Search /

Contents

Page Source (GitHub)

PEP 636 – Structural Pattern Matching: Tutorial

Author: Daniel F Moisset <dfmoisset at gmail.com>
Sponsor: Guido van Rossum <guido at python.org>

BDFL-Delegate:
Discussions-To: Python-Dev list
Status: Final
Type: Informational
Created: 12-Sep-2020
Python-Version: 3.10
Post-History: 22-Oct-2020, 08-Feb-2021
Resolution: Python-Committers message

▶ Table of Contents

Abstract

This PEP is a tutorial for the pattern matching introduced by PEP 634.

PEP 622 proposed syntax for pattern matching, which received detailed discussion both from the community and the Steering Council. A frequent concern was about how easy it would be to explain (and learn) this feature. This PEP addresses that concern providing the kind of document which developers could use to learn about pattern matching in Python.

This is considered supporting material for PEP 634 (the technical specification for pattern matching) and PEP 635 (the motivation and rationale for having pattern matching and design considerations).

For readers who are looking more for a quick review than for a tutorial, see Appendix A.

Tutorial

As an example to motivate this tutorial, you will be writing a text adventure. That is a form of interactive fiction where the user enters text commands to interact with a fictional world and receives text descriptions of what happens. Commands will be simplified forms of natural language like `get sword`, `attack dragon`, `go north`, `enter shop` or `buy cheese`.

Matching sequences

Your main loop will need to get input from the user and split it into words, let's say a list of strings like this:

```
command = input("What are you doing next? ")
# analyze the result of command.split()
```

The next step is to interpret the words. Most of our commands will have two words: an action and an object. So you may be tempted to do the following:

```
[action, obj] = command.split()
... # interpret action, obj
```

The problem with that line of code is that it's missing something: what if the user types more or fewer than 2 words? To prevent this problem you can either check the length of the list of words, or capture the `ValueError` that the statement above would raise.

You can use a matching statement instead:

```
match command.split():
    case [action, obj]:
        ... # interpret action, obj
```

The match statement evaluates the **"subject"** (the value after the `match` keyword), and checks it against the **pattern** (the code next to `case`). A pattern is able to do two different things:

- Verify that the subject has certain structure. In your case, the `[action, obj]` pattern matches any sequence of exactly two elements. This is called **matching**
- It will bind some names in the pattern to component elements of your subject. In this case, if the list has two elements, it will bind `action = subject[0]` and `obj = subject[1]`.

If there's a match, the statements inside the case block will be executed with the bound variables. If there's no match, nothing happens and the statement after `match` is executed next.

Note that, in a similar way to unpacking assignments, you can use either parenthesis, brackets, or just comma separation as synonyms. So you could write `case action, obj` or `case (action, obj)` with the same meaning. All forms will match any sequence (for example lists or tuples).

Matching multiple patterns

Even if most commands have the action/object form, you might want to have user commands of different lengths. For example, you might want to add single verbs with no object like `look` or `quit`. A match statement can (and is likely to) have more than one `case`:

```
match command.split():
    case [action]:
        ... # interpret single-verb action
    case [action, obj]:
        ... # interpret action, obj
```

The match statement will check patterns from top to bottom. If the pattern doesn't match the subject, the next pattern will be tried. However, once the *first* matching pattern is found, the body of that case is executed, and all further cases are ignored. This is similar to the way that an `if/elif/elif/...` statement works.

Matching specific values

Your code still needs to look at the specific actions and conditionally execute different logic depending on the specific action (e.g., `quit`, `attack`, or `buy`). You could do that using a chain of `if/elif/elif/...`, or using a dictionary of functions, but here we'll leverage pattern matching to solve that task. Instead of a variable, you can use literal values in patterns (like `"quit"`, `42`, or `None`). This allows you to write:

```
match command.split():
    case ["quit"]:
        print("Goodbye!")
        quit_game()
    case ["look"]:
        current_room.describe()
    case ["get", obj]:
        character.get(obj, current_room)
    case ["go", direction]:
        current_room = current_room.neighbor(direction)
    # The rest of your commands go here
```

A pattern like `["get", obj]` will match only 2-element sequences that have a first element equal to `"get"`. It will also bind `obj = subject[1]`.

As you can see in the `go` case, we also can use different variable names in different patterns.

Literal values are compared with the `==` operator except for the constants `True`, `False` and `None` which are compared with the `is` operator.

