Pyltalia Pattern Matching Talk Documentation

Release 1.0

Raymond Hettinger

Jun 05, 2022

CONTENTS

PyItalia 2022

CHAPTER

ONE

RAYMOND HETTINGER

My Mission Train thousands of Python Programmers

Contact Info raymond dot hettinger at gmail dot com

Twitter Account @raymondh

Date June 4, 2022

Contents:

1.1 Introduction

When thinking about structural pattern matching, take a "grammar first" approach. Everything in a case clause acts completely different from the same code outside of a case clause.

- A variable name *always* triggers a *capture* pattern. It *always* assigns and *always* matches.
- Parentheses *always* triggers a *class* pattern.
- Dots *always* trigger a *value* pattern.
- Square brackets *always* trigger a *sequence* pattern.
- Curly braces *always* trigger a *mapping* pattern.
- Literal numbers and strings always trigger an equality test.
- `None`, `True`, and `False` always compare on identity.
- _ *always* triggers a wildcard pattern and matches everything. You can only have one of these and it must be last.

1.1.1 General guidelines

Here are some helpful tips:

1) Remember that cases are a big if/elif chain and the first matching case wins.
2) Order cases first for correctness. This means putting specific cases before general cases.
3) Order cases secondarily for speed. This means putting common cases before rare cases.
4) To replace a literal with a variable or expression, use the value pattern as shown in the next section.
5) If the cases are exhaustive, always add a catchall *wildcard* case. The catches errors for unexpected cases. More importantly, it catches a preceding case when you accidentally use a variable in case statement when you needed a *value* pattern.

1.2 Miracle Tools

The pattern matching grammar has some builtin limitations.

Here are several tools that extend the capabilities to solve common challenges.

```
import re

class Var:
    pass

class Const:
    pass

class FuncCall:
    "Descriptor to convert fc.name to func(name)."

    def __init__(self, func):
        self.func = func

    def __set_name__(self, owner, name):
        self.name = name

    def __get__(self, obj, objtype=None):
        return self.func(self.name)

class RegexEqual(str):
    "Override str.__eq__ to match a regex pattern."

    def __eq__(self, pattern):
        return bool(re.search(pattern, self))

class InSet(set):
    "Override set.__eq__ to test set membership."

    def __eq__(self, elem):
        return elem in self
```

1.2.1 Namespaces for the Class Pattern

The `Const` and `Var` classes provide data holders for constants and variables:

```
>>> Const.pi = 3.1415926535
>>> Const.pi
3.1415926535

>>> Var.x = 10
>>> Var.x += 1
>>> Var.x
11
```

This is used in case clauses to replace a *literal* pattern with a variable by using the *value* pattern:

```
>>> match 11:
...     case Const.pi:
...         print('Matches "pi"')
...     case Var.x:
...         print('Matches "x"')
...
Matches "x"
```

1.2.2 Dynamic Dispatch to Function Calls

The `FuncCall` class is a descriptor that passes the attribute name to function call.

Here we pass the attribute names `x` and `y` to the function `ord`:

```
>>> class A:
        x = FuncCall(ord)
        y = FuncCall(ord)

>>> A.x
120
>>> A.y
121
```

This is used in case clauses to call arbitrary functions using the *value* pattern.

This is needed when for impure functions where the value can change between successive calls (otherwise you could use `Const` or `Var` tools shown above).

For example, consider a language translation function that changes its result depending on the current language setting. We could create a namespace with dynamic lookups:

```
class Directions:
    north = FuncCall(translate)
    south = FuncCall(translate)
    east = FuncCall(translate)
    west = FuncCall(translate)
```

In the match/case statement, we use the *value* pattern to trigger a new function call:

```
def convert(direction):
    match direction:
        case Directions.north:
            return 1, 0
        case Directions.south:
            return -1, 0
        case Directions.east:
            return 0,  1
        case Directions.west:
            return 0, -1
        case _:
            raise ValueError(_('Unknown direction'))
    print('Adjustment:', adj)
```

The tool is used like this:

```
>>> set_language('es')    # Spanish
>>> convert('sur')
(-1, 0)
>>> set_language('fr')    # French
>>> convert('nord')
(1, 0)
```

The case statements match the current language setting and will change when the language setting changes.

