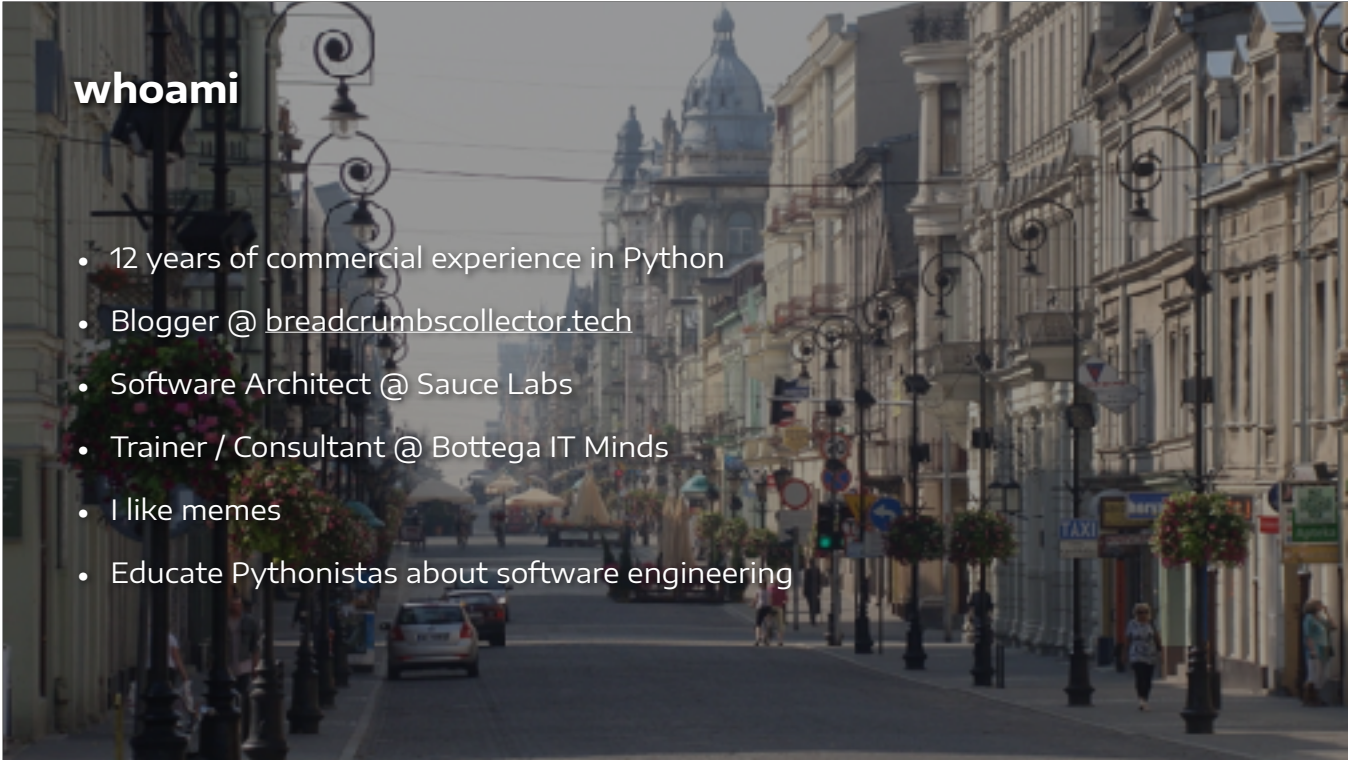


Having fun with Pydantic and pattern matching

Sebastian Buczyński





whoami

- 12 years of commercial experience in Python
- Blogger @ breadcrumbscollector.tech
- Software Architect @ Sauce Labs
- Trainer / Consultant @ Bottega IT Minds
- I like memes
- Educate Pythonistas about software engineering

I've been professionally working with Python for over 12 years by now

I blog under breadcrumbscollector.tech

I work as Software Architect in Sauce Labs and part-time I do some trainings & consulting with another company, Bottega IT Minds.

Apart from computers, I like memes so you can expect some on the slides.

My personal mission is to spread knowledge about software engineering amongst Pythonistas.

