

# *Match all things Python:*

## *Advanced parsing of structured data using Python's new match statement*

Python Meeting Düsseldorf – 17.04.2024

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** (PEP 100, DB-API, platform module, ...)
  - **Co-founder Python Meeting Düsseldorf**
  - Based in Düsseldorf, Germany
  
  - More details: <http://malemburg.com>



# Intro: Recap of the new Python match statement

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

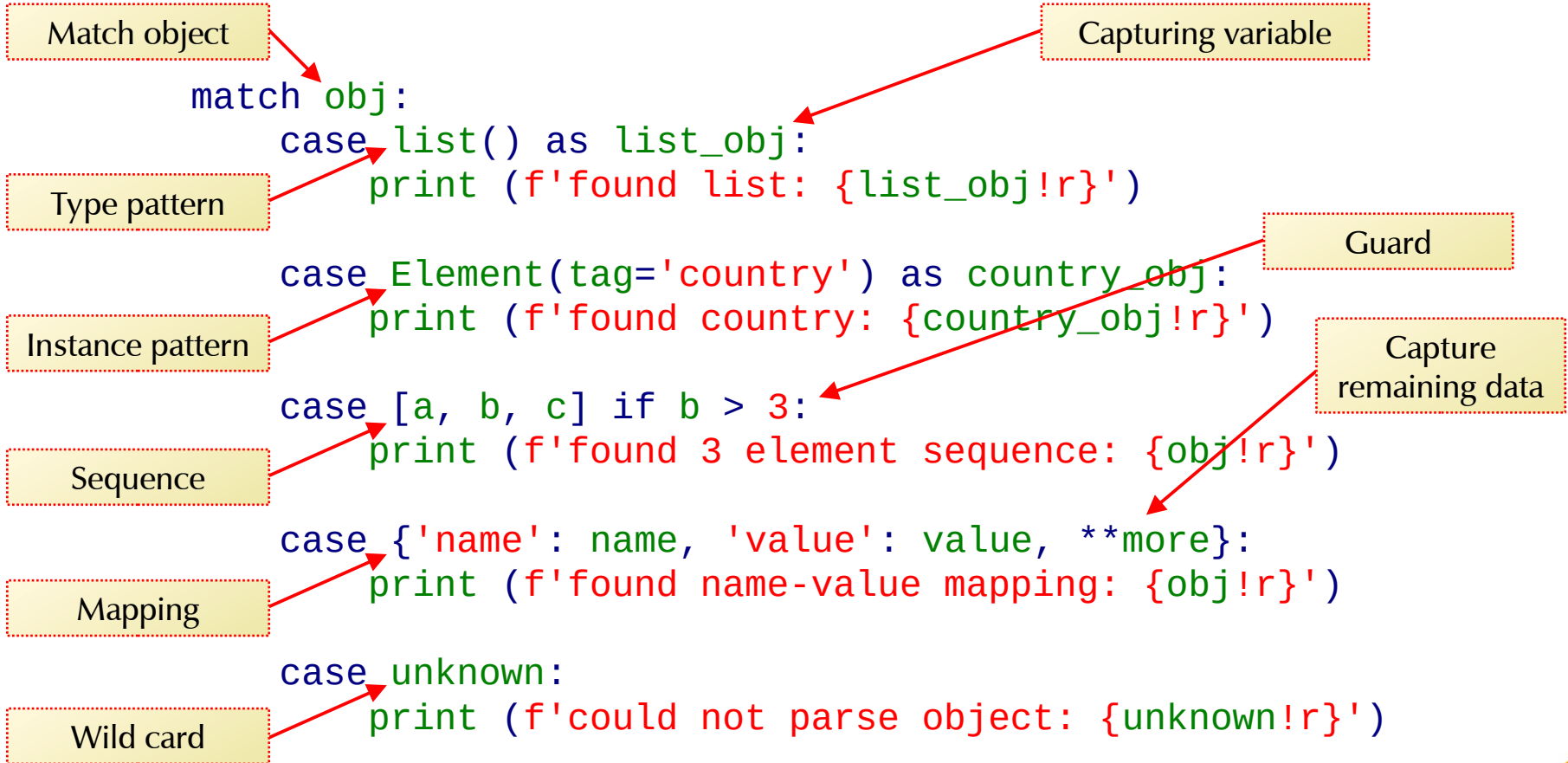
    case Element(tag='country') as country_obj:
        print (f'found country: {country_obj!r}')

