Pattern Matching in Python
Learning why Python and patterns are a perfect match

EuroPython 2021
July 29, 2021

Daniel Moisset
Software Engineering Technical Trainer

TechAtBloomberg.com
Hello!

I am Daniel Moisset

I co-authored the Pattern Matching PEP “trilogy” (PEP 634-636)

You can find me at @dmoisset
Pattern Matching comes with the latest Python 3.10b4 (since alpha 6).

Final 3.10 is planned for October 2021.
You’re in a maze of twisty little objects, all alike.
The problem

Working with the “shape” of an object
What is the “shape” of an object?

- Type
- Length (of a sequence)
- Present keys (of a mapping)
- Present attributes (of an instance)
- Exact value

And we can combine and nest these...
PATTERN MATCHING

IS THIS A BOOLEAN EXPRESSION?
**Insight:** you’re checking the shape of your object to extract data from it!

<table>
<thead>
<tr>
<th>Condition</th>
<th>Data extraction</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>isinstance(x, Duck)</code></td>
<td><code>foo(x.quack())</code></td>
</tr>
<tr>
<td><code>len(x) &gt; 2</code></td>
<td><code>foo(x[0], x[1])</code></td>
</tr>
<tr>
<td>&quot;key&quot; in x</td>
<td><code>foo(x[&quot;key&quot;])</code></td>
</tr>
<tr>
<td><code>hasattr(x, &quot;beak&quot;)</code></td>
<td><code>foo(x.beak)</code></td>
</tr>
</tbody>
</table>
Patterns
Describing shape+extraction
We need some syntax to decompose an object and assign into variables...

...wait, we already have something like that:

```python
a, *b, c = [1, 3, 5, 7, 9]
```

Let’s crank this idea up to 11!
Object decomposition mimics object creation:

```
print([a, *b, c])
[1, 3, 5, 7, 9]
```
Can the *pattern* generate the *subject*?

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Subject</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>x</td>
<td>True</td>
</tr>
<tr>
<td>x</td>
<td>[1, 2, 3]</td>
</tr>
<tr>
<td>[x, y]</td>
<td>[1, 2, 3]</td>
</tr>
<tr>
<td>[x, y]</td>
<td>[&quot;foo&quot;, 42]</td>
</tr>
<tr>
<td>int(x)</td>
<td>3.0</td>
</tr>
</tbody>
</table>

Note: The pattern matches the subject in the following cases: 1, True, [1, 2, 3], ["foo", 42]. The pattern does not match in the cases of int(x) = 3.0.
Some simple patterns:

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Matches if</th>
<th>Extracts</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>obj == 123</td>
<td>-</td>
</tr>
<tr>
<td>True</td>
<td>obj is True</td>
<td>-</td>
</tr>
<tr>
<td>os.SEEK_END</td>
<td>obj == os.SEEK_END</td>
<td>-</td>
</tr>
<tr>
<td>x</td>
<td>always!</td>
<td>x = obj</td>
</tr>
<tr>
<td>_</td>
<td>always!</td>
<td>-</td>
</tr>
<tr>
<td>Duck()</td>
<td>isinstance(obj, Duck)</td>
<td>-</td>
</tr>
</tbody>
</table>
Some composite patterns:

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Matches if</th>
<th>Extracts</th>
</tr>
</thead>
<tbody>
<tr>
<td>(123, a)</td>
<td>obj is a sequence</td>
<td>a = obj[1]</td>
</tr>
<tr>
<td></td>
<td>len(obj) == 2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>obj[0] == 123</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[int(), *mid, int()]</td>
<td>obj is a sequence</td>
<td>mid = obj[1:-1]</td>
</tr>
<tr>
<td></td>
<td>len(obj) &gt;= 2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>isinstance(obj[0], int)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>isinstance(obj[-1], int)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>{&quot;name&quot;: n, &quot;age&quot;: a}</td>
<td>obj is a mapping</td>
<td>n = obj[&quot;name&quot;]</td>
</tr>
<tr>
<td></td>
<td>&quot;name&quot; in obj</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&quot;age&quot; in obj</td>
<td>a = obj[&quot;age&quot;]</td>
</tr>
</tbody>
</table>
### Extra syntax: alternatives and as

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Matches if</th>
<th>Extracts</th>
</tr>
</thead>
<tbody>
<tr>
<td>`200</td>
<td>404</td>
<td>500`</td>
</tr>
<tr>
<td>`[bool()</td>
<td>&quot;yes&quot;</td>
<td>&quot;no&quot; as flag, *]`</td>
</tr>
<tr>
<td></td>
<td><code>len(obj) &gt;= 1</code> and <code>isinstance(obj[0], bool) or obj[0] in (&quot;yes&quot;, &quot;no&quot;)</code></td>
<td></td>
</tr>
<tr>
<td>`[x]</td>
<td>{&quot;value&quot;: x}</td>
<td>x`</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>x = obj[0]</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>x = obj[&quot;value&quot;]</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>x = obj</code></td>
</tr>
</tbody>
</table>
## Class patterns

