Prof. Aviv Zohar

Semester A, 2026–7

Linters and formatters

Two tools to run while the agent works



A linter reports. A formatter rewrites the layout.

cart.py — as written

```
1 import os
2
3 def total(items,unit = 'EUR'):
4   sum = 0
5   for x in items:
6     if x == x: sum += x
7   return f'{sum} {unit}'
```

← never used

← hides the built-in sum

← always true

Linter: reports the problems and convention violations its rules select. By default it changes nothing.

A linter reports. A formatter rewrites the layout.

cart.py — as written

```python
import os

def total(items,unit = 'EUR'):
  sum = 0
  for x in items:
    if x == x: sum += x
  return f'{sum} {unit}'
```

formatter



cart.py — formatted

```python
import os


def total(items, unit="EUR"):
    sum = 0
    for x in items:
        if x == x:
            sum += x
    return f"{sum} {unit}"
```

Formatter: one canonical layout, decided for you. The program does exactly what it did.

ⓐ line breaks
ⓑ indentation
ⓒ comma spacing
ⓓ quotes

In a formatted project, the diff shows the change

without a formatter

price.py -3 +3

```
- def price(items,tax = 0.17):
-     net  = total( items )
-     return net*(1+tax)
+ def price(items, tax=0.18):
+     net = total(items)
+     return net * (1 + tax)
```

Asked for: a higher tax rate.
The diff: three lines out, three in.

In a formatted project, the diff shows the change

without a formatter

price.py -3 +3

```
- def price(items,tax =0.17):
-     net  = total( items )
-     return net*(1+tax)
+ def price(items, tax=0.18):
+     net = total(items)
+     return net * (1 + tax)
```

`0.17 → 0.18` changes behaviour. The rest is spacing, and spacing invites skimming.

In a formatted project, the diff shows the change

without a formatter

price.py -3 +3

```
- def price(items,tax =0.17):
-     net  = total( items )
-     return net*(1+tax)
+ def price(items, tax=0.18):
+     net = total(items)
+     return net * (1 + tax)
```

`0.17 → 0.18` changes behaviour. The rest is spacing, and spacing invites skimming.

already formatted

price.py -1 +1

```
- def price(items, tax=0.17):
+ def price(items, tax=0.18):
```

Formatted before the edit, and again after it: the spacing edits are gone. One line out, one in.

In a formatted project, the diff shows the change

without a formatter

price.py -3 +3

```
- def price(items,tax = 0.17):
-     net  = total( items )
-     return net*(1+tax)
+ def price(items, tax=0.18):
+     net = total(items)
+     return net * (1 + tax)
```

already formatted

price.py -1 +1

```
- def price(items, tax=0.17):
+ def price(items, tax=0.18):
```

Layout settled in advance leaves only **the edits that matter**, even in a large diff from an agent.

One command for each job, in each language

	Python	TypeScript with the project's ESLint config
check lint	`$ ruff check .`	`$ npx eslint .`

reports, changes nothing

One command for each job, in each language

	Python	TypeScript with the project's ESLint config
check lint	`$ ruff check .`	`$ npx eslint .`
fix lint, then repair	`$ ruff check --fix .`	`$ npx eslint --fix .`

fixes what it can, reports the rest

One command for each job, in each language

	Python	TypeScript with the project's ESLint config
check lint	`$ ruff check .`	`$ npx eslint .`
fix lint, then repair	`$ ruff check --fix .`	`$ npx eslint --fix .`
format write mode	`$ ruff format .`	`$ npx prettier --write .`

rewrites into one canonical form

One command for each job, in each language

	Python	TypeScript with the project's ESLint config
check lint	`$ ruff check .`	`$ npx eslint .`
fix lint, then repair	`$ ruff check --fix .`	`$ npx eslint --fix .`
format write mode	`$ ruff format .`	`$ npx prettier --write .`
format check check mode	`$ ruff format --check .`	`$ npx prettier --check .`

reports, changes no files

`exit 1` on lint errors, or when the layout needs changing: a program sees the failure.

