Python `range()` Function

Introduction

When working with `for` loops, you often need a sequence of numbers.

For example:

- Count from 1 to 10
- Print even numbers
- Generate a countdown
- Repeat an action 5 times
- Process items using their indexes
- Create numeric sequences for algorithms
- Build patterns and tables

Python provides the `range()` function for this purpose.

The basic idea is:

> **`range()` generates a sequence of numbers according to a start, stop, and optional step.**

Example:

```
for number in range(5):
    print(number)
```

Output:

```
0
1
2
3
4
```

Understanding `range()` properly is important because it appears constantly in Python loops, indexing, algorithms, patterns, and problem solving.

---

1. Why Do We Need `range()`?

Suppose you want to print numbers from 0 to 4.

Without `range()`:

```
print(0)
print(1)
print(2)
print(3)
print(4)
```

This is repetitive.

With `range()`:

```
for number in range(5):
    print(number)
```

The loop automatically receives each number.

```
range(5)
   ↓
0
1
2
3
4
```

This makes the code shorter, clearer, and scalable.

---

2. The Mental Model

Think of:

```
range(5)
```

as instructions saying:

> Start at 0 and keep increasing by 1 until you reach 5, but do not include 5.

So:

```
0 → 1 → 2 → 3 → 4 → STOP
```

The most important rule is:

> **The stop value is excluded.**

This is called a **half-open range**.

---

3. Basic Syntax

Python provides three common forms:

```
range(stop)
```

```
range(start, stop)
```

```
range(start, stop, step)
```

The arguments mean:

Argument	Meaning
`start`	Where the sequence begins
`stop`	Where the sequence stops before
`step`	How much the value changes each time

---

4. `range(stop)`

The simplest form is:

```
range(5)
```

It produces:

```
0
1
2
3
4
```

Why doesn't it start at 1?

Because when only one argument is provided, Python assumes:

```
start = 0
step = 1
```

So:

```
range(5)
```

is conceptually:

```
range(0, 5, 1)
```

---

5. Why Is 5 Excluded?

This is one of the most important things to understand.

```
range(1, 5)
```

produces:

```
1
2
3
4
```

not:

```
1
2
3
4
5
```

The stop value marks the boundary where the sequence ends.

Think:

```
start included
stop excluded
```

Therefore:

```
range(1, 6)
```

is what you use when you want:

```
1 2 3 4 5
```

---

6. `range(start, stop)`

You can specify where the sequence starts.

```
for number in range(3, 8):
    print(number)
```

Output:

```
3
4
5
6
7
```

Here:

```
start = 3
stop = 8
step = 1
```

The sequence stops before 8.

---

7. `range(start, stop, step)`

The third argument controls the amount by which the number changes.

```
for number in range(0, 11, 2):
    print(number)
```

Output:

```
0
2
4
6
8
10
```

Here:

```
start = 0
stop = 11
step = 2
```

The sequence moves by 2 each time.

---

8. Understanding Step

Compare:

```
range(0, 6, 1)
```

```
0 1 2 3 4 5
```

and:

```
range(0, 6, 2)
```

```
0 2 4
```

and:

```
range(0, 6, 3)
```

```
0 3
```

The `step` determines how quickly the sequence moves.

---

9. Step Does Not Mean Number of Values

This is a common misunderstanding.

In:

```
range(0, 10, 2)
```

the `2` does **not** mean "generate two numbers."

It means:

> Move forward by 2 after every value.

So:

```
0
2
4
6
8
```

---

10. Counting From 1

If you want:

```
1
2
3
4
5
```

write:

```
for number in range(1, 6):
    print(number)
```

Remember:

```
start = 1
stop = 6
```

Since 6 is excluded, the last value is 5.

---

11. Counting to 10

To print 1 through 10:

```
for number in range(1, 11):
    print(number)
```

Output:

```
1
2
3
4
5
6
7
8
9
10
```

The common pattern is:

```
range(1, desired_last_number + 1)
```

So:

```
range(1, 11)
```

for 1–10.

---

12. Counting Down

`range()` can also move backward.

