ORDINA
Ahead of change

INTRODUCING STRUCTURAL PATTERN MATCHING

PEP 634, 635, 636

5 October 2021

Sebastiaan Zeeff

Ordina Pythoneers

PERSONAL INTRODUCTION

- Sebastiaan Zeeff (35)
- Software Engineer for the Ordina Pythoneers
- Python Discord

Larry Hastings commenting on "Gauging sentiment on pattern matching"
-- https://discuss.python.org/t/gauging-sentiment-on-pattern-matching/5770/21

“I see the match statement as a [Domain-Specific Language] contrived to look like Python, and to be used inside of Python, but with very different semantics.

Larry Hastings commenting on "Gauging sentiment on pattern matching"
-- https://discuss.python.org/t/gauging-sentiment-on-pattern-matching/5770/21

“I see the match statement as a [Domain-Specific Language] contrived to look like Python, and to be used inside of Python, but with very different semantics. When you enter a PEP 634 match statement, the rules of the language change completely, and code that looks like existing Python code does something surprisingly very different.”

Larry Hastings commenting on "Gauging sentiment on pattern matching"
-- https://discuss.python.org/t/gauging-sentiment-on-pattern-matching/5770/21

*“I see the match statement as a **[Domain-Specific Language]** contrived to look like Python, and to be used inside of Python, but with very different semantics. When you enter a PEP 634 match statement, the rules of the language change completely, and **code that looks like existing Python code does something** surprisingly very **different**.”*

Larry Hastings commenting on "Gauging sentiment on pattern matching"
-- https://discuss.python.org/t/gauging-sentiment-on-pattern-matching/5770/21

*“I see the match statement as a **[Domain-Specific Language]** contrived to look like Python, and to be used inside of Python, but with very different semantics. When you enter a PEP 634 match statement, the rules of the language change completely, and **code that looks like existing Python code does something** surprisingly very **different**.”*

Larry Hastings commenting on "Gauging sentiment on pattern matching"
-- https://discuss.python.org/t/gauging-sentiment-on-pattern-matching/5770/21

“…but it is Python.”

Jan-Hein Bührman, private correspondence

PART I: MATCHING BY STRUCTURE AND SHAPE

What is Structural Pattern Matching?

- Structural Pattern Matching is not ***just*** *a C-style Switch statement.*

What is Structural Pattern Matching?

```
switch (status_code)
{
    case 200:
        printf("Got an OK response from server!");
        break;
    case 404:
        printf("The resource wasn't found...");
        break;
    default:
        printf("Something unexpected happened!");
        break;
}
```

What is Structural Pattern Matching?

```
match status_code:
    case 200:
        print("Got an OK response from server!")
    case 404:
        print("The resource wasn't found...")
    case _:
        print("Something unexpected happened!")
```

What is Structural Pattern Matching?

```
if status_code == 200:
    print("Got an OK response from server!")
elif status_code == 404:
    print("The resource wasn't found...")
else:
    print("Something unexpected happened!")
```

What is Structural Pattern Matching?

```
match status_code:
    case 200:
        print("Got an OK response from server!")
    case 404:
        print("The resource wasn't found...")
    case _:
        print("Something unexpected happened!")
```

What is Structural Pattern Matching?

- Structural Pattern Matching is not ***just*** *a C-style Switch statement.*
- If it's not *just* matching literals, what ***is*** Structural Pattern Matching?

On Shape and Structure

- *Structural* Pattern Matching tries matching objects by their "*shape*".

