Learn Python Programming from Scratch

Topic: Match-Case Statements in Python

1. What are Match-Case Statements?

Match-case statements, introduced in Python 3.10, provide a powerful pattern matching feature similar to switch statements in other languages. They offer a more elegant and readable way to handle multiple conditional checks compared to long if-elif chains. Think of them as intelligent decision trees that can match values, patterns, and even complex data structures.

2. Basic Match-Case Syntax

The basic syntax uses `match` followed by an expression, and `case` for each pattern to match against.

In [1]:

```python
# Basic match-case example
def process_grade(letter):
    match letter:
        case 'A':
            return "Excellent! 90-100%"
        case 'B':
            return "Good! 80-89%"
        case 'C':
            return "Average! 70-79%"
        case 'D':
            return "Below Average! 60-69%"
        case 'F':
            return "Failing! Below 60%"
        case _:  # Default case (wildcard)
            return "Invalid grade"

# Test the function
print(f"Grade 'A' : {process_grade('A')}")  # Output: Excellent! 90-100%
print(f"Grade 'X' : {process_grade('X')}")  # Output: Invalid grade
```

```
Grade 'A' : Excellent! 90-100%
Grade 'X' : Invalid grade
```

3. Pattern Matching with Multiple Values

Match-case can handle multiple values in a single case and supports various patterns.

In [2]:

```python
# Multiple values and guard conditions
def classify_number(num):
    match num:
        case 0:
            return "Zero"
        case 1 | 2 | 3:  # Multiple values with OR operator
            return "Small positive"
        case n if n < 0:  # Guard condition
            return f"Negative number: {n}"
        case n if n > 100:
            return f"Large number: {n}"
        case n if 4 <= n <= 10:
            return f"Medium number: {n}"
        case _:
            return f"Other positive number: {num}"

# Test with different numbers
numbers = [0, 2, -5, 150, 7, 25]
for num in numbers:
    print(f"{num}: {classify_number(num)}")
```

```
0: Zero
2: Small positive
-5: Negative number: -5
150: Large number: 150
7: Medium number: 7
25: Other positive number: 25
```

4. Matching Data Structures

Match-case excels at pattern matching with lists, tuples, and dictionaries.

In [3]:

```
# Pattern matching with data structures
def analyze_coordinates(point):
    match point:
        case (0, 0):
            return "Origin point"
        case (0, y):
            return f"On Y-axis at {y}"
        case (x, 0):
            return f"On X-axis at {x}"
        case (x, y) if x == y:
            return f"On diagonal at ({x}, {y})"
        case (x, y) if x > 0 and y > 0:
            return f"First quadrant: ({x}, {y})"
        case (x, y) if x < 0 and y > 0:
            return f"Second quadrant: ({x}, {y})"
        case (x, y) if x < 0 and y < 0:
            return f"Third quadrant: ({x}, {y})"
        case (x, y) if x > 0 and y < 0:
            return f"Fourth quadrant: ({x}, {y})"
        case _:
            return "Invalid coordinates"

# Test coordinate analysis
coordinates = [(0, 0), (0, 5), (3, 0), (2, 2), (4, -3), (-2, 5)]
for coord in coordinates:
    print(f"{coord}: {analyze_coordinates(coord)}")
```

```
(0, 0): Origin point
(0, 5): On Y-axis at 5
(3, 0): On X-axis at 3
(2, 2): On diagonal at (2, 2)
(4, -3): Fourth quadrant: (4, -3)
(-2, 5): Second quadrant: (-2, 5)
```

5. Object and Class Pattern Matching

Match-case can match against object types and extract attributes.

