Lecture 4: Loops and Iteration

INF 605 · 2026-09-03 and 2026-09-08

Two class meetings, one lecture.

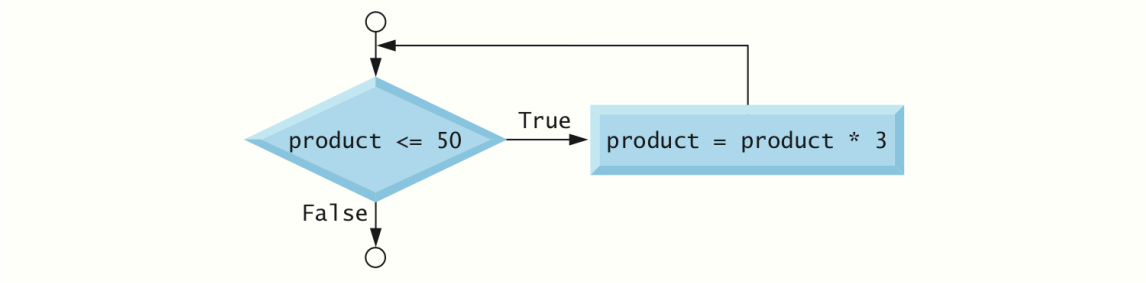
Open in Colab → **File → Save a copy in Drive**. You cannot edit the class original.

Start-of-class quiz

Type answers in the Canvas attendance quiz (empty boxes only). The questions stay here.

1. Last class was **Decision Making in Python**. Name one thing we actually ran or wrote.
2. Last class, which name or API should you still remember for today's first demo?

A flowchart for `while`



Textbook flowchart (Deitel): while the condition is True, run the body and check again.

The `while` statement

A `while` loop repeats its indented block as long as the condition is `True`. Last class, `if` asked the question once. `while` asks it again after each pass.

A countdown: the condition is `count > 0`. Each pass prints and subtracts 1.

In [5]:
```
count = 5
print("countdown:")
while count > 0:
    print(count)
    count = count - 1
print("done")
```

```
countdown:
5
4
3
2
1
done
```

5, 4, 3, 2, 1 print. When `count` is 0, `count > 0` is False, so `done` prints.

Textbook flowchart (Deitel): while `product <= 50` is True, multiply `product` by 3. When it is False, leave the loop.

In [7]:
```
# FIGURE: while_statement.png
print("while flowchart: True runs the body and checks again; False leaves.")
```

```
while flowchart: True runs the body and checks again; False leaves.
```

Textbook example. `product` starts at 3.

In [9]:
```
product = 3
while product <= 50:
    product = product * 3
print(product)
```

```
81
```

The values are 3, then 9, then 27, then 81. `81 <= 50` is False, so the loop stops and `print` shows 81.

To prevent an infinite loop, something in the `while` suite must change `product`'s value, so the condition eventually becomes False.

Adding numbers 1 through 5. `total` starts at 0. Each pass adds the current number.

In [12]:
```
total = 0
number = 1
while number <= 5:
    total = total + number
    print("added", number, "total", total)
    number = number + 1
print("final", total)
```

```
added 1 total 1
added 2 total 3
added 3 total 6
added 4 total 10
added 5 total 15
final 15
```

1+2+3+4+5 is 15. Both `total` and `number` change each pass.

Augmented assignments

`total += number` is the same as `total = total + number`. The other abbreviations work the same way.

Augmented assignment	Sample	Meaning
`+=`	`c += 7`	`c = c + 7`
`-=`	`d -= 4`	`d = d - 4`
`*=`	`e *= 5`	`e = e * 5`
`/=`	`g /= 2`	`g = g / 2`
`//=`	`g //= 2`	`g = g // 2`
`%=`	`h %= 9`	`h = h % 9`

In [15]:
```
total = 0
number = 1
while number <= 5:
    total += number
    number += 1
print(total)
```

15

Same 15. `total += number` adds. `number += 1` moves to the next integer.

Exercise (while)

Count down from 10 to 1, then print `Happy New Year!`. Use `while` and `-=`.

