15-110 Principles of Computing – S19

Lecture 9:
Iteration 2

Teacher:
Gianni A. Di Caro

جامعة كارنيجي ميلون في قطر
Carnegie Mellon University Qatar

Loops so far: definite loops, `for` construct

✓ Repeat a set of actions a **defined number of times** (at *most*)

```
for variable in sequence:
    actions
```

sequence
- tuple
- list

`len(sequence)`
iterations (at most)

variable
- *loop index*: each time the variable is set to the value of the next item in the sequence

Loops so far: definite loops, `for` construct

✓ Repeat a set of actions a **defined number of times** (at *most*)

```
for variable in sequence:
    actions
```

sequence
- tuple
- list

`len(sequence)`
iterations (at most)

variable
loop index: each time the variable is set to the value of the next item in the sequence

```
sum = 0
for n in [6, 3, 5, 7]:
    sum += n
    print(sum, n)
average = sum / 4
```

Loops so far: definite loops, `for` construct

✓ Repeat a set of actions a **defined number of times** (at *most*)

```
for variable in sequence:
    actions
```

sequence
- tuple
- list

`len(sequence)`
iterations (at most)

variable
loop index: each time the variable is set to the value of the next item in the sequence

```
sum = 0
for n in [6, 3, 5, 7]:
    sum += n
    print(sum, n)
average = sum / 4
```

Is equivalent to:

Loops so far: definite loops, `for` construct

✓ Repeat a set of actions a **defined number of times** (at *most*)

sequence
- tuple
- list

`len(sequence)`
iterations (at most)

variable
loop index: each time the variable is set to the value of the next item in the sequence

```
for variable in sequence:
    actions
```

```
sum = 0
sequence = [6, 3, 5, 7]
```

```
sum = 0
for n in [6, 3, 5, 7]:
    sum += n
    print(sum, n)
average = sum / 4
```

Is equivalent to:

Loops so far: definite loops, `for` construct

✓ Repeat a set of actions a **defined number of times** (at *most*)

sequence
- tuple
- list

`len(sequence)`
iterations (at most)

variable
loop index: each time the variable is set to the value of the next item in the sequence

```
for variable in sequence:
    actions
```

```
sum = 0
sequence = [6, 3, 5, 7]
n = sequence[0]
sum += n
print(sum, n)
```

Is equivalent to:

```
sum = 0
for n in [6, 3, 5, 7]:
    sum += n
    print(sum, n)
average = sum / 4
```

Loops so far: definite loops, `for` construct

✓ Repeat a set of actions a **defined number of times** (at *most*)

`for` *`variable`* `in` *`sequence`*`:`
    `actions`

`sequence`
- tuple
- list

`len(sequence)`
iterations (at most)

`variable`
- *loop index*: each time the variable is set to the value of the next item in the sequence

```
sum = 0
sequence = [6, 3, 5, 7]
```

Iteration 1
```
n = sequence[0]
sum += n
print(sum, n)
```

```
sum = 0
for n in [6, 3, 5, 7]:
    sum += n
    print(sum, n)
average = sum / 4
```

Is equivalent to:

Loops so far: definite loops, `for` construct

✓ Repeat a set of actions a **defined number of times** (at *most*)

`for variable in sequence:`
`    actions`

sequence
- tuple
- list

`len(sequence)` **iterations** (at most)

variable
- *loop index*: each time the variable is set to the value of the next item in the sequence

```
sum = 0
sequence = [6, 3, 5, 7]
```

Iteration 1
```
n = sequence[0]
sum += n
print(sum, n)
```

```
n = sequence[1]
sum += n
print(sum, n)
```

Is equivalent to:

```
sum = 0
for n in [6, 3, 5, 7]:
    sum += n
    print(sum, n)
average = sum / 4
```

Loops so far: definite loops, `for` construct

✓ Repeat a set of actions a **defined number of times** (at *most*)

`for` *`variable`* `in` *`sequence`*`:`

    actions

sequence: tuple, list

`len(sequence)` **iterations** (at most)

variable: *loop index*: each time the variable is set to the value of the next item in the sequence

```
sum = 0
sequence = [6, 3, 5, 7]
```

Iteration 1:

```
n = sequence[0]
sum += n
print(sum, n)
```

Iteration 2:

```
n = sequence[1]
sum += n
print(sum, n)
```

Is equivalent to:

```
sum = 0
for n in [6, 3, 5, 7]:
    sum += n
    print(sum, n)
average = sum / 4
```

