Салимова А.Р., Васильева К.А. Сопоставление с образцом в языке программирования Python // Международный журнал информационных технологий и энергоэффективности. – 2023. –
Т. 8 № 4(30) с. 30–40

Международный журнал информационных технологий и энергоэффективности

Сайт журнала:
http://www.openaccessscience.ru/index.php/ijcse/

УДК 004.432.2

СОПОСТАВЛЕНИЕ С ОБРАЗЦОМ В ЯЗЫКЕ ПРОГРАММИРОВАНИЯ PYTHON

[1]Салимова А.Р., [2]Васильева К.А.
МИРЭА - Российский технологический университет, Москва, Россия (119454, г. Москва, пр. Вернадского, 78), e-mail: [1]alnsalimova@mail.ru, [2]vasilievaxeniaa@gmail.com

В данной статье проводится анализ нового решения в языке программирования Python — сопоставления с образцом. В статье также рассматривается подобный шаблон в языке программирования Haskell. Помимо этого, сопоставление с образцом сравнивается с шаблоном Visitor, который реализован в языке программирования Python. Приводится пример простого калькулятора, реализованного при помощи сопоставления с образцом и шаблона Visitor. Также приводится пример тестирования обоих решений.

Ключевые слова: Python, сопоставление с образцом, Haskell.

PATTERN MATCHING IN PYTHON PROGRAMMING LANGUAGE

[1]Salimova A.R., [2]Vasilieva K.A.
MIREA - Russian Technological University, Moscow, Russia (119454, Moscow, Vernadskogo Ave., 78), e-mail: [1]alnsalimova@mail.ru, [2]vasilievaxeniaa@gmail.com

This article analyzes a new solution in the Python programming language — pattern matching. The article also discusses a similar pattern in the Haskell programming language. In addition, pattern matching is compared with the Visitor template, which is implemented in the Python programming language. An example of a simple calculator implemented using pattern matching and the Visitor template is given. An example of testing both solutions is also provided.

Keywords: Python, pattern matching, Haskell.

Введение

Сопоставление с образцом (англ. pattern matching) — метод анализа и обработки структур данных в языках программирования, основанный на выполнении определённых инструкций в зависимости от совпадения исследуемого значения с тем или иным образцом, в качестве которого может использоваться константа, предикат, тип данных или иная поддерживаемая языком конструкция [7]. Как правило, имеется возможность указать более одного образца и связанного с ним действия. Сопоставление с образцом часто встречается в функциональных языках программирования, таких как языки семейства ML и Haskell, в том числе в виде охранных выражений.

Match в языке Haskell.

Рассмотрение данной конструкции стоит начать с ее применения в функциональных языках программирования, а именно в языке Haskell.

Как описывалось ранее, сопоставление с образцом используется для сопоставления заданного значения и соответствующего возврата результата. Для начала, рассмотрим пример реализации сопоставления с образцом в Haskell. Синтаксис в данном языке предельно прост и понятен:

Листинг 1 – Простой пример синтексиса

```
имя_шаблона значение = возвращаемое_значение
```

Приведенные выше строки описывают создание шаблона. Так как выше уже было рассмотрено, что сопоставление с образцом используется для сопоставления значения с определенным образцом, рассмотрим пример с несколькими шаблонами. В Haskell шаблон позволяет сопоставить любой тип, такой как число, строка, символ, кортеж, список и т.д [1]. Само сопоставление с образцом можно выполнять, основываясь на предыдущем примере:

Листинг 2 – Простой пример сопоставления с образцом

```
newFunc :: (Integral a) => a -> String
newFunc 10 = "ten!"
newFunc 20 = "twenty!"
newFunc 30 = "thirty!"
newFunc x = "not matching anything here!!"
```

Как и в первом примере, задается имя шаблона, тип получаемого значения и тип возвращаемого значения. В данном случае последний шаблон используется как стандартный, то есть в тех случаях, когда ни один из предыдущих не подошел. Стоит отметить, что синтаксис у данного паттерна прост и примитивен, но данная конструкция является крайне полезной в задачах сопоставления.

