

Day 5 Loops

Self-Study Guide · About 60 minutes · With answer key

By the end you will be able to

- Write while loops that stop, including loops that keep asking until the user types what is needed.
- Loop over the characters of a string and over range(start, stop, step), and choose between for and while.
- Count, sum, and find the largest value with accumulators, and control a loop with break, continue, and else.
- Trace a loop by hand, including nested loops and loops that use enumerate().

What to remember from earlier

- Assignment replaces a value, so `count = count - 1` works out the right side with the old value and stores the result. ([Day 2: Variables, numbers and basic types, 2.1 Variables: Giving a Value a Name](#))
- The remainder operator `%` gives what is left over after dividing. ([Day 2: Variables, numbers and basic types, 2.4 Floor Division, Remainders, and Powers](#))
- `len()` counts a string's characters, and in asks whether one string is inside another. ([Day 3: Strings, 3.2 Length, Concatenation, and in](#))
- An f-string puts values into text, and a format specifier such as `:.1f` shows one decimal place. ([Day 3: Strings, 3.8 f-strings and Format Specifiers](#))
- `input()` always gives back a string, so convert it with `int()` before doing math with it. ([Day 3: Strings, 3.9 Reading Input](#))
- Conditions joined with `and`, `or`, and `not` decide which indented `if`, `elif`, or `else` block runs. ([Day 4: Making decisions](#))
- `is None` checks whether a name refers to `None`, the value that means "no value yet". ([Day 4: Making decisions, 4.8 Equality Versus Identity](#))

Make a `day05` folder for today's files, open it in your editor, and run each script from its terminal, as on Days 1 to 4.

On Day 4 your programs learned to choose. But every line still ran at most once: the café menu answered one command and stopped, and printing the numbers 1 to 100 would take 100 `print` lines. Today you learn to **repeat**: run the same block again and again, as many times as the job needs, and stop at the right moment.

5.1 while Loops

Day 1 printed a countdown with one `print` per number. That works for 3, 2, 1, but not for a countdown from 1000, and not when the program cannot know in advance how many times to repeat. You need a way to say "keep doing this **while** something is true".

A **loop** runs a block of code repeatedly. A **while loop** checks a condition, runs its block if the condition is `True`, then goes back and checks the condition again. It stops when the condition is `False`. Its shape looks like an `if` statement from Day 4:

```
1 while condition:
2     statement
3     statement
```

- `while` is a keyword, followed by a condition and a colon.
- The indented block is the **loop body**: the lines that repeat.
- Each run of the body is called an **iteration** (or a pass).
- When the condition is `False`, Python skips the body and carries on with the first line after it, back at the left edge.

The difference from `if`: an `if` asks its question once. A `while` asks again after every pass.

WORKED EXAMPLE A countdown

countdown.py

```
1 count = 3
2 while count > 0:
3     print(count)
4     count = count - 1
5 print("Liftoff!")
```

```
3
2
1
Liftoff!
```

Check	count	count > 0	What happens
1st	3	True	Print 3, count becomes 2
2nd	2	True	Print 2, count becomes 1
3rd	1	True	Print 1, count becomes 0
4th	0	False	Leave the loop, print Liftoff!

Line 4 is what makes the loop stop. It uses the Day 2 rule for assignment: work out `count - 1` with the old value, then store the result in `count`. Each pass moves `count` one step closer to making the condition `False`. The condition was checked four times but the body ran three times.

Now change only the starting value, to 0:

countdown_zero.py

```
1 count = 0
2 while count > 0:
3     print(count)
4     count = count - 1
5 print("Liftoff!")
```

```
Liftoff!
```

The very first check, `0 > 0`, is `False`, so the body never runs. A while loop checks **before** each pass, so its body can run zero times.

KEY IDEA

`while condition:` runs its body again and again as long as the condition is `True`, checking before every pass. Something in the body must change the condition, or the loop never ends.

WATCH OUT

The infinite loop. Leave out the line that changes `count`:

```
1 count = 3
2 while count > 0:
3     print(count)
4 print("Liftoff!")
```

`count` stays 3 forever, so `count > 0` is always `True`, and the loop never stops. A loop that never ends is an **infinite loop**. The program keeps printing 3 until you stop it: click in the terminal and press **Ctrl+C** (the Ctrl key and C together). Python stops the program and prints a `KeyboardInterrupt` message. Whenever you write a while loop, find the line in the body that moves the condition toward `False`. The fix here is the line `count = count - 1` from `countdown.py`.

PAUSE AND TRY

Question 1 Tracing

Savings double every week, starting at 10. What does this program print? Make a trace table of `savings` and `weeks` at each check of the condition.

`savings.py`

```
1 savings = 10
2 weeks = 0
3 while savings < 50:
4     savings = savings * 2
5     weeks = weeks + 1
6 print(weeks, savings)
```

5.2 Asking Until the Answer Is Right

The Day 4 menu handled one command and stopped, and Day 4 promised that today you would see how to keep asking. A while loop is the tool, because the program cannot know how many commands the user will type. It keeps going until the user types `quit`.

WORKED EXAMPLE A menu that keeps asking

This run types `Hours`, then `tea`, then `quit`:

keep_asking.py

```
1 command = ""
2 while command != "quit":
3     command = input("Command: ").strip().lower()
4     if command == "hours":
5         print("Open 8:00 to 18:00.")
6     elif command == "price":
7         print("A coffee costs 3.50.")
8     elif command != "quit":
9         print("Unknown command:", command)
10 print("Goodbye!")
```

```
Command: Hours
Open 8:00 to 18:00.
Command: tea
Unknown command: tea
Command: quit
Goodbye!
```

- Line 1 gives `command` a starting value. The condition on line 2 needs a value to compare, and `""` is not `"quit"`, so the first check is `True` and the body runs.
- Each pass reads one command and answers it with the if chain from Day 4.
- When the user types `quit`, the chain prints nothing (the last rung, `elif command != "quit":`, keeps `"quit"` from being called unknown). Then the loop goes back to line 2, `"quit" != "quit"` is `False`, and the program says goodbye.

