Code Translation Using Generative AI

Estimated time needed: 30 minutes

As a software developer, you may be very comfortable coding in a particular language. But often you have requests from client, to use a different language than what you are comfortable with. In the past, in such situations, you would either have had to learn the language or have to give up on the client and lose business. But today, generative AI can make it much simpler for developers.

Please remember that the prompts that you feed to generative AI are like a conversation with a subject matter expert and the consecutive prompts depend on the previous ones and the response received. Change the wording if required to get a specific desired result. The example showcases one possible chat conversation to attain the objective.

Learning Objectives

At the end of the lab, you will be able to:

1. Use generative AI to translate the code written in one language to another, retaining the logic and workflow as is.
2. Convert JavaScript CRUD code to Python CRUD code without any scripting in Python.

Prerequisites

Know at least one programming language. The lab was created with some sample code in JavaScript. It will be good if you know how to read the JavaScript code.

Set up chat title and prompt instruction

When the chat session opens, a new chat conversation begins. Give the chat conversation an appropriate title. This will help you revisit the chat conversation. It is a good practice to segregate the conversations topically as it will help in continuing the conversation at a later point.

Also, provide prompt instructions that are specific to the conversation in this particular lab. Let's get started with the task of translating JavaScript code to Python.

> Please note that generative AI is an evolving field. As you attempt the labs, your experience and output might be different than what is seen here.

Tool capability to translate

You are already comfortable coding in JavaScript. You have created the code in JavaScript. You will now use gen AI model to create the same in Python.

Firstly, you need to determine if the gen AI tool has the capability to do the required translation. In the prompt type:

```
I have a requirement to code in Python. If I provide the JavaScript code, can you help me create the Python equivalent of the code?
```

If the GenAI model has the ability to translate, you will get an affirmative response in line with what is shown below. Python is one of the most commonly used languages, and this task of translating from JS to Python will be doable. But if there are other newer or scarcely used languages, you will need to use your judgment to perceive confidence in the response and decide whether to continue to use the tool, either entirely or as a support tool. Iterative prompts can be used to ensure the correctness of the responses.

▼ Click here to view the sample response generated

Yes — I can help translate JavaScript code to Python. Please provide the JavaScript code you want converted, and I will produce an equivalent Python implementation.

Here’s how I typically approach the translation to ensure accuracy and readability:

- Map JavaScript syntax and idioms to Python equivalents while preserving behavior
- Add clear comments and optional type hints (PEP 484) for readability
- Use Pythonic constructs (e.g., list comprehensions, idiomatic error handling)
- Preserve function and variable naming where reasonable, or suggest Pythonic alternatives
- Replace browser/JS-specific APIs with appropriate Python libraries (e.g., fetch -> requests or aiohttp, setInterval -> asyncio sleep loop)
- Provide error handling and input validation as appropriate
- Optionally include a small usage example or tests if you want

What I need from you to get started:

- Paste the JavaScript code (or share a link to it)
- Target Python version (e.g., 3.9, 3.11) and any style preferences (PEP 8)
- Any libraries to replace or avoid (e.g., using requests instead of fetch, or using asyncio for asynchronous code)
- Whether to implement synchronously or with asyncio (if the JS code uses async patterns)
- Any constraints or expectations on performance, error handling, or test coverage

Once you provide the JS code, I will respond with the Python equivalent, including well-documented code, type hints if requested, and notes on any behavioral caveats.

What I will deliver

- A Python 3.x module or function(s) implementing the same logic as your JavaScript code
- Optional type hints and docstrings for clarity
- Clear, well-commented code explaining non-trivial logic
- Error handling and input validation aligned with Python practices
- A short usage example or tests if you request them
- Guidance on any library substitutions (e.g., requests instead of fetch, asyncio vs sync)

What I need from you

- The JavaScript code to translate (paste here or share a link)
- Target Python version (e.g., 3.9, 3.11)
- Any preferred libraries or constraints (e.g., avoid external deps, use stdlib)
- Whether to implement synchronously or using asyncio
- Any specific API surface you want to preserve (function names, signatures)