Loops so far: definite loops, `for` construct

✓ Repeat a set of actions a **defined number of times** (at *most*)

`for` *`variable`* `in` *`sequence`*`:`

    actions

sequence: tuple, list

`len(sequence)` **iterations** (at most)

variable: *loop index*: each time the variable is set to the value of the next item in the sequence

```
sum = 0
for n in [6, 3, 5, 7]:
    sum += n
    print(sum, n)
average = sum / 4
```

Is equivalent to:

```
sum = 0
sequence = [6, 3, 5, 7]
```

Iteration 1:

```
n = sequence[0]
sum += n
print(sum, n)
```

Iteration 2:

```
n = sequence[1]
sum += n
print(sum, n)
```

```
n = sequence[2]
sum += n
print(sum, n)
```

Loops so far: definite loops, `for` construct

✓ Repeat a set of actions a **defined number of times** (at *most*)

`for` *`variable`* `in` *`sequence`*`:`

    actions

sequence: tuple, list

`len(sequence)` **iterations** (at most)

variable: *loop index*: each time the variable is set to the value of the next item in the sequence

```
sum = 0
for n in [6, 3, 5, 7]:
    sum += n
    print(sum, n)
average = sum / 4
```

Is equivalent to:

```
sum = 0
sequence = [6, 3, 5, 7]
```

Iteration 1:
```
n = sequence[0]
sum += n
print(sum, n)
```

Iteration 2:
```
n = sequence[1]
sum += n
print(sum, n)
```

Iteration 3:
```
n = sequence[2]
sum += n
print(sum, n)
```

Loops so far: definite loops, `for` construct

✓ Repeat a set of actions a **defined number of times** (at *most*)

`for` *variable* `in` *sequence*:
    actions

sequence: tuple, list

`len(sequence)` **iterations** (at most)

variable: *loop index*: each time the variable is set to the value of the next item in the sequence

```
sum = 0
for n in [6, 3, 5, 7]:
    sum += n
    print(sum, n)
average = sum / 4
```

Is equivalent to:

```
sum = 0
sequence = [6, 3, 5, 7]
```

Iteration 1:
```
n = sequence[0]
sum += n
print(sum, n)
```

Iteration 2:
```
n = sequence[1]
sum += n
print(sum, n)
```

Iteration 3:
```
n = sequence[2]
sum += n
print(sum, n)
```

```
n = sequence[3]
sum += n
print(sum, n)
```

Loops so far: definite loops, `for` construct

✓ Repeat a set of actions a **defined number of times** (at *most*)

sequence: tuple, list

`len(sequence)` **iterations** (at most)

variable: *loop index*: each time the variable is set to the value of the next item in the sequence

```
sum = 0
for n in [6, 3, 5, 7]:
    sum += n
    print(sum, n)
average = sum / 4
```

Is equivalent to:

`for variable in sequence:`

`actions`

```
sum = 0
sequence = [6, 3, 5, 7]
```

Iteration 1:
```
n = sequence[0]
sum += n
print(sum, n)
```

Iteration 2:
```
n = sequence[1]
sum += n
print(sum, n)
```

Iteration 3:
```
n = sequence[2]
sum += n
print(sum, n)
```

Iteration 4:
```
n = sequence[3]
sum += n
print(sum, n)
```

Loops so far: definite loops, `for` construct

✓ Repeat a set of actions a **defined number of times** (at *most*)

`for` *variable* `in` *sequence*:
    actions

sequence: tuple, list

`len(sequence)` **iterations** (at most)

variable: *loop index*: each time the variable is set to the value of the next item in the sequence

```
sum = 0
for n in [6, 3, 5, 7]:
    sum += n
    print(sum, n)
average = sum / 4
```

Is equivalent to:

```
sum = 0
sequence = [6, 3, 5, 7]
```

Iteration 1:
```
n = sequence[0]
sum += n
print(sum, n)
```

Iteration 2:
```
n = sequence[1]
sum += n
print(sum, n)
```

Iteration 3:
```
n = sequence[2]
sum += n
print(sum, n)
```

Iteration 4:
```
n = sequence[3]
sum += n
print(sum, n)
```

```
average = sum / 4
```

Loops so far: definite loops, `for` construct

✓ Repeat a set of actions a **defined number of times** (at *most*)

