

Day 3 Strings

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

By the end you will be able to

- Write strings with single, double, or triple quotes and use escape characters for quotes and new lines.
- Pick out any character or part of a string with indexing and slicing.
- Clean up and search text with string methods, and build formatted output with f-strings.
- Read what the user types with `input()` and convert it to a number when you need one.

What to remember from earlier

- Text in quotes, like "Hello", is a string. ([Day 1: Setup and your first program, 1.4 Printing Text and Numbers](#))
- A variable is a name that refers to a value, so you can store something now and use it later. ([Day 2: Variables, numbers and basic types, 2.1 Variables: Giving a Value a Name](#))
- `type()` tells you which type a value is, such as `int`, `float`, or `str`. ([Day 2: Variables, numbers and basic types, 2.3 Integers and Floats](#))
- A `bool` is either `True` or `False`. ([Day 2: Variables, numbers and basic types, 2.6 True, False, and None](#))
- `int()`, `float()`, and `str()` convert a value to another type. ([Day 2: Variables, numbers and basic types, 2.7 Converting Between Types](#))
- A string of digits like "15" is text, not a number, so subtracting from it is a `TypeError`. ([Day 2: Variables, numbers and basic types, 2.7 Converting Between Types](#))
- `round(x, 2)` rounds a float for display, but it cannot show trailing zeros such as 2.50. ([Day 2: Variables, numbers and basic types, 2.9 Floats Are Close, Not Exact](#))

Make a `day03` folder for today's files, open it in your editor, and run each script from its terminal, as on Days 1 and 2.

3.1 Quotes and Escape Characters

Since Day 1 you have written text like "Hello". Python calls a piece of text a **string**, and its type is `str`. A string is a sequence of **characters**: letters, digits, spaces, punctuation, each one a single symbol. Today you learn to take strings apart, change them, and build them into neat output. First, how do you put a quote mark **inside** a string, when quote marks are what start and end it?

Python lets you write a string with either kind of quote, "double" or 'single'. Both make exactly the same string. Having two kinds means you can use one kind to hold the other:

quotes.py

```
1 print("Hello, Ana")
2 print('Hello, Ana')
3 print("It's raining")
4 print('She said "hi"')
```

```
Hello, Ana
Hello, Ana
It's raining
She said "hi"
```

Lines 1 and 2 print the same thing: the quotes are not part of the text. In line 3 the apostrophe is safe because the string is in double quotes, and in line 4 the double quotes are safe inside single quotes. This course uses double quotes unless the text holds a " .

WATCH OUT

An apostrophe inside single quotes. Python ends a string at the first quote that matches the opening one:

apostrophe_in_single.py

```
1 print('It's raining')
```

```
File ".../day03/apostrophe_in_single.py", line 1
    print('It's raining')
          ^
```

SyntaxError: unterminated string literal (detected at line 1)

Python reads 'It' as a whole string, then sees s raining, then a new ' that is never closed. It is the same "unterminated string literal" error you met on Day 1. Use double quotes: `print("It's raining")`.

Sometimes you need both kinds of quote, or a character you cannot type into a string at all, such as a line break. For these, Python uses an **escape character**: a backslash `\` followed by a letter or symbol, which together stand for one special character. ("Escape" because the backslash escapes the normal meaning of the next character.)

Escape	Means	Example	Prints
<code>\n</code>	New line	<code>"one\ntwo"</code>	one and two
<code>\t</code>	Tab (jumps to the next column)	<code>"Name:\tAna"</code>	Name: then a gap
<code>\"</code>	A double quote	<code>"say \"hi\""</code>	say "hi"
<code>\'</code>	A single quote	<code>'It\'s'</code>	It's
<code>\\</code>	One backslash	<code>"C:\\new_folder"</code>	C:\new_folder

WORKED EXAMPLE Escapes in action

escapes.py

```
1 print("Line one\nLine two")
2 print("Name:\tAna")
3 print("She said \"hi\"")
4 print('It\'s raining')
5 print("C:\\new_folder")
```

```
Line one
Line two
Name: Ana
She said "hi"
It's raining
C:\new_folder
```

Line 1 prints on two lines: `\n` is a line break, not two characters. Line 3's `\"` keeps the quotes from ending the string. Line 5 needs `\\`: a single `\` before `n` would make `\n`, a new line, and the output would break after `c:`.

For text that spans several lines, such as a poem or a menu, writing `\n` everywhere is hard to read. A **triple-quoted string** starts and ends with three quote marks, `"""` (or `'''`), and can run over several lines. Every line break inside it becomes part of the string:

poem.py

```
1 poem = """Roses are red,
2 violets are blue,
3 this string has three lines."""
4 print(poem)
```

```
Roses are red,
violets are blue,
this string has three lines.
```

Finally, the REPL shows a string differently from `print`. The REPL shows a string as you would type it into code: in quotes (single unless the text holds an apostrophe), with escapes written out. `print` shows the characters themselves:

```
>>> "It's raining"
"It's raining"
>>> 'Hello'
'Hello'
>>> "Line one\nLine two"
'Line one\nLine two'
>>> print("Line one\nLine two")
Line one
Line two
```

KEY IDEA

A string is text in matching single, double, or triple quotes. A backslash starts an escape character: `\n` is a new line, `\t` a tab, `\"` and `\'` quote marks, and `\\` one backslash.

PAUSE AND TRY

Question 1 Tracing

What does this program print?

predict_escapes.py

```
1 print("Ready,\nset,\ngo!")
2 print("Say \"cheese\"")
3 print('Don\'t type \\n')
```

Question 2 Multiple choice

Which line prints Ana's "cat" without an error?

- A. `print('Ana's "cat"')`
- B. `print("Ana's "cat"")`
- C. `print("Ana's \"cat\"")`
- D. `print("Ana\'s "cat"")`

3.2 Length, Concatenation, and in

Once text is in a string, the first questions are often simple: how long is it, can I stick two strings together, and does it contain some word? Python answers each with one short piece of syntax.