Кроме простого сопоставления с образцом необходимо также привести пример сопоставления с образцом при помощи case, Так как данная конструкция является альтернативой к предыдущей реализации [2]. В данном случае синтаксис так же прост:

Листинг 3 – Простой пример сопоставления с образцом с помощью case

```
имя_шаблона значение =
  case значение of
    значение_1 -> действие
  значение_2 -> действие
  _ -> действие
```

Структура кода, как и в предыдущем примере, довольно проста. Вначале объявляется имя шаблона, затем название переменной, сопоставление которой будет осуществляться. После этого прописывается ключевое слово “case” для искомого значения и ключевое слово “of”. После этого следует перечень значений, которые могут быть сопоставлены, и действия, которые будут выполняться в случае совпадения с шаблоном. Также здесь может быть

применен символ wildcard (“_”). Данный символ обозначает блок программы, который выполнится, если ни одно из предыдущих значений не подойдет.

Так как основным примером иллюстрации сопоставления с образцом в Python в данной статье является пример с сопоставлением экземпляров классов, стоит рассмотреть пример подобной реализации на языке Haskell. В данном примере представлен рекурсивный тип данных. Создаваемый объект может иметь только два состояния — пуст или содержит значение. Данная конструкция достаточно удобна для понимания обработки составных типов данных в Haskell с помощью сопоставления с образцом, так как не содержит ошибки с исчерпаемостью альтернатив, которая будет рассмотрена далее [12]. В качестве простой функции для демонстрации сопоставления с образцом была реализована функция проверки значений всех листьев дерева.

Листинг 4 – Пример с рекурсивным типом данных

```
data Tree = Empty | Node Tree Integer Tree
   deriving (Show)

my_tree = (Node  (Node (Node Empty 4 Empty) 7 (Node Empty 7 Empty)) 1 (Node Empty 5 (Node Empty 2 Empty)))
smallTree:: Tree -> Bool
smallTree x = case x of
  Empty -> True
  Node a b c -> b < 10 && smallTree a && smallTree c

main :: IO()
main = print (smallTree my_tree)
```

Рассматривая язык программирования Haskell, стоит обязательно осветить тему исчерпаемости альтернатив. Сам термин означает, что в примененном паттерне описаны все возможные исходы [3]. Это защищает программу от ошибок и непредвиденной остановки. Важной особенностью реализации сопоставления с образцом в Haskell является то, что Haskell выявит ошибку связанную с исчерпаемостью альтернатив на этапе компиляции, что приведет к остановке сбора программы [4]. Однако в языке Python этого не происходит. Если какой-либо исход, который не описан в паттерне, будет подан в программу, Python остановит выполнение кода только тогда, когда дойдет до этого места в программе [6]. В Haskell строка, содержащая символ wildcard позволяет избежать ошибки исчерпаемости альтернатив, поэтому если данная строка будет отсутствовать, Haskell выдаст ошибку.

Завершая обзор сопоставления с образцом в языке Haskell, необходимо отметить, что сопоставление с образцом позволяет легко найти соответствующее значение внутри списка, кортежа, числа, строки и т.д. Кроме того, синтаксис для сопоставления с образцом прост в использовании и реализации в Haskell. Данная конструкция работает так же, как и любой другой язык программирования, где некоторые значения используются для сопоставления с шаблоном и получения желаемого результата. Далее перейдем к рассмотрению сопоставления с образцом в языке Python.

Match в языке Python

Как можно заметить из предыдущего примера, конструкция, похожая на данный паттерн, реализована уже во многих языках программирования, поэтому назвать ее новой нельзя. Однако без данной функции в Python было слишком много кода. Программы часто должны обрабатывать данные, которые различаются по типу, наличию атрибутов/ключей или количеству элементов. Данная конструкция может сравнивать не только константы, но и типы, атрибуты, а также может делать вложенные проверки.