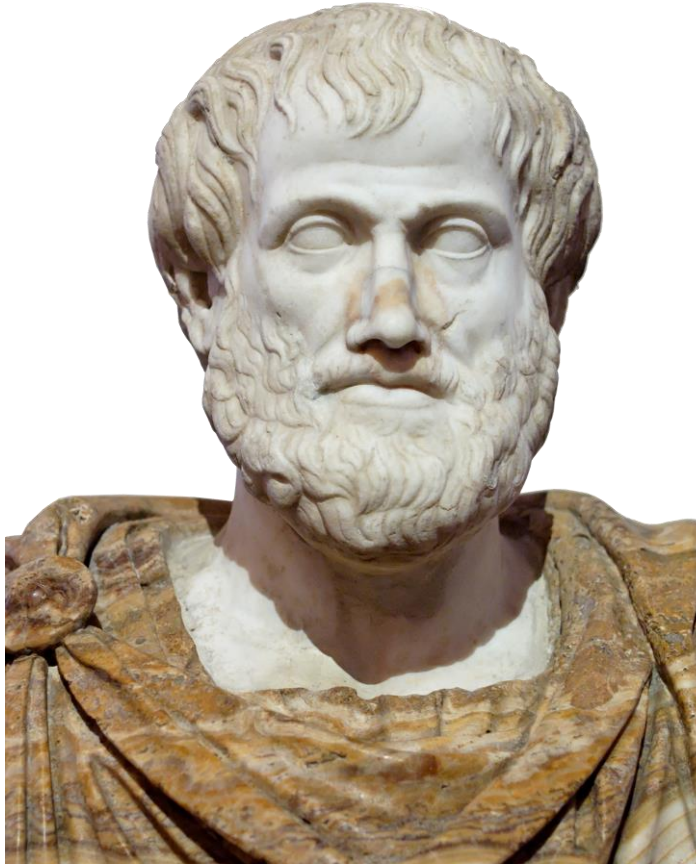
On Shape and Structure

- *Structural* Pattern Matching tries matching objects by their "*shape*".

- What is the shape of an object?

On Shape and Structure

- *Structural* Pattern Matching tries matching objects by their "*shape*".
- What is the shape of an object?



The shape of an object

```
the_answer = 42
game_command = ["MOVE", "NORTH", 10]
my_introduction = {"name": "Sebastiaan", "age": 35}
```

The shape of an object

```
the_answer = 42
game_command = ["MOVE", "NORTH", 10]
my_introduction = {"name": "Sebastiaan", "age": 34}
```

- How would you describe the shape of these objects?

The shape of an object

```
the_answer = 42
game_command = ["MOVE", "NORTH", 10]
my_introduction = {"name": "Sebastiaan", "age": 34}
```

The shape of an object

```
the_answer = 42
game_command = ["MOVE", "NORTH", 10]
my_introduction = {"name": "Sebastiaan", "age": 34}
```

- The value of the object

The shape of an object

```
the_answer = 42
game_command = ["MOVE", "NORTH", 10]
my_introduction = {"name": "Sebastiaan", "age": 34}
```

- The value of the object
- The type of the object

The shape of an object

```
the_answer = 42
game_command = ["MOVE", "NORTH", 10]
my_introduction = {"name": "Sebastiaan", "age": 34}
```

- The value of the object
- The type of the object
- For sequences: Which and/or how many elements does a sequence have?

The shape of an object

```
the_answer = 42
game_command = ["MOVE", "NORTH", 10]
my_introduction = {"name": "Sebastiaan", "age": 34}
```

- The value of the object
- The type of the object
- For sequences: Which and/or how many elements does a sequence have?
- For mappings: Which keys does a mapping have?

The shape of an object

```
the_answer = 42
game_command = ["MOVE", "NORTH", 10]
my_introduction = {"name": "Sebastiaan", "age": 34}
```

- The value of the object
- The type of the object
- For sequences: Which and/or how many elements does a sequence have?
- For mappings: Which keys does a mapping have? What kind of values?

The shape of an object

```
the_answer = 42
game_command = ["MOVE", "NORTH", 10]
my_introduction = {"name": "Sebastiaan", "age": 34}
```

- The value of the object
- The type of the object
- For sequences: Which and/or how many elements does a sequence have?
- For mappings: Which keys does a mapping have? What kind of values?
- For all types: Which attributes does this object have?

Fill in the blanks: A match statement.

```
match <target>:
    case <pattern> [if <guard>]:
        <block of code>
    case <pattern> [if <guard>]:
        <block of code>
    ...
```

Fill in the blanks: A match statement.

```
match <target>:
    case <pattern> [if <guard>]:
        <block of code>
    case <pattern> [if <guard>]:
        <block of code>
    ...
```

```
match my_target:
    case int():
        print("My target is an instance of `int`")
    case str():
        print("My target is an instance of `str`")
```

Fill in the blanks: A match statement.

```
match <target>:
    case <pattern> [if <guard>]:
        <block of code>
    case <pattern> [if <guard>]:
        <block of code>
    ...
```

```
match my_target:
    case int() if my_target > 100:
        print("This integer is larger than 100.")
    case str() if len(my_target) == 3:
        print("This string has a length of 3")
```

ORDINA
Ahead of change

PART II: THE PATTERNS

Wildcard Pattern

```
match my_target:
    case _:
        print("This always matches!")
```

- A wildcard pattern **always** matches.