You need a negative step.

```
for number in range(5, 0, -1):
    print(number)
```

Output:

```
5
4
3
2
1
```

Here:

```
start = 5
stop = 0
step = -1
```

Because the stop value is excluded, 0 isn't printed.

---

13. Why Does the Step Need to Be Negative?

Consider:

```
range(5, 0)
```

The default step is:

```
+1
```

Python would try to move:

```
5 → 6 → 7 → 8 → ...
```

But the target boundary is 0.

So the sequence cannot progress toward the stop value.

For descending sequences, use:

```
-1
```

Example:

```
range(5, 0, -1)
```

---

14. Countdown From 10

```
for number in range(10, 0, -1):
    print(number)
```

Output:

```
10
9
8
7
6
5
4
3
2
1
```

This pattern is useful for:

- countdown timers
- reverse iteration
- decreasing values
- algorithmic problems
- game logic

---

15. Counting Down by 2

```
for number in range(10, 0, -2):
    print(number)
```

Output:

```
10
8
6
4
2
```

The step is `-2`.

---

16. Even Numbers

You can generate even numbers using:

```
for number in range(0, 11, 2):
    print(number)
```

Output:

```
0
2
4
6
8
10
```

This is often cleaner than generating every number and checking:

```
if number % 2 == 0:
```

when you already know you only need even numbers.

---

17. Odd Numbers

Start at 1 and use a step of 2:

```
for number in range(1, 11, 2):
    print(number)
```

Output:

```
1
3
5
7
9
```

---

18. `range()` With a `for` Loop

The most common use is:

```
for number in range(5):
    print(number)
```

Execution:

```
range(5)
   ↓
0 → number
   ↓
1 → number
   ↓
2 → number
   ↓
3 → number
   ↓
4 → number
   ↓
finished
```

The loop variable receives one value at a time.

---

19. Repeating an Action

Sometimes you don't actually care about the number.

You simply want to repeat something a fixed number of times.

```
for _ in range(5):
    print("Hello")
```

Output:

```
Hello
Hello
Hello
Hello
Hello
```

The `_` communicates:

> I don't need the current value.

---

20. Why Use `_`?

You could write:

```
for number in range(5):
    print("Hello")
```

But `number` isn't being used.

A cleaner convention is:

```
for _ in range(5):
    print("Hello")
```

This tells another developer:

> The loop variable itself isn't important.

---

21. `range()` and Indexes

One important use of `range()` is working with indexes.

Suppose:

```
names = ["Ali", "Sara", "Ahmed"]
```

You could write:

```
for index in range(len(names)):
    print(index)
```

Output:

```
0
1
2
```

Then:

```
for index in range(len(names)):
    print(names[index])
```

Output:

```
Ali
Sara
Ahmed
```

---

22. `range(len(list))`

This pattern is sometimes useful when you specifically need indexes:

```
for index in range(len(names)):
    print(index, names[index])
```

Output:

```
0 Ali
1 Sara
2 Ahmed
```

However, when you need both the index and value, Python usually provides a cleaner option:

```
for index, name in enumerate(names):
    print(index, name)
```

So remember:

> Use `range(len(...))` when you genuinely need index-based logic. Use `enumerate()` when you simply need the index and item together.

---

23. `range()` and List Indexing

Suppose:

```
languages = ["Python", "JavaScript", "Dart", "Java"]
```

The indexes are:

```
Python      → 0
JavaScript  → 1
Dart        → 2
Java        → 3
```

You can process them:

```
for index in range(len(languages)):
    print(index, languages[index])
```

Output:

```
0 Python
1 JavaScript
2 Dart
3 Java
```

---

24. `range()` Does Not Directly Return a List

Consider:

```
numbers = range(5)
```

`numbers` is a `range` object.

You can check:

```
print(type(numbers))
```

Output:

```
<class 'range'>
```

If you specifically need a list:

```
numbers = list(range(5))
```

Now:

```
print(numbers)
```

Output:

```
[0, 1, 2, 3, 4]
```

---

25. Why Is `range()` Its Own Object?

Python doesn't need to create and store every number as a normal list just to iterate over them.

For example:

```
range(1000000)
```

represents a numeric sequence without requiring a million-element list to be created first.

This makes `range()` efficient for iteration.

