Pattern matching
in python?

Lukas Prokop
7th of July 2015

Pattern matching quiz

I will show you patterns and usecases.

You will tell me whether this is *pattern matching*.

Scala Pattern Matching

Scala supports the notion of *case classes*. Case classes are regular classes which export their **constructor parameters** and which provide a **recursive decomposition mechanism via pattern matching**.

— A Tour of Scala: Case Classes

Scala's pattern matching statement is **most useful** for matching on **algebraic types** expressed via case classes.

— Scala Documentation: Tutorial Pattern Matching

Scala Pattern Matching

```
1  object AScalaClass {
2    class Atom(name: String, atomicNumber: Int) {
3      def atomicNumber(): Int = atomicNumber
4      override def toString(): String = name
5    }
6
7    def main(args: Array[String]) {
8      val element = new Atom("Hydrogen", 1)
9      println("This is an " + element + " atom "
10           + "with atomic number "
11           + element.atomicNumber() + "!")
12   }
13 }
```

Scala Pattern Matching

```
1  object RandScalaClass {
2    class Atom(name: String, atomicNumber: Int) {
3      def atomicNumber(): Int = atomicNumber
4      override def toString(): String = name
5    }
6
7    def main(args: Array[String]) {
8      val es = Array(new Atom("Hydrogen", 1), new Atom("Helium", 2))
9      val rand = scala.util.Random.nextInt(2)
10     println("This is an " + es(rand) + " atom!")
11   }
12 }
```

Scala Pattern Matching

```
1  object RandBranchedScalaClass {
2    class Atom(name: String, atomicNumber: Int) {
3      def atomicNumber(): Int = atomicNumber
4      override def toString(): String = name
5    }
6
7    def main(args: Array[String]) {
8      val es = Array(new Atom("Hydrogen", 1), new Atom("Helium", 2))
9      val rand = scala.util.Random.nextInt(2)
10     println("This is an " + es(rand) + " atom!")
11     if (es(rand).toString() == "Hydrogen")
12       println(" with atomic number " + es(rand).atomicNumber())
13   }
14 }
```

Scala Pattern Matching

```
1  object RandMatchedScalaClass {
2    class Atom(name: String, atomicNumber: Int) {
3      def atomicNumber(): Int = atomicNumber
4      override def toString(): String = name
5    }
6
7    def main(args: Array[String]) {
8      val es = Array(new Atom("Hydrogen", 1), new Atom("Helium", 2))
9      val rand = scala.util.Random.nextInt(2)
10     es(rand).toString() match {
11       case "Hydrogen" => println("This is an " + es(rand) + " atom"
12             + " with atomic number " + es(rand).atomicNumber() + "!")
13       case _ => println("This is an " + es(rand) + " atom!")
14     }
15   }
16 }
```

Scala Pattern Matching

```
1  object EquivScalaClass {
2    class Atom(name: String, atomicNumber: Int) {
3      def atomicNumber(): Int = atomicNumber
4      override def toString(): String = name
5    }
6
7    def main(args: Array[String]) {
8      val es = Array(new Atom("Hydrogen", 1),
9                     new Atom("Helium", 2))
10     println(es(0) == es(1))
11     // prints 'false'
12   }
13 }
```

Scala Pattern Matching

- Equivalence is violated.
- ~~Bananas~~ Object members for scale!
  Assume instances have more than 12 attributes
  - Branching should not depend on string representation (might not be unique, compare with python's repr)
  - Assume branching depends on any attribute
    - One method per attribute
    - ...and `toString` and `equals`!
    - Remember name of each attribute method!
    - `match` statement performs branching

Scala Pattern Matching

```
1  object AttributedScalaClass {
2    class Atom(name: String, atomicNumber: Int)
3
4    def main(args: Array[String]) {
5      println(new Atom("Helium", 2) == new Atom("Helium", 2))
6      // prints 'false'
7    }
8  }
```

Objects are already fully specified!

