1. Key terms

- An **iterable** is an object you can loop over (e.g., a list, tuple, string, or a custom object that implements the protocol). Medium+2Real Python+2
- An **iterator** is an object that actually does the iteration: it knows how to fetch the "next" item, and when the items are exhausted, it signals that with a `StopIteration` exception. Real Python+1
- The methods involved:
    - `__iter__()` — called on the iterable to get the iterator.
    - `__next__()` — called on the iterator to get each next item; once there is no next item, it must raise `StopIteration`. GeeksforGeeks+1

---

2. What the `for` loop *really* does

When you write something like:

```
for x in some_iterable:
    # body
```

Under the hood Python more or less does this:

```
# roughly equivalent:
iterator = iter(some_iterable)          # calls
some_iterable.__iter__()
while True:
    try:
        x = next(iterator)               # calls iterator.__next__()
    except StopIteration:
        break
    # body of the for loop using x
```

This means:

- `iter(some_iterable)` tries to get an iterator out of the iterable (via `__iter__()`, or fallback to older protocols). Real Python+1
- Then repeatedly call `next(iterator)`, which calls `iterator.__next__()`.
- When `__next__()` raises `StopIteration`, the loop ends (no error to user, just end of loop).

---

3. Why these methods matter & how custom objects use them

- If you define a custom class and you want it to be usable in a `for` loop, you typically implement `__iter__()` and `__next__()`. For example:

```
class MyCounter:
    def __init__(self, low, high):
        self.current = low
        self.high = high

    def __iter__(self):
        return self

    def __next__(self):
        if self.current > self.high:
            raise StopIteration
        result = self.current
        self.current += 1
        return result
```

Now `for n in MyCounter(1,3):` will iterate through 1,2,3 then stop. This example is directly discussed in articles on the iterator protocol. GeeksforGeeks+1

- One nuance: An object can be *just* iterable (its `__iter__()` returns a *different* iterator object) – or the object itself can be the iterator (its `__iter__()` returns `self`). The latter is simpler but has implications if you try to iterate multiple times. Stack Overflow+1

---

4. Mapping back to loops: what happens step-by-step

Let's imagine `some_iterable = [10, 20, 30]`. Then:

1. The `for` loop asks: `iterator = iter(some_iterable)`.
    - For a list, `list.__iter__()` returns an iterator object (of type list_iterator) that has its internal state pointing to the start of the list.
2. Enter loop: call `next(iterator)` → calls its `__next__()`, returns `10`.
3. Execute loop body with `x = 10`.
4. Then loop back: call `next(iterator)` → returns `20`.
5. Loop body with `x = 20`.
6. Repeat: `next(iterator)` → returns `30`.
7. Loop body with `x = 30`.
8. Next: `next(iterator)` → now internal index is beyond the list → raises `StopIteration`.
9. The `for` sees `StopIteration` → exits cleanly, loop ends.
10. The code after the loop executes.

---

5. Why `StopIteration`?

- `StopIteration` is the signal from the iterator that there are no more elements.
- The `for` loop handles `StopIteration` internally — you don't usually see it in your code.
- If you manually used `next(iterator)` outside of a loop, then you would catch `StopIteration` yourself if you wanted to handle exhaustion. Real Python+1

6. Some questions for you (to check understanding)

- What is returned by `iter()` when called on an iterable? Why is this important?
- If an object has only `__next__()` but no `__iter__()`, can you use it in a `for` loop? What will go wrong?
- What is the difference between an object being **iterable** vs being an **iterator**?
- Why do we raise `StopIteration` rather than return something like `None` to indicate the end of iteration?

