

Day 1 Setup and your first program

Self-Study Guide · About 60 minutes

By the end you will be able to

- Install Python, check its version from the terminal, and use the REPL as a calculator.
- Create a .py file and run it from the terminal.
- Use print, comments, and top-to-bottom order to write a short program.
- Read a SyntaxError or NameError and explain what it is pointing at.

What to remember from earlier

This is the first day, so you need no programming experience. You only need everyday computer skills: saving a file into a folder, finding that folder again, and typing exactly what you see. If you have ever followed instructions like "type this into the box and press Enter," you are ready.

1.1 Installing Python and Opening a Terminal

A computer does exactly what it is told, one instruction at a time, and nothing more. A **program** is a list of instructions written for a computer, and a **programming language** is a language with strict rules for writing those instructions. This course teaches the programming language **Python**, one of the most popular languages in the world because its programs are short and read almost like English.

Your computer does not understand Python on its own. It needs a program called the **Python interpreter**, which reads Python instructions and carries them out. "Installing Python" means installing the interpreter. Today you will also install a **code editor**, an app for writing program files. Any plain text editor works, but a code editor colors the parts of your code and catches some mistakes as you type.

Install the tools

1. Go to python.org, open **Downloads**, and download the newest Python 3 for your computer (Windows or macOS). Any version 3.12 or newer works for this course. On Linux, Python is usually installed already.
2. Run the installer. On Windows, if the installer shows a checkbox to add Python to PATH, tick it; that lets the terminal find Python.
3. Install VS Code from code.visualstudio.com, open it, and install the **Python** extension from the Extensions panel.

Check that it worked

To talk to the interpreter you use a **terminal**: a window where you type a command, press Enter, and read the reply as text. Every computer has one (Terminal on macOS and Linux, PowerShell or Terminal on Windows), and VS Code has one built in: choose **Terminal → New Terminal**.

Type this command into the terminal and press Enter:

```
python --version
```

The command has two parts: `python` names the program to run, and `--version` asks it only to report which version it is. If Python is installed, the terminal answers with one line that starts with `Python 3.` followed by the rest of the version number. Version numbers change often, so yours will not match anyone else's exactly; what matters is that it starts with `Python 3` and is 3.12 or newer.

KEY IDEA

The Python interpreter is the program that runs Python code. Typing `python --version` in a terminal checks that it is installed and shows which version you have.

Now vary one thing: the computer. On macOS and Linux the command is often `python3` instead of `python`, and on Windows `py` sometimes works when `python` does not. Each is the same interpreter under a different name:

```
python3 --version
py --version
```

Use whichever one answers with `Python 3`, and use that same word everywhere this course says `python`.

WATCH OUT

"Command not found" or "Python 2." If the terminal says it does not recognize `python`, the interpreter is either not installed or the terminal cannot find it. Try `python3` (macOS, Linux) or `py` (Windows). If none of them work on Windows, run the installer again and tick the PATH checkbox. Then **close the terminal and open a new one**: a terminal that was already open does not notice new installs. If the reply starts with `Python 2`, that is an old version that some computers keep for other programs; use `python3` instead.

PAUSE AND TRY

Question 1 Short answer

Sam types `python --version` and the terminal says the command was not found. Name two things Sam should try, and explain why each might help.

1.2 The REPL: Python as a Calculator

You have Python installed. What is the fastest way to try one small instruction and see what Python does with it?

Python has an interactive mode for exactly this. Type `python` (with nothing after it) in the terminal and press Enter. The terminal changes: a few lines about the version appear, then a new prompt, `>>>`. That prompt means "Python is listening; type one instruction." This mode is called the **REPL**, short for **Read, Evaluate, Print, Loop**: Python **reads** what you type, **evaluates** it (works out its value), **prints** the result, and **loops** back to `>>>` for the next one.

The simplest thing to type is an **expression**: a piece of code that Python can work out to a value, such as `2 + 3`. Python uses `+` to add, `-` to subtract, `*` (an asterisk) to multiply, and `/` to divide.

