第13章. 模式匹配

13-1. match语句概述

（教程：4.6）
（语言参考手册：8.6.1～8.6.3、8.6.4.5）

“模式匹配（pattern matching）”用于判断某个对象是否具有特定的模式，本质上是一系列==比较、ID检测和isinstance()判断。理论上，可以用if语句实现模式匹配，但这样做在很多情况下显得相当笨拙。为了精简代码，Python提供了match语句，它源自C中的switch语句，但执行逻辑存在差异，功能被极大地扩展，其语法为：

```
match subject_expr:
    case pattern1 [guard1]:
        suite
    case pattern2 [guard2]:
        suite
    ...
```

其中subject_expr代表主题表达式；pattern1，pattern2，……代表模式；guard1，guard2，……代表约束项。而约束项的语法为：

```
if expr
```

其含义是对表达式expr求值，如果得到的对象的逻辑值检测结果为True则接受该主题，否则拒绝该主题。

match语句的执行逻辑是这样的：首先设置一个游标，让其指向第一个case子句，然后执行如下算法：

➤ 步骤一：对subject_expr求值，得到的对象即用来匹配的主题。
➤ 步骤二：尝试将主题与游标指向的case子句中的patternN匹配。如果匹配成功，则前往步骤三；否则前往步骤五。
➤ 步骤三：如果patternN后面具有guardN，则基于guardN进行判断，接受该主题则前往步骤四，否则前往步骤五。如果guardN被省略，则直接前往步骤四。
➤ 步骤四：执行游标指向的case子句下的suite，然后前往步骤六。
➤ 步骤五：如果游标指向的case子句下面还有case子句，则让游标指向后者，回到步骤二；否则，前往步骤六。
➤ 步骤六：match语句执行完成。

下面用一个例子说明match语句的执行逻辑：

```
#定义一个模拟菜单的函数。
def menu_simulate(choice):
    """This function simulates a menu.

        The virtual menu provides three items:
        Setting: chosen by input 1
        Help: chosen by input 2
        Logout: chosen by input 3
    """

    legal = False

    match choice:
        case 1:
            legal = True
            print("Setting\n")
        case 2:
            legal = True
            print("Help\n")
        case 3:
            legal = True
            print("Logout\n")

    if legal:
        return 0
    else:
        return 1


#仅当脚本在顶层环境中运行时才执行。
if __name__ == '__main__':
    choice = 0
    while choice != 3:
        choice = int(input("Menu:\n1 Setting\n2 Help\n3 Logout\n"))
        if menu_simulate(choice):
            print(str(choice) + ' is an illegal menu item!\n')
```

请将上述代码保存为match1.py，然后通过如下命令行验证：

```
$ python3 match1.py
Menu:
1 Setting
2 Help
3 Logout
1
Setting

Menu:
1 Setting
2 Help
3 Logout
2
Help

Menu:
1 Setting
2 Help
```

```
3 Logout
4
4 is an illegal menu item!

Menu:
1 Setting
2 Help
3 Logout
3
Logout
```

在该列子中函数menu_simulate()包含的match语句中的主题来自用户的输入，它被转换为一个整数。三个case子句中的模式都只是一个简单的整数（下节会说明这是字面值模式的一种），且没有约束项。主题与模式的匹配本质上是一次==比较，当主题不是1、2或3时，match语句将不执行任何suite，因为该主题不匹配任意case子句给出的模式。

从上面的例子可以看出，有时候主题可能具有的模式非常多，而有效模式却只有几种。在这种情况下，一种策略是在match语句中仅针对主题匹配有效模式的情况进行处理，将无效模式留给其他代码，如上面的例子；另一种策略是在match语句中用一个可以匹配主题可能具有的所有模式的case子句兜底，然后在该case子句对应的suite中处理主题匹配无效模式的情况。下面的match2.py是对match1.py的改写：

```
def menu_simulate(choice):
    """This function simulates a menu.

        The virtual menu provides three items:
        Setting: chosen by input 1
        Help: chosen by input 2
        Logout: chosen by input 3
    """

    match choice:
        case 1:
            print("Setting\n")
        case 2:
            print("Help\n")
        case 3:
            print("Logout\n")
        case _:    #这是兜底case子句。
            print(str(choice) + ' is an illegal menu item!\n')


if __name__ == '__main__':
    choice = 0
    while choice != 3:
        choice = int(input("Menu:\n1 Setting\n2 Help\n3 Logout\n"))
        menu_simulate(choice)
```

可以用相同的方法验证match2.py与match1.py的功能完全相同，但match2.py更加简洁，更具有逻辑性，因为对菜单的操作完全由menu_simulate()处理。