---

26. Converting range() to a List

You can convert it:

```
numbers = list(range(5))
```

Result:

```
[0, 1, 2, 3, 4]
```

You might do this when you actually need a list object.

But if you're simply looping:

```
for number in range(5):
    print(number)
```

there is usually no reason to convert it to a list first.

---

27. Negative Numbers

`range()` can generate negative values.

```
for number in range(-3, 4):
    print(number)
```

Output:

```
-3
-2
-1
0
1
2
3
```

The same start-inclusive, stop-exclusive rule still applies.

---

28. Negative Step With Negative Numbers

```
for number in range(3, -4, -1):
    print(number)
```

Output:

```
3
2
1
0
-1
-2
-3
```

The stop value `-4` is excluded.

---

29. Empty Ranges

Sometimes `range()` produces no values.

For example:

```
range(5, 1)
```

The default step is `+1`.

Python would need to move:

```
5 → 6 → 7 → ...
```

but it cannot reach the lower stop boundary.

Therefore:

```
for number in range(5, 1):
    print(number)
```

prints nothing.

---

30. Empty Descending Range

Similarly:

```
range(1, 5, -1)
```

doesn't produce values because the sequence is supposed to decrease, but the stop boundary is higher.

For descending:

```
range(5, 1, -1)
```

works:

```
5
4
3
2
```

---

31. Important Rule for Direction

Ask:

> **Is my stop value greater or smaller than my start value?**

Moving upward

Use a positive step:

```
range(1, 6, 1)
```

Moving downward

Use a negative step:

```
range(5, 0, -1)
```

A simple mental model:

```
Start < Stop → usually positive step
Start > Stop → usually negative step
```

---

32. The Three Forms

Memorize these:

One argument

```
range(stop)
```

Example:

```
range(5)
```

Means:

```
0 1 2 3 4
```

Two arguments

```
range(start, stop)
```

Example:

```
range(2, 6)
```

Means:

```
2 3 4 5
```

Three arguments

```
range(start, stop, step)
```

Example:

```
range(2, 10, 2)
```

Means:

```
2 4 6 8
```

---

33. Common `range()` Examples

```
range(5)
```

```
0 1 2 3 4
```

```
range(1, 5)
```

```
1 2 3 4
```

```
range(2, 10, 2)
```

```
2 4 6 8
```

```
range(10, 0, -1)
```

```
10 9 8 7 6 5 4 3 2 1
```

```
range(10, 0, -2)
```

```
10 8 6 4 2
```

---

34. `range()` With Conditions

You can combine `range()` with `if`.

```
for number in range(1, 11):
    if number % 2 == 0:
        print(number)
```

Output:

```
2
4
6
8
10
```

But if you already know you want even numbers, this is more direct:

```
for number in range(2, 11, 2):
    print(number)
```

Both are valid.

The second expresses the intended sequence directly.

---

35. `range()` for Multiplication Tables

A classic beginner exercise is a multiplication table.

```
number = 5

for i in range(1, 11):
    print(number * i)
```

Output:

```
5
10
15
20
25
30
35
40
45
50
```

You can make it more readable:

```
number = 5

for i in range(1, 11):
    print(f"{number} × {i} = {number * i}")
```

Output:

```
5 × 1 = 5
5 × 2 = 10
5 × 3 = 15
...
5 × 10 = 50
```

---

36. `range()` for Patterns

You can use `range()` to control pattern size.

```
for row in range(1, 6):
    print("*" * row)
```

Output:

```
*
**
***
****
*****
```

Here `range()` determines how many rows are generated.

---

37. Nested `range()` Loops

You can use multiple ranges together.

```
for row in range(3):
    for column in range(4):
        print("*", end=" ")

    print()
```

Output:

```
* * * *
* * * *
* * * *
```

The outer `range(3)` controls rows.

The inner `range(4)` controls columns.

---

38. `range()` for a Grid

Suppose you are building a game board.

```
rows = 3
columns = 3

for row in range(rows):
    for column in range(columns):
        print(row, column)
```

Output:

```
0 0
0 1
0 2
1 0
```

```
1 1
1 2
2 0
2 1
2 2
```

This pattern appears in:

- grids
- matrices
- board games
- image processing
- simulations
- algorithms

---

39. `range()` for Fixed Repetition

Suppose you want to run a task 10 times:

```
for _ in range(10):
    print("Running task")
```

This is different from:

```
while condition:
```

because you already know the number of repetitions.