Wildcard Pattern

```
match my_target:
    case _:
        print("This always matches!")
```

- A wildcard pattern **always** matches.
- The case above is **a tautology**: It always matches.
  - Tautological cases must come last within a match statement.

Wildcard Pattern

```
match my_target:
    case _:
        print("This always matches!")
```

- A wildcard pattern **always** matches.
- The case above is **a tautology**: It always matches.
  - Tautological cases must come last within a match statement.
- Wildcard patterns, like any pattern, can be embedded within other patterns.

Capture Pattern

```
match my_target:
    case some_name:
        print(f"I matched with {some_name!r}!")
```

- Just like a Wildcard Pattern, a Capture Pattern **always** matches.
  - The same rules for tautologies apply here as well!
- It will **capture the matched value** by *binding* a name to it.
  - Note: A Wildcard Pattern **does not** bind the underscore!
- As with all patterns, it can be used nested within another pattern.

Literal Pattern

```
match my_target:
    case 100:
        print("Matched with the int `100`")
    case "hello":
        print("Matched with the str ``'hello'``")
    case True:
        print("Matched with `True`")
```

Matched literals:

- Numbers (incl. complex)
- Strings
- True
- False
- None

- The Literal Pattern matches some types of literals
- Numbers and strings are compared using equality (`a == b`)
- The "singleton literals" are compared by identity (`a is b`)

OR Pattern

```
match my_target:
    case 100 | 200 | 300 | 400:
        print("Look at all those numbers!")
    case "hello" | "goodbye":
        print("Are you leaving so soon?")
    case None | _:
        print("Hey, we've embedded a Wildcard Pattern")
```

- You can combine subpatterns in an OR-pattern using `|` (pipe)
- Each subpattern **must bind the same set of names**

Sequence Pattern

```
player_command = ["MOVE", "NORTH", 10]

match player_command:
    case "MOVE", direction, distance:
        move_player(direction, distance)
```

- This a sequence pattern: subpatterns separated by commas
- It matches instances of collections.abc.Sequence (except str, bytes, bytearray)

Sequence Pattern

```
player_command = ["MOVE", "NORTH", 10]

match player_command:
    case ("MOVE", direction, distance):
        move_player(direction, distance)
```

- You can also wrap a sequence pattern in parentheses for a tuple-look
- It still matches all sequences (except str, bytes, bytearray), not *just* tuples!

Sequence Pattern

```
player_command = ["MOVE", "NORTH", 10]

match player_command:
    case ["MOVE", direction, distance]:
        move_player(direction, distance)
```

- This pattern matches still matches all sequences, not just list!

Sequence Pattern with a Star Pattern

```
player_command = ["MOVE", "NORTH", 10]

match player_command:
    case ["MOVE", direction, distance]:
        move_player(direction, distance)
    case ["SELL", *items]:
        sell_items(items)
```

- You can use a Star Pattern to capture a variable number of values.
- This will bind the name `items` to a list with the captured values.

Sequence Pattern with a Star Pattern

```
player_command = ["MOVE", "NORTH", 10]

match player_command:
    case ["MOVE", direction, distance]:
        move_player(direction, distance)
    case ["SELL", *items]:
        sell_items(items)
    case ["TRAVEL", *intermediate, destination]:
        travel(*intermediate, destination)
```

- Like with unpacking/packing, the star pattern does not have to be at the end

Sequence Pattern with a Star Pattern

```
match player_command:
    case ["FIRST", *_, "LAST"]:
        print("This sequence starts with FIRST and ends with LAST")
```

- You can also use the Wildcard Pattern with a star expression.
- In this case, we dismiss all elements between the first and last, if present.