In [ ]:
```
# Exercise — your attempt
count = 10
print("New Year countdown:")
# while count > 0:
#     print the current count
#     count -= 1
print("Happy New Year!")
```

Solution

In [20]:
```
# Solution
count = 10
print("New Year countdown:")
while count > 0:
    print(count)
    count -= 1
print("Happy New Year!")
```

```
New Year countdown:
10
9
8
7
6
5
4
3
2
1
Happy New Year!
```

The `for` statement

`for` repeats for each item in a sequence. You do not write the condition yourself.

A string is a sequence of individual characters.

Textbook flowchart (Deitel): if there are more items, assign the next one to the target variable and run the suite. If not, leave.

In [22]:
```
# FIGURE: for_statement.png
print("for flowchart: more items -> assign next item -> run the body.")
```

```
for flowchart: more items -> assign next item -> run the body.
```

Textbook example. Target variable: `character`. Sequence: `'Programming'`.

In [24]:
```
for character in "Programming":
    print(character, end="  ")
print()
```

```
P  r  o  g  r  a  m  m  i  n  g
```

Upon entering the `for` loop, Python assigns the `P` in `Programming` to `character`. After the suite, it assigns `r`, then runs the suite again. It continues while there are more items.

`end="  "` is a keyword argument to `print`. It puts two spaces after each character instead of a new line.

In [26]:
```
print(10, 20, 30)
print(10, 20, 30, sep=", ")
```

```
10 20 30
10, 20, 30
```

`sep` is the string between items. Default is a space. `sep=", "` puts a comma and a space.

Function `range`

`range` is a sequence of integers. `range(stop)` starts at 0 and goes up to, but not including, `stop`.

In [29]:
```
for counter in range(10):
    print(counter, end=" ")
print()
```

```
0 1 2 3 4 5 6 7 8 9
```

0 through 9. 10 is not included. Assuming `range(10)` contains 10 is an off-by-one error.

In [31]:
```
for number in range(5, 10):
    print(number, end=" ")
print()
```

```
5 6 7 8 9
```

`range(5, 10)` is 5, 6, 7, 8, 9. Still exclusive of the stop value.

In [33]:
```
for number in range(0, 10, 2):
    print(number, end=" ")
print()
```

```
0 2 4 6 8
```

The third argument is the step. 0, 2, 4, 6, 8. 10 is not included.

In [35]:
```
for number in range(10, 0, -1):
    print(number, end=" ")
print()
```

```
10 9 8 7 6 5 4 3 2 1
```

A negative step counts down. 10 down to 1. 0 is the stop, so it is not printed.

Exercise (range)

Print the even numbers from 2 through 20 (include 20). Use `for` and `range`.

In [ ]:
```
# Exercise — your attempt
# for n in range(?, ?, ?):
#     print n
result = None
print(result)
```

Solution

In [40]:
```
# Solution
for n in range(2, 21, 2):
    print(n)
```

```
2
4
6
8
10
12
14
16
18
20
```

`for` over a list

A list is a comma-separated collection in square brackets. `for` takes one item at a time.

In [42]:
```
total = 0
for number in [2, -3, 0, 17, 9]:
    total = total + number
print(total)
```

```
25
```

2 + -3 + 0 + 17 + 9 is 25.

In [44]:
```
names = ["Ada", "Grace", "Alan"]
for name in names:
    print("hello", name)
```

```
hello Ada
hello Grace
hello Alan
```

Each pass binds `name` to the next string. No index is required if you only need the value.

`enumerate` gives the index and the value together. `start=1` numbers from 1 instead of 0.

In [47]:
```
names = ["Ada", "Grace", "Alan"]
for index, name in enumerate(names, start=1):
    print(index, name)
```

```
1 Ada
2 Grace
3 Alan
```

`(1, "Ada")`, then `(2, "Grace")`, then `(3, "Alan")`.

Exercise (lists)

Print this shopping list numbered from 1. Use `enumerate`.