- The built-in function `len()` gives the **length** of a string: how many characters it has. Spaces and punctuation count, and the **empty string** `""` has length 0.
- **Concatenation** means joining strings end to end, and `+` does it for strings. Python adds nothing between them; a space appears only if you include one.
- **Repetition**: `*` with a string and an int repeats the string that many times.

```
>>> len("Python")
6
>>> len("New York")
8
>>> len("")
0
>>> "snow" + "ball"
'snowball'
>>> "snow" + " " + "ball"
'snow ball'
>>> "ha" * 3
'hahaha'
>>> "-" * 10
'-----'
```

Notice that `+` and `*` mean something different for strings than for numbers. The type of the values decides what an operator does, as Day 2 showed with `/`.

WORKED EXAMPLE A name badge

name_badge.py

```

1 first_name = "Grace"
2 last_name = "Hopper"
3 full_name = first_name + " " + last_name
4 border = "*" * (len(full_name) + 4)
5 print(border)
6 print("* " + full_name + " *")
7 print(border)
8 print("Characters:", len(full_name))

```

```

*****
* Grace Hopper *
*****
Characters: 12

```

Line	Worked out	Value
3	"Grace" + " " + "Hopper"	"Grace Hopper"
4	len is 12, plus 4 is 16	"*" repeated 16 times
6	"* " + full_name + " *"	"* Grace Hopper *"

The border is 4 characters longer than the name to make room for "*" and "*". Change `first_name` to "Ada" and the border shrinks to fit, because it is worked out from `len(full_name)`.

Two more operators ask yes-or-no questions about strings, so they give a bool (Day 2):

- `in` asks whether one string appears inside another: "York" `in` "New York" is `True`. The part you look for is a **substring**, a run of characters inside a string.
- `==` asks whether two values are equal: "stop" `==` "stop" is `True`. It is two equals signs; a single `=` assigns. Day 4 covers comparisons fully; today you only need `==` to check whether two strings match.

```

>>> "York" in "New York"
True
>>> "york" in "New York"
False
>>> "ark" in "parking"
True
>>> "stop" == "stop"
True
>>> "stop" == "Stop"
False

```

Both are **case-sensitive**: "y" and "Y" are different characters, so "york" is not in "New York".

KEY IDEA

`len(text)` counts characters, `+` joins strings, and `text * n` repeats one. part in text and `a == b` give `True` or `False`, and both care about capital letters.

WATCH OUT

Joining a string and a number with `+`. `+` joins strings or adds numbers, but it will not guess which you mean when you mix them:

age_label.py

```
1 age = 15
2 print("Age: " + age)
```

```
Traceback (most recent call last):
  File "../day03/age_label.py", line 2, in <module>
    print("Age: " + age)
                ~~~~~^~~~~~
TypeError: can only concatenate str (not "int") to str
```

Read it as "a string can only be joined to another string, not to an int." Convert the number first, `"Age: " + str(age)`, or let `print` separate its arguments with commas, `print("Age:", age)`. Section 3.8 shows a third, neater way.

PAUSE AND TRY**Question 3** Tracing

Predict exactly what the REPL prints for each line.

```
>>> len("hi there")
>>> "ab" * 2 + "c"
>>> "3" + "4"
>>> "car" in "Scar"
```

3.3 Indexing: One Character at a Time

To print someone's initials, you need the **first** character of a name, not the whole string. Python numbers every character's position so you can ask for one.

The position of a character is its **index**. Indexes start at **0**, not 1: the first character is at index 0, the second at index 1, and so on. Think of the index as "how many characters come before this one": the first character has 0 before it. To get the character at an index, write the index in square brackets after the string. This is called **indexing**.

Python also counts from the end with **negative indexes**: `-1` is the last character, `-2` the one before it. Here are both numberings for `"Python"`:

Character	Index	Negative index
P	0	-6
y	1	-5
t	2	-4
h	3	-3

o	4	-2
n	5	-1

```
>>> word = "Python"
>>> word[0]
'P'
>>> word[3]
'h'
>>> word[5]
'n'
>>> word[-1]
'n'
>>> word[-6]
'P'
>>> word[len(word) - 1]
'n'
```

A string of 6 characters has indexes 0 to 5, so the last index is always `len(word) - 1`. `word[-1]` is a shorter way to say the same thing, and it works for a string of any length. Indexing gives a new string of length 1; Python has no separate type for a single character.

Now vary the index past the end:

index_too_far.py

```
1 word = "Python"
2 print(word[6])
```

```
Traceback (most recent call last):
  File "../day03/index_too_far.py", line 2, in <module>
    print(word[6])
          ~~~~^^^
IndexError: string index out of range
```

An **IndexError** means you asked for a position that does not exist. "Python" has 6 characters, but because counting starts at 0, there is no index 6.

KEY IDEA

`text[i]` is the character at index `i`. Indexes run from 0 to `len(text) - 1`, and negative indexes count from the end, with `-1` the last character. An index past the end is an **IndexError**.

WATCH OUT

Counting from 1. It is natural to call the first letter "letter 1", but `word[1]` is the **second** character, 'y'. Subtract 1 from the everyday position to get the index. That is also why `word[len(word)]` is always an **IndexError**.

PAUSE AND TRY**Question 4** Tracing

Predict exactly what the REPL prints for each line. Draw the index table for "Lisbon" first.

```
>>> city = "Lisbon"
>>> city[1]
>>> city[-2]
>>> city[len(city) - 3]
```

3.4 Slicing

A date stored as "2026-10-09" holds a year, a month, and a day. Indexing gets one character at a time; to get the four-character year you need a **range** of positions.

A **slice** is a part of a string taken from a range of indexes, written `text[start:stop]`. It starts **at** start and stops **before** stop: the character at stop is not included. So `word[0:2]` is index 0 and index 1, two characters. A handy check: a slice has `stop - start` characters.