Mental model:

```
Need exactly 10 repetitions
        ↓
Use range(10)
        ↓
for loop
```

---

40. `range()` With Functions

You can call a function repeatedly.

```
def greet():
    print("Hello")

for _ in range(3):
    greet()
```

Output:

```
Hello
Hello
Hello
```

The function controls the task.

`range()` controls how many times it happens.

---

41. `range()` in Real Applications

Suppose a system needs to generate five placeholder resources:

```
for number in range(5):
    print(f"Resource {number + 1}")
```

Output:

```
Resource 1
Resource 2
Resource 3
Resource 4
Resource 5
```

Or suppose a dashboard needs to generate 12 monthly labels:

```
for month in range(1, 13):
    print("Month:", month)
```

`range()` is useful whenever the numeric sequence itself is part of the logic.

---

42. Haas.dev Example

Imagine you're creating a simple resource dashboard and want to display numbered resources:

```
resources = [
    "Python Variables",
    "Python Conditions",
    "Python Loops",
    "Python Functions"
]

for index in range(len(resources)):
    print(f"{index + 1}. {resources[index]}")
```

Output:

```
1. Python Variables
2. Python Conditions
3. Python Loops
4. Python Functions
```

This demonstrates how `range()` can work with indexes.

However, Python provides a more direct solution:

```
for index, resource in enumerate(resources, start=1):
    print(f"{index}. {resource}")
```

Knowing both approaches is useful.

---

43. `range()` and `enumerate()`

These can sometimes solve similar problems, but they have different purposes.

`range()`

Creates a numeric sequence.

```
for i in range(5):
    print(i)
```

`enumerate()`

Provides an index while iterating through an existing iterable.

```
for i, item in enumerate(items):
    print(i, item)
```

Use `range()` when you need numbers.

Use `enumerate()` when you need positions of items.

---

44. `range()` and `while`

You can sometimes implement the same repetition with either.

Using `for`:

```
for number in range(5):
    print(number)
```

Using `while`:

```
number = 0

while number < 5:
    print(number)
    number += 1
```

Both produce:

```
0
1
2
3
4
```

But the `for` version directly expresses:

> Iterate through the numbers 0 through 4.

The `while` version requires manually managing the counter.

---

45. Why `for range()` Is Usually Better for Fixed Counts

Compare:

```
number = 0

while number < 10:
    print(number)
    number += 1
```

with:

```
for number in range(10):
    print(number)
```

The second version is simpler.

There is no manual:

```
number += 1
```

and less opportunity to accidentally create an infinite loop.

---

46. Common Mistake: Off By One

One of the most common errors is writing:

```
range(1, 10)
```

when you actually want:

```
1 through 10
```

But:

```
range(1, 10)
```

produces only:

```
1 through 9
```

Correct:

```
range(1, 11)
```

This type of error is called an **off-by-one error**.

---

47. How to Avoid Off By One Errors

Before writing a range, ask:

What should be the first value?

That's your `start`.

What should be the last value?

Because `stop` is excluded:

```
stop = last value + 1
```

For example, desired:

```
1, 2, 3, 4, 5
```

Write:

```
range(1, 6)
```

---

48. Common Mistake: Forgetting Negative Step

Wrong:

```
for number in range(10, 0):
    print(number)
```

This prints nothing.

Correct:

```
for number in range(10, 0, -1):
    print(number)
```

---

49. Common Mistake: Wrong Step Direction

Wrong:

```
range(1, 10, -1)
```

You are starting at 1 but trying to move downward toward 10.

Correct for ascending:

```
range(1, 10, 1)
```

Correct for descending:

```
range(10, 1, -1)
```

---

50. Common Mistake: Thinking the Stop Is Included

Wrong assumption:

```
range(5, 10)
```

means:

```
5 6 7 8 9 10
```

Actual result:

```
5 6 7 8 9
```

Remember:

> **Start included, stop excluded.**

---

51. Common Mistake: Using a Decimal Step

This does not work:

```
range(0, 1, 0.1)
```

`range()` requires integer arguments.