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

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

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

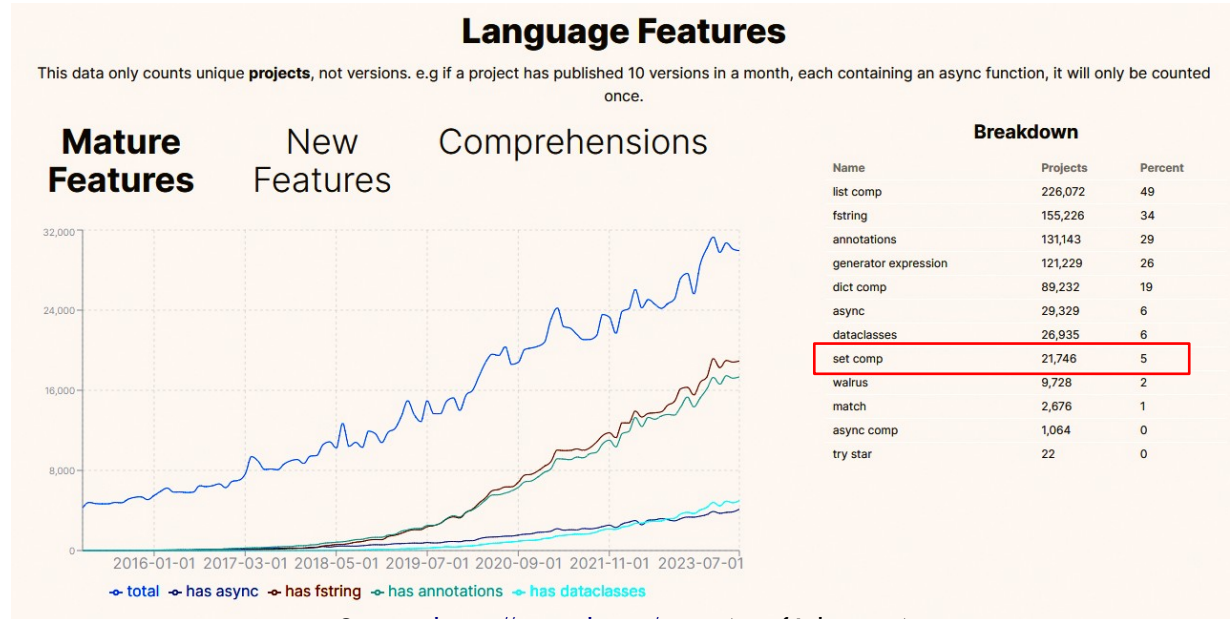
# Intro: Recap of the new Python match statement



# Match statement: Popularity

- Introduced in Python 3.10 (2021)
- 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**



# 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 :-)

# Some techniques used in advanced parsing

- OR parsing: 

```
case ["yes" | "y" | "on" | "true" | "1"] as option:
```
- Parsing optional / remaining arguments: 

```
case [1, 2, *args]:  
case [1, _, _, 4]:  
case {'name':name, **more}:
```
- Instance parsing: 

```
case Element(tag='country', attrib={'name': name}):
```
- Mixing local variables and capturing variables: 

```
case Call(func=Name(id='isinstance'),  
          args=[Name(id=params.varname),  
                Name(id=typename)]):
```

more...  
see Raymond Hettinger's  
PyCon Italia 2022 talk

# Keeping backwards compatibility

- *match* code will cause a **SyntaxError in Python 3.9 and lower**
- Can be emulated using *if*-constructs

- **Solution:**

- Put new code into separate module
- Use try-import to switch between implementations

or: Require Python 3.10+

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

    case Element(tag='country') as country_obj:
        print (f'found dict: {dict_obj!r}')