从match2.py可以看出，“_”可以匹配任意主题，被称为“通配符模式（wildcard pattern）”。在第2章已经说明，“match”、“case”和“_”是仅在match语句中才有特殊含义的软关键字。这里要进一步强调，“_”仅在case子句中作为模式时才具有特殊含

义，被解读为通配符模式。此外，“_”本身可以匹配任意主题，但依然受约束项的影响，当具有约束项时就不一定能实现兜底的效果，例如如下case子句中的suite在任何情况下都不会被执行：

```
case _ if False:
    pass
```

如果一个case子句使用了通配符模式，且没有约束项，则必定能匹配任何主题。下面还会介绍其他构造能匹配任何主题的case子句的方法。这样的case子句又被称为“不可拒绝的case块（irrefutable case block）”，在每个match语句中至多出现一次，且如果出现则必须是最后一个case子句，否则会被视为语法错误（因为它导致后续的case子句永远不会被使用）。

易知，决定match语句中哪个suite被执行的最重要因素是模式，而match语句支持多种模式。下面将分类讨论match语句支持的所有模式（除了本节已经介绍的通配符模式）。

13-2. 匹配字面值

（语言参考手册：8.6.4.3）

“字面值模式（literal patterns）”是最基本的模式，包括所有字面值， True、False和None。

我们已经知道，数字字面值又进一步分为整数字面值、浮点数字面值和虚数字面值。主题与数字字面值匹配本质上是进行==比较，而整数、浮点数和复数之间进行==比较的结果与数学上保持一致。这就导致了一个数字字面值模式可以匹配多个数字，下面是一个例子：

```
def number_match(a, b):
    match a + b:
        case 0:
            print("They are opposite numbers.")
        case 10:
            print("They are complementary numbers against 10.")
        case 100:
            print("They are complementary numbers against 100.")
        case _:
            print("They have no special relations.")
```

请将上述代码保存为match3.py，然后通过如下命令行和语句验证：

```
$ python3 -i match3.py

>>> number_match(-5, 5.0)
They are opposite numbers.
>>> number_match(2.6+5j, 7.4-5j)
They are complementary numbers against 10.
>>> number_match(102, -2.0)
```

```
They are complementary numbers against 100.
>>> number_match(-1+9j, 1-9j)
They are opposite numbers.
>>> number_match(-1+9j, 1+9j)
They have no special relations.
>>>
```

字符串字面值作为模式时，既不能具有r/R前缀，也不能具有f/F前缀，但可以是通过三引号括起来的多行字符串。此时本质上依然是主题与模式进行==比较。下面是一个例子：

```
def to_arabic(s):
    match s.strip().lower():
        case 'zero':
            return 0
        case 'one':
            return 1
        case 'two':
            return 2
        case 'three':
            return 3
        case 'four':
            return 4
        case 'five':
            return 5
        case 'six':
            return 6
        case 'seven':
            return 7
        case 'eight':
            return 8
        case 'nine':
            return 9
        case _:
            return None
```

请将上述代码保存为match4.py，然后通过如下命令行和语句验证：

```
$ python3 -i match4.py

>>> to_arabic('One')
1
>>> to_arabic('FIVE ')
5
>>> to_arabic('    zero')
0
>>> to_arabic('Eighteen')
>>>
```

字节串字面值作为模式时，不仅能匹配字节串，还能匹配字节数组，因为同样进行的是==比较。下面的例子说明了这点：

```
>>> match b'a':
...     case b'a':
```

```
...         print('OK')
...
OK
>>> match bytearray(b'a'):
...     case b'a':
...         print("OK")
...
OK
>>> match b'a':
...     case bytearray(b'a'):
...         print("OK")
...
>>>
```

注意该例子同时说明了，当以“bytearray(...)”作为模式时，无法匹配相应的字节串，哪怕它们相等。这是因为“bytearray(...)”并非字面值模式，而是后面将要讨论的类模式。

当以True、False和None作为模式时，进行的是ID检测而非==比较，以避免触发逻辑值检测。下面的例子说明了这点：

```
def match_specials(v):
    match v:
        case True:
            print("True")
        case False:
            print("False")
        case None:
            print("None")
        case _:
            print("Unkown")
```

请将上述代码保存为match5.py，然后通过如下命令行和语句验证：

```
$ python3 -i match5.py

>>> match_specials(True)
True
>>> match_specials(False)
False
>>> match_specials(None)
None
>>> match_specials(1)
Unknown
>>> match_specials(bool(1))
True
>>> match_specials(0)
Unknown
>>> match_specials(bool(0))
False
>>> match_specials(bool(None))
False
>>>
```