Сопоставление с образцом представляет собой конструкцию для сопоставления шаблонов. В самой же конструкции есть функция match, которая может сопоставить исходное выражение с заданным шаблоном. Сама функция похожа на конструкцию if/else — если match находит совпадение с шаблоном, то выполняет заданные действия, в противном случае пропускает соответствующие действия [11]. Но, несмотря на свою схожесть с простой конструкцией, функциональных возможностей у match гораздо больше. Эта функция представляет возможность извлечения данных из структур с составным типом данных, а также возможность применения различных действий к разным частям структуры.

Конструкция оператора сопоставления выглядит следующим образом:

Листинг 5 – Сопоставление с образцом в Python

```
match expression:
        case pattern_1:
                action_1
        case pattern_2:
                action_2
        case _:
                default_action
```

Одной важной особенностью конструкции match является то, что блоки case нельзя оставлять пустыми. Такие конструкции присутствуют в некоторых языках, но в Python это недопустимо.

Кроме простых переменных в качестве шаблона могут выступать последовательности элементов, разделенные запятой и заключенные в скобки. В первую очередь рассмотрим кортежи. В этих наборах данных до запятой могут быть указаны как единичные значения, так и набор значений. Также здесь возможен пропуск элементов [8]. Кроме этого с помощью сопоставления с образцом возможно обрабатывать массивы. Данная обработка очень похожа на обработку кортежа. Здесь точно также предусмотрена обработка заданных значений, переменных и знака wildcard. Кроме этого можно сравнивать массивы неопределенной длинны, элементы массивов с многовариантными значениями и многовариантные массивы. В случае обработки массивов можно также передавать набор значений. Для этого нужно передать перечень величин, которые должен иметь атрибут, используя символ “|” [9].

Помимо картежей и массивов сопоставление с образцом позволяет проводить анализ словаря. С помощью данной конструкции можно проверить присутствие в словаре определенных ключей и значений. Кроме простого сравнения с заданными значениями сопоставление с образцом позволяет проверять набор значений в словаре и набор словарей. Чтобы получать неограниченное количество значений, необходимо воспользоваться двумя символами "*" [10].

Ключевой пример

Последний тип многомерных данных, который сопоставление с образцом позволяет обрабатывать — классы. В случае с классами происходит все то же самое, как и при обработке массивов, кортежей и словарей. В качестве шаблона задается объект класса. Далее в case вызывается конструктор, в котором задаются атрибуты класса. Значения атрибутов могут быть заданы конкретным значением или же переменной, в которую запишется значение при выполнении match. Также в качестве шаблона можно посылать объекты разных классов. Для этого нужно вызвать конструкторы этих классов и соединить их символом “|”. Также рекомендуется использовать символ wildcard — символ, который позволяет обрабатывать ситуацию, при которой ни один из шаблонов не подошел.

Данный паттерн имеет много достоинств, но возникает вопрос, действительно ли он полезен в Python? Для ответа на этот вопрос стоит сравнить сопоставление с образцом и шаблон Visitor.

Visitor — поведенческий шаблон проектирования, описывающий операцию, которая выполняется над объектами других классов [5]. При изменении Visitor нет необходимости редактировать обслуживаемые классы. Шаблон демонстрирует классический приём восстановления информации о потерянных типах, не прибегая к понижающему приведению типов.

Данный шаблон очень полезен в случаях, когда над разными классами нужно произвести одну и ту же или схожую операцию. Подобный пример будет рассмотрен далее.

В качестве примера будет рассмотрена реализация простого калькулятора, который может посчитать выражение, напечатать само выражение, а также вывести код стековой машины для данного примера.

Сама идея шаблона Visitor в данном примере очень проста. Создается класс Visitor с функцией, которая будет самостоятельно генерировать функции для нужного класса. Класс PrintVisitor содержит функции для печати нашего выражения. Как можно отметить, все они начинаются с ключевого слова visit и содержат в себе название класса. Класс CalcVisitor содержит операции для вычисления заданного выражения, а класс StackVisitor — функции для печати кода стековой машины.

