Digon.IO

How Not to Use Python Match Cases

Fabian Preiß

About Me

Fabian Preiß

- Technical Co-Founder @ Digon.IO GmbH
- Long time Python Developer (>12 years)
- Background in DataScience (physics)

Introduction

`match/case` in Python

- Related `switch` and `case` statements were first proposed to python 2.6 in 2001 (PEP: 275)
- 20 years later, Python 3.10 was released with the `match` and `case` statements
- The feature is one of the most extensive additions to Python in recent years

Introduction

A most basic example

```python
def accept_cookie(
    response: str,
) -> str:
    if response == "yes":
        return "🍪"
    elif response == "no":
        return "No problem."
    else:
        return "I didn't get that."
```

```python
def accept_cookie(
    response: str,
) -> str:
    match response:
        case "yes":
            return "🍪"
        case "no":
            return "No problem."
        case _:
            return "I didn't get that."
```

Introduction

A most basic example

```python
def accept_cookie(
    response: str,
) -> str:
    if response == "yes":
        return "🍪"
    elif response == "no":
        return "No problem."
    else:
        return "I didn't get that."
```

```python
def accept_cookie(
    response: str,
) -> str:
    match response:
        case "yes":
            return "🍪"
        case "no":
            return "No problem."
        case _:
            return "I didn't get that."
```

Is this obviously better?

Introduction

A most basic example

```python
def accept_cookie(
    response: str,
) -> str:
    if response == "yes":
        return "🍪"
    elif response == "no":
        return "No problem."
    else:
        return "I didn't get that."
```

```python
def accept_cookie(
    response: str,
) -> str:
    match response:
        case "yes":
            return "🍪"
        case "no":
            return "No problem."
        case _:
            return "I didn't get that."
```

Is this obviously better?

Pick your poison 🤷

Introduction

But there is much more…

About 13% of the current python grammar is dedicated to the `match/case` syntax alone

Introduction

So what does `match/case` promise?

- Alternative to `if/elif/else` chains
- Less nesting and more readable code
- Destructuring objects into variables
- Potential for future performance improvements

Introduction

Does `match/case` deliver? Opinions...

Introduction

Does `match/case` deliver? Opinions…

- Alternative to `if/elif/else` chains?

Introduction

Does `match/case` deliver? Opinions…

- Alternative to `if/elif/else` chains?
  - Most of the time ✅

Introduction

Does `match/case` deliver? Opinions…

- Alternative to `if/elif/else` chains?
  - Most of the time ✅
- Less nesting and more readable code?

Introduction

Does `match/case` deliver? Opinions…

- Alternative to `if/elif/else` chains?
  - Most of the time ✅
- Less nesting and more readable code?
  - It depends ✅/❌

Introduction

Does `match/case` deliver? Opinions…

- Alternative to `if/elif/else` chains?
  - Most of the time ✅
- Less nesting and more readable code?
  - It depends ✅/❌
- Destructuring objects into variables?

Introduction

Does `match/case` deliver? Opinions…

- Alternative to `if/elif/else` chains?
  - Most of the time ✅
- Less nesting and more readable code?
  - It depends ✅/❌
- Destructuring objects into variables?
  - Helpful but complicated at times ✅/❌

Introduction

Does `match/case` deliver? Opinions…

- Alternative to `if/elif/else` chains?
  - Most of the time ✅
- Less nesting and more readable code?
  - It depends ✅/❌
- Destructuring objects into variables?
  - Helpful but complicated at times ✅/❌
- Potential for future performance improvements?

Introduction

Does `match/case` deliver? Opinions…

- Alternative to `if/elif/else` chains?
  - Most of the time ✅
- Less nesting and more readable code?
  - It depends ✅/❌
- Destructuring objects into variables?
  - Helpful but complicated at times ✅/❌
- Potential for future performance improvements?
  - To be seen…

Motivation

So where does it shine?

Motivation

So where does it shine?

Parsing of diverse data structures!

Motivation

So where does it shine?

Parsing of diverse data structures!

```python
JSONScalars: TypeAlias = None | bool | int | float | str

JSONSerializable: TypeAlias = (
    JSONScalars | list["JSONSerializable"] | dict[JSONScalars, "JSONSerializable"]
)


def to_json_serializable(unknown: Any) -> JSONSerializable:
    match unknown:
        case list(values):
            return [to_json_serializable(value) for value in values]
        case dict(values):
            return {key: to_json_serializable(value) for key, value in values.items()}
        case None | bool() | int() | float() | str():
            return unknown
        case _:
            raise ValueError(f"Unsupported type {type(unknown)} for JSON serialization")
```

Motivation

So where does it shine?