The same idea checks typed input. A program that needs an age from 1 to 120 can keep asking until it gets one. This run types 0, 200, then 15:

age_until_valid.py

```
1 age = int(input("Age (1 to 120): "))
2 while not 1 <= age <= 120:
3     print("Please type an age from 1 to 120.")
4     age = int(input("Age (1 to 120): "))
5 print(f"Thanks! You are {age}.")
```

```
Age (1 to 120): 0
Please type an age from 1 to 120.
Age (1 to 120): 200
Please type an age from 1 to 120.
Age (1 to 120): 15
Thanks! You are 15.
```

The condition is the Day 4 chained comparison with `not` in front: "while the age is **not** between 1 and 120". The body asks again, which is the line that can make the condition `False`. If the first answer is already valid, the body runs zero times. (If the user types a word instead of a number, `int()` still stops with a `ValueError`; Day 15 shows how to handle that.)

KEY IDEA

To keep asking, give the variable a value before the loop, check it in the condition, and read a new value inside the body. The new value inside the body is what lets the loop end.

WATCH OUT

No value before the first check. Python works out the condition before the first pass, so the variable must already exist:

keep_asking_no_start.py

```
1 while command != "quit":
2     command = input("Command: ").strip().lower()
3     print("You typed", command)
4     print("Goodbye!")
```

```
Traceback (most recent call last):
  File "../day05/keep_asking_no_start.py", line 1, in <module>
    while command != "quit":
          ^^^^^^^
NameError: name 'command' is not defined
```

Line 1 uses `command` before anything has been stored in it. Give it a starting value that lets the loop begin, such as `command = ""`, or read the first answer with `input()` before the loop, as `age_until_valid.py` does.

PAUSE AND TRY**Question 2** Multiple choice

The user types `Open Sesame`, then `open sesame`. How many times does this program print `Wrong, try again.?`

password.py

```
1 password = input("Password: ")
2 while password != "open sesame":
3     print("Wrong, try again.")
4     password = input("Password: ")
5     print("Welcome!")
```

- A. 0 times
- B. 1 time
- C. 2 times
- D. Forever: it is an infinite loop

5.3 for Loops Over Strings

To print each letter of a word on its own line, you could count positions with a while loop and index each one (Day 3). But "do something with each character" is so common that Python has a loop made for it.

A **for loop** runs its body once for **each item** in a sequence. On Day 3 you learned that a string is a sequence of characters, so a for loop over a string runs once per character:

```
1 for name in sequence:
2     statement
```

- `for` is a keyword. `name` is the **loop variable**: before each pass, Python assigns the next item to it.
- `in` is followed by the sequence to go through, then a colon.
- The indented body runs once per item. When there are no items left, the loop ends.

The `in` here is part of the `for` statement. It is not the Day 3 `in` that asks "is this inside that?", although both read naturally as English.

WORKED EXAMPLE Spelling a word

spell_word.py

```
1 word = "loop"
2 for letter in word:
3     print(letter)
4 print("Done")
```

```
l
o
o
p
Done
```

Pass	Letter	Prints
1	"l"	l
2	"o"	o
3	"o"	o
4	"p"	p

There is no condition to write and no counter to update. The loop knows the string has four characters, gives them to `letter` one at a time, and stops after the last one. That is why a for loop over a string cannot run forever.

The body can do anything with the loop variable, such as call a method on it or put it in an f-string:

cheer.py

```
1 name = "Ana"
2 for letter in name:
3     print(f"Give me {letter.upper()}!")
4 print(f"What does that spell? {name}!")
```

```
Give me A!  
Give me N!  
Give me A!  
What does that spell? Ana!
```

The loop variable changes on every pass, but the body's code is the same each time. Line 4 is back at the left edge, so it runs once, after the loop.

KEY IDEA

`for letter in text:` runs the body once per character of `text`, with `letter` holding the current character. The loop ends by itself after the last character.

WATCH OUT

Expecting the loop variable to change the string. This tries to make a word capitals, one letter at a time:

upper_attempt.py

```
1 word = "loop"  
2 for letter in word:  
3     letter = letter.upper()  
4 print(word)
```

loop

Line 3 stores a capital letter in `letter`, but `letter` is just a name for one character. Changing what it refers to does nothing to `word`, and strings cannot be changed anyway (Day 3). To show the word in capitals, use `word.upper()`. To build a changed copy character by character, use the accumulator pattern in 5.6.

PAUSE AND TRY

Question 3 Tracing

What does this program print?

brackets.py

```
1 message = "a b!"  
2 for ch in message:  
3     print(f"[{ch}]")
```

5.4 Counting with range()

Printing the numbers 1 to 100 is not a job about a string. You need a sequence of **numbers** to loop over.

The built-in function `range()` makes a sequence of whole numbers for a for loop to go through. It takes up to three arguments:

Call	Gives
<code>range(stop)</code>	0, 1, 2, ... up to but not including stop
<code>range(start, stop)</code>	start, start + 1, ... up to but not including stop
<code>range(start, stop, step)</code>	start, then adds step each time, stopping before stop

```
>>> for n in range(4):
...     print(n)
...
0
1
2
3

>>> for n in range(2, 6):
...     print(n)
...
2
3
4
5
```

In the REPL, a line ending in a colon starts a block, so the prompt changes to `...` for the indented lines. Press Enter on an empty `...` line to finish the loop and run it. `range(4)` starts at 0 and gives four numbers, 0 to 3. `range(2, 6)` starts at 2 and stops **before** 6.

Why is the stop value left out of the numbers? Because then `range(n)` gives exactly `n` numbers, and the numbers 0 to `n - 1` match the indexes of a string of length `n` (Day 3). Read `range(2, 6)` as "from 2, stop at 6".

The third argument, the **step**, is how much to add each time. A negative step counts down:

```
>>> for n in range(1, 10, 3):
...     print(n)
...
1
4
7

>>> for n in range(10, 0, -3):
...     print(n)
...
10
7
4
1
```

`range(1, 10, 3)` gives 1, 4, 7; the next would be 10, which is not before the stop. `range(10, 0, -3)` counts down from 10 and stops before reaching 0. With a negative step, start must be bigger than stop, or there are no numbers at all.