13-3. 匹配值

（教程：4.6）
（语言参考手册：8.6.4.6）

match语句支持以对象的属性作为模式，这等价于将主题与该属性引用的对象进行==比较，因此被称为“值模式（value patterns）”。这一机制的重要性在于，不需要修改match语句本身，只需要改变属性引用的对象，就可以改变match语句的功能。下面是一个例子：

```
class PrimaryColor:
    def __init__(self):
        self.red = 'red'
        self.green = 'green'
        self.blue = 'blue'


pri_co = PrimaryColor()


def pure_color(value):
    match value:
        case pri_co.red:
            print("This color is purely red.")
        case pri_co.green:
            print("This color is purely green.")
        case pri_co.blue:
            print("This color is purely blue.")
        case _:
            print("This color is mixed.")
```

请将上述代码保存为match6.py，然后通过如下命令行和语句验证：

```
python3 -i match6.py

>>> pure_color('red')
This color is purely red.
>>> pure_color('green')
This color is purely green.
>>> pure_color('blue')
This color is purely blue.
>>> pure_color('yellow')
This color is mixed.
>>>
>>> pure_color('#ff0000')
This color is mixed.
>>> pure_color('#00ff00')
This color is mixed.
>>> pure_color('#0000ff')
This color is mixed.
>>> pure_color('#ffff00')
This color is mixed.
>>>
>>> pri_co.red = '#ff0000'
>>> pri_co.green = '#00ff00'
>>> pri_co.blue = '#0000ff'
>>>
```

```
>>> pure_color('red')
This color is mixed.
>>> pure_color('green')
This color is mixed.
>>> pure_color('blue')
This color is mixed.
>>> pure_color('yellow')
This color is mixed.
>>>
>>> pure_color('#ff0000')
This color is purely red.
>>> pure_color('#00ff00')
This color is purely green.
>>> pure_color('#0000ff')
This color is purely blue.
>>> pure_color('#ffff00')
This color is mixed.
>>>
```

在该例子中，pure_color()函数以全局变量pri_co引用的对象的属性red、green和blue作为模式。起初它们分别引用字符串“red”、“green”和“blue”，故无法将“#ff0000”、“#00ff00”和“#0000ff”识别为纯色；但它们被修改为引用后者后，后者被识别为纯色，但前者不再被识别为纯色。

13-4. 匹配类

（教程：4.6）
（语言参考手册：3.3.10、8.6.4.10）

match语句还支持将主题匹配到一个类，且指定属性匹配到某些子模式。为此而引入的模式被称为“类模式（class patterns）”。类模式的格式为：

```
clsname([pos_pattern, ...,] key=key_pattern, ...])
```

其中clsname为引用某个以type作为元类的类对象的标识符，用于对主题的类型进行限制；pos_pattern代表位置模式参数，key_pattern代表关键字模式参数，都用于对主题的属性引用的对象进行限制。此时匹配的具体算法是：

➤ 步骤一. 首先对主题进行isinstance()判断。如果它不是clsname的实例，则匹配失败；否则前往步骤二。
➤ 步骤二. 如果没有模式参数，则匹配成功。否则，依次尝试匹配每个模式参数和主题的相应属性，具体进行==比较还是ID测试取决于模式参数的类型，仅当所有模式参数都匹配成功时才匹配成功。

下面的例子说明了没有模式参数的类模式的用法（注意该例子导入了Point2D.py和Point2DFixed.py中定义的类，请确保这两个模块可被访问）：

```
from Point2D import Point2D
from Point2DFixed import Point2DFixed

def match_point(p):
    match p:
        case Point2D():
            print("This is a Point2D point.")
        case Point2DFixed():
            print("This is a Point2DFixed point.")
```

请将上述代码保存为match7.py，然后通过如下命令行和语句验证：

```
$ python3 -i match7.py

>>> p1 = Point2D(0, 0)
>>> p2 = Point2DFixed(0, 0)
>>> p3 = (0, 0)
>>> match_point(p1)
This is a Point2D point.
>>> match_point(p2)
This is a Point2DFixed point.
>>> match_point(p3)
>>>
```

下面的例子中的类模式仅具有关键字模式参数：

```
from Point2DFixed import Point2DFixed

def match_point(p):
    match p:
        case Point2DFixed(x=0, y=0):
            print("This point is the origin.")
        case Point2DFixed(x=0):
            print("This point is on the y-axis.")
        case Point2DFixed(y=0):
            print("This point is on the x-axis.")
        case Point2DFixed(area=1):
            print("This point is on the unit circle.")
        case _:
            print("This point is not special.")
```

请将上述代码保存为match8.py，然后通过如下命令行和语句验证：

```
$ python3 -i match8.py

>>> p1 = Point2DFixed(0, 0)
>>> p2 = Point2DFixed(0, 1)
>>> p3 = Point2DFixed(-1, 0)
>>> p4 = Point2DFixed(-1, 1)
>>> p5 = Point2DFixed(0.8, 0.6)
>>> match_point(p1)
This point is the origin.
>>> match_point(p2)
```

```
This point is on the y-axis.
>>> match_point(p3)
This point is on the x-axis.
>>> match_point(p4)
This point is not special.
>>> match_point(p5)
This point is not special.
>>> p5.area = 1
>>> match_point(p5)
This point is on the unit circle.
>>>
```