If you need decimal values, other approaches are more appropriate.

For example, you can generate integer steps and calculate the decimal:

```
for i in range(11):
    value = i / 10
    print(value)
```

Output:

```
0.0
0.1
0.2
...
1.0
```

This is often safer than trying to use floating-point values as direct range steps.

---

52. `range()` Requires Integer-Like Arguments

Valid:

```
range(10)
```

```
range(1, 10)
```

```
range(1, 10, 2)
```

Not valid:

```
range(0.5, 5)
```

or:

```
range(1, 10, 0.5)
```

The standard `range()` function works with integers.

---

53. Step Cannot Be Zero

This is invalid:

```
range(1, 10, 0)
```

Python raises an error because a sequence cannot move forward or backward with a step of zero.

You need a non-zero step:

```
range(1, 10, 1)
```

or:

```
range(10, 1, -1)
```

---

54. Using Variables With range()

The values don't have to be hardcoded.

```
start = 1
end = 10

for number in range(start, end + 1):
    print(number)
```

Output:

```
1
2
3
4
5
6
7
8
9
10
```

This makes your program more flexible.

---

55. User Controlled range()

You can ask the user for values:

```
start = int(input("Start: "))
end = int(input("End: "))

for number in range(start, end + 1):
    print(number)
```

If the user enters:

```
3
7
```

the output is:

```
3
4
5
6
7
```

This is a simple example of combining:

- input
- type conversion
- variables
- `range()`
- `for` loops

---

56. Even Numbers Between User Inputs

```
start = int(input("Start: "))
end = int(input("End: "))

for number in range(start, end + 1):
    if number % 2 == 0:
        print(number)
```

A more direct approach can sometimes adjust the starting point and use a step of 2, but the conditional version is useful when you're learning the relationship between loops and conditions.

---

57. Range Length

You can find how many values a range contains:

```
numbers = range(1, 6)

print(len(numbers))
```

Output:

```
5
```

This works because the range represents five values:

```
1 2 3 4 5
```

---

58. Membership Testing

You can check whether a number belongs to a range:

```
numbers = range(1, 11)

print(5 in numbers)
```

Output:

```
True
```

And:

```
print(15 in numbers)
```

Output:

```
False
```

This can be useful when working with numeric boundaries.

---

59. Converting Range to Different Structures

You can convert a range to a list:

```
list(range(5))
```

Result:

```
[0, 1, 2, 3, 4]
```

You can also create a tuple:

```
tuple(range(5))
```

Result:

```
(0, 1, 2, 3, 4)
```

But conversion isn't necessary when you simply need to iterate.

---

60. Range Is Reusable

A range object can be iterated over multiple times.

```
numbers = range(1, 4)

for number in numbers:
    print(number)
```

```
for number in numbers:
    print(number)
```

Both loops can iterate over the range.

This differs from some one-time iterator objects.

---

61. Advanced: The Internal Structure

You can inspect a range:

```
numbers = range(2, 10, 2)

print(numbers)
```

Output:

```
range(2, 10, 2)
```

It stores the rules describing the sequence:

```
start = 2
stop = 10
step = 2
```

It does not need to display itself as:

```
[2, 4, 6, 8]
```

unless you convert it to a list.

---

62. Advanced: `start`, `stop`, and `step`

A range object exposes these properties:

```
numbers = range(2, 10, 2)

print(numbers.start)
print(numbers.stop)
print(numbers.step)
```

Output:

```
2
10
2
```

This can help you understand that `range()` represents a sequence definition.

---

63. Advanced: Large Ranges

This is possible:

```
numbers = range(10_000_000)
```

The range object itself is compact.

But be careful if you convert it into a list:

```
numbers = list(range(10_000_000))
```

Now Python must create and store all those values as list elements.

So:

```
range(10_000_000)
```

and:

```
list(range(10_000_000))
```

have very different memory implications.

---

64. `range()` in Algorithm Design

Many algorithms need to process positions.

For example:

```
numbers = [10, 20, 30, 40]

for i in range(len(numbers)):
    print(i)
```

Output:

```
0
1
2
3
```

This becomes useful when the algorithm needs to compare an item with another position.