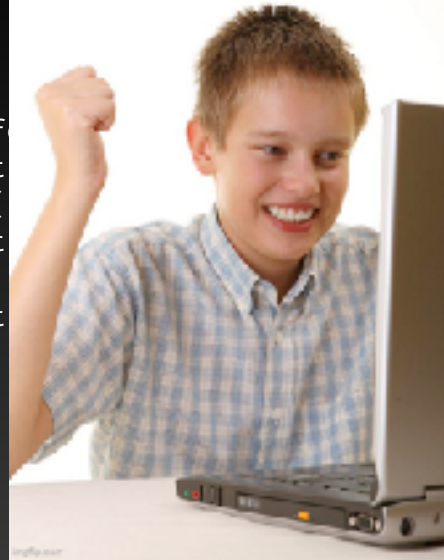
But today we're going to talk about something different...

```
match latte:
    case Coffee():
        print("It's caffee latte or latte machiato!")
    case Tea():
        print("It's Matcha Latte")
    case _:
        print("Huh, no idea what it is")
```

...namely this beauty - match..case syntax and pattern matching in general.
Now, some people seeing this are like

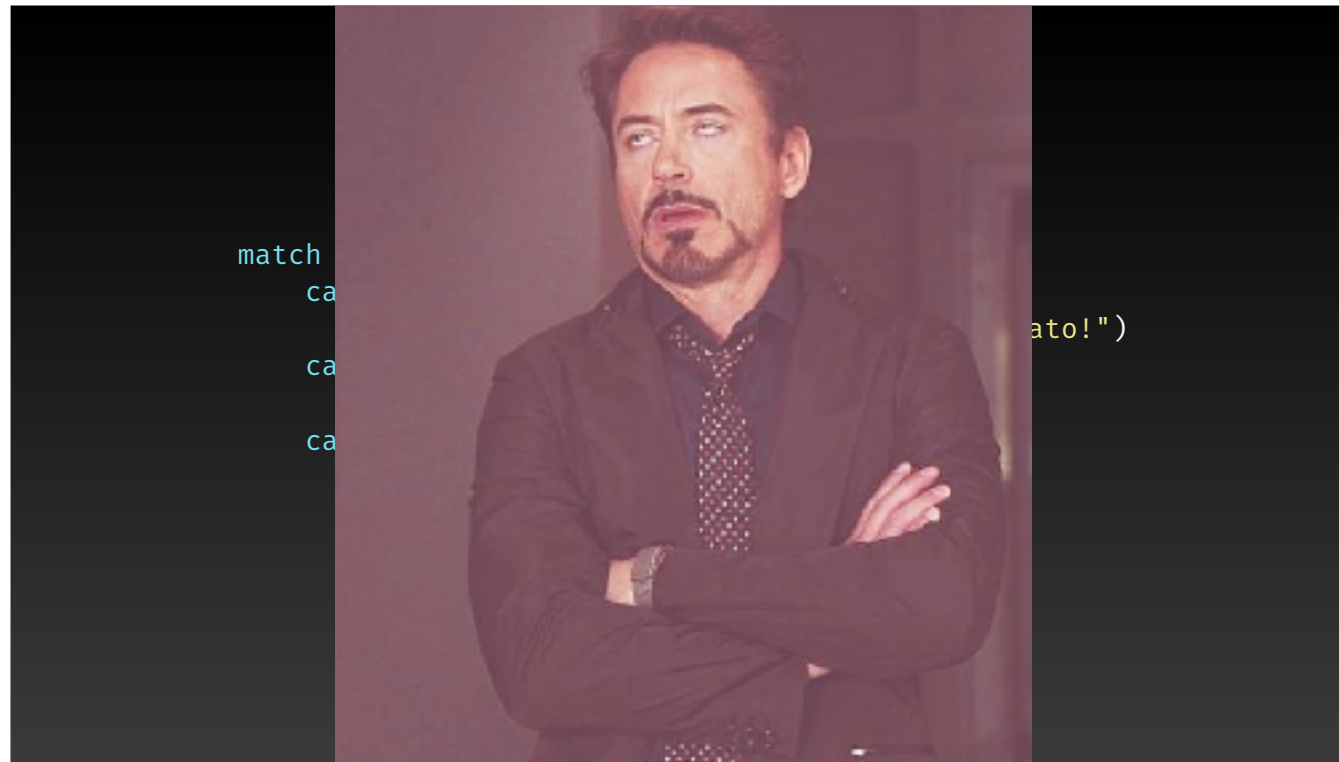
PYTHON FINALLY
GETTING SWITCH!

```
match latte:  
    case Coff  
        print  
    case Tea(  
        print  
    case _:  
        print
```



```
        e machiato!")
```

PYTHON FINALLY GETTING SWITCH..CASE! YEAH!
Except...



