Seed Programming

Python Fundamentals

Lecture 03: Functions and the `for` Loop

1 Learning Objectives

By the end of this lecture, students should be able to:

- Explain why functions are needed.
- Differentiate between built-in and user-defined functions.
- Define and call functions.
- Explain arguments and parameters.
- Trace function execution using the Visual Studio Code debugger.
- Explain modular programming.
- Use the `for` loop and `range()`.
- Compare `while` and `for` loops.
- Use the `break` statement.
- Build a menu-driven program.

2 Review

Before introducing new material, it is useful to revisit three ideas from the previous lecture because functions and `for` loops build directly on them.

2.1 Debugging

Debugging is the process of stepping through a program line by line to observe how the values of variables change over time. The Visual Studio Code debugger lets us set breakpoints, run the program to a selected line, and then execute one line at a time while watching the Variables panel.

This tool is especially useful in this lecture because it allows us to see how arguments enter functions and how loop variables change during each iteration.

2.2 The `input()` Function

The `input()` function pauses the program, displays a prompt, waits for the user to type something and press Enter, and returns the entered value as a string.

```
1 name = input("Enter your name: ")
```

2.3 The `while` Loop

A `while` loop repeats a block of statements as long as a condition remains true. It normally requires three ingredients: a variable initialized before the loop, a condition checked before every iteration, and an update that eventually makes the condition false.

```
1 i = 1
2 while i <= 5:
3     print(i)
4     i = i + 1
```

If one of these ingredients is missing or incorrect, the loop may never run or may run forever. With these ideas reviewed, we can ask a more interesting question: what should we do when the same block of code is needed in several places?

3 Why Do We Need Functions?

Consider a program that prints multiplication tables for `2`, `5`, and `7` at different points. Using only variables and loops, we might copy the same logic three times and change the number in each copy.

```
# Table for 2
i = 1
while i <= 10:
    print(2, "x", i, "=", 2 * i)
    i = i + 1

# Table for 5
i = 1
while i <= 10:
    print(5, "x", i, "=", 5 * i)
    i = i + 1

# Table for 7
i = 1
while i <= 10:
    print(7, "x", i, "=", 7 * i)
    i = i + 1
```

This approach works, but it creates unnecessary repetition.

3.1 Code Duplication

Copying and pasting the same code repeatedly is called *code duplication*. It causes several problems:

- **Longer programs.** Every copy adds more lines without adding a new capability.
- **Difficult maintenance.** If the output format changes, every copy must be found and edited separately.
- **More mistakes.** Each copied block can contain a typo or become inconsistent with the others.

What we really want is to write the multiplication-table logic once and run it with a different number whenever needed. A function provides exactly this behavior.

4 Built-in Functions

Students have used functions since their first Python program:

```
name = input("Enter your name: ")
print("Welcome", name)
```

The `print()` and `input()` functions are *built-in functions*: pieces of ready-made code included with Python. To use a built-in function, we need to know:

- its name,
- what input it expects, and
- what it does.

A function can be viewed as a black box that accepts some input, performs a task, and may produce output. The caller does not need to know every internal implementation detail.

5 User-Defined Functions

Python also allows us to create our own functions with the `def` keyword. A user-defined function packages a block of code under a name so that it can be executed again later.

```
def tablePrinter(Table):
    i = 1
    while i < 10:
        print(Table, "x", i, "=", Table*i)
        i = i + 1
```

Important parts of this definition are:

- `def` tells Python that a function is being defined.
- `tablePrinter` is the function name.
- `Table` is a parameter that receives the table number.
- The indented statements form the function body.

The body does not run when Python reads the `def` statement. It runs only when the function is called.

5.1 Calling a Function

Defining a function does not execute it. To run the function, write its name followed by parentheses containing the value to pass into it.

```
table = 5
tablePrinter(table)
```

The same function can be called repeatedly with different values:

```
tablePrinter(2)
tablePrinter(5)
tablePrinter(7)
```

The multiplication logic is now written only once. If its output format needs to change, there is only one function body to edit.

For example, a call to `tablePrinter(5)` begins with `i = 1`. Each iteration prints a row and then increases `i`. The final row is `5 x 9 = 45`; after the increment, `i` is `10`, so the condition `i < 10` is false. The table stops at `9`, not `10`, because the condition uses strictly less than.