For example:

```
for i in range(len(numbers) - 1):
    print(numbers[i], numbers[i + 1])
```

Output:

```
10 20
20 30
30 40
```

This type of indexing is common in DSA problems.

---

65. `range()` in Pair Comparisons

Suppose you want to compare neighboring values.

```
numbers = [10, 20, 15, 30]

for i in range(len(numbers) - 1):
    if numbers[i] > numbers[i + 1]:
        print("Decrease found")
```

The range controls valid indexes.

Why:

```
len(numbers) - 1
```

?

Because the code accesses:

```
numbers[i + 1]
```

The last valid index cannot be used with `i + 1`.

This is where understanding range boundaries becomes important for avoiding index errors.

---

66. `range()` and Nested Algorithms

Suppose you want to compare every pair of elements:

```
numbers = [1, 2, 3, 4]

for i in range(len(numbers)):
    for j in range(i + 1, len(numbers)):
        print(numbers[i], numbers[j])
```

Output:

```
1 2
1 3
1 4
2 3
2 4
3 4
```

This is a common pattern in algorithmic problem solving.

The range boundaries determine exactly which pairs are processed.

---

67. Haas.dev Example: Resource Numbering

Suppose you have:

```
resources = [
    "Python Variables",
    "Python Conditions",
    "Python Loops",
    "Python Functions",
    "Python Lists"
]
```

Using `range()`:

```
for index in range(len(resources)):
    print(f"{index + 1}. {resources[index]}")
```

Output:

```
1. Python Variables
2. Python Conditions
3. Python Loops
4. Python Functions
5. Python Lists
```

The index starts at 0, so:

```
index + 1
```

creates human-friendly numbering.

---

68. Haas.dev Example: Generating Resource IDs

Suppose you need temporary numeric IDs:

```
for number in range(1, 6):
    resource_id = f"PY-{number:03}"
    print(resource_id)
```

Output:

```
PY-001
PY-002
PY-003
PY-004
PY-005
```

This shows how `range()` can provide a controlled sequence that is used inside another operation.

---

69. Haas.dev Example: Page Numbers

Imagine a resource viewer with 10 pages:

```
for page in range(1, 11):
    print("Page", page)
```

Output:

```
Page 1
Page 2
...
Page 10
```

This is a simple example of numeric iteration in an application.

---

70. Choosing the Correct range()

Before writing `range()`, identify four things:

1. First value

What number should appear first?

2. Last value

What number should appear last?

3. Direction

Are you going upward or downward?

4. Step

How much should the value change?

Then translate:

```
First value → start
Boundary after last value → stop
Amount of movement → step
```

Example:

Desired:

```
10, 8, 6, 4, 2
```

Think:

```
start = 10
stop = 0
step = -2
```

Therefore:

```
range(10, 0, -2)
```

---

71. A Quick Translation Method

Desired:

```
1 2 3 4 5
```

Write:

```
range(1, 6)
```

Desired:

```
0 1 2 3 4
```

Write:

```
range(5)
```

Desired:

```
2 4 6 8 10
```

Write:

```
range(2, 11, 2)
```

Desired:

```
10 9 8 7 6 5 4 3 2 1
```

Write:

```
range(10, 0, -1)
```

Desired:

```
10 8 6 4 2
```

Write:

```
range(10, 0, -2)
```

---

72. Common Mistakes Checklist

When your `range()` isn't working, check:

- Did you remember that `stop` is excluded?
- Did you choose the correct starting number?
- Does the step have the correct sign?
- Is the step zero?
- Are you accidentally using a decimal?
- Are you trying to count downward without a negative step?
- Are your index boundaries correct?
- Do you actually need `range()`, or would direct iteration be clearer?

---

73. Debugging range()

You can inspect a range by converting it to a list:

```
print(list(range(1, 10, 2)))
```

Output:

```
[1, 3, 5, 7, 9]
```

This is a very useful debugging technique.

