PEP 634: Structural Pattern Matching

Structural pattern matching has been added in the form of a match statement and case statements of patterns with associated actions. Patterns consist of sequences, mappings, primitive data types as well as class instances. Pattern matching enables programs to extract information from complex data types, branch on the structure of data, and apply specific actions based on different forms of data.

Syntax and operations

The generic syntax of pattern matching is:

```
match subject:
    case <pattern_1>:
        <action_1>
    case <pattern_2>:
        <action_2>
    case <pattern_3>:
        <action_3>
    case _:
        <action_wildcard>
```

A match statement takes an expression and compares its value to successive patterns given as one or more case blocks. Specifically, pattern matching operates by:

1. using data with type and shape (the `subject`)
2. evaluating the `subject` in the `match` statement
3. comparing the subject with each pattern in a `case` statement from top to bottom until a match is confirmed.
4. executing the action associated with the pattern of the confirmed match
5. If an exact match is not confirmed, the last case, a wildcard `_`, if provided, will be used as the matching case. If an exact match is not confirmed and a wildcard case does not exist, the entire match block is a no-op.

Declarative approach

Readers may be aware of pattern matching through the simple example of matching a subject (data object) to a literal (pattern) with the switch statement found in C, Java or JavaScript (and many other languages). Often the switch statement is used for comparison of an object/expression with case statements containing literals.

More powerful examples of pattern matching can be found in languages such as Scala and Elixir. With structural pattern matching, the approach is “declarative” and explicitly states the conditions (the patterns) for data to match.

While an “imperative” series of instructions using nested “if” statements could be used to accomplish something similar to structural pattern matching, it is less clear than the “declarative” approach. Instead the “declarative” approach states the conditions to meet for a match and is more readable through its explicit patterns. While structural pattern matching can be used in its simplest form comparing a variable to a literal in a case statement, its true value for Python lies in its handling of the subject’s type and shape.

Simple pattern: match to a literal

Let’s look at this example as pattern matching in its simplest form: a value, the subject, being matched to several literals, the patterns. In the example below, `status` is the subject of the match statement. The patterns are each of the case statements, where literals represent request status codes. The associated action to the case is executed after a match:

```python
def http_error(status):
    match status:
        case 400:
            return "Bad request"
        case 404:
            return "Not found"
        case 418:
            return "I'm a teapot"
        case _:
            return "Something's wrong with the internet"
```

If the above function is passed a `status` of 418, “I’m a teapot” is returned. If the above function is passed a `status` of 500, the case statement with `_` will match as a wildcard, and “Something’s wrong with the internet” is returned. Note the last block: the variable name, `_`, acts as a wildcard and insures the subject will always match. The use of `_` is optional.

You can combine several literals in a single pattern using `|` (“or”):

```python
case 401 | 403 | 404:
    return "Not allowed"
```

Behavior without the wildcard

If we modify the above example by removing the last case block, the example becomes:

```
def http_error(status):
    match status:
        case 400:
            return "Bad request"
        case 404:
            return "Not found"
        case 418:
            return "I'm a teapot"
```
Without the use of _ in a case statement, a match may not exist. If no match exists, the behavior is a no-op. For example, if status of 500 is passed, a no-op occurs.

Patterns with a literal and variable

Patterns can look like unpacking assignments, and a pattern may be used to bind variables. In this example, a data point can be unpacked to its x-coordinate and y-coordinate:

```
# point is an (x, y) tuple
match point:
    case (0, 0):
        print("Origin")
    case (0, y):
        print(f"Y={y}")
    case (x, 0):
        print(f"X={x}")
    case (x, y):
        print(f"X={x}, Y={y}")
    case _:
        raise ValueError("Not a point")
```
The first pattern has two literals, (0, 0), and may be thought of as an extension of the literal pattern shown above. The next two patterns combine a literal and a variable, and the variable binds a value from the subject (point). The fourth pattern captures two values, which makes it conceptually similar to the unpacking assignment (x, y) = point.

