

# M003. Syntax Essentials — Indentation, statements, blocks

This module gives you a solid, practical grasp of Python's core syntax: how indentation defines blocks, how statements are structured, and how readable programs are composed from these building pieces. You'll learn the exact rules Python enforces (spaces, newlines, and continuation), see idiomatic examples, and practice turning pseudocode into clean, executable Python.


- **Who this is for:** beginners and practitioners who want a precise, hands-on mental model of Python syntax.
- **You'll get:** clear rules for indentation and blocks, statement patterns (simple and compound), line continuation options, and pitfalls to avoid.

## Table of contents

1. Python syntax at a glance
2. Whitespace and indentation rules
3. Blocks (suites): how code groups are formed
4. Statements: simple vs. compound
5. Line structure: newlines, continuation, semicolons
6. Comments and docstrings
7. Keywords and naming conventions
8. Control-flow blocks: if/elif/else, for, while, try/except
9. Function and class blocks
0. Best practices and common pitfalls

## 1. Python syntax at a glance


A tour of core syntax with small, focused examples. Each code block is followed by line-by-line explanations that highlight indentation, statements, and block structure.

 **Interactive Guide**

## 1.1 Module header, comments, and constants

Module docstring, comments, and a constant

python

 Copy

Edit

Reset

```
1  
2     """Utility functions for circle geometry."""  
3     # This comment documents the next line.  
    PI = 3.14159    # Module-level constant
```

Run

Clear

1. **Line 1:** the first string literal in a module is the module docstring.
2. **Line 2:** comments start with `#` and end at the newline.`

3. **Line 3:** a simple assignment statement; inline comments are allowed after code.

## 1.2 Functions and indented blocks (suites)

Defining a function with a nested block python Copy Edit Reset




```
1
2 def circle_area(radius: float) -> float:
3     """Return the area of a circle; raises ValueError for negative radius."""
4     if radius < 0:
5         raise ValueError("radius must be non-negative")
6     area = PI * (radius ** 2)
    return area
```

Run Clear



1. **Line 1:** `def ...:` is a compound statement; the colon introduces an indented suite.
2. **Line 2:** a docstring for the function (first string literal in the block).
3. **Line 3:** an `if` header starts another block nested inside the function.
4. **Line 4:** the `raise` statement is part of the `if` block (same indentation).
5. **Line 5:** a simple assignment inside the function block (aligned with the `if`).

6. **Line 6:** ``return`` exits the function; indentation shows it belongs to the function block, not the ``if``.

### 1.3 Simple vs. compound statements




Multiple simple statements vs. readability python  Copy  Edit  Reset

1  
  
a = 1; b = 2; c = a + b # multiple simple statements on one line (legal, not idiomatic)



 Run  Clear

1. **Line 1:** semicolons can separate simple statements on a single physical line; prefer one statement per line for clarity.

### 1.4 Line structure and continuation

Explicit continuation with backslash (use sparingly) python  Copy  Edit  Reset

1  
  
path = "C:/" "Users/" "name"

 Run  Clear

1. **Lines 1–3:** the trailing `` escapes the newline to continue the statement; avoid any spaces after the backslash.

Implicit continuation inside (), [], {}python

CopyEditReset

1  
2  
3  
4  
5

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

RunClear

1. **Lines 1–5:** brackets allow multi-line literals without a backslash; the block is visually indented for readability.

## 1.5 Control flow blocks and exception suites

for with try/except/else/finallypython

CopyEditReset

1234567891011

```
radii = [3, -1, 5]
results = []
for r in radii:
    try:
        a = circle_area(r)
    except ValueError:
        a = None
    else:
        results.append(a)
```

0%

RunClear

1. **Line 1:** a list literal assigned to a name (simple statement).
2. **Line 2:** initialize an empty list to collect results.
3. **Line 3:** `for` starts a compound statement; the colon introduces the loop block.
4. **Lines 4–11:** the `try` suite contains `except`, optional `else`, and `finally` blocks, each with its own indentation.

5. **Line 6:** catching a specific exception narrows error handling and is preferred over a bare `except:`.

## 1.6 Comprehensions and f-strings

List comprehension with inline conditional and f-stringspythonCopyEditReset

```
1
2
3     report = [
4         f"r={r}: area={circle_area(r) if r >= 0 else 'invalid'}"
5         for r in radii
6     ]
```

RunClear

- Lines 1–4:** comprehension spans multiple lines using brackets for implicit continuation.
- Line 2:** an f-string embeds expressions inside `{...}` for readable formatting.

