Collections and Iteration: Looping Through Data

Applied Python Programming with AI and Raspberry Pi Interfaces

Instructor: Dr. Vikas Thammanna Gowda Semester: ABCD 20YX

Module Overview: *This module introduces students to Python's fundamental collection data types and the looping structures used to process them. Students will learn how to store, organize, and manipulate data using* ***lists****,* ***tuples****,* ***sets****, and* ***dictionaries****, each offering unique capabilities for handling information efficiently. The module also covers* ***iteration techniques****, focusing on the use of* `while` *and* `for` *loops to process elements in collections. Students will explore the differences between these loop types, when to use each, and how to integrate them into problem-solving workflows. Practical examples and exercises will emphasize working with collections to store data, perform lookups, iterate through items, and apply modifications where appropriate.* **Learning Objectives:**

1. *Create, modify, and access elements in a list to store and manage ordered data.*
2. *Define and utilize tuples to store fixed, immutable sequences of data.*
3. *Use sets to store unique values and perform operations such as unions, intersections, and differences.*
4. *Create and manipulate dictionaries to store data as key–value pairs for fast lookups.*
5. *Use* `while` *loops to perform repeated actions until a specified condition is met.*
6. *Use* `for` *loops to iterate over elements in a collection or range efficiently.*
7. *Combine collection types with loop structures to process, analyze, and transform data.*

Contents

Chapter 1

Collection Data types

1.1 Introduction

In Python, a *collection* is a data type that can hold multiple values under a single name. Collections help us organize and work with groups of related items—like a list of student names, a tuple of coordinates, a set of unique vocabulary words, or a dictionary mapping student IDs to grades. Python provides four built-in collection types:

- **List**: An ordered, *mutable* sequence of items.
- **Tuple**: An ordered, *immutable* sequence of items.
- **Set**: An *unordered* collection of *unique* items.
- **Dictionary**: A mapping of *keys* to *values* (an unordered collection of key-value pairs).

Together, these four **data structures** let you handle almost any grouping of data you might need in your programs.

1.2 List

Definition: A list is an ordered sequence of elements, enclosed in square brackets `[ ]`. (Ordering starts at 0 and goes until length - 1)
Mutability: You can change (add, remove, or modify) elements after the list is created.
Typical Uses: Storing a sequence of items where order matters or duplicates are allowed.

1.2.1 Key Operations

- `append(item)`: add to end
- `insert(i, item)`: insert at index `i`
- `remove(item)`: delete first matching
- `pop()` or `pop(i)`: remove and return
- `len(list)`: number of items

1.2.2 Illustration

Example:

```
students = ["Alice", "Bob", "Charlie", "Bob"]
print(students)


# Accessing elements by index (0-based)
first = students[0]
last = students[-1]

# Adding items
students.append("Diana")     # adds to end

# Removing items
students.remove("Bob")       # removes first "Bob"

print(students)
```

Output:

```
['Alice', 'Bob', 'Charlie', 'Bob']
['Alice', 'Charlie', 'Bob', 'Diana']
```

1.3 Tuple

Definition: A tuple is like a list but immutable, enclosed in parentheses ().
Immutability: Once created, you cannot change its elements.
Typical Uses: Fixed collections—such as coordinates (x, y), or days of the week—that should not change.

1.3.1 Key Properties

- Ordered (indexing and slicing possible)
- Immutable (elements cannot be changed once set)
- Faster for fixed data

1.3.2 Illustration

Example:

```
# Creating a tuple of coordinates
point = (3, 7)
print(point)

# Accessing elements
x, y = point            # tuple unpacking
print("x =", x, "y =", y)

# Single-element tuple needs a trailing comma
singleton = ("only one",)
print(type(singleton))
```

Output:

```
(3, 7)
x = 3 y = 7
<class 'tuple'>
```

1.4 Set

Definition: A set is an unordered collection of unique items, enclosed in braces {}.
Uniqueness: Duplicate entries are automatically removed.
Typical Uses: When you need to test membership quickly or want only one of each item. For example, unique words in a text.

