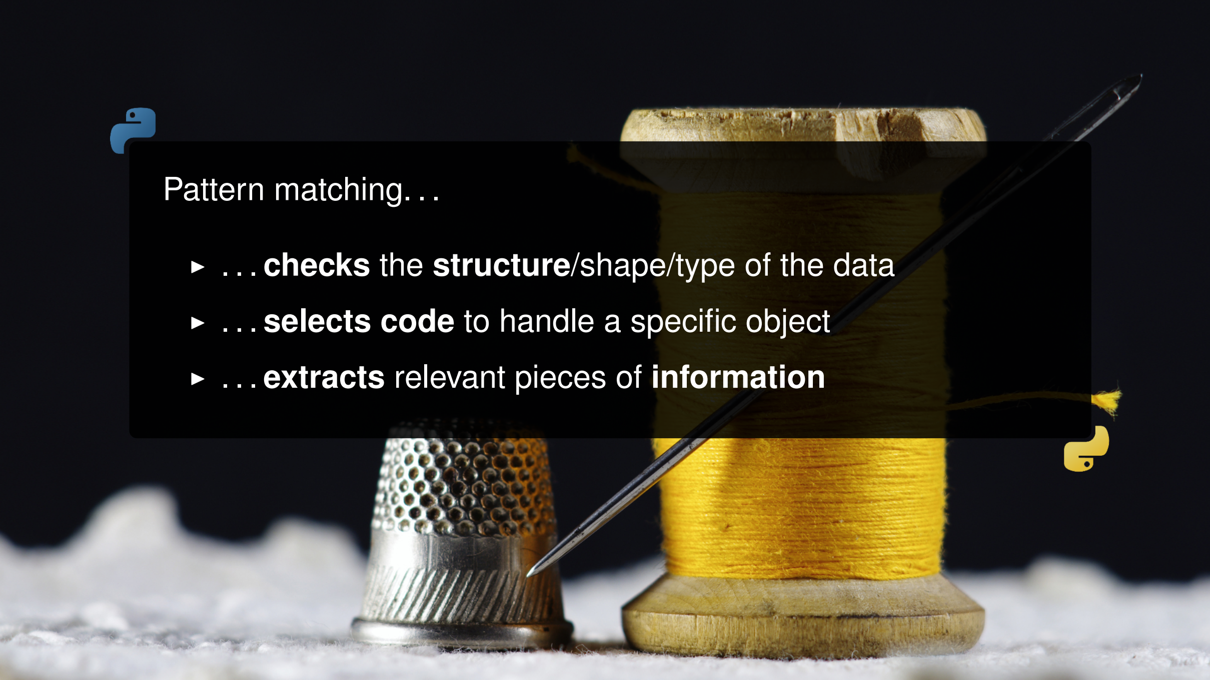
The Road to Pattern Matching in Python

Tobias Kohn

Pattern matching is simple…


PLASTIC
GLASS
PAPER
OTHER
. . . yet versatile and malleable

Pattern Matching
So, what is it really?


Pattern matching...
▸ ...**checks** the **structure**/shape/type of the data
▸ ...**selects code** to handle a specific object
▸ ...**extracts** relevant pieces of **information**


```
circle( x, y, radius )
```

$$A = \pi \times r^2$$


```
rectangle( x, y, width, height )
```

$$A = w \times h$$

```python
def area(shape):
    if isinstance(shape, circle):
        radius = shape.radius
        return math.pi * radius ** 2

    elif isinstance(shape, rectangle):
        wd, ht = shape.width, shape.height
        return wd * ht

a = area( Circle(40, 50, 100) )
```

```python
def area(shape):
    if isinstance(shape, circle):
        radius = shape.radius
        return math.pi * radius ** 2

    elif isinstance(shape, rectangle):
        wd, ht = shape.width, shape.height
        return wd * ht


a = area( Circle(40, 50, 100) )
```

```python
def area(shape):
    if isinstance(shape, circle):
        radius = shape.radius
        return math.pi * radius ** 2

    elif isinstance(shape, rectangle):
        wd, ht = shape.width, shape.height
        return wd * ht

a = area( Circle(40, 50, 100) )
```

```python
def area(shape):
    match shape:
        case circle(_, _, radius):
            return math.pi * radius ** 2

        case rectangle(_, _, wd, ht):
            return wd * ht


a = area( Circle(40, 50, 100) )
```