...it's not switch..case. match..case and pattern matching in general is SO, SO much more.
And I will show you what it means during the talk.

BTW, who already uses match..case syntax?

```
price = 200  
match price:  
    ...
```

Let's start with a really short introduction to those who don't know or use this syntax yet. Let's meet this fella together, shall we?

Say we have an integer 200 named price. We then use match on the name...

```
price = 200

match price:
    case 100:
        print("Cheap!")
    case 200:
        print("Expensive!")
    case _:
        print("No idea")
```

...providing multiple cases to match against.

```
price = 200

match price:
    case 100:
        print("Cheap!")
    case 200:
        print("Expensive!")
    case _:
        print("No idea")
```

Say, for starters let's see if price is 100. If it is, then we print „Cheap”


```
price = 200

match price:
    case 100:
        print("Cheap!")
    case 200:
        print("Expensive!")
    case _:
        print("No idea")
```

Then let's try with another value 200, but this time we print „Expensive!“

```
price = 200

match price:
    case 100:
        print("Cheap!")
    case 200:
        print("Expensive!")
    case _:
        print("No idea")
```

And finally, we have „catch-all” - if no other case was matched successfully, then we land here.

What we'll see? #1

```
price = 200

match price:
    case 100:
        print("Cheap!")
    case 200:
        print("Expensive!")
    case _:
        print("No idea")
```

Let's check how we are doing so far. What will we see on the screen?

What we'll see? #2

```
price = 2137

match price:
    case 100:
        print("Cheap!")
    case 200:
        print("Expensive!")
    case _:
        print("No idea")
```

And now, when price is 2137?

So far, this is pretty much all that switch...case in other programming languages can do.
BUT this is also where possibilities of switch...case end. And we're even barely warmed up.

What we'll see? #3

```
data = (1, 2, 3)

match data:
    case {}:
        print("It's a dict!")
    case tuple():
        print("Huh, a tuple!")
    case _:
        print("No idea")
```

Let's try matching on type. There we have a 3-element tuple called „data“. What we'll see on the screen?

What we'll see? #4

```
data = {"name": "Sebastian"}

match data:
    case {}:
        print("It's a dict!")
    case tuple():
        print("Huh, a tuple!")
    case _:
        print("No idea")
```

It's „huh, a tuple“.

How about now?

What we'll see? #5

```
data = {"name": "Sebastian"}

match data:
    case {"name": _}:
        print("That's a dict with 'name'")
    case {}:
        print("It's a dict!")
    case _:
        print("No idea")
```

We should see „It's a dict!“

And now?

What we'll see? #6

```
data = {"name": "Sebastian"}

match data:
    case {}:
        print("It's a dict!")
    case {"name": _}:
        print("That's a dict with 'name'")
    case _:
        print("No idea")
```

We've got „That's a dict with ,name'"" over there.

How about this one?

What we'll see? #6

```
data = {"name": "Sebastian"}

match data:
    case {"name": "Sebastian"}:
        print("It's a dict, with name 'Sebastian'")
    case {"name": "Sebastian"}:
        print("It's a dict, with name 'Sebastian'")
    case _:
        print("No idea")
```

Ordering of ,cases' is significant

We'll see It's a dict even though we have another case matching the value more accurately.

Which leads us to one of key take aways from this talk...

If you want to use match...case, remember that ORDERING OF CASES IS SIGNIFICANT.

Let's see what happens if you neglect it



This is what happens when you neglect ordering...

...just like with exceptions

```
try:
    ...
except Exception:
    ...
except ValueError:
    ... # dead code!
```

This is also very similar to exceptions handling. „except” blocks are evaluated from top to the bottom.

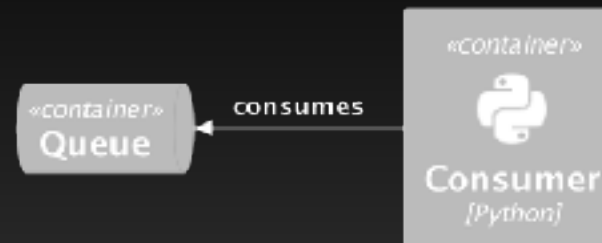
If you put a superclass of an exception at the top, it will swallow everything like a black hole and make contents of other „except” blocks a dead code.