从这个例子可以看出，关键字模式参数中的关键字就是属性名，其含义是主题具有该属性名的属性所引用的对象必须匹配指定的子模式。如果主题不具有采用该属性名的属性，或者采用该属性名的属性所引用的对象不匹配指定的子模式，则匹配失败。

位置模式参数中没有关键字，它们又是如何指定属性的呢？这依赖于让类模式中的类实现魔术属性__match_args__，该属性引用一个字符串元组，其中每个字符串都代表一个属性名。第n个位置模式参数对应的属性名是subject.__match_args__[n-1]。下面的例子说明了位置模式参数的用法：

```
from Point2DFixed import Point2DFixed


class Point(Point2DFixed):
    __match_args__ = ('x', 'y')


def match_point(p):
    match p:
        case Point(0, 0):
            print("This point is the origin.")
        case Point(0):
           print("This point is on the y-axis.")
       case Point(_, 0):
            print("This point is on the x-axis.")
        case Point(_, _, area=1):
           print("This point is on the unit circle.")
        case _:
            print("This point is not special.")
```

请将上述代码保存为match9.py，然后通过如下命令行和语句验证：

```
$ python3 -i match9.py

>>> p1 = Point(0, 0)
>>> p2 = Point(0, 1)
>>> p3 = Point(-1, 0)
>>> p4 = Point(-1, 1)
>>> p5 = Point(0.8, 0.6)
>>> match_point(p1)
This point is the origin.
>>> match_point(p2)
This point is on the y-axis.
>>> match_point(p3)
```

```
This point is on the x-axis.
>>> match_point(p4)
This point is not special.
>>> match_point(p5)
This point is not special.
>>> p5.area = 1
>>> match_point(p5)
This point is on the unit circle.
>>>
```

在该例子中，由于Point.__match_args__引用了('x', 'y')，所以第一个位置模式参数对应属性x，第二个位置模式参数对应属性y。该例子同时说明了，位置模式参数和关键字模式参数可以同时存在，但后者必须位于前者之后。

当然，一条match语句中的类模式可以分别引用不同的类对象，其模式参数也可以自由组合。下面是一个例子：

```
class Point(Point2DFixed):
    __match_args__ = ('x', 'y')


def match_point(p):
    match p:
        case Point2D(x=0, y=0):
            print("This Point2D point is the origin.")
        case Point2DFixed(x=0):
            print("This Point2DFixed point is on the y-axis.")
        case Point(_, 0):
            print("This Point point is on the x-axis.")
        case Point2D():
            print("This Point2D point is not special.")
        case Point2DFixed():
            print("This Point2DFixed point is not special.")
        case Point():
            print("This Point point is not special.")
```

请将上述代码保存为match10.py，然后通过如下命令行和语句验证：

```
$ python3 -i match10.py

>>> p1 = Point2D(0, 0)
>>> p2 = Point2D(0, 1)
>>> p3 = Point2D(-1, 0)
>>> p4 = Point2D(-1, 1)
>>> p5 = Point2DFixed(0, 0)
>>> p6 = Point2DFixed(0, 1)
>>> p7 = Point2DFixed(-1, 0)
>>> p8 = Point2DFixed(-1, 1)
>>> p9 = Point(0, 0)
>>> p10 = Point(0, 1)
>>> p11 = Point(-1, 0)
>>> p12 = Point(-1, 1)
>>> match_point(p1)
This Point2D point is the origin.
>>> match_point(p2)
This Point2D point is not special.
```

```
>>> match_point(p3)
This Point2D point is not special.
>>> match_point(p4)
This Point2D point is not special.
>>> match_point(p5)
This Point2DFixed point is on the y-axis.
>>> match_point(p6)
This Point2DFixed point is on the y-axis.
>>> match_point(p7)
This Point2DFixed point is not special.
>>> match_point(p8)
This Point2DFixed point is not special.
>>> match_point(p9)
This Point2DFixed point is on the y-axis.
>>> match_point(p10)
This Point2DFixed point is on the y-axis.
>>> match_point(p11)
This Point point is on the x-axis.
>>> match_point(p12)
This Point2DFixed point is not special.
>>>
```

上述例子的验证结果可能出乎你的意料，但请注意Point2DFixed是Point的基类，所以任何Point的实例也是Point2DFixed的实例。

最后，当类模式的clsname对应的是如下内置类型之一时，对于位置模式参数将采取特殊处理：bool、int、float、str、bytes、bytearray、tuple、list、dict、set和frozenset。这些内置类型仅支持一个模式参数，且必须是位置模式参数，而用于匹配该参数的是主题本身而非它的某个属性。这一语法特性的主要功能是同时限定主题的类型与值。作为一些典型的例子，类模式int(0)特指整数0，无法匹配0j和0.0；而类模式float(-3.5)特指浮点数-3.5，无法匹配-3.5+0j。

