

Day 2 Variables, numbers and basic types

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

By the end you will be able to

- Store values in variables with clear snake_case names and trace how they change.
- Predict the type and value of any arithmetic expression, including floor division, remainder, and powers.
- Check a value's type with `type()` and convert between `int`, `float`, and `str`.
- Explain why `0.1 + 0.2` is not exactly `0.3` and round a result for display.

What to remember from earlier

- At the REPL's `>>>` prompt Python shows each result right away, but a saved `.py` script shows only what you print. ([Day 1: Setup and your first program, 1.3 Writing and Running a Script](#))
- A script runs from the top line to the bottom, one line at a time. ([Day 1: Setup and your first program, 1.5 Python Reads Top to Bottom](#))
- `print` can show several values separated by commas, all on one line. ([Day 1: Setup and your first program, 1.4 Printing Text and Numbers](#))
- Python does `*` and `/` before `+` and `-`, parentheses go first, and `/` always gives a decimal point, such as `4.0`. ([Day 1: Setup and your first program, 1.2 The REPL: Python as a Calculator](#))
- A word without quotes is a name, and a name Python does not know stops the program with a `NameError`. ([Day 1: Setup and your first program, 1.9 Reading a NameError](#))

Make a `day02` folder for today's files, open it in your editor, and run each script from its terminal, just as you did with `day01`.

2.1 Variables: Giving a Value a Name

On Day 1, every number you used was typed straight into the expression, like `30 - 12`. That works once. But imagine a program that uses a ticket price in five places: if the price changes, you must find and edit all five, and missing one gives a wrong answer. Programs need a way to work a value out once, remember it, and refer to it by name.

A **variable** is a name that refers to a value. You create one with an **assignment statement**, which has three parts: a name, an equals sign, and an expression.

```
name = expression
```

Python runs an assignment in two steps. First it works out the expression on the **right** side. Then it attaches the name on the **left** side to that value. From then on, wherever you write the name, Python uses the value it refers to. The `=` sign is called the **assignment operator**. It does not mean "is equal to" as in math; read `price = 4` as "price **gets** 4" or "let price refer to 4".

Think of a variable as a name tag stuck on a value. Python keeps the value, and the tag lets you find it again.

WORKED EXAMPLE A ticket total

Watch what each name refers to after each line.

ticket_total.py

```
1 ticket_price = 12
2 tickets_sold = 35
3 total = ticket_price * tickets_sold
4 print("Tickets sold:", tickets_sold)
5 print("Total:", total)
```

```
Tickets sold: 35
Total: 420
```

Line	ticket_price	tickets_sold	total
1	12		
2	12	35	
3	12	35	420
4	12	35	420
5	12	35	420

Lines 1 and 2 each work out a number and attach a name to it. Line 3 looks up both names, works out $12 * 35$, and attaches `total` to 420. Lines 4 and 5 look up a name and print its value; they change nothing.

Line 3 never mentions 12 or 35; it uses the names, and Python looks up their values. If the price changes, you edit line 1 and everything after it uses the new price.

At the REPL, an assignment shows nothing, because a statement that only attaches a name has no value to show. Typing the name by itself is an expression, so the REPL shows its value:

```
>>> price = 4
>>> price
4
>>> price * 3
12
```

Now vary one thing: assign to a name that already has a value. The name moves to the new value, and the old value is forgotten. This lets you **update** a variable, even using its own old value on the right side.

score_update.py

```
1 score = 10
2 print("Start:", score)
3 score = score + 5
4 print("After bonus:", score)
5 score = 0
6 print("After reset:", score)
```

Start: 10
After bonus: 15
After reset: 0

Line	Right side worked out	score after the line
1	10	10
3	score + 5 is 10 + 5	15
5	0	0

Line 3, `score = score + 5`, would be nonsense as a math equation, but as an assignment it makes sense: the right side runs first, using the **current** value 10, and only then is `score` attached to the result, 15.

KEY IDEA

An assignment `name = expression` works out the right side first, then makes the name on the left refer to that value. Assigning again replaces the old value.

WATCH OUT

Using a variable before it has a value. A script runs top to bottom, so a name exists only after the line that assigns it has run:

use_before_assign.py

```
1 print("Total:", total)
2 total = 12 * 35
```

```
Traceback (most recent call last):
  File ".../day02/use_before_assign.py", line 1, in <module>
    print("Total:", total)
                      ^^^^^
NameError: name 'total' is not defined
```

This is the same `NameError` you met on Day 1: when line 1 runs, Python does not yet know any name `total`, because line 2 has not run. Move the assignment above the `print`. The same error appears if you misspell a variable's name, because to Python a misspelled name is simply a different, unknown name.

PAUSE AND TRY

Question 1 Tracing

What does this program print? Make a table of `apples` and `pears` after each line.

fruit_trace.py

```
1 apples = 6
2 apples = apples + 4
3 pears = apples - 3
4 apples = 2
5 print(apples, pears)
```

Question 2 Debugging

This program should print the area of an 8 by 20 rectangle. Instead it shows the message below. Explain the mistake and write the fix.

reversed_assignment.py

```
1 width = 8
2 20 = height
3 print(width * height)
```

```
File ".../day02/reversed_assignment.py", line 2
```

```
    20 = height
```

```
    ^^
```

```
SyntaxError: cannot assign to literal here. Maybe you meant '==' instead of '='?
```

2.2 Naming Rules and snake_case

You get to choose every variable name, so two questions follow: which names does Python accept, and which names make a program easy to read?

Python's rules for names (programmers call them **identifiers**) are short:

1. A name may contain letters, digits, and the underscore `_`, and nothing else: no spaces, hyphens, or other symbols.
2. A name cannot start with a digit.
3. Capital and small letters count as different (Python is **case-sensitive**, as Day 1 showed), so `score` and `Score` are two different names.
4. A name cannot be a **keyword**, one of about 35 words Python reserves for its own grammar, such as `class`, `if`, `for`, `True`, `False`, and `None`. You will learn most of them over the coming days.

Why these rules? Python has to tell names apart from everything else on a line. A name starting with a digit would look like a number, a space would make one name look like two, and a hyphen looks like a minus sign.

On top of the rules there is a **convention**, a habit every Python programmer follows even though Python does not enforce it. Python's official style guide says variable names use **snake_case**: all lowercase, with words joined by underscores, like `ticket_price` or `max_speed_kmh`. Choose names that say what the value means: `tickets_sold` tells a reader far more than `t` or `x`.