1.4.1 Key Operations

- `add(item)`: add element
- `remove(item)`: remove element (error if missing)
- `discard(item)`: remove if present
- `union()/intersection()/difference()`

1.4.2 Illustration

Example:

```
# Creating a set of vocabulary words
words = {"apple", "banana", "apple", "cherry"}
print(words)


# Adding and removing items
words.add("date")
words.remove("banana")

# Membership test
if "apple" in words:
    print("We have an apple!")

# Set operations
even = {2, 4, 6, 8}
odd  = {1, 3, 5, 7}
all_numbers = even.union(odd)           # union
common = even.intersection({4,5}) # intersection
diff = even.difference({6,8})   # difference

print("All numbers:", all_numbers)
print("Common numbers:", common)
print("Difference:", diff)
```

Output:

```
{'banana', 'cherry', 'apple'}
We have an apple!
All numbers: {1, 2, 3, 4, 5, 6, 7, 8}
Common numbers: {4}
Difference: {2, 4}
```

1.5 Dictionary

Definition: A dictionary stores key-value pairs, enclosed in braces with colons (`:`)

Syntax:

```
{key: value}
```

Uniqueness: Since Python 3.7, insertion-order is preserved, but keys are still looked up by hash, not position.
Typical Uses: When you need to look up information by a unique key—such as student IDs, product SKUs, or word - definition pairs.

1.5.1 Key Methods

- `dict.keys()/values()/items()`
- `get(key, default)`: safe lookup
- `pop(key)`: remove and return value
- `clear()`: remove all entries

1.5.2 Illustration

Example:

```
# Creating a dictionary mapping student IDs to grades
grades = {"S001": 85, "S002": 92, "S003": 78}
print(grades)

# Accessing values by key
grade_s002 = grades["S002"]   # 92

# Adding or updating entries
grades["S004"] = 88           # add new key
grades["S001"] = 90           # update existing

# Removing entries
del grades["S003"]            # removes key "S003"


# Getting all keys or all values
all_ids   = grades.keys()
all_scores = grades.values()

print("All keys:", all_ids)
print("All values:", all_scores)
```

Output:

```
{'S001': 85, 'S002': 92, 'S003': 78}
All keys: dict_keys(['S001', 'S002', 'S004'])
All values: dict_values([90, 92, 88])
```

1.6 Comparison of Collections

Collection	Ordered?	Mutable?	Duplicates?	Best For
List	Yes	Yes	Yes	Sequences you need to change (e.g., playlists)
Tuple	Yes	No	Yes	Fixed data (e.g., coordinates)
Set	No	Yes	No	Unique items & fast membership tests
Dictionary	Yes	Yes	Keys unique	Mapping keys to values (e.g., lookups by ID)

Chapter 2

Loops and Iteration

2.1 Introduction to Loops

2.1.1 What is a loop and why do we use it?

A **loop** is a programming construct that allows a set of instructions to be executed repeatedly until a certain condition is met. In simpler terms, it's like hitting a "repeat" button on a block of code so that we don't have to write the same instructions over and over. We use loops to automate repetitive tasks, to iterate over data structures (like all elements in a list), and to run simulations or processes that require many repeated steps. By using loops, a program can perform tasks multiple times efficiently – for example, processing every item in a list with just a few lines of code, instead of writing one line of code per item.

Without loops, if we needed to do something 100 times, we would have to copy-paste the code 100 times (which is impractical). Loops let us *reuse* a single block of code and specify how many times or under what conditions it should run. This makes programs shorter, easier to read, and less error-prone.

2.1.2 The four components of a loop

In general, a loop has four key components:

- **Initialization**: Set up any variables needed for the loop. For example, initialize a counter variable to a starting value.
- **Condition (Continuation Test)**: A boolean expression that is checked before each iteration of the loop. If the condition is true, the loop body executes; if it is false, the loop stops. This condition determines whether another iteration should happen.
- **Body**: The set of instructions that executes each time through the loop (the repeated code block).
- **Update**: Change something in the loop's state that will eventually make the condition false. Often this is an update to the loop variable (e.g., incrementing a counter) so that the loop moves toward termination.