Pattern Matching

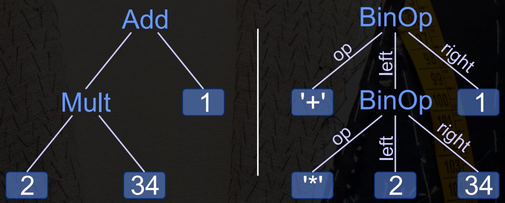
1. Run specialised code based on type and structure of your object;
2. Automatically extract relevant data/attributes from an object

A Closer Look at the Fabric

How do we make things fit?

Premise data is organised in graphs and trees (using objects)

Example the expression `2 * 34 + 1` has a tree-structure:




```
BinOp(op='+', left=BinOp('*', 2, 34), right=1)
```

Patterns naturally exhibit the same tree-structure as objects, e.g.:

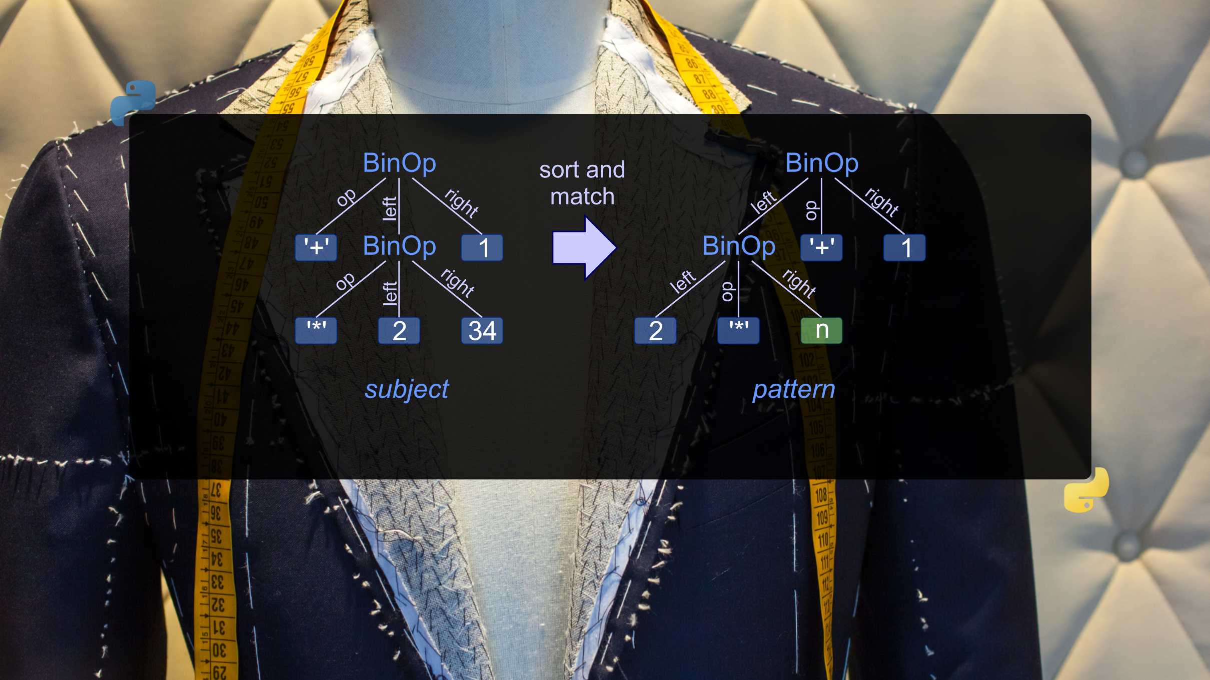
```
match s:
    case BinOp(BinOp(2, '*', n), '+', 1):
        # odd number
        ...
```