WORKED EXAMPLE Names that work

good_names.py

```

1 first_place = "Ana"
2 player2_score = 18
3 max_speed_kmh = 120
4 print(first_place, player2_score, max_speed_kmh)

```

Ana 18 120

All three names follow the rules and the convention. `player2_score` contains a digit, which is fine because it does not **start** with one. `max_speed_kmh` adds the unit, a good habit that prevents mix-ups between kilometres and miles.

Now vary the names so they break the rules. Each of these programs stops with a `SyntaxError` before anything runs:

name_starts_with_digit.py

```

1 2nd_place = "Ben"
2 print(2nd_place)

```

SyntaxError: invalid decimal literal

name_with_space.py

```

1 my score = 10
2 print(my score)

```

SyntaxError: invalid syntax

name_is_keyword.py

```

1 class = "Algebra"
2 print(class)

```

SyntaxError: invalid syntax

In the first, Python reads `2` as the start of a number and then finds letters glued to it, which no number can contain ("decimal literal" means a number written in ordinary digits). In the second, Python sees two names, `my` and `score`, side by side, which breaks the grammar. In the third, `class` is a keyword, so it cannot be a variable name. The fixes are `second_place`, `my_score`, and a name like `class_name`.

Name	Allowed?	Why
<code>total_cost</code>	Yes	Letters and underscores; good snake_case
<code>player2</code>	Yes	A digit is fine after the first character
<code>2nd_place</code>	No	Starts with a digit
<code>total cost</code>	No	Contains a space
<code>total-cost</code>	No	Python reads <code>-</code> as minus

class	No	A keyword
TotalCost	Yes	Allowed, but not the convention for variables

KEY IDEA

A variable name uses letters, digits, and underscores, does not start with a digit, and is not a keyword. By convention, Python variable names are descriptive snake_case, like `tickets_sold`.

WATCH OUT

Two names that differ only in capitals. Python is case-sensitive, so typing `Score` when you meant `score` does not cause an error: it quietly creates a second variable, and `score` keeps its old value. Sticking to all-lowercase snake_case avoids this. Also avoid naming a variable after something Python already uses, like `print` or `type` (2.3): Python allows it, but the name then refers to your value instead of Python's tool.

PAUSE AND TRY**Question 3** Multiple choice

Which is a valid variable name that also follows the snake_case convention?

- A. `final score`
- B. `final_score`
- C. `2final_score`
- D. `final-score`

2.3 Integers and Floats

On Day 1, `8 / 2` printed `4.0` instead of `4`, and the lesson promised an explanation. Here it is.

Every value in Python has a **type**: the kind of value it is. The type decides how the value is stored and what you can do with it. Numbers come in two main types:

- An **int** (short for **integer**) is a whole number, with no decimal point: `42`, `0`, `-7`. Python ints can be as large as you like.
- A **float** (short for **floating-point number**) is a number with a decimal point: `3.5`, `0.45`, `-2.0`. Floats can hold fractions, but (as 2.9 shows) only approximately.

`42` and `42.0` are the same amount, but not the same type. The decimal point is what makes a number a float.

To ask Python for a value's type, call the built-in function `type()` with the value as its argument. The REPL shows the answer as `<class 'int'>` or `<class 'float'>`; "class" is Python's word for a type, so read `<class 'int'>` as "the type is int".

```
>>> type(42)
<class 'int'>
>>> type(-7)
<class 'int'>
>>> type(42.0)
<class 'float'>
>>> type(3.5)
<class 'float'>
```

When you combine two numbers, the types of the inputs decide the type of the result:

- `+`, `-`, and `*` with two ints give an int.
- If **either** number is a float, the result is a float.
- `/` **always** gives a float, even when both numbers are ints and the division comes out even.

```
>>> 6 + 4
10
>>> 6 + 4.0
10.0
>>> 3 * 2.5
7.5
>>> 8 / 2
4.0
>>> type(8 / 2)
<class 'float'>
```

Why does `/` always give a float? Most divisions do not come out even (`7 / 2` is 3.5), and if `/` gave an int only when the answer happened to be whole, a program could not know which type to expect. So `/` is predictable: always a float.

WORKED EXAMPLE The types in a book order

book_types.py

```
1 pages = 320
2 weight_kg = 0.45
3 print(pages, type(pages))
4 print(weight_kg, type(weight_kg))
5 print(pages * 2, type(pages * 2))
6 print(pages / 2, type(pages / 2))
```

```
320 <class 'int'>
0.45 <class 'float'>
640 <class 'int'>
160.0 <class 'float'>
```

Expression	Inputs	Result	Why
pages	int	320	Whole number, no point
weight_kg	float	0.45	Has a decimal point
pages * 2	int, int	640	* of two ints is int
pages / 2	int, int	160.0	/ always gives float

Notice `type()` works on a variable too: it reports the type of the value the name refers to. Lines 5 and 6 work out the expression first, then ask for its type.

KEY IDEA

An int is a whole number and a float has a decimal point. Mixing an int with a float gives a float, and `/` always gives a float. `type(value)` tells you which type you have.

WATCH OUT

Expecting an even division to give an int. `8 / 2` is `4.0`, not `4`, so a result like a number of people or pages ends up as a float. If you need a whole-number answer from a division, use `//` from `2.4` instead of `/`.

PAUSE AND TRY

Question 4 Multiple choice

What is the value and type of `10 / 5`?

- A. `2`, an int
- B. `2.0`, a float
- C. `2`, a float
- D. `2.0`, an int

Question 5 Tracing

Predict exactly what the REPL prints for each line.

```
>>> 3 + 4.0
>>> type(9 - 2)
>>> type(9 / 3)
```

2.4 Floor Division, Remainders, and Powers

A movie is 135 minutes long. How many hours and minutes is that? `135 / 60` gives `2.25`, which is true but not what you want. You want "2 hours and 15 minutes": a whole number of hours, and the minutes left over. Python has an operator for each part.

- **Floor division** `//` divides and then rounds **down** to a whole number. `135 // 60` is 2, because 60 fits into 135 two whole times. ("Floor" means the whole number at or below the answer, like the floor below you.)
- **Modulo** `%` (also called the **remainder** operator) gives what is left over after floor division. `135 % 60` is 15, because two 60s make 120, and `135 - 120` is 15. The `%` sign here has nothing to do with percentages.
- **Exponentiation** `**` raises a number to a power. `2 ** 10` means 2 multiplied by itself 10 times, which is 1024.

```
>>> 135 / 60
2.25
>>> 135 // 60
2
>>> 135 % 60
15
>>> 2 ** 10
1024
>>> 10 ** 2
100
>>> 9 ** 0.5
3.0
```

Check `//` and `%` against each other: 2 whole hours of 60 minutes plus 15 left over is $2 \times 60 + 15 = 135$, the number you started with. That check always works, because `//` counts the whole groups and `%` counts what is left.