WORKED EXAMPLE Using the REPL as a calculator

Here is a REPL session. Lines that start with `>>>` are what you type (without typing the `>>>` itself); the line under each one is Python's reply.

```
>>> 2 + 3
5
>>> 10 - 4
6
>>> 6 * 7
42
>>> 7 / 2
3.5
>>> 8 / 2
4.0
```

Walk through it the way the REPL does:

You type	Python works out	Python prints
<code>2 + 3</code>	add 2 and 3	5
<code>10 - 4</code>	subtract 4 from 10	6
<code>6 * 7</code>	multiply 6 by 7	42
<code>7 / 2</code>	divide 7 by 2	3.5
<code>8 / 2</code>	divide 8 by 2	4.0

Notice the last line: dividing with `/` always gives an answer with a decimal point, even when it comes out even. Python keeps whole numbers and decimal numbers as two different kinds of value, and Day 2 explains why. For today, read `4.0` as "4".

Now vary one thing: the order of the operations. Python follows the same order of operations as math class: multiplication and division happen before addition and subtraction, and parentheses go first.

```
>>> 2 + 3 * 4
14
>>> (2 + 3) * 4
20
```

In `2 + 3 * 4`, Python multiplies `3 * 4` first to get 12, then adds 2 to get 14. In `(2 + 3) * 4`, the parentheses make Python add first to get 5, then multiply by 4 to get 20. The numbers are the same; only the order changed, and so did the answer.

To leave the REPL, type `exit()` and press Enter. The `>>>` prompt disappears and you are back at the ordinary terminal.

KEY IDEA

In the REPL, type an expression after `>>>` and Python prints its value right away. It is the quickest way to test an idea, and you leave it with `exit()`.

WATCH OUT

Typing a terminal command at `>>>`. The terminal and the REPL look alike, but they understand different things. The terminal understands commands like `python --version`; the REPL understands only Python. Typing a terminal command into the REPL causes an error:

```
>>> python --version
Traceback (most recent call last):
...
NameError: name 'python' is not defined
```

The REPL prints a few more lines where `...` appears here. You will learn to read this kind of message in 1.9; for now, notice that the REPL did not understand. The fix is to type `exit()` first, then run the command at the ordinary terminal prompt. The reverse mistake also happens: Python code typed at the ordinary terminal prompt gets an error from the terminal, not an answer from Python. Look for `>>>` before typing Python.

PAUSE AND TRY

Question 2 Tracing

Predict exactly what the REPL prints for each line. Then type them into the REPL to check.

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

1.3 Writing and Running a Script

The REPL forgets everything when you close it. Real programs need to be saved, run again tomorrow, and shared with a friend. For that you write the instructions into a file.

A **script** is a file of Python code that the interpreter runs from the first line to the last. Python script files end in `.py`. You write the file in your editor, save it, and then ask the interpreter to run it from the terminal.

WORKED EXAMPLE Your first script

1. Make a folder named `day01` somewhere easy to find, such as your Documents folder.
2. In VS Code, choose **File** → **Open Folder** and open `day01`.
3. Choose **File** → **New File**, name it `hello.py`, and type these two lines:

`hello.py`

```
1 print("Hello, world!")
2 print("I am learning Python.")
```

1. Save the file, then open the terminal with **Terminal** → **New Terminal**. It starts inside `day01`, the folder you opened.
2. Run the script by typing `python` followed by the file name:

```
python hello.py
```

The interpreter opens `hello.py`, runs line 1, then line 2, and the terminal shows:

```
Hello, world!
I am learning Python.
```

You have written and run your first program. Every later day works the same way: edit the file, save, run it again with the same command.

Scripts and the REPL do not behave the same in one important way. Vary one thing: put a bare expression in a script, as you would in the REPL.

`silent_sum.py`

```
1 print("Adding 2 and 3:")
2 2 + 3
```

```
Adding 2 and 3:
```

Line 2 is never shown. The interpreter still works out `2 + 3`, but a script only shows what you explicitly tell it to show; the automatic "print the value" step belongs to the REPL. To see the answer in a script, ask for it with `print`:

`printed_sum.py`

```
1 print("Adding 2 and 3:")
2 print(2 + 3)
```

```
Adding 2 and 3:
5
```

Question	REPL	Script
How do I start it?	<code>python</code>	<code>python name.py</code>
Is the code saved?	No, it is gone when you exit	Yes, in the <code>.py</code> file
Values shown by itself?	Yes, after every expression	No, only what you <code>print</code>
Best for	Trying one small idea	A whole program you will run again

KEY IDEA

A script is a saved `.py` file that you run with `python name.py`. Unlike the REPL, a script shows only what you print.