`for` *variable* `in` *sequence*:
    actions

sequence: tuple, list

`len(sequence)` **iterations** (at most)

variable: *loop index*: each time the variable is set to the value of the next item in the sequence

```
sum = 0
for n in [6, 3, 5, 7]:
    sum += n
    print(sum, n)
average = sum / 4
```

Is equivalent to:

```
sum = 0
sequence = [6, 3, 5, 7]
```

Iteration 1:
```
n = sequence[0]
sum += n
print(sum, n)
```
→ 6 6

Iteration 2:
```
n = sequence[1]
sum += n
print(sum, n)
```

Iteration 3:
```
n = sequence[2]
sum += n
print(sum, n)
```

Iteration 4:
```
n = sequence[3]
sum += n
print(sum, n)
```

```
average = sum / 4
```

Loops so far: definite loops, `for` construct

✓ Repeat a set of actions a **defined number of times** (at *most*)

`for` *variable* `in` *sequence*:
    actions

sequence: tuple, list

`len(sequence)` **iterations** (at most)

variable: *loop index*: each time the variable is set to the value of the next item in the sequence

```
sum = 0
for n in [6, 3, 5, 7]:
    sum += n
    print(sum, n)
average = sum / 4
```

Is equivalent to:

```
sum = 0
sequence = [6, 3, 5, 7]
```

Iteration 1:
```
n = sequence[0]
sum += n
print(sum, n)      → 6 6
```

Iteration 2:
```
n = sequence[1]
sum += n
print(sum, n)      → 9 3
```

Iteration 3:
```
n = sequence[2]
sum += n
print(sum, n)
```

Iteration 4:
```
n = sequence[3]
sum += n
print(sum, n)
```

```
average = sum / 4
```

Loops so far: definite loops, `for` construct

✓ Repeat a set of actions a **defined number of times** (at *most*)

`for` *variable* `in` *sequence*:
    actions

sequence: tuple, list

`len(sequence)` **iterations** (at most)

variable: *loop index*: each time the variable is set to the value of the next item in the sequence

```
sum = 0
for n in [6, 3, 5, 7]:
    sum += n
    print(sum, n)
average = sum / 4
```

Is equivalent to:

```
sum = 0
sequence = [6, 3, 5, 7]
```

Iteration 1:
```
n = sequence[0]
sum += n
print(sum, n)      → 6 6
```

Iteration 2:
```
n = sequence[1]
sum += n
print(sum, n)      → 9 3
```

Iteration 3:
```
n = sequence[2]
sum += n
print(sum, n)      → 14 5
```

Iteration 4:
```
n = sequence[3]
sum += n
print(sum, n)
```

```
average = sum / 4
```

Loops so far: definite loops, `for` construct

✓ Repeat a set of actions a **defined number of times** (at *most*)

`for` *variable* `in` *sequence*:
    actions

sequence: tuple, list

`len(sequence)` **iterations** (at most)

variable: *loop index*: each time the variable is set to the value of the next item in the sequence

```
sum = 0
for n in [6, 3, 5, 7]:
    sum += n
    print(sum, n)
average = sum / 4
```

Is equivalent to:

```
sum = 0
sequence = [6, 3, 5, 7]
```

Iteration 1:
```
n = sequence[0]
sum += n
print(sum, n)      → 6 6
```

Iteration 2:
```
n = sequence[1]
sum += n
print(sum, n)      → 9 3
```

Iteration 3:
```
n = sequence[2]
sum += n
print(sum, n)      → 14 5
```

Iteration 4:
```
n = sequence[3]
sum += n
print(sum, n)      → 21 7
```

```
average = sum / 4
```

Loops so far: definite loops, `for` construct

- ✓ Any sequence is a valid one to index the loop

Loops so far: definite loops, `for` construct

- ✓ Any sequence is a valid one to index the loop

```
for i in [(1,3), ('a', 2), (True, 'hello', 5)]:
    print('This is a loop iteration')
```

Loops so far: definite loops, `for` construct

- ✓ Any sequence is a valid one to index the loop

```
for i in [(1,3), ('a', 2), (True, 'hello', 5)]:
    print('This is a loop iteration')
```

```
sum = 0
for i in [(1,3), ('a', 2), (True, 5, 'hello')]:
  sum += i[1]
  print('Loop variable:', i, 'Sum:', sum)
```