The last line raises 9 to the power 0.5, which is the square root of 9. It gives `3.0`, a float, because one of the inputs is a float: the type rules from 2.3 apply to `//`, `%`, and `**` too.

WORKED EXAMPLE Minutes to hours and minutes

minutes_to_hours.py

```
1 total_minutes = 135
2 hours = total_minutes // 60
3 minutes = total_minutes % 60
4 print(total_minutes, "minutes is", hours, "hours and", minutes, "minutes")
```

```
135 minutes is 2 hours and 15 minutes
```

Line	Expression	Worked out	Value
2	<code>total_minutes // 60</code>	$135 \div 60$ is 2.25, round down	2
3	<code>total_minutes % 60</code>	$135 - 2 \times 60 = 135 - 120$	15

Line 4 prints six arguments, separated by spaces. Change `total_minutes` to 200 and run it again: you get 3 hours and 20 minutes.

Now vary the numbers, to see what the rules really depend on:

```
>>> 14 % 2
0
>>> 15 % 2
1
>>> 7.5 // 2
3.0
>>> -7 // 2
-4
```

- `% 2` gives 0 for every even number and 1 for every odd number, so the remainder tells you whether a number is even.
- `7.5 // 2` gives `3.0`: still rounded down to a whole number, but a float, because 7.5 is a float.

- `-7 // 2` gives `-4`, not `-3`. The exact answer is `-3.5`, and rounding **down** means toward the lower number, which for negatives is `-4`. Floor division always rounds down, never just "chops off" the decimal.

One more edge case: dividing by zero has no answer, so `/`, `//`, and `%` all stop the program with a `ZeroDivisionError`:

share_pizza.py

```
1 pizza_slices = 12
2 people = 0
3 print("Each person gets", pizza_slices // people)
```

```
Traceback (most recent call last):
  File "../day02/share_pizza.py", line 3, in <module>
    print("Each person gets", pizza_slices // people)
                                ~~~~~^~~~~~
ZeroDivisionError: division by zero
```

Read it as you learned on Day 1, from the bottom up: the last line names the error, and the `^^` marks point at the `//` that failed. Like a `NameError`, it happens while the program runs, so any lines before line 3 would already have run.

Operator	Name	With 135 and 60	Type of result with two ints
<code>/</code>	Division	2.25	Always float
<code>//</code>	Floor division	2	int
<code>%</code>	Modulo	15	int
<code>**</code>	Exponent	A 128-digit number	int, if the power is 0 or more

KEY IDEA

`//` divides and rounds down, `%` gives the remainder, and `**` raises to a power. Together, `a // b` whole groups and `a % b` left over always rebuild `a`.

PAUSE AND TRY

Question 6 Tracing

Predict exactly what the REPL prints for each line.

```
>>> 17 // 5
>>> 17 % 5
>>> 17 / 5
>>> 2 ** 5
```

2.5 Operator Precedence

With seven arithmetic operators, an expression like `10 + 3 ** 2 * 2 - 8 // 3` needs a clear rule for what happens first. Day 1 gave part of it: multiplication and division before addition and subtraction, parentheses first. Here is the full rule for today's operators.

Operator precedence is the order in which Python applies operators. From first to last:

Order	Operators	Notes
1	()	Parentheses, innermost first
2	**	Power; groups from the right
3	- before a number	A negative sign, as in <code>-3</code>
4	* / // %	Left to right
5	+ -	Left to right

Operators on the same row are applied left to right, in the order they appear, with one exception: a chain of `**` is worked out from the right.

WORKED EXAMPLE Evaluating step by step

Evaluate `10 + 3 ** 2 * 2 - 8 // 3` the way Python does, one operator at a time:

Step	Rule used	Expression now
0	Start	<code>10 + 3 ** 2 * 2 - 8 // 3</code>
1	<code>**</code> first: <code>3 ** 2</code> is 9	<code>10 + 9 * 2 - 8 // 3</code>
2	<code>*</code> next, leftmost: <code>9 * 2</code> is 18	<code>10 + 18 - 8 // 3</code>
3	Then <code>//</code> : <code>8 // 3</code> is 2	<code>10 + 18 - 2</code>
4	<code>+</code> and <code>-</code> left to right: <code>28 - 2</code>	<code>26</code>

```
>>> 10 + 3 ** 2 * 2 - 8 // 3
26
>>> 100 / 10 / 2
5.0
>>> 2 ** 3 ** 2
512
>>> (2 ** 3) ** 2
64
```

The REPL agrees: 26. The next lines show the same-row rules. `100 / 10 / 2` goes left to right: `100 / 10` is 10.0, then `/ 2` is 5.0. `2 ** 3 ** 2` goes right to left: `3 ** 2` is 9 first, then `2 ** 9` is 512. Parentheses change it: `(2 ** 3) ** 2` is `8 ** 2`, which is 64.

Now vary one thing in a real calculation: the parentheses. To average two test scores, you add them and divide by 2.

```
>>> 88 + 94 / 2
135.0
>>> (88 + 94) / 2
91.0
```

Without parentheses, `/` happens before `+`, so Python divides only 94 by 2 and adds 88, giving 135.0, which is not an average. With parentheses, the addition happens first, and the average is 91.0.

KEY IDEA

Python applies parentheses, then `**`, then a negative sign, then `*` `/` `//` `%`, then `+` `-`, left to right within a row. When you are not sure, add parentheses: they make the order certain and easier to read.

WATCH OUT

Squaring a negative number. `**` comes before the negative sign, so `-3 ** 2` means "the negative of `3 ** 2`":

```
>>> -3 ** 2
-9
>>> (-3) ** 2
9
```

You get -9, not 9. To square -3, put it in parentheses: `(-3) ** 2`.

PAUSE AND TRY**Question 7** Tracing

Predict exactly what the REPL prints for each line. Show the order of the steps.

```
>>> 7 + 8 // 3 * 2
>>> (7 + 8) // 3 * 2
>>> 2 * 3 ** 2
```

Question 8 Multiple choice

Which expression has the value 25?

- A. `-5 ** 2`
- B. `(-5) ** 2`
- C. `10 + 5 ** 2 // 2`
- D. `5 + 5 * 2`

2.6 True, False, and None

Not every fact a program stores is a number. Is the library open? Has a player finished? Some facts are simply yes or no, and some values do not exist yet, like the winner of a game that has not ended. Python has types for both.

