INF1-IP

Introduction to Programming

Lecture 14 – Code quality 2: Community conventions

Matteo Degiacomi

> Code is read more than it is written

Python code is read **much more often** than it is written, often by its own author.

Today: community conventions

- The Zen of Python and PEPs
- PEP 8, Python's style guide
- Constants
- Pure functions and side effects
- The mutable default trap
- Type hints



An easter egg: import this

A one-line program. Nothing is missing!

```
import this
```

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
...
```

PEP: *Python Enhancement Proposal*, the way the community proposes changes to Python.

e.g., PEP 285 added `bool` in Python 2.3. Before that: `0` and `1`.

PEP 8: the official *Style Guide for Python Code*.

PEP 20: *The Zen of Python*. The principles behind Python's design.

PEP 8: Python's style guide

Conventions we've been using all along:

- `lowercase_with_underscores` for variable and function names
- `CapWords` for class names
- four spaces per indentation level
- lines of at most 79 characters

```
class CamelCase:
    """Very long lines should be avoided where possible and broken up after 79
    characters"""
    lower_case = "a class attribute"
```

> Conventions, not rules

Goal: keeping code **readable** and **consistent**.

Your code still runs if you break them:

- a line longer than 79 characters
- tabs instead of spaces (but **mixing** the two is not allowed)

Good style differs between languages.

"A Foolish Consistency is the Hobgoblin of Little Minds"
(R. W. Emerson, "*Self-Reliance*", 1841, quoted in PEP 8)



Constants

Predict: if the threshold became £100, which line would we change?

```
DISCOUNT_THRESHOLD = 50
DISCOUNT_RATE = 0.10

def apply_discount(total):
    if total >= DISCOUNT_THRESHOLD:
        return total * (1 - DISCOUNT_RATE)
    return total

print(apply_discount(60))
```

→ 54.0

A **constant**: a value we intend to keep unchanged.

Python lets us reassign any name: `CAPITALS` signal the intent.

Keep constants together at the top, after the imports.

> Pure functions and side effects

A function is **pure** if:

- its result depends **only on its arguments** (same arguments, same result), and
- it has **no side effects**: it changes nothing outside itself.

Side effects: printing, writing a file, changing a list passed in as an argument.

```
def pure_total(price, quantity):
    return price * quantity

print(pure_total(2.5, 3))
print(pure_total(2.5, 3))
```

```
→ 7.5   7.5
```

A function that breaks either rule is **impure**. Most programs need some impure functions: **document** their side effects.

An impure function

Predict: we call `with_tax(10)` twice. Do we get the same result?

```
tax_rate = 0.2

def with_tax(price):
    return price * (1 + tax_rate)

print(with_tax(10))
tax_rate = 0.25
print(with_tax(10))
```

```
12.0
12.5
```

`with_tax()` reads `tax_rate`, which is not one of its arguments.

Same argument, different result: **impure**.

Reading `input()` or the current time does the same.

{ Change it, or return a copy?

Predict: which lists contain `"practise"` after each call?

```
def add_task(task_list, task):
    task_list.append(task)

def with_task(task_list, task):
    updated = task_list.copy()
    updated.append(task)
    return updated

original = ["read"]
result = with_task(original, "practise")
print(original, result)
add_task(original, "practise")
print(original, result)
```

```
['read'] ['read', 'practise']
['read', 'practise'] ['read', 'practise']
```

- `add_task()`: changes the caller's list, so it is **impure**
- `with_task()`: returns a new list, so it is **pure**

How would you write their **docstrings**?

{ Try it yourself #1: return, then print

```
def calculate_and_show_mean(numbers):
    result = sum(numbers) / len(numbers)
    print(f"Mean: {result:.1f}")
```

- Split the code above into:
  - `mean(numbers)`, which **returns** the mean
  - `show_mean(numbers)`, which **prints** it
- Add a docstring saying that `mean()` requires a non-empty list
- Test (with assert) a one-item list as well as several values

About 3 minutes

Example solution #1

```python
def mean(numbers):
    """Return the mean of a non-empty list of numbers."""
    return sum(numbers) / len(numbers)

def show_mean(numbers):
    print(f"Mean: {mean(numbers):.1f}")

assert mean([4]) == 4
assert mean([2, 4, 6]) == 4
show_mean([2, 4, 6])
```

```
→ Mean: 4.0
```

`mean()` is pure, so it is easy to test. `show_mean()` is impure: it prints.

! A list as a default

Predict: will `first` and `second` be equal lists, the same list, or different lists?

