Match all things Python:

Parsing structured content with Python's new match statement

FOSDEM 2024, Brussels – 04.02.2024

Marc-André Lemburg :: eGenix.com GmbH

Speaker Introduction

- Marc-André Lemburg
  - Python since 1994
  - Studied Mathematics
  - CEO eGenix.com GmbH
  - Senior Solution Architect, Consulting CTO and Coach
  - EuroPython Society Fellow and former Chair
  - Python Software Foundation Fellow and former Director
  - Python Core Developer (Unicode, DB-API, platform module)
  - Co-founder Python Meeting Düsseldorf
  - Based in Düsseldorf, Germany

  - More details: https://malemburg.com/

Welcome the new Python match statement

```
match obj:

    case list() as list_obj:
        print (f'found list: {list_obj!r}')

    case dict() as dict_obj:
        print (f'found dict: {dict_obj!r}')

    case [a, b, c]:
        print (f'found 3 element sequence: {obj!r}')

    case {'name': name, 'value': value}:
        print (f'found name-value mapping: {obj!r}')

    case unknown:
        print (f'could not parse object: {unknown!r}')
```

Match statement: Motivation

- Better syntax for long and nested if-elif-elif-else constructs
  - Similar to, but a lot more advanced than the C switch statement
- Good support for easy matching of nested structures
  - Can be generalized to more complex structures as well
- Good support for type based matching
  - Both for builtin types and user defined classes
- Good support for combining matching and parsing
  - Avoids duplicate effort

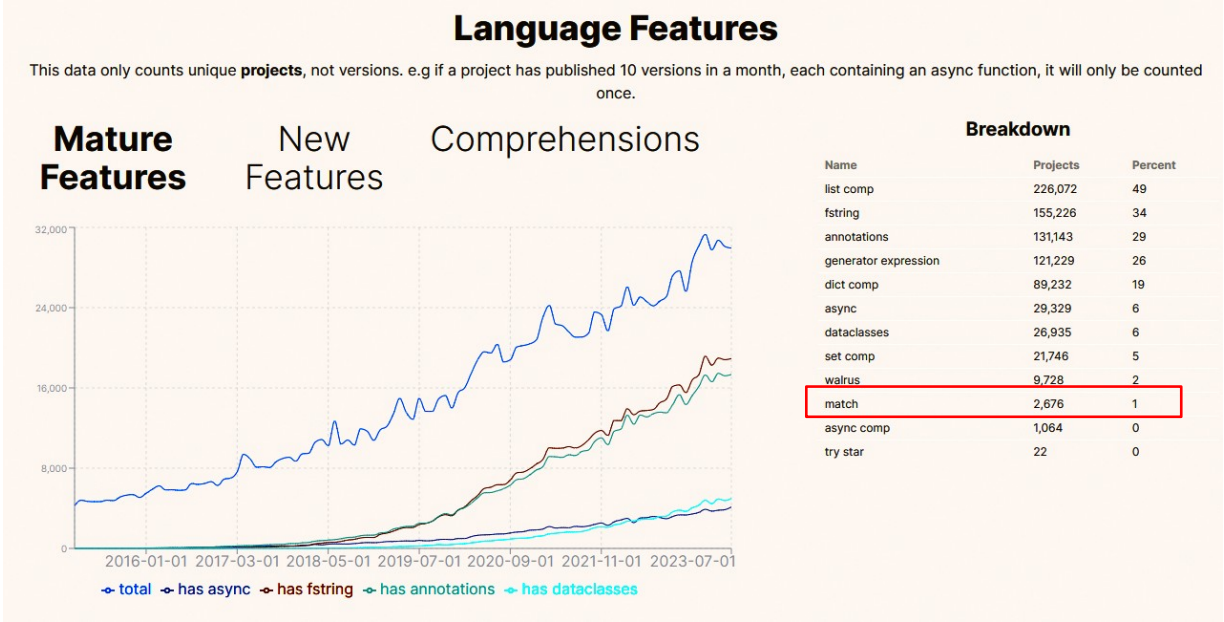
Match statement: History

- Introduced in Python 3.10
  - Released 2021-10-04
  - More than 2 years ago
- Basic idea had been cooking for ages
  - see e.g. PEP 275 from 2001

Source: https://www.python.org/downloads/release/python-3100/

Match statement: Popularity

- How popular is this new feature ?
  - Only 2,676 PyPI packages use the match statement
  - That's about 0.55% of all packages on PyPI



Source: https://py-code.org/stats (as of 2023-07-01)

Match statement: Documentation

- PEPs
  - PEP 635 – Structural Pattern Matching: Motivation and Rationale
    - Discussion about syntax – not a good intro
  - PEP 636 – Structural Pattern Matching: Tutorial
    - Best way to start to learn the new syntax
  - PEP 634 – Structural Pattern Matching: Specification
    - In-depth spec for how things work
- Python documentation: match statement
  - Not much different than the PEPs :-(