🧩 1. Iterable vs Iterator

Iterable

An **iterable** is *any* object that can give you an **iterator** when you call `iter()` on it.
Examples:

```
list, tuple, dict, set, str, range, file
```

Formally, an object is *iterable* if it defines a `__iter__()` method that returns an **iterator object**.

```
mylist = [1, 2, 3]
it = iter(mylist)   # internally calls mylist.__iter__()
```

Iterator

An **iterator** is an object that represents a *stream of data*.
It has two required methods:

- `__iter__()` → returns itself (the iterator object itself)

- `__next__()` → returns the next item in the stream; when finished, raises `StopIteration`

```
next(it)   # internally calls it.__next__()
```

⚙️ 2. How iteration tools (`for`, comprehensions) work internally

Python's `for` loop, generator expressions, and list comprehensions **all depend** on the iterator protocol:

```
for element in iterable:
    # body
```

is equivalent to:

```
_iterator = iter(iterable)
while True:
    try:
        element = next(_iterator)
    except StopIteration:
        break
    # body
```

So the `for` loop doesn't have special syntax — it's just hiding these `iter()` and `next()` calls and automatically catching `StopIteration`.

🔍 3. `__next__` and the role of StopIteration

The `__next__()` method **returns one element at a time**.
When there are no elements left, it must **raise** `StopIteration`.

If it *didn't*, the loop would go on forever, because Python wouldn't know it's finished.

You can think of the iterator as having an **internal pointer** (a “cursor”) that moves through the data.
Calling `__next__()` moves the pointer one step forward.

Example:

```
myList = [1, 2, 3, 4]
I = iter(myList)

print(next(I))  # 1
print(next(I))  # 2
print(next(I))  # 3
print(next(I))  # 4
print(next(I))  # StopIteration!
```

🧠 4. The “memory address” confusion

You noticed this:

```
I = iter(myList)
print(I)
print(next(I))
print(I)
```

Output:

```
<list_iterator object at 0x1010a0c10>
1
<list_iterator object at 0x1010a0c10>
```

Why doesn’t the memory address change?
Because the *iterator object itself* doesn’t change — it’s the *internal pointer* inside it that advances.
Think of it like a **bookmark inside a book** — the book is the same, but the bookmark moves to the next page internally.

So:

- The **iterator object** stays at the same memory address.
- The **cursor position** inside the iterator changes each time you call `__next__()`.

📂 5. Why `file` objects are special iterables

When you open a file:

```
f = open('main.py')
```

That `f` object is **already** an iterator.

✅ So you don't need:

```
iter(f)
```

because `f.__iter__()` returns itself (`return self`).

You can verify:

```
print(iter(f) is f)          # True
print(f.__iter__() is f)     # True
```

This is **not true for lists**, because:

```
L = [1, 2, 3]
print(iter(L) is L)   # False
```

- `list` objects are **iterables** (they can produce iterators)
- `file` objects are **iterators** (they *are* their own iterator)

📜 6. Why `readline()` vs `__next__()`

When you do:

```
line = f.__next__()
```

You’re using the raw iterator protocol. If the file ends, `__next__()` raises `StopIteration`.
If you don’t handle it, your program **crashes**.

But when you do:

```
line = f.readline()
```

`readline()` catches `StopIteration` internally and instead returns an **empty string (`""`)** when there’s no more data.
That’s why the recommended safe loop is:

```
while True:
    line = f.readline()
    if not line:
        break
    print(line, end='')
```

This handles the end-of-file gracefully without exceptions.

🧱 7. `.readlines()` and memory usage

`.readlines()` loads the **entire file into memory** as a list of strings.

```
lines = open('main.py').readlines()
```

For small files this is fine, but for large files (say 5 GB logs), it will crash your memory.

That’s why modern Python prefers:

```
for line in open('main.py'):
    print(line, end='')
```

This uses the **iterator** of the file — it reads one line at a time, not all at once.

🧭 8. Dictionary and range examples

Dictionary

A `dict` is also iterable.
 Iterating over it gives **keys** by default:

```
D = {'a': 1, 'b': 2}
I = iter(D)

print(next(I))  # 'a'
print(next(I))  # 'b'
```

When there are no keys left, `StopIteration` is raised.

Range

A `range` object is a lightweight sequence object that *generates numbers on demand*, not a prebuilt list.

```
R = range(5)
I = iter(R)

print(next(I))  # 0
print(next(I))  # 1
print(next(I))  # 2
print(next(I))  # 3
print(next(I))  # 4
print(next(I))  # StopIteration
```

So `range` is also iterable, but its iterator stops when it reaches the end.