The function can be wrapped in a task that obtains input from the user:

```
def Task1():
    T = input("Which Table: ")
    T = int(T)
    tablePrinter(T)
```

The `input()` function returns a string, so `int(T)` converts the input to an integer. Without this conversion, multiplying by `i` would repeat a string rather than perform numerical multiplication.

6 Arguments and Parameters

The terms *argument* and *parameter* describe different parts of a function call:

- A **parameter** is a name written in the function definition. In `def tablePrinter(Table):`, `Table`

is the parameter.
- An **argument** is the value supplied when the function is called. In `tablePrinter(table)`, the value stored in `table` is the argument.

This can be observed with the Visual Studio Code debugger. Set a breakpoint on the first line of `tablePrinter`, run the program, and step into the call. The following events occur:

- Python evaluates the argument before entering the function.
- The parameter `Table` is created as a new local name inside the function and receives the argument's value.
- The original variable `table` remains available in the calling code.
- The parameter exists only during the function call and disappears when the function finishes.

For the integer values used here, reassigning `Table` inside the function does not change the outer variable `table`.

7 Modular Programming

Modular programming means dividing a large program into smaller, focused parts. A useful analogy is a car: one team may build the engine, another the braking system, and another the dashboard. Each team focuses on one module, and the complete car is assembled from these independently developed parts.

Programs can be organized in the same way. Instead of one long block of code, a program can contain small functions responsible for tasks such as printing a table, validating input, calculating a total, or displaying a menu.

Benefits of modular programming include:

- **Reusability.** A function written once can be called many times.
- **Easier maintenance.** A change is made in the function responsible for that task.
- **Easier debugging.** Each function can be tested and fixed separately.
- **Team collaboration.** Different programmers can develop different functions as long as they agree on how those functions are used.

8 Introduction to the `for` Loop

One common mistake with a `while` loop is forgetting its update statement:

```
1 i = i + 1
```

Other common mistakes include using the wrong increment or starting from the wrong initial value. Python provides the `for` loop for situations where a task should repeat over a known sequence of values.

8.1 Syntax and `range()`

The general form is:

```
1 for i in range(start, stop, step):
2     statements
```

The `range(start, stop, step)` function generates values beginning at `start`, changing by `step`, and stopping before `stop`. The stop value is not included.

Range	Generated values
`range(1, 11)`	`1, 2, 3, ..., 10`
`range(0, 10, 2)`	`0, 2, 4, 6, 8`
`range(10, 0, -1)`	`10, 9, 8, ..., 1`

When `step` is omitted, it defaults to `1`. On every iteration, Python automatically assigns the next generated value to the loop variable.

8.2 Comparing `while` and `for` Loops

Both loops repeat code, but they are suited to different situations.

	Use when	Example
`while` loop	The number of repetitions is not known in advance and depends on a condition.	Keep asking for input until the user enters `"quit"`.
`for` loop	The number of repetitions or sequence of values is known.	Print the numbers from `1` through `10`.

A `for` loop over `range()` can be rewritten as a `while` loop, but the `for` version is often shorter and clearly communicates that a known sequence is being processed.

9 Pattern-Printing Functions

Repeated strings and loops can be combined to draw rectangular, triangular, and diamond-shaped patterns. These examples also show how counters may control the number of rows, symbols, and spaces.

9.1 Rectangle

```
1 def printRectangle(L, W, sym):
2     for i in range(1, L+1):
3         print(sym*W)
```

The range contains exactly `L` values, so the function prints `L` rows. On every iteration, `sym*W` produces the same row of `W` symbols. The loop variable `i` is not used in the body; it only controls how many times the body runs. For example, `printRectangle(3, 4, "*")` produces:

```
****
****
****
```

9.2 Growing Left-Aligned Triangle

```
1 def printT1(H, sym):
2     st = 1
3     for ln in range(1, H+1):
4         print(sym*st)
5         st += 1
```

The counter `st` stores the number of symbols to print. It begins at `1` and increases after every row.

A trace of `printT1(4, "*")` is:

ln	st before printing	Output
1	1	*
2	2	**
3	3	***
4	4	****

The code and trace logic are correct: each row prints one more symbol than the previous row.