13-5. 多个模式的合并

（教程：4.6）
（语言参考手册：8.6.3、8.6.4.1）

很多时候，match语句需要对多个模式做相同的处理，而如果为每个模式都提供一个case子句，代码就会显得冗长。为了追求简洁，我们可以用“|”将多个模式连接起来，当成一个模式使用，即：

```
pattern1 | pattern2 | ... | patternN
```

这种模式被称为“或模式（or patterns）”，表示按照pattern1、pattern2、……、patternN的顺序依次尝试匹配主题与子模式，只要有一个子模式匹配成功就无需尝试后续的子模式，仅当所有子模式都不匹配时才失败。

或模式允许包含能匹配任何主题的子模式，但它必须是最后一个子模式，否则被视为语法错误。显然，包含能匹配任何主题的子模式的或模式在没有约束项的情况下形成不可拒绝的

case块。一般而言，没必要让或模式包含能匹配任何主题的子模式，因为这与通配符模式没有区别。

下面是一个或模式的简单例子，它将英文字母分为元音字母和辅音字母两类：

```
def identify_vowel(letter):
    match letter.lower():
        case 'a' | 'e' | 'i' | 'o' | 'u':
            print("This letter is a vowel.")
        case _:
            print("This letter is a consonant.")
```

请将上述代码保存为match11.py，然后通过如下命令行和语句验证：

```
$ python3 -i match11.py

>>> identify_vowel('l')
This letter is a consonant.
>>> identify_vowel('A')
This letter is a vowel.
>>> identify_vowel('C')
This letter is a consonant.
>>> identify_vowel('e')
This letter is a vowel.
>>>
```

注意如果不使用或模式，则该例子中的match语句需要包含6个case子句，而不是仅2个。

下面是一个以类模式作为或模式的子模式的例子：

```
from Point2D import Point2D
from Point2DFixed import Point2DFixed

def match_point(p):
    match p:
        case Point2D(x=0, y=0) | Point2DFixed(x=0, y=0) :
            print("This point is the origin.")
        case Point2D(x=0) | Point2DFixed(x=0) :
            print("This point is on the y-axis.")
        case Point2D(y=0) | Point2DFixed(y=0):
            print("This point is on the x-axis.")
        case Point2D() | Point2DFixed():
            print("This Point point is not special.")
```

请将上述代码保存为match12.py，然后通过如下命令行和语句验证：

```
$ python3 -i match12.py

>>> p1 = Point2D(0, 0)
>>> p2 = Point2D(0, 1)
```

```
>>> p3 = Point2D(-1, 0)
>>> p4 = Point2D(-1, 1)
>>> p5 = Point2DFixed(0, 0)
>>> p6 = Point2DFixed(0, 1)
>>> p7 = Point2DFixed(-1, 0)
>>> p8 = Point2DFixed(-1, 1)
>>> match_point(p1)
This point is the origin.
>>> match_point(p2)
This point is on the y-axis.
>>> match_point(p3)
This point is on the x-axis.
>>> match_point(p4)
This Point point is not special.
>>> match_point(p5)
This point is the origin.
>>> match_point(p6)
This point is on the y-axis.
>>> match_point(p7)
This point is on the x-axis.
>>> match_point(p8)
This Point point is not special.
>>>
```

13-6. 在匹配过程中赋值

（教程：4.6）
（语言参考手册：8.6.2、8.6.3、8.6.4.2、8.6.4.4）

第3章已经说明match语句是将对象绑定到标识符的操作之一，这是因为在匹配过程中可以同时完成赋值。这有点类似于赋值表达式，但存在微妙的不同。

在match语句中实现赋值的第一种方法是使用“捕获模式（capture patterns）”，也就是一个标识符。捕获模式也可以匹配任何主题，当不存在约束项时形成不可拒绝的case块。捕获模式与通配符模式的区别在于前者确认匹配之后也就完成了赋值，被赋值的标识符可以在约束项和suite中使用。注意该标识符或者引用主题本身（以捕获模式作为模式），或者引用主题的某个属性（以捕获模式作为模式参数）。

