

PYTHON LESSON

Displaying Output - Advanced

Displaying Output - Advanced

The first Displaying Output lesson covered `print()`, `sep`, `end` and f-strings. This lesson is about the next problem: printing collections (lists, tuples, dictionaries, nested combinations of them) so the result is neat rather than a wall of brackets and quotes. The star of the show is `str.join()`, but it has several companions: unpacking with `*`, `repr`, dynamic field widths, `pprint` and `json.dumps`. The lesson finishes with a critical look at whether a loop like this belongs inline or in a function. As requested, the functions in this lesson are discussed in words only, and none of the examples define one.

COMING FROM JAVA / JAVASCRIPT

`join` lives on the separator in Python, not on the collection. JavaScript has `arr.join(", ")`, which only works on arrays. Java has `String.join(", ", list)`, which is much closer to Python's design. In Python you write `", ".join(items)`, and because it is a *string* method it accepts any iterable: a list, a tuple, a set, dictionary keys, a generator, even another string. The second difference is strictness. JavaScript's `join` quietly converts numbers (and turns `null` into an empty string); Python's raises `TypeError` unless every item is already a string. The third is that Python has no `JSON.stringify` on the global scope: the equivalent is `json.dumps()` from the standard `json` module.

FASTAPI RELEVANCE

A FastAPI endpoint that returns a dict sends it to the client as JSON, and `json.dumps(data, indent=2)` is the quickest way to see in the terminal what that response body will look like. `pprint` is the same idea for a Python-flavoured view (`True` and `None` stay as they are, tuples stay tuples). Knowing the two apart saves confusion when a value changes shape on the way out: tuples become JSON arrays, `True` becomes `true` and `None` becomes `null`.

The Toolbox

Ten tools, most of them one line long. The first three are about turning a collection into text; the rest are about making that text look good.

JOINING

`", ".join(items)`

Called on the *separator*, not the list. It takes any iterable of strings and returns one new string with the separator placed *between* the items, never after the last one.

```
tags = ["red", "green", "blue"]
print(", ".join(tags))       # red, green, blue
print(" ~~~ ".join(tags))    # red ~~~ green ~~~ blue
print("".join(tags))         # redgreenblue
```

JOIN NEEDS STRINGS

`", ".join(str(n) for n in nums)`

Every item must already be a `str`, otherwise `TypeError`. Convert on the way in with a generator expression, or with `map(str, nums)`.

```
nums = [3, 1, 4]
# ", ".join(nums)    # TypeError
print(", ".join(str(n) for n in nums))    # 3, 1, 4
print(", ".join(map(str, nums)))          # 3, 1, 4
```

UNPACKING INTO PRINT()

`print(*items, sep=", ")`

The `*` spreads a collection into separate arguments. `print()` converts each one to a string itself, so numbers need no `str()` at all, and `sep` does the joining.

```
nums = [3, 1, 4]
print(*nums)                 # 3 1 4
print(*nums, sep=", ")       # 3, 1, 4
print(*["a", "b"], sep="\n")  # a, then b on the next line
```

REPR: SHOW THE QUOTES

`f"{value!r}"`

`!r` (or `repr()`) shows a value the way you would type it in code, so `"7"` and `7` st looking identical. Collections already use it for their items.

```
for v in ("7", 7):
    print(f"{v} / {v!r}")      # 7 / '7'  then  7 / 7
print(["a", 1])                # ['a', 1]
```

DYNAMIC FIELD WIDTH

`f"{name:<{width}}"`

A second pair of braces *inside* the format spec lets the width come from a variable, so columns can fit the longest value in the data instead of a hard-coded number.

```
names = ["Ann", "Bartholomew"]
width = len(max(names, key=len))
for n in names:
```

PPRINT

`pprint(data)`

From the standard library. Prints nested lists, tuples and dicts over several indented lines once they get too wide for one. It prints directly and returns `None`.

```
from pprint import pprint
data = {"a": [1, 2, 3], "b": {"c": (4, 5)}}
pprint(data, width=20)
```

```
    print(f"{n:<{width}}|")
# Ann        |
# Bartholomew|
```

```
# {'a': [1, 2, 3],
#  'b': {'c': (4, 5)}}
```

JSON.DUMPS

`json.dumps(data, indent=2)`

Returns a neatly indented string in JSON notation, so wrap it in `print()`. The output looks much like the snippet you started from, but it is real, valid JSON.

```
import json
data = {"a": [1, 2], "ok": True}
print(json.dumps(data, indent=2))
# {
#   "a": [
#     1,
#     2
#   ],
#   "ok": true
# }
```