Either number can be left out. Leave out start and the slice begins at the start of the string; leave out stop and it runs to the end.

```
>>> word = "Python"
>>> word[0:2]
'Py'
>>> word[2:5]
'tho'
>>> word[:3]
'Pyt'
>>> word[3:]
'hon'
>>> word[-3:]
'hon'
>>> word[:]
'Python'
>>> word[4:100]
'on'
```

- `word[2:5]` takes indexes 2, 3, and 4: 'tho'.
- `word[:3]` is the first three characters, and `word[3:]` is everything from index 3 on. Together they rebuild "Python".
- `word[-3:]` starts three from the end: the last three characters.
- `word[4:100]` goes past the end, but a slice never causes an `IndexError`: it stops at the end of the string. Only indexing a single position does.

WORKED EXAMPLE Taking a date apart

date_parts.py

```

1 date = "2026-10-09"
2 year = date[0:4]
3 month = date[5:7]
4 day = date[8:]
5 print("Year:", year)
6 print("Month:", month)
7 print("Day:", day)

```

```

Year: 2026
Month: 10
Day: 09

```

Part	Indexes	Slice
2026	0 to 3	date[0:4]
-	4	skipped
10	5 to 6	date[5:7]
-	7	skipped
09	8 to 9	date[8:]

Each stop is one past the last index wanted, and `date[8:]` runs to the end. Each part is still a string: day is "09", not the number 9.

A slice can take a third number, the **step**: `text[start:stop:step]` takes every `step`-th character. A step of 2 takes every second character. A **negative step** walks backwards, so `text[::-1]`, with start and stop left out, is the whole string reversed.

```

>>> letters = "abcdefgh"
>>> letters[0:6:2]
'ace'
>>> letters[:2]
'ac'
>>> letters[1:2]
'b'
>>> letters[::-1]
'hgfedcba'
>>> "stressed"[::-1]
'desserts'

```

`letters[0:6:2]` takes indexes 0, 2, and 4. `letters[1:2]` starts at 1 and takes every second character to the end. `[::-1]` is the usual way to reverse a string in Python, and with `==` from 3.2 it tells you whether a word reads the same both ways.

KEY IDEA

`text[start:stop]` takes characters from `start` up to, but not including, `stop`; a missing `start` or `stop` means the beginning or the end. `text[start:stop:step]` takes every `step`-th character, and `text[::-1]` reverses the string.

WATCH OUT

Expecting stop to be included. To get "Pyth" from "Python", students often write `word[0:3]`, which gives only 'Pyt'. The stop index is excluded, so you need `word[0:4]`. Check with `stop - start: 4 - 0 = 4` characters.

PAUSE AND TRY**Question 5** Tracing

Predict exactly what the REPL prints for each line.

```
>>> item = "notebook"
>>> item[0:4]
>>> item[4:]
>>> item[1:6:2]
>>> item[::-1]
```

Question 6 Multiple choice

`code = "AB-2026-X"`. Which expression gives "2026"?

- A. `code[3:6]`
- B. `code[3:7]`
- C. `code[2:7]`
- D. `code[-6:-1]`

3.5 Strings Are Immutable

Suppose `word` holds "cat" and you want "bat". Indexing reads a character, so it seems you could assign to one:

`change_letter.py`

```
1 word = "cat"
2 word[0] = "b"
3 print(word)
```

Traceback (most recent call last):

```
File "../day03/change_letter.py", line 2, in <module>
    word[0] = "b"
    ~~~~^
```

TypeError: 'str' object does not support item assignment

Python refuses. A string is **immutable**: once made, its characters can never change. ("Item assignment" means assigning to one position.) You can still get `"bat"`. Build a **new** string from the pieces you want, and point the name at it:

change_letter_fixed.py

```
1 word = "cat"
2 word = "b" + word[1:]
3 print(word)
```

bat

Line 2 makes a new string, `"b" + "at"`, and rebinds `word` to it (rebinding is from Day 2). The old `"cat"` is not changed; the name simply refers to a different string now.

Why make strings unchangeable? Because a value that cannot change is safe to share. If two names refer to the same string, nothing done through one name can surprise the other. It also means every string operation you learn today, slices and the methods in 3.6, leaves the original alone and gives you a **new** string.

KEY IDEA

Strings are immutable: you cannot change a character in place. To "change" a string, build a new one and assign it to the name.

PAUSE AND TRY

Question 7 Debugging

This program should print `Hello world`, with a capital H. It stops with an error. Quote the last line of the error, explain it, and write the fix.

capitalize_title.py

```
1 title = "hello world"
2 title[0] = title[0].upper()
3 print(title)
```

3.6 String Methods

Text that people type is messy: extra spaces, random capitals, the wrong punctuation. Cleaning it with slices alone would be slow. Python gives every string a set of ready-made tools for these jobs.

A **method** is a function that belongs to a value. You call it by writing the value, a dot, the method's name, and parentheses: `name.upper()`. The value before the dot is what the method works on, and any arguments go inside the parentheses, as with `print` and `len`. Because strings are immutable (3.5), a method never changes the string: it **returns** (gives back) a new value.

Method	Returns	Example	Result
--------	---------	---------	--------

<code>upper(), lower()</code>	A copy in capitals / small letters	<code>"Hi".upper()</code>	<code>'HI'</code>
<code>strip()</code>	A copy without spaces, tabs, or new lines at the ends	<code>" hi ".strip()</code>	<code>'hi'</code>
<code>replace(old, new)</code>	A copy with every old replaced by new	<code>"a-b-c".replace("-", "")</code>	<code>'abc'</code>
<code>find(part)</code>	The index where part first starts, or -1	<code>"hello".find("l")</code>	2
<code>startswith(part)</code>	True if the string begins with part	<code>"hello".startswith("he")</code>	True
<code>count(part)</code>	How many times part appears	<code>"banana".count("a")</code>	3

Spaces, tabs, and new lines are together called **whitespace**; `strip` removes it from both ends but not from the middle.

```
>>> name = " Ada Lovelace "
>>> name.strip()
'Ada Lovelace'
>>> name.upper()
' ADA LOVELACE '
>>> name.strip().lower()
'ada lovelace'
>>> name
' Ada Lovelace '
```

The last line shows `name` itself is unchanged, spaces and all. `name.strip().lower()` is a **method chain**: Python runs it left to right, so `strip()` returns `'Ada Lovelace'` and `lower()` works on that new string.