In [4]:

```
# Object pattern matching with classes
class Student:
    def __init__(self, name, age, grade):
        self.name = name
        self.age = age
        self.grade = grade

class Teacher:
    def __init__(self, name, subject, experience):
        self.name = name
        self.subject = subject
        self.experience = experience

def process_person(person):
    match person:
        case Student(name=name, age=age, grade=grade) if age < 13:
            return f"Elementary student {name} in grade {grade}"
        case Student(name=name, age=age, grade=grade) if 13 <= age <= 18:
            return f"High school student {name} (age {age}) in grade {grade}"
        case Student(name=name, age=age, grade=grade):
            return f"College student {name} (age {age}) in grade {grade}"
        case Teacher(name=name, subject=subject, experience=exp) if exp < 5:
            return f"New teacher {name} teaches {subject} ({exp} years exp)"
        case Teacher(name=name, subject=subject, experience=exp):
            return f"Experienced teacher {name} teaches {subject} ({exp} years e
        case _:
            return "Unknown person type"

# Test with different person objects
people = [
    Student("Alice", 12, 7),
    Student("Bob", 16, 11),
    Student("Charlie", 20, "Sophomore"),
    Teacher("Dr. Smith", "Mathematics", 3),
    Teacher("Prof. Johnson", "Physics", 15)
]

for person in people:
    print(process_person(person))
```

```
Elementary student Alice in grade 7
High school student Bob (age 16) in grade 11
College student Charlie (age 20) in grade Sophomore
New teacher Dr. Smith teaches Mathematics (3 years exp)
Experienced teacher Prof. Johnson teaches Physics (15 years exp)
```

6. Advanced Pattern Matching Features

Explore advanced features like capturing subpatterns and complex matching scenarios.

In [5]:

```
# Advanced pattern matching with lists and nested structures
def analyze_data_structure(data):
    match data:
        case []:  # Empty list
            return "Empty list"
        case [x]:  # Single element
            return f"Single element: {x}"
        case [x, y]:  # Exactly two elements
            return f"Pair: {x}, {y}"
        case [x, *rest]:  # First element and rest
            return f"First: {x}, Rest: {rest} (length: {len(rest)})"
        case {"type": "user", "name": str(name), "age": int(age)}:
            return f"User: {name}, Age: {age}"
        case {"type": "product", "name": str(name), "price": float(price)}:
            return f"Product: {name}, Price: ${price}"
        case str(s) if len(s) > 10:
            return f"Long string: {s[:10]}..."
        case int(n) if n > 1000:
            return f"Large integer: {n}"
        case _:
            return f"Other data type: {type(data).__name__}"

# Test with various data structures
test_data = [
    [],
    [42],
    [1, 2],
    [1, 2, 3, 4, 5],
    {"type": "user", "name": "Alice", "age": 25},
    {"type": "product", "name": "Laptop", "price": 999.99},
    "This is a very long string for testing",
    5000,
    3.14159
]

for data in test_data:
    print(f"{str(data)[:30]}: {analyze_data_structure(data)}")
```

```
[]: Empty list
[42]: Single element: 42
[1, 2]: Pair: 1, 2
[1, 2, 3, 4, 5]: First: 1, Rest: [2, 3, 4, 5] (length: 4)
{'type': 'user', 'name': 'Alic: User: Alice, Age: 25
{'type': 'product', 'name': 'L: Product: Laptop, Price: $999.99
This is a very long string for: Long string: This is a ...
5000: Large integer: 5000
3.14159: Other data type: float
```

Exercises

1. Create a match-case statement for HTTP status codes (200, 404, 500, etc.).
2. Build a calculator using match-case for different operations (+, -, *, /).
3. Write a pattern matcher for different geometric shapes with their properties.
4. Create a menu system using match-case for user choices.
5. Implement a data validator using match-case for different input types.

Practical Examples

Let's explore practical applications of match-case statements in real-world scenarios. These examples show how pattern matching can simplify complex decision-making logic.

Restaurant Order Processing System

Here's a comprehensive restaurant system that uses match-case for order processing, pricing, and customer management.

