```
# 🔢 Python `enumerate()` – The Complete Guide
```

📌 What is `enumerate()`?

`enumerate()` is a **built-in Python function** that lets you **loop through an iterable (like a list or tuple)** and **track the index** of each item **at the same time** — all without writing extra code. 🧙

🧠 Why Use `enumerate()`?

Without `enumerate()`:

```python
fruits = ["apple", "banana", "mango"]
for i in range(len(fruits)):
    print(i, fruits[i])
```

With `enumerate()` (cleaner!):

```python
for i, fruit in enumerate(fruits):
    print(i, fruit)
```

✅ **Cleaner** ✅ **More Pythonic** ✅ **Less error-prone**

🧪 Syntax

```python
enumerate(iterable, start=0)
```

- `iterable` – any iterable object (list, tuple, string, etc.)
- `start` – index to start counting from (default is `0`)

🔍 Basic Example

```python
names = ["Alice", "Bob", "Charlie"]

for index, name in enumerate(names):
    print(f"{index}: {name}")
```

📤 **Output**:

```
0: Alice
1: Bob
2: Charlie
```

🎯 Custom Start Index

```python
for idx, name in enumerate(names, start=1):
    print(f"{idx}: {name}")
```

📤 **Output**:

```
1: Alice
2: Bob
3: Charlie
```

🧵 Use Case Examples

✅ 1. Index-based Updates

```python
nums = [10, 20, 30]
for i, val in enumerate(nums):
    nums[i] = val + 5
print(nums)
```

📤 Output: `[15, 25, 35]`

✅ 2. Parallel Iteration with Index

```python
questions = ["Your name?", "Your age?"]
answers = ["Darshan", "21"]

for i, (q, a) in enumerate(zip(questions, answers), start=1):
    print(f"Q{i}: {q} → A: {a}")
```

✅ 3. Finding Positions

```
text = "hello world"
for index, char in enumerate(text):
    if char == "o":
        print(f"Found 'o' at index {index}")
```

☑ 4. With Tuples & Lists

```
data = [("Darshan", 20), ("Vasani", 21)]
for i, (name, age) in enumerate(data):
    print(f"{i}: {name} is {age} years old")
```

☑ 5. Skipping Items Conditionally

```
words = ["skip", "take", "ignore", "process"]
for idx, word in enumerate(words):
    if idx % 2 == 0:
        continue
    print(f"{idx}: {word}")
```

🚫 Common Mistake

```
for fruit in enumerate(["apple", "banana"]):
    print(fruit)
```

📤 Output:

```
(0, 'apple')
(1, 'banana')
```

🔎 Explanation:

- It works — but returns a **tuple**.
- You must unpack it like this:

```
for i, fruit in enumerate(["apple", "banana"]):
    print(i, fruit)
```

💡 Behind the Scenes

```
list(enumerate(["x", "y", "z"]))  # [(0, 'x'), (1, 'y'), (2, 'z')]
```

`enumerate()` returns an **enumerate object**, which is an iterator that yields `(index, value)` pairs.

🧾 Summary Table

Feature	Description
Cleaner loops	Replaces `range(len(...))`
Custom index	Use `start=1` or any number
Works on any iterable	Strings, lists, tuples, generators
Tuple output	Returns `(index, value)`
Memory-efficient	Uses iterators internally (like `zip`)

✅ Best Practices

Tip	Description
Use `enumerate()` over `range(len(...))`	More readable
Always unpack `index, value`	Avoid confusion
Combine with `zip()` for dual iteration	Helpful in interviews

🧠 Real Life Analogy

> You're a teacher calling attendance. Instead of saying "next," you say: `1. Alice`, `2. Bob`, `3. Charlie`
> That's exactly what `enumerate()` does! 🎓

🧙 Bonus Challenge

🔧 **Problem:** Loop through a list and print only items at even indexes.

```
items = ["pen", "book", "pencil", "eraser"]

for idx, item in enumerate(items):
    if idx % 2 == 0:
        print(f"Even index {idx}: {item}")
```

🐍 Part 1: Error Handling in Pure Python

☑️ Why Error Handling?

To catch and handle runtime errors gracefully without crashing the program.

🎯 Basic Syntax

```
try:
    # risky code
except SomeException as e:
    # handle error
else:
    # if no exception occurs
finally:
    # always runs
```

💥 Common Exceptions

Exception	Description
`ZeroDivisionError`	Division by zero attempted
`TypeError`	Wrong data type used
`ValueError`	Invalid value
`KeyError`	Missing dictionary key
`FileNotFoundError`	File path doesn't exist

🧪 Example

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except ZeroDivisionError:
    print("Can't divide by zero!")
except ValueError:
    print("Invalid number entered.")
else:
    print("Result:", result)
finally:
    print("Execution complete.")
```

⚡ Part 2: Error Handling in FastAPI

FastAPI uses `HTTPException` and **custom exception handlers** for robust APIs.

🔧 Using HTTPException

```python
from fastapi import FastAPI, HTTPException

app = FastAPI()

@app.get("/items/{item_id}")
def read_item(item_id: int):
    if item_id == 0:
        raise HTTPException(status_code=404, detail="Item not found")
    return {"item_id": item_id}
```

🛠 Custom Exception & Handler

```python
from fastapi import Request
from fastapi.responses import JSONResponse

class UnicornException(Exception):
    def __init__(self, name: str):
        self.name = name