A **bool** (short for **Boolean**) is a value that is either `True` or `False`. Those are the only two bool values. They are keywords, spelled with a capital letter and no quotes. On Day 4 you will make bools by comparing values (is the score over 50?); today you store and print them.

`None` is a special value that means "no value here". Its type is called `NoneType`, and `None` is its only value. A program uses `None` for something that is not known yet, such as a due date for a library card with no books out.

```
>>> is_open = True
>>> is_open
True
>>> type(is_open)
<class 'bool'>
>>> type(False)
<class 'bool'>
>>> winner = None
>>> winner
>>> print(winner)
None
>>> type(winner)
<class 'NoneType'>
```

Look at the line `>>> winner`. The REPL shows nothing, because it does not display a `None` result (there is no value to show). `print` does show it, as the word `None`. If the REPL ever seems to show nothing, `None` may be the reason.

WORKED EXAMPLE A library card

library_card.py

```
1 books_borrowed = 3
2 has_overdue_books = False
3 next_due_date = None
4 print("Books:", books_borrowed)
5 print("Overdue:", has_overdue_books)
6 print("Next due date:", next_due_date)
```

```
Books: 3
Overdue: False
Next due date: None
```

Name	Value	Type	Meaning
books_borrowed	3	int	A count
has_overdue_books	False	bool	A yes-or-no fact
next_due_date	None	NoneType	Not known; no books are due

A bool's name often starts with `is_` or `has_`, so that it reads like a yes-or-no question.

KEY IDEA

A bool is `True` or `False`, capitalized and without quotes. `None` means "no value" and has the type `NoneType`.

WATCH OUT

Writing true in lowercase. Many other languages use lowercase `true`, and students carry the habit over:

lowercase_true.py

```
1 is_member = true
2 print("Member:", is_member)
```

```
Traceback (most recent call last):
  File "../day02/lowercase_true.py", line 1, in <module>
    is_member = true
                ^^^^^
NameError: name 'true' is not defined. Did you mean: 'True'?
```

Without the capital, `true` is just an unknown name. Python even suggests the fix: `True`.

PAUSE AND TRY**Question 9** Multiple choice

Which value has the type `bool`?

- A. `"False"`
- B. `False`
- C. `false`
- D. `None`

2.7 Converting Between Types

Numbers do not always arrive as numbers. On Day 3, a program will ask the user to type their age, and whatever they type arrives as text: the string `"15"`, not the int `15`. Files hold text too. The digits look the same, but the type is different, and the type decides what you can do:

text_minus_number.py

```
1 age_text = "15"
2 print(age_text - 1)
```

```
Traceback (most recent call last):
  File "../day02/text_minus_number.py", line 2, in <module>
    print(age_text - 1)
              ~~~~~^~
TypeError: unsupported operand type(s) for -: 'str' and 'int'
```

A **TypeError** means an operation was used on a type that does not support it. An **operand** is a value an operator works on, and `str` is Python's name for the string type. So the message says: `-` does not work with a string and an int. Subtracting from text makes no sense, and Python will not guess that you meant the number.

The fix is **type conversion**: making a new value of a different type from an existing one. Python's type names double as conversion functions:

- `int(value)` makes an int: from a string of digits, or from a float by dropping everything after the decimal point.
- `float(value)` makes a float: from a string like `"2.5"`, or from an int.
- `str(value)` makes a string: the text you would see if you printed the value.

```
>>> int("42")
42
>>> int("42") - 1
41
>>> float("2.5")
2.5
>>> float(3)
3.0
>>> int(3.9)
3
>>> int(-3.9)
-3
>>> str(42)
'42'
>>> type(str(42))
<class 'str'>
```

Notice three things:

- `int(3.9)` gives 3, not 4. `int()` does not round; it drops the decimal part, so `int(-3.9)` is -3. (2.9 shows how to round.)
- `str(42)` shows as `'42'` with quotes. The REPL puts quotes around a string to show you it is text; print would show just 42. The REPL uses single quotes, which mean the same as double quotes.
- Converting makes a **new** value. It does not change the original: after `int(price_text)`, `price_text` is still a string.

WORKED EXAMPLE Prices that arrive as text

ticket_from_text.py

```
1 price_text = "12"
2 quantity_text = "3"
3 price = int(price_text)
4 quantity = int(quantity_text)
5 print("Total:", price * quantity)
6 print(type(price_text), type(price))
```

```
Total: 36
<class 'str'> <class 'int'>
```

Line	Name	Value	Type
1	price_text	"12"	str
2	quantity_text	"3"	str

3	price	12	int
4	quantity	3	int

Line 5 multiplies the two ints, $12 \times 3 = 36$. Line 6 confirms that converting did not change `price_text`: it is still a `str`, while `price` is an `int`.

Now vary the text: give `int()` a string with a decimal point.

`int_from_decimal_text.py`

```
1 height_text = "1.75"
2 height = int(height_text)
3 print("Height:", height)
```

```
Traceback (most recent call last):
  File ".../day02/int_from_decimal_text.py", line 2, in <module>
    height = int(height_text)
ValueError: invalid literal for int() with base 10: '1.75'
```

A `ValueError` means the type was acceptable (`int()` does take strings) but this particular value was not. `int()` accepts only a string of whole-number digits, optionally with a minus sign ("base 10" just means ordinary digits 0–9), and "1.75" is not one. `int("forty")` fails the same way. For text with a decimal point, use `float()`:

`float_from_decimal_text.py`

```
1 height_text = "1.75"
2 height = float(height_text)
3 print("Height:", height)
```

```
Height: 1.75
```

KEY IDEA

`int()`, `float()`, and `str()` make a new value of that type. `int()` drops the decimal part of a float, and it accepts only whole-number text; use `float()` for text with a decimal point.

WATCH OUT

Expecting `int()` to round. `int(2.99)` is 2, not 3. `int()` always drops the decimal part. If you want the nearest whole number, use `round()` from 2.9.

PAUSE AND TRY

Question 10 Tracing

Predict exactly what the REPL prints for each line. Watch for quotes.

```
>>> int(7.99)
>>> float("10")
>>> str(5 + 5)
>>> int("8") * 2
```

Question 11 Debugging

This program should print how many years until age 18. It stops with an error. Quote the last line of the error, explain it, and write the fix.

years_left.py

```
1 age_text = "15"
2 years_left = 18 - age_text
3 print("Years until 18:", years_left)
```

2.8 Dynamic Typing: Names Can Change Type

In 2.1 you reassigned a variable to a new number. Can a name that refers to an int later refer to a float, or a string? In some programming languages, no: you declare that a variable holds whole numbers, and it may only ever hold whole numbers. Python works differently.