WATCH OUT

Running the file from the wrong place. If the terminal is not in the folder that holds `hello.py`, the interpreter reports that it can't open the file, because it looks for `hello.py` in the terminal's current folder. Open the `day01` folder in VS Code and use its terminal, so they always match. Also check that you **saved** the file: the interpreter runs the version on disk, not the unsaved text in your editor. And run `python hello.py` at the ordinary terminal prompt, not at `>>>`, where it is a Python error.

PAUSE AND TRY**Question 3** Tracing

This file is saved as `predict_script.py` and run with `python predict_script.py`. What does the terminal show?

`predict_script.py`

```
1 print("Start")
2 4 * 5
3 print(4 * 5)
```

Question 4 Multiple choice

Which job is the REPL better suited for than a script?

- A. A program you want to run again next week
- B. Checking what `15 * 4` gives before you use it
- C. A program you want to send to a friend
- D. A program with forty lines

1.4 Printing Text and Numbers

Scripts only show what you print, so `print` is the most important tool of the day. It is worth knowing exactly how it treats text and numbers.

`print` is a **function**: a named instruction that does one job when you use it. Using a function is called **calling** it, and you call a function by writing its name followed by parentheses: `print(...)`. The value inside the parentheses is the **argument**, the thing you hand to the function to work on. `print` displays its argument in the terminal and then moves to a new line.

Text in Python goes inside double quotes, like `"Hello"`. A piece of text in quotes is called a **string**. (Single quotes work too; Day 3 covers strings in depth.) Numbers are written without quotes.

The quotes change everything. Python treats whatever is inside quotes as plain text to copy exactly, and it works out anything outside quotes as an expression.

WORKED EXAMPLE Quotes or no quotes

`print_basics.py`

```
1 print("Total cost:")
2 print(4 + 5)
3 print("4 + 5")
```

```
Total cost:
9
4 + 5
```

Line	Argument	What Python does	Printed
1	"Total cost:"	It is a string, so copy the text inside the quotes	Total cost:
2	4 + 5	No quotes, so work out the expression: 9	9
3	"4 + 5"	It is a string, so copy the characters exactly	4 + 5

Lines 2 and 3 differ only by the quotes, and that is the whole difference between doing the math and showing the math.

Now vary the number of arguments. `print` can take several, separated by commas; it prints them on one line with one space between each. And `print()` with nothing inside prints an empty line.

`print_commas.py`

```
1 print("Tickets:", 3 * 12)
2 print()
3 print("Thanks for visiting!")
```

```
Tickets: 36
```

```
Thanks for visiting!
```

Line 1 has two arguments: the string `"Tickets:"` and the expression `3 * 12`. Python works out 36, then prints the text, a space, and the number. Line 2 prints a blank line, and line 3 prints the last string.

KEY IDEA

`print` displays its arguments on one line, separated by spaces, then moves to the next line. Text in quotes is printed exactly as written; an expression without quotes is worked out first.

WATCH OUT

Putting quotes around math you want worked out. `print("6 * 7")` prints `6 * 7`, not `42`, because the quotes turn it into text. Remove the quotes when you want the answer. The opposite mistake, leaving the quotes off words, causes an error you will meet in 1.9.

PAUSE AND TRY**Question 5** Multiple choice

Which line prints `7`?

- A. `print("3 + 4")`
- B. `print(3 + 4)`
- C. `print("3", "+", "4")`
- D. `print(34)`

Question 6 Tracing

What does this line print?

price_line.py

```
1 print("Price:", 2 * 5, "dollars")
```

1.5 Python Reads Top to Bottom

Your scripts so far have had a few lines each, and their output appeared in the same order as the lines. That is not a coincidence, and it is the rule every Python program follows.