The other four methods search and replace:

```
>>> sentence = "the cat sat on the mat"
>>> sentence.replace("cat", "dog")
'the dog sat on the mat'
>>> sentence.count("at")
3
>>> sentence.find("sat")
8
>>> sentence.find("dog")
-1
>>> sentence.startswith("the")
True
>>> sentence.startswith("cat")
False
```

- `replace` changed every `"cat"` (there is one).
- `count("at")` finds `at` in `"cat"`, `"sat"`, and `"mat"`: 3.
- `find("sat")` is 8: `"the cat "` is 8 characters before it. `find` gives `-1` when the part is missing, because `-1` can never be a real starting position.
- `startswith` gives a bool, like `in`, but checks only the beginning.

WORKED EXAMPLE Cleaning an email address

clean_email.py

```

1 raw_email = " Maya.Lopez@Example.com "
2 email = raw_email.strip().lower()
3 at_position = email.find("@")
4 print("Cleaned:", email)
5 print("Name part:", email[:at_position])
6 print("Domain:", email[at_position + 1:])
7 print("Dots:", email.count("."))

```

```

Cleaned: maya.lopez@example.com
Name part: maya.lopez
Domain: example.com
Dots: 2

```

Line	Worked out	Value
2	strip(), then lower()	"maya.lopez@example.com"
3	"maya.lopez" is 10 characters	10
5	email[:10]	"maya.lopez"
6	email[11:]	"example.com"

find gives the @'s index, and the slices use it: everything before it, and everything after it (+ 1 skips the @ itself). The program works for any address, because it never uses a fixed number.

WATCH OUT

Calling a method and throwing away the result. A method returns a new string; it does not change the old one:

forgot_result.py

```

1 name = "ada"
2 name.upper()
3 print(name)

```

ada

Line 2 makes 'ADA' and then discards it, because nothing stores it. Assign the result:
 name = name.upper().

PAUSE AND TRY**Question 8** Tracing

Predict exactly what the REPL prints for each line. Note the spaces at both ends of code.

```
>>> code = " ab-12-ab "  
>>> code.strip().upper()  
>>> code.count("ab")  
>>> code.replace("-", "")  
>>> code.find("12")
```

3.7 Splitting and Joining

A sentence is made of words, and a date of numbers separated by dashes. Often you want the separate pieces, or you want to glue pieces back together with something new between them. Two more methods do exactly that.

`split()` breaks a string into pieces. With no argument it splits wherever there is whitespace, and ignores extra spaces. With an argument, such as `split("-")`, it splits at each copy of that **separator** and drops the separators.

`join()` does the opposite: it glues pieces together, putting the string it is called on between each pair. Read `"-".join(pieces)` as "join the pieces with a dash".

```
>>> "Ada Lovelace".split()  
['Ada', 'Lovelace']  
>>> "  red  green blue ".split()  
['red', 'green', 'blue']  
>>> "2026-10-09".split("-")  
['2026', '10', '09']  
>>> "-".join("abc")  
'a-b-c'  
>>> "/" .join("2026-10-09".split("-"))  
'2026/10/09'
```

The result of `split` is shown in square brackets, like `['Ada', 'Lovelace']`. That is a **list**: one value holding several values in order. Lists get a whole day of their own (Day 6). Today, just read the printed result as "these strings, in this order".

`join` needs the pieces to glue. A string is a sequence of characters, so `"-".join("abc")` puts a dash between each character. The last line splits a date at its dashes and joins the pieces with slashes, turning `2026-10-09` into `2026/10/09`. (`"2026-10-09".replace("-", "/")` gives the same here.)

KEY IDEA

`text.split()` breaks a string into a list of pieces at whitespace, or at a separator you give.
`separator.join(pieces)` glues pieces into one string with the separator between them.

WATCH OUT

Calling join on the pieces instead of the separator. Writing `"abc".join("-")` reads naturally but means "join the characters of `"-"` with `abc` between them". With only one character there is nothing to put between, so you get just `'-'`. The separator always comes first: `"-".join("abc")`.

PAUSE AND TRY**Question 9** Tracing

Predict exactly what the REPL prints for each line.

```
>>> "one,two,,three".split(",")
>>> "+".join("123")
>>> " ".join("to be or not".split())
>>> "".join("a b c".split())
```

3.8 f-strings and Format Specifiers

Day 2 left a promise: `round(2.50, 2)` shows `2.5`, and a price should show as `2.50`. And building a line like `"Total: " + str(total)` with `+` and `str()` is clumsy. One tool fixes both.

An **f-string** is a string with the letter `f` just before its opening quote. Inside it, anything in curly braces `{ }` is worked out as an expression, and its value is placed into the text. A value placed into a string this way is said to be **interpolated**. The expression can be a name or any calculation, and Python converts the value to text for you, so there is no need for `str()`.

WORKED EXAMPLE Ticket sentences

`f_string_basics.py`

```
1 name = "Ana"
2 tickets = 3
3 price = 12.5
4 print(f"{name} bought {tickets} tickets.")
5 print(f"Total: {tickets * price}")
6 print(f"Total: {tickets * price:.2f}")
```

```
Ana bought 3 tickets.
Total: 37.5
Total: 37.50
```

Line 4 places three names into one sentence: no `+`, no `str()`, and the spaces are exactly where they are in the text. Line 5 works out `3 * 12.5` inside the braces and shows `37.5`. Line 6 is the same calculation with `:.2f` after it, which shows exactly two decimal places.

The `:.2f` is a **format specifier**: instructions, after a colon inside the braces, for how to show the value. Take `.2f` apart:

- `:` separates the expression from the instructions.
- `.2` means two digits after the decimal point.
- `f` means **fixed-point**: always show exactly that many decimal places, rounding or adding zeros as needed.

