Switch statement

October 24, 2025

Python switch statement

Dr. Labeed Al-Saad

Introduction The switch statement is a powerful control flow structure that allows a variable to be tested for equality against multiple cases. While Python didn't originally have a traditional switch statement, modern versions offer elegant alternatives that serve the same purpose.

What is a Switch Statement? A switch statement is a control flow structure that evaluates an expression and matches it to one of several cases, executing the corresponding block of code. It's essentially a cleaner alternative to long chains of if-elif-else statements.

Python's Approach to Switch Python introduced the match-case statement in version 3.10, which provides switch-like functionality with even more power than traditional switch statements.

Method 1: Using Match-Case (Python 3.10+) Basic Syntax

[ ]:
```
match expression:
    case pattern1:
        # code for pattern1
    case pattern2:
        # code for pattern2
    case _:
        # default case
```

Practical Examples Example 1: Basic Day of Week

[3]:
```
def GetDayType(day):
    match day.lower(): # day.lower() is a methid used to convert "day name" to␣
 ↪lower case status
        case "monday" | "tuesday" | "wednesday" | "thursday" | "friday":
            return "weekday"
        case "saturday" | "sunday":
            return "weekend"
        case _:
            return "invalid day"
# Uasage
print(GetDayType("Monday"))
print(GetDayType("SUNDAY"))
print(GetDayType("Funday"))
```

```
weekday
weekend
invalid day
```

Example 2: HTTP Status Codes

[8]:
```python
def handle_http_status(code):
    match code:
        case 200:
            return "OK seccess"
        case 301 | 302:
            return "Redirect"
        case 404:
            return "Not found"
        case _: return f"unknown status: {code}"
# Usage
print(handle_http_status(200))
print(handle_http_status(404))
print(handle_http_status(999))
```

```
OK seccess
Not found
unknown status: 999
```

Example 3: Advanced Pattern Matching How can I write a single, clean function that handles completely different types of data structures (like lists, dictionaries, and strings) and performs different actions based on both the type of data and its internal structure or values?

[12]:
```python
def process_data(data):
    match data:
        case [x, y, z]:
            return f"List with three elements: {x}, {y}, {z}"
        case {"name": name, "age": age}:
            return f"this pesron called: {name}, and his/her age is {age} years␣
 ↪old"
        case str() as text if len(text) > 10:
            return f"Long string:{text[:10]}..."
        case str() as text:
            return f"short string: {text}"
        case _: "unknown data type"
# usage
print(process_data([1, 2, 3]))
print(process_data({"name": "Alice", "age": 30}))
print(process_data("Hello world"))
```

```
List with three elements: 1, 2, 3
this pesron called: Alice, and his/her age is 30 years old
Long string:Hello worl…
```

Method 2: Using Dictionary Mapping (All Python Versions)

Basic Approach

[ ]:
```
def switch_dict(value):
    return {
        'case1': lambda: "Result for case1",
        'case2': lambda: "Result for case2",
        'case3': lambda: "Result for case3"
    }.get(value, lambda: "Default case")()
```

.get(value, lambda: "Default case")- This uses the dictionary'sget()method to look up the key specified by thevalue‘ parameter. If the key exists in the dictionary, it returns the corresponding value (which is a lambda function). If the key doesn't exist, it returns the default value provided as the second argument (another lambda function that returns "Default case").

Practical Examples

Example 1: Calculator

[14]:
```
def calculator(operator, a, b):
    operations = {
        '+': lambda: a + b,
        '-': lambda: a - b,
        '*': lambda: a * b,
        '/': lambda: a / b if b != 0 else "Error: Division by zero"
    }
    return operations.get(operator, lambda: "invalid operator")() #The empty␣
 ↪parentheses `()` at the end of the expression
                                                                  #`operations.
 ↪get(operator, lambda: "invalid operator")()`
                                                                  # are used to␣
 ↪immediately invoke/call the lambda function
                                                                  # that was␣
 ↪retrieved from the dictionary

# Usage
print(calculator('+', 5, 3))
print(calculator('*', 10, 3))
print(calculator('/', 5, 0))
```

```
8
30
Error: Division by zero
```

Example 2: Grade Converter

[19]:
```
def get_grade_description(grade):
    grade_descriptions = {
        'A': "Excellent",
        'B': "Good",
        'C': "Average",
```

```
        'D': "Below Average",
        'F': "Fail"
    }
    return grade_descriptions.get(grade.upper(), "Invalid grade")

# Usage
print(get_grade_description('A'))  # Output: Excellent
print(get_grade_description('C'))  # Output: Average
print(get_grade_description('X'))
```

```
Excellent
Average
Invalid grade
```

Method 3: Using Classes for Complex Logic

[20]:
```
class Switch:
    def __init__(self, value):
        self.value = value
        self.matched = False

    def case(self, pattern, func):
        if not self.matched and (pattern == self.value or callable(pattern) and␣
 ↪pattern(self.value)):
            func()
            self.matched = True
        return self

    def default(self, func):
        if not self.matched:
            func()
        return self

# Usage
def process_number(num):
    Switch(num)\
        .case(1, lambda: print("One"))\
        .case(lambda x: 2 <= x <= 5, lambda: print("Between 2 and 5"))\
        .case(10, lambda: print("Ten"))\
        .default(lambda: print("Other number"))

process_number(3)
process_number(10)
process_number(7)
```

```
Between 2 and 5
Ten
Other number
```

Summary

Method: Match-Case; Python Version: 3.10+; Use Case: Complex pattern matching; Pros: Clean, powerful, readable; Cons: Newer Python only

Method: Dictionary Mapping; Python Version: All versions; Use Case: Simple value mapping; Pros: Fast, compatible; Cons: Limited omplexity

Method: Class-based; Python Version: All versions; Use Case: Complex conditional logic; Pros: Flexible, reusable; Cons: More verbose