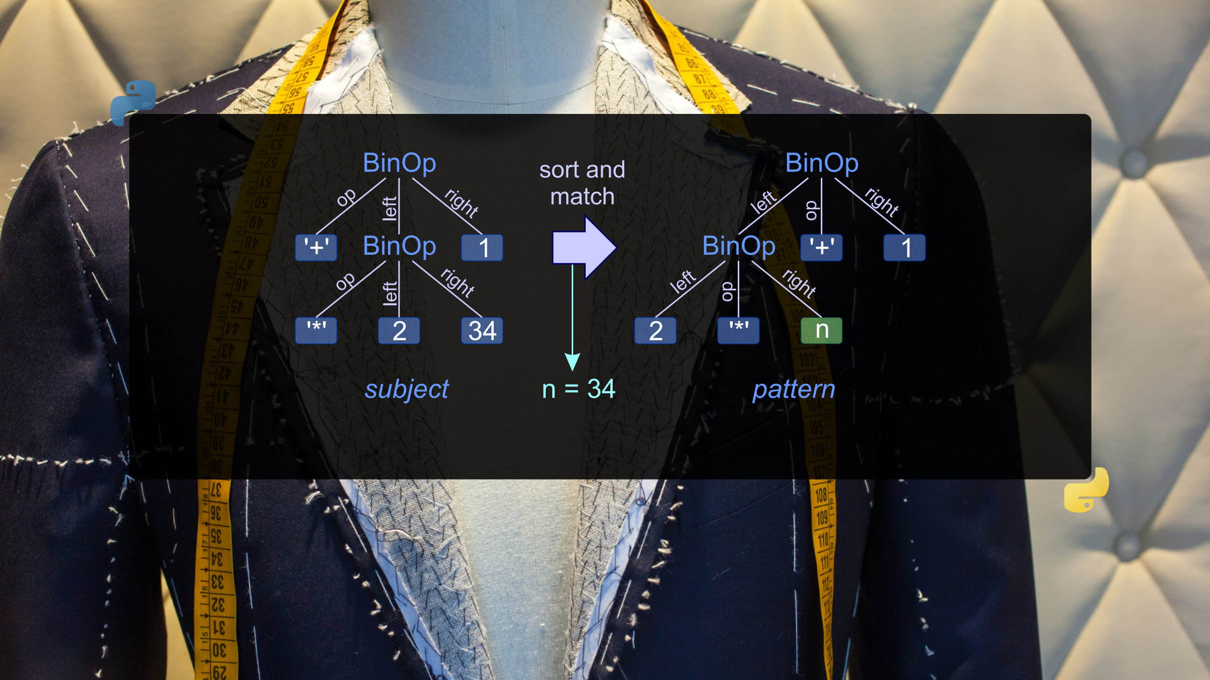
Matching objects to patterns

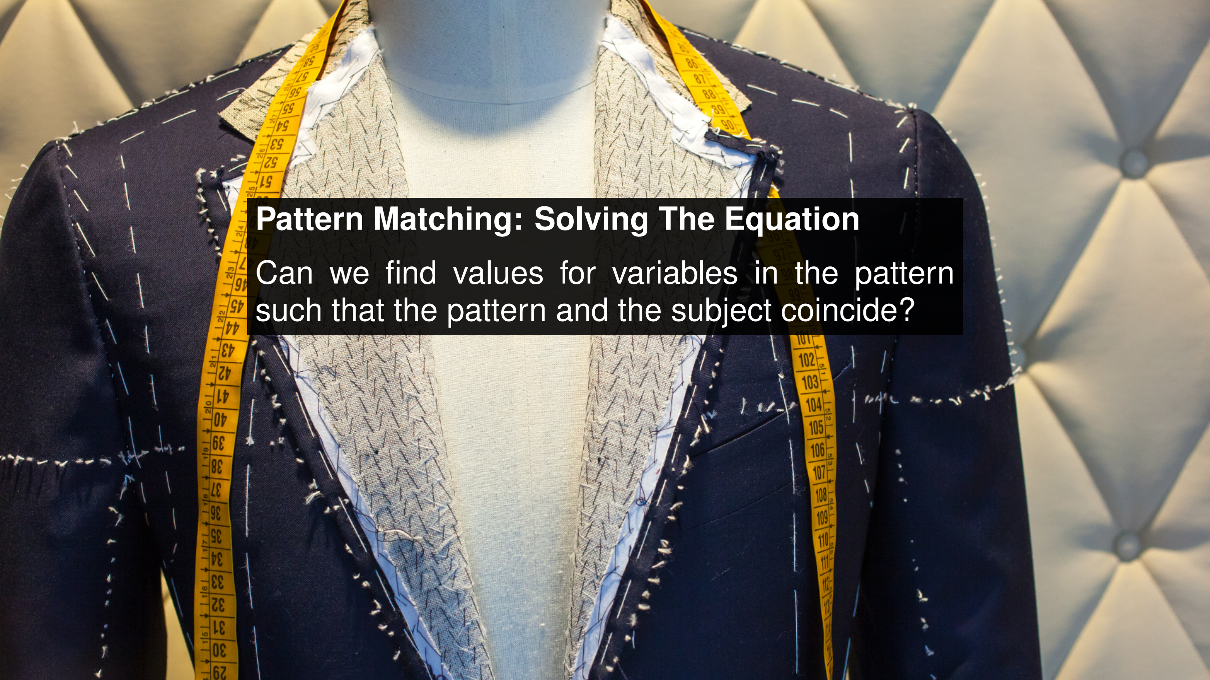
- attributes are inherently *unordered*
- how do we map attributes to positions?
- `BinOp(x, '+', 1)` vs `BinOp('+', x, 1)`