KEY IDEA

`range(start, stop, step)` counts from `start` by `step` and stops **before** `stop`. `start` defaults to 0 and `step` to 1. To include a number `n` at the end, the stop must be one past it: `range(1, n + 1)`.

WATCH OUT

Off by one at the end. To loop over 1 to 10, it is natural to write `range(1, 10)`. But 10 is the stop, and the stop is never included. Check with `in` and `len()` from Day 3, which work on a range too:

```
>>> 10 in range(1, 10)
False
>>> len(range(1, 10))
9
>>> 10 in range(1, 11)
True
>>> len(range(1, 11))
10
```

`range(1, 10)` has only nine numbers. Use `range(1, 11)`. A mistake where a loop runs one time too many or too few is called an **off-by-one error**, and the stop value of `range` is the usual cause.

PAUSE AND TRY**Question 4** Multiple choice

Which call gives exactly 5, 10, 15, 20?

- A. `range(5, 20, 5)`
- B. `range(5, 21, 5)`
- C. `range(0, 20, 5)`
- D. `range(5, 20)`

5.5 Choosing for or while

You now have two loops, and many jobs can be done with either. Here is the spelling program from 5.3 written with a while loop and an index:

`spell_while.py`

```
1 word = "loop"
2 index = 0
3 while index < len(word):
4     print(word[index])
5     index = index + 1
6 print("Done")
```

```
l
o
o
p
Done
```

The output is the same, but this version needs three extra pieces: a counter that starts at 0 (line 2), a condition that stops at the length (line 3), and an update (line 5). Each is a place to make a mistake: start at 1 and you skip the first letter; write `<=` and `word[4]` stops the program with an `IndexError`; forget line 5 and the loop never ends. The `for` loop does all three for you.

Use	When	Examples
<code>for</code>	You know what to go through: each character, or each number in a range	Spell a word; print 1 to 100
<code>while</code>	You repeat until something happens, and cannot know in advance how many times	Keep asking until <code>quit</code> ; double until over 50

A quick test: if you can say "for each...", use `for`. If you say "until..." or "as long as...", use `while`.

KEY IDEA

Use `for` when the items or the number of passes are known before the loop starts. Use `while` when the loop must run until a condition changes, such as a valid answer or a target reached.

PAUSE AND TRY

Question 5 Short answer

For each job, say whether you would use `for` or `while`, and why: (a) print every character of a city name; (b) keep asking for a PIN until it is 4 characters long; (c) print the even numbers from 2 to 20.

5.6 Counting and Summing

A loop that only prints is useful, but most loops work something out: how many vowels in a sentence, the total of 100 numbers. To get one answer from many passes, the program needs to remember something **between** passes.

An **accumulator** is a variable that collects a result across the passes of a loop. The pattern always has three parts:

1. **Before the loop:** set the accumulator to a starting value (0 for a count or a total).
2. **Inside the loop:** update it on each pass that matters.
3. **After the loop:** use the finished result.

A **counter** is an accumulator that goes up by 1 each time something happens:

WORKED EXAMPLE Counting vowels

count_vowels.py

```

1 text = "Programming is fun"
2 vowels = 0
3 for letter in text.lower():
4     if letter in "aeiou":
5         vowels = vowels + 1
6 print("Vowels:", vowels)

```

Vowels: 5

Line 2 starts the count at 0. Line 3 loops over the lowercase version of the text, so a capital vowel would count too. Line 4 uses the Day 3 `in`: is this character one of the five vowels? Only then does line 5 add 1. The five vowels are the o, a, and i of "Programming", the i of "is", and the u of "fun". Line 6 is outside the loop, so it prints once, at the end.

Updating a variable from its own value is so common that Python has a short form. `count += 1` means `count = count + 1`. This is called **augmented assignment**, and it works with the other arithmetic operators too: `-=`, `*=`, and so on. A **running total** adds each value instead of 1:

sum_to_100.py

```

1 total = 0
2 for number in range(1, 101):
3     total += number
4 print("Sum:", total)

```

Sum: 5050

Pass	number	total after the pass
1	1	1
2	2	3
3	3	6
...	...	...
100	100	5050

`range(1, 101)` gives 1 to 100, and each pass adds the current number to `total`.

Strings can accumulate too. Start with the empty string and add characters with `+=`:

no_spaces.py

```

1 text = "see you soon"
2 result = ""
3 for ch in text:
4     if ch != " ":
5         result += ch
6 print(result)

```

seeyousoon

Strings cannot be changed (Day 3), so `result += ch` builds a **new** string, one character longer, and stores it in `result`. This is how to build a changed copy of a string one character at a time.

KEY IDEA

An accumulator is set **before** the loop, updated **inside** it, and used **after** it. `total += value` means `total = total + value`.

WATCH OUT

Setting the accumulator inside the loop. This should add 1, 2, 3, and 4:

sum_reset.py

```
1 for number in range(1, 5):
2     total = 0
3     total += number
4 print("Sum:", total)
```

Sum: 4

Line 2 is inside the body, so it runs on every pass and throws away the total so far. After the last pass, `total` holds only the last number. Move `total = 0` above the `for` line, so it runs once.

PAUSE AND TRY

Question 6 Tracing

What does this program print? Trace `n`, `count`, and `total`.

count_and_sum.py

```
1 total = 0
2 count = 0
3 for n in range(2, 11, 2):
4     if n % 4 == 0:
5         count += 1
6     total += n
7 print(count, total)
```

5.7 Finding the Largest Value

A weather station records five temperatures and wants the highest. A total adds every value; here you keep only the **best so far**, replacing it whenever a bigger value turns up. That is the **maximum** pattern.