⚡ Summary (in plain words)

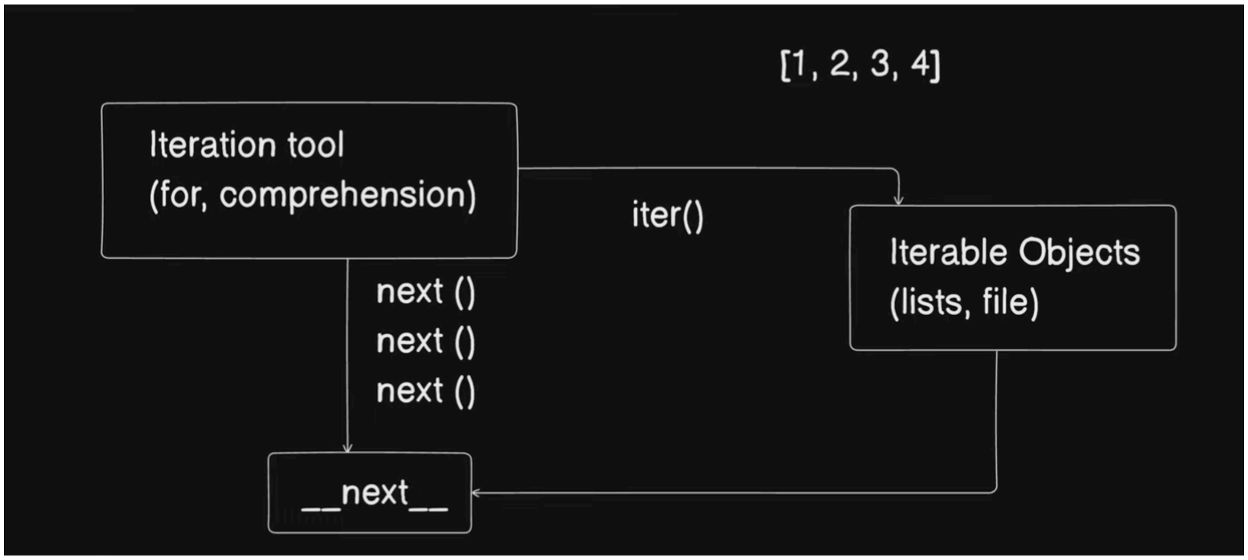
Concept	Meaning	Example	Note
Iterable	Can be looped over (has `__iter__()`)	list, str, dict, file, range	Gives you an iterator
Iterator	Remembers position, has `__next__()`	object returned by `iter()`	Returns next element, raises `StopIteration`
`for` loop	Automatically calls `iter()` + `next()`	`for x in mylist:`	Stops when `StopIteration` raised
`file` objects	Are already iterators	`f = open('a.txt')`	`iter(f) is f` is True
`readline()`	Safe way to read one line	returns `""` at EOF	avoids `StopIteration`
`__next__()`	Core method to get next element	`next(iterator)`	manually used rarely
`StopIteration`	Signals "no more data"	internally handled by `for`	you don't usually catch it

---

🧩 Mental model summary

Think of it like a **Netflix playlist**:

- The **iterable** is the entire playlist.
- The **iterator** is your remote control + the current episode tracker.
- `next()` = go to next episode.
- When the series ends → `StopIteration`.
- The `for` loop is an "auto-play" system that keeps hitting next until the show ends.



🧩 Overall idea

The diagram shows how Python connects:

- **Iterable objects** (like lists, files, etc.)
- **Iteration tools** (`for` loops, comprehensions)
- **Iterator methods** (`__iter__()`, `__next__()`)

This is the behind-the-scenes wiring that powers every `for` loop.

1️⃣ Iterable Objects box (right side)

```
Iterable Objects (lists, file)
```

This box represents any object that can be looped over — for example:

```
[1, 2, 3, 4]
open('main.py')
range(5)
```

These objects are **iterables** because they have a method called `__iter__()` which returns an **iterator**.

