Pattern Matching in Python

Copyright and AI Notice

These slides are intended for studying and teaching at Nazarbayev University in Astana, Kazakhstan.

If you want to redistribute or adapt these slides for different purposes, check the license first.

No generative AI was used in the preparation of these slides.

Pattern Matching

Pattern matching is a compact way of inspecting the structure of data. Code that uses matching is easy to write and easy to read.

Matching compares a given value to a sequence of patterns, where each pattern has code attached to it. It executes the code that is attached to the first pattern that matches.

Example

```
def trymatch(x) :
    match x :
        case 100 :
            return "one hundred"
        case str(y) :
            return f"x is a string {y}"
        case a, b:
            return f"x is the tuple or list ( {a},{b} )"
```

Try writing the same with `if`, `elif`, and `else` and you see the advantage of matching.

Tuples/Lists

Matching does not distinguish between tuples and lists.
`case (a,b):` `case [a,b]:` and also `case a,b:` have the same effect.

It will match both `(1,2)` and `[1,2]`.

If you want to specify the first elements of a tuple/list, but not how long it is, you can write for example `case ( 1,2, *a )`. Fits to lists/tuples starting with `1, 2`, the rest goes into variable `a`.

Order is Important

Cases are attempted in order. Specific cases must be put before general cases.

```
match x :
    case [ a, b ] :
        print( f"a = {a}, b = {b}" )
    case [ x, 4 ] :
        print( "cannot not happen" )
            # because the first case will have matched.
```

Exchanging order results in:

```
    case [ x, 4 ] :
        print( f"first element is {x}" )
    case [ a, b ] :
        print( f"a = {a}, b = {b}" )
            # We can be sure that b is not 4.
```

Adding Conditions

It is possible to write conditions after the cases. In order to match, the condition must be true:

```
match x :
   case [ a, b ] if a >= b :
      print( f"{a} is greater than {b}" )
   case [ a, b ] :
      print( f"{a} is not greater than {b}" )
```

The match will be succesful if `x` has form `[A,B]` and `A` indeed equals `B`. If `A` is not equal to `B`, the first case will be not a matching, and second case will match.

Adding Conditions (2)

If one writes instead

```
match x :
   case [ a, b ] :
      if a > b :
         print( f"{a} is greater than {b}" )
   case [ a, b ] :
      print( f"{a} is not greater than {b}" )
```

the first case will match `[1,2]` but have no effect.

The second case will never match.

Matching is Recursive

Matching expressions can be more complicated. In that case, the matched value will be recursively inspected:

```
case [ [ [ 1, a ] ] ] :
```

Will match against `[[[1,4]]]` with `a` = 4.

Ignoring Variables

If one does not care about the value of a variable, it can be replaced by an underscore:

```
match x :
   case [ [ [ _ ] ] ] :
      print( "a 3-nested list" )
   case [ [ _ ] ] :
      print( "2-nested" );
   case [ _ ] :
      print( "just a list" )
   case _ :
      print( "not even a list" )
```

Different Patterns in a Single Case

Different patterns can be combined into a single case by using bar (|):

```
case [ [ [ 1, a ] ] ]  |  [ [ [ a, 1 ] ] ] :
   print(a)
```

If you don't want to write the common part twice, the bar can be used inside the matching expression:

```
case [ [ 1, a ]  |  [ a, 1 ] ] :
   print(a)
```

Patterns combined with a bar must contain the same variables.

Catching Subexpressions

If one wants to store subexpressions in variables, one can use `as`:

This is useful when using a bar (`|`) because otherwise there is no way of knowing wich option was used:

```
match move:
   case ( "north", dist ) | ( "east", dist ) |
        ( "south", dist ) | ( "west", dist ) :
      print( dist )
         # But how do we know which one matched?
```

(Note that in real, the `case` must be in one line.)

Catching Subexpressions (2)

With `as`, one can remember subexpressions in variables:

```
case (( "north", dist ) | ( "east", dist ) |
      ( "south", dist ) | ( "west", dist )) as dir :
   print( dist )
   print( dir[0] )
      # Advantage is that we are sure that dir[0]
      # exists.
```

`as` can be applied on subexpressions, so that the direction can be caught directly:

```
case ( ( "north" | "east" | "south" | "west" ) as dir,
       dist ) :
   print( dist )
   print( dir )
```

Matching Keys

It is possible to match keys in a dictionary:

```
dict = { "one" : ( 1, "first" ),
         "two": ( 2, "second" ),
         "three" : ( 3, "third" ),
         "four" : ( 4, "fourth" ) }
match dict :
   case { "two" : ( nr, ord ) } :
      print( f"the ordinal of two is {ord}" )
```

Unfortunately, matching keys must be fixed. The following is impossible, although it would be nice (it's a syntax error):

```
card = "two"
match dict :
   case { card : ( nr, ord ) } :
      print( f"the ordinal of {card} is {ord}" )
```

Types and Classes

Types and classes can be specified as follows:

```
case int(x) :
   print(x)
      # Matches only integers

case [ str(s), float(n) ] :
   print(s)
      # Matches list/tuple consisting of
      # a string and a float.
      # matches [ 'x', 4.0 ] but not [ 'string', 100 ]
```

If you don't know the type of your data, use `type( )`.

Matching Fields

It is possible to match against a class with fields:

```
class Cl :
   pass

c = Cl( )
c. fld1 = 'a'
c. fld2 = 'b'

match c :
   case Cl( fld1 = 'a', fld2 = 'b' ) :
      # Will match.
```

Matching Fields (2)

If you need the values of the fields, write

```
case Cl( fld1 = A, fld2 = B ) :
   print( f"fld1 = {A} and fld2 = {B}" )
```

Conclusions

Matching is a readable and compact way of inspecting structure of data. It is much easier to write than chains of ifs and elses, and the result is more readable.

In functional languages (like Haskell or Scala) it is the main control mechanism.

In Prolog, you will see unification, which is a generalization of matching. The difference is that with unification, variables can appear both in the data and in the pattern that is used for matching.