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

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

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



# Examples parsing structured data

- **JSON**
  - Record lists (tabular data exports)
  - GeoJSON (used for geospatial data)
- **XML**
  - Country data
- **AST**
  - *if*-statement to *match*-statement refactoring



# Parsing JSON: Tabular data

- Schema:

```
schema = {  
    "type": "object",  
    "properties": {  
        "price": {"type": "number"},  
        "name": {"type": "string"},  
    },  
}
```

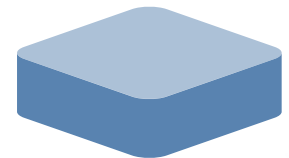
name	price	color
eggs	2.99	
eggs	2	
eggs		
eggs	3.99	brown

- Examples:

```
data_1 = {  
    "name": "eggs",  
    "price": 2.99  
}  
data_1a = {  
    "name": "eggs",  
    "price": 2  
}
```

```
data_2 = {  
    "name": "eggs"  
}  
data_3 = {  
    "name": "eggs",  
    "price": 3.99,  
    "color": "brown"  
}
```

```
table = list(data_1, data_1a, data_2, data_3)
```

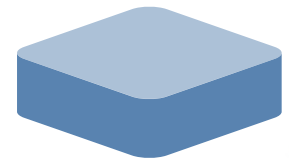


# Parsing JSON: Code

- Parse one record:

```
def parse_demo_data(instance):
    match instance:
        case dict() as data_item:
            match data_item:
                case {
                    'name': str() as name,
                    'price': int(price) | float(price),
                    **extra}:
                    if extra:
                        print (f'found extra values: {extra!r}')
                    # Process data
                    print (f'{name}: {price}')
                case wrong_values:
                    print (f'could not parse properties: {wrong_values!r}')
            case wrong_values:
                print (f'could not parse instance: {wrong_values!r}')
```

```
schema = {
    "type": "object",
    "properties": {
        "price": {"type": "number"},
        "name": {"type": "string"},
    },
}
```

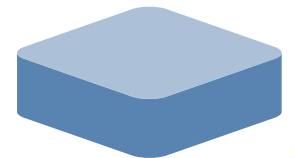


# Parsing JSON: Code

- Parse multiple records:

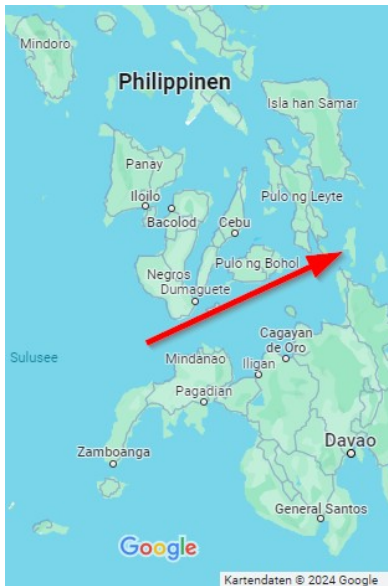
```
def parse_list_data(many_instances):  
    match many_instances:  
        case list() as data_list:  
            for instance in data_list:  
                parse_demo_data(instance)  
        case wrong_data:  
            print (f'unknown data format: {wrong_data!r}')
```

```
table = [ {'name': 'eggs', 'price': 2.99},  
          {'name': 'eggs', 'price': 2},  
          {'name': 'eggs'},  
          {'color': 'brown', 'name': 'eggs', 'price': 3.99}]
```



# Parsing GeoJSON: Data

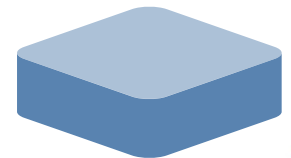
- Example:



```
# From https://geojson.org/  
GEOJSON_TYPES = (  
    'Feature',  
    'FeatureCollection',  
    'Point',  
    # more  
)
```

```
geojson_data_1 = {  
    "type": "Feature",  
    "geometry": {  
        "type": "Point",  
        "coordinates": [125.6, 10.1]  
    },  
    "properties": {  
        "name": "Dinagat Islands"  
    }  
}
```