1.2.3 Matching Regular Expressions

The `RegexEqual` class inherits from `str` and overrides the `__eq__` method to match a regular expression.

```
>>> bool(RegexEqual('hello') == 'h.*o')
True
```

This is used in the match-clause (not a case clause). It will match cases with a regex for a *literal* pattern:

```
match RegexEqual('the tale of two cities'):
    case 's...y':
        print('A sad story')
    case 't..e':
        print('A mixed tale')
    case 's..a':
        print('A long read')
```

1.2.4 Testing Set Membership

The `InSet` class inherits from `set` and overrides the `__eq__` method to test for set membership:

```
Colors = SimpleNamespace(
    warm = InSet({'red', 'orange', 'yellow'}),
    cool = InSet({'green', 'blue', 'indigo', 'violet'}),
    mixed = InSet({'purple', 'brown'})
)

match 'blue':
    case Colors.warm:
        print('warm')
    case Colors.cool:
        print('cool')
    case Colors.mixed:
        print('mixed')
```

1.3 Formatting mini language for datetime objects

Here we use pattern matching is its most basic form as a case statement.

Learning points:

- The cases are independent, so the order doesn't matter for correctness.
- The order does matter for speed, so put the most common cases first.
- The *wildcard* pattern is used at the bottom for a catchall that raises an exception. This is a best practice when the cases are supposed to be exhaustive.
- Alternatively, the catchall could be made to return the date code unchanged. This is useful for multiple substitution passes.

```
from datetime import datetime, timedelta, timezone
import calendar
import re

CST = timezone(-timedelta(hours=6), name='CST')
ts = datetime.now(tz=CST)

def replace_code(mo):
    c = mo.group(1)
    match c:
        case 'a': return calendar.day_abbr[ts.weekday()]
        case 'A': return calendar.day_name[ts.weekday()]
        case 'w': return str(ts.weekday())
        case 'd': return f'{ts.day:02d}'
        case 'b': return calendar.month_abbr[ts.month]
        case 'B': return calendar.month_name[ts.month]
        case 'm': return f'{ts.month:02d}'
        case 'y': return f'{ts.year % 100:02d}'
        case 'Y': return f'{ts.year:04d}'
        case 'H': return f'{ts.hour:02d}'
        case 'I': return f'{ts.hour % 12:02d}'
        case 'p': return 'PM' if ts.hour > 12 else 'AM'
        case 'M': return f'{ts.minute:02d}'
        case 'S': return f'{ts.second:02d}'
        case 'f': return f'{ts.microsecond:06d}'
        case 'j': return f'{ts.timetuple().tm_yday:03d}'
        case '%': return '%'
        case _: raise ValueError(f'Unknown code: {c}')

def date_format(datestr):
    return re.sub(r'%([A-Za-z])', replace_code, datestr)
```

We call the date formatter like this:

```
>>> date_format('%H:%M:%S %p %a %A %d %b %B %m %y %Y %H %I %f %p %j')
01:56:59 AM Sat Saturday 04 Jun June 06 22 2022 01 01 331226 AM 155
```

1.4 Language Tokenizer

A "tokenizer" is the first step in compiling or interpreting a language. Its goal is to break the input text into "tokens" with a token type, token value, and position information for error reporting.

In this example, we use pattern matching like a case statement.

Learning points:

- A *literal* pattern and a *value* pattern cannot be combined in one case statement to express a compound condition.
- So, the `ID` case requires a "type guard".
- Here, we intentionally omit the *wildcard* pattern as a catchall because we want fallthrough for `ASSIGN`, `END`, non-keyword `ID` and `OP`.
- There is a `MISMATCH` catchall in the regex cases. We use that for the error case to get be able to report the mismatching token and its position.

```
from typing import NamedTuple
import re

class Token(NamedTuple):
    type: str
    value: str
    line: int
    column: int

def tokenize(code):
    keywords = {'IF', 'THEN', 'ENDIF', 'FOR', 'NEXT', 'GOSUB', 'RETURN'}
    token_specification = [
        ('NUMBER',   r'\d+(\.\d*)?'),  # Integer or decimal number
        ('ASSIGN',   r':='),           # Assignment operator
        ('END',      r';'),            # Statement terminator
        ('ID',       r'[A-Za-z]+'),    # Identifiers
        ('OP',       r'[+\-*/]'),      # Arithmetic operators
        ('NEWLINE',  r'\n'),           # Line endings
        ('SKIP',     r'[ \t]+'),       # Skip over spaces and tabs
        ('MISMATCH', r'.'),            # Any other character
    ]
    tok_regex = '|'.join('(?P<%s>%s)' % pair for pair in token_specification)
    line_num = 1
    line_start = 0
    for mo in re.finditer(tok_regex, code):
        kind = mo.lastgroup
        value = mo.group()
        column = mo.start() - line_start
        match kind:
            case 'NUMBER':
                value = float(value) if '.' in value else int(value)
            case 'ID' if value in keywords:
                kind = value
            case 'NEWLINE':
                line_start = mo.end()
                line_num += 1
                continue
```

(continues on next page)

(continued from previous page)

```
            case 'SKIP':
                continue
            case 'MISMATCH':
                raise RuntimeError(f'{value!r} unexpected on line {line_num}')
        yield Token(kind, value, line_num, column)
```

We call the tokenizer like this:

```
statements = '''
    IF quantity THEN
        total := total + price * quantity;
        tax := price * 0.05;
    ENDIF;
'''

for token in tokenize(statements):
    print(token)
```

That produces this output:

```
Token(type='IF', value='IF', line=2, column=8)
Token(type='ID', value='quantity', line=2, column=11)
Token(type='THEN', value='THEN', line=2, column=20)
Token(type='ID', value='total', line=3, column=12)
Token(type='ASSIGN', value=':=', line=3, column=18)
Token(type='ID', value='total', line=3, column=21)
Token(type='OP', value='+', line=3, column=27)
Token(type='ID', value='price', line=3, column=29)
Token(type='OP', value='*', line=3, column=35)
Token(type='ID', value='quantity', line=3, column=37)
Token(type='END', value=';', line=3, column=45)
Token(type='ID', value='tax', line=4, column=12)
Token(type='ASSIGN', value=':=', line=4, column=16)
Token(type='ID', value='price', line=4, column=19)
Token(type='OP', value='*', line=4, column=25)
Token(type='NUMBER', value=0.05, line=4, column=27)
Token(type='END', value=';', line=4, column=31)
Token(type='ENDIF', value='ENDIF', line=5, column=8)
Token(type='END', value=';', line=5, column=13)
```

1.5 JSON Examples

Structural pattern matching really shines to querying JSON data with either SQL style queries or recursive queries

1.5.1 JSON Lines

This code runs an SQL style query to select game dates where the game was a victory:

```
from pprint import pp
import json

with open('data/team_history.json') as f:
    hist = json.load(f)

# SELECT date FROM History WHERE result = "won";
for game in hist:
    match game:
        case {'result': 'won', 'date': date}:
            print(f'On {date}, we won the game!')
```

Here is a sample data set for that query:

```
[
    {"date": "3/14/2013", "score": 14, "result": "lost"},
    {"date": "6/2/2013", "score": 15, "result": "lost"},
    {"date": "7/15/2013", "score": 1, "result": "won"},
    {"date": "8/14/2013", "score": 4, "result": "lost"},
    {"date": "8/19/2013", "score": 7, "result": "won"},
    {"date": "9/2/2013", "score": 2, "result": "lost"},
    {"date": "9/7/2013", "score": 5, "result": "won"},
    {"date": "10/2/2013", "score": 3, "result": "lost"},
    {"date": "10/5/2013", "score": 12, "result": "won"}
]
```

1.5.2 Standard JSON

More typically, JSON data consists of nested dictionaries.