Each line of instructions in a program is a **statement**: one complete instruction, such as one call to `print`. When you run a script, the interpreter starts at line 1 and carries out the statements **one at a time, from top to bottom**. It finishes each statement before starting the next, and it never skips ahead or rearranges them to make more sense.

WORKED EXAMPLE Following the order

morning_steps.py

```
1 print("Wake up")
2 print("Eat breakfast")
3 print("Walk to school")
```

Step	Line run	Terminal so far
1	1	Wake up
2	2	Wake up, Eat breakfast
3	3	Wake up, Eat breakfast, Walk to school

```
Wake up
Eat breakfast
Walk to school
```

Now change one thing: move the last line to the top. The statements are the same, but the order is not:

morning_steps_swapped.py

```
1 print("Walk to school")
2 print("Wake up")
3 print("Eat breakfast")
```

```
Walk to school
Wake up
Eat breakfast
```

Python ran the lines in exactly the order you wrote them, even though walking to school before waking up makes no sense. Python cannot know what you meant; it follows the order on the page.

KEY IDEA

A script's statements run one at a time, from the top line to the bottom line, in exactly the order written.

WATCH OUT

"Python will know what I meant." A program can run without any error message and still do the wrong thing, as `morning_steps_swapped.py` did. No error does not mean correct. Before you run a program, write down what it should print; afterward, compare.

PAUSE AND TRY**Question 7** Short answer

This program should count down 3, 2, 1, then print `Liftoff!`. Run it, compare with what it should do, and say which change fixes it.

countdown_wrong.py

```
1 print(3)
2 print(1)
3 print(2)
4 print("Liftoff!")
```

1.6 Comments: Notes Python Ignores

Next week you will open today's files and wonder what they were for. Programmers solve this by leaving notes in the code itself, written for people, not for Python.

A **comment** starts with `#` (the hash sign) and runs to the end of that line. The interpreter ignores everything from the `#` onward. A comment can fill a whole line, or follow code on the same line, where the code before the `#` still runs.

WORKED EXAMPLE Comments in a receipt program

receipt_comments.py

```
1 # Print a shopping receipt
2 print("Apples:", 3 * 2)
3 print("Bread:", 4) # one loaf
4 # print("Milk:", 2)
5 print("Done")
```

```
Apples: 6
Bread: 4
Done
```

Line	What the interpreter does
1	Whole line is a comment: skipped
2	Prints Apples: 6
3	Prints Bread: 4; the comment # one loaf after the code is ignored
4	Starts with #, so the whole line is a comment, even though it looks like code
5	Prints Done

Line 4 shows a common trick called **commenting out**: putting `#` in front of a line of code turns it off without deleting it. Remove the `#` and the line runs again.

Now vary where the `#` sits: inside quotes, it is just a character in the text, not the start of a comment.

hash_in_quotes.py

```
1 print("Room #12") # the # before 12 is part of the text
```

```
Room #12
```

The first # is inside the string, so it prints. The second # is outside the quotes, so the rest of the line is a comment.

KEY IDEA

A # outside quotes starts a comment, and Python ignores the rest of that line. Comments explain code to people; commenting out a line turns it off.

WATCH OUT

Commenting the wrong part of a line. A # at the start of a line turns off everything on it. Students sometimes put # in front of a `print` to add a note and are surprised when that line stops printing. Put notes on their own line, or after the code.

PAUSE AND TRY**Question 8** Tracing

What does this program print?

comment_trace.py

```
1 # Team scores
2 print("Red team:", 10 + 5)
3 # print("Blue team:", 8)
4 print("Green team:", 7) # print("Gold team:", 9)
```

1.7 Indentation Is Part of the Syntax

So far, every line you have seen starts at the very left edge. That is not just tidiness. In most writing, extra spaces at the start of a line are harmless. In Python they carry meaning.

Every language has rules for how sentences must be built; "dog the ran" breaks English grammar. The rules for how code must be written are the language's **syntax**. **Indentation** means the spaces at the start of a line, and in Python indentation is part of the syntax: it shows which lines belong

together in a group. You will build those groups on Day 4. Until then, every statement belongs at the left edge, so **any** spaces before a statement break the rules.