Match statement: How does it all work ?

```
    match obj:

        case list() as list_obj:
            print (f'found list: {list_obj!r}')

        case dict() as dict_obj:
            print (f'found dict: {dict_obj!r}')

        case [a, b, c]:
            print (f'found 3 element sequence: {obj!r}')

        case {'name': name, 'value': value}:
            print (f'found name-value mapping: {obj!r}')

        case _:
            print (f'could not parse object: {obj!r}')
```

Match statement: Explaining the different parts

```
match obj:

    case list() as list_obj:
        print (f'found list: {list_obj!r}')

    case dict() as dict_obj:
        print (f'found dict: {dict_obj!r}')

    case [a, b, c]:
        print (f'found 3 element sequence: {obj!r}')

    case {'name': name, 'value': value}:
        print (f'found name-value mapping: {obj!r}')

    case _:
        print (f'could not parse object: {obj!r}')
```

The match object

Match statement: Explaining the different parts

Match patterns

```
match obj:

    case list() as list_obj:
        print (f'found list: {list_obj!r}')

    case dict() as dict_obj:
        print (f'found dict: {dict_obj!r}')

    case [a, b, c]:
        print (f'found 3 element sequence: {obj!r}')

    case {'name': name, 'value': value}:
        print (f'found name-value mapping: {obj!r}')

    case _:
        print (f'could not parse object: {obj!r}')
```

Match statement: Explaining the different parts

Match code

```
match obj:

    case list() as list_obj:
        print (f'found list: {list_obj!r}')

    case dict() as dict_obj:
        print (f'found dict: {dict_obj!r}')

    case [a, b, c]:
        print (f'found 3 element sequence: {obj!r}')

    case {'name': name, 'value': value}:
        print (f'found name-value mapping: {obj!r}')

    case _:
        print (f'could not parse object: {obj!r}')
```

Match statement: Explaining the different parts

Capturing variables

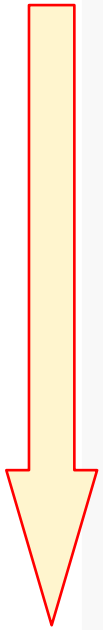
```
match obj:

    case list() as list_obj:
        print (f'found list: {list_obj!r}')

    case dict() as dict_obj:
        print (f'found dict: {dict_obj!r}')

    case [a, b, c]:
        print (f'found 3 element sequence: {obj!r}')

    case {'name': name, 'value': value}:
        print (f'found name-value mapping: {obj!r}')

    case _:
        print (f'could not parse object: {obj!r}')
```

Match statement: Explaining the different parts

```
    match obj:

        case list() as list_obj:
            print (f'found list: {list_obj!r}')

        case dict() as dict_obj:
            print (f'found dict: {dict_obj!r}')

        case [a, b, c]:
            print (f'found 3 element sequence: {obj!r}')

        case {'name': name, 'value': value}:
            print (f'found name-value mapping: {obj!r}')

        case _:
            print (f'could not parse object: {obj!r}')
```

Non-capturing wildcard

Match statement: Execution flow



```
match obj:

    case list() as list_obj:
        print (f'found list: {list_obj!r}')

    case dict() as dict_obj:
        print (f'found dict: {dict_obj!r}')

    case [a, b, c]:
        print (f'found 3 element sequence: {obj!r}')

    case {'name': name, 'value': value}:
        print (f'found name-value mapping: {obj!r}')

    case _:
        print (f'could not parse object: {obj!r}')
```

Matching is tried top to bottom

First match wins

No fall-through between cases as in C

Match statement: Pattern types

- Literals
- Values
- Sequences
- Mappings
- Builtin types
- Classes
- Wildcards

- Nested patterns
- OR patterns
- Guards

```
match obj:

    case list() as list_obj:
        print (f'found list: {list_obj!r}')
    case [a, b, c]:
        print (f'found 3 element sequence: {obj!r}')
    case {'name': name, 'value': value}:
        print (f'found name-value mapping: {obj!r}')
    case 42 | "42":
        print ('found the meaning of life')
    case int(a) if a > 1000:
        print (f'found a large int: {obj!r}')
    case _:
        print (f'could not parse object: {obj!r}')
```

Match statement: Pattern types

- Literals
  - Strings `"abc"`
  - Numbers `123  2.3456`
  - True, False, None special singletons
- Values
  - Using a dot notation: `obj.name`
  - Needed to differentiate from type matching and capturing variables
  - Need to compare equal for a match

Match statement: Pattern types

- Sequences
  - Written as `[...]` or `(...)` `[a, b, c]  (d, e, f)`
  - Matches any sequence — not just lists and tuples
  - With support for * wildcards `[a, b, *rest]`
- Mappings
  - Written as `{...}` `{'name': name, 'value': value}`
  - Matches any mapping — not just dicts
  - With support for ** wildcards `{'name': name, **rest}`
  - Only works for literal keys `{obj: value}` does not work

Match statement: Pattern types

- Arbitrary Python types/classes
  - Using the type name: `abc()`
    - Parenthesis are important to distinguish from capturing variables
  - Support for builtin types: `int(), bool(), float(), etc.`
  - Support for user classes: `Point()`
  - Support for attributes: `Point(x=2, y=3)`
  - Support for capturing variables: `str(a), int(b), MyClass(c)`