This code scans the data for matches:

```
with open('data/books.json') as f:
    catalog = json.load(f)

print('\nWhat the names and prices of the computer books?')
print('SELECT price, name) FROM Catlog WHERE genre = "Computer";')
for book in catalog.values():
    match book:
        case {'genre': 'Computer', 'title': name, 'price': price}:
            print(price, name)

print('\nWho is the author and title for bk107?')
```

(continues on next page)

(continued from previous page)

```
print('SELECT author, "-->", title FROM Catalog WHERE id = "bk107";')
match catalog:
    case {'bk107': {'author': author, 'title': title}}:
        print(author, '-->', title)
```

Excerpt of the dataset:

```
{
    "bk103": {
        "description": "After the collapse of a nanotechnology \n      society in␣
↪England, the young survivors lay the \n      foundation for a new society.",
        "title": "Maeve Ascendant",
        "price": 6.00,
        "author": "Corets, Eva",
        "publish_date": "2000-11-17",
        "genre": "Fantasy"
    },
    "bk102": {
        "description": "A former architect battles corporate zombies, \n      an evil␣
↪sorceress, and her own childhood to become queen \n      of the world.",
        "title": "Midnight Rain",
        "price": 6.00,
        "author": "Ralls, Kim",
        "publish_date": "2000-12-16",
        "genre": "Fantasy"
    },
    "bk101": {
        "description": "An in-depth look at creating applications \n      with XML.",
        "title": "XML Developer's Guide",
        "price": 44.95,
        "author": "Gambardella, Matthew",
        "publish_date": "2000-10-01",
        "genre": "Computer"
    },
    ...
```

1.5.3 Recursive JSON processing

This recipe shows the pattern for traversing arbitrary tree structured data:

```
class Var:
    pass

def path_to(target, node):
    Var.target = target
    match node:
        case Var.target:
            return f' --> {target!r}'
        case list():
            for i, subnode in enumerate(node):
                if (path := path_to(target, subnode)):
```

(continues on next page)

(continued from previous page)

```
                    return f'[{i}]' + path
        case dict():
            for key, subnode in node.items():
                if (path := path_to(target, subnode)):
                    return f'[{key!r}]' + path
    return ''
```

Sample dataset:

```
tree = {'one': ['abc',
                'def',
                'ghi',
                {'four': 4,
                 'five': 5}],
        'two': ['jkl',
                'mno',
                'BLUE',
                {'six': 6,
                 'seven': 7}],
        'three': ['qrs',
                  'BLUE',
                  'BLUE',
                  {'eight': 'BLUE',
                   'BLUE': 9}]}
```

Sample call:

```
>>> path_to(7, tree)
"['two'][3]['seven'] --> 7"
```

1.6 Data Serialization

Here we make more sophisticated use of the match/case statement to implement a serializer styled after Python's *marshal* module.

Learning points:

- The try/except is necessary because match/case can only tell whether `__reduce__` is present (it always is). It cannot tell whether the call would succeed.
- The test `True` and `False` must precede the test for `int()` because the latter is more general and would match first.
- The type tests use the *class* pattern. However, they use an instance check which would give a false positive for subclasses. So, we have to add a guard to make sure the type is an exact match.
- The *wildcard* pattern is used at the bottom for two reasons. Since only one irrefutable case is allowed, we can assure the previous cases don't accidentally match everything. Also, this tool only supports specific types, so we have to raise an error if an unknown type is encountered.
- In the deserialization code, the cases are mutually exclude, so any ordering would be correct. We can choose to optimize execution speed by putting the most common cases first.

```
from typing import TextIO, Callable
from collections import Counter
from fractions import Fraction
from decimal import Decimal

supported_type = (None | bool | int | str | list | tuple | dict
                  | bytes | Counter | Decimal | Fraction)

def dump_target(obj: supported_type, emit: Callable[[str], None]) -> None:

    reduce_errors = (TypeError, ValueError)

    def dump_obj(x: supported_type) -> None:
        try:
            func, args = x.__reduce__()
        except reduce_errors:
            pass
        else:
            emit('R')
            dump_obj(func.__name__)
            dump_obj(args)
            return
        match x:
            case True:
                emit('t')
            case False:
                emit('f')
            case int() if type(x) == int:
                assert not isinstance(x, bool)
                emit(f'I{x:x}!')
            case str() if type(x) == str:
                emit('S')
                dump_obj(len(x))
                emit(x)
            case None:
                emit('n')
            case list() if type(x) == list:
                emit('L')
                dump_obj(len(x))
                for elem in x:
                    dump_obj(elem)
            case tuple() if type(x) == tuple:
                emit('T')
                dump_obj(list(x))
            case dict() if type(x) == dict:
                emit('D')
                dump_obj(len(x))
                for key, value in x.items():
                    dump_obj(key)
                    dump_obj(value)
            case bytes() if type(x) == bytes:
                emit('B')
                dump_obj(x.hex())
```