These four components work together to implement repetition. For example, consider a simple loop that prints numbers 1 to 5: we would *initialize* a counter at 1, loop while the *condition*

"`counter <= 5`" is true, *body* prints the counter, and then *update* by adding 1 to the counter each time. Once the counter becomes 6, the condition fails and the loop stops.

It's important that the update step will eventually make the loop's condition false; otherwise, the loop would run forever. Not all loop constructs explicitly show all four components (for instance, Python's `for` loop hides some of them), but conceptually they are present in any loop structure.

2.2 The `while` Loop

The `while` loop in Python repeats a block of code as long as a given condition remains true. It is known as a **pre-test loop** (or entry-controlled loop) because the loop's condition is checked at the start of each iteration, before the body executes. The basic syntax is:

Syntax:

```
while (condition):
    <statement block>
```

Here, the `(condition)` is a boolean expression. The loop will keep running the `<body>` as long as the condition evaluates to True. The moment the condition becomes False, the loop stops and execution continues after the loop.

Always make sure the `condition` will eventually become `False`, or you'll create an infinite loop (your program runs forever!).

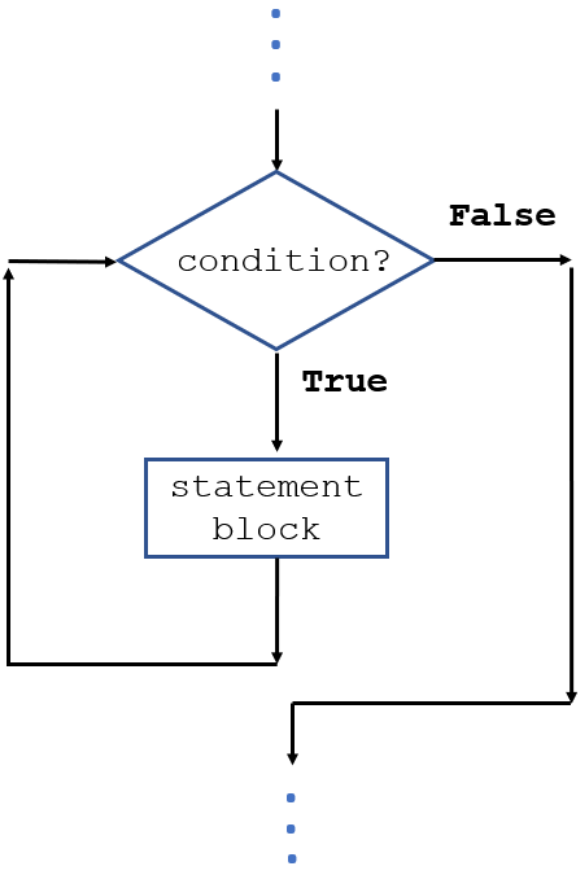

Figure 2.1: Flow chart for a `while` loop

2.2.1 Illustration

Example 1:

```
count = 1                # start with count = 1
while (count <= 5):        # keep looping while count is 1, 2, 3, 4,
    or 5
     print("Count is", count)
     count = count + 1   # increase count by 1 each time
```

Initialization: This sets up the loop's starting state. In the example:

```
count = 1
```

Condition Check: A boolean test evaluated before each iteration; the loop runs only if it's true:

```
while (count <= 5):
```

Statement Block: The indented code that executes on each pass through the loop:

```
print("Count is", count)
```

Updation: Changes the loop variable so the condition eventually becomes false:

```
count = count + 1
```


Output:

```
Count is 1
Count is 2
Count is 3
Count is 4
Count is 5
```

Execution Flow

1. Start: Dispaly a message
2. `count = 1` (initialization)
3. Check: `1 <= 5`? → true
4. Execute block: prints "Count is 1"
5. Update: `count = 2`
6. Repeat until `count = 6`, then `6 <= 5`? → `False`, exit loop.