Matching multiple values

A player may be able to drop multiple items by using a series of commands `drop key`, `drop sword`, `drop cheese`. This interface might be cumbersome, and you might like to allow dropping multiple items in a single command, like `drop key sword cheese`. In this case you don't know beforehand how many words will be in the command, but you can use extended unpacking in patterns in the same way that they are allowed in assignments:

```
match command.split():
    case ["drop", *objects]:
        for obj in objects:
            character.drop(obj, current_room)
    # The rest of your commands go here
```

This will match any sequences having "drop" as its first elements. All remaining elements will be captured in a `list` object which will be bound to the `objects` variable.

This syntax has similar restrictions as sequence unpacking: you can not have more than one starred name in a pattern.

Adding a wildcard

You may want to print an error message saying that the command wasn't recognized when all the patterns fail. You could use the feature we just learned and write `case [*ignored_words]` as your last pattern. There's however a much simpler way:

```
match command.split():
    case ["quit"]: ... # Code omitted for brevity
    case ["go", direction]: ...
    case ["drop", *objects]: ...
    ... # Other cases
    case _:
        print(f"Sorry, I couldn't understand {command!r}")
```

This special pattern which is written `_` (and called wildcard) always matches but it doesn't bind any variables.

Note that this will match any object, not just sequences. As such, it only makes sense to have it by itself as the last pattern (to prevent errors, Python will stop you from using it before).

Composing patterns

This is a good moment to step back from the examples and understand how the patterns that you have been using are built. Patterns can be nested within each other, and we have been doing that implicitly in the examples above.

There are some "simple" patterns ("simple" here meaning that they do not contain other patterns) that we've seen:

- **Capture patterns** (stand-alone names like `direction`, `action`, `objects`). We never discussed these separately, but used them as part of other patterns.
- **Literal patterns** (string literals, number literals, `True`, `False`, and `None`)
- The **wildcard pattern** `_`

Until now, the only non-simple pattern we have experimented with is the sequence pattern. Each element in a sequence pattern can in fact be any other pattern. This means that you could write a pattern like `["first", (left, right), _, *rest]`. This will match subjects which are a sequence of at least three elements, where the first one is equal to `"first"` and the second one is in turn a sequence of two elements. It will also bind `left=subject[1][0]`, `right=subject[1][1]`, and `rest = subject[3:]`

Or patterns

Going back to the adventure game example, you may find that you'd like to have several patterns resulting in the same outcome. For example, you might want the commands `north` and `go north` to be equivalent. You may also desire to have aliases for `get X`, `pick up X` and `pick X up` for any X.

The `|` symbol in patterns combines them as alternatives. You could for example write:

```
match command.split():
    ... # Other cases
    case ["north"] | ["go", "north"]:
        current_room = current_room.neighbor("north")
    case ["get", obj] | ["pick", "up", obj] | ["pick", obj, "up"]:
        ... # Code for picking up the given object
```

This is called an **or pattern** and will produce the expected result. Patterns are tried from left to right; this may be relevant to know what is bound if more than one alternative matches. An important restriction when writing or patterns is that all alternatives should bind the same variables. So a pattern `[1, x] | [2, y]` is not allowed because it would make unclear which variable would be bound after a successful match. `[1, x] | [2, x]` is perfectly fine and will always bind `x` if successful.

Capturing matched sub-patterns

The first version of our "go" command was written with a `["go", direction]` pattern. The change we did in our last version using the pattern `["north"] | ["go", "north"]` has some benefits but also some drawbacks in comparison: the latest version allows the alias, but also has the direction hardcoded, which will force us to actually have separate patterns for north/south/east/west. This leads to some code duplication, but at the same time we get better input validation, and we will not be getting into that branch if the command entered by the user is `"go figure!"` instead of a direction.

We could try to get the best of both worlds doing the following (I'll omit the aliased version without "go" for brevity):

```
match command.split():
    case ["go", ("north" | "south" | "east" | "west")]:
        current_room = current_room.neighbor(...)
        # how do I know which direction to go?
```

This code is a single branch, and it verifies that the word after "go" is really a direction. But the code moving the player around needs to know which one was chosen and has no way to do so. What we need is a pattern that behaves like the or pattern but at the same time does a capture. We can do so with an **as pattern**:

```
match command.split():
    case ["go", ("north" | "south" | "east" | "west") as direction]:
        current_room = current_room.neighbor(direction)
```

The as-pattern matches whatever pattern is on its left-hand side, but also binds the value to a name.