## 1.7 Multi-line expressions with parentheses

Parentheses enable readable multi-line expressionspythonCopyEditReset

```
1
2
3     total = (
4         sum(x for x in results)
5         if results else 0.0
6     )
```

Run

Clear

1. **Lines 1–4:** parentheses allow implicit continuation across lines; line breaks align with logical parts of the expression.

## 1.8 Printing results (simple statements)

One statement per line is clearer

python

Copy

Edit

Reset

1

2

3

```
print("areas:", results)
print("report:", report)
print("total:", total)
```

Run

Clear

1. **Lines 1–3:** each `print` call is a separate simple statement terminated by the newline.

## 2. Whitespace and indentation rules

Python uses indentation to delimit blocks. Mixing tabs and spaces in indentation is an error. The community convention is 4 spaces per indentation level; be consistent project-wide.

[<> Indentation Guide](#)



Consistent indentation (spaces)

python

Copy

Edit

Reset

```
1
2     def greet(name: str) -> None:
3         if name:
4             print(f"Hello, {name}")
5         else:
            print("Hello")
```

Run

Clear

1. **Line 1:** function header ends with `:`, introducing a suite.
2. **Line 2:** `if` header starts a nested block (4 spaces deeper).
3. **Line 3:** statement inside the `if` block; aligned under the `if`.
4. **Line 4:** `else:` starts a sibling block at the same level as `if`.
5. **Line 5:** statement inside the `else` block.
6. **Note:** use consistent 4-space indentation; never mix tabs and spaces.

### 3. Blocks (suites): how code groups are formed

A block (suite) follows a header that ends with a colon. All statements with the same indentation after the header belong to that block.

<> Blocks Guide

Multiple statements in a suitepython

CopyEditReset

1  
2  
3  
4  
5  
6

x = 10  
if x > 0:  
 y = x \* 2  
 z = y - 1  
 print(z)  
print("done")

RunClear

1. **Line 1:** simple assignment; not part of any block.
2. **Line 2:** `if` header with colon introduces a suite.
3. **Line 3:** first statement in the suite; indented.
4. **Line 4:** second statement in the same suite; aligned with line 3.
5. **Line 5:** third statement in the suite.
6. **Line 6:** this `print` is back at the outer indentation, so it is not in the `if` block.

## 4. Statements: simple vs. compound

Simple statements execute entirely on one logical line. Compound statements contain other statements, introducing suites with colons.

< > Statements Guide

Simple statements

python

Copy

Edit

Reset

1

2

3

a = 1  
b = 2  
total = a + b

Run

Clear

1. **Line 1:** simple assignment; statement ends at newline.
2. **Line 2:** another simple assignment.
3. **Line 3:** expression evaluation and assignment on one logical line.

Compound statements

python

Copy

Edit

Reset

1

2

3

4

5

6

if total > 0:  
 print("positive")  
elif total == 0:  
 print("zero")  
else:  
 print("negative")

Run

Clear

1. **Line 1:** compound header; colon introduces a suite.

2. **Line 2:** first suite body line.
3. **Line 3:** ``elif`` is part of the same compound statement; starts a sibling suite.
4. **Line 4:** body for the ``elif`` suite.
5. **Line 5:** ``else`` starts the final fallback suite.
6. **Line 6:** body of the ``else`` suite.

## 5. Line structure: newlines, continuation, semicolons

A physical line usually equals a logical line. Use implicit continuation inside brackets or parentheses; prefer it over backslashes. Semicolons are legal but non-idiomatic.

<> [Line Structure Guide](#)

**Implicit continuation** python

1

2

3

4

5

```
values = [  
    1,  
    2,  
    3,  
]
```

Run

Clear

1. **Line 1:** opening ``[`` allows the statement to continue across lines.
2. **Lines 2–4:** elements on separate physical lines are valid due to implicit continuation.
3. **Line 5:** closing bracket ends the statement.

Explicit continuation (avoid when possible)pythonCopyEditReset

1msg = "Hello, " + "World"

RunClear

1. **Line 1:** trailing backslash escapes the newline; continuation is required on the next line.
2. **Line 2:** expression continues; avoid spaces after the backslash on previous line.

Semicolons (legal, not idiomatic)pythonCopyEditReset

1a = 1; b = 2; c = a + b

RunClear

1. **Line 1:** three simple statements separated by semicolons; prefer one per line.

## 6. Comments and docstrings

Use `#` for inline comments. Docstrings are the first string literal in a module, class, or function and document the API.

<> [Comments Guide](#)

Inline commentspython

CopyEditReset

```
1
    threshold = 0.5  # probability cutoff
```

Run

Clear

1. **Line 1:** everything after `#` is ignored by the interpreter on that line.