Motivation

So where does it shine?

Unpacking of nested and complex data structures!

Motivation

So where does it shine?

Unpacking of nested and complex data structures!

Before Python 3.10, you'd likely write something like this:

Motivation

So where does it shine?

Unpacking of nested and complex data structures!

Before Python 3.10, you'd likely write something like this:

```python
customer_id = stripe_invoice.customer
subscription_id = stripe_invoice.subscription
if len(data := stripe_invoice.lines.data) == 1:
    price_id = data[0].price.id
    product_id = data[0].price.product
    start = data[0].period.start
    end = data[0].period.end
    if customer_id is not None and subscription_id is not None:
        print(customer_id, subscription_id, price_id, product_id)
```

Motivation

So where does it shine?

Unpacking of nested and complex data structures!

Before Python 3.10, you'd likely write something like this:

```python
customer_id = stripe_invoice.customer
subscription_id = stripe_invoice.subscription
if len(data := stripe_invoice.lines.data) == 1:
    price_id = data[0].price.id
    product_id = data[0].price.product
    start = data[0].period.start
    end = data[0].period.end
    if customer_id is not None and subscription_id is not None:
        print(customer_id, subscription_id, price_id, product_id)
```

```
cus_RCWzWxRNmFPtEw sub_1MowQVLkdIwHu7ixeRlqHVzs price_1QHnM1DTc7leUICsAukuDpcs prod_RA7aUpI09bik6W
```

Motivation

So where does it shine?

Unpacking of nested and complex data structures!

Motivation

So where does it shine?

Unpacking of nested and complex data structures!

Now, this becomes:

Motivation

So where does it shine?

Unpacking of nested and complex data structures!

Now, this becomes:

```python
match stripe_invoice:
    case {
        "customer": str(customer_id),
        "subscription": str(subscription_id),
        "lines": {
            "data": [
                {
                    "price": {"id": str(price_id), "product": str(product_id)},
                    "period": {"start": int(start), "end": int(end)},
                }
            ]
        },
    }:
        print(customer_id, subscription_id, price_id, product_id)
```

Motivation

So where does it shine?

Unpacking of nested and complex data structures!

Now, this becomes:

```python
match stripe_invoice:
    case {
        "customer": str(customer_id),
        "subscription": str(subscription_id),
        "lines": {
            "data": [
                {
                    "price": {"id": str(price_id), "product": str(product_id)},
                    "period": {"start": int(start), "end": int(end)},
                }
            ]
        },
    }:
        print(customer_id, subscription_id, price_id, product_id)
```

```
cus_RCWzWxRNmFPtEw sub_1MowQVLkdIwHu7ixeRlqHVzs price_1QHnM1DTc7leUICsAukuDpcs prod_RA7aUpI09bik6W
```

Motivation

So where does it shine?

Unpacking of nested and complex data structures!

Now, this becomes:

```python
match stripe_invoice:
    case {
        "customer": str(customer_id),
        "subscription": str(subscription_id),
        "lines": {
            "data": [
                {
                    "price": {"id": str(price_id), "product": str(product_id)},
                    "period": {"start": int(start), "end": int(end)},
                }
            ]
        },
    }:
        print(customer_id, subscription_id, price_id, product_id)
```

```
cus_RCWzWxRNmFPtEw sub_1MowQVLkdIwHu7ixeRlqHVzs price_1QHnM1DTc7leUICsAukuDpcs prod_RA7aUpI09bik6W
```

Plus we get additional type safety (both runtime & static)

Motivation

So where does it shine?

Unpacking of nested and complex data structures!

Motivation

So where does it shine?

Unpacking of nested and complex data structures!

Equivalent safety during runtime without `match`/`case`, becomes annoying!

Motivation

So where does it shine?

Unpacking of nested and complex data structures!