In [7]:

```
# Advanced restaurant order processing system using match-case

print("Restaurant Order Processing System")
print("=" * 35)

# Menu data structure
menu_items = {
    "appetizers": {"spring_rolls": 8.99, "nachos": 12.99, "wings": 14.99},
    "mains": {"burger": 16.99, "pasta": 18.99, "steak": 24.99, "salmon": 22.99},
    "desserts": {"cake": 6.99, "ice_cream": 4.99, "pie": 7.99},
    "beverages": {"soda": 2.99, "juice": 3.99, "coffee": 2.49, "wine": 8.99}
}

# Customer types for pricing
customer_types = {
    "regular": 1.0,      # No discount
    "student": 0.90,     # 10% discount
    "senior": 0.85,      # 15% discount
    "employee": 0.75,    # 25% discount
    "vip": 0.80         # 20% discount
}

def process_order_item(item_info):
    """Process individual order items using match-case"""
    match item_info:
        case {"category": "appetizers", "item": item_name, "quantity": qty}:
            if item_name in menu_items["appetizers"]:
                price = menu_items["appetizers"][item_name]
                return {
                    "item": item_name.replace("_", " ").title(),
                    "category": "Appetizer",
                    "price": price,
                    "quantity": qty,
                    "total": price * qty,
                    "prep_time": 10  # minutes
                }
            return {"error": f"Appetizer {item_name} not available"}

        case {"category": "mains", "item": item_name, "quantity": qty}:
            if item_name in menu_items["mains"]:
                price = menu_items["mains"][item_name]
                # Main dishes have longer prep time
                prep_time = 25 if item_name == "steak" else 20
                return {
                    "item": item_name.replace("_", " ").title(),
                    "category": "Main Course",
                    "price": price,
                    "quantity": qty,
                    "total": price * qty,
                    "prep_time": prep_time
                }
            return {"error": f"Main dish {item_name} not available"}

        case {"category": "desserts", "item": item_name, "quantity": qty}:
            if item_name in menu_items["desserts"]:
                price = menu_items["desserts"][item_name]
                return {
                    "item": item_name.replace("_", " ").title(),
                    "category": "Dessert",
                    "price": price,
                    "quantity": qty,
                    "total": price * qty,
                    "prep_time": 8
                }
            return {"error": f"Dessert {item_name} not available"}

        case {"category": "beverages", "item": item_name, "quantity": qty}:
            if item_name in menu_items["beverages"]:
                price = menu_items["beverages"][item_name]
                # Beverages are quick to prepare
                prep_time = 2 if item_name in ["soda", "juice"] else 5
                return {
                    "item": item_name.replace("_", " ").title(),
                    "category": "Beverage",
                    "price": price,
                    "quantity": qty,
                    "total": price * qty,
                    "prep_time": prep_time
                }
            return {"error": f"Beverage {item_name} not available"}

        case _:
            return {"error": "Invalid item format"}

def calculate_final_price(subtotal, customer_type, order_size):
    """Calculate final price with discounts using match-case"""
    match customer_type, order_size:
        case "student", size if size >= 3:
            discount = 0.85  # Extra discount for students with large orders
            discount_desc = "Student + Large Order"
        case "senior", size if size >= 2:
            discount = 0.80  # Extra discount for seniors
            discount_desc = "Senior + Multi-item"
        case "employee", _:
            discount = 0.75  # Employee discount always applies
            discount_desc = "Employee"
        case "vip", size if size >= 5:
            discount = 0.70  # VIP with very large order
            discount_desc = "VIP + Large Order"
        case customer_type, _ if customer_type in customer_types:
            discount = customer_types[customer_type]
            discount_desc = customer_type.title()
        case _:
```

```python
            discount = 1.0