当主题本身就是一个标识符时，我们可以在约束项和suite中直接使用该标识符来访问主题。此时捕获模式通常被作为类模式的模式参数，以使标识符引用主题的某个属性。请看下面的例子：

```
import math
from Point2DFixed import Point2DFixed


class Point(Point2DFixed):
    __match_args__ = ('x', 'y')


def match_point(p):
    match p:
        case Point(0, 0):
            print("This point is the origin.")
```

```
            case Point(0, y):
                print("This point is on the y-axis, distance from origin is " +
str(abs(y)) + ".")
            case Point(x, 0):
                print("This point is on the x-axis, distance from origin is " +
str(abs(x)) + ".")
            case Point(x, y) if abs(math.sqrt(x**2 + y**2) - 1.0) < 1e-100:
                print("This point is on the unit circle.")
            case _:
                print("This point is not special.")
```

请将上述代码保存为match13.py，然后通过如下命令行和语句验证：

```
$ python3 -i match13.py

>>> p1 = Point(0, 0)
>>> p2 = Point(0, 1)
>>> p3 = Point(-1, 0)
>>> p4 = Point(-1, 1)
>>> p5 = Point(0.8, 0.6)
>>> match_point(p1)
This point is the origin.
>>> match_point(p2)
This point is on the y-axis, distance from origin is 1.
>>> match_point(p3)
This point is on the x-axis, distance from origin is 1.
>>> match_point(p4)
This point is not special.
>>> match_point(p5)
This point is on the unit circle.
>>>
```

注意在该例子中match语句通过捕获模式x和y提取出了点的x坐标和y坐标，并在suite和约束项中使用它们。

当主题是非单个标识符形成的表达式时，我们可以用捕获模式代替通配符模式作为最后一个case子句，以使一个标识符引用主题表达式的计算结果。请看下面的例子：

```
def to_arabic(s):
    match s.strip().lower():
        case 'zero':
            print('zero means 0.')
        case 'one':
            print('one means 1.')
        case 'two':
            print('two means 2.')
        case 'three':
            print('three means 3.')
        case 'four':
            print('four means 4.')
        case 'five':
            print('five means 5.')
        case 'six':
            print('six means 6.')
        case 'seven':
            print('seven means 7.')
```

```
            case 'eight':
                print('eight means 8.')
            case 'nine':
                print('nine means 9.')
            case s:
                print(s + ' is not a number.')
```

请将上述代码保存为match14.py，然后通过如下命令行和语句验证：

```
$ python3 -i match14.py

>>> to_arabic(' FOUR')
four means 4.
>>> to_arabic(' eIGht  ')
eight means 8.
>>> to_arabic(' apple ')
apple is not a number.
>>> to_arabic('BROWN')
brown is not a number.
>>>
```

在match语句中实现赋值的第二种方法是使用“as模式（as pattern）”，其语法为：

```
pattern as identifier
```

该模式的含义是：如果主题匹配pattern，就可以匹配该as模式，同时将主题匹配的部分赋值给标识符identifier。

as模式的一个常见用法是与或模式联合使用，使得标识符最终引用主题本身。请看下面的例子：

```
def identify_vowel(letter):
    match letter.lower():
        case 'a' | 'e' | 'i' | 'o' | 'u' as l:
            print(str(l) + " is a vowel.")
        case _ as l:
            print(str(l) + " is a consonant.")
```

请将上述代码保存为match15.py，然后通过如下命令行和语句验证：

```
$ python3 -i match15.py

>>> identify_vowel('l')
l is a consonant.
>>> identify_vowel('A')
a is a vowel.
>>> identify_vowel('C')
c is a consonant.
```

```
>>> identify_vowel('e')
e is a vowel.
>>>
```

注意上面例子中的as模式“_ as l”其实可以用捕获模式“l”来代替。

as模式也可用于将主题匹配子模式的部分赋值给标识符，但此时as模式通常被应用于序列模式或映射模式中，这会在下面讨论。

13-7. 按序列匹配

（教程：4.6）
（语言参考手册：8.6.4.7、8.6.4.8）

很多情况下，我们需要一次匹配多个主题，那么最有效的方法是让这些主题成为一个序列或一个映射的元素，再将该序列或映射整体作为一个主题进行匹配。然而由于属于序列或映射的类型都有多种，使用类模式达到该目的就变得很麻烦（因为需要考虑到所有可能的类）。更好的办法是使用“序列模式（sequence patterns）”来匹配序列主题，使用“映射模式（mapping patterns）”来匹配映射主题。本节讨论序列模式，下节讨论映射模式。

序列模式即用圆括号或方括号括起来的以逗号分隔的模式序列，即：

```
(pattern1, pattern2, ..., patternN)
```

或者

```
[pattern1, pattern2, ..., patternN]
```

这两种格式几乎没有区别，但如果使用圆括号，则即便括号内只有一个子模式也必须添加逗号，否则会退化为“组模式（group pattern）”。换句话说，“(P,)”、“[P,]”和“[P]”都是序列模式，而“(P)”是组模式。组模式“(P)”和模式“P”完全等价。