Loops so far: definite loops, `for` construct

- ✓ Any sequence is a valid one to index the loop

```
for i in [(1,3), ('a', 2), (True, 'hello', 5)]:
    print('This is a loop iteration')
```

```
sum = 0
for i in [(1,3), ('a', 2), (True, 5, 'hello')]:
  sum += i[1]
  print('Loop variable:', i, 'Sum:', sum)
```

- ✓ `range(start, end, step)` function for generating sequences that are ranges of integer numbers

Loops so far: definite loops, `for` construct

- ✓ Any sequence is a valid one to index the loop

```
for i in [(1,3), ('a', 2), (True, 'hello', 5)]:
    print('This is a loop iteration')
```

```
sum = 0
for i in [(1,3), ('a', 2), (True, 5, 'hello')]:
  sum += i[1]
  print('Loop variable:', i, 'Sum:', sum)
```

- ✓ `range(start, end, step)` function for generating sequences that are ranges of integer numbers

```
for i in range(-1,10,2):
   print(i)
```

→ -1, 1, 3, 5, 7, 9

Loops so far: definite loops, `for` construct

- ✓ Any sequence is a valid one to index the loop

```
for i in [(1,3), ('a', 2), (True, 'hello', 5)]:
    print('This is a loop iteration')
```

```
sum = 0
for i in [(1,3), ('a', 2), (True, 5, 'hello')]:
  sum += i[1]
  print('Loop variable:', i, 'Sum:', sum)
```

- ✓ `range(start, end, step)` function for generating sequences that are ranges of integer numbers

```
for i in range(-1,10,2):
   print(i)
```
→ -1, 1, 3, 5, 7, 9

```
for i in range(2,9):
   print(i)
```
→ 2, 3, 4, 5, 6, 7, 8

Loops so far: definite loops, `for` construct

- ✓ Any sequence is a valid one to index the loop

```
for i in [(1,3), ('a', 2), (True, 'hello', 5)]:
    print('This is a loop iteration')
```

```
sum = 0
for i in [(1,3), ('a', 2), (True, 5, 'hello')]:
  sum += i[1]
  print('Loop variable:', i, 'Sum:', sum)
```

- ✓ `range(start, end, step)` function for generating sequences that are ranges of integer numbers

```
for i in range(-1,10,2):
    print(i)
```
→ -1, 1, 3, 5, 7, 9

```
for i in range(2,9):
    print(i)
```
→ 2, 3, 4, 5, 6, 7, 8

```
for i in range(10):
    print(i)
```
→ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

`continue`: jump to the end of the loop, skip to next iteration

- It might happen that **a part of the block of code in the for body need to be skipped for certain data items based on conditional tests**, moving straight to ne next iteration → `continue`

`continue`: jump to the end of the loop, skip to next iteration

- It might happen that **a part of the block of code in the for body need to be skipped for certain data items based on conditional tests**, moving straight to ne next iteration → `continue`

```
numbers = [30, 40, 0, 20, 0, -11, 5]
percent = []
for n in numbers:
    if n == 0 or n < 0:
        continue
    frac = n / 100
    percent.append(frac)
    print("New percentage:", frac)
print("Non zero:", len(percent))
```

`continue`: jump to the end of the loop, skip to next iteration

- It might happen that **a part of the block of code in the for body need to be skipped for certain data items based on conditional tests**, moving straight to ne next iteration → `continue`

```
numbers = [30, 40, 0, 20, 0, -11, 5]
percent = []
for n in numbers:
    if n == 0 or n < 0:
        continue
    frac = n / 100
    percent.append(frac)
    print("New percentage:", frac)
print("Non zero:", len(percent))
```