Matching objects to patterns

- attributes are inherently *unordered*
- how do we map attributes to positions?
- `BinOp(x, '+', 1)` vs `BinOp('+', x, 1)`
- use `__match_args__ = ('left', 'op', 'right')`

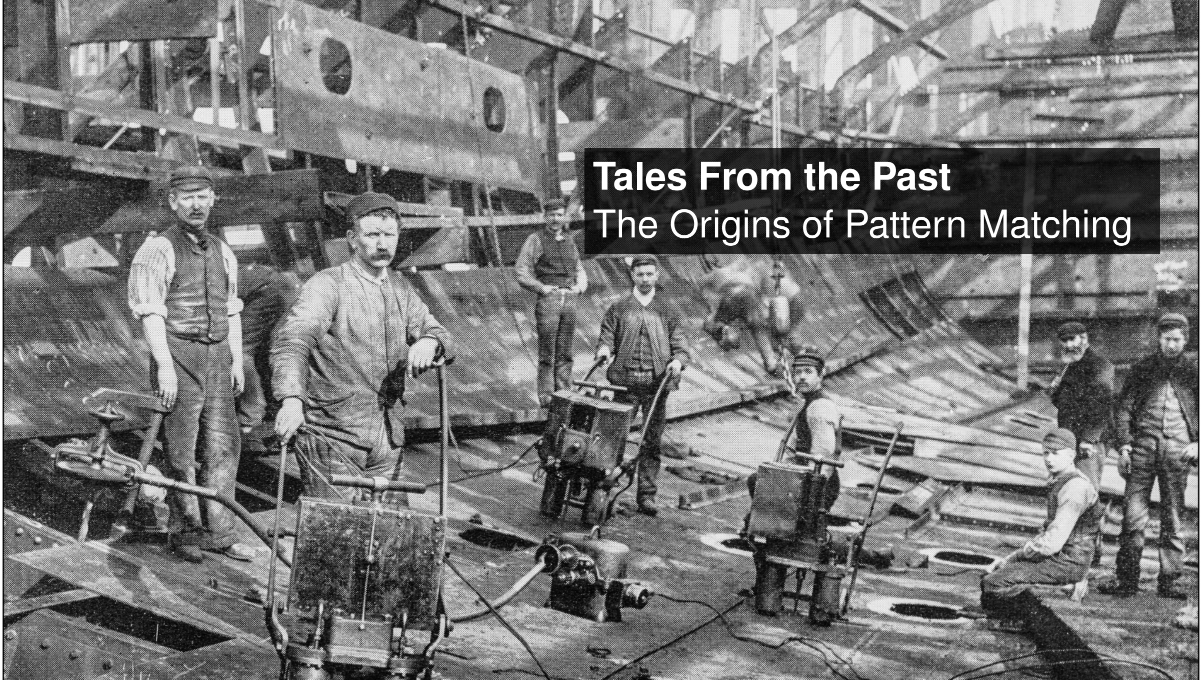

BinOp
op
left
right
'+'
BinOp
1
op
left
right
'*'
2
34
subject
sort and
match
BinOp
left
op
right
BinOp
'+'
1
left
op
right
2
'*'
n
pattern


BinOp
op
left
right
'+'
BinOp
1
op
left
right
'*'
2
34
subject
sort and
match
n = 34
BinOp
left
op
right
BinOp
'+'
1
left
op
right
2
'*'
n
pattern



Pattern Matching: Solving The Equation

Can we find values for variables in the pattern such that the pattern and the subject coincide?