Next steps

1. You paste the JS code
2. I provide the Python translation along with notes on any behavioral nuances and edge cases
3. If needed, we iterate to adjust API surface, performance tweaks, or style preferences

Provide the code to Gen AI

Paste the following code into the prompt and send it. This process will take a few seconds as it needs to process the code given and then give the equivalent.

```
// Employee class
class Employee {
  constructor(id, name, age, position) {
    this.id = id;
    this.name = name;
    this.age = age;
    this.position = position;
  }
}
// Employee data storage
let employees = [];
// Create employee
```

```
function createEmployee(id, name, age, position) {
  const employee = new Employee(id, name, age, position);
  employees.push(employee);
}
// Read employee by ID
function readEmployee(id) {
  return employees.find((employee) => employee.id === id);
}
// Update employee by ID
function updateEmployee(id, updatedData) {
  const employee = employees.find((employee) => employee.id === id);
  if (employee) {
    Object.assign(employee, updatedData);
  }
}

// Delete employee by ID
function deleteEmployee(id) {
  employees = employees.filter((employee) => employee.id !== id);
}
```

It will produce an output as Python code similar to what is shown below.

▼ Click here to view the sample response generated

This Python module provides a direct Python translation of the provided JavaScript Employee CRUD example. It defines an Employee dataclass, an in-memory store, and functions to create, read, update, and delete employees by ID. Includes input validation and basic error handling.

```
from dataclasses import dataclass
from typing import List, Optional, Union, Dict, Any
# In-memory storage for Employee records
EmployeeID = Union[int, str]
@dataclass
class Employee:
    id: EmployeeID
    name: str
    age: int
    position: str
# Global in-memory list to store employees
_employees: List[Employee] = []
def _find_employee_by_id(emp_id: EmployeeID) -> Optional[Employee]:
    """Return the first employee with matching id, or None if not found."""
    for emp in _employees:
        if emp.id == emp_id:
            return emp
    return None
def create_employee(emp_id: EmployeeID, name: str, age: int, position: str) -> Employee:
    """Create a new Employee and add it to the in-memory store.
    Args:
        emp_id: Unique identifier for the employee (int or str).
        name: Employee's name.
        age: Employee's age (non-negative integer).
        position: Job title/position.
    Returns:
        The created Employee instance.
    Raises:
        ValueError: If required fields are invalid or an employee with the same id exists.
    """
    if emp_id is None:
        raise ValueError("Employee id must be provided.")
    if not isinstance(name, str) or not name:
        raise ValueError("Name must be a non-empty string.")
    if not isinstance(age, int) or age < 0:
        raise ValueError("Age must be a non-negative integer.")
    if not isinstance(position, str) or not position:
        raise ValueError("Position must be a non-empty string.")
    if _find_employee_by_id(emp_id) is not None:
        raise ValueError(f"An employee with id {emp_id} already exists.")
    employee = Employee(id=emp_id, name=name, age=age, position=position)
    _employees.append(employee)
    return employee
def read_employee(emp_id: EmployeeID) -> Optional[Employee]:
    """Retrieve an employee by id. Returns None if not found."""
    return _find_employee_by_id(emp_id)
def update_employee(emp_id: EmployeeID, updated_data: Dict[str, Any]) -> Optional[Employee]:
    """Update attributes of an existing employee.
    updated_data keys should correspond to Employee attributes (e.g., 'name', 'age', 'position', 'id').
    Unsupported keys are ignored.
    Returns:
        The updated Employee instance, or None if the employee does not exist.
    """
    if updated_data is None or not isinstance(updated_data, dict):
        raise ValueError("updated_data must be a dictionary of attributes to update.")
    employee = _find_employee_by_id(emp_id)
    if employee is None:
        return None
    for key, value in updated_data.items():
        if hasattr(employee, key):
            setattr(employee, key, value)
        else:
            # Ignore unknown attributes to preserve API compatibility
            pass
    return employee
def delete_employee(emp_id: EmployeeID) -> bool:
    """Delete an employee by id. Returns True if an employee was removed."""
    global _employees
    initial_len = len(_employees)
    _employees = [e for e in _employees if e.id != emp_id]
    return len(_employees) < initial_len
# Optional: simple demo / tests when running this module directly
if __name__ == "__main__":
    try:
        e1 = create_employee(1, "Alice", 30, "Engineer")
        e2 = create_employee(2, "Bob", 28, "Designer")
        print(read_employee(1))
        update_employee(1, {"age": 31, "position": "Senior Engineer"})
        print(read_employee(1))
        delete_success = delete_employee(2)
        print(f"Deleted 2: {delete_success}")
        print(read_employee(2))
    except Exception as ex:
        print(f"Error: {ex}")
```