There is one more fact you need: before running anything, the interpreter reads the **whole file** to check its syntax. If it finds a broken rule anywhere, it reports the error and runs **nothing**, not even the lines above the mistake.

WORKED EXAMPLE One stray indent stops everything

unexpected_indent.py

```
1 print("Line one")
2     print("Line two")
```

Line 2 begins with four spaces. Running it prints no Line one at all, only this:

```
File ".../day01/unexpected_indent.py", line 2
    print("Line two")
IndentationError: unexpected indent
```

The first line names the file (yours shows the full path to your own folder) and the line, 2. The middle line shows that line of code. The last line names the problem: an `IndentationError`, and specifically an **unexpected indent**, meaning spaces where Python did not expect any. Delete the four spaces and both lines print.

Now vary where the extra spaces go. Blank lines and spaces **inside** a line do not count as indentation:

spacing_inside_lines.py

```
1 print("Start")
2
3 print(2+3)
4 print(2  +  3)
```

```
Start
5
5
```

The blank line is skipped, and `2+3` and `2 + 3` mean the same as `2 + 3`. Only spaces at the **start** of a line are part of the syntax. (Most Python programmers write one space on each side of `+`, as in `2 + 3`, because it is easiest to read.)

KEY IDEA

In Python, spaces at the start of a line are part of the syntax. Until you learn to group lines, every statement starts at the left edge; a stray indent is an `IndentationError` and nothing in the file runs.

WATCH OUT

Invisible spaces. An accidental space or Tab at the start of a line is hard to see, so the error can seem to come from nowhere. When you see `unexpected indent`, go to the line number it gives, put the cursor at the very start of the line, and delete until the code touches the left edge.

PAUSE AND TRY**Question 9** Multiple choice

What happens when this program runs?

`indent_question.py`

```
1 print("Welcome")
2 print("Today we learn Python")
3   print("Let's begin")
4 print("Goodbye")
```

- A. It prints `Welcome` and `Today we learn Python`, then reports an error.
- B. It reports `IndentationError: unexpected indent` for line 3 and prints nothing.
- C. It prints all four lines; the extra spaces are ignored.
- D. It prints lines 1, 2, and 4 and skips line 3.

1.8 Reading a `SyntaxError`

Everyone who writes code makes mistakes, every day. The difference between a beginner who gets stuck and one who keeps going is reading the error message instead of panicking. Python's messages tell you where to look and what is wrong.

An **error message** is the report the interpreter prints when it cannot continue. A `SyntaxError` is the general name for a broken syntax rule: a missing parenthesis, a missing quote, a missing comma. The `IndentationError` from 1.7 is one special kind of syntax error. Like it, every `SyntaxError` is found **before** the program runs, so nothing prints.

WORKED EXAMPLE A parenthesis that was never closed

`never_closed.py`

```
1 print("Hello")
2 print("World"
3 print("Bye")
```

```
File ".../day01/never_closed.py", line 2
    print("World"
          ^
```

`SyntaxError: '(' was never closed`

Read it in three parts:

1. **Where:** line 2 of `never_closed.py`.

2. **Exactly where:** the caret ^ under the code points at the (after print.
3. **What:** '(' was never closed. Every (needs a matching), and this one has none.

Add) to the end of line 2 and all three lines print. Notice that Hello did not print first: the error was found while Python checked the file, before anything ran.

Now vary the mistake: leave off a closing quote instead.

unclosed_quote.py

```
1 print("Hello)
2 print("Bye")
```

```
File ".../day01/unclosed_quote.py", line 1
```

```
    print("Hello)
```

```
        ^
```

```
SyntaxError: unterminated string literal (detected at line 1)
```

A **string literal** is a string typed directly into the code, like "Hello". **Unterminated** means it was never ended: the opening " has no closing " on that line, so Python thinks the) is part of the text. The caret points at the quote that was opened. The fix is print("Hello").

Mistake	Example	Last line of the message
Missing)	print("Hi"	SyntaxError: '(' was never closed
Missing closing quote	print("Hi)	SyntaxError: unterminated string literal ...
Stray space at the start	Spaces before print("Hi")	IndentationError: unexpected indent

Older versions of Python may word these messages differently, but they always give the line number and the kind of error.