Tales From the Past
The Origins of Pattern Matching

In the beginning was…

In the beginning was... *tuple unpacking*

In the beginning was... *tuple unpacking*

Minimalistic design: a language without field or item access

With strong static types, consider `tup = (123, 'abc')`

- `tup[0]` has type `int`
- `tup[1]` has type `str`
- what type has `tup[i]`?

Question: how do we handle dynamic data structures?

Simply put, each 'object' is either a `tuple` or `None`, e.g. linked list:

```
primes = (2, (3, (5, (7, None))))
```

2 → 3 → 5 → 7 → None

```
(x, rest) = mylist
```

Answer: alternatives / *conditional unpacking*

```python
def sum(mylist):
    result = 0
    while True:
        match mylist:
            case (n, rest):
                result += n
                mylist = rest
            case None:
                return result
```

Answer: alternatives / *conditional unpacking*

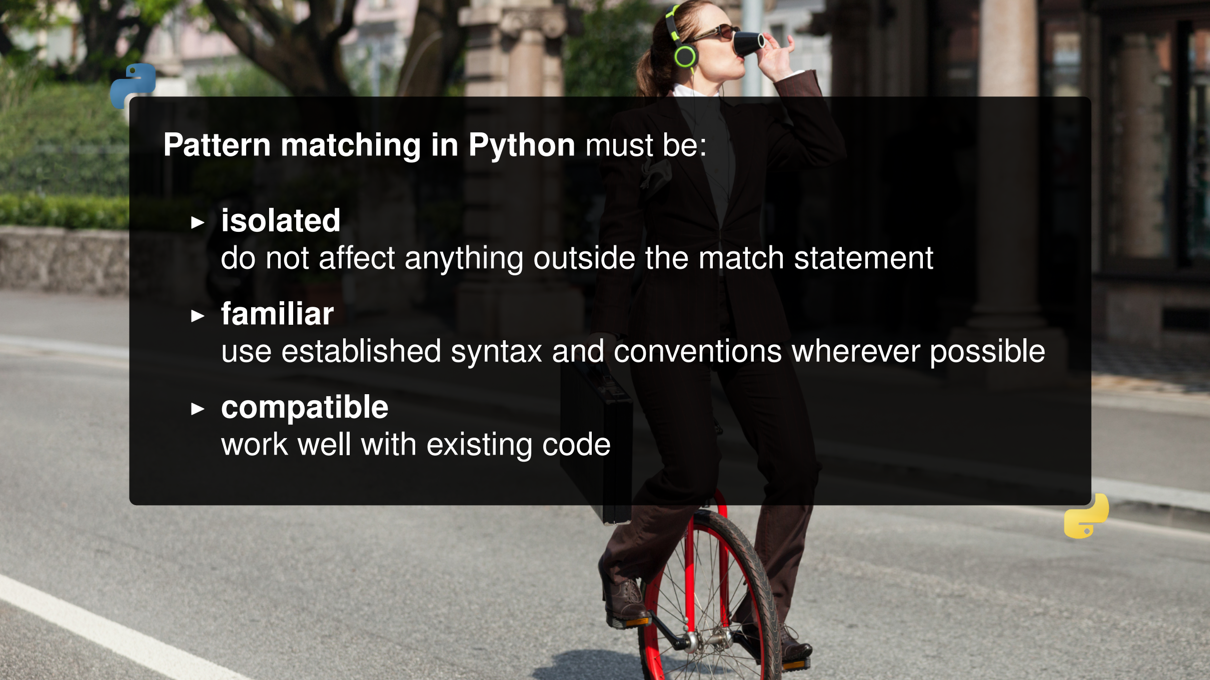
```
def sum(mylist):
    match mylist:
        case (m, (n, None)):
            return m + n
        case (n, rest):
            return n + sum(rest)
        case None:
            return 0
```