This is with the assumption that the JavaScript code has been typed properly. If there are minor coding mistakes, gen AI has the ability to correct itself and give the right output.

There are often a few lines of explanation given at the end of the response. This will not be the same every time, as the responses are AI-generated and not programmed. The essence of the message is, however, consistent. If there is no explanation, you can prompt for an explanation of the code rendered.

Explain the python code

You can skip this step if the explanation offered at the end of the previous step suffices. If required, prompt for an explanation of the code.

```
Explain this Python code step by step. For each section (class, data storage, CRUD functions, example usage), show the code block and then briefly explain what it does and why, in bullet points. End
```

The output would be similar to what is shown below. You can iteratively prompt for more specific questions you may have.

▼ Click here to view the sample response generated

# Step-by-step explanation of the Python code

This answer walks through the translated Python implementation section by section. Each section includes the relevant code block followed by concise bullet-point explanations describing what it does and why it exists.

---

1) Class: Employee (dataclass)

```
from dataclasses import dataclass
from typing import Optional, List, Dict, Any
@dataclass
class Employee:
    """Data model representing an employee."""
    id: int
    name: str
    age: int
    position: str
```

- Defines a lightweight data container for employee data.
- Uses `@dataclass` to auto-generate common boilerplate (constructor, repr, etc.).
- Type hints (`int`/`str`) communicate expected data types and help with static analysis.
- This class serves as the core data model for the in-memory store.

---

2) Data storage: In-memory list of employees

```
# In-memory storage for employees
employees: List[Employee] = []
```

- Stores Employee instances in a simple Python list, mirroring the original JavaScript array.
- Suitable for small examples and tests; for real apps, replace with a database or persistent store.
- All CRUD operations operate on this shared in-memory state.

---

3) Internal helper: _get_employee_by_id

```
def _get_employee_by_id(emp_id: int) -> Optional[Employee]:
    """Internal helper to find an employee by ID.
    Args:
        emp_id: The unique identifier of the employee.
    Returns:
        The Employee instance if found, otherwise None.
    """
    for emp in employees:
        if emp.id == emp_id:
            return emp
    return None
```

- Encapsulates the lookup logic so create/read/update/delete can reuse it.
- Returns `None` if not found, a natural Python pattern for “not found”.
- Leading underscore signals this is an internal helper, not part of the public API.

---

4) Create operation: create_employee

```
def create_employee(id: int, name: str, age: int, position: str) -> Employee:
    """Create and store a new employee.
    Raises:
        ValueError: If an employee with the given ID already exists.
    """
    if _get_employee_by_id(id) is not None:
        raise ValueError(f"Employee with id {id} already exists.")
    emp = Employee(id=id, name=name, age=age, position=position)
    employees.append(emp)
    return emp
```

- Ensures IDs are unique to avoid inconsistent state.
- Appends the new Employee to the in-memory storage and returns it.
- Uses Python-style exceptions to signal invalid operations (duplicate ID).

---

5) Read operation: read_employee

```
def read_employee(id: int) -> Optional[Employee]:
    """Read (retrieve) an employee by ID.
    Returns:
        The Employee instance if found, otherwise None.
    """
    return _get_employee_by_id(id)
```

- Retrieves an employee by ID using the helper function.
- Returns the Employee or `None` if not found (Pythonic not-found pattern).
- Simple and predictable behavior for lookups.