In Python, the **value** has a type, and the **name** does not. A name is only a tag pointing at a value, and you can move the tag to any value of any type. This is called **dynamic typing**: the type that goes with a name is decided while the program runs, by whatever value it refers to at that moment. Moving a name to a new value is called **rebinding** it.

This is where the name-tag analogy beats the idea of a box: a box sized for whole numbers could not hold text, but a tag can be stuck on anything.

WORKED EXAMPLE One name, four types

changing_types.py

```
1 reading = 72
2 print(reading, type(reading))
3 reading = 72.5
4 print(reading, type(reading))
5 reading = "72 degrees"
6 print(reading, type(reading))
7 reading = None
8 print(reading, type(reading))
```

```
72 <class 'int'>
72.5 <class 'float'>
72 degrees <class 'str'>
None <class 'NoneType'>
```

Line	reading refers to	Type of that value
1	72	int

3	72.5	float
5	"72 degrees"	str
7	None	NoneType

Every assignment is legal. `type(reading)` reports the type of the value `reading` refers to **right now**, so the same expression gives four different answers on four different lines.

KEY IDEA

In Python, values have types and names do not. Assigning a value of a different type rebinds the name, and `type(name)` reports the type of the value it refers to right now.

WATCH OUT

Reusing a name for a different kind of thing. Python allows it, but later lines may still expect the old type:

temperature_mixup.py

```
1 temperature = 21
2 temperature = "warm"
3 print(temperature - 5)
```

```
Traceback (most recent call last):
  File ".../day02/temperature_mixup.py", line 3, in <module>
    print(temperature - 5)
          ~~~~~^~~~~
TypeError: unsupported operand type(s) for -: 'str' and 'int'
```

Line 2 is legal, but it rebinds `temperature` to a string, so line 3 tries to subtract from text. Give each kind of value its own name, such as `temperature` for the number and `description` for "warm". When a name's type surprises you, print `type(name)` to find out what it really refers to.

PAUSE AND TRY

Question 12 Tracing

What does this program print? Track both the value and its type.

value_trace.py

```
1 value = 7
2 value = value * 2
3 value = value / 7
4 print(value, type(value))
5 value = str(value)
6 print(value, type(value))
```

2.9 Floats Are Close, Not Exact

Try this in the REPL, and expect a surprise:

```
>>> 0.1 + 0.2
0.30000000000000004
>>> 1.1 * 3
3.3000000000000003
>>> 1 + 2
3
```

`0.1 + 0.2` is not exactly `0.3`. Nothing is broken; every programming language that uses standard floats gets the same result. The reason is how a computer stores numbers.

A computer stores numbers in **binary**, using only the digits 0 and 1. Whole numbers convert to binary exactly, so ints are always exact: `1 + 2` is exactly 3. Many fractions do not. Think of $1/3$ in ordinary decimal: 0.3333..., with threes forever. You can write 0.33 or 0.3333333, but never exactly one third. In binary, 0.1 is like that: it repeats forever, so Python stores the closest value it can fit. That stored value is a tiny bit off, the tiny errors add up, and sometimes they show in the last digit. This is called **floating-point imprecision** (or rounding error).

The error is tiny (about 0.00000000000000004 here), so it almost never matters for a calculation. It matters when you **show** the result: nobody wants to read a price of 3.3000000000000003. The fix is to round when you display a value.

The built-in function `round(number, digits)` rounds a number to that many digits after the decimal point. With no second argument, it rounds to the nearest whole number and gives an int.

```
>>> round(0.1 + 0.2, 2)
0.3
>>> round(3.14159, 2)
3.14
>>> round(2.50, 2)
2.5
>>> round(7.6)
8
>>> type(round(7.6))
<class 'int'>
```

- `round(0.1 + 0.2, 2)` gives 0.3: the tiny error disappears once you keep only two decimal places.
- `round(2.50, 2)` shows 2.5, not 2.50. `round` changes the value, not how many zeros are shown; a float never prints trailing zeros. Day 3 shows how to always print two decimal places.
- `round(7.6)` is the int 8. Unlike `int()`, which drops the decimal part, `round()` goes to the **nearest** whole number.

WORKED EXAMPLE Splitting a cost

Three friends split a 10-dollar bill.

unit_price.py

```
1 total_cost = 10
2 items = 3
3 unit_price = total_cost / items
4 print("Exact:", unit_price)
5 print("Rounded:", round(unit_price, 2))
```

```
Exact: 3.3333333333333335
Rounded: 3.33
```

$10 / 3$ is 3.333..., with threes forever, and the stored float runs out of room after about 16 digits, so the last digit is off. `round(unit_price, 2)` keeps two decimal places for display. `unit_price` itself is unchanged: `round` makes a new value, just like the conversion functions in 2.7.

KEY IDEA

Floats are stored in binary, so many decimals, like 0.1, are stored as a very close approximation, and tiny errors can show in the last digit. Ints are exact. Use `round(value, digits)` when you display a float.

WATCH OUT

Rounding too early. Round only when you display a result, not in the middle of a calculation:

round_too_early.py

```
1 unit_price = round(10 / 3, 2)
2 print("Price of 3:", unit_price * 3)
```

```
Price of 3: 9.99
```

Rounding the share to 3.33 first threw away a third of a cent, and multiplying by 3 made the loss visible: three shares total 9.99, not 10. Keep the full value for calculations and round the final answer.