Adding conditions to patterns

The patterns we have explored above can do some powerful data filtering, but sometimes you may wish for the full power of a boolean expression. Let's say that you would actually like to allow a "go" command only in a restricted set of directions based on the possible exits from the current_room. We can achieve that by adding a **guard** to our case. Guards consist of the `if` keyword followed by any expression:

```
match command.split():
    case ["go", direction] if direction in current_room.exits:
        current_room = current_room.neighbor(direction)
    case ["go", _]:
        print("Sorry, you can't go that way")
```

The guard is not part of the pattern, it's part of the case. It's only checked if the pattern matches, and after all the pattern variables have been bound (that's why the condition can use the `direction` variable in the example above). If the pattern matches and the condition is truthy, the body of the case executes normally. If the pattern matches but the condition is falsy, the match statement proceeds to check the next case as if the pattern hadn't matched (with the possible side-effect of having already bound some variables).

Adding a UI: Matching objects

Your adventure is becoming a success and you have been asked to implement a graphical interface. Your UI toolkit of choice allows you to write an event loop where you can get a new event object by calling `event.get()`. The resulting object can have different type and attributes according to the user action, for example:

- A `KeyPress` object is generated when the user presses a key. It has a `key_name` attribute with the name of the key pressed, and some other attributes regarding modifiers.
- A `Click` object is generated when the user clicks the mouse. It has an attribute `position` with the coordinates of the pointer.
- A `Quit` object is generated when the user clicks on the close button for the game window.

Rather than writing multiple `isinstance()` checks, you can use patterns to recognize different kinds of objects, and also apply patterns to its attributes:

```
match event.get():
    case Click(position=(x, y)):
        handle_click_at(x, y)
    case KeyPress(key_name="Q") | Quit():
        game.quit()
    case KeyPress(key_name="up arrow"):
        game.go_north()
    ...
    case KeyPress():
        pass # Ignore other keystrokes
    case other_event:
        raise ValueError(f"Unrecognized event: {other_event}")
```

A pattern like `Click(position=(x, y))` only matches if the type of the event is a subclass of the `Click` class. It will also require that the event has a `position` attribute that matches the `(x, y)` pattern. If there's a match, the locals `x` and `y` will get the expected values.

A pattern like `KeyPress()`, with no arguments will match any object which is an instance of the `KeyPress` class. Only the attributes you specify in the pattern are matched, and any other attributes are ignored.

Matching positional attributes

The previous section described how to match named attributes when doing an object match. For some objects it could be convenient to describe the matched arguments by position (especially if there are only a few attributes and they have a "standard" ordering). If the classes that you are using are named tuples or dataclasses, you can do that by following the same order that you'd use when constructing an object. For example, if the UI framework above defines their class like this:

```
from dataclasses import dataclass

@dataclass
class Click:
    position: tuple
    button: Button
```

then you can rewrite your match statement above as:

```
match event.get():
    case Click((x, y)):
        handle_click_at(x, y)
```

The `(x, y)` pattern will be automatically matched against the `position` attribute, because the first argument in the pattern corresponds to the first attribute in your dataclass definition.

Other classes don't have a natural ordering of their attributes so you're required to use explicit names in your pattern to match with their attributes. However, it's possible to manually specify the ordering of the attributes allowing positional matching, like in this alternative definition:

```
class Click:
    __match_args__ = ("position", "button")
    def __init__(self, pos, btn):
        self.position = pos
        self.button = btn
        ...
```

The `__match_args__` special attribute defines an explicit order for your attributes that can be used in patterns like `case Click((x,y))`.

Matching against constants and enums

Your pattern above treats all mouse buttons the same, and you have decided that you want to accept left-clicks, and ignore other buttons. While doing so, you notice that the `button` attribute is typed as a `Button` which is an enumeration built with `enum.Enum`. You can in fact match against enumeration values like this:

```
match event.get():
    case Click((x, y), button=Button.LEFT):  # This is a left click
        handle_click_at(x, y)
    case Click():
        pass  # ignore other clicks
```

This will work with any dotted name (like `math.pi`). However an unqualified name (i.e. a bare name with no dots) will be always interpreted as a capture pattern, so avoid that ambiguity by always using qualified constants in patterns.

Going to the cloud: Mappings

You have decided to make an online version of your game. All of your logic will be in a server, and the UI in a client which will communicate using JSON messages. Via the `json` module, those will be mapped to Python dictionaries, lists and other builtin objects.