            discount_desc = "Regular Price"

    # Apply quantity discount for very large orders
    if order_size >= 10:
        discount *= 0.95  # Additional 5% off for bulk orders
        discount_desc += " + Bulk Discount"

    final_price = subtotal * discount
    savings = subtotal - final_price

    return final_price, savings, discount_desc

# Sample orders for different customers
sample_orders = [
    {
        "customer": "Alice (Student)",
        "customer_type": "student",
        "items": [
            {"category": "mains", "item": "burger", "quantity": 1},
            {"category": "beverages", "item": "soda", "quantity": 2},
            {"category": "desserts", "item": "ice_cream", "quantity": 1}
        ]
    },
    {
        "customer": "Bob (Senior)",
        "customer_type": "senior",
        "items": [
            {"category": "appetizers", "item": "wings", "quantity": 1},
            {"category": "mains", "item": "salmon", "quantity": 1},
            {"category": "beverages", "item": "wine", "quantity": 1}
        ]
    },
    {
        "customer": "Carol (VIP)",
        "customer_type": "vip",
        "items": [
            {"category": "appetizers", "item": "spring_rolls", "quantity": 2},
            {"category": "mains", "item": "steak", "quantity": 2},
            {"category": "mains", "item": "pasta", "quantity": 1},
            {"category": "desserts", "item": "cake", "quantity": 2},
            {"category": "beverages", "item": "wine", "quantity": 3}
        ]
    }
]

# Process each order
for order in sample_orders:
    customer_name = order["customer"]
    customer_type = order["customer_type"]
    items = order["items"]

    print(f"\nProcessing order for {customer_name}")
    print("-" * (20 + len(customer_name)))

    order_details = []
    subtotal = 0
    total_prep_time = 0
    order_size = 0
    # Process each item in the order
    for item_info in items:
        result = process_order_item(item_info)

        if "error" in result:
            print(f"❌ Error: {result['error']}")
            continue

        order_details.append(result)
        subtotal += result["total"]
        total_prep_time = max(total_prep_time, result["prep_time"])  # Parallel
        order_size += result["quantity"]

        print(f"{result['quantity']}x {result['item']} ({result['category']})")
        print(f"   ${result['price']:.2f} each → ${result['total']:.2f}")

    if order_details:
        # Calculate final pricing
        final_price, savings, discount_desc = calculate_final_price(
            subtotal, customer_type, order_size
        )

        # Order summary
        print(f"\nOrder Summary:")
        print(f"   Items: {order_size}")
        print(f"   Subtotal: ${subtotal:.2f}")
        print(f"   Discount: {discount_desc}")
        if savings > 0:
            print(f"   Savings: ${savings:.2f}")
        print(f"   Final Total: ${final_price:.2f}")
        print(f"   Estimated prep time: {total_prep_time} minutes")
    else:
        print("No valid items in order")

print(f"\nAll orders processed successfully!")
```

```
Restaurant Order Processing System
===================================

Processing order for Alice (Student)
-----------------------------------
1x Burger (Main Course)
   $16.99 each → $16.99
2x Soda (Beverage)
   $2.99 each → $5.98
1x Ice Cream (Dessert)
   $4.99 each → $4.99

Order Summary:
   Items: 4
   Subtotal: $27.96
   Discount: Student + Large Order
   Savings: $4.19
   Final Total: $23.77
   Estimated prep time: 20 minutes

Processing order for Bob (Senior)
--------------------------------
1x Wings (Appetizer)
   $14.99 each → $14.99
1x Salmon (Main Course)
   $22.99 each → $22.99
1x Wine (Beverage)
   $8.99 each → $8.99

Order Summary:
   Items: 3
   Subtotal: $46.97
   Discount: Senior + Multi-item
   Savings: $9.39
   Final Total: $37.58
   Estimated prep time: 20 minutes

Processing order for Carol (VIP)
-------------------------------
2x Spring Rolls (Appetizer)
   $8.99 each → $17.98
2x Steak (Main Course)
   $24.99 each → $49.98
1x Pasta (Main Course)
   $18.99 each → $18.99
2x Cake (Dessert)
   $6.99 each → $13.98
3x Wine (Beverage)
   $8.99 each → $26.97

Order Summary:
   Items: 10
   Subtotal: $127.90
   Discount: VIP + Large Order + Bulk Discount
   Savings: $42.85
   Final Total: $85.05
   Estimated prep time: 25 minutes

All orders processed successfully!
```