(continues on next page)

(continued from previous page)

```
            case _:
                raise ValueError(f'Unserializable: {x!r}')

    dump_obj(obj)

def dumps(obj: supported_type) -> str:
    chunks: list[str] = []
    dump_target(obj, emit=chunks.append)
    return ''.join(chunks)

def loads(p: str) -> supported_type:
    i = 0

    def loads_pos() -> supported_type:
        nonlocal i

        code = p[i]
        i += 1
        match code:
            case 'I':
                j = p.index('!', i)
                digits = p[i : j]
                i = j + 1
                return int(digits, 16)
            case 'S':
                n = loads_pos()
                result = p[i : i + n]
                i += n
                return result
            case 't':
                return True
            case 'f':
                return False
            case 'n':
                return None
            case 'L':
                n = loads_pos()
                return [loads_pos() for _ in range(n)]
            case 'T':
                return tuple(loads_pos())
            case 'D':
                n = loads_pos()
                return dict((loads_pos(), loads_pos()) for _ in range(n))
            case 'B':
                return bytes.fromhex(loads_pos())
            case 'R':
                funcname = loads_pos()
                args = loads_pos()
                func = globals()[funcname]
                return func(*args)
            case _:
                raise ValueError('Malformed serialization')
```

(continues on next page)

(continued from previous page)

```
    return loads_pos()

def dump(obj: supported_type, f: TextIO) -> None:
    dump_target(obj, emit=f.write)

def load(f: TextIO) -> supported_type:
    return loads(f.read())
```

A user session with the serializer looks like this:

```
>>> data = {'raymond': 'red',
...         'nested': {'outer':
...                    {'inner1': [(0, 'zero'), 1, (5, 'five')],
...                     'inner2': (Fraction(3, 4), Decimal('5.25'))}}}
...
>>> s = dumps(data)
>>> loads(s) == data
True
>>> s
'DI2!SI7!raymondSI3!redSI6!nestedDI1!SI5!outerDI2!SI6!inner1LI3!TLI2!I0!SI4!zeroI1!TLI2!
↪I5!SI4!fiveSI6!inner2TLI2!RSI8!FractionTLI2!I3!I4!RSI7!DecimalTLI1!SI4!5.25'
```

1.7 Inverting tests

When translating if/else style to match/case, a problem arises expressing negative assertions:

```
if isinstance(x, (list, dict)):
   x = json.dump(x)
elif not isinstance(x, str):    # <-- Inverted test
   x = str(x)
```

1.7.1 Basic technique

Intrinsically, the design of structural pattern matching only triggers a case when there is a positive match. However, there are two workarounds.

The easiest way is to add a guard expression. But this is a last resort because it doesn't take advantage of the pattern matching and destructuring capabilities.

The second way is to add an earlier matching test so that later cases can presume that the inverse match is true. This works nicely if the negative case was intended to be the last case. If not, it becomes a little awkward because nesting is required.

1.7.2 Guards

Take the inverted test and move it into an if-expression:

```
match x:
    case list() | dict():
        x = json.dump(x)
    case _ if not isinstance(x, str):    # <-- Inverted test
        x = str(x)
```

1.7.3 Pretest

The basic idea here is to make a positive match so that subsequent cases can assume a negative match:

```
match x:
    case list() | dict():
        x = json.dump(x)
    case str():                          # <-- Positive match
        pass
    case _:                              # <-- Inverted case
        x = str(x)
```

1.7.4 Pretest with subsequent cases

The pretest technique is elegant unless you there are additional cases to be matched:

```
if isinstance(x, (list, dict)):
   x = json.dump(x)
elif not isinstance(x, str):    # <-- Inverted test
   x = str(x)
elif x == 'quit':               # <-- Additional test
   sys.exit(0)
```