KEY IDEA

A `SyntaxError` means the code breaks Python's rules. Python finds it before running anything; read the line number, the caret, and the last line to find and fix it.

WATCH OUT

Fixing the line the message names, when the mistake is just before it. Usually the line number points straight at the mistake. Occasionally Python only notices a problem a little later, so if the named line looks perfect, check the line above it too.

PAUSE AND TRY

Question 10 Debugging

This program reports an error and prints nothing. Find the mistake, quote the error message's last line, and write the fix.

debug_syntax.py

```

1 print("Quiz time!")
2 print("Question 1)
3 print("Good luck!")

```

1.9 Reading a NameError

Some mistakes follow every syntax rule, so Python happily starts running the program, and only discovers the problem when it reaches the broken line.

Python knows a set of built-in **names**, such as `print`. When a line uses a name Python does not know, it stops at that line with a `NameError`. Because the file's syntax was fine, the program really did start, so every line **before** the mistake has already run.

WORKED EXAMPLE A misspelled function name

misspelled_print.py

```

1 print("Ready")
2 pront("Go")
3 print("Done")

```

Ready

Traceback (most recent call last):

```

File ".../day01/misspelled_print.py", line 2, in <module>
    pront("Go")
    ^^^^^

```

NameError: name 'pront' is not defined. Did you mean: 'print'?

The report Python prints when a running program stops is called a **traceback**, because it traces back to the line where things went wrong. Read it from the top:

Part	Meaning
Ready	Normal output: line 1 ran before the error
Traceback (most recent call last):	Python is about to show where the running program stopped
File "...", line 2, in <module>	The file and line; <module> just means "in the main part of your file"
pront("Go") and ^^^^^	The line itself, with carets under the name Python did not recognize
NameError: name 'pront' is not defined	The kind of error and its cause

Did you mean: 'print'?

A suggestion, when Python spots a known name that looks similar

Line 3 never ran: once a program stops with an error, the statements after it are skipped. Fix the spelling and all three lines print.

Now vary the mistake: spell `print` correctly but forget the quotes around a word.

missing_quotes.py

```
1 print("Ready")
2 print(Hello)
```

```
Ready
Traceback (most recent call last):
  File ".../day01/missing_quotes.py", line 2, in <module>
    print(Hello)
        ^^^^^
NameError: name 'Hello' is not defined
```

Without quotes, `Hello` is not text; Python treats it as a name and looks it up. It knows no name `Hello`, so it stops. The fix is `print("Hello")`. Python is also **case-sensitive**, meaning capital and small letters count as different: `Print` is a different name from `print`, and causes the same kind of error.

Question	SyntaxError	NameError
When is it found?	Before anything runs	When Python reaches that line
Does earlier output appear?	No, nothing prints	Yes, earlier lines have already run
Common causes	Missing <code>)</code> , quote, comma; stray indent	Misspelled name, missing quotes, wrong capital

As with syntax errors, older versions of Python may word these messages a little differently, and some print no "Did you mean" suggestion.

KEY IDEA

A `NameError` means Python reached a line using a name it does not know. Lines before it have already run; read the last line of the traceback and the carets to find the unknown name.

WATCH OUT

Reading only the first line of the message. The most useful line of a traceback is the **last** one, which says what went wrong, and the line just above the carets, which shows where. Students who stop reading at `Traceback (most recent call last):` miss both. Read error messages from the bottom up.

PAUSE AND TRY**Question 11** Tracing

Write everything this program shows, including the last line of the error message.

name_trace.py

```
1 print("Step 1")
2 print("Step 2:", 5 * 2)
3 print(Finished)
4 print("Step 3")
```

Question 12 Multiple choice

A program's first two lines are correct `print` statements. Line 3 has a mistake. In which case do the first two lines still print?

- A. Line 3 is missing its closing `)`.
- B. Line 3 starts with two spaces.
- C. Line 3 is `Print("Done")`.
- D. Line 3 is missing its closing quote.