Smart Device Control System

This example demonstrates using match-case for IoT device control and status management.

In [9]:

```
# Smart home device control system using advanced match-case patterns

print("Smart Device Control System")
print("=" * 28)

# Device state management
device_states = {
    "living_room_light": {"status": "on", "brightness": 75, "color": "warm_white
    "bedroom_ac": {"status": "on", "temperature": 22, "mode": "cool"},
    "kitchen_oven": {"status": "off", "temperature": 0, "timer": 0},
    "security_camera": {"status": "on", "recording": True, "motion_detection": T
    "smart_speaker": {"status": "on", "volume": 50, "playing": "jazz_playlist"}
}

def process_device_command(command):
    """Process device commands using sophisticated match-case patterns"""
    match command:
        # Light control commands
        case {"device": "light", "room": room, "action": "toggle"}:
            device_key = f"{room}_light"
            if device_key in device_states:
                current_status = device_states[device_key]["status"]
                new_status = "off" if current_status == "on" else "on"
                device_states[device_key]["status"] = new_status
                return f"{room.title()} light turned {new_status}"
            return f"❌ Light not found in {room}"

        case {"device": "light", "room": room, "action": "set_brightness", "valu
            device_key = f"{room}_light"
            if device_key in device_states:
                device_states[device_key]["brightness"] = brightness
                device_states[device_key]["status"] = "on" if brightness > 0 els
                return f"{room.title()} light brightness set to {brightness}%"
            return f"Light not found in {room}"

        case {"device": "light", "room": room, "action": "set_color", "value": c
            device_key = f"{room}_light"
            if device_key in device_states:
                valid_colors = ["warm_white", "cool_white", "red", "green", "blu
                if color in valid_colors:
                    device_states[device_key]["color"] = color
                    return f"{room.title()} light color changed to {color.replac
                return f"Invalid color. Available: {', '.join(valid_colors)}"
            return f"Light not found in {room}"

        # Air conditioning control
        case {"device": "ac", "room": room, "action": "set_temperature", "value'
            device_key = f"{room}_ac"
            if device_key in device_states:
                device_states[device_key]["temperature"] = temp
                device_states[device_key]["status"] = "on"
                return f"{room.title()} AC temperature set to {temp}°C"
            return f"❌ AC not found in {room}"

        case {"device": "ac", "room": room, "action": "set_mode", "value": mode
            device_key = f"{room}_ac"
            if device_key in device_states:
                valid_modes = ["cool", "heat", "auto", "fan"]
                if mode in valid_modes:
                    device_states[device_key]["mode"] = mode
                    return f"{room.title()} AC mode set to {mode}"
                return f"Invalid mode. Available: {', '.join(valid_modes)}"
            return f"AC not found in {room}"

        # Kitchen appliance control
        case {"device": "oven", "room": "kitchen", "action": "preheat", "value"
            device_states["kitchen_oven"]["status"] = "on"
            device_states["kitchen_oven"]["temperature"] = temp
            return f"Kitchen oven preheating to {temp}°C"

        case {"device": "oven", "room": "kitchen", "action": "set_timer", "value
            if device_states["kitchen_oven"]["status"] == "on":
                device_states["kitchen_oven"]["timer"] = minutes
                return f"Kitchen oven timer set for {minutes} minutes"
            return "Oven must be on to set timer"

        # Security system control
        case {"device": "camera", "room": room, "action": "toggle_recording"}:
            device_key = f"{room}_camera"
            if device_key in device_states:
                current_recording = device_states[device_key]["recording"]
                device_states[device_key]["recording"] = not current_recording
                status = "started" if not current_recording else "stopped"
                return f"{room.title()} camera recording {status}"
            return f"Camera not found in {room}"

        case {"device": "camera", "room": room, "action": "motion_detection", "
            device_key = f"{room}_camera"
            if device_key in device_states:
                device_states[device_key]["motion_detection"] = enabled
                status = "enabled" if enabled else "disabled"
                return f"{room.title()} camera motion detection {status}"
            return f"Camera not found in {room}"

        # Smart speaker control
        case {"device": "speaker", "room": room, "action": "set_volume", "value"
            device_key = f"{room}_speaker"
            if device_key in device_states:
                device_states[device_key]["volume"] = volume
                return f"{room.title()} speaker volume set to {volume}%"
            return f"Speaker not found in {room}"

        case {"device": "speaker", "room": room, "action": "play", "value": cont
            device_key = f"{room}_speaker"
            if device_key in device_states:
                device_states[device_key]["playing"] = content
                device_states[device_key]["status"] = "on"