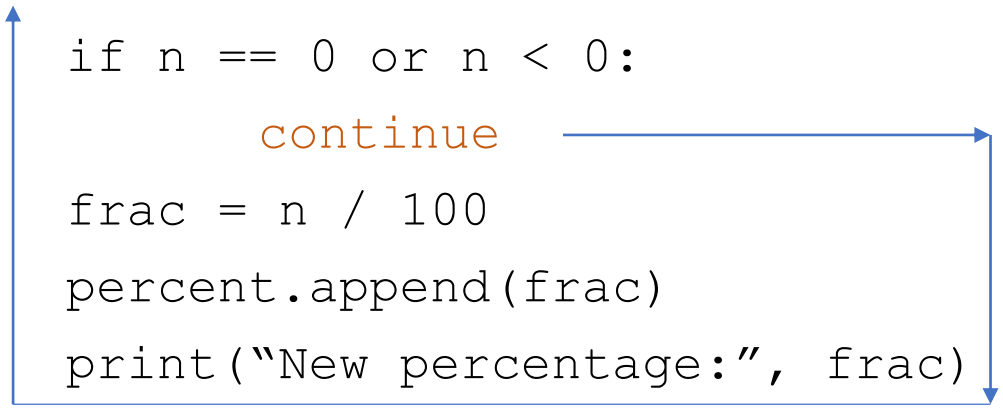
jump to the end of the loop code block

→ new iteration starts: `n` gets its next value

`continue`: jump to the end of the loop, skip to next iteration

- It might happen that **a part of the block of code in the for body need to be skipped for certain data items based on conditional tests**, moving straight to ne next iteration → `continue`

```
numbers = [30, 40, 0, 20, 0, -11, 5]
percent = []
for n in numbers:
    if n == 0 or n < 0:
        continue
    frac = n / 100
    percent.append(frac)
    print("New percentage:", frac)
print("Non zero:", len(percent))
```



Iteration 1

`n = 30`

Executed instructions:

`if, append, print`

`percent: [0.3]`

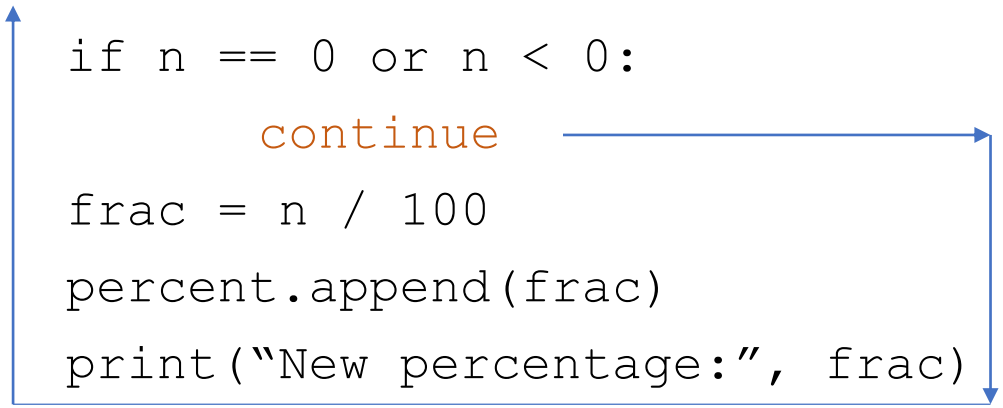
jump to the end of the loop code block

→ new iteration starts: `n` gets its next value

`continue`: jump to the end of the loop, skip to next iteration

- It might happen that **a part of the block of code in the for body need to be skipped for certain data items based on conditional tests**, moving straight to ne next iteration → `continue`

```
numbers = [30, 40, 0, 20, 0, -11, 5]
percent = []
for n in numbers:
    if n == 0 or n < 0:
        continue
    frac = n / 100
    percent.append(frac)
    print("New percentage:", frac)
print("Non zero:", len(percent))
```



Iteration 1

`n = 30`

Executed instructions:

`if, append, print`

`percent: [0.3]`

Iteration 2

`n = 40`

Executed instructions:

`if, append, print`

`percent: [0.3,0.4]`

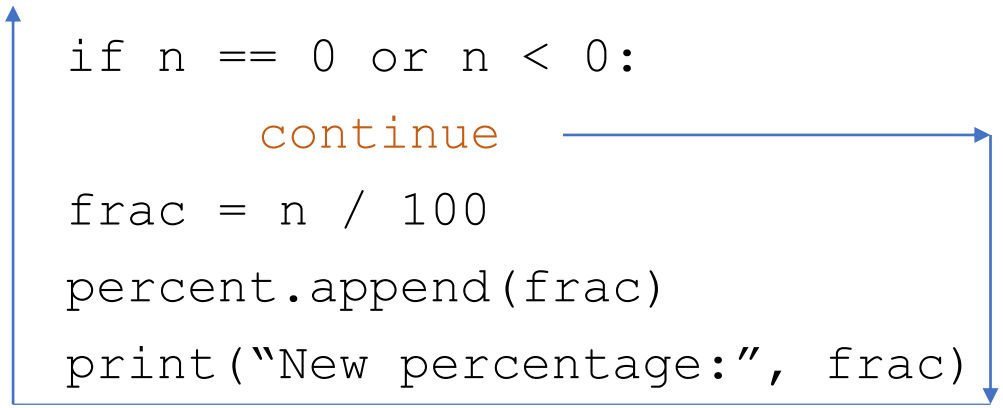
jump to the end of the loop code block

→ new iteration starts: `n` gets its next value

`continue`: jump to the end of the loop, skip to next iteration

- It might happen that **a part of the block of code in the for body need to be skipped for certain data items based on conditional tests**, moving straight to ne next iteration → `continue`

```
numbers = [30, 40, 0, 20, 0, -11, 5]
percent = []
for n in numbers:
    if n == 0 or n < 0:
        continue
    frac = n / 100
    percent.append(frac)
    print("New percentage:", frac)
print("Non zero:", len(percent))
```