Scala Pattern Matching

```
1  object ScalaCaseClass {
2    case class Atom(name: String, atomicNumber: Int)
3        universal solution to all mentioned problems!
4    def main(args: Array[String]) {
5      println(new Atom("Helium", 2) == new Atom("Helium", 2))
6      // prints 'true'
7    }
8  }
```

Scala Pattern Matching

```
1  object MatchedCaseClass {
2    case class Atom(name: String, atomicNumber: Int)
3
4    def main(args: Array[String]) {
5      val es = Array(new Atom("Hydrogen", 1),
6                     new Atom("Helium", 2))
7      val rand = scala.util.Random.nextInt(es.length)
8      es(rand) match {
9        case Atom("Hydrogen", ano) =>
10         println("Hydrogen's atomic number is " + ano)
11       case Atom(aname, _) => println("Atom " + aname)
12     }
13   }
14 }
```

Scala Pattern Matching

```
1   object Extractor {
2     object Manipulative {
3       def apply(x: Int): Int = x * 2
4       def unapply(z: Int): Option[Int] =
5         if (z % 2 == 0) Some(z / 2)
6         else None
7     }
8
9     def main(args: Array[String]) {
10      // EXPANSION: val x = Manipulative.apply(21)
11      val x = Manipulative(21)
12      println(x) // "42", type int
13      x match {
14        // case 42 => println("I am also matched")
15        // EXPANSION: Manipulative.unapply(42)
16        case Manipulative(n) => println("I matched " + n)
17      }
18    }
19  }
```

Pattern matching in Scala is independent of case classes. Case classes are just an application.

In general pattern matching is handled by (un)apply methods.

Mathematica Cases

$$\text{In}[1] := \text{Cases}[\{1, 1, f[a], 2, 3, y, f[8], 9, f[10]\}, \_\text{Integer}]$$
$$\text{Out}[1] = \{1, 1, 2, 3, 9\}$$

$$\text{In}[2] := \text{Cases}[\{1, f[a], 3, y, f[8]\}, \text{Except}[\_\text{Integer}]]$$
$$\text{Out}[2] = \{f[a], y, f[8]\}$$

$$\text{In}[3] := \text{Cases}[\{1, 1, f[a], 2, 3, y, f[8], 9, f[10]\}, f[x\_] \text{->} x]$$
$$\text{Out}[3] = \{a, 8, 10\}$$

Mathematica Cases

$$\text{In}[4] := \text{Cases}[\_\text{Integer}][\{1, 1, f[a], 2, 3, y, f[8], 9, f[10]\}]$$

$$\text{Out}[4] = \{1, 1, 2, 3, 9\}$$

$$\text{In}[5] := \text{Cases}[\{\{1, 2\}, \{2\}, \{3, 4, 1\}, \{5, 4\}, \{3, 3\}\},$$
$$\{a\_, b\_\} \text{ -> } a + b]$$

$$\text{Out}[5] = \{3, 9, 6\}$$

Rust pattern matching

A match expression **branches on a pattern**. The **exact form** of matching that occurs **depends on the pattern**. Patterns consist of some **combination of literals**, **destructured arrays** or **enum constructors**, **structures** and **tuples**, **variable binding specifications**, **wildcards** (..), and **placeholders** (_).

A match expression has a **head expression**, which is the value to compare to the patterns. The type of the patterns must equal the type of the head expression.

—https://doc.rust-lang.org/stable/reference.html#match-expressions

Rust pattern matching

```
1  extern crate rand;
2
3  fn main() {
4      let rand: f32 = rand::random::<f32>();
5      let index: i32 = (6.0 * rand) as i32;
6
7      print!("{}", index);
8      match index {
9          1 => println!("st"),
10         2 => println!("nd"),
11         _ => println!("th"),
12     }
13 }
```

Rust pattern matching

```
1  extern crate rand;
2
3  fn main() {
4      let rand: f32 = rand::random::<f32>();
5      let index: i32 = (6.0 * rand) as i32;
6
7      print!("{}", index);
8      match index {
9          1 => { println!("st"); }
10         2 => { println!("nd"); }
11         0|3|4|5 => { println!("th"); }
12         6...7 => { println!("impossible"); }
13         _ => { println!("even less possible"); }
14     }
15 }
```

Rust pattern matching

```
1  fn main() {
2      let tup = ("pygraz", "spektral",
3                 "7th of July 2015", "http://pygraz.org/");
4      let (event, location, date, _) = tup;
5
6      println!("{} at {} on {}", event, location, date);
7  }
```

Rust pattern matching

```
1   fn main() {
2       let tup = ("pygraz", "spektral",
3                  "7th of July 2015", "http://pygraz.org/");
4
5       match tup {
6           ("rustgraz", _, _, _)
7               => println!("Rust event detected!"),
8           (event, _, when, _)
9               => println!("Event: {}\nDate: {}", event, when),
10      }
11  }
```

Rust pattern matching

```
1  fn main() {
2      let tup = ("pygraz", "spektral",
3                 "7th of July 2015", "http://pygraz.org/");
4
5      match tup {
6          (event, _, when, _) if event.contains("rust")
7              => println!("Rust event on {} detected!", when),
8          (event, _, when, _)
9              => println!("Event: {}\nDate: {}", event, when),
10     }
11 }
```