Notice that if the loop's condition is `False` from the start, the loop body will not execute even once. For example, if we had initialized `count = 10` in the above code, the condition `count <= 5` would be `False` immediately and the print inside the loop would never run.

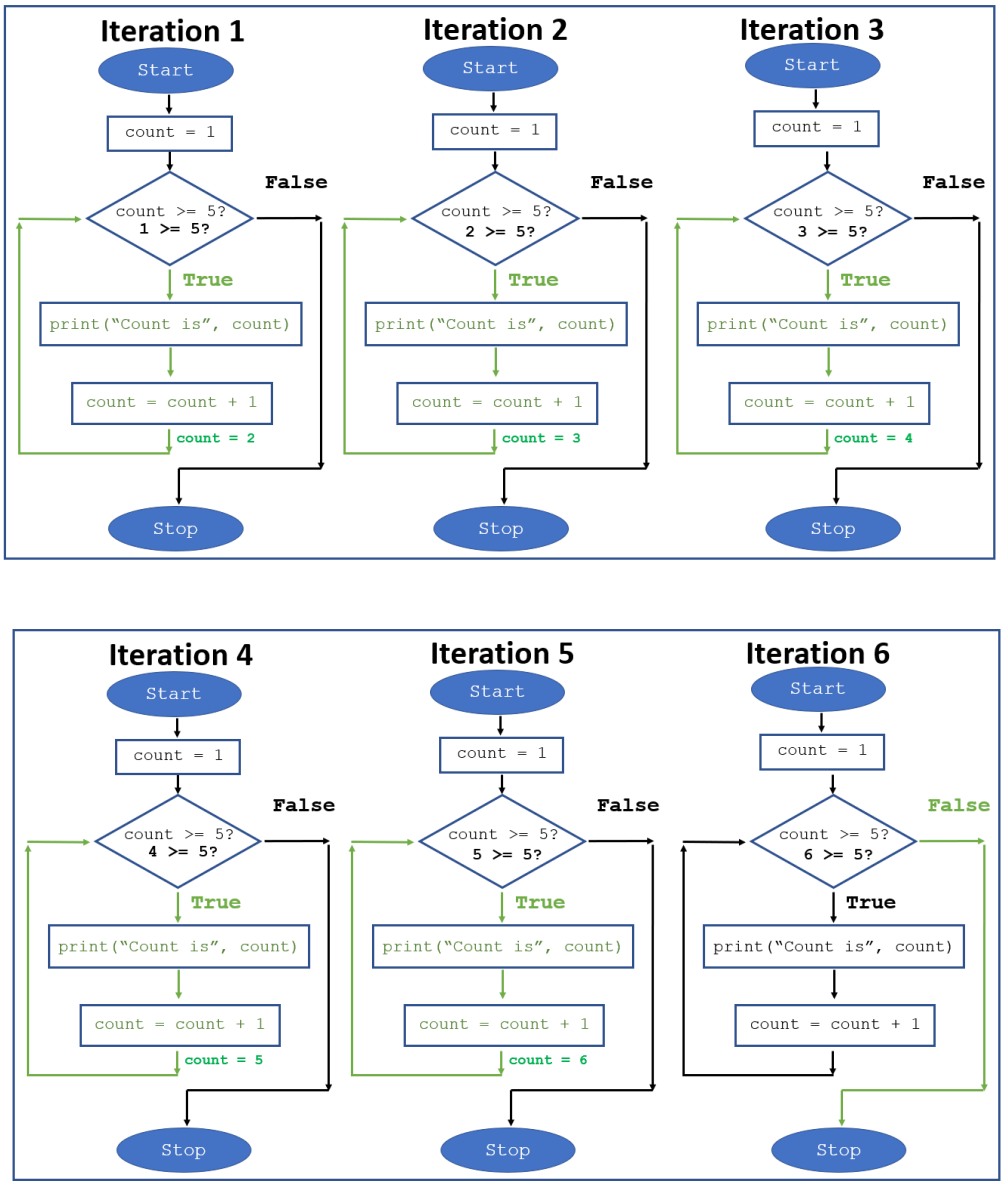

Figure 2.2: Flow chart for Example 1 at each iteration.