FILE= AND FLUSH=

`print(msg, file=sys.stderr)`

`print()` has two more keyword arguments. `file` chooses where the text goes (the normal output channel, the error channel, or an open file); `flush=True` pushes it out immediately instead of waiting.

```
import sys
print("Result: 42")                          # normal outpu
print("Low disk space", file=sys.stderr)     # error channe
```

MORE FORMAT SPECS

`f"{n:,.2f}" f"{r:.1%}"`

A comma adds thousands separators, `%` multiplies by 100 and adds the sign, and `+` forces a sign on positive numbers. They combine freely with width and precision.

```
n = 1234567.891
print(f"{n:,.2f}")        # 1,234,567.89
print(f"{0.256:.1%}")     # 25.6%
print(f"{5:+d}")          # +5
```

INDENTING BY DEPTH

`" " * depth + text`

Multiplying a string repeats it, which gives a quick way to indent each line by its nesting level when printing a tree-shaped structure.

```
for depth, name in enumerate(["root", "child", "leaf"]):
    print("  " * depth + name)
# root
#   child
#     leaf
```

🔗 What join() Actually Does

Three facts explain almost everything about `join`.

1. **The string you call it on is the glue.** In `" ~~~ ".join(value)` the separator is `" ~~~ "`, and `value` is the thing being walked through. Python takes the first item, then for each remaining item adds the glue and then the item. The glue goes *between* items only. That is why `join` beats a hand-written loop with `end=", "`, which always leaves a stray separator after the last item.
2. **It returns a new string and prints nothing.** `join` only builds text, so you hand the result to `print()` or place it inside an f-string, exactly as the snippet does with `{" ~~~ ".join(value)}`. Because it builds the whole string in one go, it is also the efficient way to assemble text from many pieces, rather than `+=` inside a loop.
3. **Every item must be a string.** A list of numbers, booleans or nested lists raises `TypeError`. The usual fix is `", ".join(str(item) for item in value)`, or the shorter `", ".join(map(str, value))`. When the items are strings already, as in the snippet, you can pass the collection straight in.

When to reach for it: whenever you have a collection of pieces and want one line of text from them: a comma-separated list for a sentence or a CSV row, a path built from folder names, a multi-line block (`"\n".join(lines)`) printed with a single `print()`, or the contents of a list or tuple dropped into the middle of a label. **When not to:** if you only want the items on one line and they are not all strings, `print(*items, sep=", ")` is shorter and needs no conversion; and if the structure is nested, `pprint` or `json.dumps` will do the whole job for you.

Example 1: join Basics, and the TypeError

Everything above in one script. The three commented-out or converted lines show the number problem and its two fixes. The last line builds a sum expression from the same list.

```python
fruit = ["apple", "pear", "plum"]
nums = [3, 1, 4]

print(", ".join(fruit))
print(" | ".join(fruit))
print("\n".join(fruit))
print("-".join("abc"))        # a string is an iterable of characters
print(", ".join([]))          # an empty list gives an empty string
print(", ".join(["solo"]))    # one item: no separator at all

# print(", ".join(nums))      # TypeError: expected str instance, int found
print(", ".join(str(n) for n in nums))
print(", ".join(map(str, nums)))
print(f"Total: {' + '.join(map(str, nums))} = {sum(nums)}")

# apple, pear, plum
# apple | pear | plum
# apple
# pear
# plum
# a-b-c
#
# solo
# 3, 1, 4
# 3, 1, 4
# Total: 3 + 1 + 4 = 8
```

✳ The Star Alternative: print(*items)

`print()` already accepts any number of arguments, joins them with `sep` and converts each to a string. The `*` operator turns one collection into that many arguments, which gives you `join`'s result without the string conversion step.

Example 2: Unpacking Lists, Ranges and Dictionaries

Note what unpacking a dictionary gives you: the keys. Unpack `.items()` to get (key, value) tuples, and set `sep="\n"` to put each on its own line.

```python
names = ["Ann", "Bo", "Cyd"]
scores = [88, 92, 79]
prices = {"Notebook": 4.5, "Pen": 1.2}

print(*names)
print(*names, sep=", ")
print(*scores, sep=" -> ")
print(*range(1, 6))
print(*prices)                  # a dict unpacks to its keys
print(*prices.items(), sep="\n")  # each item is a (key, value) tuple
print(names)                    # without the star: brackets and quotes

# Ann Bo Cyd
# Ann, Bo, Cyd
# 88 -> 92 -> 79
# 1 2 3 4 5
# Notebook Pen
# ('Notebook', 4.5)
# ('Pen', 1.2)
# ['Ann', 'Bo', 'Cyd']
```

> **Choosing between them:** *use `print(*items, sep=...)` when you are printing straight away and the items might not be strings. Use `join` when you need the text as a value: inside an f-string, in a variable, in a file. The star only works inside a call, so it cannot sit inside an f-string's text.*

🔬 Dissecting the Snippet

Here is the code from the dictionary-iteration lesson again, piece by piece, followed by what it actually prints for a student record.