Caveat: be aware of partial matches binding names

```
target = [1, 2, 3, 4]

match target:
    case [a, b, 1000, d]:
        print("This won't match, 3 != 100")
```

- CPython will already bind names while trying to match a pattern
- Here, it fails at element 3, after having bound **a** & **b**, but before binding **d**
- **a** & **b** will remain bound to **1** and **2**

Mapping Pattern

```
my_introduction = {"name": "Sebastiaan", "age": 35}

match my_introduction:
    case {"age": 35}:
        print("Someone's age is 35.")
    case {"name": "Sebastiaan", "age": 35}:
        print("Sebastiaan is 35 years old.")
    case _:
        print("I wasn't able to match anything.")
```

Mapping Pattern

```
my_introduction = {"name": "Sebastiaan", "age": 35}

match my_introduction:
    case {"age": 35}:
        print("Someone's age is 35.")
    case {"name": "Sebastiaan", "age": 35}:
        print("Sebastiaan is 35 years old.")
    case _:
        print("I wasn't able to match anything.")
```

- This will match the first case block!

Mapping Pattern

```
my_introduction = {"name": "Sebastiaan", "age": 35}

match my_introduction:
    case {"name": "Sebastiaan", "age": 35}:
        print("Sebastiaan is 35 years old.")
    case {"age": 35}:
        print("Someone's age is 35.")
    case _:
        print("I wasn't able to match anything.")
```

- Lesson: match more specific patterns first!

Mapping Pattern

```
my_introduction = {"name": "Sebastiaan", "age": 35}

match my_introduction:
    case {"name": "Sebastiaan", "age": age}:
        print(f"Sebastiaan is {age} years old.")
```

- You can use any pattern for the values, including a Capture Pattern.

Class Pattern

```
person = Person(name="Sebastiaan")  # person.name = "Sebastiaan"

match person:
    case Person():
        print("This is an instance of Person")
```

- A Class Pattern is a type followed by parenthesis; e.g., `ClassName()`

Class Pattern

```
person = Person(name="Sebastiaan")  # person.name = "Sebastiaan"

match person:
    case Person():
        print("This is an instance of Person")
```

- A Class Pattern is a type followed by parenthesis; e.g., `ClassName()`
- The match target will then be checked with `isinstance(target, ClassName)`

Class Pattern: Matching attributes

```
person = Person(name="Sebastiaan")  # person.name = "Sebastiaan"
person.age = 35

match person:
    case Person(age=35):
        print("This person is 35.")
```

- You can also match objects by their attributes.
- Here, the object is matched using a keyword attribute pattern.

Class Pattern: Matching attributes

```
person = Person(name="Sebastiaan")  # person.name = "Sebastiaan"
person.age = 35

match person:
    case Person(age=captured_age):
        print(f"This person's age is {captured_age}.")
```

- We can also use other patterns as value patterns.
- Here we use a Capture Pattern to capture the value.

PART III: A PRACTICAL EXAMPLE

Parsing a Search API response

- We want to use a Search API that responds in the following way:

```
response = {
    "query": "<your query>",
    "results": [<a list with results matching the query>],
}
```

- If the query did not result in any matches, the list will be empty.

Parsing a Search API response

```
response = requests.get("https://search.local?q=hello")

match response.json():
    case {"query": query, "result": []}:
        handle_no_results(query)
    case {"query": query, "result": [result]}:
        handle_single_result(query, result)
    case {"query": query, "result": [*results]}:
        handle_multiple_results(query, results)
    case error_response:
        log.error("Got an unexpected response", response=error_response)
```

WRAPPING IT UP

ORDINA

Summary

- Structural Pattern Matching is a very powerful tool
- It does require you to learn a new mini-language
- I've had a lot of fun playing around with Structural Pattern Matching

ORDINA
Ahead of change

Slides available: https://sebastiaanzeeff.nl

- Twitter: @SebastiaanZeeff
- LinkedIn: https://www.linkedin.com/in/sebastiaanzeeff/
- GitHub: https://github.com/sebastiaanz
- Python Discord: @Sebastiaan#0008

ORDINA
Ahead of change

Slides available: https://sebastiaanzeeff.nl

- Twitter: @SebastiaanZeeff
- LinkedIn: https://www.linkedin.com/in/sebastiaanzeeff/
- GitHub: https://github.com/sebastiaanz
- Python Discord: @Sebastiaan#0008

QUESTIONS?