2.2.2 Loop condition evaluation

In a `while` loop, the condition is evaluated at the beginning of each iteration (including the first one). This means the loop will only run if the condition is true, and it will stop as soon as the condition becomes false. As a consequence:

If the condition is initially false, the loop body will not execute at all. The loop must be designed such that something in the loop (or outside it) will eventually make the condition false; otherwise, the loop would continue forever.

It's also worth noting that the condition is re-checked *before* each iteration, not after. This is why `while` loops are good for scenarios where you don't know in advance how many times the loop will execute – the loop will keep going until some condition changes. Typical use cases for `while` loops include waiting for a certain event or condition (e.g., waiting for user input, or looping until a

calculation converges, etc.). They are often used when the number of iterations is not known ahead of time.

As a general tip, if your loop is running too many or too few times, or not stopping at all, check these aspects: initial value, condition, and update logic. They are usually the source of the problem when a loop misbehaves.

2.3 The `for` Loop

Python's `for` loop is used to iterate over the elements of a sequence (such as a list, tuple, string, or other iterable). It is a **definite loop**, meaning it iterates a predetermined number of times (once for each element in the sequence). In Python, you typically use `for` loops when you know in advance the collection of items or the number of iterations you need to loop over.

Unlike `while`, the `for` loop in Python does not require explicit initialization, condition, and update for a loop counter – those details are handled internally when you iterate over a sequence. This makes `for` loops convenient and less error-prone for many common use cases, especially iterating through lists or other collections.

2.3.1 Iterating over sequences (list, tuple, string)

The syntax of a `for` loop is:

Syntax:

```
for <variable> in <sequence>:
    <body of loop>
```

Here, `<sequence>` is something that Python can iterate over (like a list, tuple, string, range, etc.), and `<variable>` is a name that will take on each value from the sequence in turn.

For example, if we have a list of names and we want to greet each name, we can do:

Example 2:

```
names = ["Alice", "Bob", "Cathy"]
for name in names:
    print("Hello,", name)
```

This loop will run three times, and each time the variable `name` will be set to the next element of the list. The output will be:

Output:

```
Hello, Alice
Hello, Bob
Hello, Cathy
```

We did not need to use an index or explicitly increment anything; the `for` loop handled that for us. You can use a `for` loop with any kind of sequence or iterable. For example:

Example 3: Iterating over a tuple

```
coordinates = (10, 20, 30)
for value in coordinates:
    print(value, end ="\n")
```

Output:

```
10
20
30
```

Example 4: Iterating over a a string

```
for ch in "PYTHON":
    print(ch, end ="\n")
```

Output:

```
P
Y
T
H
O
N
```

In the string example, the loop runs once for each character in the string. Similarly, you can iterate over other iterable objects (like sets, or even custom iterable classes) in the same way.

The Python `for` loop essentially abstracts the "initialize, condition, update" parts for you: it internally gets an iterator from the sequence and repeatedly fetches the next item until the sequence is exhausted.

2.3.2 Using `range()` for numeric loops

Often we want to loop a specific number of times or over a range of numeric values. Python provides the built-in `range()` function for this purpose. `range()` generates a sequence of numbers. Common ways to use it:

- `range(n)` generates numbers from 0 up to $n-1$ (inclusive). For example, `range(5)` produces the sequence 0, 1, 2, 3, 4 (five numbers, starting at 0). This is handy if you want to loop n times.
- `range(start, end)` generates numbers from `start` up to `end-1`. For example, `range(3, 7)` yields 3, 4, 5, 6.
- `range(start, end, step)` allows specifying a step (increment). For example, `range(2, 10, 2)` would generate even numbers 2, 4, 6, 8. The sequence starts at 2, stops before 10, and increments by 2 each time.

