already exist?

A. `update()`

B. `setdefault()`

C. `copy()`

D. `pop()`

Answer: B

---

54. Interview Questions

Beginner

1. What are dictionary methods in Python?
2. What does `get()` do?
3. What is the difference between `get()` and `[]`?
4. What does `keys()` return?
5. What does `values()` return?
6. What does `items()` return?
7. What does `update()` do?
8. What does `pop()` do?
9. What does `popitem()` do?
10. What does `clear()` do?

Intermediate

11. What is the difference between `pop()` and `popitem()`?
12. How can you provide a default value with `get()`?
13. What is `setdefault()` used for?
14. What is the difference between `setdefault()` and direct assignment?
15. What does `copy()` do?
16. What is a shallow copy?
17. What does `dict.fromkeys()` do?
18. Why should you be careful with mutable default values in `fromkeys()`?
19. What do dictionary view objects represent?
20. How can `get()` be used to count values?

Problem Solving

21. How would you safely access optional user data?
22. How would you count the frequency of words using a dictionary?
23. How would you group multiple values under the same key?
24. How would you create a copy of a dictionary before modifying it?
25. Which dictionary method would you use to remove a specific key and retrieve its value?

---

55. Complete Dictionary Methods Cheat Sheet

```
# Safe access
student.get("name")
student.get("email", "Not provided")

# Keys
student.keys()

# Values
student.values()

# Key-value pairs
student.items()

# Add/update
student.update({
    "age": 25,
    "course": "Python"
})

# Remove specific key
student.pop("age")

# Remove last inserted item
student.popitem()

# Add only if missing
student.setdefault("country", "Pakistan")

# Remove everything
student.clear()
```

```
# Copy
student_copy = student.copy()

# Create dictionary from keys
data = dict.fromkeys(["a", "b", "c"], 0)
```

---

56. The Dictionary Methods Mental Model

Think of dictionary methods in five groups:

```
READ
│
├── get()
├── keys()
├── values()
└── items()

ADD / UPDATE
│
├── update()
└── setdefault()

REMOVE
│
├── pop()
├── popitem()
└── clear()

COPY / CREATE
│
├── copy()
└── fromkeys()
```

This makes the methods easier to remember.

Instead of memorizing isolated method names, remember what **job** each method performs.

---

57. Key Takeaways

- `get()` safely retrieves a value.
- `keys()` gives access to dictionary keys.
- `values()` gives access to dictionary values.

- `items()` gives key-value pairs.
- `update()` adds or changes multiple items.
- `pop()` removes a specific key and returns its value.
- `popitem()` removes the last inserted pair.
- `setdefault()` adds a key only when it is missing.
- `clear()` removes all items.
- `copy()` creates a shallow copy.
- `fromkeys()` creates a dictionary from a collection of keys.
- Dictionary methods are especially useful when processing real-world structured data.
- `get()` and `setdefault()` solve different problems: one safely **reads**, while the other can **create missing data**.
- `keys()`, `values()`, and `items()` are particularly important when looping through dictionaries.

The core idea:

```
Need to READ?
→ get()

Need KEYS?
→ keys()

Need VALUES?
→ values()

Need BOTH?
→ items()

Need to ADD/UPDATE?
→ update()

Need to ADD ONLY IF MISSING?
→ setdefault()

Need to REMOVE?
→ pop()

Need to REMOVE THE LAST ITEM?
→ popitem()
```

```
Need to EMPTY?
→ clear()

Need a COPY?
→ copy()
```

Visit haas.dev for more resources and guides.

https://dev-roast-app.vercel.app

https://dev-roast-app.vercel.app/resources