# GEOJSON

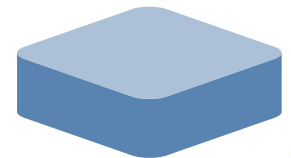


# Parsing GeoJSON: Code

- Parse a GeoJSON record:

```
def parse_geojson_data(instance):
    match instance:
        case {'type': obj_type, **other_members} as obj:
            print (f'new object of type {obj_type!r}')
            if obj_type not in GEOJSON_TYPES:
                print (f'this type is not a valid GeoJSON type')
                return None
            for obj_member in other_members.items():
                match obj_member:
                    case ('geometry', geometry):
                        print (ffound geometry {geometry!r}')
                    case ('properties', properties):
                        print (ffound properties {properties!r}')
                    case wrong_member:
                        print (f'could not parse member: {wrong_member!r}')
        case wrong_values:
            print (f'could not parse object: {wrong_values!r}')
```

```
geojson_data_1 = {
    "type": "Feature",
    "geometry": {
        "type": "Point",
        "coordinates": [125.6, 10.1]
    },
    "properties": {
        "name": "Dinagat Islands"
    }
}
```

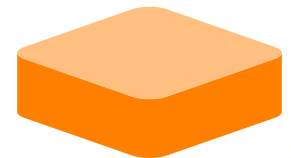


# Parsing XML: Data

- Example:

```
<?xml version="1.0"?>
<data>
  <country name="Liechtenstein">
    <rank>1</rank>
    <year>2008</year>
    <gdppc>141100</gdppc>
    <neighbor name="Austria" direction="E"/>
    <neighbor name="Switzerland" direction="W"/>
  </country>
  <country name="Singapore">
    <rank>4</rank>
    <year>2011</year>
    <gdppc>59900</gdppc>
    <neighbor name="Malaysia" direction="N"/>
  </country>
```

```
<country name="Panama">
  <rank>68</rank>
  <year>2011</year>
  <gdppc>13600</gdppc>
  <neighbor name="Costa Rica" direction="W"/>
  <neighbor name="Colombia" direction="E"/>
</country>
</data>
```



From <https://docs.python.org/3/library/xml.etree.elementtree.html>

# Parsing XML: Code

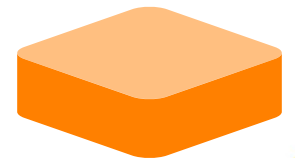
- Parse countries:

```
def parse_countries_1(tree):
    countries = {}
    for child in tree:
        match child:
            case Element(tag='country',
                          attrib={'name': name}) as country:
                for child in country:
                    neighbors = {}
                    match child:
                        case Element(tag='rank', text=rank):
                            rank = int(rank)
                        case Element(tag='year', text=year):
                            year = int(year)
                        case Element(tag='gdppc', text=gdppc):
                            gdppc = float(gdppc)
                        case Element(
                            tag='neighbor',
                            attrib={'name': nb_name,
                                    'direction': nb_direction}):
                            neighbors[nb_name] = nb_direction
```

```
<country name="Panama">
  <rank>68</rank>
  <year>2011</year>
  <gdppc>13600</gdppc>
  <neighbor name="Costa Rica" direction="W"/>
  <neighbor name="Colombia" direction="E"/>
</country>
```

```
        case wrong_data:
            raise TypeError(
                error_string(wrong_data,
                             'country element'))
    countries[name] = dict(
        rank=rank,
        year=year,
        gdppc=gdppc,
        neighbors=neighbors,
    )
    case wrong_data:
        raise TypeError(error_string(wrong_data,
                                     'country'))

return countries
```



Note: This does not detect missing child elements. See repo for a version which does.