Alright, this match...case is a neat gimmick. Can it actually be useful in production?

Let's see.

Consuming stream of events



Assume we're consuming a never-ending stream of events or messages if you prefer.

We take them from some kind of queue - it can be RabbitMQ, Redis, Kafka, some cloud-native solution - you name it. Doesn't matter.

What matters is that we fetch messages from this stream and handle it one by one.

How does an individual message look like?

Consuming stream of events: data

```
{"type": "AccountCreated", "id": "64160ccb", "name": "Matthew"}
```

Technically such things are passed as binaries, but let's assume we use JSON encoding and each piece can be decoded to a Python dictionary - like the one you see on the screen.

The typical message will contain „type” field to inform us what kind of information is this and some accompanying data, like „id” and „name” in this case.

Consuming stream of events: data

```
{"type": "AccountCreated", "id": "64160ccb", "name": "Matthew"}  
{"type": "AccountUpdated", "id": "bb915f85", "old_status": "trial",  
"new_status": "paid"}  
{"type": "AccountDeleted", "id": "57ae55c5"}
```

There will be quite a few types of such messages - along with „AccountCreated” we also have „AccountUpdated” or AccountDeleted. Each has its own unique set of fields.

Consuming stream of events: data

```
{"type": "AccountCreated", "id": "64160ccb", "name": "Matthew"}  
{"type": "AccountUpdated", "id": "bb915f85", "old_status": "trial",  
"new_status": "paid"}  
{"type": "AccountDeleted", "id": "57ae55c5"}  
{"type": "AccountUpdated", "id": "fede79b2", "old_status": "paid",  
"new_status": "trial"}  
{"type": "AccountCreated", "id": "23826148", "name": "John"}
```

Messages of different types are mixed in this single stream of data and they concern many different users.

Consuming stream of events: data

```
{ "type": "AccountCreated", "id": "64160ccb", "name": "Matthew" }  
{ "type": "AccountUpdated", "id": "bb915f85", "old_status": "trial",  
  "new_status": "paid" }  
{ "type": "AccountDeleted", "id": "57ae55c5" }  
{ "type": "AccountUpdated", "id": "fede79b2", "old_status": "paid",  
  "new_status": "trial" }  
{ "type": "AccountCreated", "id": "23826148", "name": "John" }  
{ "username": "DrWho" }
```

To make this example a bit more interesting and also closer to reality, let's also assume that some messages will have unexpected format.

For example, the last one you can see has just „username” field with no „type”. We'll have to deal with it - somehow.

```
def handle(payload: dict) → None:  
    ... # let's code this!
```

Okay, so we're familiar with input data.

Our actual task is to code handling of each message.

We know we get them one by one as dict each, so let's do it!

```
def handle(payload: dict) → None:
    # code like it's 2020 😎
    if payload.get("type") == "AccountCreated":
        ...
    elif payload.get("type") == "AccountDeleted":
        ...
    elif payload.get("type") == "AccountUpdated":
        ...
    else:
        print("Omg, what is this?!")
```

If we're stuck with Python older than 3.10 or don't know `match..case`, we could do it like this.

We bash together some `if`, `elif`, `else` statements, combine it with checks for optional fields using „`get`” method and we're done.

Except...

```
def handle(payload: dict) → None:
    # code like it's 2020 😊
    # ...and pretend we're in control 🤖🤖🤖
    if payload.get("type") == "AccountCreated":
        ...
    elif payload.get("type") == "AccountDeleted":
        ...
    elif payload.get("type") == "AccountUpdated":
        if payload["new_status"] == "paid":
            ...
        elif payload["new_status"] == "trial":
            ...
    else:
        print("Omg, what is this?!")
```

...this can escalate and doesn't scale too well with a number of messages or special cases we'd like to tell from another.

For example, type of message, „AccountUpdated“, can require a different handling. For example, we distinguish updating to paid and trial account to treat these situation separately.