В функциональных языках программирования достаточно часто применяют конструкцию вида:

Листинг 6 – Популярная конструкция в функциональных языках

```
def visit(self, num):
    op = 'visit_'+type(num).__name__
    return getattr(self, op)(num)
```

Она позволяет динамически создавать функции для каждого класса. В данном случае такая конструкция будет располагаться в классе Visitor. В реализации простого калькулятора обоими способами будут использоваться следующие классы:

Листинг 7 – Используемые классы для первого варианта решения

```
class Num:
    def __init__(self, num):
        self.num = num
```

```
class BinOp:
    def __init__(self, num1, num2):
        self.num1 = num1
        self.num2 = num2

class AddBinOp):
    pass

class Mul(BinOp):
    pass
```

Теперь рассмотрим сам программный код.

Листинг 8 – Программный код для первого варианта решения

```
class PrintVisitor(Visitor):
    def visit_Num(self, num):
        return str(num.num)

    def visit_BinOp(self, num, operation):
        return f'({self.visit(num.num1)} {operation} {self.visit(num.num2)})'

    def visit_Add(self, num):
        return self.visit_BinOp(num, '+')

    def visit_Mul(self, num):
        return self.visit_BinOp(num, '*')

class CalcVisitor(Visitor):
    def visit_Num(self, new):
        return str(new.num)

    def visit_BinOp(self, num, operation):
        return eval(f'{self.visit(num.num1)}{operation}{self.visit(num.num2)}')

    def visit_Add(self, num):
        return self.visit_BinOp(num, '+')

    def visit_Mul(self, num):
        return self.visit_BinOp(num, '*')

class StackVisitor(Visitor):
    def visit_Num(self, new):
        return f'PUSH {str(new.num)}\n'
```

```
    def visit_BinOp(self, num, operation):
        return f'{self.visit(num.num1)}{self.visit(num.num2)}{operation}\n'

    def visit_Add(self, num):
        return self.visit_BinOp(num, 'Add')

    def visit_Mul(self, num):
        return self.visit_BinOp(num, 'Mul')
```

Как можно отметить, в программе реализованы классы без лишних строчек кода. Но, если появится необходимость добавить какую-то новую операцию, например вычитание, необходимо будет инициализировать новый класс и добавить новые функции во все классы. Данный метод не очень удобен, если программа предполагает наличие многих классов или добавление новых классов с течением времени. Но шаблон Visitor позволяет упростить выполнение различных операций, так как устраняет дублирование кода для разных объектов.

Теперь рассмотрим реализацию этой же задачи, только с использованием сопоставления с образцом. В данной реализации понадобятся классы, приведенные в Листинге 9 ниже.

Листинг 9 – Классы для второго варианта решения

```
class Num:
    def __init__(self, num):
        self.num = num
class Add:
    def __init__(self, num1, num2):
        self.num1 = num1
        self.num2 = num2
class Mul:
    def __init__(self, num1, num2):
        self.num1 = num1
        self.num2 = num2
```

В данном случае шаблоны отсутствуют, а для каждого case необходимо прописывать свое действие. По этой причине нет возможности сделать универсальную реализацию для всех типов вычислений, возникает необходимость в написании фактически одного и того же кода для каждого варианта.

Листинг 10 – Программный код для второго варианта решения

```
def calc(value):
    match value:
        case Add(num1=num1, num2=num2):
            return calc(num1) + calc(num2)
        case Mul(num1=num1, num2=num2):
            return calc(num1) * calc(num2)
        case Num(num=num):
```

```
            return num

    def print_str(value):
        match value:
            case Add(num1=num1, num2=num2):
                return f"({print_str(num1)} + {print_str(num2)})"
            case Mul(num1=num1, num2=num2):
                return f"({print_str(num1)} * {print_str(num2)})"
            case Num(num=num):
                return num

    def print_stack(value):
        match value:
            case Add(num1=num1, num2=num2):
                return f"{print_stack(num1)}{print_stack(num2)}ADD\n"
            case Mul(num1=num1, num2=num2):
                return f"{print_stack(num1)}{print_stack(num2)}MUL\n"
            case Num(num=num):
                return f'PUSH {num}\n'
```

Как видно из Листинга 10, реализация с использованием сопоставления с образцом занимает меньше строк кода и выглядит более понятно и менее громоздко. Также стоит отметить, при необходимости добавления новой операции, необходимо будет инициализировать новый класс и добавить обработку нового case в каждую функцию. Нет необходимости в создании новых функций и добавлении их в каждый класс, как это происходит в предыдущем примере. Поэтому весь код выглядит проще и позволяет гораздо быстрее обновлять программу.