# Parsing ASTs: Data

- Example function and AST:

```
def example(x):  
    if isinstance(x, int) and x > 12:  
        print ('Int > 12: {x}')    elif isinstance(x, float):  
        print ('Float: {x}')    elif isinstance(x, str) and len(x) > 5:  
        print ('String with more than 5 chars: {x}')    else:  
        print ('Unknown type: {x}')
```

- **Idea:** Refactor this into an equivalent function using the *match* statement

```
Module(  
  body=[  
    FunctionDef(  
      name='example',  
      args=arguments(  
        posonlyargs=[],  
        args=[  
          arg(arg='x')],  
        kwonlyargs=[],  
        kw_defaults=[],  
        defaults=[]),  
      body=[  
        If(  
          test=BoolOp(  
            op=And(),  
            values=[  
              Call(  
                func=Name(id='isinstance', ctx=Load()),  
                args=[  
                  Name(id='x', ctx=Load()),  
                  Name(id='int', ctx=Load()),  
                ],  
                keywords=[]),  
              Compare(  
                left=Name(id='x', ctx=Load()),  
                ops=[  
                  Gt(),  
                ],  
                comparators=[  
                  Constant(value=12)])),  
            ]),  
          body=[  
            Expr(  
              value=Call(  
                func=Name(id='print', ctx=Load()),  
                args=[  
                  Constant(value='Int > 12: {x}')],  
                Keywords=[]),  
            ]),  
          ]),  
        ]),  
    ]),  
  ... lots more
```



# Parsing ASTs: Code

- Parse function AST & refactor:

```
def if_match_refactor(fct):
    source = inspect.getsource(fct)
    tree = ast.parse(source) # AST.Module
    # Find functions
    for node in tree.body:
        match node:
            case FunctionDef(
                name,
                args=arguments(args=[arg(arg=varname)])):
                # Refactor function name with variable varname
                function_refactor(node, name, varname)
            case _:
                # skip
                pass
    return tree
```

```
def function_refactor(fct_node, name, varname):
    # Find if-elif-else
    new_body = []
    for node in fct_node.body:
        match node:
            case If(test, body, orelse):
                # Scan for match cases written as if-elif chain
                cases, orelse = scan_if(node, varname,
                                       test, body, orelse)
                if cases:
                    # Refactor to use match instead
                    node = if_refactor(node, varname,
                                       cases, orelse)
            case _:
                # Not an if node
                break
    new_body.append(node)
    fct_node.body = new_body
```



# Parsing ASTs: Code

- Scan *if-elif-else* chains recursively:

```
def scan_if(node, varname, test, body, orelse, cases=None):
    # Trick for non-capturing vars
    class params:
        pass
    params.varname = varname
    # Check for possible match refactoring if-candidate
    match test:
        case Call(func=Name(id='isinstance'),
                  args=[Name(id=params.varname),
                        Name(id=typename)]):
            # isinstance(varname, typename)
            new_case = (varname, typename, None, body)
        case BoolOp(op=And(),
                   values=[
                       Call(func=Name(id='isinstance'),
                             args=[Name(id=params.varname),
                                     Name(id=typename)]),
                       condition
                   ]):
            # isinstance(varname, typename) and condition
            new_case = (varname, typename, condition, body)
```

```
        case _:
            # Unsupported if-variant
            return cases, node
    # Add new case
    if cases is None:
        cases = []
    cases.append(new_case)
    # Recursively check whether there are more if-elif cases
    match orelse:
        case [If(elif_test, elif_body, elif_orelse) as elif_node]:
            cases, orelse = scan_if(
                elif_node, varname,
                elif_test, elif_body, elif_orelse,
                cases)
        case _:
            pass
    return cases, orelse
```



# Parsing ASTs: Code

- Refactor *if-elif-else* into *match* statements:

```
def if_refactor(if_node, varname, cases, orelse):
    # Build match_cases
    case_nodes = []
    match_case(
        pattern=MatchClass(
            cls=Name(id=typename),
            patterns=[],
            kwd_attrs=[],
            kwd_patterns=[]),
            guard=condition,
            body=body)
    for (varname, typename, condition, body) in cases
    ]
    case_nodes.append(
        match_case(
            pattern=MatchAs(),
            body=orelse)
    )
```

```
# Build Match node
match_node = Match(
    subject=Name(id=varname),
    cases=case_nodes,
)
return match_node
```