Putting it together

Maya is making a sign for her school's bake sale. Follow her through three runs; each stage uses a different part of today's lesson.

Attempt 1 (1.6, 1.8). Maya starts with a comment saying what the program is for, then the sign's lines:

bake_sale_attempt1.py

```
1 # Bake sale sign
2 print("BAKE SALE TODAY")
3 print("Muffins left:", 30 - 12
4 print("Cookies left:", 48 - 20)
5 print("All money goes to the library")
```

```
File ".../day01/bake_sale_attempt1.py", line 3
    print("Muffins left:", 30 - 12
      ^
```

SyntaxError: '(' was never closed

Nothing prints, not even `BAKE SALE TODAY`, because a syntax error stops the whole file before it runs. The caret points at line 3's `(`, which has no `)`. Maya adds the `)`.

Attempt 2 (1.5, 1.9). Now the syntax is fine, so the program starts running from the top:

bake_sale_attempt2.py

```
1 # Bake sale sign
2 print("BAKE SALE TODAY")
3 print("Muffins left:", 30 - 12)
4 prnt("Cookies left:", 48 - 20)
5 print("All money goes to the library")
```

```
BAKE SALE TODAY
Muffins left: 18
Traceback (most recent call last):
  File ".../day01/bake_sale_attempt2.py", line 4, in <module>
    prnt("Cookies left:", 48 - 20)
    ^^^^^
NameError: name 'prnt' is not defined. Did you mean: 'print'?
```

Lines 2 and 3 ran (line 1 is a comment), then line 4 stopped the program. Reading the message from the bottom up, Maya sees the misspelling and Python's suggestion.

Attempt 3 (1.4). With `print` spelled correctly, every line runs:

bake_sale.py

```
1 # Bake sale sign
2 print("BAKE SALE TODAY")
3 print("Muffins left:", 30 - 12)
4 print("Cookies left:", 48 - 20)
5 print("All money goes to the library")
```

```
BAKE SALE TODAY
Muffins left: 18
Cookies left: 28
All money goes to the library
```

Attempt	Kind of problem	Found	What printed
1	<code>SyntaxError</code>	Before running	Nothing
2	<code>NameError</code>	While running	The first two lines
3	None	Runs to the end	All four lines

Vocabulary

Term	Definition	Example
program	A list of instructions for a computer	<code>hello.py</code>
Python interpreter	The program that reads and runs Python code	What python starts

code editor	An app for writing program files	VS Code
terminal	A window where you type commands and read text replies	Where you type <code>python hello.py</code>
REPL	Python's interactive mode: read, evaluate, print, loop	The <code>>>></code> prompt
expression	Code that Python works out to a value	<code>2 + 3</code>
script	A saved <code>.py</code> file that Python runs from top to bottom	<code>hello.py</code>
function / calling	A named instruction that does one job / using it, with parentheses	<code>print(...)</code>
argument	A value handed to a function inside its parentheses	<code>"Hi"</code> in <code>print("Hi")</code>
string / string literal	Text in quotes / a string typed directly into the code	<code>"Hello"</code>
statement	One complete instruction	<code>print("Done")</code>
comment / commenting out	Text after <code>#</code> that Python ignores / turning a line off with <code>#</code>	<code># one loaf</code>
syntax	The rules for how code must be written	Every <code>(</code> needs a <code>)</code>
indentation	Spaces at the start of a line; part of Python's syntax	Spaces before <code>print("Hi")</code>
error message / traceback	The interpreter's report when it cannot continue / the report of where a running program stopped	Traceback (most recent call last):
SyntaxError	A broken syntax rule, found before anything runs	<code>'('</code> was never closed
IndentationError	A syntax error caused by wrong indentation	unexpected indent
NameError	A name Python does not know, found when that line runs	name <code>'pront'</code> is not defined
case-sensitive	Capital and small letters count as different	<code>Print</code> is not <code>print</code>

Summary

- The **Python interpreter** runs Python code. `python --version` in a **terminal** checks that it is installed.
- The **REPL** (`>>>`) shows the value of each **expression** at once; a **script** is a saved `.py` file run with `python name.py`, and it shows only what you `print`.
- `print` displays its arguments separated by spaces; text in quotes is copied exactly, and expressions without quotes are worked out first. **Comments** start with `#` and are ignored.