Here's an example of using `range()` in a loop:

Example 5:

```
for i in range(5):
    print(i, end = " ")
```

Output:

```
0 1 2 3 4
```

We started at 0 and went up to 4. If we wanted 1 through 5 instead, we could use `range(1, 6)`. For instance:

Example 6:

```
for i in range(1, 6):
    print(i, end = " ")
```

Output:

```
1 2 3 4 5
```

Notice that `range(1,6)` stops at 5 (not including 6). The end value in `range` is always non-inclusive, which is a common source of off-by-one confusion for beginners.

We can also loop with a step. Example:

Example 7:

```
for j in range(0, 10, 3):
    print(j, end = " ")
```

Output:

```
0 3 6 9
```

Negative steps are allowed too (e.g., `range(10, 0, -1)` would count down from 10 to 1).

Using `range()` is a very common idiom for loops where you need an index or just want to repeat something a fixed number of times.

2.3.3 Loop variables and scope

One important detail in Python is that the loop variable in a `for` loop remains defined even after the loop finishes. In other languages (like C or Java), a loop variable may be limited in scope to the loop itself, but in Python it will still exist after the loop, holding the last value it was assigned. For example:

Example 8:

```
colors = ["red", "green", "blue"]
for color in colors:
    print(color)
print("Last color was:", color)
```

Output:

```
red
green
blue
Last color was: blue
```

After the loop, the variable `color` is still accessible and it retains the value `"blue"` (the last item in the list). This happens because Python's `for` loop does not create a new inner scope for the loop variable. It's something to be mindful of. If you reuse the same variable name later, it will still hold that last value unless you assign it a new value.

If you don't need the loop variable after the loop, there's no harm, but if you do plan to use the variable name again, you might want to choose a different name for the loop to avoid confusion. It's also a common practice to use a placeholder variable name like _ (underscore) if the loop variable isn't actually used in the loop body, to signal that it's just a throwaway.

Note: List comprehensions (and other comprehensions) in Python 3 have their own scope for the loop variable, so they don't leak the variable to the surrounding scope. But normal `for` loops do not have this isolation.

2.3.4 Looping over dictionaries (`.keys()`, `.values()`, `.items()`)

Dictionaries are collections of key-value pairs, and looping through dictionaries is a common task. By default, iterating directly over a dictionary iterates over its keys. For example:

Example 9:

```
my_dict = {"a": 1, "b": 2, "c": 3}
for key in my_dict:
    print(key)
```

Output:

```
a
b
c
```

It's equivalent to looping over `my_dict.keys()`.

If you want to iterate over the values of the dictionary, you can use the `.values()` method:

Example 10:

```
for val in my_dict.values():
    print(val)
```

Output:

```
1
2
3
```

Often, we want to iterate over both the keys and values together. Python provides the `.items()` method for dictionaries, which returns an iterable of *(key, value)* tuples. We can unpack these pairs directly in a `for` loop:

Example 11:

```
for key, value in my_dict.items():
    print(key, "->", value)
```

Output:

```
a -> 1
b -> 2
c -> 3
```

Using `items()` is very handy because it gives you both pieces of information at once, without needing to do a separate lookup for the value inside the loop.

To summarize:

- `for key in dict:` iterates over keys.
- `for value in dict.values():` iterates over values.
- `for key, value in dict.items():` iterates over key-value pairs.

Looping over dictionaries in these ways is a common pattern for tasks like summing values, printing content, or any operation where you need to examine both keys and values.

2.4 Loop Control Statements

2.4.1 `break`: Using `break` to exit early

Sometimes we want to end a loop prematurely when a certain event happens. The `break` statement allows us to do that. When Python encounters a `break` inside a loop, it immediately exits the **innermost** loop, and the program continues after the loop.

Using `break` is a common way to avoid infinite loops or to stop the loop when a goal is achieved. For example, suppose we have a loop that could potentially run a long time, but we want to stop if a certain condition occurs:

Example:

```
i = 1
while i <= 10:
    if i == 5:
        break    # exit loop early when i equals 5
    print(i)
    i += 1
print("Loop ended")
```

Output:

```
1
2
3
4
Loop ended
```

In this code, the loop will print 1, 2, 3, 4 and then when `i` becomes 5, the `if` condition is `True` and `break` is executed. The `break` immediately terminates the while loop, so the numbers 5 through 10 are never printed. The program then continues with the `print("Loop ended")` after the loop. This technique is useful if, for instance, you are searching for something and want to stop once you've found it, or if a certain error condition arises and you need to halt the loop.