# Parsing ASTs: Fully automatic refactoring


- Example:

```
def example(x):  
    if isinstance(x, int) and x > 12:  
        print ('Int > 12: {x}')  
    elif isinstance(x, float):  
        print ('Float: {x}')  
    elif isinstance(x, str) and len(x) > 5:  
        print ('String with more than 5 chars: {x}')  
    else:  
        print ('Unknown type: {x}')
```

```
def example(x):  
    match x:  
        case int() if x > 12:  
            print('Int > 12: {x}')  
        case float():  
            print('Float: {x}')  
        case str() if len(x) > 5:  
            print('String with more than 5 chars: {x}')  
        case _:  
            print('Unknown type: {x}')
```



# What about performance ?

Name	Match Variant	If Variant	Notes
Int	111	52	
Int capvars	116 (generic) 548 (inlined)	56	<i>Avoid inlined capvars (?)</i>
List ints	61	82	
Str	102	54	
Int guards	143	49	
Complex obj with capvars	663 (generic) 3410 (inlined)	306	<i>Avoid inlined capvars (?)</i>
Float or int	104	62	

Using Python 3.11. All numbers refer to nanoseconds. See repo for benchmark details

# Performance Bug with Captured Variables ?

Name	Match Variant	If Variant	Notes
Int	111	52	
Int capvars	116 (generic) 548 (inlined)	56	<i>Avoid inlined capvars (?)</i>
List ints	61	82	<i>Generic: case int() as a: Inlined: case int(a):</i>
Str	102	54	
Int guards	143	49	
Complex obj with capvars	663 (generic) 3410 (inlined)	306	<i>Avoid inlined capvars (?)</i>
Float or int	104	62	

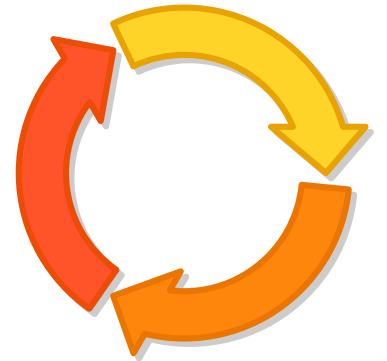
Using Python 3.11. All numbers refer to nanoseconds. See repo for benchmark details

# Benchmarking: Conclusions

- *match* is **about 2 times slower** than *if* in most cases
- *match* **shines** for **sequence matching**

```
match obj:  
    case [a, b, c]:  
        do_something()
```

- Python 3.10 – 3.12: still first implementation of the PEPs
  - No significant optimizations have been applied since
  - Some parts appear to have performance bugs (inlined capvars)





# Overall conclusion: Advanced parsing with *match*

- *match* isn't fast (yet)
  - Lot's of room for improvements
- *match* is **great for complex parsing tasks** where readability counts:

```
if isinstance(obj, int) and obj > 20:
    value = obj
elif (isinstance(obj[0], int) and
      isinstance(obj[1], int) and
      isinstance(obj[2], dict) and
      ('abc' in obj[2]) and
      isinstance(obj[2]['abc'], int) and
      isinstance(obj[3], tuple) and
      isinstance(obj[3][0], int) and
      isinstance(obj[3][1], int) and
      isinstance(obj[3][2], int)):
    [a, b, q1, (d, e, f)] = obj
    c = q1['abc']
else:
    pass
```

```
match obj:
    case int() as value if value > 20:
        pass
    case [int() as a, int() as b,
          {'abc': int() as c},
          (int() as d, int() as e, int() as f)]:
        pass
    case _:
        pass
```



**Main takeaway:** Never stop **learning** and **trying out new things**



These  
are  
exciting  
times

**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](mailto:mal@egenix.com)

Phone: +49 211 9304112

Fax: +49 211 3005250

Web: <https://www.egenix.com/>

LinkedIn:



# References

- Some content taken from:
  - <https://py-code.org/stats>
  - [https://de.wikipedia.org/wiki/Tango\\_Desktop\\_Project](https://de.wikipedia.org/wiki/Tango_Desktop_Project)
- 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