In [ ]:
```
# Exercise — your attempt
items = ["milk", "eggs", "bread"]
# for i, item in enumerate(items, start=1):
#     print i and item
```

Solution

In [52]:
```
# Solution
items = ["milk", "eggs", "bread"]
for i, item in enumerate(items, start=1):
    print(i, item)
```

```
1 milk
2 eggs
3 bread
```

`break` and `continue`

`break` leaves the loop immediately. The rest of the items are not processed.

In [54]:
```
for number in range(100):
    if number == 10:
        break
    print(number, end=" ")
print()
```

```
0 1 2 3 4 5 6 7 8 9
```

0 through 9 print. When `number` is 10, `break` exits. 10 and the rest never print.

`continue` skips the rest of this pass and goes to the next item. In a `while`, Python then tests the condition again.

In [57]:
```
for number in range(10):
    if number == 5:
        continue
    print(number, end=" ")
print()
```

```
0 1 2 3 4 6 7 8 9
```

0 through 9 except 5. `continue` skips the `print` for 5, then the loop takes 6.

Exercise (continue)

Print only the positive numbers. Use `continue` when a value is `<= 0`.

In [ ]:
```
# Exercise — your attempt
nums = [4, -1, 7, 0, 3]
# for n in nums:
#     if n <= 0:
#         continue
#     print n
```

Solution

In [62]:
```
# Solution
nums = [4, -1, 7, 0, 3]
for n in nums:
    if n <= 0:
        continue
    print(n)
```

```
4
7
3
```

Nested `for`

A `for` inside another `for`. Read the outer loop first, then the inner one. The inner loop runs all the way through for each outer value.

In [64]:
```
for row in range(1, 4):
    for col in range(1, 4):
        print(row * col, end=" ")
    print()
```

```
1 2 3
2 4 6
3 6 9
```

Outer `row` is 1, then 2, then 3. Inner `col` is 1, 2, 3 on every row. `print()` with no arguments ends the line so the next row starts fresh.

Exercise (nested)

Print this 3×3 grid of coordinates. Use two `for` loops.

```
(0,0) (0,1) (0,2)
(1,0) (1,1) (1,2)
(2,0) (2,1) (2,2)
```

In [ ]:
```
# Exercise — your attempt
# for r in range(3):
#     for c in range(3):
#         print the pair, end with a space
#     print()
```

Solution

In [69]:
```
# Solution
for r in range(3):
    for c in range(3):
        print(f"({r},{c})", end=" ")
    print()
```

```
(0,0) (0,1) (0,2)
(1,0) (1,1) (1,2)
(2,0) (2,1) (2,2)
```

Sequence-controlled class average

A **requirements statement** describes what a program should do, not how.

> A class of ten students took a quiz. Their grades (integers in the range 0–100) are 98, 76, 71, 87, 83, 90, 57, 79, 82, 94. Determine the class average on the quiz.

The algorithm must keep a running total, count the grades, then divide.

Pseudocode:

Set total to zero
Set grade counter to zero
Set grades to a list of the ten grades

For each grade in the grades list:

> *Add the grade to the total*
> *Add one to the grade counter*

Set the class average to the total divided by the number of grades
Display the class average

Variables for totaling and counting start at zero.

In [71]:
```
# initialization phase
total = 0
grade_counter = 0
grades = [98, 76, 71, 87, 83, 90, 57, 79, 82, 94]

# processing phase
for grade in grades:
    total += grade
    grade_counter += 1

# termination phase
```

```
average = total / grade_counter
print(f"Class average is {average}")
```

```
Class average is 81.7
```

Three phases: initialize the variables, process every grade, then compute and display the result. `grade_counter` is 10, not hard-coded in the division, so the same loop works if the list is longer or shorter.

Exercise (class average)

`scores` is a list of four quiz scores. Print the class average with an f-string.

In [ ]:
```
# Exercise — your attempt
scores = [80, 90, 70, 100]
total = 0
# for s in scores:
#     total += s
# average = total / len(scores)
# print the average
```

Solution

In [76]:
```
# Solution
scores = [80, 90, 70, 100]
total = 0
for s in scores:
    total += s
average = total / len(scores)
print(f"Class average is {average}")
```

```
Class average is 85.0
```

Formatted strings

An **f-string** inserts values into a string. Put `f` before the opening quote. `{name}` is replaced by the value of `name`.