Pattern matching

Extend *tuple unpacking* to handle *dynamic data structures*

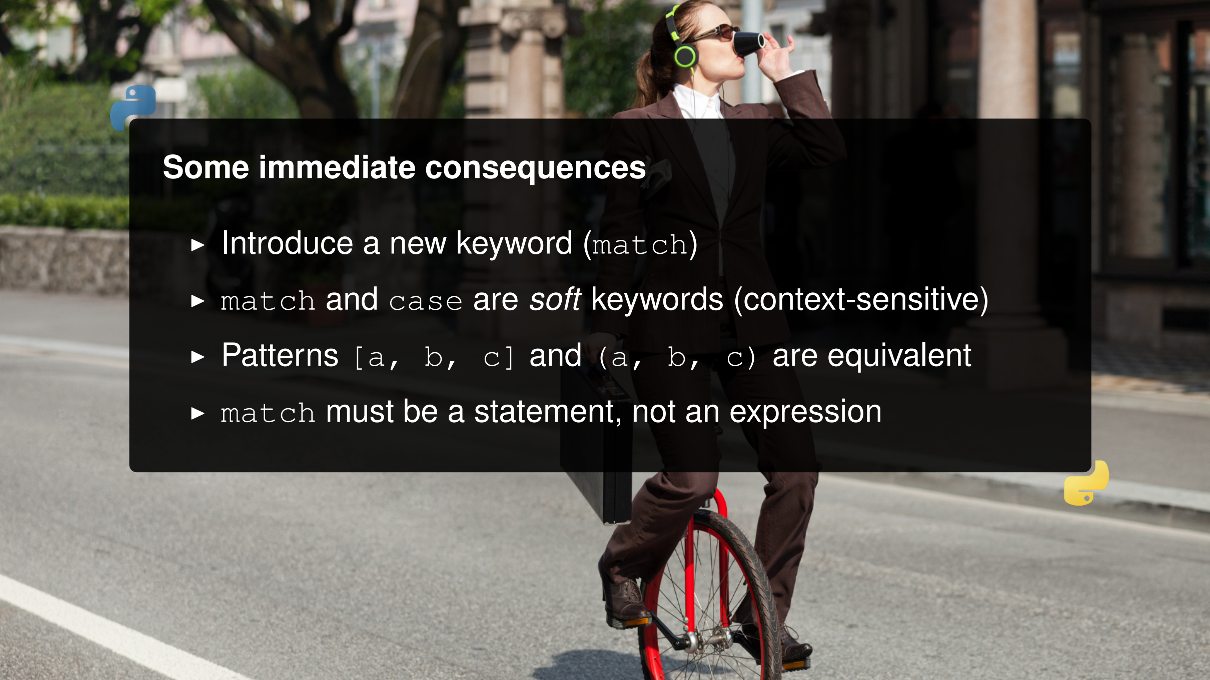
Changing the Present

The Challenge of Embracing a New Paradigm



Pattern matching in Python must be:

- **isolated**
  do not affect anything outside the match statement
- **familiar**
  use established syntax and conventions wherever possible
- **compatible**
  work well with existing code



Some immediate consequences

- Introduce a new keyword (`match`)
- `match` and `case` are *soft* keywords (context-sensitive)
- Patterns `[a, b, c]` and `(a, b, c)` are equivalent
- `match` must be a statement, not an expression

Conditional vs unconditional unpacking

```
match some_iterator:
    case (a, b, c, 0):
        ...
    case (a, b, c, *rest):
        ...
    case x:
        # do not consume elements from the
        # iterator in this case
```

Annotations / type hints

Could we use type hints to specify the type/class of variables?

```
match some_expr:
    case (i: int):
        ...
    case [s: str, t: str]:
        ...
```

Annotations / type hints

Could we use type hints to specify the type/class of variables?

```
match some_expr:
    case (i: int):
        ...
    case [s: str, t: str]:
        ...
```

