**Loops in Python**

Python provides two main loop constructs for repeating code: **for loops** and **while loops**. A for loop iterates over the *items of a sequence* (like a list, string, or range). This is often the simplest way to process each element in a collection. A while loop repeatedly executes a block of code *as long as a condition remains true*. Together with loop *control statements* (break, continue, pass) and optional else clauses, Python loops provide flexible control flow.

**for Loops**

A for loop runs a block of code once for each item in an iterable (such as a list, tuple, string, or range). It automatically stops when the iterable is exhausted. Common uses include:

* **Iterating over a list:**
* fruits = ['apple', 'banana', 'cherry']
* for fruit in fruits:
* print(fruit)
* # Output: apple banana cherry
* **Iterating over a string:** Each character of the string is treated as an item.
* for ch in "Hello":
* print(ch)
* # Output: H e l l o
* **Iterating over a range of numbers:** Using range() to generate a sequence of integers.
* for i in range(3):
* print(i)
* # Output: 0 1 2
* **Iterating with indexes:** You can combine range(len(...)) or use enumerate() (not shown) to get indexes and values.
* colors = ['red', 'green', 'blue']
* for index in range(len(colors)):
* print(index, colors[index])
* # Output: 0 red 1 green 2 blue

Using a for loop is often the “simplest and most common” way to process each element in a list or other collection. Because it directly accesses each element, it avoids manual index management and is idiomatic in Python.

**while Loops**

A while loop repeats a block of code as long as a given condition is true. It is useful when the number of iterations is not known beforehand or when you need to loop until a specific condition changes. For example:

* **Counting with while:**
* count = 0
* while count < 3:
* print(count)
* count += 1
* # Output: 0 1 2
* **Looping until a condition:** For instance, reading input until a sentinel value:
* line = input("Enter text (empty to stop): ")
* while line != "":
* print("You entered:", line)
* line = input("Enter text (empty to stop): ")
* **Infinite loop (until break):**
* while True:
* data = receive\_data()
* if not data:
* break # Exit when no more data
* process(data)

If you omit a condition, a while loop could run forever (until interrupted). For example, while True: print("Hi") will loop infinitely unless a break is used. Thus, always ensure the loop’s condition will eventually become false or use break to avoid infinite loops.

**Loop Control Statements (break, continue, pass)**

Inside any loop, Python supports three control statements:

* **break:** Immediately exits the enclosing loop. Execution resumes at the statement following the loop.
* for n in range(1, 6):
* if n == 3:
* break
* print(n)
* # Output: 1 2

Here, when n == 3, the break stops the loop entirely.

* **continue:** Immediately jumps to the next iteration of the loop, skipping the remaining code in the loop body for the current iteration.
* for n in range(1, 6):
* if n % 2 == 0:
* continue
* print(n)
* # Output: 1 3 5

In this example, continue skips printing even numbers.

* **pass:** Does nothing; it’s a no-op placeholder where a statement is syntactically required but no action is needed. For example:
* while True:
* pass # Busy-wait or placeholder

or defining an empty class or function:

class Empty:

pass

The break and continue statements are often used to fine-tune loop behavior (e.g. exiting early or skipping items). The pass statement, by contrast, has no effect on loop flow; it simply satisfies the syntax when you need an empty block.

**else Clause with Loops**

Python loops can have an optional else clause that executes **after** the loop finishes normally (i.e. without encountering a break). This applies to both for and while loops:

for i in range(3):

print(i)

else:

print("Loop finished without break")

# Output: 0 1 2 Loop finished without break

for i in range(5):

if i == 10:

break

else:

print("No break occurred")

In the second example, the else block does **not** run, because the loop terminated via break. In general, the else block runs only if the loop ends by exhausting the iterable (for for loops) or the condition becoming false (for while loops), and *not* by a break. This feature is useful for post-loop logic like searching: for example, you can loop to find an item and use else to handle “not found” cases.

**Nested Loops**

A **nested loop** is a loop inside another loop. Python allows nesting any combination of for and while loops. The “inner” loop runs completely every time the “outer” loop runs one iteration. For example:

# Nested for loops

for i in [1, 2]:

for j in [4, 5]:

print(i, j)

# Output:

# 1 4

# 1 5

# 2 4

# 2 5

# Nested while loops

x = [1, 2]

y = [4, 5]

i = 0

while i < len(x):

j = 0

while j < len(y):

print(x[i], y[j])

j += 1

i += 1

Nested loops are useful for tasks like iterating over multi-dimensional data (matrices, grids) or producing combinations of items. However, they can be less efficient: each additional nested loop multiplies the total number of iterations (e.g. two nested loops often give *O(n²)* complexity). Use them only when necessary, and consider alternative algorithms for large data.

**Comparison: for vs while**

| **Aspect** | **for loop** | **while loop** |
| --- | --- | --- |
| **Syntax** | for var in iterable:…body… | while condition:…body… |
| **Use Case** | Iterating over a known sequence of items (list, string, tuple, etc.). | Repeating until a condition changes, especially when number of iterations isn’t fixed. |
| **Termination** | Stops after the iterable is exhausted, or by break. | Stops when the condition becomes false, or by break. |
| **Typical Usage** | Best for processing each element of a collection. E.g., “for each element in a list”. | Best for loops that run until some condition occurs (e.g. user input, complex logic). |
| **Performance** | Often more efficient for fixed sequences, since Python handles iteration in C and knows how many steps are needed. | Can be equally efficient in simple cases, but may incur more overhead if updating and checking conditions each iteration. |
| **When to Use** | Use when you *have* an iterable or fixed range of values. | Use when the loop must continue until a condition (not necessarily a counter) changes. |