Structural pattern matching in Python: a reconstruction

Thierry Martinez, QAT/SED

Tuesday 19 March, Inria Paris developer meetup

Structural pattern matching in Python

Pattern matching originates from functional languages and has been adopted across various programming languages.

It is a convenient construct for case analysis and data destructuring.

```
match x:
    case []:
        print("Empty list")
    case [Point(x, y)]:
        print(f"Single point: {x}, {y}")
    case [Point(x1, y1), Point(x2, y2)] if y1 = y2:
        print(f"Horizontal line: {x1} - {x2}, {y1}")
    case int(i):
        print(f"Integer: {i}")
    case _:
        print("Something else")
```

Structural pattern matching in Python: history

- **Proposal:** 23 June 2020
  Brandt Bucher *et alii*,
  PEP 622 – Structural Pattern Matching

- **First Specification:** 12 September 2020
  Brandt Bucher, Guido van Rossum,
  PEP 634 – Structural Pattern Matching: Specification
  "*This PEP is a historical document.*"

- **Released:** 4 October 2021, Python 3.10

- **Current documentation (and specification):**
  Python documentation, *¶ 8.6. The match statement*

Motivation for a reconstruction

Pull Request on GitHub project `graphix`:
Refactor measure operator in a new pauli module #122
`https://github.com/TeamGraphix/graphix/pull/122`

```
match a, b:
    case Axis.X, Axis.Y:
        return Plane.XY
    case Axis.Y, Axis.Z:
        return Plane.YZ
    case Axis.X, Axis.Z:
        return Plane.XZ
```

In the pull-request discussion, `@shinich1`, the principal developer, said: *[...] it's good to keep supporting 3.9.*

Is there a refactor tool to **translate `match`-blocks to code compatible with Python prior to 3.10**?

A more fundamental motivation: curiosity!

- I haven't found such a tool: the goal now is to build a custom refactor tool.

- An opportunity to learn about the **general frameworks for building a refactor tool** for Python.

- An opportunity to **dive into the Python pattern-matching specification**.

General Frameworks for Refactoring Python Code

- **Bowler** *(outdated)*:
  - Built on fissix, a backport of `lib2to3` removed from the standard Python library in Python 3.10.
  - Does not support the new PEG-based parser introduced in 3.10.

- **LibCST**, open-source project from *Instagram*:
  - A Concrete Syntax Tree (CST) parser and serializer library for Python.
  - Loss-less parser: keeps all formatting details (comments, whitespaces, parentheses, etc.).
  - Aims to be as convenient as an Abstract Syntax Tree (AST).
  - Functional flavour: mypy-compliant, structures are immutable (functional update with `node.with_changes(key=value)`...

`match_transformer` tool

`https://github.com/thierry-martinez/match_transformer`

- A refactor tool based on LibCST (keeps all formatting details).
- Translate all `match`-blocks into legacy code.
- Passes (almost) all the Python `pattern-matching test-suite`:
  - Dynamic parsing of `match`-blocks (via `eval`) is not supported.
  - Traces are not preserved (tests that use `_trace()` method to track line-numbers in traces are broken).
- No code duplication, **preserves flow-control** (`break`, `continue`, `return`) and **context** (`globals()` and `locals()`).
- Not much room for performance optimisation.

Generated code is mostly readable and can be used in commits

```
# match a, b:
#     case Axis.X, Axis.Y:
#         return Plane.XY
#     case Axis.Y, Axis.Z:
#         return Plane.YZ
#     case Axis.X, Axis.Z:
#         return Plane.XZ
if a == Axis.X and b == Axis.Y:
    return Plane.XY
elif a == Axis.Y and b == Axis.Z:
    return Plane.YZ
elif a == Axis.X and b == Axis.Z:
    return Plane.XZ
```

LibCST and pretty-printing

LibCST is quite convenient for building syntax trees and pretty-printing.

However, we should keep in mind that it is a **Concrete Syntax Tree**, and the validators are not complete (though they are still present).

```
>>> m = cst.Module([])
>>> m.code_for_node(
...     cst.BinaryOperation(
...         cst.Integer("1"), cst.Multiply(),
...         cst.BinaryOperation(
...             cst.Integer("2"), cst.Add(), cst.Integer("3"))))
'1 * 2 + 3'
>>> m.code_for_node(
...     cst.BinaryOperation(
...         cst.Integer("1"), cst.Multiply(),
...         cst.BinaryOperation(
...             cst.Integer("2"), cst.Add(), cst.Integer("3"),
...             lpar=[cst.LeftParen()],
...             rpar=[cst.RightParen()])))
'1 * (2 + 3)'
```

Pattern-matching allows dictionary key functional removal

```
match {"a": 1, "b": 2}:
    case {"a": _, **d}:
        assert d == {"b": 2}
```

Generated code:

```
d = [key: value for key, value in subject.items()
    if key not in {"a"}]
```

Failed bindings are specified irrelevant... but tested

The documentation says:

Note: *During failed pattern matches, some subpatterns may succeed. Do not rely on bindings being made for a failed match. Conversely, do not rely on variables remaining unchanged after a failed match.*

```
def test_patma_042(self):
    x = 2
    y = None
    match x:
        case (0 as z) |
             (1 as z) |
             (2 as z) if
                z == x % 2:
             y = 0
    self.assertEqual(x, 2)
    self.assertIs(y, None)
    self.assertEqual(z, 2)
```

```
test = subject == 0 or
    subject == 1 or
    subject == 2
if test: z = subject
if test and z == x % 2:
    del subject
    del test
    y = 0
else:
    del test
    del subject
```

Proper handling of side-effects

```
def test_patma_081(self):
    x = 0
    match x:
        case 0 if not (x := 1):
            y = 0
        case (0 as z):
            y = 1
    self.assertEqual(x, 1)
    self.assertEqual(y, 1)
    self.assertEqual(z, 0)
```

```
subject = (x)
if subject == 0 and not (x := 1):
    del subject
    y = 0
elif subject == 0:
    z = subject
    del subject
    y = 1
else:        del subject
```

Sequences and mappings: do not believe the specification!

The specification says:

In pattern matching, a sequence is defined as one of the following:

- *a class that inherits from `collections.abc.Sequence`*
- *a Python class that has been registered as collections.abc.Sequence*
- *a builtin class that has its (CPython) Py_TPFLAGS_SEQUENCE bit set*
- *a class that inherits from any of the above*

In practice, `match` only tests for `Py_TPFLAGS_SEQUENCE` and `Py_TPFLAGS_MAPPING`, that are mutually exclusive (but not accessible in pure Python).

```python
class M1(collections.UserDict, collections.abc.Sequence):
    pass
match x:
    case [*_]: # do not match
        return "seq"
    case {}:
        return "map" # it is a map!
```

Use `M1.__mro__` for depth-first search of collection class in ancestors.

Conclusion

- `LibCST` is a convenient library for automating complex Python code refactoring.
- We can use `match`-blocks in projects using Python versions earlier than 3.10 and employ `match_transformer` for their translation.
- Python has some very peculiar corner cases, even in the newly-designed parts of the language!
- It would be interesting to have a tool in the opposite direction, that transforms `if-elif-else` chains into `match`-blocks.