```
def add_label(label, labels=[]):
    labels.append(label)
    return labels

first = add_label("first")
second = add_label("second")
print(first)
print(second)
```

```
['first', 'second']
['first', 'second']
```

Watch out:

- The default `[]` is made *once*, when `def` is run, not on every call.
- Every call omitting the `labels` argument will refer to that same default list.
- Appending to the default list modifies it. This change will still be there in the next call.

i.e., here appending `"second"` changed what we see through `first`.

The fix: None means "not supplied"

Make a **new** list on each call. **Predict:** what happens to `existing`?

```
def add_label(label, labels=None):
    if labels is None:
        labels = []
    labels.append(label)
    return labels

print(add_label("first"))
print(add_label("second"))

existing = ["existing"]
print(add_label("extra", labels=existing))
print(existing)
```

```
['first']
['second']
['existing', 'extra']
['existing', 'extra']
```

`existing` changes: the caller supplied it. Say so in the docstring.

PEP 8: compare with `None` using `is` (more in Lecture 17).

[Extra] Try it yourself #2: dictionaries too

- Predict what `count_word("cat")`, then `count_word("dog")` return
- Run it, fix it with `None`, and assert that the calls are independent
- Why would `if not counts:` fail for a caller's empty dictionary?

```
def count_word(word, counts={}):
    if word not in counts:
        counts[word] = 0
    counts[word] += 1
    return counts
```

About 3 minutes

[Extra] Example solution #2

```
def count_word(word, counts=None):
    if counts is None:
        counts = {}
    if word not in counts:
        counts[word] = 0
    counts[word] += 1
    return counts

assert count_word("cat") == {"cat": 1}
assert count_word("dog") == {"dog": 1}

supplied = {}
count_word("cat", counts=supplied)
assert supplied == {"cat": 1}
```

No output is good news: every assertion passed.

`if not counts:` is also true for an **empty** dictionary: `supplied` would be swapped for a new one, stay empty, and the last assertion would fail.

Other solutions are possible.

Type hints

`name: type` for each parameter, `-> type` for the result. **Predict:** the second call?

```
def double(n: int) -> int:
    return n * 2

print(double(2))
print(double("ha"))
```

```
4
haha
```

Hints are a **convention**: Python does **not** check or convert the arguments.

Many editors use them to highlight a wrong type.

Note: *type hint* and *type annotation* mean the same thing.

{ Hints for collections, and |

Predict: why does `find_room()` return `str | None`?

```
def mean(numbers: list[float]) -> float:
    """Return the arithmetic mean of a non-empty list."""
    return sum(numbers) / len(numbers)

def find_room(code: str,
              rooms: dict[str, str]) -> str | None:
    if code in rooms:
        return rooms[code]
    return None

print(mean([1.0, 2.5, 4.0]))
print(find_room("IP", {"IP": "G.07"}))
print(find_room("CriSP", {"IP": "G.07"}))
```

```
2.5
G.07
None
```

`|` reads as *or* (Python 3.10+).

Hints + docstring: the function's **contract**.

A hint cannot say *non-empty*: that belongs in the docstring.

Try it yourself #3: annotate a function

- Write `clamp(value, lower=0.0, upper=1.0)` with `float` hints for its parameters and result
- Return the nearest bound if `value` is outside the interval, otherwise `value` itself
- Document that `lower` must not be greater than `upper`
- Extra: Using `assert, test` its behaviour for input values below, at, inside and above the bounds.

About 4 minutes · in pairs

Example solution #3

```python
def clamp(value: float, lower: float = 0.0, upper: float = 1.0) -> float:
    """Limit value to the inclusive bounds. Requires lower <= upper."""
    if value < lower:
        return lower
    if value > upper:
        return upper
    return value

assert clamp(-0.1) == 0.0
assert clamp(0.0) == 0.0
assert clamp(0.5) == 0.5
assert clamp(1.0) == 1.0
assert clamp(1.1) == 1.0
assert clamp(25.0, lower=10.0, upper=20.0) == 20.0
```

> Summary

- PEPs are how the Python community proposes changes; PEP 8 is its style guide
- Conventions, not rules: aim for **readable** and **consistent** code
- Name fixed values as `CONSTANTS`, at the top
- Pure functions are easy to test; **document** side effects
- Avoid mutable defaults (`[]`, `{}`): use `None` and create the object inside
- Type hints document expected types; Python does not check them

Homework: the consolidation challenge at the end of the notebook.