If you're unsure what your range will produce, test:

```
list(your_range)
```

For example:

```
print(list(range(10, 0, -2)))
```

Output:

```
[10, 8, 6, 4, 2]
```

---

74. Mini Quiz

Question 1

What does this produce?

```
range(5)
```

A.

```
1 2 3 4 5
```

B.

```
0 1 2 3 4
```

C.

```
0 1 2 3 4 5
```

D.

```
5 4 3 2 1
```

Question 2

Which range produces:

```
1 2 3 4 5
```

A.

```
range(1, 5)
```

B.

```
range(1, 6)
```

C.

```
range(0, 5)
```

D.

```
range(5)
```

Question 3

What does this produce?

```
range(10, 0, -2)
```

A.

```
10 8 6 4 2
```

B.

```
10 8 6 4 2 0
```

C.

```
0 2 4 6 8 10
```

D. Nothing

---

75. Mini Quiz Answers

Question 1

B

```
range(5)
```

produces:

```
0 1 2 3 4
```

Question 2

B

```
range(1, 6)
```

produces:

```
1 2 3 4 5
```

because 6 is excluded.

Question 3

A

```
range(10, 0, -2)
```

produces:

```
10 8 6 4 2
```

The stop value 0 is excluded.

---

76. 5 Minute Challenge

Create a program that prints:

```
1
2
3
4
5
6
7
8
9
10
```

using `range()`.

Then modify it to print:

```
2
4
6
8
10
```

Then create a countdown:

```
10
9
8
7
6
5
4
3
2
1
```

Finally, print:

```
10
8
6
4
2
```

without using an `if` condition.

The goal is to become comfortable choosing:

```
start
stop
step
```

---

77. Practice Exercises

Exercise 1

Print numbers from 0 to 20.

Exercise 2

Print numbers from 1 to 50.

Exercise 3

Print even numbers from 0 to 100.

Exercise 4

Print odd numbers from 1 to 99.

Exercise 5

Print a countdown from 20 to 1.

Exercise 6

Print multiples of 5 from 5 to 50.

Expected:

```
5
10
15
```

```
20
25
30
35
40
45
50
```

Exercise 7

Print numbers from -10 to 10.

Exercise 8

Print:

```
100
90
80
70
...
10
```

Exercise 9

Ask the user for a number and print its multiplication table from 1 to 10.

Exercise 10

Given:

```
languages = ["Python", "JavaScript", "Dart", "Java"]
```

Use `range()` to print every language with its index.

Exercise 11

Print this pattern:

```
*
**
***
****
*****
```

using `range()`.

Exercise 12

Print this pattern:

```
*****
****
***
**
*
```

using `range()` and a negative step.

---

78. Mini Project: Multiplication Table Generator

Build a program that asks the user for a number.

Example:

```
Enter a number: 7
```

Then generate:

```
7 × 1 = 7
7 × 2 = 14
7 × 3 = 21
7 × 4 = 28
7 × 5 = 35
7 × 6 = 42
7 × 7 = 49
7 × 8 = 56
7 × 9 = 63
7 × 10 = 70
```

Requirements:

- Use `input()`.
- Convert the input to an integer.
- Use `range()`.
- Use a `for` loop.
- Do not manually write ten `print()` statements.

---

79. Mini Project Extension: Resource Page Generator

Use:

```
resources = [
    "Python Variables",
    "Python Conditions",
    "Python Loops",
    "Python Functions",
    "Python Lists"
]
```

Create numbered output:

```
1. Python Variables
2. Python Conditions
3. Python Loops
4. Python Functions
5. Python Lists
```

First solve it using:

```
range()
```

Then solve it using:

```
enumerate()
```

Compare both approaches.

This teaches an important engineering lesson:

> There may be multiple ways to solve a problem, but some approaches communicate the intent more clearly.

---

80. Interview Questions

Beginner

1. What is `range()` in Python?

`range()` creates a range object representing a sequence of integers.

2. What does `range(5)` produce?

```
0, 1, 2, 3, 4
```

3. Is the stop value included?

No.

4. What are the three arguments of `range()`?

```
start
stop
step
```

5. What is the default start?

`0`.

6. What is the default step?

`1`.

Intermediate

7. How do you count backward with `range()`?

Use a negative step:

```
range(10, 0, -1)
```

8. Why does `range(1, 10)` stop at 9?

Because the stop value is excluded.