Our client will receive a list of dictionaries (parsed from JSON) of actions to take, each element looking for example like these:

- `{"text": "The shop keeper says 'Ah! We have Camembert, yes sir'", "color": "blue"}`
- If the client should make a pause `{"sleep": 3}`
- To play a sound `{"sound": "filename.ogg", "format": "ogg"}`

Until now, our patterns have processed sequences, but there are patterns to match mappings based on their present keys. In this case you could use:

```
for action in actions:
    match action:
        case {"text": message, "color": c}:
            ui.set_text_color(c)
            ui.display(message)
        case {"sleep": duration}:
            ui.wait(duration)
        case {"sound": url, "format": "ogg"}:
            ui.play(url)
        case {"sound": _, "format": _}:
            warning("Unsupported audio format")
```

The keys in your mapping pattern need to be literals, but the values can be any pattern. As in sequence patterns, all subpatterns have to match for the general pattern to match.

You can use `**rest` within a mapping pattern to capture additional keys in the subject. Note that if you omit this, extra keys in the subject will be ignored while matching, i.e. the message `{"text": "foo", "color": "red", "style": "bold"}` will match the first pattern in the example above.

Matching builtin classes

The code above could use some validation. Given that messages came from an external source, the types of the field could be wrong, leading to bugs or security issues.

Any class is a valid match target, and that includes built-in classes like `bool` `str` or `int`. That allows us to combine the code above with a class pattern. So instead of writing `{"text": message, "color": c}` we can use `{"text": str() as message, "color": str() as c}` to ensure that `message` and `c` are both strings. For many builtin classes (see PEP 634 for the whole list), you can use a positional parameter as a shorthand, writing `str(c)` rather than `str() as c`. The fully rewritten version looks like this:

```
for action in actions:
    match action:
        case {"text": str(message), "color": str(c)}:
            ui.set_text_color(c)
            ui.display(message)
        case {"sleep": float(duration)}:
            ui.wait(duration)
        case {"sound": str(url), "format": "ogg"}:
            ui.play(url)
        case {"sound": _, "format": _}:
            warning("Unsupported audio format")
```

Appendix A – Quick Intro

A match statement takes an expression and compares its value to successive patterns given as one or more case blocks. This is superficially similar to a switch statement in C, Java or JavaScript (and many other languages), but much more powerful.

The simplest form compares a subject value against one or more literals:

```
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"
```

Note the last block: the "variable name" `_` acts as a *wildcard* and never fails to match.

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

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

Patterns can look like unpacking assignments, and can be used to bind variables:

```
# 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")
```

Study that one carefully! The first pattern has two literals, and can be thought of as an extension of the literal pattern shown above. But 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`.

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

```
from dataclasses import dataclass

@dataclass
class Point:
    x: int
    y: int

def where_is(point):
    match point:
        case Point(x=0, y=0):
            print("Origin")
        case Point(x=0, y=y):
            print(f"Y={y}")
        case Point(x=x, y=0):
            print(f"X={x}")
        case Point():
            print("Somewhere else")
        case _:
            print("Not a point")
```

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)
```

Patterns can be arbitrarily nested. For example, if we have a short list of points, we could match it like this:

```
match points:
    case []:
        print("No points")
    case [Point(0, 0)]:
        print("The origin")
    case [Point(x, y)]:
        print(f"Single point {x}, {y}")
    case [Point(0, y1), Point(0, y2)]:
        print(f"Two on the Y axis at {y1}, {y2}")
    case _:
        print("Something else")
```

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 points:
    case [Point(x, y)] if x == y:
        print("Y = X at {x}")
    case [Point(x, y)]:
        print("Not on the diagonal")
```

Several other key features:

- Like unpacking assignments, tuple and list patterns have exactly the same meaning and actually match arbitrary sequences. An important exception is that they don't match iterators or strings. (Technically, the subject must be an instance of `collections.abc.Sequence`.)
- 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 it is not allowed.)
- Subpatterns may be captured using the `as` keyword:

```
case (Point(x1, y1), Point(x2, y2) as p2): ...
```

- Most literals are compared by equality, however the singletons `True`, `False` and `None` are compared by identity.
- Patterns may use named constants. These must be dotted names to prevent them from being interpreted as capture variable:

```
from enum import Enum
class Color(Enum):
    RED = 0
    GREEN = 1
    BLUE = 2

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 :(")
```

Copyright

This document is placed in the public domain or under the CC0-1.0-Universal license, whichever is more permissive.

Source: https://github.com/python/peps/blob/main/peps/pep-0636.rst

Last modified: 2020-02-01 09:34:37 GMT