PAUSE AND TRY**Question 13** Tracing

Predict exactly what the REPL prints for each line.

```
>>> round(7 / 3, 2)
>>> round(1.1 * 3, 1)
```

Putting it together

Jordan is planning a 523-kilometre road trip at an average 90 km/h. The car uses 6.8 litres of fuel per 100 km, and fuel costs 1.89 per litre. The distance arrives as text (as if typed in), so it has to be converted. The `#` comments are from Day 1; the blank lines just split the program into steps.

road_trip.py

```
1 # Road trip planner
2 distance_text = "523" # kilometres, as text
3 distance_km = int(distance_text)
4 speed_kmh = 90
5 litres_per_100_km = 6.8
6 price_per_litre = 1.89
7
8 total_minutes = distance_km * 60 // speed_kmh
9 hours = total_minutes // 60
10 minutes = total_minutes % 60
11
12 fuel_litres = distance_km / 100 * litres_per_100_km
13 fuel_cost = fuel_litres * price_per_litre
14
15 print("Distance:", distance_km, "km")
16 print("Driving time:", hours, "hours", minutes, "minutes")
17 print("Fuel needed:", round(fuel_litres, 1), "litres")
18 print("Fuel cost:", round(fuel_cost, 2))
```

```
Distance: 523 km
Driving time: 5 hours 48 minutes
Fuel needed: 35.6 litres
Fuel cost: 67.22
```

Follow the values and types, in the order Python runs the lines:

Name (line)	Worked out	Value	Type
distance_text (2)	A string literal	"523"	str
distance_km (3)	int("523") (2.7)	523	int
total_minutes (8)	523 * 60 is 31380, then // 90 (2.5: left to right)	348	int
hours (9)	348 // 60 (2.4)	5	int
minutes (10)	348 % 60, since 5 × 60 = 300 (2.4)	48	int
fuel_litres (12)	523 / 100 * 6.8; / makes it a float (2.3)	35.564...	float
fuel_cost (13)	fuel_litres * 1.89	67.21596...	float

The last two lines round only when printing (2.9): `round(fuel_litres, 1)` shows 35.6 and `round(fuel_cost, 2)` shows 67.22. Notice line 8: `*` and `//` share a precedence row, so Python multiplies first because it comes first, giving whole minutes. Writing `distance_km // speed_kmh * 60` instead would round down to 5 hours **before** converting to minutes and lose the 48 extra minutes.

Vocabulary

Term	Definition	Example
variable	A name that refers to a value	<code>price</code>
assignment statement	<code>name = expression</code> : work out the right side, then attach the name to it	<code>price = 4</code>
assignment operator	The <code>=</code> sign in an assignment; read it as "gets"	<code>total = 0</code>
identifier	A name you choose, following Python's naming rules	<code>tickets_sold</code>
keyword	A word Python reserves for its grammar; it cannot be a name	<code>class</code> , <code>True</code> , <code>None</code>
snake_case	The convention of lowercase words joined by underscores	<code>max_speed_kmh</code>
type	The kind of a value, which decides what you can do with it	<code>int</code> , <code>float</code> , <code>str</code>
int (integer)	A whole number with no decimal point	<code>42</code> , <code>-7</code>
float (floating-point number)	A number with a decimal point, stored approximately	<code>3.5</code> , <code>2.0</code>
floor division	<code>//</code> : divide, then round down to a whole number	<code>135 // 60</code> is 2
modulo (remainder)	<code>%</code> : what is left over after floor division	<code>135 % 60</code> is 15
exponentiation	<code>**</code> : raise a number to a power	<code>2 ** 10</code> is 1024
operator precedence	The order in which Python applies operators	<code>**</code> before <code>*</code> before <code>+</code>
bool (Boolean)	A value that is <code>True</code> or <code>False</code>	<code>has_overdue_books = False</code>
None	The value meaning "no value"; its type is <code>NoneType</code>	<code>next_due_date = None</code>
type conversion	Making a new value of another type with <code>int()</code> , <code>float()</code> , or <code>str()</code>	<code>int("42")</code> is 42
TypeError / operand	An operator used on an unsupported type / a value an operator works on	<code>"15" - 1</code>
ValueError	A function got the right type but an unusable value	<code>int("1.75")</code>
ZeroDivisionError	Dividing by zero	<code>12 // 0</code>
dynamic typing / rebinding	Names have no fixed type / moving a name to a new value	<code>reading = "warm"</code> after 72
floating-point imprecision	Tiny errors because many decimals cannot be stored exactly in binary	<code>0.1 + 0.2</code>

Summary

- A **variable** is a name that refers to a value. An **assignment** works out the right side first, then attaches the name on the left; assigning again replaces the value. Names use letters, digits,

and underscores, never start with a digit, are not **keywords**, and by convention are descriptive **snake_case**.

- Every value has a **type**: **int** (whole numbers), **float** (decimals), **bool** (`True` or `False`), **None** (no value), and **str** (text). `type()` reports it.
- `/` always gives a float, `//` divides and rounds down, `%` gives the remainder, and `**` raises to a power. A float anywhere in a calculation makes the result a float.
- **Precedence**: parentheses, then `**`, then a negative sign, then `*` `/` `//` `%`, then `+` `-`, left to right within a row. Parentheses make any order certain.
- `int()`, `float()`, and `str()` make new values of another type. Python is **dynamically typed**: a name can be rebound to a value of any type.
- Floats are stored approximately, so `0.1 + 0.2` shows a tiny error. Calculate with full values and `round()` only for display.

Practice

Do these in your `day02` folder. Type each program yourself, run it, and only then open the answer. Change the starting values and run them again to test your program.

Question 14 Short answer

Fahrenheit to Celsius. Write a script that stores a temperature in Fahrenheit in a variable, converts it to Celsius with the formula $C = (F - 32) \times 5 \div 9$, and prints both, with the Celsius value rounded to one decimal place. Try 72 and 212.

Question 15 Short answer

Seconds to hours, minutes, and seconds. Store a number of seconds in a variable, such as 7384. Use `//` and `%` to work out the hours, the minutes, and the seconds left over, and print them on one line.

Question 16 Short answer

Compound interest. A savings account starts with a **principal** (the starting amount) and grows by a fixed **rate** each year, with each year's interest added to the balance. After `years` years the balance is $\text{principal} \times (1 + \text{rate})$ to the power `years`. Store a principal of 1000, a rate of 0.05 (5%), and 10 years in variables; print the final amount and the interest earned, both rounded to two decimal places.

Review questions

Question 17 Multiple choice

What are the value and type of `7 // 2 * 2.0 + 1`?

- A. `7.0`, a float
- B. `8.0`, a float
- C. `7`, an int
- D. `9.0`, a float

Question 18 Tracing

This program makes change for 87 cents using quarters (25 cents) and dimes (10 cents). What does it print? Track `coins`, `quarters`, and `dimes`.

`coins.py`

```
1 coins = 87
2 quarters = coins // 25
3 coins = coins % 25
4 dimes = coins // 10
5 coins = coins % 10
6 print(quarters, dimes, coins)
```

Question 19 Debugging

This program should print the area of a 2.5 by 4 metre floor. It stops with an error. Quote the last line of the error, explain it, and write the fix.

`floor_area.py`

```
1 length_text = "2.5"
2 width_text = "4"
3 area = int(length_text) * int(width_text)
4 print("Area:", area)
```

Question 20 Debugging

Lena wants to print the average of the scores 80 and 90. Her program has two mistakes: one stops it before it runs, and one would give the wrong answer even after the first is fixed. Find both and write the fixed program.