Iteration 1

`n = 30`

Executed instructions:

`if, append, print`

`percent: [0.3]`

Iteration 2

`n = 40`

Executed instructions:

`if, append, print`

`percent: [0.3,0.4]`

Iteration 3

`n = 0`

Executed instructions:

`if, continue`

`percent: [0.3,0.4]`

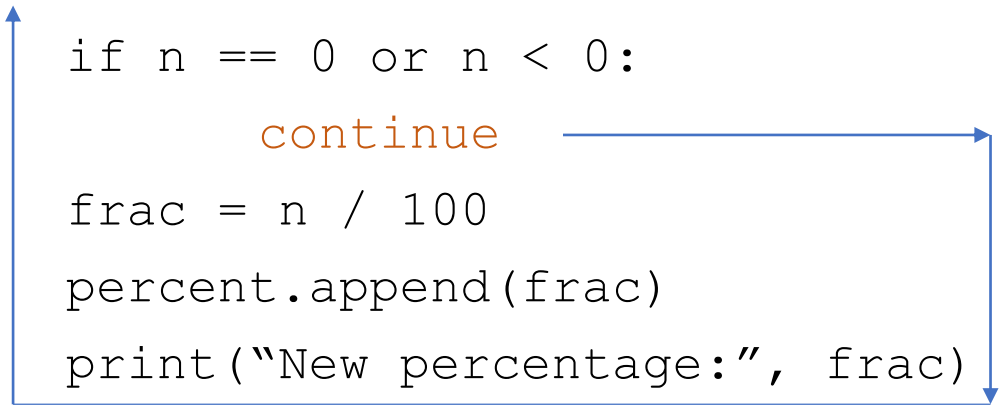
jump to the end of the loop code block

→ new iteration starts: `n` gets its next value

`continue`: jump to the end of the loop, skip to next iteration

- It might happen that **a part of the block of code in the for body need to be skipped for certain data items based on conditional tests**, moving straight to ne next iteration → `continue`

```
numbers = [30, 40, 0, 20, 0, -11, 5]
percent = []
for n in numbers:
    if n == 0 or n < 0:
        continue
    frac = n / 100
    percent.append(frac)
    print("New percentage:", frac)
print("Non zero:", len(percent))
```



Iteration 1

`n = 30`

Executed instructions:

`if, append, print`

`percent: [0.3]`

Iteration 2

`n = 40`

Executed instructions:

`if, append, print`

`percent: [0.3,0.4]`

Iteration 3

`n = 0`

Executed instructions:

`if, continue`

`percent: [0.3,0.4]`

Iteration 4

`n = 20`

Executed instructions:

`if, append, print`

`percent: [0.3,0.4,0.2]`

jump to the end of the loop code block

→ new iteration starts: `n` gets its next value

`break:` jump out of the loop (that *at most*)

- It might happen that *according to a conditional test*, the **loop must be interrupted** without performing any further instructions, moving the program counter to the first instruction after the loop → `break`

`break`: jump out of the loop (that *at most*)

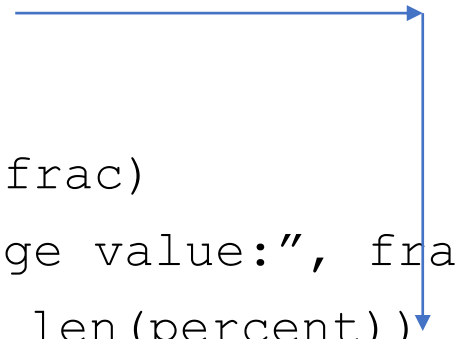
- It might happen that *according to a conditional test*, the **loop must be interrupted** without performing any further instructions, moving the program counter to the first instruction after the loop → `break`

```
numbers = [30, 40, 0, 20, 0, -11, 5]
percent = []
for n in numbers:
    if n == 0 or n < 0:
            print("Value not allowed!")
            break
    frac = n / 100
    percent.append(frac)
    print("Percentage value:", frac)
print("Non zero:", len(percent))
```

