Department of Computer Engineering
University of Peradeniya
CO523 - Programming Languages

Lab 06: Expressions, Control Structures, and Exception Handling

Objectives

- Understand expressions and operators
- Use conditional control structures
- Apply loop constructs
- Implement recursion
- Handle runtime errors using exceptions

1. Expressions

An expression is a combination of variables, constants, and operators that produces a value. Expressions are the basic building blocks of any program. They are used to calculate results, compare values, and make decisions. Expressions can be arithmetic (mathematical), relational (comparison), or logical.

Arithmetic expressions use operators such as +, -, *, /, and %. Relational expressions compare two values and produce a boolean result (true or false), such as >, <, ==, and !=. Logical expressions combine conditions using and, or, and not.

e.g.;

```
a = 10
b = 3
print(a + b)
print(a * b)
print(a > b)
print(a % b)
```

Result Explanation

- `a + b` adds 10 and 3, giving 13.
- `a * b` multiplies 10 and 3, giving 30.
- `a > b` checks if 10 is greater than 3, which is true.
- `a % b` finds the remainder when 10 is divided by 3, which is 1.

When an expression contains multiple operators, Python follows specific rules to determine the order of evaluation, known as operator precedence. Just like in standard mathematics, parentheses have the highest precedence and are used to force a specific order. Multiplication and division are prioritized over addition and subtraction.

If two operators have the same level of precedence, Python uses associativity to decide which goes first. Most operators are left-associative, meaning they are evaluated from left to right. However, assignment operators are a notable exception; they are right-associative, meaning the rightmost part of the expression is evaluated first. This allows for "chained assignments" such as `a = b = c = 10`, where the value 10 is passed from right to left.

1.2. Short-circuit evaluation

Short-circuiting is a programming concept in which the compiler skips the execution or evaluation of some sub-expressions in a logical expression. The compiler stops evaluating the further sub-expressions as soon as the value of the expression is determined.

e.g.;

```python
def check_value():
    print("Checking the second operand...")
    return True

x = 0
# Short-circuit prevents a ZeroDivisionError
if x != 0 and (10 / x) > 1:
    print("Success")
else:
    print("Short-circuited: Division was never attempted.")

# Demonstrating evaluation order
print(False and check_value())  # check_value() is never called
```

In the example above, the expression x != 0 evaluates to False. Because of short-circuiting, Python never attempts to evaluate (10 / x) > 1, which saves the program from a division-by-zero crash.

2. Conditional Control Structures

Conditional control structures decide which part of the program will run based on conditions. The most common structure is the if–elif–else statement. A condition is an expression that evaluates to either true or false.

If the condition in the if statement is true, that block runs. If not, the program checks the elif condition. If all conditions are false, the else block runs.

e.g.;

```
marks = int(input("Enter marks: "))

if marks >= 75:
    print("Distinction")
elif marks >= 50:
    print("Pass")
else:
    print("Fail")
```

- If marks are 80, the first condition is true and “Distinction” is printed.
- If marks are 60, the first is false but second is true, so “Pass” is printed.
- If marks are 40, both conditions are false, so “Fail” is printed.

3. Loop Constructs

A loop is a fundamental programming concept that allows for a set of instructions to be executed repeatedly based on a condition. It relies on a condition and keeps executing the block of code as long as the condition remains true. Typically, a loop uses constant memory, as it doesn’t require additional memory for each iteration. Generally, a loop is faster and more memory-efficient, and can be more straightforward and intuitive for simple tasks.

The most common types of loops include:

For Loops: Repeat a specific number of times.

While Loops: Repeat as long as a condition is true.

Do-While Loops: Execute the loop body at least once, then repeat as long as a condition is true.

3.1. For Loop

For loops is used to iterate over a sequence such as a list, tuple, string or range. It allow to execute a block of code repeatedly, once for each item in the sequence.

e.g.;

```
n = 4
for i in range(0, n):
    print(i)
```

This code prints the numbers from 0 to 3 (inclusive) using a for loop that iterates over a range from 0 to n-1 (where n = 4).

3.2. While Loop

A **While Loop** is used to execute a block of statements repeatedly until a given condition is satisfied. When the condition becomes false, the line immediately after the loop in the program is executed.

e.g.;4

```
count = 0
while (count < 3):
    count = count + 1
    print("Hello World")
```

In above code, loop runs as long as the condition *count < 3* is true. It increments the counter by 1 on each iteration and prints "Hello World" three times.