`average_two_mistakes.py`

```
1 average score = 80 + 90 / 2
2 print(average score)
```

Go further

The official Python Tutorial's introduction shows the same operators, plus a few extra details, in its Numbers subsection: [An Informal Introduction to Python: Numbers](#).

Reflection

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

- ☐ I can create a variable and trace how its value changes. [Reread: 2.1 Variables: Giving a Value a Name](#)
- ☐ I can tell whether a name is legal and write it in `snake_case`. [Reread: 2.2 Naming Rules and snake_case](#)
- ☐ I can predict whether a result is an int or a float. [Reread: 2.3 Integers and Floats](#)
- ☐ I can use `//` and `%` to split a quantity into whole groups and a remainder. [Reread: 2.4 Floor Division, Remainders, and Powers](#)
- ☐ I can evaluate an expression in precedence order. [Reread: 2.5 Operator Precedence](#)

- ☐ I can use True, False, and None correctly. Reread: 2.6 True, False, and None
- ☐ I can convert values with int(), float(), and str(). Reread: 2.7 Converting Between Types
- ☐ I can explain how a name can refer to values of different types. Reread: 2.8 Dynamic Typing: Names Can Change Type
- ☐ I can explain why $0.1 + 0.2$ is not exactly 0.3 and round a result for display. Reread: 2.9 Floats Are Close, Not Exact

Check Your Reasoning

Answers and explanations for every question, numbered to match.

Question 1

2 7

Line	Right side worked out	apples	pears
1	6	6	
2	apples + 4 is 6 + 4	10	
3	apples - 3 is 10 - 3	10	7
4	2	2	7

Line 4 changes `apples` to 2, but `pears` keeps 7. Line 3 used the value `apples` had **then**; changing `apples` later does not go back and change `pears`. If you wrote `2 - 1`, you recalculated `pears` from the new `apples`.

Reread *2.1 Variables: Giving a Value a Name* if your output differed.

Question 2

Line 2 is written backwards. In an assignment the **name** must be on the left and the value on the right. `20 = height` asks Python to attach the number 20 to a value, which makes no sense: a number typed into the code (a **literal**, like a string literal) cannot be given a new value. Swap the two sides:

reversed_assignment_fixed.py

```
1 width = 8
2 height = 20
3 print(width * height)
```

160

Because it is a `SyntaxError`, nothing ran before the fix, not even line 1. The message's suggestion about `==` refers to comparing two values, which Day 4 teaches; it is not the fix here.

Reread *2.1 Variables: Giving a Value a Name* if you missed it.

Question 3

B. `final_score` uses only lowercase letters and an underscore, and does not start with a digit.

- **A** is wrong: the space makes Python see two names, a `SyntaxError`.
- **C** is wrong: a name cannot start with a digit.
- **D** is wrong: Python reads the hyphen as minus, so it means `final` minus `score`, which cannot be assigned to.

Reread *2.2 Naming Rules and snake_case* if you chose differently.

Question 4

B. `/` always gives a float, even when the division comes out even, so the answer is `2.0`.

- **A** is wrong: it assumes an even division gives an int. With `/` it never does.
- **C** is wrong: a float always shows a decimal point, so the float 2 appears as `2.0`.
- **D** is wrong: a number with a decimal point is a float, never an int.

Reread 2.3 *Integers and Floats* if you chose differently.

Question 5

```
>>> 3 + 4.0
7.0
>>> type(9 - 2)
<class 'int'>
>>> type(9 / 3)
<class 'float'>
```

- `3 + 4.0` mixes an int and a float, so the result is the float `7.0`.
- `9 - 2` is two ints with `-`, so the result, `7`, is an int.
- `9 / 3` uses `/`, so the result, `3.0`, is a float even though 9 divides evenly by 3.

Reread 2.3 *Integers and Floats* if any answer differed.

Question 6

```
>>> 17 // 5
3
>>> 17 % 5
2
>>> 17 / 5
3.4
>>> 2 ** 5
32
```

- `17 // 5`: 5 fits into 17 three whole times, so **3**.
- `17 % 5`: three 5s make 15, and `17 - 15` leaves **2**.
- `17 / 5`: ordinary division, always a float: **3.4**.
- `2 ** 5`: $2 \times 2 \times 2 \times 2 \times 2$ is **32**.

Reread 2.4 *Floor Division, Remainders, and Powers* if any answer differed.

Question 7

```
>>> 7 + 8 // 3 * 2
11
>>> (7 + 8) // 3 * 2
10
>>> 2 * 3 ** 2
18
```

- `7 + 8 // 3 * 2`: `//` and `*` share a row, so left to right: `8 // 3` is 2, then `2 * 2` is 4, then `7 + 4` is **11**.
- `(7 + 8) // 3 * 2`: parentheses first, 15; then `15 // 3` is 5; then `5 * 2` is **10**.
- `2 * 3 ** 2`: `**` first, 9; then `2 * 9` is **18**. (Not 36: Python does not multiply 2×3 first.)

Reread 2.5 *Operator Precedence* if any answer differed.

Question 8

B. The parentheses make -5 one number, which is then squared: $(-5) \times (-5) = 25$.

- **A** is wrong: `**` comes before the negative sign, so it is $-(5 ** 2)$, which is -25.
- **C** is wrong: `**` first gives 25, then $25 // 2$ is 12, then $10 + 12$ is 22.
- **D** is wrong: `*` before `+` gives $5 + 10$, which is 15. Only $(5 + 5) * 2$ would add first, giving 20.

Reread 2.5 *Operator Precedence* if you chose differently.

Question 9

B. `False` is one of the two bool values, written with a capital F and no quotes.

- **A** is wrong: the quotes make it a string, five letters of text.
- **C** is wrong: lowercase `false` is not a value Python knows, so using it is a `NameError`.
- **D** is wrong: `None` means "no value", and its type is `NoneType`, not `bool`.

Reread 2.6 *True, False, and None* if you chose differently.

Question 10

```
>>> int(7.99)
7
>>> float("10")
10.0
>>> str(5 + 5)
'10'
>>> int("8") * 2
16
```

- `int(7.99)` drops the decimal part: **7** (not 8; it does not round).
- `float("10")` makes a float, so it shows a decimal point: **10.0**.
- `str(5 + 5)` works out 10 first, then makes the string `'10'`. The REPL shows the quotes because it is a string.
- `int("8") * 2` converts the text to the int 8, then multiplies: **16**.

Reread 2.7 *Converting Between Types* if any answer differed.