序列模式中至多包含一个带星号的子模式，该子模式的格式为：

```
*identifier
```

或者

```
*_
```

这种子模式可以出现在序列模式中的任意位置，并不需要是最后一个子模式。

如果序列模式中没有带星号的子模式，那么它包含了多少个子模式，能够匹配的序列主题就必须有多少个元素。这属于固定长度的序列模式。而当序列模式中包含带星号的子模式时，假设它的子模式数为n，则能够匹配的序列主题中的元素数量可以是n-1~+∞，因为带星号的子模式可以匹配任意个（包括0个）元素。

具体来说，在匹配序列模式与序列主题时，会按照如下算法尝试匹配：

➤ 步骤一. 从序列主题开头向后取出元素，依次与序列模式中带星号的子模式之前的子模式配对，直到遇到带星号的子模式。
➤ 步骤二. 从序列末尾向前取出元素，依次与序列模式中带星号的子模式之后的子模式配对，直到遇到带星号的子模式。
➤ 步骤三. 将剩下的元素重新打包成一个子序列主题，与带星号的子模式配对。

这一算法中，如果序列主题中的元素不够，则直接匹配失败。否则，仅当所有子模式都匹配时才匹配成功。注意带星号的子模式总是能匹配子序列主题，如果它包含一个标识符（捕获模式），则还会将子序列主题赋值给该标识符。

最后需要强调，序列主题的类型必须是str、bytes和bytearray之外的序列，因为字符串、字节串和字节数组作为主题时是被视为一个整体的。

下面是一个序列模式的简单例子：

```
    from Point2DFixed import Point2DFixed


    class Point(Point2DFixed):
        __match_args__ = ('x', 'y')


    def identify_shape(points):
        match points:
            case []:
                print("No points.")
            case [Point()]:
                print("A single point.")
            case [Point(), Point()]:
                print("A line segment.")
            case [Point(), Point(), Point()]:
                print("A triangle.")
            case [Point(), Point(), *_, Point(), Point()]:
                print("A polygon.")
```

请将上述代码保存为match16.py，然后通过如下命令行和语句验证：

```
$ python3 -i match16.py

>>> p1 = Point(0, 0)
>>> p2 = Point(0, 1)
>>> p3 = Point(-1, 0)
>>> p4 = Point(-1, 1)
>>> p5 = Point(0.8, 0.6)
>>> identify_shape(())
No points.
>>> identify_shape((p1,))
A single point.
>>> identify_shape([p1])
A single point.
>>> identify_shape((p1, p2))
A line segment.
>>> identify_shape((p1, p2, p3))
A triangle.
>>> identify_shape([p1, p2, p3, p4])
A polygon.
>>> identify_shape((p1, p2, p3, p4, p5))
A polygon.
>>>
```

上一节已经提到了as模式可以被应用于序列模式。下面用一个例子来说明：

```
from Point2DFixed import Point2DFixed


class Point(Point2DFixed):
    __match_args__ = ('x', 'y')


def identify_shape(points):
    match points:
        case []:
            print("No points.")
        case [Point() as p1]:
            print("A single point at ("
                    + str(p1.x) + ", "
                    + str(p1.y) + ").")
        case [Point() as p1, Point() as p2]:
            print("A line segment from ("
                    + str(p1.x) + ", "
                    + str(p1.y) + ") to ("
                    + str(p2.x) + ", "
                    + str(p2.y) + ").")
        case [Point() as p1, *middle, Point() as p2]:
            print("A polygonal line segment from ("
                    + str(p1.x) + ", "
                    + str(p1.y) + ") to ("
                    + str(p2.x) + ", "
                    + str(p2.y) + ") with "
                    + str(len(middle)) + " middle points.")
```

请将上述代码保存为match17.py，然后通过如下命令行和语句验证：

```
$ python3 -i match17.py

>>> p1 = Point(0, 0)
>>> p2 = Point(0, 1)
>>> p3 = Point(-1, 0)
>>> p4 = Point(-1, 1)
>>> p5 = Point(0.8, 0.6)
>>> identify_shape([p5])
A single point at (0.8, 0.6).
>>> identify_shape([p2, p4])
A line segment from (0, 1) to (-1, 1).
>>> identify_shape([p2, p1, p5, p4])
A polygonal line segment from (0, 1) to (-1, 1) with 2 middle points.
>>> identify_shape([p1, p3, p5])
A polygonal line segment from (0, 0) to (0.8, 0.6) with 1 middle points.
>>>
```