---

6) Update operation: update_employee

```
def update_employee(id: int, updated_data: Dict[str, Any]) -> Optional[Employee]:
    """Update fields of an existing employee using a dict of updates.
    Only known fields are updated. The function will raise if an attempt is
    made to change the immutable 'id' field.
    Args:
        id: The ID of the employee to update.
        updated_data: A dictionary of fields to update (e.g., {"name": "New Name"}).
    Returns:
        The updated Employee instance, or None if the employee does not exist.
    """
    emp = _get_employee_by_id(id)
    if emp is None:
        return None
    allowed = {"name", "age", "position"}
    for key, value in updated_data.items():
        if key in allowed:
            setattr(emp, key, value)
        elif key == "id":
            # Do not allow updating the primary key
            raise ValueError("Cannot update the 'id' of an existing employee.")
        else:
            # Ignore unknown keys (could also log a warning)
            pass
    return emp
```

- Looks up the target employee, then applies updates from a dictionary.
- Uses an allowlist to prevent unintended fields from being modified.
- Explicitly prevents changing the ‘id’ to preserve primary key integrity.
- Unknown keys are ignored by default.

---

7) Delete operation: delete_employee

```
def delete_employee(id: int) -> bool:
        """Delete the employee with the given ID.
        Returns:
            True if an employee was deleted, False if not found.
        """
        global employees
        before = len(employees)
        employees = [emp for emp in employees if emp.id != id]
        return len(employees) < before
```

- Removes the employee by filtering the storage list and reassigning it.
- Returns `True` if a deletion actually occurred, otherwise `False`.
- Simple in-memory deletion; thread-safety considerations apply in concurrent environments.

---

8) Script usage: main example

```
    if __name__ == "__main__":
        # Simple usage example
        try:
            e1 = create_employee(1, "Alice", 30, "Engineer")
            e2 = create_employee(2, "Bob", 28, "Designer")
            print("Created:", e1, e2)
            print("Read 1:", read_employee(1))
            update_employee(1, {"age": 31, "position": "Senior Engineer"})
            print("After update 1:", read_employee(1))
            delete_employee(2)
            print("All employees:", employees)
        except Exception as ex:
            print("Error:", ex)
```

- Demonstrates a full CRUD flow: create two employees, read one, update it, then delete another.
- Prints current state to quickly validate behavior.
- Wrapped in a try/except to surface errors cleanly during demonstration.

---

9) Quick overall summary

- Defines an Employee dataclass as the data model.
- Uses an in-memory list as a simple data store for demonstration and testing.
- CRUD operations have clear, well-typed signatures and sensible error handling:
    - `create_employee` enforces unique IDs and returns the created instance.
    - `read_employee` returns the matching Employee or `None`.
    - `update_employee` applies allowed field updates and prevents changing the ID.

  - `delete_employee` removes the matching Employee and reports success.
- A small main demo shows end-to-end usage.
- For production, replace the in-memory store with a persistent datastore, add validation and tests, and consider concurrency controls.

Versions of Python

It is important to know which version of Python the code will work with. There might be requirements to provide older versions for backward compatibility, especially when you are going to create software that fits into other existing software. Type the next prompt as:

```
Explain which Python versions this code runs on, noting any features and their minimum version. Include recommended versions and a short conclusion, formatted in clear sections with bullet points.
```

The output would be similar to what is given below.

▼ Click here to view the sample response generated

Python Version Compatibility for the In-Memory Employee CRUD Module

This explanation covers the Python version compatibility for the in-memory Employee CRUD module that mirrors the provided JavaScript example. It highlights the minimum Python version required by the language features used, recommended versions for long-term maintenance, and practical notes when targeting older runtimes.