Match statement: Pattern types

- All of the above in nested form

```
[a, {'name': name, **rest}, b, *more]
```

- OR combinations of the above using | *(pipe)*

```
int(a) | float(a)
[a, *more] | {'a': value, *more}
```

- Guards using the *if syntax*
  - Condition checked when the pattern matches
  - Can use already parsed data

```
[a, b] if a > 10
_ if obj in value_set
```

Match statement: Pattern types

- Wildcard pattern `case _:` at the end
  - Matches anything
  - Does not capture the value in a variable _

```
match obj:
    ...
    case _:
        print (f'could not parse object: {obj!r}')
```

- Wildcard pattern `case unknown:` at the end
  - Matches anything
  - Captures the value in the given variable

```
match obj:
    ...
    case unknown:
        print (f'could not parse object: {unknown!r}')
```

Match statement: Capturing values

- Bind parsed values to variables
  - Useful for processing parsed values
  - Bound values can be used directly in match code
  - Also available outside the match block, after match code execution

- Explicit form: `list() as some_list`
  - *list()* defines the type to check
  - *some_list* gets the list as value

  - Always works
  - Easy to understand

Match statement: Capturing values

- Implicit form: `[a, b]` `{'name': name}` `str(a)`
  - Binding variables embedded into type check
  - Works well, if the variable names are well chosen
  - If not, easy to confuse with types
- Works with some builtin types
  - bool, bytearray, bytes, dict, float, frozenset, int, list, set, str, and tuple
- Works with classes: `Point(a, b)`
  - Classes need special support for this (see PEP 634)
- Doesn't work with ABCs
  - Use explicit variant instead

Match statement: Where from here ?

- Regular expression matching does not work directly
  - Often needed for shell script like Python scripts
  - Can be had using helpers making use of the `.__eq__()` slot.
- Set member matching
  - Can be had using wildcard guards: `case _ if obj in value_set:`
  - Or (less efficient) using the OR pattern: `case 2 | 4 | 8:`
- Optimizations
  - Code using match does not necessarily run faster than corresponding *if-statement* based code

May be added in some later Python version...

Gotchas: Sequence and mapping patterns

- The sequence pattern matches any sequence, not only tuples or lists:
  - Use *list((a, b))* to parse only lists
  - Use *tuple((a, b))* to parse only tuples
- Same thing for mapping patterns

```
def test_match_obj3(obj):
    match obj:
        case (a, b):
            print (f'found a tuple')
        case [a, b]:
            print (f'found a list')
        case wrong_values:
            print (f'could not parse object:'
                   f' {wrong_values!r}')

t = (1, 2)
test_match_obj3(t)
# prints: found a tuple

l = [2, 3]
test_match_obj3(l)
# prints: found a tuple
```

Gotchas: Wildcard patterns

- Wildcard patterns may only be used at the end:
  - This is true for capturing wildcard patterns and non-capturing (_) ones
  - Not true for wildcard patterns with guards

```
def test_match_obj1(obj):
    match obj:
        case wrong_values:
            print (f'could not parse object:'
                   f' {wrong_values!r}')
        case list():
            print (f'found list')
        case dict():
            print (f'found dict')

# gives a SyntaxError

        case _ if obj > 0:

# works fine
```

Gotchas: Forgetting parenthesis

- Typos can easily result in broken code:
  - *dict* in this case is interpreted as capturing variable, not as a type check
  - As a result, parsing is wrong and you could easily break other code using *dict()* later on

```
def test_match_obj2(obj):
    match obj:
        case list():
            print (f'found list')
        case { 'properties': dict }:
            print (f'found dict with properties'
                   f' {dict!r}')
        case wrong_values:
            print (f'could not parse object: '
                   f'{wrong_values!r}')

obj = {'a': 1, 'properties': [2, 3, 4]}
test_match_obj2(obj)
# prints: found dict with properties [2, 3, 4]
```

Additional Resources

- Raymond Hettinger gave a good talk at PyCon Italia 2022
  - Search for "PyItalia 2022 Pattern Matching Talk"
    - Slides
    - Video
  - Includes ways to get around some of the current limitations

Main takeaway: Never stop learning and try out new things...

Thank you for your attention !

Time for discussion

Contact

eGenix.com Software, Skills and Services GmbH
Marc-André Lemburg
Pastor-Löh-Str. 48
D-40764 Langenfeld
Germany

eMail: mal@egenix.com
Phone: +49 211 9304112
Fax: +49 211 3005250
Web: https://www.egenix.com/

LinkedIn:

References

- Some content taken from:
  - https://www.python.org/downloads/release/python-3100/
  - https://py-code.org/stats
- Several photos taken from Pixabay and Unsplash
- Some screenshots taken from the mentioned websites
- All other graphics and photos are (c) eGenix.com or used with permission
- Details are available on request
- Logos are trademarks of their respective trademark holders