A question for you - is there anything else you don't like in this solution?

```
def handle(payload: dict) → None:
    # HAMMER TIME! ↖
    match payload:
        case _:
            if payload.get("type") == "AccountCreated":
                ...
            elif payload.get("type") == "AccountDeleted":
                ...
            elif payload.get("type") == "AccountUpdated":
                if payload["new_status"] == "paid":
                    ...
                elif payload["new_status"] == "trial":
                    ...
            else:
                print("Omg, what is this?!")
```

Let's refactor it with match...case then.

First, we introduce match on payload with just a default case. We'll do this step by step.

Now, everything falls under this catch-all, underscore. The function is even worse than it used to be, because we've added 2 levels of indentation, but bear with me.

```
def handle(payload: dict) → None:
    # HAMMER TIME! ↖
    match payload:
        case {"type": "AccountCreated"}:
            ...
        case _:
            if payload.get("type") == "AccountDeleted":
                ...
            elif payload.get("type") == "AccountUpdated":
                if payload["new_status"] == "paid":
                    ...
                elif payload["new_status"] == "trial":
                    ...
            else:
                print("Omg, what is this?!")
```

We change a call to „get” method with comparing string into a pattern...

```
def handle(payload: dict) -> None:
    # HAMMER TIME! ⚡
    match payload:
        case {"type": "AccountCreated"}:
            ...
        case {"type": "AccountDeleted"}:
            ...
        case _:
            if payload.get("type") == "AccountUpdated":
                if payload["new_status"] == "paid":
                    ...
                elif payload["new_status"] == "trial":
                    ...
            else:
                print("Omg, what is this?!")
```

For AccountDeleted message we do exactly the same thing

```
def handle(payload: dict) → None:
    # HAMMER TIME! ⚡
    match payload:
        case {"type": "AccountCreated"}:
            ...
        case {"type": "AccountDeleted"}:
            ...
        case {"type": "AccountUpdated"}:
            if payload["new_status"] == "paid":
                ...
            elif payload["new_status"] == "trial":
                ...
        case _:
            print("Omg, what is this?!")
```

AccountUpdated is dealt with in two steps. The first one is the same. We could leave nested if...elif but why don't we take another step...?


```
def handle(payload: dict) -> None:
    # HAMMER TIME! ↖
    match payload:
        case {"type": "AccountCreated"}:
            ...
        case {"type": "AccountDeleted"}:
            ...
        case {"type": "AccountUpdated", "new_status": "paid"}:
            ...
        case {"type": "AccountUpdated", "new_status": "trial"}:
            ...
        case _:
            print("Omg, what is this?!")
```

This way, our refactoring is complete and we eliminated some code nesting.

But you know what? I still don't like the outcome. It smells a bit like a dead snake here.

```
def handle(payload: dict) -> None:
    # HAMMER TIME!
    match payload:
        case {"type": "AccountCreated"}:
            ...
        case {"type": "AccountDeleted"}:
            ...
        case {"type": "AccountUpdated", "new_status": "paid"}:
            ...
        case {"type": "AccountUpdated", "new_status": "trial"}:
            ...
        case _:
            print("Omg, what is this?!")
```



My problem (and yours should be too) is with passing dictionaries around. This is so Python2-ish.

Back in the day, we passed around lots of dicts and it was always difficult to figure out what's inside.

Let's fix it using Pydantic.

Pydantic model

```
{"type": "AccountCreated", "id": "64160ccb", "name": "Mateusz"}
```

We know the schemas of expected messages. For example, we know AccountCreated message has type, id and name fields

Pydantic model

```
data = {"type": "AccountCreated", "id": "64160ccb", "name": "Mateusz"}

class AccountCreated(BaseModel):
    type: str
    id: str
    name: str

AccountCreated(**data)
# AccountCreated(type='AccountCreated', id='64160ccb', name='Mateusz')
```

So we can write Pydantic model describing that.

Thanks for that we get validation, typing and type coercion.

Pydantic model with a constant

```
data = {"type": "AccountCreated", "id": "64160ccb", "name": "Mateusz"}

class AccountCreated(BaseModel):
    type: Literal["AccountCreated"]
    id: str
    name: str

AccountCreated(type="BAZINGA", id="123", name="Seba")
# raises exception
```

Let's use a little trick that will be helpful later - let's specify value of „type” because there is only ONE allowed value - AccountCreated, we can describe it using typing.Literal.