Minimum Python version to run this code

- The most impactful feature is the `dataclass` decorator, which requires **Python 3.7** or newer. For Python 3.6, you can use the `dataclasses` backport package, though it may not be feature-complete or future-proof.
- Other language features (type hints via `typing`, `Optional`, `List`, `Dict`, `Any`; f-strings; variable annotations) are available from Python 3.5+ (`typing`), 3.6+ (variable annotations and f-strings). The code uses these without additional dependencies.
- The module relies only on the **standard library**, so there are no external package requirements beyond the Python version itself.

Key features used and their version requirements

- **Dataclasses (`@dataclass`)**: Lightweight data model; minimum Python 3.7.
- **Type hints** (`typing.Optional`, `List`, `Dict`, `Any`): For readability and tooling; available since Python 3.5+.
- **Variable annotations** (e.g., `employees: List[Employee] = []`): Supported since Python 3.6.
- **f-strings**: Readable string formatting; available since Python 3.6.
- **Basic collection operations**: Lists, list comprehensions, dictionary-based updates; standard and broadly compatible.
- No reliance on deprecated features or third-party libraries; compatibility is mainly tied to Python version rather than runtime environment.

Recommended Python versions and rationale

- **Minimum practical target**: Python 3.7+ to use dataclasses without backport concerns.
- **Better typing, performance, and newer language features**: Python 3.9–3.11+. These versions provide broader standard library improvements and ongoing security updates.

- **Starting a new project**: Use the latest stable release you can reasonably support (e.g., Python 3.11 or newer at setup).
- **Enterprise/long-term stability**: Adopt a version with extended security updates (ESU) and align with your CI/CD pipelines.

Backward compatibility and migration notes

- Supporting **Python 3.6**: Install the `dataclasses` backport and test thoroughly.
- Supporting **pre-3.6 runtimes**: Rewrite the dataclass into a regular class and manually implement `__init__`, `__repr__`, and other boilerplate; this reduces readability and increases maintenance burden.
- **Upgrading Python versions**: Run your test suite to catch changes in typing behavior or runtime semantics, although this code uses standard features with broad compatibility.

Conclusion

- The Python module is **compatible with Python 3.7+** out of the box due to dataclasses.
- Using Python **3.9–3.11+** provides better tooling, performance, and typing ergonomics, making it the recommended range for new projects.
- Supporting older versions may require minor adaptations (dataclasses backport, manual class definitions) and extra testing.
- Overall, target a **recent Python version (3.9+ or 3.11+)** for new development, with **3.7+** as the baseline for older environments.

Include version information in code

It is easier to include the Python version information in the file. Type the next prompt as:

```
Update the Python code to include comments specifying which Python versions support it. Keep the code unchanged except for adding a comment section listing minimum, recommended, and newer versions. I
```

It will reproduce the same code as before, with the comment statement about the version included.

