流程控制

IlleniumDillon

2024 年 6 月 3 日

1 if 语句

可有零个或多个 elif 部分，以及一个可选的 else 部分。关键字 'elif' 是 'else if' 的缩写，用于避免过多的缩进。一个 if ··· elif ··· elif ··· 序列可以看作其他语言中的 switch 或 case 语句。如果是把一个值与多个常量比较，或者检查特定类型或属性，match 语句更有用。

```
x = int(input("Please enter an integer: "))
if x < 0:
    x = 0
    print('Negative changed to zero')
elif x == 0:
    print('Zero')
elif x == 1:
    print('Single')
else:
    print('More')
```

2 match 语句

match 语句是一种更通用的形式，可以替代一系列 if ··· elif ··· elif ··· 语句。它比一系列 if ··· elif ··· elif ··· 语句更简洁，更易读。它接受一个表

达式，然后按顺序检查每个模式，并执行与第一个匹配的模式相关联的代码块。最简单形式是将一个主语值与一个或多个字面值进行比较：

```
def http_error(status):
    match status:
        case 400:
            return "Bad request"
        case 404:
            return "Not found"
        case 418:
            return "I'm a teapot"
        case _:
            return "Something else"
```

注意最后一个 case 语句，它使用了一个通配符模式，即一个下划线。这个模式匹配任何值，这使得 match 语句总是有一个分支可以执行。如果没有 case 匹配成功，则不会执行任何分支。

可以使用 — 运算符将多个模式组合在一起，以便在一个分支中匹配多个模式：

```
case 401 | 403 | 404:
    return "Not allowed"
```

形如解包赋值的模式可被用于绑定变量：

第一个模式有两个字面值，可视为前述字面值模式的扩展。接下来的两个模式结合了一个字面值和一个变量，变量绑定了来自主语（point）的一个值。第四个模式捕获了两个值，使其在概念上与解包赋值 (x, y) = point 类似。

如果使用类来组织数据，可以使用模式匹配来检查类的属性：

可以通过在类定义中添加 __match_args__ 属性来指定模式匹配的参数。这个属性是一个元组，包含了类的参数名称。在这个例子中，Point 类有两个参数， x 和 y。这个属性告诉 match 语句如何解构 Point 实例。

```python
# point is an (x, y) tuple
match point:
    case (0, 0):
        print("Origin")
    case (0, y):
        print(f"Y={y}")
    case (x, 0):
        print(f"X={x}")
    case (x, y):
        print(f"X={x}, Y={y}")
    case _:
        raise ValueError("Not a point")
```

可以向模式添加条件，以便在模式匹配成功时执行更复杂的检查，如果约束项为假值，则 match 将继续尝试下一个 case 语句块。请注意值的捕获发生在约束项被求值之前。:

match语句还有一些其他特性：

与解包赋值类似，元组和列表模式具有完全相同的含义并且实际上都能匹配任意序列，区别是它们不能匹配迭代器或字符串。

序列模式支持扩展解包：[x, y, *rest] 和 (x, y, *rest) 和相应的解包赋值做的事是一样的。接在 * 后的名称也可以为 _，所以 (x, y, *_) 匹配含至少两项的序列，而不必绑定剩余的项。

映射模式：”bandwidth”: b, ”latency”: l 从字典中捕获 ”bandwidth” 和 ”latency” 的值。额外的键会被忽略，这一点与序列模式不同。**rest 这样的解包也支持。但 **_ 将会是冗余的，故不允许使用。

as 模式：x as name 会将 x 匹配的值绑定到 name 上，这样可以在模式中使用它。这对于在模式中使用值的多个副本很有用，或者在模式中使用一个值的属性。

模式可以使用具名常量。它们必须作为带点号的名称出现，以防止它们被解释为用于捕获的变量。

```
class Point:
    __match_args__ = ('x', 'y')
    def __init__(self, x, y):
        self.x = x
        self.y = y

match points:
    case []:
        print("No points")
    case [Point(0, 0)]:
        print("The origin")
    case [Point(x, y)]:
        print(f"Single point {x}, {y}")
    case [Point(0, y1), Point(0, y2)]:
        print(f"Two on the Y axis at {y1}, {y2}")
    case _:
        print("Something else")
```

```
match point:
    case Point(x, y) if x == y:
        print(f"Diagonal point {x}, {y}")
    case Point(x, y):
        print(f"Ordinary point {x}, {y}")
```

```
case Point(0, 0) as origin:
    print("The origin is at", origin)
...
case Point(0, 0) as Point.ORIGIN:
    print("The origin")
```

3 while 语句

while 语句用于在表达式保持为真的情况下重复地执行, 这将重复地检验表达式，并且如果其值为真就执行第一个子句体；如果表达式值为假（这可能在第一次检验时就发生）则如果 else 子句体存在就会被执行并终止循环。第一个子句体中的 break 语句在执行时将终止循环且不执行 else 子句体。第一个子句体中的 continue 语句在执行时将跳过子句体中的剩余部分并返回检验表达式。

4 for 语句

for 语句用于迭代一个序列（例如列表、元组或字符串）或其他可迭代对象。 for 循环的语法如下：

```
for target_list in starred_list:
    statements(s)
else:
    final_statement(s)
```

starred_list 表达式会被求值一次；它应当产生一个 iterable 对象。将针对该可迭代对象创建一个 iterator。随后该迭代器所提供的第一个条目将使用标准的赋值规则被赋值给目标列表，而代码块将被执行。此过程将针对该迭代器所提供每个条目重复进行。当迭代器被耗尽时，如果存在 else 子句中的代码块，则它将被执行，并终结循环。

第一个子句体中的 break 语句在执行时将终止循环且不执行 else 子句体。第一个子句体中的 continue 语句在执行时将跳过子句体中的剩余部分并转往下一项继续执行，或者在没有下一项时转往 else 子句执行。

for 循环会对目标列表中的变量进行赋值。这将覆盖之前对这些变量的所有赋值，包括在 for 循环体中的赋值。目标列表中的名称在循环结束时不会被删除，但是如果序列为空，则它们将根本不会被循环所赋值。

5 range() 语句

内置函数 range() 用于生成等差数列，生成的序列绝不会包括给定的终止值； range(10) 生成 10 个值——长度为 10 的序列的所有合法索引。range 可以不从 0 开始，且可以按给定的步长递增（即使是负数步长）。

class range(start, stop[, step]) range 构造器的参数必须为整数（可以是内置的 int 或任何实现了 __index__() 特殊方法的对象）。如果省略 step 参数，则默认为 1。如果省略 start 参数，则默认为 0。

如果 step 为正值，确定 range r 内容的公式为

$$r[i] = start + step * i \ when \ i \geq 0 \ and \ r[i] < stop$$

如果 step 为负值，确定 range r 内容的公式为:

$$r[i] = start + step * i \ when \ i \geq 0 \ and \ r[i] > stop$$

6 pass 语句

pass 语句什么也不做。它用于那些语法上需要语句但程序不需要执行任何操作的场合。

```
while True:
    pass  # Busy-wait for keyboard interrupt (Ctrl+C)
```