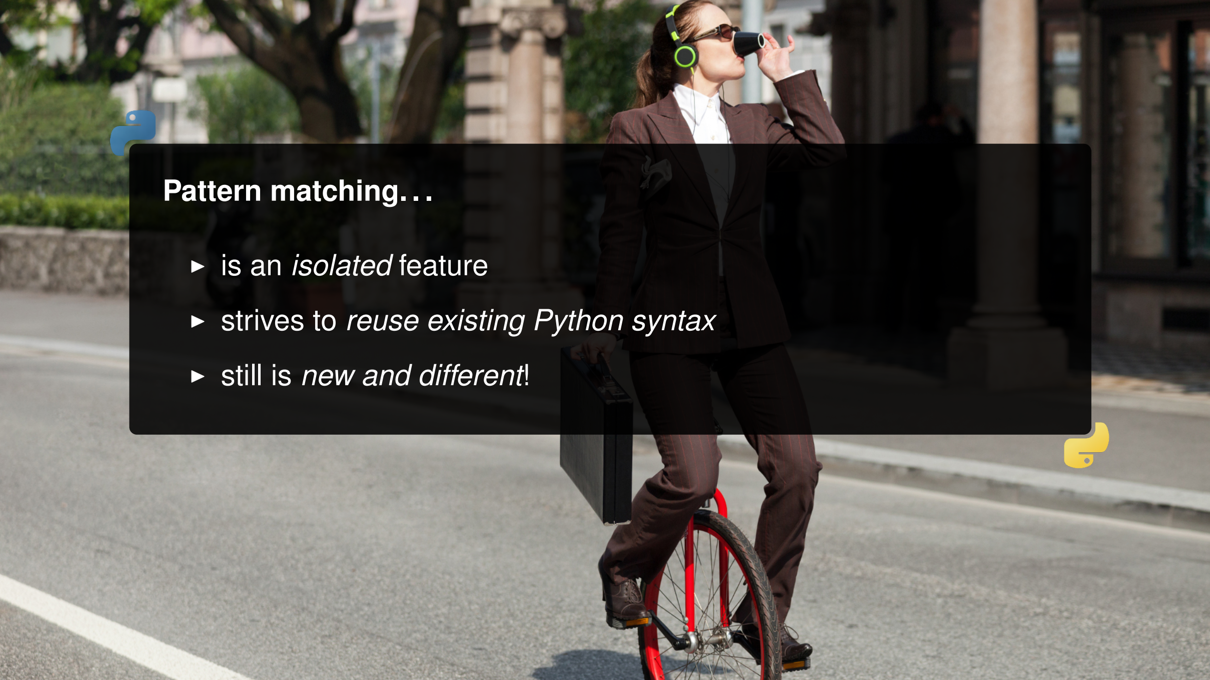
No – annotations are never enforced by the interpreter

Annotations / type hints

Could we use type hints to specify the type/class of variables?

```
match some_expr:
    case int(i):
        ...
    case [str(s), str(t)]:
        ...
```

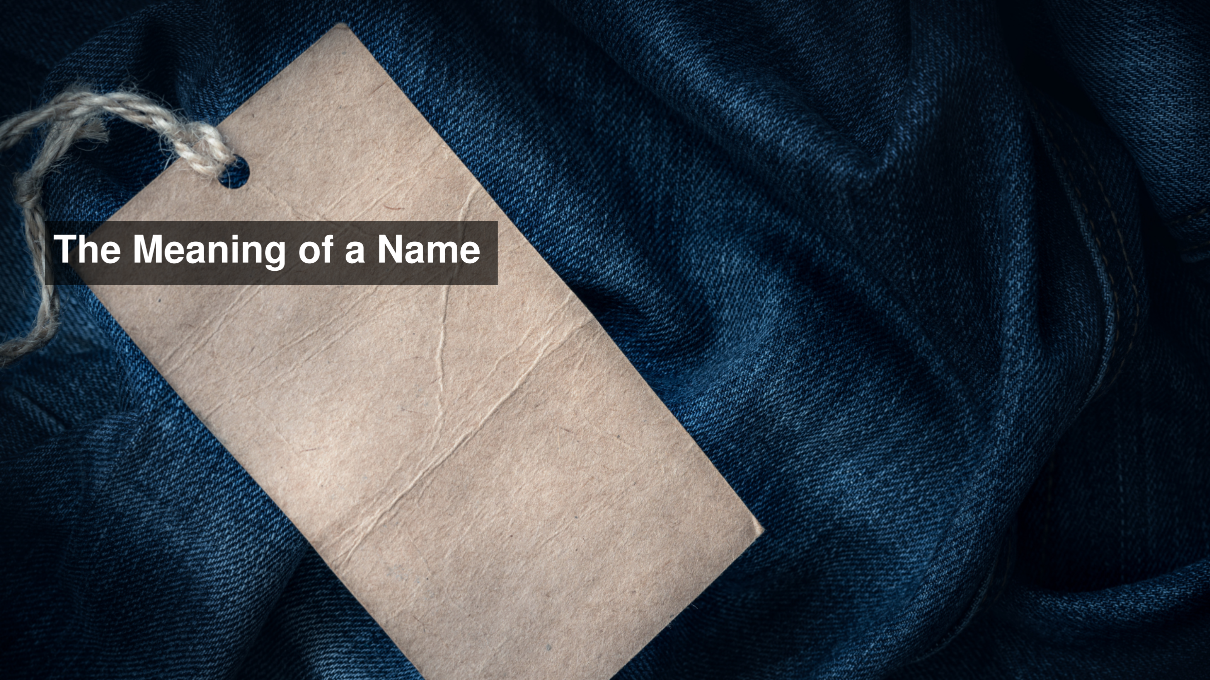
No – annotations are never enforced by the interpreter