▼ Click here to view the sample response generated

Added a module-level compatibility note as a docstring at the top of the Python module to specify minimum, recommended, and newer Python versions. The rest of the code is unchanged and continues to implement the in-memory Employee CRUD API. The docstring provides guidance for environment choices without altering runtime behavior.

```
"""
Version compatibility notes
Minimum supported Python version: 3.7 (dataclasses)
Recommended for new projects: Python 3.9–3.11
Newer versions: Python 3.12+ (when available)
Notes:
- The code uses dataclasses (Python 3.7+), typing (3.5+), f-strings (3.6+), and standard library features.
- If you must run on Python 3.6, dataclasses backport is needed and some typing features might be slightly different.
Summary:
This section documents the minimum, recommended, and newer Python versions to help you choose an environment. The code itself remains unchanged.
"""
from dataclasses import dataclass
from typing import Optional, List, Dict, Any
@dataclass
class Employee:
    """Data model representing an employee."""
    id: int
    name: str
    age: int
    position: str
# In-memory storage for employees
employees: List[Employee] = []
def _get_employee_by_id(emp_id: int) -> Optional[Employee]:
    """Internal helper to find an employee by ID.
    Args:
        emp_id: The unique identifier of the employee.
    Returns:
        The Employee instance if found, otherwise None.
    """
    for emp in employees:
        if emp.id == emp_id:
            return emp
    return None
def create_employee(id: int, name: str, age: int, position: str) -> Employee:
    """Create and store a new employee.
    Raises:
        ValueError: If an employee with the given ID already exists.
    """
    if _get_employee_by_id(id) is not None:
        raise ValueError(f"Employee with id {id} already exists.")
    emp = Employee(id=id, name=name, age=age, position=position)
    employees.append(emp)
    return emp
def read_employee(id: int) -> Optional[Employee]:
    """Read (retrieve) an employee by ID.
    Returns:
        The Employee instance if found, otherwise None.
    """
    return _get_employee_by_id(id)
def update_employee(id: int, updated_data: Dict[str, Any]) -> Optional[Employee]:
    """Update fields of an existing employee using a dict of updates.
    Only known fields are updated. The function will raise if an attempt is
    made to change the immutable 'id' field.
    Args:
        id: The ID of the employee to update.
        updated_data: A dictionary of fields to update (e.g., {"name": "New Name"}).
    Returns:
        The updated Employee instance, or None if the employee does not exist.
    """
    emp = _get_employee_by_id(id)
    if emp is None:
        return None
    allowed = {"name", "age", "position"}
    for key, value in updated_data.items():
        if key in allowed:
            setattr(emp, key, value)
        elif key == "id":
            # Do not allow updating the primary key
            raise ValueError("Cannot update the 'id' of an existing employee.")
        else:
            # Ignore unknown keys (could also log a warning)
            pass
    return emp
def delete_employee(id: int) -> bool:
    """Delete the employee with the given ID.
    Returns:
        True if an employee was deleted, False if not found.
    """
    global employees
    before = len(employees)
    employees = [emp for emp in employees if emp.id != id]
    return len(employees) < before
if __name__ == "__main__":
    # Simple usage example
    try:
        e1 = create_employee(1, "Alice", 30, "Engineer")
        e2 = create_employee(2, "Bob", 28, "Designer")
        print("Created:", e1, e2)
        print("Read 1:", read_employee(1))
        update_employee(1, {"age": 31, "position": "Senior Engineer"})
        print("After update 1:", read_employee(1))
        delete_employee(2)
        print("All employees:", employees)
    except Exception as ex:
        print("Error:", ex)
```

Including comments

Comments are extremely important for code readability. It is always good coding practice to include comments explaining what the code does. Type the next prompt as:

```
Add detailed comments to the existing Python code to explain:
The purpose of each class and function
Each attribute in the Employee class
Key steps in the CRUD operations
Example usage
Also, provide a short summary explaining how the comments improve readability.
```

