While-Loops

Table of contents

Outline . . . 1
Learning objectives . . . 1
Motivation . . . 2
The two definitions . . . 2
Accumulator pattern . . . 2
When the condition never changes . . . 3
Exercise: guess the number . . . 4
Exercise: guess the dice roll . . . 5
Further exercises . . . 5
Summary . . . 6
Next lecture . . . 6

See csse1001[1] for course logistics — this note covers Lecture 4A's technical content.

Outline

- Why we need to repeat code, possibly indefinitely
- The while-loop
- Simulating a do-while, and reading user input

Learning objectives

1. Loops are a control structure for repeating code.
2. A while-loop repeats code while a condition holds — and the condition is tested *before* each repetition.
3. `break` leaves a loop immediately; `while True` + `break` is how Python writes a do-while.
4. `input()` reads from the keyboard; `random.randint` generates random integers.

[1] courses/csse1001/csse1001.pdf

Motivation

There are (at least) two scenarios where *repeating possibly indefinitely* is **necessary**:

1. prompting the user for valid input, and
2. playing a random game (i.e. guessing dice rolls).

The common feature is that neither has a repetition count you could work out in advance, which is what a while-loop buys you over the counted `for`-loop of 2025-08-25-for-loops[2].

The two definitions

> **Definition (Loop).** A *loop* is a *control structure* that repeats code that belongs to it.
>
> **Definition (While-Loop).** A *while-loop* is a *control structure* that repeats code while some *condition is satisfied*.

In Python that is written

```
while <condition>:
    <code>
```

and it behaves like this:

Reading Figure 1, the only thing separating a while-loop from an `if` is the back-edge from the body to the diamond. See python-while-loops[3] for the full reference on loops, while-loops, `break`, augmented assignment operators (`+=`, `*=`, `/=`, `%=`), simulating a do-while, `input()`, and `random.randint`.

Accumulator pattern

The loop variable is updated inside the body so that the condition eventually fails:

```
>>> x = 0
>>> while x < 10:
...     x = x + 1          # accumulator
>>> x
10
```

[2] courses/csse1001/2025-08-25-for-loops.pdf
[3] courses/csse1001/python-while-loops.pdf

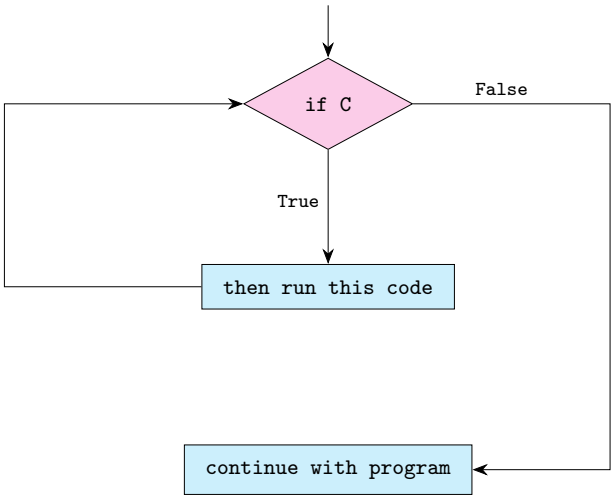

Figure 1: The while-loop as a flowchart. The condition `C` is tested *first*: while it is `True` control drops into the body and then loops straight back to the test, and only when it is `False` does control fall through to the rest of the program. The pink diamond is the decision and the blue boxes are code.

Using an augmented assignment operator (see python-while-loops[4]) makes the accumulation more concise — `x += 1` is exactly `x = x + 1`:

```
>>> x = 0
>>> while x < 10:
...     x += 1              # accumulator
>>> x
10
```

The loop stops at exactly `10`, not `9`: the test `x < 10` is re-checked after the body has already pushed `x` to `10`.

When the condition never changes

If the condition is never falsified the loop never stops:

```
>>> x = 0
>>> while True:
...     print(x)
```

[4] courses/csse1001/python-while-loops.pdf

```
...     x += 1
0
1
2
3 ...
```

There is usually a key-stroke, typically `ctrl+c`, that terminates a loop — it is a good idea to learn what it is in your IDE.

The mirror case is a condition that is false to begin with. The body then never runs at all, because the test comes first:

```
>>> x = 0
>>> while False:
...     print(x)
...     x += 1
```

Nothing prints.

Exercise: guess the number

`x = input()` waits for input from the keyboard and then assigns it to `x` with `type(x) is str` — so a numeric guess has to be converted with `int(...)`.

Write a function `foo(target: int) -> int` that prompts the user to input a number until the user guesses some (secret) target. For each guess the function should print `Too high`, `Too low`, or `That's it!` depending on the guess. *Return the number of guesses required.*

```
>>> def foo(target: int) -> int:
...     number_of_guesses = 0
...     while True:
...         guess = int(input("Guess: "))
...         number_of_guesses += 1
...         if guess > target:
...             print("Too high")
...         elif guess < target:
...             print("Too low")
...         else:
...             print("That's it!")
...             return number_of_guesses
>>> foo(10)
Guess:
```

The loop is deliberately `while True`: there is no condition available at the top of the loop that says "the user has not guessed yet", because the guess does not exist until the body has run. The exit is the `return` in the `else` branch, which leaves the function (and so the loop) outright.

Exercise: guess the dice roll

Write a *function* called `bar() -> None` that prompts the user to predict the result of a six-sided dice roll. Repeat the prompt until the user predicts correctly.

Random number generation is handled by an *external library*. It is **not** built-in and therefore must be imported (see python-while-loops[5] for both import forms):

```
>>> from random import randint
>>> def bar() -> None:
...     """
...     Prompt user to predict the result of a six sided
...     dice roll until the guess is correct.
...     """
...     while True:
...         guess = int(input("Guess in [1..6]: "))
...         roll  = randint(1, 6)
...         if roll == guess:
...             break
...     return
>>> bar()
Guess in [1..6]:
```

Here the exit is a `break` rather than a `return`, because the function has nothing to hand back. The die is re-rolled every iteration, so the user is predicting a fresh roll each time rather than searching for one fixed hidden number.

Further exercises

Open exercises:

- Write a function that prints the result of a six-sided dice roll until a six is rolled, and returns the number of rolls required.
- Write a function that accumulates the result of dice rolls until some threshold is met/passed, and returns the number of rolls required.

[5] courses/csse1001/python-while-loops.pdf

- Develop functions for adding and multiplying positive integers together, restricting ourselves to exactly two arithmetic operations — `succ` (adding one) and `pred` (subtracting one). The two building blocks are:

```
def succ(x: int) -> int:
    return x+1

def pred(x: int) -> int:
    return x-1
```

Both the add and the multiply function are accumulators driven by a while-loop.

Summary

The *while-loop* is a control structure for repeating code a possibly infinite number of times.

Next lecture

2025-08-18-non-primitive-data[6] — lists and dictionaries.

[6] courses/csse1001/2025-08-18-non-primitive-data.pdf