Change the 2 to get a different number of places: `:.1f` for one, `:.3f` for three.

```
>>> price = 2.5
>>> f"{price:.2f}"
'2.50'
>>> f"{3.14159:.3f}"
'3.142'
>>> f"{0.1 + 0.2:.2f}"
'0.30'
>>> f"{7:.2f}"
'7.00'
>>> round(2.5, 2)
2.5
```

Each line gives a string with exactly the decimals asked for: 2.50, 3.142 (rounded), 0.30 (the floating-point error from Day 2 is hidden in the display), and 7.00 from a plain int. Compare `round` and `:.2f` side by side:

Tool	Gives	2.5 becomes	Use it for
<code>round(value, 2)</code>	A number	2.5	A rounded value to compute with
<code>f"{value:.2f}"</code>	A string	'2.50'	Showing a value, such as money

KEY IDEA

In an f-string, `f"...{expression}..."`, each expression in braces is worked out and placed into the text. `{value:.2f}` shows a number with exactly two decimal places.

WATCH OUT

Forgetting the `f`. Without it, the braces are ordinary characters:

missing_f.py

```
1 name = "Ana"
2 print("Hello, {name}!")
```

```
Hello, {name}!
```

There is no error, so this mistake can slip through. If braces show up in your output, check for the `f` before the quote.

PAUSE AND TRY

Question 10 Tracing

Predict exactly what the REPL prints for each line.

```
>>> width = 4
>>> height = 2.5
>>> f"{width} x {height} = {width * height}"
>>> f"Area: {width * height:.1f} m"
>>> f"{1 / 3:.2f}"
>>> f"{2 / 3:.2f}"
```

3.9 Reading Input

Every program so far has used the same values each time it ran. To ask the user something, such as their name or age, a program needs to read what they type.

The built-in function `input()` shows a message, called a **prompt**, then waits. The user types a line and presses Enter, and `input()` returns what they typed, without the Enter, as a string. Put a space at the end of the prompt so the typing does not touch it.

`greeting.py`

```
1 name = input("What is your name? ")
2 print(f"Nice to meet you, {name}!")
```

When you run it and type `Ana`, the terminal shows:

```
What is your name? Ana
Nice to meet you, Ana!
```

Line 1 prints the prompt and pauses until Enter is pressed. Then `name` refers to `"Ana"`, and line 2 places it into an f-string.

Now vary the program to ask for two numbers and add them. Watch what happens when the user types 5 and 3:

`add_wrong.py`

```
1 first = input("First number: ")
2 second = input("Second number: ")
3 print("Sum:", first + second)
```

```
First number: 5
Second number: 3
Sum: 53
```

The sum is 53, with no error message. `input()` **always returns a string**, whatever the user types. `first` is `"5"` and `second` is `"3"`, so `+` joins them (`3.2`). `input` cannot know you wanted a number: someone might type a name, a phone number with a leading 0, or "maybe", so Python leaves the converting to you.

Convert with `int()` or `float()` from Day 2. You can wrap the call around `input()` directly:

add_right.py

```
1 first = int(input("First number: "))
2 second = int(input("Second number: "))
3 print("Sum:", first + second)
```

```
First number: 5
Second number: 3
Sum: 8
```

Python works out `int(input("First number: "))` from the inside out: `input` runs first and returns "5", then `int("5")` gives the int 5. Use `float(input(...))` for decimals such as prices. If the user types something that is not a number, such as `five`, the conversion stops with the `ValueError` from Day 2; Day 15 shows how to handle bad input.

KEY IDEA

`input(prompt)` shows the prompt, waits for a line, and always returns it as a string. Wrap it in `int()` or `float()` when you need a number: `age = int(input("Age: "))`.

WATCH OUT

Doing math with the raw input. Day 2 promised this one. Typing 15 gives the string "15":

age_next_year.py

```
1 age = input("How old are you? ")
2 print("Next year you will be", age + 1)
```

```
How old are you? 15
Traceback (most recent call last):
  File ".../day03/age_next_year.py", line 2, in <module>
    print("Next year you will be", age + 1)
                                ~~~~^~~
TypeError: can only concatenate str (not "int") to str
```

`age + 1` tries to join the string "15" and the int 1. Convert as you read:

`age = int(input("How old are you? "))`, and the program prints `Next year you will be 16`.

PAUSE AND TRY

Question 11 Tracing

The user types `0s1o`, then `3`. What does the whole terminal session show?

predict_input.py

```
1 city = input("City: ")
2 nights = input("Nights: ")
3 print(city.upper(), nights * 2)
4 print(f"{city[:3]}-{int(nights) * 2}")
```

Putting it together

A café wants a small program that prints an order slip. The user types a name (with stray spaces), a drink, the number of cups, and the price per cup. This run types `Sam`, `Latte`, `2`, and `3.75`:

coffee_order.py

```
1 # Coffee order receipt
2 name = input("Name: ").strip()
3 drink = input("Drink: ").strip().lower()
4 cups = int(input("Cups: "))
5 price = float(input("Price per cup: "))
6
7 total = cups * price
8 order_code = name[:3].upper() + str(cups)
9
10 print("-" * 24)
11 print(f"Order {order_code} for {name}")
12 print(f"{cups} x {drink}")
13 print(f"Total: {total:.2f}")
14 print("-" * 24)
```

```
Name: Sam
Drink: Latte
Cups: 2
Price per cup: 3.75
-----
Order SAM2 for Sam
2 x latte
Total: 7.50
-----
```

Line	Worked out	Value	Idea
2	<code>" Sam".strip()</code>	<code>"Sam"</code>	3.6, 3.9
3	<code>"Latte".strip().lower()</code>	<code>"latte"</code>	3.6
4	<code>int("2")</code>	<code>2</code>	3.9
5	<code>float("3.75")</code>	<code>3.75</code>	3.9
7	<code>2 * 3.75</code>	<code>7.5</code>	Day 2
8	<code>"Sam"[:3].upper() + str(2)</code>	<code>"SAM2"</code>	3.4, 3.2
10	<code>"-" * 24</code>	<code>24 dashes</code>	3.2
13	<code>f"Total: {7.5:.2f}"</code>	<code>"Total: 7.50"</code>	3.8

Lines 2 and 3 call methods straight on what `input` returns, so the stray spaces never reach a variable. Line 8 slices the name, makes it capitals, and joins the cup count as a string: `+` needs both sides to be strings, so `str(cups)` is required. Lines 11 to 13 build the slip with f-strings, and `:.2f` shows `7.5` as `7.50`.