Исходя из двух примеров, можно сделать вывод, что сопоставление с образцом целесообразно использовать для сопоставления объектов, так как данная конструкция осуществляет всю обработку объекта класса внутри себя. Это позволяет избавиться от громоздкого кода и осуществить простое и понятное обновление программного кода. Однако в остальных ситуациях не совсем уместно использовать конструкцию match, так как на данный момент в Python существуют более универсальные конструкции и шаблоны.

Тестирование

После рассмотрения каждого варианта реализации стоит проиллюстрировать, какая из программ будет работать с большей скоростью. Для этого было написано пять различных по сложности и составу тестов. Сами тесты можно рассмотреть в Таблице 1:

Таблица 1 — Перечень тестов

№	Состав теста
1	Add(Num(5), Mul(Num(4), Num(6)))
2	Mul(Num(8), Mul(Num(9), Mul(Num(10), Mul(Num(2), Mul(Num(3), Mul(Num(5), Mul(Num(4), Num(6))))))))
3	Add(Num(7), Add(Num(8), Add(Num(9), Add(Num(10), Add(Num(2), Add(Num(3), Add(Num(5), Add(Num(4), Num(6))))))))))
4	Add(Num(7), Mul(Num(8), Add(Num(9), Mul(Num(10), Add(Num(2), Mul(Num(3), Add(Num(5), Mul(Num(4), Num(6))))))))))
5	Add(Num(7), Mul(Num(8), Add(Num(9), Mul(Num(10), Add(Num(2), Mul(Num(3), Add(Num(5), Mul(Num(4), Add(Num(6), Mul(Num(10), Add(Num(18), Num(13))))))))))))

После выполнения данных тестов для обеих функций, были получены результаты, которые представлены в Таблице 2.

Таблица 2 — Время выполнения тестов для обоих решений

№	Время выполнения visitor/мкс	Время выполнения match/мкс
1	59	18,2
2	140,8	40,6
3	164,5	47,6
4	210,8	52
5	245,9	68

Теперь можно оценить полученные результаты. Как видно из предыдущей таблицы, программа, реализованная с помощью сопоставления с образцом, во всех случаях выполняется быстрее, чем программа с использованием шаблона Visitor. Если обратить внимание на четвертый тест, можно заметить, что только в этом случае шаблон Visitor приблизился по скорости к сопоставлению с образцом. Но тем не менее выполнялся немного дольше. Исходя из этого, можно сделать вывод, что сопоставление с образцом выигрывает по скорости у шаблона Visitor. Таким образом, с точки зрения читаемости кода и скорости выполнения, более целесообразно будет использование сопоставления с образцом.

Заключение

В заключение данной статьи стоит отметить, что при использовании сопоставления с образцом программный код может стать визуально проще и понятнее. Данный паттерн будет достаточно удобен при сопоставлении объектов классов, в иных ситуациях рациональнее будет использование более простых и универсальных конструкций языка. Это подтверждает пример, рассмотренный выше. Использование сопоставления с образцом не является универсальным решением, поэтому не стоит его использовать вместо привычного if/else, если нет в этом необходимости.