Important: If you have nested loops (a loop inside another loop), `break` will only break out of the innermost loop that contains it. It does not automatically terminate outer loops. If you need to break out of multiple levels of loops at once, you typically need to use additional logic (such as setting a flag, or raising an exception, etc.).

The `break` statement, as discussed, causes an immediate exit from the loop in which it appears. It works the same way in `for` loops as it does in `while` loops.

Example:

```
for n in range(1, 10):
    if ((n % 2) == 0):     # if n is even
        print("Found an even number:", n)
        break
    print("Checked", n, "- not even")
```

In this code, the loop is looking for an even number. It will check each number from 1 upward. The first time it finds an even number (which will be 2), it will print "Found an even number: 2" and then `break` will terminate the loop. The loop does not continue to 3, 4, etc., once the break executes. The output would be:

Output:

```
Checked 1 - not even
Found an even number: 2
```

After that, the loop stops completely.

Key points about `break`: - It exits only the innermost loop where it's called. - If you have outer loops, they will continue after the inner loop is broken out of. - It's often used with a condition (usually inside an `if`) to decide when to break.

Common uses of `break` include searching (stop when you found what you need) and safety escapes (stop the loop if something goes wrong or if a certain limit is reached).

2.4.2 `continue`: Using `continue` to skip to the next iteration

The `continue` statement is another loop control tool. When encountered inside a loop, it causes the loop to immediately skip the rest of the body for the current iteration, and then go back to the

top of the loop and start the next iteration. In other words, `continue` jumps to the next iteration of the loop, bypassing any code below it in the loop body for that iteration.

Here's a simple example to illustrate `continue`. Let's modify our previous loop to skip printing the number 3:

Example:

```
i = 0
while i < 5:
    i += 1
    if i == 3:
        continue   # skip the rest of this iteration when i is 3
    print(i)
print("Loop ended")
```

In this loop, we increment `i` at the start of each iteration. When `i` becomes 3, the `if` condition is true, and `continue` is executed. This causes the loop to immediately jump back to the top, re-check the condition `i < 5` (which will be true, since `i` is 3 at that point), and start the next iteration. Crucially, the `print(i)` line is skipped when `i == 3`. As a result, this code will print: 1, 2, 4, 5. The number 3 is not printed, because when `i` was 3 we used `continue` to skip that iteration.

Output:

```
1
2
4
5
Loop ended
```

Using `continue` is useful when you want to skip over certain data or iterations that you're not interested in processing, while still continuing the loop for the remaining items. For example, you might use `continue` to skip blank lines when processing lines from a file, or to skip invalid inputs in a loop without aborting the whole loop.

Caution: Just like with `while` loops in general, be careful when using `continue` that you don't accidentally create an infinite loop. If `continue` causes you to skip the part of the code that updates the loop variable, you might end up never reaching the loop's termination. In the example above, we update `i` *before* the `continue`, ensuring that `i` still increases even on the iterations we skip the print.

Python provides special statements that change the normal flow of loops. We have already introduced two of them: `break` and `continue`. In this section, we will summarize these and also discuss the loop `else` clause, which is a lesser-known feature of Python loops.

The `continue` statement functions the same in both `while` and `for` loops.
For example, using `continue` in a `for` loop:

Example:

```
for n in range(1, 6):
    if n == 3:
        continue
    print(n)
print("Loop ended")
```

This loop will print: `1, 2, 4, 5`. When `n` is 3, the `continue` statement is executed, which means the print statement is skipped for that iteration. The loop then moves on to `n = 4`.

Output:

```
1
2
4
5
Loop ended
```

This is similar to the while-loop example we saw earlier, but here it's in a `for` loop context. The utility of `continue` is the same: use it when you want to skip certain iterations but not abort the whole loop. Perhaps you're processing a list of records and want to skip any record that has missing data – a `continue` would be perfect for that (skip that record and continue with the next one).