Function docstringpython

CopyEditReset

```
1
2
3     def normalize(x: float) -> float:
4         """Return x clamped to [0.0, 1.0]."""
5         if x < 0.0:
6             return 0.0
7         if x > 1.0:
8             return 1.0
9         return x
```

Run

Clear

1. **Line 1:** function header introduces a suite.
2. **Line 2:** first string literal inside the suite becomes the function docstring.
3. **Lines 3–6:** body statements implement the function logic.

Class docstringpython

CopyEditReset

```
1
2
3 class Counter:
4     """A minimal counter with increment and reset."""
5     def __init__(self) -> None:
6         self.value = 0
7     def inc(self) -> None:
8         self.value += 1
9     def reset(self) -> None:
10        self.value = 0
```

RunClear

1. **Line 1:** class header; colon introduces the class suite.
2. **Line 2:** first string literal in the class suite is the class docstring.
3. **Lines 3–8:** method definitions are nested suites within the class.

## 7. Keywords and naming conventions

Don't use reserved keywords as identifiers (e.g., `if`, `class`, `for`). Follow PEP 8 naming: `snake_case` for variables/functions, `PascalCase` for classes, and `UPPER_CASE` for constants.

[<> Naming Guide](#)

PEP 8-aligned namespythonCopyEditReset

```
1
2     MAX_RETRIES = 3
3
4     def fetch_data(url: str) -> str:
5         return url
6
7     class RetryPolicy:
8         pass
```

RunClear

1. **Line 1:** constant in `'UPPER_CASE'`.
2. **Lines 3–4:** function named with `'snake_case'`.
3. **Lines 6–7:** class named with `'PascalCase'`.

## 8. Control-flow blocks: if/elif/else, for, while, try/except

Control flow statements control the execution order of your code. They include conditional statements, loops, and exception handling.

<> Control Flow Guide



if / elif / else python

 Copy

Edit

Reset

```
1
2
3     def sign(x: int) -> str:
4         if x > 0:
5             return "positive"
6         elif x == 0:
7             return "zero"
8         else:
9             return "negative"
```

Run

Clear

1. **Line 1:** function header; returns a string label.
2. **Line 2:** first branch; positive case.
3. **Line 3:** return ends execution for this branch.
4. **Line 4:** `elif` branch checks equality to zero.
5. **Line 5:** return for zero case.
6. **Lines 6–7:** `else` fallback handles negatives.

## for and while python

 Copy

Edit

Reset

```
1
2   for i in range(3):
3       print(i)
4
5   n = 3
6   while n > 0:
7       n -= 1
```

Run

Clear

1. **Lines 1–2:** `for` loop iterates 0..2; body prints the index.
2. **Line 4:** initialize counter `n`.
3. **Lines 5–6:** `while` loop decrements until the condition becomes false.

## try / except / else / finally python

 Copy

Edit

Reset

```
1
2   try:
3       risky()
4   except ValueError as e:
5       handle(e)
6   else:
7       log("ok")
8   finally:
9       cleanup()
```

Run

Clear

1. **Line 1:** `try` suite introduces protected code.
2. **Line 2:** code that may raise `ValueError`.
3. **Lines 3–4:** handler for `ValueError`.
4. **Lines 5–6:** `else` runs only if no exception occurred.
5. **Lines 7–8:** `finally` always runs, success or failure.

## 9. Function and class blocks

Functions and classes define their own suites. Methods are indented within the class suite.

<> Functions & Classes Guide

Functions

python

Copy

Edit

Reset

1

2

```
def add(a: int, b: int) -> int:
    return a + b
```

Run

Clear

1. **Line 1:** function with type hints; introduces a suite.
2. **Line 2:** single return statement in the function body.

## Classes and methods python

[Copy](#)[Edit](#)[Reset](#)

```
1
2
3 class Point:
4     def __init__(self, x: float, y: float) -> None:
5         self.x = x
6         self.y = y
7     def move(self, dx: float, dy: float) -> None:
8         self.x += dx
9         self.y += dy
```

[Run](#)[Clear](#)

1. **Line 1:** class header defines the type ``Point``.
2. **Lines 2–4:** constructor stores coordinates on ``self``.

3. **Lines 5–7:** method modifies instance state within its suite.

## 10. Best practices and common pitfalls

- **Prefer implicit continuation** inside brackets over backslashes.
- **One statement per line** for readability; avoid semicolons.
- **Consistent 4-space indentation**; never mix tabs and spaces.
- **Use docstrings** for public APIs; keep inline comments concise and useful.
- **Avoid shadowing** built-ins or keywords with variable names.

< > **Best Practices Guide**

### Continue learning

Navigate between modules in the Python Developer journey.

← Previous: Course Overview (LMS)

Next: M004. Numbers — Arithmetic: ints, floats, decimals, fractions →