Question 11

```
TypeError: unsupported operand type(s) for -: 'int' and 'str'
```

`age_text` holds the string `"15"`, and `-` cannot subtract a string from an int. Convert the text to an int first:

years_left_fixed.py

```
1 age_text = "15"
2 years_left = 18 - int(age_text)
3 print("Years until 18:", years_left)
```

```
Years until 18: 3
```

`int(age_text)` makes the int 15, so line 2 works out $18 - 15 = 3$. `age_text` itself is still a string.

Reread 2.7 *Converting Between Types* if you missed it.

Question 12

```
2.0 <class 'float'>
2.0 <class 'str'>
```

Line	Worked out	value	Type
1	7	7	int
2	7 * 2	14	int
3	14 / 7, / is float	2.0	float
5	str(2.0)	"2.0"	str

Both lines print 2.0, because `print` shows a string without quotes. Only `type()` reveals that the second one is text.

Reread *2.8 Dynamic Typing: Names Can Change Type* if your output differed.

Question 13

```
>>> round(7 / 3, 2)
2.33
>>> round(1.1 * 3, 1)
3.3
```

- `7 / 3` is 2.333..., and two decimal places give **2.33**.
- `1.1 * 3` is 3.3000000000000003 (a float error), and one decimal place gives **3.3**.

Reread *2.9 Floats Are Close, Not Exact* if any answer differed.

Question 14

One possible solution:

fahrenheit_to_celsius.py

```
1 # Convert a temperature from Fahrenheit to Celsius
2 fahrenheit = 72
3 celsius = (fahrenheit - 32) * 5 / 9
4 print(fahrenheit, "degrees Fahrenheit is", round(celsius, 1), "degrees Celsius")
```

```
72 degrees Fahrenheit is 22.2 degrees Celsius
```

The parentheses make Python subtract 32 **before** multiplying; without them, `fahrenheit - 32 * 5 / 9` would multiply first and give a wrong answer. The `/` makes `celsius` a float, 22.22222222222222, so `round(celsius, 1)` shows 22.2. With 212, the program prints 100.0.

Reread *2.5 Operator Precedence* and *2.9 Floats Are Close, Not Exact* if your result differed.

Question 15

One possible solution:

seconds_breakdown.py

```

1 # Split a number of seconds into hours, minutes and seconds
2 total_seconds = 7384
3 hours = total_seconds // 3600
4 seconds_left = total_seconds % 3600
5 minutes = seconds_left // 60
6 seconds = seconds_left % 60
7 print(total_seconds, "seconds is", hours, "hours,", minutes, "minutes and", seconds,
  "seconds")

```

7384 seconds is 2 hours, 3 minutes and 4 seconds

Line	Expression	Worked out	Value
3	total_seconds // 3600	3600 seconds per hour	2
4	total_seconds % 3600	7384 - 7200	184
5	seconds_left // 60	whole minutes in 184	3
6	seconds_left % 60	184 - 180	4

Check: $2 \times 3600 + 3 \times 60 + 4 = 7384$. Every value is an int because `//` and `%` of two ints give ints. If you used `/`, you got decimals like 2.0511....

Reread 2.4 *Floor Division, Remainders, and Powers* if your result differed.

Question 16

One possible solution:

compound_interest.py

```

1 # Compound interest: the balance grows by the rate once per year
2 principal = 1000
3 rate = 0.05
4 years = 10
5 amount = principal * (1 + rate) ** years
6 interest = amount - principal
7 print("Final amount:", round(amount, 2))
8 print("Interest earned:", round(interest, 2))

```

Final amount: 1628.89
Interest earned: 628.89

`(1 + rate) ** years` needs the parentheses: `**` comes before `+`, so `1 + rate ** years` would raise only 0.05 to the 10th power. Then the balance is multiplied by 1000, giving 1628.894626777442, a float. The interest is that amount minus the principal. Both are rounded only when printed, so no accuracy is lost along the way.

Reread 2.5 *Operator Precedence* and 2.9 *Floats Are Close, Not Exact* if your result differed.

Question 17

A. `//` and `*` share a row, so left to right: `7 // 2` is 3; `3 * 2.0` is 6.0, a float because 2.0 is a float; then `6.0 + 1` is 7.0, still a float.

- **B** is wrong: it treats `//` as `/`, giving $3.5 \times 2.0 + 1 = 8.0$.

- **C** is wrong: the value is right, but `2.0` is a float, so the result is a float.
- **D** is wrong: it adds `2.0 + 1` first, but `+` comes after `*`.

Reread *2.5 Operator Precedence* and *2.3 Integers and Floats* if you chose differently.

Question 18

```
3 1 2
```

Line and work	coins	quarters	dimes
1: 87	87		
2: 87 // 25	87	3	
3: 87 % 25	12	3	
4: 12 // 10	12	3	1
5: 12 % 10	2	3	1

Line 3 rebinds `coins` to what is left after the quarters, $87 - 75 = 12$, and line 4 uses that new value. The answer is 3 quarters, 1 dime, and 2 cents left. If you got `3 8 7`, you used 87 instead of 12 on lines 4 and 5, forgetting that line 3 rebound `coins`.

Reread *2.1 Variables: Giving a Value a Name* and *2.4 Floor Division, Remainders, and Powers* if your output differed.

Question 19

```
ValueError: invalid literal for int() with base 10: '2.5'
```

`int()` accepts only text that is a whole number, and `"2.5"` has a decimal point. (Even `int(2.5)` on the float would give 2, losing half a metre.) Use `float()` to keep the decimal part:

`floor_area_fixed.py`

```
1 length_text = "2.5"
2 width_text = "4"
3 area = float(length_text) * float(width_text)
4 print("Area:", area)
```

```
Area: 10.0
```

2.5×4.0 is 10.0, a float because both inputs are floats.

Reread *2.7 Converting Between Types* if you missed it.

Question 20

The first mistake is the space in the name `average score`, which Python reads as two names side by side:

```
SyntaxError: invalid syntax
```

Renaming it to `average_score` fixes the syntax, but the program would then print 125.0, because `/` comes before `+`: Python works out $90 / 2 = 45.0$ and adds 80. The second fix is parentheses around the sum:

average_two_mistakes_fixed.py

```
1 average_score = (80 + 90) / 2
2 print(average_score)
```

```
85.0
```

The first mistake was found before anything ran, because it breaks the naming rules. The second gave no error at all; only knowing the right answer (85) reveals it.

Reread *2.2 Naming Rules and snake_case* and *2.5 Operator Precedence* if you missed either one.