Pydantic models

```
class AccountCreated(BaseModel):  
    ...  
  
class AccountDeleted(BaseModel):  
    ...  
  
class AccountUpdated(BaseModel):  
    type: Literal["AccountUpdated"]  
    id: str  
    old_status: Literal["trial", "paid"]  
    new_status: Literal["trial", "paid"]
```

In a same way, we define models for all other known messages

```
def handle(payload: dict) → None:
    match payload:
        case {"type": "AccountCreated"}:
            ...
        case {"type": "AccountDeleted"}:
            ...
        case {"type": "AccountUpdated", "new_status": "paid"}:
            ...
        case {"type": "AccountUpdated", "new_status": "trial"}:
            ...
        case _:
            print("Omg, what is this?!")
```

Now we can come back to our code with match...case and once we recognize the message type, we can create an instance of matching model


```
def handle(payload: dict) → None:
    match payload:
        case {"type": "AccountCreated"}:
            event = AccountCreated(**payload)
        case {"type": "AccountDeleted"}:
            event = AccountDeleted(**payload)
        case {"type": "AccountUpdated", "new_status": "paid"}:
            event = AccountUpdated(**payload)
        case {"type": "AccountUpdated", "new_status": "trial"}:
            event = AccountUpdated(**payload)
        case _:
            print("Omg, what is this?!")
```

But you know what? Pydantic model also KINDA described a pattern - it contains fields and even has some literal values for them... And the cool thing is we can do this pattern matching thing with just Pydantic alone.


```
def handle(payload: dict) → None:
    supported_model = (
        AccountCreated | AccountDeleted | AccountUpdated
    )

    match payload:
        case {"type": "AccountCreated"}:
            ...
```

First, we create a union of handled models to express the fact the message can be one of them.

```
def handle(payload: dict) → None:
    supported_model = (
        AccountCreated | AccountDeleted | AccountUpdated
    )
    adapter = TypeAdapter(supported_model) 

    match payload:
        case {"type": "AccountCreated"}:
            ...
```

Then, we use some Pydantic magic - we create an instance of TypeAdapter, that accepts our union as an argument.

```
def handle(payload: dict) → None:
    supported_model = (
        AccountCreated | AccountDeleted | AccountUpdated
    )
    adapter = TypeAdapter(supported_model)
    event = adapter.validate_python(payload)
    # event will be an instance of a supported model!

    match payload:
        case {"type": "AccountCreated"}:
            ...
```

TypeAdapter can convert raw dictionaries to a specific object. In this case, output will be one of supported models!

```
def handle(payload: dict) → None:
    supported_model = (
        AccountCreated | AccountDeleted | AccountUpdated
    )
    adapter = TypeAdapter(supported_model)
    event = adapter.validate_python(payload)

    match event:
        case AccountCreated():
            ...
        case AccountUpdated(new_status="paid"):
            ...
        case AccountUpdated(new_status="trial"):
            ...
```

Finally, we can match on types and objects' attributes. Plus we now have data conversion, validation and full typing.

Except...this code will explode in production.

Can you see why?

```
def handle(payload: dict) → None:
    supported_model = (
        AccountCreated | AccountDeleted | AccountUpdated | Any
    )
    adapter = TypeAdapter(supported_model)
    event = adapter.validate_python(payload)

    match event:
        case AccountCreated():
            ...
        case AccountUpdated(new_status="paid"):
            ...
        case AccountUpdated(new_status="trial"):
            ...
        case _:
            print("Omg, what is this?!")
```

We need Any added at the end of union. There are situations when nothing will match and then TypeAdapter will raise an exception.

Since it went so well and easy with Pydantic models, we can push it even harder.

de



ny

Using typing, of course.

```
def handle(payload: dict) -> None:
    supported_model = (
        AccountCreated | AccountDeleted | AccountUpdated | Any
    )
    adapter = TypeAdapter(supported_model)
    event = adapter.validate_python(payload)

    match event:
        case AccountCreated():
            ...
        case AccountUpdated(new_status="paid"):
            ...
        case AccountUpdated(new_status="trial"):
            ...
        case _:
            print("Omg, what is this?!")
```

For starters, we can create dedicated types for these two fellas

```
class AccountUpdated(BaseModel):  
    type: Literal["AccountUpdated"]  
    id: str  
    old_status: Literal["trial", "paid"]  
    new_status: Literal["trial", "paid"]
```

So single AccountUpdated becomes


```
class AccountUpdatedToPaid(BaseModel):  
    type: Literal["AccountUpdated"]  
    id: str  
    old_status: Literal["trial"]  
    new_status: Literal["paid"]  
  
class AccountUpdatedToTrial(BaseModel):  
    type: Literal["AccountUpdated"]  
    id: str  
    old_status: Literal["paid"]  
    new_status: Literal["trial"]
```

...AccountUpdatedToPaid and AccountUpdatedToTrial.