Vocabulary

Term	Definition	Example
string (<code>str</code>)	A sequence of characters: text	<code>"Hello"</code>
character	One letter, digit, space, or symbol in a string	<code>"H"</code>
escape character	A backslash and a symbol that stand for one special character	<code>\n, \"</code>
triple-quoted string	A string in <code>"""</code> that can span several lines	<code>poem = """Roses..."""</code>
length	How many characters a string has; <code>len()</code> gives it	<code>len("cat")</code> is 3
empty string	The string with no characters	<code>""</code>
concatenation	Joining strings end to end with <code>+</code>	<code>"snow" + "ball"</code>
repetition	Repeating a string with <code>*</code> and an int	<code>"ha" * 3</code>
substring	A run of characters inside a string	<code>"York"</code> in <code>"New York"</code>
case-sensitive	Treating capital and small letters as different	<code>"a" == "A"</code> is <code>False</code>
index / indexing	A character's position, from 0 / getting the character at one	<code>word[0]</code>
negative index	A position counted from the end; -1 is the last character	<code>word[-1]</code>
IndexError	Asking for a position that does not exist	<code>"cat"[3]</code>
slice / step	Characters from <code>start</code> up to but not including <code>stop</code> / the gap between them	<code>word[1:4:2]</code>
immutable	Cannot be changed once made	<code>word[0] = "b"</code> fails
method / method chain	A function that belongs to a value / calls run left to right	<code>name.strip().lower()</code>
whitespace	Spaces, tabs, and new lines	<code>" hi\n"</code>
separator	The string that <code>split</code> cuts at or <code>join</code> puts between pieces	<code>"-"</code>
list	One value holding several in order, in square brackets (Day 6)	<code>['Ada', 'LoveLace']</code>
f-string / interpolated	A string starting with <code>f</code> / a value placed into it from braces	<code>f"Hi, {name}"</code>
format specifier	Instructions after <code>:</code> in braces for how to show a value	<code>{price:.2f}</code>
prompt	The message <code>input()</code> shows before the user types	<code>input("Age: ")</code>

Summary

- A **string** is text in single, double, or triple quotes. **Escape characters** such as `\n`, `\"`, and `\\` stand for characters you cannot type directly.
- `len()` counts characters, `+` joins strings, `*` repeats them, and `in` and `==` give `True` or `False`.
- **Indexes** start at 0, and negative indexes count from the end. A **slice** `text[start:stop:step]` stops **before** `stop`, and `text[::-1]` reverses.
- Strings are **immutable**, so slices and **methods** (`upper`, `lower`, `strip`, `replace`, `find`, `startswith`, `count`, `split`, `join`) return new values and leave the original alone.

- An **f-string** places values into text, and `:.2f` shows exactly two decimal places.
- `input()` always returns a string; convert with `int()` or `float()` before doing math.

Practice

Do these in your `day03` folder. Type each program yourself, run it, and only then open the answer. Run each one several times with different typed input.

Question 12 Short answer

Initials. Ask the user for their full name (a first and a last name separated by a space) and print their initials in capitals, followed by dots, such as `A.L.` for `ada loveLace`. Hint: `find` can tell you where the space is.

Question 13 Short answer

Reverse and palindrome. A **palindrome** reads the same forwards and backwards, like `"level"`. Ask for a word, print it reversed using slicing, then print whether it is a palindrome as `True` or `False`. Make `Racecar` count as a palindrome even with its capital R.

Question 14 Short answer

Receipt line. Ask for a price and a quantity, then print one receipt line showing the quantity, the price, and the total, with both money amounts to two decimal places, such as `3 x 4.50 = 13.50`.

Review questions

Question 15 Tracing

What does this program print?

`book_title.py`

```
1 title = " the hobbit "  
2 clean = title.strip()  
3 print(clean.upper())  
4 print(clean[4:].replace("b", "B"))  
5 print(len(clean), clean.count("t"))
```

Question 16 Multiple choice

Which expression gives the string `'2.50'`?

- A. `round(2.5, 2)`
- B. `f"{2.5:.2f}"`
- C. `f"{2.5}:.2f"`
- D. `"{2.5:.2f}"`

Question 17 Tracing

What does this program print? Track `word` and `new_word`.

stone.py

```

1 word = "stone"
2 word.upper()
3 new_word = word.replace("s", "")
4 word = word + "s"
5 print(word, new_word)
6 print(word.find("t"), new_word.find("s"))

```

Question 18 Debugging

This program should print the cost of the tickets at 12.50 each. The user types 3, and it stops with the error below. Explain it and write the fix.

ticket_cost.py

```

1 tickets = input("Tickets: ")
2 total = tickets * 12.5
3 print(f"Total: {total:.2f}")

```

```

Tickets: 3
Traceback (most recent call last):
  File "../day03/ticket_cost.py", line 2, in <module>
    total = tickets * 12.5
            ~~~~~^~~~~~
TypeError: can't multiply sequence by non-int of type 'float'

```

Question 19 Short answer

A card number is stored as the string "4111222233334444". Write a program that prints it with every digit except the last four replaced by *, and that still works if the number has a different length.

Go further

The official Python Tutorial covers strings, indexing, and slicing in [An Informal Introduction to Python: Text, and f-strings in Input and Output: Formatted String Literals](#).