Patterns and classes

If you are using classes to structure your data, you can use as a pattern the class name followed by an argument list resembling a constructor. This pattern has the ability to capture class attributes into variables:

```
class Point:
    x: int
    y: int
```

```
def location(point):
    match point:
        case Point(x=0, y=0):
            print("Origin is the point's location.")
        case Point(x=0, y=y):
            print(f"Y={y} and the point is on the y-axis.")
        case Point(x=x, y=0):
            print(f"X={x} and the point is on the x-axis.")
        case Point():
            print("The point is located somewhere else on the plane.")
        case _:
            print("Not a point")
```

Patterns with positional parameters

You can use positional parameters with some builtin classes that provide an ordering for their attributes (e.g. dataclasses). You can also define a specific position for attributes in patterns by setting the `__match_args__` special attribute in your classes. If it's set to ("x", "y"), the following patterns are all equivalent (and all bind the `y` attribute to the `var` variable):

```
Point(1, var)
Point(1, y=var)
Point(x=1, y=var)
Point(y=var, x=1)
```

Nested patterns

Patterns can be arbitrarily nested. For example, if our data is a short list of points, it could be matched like this:

```
match points:
    case []:
        print("No points in the list.")
    case [Point(0, 0)]:
        print("The origin is the only point in the list.")
    case [Point(x, y)]:
        print(f"A single point {x}, {y} is in the list.")
    case [Point(0, y1), Point(0, y2)]:
        print(f"Two points on the Y axis at {y1}, {y2} are in the list.")
    case _:
        print("Something else is found in the list.")
```

Complex patterns and the wildcard

To this point, the examples have used `_` alone in the last case statement. A wildcard can be used in more complex patterns, such as `('error', code, _)`. For example:

```
match test_variable:
    case ('warning', code, 40):
        print("A warning has been received.")
    case ('error', code, _):
        print(f"An error {code} occurred.")
```
In the above case, test_variable will match for (‘error’, code, 100) and (‘error’, code, 800).

Guard

We can add an if clause to a pattern, known as a “guard”. If the guard is false, match goes on to try the next case block. Note that value capture happens before the guard is evaluated:

```
match point:
    case Point(x, y) if x == y:
        print(f"The point is located on the diagonal Y=X at {x}.")
    case Point(x, y):
        print(f"Point is not on the diagonal.")
```
Other Key Features

Several other key features:

- Like unpacking assignments, tuple and list patterns have exactly the same meaning and actually match arbitrary sequences. Technically, the subject must be a sequence. Therefore, an important exception is that patterns don’t match iterators. Also, to prevent a common mistake, sequence patterns don’t match strings.

- Sequence patterns support wildcards: `[x, y, *rest]` and `(x, y, *rest)` work similar to wildcards in unpacking assignments. The name after `*` may also be `_`, so `(x, y, *_)` matches a sequence of at least two items without binding the remaining items.

- Mapping patterns: `{"bandwidth": b, "latency": l}` captures the `"bandwidth"` and `"latency"` values from a dict. Unlike sequence patterns, extra keys are ignored. A wildcard `**rest` is also supported. (But `**_` would be redundant, so is not allowed.)

- Subpatterns may be captured using the `as` keyword:
  ```
  case (Point(x1, y1), Point(x2, y2) as p2): ...
  ```

-

This binds x1, y1, x2, y2 like you would expect without the as clause, and p2 to the entire second item of the subject.

- Most literals are compared by equality. However, the singletons True, False and None are compared by identity.

- Named constants may be used in patterns. These named constants must be dotted names to prevent the constant from being interpreted as a capture variable:

```python
from enum import Enum

class Color(Enum):
    RED = 0
    GREEN = 1
    BLUE = 2

color = Color.GREEN
match color:
    case Color.RED:
        print("I see red!")
    case Color.GREEN:
        print("Grass is green")
    case Color.BLUE:
        print("I'm feeling the blues :(")
```