Pattern matching…

- is an *isolated* feature
- strives to *reuse existing Python syntax*
- still is *new and different*!

The Meaning of a Name



The meaning of a name

```
from math import pi

match x:
    case pi:
        ...
```

How shall we interpret ‘`case pi`’?

- match only if $x = \pi$
- match anything and set $pi := x$

The meaning of a name

- Languages with *declarations* (`var x = ...`) can differentiate
- Others distinguish based on *spelling*: `pi` vs `Pi`
- Only bind *local* names: `pi` vs `math.pi`
- Make all names binding targets (i.e. always overwrite `pi`)

The meaning of a name

- Languages with *declarations* (`var x = ...`) can differentiate
- Others distinguish based on *spelling*: `pi` vs `Pi`
- Only bind *local* names: `pi` vs `math.pi` $\leftarrow$ ***most Pythonic***
- Make all names binding targets (i.e. always overwrite `pi`)

The meaning of a name

```
match mytuple:
    case (x, x):
        ...
    case 2 | n:
        ...
```

How shall we interpret '`case (x, x)`'?

- Tuple with two equal elements?
- Bind *x* to the second element?

The meaning of a name

```
match mytuple:
    case (x, x):
        ...
    case 2 | n:
        ...
```

How shall we interpret '**`case`** `2|n`'?

- Only bind *n* if it is not 2?

The meaning of a name

```
match mytuple:
    case (x, x):
        ...
    case 2 | n:
        ...
```

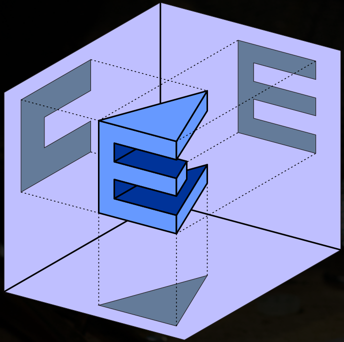
Don't allow either of these variants!

- Bind all occurring names to values
- Each name is bound exactly once

The meaning of a name

- Simple names are binding targets
- Attributes provide value constraints
- The set of binding targets is deterministic

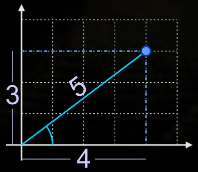
A Vision of the Future
Bespoke Patterns



Objects are complex

- An object can have more than one 'shape'
- There is more than one way to look at/view an object

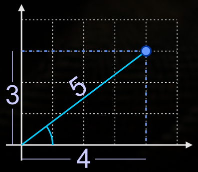
Objects are complex—example



$$4 + 3j = 5\measuredangle 38.9^\circ$$

```
case crect(x, y):
    ...
case cpolar(r, angle):
    ...
```

Objects are complex—example



$4 + 3j = 5\measuredangle 38.9^\circ$

```
case crect(x, y):
    ...
case cpolar(r, angle):
    ...
```

`crect` and `cpolar` are not classes, but ***views*** of an object

Objects are complex

```
class crect:
    def __match__(s):
        if isinstance(s, complex):
            return Yes(s)
        elif isinstance(s, vector2D):
            return Yes(complex(s[0], s[1]))
        else:
            return No
```


Pattern Matching
Taylor Your Code to Your Data

The Road to Pattern Matching in Python

Special thanks to Brandt Bucher, Ivan Levkivskyi, Daniel Moisset, Guido van Rossum, Talin