PIECE	WHAT IT DOES
`for key, value in student_info.items():`	Walks the dictionary one entry at a time, unpacking each (key, value) pair into two variables.
`isinstance(value, list)`	Asks whether this entry's value is a list. The `if / elif / else` chain then picks a different print statement for each type: list, tuple, dict, and everything else.
`f'"{key}" : ...'`	The outer quotes are *single*, so the double quotes inside are ordinary characters that appear in the output (`"name" : ...`). Only what is inside `{ }` is evaluated.
`{" ~~~ ".join(value)}`	An expression inside the braces: join the list's items with `~~~` between them. It uses double quotes inside a single-quoted f-string on purpose (see the gotchas).
`{" ".join(value)}` (tuple branch)	The same idea with a plain space as the glue, so a tuple prints as words separated by spaces.
the dict branch	Prints the key on a line of its own, then a second loop over the inner dictionary, indented by two spaces, one line per inner pair.
the `else` branch	Strings, numbers, booleans: anything that needs no special treatment prints as `"key" : "value"` .
the trailing `,`	A literal comma at the end of every line, imitating how a dictionary looks when written out in code.

Example 3: The Snippet on a Student Record

Same code, new dictionary: a string, an int, a list, a tuple and a nested dict. Notice the age comes out as `"21"`, in quotes, because the `else` branch quotes everything.

```
student_info = {
    "name": "Priya Nair",
    "age": 21,
    "modules": ["Python", "Databases", "Web Design"],
    "grades": ("A", "B", "A"),
    "address": {"city": "Edinburgh", "postcode": "EH1 1AA"},
}

for key, value in student_info.items():
    if isinstance(value, list):
        print(f'"{key}" : {" ~~~ ".join(value)},')
    elif isinstance(value, tuple):
        print(f'"{key}" : {" ".join(value)},')
    elif isinstance(value, dict):
        print(f'"{key}" :')
        for inner_key, inner_value in value.items():
            print(f'  "{inner_key}" : "{inner_value}",')
    else:
        print(f'"{key}" : "{value}",')

# "name" : "Priya Nair",
# "age" : "21",
# "modules" : Python ~~~ Databases ~~~ Web Design,
# "grades" : A B A,
# "address" :
```

```
#   "city" : "Edinburgh",
#   "postcode" : "EH1 1AA",
```

Where the snippet is fragile

1. **A list of numbers crashes it.** If `"modules"` were `[101, 205]`, `" ~~~ ".join(value)` would raise `TypeError`. It works here only because every list and tuple happens to hold strings.
2. **The `else` branch quotes numbers and booleans.** `21` and `True` print as `"21"` and `"True"`, so a number looks like a string.
3. **It goes exactly one level deep.** A list inside the inner dictionary would print in raw bracket form, because the inner loop has no type checks of its own. Handling it means copying the whole `if / elif` chain inside the inner loop, and then again inside *that*.
4. **Every line ends with a comma, including the last.** It looks like JSON but is not valid JSON.

None of this matters for a classroom exercise. All four are exactly the kind of thing that starts to matter when the code has to cope with other dictionaries, which brings us to the reason for the next two examples and the function discussion after them.

♻️ The Same Pattern, Different Data

The structure is identical (`.items()`, an `isinstance` chain, a nested loop) but this version fixes the fragile points: it converts list items with `str()`, merges lists and tuples into one branch, drops the quotes, and aligns the keys in a column.

Example 4: A Library Record With Numbers and a Boolean

`{key:<10}` pads every key to 10 characters so the values line up. `str(item) for item in value` lets the list of ratings through `join`. The single-item tuple is written `("Luciano Ramalho",)`: the trailing comma is what makes it a tuple.

```
library_book = {
    "title": "Fluent Python",
    "year": 2022,
    "available": True,
    "ratings": [5, 4, 5, 3],
    "authors": ("Luciano Ramalho",),
    "location": {"floor": 2, "shelf": "QA76"},
}

for key, value in library_book.items():
    if isinstance(value, (list, tuple)):
        print(f"{key:<10}: {', '.join(str(item) for item in value)}")
    elif isinstance(value, dict):
        print(f"{key:<10}:")
        for inner_key, inner_value in value.items():
            print(f"  {inner_key:<8}= {inner_value}")
    else:
        print(f"{key:<10}: {value}")

# title     : Fluent Python
# year      : 2022
# available : True
# ratings   : 5, 4, 5, 3
# authors   : Luciano Ramalho
# location  :
#   floor   = 2
#   shelf   = QA76
```