- Statements run one at a time, top to bottom, in the order written. In Python, **indentation** is part of the **syntax**, so for now every statement starts at the left edge.
- A **SyntaxError** (including an **IndentationError**) is found before anything runs, so nothing prints. A **NameError** happens when Python reaches an unknown name, after earlier lines have run. Read error messages from the bottom up.

Practice

Do these in your `day01` folder. Type each program yourself, run it, and only then open the answer.

Question 13 Short answer

Name and address. Write a script that prints your name, then a three-line address, using one `print` per line. (Use a made-up address if you prefer.)

Question 14 Short answer

REPL calculator. Start the REPL and use it as a calculator for five expressions. Use all four operators (+, -, *, /) and at least one pair of parentheses. Predict each answer before you press Enter.

Question 15 Short answer

Break it on purpose. Make three copies of your name-and-address program. Break each one a different way: remove a closing quote, misspell `print`, and add spaces at the start of a line. Run each, and for each one write down the kind of error, the line number, and whether anything printed first.

Review questions

Question 16 Multiple choice

A file `answer.py` contains just one line: `6 * 7`. What appears when you run `python answer.py`?

- A. 42
- B. `6 * 7`
- C. Nothing
- D. `NameError: name 'answer' is not defined`

Question 17 Tracing

Write exactly what this program prints, including any blank line.

`review_trace.py`

```
1 # Movie night plan
2 print("Movie night")
3 print("Snacks:", 3 + 4) # popcorn and juice
4 # print("Cancelled")
5 print()
6 print("Tickets:", 2 * 9)
7 print("2 * 9")
```

Question 18 Debugging

This program should print `Total: 9`. Instead it shows the message below. Use the caret and the message to find the mistake, and write the fix.

`review_missing_comma.py`

```
1 print("Total:" 4 + 5)
```

```
File ".../day01/review_missing_comma.py", line 1
    print("Total:" 4 + 5)
                  ^^^^^^^^^^^^^^^^^^^
```

`SyntaxError: invalid syntax. Perhaps you forgot a comma?`

Question 19 Multiple choice

Program P and program Q each have four lines. They differ only in line 3.

program_p.py

```
1 print("A")
2 print("B")
3 printn("C")
4 print("D")
```

program_q.py

```
1 print("A")
2 print("B")
3 print("C"
4 print("D")
```

Which statement is true?

- A. Both print A and B, then stop with an error.
- B. P prints A and B then stops; Q prints nothing.
- C. P prints nothing; Q prints A and B then stops.
- D. Both print nothing.

Question 20 Short answer

Ravi runs a ten-line script. It prints four lines and then shows a `NameError` for line 5. Ravi says, "Line 5 is the only mistake, so once I fix it the program will work." Is Ravi right? Explain what the error does and does not tell him about lines 1–4 and lines 6–10.

Go further

The official Python Tutorial's first chapters cover why Python is worth learning and how to start the interpreter and the REPL: Whetting Your Appetite and Using the Python Interpreter.

Reflection

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

- ☐ I can check which version of Python is installed from the terminal. Reread: 1.1 Installing Python and Opening a Terminal
- ☐ I can use the REPL as a calculator and leave it. Reread: 1.2 The REPL: Python as a Calculator
- ☐ I can create a .py file and run it from the terminal. Reread: 1.3 Writing and Running a Script
- ☐ I can predict what print shows with and without quotes. Reread: 1.4 Printing Text and Numbers
- ☐ I can trace a program from top to bottom. Reread: 1.5 Python Reads Top to Bottom
- ☐ I can write a comment and comment out a line. Reread: 1.6 Comments: Notes Python Ignores
- ☐ I can explain why a stray indent stops a whole program. Reread: 1.7 Indentation Is Part of the Syntax

- ☐ I can find the mistake a `SyntaxError` is pointing at. Reread: 1.8 Reading a `SyntaxError`
- ☐ I can read a `NameError` and say which lines ran before it. Reread: 1.9 Reading a `NameError`