Equivalent safety during runtime without `match`/`case`, becomes annoying!

```python
if (
    isinstance(stripe_invoice, dict)
    and isinstance(stripe_invoice.get("customer"), str)
    and isinstance(stripe_invoice.get("subscription"), str)
    and isinstance(stripe_invoice.get("lines"), dict)
    and isinstance(stripe_invoice["lines"].get("data"), list)
    and len(stripe_invoice["lines"]["data"]) == 1
    and isinstance(stripe_invoice["lines"]["data"][0], dict)
    and isinstance(stripe_invoice["lines"]["data"][0].get("price"), dict)
    and isinstance(stripe_invoice["lines"]["data"][0]["price"].get("id"), str)
    and isinstance(stripe_invoice["lines"]["data"][0]["price"].get("product"), str)
    and isinstance(stripe_invoice["lines"]["data"][0].get("period"), dict)
    and isinstance(stripe_invoice["lines"]["data"][0]["period"].get("start"), int)
    and isinstance(stripe_invoice["lines"]["data"][0]["period"].get("end"), int)
):
    customer_id = stripe_invoice["customer"]
    subscription_id = stripe_invoice["subscription"]
    price_id = stripe_invoice["lines"]["data"][0]["price"]["id"]
    product_id = stripe_invoice["lines"]["data"][0]["price"]["product"]
    start = stripe_invoice["lines"]["data"][0]["period"]["start"]
    end = stripe_invoice["lines"]["data"][0]["period"]["end"]

    print(customer_id, subscription_id, price_id, product_id)
```

Motivation

So where does it shine?

Unpacking of nested and complex data structures!

Equivalent safety during runtime without `match`/`case`, becomes annoying!

```python
if (
    isinstance(stripe_invoice, dict)
    and isinstance(stripe_invoice.get("customer"), str)
    and isinstance(stripe_invoice.get("subscription"), str)
    and isinstance(stripe_invoice.get("lines"), dict)
    and isinstance(stripe_invoice["lines"].get("data"), list)
    and len(stripe_invoice["lines"]["data"]) == 1
    and isinstance(stripe_invoice["lines"]["data"][0], dict)
    and isinstance(stripe_invoice["lines"]["data"][0].get("price"), dict)
    and isinstance(stripe_invoice["lines"]["data"][0]["price"].get("id"), str)
    and isinstance(stripe_invoice["lines"]["data"][0]["price"].get("product"), str)
    and isinstance(stripe_invoice["lines"]["data"][0].get("period"), dict)
    and isinstance(stripe_invoice["lines"]["data"][0]["period"].get("start"), int)
    and isinstance(stripe_invoice["lines"]["data"][0]["period"].get("end"), int)
):
    customer_id = stripe_invoice["customer"]
    subscription_id = stripe_invoice["subscription"]
    price_id = stripe_invoice["lines"]["data"][0]["price"]["id"]
    product_id = stripe_invoice["lines"]["data"][0]["price"]["product"]
    start = stripe_invoice["lines"]["data"][0]["period"]["start"]
    end = stripe_invoice["lines"]["data"][0]["period"]["end"]

    print(customer_id, subscription_id, price_id, product_id)
```

```
cus_RCWzWxRNmFPtEw sub_1MowQVLkdIwHu7ixeRlqHVzs price_1QHnM1DTc7leUICsAukuDpcs prod_RA7aUpI09bik6W
```

What to look out for?

Capture patterns

What to look out for?

Capture patterns

```
1 def validate_user_id(x: int | str) -> None:
2     match x:
3         case 0: # literal pattern
4             print("Found system user!")
5         case str:
6             print(f"Found user with string ID {x}")
```

```
1 validate_user_id(0)
```

What to look out for?

Capture patterns

```python
def validate_user_id(x: int | str) -> None:
    match x:
        case 0: # literal pattern
            print("Found system user!")
        case str:
            print(f"Found user with string ID {x}")
```

```
validate_user_id(0)
```

What to look out for?

Capture patterns

```
1 def validate_user_id(x: int | str) -> None:
2     match x:
3         case 0: # literal pattern
4             print("Found system user!")
5         case str:
6             print(f"Found user with string ID {x}")
```

```
1 validate_user_id(0)
```

```
Found system user!
```

What to look out for?

Capture patterns

```python
def validate_user_id(x: int | str) -> None:
    match x:
        case 0:
            print("Found system user!")
        case str:  # capture pattern
            print(f"Found user with string ID {x}")
```

```
validate_user_id("78759548-eb9f-4ef8-9484-120daba2657b")
```

What to look out for?