In [78]:
```
name = "Ada"
n = 3
print(f"{name} wrote {n} notes.")
print(f"The sum of 2 and 5 is {2 + 5}.")
```

```
Ada wrote 3 notes.
The sum of 2 and 5 is 7.
```

Follow a replacement with a colon and a **format specifier**. `.2f` means a floating-point number with two digits after the decimal point.

In [80]:
```
pi = 3.1415926535
print(f"pi rounded to two places is {pi:.2f}")
print(f"Class average is {81.7:.2f}")
```

```
pi rounded to two places is 3.14
Class average is 81.70
```

Sentinel-controlled iteration

Sometimes you do not know how many grades there will be. A **sentinel** (also called a signal, dummy, or flag) marks the end of the data. It must not be a legal grade. Here the sentinel is `-1`.

Read a value, then loop while it is not the sentinel. Add the grade, then read the next one. Do **not** add `-1` into the total.

If the user enters the sentinel first, `grade_counter` stays 0. Do not divide by zero — print that no grades were entered.

This notebook uses a list that already ends with `-1`, so it can run without typing. The same loop works with `input` in a live session.

In [82]:
```
# initialization
total = 0
grade_counter = 0
entered = [98, 76, 71, 87, -1]  # -1 is the sentinel, not a grade

# processing: look at the first value, then loop
index = 0
grade = entered[index]
index += 1
while grade != -1:
    total += grade
    grade_counter += 1
    grade = entered[index]
    index += 1

# termination
if grade_counter != 0:
    average = total / grade_counter
    print(f"Class average is {average:.2f}")
else:
    print("No grades were entered")
```

```
Class average is 83.00
```

98, 76, 71, 87 go into the total. `-1` stops the loop and is not added. `{average:.2f}` prints two decimal places.

In a live program the same test is `while grade != -1`, and each new grade comes from `int(input('Enter grade, -1 to end: '))`.

Exercise (sentinel)

`entered` ends with the sentinel `-1`. Print how many real grades were counted. Do not count `-1`.

In [ ]:
```
# Exercise — your attempt
entered = [10, 20, 30, -1]
count = 0
# walk the list; stop at -1; count the others
print("count:", count)
```

Solution

In [87]:
```
# Solution
entered = [10, 20, 30, -1]
count = 0
for g in entered:
    if g == -1:
        break
    count += 1
print("count:", count)
```

```
count: 3
```

Comprehensive exercises

Comprehensive exercise 1

Add the integers 1 through 100. Store the result in `total` and print it. Use `for`, `range`, and `+=`.

In [89]:
```
total = 0
# for n in range(1, 101):
#     total += n
print("sum:", total)
```

```
sum: 0
```

Solution

In [91]:
```
total = 0
for n in range(1, 101):
    total += n
print("sum:", total)
```

```
sum: 5050
```

Comprehensive exercise 2

`grades` is a list of scores. Print the first score that is at least 70, then stop. If none pass, print `none`.

In [93]:
```
grades = [55, 62, 71, 88]
found = False
# for g in grades:
#     if g >= 70:
#         print g, set found, break
# after the loop, if not found: print none
```

Solution

In [95]:
```
grades = [55, 62, 71, 88]
found = False
for g in grades:
    if g >= 70:
        print("first passing:", g)
        found = True
        break
if not found:
    print("none")
```

```
first passing: 71
```

Takeaways

1. `while` repeats while the condition is True. Something in the body must change that condition.
2. `for` repeats for each item in a sequence. A string, a list, and `range(...)` are sequences.
3. `range(stop)` is 0 through `stop - 1`. The stop value is not included.
4. `+=` means add and store. `break` leaves the loop. `continue` skips this pass.
5. Nested `for`: the inner loop finishes for every outer value.
6. A running total plus a counter gives an average. A sentinel such as `-1` ends input without being added.
7. f-strings insert values; `{x:.2f}` prints two decimal places.
8. Next class: lists.