03. Control Flow

October 16, 2025

0.1 1. The `else` clause

Somewhat surprising, the `else` clause in Python can be used together with multiple statements: - `if` statement - `for` statement - `while` statement - `try` statement

0.1.1 1.1. `else` Clause with Loops

The `else` clause in a `while` loop is executed when the loop terminates naturally (i.e., when the condition becomes `False`). If the loop is terminated by a `break` statement, the `else` clause will not be executed.

[1]:
```
n = 3
while n > 0:
    print(n)
    n -= 1
else:
    print("Loop ended naturally")
```

```
3
2
1
Loop ended naturally
```

If a `break` statement is used:

[2]:
```
n = 3
while n > 0:
    print(n)
    if n == 2:
        break
    n -= 1
else:
    print("Loop ended naturally")
```

```
3
2
```

The `else` clause in a `for` loop is executed when the loop finishes iterating over all items. Similar to the `while` loop, if the loop is terminated by a `break` statement, the `else` clause will not be executed.

[3]:
```
for i in range(3):
    print(i)
else:
    print("Completed all iterations")
```

```
0
1
2
Completed all iterations
```

If a `break` statement is used:

[4]:
```
for i in range(3):
    print(i)
    if i == 1:
        break
else:
    print("Completed all iterations")
```

```
0
1
```

0.1.2 Exercises 1.1

1. Write a program that displays all prime numbers between 1 and 100. Do it in two versions: with and without an `else` clause for the `for`.
2. Write an `authenticate_user` function that receives `correct_password` as parameter. The function should ask the user for the password, for a maximum of 3 attempts. If the password is correct, print `Access granted`. If the password is incorrect, display the number of attempts left. If the attempts are exhausted, print `Access denied`.

0.1.3 1.2. `else` Clause with `try` Statement

The `try … except` statement has an optional `else` clause, which, when present, must follow all `except` clauses. It is useful for code that must be executed if the try clause does not raise an exception. The use of the `else` clause is better than adding additional code to the `try` clause because it avoids accidentally catching an exception that wasn't raised by the code being protected by the `try … except` statement.

[3]:
```
try:
    x = 10 / 2
except ZeroDivisionError:
    print("Division by zero error")
else:
    print("Division successful:", x)
```

```
Division successful: 5.0
```

If an exception is raised:

[6]:
```
try:
    x = 10 / 0
except ZeroDivisionError:
    print("Division by zero error")
else:
    print("Division successful:", x)
```

```
Division by zero error
```

0.1.4 Exercises 1.2

1. Create a function `process_file` that takes a filename as an argument. If the file cannot be open, print a message with the reason. If the file was successfully open, print the number of distinct words in the file (word = any succession of characters that are not whitespace).

0.2 2. Assignment expressions (:=)

Assignment expressions allow assigning values to variables as part of an expression. Also known as "the walrus operator" due to its resemblance to the eyes and tusks of a walrus, the `:=` operator was introduced in Python 3.8. It provides a way to both evaluate and assign a value in a single, concise statement, enhancing readability and efficiency in certain scenarios.

Syntax

```
variable := expression
```

Key Features

1. **Inline Assignment:**
   - The walrus operator enables the assignment of a value to a variable within an expression, such as in a condition of an `if` statement, a loop, or a comprehension.
2. **Improved Readability:**
   - By combining assignment and condition checking, the walrus operator can make code more readable and reduce redundancy.
3. **Performance:**
   - It can improve performance by avoiding repeated evaluations of the same expression.

Examples

Using the Walrus Operator in `while` Loops

- Before the walrus operator:

[2]:
```
line = input("Enter something: ")
while line != "quit":
    print(f"You entered: {line}")
    line = input("Enter something: ")
```

```
Enter something:  something
```

```
You entered: something
```

```
Enter something:  something else

You entered: something else

Enter something:  quit
```

- With the walrus operator:

[3]:
```
while (line := input("Enter something: ")) != "quit":
    print(f"You entered: {line}")
```

```
Enter something:  hello

You entered: hello

Enter something:  world

You entered: world

Enter something:  quit
```

Using the Walrus Operator in `if` Statements

- Before the walrus operator:

[4]:
```
some_list = [7, 8, 2, 4, 7, 1]
value = len(some_list)
if value > 5:
    print(f"List is too long ({value} elements)")
```

```
List is too long (6 elements)
```

- With the walrus operator:

[5]:
```
some_list = [7, 8, 2, 4, 7, 1]
if (value := len(some_list)) > 5:
    print(f"List is too long ({value} elements)")
```

```
List is too long (6 elements)
```

Find more examples in the PEP documenting assignment expressions.

Best Practices

- **Readability First:** While the walrus operator can make code more concise, it's important to ensure that the code remains readable.
- **Use Judiciously:** Avoid overusing the walrus operator, especially in complex expressions where it might reduce clarity.
- **Python 3.8+:** Remember that the walrus operator is available only in Python 3.8 and later.

0.2.1 Exercises 2

For the first two exercises below, you can write two implementations: with and without walrus operator, and compare the two implementations.

1. Write a program that saves user inputs in a list. The program should stop asking the user to enter a word when the user doesn't enter anything (presses Enter).
2. Open and parse users.json using json. Save the result in a Python variable. Iterate over users and, for each user that is between 25 and 50 years old (including the limits) and has a name (key `name`), display a message: `User <name> is <age> years old.`
3. Can you think of another scenario where a walrus operator would simplify things?

0.3 3. `match` Statements

Introduced in Python 3.10, the `match` statement brings pattern matching capabilities to Python, allowing for more expressive and readable code when dealing with complex data structures. This feature is particularly useful for handling multiple conditions and branching logic in a clean and concise manner. Only the first pattern that matches gets executed and it can also extract components (sequence elements or object attributes) from the value into variables.