9. Can `range()` use decimal values?

No. Standard `range()` works with integers.

10. Can the step be zero?

No.

11. What is the difference between `range()` and `list(range())`?

`range()` creates a range object representing the sequence, while `list(range())` materializes the values into a list.

12. When would you use `enumerate()` instead of `range(len(...))`?

When you need both the index and the corresponding item from an iterable.

---

81. Advanced Thinking: Why Stop Is Exclusive

The exclusive stop boundary is useful because ranges can be combined naturally.

For example:

```
range(0, 5)
```

has five values:

```
0 1 2 3 4
```

This matches the five indexes of a five-element list:

```
index: 0 1 2 3 4
```

So:

```
for i in range(len(items)):
```

naturally generates every valid index.

The same principle also makes adjacent ranges easy to reason about:

```
range(0, 5)
range(5, 10)
```

Together they cover:

```
0 1 2 3 4 5 6 7 8 9
```

without overlapping at 5.

---

82. Advanced Thinking: Off By One Errors

An off-by-one error occurs when your loop runs one time too many or one time too few.

For example:

```
for i in range(1, 10):
```

runs 9 times.

If you expected 10 iterations, you need:

```
for i in range(1, 11):
```

When debugging, always identify:

```
First value
Last intended value
Actual stop boundary
Number of iterations
```

This habit becomes extremely important in DSA and real-world programming.

---

83. Advanced Thinking: range() and Complexity

Consider:

```
for i in range(n):
    print(i)
```

The loop runs approximately `n` times.

Its time complexity is:

```
O(n)
```

Now consider:

```
for i in range(n):
    for j in range(n):
        print(i, j)
```

The inner operation runs approximately:

```
n × n
```

times.

So the complexity is:

```
O(n²)
```

Understanding `range()` therefore becomes useful when learning algorithm complexity.

---

84. Advanced Thinking: Range With Index-Based Algorithms

Consider:

```
numbers = [5, 8, 2, 9, 1]

for i in range(len(numbers)):
    print(numbers[i])
```

Here `range()` generates valid indexes.

But if the algorithm needs neighboring values:

```
for i in range(len(numbers) - 1):
    current = numbers[i]
    next_value = numbers[i + 1]

    print(current, next_value)
```

The `-1` ensures `i + 1` remains a valid index.

This is an example of how understanding boundaries directly affects algorithm correctness.

---

85. `range()` Cheat Sheet

Basic

```
range(5)
```

```
0 1 2 3 4
```

Start and stop

```
range(2, 7)
```

```
2 3 4 5 6
```

Step

```
range(0, 10, 2)
```

```
0 2 4 6 8
```

Countdown

```
range(5, 0, -1)
```

```
5 4 3 2 1
```

Countdown by 2

```
range(10, 0, -2)
```

```
10 8 6 4 2
```

Convert to list

```
list(range(5))
```

```
[0, 1, 2, 3, 4]
```

Fixed repetitions

```
for _ in range(5):
    ...
```

Index iteration

```
for i in range(len(items)):
    ...
```

Remember

```
start → included
stop  → excluded
step  → movement
```

---

86. Final Summary

The `range()` function is one of the most important tools for working with numeric iteration in Python.

The three forms are:

```
range(stop)
```

```
range(start, stop)
```

```
range(start, stop, step)
```

The most important rule is:

> **The start value is included, but the stop value is excluded.**

Examples:

```
range(5)
```

produces:

```
0 1 2 3 4
```

```
range(1, 6)
```

produces:

```
1 2 3 4 5
```

```
range(2, 11, 2)
```

produces:

```
2 4 6 8 10
```

```
range(10, 0, -1)
```

produces:

```
10 9 8 7 6 5 4 3 2 1
```

Use `range()` when you need a numeric sequence or a fixed number of loop iterations.

Use `enumerate()` when you're iterating over an existing collection and need its indexes.

The real skill is not memorizing `range()` syntax. It is being able to translate a requirement such as:

> "Start at 10, go down by 2, and stop before 0."

into:

```
range(10, 0, -2)
```

That ability becomes increasingly important as you move from basic Python into algorithms, data structures, and real application development.

Visit haas.dev for more resources and guides.

haas.dev

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