3.2.1. Infinite While Loop

If we want a block of code to execute infinite number of times then we can use the while loop to do so.

Code given below uses a 'while' loop with the condition "True", which means that the loop will run infinitely until we break out of it using "break" keyword or some other logic.

e.g.;

```
while (True):
    print("Hello World")
```

Note: *It is not recommended to use this type of loop because the loop condition is always true, resulting in an infinite loop. Such loops continue executing indefinitely unless they are explicitly terminated using an external interruption.*

3.3. Do while loop

Do while loop is a type of control looping statement that can run any statement until the condition statement becomes false specified in the loop. In do while loop the statement runs at least once no matter whether the condition is false or true.

In Python, there is no construct defined for do while loop. Python loops only include for loop and while loop but we can modify the while loop to work as do while as in any other languages such as C, C++ and Java.

e.g.;

Do While loop in C language

```
#include <stdio.h>

int main() {
    // Loop variable declaration and initialization
    int i = 0;

    // do while loop
    do {
        printf("Hello\n");
        i++;
    } while (i < 3);

    return 0;
}
```

The do...while loop in this C program prints "Hello" three times by executing the loop body at least once and continuing until the condition i < 3 becomes false.

3.4. Nested Loops

A nested loop is a loop inside another loop. The inner loop executes completely every time the outer loop executes once. Nested loops are useful when working with multidimensional data, such as matrices or grids, or when you need to repeat a set of operations multiple times in a structured way. Most programming languages support nested loops using for, while, or other loop constructs.

e.g;

```
rows = 5

for i in range(1, rows + 1):      # Outer loop for rows
    for j in range(1, i + 1):     # Inner loop for numbers in a row
        print(j, end=" ")         # Print numbers on the same line
    print()                        # Move to next line after inner loop
```

The program prints a right-angled number triangle, where each row contains numbers starting from 1 up to the row number. The **outer loop** controls the number of rows, and the **inner loop** prints the numbers in each row on the same line. After each row, a new line is started to form the triangular pattern.

3.5. Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed. They allows to exit a loop prematurely or skip iterations instead of letting the loop run normally to its end.

3.5.1. continue Statement

The continue statement skips the current iteration and proceeds to the next iteration of the loop.

e.g.;

```python
# Using continue to skip an iteration
for i in range(10):
    if i % 2 == 0:
        continue
    print(i)
```

In this example the loop prints odd numbers from 0 to 9. When i is even, the continue statement skips the current iteration.

3.5.2. break Statement

The break statement is used to exit the loop prematurely when a certain condition is met.

e.g.;

```python
# Using break to exit the loop
for i in range(10):
    if i == 5:
        break
    print(i)
```

The loop prints numbers 0 to 4 and then stops completely when i becomes 5 due to the break statement.

4. Recursion

Recursion is a technique where a function calls itself in order to break down a problem into smaller, more manageable pieces. The idea is that the problem can be solved by solving one or more smaller instances of the same problem.

A recursive approach generally involves:

Base Case(s): These are the conditions where the function stops calling itself.

Recursive Case: The scenario in which the function calls itself, usually with modified arguments.

Every recursive function must have a base case to stop the recursion. Without a base case, the function will run forever and cause an error.

Recursion breaks down a problem into smaller instances of the same problem until it reaches the base case. Each recursive call creates a new layer on the call stack, which can lead to increased memory consumption, and deep recursions can result in a stack overflow. It offers a more elegant and concise solution for certain problems, especially those that have a naturally recursive structure like the Fibonacci sequence or tree traversals. Recursion can be slower and less memory-efficient due to overheads of function calls, but it may be more intuitive for complex problems.

e.g.;

Calculating a factorial is the simplest example of recursion that will really help to understand how it works. The factorial of a number is the multiplication of all numbers from 1 up to that number.

For example, the factorial of 5 is 120 – that is 5×4×3×2×1.