Rust pattern matching

```
1   fn main() {
2       struct Point {
3           x: i32,
4           y: i32,
5       }
6
7       let origin = Point { x: 0, y: 0 };
8
9       match origin {
10          Point { x, y } => println!("({},{})", x, y),
11      }
12  }
```

Rust pattern matching

```
1   fn main() {
2       struct Person {
3           name: Option<String>,
4       }
5
6       let name = "Steve".to_string();
7       let x: Option<Person> = Some(Person { name: Some(name) });
8       match x {
9           Some(Person { name: ref a @ Some(_), .. })
10              => println!("{:?}", a),
11          _ => {}
12      }
13  }
```

Haskell one-way pattern matching

```
1  fib 0 = 0
2  fib 1 = 1
3  fib n = fib (n-1) + fib (n-2)
4
5  main = print (fib 5)
```

Python3 re

```
1  import re
2  import sys
3
4  INT = re.compile(r""^\d+$"")
5  STR = re.compile(r""^(["""])[^""]*\1$"")
6  FLOAT = re.compile(r""^(\d+|\d+\.\d+|10e\d+|[—+]inf)$"")
7
8  if __name__ == '__main__':
9      main(sys.argv[1])
```

Python3 re

```
1  def main(arg):
2      if INT.search(arg):
3          print("{} is an integer".format(arg))
4      elif STR.search(arg):
5          print("{} is a string".format(arg))
6      elif FLOAT.search(arg):
7          print("{} is a floating point number".format(arg))
8      else:
9          print("{} is an unknown type".format(arg))
```

CSS selectors

- a
- a.external
- a emph
- :first-child
- a[rel =copyright], a[data-community*=pygraz]
- div:empty
- article:lang(fr)
- input:not(:checked)

XPath / XSLT

```
//html/body/*[
  local-name() = "h1" or
  local-name() = "h2"
]/text()
```

```
1  <xsl:variable name="months" select="'January', 'February', 'March', 'April'"/>
2  <xsl:function name="custom:convert-date">
3    <xsl:param name="value" />
4    <xsl:analyze-string select="normalize-space($value)"
5        regex="([0-9]{4})/([0-9]{1,2})/([0-9]{1,2})">
6      <xsl:matching-substring>
7        <xsl:number value="regex-group(1)" format="0001"/>
8        <xsl:text>-</xsl:text>
9        <xsl:number value="regex-group(2)" format="01"/>
10       <xsl:text>-</xsl:text>
11       <xsl:number value="regex-group(3)" format="01"/>
12     </xsl:matching-substring>
13   </xsl:analyze-string>
14 </xsl:function>
```

Clojure case function

```
1  (defn xor
2    [a b]
3    (case [a b]
4      ([true false] [false true]) true
5      false))
6
7  (xor true true) ; false
8  (xor false false) ; false
9  (xor false true) ; true
10 (xor true false) ; true
```

Clojure fn dispatching

```
1  (def mult
2    (fn this
3      ([] 1)
4      ([x] x)
5      ([x y] (* x y))
6      ([x y & more] (apply this (this x y) more))))
7
8  (mult 42 11) ; 462
```

Clojure let bindings

```
1  (defn edge-str [arg]
2    (let [[v w] arg]
3      (str "v=" v " w=" w)))
4
5  (edge-str [3 4])
6
7  (use '[clojure.string :only (join)])
8  (defn print-langs [{name :name [lang & more] :langs}]
9    (str name " programming in " lang " and others including "
10     (join ", " more)))
11
12 (print-langs {:name "Lukas" :langs ["Go", "Python"]})
```

Clojure multimethods

```
1  (defmulti area :Shape)
2  (defmethod area :Rect [r]
3        (* (:wd r) (:ht r)))
4  (defmethod area :Circle [c]
5        (* (. Math PI) (* (:radius c) (:radius c))))
6  (defmethod area :default [x] :very-much-invalid)
7
8  (defn rect [wd ht] {:Shape :Rect :wd wd :ht ht})
9  (defn circle [radius] {:Shape :Circle :radius radius})
10 (area (rect 4 13))
11 (area (circle 12))
12 (area {})
```

Python3 if elif

```
1   a = 5
2
3   if a is None:
4       print(”a is of value None”)
5   elif a > 20:
6       print(”great value”)
7   elif bin(a).endswith(”0”):
8       print(”Even value”)
9   elif a:
10      print(”a is considered positive”)
```