Capture patterns

```python
def validate_user_id(x: int | str) -> None:
    match x:
        case 0:
            print("Found system user!")
        case str:  # capture pattern
            print(f"Found user with string ID {x}")
```

```python
validate_user_id("78759548-eb9f-4ef8-9484-120daba2657b")
```

```
Found user with string ID 78759548-eb9f-4ef8-9484-120daba2657b
```

What to look out for?

Capture patterns

```python
def validate_user_id(x: int | str) -> None:
    match x:
        case 0:
            print("Found system user!")
        case str:  # capture pattern
            ...

    print(f"NO! Our Built-in str: {str}")
```

```python
validate_user_id("78759548-eb9f-4ef8-9484-120daba2657b")
```

What to look out for?

Capture patterns

```python
def validate_user_id(x: int | str) -> None:
    match x:
        case 0:
            print("Found system user!")
        case str:  # capture pattern
            ...

    print(f"NO! Our Built-in str: {str}")
```

```python
validate_user_id("78759548-eb9f-4ef8-9484-120daba2657b")
```

```
NO! Our Built-in str: 78759548-eb9f-4ef8-9484-120daba2657b
```

What to look out for?

Capture patterns

```python
def validate_user_id(x: int | str) -> None:
    match x:
        case 0:
            print("Found system user!")
        case str:  # irrefutable capture pattern
            ...
        case _:  # irrefutable capture pattern (conflicts with above)
            print("Everything else!")
```

What to look out for?

Capture patterns

```
 1 def validate_user_id(x: int | str) -> None:
 2     match x:
 3         case 0:
 4             print("Found system user!")
 5         case str:  # irrefutable capture pattern
 6             ...
 7         case _:  # irrefutable capture pattern (conflicts with above)
 8             print("Everything else!")
```

```
  Cell In[129], line 5
    case str:  # irrefutable capture pattern
         ^
SyntaxError: name capture 'str' makes remaining patterns unreachable
```

What to look out for?

Capture patterns

```
 1  def validate_user_id(x: int | str) -> None:
 2      match x:
 3          case 0:
 4              print("Found system user!")
 5          case str:  # irrefutable capture pattern
 6              ...
 7          case _:  # irrefutable capture pattern (conflicts with above)
 8              print("Everything else!")
```

```
  Cell In[129], line 5
    case str:  # irrefutable capture pattern
         ^
SyntaxError: name capture 'str' makes remaining patterns unreachable
```

It is good practice, to end the match statement with a wildcard (_) to catch any unmatched cases.

What to look out for?

Match for types with the class pattern!

What to look out for?

Match for types with the class pattern!

```python
def validate_user_id(x: int | str) -> None:
    match x:
        case 0:
            print("This is a literal pattern.")
        case str(uid) if uid.isupper():
            print(f"Found UPPER user with string ID {uid=}.")
        case str() as uid:
            print(f"Found user with string ID {uid=}.")
        case _:
            print("Everything else!")
```

```python
validate_user_id("78759548-EB9F-4EF8-9484-120DABA2657B")
```

What to look out for?

Match for types with the class pattern!

```python
def validate_user_id(x: int | str) -> None:
    match x:
        case 0:
            print("This is a literal pattern.")
        case str(uid) if uid.isupper():
            print(f"Found UPPER user with string ID {uid=}.")
        case str() as uid:
            print(f"Found user with string ID {uid=}.")
        case _:
            print("Everything else!")
```

```python
validate_user_id("78759548-EB9F-4EF8-9484-120DABA2657B")
```

```
Found UPPER user with string ID uid='78759548-EB9F-4EF8-9484-120DABA2657B'.
```

What to look out for?

Match for types with the class pattern!

```python
def validate_user_id(x: int | str) -> None:
    match x:
        case 0:
            print("This is a literal pattern.")
        case str(uid) if uid.isupper():
            print(f"Found UPPER user with string ID {uid=}.")
        case str() as uid:
            print(f"Found user with string ID {uid=}.")
        case _:
            print("Everything else!")
```

```python
validate_user_id("78759548-eb9f-4ef8-9484-120daba2657b")
```

What to look out for?

Match for types with the class pattern!

```python
def validate_user_id(x: int | str) -> None:
    match x:
        case 0:
            print("This is a literal pattern.")
        case str(uid) if uid.isupper():
            print(f"Found UPPER user with string ID {uid=}.")
        case str() as uid:
            print(f"Found user with string ID {uid=}.")
        case _:
            print("Everything else!")
```

```python
validate_user_id("78759548-eb9f-4ef8-9484-120daba2657b")
```

```
Found user with string ID uid='78759548-eb9f-4ef8-9484-120daba2657b'.
```

Let's extend the json serializer with enums

Other surprises?