                return f"{room.title()} speaker now playing: {content.replace('_
            return f"Speaker not found in {room}"
        # Device status queries
        case {"device": device_type, "room": room, "action": "status"}:
            device_key = f"{room}_{device_type}"
            if device_key in device_states:
                state = device_states[device_key]
                status_info = f"{room.title()} {device_type} status:\n"
                for key, value in state.items():
                    if key == "status":
                        emoji = "✅" if value == "on" else "❌"
                        status_info += f"   {emoji} Status: {value.title()}\n"
                    else:
                        status_info += f"   • {key.replace('_', ' ').title()}: {
                return status_info.strip()
            return f"{device_type.title()} not found in {room}"

        # Bulk operations
        case {"action": "all_lights", "command": "off"}:
            lights_turned_off = []
            for device_key, state in device_states.items():
                if "light" in device_key:
                    state["status"] = "off"
                    room_name = device_key.replace("_light", "").replace("_", "
                    lights_turned_off.append(room_name.title())
            return f"All lights turned off: {', '.join(lights_turned_off)}"

        case {"action": "energy_save_mode"}:
            changes = []
            for device_key, state in device_states.items():
                if "light" in device_key:
                    state["brightness"] = 30
                    changes.append(f"Light dimmed in {device_key.replace('_light
                elif "ac" in device_key:
                    state["temperature"] = 25  # Energy efficient temperature
                    changes.append(f"AC optimized in {device_key.replace('_ac',
            return f"Energy save mode activated:\n   • " + "\n   • ".join(change

        case _:
            return "Unknown command or invalid parameters"

# Test commands for the smart home system
test_commands = [
    {"device": "light", "room": "living_room", "action": "toggle"},
    {"device": "light", "room": "living_room", "action": "set_brightness", "valu
    {"device": "light", "room": "bedroom", "action": "set_color", "value": "blue
    {"device": "ac", "room": "bedroom", "action": "set_temperature", "value": 2
    {"device": "ac", "room": "bedroom", "action": "set_mode", "value": "cool"},
    {"device": "oven", "room": "kitchen", "action": "preheat", "value": 180},
    {"device": "oven", "room": "kitchen", "action": "set_timer", "value": 45},
    {"device": "camera", "room": "security", "action": "toggle_recording"},
    {"device": "speaker", "room": "smart", "action": "set_volume", "value": 75},
    {"device": "speaker", "room": "smart", "action": "play", "value": "classica
    {"device": "light", "room": "living_room", "action": "status"},
    {"action": "all_lights", "command": "off"},
    {"action": "energy_save_mode"}
]

print("Executing Smart Home Commands:")
print("-" * 30)
```

```python
for i, command in enumerate(test_commands, 1):
    print(f"\n{i}. Command: {command}")
    result = process_device_command(command)
    print(f"   Result: {result}")

# Final system status
print(f"\nFinal System Status:")
print("=" * 22)
for device_key, state in device_states.items():
    device_name = device_key.replace("_", " ").title()
    status = "Online" if state["status"] == "on" else "Offline"
    print(f"{device_name}: {status}")
```

```
Smart Device Control System
============================
Executing Smart Home Commands:
------------------------------

1. Command: {'device': 'light', 'room': 'living_room', 'action': 'toggle'}
   Result: Living_Room light turned off

2. Command: {'device': 'light', 'room': 'living_room', 'action': 'set_brightnes
s', 'value': 50}
   Result: Living_Room light brightness set to 50%

3. Command: {'device': 'light', 'room': 'bedroom', 'action': 'set_color', 'valu
e': 'blue'}
   Result: Light not found in bedroom

4. Command: {'device': 'ac', 'room': 'bedroom', 'action': 'set_temperature', 'val
ue': 20}
   Result: Bedroom AC temperature set to 20°C

5. Command: {'device': 'ac', 'room': 'bedroom', 'action': 'set_mode', 'value': 'c
ool'}
   Result: Bedroom AC mode set to cool

6. Command: {'device': 'oven', 'room': 'kitchen', 'action': 'preheat', 'value': 1
80}
   Result: Kitchen oven preheating to 180°C

7. Command: {'device': 'oven', 'room': 'kitchen', 'action': 'set_timer', 'value':
45}
   Result: Kitchen oven timer set for 45 minutes

8. Command: {'device': 'camera', 'room': 'security', 'action': 'toggle_recordin
g'}
   Result: Security camera recording stopped

9. Command: {'device': 'speaker', 'room': 'smart', 'action': 'set_volume', 'valu
e': 75}
   Result: Smart speaker volume set to 75%

10. Command: {'device': 'speaker', 'room': 'smart', 'action': 'play', 'value': 'c
lassical_music'}
   Result: Smart speaker now playing: classical music

11. Command: {'device': 'light', 'room': 'living_room', 'action': 'status'}
   Result: Living_Room light status:
   ✅ Status: On
   • Brightness: 50
   • Color: warm_white

12. Command: {'action': 'all_lights', 'command': 'off'}
   Result: All lights turned off: Living Room

13. Command: {'action': 'energy_save_mode'}
   Result: Energy save mode activated:
   • Light dimmed in living room
   • AC optimized in bedroom

Final System Status:
```

```
======================
Living Room Light: Offline
Bedroom Ac: Online
Kitchen Oven: Online
Security Camera: Online
Smart Speaker: Online
```