2.4.3 `else` clause on loops: when and how it runs

Python has an interesting feature where loops (`for` or `while`) can have an `else` clause attached. The syntax is:

Syntax:

```
for item in collection:
    ...  # loop body
else:
    ...  # else-block
```

and similarly for a `while` loop:

Syntax:

```
while condition:
    ...  # loop body
else:
    ...  # else-block
```

The **loop `else`** clause executes after the loop finishes *only if the loop was not terminated by a `break`*. In other words:

- If the loop runs to completion (the `for` goes through all items, or the `while` condition becomes false naturally) without encountering a `break`, then the `else` block will execute.
- If the loop is exited early because of a `break` (or some other interruption like a `return` or exception), then the `else` block is skipped.

It might help to think of the `else` as meaning "no break occurred" or "loop completed normally." This is very useful in search scenarios, for example, when you want to distinguish between "found the item" and "didn't find the item."

Here's a practical example using a `for-else` loop to search for a number in a list:

Example:

```
nums = [2, 4, 6, 8]
target = 5
for num in nums:
    if num == target:
        print("Found", target)
        break
else:
    print(target, "not found in list")
```

In this code, we iterate over `nums`. If we find the `target` (5 in this case), we print a message and break out of the loop. If we go through the entire list and never break (meaning we never found the target), then the `else` clause will execute and print "5 not found in list."

Let's consider two scenarios:

- If `target` were 6, the loop would break when `num == 6`, and thus the `else` block would be skipped. The output would just be "Found 6".
- For `target = 5` (which isn't in the list), the loop completes without finding it, no break happens, and then the else block prints "5 not found in list."

The loop `else` clause can be used similarly with `while` loops—for example, if you were searching using a while loop and broke out when found, the else could handle the "not found" case.

This feature is somewhat unique to Python and can be confusing at first, because people might think the `else` works like it does with `if`. But in the context of loops, `else` doesn't mean "if the loop condition is false"; it specifically means "if the loop wasn't terminated by break." If the loop iterates zero times (for example, if the sequence was empty, or the while condition was false initially), that still counts as "normal completion" (no break), so the else would run in that case too.

In summary, use the loop `else` when you have a loop that may or may not break, and you want to run some code only in the case where it didn't break. A common pattern is searching: do something if found and break, else (after loop) handle the not-found case.

2.4.4 Common pitfalls (off-by-one errors, infinite loops)

When using `while` loops, there are a few common pitfalls to watch out for:

- **Off-by-one errors**: This happens when the loop runs one time too many or one time too few because of an incorrect condition or update. For example, you might use `<=` when you meant `<`, or initialize a counter incorrectly, causing the loop to iterate one extra or one fewer time than intended. Off-by-one errors are easy to introduce by mistake and can lead to incorrect results.

Always double-check the loop's start and end conditions. For instance, if you intended to loop 10 times, make sure your initialization and condition actually produce 10 iterations (and not 9 or 11).

- **Infinite loops**: An infinite loop occurs when the loop's condition never becomes false, so the loop never terminates:contentReference[oaicite:3]index=3. This often happens if you forget to update the loop variable inside the loop, or if you update it incorrectly. For example, if you accidentally omit `i += 1` in the earlier example, `i` would stay 1 forever and `while i <= 5` would never stop (this would print an endless stream of 1s until you forcefully stop the program). Infinite loops can also occur if the condition is always true (e.g., `while True: ...`) and there is no `break` inside to escape. To avoid accidental infinite loops, make sure that something in the loop eventually makes the condition false. If you do intentionally write an infinite loop (such as `while True:`) for a program that runs continuously, ensure you have a `break` or some other exit mechanism at the appropriate time.

- **Forgetting the update step**: This is related to infinite loops. In a `while` loop, it's your responsibility to update any variables involved in the condition. If you forget to change those variables, the condition may remain true forever. Always include the update (e.g., increment or decrement) in the right place inside the loop.