Ask the agent to run the linter and formatter **before it reports that it is done.**

Read the report, fix the cause, run the check again

billing.py

```
1 import os
2
3
4 def total(prices):
5     return sum(prices)
```

ⓐ the rule and its finding
ⓑ file, line and column
ⓒ the suggested fix

shell

```
~/shop $ ruff check .
F401 [*] `os` imported but unused
 --> billing.py:1:8
  |
1 | import os
  |        ^^
help: Remove unused import: `os`
Found 1 error.
[*] 1 fixable with the --fix option.
```

exit 1

Read the report, fix the cause, run the check again

billing.py — fixed

```
- import os
2
3
4 def total(prices):
5     return sum(prices)
```

[*] Ruff can fix this.

--fix removed the import.

shell

```
~/shop $ ruff check .
F401 [*] `os` imported but unused
 --> billing.py:1:8
  |
1 | import os
  |        ^^
help: Remove unused import: `os`
:
Found 1 error.
[*] 1 fixable with the --fix option.
~/shop $ ruff check --fix .
Found 1 error (1 fixed, 0 remaining).
```

Read the report, fix the cause, run the check again

billing.py — fixed

```
- import os
2
3
4 def total(prices):
5     return sum(prices)
```



shell

```
~/shop $ ruff check .
F401 [*] `os` imported but unused
 --> billing.py:1:8
  |
1 | import os
  |        ^^
help: Remove unused import: `os`
:
Found 1 error.
[*] 1 fixable with the --fix option.
~/shop $ ruff check --fix .
Found 1 error (1 fixed, 0 remaining).
~/shop $ ruff check .
All checks passed!
```

exit 0

A linter can check your project's own convention

CLAUDE.md — instruction file

```
1 ## Database
2 - All database access lives in db/.
3 - Only db/ may import sqlite3.
```

A sentence: it asks the agent to keep the convention.

shop/ the project

report.py	`import sqlite3`
db/__init__.py	`import sqlite3`

A linter can check your project's own convention

CLAUDE.md — instruction file

```
1 ## Database
2 - All database access lives in db/.
3 - Only db/ may import sqlite3.
```

shop/ the project

ⓐ report.py — `import sqlite3` ✗

ⓑ db/__init__.py — `import sqlite3` ✓

pyproject.toml — Ruff: banned-api

```
[tool.ruff.lint]
extend-select = ["TID251"]
[tool.ruff.lint.flake8-tidy-imports.banned-api]
"sqlite3".msg = "Use the db package."          ⓐ
[tool.ruff.lint.per-file-ignores]
"db/*" = ["TID251"]                            ⓑ
```

In ESLint, the same rule is `no-restricted-imports`.

ⓐ bans `sqlite3`
ⓑ except in `db/`
The sentence asks. The rule reports every violation, each time it runs.

A rule can fire on code that is correct

cli.py a command-line tool

```
3 print(f"scanned {n} files")
4 print(f"found {k} issues")
5 print("done")
```

T201
T201
T201

A rule that forbids printing,
in a tool whose job is to print.
All three reported.
All three correct.

A rule can fire on code that is correct

cli.py a command-line tool

```
3 print(f"scanned {n} files")  # noqa: T201
4 print(f"found {k} issues")  # noqa: T201
5 print("done")  # noqa: T201
```

Suppression comments make the reports disappear. These lines are no longer checked for this rule.

A rule can fire on code that is correct

cli.py a command-line tool

```
3 print(f"scanned {n} files")
4 print(f"found {k} issues")
5 print("done")
```

pyproject.toml two lines

```
[tool.ruff.lint.per-file-ignores]
"cli.py" = ["T201"]
```

only `cli.py` may print. Every other file is still checked.

In ESLint: a configuration override.

A rule can fire on code that is correct

cli.py a command-line tool

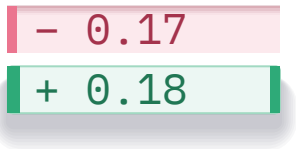
```
3 print(f"scanned {n} files")
4 print(f"found {k} issues")
5 print("done")
```

pyproject.toml two lines

```
[tool.ruff.lint.per-file-ignores]
"cli.py" = ["T201"]
```

A noisy rule leaves **clutter, and less trust in its warnings.**
Start with the standard configuration, and let real work decide.

Recap



Formatter

Keeps your diffs about behaviour: the layout is settled before anyone reads them.



Linter

Carries your project's rules: likely bugs and broken conventions, caught in seconds, fixed by the agent before you read the code.

Start with your project's standard setup.