9.3 Right-Aligned Triangle

```
def printT2(H, sym):
    sp = H-1
    st = 1
    for i in range(1, H+1):
        print(" "*sp, end="")
        print(sym*st)
        sp -= 1
        st += 1
```

Here two counters move in opposite directions. The number of leading spaces, `sp`, decreases while the number of symbols, `st`, increases. The argument `end=""` prevents the first `print()` call from moving to a new line, so the spaces and symbols appear on the same row. Calling `printT2(4, "*")` produces:

```
   *
  **
 ***
****
```

Tasks can collect input and call the two triangle functions:

```
def Task3():
    Height = int(input("Height: "))
    sym = input("Symbol: ")
    printT1(Height, sym)

def Task4():
    Height = int(input("Height: "))
    sym = input("Symbol: ")
    printT2(Height, sym)
```

9.4 Diamond Upper Triangle and Default Parameters

```
def DUT(H, sym, sln=1):
    sp = H-1
    st = 1
    for ln in range(1, H+1):
        if ln >= sln:
            print(" "*sp, end="")
            print(sym*st)
```

```
8           sp -= 1
9           st += 2
```

The parameter `sln` means "start line" and has the default value `1`. If the caller omits it, printing begins on the first row. Rows before `sln` are not printed, but `sp` and `st` are still updated so that later rows keep their correct widths. Because `st` grows by `2`, the row widths are odd numbers: `1`, `3`, `5`, and so on. Thus, `DUT(4, "*")` produces:

```
   *
  ***
 *****
*******
```

9.5 Diamond Lower Triangle

```
1  def DLT(H, sym, sln=1):
2      sp = 0
3      st = 2*H-1
4      for ln in range(1, H+1):
5          if ln >= sln:
6              print(" "*sp, end="")
7              print(sym*st)
8          sp += 1
9          st -= 2
```

This function mirrors `DUT`. It begins with `2*H-1` symbols, then adds one leading space and removes two symbols after every row.

9.6 Combining the Diamond Halves

```
1  def Diamond(H, sym):
2      DUT(H, sym)
3      DLT(H, sym, 2)
```

The first call prints the complete upper half. The second call passes `2` as `sln`, skipping the lower triangle's first row because it would duplicate the widest row already printed by `DUT`. This illustrates modular programming: two independent functions are combined to construct a larger pattern.

9.7 Pyjama Pattern

```
1  def Pyjama(H, sym):
2      st = H
3      sp = 0
4      for ln in range(1, H+1):
5          print(sym*st, end="")
6          print(" "*sp, end="")
7          print(" "*sp, end="")
8          print(sym*st)
9          st -= 1
10         sp += 1
```

Each row is assembled from four pieces: a left block of symbols, two blocks of spaces, and a right block of symbols. The symbol blocks shrink while the central gap grows. For `Pyjama(3, "*")`, the result is:

```
******
**  **
*    *
```

If the file ends with the following call, the pattern is printed immediately when the script runs:

```
Pyjama(10, "*")
```

The other functions are only defined until a task or another statement calls them.

Remark. *When pattern output is incorrect, trace the symbol and space counters separately and check whether each counter should increase, decrease, or remain fixed.*

10 Menu-Driven Program

A menu-driven program repeatedly displays options, lets the user choose a task, performs it, and then asks whether the user wants to continue.

```
while True:
    print("1. Task 1")
    print("2. Task 2")

    task = input("Which task to run? ")

    if task == "1":
        pass

    if task == "2":
        pass

    choice = input("Do you want to continue? ")

    if choice == "N" or choice == "n":
        break

print("Good Bye...")
```

The structure works as follows:

- `while True` starts an intentional loop that keeps displaying the menu.
- The `print()` calls display the options, and `input()` reads the user's selection.
- Each `if` block represents a menu option. The `pass` statements are placeholders that can later be replaced by function calls.
- The program asks whether the user wants to continue.
- If the answer is `"N"` or `"n"`, `break` exits the loop.
- After the loop ends, the program prints `"Good Bye..."`.

This combination of an intentional infinite loop, menu choices, task blocks, and an exit condition is a common program structure.

11 The `break` Statement

The `break` statement immediately terminates the nearest enclosing `while` or `for` loop and transfers control to the first statement after that loop. Python does not return to the top of the loop to check its condition again.

In the menu program, this behavior is necessary because `while True` never becomes false by itself. When the user enters `"n"`, `break` exits the loop so that execution can continue with the goodbye message.

The `break` statement is also useful when:

- searching until a target value is found,
- reading input until a sentinel value is entered, or
- repeating a menu until the user decides to exit.

12 Summary

Students should now be able to:

- Create and call functions.
- Explain arguments and parameters.
- Apply modular programming.
- Use the `for` loop and `range()`.
- Debug logical errors.
- Use the `break` statement.
- Develop simple menu-driven applications.