What should "the best so far" be before you have seen any value? Day 2's `None` means exactly "no value yet", and Day 4's `is None` checks for it. This run types 12, 17.5, 9, 17, and 14:

WORKED EXAMPLE The highest temperature

highest_temp.py

```

1 highest = None
2 for day in range(1, 6):
3     temp = float(input(f"Day {day}: "))
4     if highest is None or temp > highest:
5         highest = temp
6 print("Highest:", highest)

```

```

Day 1: 12
Day 2: 17.5
Day 3: 9
Day 4: 17
Day 5: 14
Highest: 17.5

```

Day	temp	Condition on line 4	highest after
1	12.0	highest is None is True	12.0
2	17.5	17.5 > 12.0 is True	17.5
3	9.0	9.0 > 17.5 is False	17.5
4	17.0	17.0 > 17.5 is False	17.5
5	14.0	14.0 > 17.5 is False	17.5

On the first pass, `highest is None` is `True`, so `or` stops there (short-circuit evaluation, Day 4) and the comparison `temp > highest` never runs. That matters, because `>` cannot compare a number with `None`:

```

>>> highest = None
>>> 12.0 > highest
Traceback (most recent call last):
...
TypeError: '>' not supported between instances of 'float' and 'NoneType'

```

From the second pass on, `highest` holds a number and the comparison does the work. The prompt uses the loop variable in an f-string, so each prompt names its day.

To find the **minimum**, the smallest value, use the same pattern with `<`.

KEY IDEA

To find the largest value, start with `highest = None`, and on each pass replace it when `highest is None or value > highest`. Use `<` for the smallest.

WATCH OUT

Starting the maximum at 0. It seems harmless, since temperatures are usually above 0. Here is a winter week, typing -3, -1.5, -7, -4, and -2:

highest_zero.py

```
1 highest = 0
2 for day in range(1, 6):
3     temp = float(input(f"Day {day}: "))
4     if temp > highest:
5         highest = temp
6 print("Highest:", highest)
```

```
Day 1: -3
Day 2: -1.5
Day 3: -7
Day 4: -4
Day 5: -2
Highest: 0
```

No temperature is above 0, so `highest` is never replaced, and the program reports 0, a temperature that never happened. With `highest = None` and the condition from `highest_temp.py`, the same five values give `Highest: -1.5`. Start from "no value yet", not from a number you hope is small enough.

PAUSE AND TRY

Question 7 Debugging

This program should print the lowest of three game scores. The user types 72, 64, and 91, and it prints `Lowest: 0`. Explain the bug and write the fix.

lowest_bug.py

```
1 lowest = 0
2 for game in range(1, 4):
3     score = int(input(f"Game {game} score: "))
4     if score < lowest:
5         lowest = score
6 print("Lowest:", lowest)
```

5.8 break and continue

Some loops should stop early: once you have found the first digit in a product code, there is no point looking at the rest. Others should skip some items, such as spaces. Python has two statements for these jobs. Both go inside the loop body, usually inside an `if`.

`break` ends the loop at once. Python jumps to the first line after the loop, skipping the rest of the body and every remaining pass.

first_digit.py

```
1 code = "AB7C9"
2 for ch in code:
3     if ch in "0123456789":
4         print("First digit:", ch)
5         break
6     print("Not a digit:", ch)
7 print("Search over")
```

```
Not a digit: A
Not a digit: B
First digit: 7
Search over
```

A and B are not in "0123456789", so for them the `if` is skipped and line 6 runs. At 7, the `if` runs, prints, and `break` ends the loop. Line 6 does not run for 7, and c and 9 are never looked at.

`break` gives the "keep asking" loop from 5.2 a common shorter shape: `while True:` is a loop whose condition never becomes `False`, so the only way out is a `break`. This run types `hello`, then `quit`:

until_quit.py

```
1 while True:
2     command = input("Command: ").strip().lower()
3     if command == "quit":
4         break
5     print("You typed", command)
6 print("Goodbye!")
```

```
Command: hello
You typed hello
Command: quit
Goodbye!
```

There is no starting value to set, because the condition no longer uses `command`. The loop reads first, then decides whether to stop.

`continue` ends only the **current pass**. Python skips the rest of the body and goes straight to the next pass (in a `while` loop, back to the condition check):

skip_spaces.py

```
1 for ch in "a b c":
2     if ch == " ":
3         continue
4     print(ch.upper())
5 print("Done")
```

```
A
B
C
Done
```

For each space, `continue` skips line 4, so spaces are not printed. The loop itself goes on to the next character.

Statement	Ends	Then Python goes to
<code>break</code>	The whole loop	The first line after the loop
<code>continue</code>	Only this pass	The next pass

KEY IDEA

`break` leaves the loop completely; `continue` skips to the next pass. `while True:` with a `break` inside is the usual way to repeat until the user says stop.

WATCH OUT

`continue` **before the update in a while loop**. This is meant to print 0 to 4 but skip 2:

```
1 n = 0
2 while n < 5:
3     if n == 2:
4         continue
5     print(n)
6     n += 1
```

When `n` is 2, `continue` jumps back to the condition before `n += 1` runs. `n` stays 2, the condition stays `True`, and the loop never ends (stop it with Ctrl+C). In a while loop, update the variable **before** any `continue`:

skip_three.py

```
1 n = 0
2 while n < 5:
3     n += 1
4     if n == 3:
5         continue
6     print(n)
```

```
1
2
4
5
```

Now `n` goes up first on every pass, so the pass that skips 3 still moves the loop forward. (Since the update comes first, this version counts 1 to 5.) A for loop does not have this problem, because it moves to the next item by itself.

PAUSE AND TRY

Question 8 Tracing

What does this program print?

no_way.py

```
1 result = ""
2 for ch in "no-way!out":
3     if ch == "-":
4         continue
5     if ch == "!":
6         break
7     result += ch
8 print(result)
```

5.9 The Loop else Clause

After a search loop, you often need to know **why** it ended: did `break` stop it because the thing was found, or did it run out of items? Python lets a loop answer that question itself.

A loop can have an **else clause**, written at the same indentation as `for` or `while`. Its block runs when the loop ends **normally**: a `for` loop ran out of items, or a `while` loop's condition became `False`. It does **not** run when `break` ended the loop. Think of it as "if no break".