该例子说明了将as模式应用于序列模式的目的主要是为了提取主题中匹配子模式的部分。此外该例子还演示了通过带星号的子模式完成赋值的技巧。

下面则是一个将捕获模式应用于序列模式的例子：

```
import math
from Point2DFixed import Point2DFixed


class Point(Point2DFixed):
    __match_args__ = ('x', 'y')


def get_distance(p1=None, p2=None):
    match p1, p2:
        case (None, None):
            print("No points.")
        case (Point(x1, y1), None):
            print("A single point, the distance from it to origin is "
                    + str(math.sqrt(x1**2 + y1**2)) + ".")
        case (Point(x1, y1), Point(x2, y2)):
            print("The distance between the two points is "
                    + str(math.sqrt((x1 - x2)**2 + (y1 - y2)**2)) + ".")
```

请将上述代码保存为match18.py，然后通过如下命令行和语句验证：

```
$ python3 -i match18.py

>>> p1 = Point(0, 0)
>>> p2 = Point(0, 1)
>>> p3 = Point(-1, 0)
>>> p4 = Point(-1, 1)
>>> p5 = Point(0.8, 0.6)
>>> get_distance()
No points.
>>> get_distance(p1)
A single point, the distance from it to origin is 0.0.
>>> get_distance(p5)
```

```
    A single point, the distance from it to origin is 1.0.
    >>> get_distance(p2, p3)
    The distance between the two points is 1.4142135623730951.
    >>> get_distance(p2, p4)
    The distance between the two points is 1.0.
    >>>
```

注意该例子还额外说明了match语句中的主题可以是逗号分隔的表达式列表，它只能匹配序列模式。

13-8. 按映射匹配

（语言参考手册：8.6.4.9）

映射模式即用大括号括起来的以逗号分隔的键值对，键与映射类型中的规则相同，值则是子模式，即：

```
    {key1: pattern1, key2: pattern2, ..., keyN: patternN)}
```

映射模式可以至多包含一个带双星号的子模式，其语法为：

```
    **identifier
```

注意与序列模式中的带星号的子模式不同，映射模式中带双星号的子模式必须位于所有键值对之后，自身并不是键值对。

同样，当映射模式不包含带双星号的子模式时，属于固定长度的映射模式，这意味着映射主题的键必须与映射模式的键一一对应。而当映射模式包含带双星号的子模式时，映射主题必须具有映射模式中的所有键，但还可以具有其他的键，后者会被重新打包成一个子映射主题与带双星号的子模式配对。需要强调的是，如果主题具有__missing__魔术属性，就能够对任意键返回一个值，但这不意味着它具有任意键。只有能够通过其get()属性访问的键才是主题具有的键。

在确定了映射主题与映射模式之间键的对应关系后，接下来就是按照键检查映射主题中的值是否匹配映射模式中的子模式。显然，仅当所有子模式都匹配时该匹配才成功。带双星号的子模式总是匹配成功，且它总是会同时完成赋值（双星号之后不能跟通配符模式）。

映射模式中不能出现重复的键。如果两个键相等，则会导致在运行前抛出SyntaxError异常；如果两个键不相等但具有相同的哈希值，则会导致在运行时抛出ValueError异常。

下面是一个映射模式的例子：

```
    def identify_race(person):
        match person:
            case {
                    'skin': 'white' | 'brown',
                    'hair': 'gray' | 'blonde' | 'yellow' | 'red' | 'brown' |
'black',
                    'pupil': 'gray' | 'blue' | 'green' | 'red' | 'gold',
                    **info}:
                print(info['name'] + ', '
                        + str(info['age']) + ' years old, '
                        + 'belongs to the white race.')
            case {
                    'skin': 'yellow' | 'white',
                    'hair': 'black' | 'brown' | 'red',
                    'pupil': 'black' | 'brown',
                    **info}:
                print(info['name'] + ', '
                       + str(info['age']) + ' years old, '
                       + 'belongs to the yellow race.')
            case {
                    'skin': 'black' | 'brown',
                    'hair': 'black',
                    'pupil': 'black' | 'brown',
                    **info}:
                print(info['name'] + ', '
                       + str(info['age']) + ' years old, '
                       + 'belongs to the black race.')
```

请将上述代码保存为match19.py，然后通过如下命令行和语句验证：

```
    $ python3 -i match19.py

    >>> p1 = {'name': 'Mark Roy', 'age': 17, 'skin': 'brown', 'hair': 'yellow',
'pupil': 'blue'}
    >>> identify_race(p1)
    Mark Roy, 17 years old, belongs to the white race.
    >>> p2 = {'name': 'Min Li', 'age': 23, 'skin': 'yellow', 'hair': 'black',
'pupil': 'brown'}
    >>> identify_race(p2)
    Min Li, 23 years old, belongs to the yellow race.
    >>> p3 = {'name': 'Antoine Noa', 'age': 21, 'skin': 'brown', 'hair':
'black', 'pupil': 'black'}
    >>> identify_race(p3)
    Antoine Noa, 21 years old, belongs to the black race.
    >>>
```

显然，捕获模式和as模式也能应用于映射模式中，与应用于序列模式中的方式没有区别。