Compare with PEP 3103

Java switch-case

```
1  class SwitchCase {
2      public static void main(String[] args) {
3          switch ("Hello World") {
4              case "Hello World":
5                  System.out.println(
6                    "An English greeting");
7                  break;
8              case "こんにちは":
9                  System.out.println(
10                   "Oh, someone's speaking " +
11                   " Japanese!");
12                 break;
13 } } }
```

C switch-case

```
1   #include <stdio.h>
2
3   int main(int argc, char* argv[]) {
4     int a;
5     for (a = 1; a < argc; a++) {
6       switch ((argv[a][0] - 'a') / 2) {
7         case 0:
8           printf("Argument %d starts with a or b\n", a);
9           break;
10        case 1:
11          printf("Argument %d starts with c or d\n", a);
12          break;
13          // ...
14        default:
15          printf("Argument %d starts with an non-alphabetic character\n", a);
16          break;
17    } }
18    return 0;
19  }
```

C switch-case

```
1   #include <stdio.h>
2
3   int main(int argc, char* argv[]) {
4     int a;
5     for (a = 1; a < argc; a++) {
6       switch ((argv[a][0] - 'a') / 2) {
7         case 0:
8           printf("Argument %d starts with a or b\n", a);
9           break;
10        case 1:
11          printf("Argument %d starts with c or d\n", a);
12          break;
13          // ...
14        default:
15          printf("Argument %d starts with an non-alphabetic character\n", a);
16          break;
17    } }
18    return 0;
19  }
```

C builds hashmap in background

C switch-case

```
1   void device(char* from, char* to, int count) {
2       register int n = (count + 7) / 8;
3       switch (count % 8) {
4       case 0: do { *to++ = *from++;
5       case 7: *to++ = *from++;
6       case 6: *to++ = *from++;
7       case 5: *to++ = *from++;
8       case 4: *to++ = *from++;
9       case 3: *to++ = *from++;
10      case 2: *to++ = *from++;
11      case 1: *to++ = *from++;
12                  } while (--n > 0);
13      }
14  }
```

C switch-case

```
1   void device(char* from, char* to, int count) {
2       register int n = (count + 7) / 8;
3       switch (count % 8) {
4       case 0: do { *to++ = *from++;
5       case 7: *to++ = *from++;
6       case 6: *to++ = *from++;
7       case 5: *to++ = *from++;
8       case 4: *to++ = *from++;
9       case 3: *to++ = *from++;
10      case 2: *to++ = *from++;
11      case 1: *to++ = *from++;
12                  } while (--n > 0);
13      }
14  }
```

so-called “Duff’s device”

So? What is pattern matching?

- Scala Case classes
- Scala match (un)apply
- Mathematica Cases
- Rust match statement
- Rust destructuring
- Haskell one-way pattern matching
- Regular expressions
- CSS selectors
- XPath
- Clojure case function
- Clojure fn dispatching
- Clojure let bindings
- Clojure multimethods
- Python3 if elif
- Java switch-case
- C switch-case

So? What is pattern matching?

- Scala Case classes
- Scala match (un)apply
- Mathematica Cases
- Rust match statement
- Rust destructuring
- Haskell one-way pattern matching
- Regular expressions
- CSS selectors
- XPath
- Clojure case function
- Clojure fn dispatching
- Clojure let bindings
- Clojure multimethods
- Python3 if elif
- Java switch-case
- C switch-case

according to documentation

Jeffrey Friedl

Wikipedia

Pattern matching is the **act of checking** a given sequence of *tokens* for the presence of the constituents of some *pattern*. In contrast to pattern recognition, the match usually has to be exact. The patterns generally have the form of either sequences or tree structures. Uses of pattern matching include outputting the locations (if any) of a pattern within a token sequence, to output some component of the matched pattern, and to substitute the matching pattern with some other token sequence (i.e., search and replace).

Sequence patterns (e.g., a text string) are often described using regular expressions and matched using techniques such as backtracking.

Wikipedia

Tree patterns are used in some programming languages as a general tool to process data based on its structure, e.g., Haskell, ML, Scala and the symbolic mathematics language Mathematica have special syntax for expressing **tree patterns** and a **language construct for conditional execution and value retrieval** based on it. For simplicity and efficiency reasons, these tree patterns lack some features that are available in regular expressions.