WORKED EXAMPLE Does the password have a digit?

digit_check.py

```
1 password = "sunshine"
2 for ch in password:
3     if ch in "0123456789":
4         print("Has a digit.")
5         break
6 else:
7     print("No digits: add one to make it stronger.")
```

```
No digits: add one to make it stronger.
```

No character of "sunshine" is a digit, so `break` never runs. The loop ends after the last character, and the `else` block runs. Now change one letter, so the password is "sun5hine":

digit_check_found.py

```
1 password = "sun5hine"
2 for ch in password:
3     if ch in "0123456789":
4         print("Has a digit.")
5         break
6 else:
7     print("No digits: add one to make it stronger.")
```

```
Has a digit.
```

At 5, the `if` prints and `break` ends the loop, so the `else` block is skipped.

Notice the indentation: this `else` lines up with `for`, not with `if`. It belongs to the loop. The same works on a while loop: its `else` runs when the condition becomes `False`, but not after a `break`.

KEY IDEA

A loop's `else` block runs when the loop ends without `break`. Use it after a search for "not found".

WATCH OUT

Indenting the `else` under the `if`. One indentation level changes which statement the `else` belongs to:

`digit_check_wrong_else.py`

```
1 password = "ab3"
2 for ch in password:
3     if ch in "0123456789":
4         print("Has a digit.")
5         break
6     else:
7         print("No digits.")
```

```
No digits.
No digits.
Has a digit.
```

Here `else` lines up with `if`, so it is the Day 4 `if / else`, and it runs for **every** character that is not a digit. The program says "No digits." twice and then finds one. For "not found after checking them all", the `else` must line up with `for`.

PAUSE AND TRY**Question 9** Multiple choice

What does this program print?

`even_search.py`

```
1 for n in range(3, 10, 3):
2     if n % 2 == 0:
3         print("Even:", n)
4         break
5 else:
6     print("No even numbers")
7 print("Done")
```

- A. Even: 6 then Done
- B. Even: 6 then No even numbers then Done
- C. No even numbers then Done
- D. Done only

5.10 Nested Loops

A theater has rows A and B, with seats 1 to 3 in each. To list every seat, you need to go through the seats **for each row**: a loop inside a loop.

A **nested loop** is a loop inside the body of another loop. The **outer loop** runs once per row; on every one of its passes, the **inner loop** runs **all the way through**.

WORKED EXAMPLE Every seat in the theater

seats.py

```
1 for row in "AB":
2     for seat in range(1, 4):
3         print(f"{row}{seat}")
4 print("All seats listed")
```

```
A1
A2
A3
B1
B2
B3
All seats listed
```

Outer pass	row	Inner loop gives seat	Prints
1	"A"	1, 2, 3	A1, A2, A3
2	"B"	1, 2, 3	B1, B2, B3

The inner loop starts over from 1 on each outer pass. Two rows times three seats is six lines: the inner body runs 2×3 times. Line 4 is outside both loops, so it runs once.

To print a grid, build each row as a string in the inner loop (the string accumulator from 5.6), then print it once the inner loop is done. To line the numbers up in columns, Day 3's format specifiers can take a **width**: `f"{value:3}"` shows a number in at least 3 characters, adding spaces on the left.

dice_sums.py

```
1 for first in range(1, 4):
2     line = ""
3     for second in range(1, 4):
4         line += f"{first + second:3}"
5     print(line)
```

```
2 3 4
3 4 5
4 5 6
```

Each row's value is `first + second`. `line = ""` is inside the outer loop, so each row starts empty, and `print(line)` is at the outer loop's level, so it runs once per row, after the inner loop has added all three numbers.

KEY IDEA

In a nested loop, the inner loop runs completely on every pass of the outer loop, so the inner body runs (outer passes) × (inner passes) times. Indentation decides which loop each line belongs to.

WATCH OUT

Printing at the wrong level. Indent `print(line)` one level too far (shown here with two rows):

`dice_sums_wrong.py`

```
1 for first in range(1, 3):
2     line = ""
3     for second in range(1, 4):
4         line += f"{first + second:3}"
5     print(line)
```

```
2
2 3
2 3 4
3
3 4
3 4 5
```

Now `print` is inside the inner loop, so it prints the half-built row after every number. Ask of each line: "should this happen once per number, or once per row?" and indent it to match.

PAUSE AND TRY**Question 10** Tracing

What does this program print? Watch the inner loop's range: it depends on `outer`.

`nested_total.py`

```
1 total = 0
2 for outer in range(1, 4):
3     for inner in range(outer):
4         total += 1
5     print(outer, total)
```

5.11 `enumerate()`: Index and Value Together

A for loop over a string gives each character but not its position. To print "Letter 1: c", you need both. 5.5 got positions from a while loop and a counter. With a for loop, you can go through `range(len(word))`, the indexes of `word`, and index with `word[index]`:

range_len.py

```
1 word = "code"
2 for index in range(len(word)):
3     print(index, word[index])
```

```
0 c
1 o
2 d
3 e
```

It works, but you have to index the string yourself. Python has a built-in function that hands you both.

`enumerate()` goes through a sequence and gives, on each pass, the index **and** the item together. Give the for loop **two** loop variable names, separated by a comma: the first receives the index, the second the item.

enumerate_word.py

```
1 word = "code"
2 for index, letter in enumerate(word):
3     print(index, letter)
```

```
0 c
1 o
2 d
3 e
```

Same output, and no `len`, no `range`, and no `word[index]`. Day 7 explains how this two-name form works in general.

People usually count from 1, not 0. Pass a second argument, the number to start counting at:

enumerate_start.py

```
1 word = "code"
2 for position, letter in enumerate(word, 1):
3     print(f"Letter {position}: {letter}")
```

```
Letter 1: c
Letter 2: o
Letter 3: d
Letter 4: e
```

`enumerate(word, 1)` only changes the numbers it gives you; the characters are the same, and `word[1]` is still "o".

KEY IDEA

`for index, item in enumerate(sequence):` gives each item together with its index, starting at 0, or at the number you pass as a second argument.

WATCH OUT

Giving only one name. With one loop variable, it receives the index and the item as a pair:

enumerate_one_name.py

```
1 for item in enumerate("hi"):
2     print(item)
```

```
(0, 'h')
(1, 'i')
```

Each `item` is a pair shown in parentheses, which is not usually what you want to print. Give two names, for `index`, `letter` in `enumerate("hi"):`, to get the two parts separately.