Reflection

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

- ☐ I can write strings with quotes inside them and use escape characters. Reread: [3.1 Quotes and Escape Characters](#)
- ☐ I can use len, +, *, in, and == with strings. Reread: [3.2 Length, Concatenation, and in](#)
- ☐ I can get any character with a positive or negative index. Reread: [3.3 Indexing: One Character at a Time](#)
- ☐ I can slice any part of a string, use a step, and reverse a string. Reread: [3.4 Slicing](#)
- ☐ I can explain why a string cannot be changed in place, and build a new one instead. Reread: [3.5 Strings Are Immutable](#)
- ☐ I can clean and search text with string methods. Reread: [3.6 String Methods](#)

- ☐ I can split a string into pieces and join pieces back together. Reread: 3.7 Splitting and Joining
- ☐ I can build formatted output with f-strings and `:.2f`. Reread: 3.8 f-strings and Format Specifiers
- ☐ I can read input and convert it to a number. Reread: 3.9 Reading Input

Check Your Reasoning

Answers and explanations for every question, numbered to match.

Question 1

```
Ready,  
set,  
go!  
Say "cheese"  
Don't type \n
```

- Line 1 has two `\n` escapes, so it prints on three lines.
- Line 2's `\"` escapes print as plain double quotes.
- Line 3 is in single quotes, so the apostrophe needs `\'`. Then `\\` prints one backslash, and the `n` after it is an ordinary letter, so you see `\n` as two characters, not a line break.

Reread *3.1 Quotes and Escape Characters* if your output differed.

Question 2

C. The double quotes around the text are escaped with `\"`, so they do not end the string, and the apostrophe is safe inside double quotes.

- **A** is wrong: the apostrophe ends the single-quoted string after `Ana`, a `SyntaxError`.
- **B** is wrong: the `"` before `cat` ends the string, and `cat` becomes a stray name, a `SyntaxError`.
- **D** is wrong: `\'` is fine, but the unescaped `"` before `cat` still ends the string.

Reread *3.1 Quotes and Escape Characters* if you chose differently.

Question 3

```
>>> len("hi there")  
8  
>>> "ab" * 2 + "c"  
'ababc'  
>>> "3" + "4"  
'34'  
>>> "car" in "Scar"  
True
```

- `"hi there"` has 2 letters, a space, and 5 letters: **8**. The space counts.
- `*` comes before `+`, so `"ab" * 2` is `'abab'`, then `+"c"` gives `'ababc'`.
- `"3"` and `"4"` are strings, so `+` joins them: `'34'`, not 7.
- `"car"` appears inside `"Scar"` (the last three letters), so **True**.

Reread *3.2 Length, Concatenation, and in* if any answer differed.

Question 4

```
>>> city = "Lisbon"
>>> city[1]
'i'
>>> city[-2]
'o'
>>> city[len(city) - 3]
'b'
```

The indexes are L0 i1 s2 b3 o4 n5, and from the end n-1 o-2 b-3 s-4 i-5 L-6.

- `city[1]` is the second character, `'i'` (not `'L'`, which is index 0).
- `city[-2]` is second from the end, `'o'`.
- `len(city)` is 6, so the index is $6 - 3 = 3$, which holds `'b'`.

Reread 3.3 *Indexing: One Character at a Time* if any answer differed.

Question 5

```
>>> item = "notebook"
>>> item[0:4]
'note'
>>> item[4:]
'book'
>>> item[1:6:2]
'oeo'
>>> item[::-1]
'koobeton'
```

The indexes are n0 o1 t2 e3 b4 o5 o6 k7.

- `item[0:4]` takes indexes 0 to 3: `'note'`.
- `item[4:]` takes 4 to the end: `'book'`.
- `item[1:6:2]` takes 1, 3, and 5 (6 is excluded): o, e, o, so `'oeo'`.
- `item[::-1]` is the whole word backwards: `'koobeton'`.

Reread 3.4 *Slicing* if any answer differed.

Question 6

B. The 2 is at index 3 and the last 6 at index 6, so the stop must be one past it, 7. Check: $7 - 3 = 4$ characters.

- **A** is wrong: the stop is excluded, so it gives `'202'`.
- **C** is wrong: index 2 is the dash, so it gives `'-2026'`.
- **D** is wrong: -6 is the 2 but -1 is x, and stopping before it still includes the second dash: `'2026-'`.

Reread 3.4 *Slicing* if you chose differently.

Question 7

```
TypeError: 'str' object does not support item assignment
```

`title[0].upper()` on the right side is fine: it reads index 0 and returns the new string "H". The problem is the left side, which tries to put that character into position 0 of `title`, and strings are immutable. Build a new string from the capital letter and the rest of the text, and rebind `title`:

capitalize_title_fixed.py

```
1 title = "hello world"
2 title = title[0].upper() + title[1:]
3 print(title)
```

Hello world

`title[1:]` is "ello world", so line 2 makes "Hello world".

Reread 3.5 *Strings Are Immutable* if you missed it.

Question 8

```
>>> code = " ab-12-ab "
>>> code.strip().upper()
'AB-12-AB'
>>> code.count("ab")
2
>>> code.replace("-", "")
' ab12ab '
>>> code.find("12")
4
```

- `strip()` removes the end spaces, then `upper()` makes 'AB-12-AB'.
- "ab" appears twice.
- `replace` removes both dashes but keeps the end spaces: ' ab12ab '.
- Counting from 0, the space is 0, a 1, b 2, - 3, so "12" starts at 4.

Reread 3.6 *String Methods* if any answer differed.

Question 9

```
>>> "one,two,,three".split(",")
['one', 'two', '', 'three']
>>> "+".join("123")
'1+2+3'
>>> ".join("to be or not".split())
'to be or not'
>>> "".join("a b c".split())
'abc'
```

- Splitting at each comma gives four pieces. Between the two commas there is nothing, so one piece is the empty string ''.
- "+" goes between each character of "123": '1+2+3'.
- `split()` with no argument ignores the extra spaces, so the words are joined back with single spaces: 'to be or not'.
- Joining with the empty string "" puts nothing between the pieces: 'abc'.

Reread 3.7 *Splitting and Joining* if any answer differed.