Key Match-Case Rules to Remember

Let's review the essential rules and best practices for using match-case statements effectively:

- Match-case is available in Python 3.10+ - ensure compatibility before using
- Use `_` as the wildcard pattern to catch all unmatched cases (like default in switch)
- Patterns are checked in order - place more specific patterns before general ones
- Guard conditions with `if` allow additional filtering within cases
- Use `|` (OR operator) to match multiple values in a single case
- Capture patterns with variables to extract and use matched values
- Object pattern matching requires the class to support it (dataclasses work well)
- Match-case can be more readable than long if-elif chains for complex conditions
- Pattern matching with destructuring works great for lists, tuples, and dictionaries
- Use match-case for complex data validation and routing logic
- Consider performance - match-case can be faster than if-elif for many conditions
- Combine guard conditions strategically to create powerful filtering logic

In [10]:

```python
# Advanced match-case best practices and performance examples

print("Match-Case Best Practices Demo")
print("=" * 30)

# Example 1: HTTP API Response Handler
print("\nHTTP API Response Handler")
print("-" * 25)

def handle_api_response(response):
    """Handle different API response patterns efficiently"""
    match response:
        case {"status": 200, "data": data, "message": msg}:
            return f"Success: {msg} | Data items: {len(data) if isinstance(data

        case {"status": 201, "data": {"id": new_id}, "message": msg}:
            return f"Created: {msg} | New ID: {new_id}"

        case {"status": code, "error": error_msg} if 400 <= code < 500:
            return f"Client Error ({code}): {error_msg}"

        case {"status": code, "error": error_msg} if 500 <= code < 600:
            return f"Server Error ({code}): {error_msg}"

        case {"status": 204}:
            return "Success: No content returned"

        case {"status": code} if code not in [200, 201, 204]:
            return f"Unexpected status code: {code}"
        case _:
            return "Invalid response format"

# Test API responses
api_responses = [
    {"status": 200, "data": [1, 2, 3, 4, 5], "message": "Users retrieved"},
    {"status": 201, "data": {"id": 12345}, "message": "User created successfully
    {"status": 400, "error": "Invalid request parameters"},
    {"status": 500, "error": "Database connection failed"},
    {"status": 204},
    {"status": 418},  # I'm a teapot!
    {"invalid": "response"}
]

for response in api_responses:
    result = handle_api_response(response)
    print(f"Response {str(response)[:40]}: {result}")

# Example 2: Data Processing Pipeline
print(f"\nData Processing Pipeline")
print("-" * 24)

def process_data_batch(batch):
    """Process different types of data batches efficiently"""
    match batch:
        case {"type": "csv", "file_path": path, "rows": rows} if rows > 1000:
            return f"Large CSV processing: {path} ({rows:,} rows) - Using chunke

        case {"type": "csv", "file_path": path, "rows": rows}:
            return f"Small CSV processing: {path} ({rows} rows) - Using standard

        case {"type": "json", "data": data, "schema_version": version} if versic
            return f"Modern JSON processing: v{version} with {len(data)} records

        case {"type": "json", "data": data, "schema_version": version}:
            return f"Legacy JSON processing: v{version} with {len(data)} records

        case {"type": "xml", "file_path": path, "size_mb": size} if size > 100:
            return f"Large XML processing: {path} ({size}MB) - Using streaming p

        case {"type": "xml", "file_path": path, "size_mb": size}:
            return f"Small XML processing: {path} ({size}MB) - Using DOM parser"

        case {"type": "binary", "format": fmt, "size_mb": size} if fmt in ["parc
            return f"Optimized binary processing: {fmt} format ({size}MB)"

        case {"type": "stream", "source": source, "real_time": True}:
            return f"Real-time stream processing: {source} - Using event-driven

        case {"type": "stream", "source": source, "real_time": False}:
            return f"Batch stream processing: {source} - Using micro-batch proce

        case _:
            return "Unsupported data batch format"

# Test data batches
data_batches = [
    {"type": "csv", "file_path": "/data/large_dataset.csv", "rows": 50000},
    {"type": "csv", "file_path": "/data/small_dataset.csv", "rows": 100},
    {"type": "json", "data": [{"a": 1}, {"b": 2}], "schema_version": 2.1},
    {"type": "json", "data": [{"x": 1}], "schema_version": 1.0},
    {"type": "xml", "file_path": "/data/huge_file.xml", "size_mb": 250},
    {"type": "xml", "file_path": "/data/config.xml", "size_mb": 5},