PAUSE AND TRY**Question 11** Tracing

What does this program print?

find_a.py

```
1 word = "banana"
2 for i, letter in enumerate(word):
3     if letter == "a":
4         print("a at", i)
```

Putting it together

A teacher types quiz scores one at a time and types `done` to finish. The program skips anything outside 0 to 100, then reports how many scores it counted, their average, and the highest. This run types 72, 105, 88, 91, and `done`:

score_report.py

```

1 count = 0
2 total = 0
3 highest = None
4 while True:
5     entry = input("Score (or done): ").strip().lower()
6     if entry == "done":
7         break
8     score = int(entry)
9     if not 0 <= score <= 100:
10         print("Skipped: scores run from 0 to 100.")
11         continue
12     count += 1
13     total += score
14     if highest is None or score > highest:
15         highest = score
16 if count > 0:
17     print(f"{count} scores, average {total / count:.1f}, highest {highest}")
18 else:
19     print("No scores entered.")

```

```

Score (or done): 72
Score (or done): 105
Skipped: scores run from 0 to 100.
Score (or done): 88
Score (or done): 91
Score (or done): done
3 scores, average 83.7, highest 91

```

Typed	What happens	count	total	highest	Idea
72	In range; highest is None	1	72	72	5.6, 5.7
105	not 0 <= 105 <= 100 is True: continue	1	72	72	5.8
88	88 > 72	2	160	88	5.7
91	91 > 88	3	251	91	5.7
done	break	3	251	91	5.8

- `while True:` with `break` (5.8) fits because the program cannot know how many scores will be typed.
- `continue` skips the 105 before any accumulator changes, so it is neither counted nor added.
- The input is checked for `done` **before** `int()`, because `int("done")` would stop the program with a `ValueError`.
- After the loop, `251 / 3` is 83.666..., which `:.1f` shows as 83.7 (Day 3). The `if count > 0:` guard means that typing `done` straight away prints a message instead of dividing by zero.

Vocabulary

Term	Definition	Example
loop	Code that runs a block repeatedly	<code>while</code> or <code>for</code>

while loop	Repeats its body as long as its condition is true, checking before each pass	<code>while count > 0:</code>
loop body	The indented lines that repeat	<code>print(count)</code>
iteration	One run of the loop body; a pass	the 3rd pass
infinite loop	A loop whose condition never becomes false	stop it with Ctrl+C
for loop	Runs its body once for each item in a sequence	<code>for letter in word:</code>
loop variable	The name that holds the current item on each pass	<code>letter</code>
range()	Makes a sequence of whole numbers, stopping before stop	<code>range(1, 11)</code>
step	The amount <code>range</code> adds each time; negative counts down	<code>range(10, 0, -1)</code>
off-by-one error	A loop that runs one time too many or too few	<code>range(1, 10)</code> for 1 to 10
accumulator	A variable that collects a result across passes	<code>total</code>
counter	An accumulator that goes up by 1 each time something happens	<code>count += 1</code>
running total	An accumulator that adds each value	<code>total += price</code>
augmented assignment	Updates a variable from its own value	<code>total += 5</code>
maximum / minimum	The largest / smallest value seen	<code>highest</code>
break	Ends the whole loop at once	<code>if found: break</code>
continue	Skips the rest of this pass	<code>if ch == " ": continue</code>
else clause (loop)	Runs when the loop ends without <code>break</code>	<code>else: print("Not found")</code>
nested loop	A loop inside another loop's body	rows and seats
outer / inner loop	The loop that contains / the loop inside	row / seat
width	A format specifier that pads a value to a minimum number of characters	<code>f"{n:3}"</code>
enumerate()	Gives each item of a sequence together with its index	<code>enumerate(word, 1)</code>

Summary

- A **while loop** repeats while its condition is true and checks before every pass, so its body can run zero times. Something in the body must move the condition toward `False`, or you get an **infinite loop**.
- A **for loop** runs once per item: each character of a string, or each number from `range(start, stop, step)`, which stops **before** `stop`. Use `for` for "each...", while for "until...".
- An **accumulator** is set before the loop, updated inside it, and used after it: counters, running totals, built-up strings, and the largest or smallest value (start that one at `None`).
- **break** ends a loop, **continue** skips to the next pass, and a loop's **else** runs only when no `break` happened.
- In a **nested loop**, the inner loop runs completely on every outer pass. `enumerate()` gives each item with its index.

Practice

Do these in your `day05` folder. Type each program yourself, run it, and only then open the answer.

Question 12 Short answer

FizzBuzz. Print the numbers from 1 to 100, but print `Fizz` instead of a number divisible by 3, `Buzz` instead of a number divisible by 5, and `FizzBuzz` instead of a number divisible by both.

Question 13 Short answer

Guessing game. Store a secret number from 1 to 10 in a variable. Give the player 3 guesses. After each wrong guess, say whether it was too low or too high. If they guess it, congratulate them and stop asking; if they use all 3 guesses, tell them the number. (Day 13 shows how to pick a random secret; today, write it into the code.)

Question 14 Short answer

Multiplication table. Use nested loops to print the multiplication table for 1 to 5: five rows of five products, lined up in columns.

Review questions

Question 15 Tracing

What does this program print? Trace `n` and `total` through every pass.

`skip_multiples.py`

```
1 n = 0
2 total = 0
3 while n < 6:
4     n += 1
5     if n % 3 == 0:
6         continue
7     total += n
8 print(n, total)
```

Question 16 Debugging

This program should add the numbers 1 to 10, which gives 55. It prints `Sum of 1 to 10: 45`. Explain the bug and write the fix.

`sum_one_to_ten.py`

```
1 total = 0
2 for number in range(1, 10):
3     total += number
4 print("Sum of 1 to 10:", total)
```

Question 17 Short answer

A program asks for a number of tickets and must accept only 1 to 6. A classmate writes `for attempt in range(3):` around the question. Explain why a while loop fits this job better, and describe what its condition and body would be.