Syntax

```
match subject:
    case pattern1:
        # action for pattern1
    case pattern2:
        # action for pattern2
    ...
    case _:
        # default action if no patterns match
```

- **`match` Statement:** The keyword `match` is followed by the subject expression to be matched.
- **`case` Clauses:** Each `case` clause specifies a pattern and the corresponding action if the pattern matches.
- **Wildcard (`_`):** The wildcard `_` acts as a catch-all pattern, matching any value that hasn't matched any previous patterns.

Examples

Matching Simple Values Consider a function that returns a message based on HTTP status codes. The `match` statement simplifies the branching logic:

[11]:
```
def http_status(code):
    match code:
        case 200:
            return "OK"
        case 404:
            return "Not Found"
        case 401 | 403 | 405:  # several values can be combined in a single␣
↪pattern
            return "Not Allowed"
        case 500:
            return "Internal Server Error"
```

```
        case _:
            return "Unknown Status"
```

[12]:
```
http_status(200)
```

[12]:
```
'OK'
```

[13]:
```
http_status(201)
```

[13]:
```
'Unknown Status'
```

[14]:
```
http_status(403)
```

[14]:
```
'Not Allowed'
```

In this example, the function `http_status` uses a `match` statement to return messages based on the HTTP status code provided. The `_` case acts as a default, catching any status codes not explicitly handled.

Binding variables In the example above, we matched against numeric literals - actual values being compared to our variable. But we can also match against sequences, and also we can extract parts of the matched expression into variables:

[1]:
```
def process_tuple(point):
    match point:
        case (0, 0):
            print("Origin")
        case (0, y):
            print(f"Point on y axis (Y={y})")
        case (x, 0):
            print(f"Point on x axis (X={x})")
        case (x, y):
            print(f"X={x}, Y={y}")
        case _:
            print("Not a point")

process_tuple((0, 0))
process_tuple((15, 0))
process_tuple((0, 6))
process_tuple((4, 5))
process_tuple((0, 2, 3))
process_tuple(100)
```

```
Origin
Point on x axis (X=15)
Point on y axis (Y=6)
X=4, Y=5
Not a point
Not a point
```

Matching Dictionaries You may need to handle dictionaries with specific keys:

[2]:
```
def process_dict(d):
    match d:
        case {"key1": value1, "key2": value2}:
            print(f"Dictionary with key1={value1}, key2={value2}")
        case {"key1": value1}:
            print(f"Dictionary with key1={value1} and other keys")
        case _:
            print("Other dictionary")

process_dict({"key1": 1, "key2": 2, "key3": 3})
process_dict({"key1": 1, "key3": 3})
process_dict({"key3": 3, "key4": 4})
```

```
Dictionary with key1=1, key2=2
Dictionary with key1=1 and other keys
Other dictionary
```

The function `process_dict` matches dictionaries based on the presence of specific keys and handles them accordingly.

Matching Class Instances If you are using classes to structure your data you can use the class name followed by an argument list resembling a constructor, but with the ability to capture attributes into variables:

[19]:
```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

def process_point(point):
    match point:
        case Point(x=0, y=0):
            print("Origin")
        case Point(x=0):
            print(f"Y={point.y}")
        case Point(x=x, y=0):
            print(f"X={x}")
        case Point():
            print("Somewhere else")
        case _:
            print("Not a point")
```

[16]:
```
point = Point(1, 2)
process_point(point)
```

```
Somewhere else
```

[17]:
```
point = Point(0, 2)
process_point(point)
```

```
Y=2
```

[18]:
```
not_a_point = (1, 2)
process_point(not_a_point)
```

```
Not a point
```

In this example, the `process_point` function matches an instance of the `Point` class and extracts its attributes.

Using Guards We can add an if clause to a pattern, known as a "guard". If the guard is false, match goes on to try the next case block. Note that value capture happens before the guard is evaluated:

[6]:
```
def is_diagonal(point):
    match point:
        case (x, y) if x == y:
            print(f"Y=X at {x}")
            return True
        case _:
            print(f"Not on the diagonal")
            return False
```

[11]:
```
is_diagonal((2, 2))
```

```
Y=X at 2
```

[11]:
```
True
```

Advantages of Match Expressions

- **Readability:** Provides a clear and concise way to handle complex conditional logic.
- **Expressiveness:** Enables pattern matching for different data structures, including sequences and mappings.
- **Maintainability:** Improves code maintainability by reducing nested `if-elif-else` structures.

0.3.1 Exercises 3

1. Write a function `describe_day` that takes an integer representing a day of the week (1 for Monday, 2 for Tuesday, etc.) and returns a description of the day. Use the `match` statement to handle the following cases:
   - If the day is 1, return "Start of the work week".
   - If the day is 2, 3, or 4, return "Middle of the work week".
   - If the day is 5, return "End of the work week".
   - If the day is 6 or 7, return "Weekend".
   - For any other value, return "Invalid day".

2. Use data in users.json. Using a match expression, create a list of tuples `(name, age)`, where `name` will be either the value under key `name` or the values under `first_name` and `last_name` concatenated with a space between them.
3. Create a function `categorize_date` that takes a date object and categorizes it based on the month and day. Use the `match` statement to handle the following cases:
    - If the date is January 1st, return "New Year's Day".
    - If the date is December 25th, return "Christmas Day".
    - If the date is your birthday, return "My Birthday".
    - If the month is December, but not December 25th, return "December".
    - If the month is a summer month (June, July, August), return "Summer".
    - For any other date, return "Regular day".