It can also be represented mathematically as 5×(5−1)! which means that if we know the value of (5−1)! we can easily get the factorial by just multiplying 5 by it.

```
Factorial of 4 = 4×(4-1)!

Factorial of 3 = 3×(3-1)!

Factorial of 2 = 2×(2-1)!

Factorial of 1 = 1

Factorial of 0 = 1
```

This is python example for finding the factorial of a number

```python
def factorial(n):
    if n == 0 or n == 1:
        return 1
    return n * factorial(n-1)

print(factorial(5))
```

Result Explanation

- factorial(5) = 5 × factorial(4)
- factorial(4) = 4 × factorial(3)
- factorial(3) = 3 × factorial(2)
- factorial(2) = 2 × factorial(1)
- factorial(1) = 1

So, the final result is 120.

In the above example, we have used a stopping condition (base case) for the recursion. But what if we don’t add a stopping condition or what if the function we write never meets the stopping condition?

5. Exception Handling

Exception handling is a programming concept used to manage errors that occur during the execution of a program. When an error occurs, the normal flow of the program is disrupted.

The program creates an "exception" object that contains information about the error. The process of responding to this exception is called "exception handling".

Exception Handling mainly revolves around two concepts one is Exception other is Handling.

Exception: An exception is an unwanted event that interrupts the normal flow of the program.

Handling: A block of Code that Handles or continues the flow of the program even after the exception is occurred (e.g. Try-Catch Block).

5.1. Components of Exception Handling

Exception handling typically involves three main components:

Try Block: The code that may potentially throw an exception is enclosed within a try block. If an exception occurs within this block, the control is transferred to the corresponding catch block.

Catch Block: This block catches and handles the exceptions thrown within the try block. Each catch block is associated with a specific type of exception, allowing developers to handle different types of errors separately.

Finally Block (Optional): The finally block is executed regardless of whether an exception occurs or not. It is commonly used to perform clean-up tasks, such as closing files or releasing resources.

e.g.;

In Python, the try-except-finally statement facilitates structured exception handling with three essential components: the try block, the except block, and the finally block.

try Block: Encloses code where exceptions may occur, anticipating potential errors.

except Block: Follows the try block and handles exceptions that occur within it, allowing for specific actions based on exception types.

finally Block: Optional block that follows the try and except blocks, executing code always, typically used for cleanup tasks like releasing resources.

```python
try:
    numerator = 10
    denominator = 0

    # Attempt division
    result = numerator / denominator
    print("Result of division:", result)

except (ZeroDivisionError, ValueError) as e:
    # Handle division by zero and invalid input
    print("Error:", e)
```

```
finally:
    # Always executes
    print("Operation completed.")
```

In this code, a ZeroDivisionError occurs because the denominator is 0, so the program jumps to the except block and prints the error message. The finally block always executes, so “Operation completed.” is printed regardless of whether an exception occurred.

Exception Handling Best Practices are essential for writing robust and reliable code. They include catching specific exceptions rather than using a general catch-all, providing clear and informative error messages, and handling exceptions at the appropriate level. Additionally, using finally blocks for resource cleanup, differentiating between checked and unchecked exceptions, documenting exception handling, and thoroughly testing all exception scenarios help ensure that the program behaves correctly under all conditions.

Lab Tasks

Task 01

Write a program that:

- Takes an integer input
- Checks if it is positive, negative, or zero
- If positive, find its factorial using recursion
- If negative, find the sum of digits using a loop
- Use exception handling for invalid input

Task 02

Write a program that:

- Takes a number `n`
- Prints a number pattern using loops:
  1
  12
  123
  … up to n
- Then calculates `n` raised to power 5 using recursion

Task 03

Create a menu-based program:

1. Find factorial (loop)
2. Find factorial (recursion)
3. Reverse a number (loop)
4. Reverse a number (recursion)
5. Exit

Use:

- if–else for menu selection
- loops for repetition
- recursion where required
- exception handling for invalid choices and input

Task 04

Build a calculator that supports:

- Addition, subtraction, multiplication, division
- Uses loops to continue until user exits
- Uses exception handling for:

    - Division by zero
    - Invalid input
- Uses expressions for all calculations

Task 05

Explain what happens if a recursive function does not have an exit (base) condition. Write a simple Python example of a recursive function without a base case, and describe the result when it is executed.

Submission

Create a Zip PDF file containing all your source codes and explanations. Name the file CO523_Lab06_EXXYYY.zip, where EXXYYY represents your E-number. Ensure all code is properly formatted and readable before submission.

Upload your answer sheet to the FEeLS by the given deadline.