Question 18 Tracing

What does this program print?

first_vowel.py

```
1 word = "rhythm"
2 for index, letter in enumerate(word):
3     if letter in "aeiou":
4         print(f"First vowel at {index}")
5         break
6 else:
7     print("No vowels")
```

Question 19 Multiple choice

How many lines does this program print?

pairs.py

```
1 for a in range(2):
2     for b in range(3):
3         print(a, b)
4     print("--")
```

- A. 5
- B. 6
- C. 8
- D. 12

Go further

The official Python Tutorial covers today's loops in [More Control Flow Tools: for Statements, The range\(\) Function, break and continue Statements, and else Clauses on Loops](#). Its for examples loop over lists, which you meet tomorrow on Day 6.

Reflection

Check each statement. If the answer is "not yet," reread the linked subsection.

- ☐ I can write a while loop and point to the line that makes it stop. Reread: [5.1 while Loops](#)
- ☐ I can write a loop that keeps asking until the user types what is needed. Reread: [5.2 Asking Until the Answer Is Right](#)
- ☐ I can loop over the characters of a string with for. Reread: [5.3 for Loops Over Strings](#)
- ☐ I can predict the numbers any range() call gives, including the stop. Reread: [5.4 Counting with range\(\)](#)
- ☐ I can choose between for and while and explain why. Reread: [5.5 Choosing for or while](#)
- ☐ I can count and sum with an accumulator set before the loop. Reread: [5.6 Counting and Summing](#)
- ☐ I can find the largest or smallest value starting from None. Reread: [5.7 Finding the Largest Value](#)
- ☐ I can use break and continue and explain the difference. Reread: [5.8 break and continue](#)
- ☐ I can use a loop else for "not found". Reread: [5.9 The Loop else Clause](#)

- ☐ I can trace a nested loop and count how many times its body runs. Reread: 5.10 Nested Loops
- ☐ I can loop over index and item together with `enumerate()`. Reread: 5.11 `enumerate()`: Index and Value Together

Check Your Reasoning

Answers and explanations for every question, numbered to match.

Question 1

3 80

Check	savings	weeks	savings < 50
1st	10	0	True
2nd	20	1	True
3rd	40	2	True
4th	80	3	False

The loop does not stop the moment `savings` passes 50. It stops at the next **check**: the body that turned 40 into 80 runs to the end, and only then does `80 < 50` fail. If you answered 2 40 or 3 50, remember that the body always finishes before the condition is checked again.

Reread *5.1 while Loops* if your output differed.

Question 2

B. The terminal shows:

```
Password: Open Sesame
Wrong, try again.
Password: open sesame
Welcome!
```

`"Open Sesame" != "open sesame"` is `True`, because comparisons are case-sensitive, so the body runs once and asks again. The second answer matches, the condition is `False`, and the loop ends.

- **A** is wrong: the first answer has capitals, so it does not match.
- **C** is wrong: the second answer matches, so the body does not run again.
- **D** is wrong: line 4 reads a new password on every pass, so the loop can end.

Reread *5.2 Asking Until the Answer Is Right* if you chose differently.

Question 3

```
[a]
[ ]
[b]
[!]
```

`"a b!"` has four characters: `a`, a space, `b`, and `!`. A space is a character like any other, so it gets its own pass and its own line. If you printed three lines, count the characters with `len("a b!")`, which is 4.

Reread *5.3 for Loops Over Strings* if your output differed.

Question 4

B. Start at 5 and add 5: 5, 10, 15, 20. The next, 25, is not before 21, so it stops. Any stop from 21 to 25 would also work.

- **A** is wrong: 20 is the stop, and the stop is never included, so it gives 5, 10, 15.
- **C** is wrong: it starts at 0, giving 0, 5, 10, 15.
- **D** is wrong: with no step, it counts by 1: 5, 6, 7, ... 19.

Reread 5.4 *Counting with range()* if you chose differently.

Question 5

Model answer:

- (a) `for`: you go through each character of a known string, `for ch in city:`.
- (b) `while`: you cannot know how many tries the user needs; repeat until `len(pin) == 4`.
- (c) `for`: the numbers are known in advance, `for n in range(2, 21, 2):`.

A correct answer links `for` to "each item of something known" and `while` to "until a condition changes". For (c), check the stop: `range(2, 20, 2)` would leave out 20.

Reread 5.5 *Choosing for or while* if your reasoning differed.

Question 6

2 30

n	n % 4 == 0	count	total
2	False	0	2
4	True	1	6
6	False	1	12
8	True	2	20
10	False	2	30

`range(2, 11, 2)` gives 2, 4, 6, 8, 10; 11 is the stop, and 10 is the last even number before it. `count` only goes up for multiples of 4, but `total += n` is not inside the `if`, so it adds every `n`.

Reread 5.6 *Counting and Summing* if your output differed.

Question 7

`lowest` starts at 0, and no score is below 0, so `score < lowest` is never `True` and 0 is never replaced. Start from "no value yet" and let the first score in:

`lowest_fixed.py`

```
1 lowest = None
2 for game in range(1, 4):
3     score = int(input(f"Game {game} score: "))
4     if lowest is None or score < lowest:
5         lowest = score
6 print("Lowest:", lowest)
```

```
Game 1 score: 72
Game 2 score: 64
Game 3 score: 91
Lowest: 64
```

The first pass stores 72 because `lowest` is `None`. 64 is smaller, so it replaces 72; 91 is not.

Reread 5.7 *Finding the Largest Value* if you missed it.

Question 8

```
noway
```

ch	What happens	result
n	Added	"n"
o	Added	"no"
-	continue: skipped	"no"
w	Added	"now"
a	Added	"nowa"
y	Added	"noway"
!	break: loop ends	"noway"

The `-` is skipped but the loop goes on; the `!` ends the loop, so `out` is never reached. The check for `!` comes before `result += ch`, so the `!` itself is not added either.

Reread 5.8 *break and continue* if your output differed.

Question 9

A.

```
Even: 6
Done
```

`range(3, 10, 3)` gives 3, 6, 9. 3 is odd, so the loop goes on. 6 is even, so it prints and `break` ends the loop. Because the loop ended with `break`, the `else` is skipped. `Done` is after the loop, so it always prints.

- **B** is wrong: the `else` block does not run after a `break`.
- **C** is wrong: 6 is in the range and is even.
- **D** is wrong: 6 is found and printed before the loop ends.