▼ Click here to view the sample response generated

The updated Python code adds extensive inline comments to improve readability, without changing runtime behavior. Key additions include: (1) clear descriptions for the Employee data model and each attribute; (2) section headers that explain storage and CRUD operations; (3) detailed, actionable comments around validation rules (e.g., ensuring unique IDs, preventing ID changes); (4) annotated example usage to illustrate end-to-end CRUD flow; and (5) a final readability summary explaining how the comments aid understanding and maintenance.

```
# Version compatibility notes
# Minimum supported Python version: 3.7 (dataclasses)
# Recommended for new projects: Python 3.9–3.11
# Newer versions: Python 3.12+ (when available)
#
# Notes:
# - The code uses dataclasses (Python 3.7+), typing (3.5+), f-strings (3.6+),
#   and standard library features. If you must run on Python 3.6, consider
#   the dataclasses backport and be aware some typing features might differ.
#
# Summary:
# - This compatibility section documents the minimum, recommended, and newer
#   Python versions to help you choose an environment. The code itself remains
#   unchanged aside from these version notes.
from dataclasses import dataclass
from typing import Optional, List, Dict, Any
# --- 1) Data model: Employee ---
# The Employee class represents a single employee record in memory.
# Each attribute is documented to explain its purpose and expected type.
@dataclass
class Employee:
    # Unique identifier for the employee
    id: int
    # Employee full name
    name: str
    # Age of the employee in years
    age: int
    # Job title or position within the organization
    position: str
# --- 2) Data storage: In-memory list of employees ---
# A simple in-memory storage (a Python list) for demonstration and testing.
# The list holds Employee instances and is the basis for CRUD operations.
employees: List[Employee] = []
# --- 3) Internal helper: _get_employee_by_id ---
def _get_employee_by_id(emp_id: int) -> Optional[Employee]:
    # Internal helper: find and return the Employee with the given ID, or None if not found
    # This centralizes the lookup logic so all CRUD operations can reuse it
    for emp in employees:
        if emp.id == emp_id:
            return emp
    return None
# --- 4) Create operation: create_employee ---
def create_employee(id: int, name: str, age: int, position: str) -> Employee:
    # Create a new Employee and add to in-memory storage
    # First, ensure no existing employee has the same ID to preserve uniqueness
    if _get_employee_by_id(id) is not None:
        raise ValueError(f"Employee with id {id} already exists.")
    # Build the Employee object and append to storage
    emp = Employee(id=id, name=name, age=age, position=position)
    employees.append(emp)
    return emp
# --- 5) Read operation: read_employee ---
def read_employee(id: int) -> Optional[Employee]:
    # Retrieve an employee by ID; returns None if not found
    # Delegates to the shared helper for consistency
    return _get_employee_by_id(id)
# --- 6) Update operation: update_employee ---
def update_employee(id: int, updated_data: Dict[str, Any]) -> Optional[Employee]:
    # Update allowed fields of an existing employee using a dictionary
    # Look up the employee; if not found, return None
    emp = _get_employee_by_id(id)
    if emp is None:
        return None
    # Only these fields may be updated
    allowed = {"name", "age", "position"}
    for key, value in updated_data.items():
        if key in allowed:
            # Apply allowed field updates
            setattr(emp, key, value)
        elif key == "id":
            # Prohibit changing the immutable primary key
            raise ValueError("Cannot update the 'id' of an existing employee.")
        else:
            # Ignore unknown keys to avoid silent failures
            pass
    return emp
# --- 7) Delete operation: delete_employee ---
def delete_employee(id: int) -> bool:
    # Delete the employee with the given ID; return True if deleted
    global employees
    before = len(employees)
    # Create a new list excluding the employee with the given ID
    employees = [emp for emp in employees if emp.id != id]
    return len(employees) < before
# --- 8) Example usage: demonstrate a simple CRUD flow ---
if __name__ == "__main__":
    # Demonstration of create, read, update, and delete operations
    try:
        e1 = create_employee(1, "Alice", 30, "Engineer")
        e2 = create_employee(2, "Bob", 28, "Designer")
        print("Created:", e1, e2)
        print("Read 1:", read_employee(1))
        update_employee(1, {"age": 31, "position": "Senior Engineer"})
        print("After update 1:", read_employee(1))
        delete_employee(2)
        print("All employees:", employees)
    except Exception as ex:
        print("Error:", ex)
# --- 9) Readability enhancement: why these comments help ---
# - The purpose of each section (class, storage, CRUD, usage) is stated up front.
# - Each Employee attribute has a clear description of its role and type.
# - CRUD steps include validation details (e.g., unique IDs, immutability of 'id').
# - The example usage is annotated to show the flow and expected outputs.
# Summary:
# The added comments explain the intent, data model, and CRUD logic in plain language,
# making the code approachable for new readers and easier to maintain or extend.
```

Conclusion

Congratulations! You have created Python code without knowing or knowing little about the language. If you are happy with the output produced, you may copy the code, use it, and engage generative AI to enhance the code with other functionalities. If you want to improvise, you may ask additional relevant questions. Generative AI depends on external sources to supplement the responses with more facts and realism. However, it is your prerogative as a software developer to run the software and see if it is created as per the requirements.

You can always visit the conversation during the live session using the history option by clicking on the burger menu on the top-left.

Then choose the chat by title among all the labs listed out.

Author(s)

Ramanujam
Lavanya TS

© IBM Corporation. All rights reserved.