🪄 Letting the Standard Library Do It

If the aim is just to look at a nested structure, the loop above is more work than needed. `pprint` *and* `json.dumps` *handle any depth and any mix of types.*

Example 5: print vs pprint vs json.dumps

The same student dictionary three ways. Look for what changes: `pprint` *keeps Python notation, while* `json.dumps` *converts the tuple to a list and, with* `indent=2`*, spreads everything out.*

```
student_info = {
    "name": "Priya Nair",
    "age": 21,
    "modules": ["Python", "Databases", "Web Design"],
    "grades": ("A", "B", "A"),
    "address": {"city": "Edinburgh", "postcode": "EH1 1AA"},
}
import json
from pprint import pprint

print(student_info)
print()
pprint(student_info, width=50, sort_dicts=False)
print()
print(json.dumps(student_info, indent=2))

# {'name': 'Priya Nair', 'age': 21, 'modules': ['Python', 'Databases', 'Web Design'], 'grades': ('A', 'B', 'A'), '
#
# {'name': 'Priya Nair',
#  'age': 21,
#  'modules': ['Python', 'Databases', 'Web Design'],
#  'grades': ('A', 'B', 'A'),
#  'address': {'city': 'Edinburgh',
#              'postcode': 'EH1 1AA'}}
#
# {
#   "name": "Priya Nair",
#   "age": 21,
#   "modules": [
#     "Python",
#     "Databases",
#     "Web Design"
#   ],
#   "grades": [
#     "A",
#     "B",
#     "A"
#   ],
#   "address": {
#     "city": "Edinburgh",
#     "postcode": "EH1 1AA"
#   }
# }
```

Which one? `pprint` *for debugging (it shows exactly what Python holds).* `json.dumps` *when the output is headed for an API, a file or another program. A hand-written loop when you need a specific house style, like the one in the snippet.*

🔪 Case Study: A Dictionary as an Aligned Table

Dynamic field widths and the extra format specs together: the column width comes from the data itself, so adding a longer name later never breaks the layout.

Example 6: Prices Aligned to the Longest Name

`width` is computed once from the longest key, then used inside the format spec as `{name:<{width}}`. The separator line is sized from the same number.

```
prices = {"Notebook": 4.5, "Pen": 1.2, "Highlighter set": 12.99}

width = max(len(name) for name in prices)
total = 0
for name, price in prices.items():
    print(f"{name:<{width}}  £{price:>7.2f}")
    total += price

print("-" * (width + 10))
print(f"{'Total':<{width}}  £{total:>7.2f}")
print(f"{'Items':<{width}}  {len(prices):>8}")

# Notebook         £   4.50
# Pen              £   1.20
# Highlighter set  £  12.99
# --------------------------
# Total            £  18.69
# Items                   3
```

🧩 A Critical Look: Loop Inline, or in a Function?

The idea: instead of repeating the `isinstance` chain, write a function that receives the data, works out each value's type and prints it appropriately. Functions come later in the course, so this section describes the design in words only. Your instinct is sound, but how sound depends on the situation.

What the function would look like

It is worth being precise about the design, because the version you described has a choice in it. You said you would pass over the key and value, then wondered whether passing the whole dictionary is better. **It is, for most purposes.** If the caller passes one pair at a time, the caller still has to write the `for key, value in ... .items()` loop each time; only the type chain has moved. If the caller passes the dictionary, the function owns the loop too, and the calling code collapses to one line per dictionary. Passing a single pair only wins when you want to print a handful of selected entries, or in a different order.

There are two further refinements worth knowing about. A function that **returns the text** instead of printing it is more flexible, because the caller decides whether to print it, save it to a file or check it. And a function that receives a dictionary can, for a nested dictionary, **call itself** on the inner one. That is something an inline loop cannot do: it can only handle as many levels as you physically copy the code for.

FOR A FUNCTION	AGAINST A FUNCTION
The type-testing chain is written **once**. Fixing the quote problem or adding a `set` branch is one edit, not one per place it was pasted.	In a short script that prints one dictionary once, it adds a name, a header and a jump for the reader to follow, for no saving.
The call site reads like a sentence (a descriptive name and one dictionary), so the main program is easier to read.	You are designing for dictionaries you have not seen yet. A function written around one example often needs reworking for the second.
Different dictionaries need no new code: pass any of them.	The name matters. A vague one ends up being worse for readability than the loop it replaced.
It can handle any nesting depth by calling itself, which the inline version cannot do at all.	Python reads a script top to bottom, so a function must be defined *before* the line that calls it. Java does not care about order within a class, and JavaScript hoists function declarations; Python does neither.