Let's extend the json serializer with enums

Other surprises?

```python
class RegistrationMode(StrEnum):
    ENABLED = "enabled"
    ON_INVITE = "on_invite"
    DISABLED = "disabled"


def to_json_serializable(unknown: Any) -> JSONSerializable:
    match unknown:
        case list(values):
            return [to_json_serializable(value) for value in values]
        case dict(values):
            return {key: to_json_serializable(value) for key, value in values.items()}
        case tuple(values):
            return [to_json_serializable(value) for value in values]
        case None | bool() | int() | float() | str():
            return unknown
        case Enum():
            return unknown.value
        case _ as unreachable:
            assert_never(unreachable)
```

```python
to_json_serializable(RegistrationMode.ON_INVITE)
```

Let's extend the json serializer with enums

Other surprises?

```python
class RegistrationMode(StrEnum):
    ENABLED = "enabled"
    ON_INVITE = "on_invite"
    DISABLED = "disabled"


def to_json_serializable(unknown: Any) -> JSONSerializable:
    match unknown:
        case list(values):
            return [to_json_serializable(value) for value in values]
        case dict(values):
            return {key: to_json_serializable(value) for key, value in values.items()}
        case tuple(values):
            return [to_json_serializable(value) for value in values]
        case None | bool() | int() | float() | str():
            return unknown
        case Enum():
            return unknown.value
        case _ as unreachable:
            assert_never(unreachable)
```

```python
to_json_serializable(RegistrationMode.ON_INVITE)
```

```
<RegistrationMode.ON_INVITE: 'on_invite'>
```

Let's extend the json serializer with enums

Other surprises?

```python
class RegistrationMode(StrEnum):
    ENABLED = "enabled"
    ON_INVITE = "on_invite"
    DISABLED = "disabled"


def to_json_serializable(unknown: Any) -> JSONSerializable:
    match unknown:
        case list(values):
            return [to_json_serializable(value) for value in values]
        case dict(values):
            return {key: to_json_serializable(value) for key, value in values.items()}
        case tuple(values):
            return [to_json_serializable(value) for value in values]
        case None | bool() | int() | float() | str():
            return unknown
        case Enum():
            return unknown.value
        case _ as unreachable:
            assert_never(unreachable)
```

Let's extend the json serializer with enums

Other surprises?

```python
class RegistrationMode(StrEnum):
    ENABLED = "enabled"
    ON_INVITE = "on_invite"
    DISABLED = "disabled"


def to_json_serializable(unknown: Any) -> JSONSerializable:
    match unknown:
        case list(values):
            return [to_json_serializable(value) for value in values]
        case dict(values):
            return {key: to_json_serializable(value) for key, value in values.items()}
        case tuple(values):
            return [to_json_serializable(value) for value in values]
        case Enum():
            return unknown.value
        case None | bool() | int() | float() | str():
            return unknown
        case _ as unreachable:
            assert_never(unreachable)
```

```python
to_json_serializable(RegistrationMode.ON_INVITE)
```

Let's extend the json serializer with enums

Other surprises?

```python
class RegistrationMode(StrEnum):
    ENABLED = "enabled"
    ON_INVITE = "on_invite"
    DISABLED = "disabled"


def to_json_serializable(unknown: Any) -> JSONSerializable:
    match unknown:
        case list(values):
            return [to_json_serializable(value) for value in values]
        case dict(values):
            return {key: to_json_serializable(value) for key, value in values.items()}
        case tuple(values):
            return [to_json_serializable(value) for value in values]
        case Enum():
            return unknown.value
        case None | bool() | int() | float() | str():
            return unknown
        case _ as unreachable:
            assert_never(unreachable)
```

```python
to_json_serializable(RegistrationMode.ON_INVITE)
```

```
'on_invite'
```