Список литературы

1. Душкин Р.В. Функциональное программирование на языке Haskell. Учебное пособие. ДМК Пресс. Москва, 2008. — 608 с. (Дата обращения 23.03.2022)
2. Скорик И.В., Соколова М.И. Особенности функционального программирования на примере языка. Haskell, 2020. — 172 с . (Дата обращения 23.03.2022)
3. ШевченкоД.В. О Haskell по-человечески, издание 2.0, 2016. — 147 с. (Дата обращения 23.03.2022)
4. Антон Холомьёв, Учебник по Haskell. 3-е издание. 2012. — 329 с. (Дата обращения 23.03.2022)
5. Ден Бейдер, Чистый Python. Тонкости программирования для профи. 2018 . — 529 с. (Дата обращения 23.03.2022)
6. Аллен Б. Дауни, Изучение сложных систем с помощью Python. 2019 — 300 с. (Дата обращения 23.03.2022)
7. David Beazley, Brain K.Jone, Python Cookbook 3d edition. 2013 — 667 с. (Дата обращения 28.03.2022)
8. Manuel Krebber1, Henrik Barthels. MatchPy: Pattern Matching in Python, RWTH Aachen University, AICES. 2018 — 100 с. (Дата обращения 23.03.2022)
9. Krebber, Manuel, Henrik Barthels, and Paolo Bientinesi. “Efficient Pattern Matching in Python.” In Proceedings of the 7th Workshop on Python for High-Performance and Scientific Computing. [Электронный ресурс]: https://doi.org/10.1145/ 3149869.3149871. (Дата обращения 28.03.2022)
10. Соответствие структуре шаблона, конструкция match/case. [Электронный ресурс]:https://docs-python.ru/tutorial/tsikly-upravlenie-vetvleniem-python/konstruktsija-match-case/. (Дата обращения 28.03.2022)
11. Python 3.10 Match — A New Way to Find Patterns. Режим доступа: https://medium.com/short-bits/python-3-10-match-a-new-way-to-find-patterns-8452d1460407. (Дата обращения 28.03.2022)
12. Neil Mitchell, Colin Runciman. A Static Checker for Safe Pattern Matching in Haskell. 2007 — 240 с. (Дата обращения 28.03.2022)

References

1. Dushkin R.V. Functional programming in Haskell. Tutorial. DMK Press. Moscow, 2008. - 608 p. (Accessed 23.03.2022)
2. Skorik I.V., Sokolova M.I. Features of functional programming on the example of the language. Haskell, 2020. - 172 p. (Accessed 23.03.2022)
3. Shevchenko D.V. About Haskell in a human way, edition 2.0, 2016. - 147 p. (Accessed 23.03.2022)
4. Anton Kholomiev, Haskell Tutorial. 3rd edition. 2012. - 329 p. (Accessed 23.03.2022)
5. Den Bader, Pure Python. The subtleties of programming for the pros. 2018. — 529 p. (Accessed 23.03.2022)
6. Allen B. Downey, Learning Complex Systems with Python. 2019 - 300 p. (Accessed 23.03.2022)

7. David Beazley, Brain K. Jone, Python Cookbook 3d edition. 2013 - 667 p. (Accessed 28.03.2022)
8. Manuel Krebber1, Henrik Barthels. MatchPy: Pattern Matching in Python, RWTH Aachen University, AICES. 2018 - 100 p. (Accessed 23.03.2022)
9. Krebber, Manuel, Henrik Barthels, and Paolo Bientinesi. “Effective Pattern Matching in Python.” In Proceedings of the 7th Workshop on Python for High-Performance and Scientific Computing. [Electronic resource]: https://doi.org/10.1145/ 3149869.3149871. (Accessed 28.03.2022)
10. Compliance with the template structure, match/case construction. [Electronic resource]: https://docs-python.ru/tutorial/tsikly-upravlenie-vetvleniem-python/konstruktsija-match-case/. (Accessed 28.03.2022)
11. Python 3.10 Match - A New Way to Find Patterns. Access Mode: https://medium.com/short-bits/python-3-10-match-a-new-way-to-find-patterns-8452d1460407. (Accessed 28.03.2022)
12. Neil Mitchell, Colin Runciman. A Static Checker for Safe Pattern Matching in Haskell. 2007 - 240 p. (Accessed 28.03.2022)