Since we're handling it differently, why not use such a trick?

```
def handle(payload: dict) → None:
    supported_model = (
        ... | Any
    )
    adapter = TypeAdapter(supported_model)
    event = adapter.validate_python(payload)

    match event:
        case AccountCreated():
            ...
        case AccountUpdatedToPaid():
            ...
        case AccountUpdatedToTrial():
            ...
        case _:
            print("Omg, what is this?!")
```

Now that we reduced all different situations into separate types, this gives us another very interesting possibility.

We can use `@functools.singledispatch` that has been with us since Python 3.4

```
import functools

@functools.singledispatch
def event_handler(event: Any) → None:
    print("Omg, what is this?!")
```

First, we define our catch-all handler for unexpected types.

```
import functools

@functools.singledispatch
def event_handler(event: Any) → None:
    print("Omg, what is this?!")

@event_handler.register
def handle_account_created(event: AccountCreated) → None:
    ...

@event_handler.register
def handle_account_updated_to_paid(event: AccountUpdatedToPaid) → None:
    ...
```

Then we „register” handlers for various specialized types. For example, each message type will get its own handler.

```
def handle(payload: dict) -> None:
    supported_model = (
        AccountCreated
        | AccountDeleted
        | AccountUpdatedToTrial
        | AccountUpdatedToPaid
        | Any
    )
    adapter = TypeAdapter(supported_model)
    event = adapter.validate_python(payload)

    event_handler(event)
```

Eventually, our initial function is reduced to THIS.

We use Pydantic models to validate and convert message to a very specific type that's later handled using `functools singledispatch`.

We could also move a union of supported models outside of the function to be in a situation when no code changes are required here whenever we add or remove a message type support.

Time to wrap up the entire talk.

match...case on types

```
match latte:  
    case Coffee():  
        print("It's caffee latte or latte machiato!")  
    case Tea():  
        print("It's Matcha Latte")  
    case _:  
        print("Huh, no idea what is it")
```

We started off from match...case statement, which is much more powerful than switch in other languages because it can also match on types...

match...case on attributes' values

```
match event:  
    case AccountCreated():  
        ...  
    case AccountUpdated(new_status="paid"):  
        ...  
    case AccountUpdated(new_status="trial"):  
        ...  
    case _:  
        print("Omg, what is this?!")
```

...or objects' attributes. Switch can not do that.

pattern matching using Pydantic

```
class AccountCreated(BaseModel):
    type: Literal["AccountCreated"]
    id: str
    name: str

class AccountDeleted(BaseModel):
    type: Literal["AccountDeleted"]
    id: str

supported_model = (
    AccountCreated | AccountDeleted | AccountUpdated
)
adapter = TypeAdapter(supported_model)
event = adapter.validate_python(payload)
```

Then we saw how to pattern-match data and convert it to a Pydantic model using union and TypeAdapter.

What is wrong in this snippet?

pattern matching using Pydantic

```
class AccountCreated(BaseModel):
    type: Literal["AccountCreated"]
    id: str
    name: str

class AccountDeleted(BaseModel):
    type: Literal["AccountDeleted"]
    id: str

supported_model = (
    AccountCreated | AccountDeleted | AccountUpdated | Any
)
adapter = TypeAdapter(supported_model)
event = adapter.validate_python(payload)
```

Missing Any. We should be ready for unexpected. We can do it either by using Any or explicit exception handling.

Dispatch based on type using @singledispatch

```
import functools

@functools.singledispatch
def event_handler(event: Any) -> None:
    print("Omg, what is this?!")

@event_handler.register
def handle_account_created(event: AccountCreated) -> None:
    ...
```

Finally, we saw how singledispatch can make our code extensible by calling handlers specified for a given type.

Start writing classes

'cause types are awesome!

Sebastian Buczyński

breadcrumbscollector.tech