FOR A FUNCTION	AGAINST A FUNCTION
It can be tested and debugged in isolation, and moved into its own file and reused across programs.	For plain inspection, the standard library may already be the better function: `pprint` and `json.dumps` are exactly this, written and tested by someone else.

Which is better, and when

SITUATION	BETTER CHOICE	WHY
Small script, one dictionary, printed once	Inline loop	*Nothing is repeated, so there is nothing to save. Everything is visible in one place.*
Learning how loops, `.items()` and `isinstance` work	Inline loop	*The loop is the subject. Hiding it in a function hides the lesson. This is a good reason for your teacher's rule.*
The same block pasted a second time	Either	*A common rule of thumb is to copy once, and turn it into a function the third time.*
Larger program, many different dictionaries	Function (pass the dictionary)	*One definition, many calls, one place to fix bugs.*
Data of unknown shape, such as an API response	A function that calls itself, or `json.dumps`	*Nesting depth cannot be predicted, so a fixed number of copied loops will eventually be wrong.*
Just inspecting data while debugging	`pprint` / `json.dumps`	*Zero code to write, and it handles every type.*
Code others will read or maintain	Function, well named	*The reader learns what it does from the name, and only reads the detail if they need to.*

One more critical point. A growing `if / elif` chain that tests types is, in Python terms, slightly against the grain: Python usually prefers asking an object to do something rather than asking what it is. For display code that is accepted, because the alternative (making each type print itself) is more machinery than the job deserves. If the chain keeps growing past four or five types, though, that is the sign to stop and ask whether `pprint` or `json.dumps` already does it.

📋 Quick Reference — Advanced Output

TASK	SYNTAX	NOTES
Join strings with a separator	`", ".join(items)`	*Call it on the separator. All items must be `str`.*
Join non-strings	`", ".join(map(str, items))`	*Or `str(i) for i in items`.*
Print a collection on one line	`print(*items, sep=", ")`	*No conversion step needed; works only as a call.*
One item per line	`print(*items, sep="\n")`	*Or `print("\n".join(items))` for strings.*
Show the quotes on a value	`f"{x!r}"` / `repr(x)`	*Tells `"7"` from `7`.*
Width from a variable	`f"{x:<{w}}"`	*Nested braces inside the spec.*
Thousands separator / percent	`f"{n:,}"` / `f"{r:.1%}"`	*Combine with width and precision.*
Pretty-print nested data	`pprint(data)`	*`from pprint import pprint`. `sort_dicts=False` keeps your key order.*
JSON text	`print(json.dumps(data, indent=2))`	*Returns a string. `sort_keys=True` sorts the keys.*
Send text to the error channel	`print(msg, file=sys.stderr)`	*`import sys` first. `flush=True` forces immediate output.*
Indent by nesting level	`" " * depth + text`	*String repetition.*

> **⚠ Gotchas for Java / JavaScript Programmers:**
>
> `["a", "b"].join(", ")` **raises** `AttributeError`: *lists have no `join` method. The separator is the object: `", ".join(["a", "b"])`.*
>
> **Numbers are not converted for you**: *where JavaScript's `[1, 2].join()` returns `"1,2"`, Python's `",".join([1, 2])` raises `TypeError`. Convert with `map(str, ...)`, or use `print(*items)`.*

Reusing the outer quote inside an f-string's braces is a `SyntaxError` before Python 3.12: `f"{", ".join(x)}"` *only works on 3.12 or later. That is why the snippet wraps the whole f-string in single quotes: it uses double quotes both for the literal text and inside the expression. On any version, choosing different quote types is the safe habit.*

`print(pprint(data))` prints the data and then `None` : `pprint` *prints on its own and returns nothing. If you want the text as a string, use* `pprint.pformat(data)` *instead.*

`json.dumps` is not a drop-in for `print` : *it returns a string, turns tuples into lists, writes* `True` *as* `true` *and* `None` *as* `null` *, and raises* `TypeError` *for values JSON has no form for, such as a set.*

Printing a collection directly shows `repr` of its items: `print(["a", "b"])` *gives* `['a', 'b']` *with brackets and quotes. That is correct behaviour;* `join` *or* `print(*...)` *is how you ask for something different.*