@app.exception_handler(UnicornException)
async def unicorn_exception_handler(request: Request, exc: UnicornException):
    return JSONResponse(
        status_code=418,
        content={"message": f"Oops! {exc.name} did something wrong!"},
    )
```

✅ This allows better control over your API behavior.

🧱 Part 3: Error Handling in Django

✅ Built-in Views for 404/500

In `settings.py`, Django automatically handles errors via these views:

- `handler404`
- `handler500`
- `handler403`
- `handler400`

You can override them in `urls.py`:

```
handler404 = 'myapp.views.custom_404'
```

```
def custom_404(request, exception):
    return render(request, "404.html", status=404)
```

🔐 Using `try-except` in Views

```
def get_product(request, id):
    try:
        product = Product.objects.get(pk=id)
    except Product.DoesNotExist:
        return HttpResponse("Product not found", status=404)
    return render(request, "product.html", {"product": product})
```

☑ Prefer using `get_object_or_404()` for conciseness.

💬 Validation Errors (Forms)

```
from django import forms

class ProductForm(forms.Form):
    name = forms.CharField(max_length=100)

    def clean_name(self):
        data = self.cleaned_data['name']
        if "@" in data:
            raise forms.ValidationError("Invalid character '@'")
        return data
```

💧 Part 4: Error Handling in Flask

Flask handles errors using decorators or global handlers.

🚨 Built-in Error Handler

```
from flask import Flask, jsonify

app = Flask(__name__)

@app.errorhandler(404)
```

```
def not_found(e):
    return jsonify(error="Page not found"), 404
```

🎯 Raise Custom Error

```
from flask import abort

@app.route("/divide/<int:num>")
def divide(num):
    if num == 0:
        abort(400, "Division by zero not allowed")
    return str(10 / num)
```

🧠 Custom Exception Class

```
class CustomError(Exception):
    pass

@app.errorhandler(CustomError)
def handle_custom_error(e):
    return jsonify(error=str(e)), 500
```

📦 Best Practices Across All Frameworks

✅ Use specific exceptions (avoid `except:`).

✅ Log exceptions (`logging` module or service like Sentry).

✅ Don't expose raw errors to users.

✅ Keep response consistent in APIs: `status`, `message`, and `errorCode`.

✅ Use middleware for global exception handling (especially in FastAPI/Flask).

🧪 Testing Tip

```
curl -X GET http://127.0.0.1:8000/items/0
```

🧾 Summary

Feature	Python	FastAPI	Django	Flask
Basic try/except	✅	✅	✅	✅
HTTPException	❌	✅	🚫 (custom views)	✅ (`abort`)
Custom Exception	✅	✅ `@exception_handler`	✅ (view logic)	✅ `@errorhandler`
Response Shaping	❌	✅	✅ (HttpResponse)	✅ (JSONResponse)

🧩 Python JSON Essentials: `json.load()` vs `json.dump()` 🎯

Python's `json` module is your go-to for **working with JSON files** — whether you're reading from or writing to them.

📦 First, Import the Module

```python
import json
```

🔍 `json.load()`: Read JSON from a File

> 📖 Think of `json.load()` as **loading structured JSON data from a `.json` file into a Python object** (like dict or list).

🧠 Syntax:

```python
data = json.load(file_object)
```

✅ Example:

Suppose you have a file `data.json`:

```json
{
  "name": "Darshan",
  "age": 23,
  "skills": ["Python", "React", "Node"]
}
```

🧪 Python Code to Read:

```
```

```python
import json

with open('data.json', 'r') as file:
    data = json.load(file)

print(data['name'])  # Output: Darshan
print(data['skills'])  # Output: ['Python', 'React', 'Node']
```

🔁 What You Get:

A Python dictionary!

📝 `json.dump()`: Write Python Data to a JSON File

> 💾 Use `json.dump()` to **serialize** Python objects (like dict, list) into a file as JSON format.

🧠 Syntax:

```python
json.dump(data, file_object, indent=4)
```

✅ Example:

```python
import json

data = {
    "name": "Darshan",
    "city": "Ahmedabad",
    "languages": ["Python", "JavaScript"]
}

with open('output.json', 'w') as file:
    json.dump(data, file, indent=4)
```

📂 What you get in `output.json`:

```json
{
    "name": "Darshan",
    "city": "Ahmedabad",
    "languages": [
        "Python",
        "JavaScript"
    ]
}
```

🎨 `indent=4` makes it **pretty printed**!

🤔 When to Use What?

Task	Use
☑ Read from `.json` file	`json.load(file)`
☑ Write to `.json` file	`json.dump(data, file)`
☑ Convert JSON string to Python	`json.loads(string)`
☑ Convert Python to JSON string	`json.dumps(data)`

⚠ Common Mistakes

✖ **Using `json.load()` on a JSON string** ☑ Use `json.loads()` instead.

✖ **Writing JSON to file using `dumps()`** ☑ Use `json.dump()` for writing directly to a file.

🎯 Real-Life Use Case

```
# Saving user preferences
preferences = {
    "theme": "dark",
    "notifications": True,
    "volume": 70
}

# Save to file
with open('prefs.json', 'w') as f:
    json.dump(preferences, f, indent=2)

# Later... load from file
with open('prefs.json', 'r') as f:
    prefs = json.load(f)

print(prefs["theme"])  # Output: dark
```

🧙 Summary Spellbook

Function	Purpose	Works With
`json.load()`	Read JSON from file → Python object	File
`json.dump()`	Write Python object → JSON file	File
`json.loads()`	Read JSON from string → Python	String

Function	Purpose	Works With
`json.dumps()`	Convert Python object → JSON string	String