So when Python sees something like:

```
for x in [1, 2, 3, 4]:
```

It internally calls:

```
iter([1, 2, 3, 4])
```

That's what the arrow labeled `iter()` in the diagram means:
➡️ it points from "Iterable Objects" to "Iteration tool".

2️⃣ Iteration tool box (left side)

```
Iteration tool
(for, comprehension)
```

This represents anything in Python that *drives* iteration — examples:

- `for` loops
- list/set/dict comprehensions
- `sum()`, `max()`, `min()` — these also use iteration internally!

When you write:

```
for num in [1, 2, 3, 4]:
    print(num)
```

What happens internally is:

```
_iterator = iter([1, 2, 3, 4])
while True:
    try:
        num = next(_iterator)
        print(num)
    except StopIteration:
        break
```

That's what the **"next() → next() → next()"** arrows in the diagram represent —
the `for` loop repeatedly calling `next()` on the iterator.

3️⃣ `__next__` box (bottom)

```
__next__
```

This represents the **method on the iterator object** that actually returns each item.

Every time the iteration tool (like a `for` loop) calls `next()`, Python internally does:

```
_iterator.__next__()
```

So the flow is:

```
Iteration tool  →  next()  →  iterator's __next__()
```

Each call to `__next__()` returns one item from the data and advances the internal cursor.

When there's nothing left, it raises `StopIteration`.

That's how Python knows the loop is over.

4️⃣ The circular flow (bottom right arrow back to iterable)

Notice that the arrow loops back from `__next__` to the "Iterable Object".

That means `__next__()` doesn't live on the **original iterable** (like a list),
but on the **iterator** returned by `iter()`.

So conceptually:

1. You start with `[1,2,3,4]` (an iterable)
2. `iter()` creates an *iterator object* that remembers your current position
3. Each `__next__()` call fetches the next element *from that iterable's data*

So the loop "flows" through these steps:

`for → iter() → iterator → __next__() → element`

5️⃣ The arrows in meaning order

Let's "read" the diagram in logical flow:

1. **Iterable Objects** (`[1,2,3,4]`, `file`, `range`, etc.)
2. The **iteration tool** (`for`, comprehension) calls **`iter()`** on it
   ➡️ Produces an **iterator**
3. Then repeatedly calls **`next()`** on that iterator
   ➡️ Which calls the iterator's **`__next__()`** method
4. `__next__()` yields one value at a time
   ➡️ When finished, raises `StopIteration`
5. The `for` loop catches that exception internally and stops looping

---

🧠 Example matching the diagram

Let's write code that literally represents the flow:

```
data = [1, 2, 3, 4]       # Iterable Object
it = iter(data)            # iter() called by the for loop internally

print(next(it))  # -> calls __next__() and returns 1
print(next(it))  # -> returns 2
print(next(it))  # -> returns 3
print(next(it))  # -> returns 4
print(next(it))  # -> raises StopIteration
```

Now, replace the `next()` calls with a `for` loop:

```
for num in data:
    print(num)
```

Internally, Python does the same 5 steps from the diagram — just automatically.

6️⃣ Special case — file object

In the diagram's context, the example `file` is special because:

```
f = open('main.py')
```

- The `file` object **is already its own iterator**.
  So `iter(f)` just returns `f` itself.
- Each call to `next(f)` or `f.__next__()` reads the next line.
- When the file ends → raises `StopIteration`.

That's why you can write:

```
for line in open('main.py'):
    print(line)
```

and it works perfectly — same mechanism as `[1, 2, 3, 4]`.

🔁 In summary, the diagram shows:

Ste p	Action	Who does it	Explanation
1	`iter()`	`for` loop	Converts iterable into iterator
2	`next()` repeatedly	`for` loop	Asks iterator for next item
3	`__next__()`	Iterator object	Returns next value or raises `StopIteration`
4	Loop ends	`for` loop	When `StopIteration` occurs

So this small diagram basically visualizes the **entire iterator protocol** in Python —
the invisible mechanism that powers all loops, comprehensions, generators, and many built-in functions.