    {"type": "binary", "format": "parquet", "size_mb": 150},
    {"type": "stream", "source": "kafka_topic", "real_time": True},
    {"type": "stream", "source": "file_watcher", "real_time": False}
]

for batch in data_batches:
    result = process_data_batch(batch)
    print(f"Batch: {result}")

# Example 3: Performance Comparison
print(f"\nPerformance Optimization Example")
print("-" * 30)

def classify_transaction_old(transaction):
    """Old way using if-elif chains"""
    if transaction["amount"] < 0:
        if abs(transaction["amount"]) > 1000:
            return "Large Withdrawal"
        elif transaction["category"] == "grocery":
            return "Grocery Purchase"
        elif transaction["category"] == "gas":
            return "Gas Purchase"
        else:
            return "Small Purchase"
    elif transaction["amount"] > 0:
        if transaction["amount"] > 5000:
            return "Large Deposit"
        elif transaction["source"] == "salary":
            return "Salary Deposit"
        elif transaction["source"] == "refund":
            return "Refund"
        else:
            return "Other Deposit"
    else:
        return "Zero Amount"

def classify_transaction_new(transaction):
    """New way using match-case"""
    match transaction:
        case {"amount": amt, "category": "grocery"} if amt < 0:
            return "Grocery Purchase"
        case {"amount": amt, "category": "gas"} if amt < 0:
            return "Gas Purchase"
        case {"amount": amt} if amt < -1000:
            return "Large Withdrawal"
        case {"amount": amt} if amt < 0:
            return "Small Purchase"
        case {"amount": amt, "source": "salary"} if amt > 0:
            return "Salary Deposit"
        case {"amount": amt, "source": "refund"} if amt > 0:
            return "Refund"
        case {"amount": amt} if amt > 5000:
            return "Large Deposit"
        case {"amount": amt} if amt > 0:
            return "Other Deposit"
        case {"amount": 0}:
            return "Zero Amount"
        case _:
            return "Invalid Transaction"

# Test both approaches
test_transactions = [
    {"amount": -50, "category": "grocery"},
    {"amount": -30, "category": "gas"},
    {"amount": -1500, "category": "shopping"},
    {"amount": 3000, "source": "salary"},
    {"amount": 100, "source": "refund"},
    {"amount": 10000, "source": "investment"},
    {"amount": 0}
]

print("Comparing classification methods:")
for transaction in test_transactions:
    old_result = classify_transaction_old(transaction)
    new_result = classify_transaction_new(transaction)
    status = "✅" if old_result == new_result else "❌"
    print(f"{status} {str(transaction)[:35]}: {new_result}")

print(f"\nMatch-case provides cleaner, more maintainable code!")
```

```
Match-Case Best Practices Demo
==============================

HTTP API Response Handler
-------------------------
Response {'status': 200, 'data': [1, 2, 3, 4, 5],: Success: Users retrieved | Dat
a items: 5
Response {'status': 201, 'data': {'id': 12345}, ': Created: User created successf
ully | New ID: 12345
Response {'status': 400, 'error': 'Invalid reques: Client Error (400): Invalid re
quest parameters
Response {'status': 500, 'error': 'Database conne: Server Error (500): Database c
onnection failed
Response {'status': 204}: Success: No content returned
Response {'status': 418}: Unexpected status code: 418
Response {'invalid': 'response'}: Invalid response format

Data Processing Pipeline
------------------------
Batch: Large CSV processing: /data/large_dataset.csv (50,000 rows) - Using chunke
d processing
Batch: Small CSV processing: /data/small_dataset.csv (100 rows) - Using standard
processing
Batch: Modern JSON processing: v2.1 with 2 records
Batch: Legacy JSON processing: v1.0 with 1 records - Using compatibility mode
Batch: Large XML processing: /data/huge_file.xml (250MB) - Using streaming parser
Batch: Small XML processing: /data/config.xml (5MB) - Using DOM parser
Batch: Optimized binary processing: parquet format (150MB)
Batch: Real-time stream processing: kafka_topic - Using event-driven processing
Batch: Batch stream processing: file_watcher - Using micro-batch processing

Performance Optimization Example
------------------------------
Comparing classification methods:
✅ {'amount': -50, 'category': 'grocer: Grocery Purchase
✅ {'amount': -30, 'category': 'gas'}: Gas Purchase
✅ {'amount': -1500, 'category': 'shop: Large Withdrawal
✅ {'amount': 3000, 'source': 'salary': Salary Deposit
✅ {'amount': 100, 'source': 'refund'}: Refund
✅ {'amount': 10000, 'source': 'invest: Large Deposit
✅ {'amount': 0}: Zero Amount

Match-case provides cleaner, more maintainable code!
```

Course Information

Learn Python Programming from Scratch

Author: Prakash Ukhalkar

Topic: Python Control Flow - Match-Case Statements

Built with ❤️ for the Python learning community