Using `match/case` in FastAPI

Managing Parameter Combinations

Using `match/case` in FastAPI

Managing Parameter Combinations

```python
@router_projects.get("")
async def _(
    *,
    org_id: int | None = None,
    user_id: str | None = None,
    ctx: Context = Security(auth),
) -> list[V1_Project]:
    match (org_id, user_id, ctx.user):
        case (int(org_id), None, UserModel()):
            projects = read_projects_by_organization(org_id=org_id, ctx=ctx)
            ...
        case (None, str(user_id), UserModel()):
            projects = read_projects_by_user(user_id=user_id, ctx=ctx)
            ...
        case (None, None, UserModel()):
            projects = read_projects_by_user(user_id=ctx.user.id, ctx=ctx)
            ...
        case _:
            raise HTTPException() from None

    return [_V1_Project_factory(project=project) for project in projects]
```

Takeaways

Takeaways

- `if` stays relevant, but `elif`/`else` got a competitor

Takeaways

- `if` stays relevant, but `elif`/`else` got a competitor
- Use to branch based on structure not value

Takeaways

- `if` stays relevant, but `elif`/`else` got a competitor
- Use to branch based on structure not value
- Python 3.9 is soon end of life

Takeaways

- `if` stays relevant, but `elif`/`else` got a competitor
- Use to branch based on structure not value
- Python 3.9 is soon end of life
  - => Almost no reason not to try out `match`/`case`

Takeaways

- `if` stays relevant, but `elif`/`else` got a competitor
- Use to branch based on structure not value
- Python 3.9 is soon end of life
  - => Almost no reason not to try out `match`/`case`
  - Except that language models are barely trained on it

Thank you 👋

Questions?

Thank you 👋

Questions?

Backup Slides (Surprising behaviours)

Tuples

How can we match on a tuple?

Tuples

How can we match on a tuple?

```
1 def match_tuple(tup):
2     match tup:
3         case (x, y):
4             return f"Matched {x} and {y}"
5
6
7 # Matches our tuple!
8 match_tuple((1, 2))
```

Tuples

How can we match on a tuple?

```
1 def match_tuple(tup):
2     match tup:
3         case (x, y):
4             return f"Matched {x} and {y}"
5
6
7 # Matches our tuple!
8 match_tuple((1, 2))
```

Tuples

How can we match on a tuple?

```python
def match_tuple(tup):
    match tup:
        case (x, y):
            return f"Matched {x} and {y}"


# Matches our tuple!
match_tuple((1, 2))
```

```
'Matched 1 and 2'
```

Tuples

How can we match on a tuple?

Tuples

How can we match on a tuple?

But there is a catch

Tuples

How can we match on a tuple?

```
1  def match_tuple(tup):
2      match tup:
3          case (x, y):
4              return f"Matched {x} and {y}"
5
6
7  # also matches on a list!
8  match_tuple([1, 2])
```

Tuples

How can we match on a tuple?

```
1  def match_tuple(tup):
2      match tup:
3          case (x, y):
4              return f"Matched {x} and {y}"
5
6
7  # also matches on a list!
8  match_tuple([1, 2])
```

Tuples

How can we match on a tuple?

```
1  def match_tuple(tup):
2      match tup:
3          case (x, y):
4              return f"Matched {x} and {y}"
5
6
7  # also matches on a list!
8  match_tuple([1, 2])
```

```
'Matched 1 and 2'
```

Tuples

Recall the class pattern!

Tuples

Recall the class pattern!

We can have an `isinstance` equivalent check

Tuples

Recall the class pattern!

We can have an `isinstance` equivalent check

```python
@dataclass
class Point:
    x: int
    y: int


point = Point(2, 4)

match point:
    case Point(x=3, y=y):
        print("Does not match")
    case Point(2, y):
        print(f"Matches and y is {y}")
```

Tuples

Recall the class pattern!

We can have an `isinstance` equivalent check

```python
@dataclass
class Point:
    x: int
    y: int


point = Point(2, 4)

match point:
    case Point(x=3, y=y):
        print("Does not match")
    case Point(2, y):
        print(f"Matches and y is {y}")
```

Tuples

Recall the class pattern!

We can have an `isinstance` equivalent check

```python
@dataclass
class Point:
    x: int
    y: int


point = Point(2, 4)

match point:
    case Point(x=3, y=y):
        print("Does not match")
    case Point(2, y):
        print(f"Matches and y is {y}")
```

```
Matches and y is 4
```

Tuples

So how can we match on a tuple?

Tuples

So how can we match on a tuple?

Unfortunately tuples expect special treatment

Tuples

So how can we match on a tuple?

Unfortunately tuples expect special treatment

```
1 match (1, 2):
2     case tuple(x, y):
3         ...
```

Tuples

So how can we match on a tuple?