The easiest solution here is to to move the additional test to occur before the inverted test:

```
match x:
    case list() | dict():
        x = json.dump(x)
    case 'quit':                    # <-- Moved the test up
        sys.exit(0)
    case str():                     # <-- Positive match
        pass
    case _:                         # <-- Inverted case
        x = str(x)
```

It's not always possible to reorder the tests. If so, then a new level of nested matching can be introduced:

```
case list() | dict():
    x = json.dump(x)
case str():                     # <-- Positive match
    match x:                    # <-- Nested match
```

(continues on next page)

(continued from previous page)

```
        case 'quit':            # <-- Inner case
            sys.exit(0)
case _:                         # <-- Inverted case
    x = str(x)
```

1.8 Problems and Solutions

Here we walk through common problems and their solutions

1.8.1 Matching a Set or Frozenset

Pattern matching is designed around sequences, mappings, classes, and literals. It makes no provisions for handing `set()` or `frozenset`.

The is easily handled by replacing a set literal with a *value* pattern:

```
class Const:
    red_team = {'adam', 'becky', 'sue'}
    blue_team = {'bob', 'sally', 'mike'}


match {'mike', 'sally', 'bob'}:
    case Const.red_team:
        print('Red Team')
    case Const.blue_team:
        print('Blue Team')
```

This outputs:

```
Blue Team
```

1.8.2 Matching exact types

The *class* pattern performs *instance()* checks that will match subclasses as well as exact type matches.

Sometimes you need to exclude subclasses.

The core pattern matching grammar makes no provision for this, so you need to add a type guard:

```
case list() if type(x) == list:
    ...
```

1.8.3 Mixing general and specific cases

Some cases may be overlapping. In particular, general cases may subsume specific cases.

The solution here is to take advantage of the "first match rule" and put the specific cases before the general cases:

```
match(x):
     case list():
         ...
     case tuple():
         ...
     case Iterable():
         ...
```

1.8.4 Converting hasattr() tests for duck typing

Sometimes you need cases that perform `hasattr()` matching.

One common example is recognizing named tuples and dataclasses by looking for a `_fields` attribute:

```
if hasattr(candidate, '_fields'):
   do_action()
```

We can use the *class* pattern for this purpose and add the specific fields:

```
match candidate:
   case object(_fields=_):
       do_action()
```

Here assign to _ because the value of `_fields` isn't needed, but you can use some other variable name if you need to capture the value.

Here's a worked-out example of duck typing with structural pattern matching:

```
from typing import NamedTuple
from dataclasses import dataclass

class Whale(NamedTuple):
    name: str
    num_fins: int

@dataclass
class Vehicle:
    name: str
    num_wheels: int

subject = Vehicle('bicycle', 2)

match subject:
    case object(num_fins=n):
        print(f'Found {n} fins')
    case object(num_wheels=_):
        print(f'Found wheeled object')
    case _:
        print('Unknown')
```

This script outputs:

```
Found wheeled object
```

1.8.5 Converting impossible test cases

Sometimes you need to convert if/elif chains to match/case, but the test condition is too complex.

Just use the *capture* pattern with a *guard*:

```
match value:
    case x if first_complex_test(x):
        ...
    case x if second_complex_test(x):
        ...
```

1.8.6 Fooling a lint tool

If a lint tool flags a case with a *wildcard* pattern and a guard:

```
match value:
    case _ if test(): ...
    case 'xyz': ...
```

There is an interesting universal substitute using the *class* pattern:

```
match value:
    case object() if test(): ...
    case 'xyz': ...
```

This creates an "undetectable" wildcard. Mostly, this should never be needed, but it is helpful to know that *anything* will match `object()`.

1.8.7 Matching special values for floats

Case clauses allow floats in the *literal* pattern, but only the finite values. To handle positive infinite values, use the *value* pattern. The case of `Nan` is requires a guard because NaNs do not compare equal to one another:

```
match x:
    case _ if math.isfinite(x): ...   # Finite Value
    case _ if math.isnan(x): ...      # Nan
    case math.inf: ...                # Infinity
    case _: ...                       # Negative Infinity
```