Scala Case Classes – Usecases

```
1   object TreeRepresentation {
2     sealed abstract class Tree
3     case class Node(elem: Int, left: Tree, right: Tree) extends Tree
4     case object Leaf extends Tree
5
6     def inOrder(t: Tree): List[Int] = t match {
7       case Node(e, l, r) => inOrder(l) ::: List(e) ::: inOrder(r)
8       case Leaf => List()
9     }
10
11    def main(args: Array[String]) {
12      val tree = new Node(2, new Node(4, Leaf, Leaf), Leaf)
13      println(inOrder(tree))
14    }
15  }
```

Scala Case Classes – Usecases

```
1  object AbstractSyntaxTree {
2    sealed abstract class ASTree
3    case class Value(value: Any) extends ASTree
4    case class UnaryOperator(op: String, child: ASTree) extends ASTree
5    case class BinaryOperator(left: ASTree, op: String,
6          right: ASTree) extends ASTree
7    case class TernaryOperator(left: ASTree, op1: String,
8          center: ASTree, op2: String, right: ASTree) extends ASTree
9  }
```

Scala Case Classes – Usecases

```
1  object AbstractSyntaxTree {
2    sealed abstract class ASTree
3    case class Value(value: Any) extends ASTree
4    case class UnaryOperator(op: String, child: ASTree) extends ASTree
5    ...
6
7    def reprTree(ast: ASTree): String = ast match {
8      case UnaryOperator(op, child) => " " + op + reprTree(child)
9      case BinaryOperator(l, o, r) => "" + reprTree(l) + o + reprTree(r)
10     case TernaryOperator(l, o, c, p, r) =>
11       "" + reprTree(l) + o + reprTree(c) + p + reprTree(r)
12     case Value(v) => s" $v "
13   }
14 }
```

Scala Case Classes – Usecases

```
1  object AbstractSyntaxTree {
2    sealed abstract class ASTree
3    case class Value(value: Any) extends ASTree
4    case class UnaryOperator(op: String, child: ASTree) extends ASTree
5    ...
6
7    val vari = new Value("variable")
8    val tree = new TernaryOperator(
9          new BinaryOperator(vari, ">", new Value(0)),
10         "?", vari, ":", new UnaryOperator("-", vari))
11
12   def main(args: Array[String]) {
13     println(reprTree(tree))
14     // prints ' variable > 0 ? variable : - variable '
15   }
16 }
```

Definition

So I guess, pattern matching is some kind of testing whether a **structure** corresponds to some **pattern** with **optional binding** to elements for further processing.

Structures can be sequences, class hierarchies, data types or constructor arguments.

What's possible in python?

```
1  tupl = ("Dorian", "Ernesto", "Florian", "Horst", "Lukas")
2
3  head, *_ = tupl
4  print(head)
5
6  *_, tail = tupl
7  print(tail)
8
9  head, *_, tail = tupl
10 print(head, tail)
```

PEP 3132 – Extended Iterable Unpacking

What's possible in python?

```
1 nested = (1, (2, 3), 4)
2 (a, (b, c), d) = nested
3 print(b, d)
```

What's possible in python?

```
1 nested = (1, (2, 3), 4)
2 (a, (b, c), d) = nested
3 print(b, d)
```

What about function arg bindings?

What's possible in python?

```
1  def basic(first, *args):
2      print(first, args)
3
4  basic(1, 2, 3, 4) # 1 (2, 3, 4)
```

```
1  # tuple parameter unpacking
2  def nested(a, (b, c), d):
3      print(a, b, c, d)
4
5  basic(1, (2, 3), 4) # 1, 2, 3, 4
```

What's possible in python?

```
1  def basic(first, *args):
2      print(first, args)
3
4  basic(1, 2, 3, 4) # 1 (2, 3, 4)
```

```
1  # tuple parameter unpacking
2  def nested(a, (b, c), d):
3      print(a, b, c, d)
4
5  basic(1, (2, 3), 4) # 1, 2, 3, 4
```

does not work, because of
PEP 3113 – Removal of Tuple Parameter Unpacking

What's possible in python?

- Regular expressions? re available.
- Destructuring by enumeration?
  Tuple unpacking available.
- Destructuring based on types?
  AFAIK violates duck typing.
- Destructuring on constructor arguments?
  Metaprogramming with `__init__`
- Multimethods? Emulate with decorators.
- Switch case? Unnecessary.

What's possible in python?

- Regular expressions? re available.
- Destructuring by enumeration?
  Tuple unpacking available.
- Destructuring based on types?
  AFAIK violates duck typing.
- Destructuring on constructor arguments?
  Metaprogramming with `__init__`
  check out eg. MacroPy
  https://github.com/lihaoyi/macropy
- Multimethods? Emulate with decorators.
  Multimethods by Guido van Rossum
  http://www.artima.com/weblogs/viewpost.jsp?thread=101605
- Switch case? Unnecessary.

Thanks

Thanks for your attention!

http://lukas-prokop.at/talks/pattern-matching-in-python/