Unfortunately tuples expect special treatment

```
 1 match (1, 2):
 2     case tuple(x, y):
 3         ...
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[142], line 2
      1 match (1, 2):
----> 2     case tuple(x, y):
      3         ...

TypeError: tuple() accepts 1 positional sub-pattern (2 given)
```

Tuples

So how can we match on a tuple?

Unfortunately tuples expect special treatment

```
 1 match (1, 2):
 2     case tuple(x, y):
 3         ...
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[142], line 2
      1 match (1, 2):
----> 2     case tuple(x, y):
      3         ...

TypeError: tuple() accepts 1 positional sub-pattern (2 given)
```

This gives us a hint...

Tuples

So how can we match on a tuple?

Tuples

So how can we match on a tuple?

Nest a sequence pattern inside the class pattern!

Tuples

So how can we match on a tuple?

Nest a sequence pattern inside the class pattern!

```
1 match (1, 2):
2     case tuple((x, y)):  # equivalent to `case tuple([x, y])`
3         print(f"Matches and x is {x} and y is {y}")
```

Tuples

So how can we match on a tuple?

Nest a sequence pattern inside the class pattern!

```
1 match (1, 2):
2     case tuple((x, y)):  # equivalent to `case tuple([x, y])`
3         print(f"Matches and x is {x} and y is {y}")
```

Tuples

So how can we match on a tuple?

Nest a sequence pattern inside the class pattern!

```
1 match (1, 2):
2     case tuple((x, y)):  # equivalent to `case tuple([x, y])`
3         print(f"Matches and x is {x} and y is {y}")
```

```
Matches and x is 1 and y is 2
```

Sets

What about sets?

Sets

What about sets?

Don’t even think about trying this:

Sets

What about sets?

Don’t even think about trying this:

```
1 match {"created", "updated"}:
2     case {"created", "updated"}:
3 #                  ^
4 # SyntaxError: invalid syntax
```

Sets

What about sets?

Sets

What about sets?

Still not a chance:

Sets

What about sets?

Still not a chance:

```
1 match {"created", "updated"}:
2     case set({"created", "updated"}):
3 #                        ^
4 # SyntaxError: invalid syntax
```

Sets

What about sets?

Still not a chance:

```
1 match {"created", "updated"}:
2     case set({"created", "updated"}):
3 #                      ^
4 # SyntaxError: invalid syntax
```

rationale: https://stackoverflow.com/a/67893188

Sets

What about sets?

Sets

What about sets?

- Possible with the constant value pattern

Sets

What about sets?

- Possible with the constant value pattern
- Requires a qualified name

Sets

What about sets?

Sets

What about sets?

```
1  class Constants:
2      SET_123 = {1, 2, 3}
3
4
5  match {1, 2, 3}:
6      case Constants.SET_123:
7          print("matched")
```

Sets

What about sets?

```
1  class Constants:
2      SET_123 = {1, 2, 3}
3
4
5  match {1, 2, 3}:
6      case Constants.SET_123:
7          print("matched")
```

Sets

What about sets?

```
1  class Constants:
2      SET_123 = {1, 2, 3}
3
4
5  match {1, 2, 3}:
6      case Constants.SET_123:
7          print("matched")
```

```
matched
```

Sets

What about sets?

Sets

What about sets?

```
1  class Constants:
2      SET_123 = {1, 2, 3}
3
4
5  match {1, 2, 3}:
6      case set(Constants.SET_123):
7          print("matches as well")
```

Sets

What about sets?

```
1 class Constants:
2     SET_123 = {1, 2, 3}
3
4
5 match {1, 2, 3}:
6     case set(Constants.SET_123):
7         print("matches as well")
```

Sets

What about sets?

```
1  class Constants:
2      SET_123 = {1, 2, 3}
3
4
5  match {1, 2, 3}:
6      case set(Constants.SET_123):
7          print("matches as well")
```

```
matches as well
```

Backup Slides (Pattern types)

literal pattern

`True` or `"hello"`

capture pattern

x

wildcard pattern

_

constant value pattern

```
Color.RED
```

sequence pattern

```
[a, *rest, b]
```

mapping pattern

```
{"user": u, "emails": [*es]}
```

class pattern

```
datetime.date(year=y, day=d)
```

OR pattern

```
[*x] | {"elems": [*x]}
```

walrus pattern

```
d := datetime(year=2020, month=m)
```