Question 10

```
>>> width = 4
>>> height = 2.5
>>> f"{width} x {height} = {width * height}"
'4 x 2.5 = 10.0'
>>> f"Area: {width * height:.1f} m"
'Area: 10.0 m'
>>> f"{1 / 3:.2f}"
'0.33'
>>> f"{2 / 3:.2f}"
'0.67'
```

- With no specifier, each value appears as `print` would show it. `4 * 2.5` is the float `10.0`.
- `:.1f` gives one decimal place: `10.0`.
- `1/3` is `0.333...`, shown to two places as `0.33`.
- `2/3` is `0.666...`, which **rounds** to `0.67`; `:.2f` rounds, it does not just cut off.

Reread 3.8 *f-strings and Format Specifiers* if any answer differed.

Question 11

```
City: Oslo
Nights: 3
OSLO 33
Osl-6
```

- `city` is `"Oslo"` and `nights` is the **string** `"3"`.
- Line 3: `city.upper()` is `OSLO`, and `nights * 2` repeats the string: `"33"`, not 6.
- Line 4: `city[:3]` is `Osl`, and `int(nights) * 2` converts first, so it is the number 6.

Reread 3.9 *Reading Input* if your output differed.

Question 12

One possible solution:

initials.py

```
1 full_name = input("Full name: ").strip()
2 space = full_name.find(" ")
3 first_initial = full_name[0].upper()
4 last_initial = full_name[space + 1].upper()
5 print(f"Initials: {first_initial}.{last_initial}.")
```

```
Full name: ada lovelace
Initials: A.L.
```

`strip()` removes stray spaces at the ends, so index 0 is really the first letter. `find(" ")` gives the space's index, 3, so the last name starts at `space + 1`, index 4. `upper()` capitalizes each initial even if the user typed in small letters. The f-string adds the dots.

Reread 3.3 *Indexing: One Character at a Time* and 3.6 *String Methods* if your result differed.

Question 13

One possible solution:

palindrome.py

```

1 word = input("Word: ").strip().lower()
2 reversed_word = word[::-1]
3 print(f"Reversed: {reversed_word}")
4 print("Palindrome:", word == reversed_word)

```

```

Word: Racecar
Reversed: racecar
Palindrome: True

```

`lower()` makes the capital R match the small r at the end; without it, `"Racecar" == "racecaR"` is `False`. `word[::-1]` reverses the word, and `word == reversed_word` gives a bool, which `print` shows as `True` or `False`. Typing Hopper prints `reppoh` and `False`.

Reread 3.4 *Slicing* and 3.2 *Length, Concatenation, and in* if your result differed.

Question 14

One possible solution:

receipt_line.py

```

1 price = float(input("Price: "))
2 quantity = int(input("Quantity: "))
3 total = price * quantity
4 print(f"{quantity} x {price:.2f} = {total:.2f}")

```

```

Price: 4.5
Quantity: 3
3 x 4.50 = 13.50

```

The price can have a decimal point, so it uses `float(input(...))`; the quantity is a whole number, so it uses `int(input(...))`. `4.5 * 3` is `13.5`, and `:.2f` shows both money values with two places. Without the conversions, `"4.5" * "3"` would be a `TypeError`.

Reread 3.8 *f-strings and Format Specifiers* and 3.9 *Reading Input* if your result differed.

Question 15

```

THE HOBBIT
hoBBit
10 2

```

Line	Worked out	Value
2	<code>strip()</code> removes the end spaces	"the hobbit"
3	<code>upper()</code> of <code>clean</code>	THE HOBBIT
4	<code>clean[4:]</code> is "hobbit", then every b becomes B	hoBBit
5	10 characters (the space counts); t in "the" and "hobbit"	10 2

`clean[4:]` starts at index 4 because "the " takes indexes 0 to 3. `replace` changes **every** b, not just the first.

Reread 3.4 *Slicing* and 3.6 *String Methods* if your output differed.

Question 16

B. An f-string with the format specifier `:.2f` inside the braces shows exactly two decimal places.

- **A** is wrong: `round` gives the number `2.5`; a float never shows a trailing zero.
- **C** is wrong: the `:.2f` is outside the braces, so it is plain text: `'2.5: .2f'`.
- **D** is wrong: without the `f`, the braces are ordinary characters: `'{2.5: .2f}'`.

Reread 3.8 *f-strings and Format Specifiers* if you chose differently.

Question 17

```
stones tone
1 -1
```

Line	Worked out	word	new_word
1	"stone"	"stone"	
2	"STONE", thrown away	"stone"	
3	remove every s	"stone"	"tone"
4	"stone" + "s"	"stones"	"tone"

Line 2 changes nothing: the result is never stored. In line 6, `t` is at index 1 of `"stones"`, and `"tone"` has no `s`, so `find` gives `-1`. If you printed `STONES`, reread the Watch Out in 3.6.

Reread 3.5 *Strings Are Immutable* and 3.6 *String Methods* if your output differed.

Question 18

`tickets` is the string `"3"`, because `input` always returns a string. A string times an int is repetition (3.2), but a string times a float has no meaning, so Python stops. ("Sequence" here means the string.) Convert the input:

`ticket_cost_fixed.py`

```
1 tickets = int(input("Tickets: "))
2 total = tickets * 12.5
3 print(f"Total: {total:.2f}")
```

```
Tickets: 3
Total: 37.50
```

Worse than the error is the near miss: with a price of 12 instead of 12.5, `"3" * 12` would quietly give `"333333333333"`. Converting input right away avoids both.

Reread 3.9 *Reading Input* if you missed it.

Question 19

One possible solution:

`mask_card.py`

```
1 card = "4111222233334444"
2 masked = "*" * (len(card) - 4) + card[-4:]
3 print(masked)
```

```
*****4444
```

`card[-4:]` is the last four characters, from 3.4. The number of stars is the length minus 4, worked out with `len` so that any length works, and `*` repeats the star that many times. A fixed `"*" * 12` would be wrong for a 15-digit card.

Reread 3.2 *Length, Concatenation, and in* and 3.4 *Slicing* if your result differed.