Reread 5.9 *The Loop else Clause* if you chose differently.

Question 10

```
1 1
2 3
3 6
```

outer	Inner loop range(outer)	Inner passes	total after
1	0	1	1

2	0, 1	2	3
3	0, 1, 2	3	6

`print` is at the outer loop's level, so it runs once per outer pass, after that pass's inner loop. `total` is set before both loops, so it keeps growing.

Reread 5.10 *Nested Loops* if your output differed.

Question 11

```
a at 1
a at 3
a at 5
```

`enumerate` counts from 0: `b` is 0, `a` is 1, `n` is 2, and so on. The `if` prints only for the three `as`, at the odd positions. There is no `break`, so the loop checks every letter.

Reread 5.11 *enumerate(): Index and Value Together* if your output differed.

Question 12

One possible solution:

fizzbuzz.py

```
1 for number in range(1, 101):
2     if number % 15 == 0:
3         print("FizzBuzz")
4     elif number % 3 == 0:
5         print("Fizz")
6     elif number % 5 == 0:
7         print("Buzz")
8     else:
9         print(number)
```

It prints 100 lines. The first 15 show every case:

```
1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
11
Fizz
13
14
FizzBuzz
```

The pattern repeats every 15 numbers, ending at 100, which prints `Buzz`. A number divisible by both 3 and 5 is divisible by 15, and that rung comes **first**: if `% 3` came first, 15 would print `Fizz` and

the chain would stop there (the rung-order mistake from Day 4). `range(1, 101)` includes 100. Run it and check that your 100th line is `Buzz`.

Reread 5.4 *Counting with range()* if your result differed.

Question 13

One possible solution:

guessing_game.py

```
1 secret = 7
2 print("I am thinking of a number from 1 to 10.")
3 for attempt in range(1, 4):
4     guess = int(input(f"Guess {attempt} of 3: "))
5     if guess == secret:
6         print(f"Yes! You got it in {attempt}.")
7         break
8     elif guess < secret:
9         print("Too low.")
10    else:
11        print("Too high.")
12 else:
13    print(f"Out of guesses. It was {secret}.")
```

Two runs. The first types 5, 8, then 7 and wins on the last guess:

```
I am thinking of a number from 1 to 10.
Guess 1 of 3: 5
Too low.
Guess 2 of 3: 8
Too high.
Guess 3 of 3: 7
Yes! You got it in 3.
```

The second types 2, 9, then 4 and runs out:

```
I am thinking of a number from 1 to 10.
Guess 1 of 3: 2
Too low.
Guess 2 of 3: 9
Too high.
Guess 3 of 3: 4
Too low.
Out of guesses. It was 7.
```

`for attempt in range(1, 4)` gives at most three guesses and numbers the prompts. A correct guess prints and `breaks`, so the loop's `else` is skipped, even on the third guess. Three wrong guesses let the loop finish normally, so the `else` reports the secret. A `while` loop with an attempts counter works too.

Reread 5.9 *The Loop else Clause* and 5.8 *break and continue* if your result differed.

Question 14

One possible solution:

times_table.py

```

1 for row in range(1, 6):
2     line = ""
3     for column in range(1, 6):
4         line += f"{row * column:4}"
5     print(line)

```

```

1  2  3  4  5
2  4  6  8 10
3  6  9 12 15
4  8 12 16 20
5 10 15 20 25

```

The outer loop picks the row number; for each one, the inner loop goes through the five columns and adds `row * column` to the row's string. Width 4 leaves room for 25 plus a space between columns. `line = ""` is inside the outer loop so each row starts empty, and `print(line)` runs once per row. Change both 6s to 11 for the full table to 10.

Reread 5.10 *Nested Loops* if your result differed.

Question 15

6 12

n after line 4	n % 3 == 0	total
1	False	1
2	False	3
3	True: skip	3
4	False	7
5	False	12
6	True: skip	12

The update comes before `continue`, so the loop always moves forward. When `n` is 6, the pass is skipped, the condition `6 < 6` is `False`, and the loop ends. The program adds every number from 1 to 6 except the multiples of 3.

Reread 5.8 *break and continue* and 5.6 *Counting and Summing* if your output differed.

Question 16

`range(1, 10)` stops **before** 10, so the loop adds 1 to 9, which is 45. This is an off-by-one error. The stop must be one past the last number you want:

sum_one_to_ten_fixed.py

```

1 total = 0
2 for number in range(1, 11):
3     total += number
4 print("Sum of 1 to 10:", total)

```

Sum of 1 to 10: 55

Reread the Watch Out in 5.4 *Counting with range()* if you missed it.

Question 17

Model answer: nobody can know in advance how many wrong answers the user will type, so the loop should repeat **until** the answer is valid, which is a while loop's job. `range(3)` would give up after three tries, even though the program still has no valid number. Read the first answer before the loop; the condition is `while not 1 <= tickets <= 6:`; the body prints a message and reads a new answer, which is what lets the loop end.

A correct answer says that the number of passes is not known in advance, gives a condition that is `True` while the input is invalid, and reads new input inside the body.

Reread 5.5 *Choosing for or while* and 5.2 *Asking Until the Answer Is Right* if your reasoning differed.

Question 18

No vowels

"rhythm" has no a, e, i, o, or u. (y sometimes sounds like a vowel, but it is not in "aeiou", and `in` checks only the characters you list.) The `if` is never `True`, `break` never runs, the loop ends after the last letter, and the loop's `else` runs. The `index` from `enumerate` is never printed.

Reread 5.9 *The Loop else Clause* and 5.11 *enumerate(): Index and Value Together* if your output differed.

Question 19

C.

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

The outer loop runs twice. Each time, the inner loop prints 3 lines, and then `print("--")`, at the outer loop's level, prints 1 more: $2 \times (3 + 1) = 8$.

- **A** counts the passes of the two loops ($2 + 3$) instead of the lines printed.
- **B** counts only the inner prints, 2×3 , and forgets the `--` lines.
- **D** treats `print("--")` as if it were inside the inner loop, running 3 times per row instead of once: $2 \times (3 + 3) = 12$.

Reread 5.10 *Nested Loops* if you chose differently.