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Matches if</th>
<th>Extracts</th>
</tr>
</thead>
</table>
| Duck(color="green")        | `isinstance(obj, Duck)`
|                             | `hasattr(obj, "color")`                                                  | obj.color == "green" |
|                             |                                                                           |                  |
| Duck(color="green", age=a) | `isinstance(obj, Duck)`
|                             | `hasattr(obj, "color")`                                                  | obj.color == "green"
|                             | `hasattr(obj, "age")`                                                    | a = obj.age       |
|                             |                                                                           |                  |
| Duck(mom=Duck(age=a))       | `isinstance(obj, Duck)`
|                             | `hasattr(obj, "mom")`                                                    | obj.mom.age       |
|                             | `isinstance(obj.mom, Duck)`                                              |                  |
|                             | `hasattr(obj.mom, "age")`                                                |                  |
Putting it all together

The `match/case` statement
Overall syntax

```python
match <subject>:
    case <pattern 1>:
        <code block 1>
    case <pattern 2> if <guard>:
        <code block 2>
    case <pattern 3>:
        <code block 3>
```
Let’s write a “slicer”

Usage: `slicer(seq, [start], stop, [step])`

`slicer(seq, stop) → seq[:stop]`
`slicer(seq, start, stop) → seq[start:stop]`
`slicer(seq, start, stop, step) → seq[start:stop:step]`

Ok, let’s implement:

```python
def slicer(seq, start=0, stop, step=1):
    Hmm... **SyntaxError**: non-default argument follows default argument
```
Pattern match the arguments!

```python
def slicer(seq, *args):
    match args:
        case [stop]: return seq[:stop]
        case [start, stop]: return seq[start:stop]
        case [start, stop, step]: return seq[start:stop:step]
        case _: raise TypeError("invalid arguments")
```

Let's look at the spec again:

- `slicer(seq, stop) → seq[:stop]`
- `slicer(seq, start, stop) → seq[start:stop]`
- `slicer(seq, start, stop, step) → seq[start:stop:step]`

As Bruce Eckel said: “Python is executable pseudocode”
FizzBuzz

for i in range(1, 101):
    match (i % 3 == 0), (i % 5 == 0):
        case True, True: print("Fizzbuzz")
        case True, _: print("Fizz")
        case _, True: print("Buzz")
        case _: print(i)
Matching on API calls

```python
resp = get_user_info(uuid).json
match resp:
    case {"error": msg}:
        raise APIError(msg)
    case {"user": {"name": n, "dob": d}} if is_birthday(d):
        render(f"Happy Birthday {n}! 🎉")
    case {"user": {"name": n}}:
        render(f"Welcome {n}!"")
    case _:
        raise APIError("Unexpected get_user_info response")
```
A thermonuclear “switch” statement

```python
def get_user_info(uuid):
    # Get user information from the server
    resp = get_user_info(uuid).status

match resp.status:
    case 200:
        process_response(resp.json)
    case 403:
        raise InvalidUser
    case 301:
        process_redirect(resp.location)
    case code if 500 <= code <= 599:
        raise ServerError
    case _:
        raise InvalidStatus
```
Other use cases

- Simple parsers
  - See PEP-636 for an example
- Traverse recursive structures (trees!)
  - Example:
    [github.com/dmoisset/patma/blob/rbtree/examples/rbtree.py](https://github.com/dmoisset/patma/blob/rbtree/examples/rbtree.py)
The pattern matching paradigm

Pattern matching as a decomposition approach
An alternative to method dispatch

```python
def area(s: Shape):
    match s:
    case Rectangle(w, h): return w * h
    case Square(side): return side ** 2
    case Circle(r): return math.pi * (r ** 2)
    case _: raise InvalidShape

# versus
class Circle(Shape):
    def area(self):
        return math.pi * self.radius ** 2
```
Pattern matching vs. OOP

- PM comes from functional programming
- "Dumb values and smart functions"
- Comparable to method dispatch
  - Emphasis on adding verbs vs. nouns
- Comparable to visitor pattern
  - But straightforward syntax
  - Additional power
Pattern matching vs. switch statement

- Checking is sequential! (a compiler *could* optimise)
- Switch is usually based on literals
- Match can use constant values, but be careful with semantics ("MY_CONST" pattern vs. "mod.MY_CONST")
Further Reading

- PEP-636: The tutorial
- PEP-635: Motivational text
- PEP-634: The spec
- github.com/gvanrossum/patma/: Some code examples (possibly outdated)
Credits

Special thanks to all the people who made and released these awesome resources for free:

- Presentation template by SlidesCarnival
- Other original images and photographs used with permission from their creators.
Thanks!

You can find me at @dmoisset and dfmoisset@gmail.com
Thank you!

https://www.bloomberg.com/careers

Questions?

TechAtBloomberg.com

© 2021 Bloomberg Finance L.P. All rights reserved.