`break`: jump out of the loop (that *at most*)

- It might happen that *according to a conditional test*, the **loop must be interrupted** without performing any further instructions, moving the program counter to the first instruction after the loop → `break`

```
numbers = [30, 40, 0, 20, 0, -11, 5]
percent = []
for n in numbers:
    if n == 0 or n < 0:
        print("Value not allowed!")
        break
    frac = n / 100
    percent.append(frac)
    print("Percentage value:", frac)
print("Non zero:", len(percent))
```



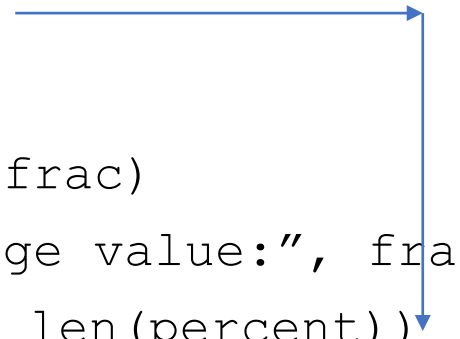
jump out of the loop

→ next program instruction is executed

`break`: jump out of the loop (that *at most*)

- It might happen that *according to a conditional test*, the **loop must be interrupted** without performing any further instructions, moving the program counter to the first instruction after the loop → `break`

```
numbers = [30, 40, 0, 20, 0, -11, 5]
percent = []
for n in numbers:
    if n == 0 or n < 0:
        print("Value not allowed!")
        break
    frac = n / 100
    percent.append(frac)
    print("Percentage value:", frac)
print("Non zero:", len(percent))
```



Iteration 1

`n = 30`

Executed instructions:

`if, append, print`

`percent: [0.3]`

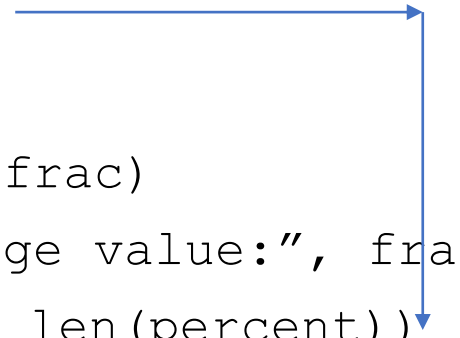
jump out of the loop

→ next program instruction is executed

`break`: jump out of the loop (that *at most*)

- It might happen that *according to a conditional test*, the **loop must be interrupted** without performing any further instructions, moving the program counter to the first instruction after the loop → `break`

```
numbers = [30, 40, 0, 20, 0, -11, 5]
percent = []
for n in numbers:
    if n == 0 or n < 0:
        print("Value not allowed!")
        break
    frac = n / 100
    percent.append(frac)
    print("Percentage value:", frac)
print("Non zero:", len(percent))
```



Iteration 1

`n = 30`

Executed instructions:

`if, append, print`

`percent: [0.3]`

Iteration 2

`n = 40`

Executed instructions:

`if, append, print`

`percent: [0.3,0.4]`

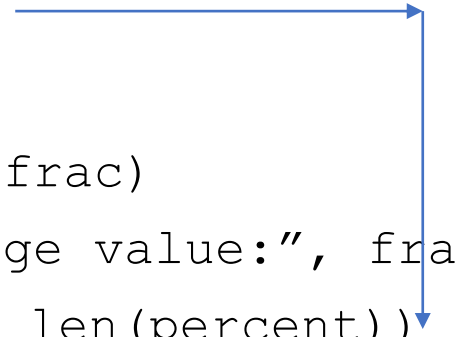
jump out of the loop

→ next program instruction is executed

`break`: jump out of the loop (that *at most*)

- It might happen that *according to a conditional test*, the **loop must be interrupted** without performing any further instructions, moving the program counter to the first instruction after the loop → `break`

```
numbers = [30, 40, 0, 20, 0, -11, 5]
percent = []
for n in numbers:
    if n == 0 or n < 0:
        print("Value not allowed!")
        break
    frac = n / 100
    percent.append(frac)
    print("Percentage value:", frac)
print("Non zero:", len(percent))
```



Iteration 1

`n = 30`

Executed instructions:

`if, append, print`

`percent: [0.3]`

Iteration 2

`n = 40`

Executed instructions:

`if, append, print`

`percent: [0.3,0.4]`

Iteration 3

`n = 0`

Executed instructions:

`if, print, break`

`percent: [0.3,0.4]`

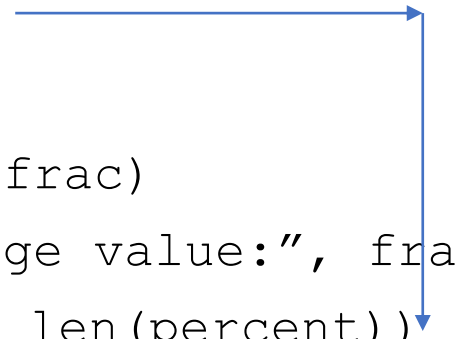
jump out of the loop

→ next program instruction is executed

`break`: jump out of the loop (that *at most*)

- It might happen that *according to a conditional test*, the **loop must be interrupted** without performing any further instructions, moving the program counter to the first instruction after the loop → `break`

```
numbers = [30, 40, 0, 20, 0, -11, 5]
percent = []
for n in numbers:
    if n == 0 or n < 0:
        print("Value not allowed!")
        break
    frac = n / 100
    percent.append(frac)
    print("Percentage value:", frac)
print("Non zero:", len(percent))
```



Iteration 1

`n = 30`

Executed instructions:

`if, append, print`

`percent: [0.3]`

Iteration 2

`n = 40`

Executed instructions:

`if, append, print`

`percent: [0.3,0.4]`

Iteration 3

`n = 0`

Executed instructions:

`if, print, break`

`percent: [0.3,0.4]`

Out of the loop

Executed instructions:

`print`

`percent: [0.3,0.4], n = 0`

jump out of the loop

→ next program instruction is executed

Modifying loop index variable and sequence during iteration?

- The loop index variable is just a variable, therefore it can (*you shouldn't*) be modified inside a loop
- Also the sequence, if modifiable (i.e., not a `range`()), can be changed (*you shouldn't*) during the iterations

Modifying loop index variable and sequence during iteration?

- The loop index variable is just a variable, therefore it can (*you shouldn't*) be modified inside a loop
- Also the sequence, if modifiable (i.e., not a `range`()), can be changed (*you shouldn't*) during the iterations

```
numbers = [30, 40, '*', 20]
percent = []
for n in numbers:
    if n == '*' :
            numbers += [1,2,3]
            continue
    n /= 100
    frac = n
    percent.append(frac)
print('Total percent:', len(percent))
```

Modifying loop index variable and sequence during iteration?

- The loop index variable is just a variable, therefore it can (*you shouldn't*) be modified inside a loop
- Also the sequence, if modifiable (i.e., not a `range`()), can be changed (*you shouldn't*) during the iterations

```
numbers = [30, 40, '*', 20]
percent = []
for n in numbers:
    if n == '*' :
            numbers += [1,2,3]
            continue
    n /= 100
    frac = n
    percent.append(frac)
print('Total percent:', len(percent))
```

Iteration 1

`n = 30`

Sequence to go:

`[40,'*',20]`

Modifying loop index variable and sequence during iteration?

- The loop index variable is just a variable, therefore it can (*you shouldn't*) be modified inside a loop
- Also the sequence, if modifiable (i.e., not a `range`()), can be changed (*you shouldn't*) during the iterations

```
numbers = [30, 40, '*', 20]
percent = []
for n in numbers:
    if n == '*' :
            numbers += [1,2,3]
            continue
    n /= 100
    frac = n
    percent.append(frac)
print('Total percent:', len(percent))
```

Iteration 1

`n = 30`

Sequence to go:

`[40,'*',20]`

Iteration 2

`n = 40`

Sequence to go:

`['*',20]`

Modifying loop index variable and sequence during iteration?

- The loop index variable is just a variable, therefore it can (*you shouldn't*) be modified inside a loop
- Also the sequence, if modifiable (i.e., not a `range`()), can be changed (*you shouldn't*) during the iterations

```
numbers = [30, 40, '*', 20]
percent = []
for n in numbers:
    if n == '*' :
            numbers += [1,2,3]
            continue
    n /= 100
    frac = n
    percent.append(frac)
print('Total percent:', len(percent))
```

Iteration 1

`n = 30`

Sequence to go:

`[40,'*',20]`

Iteration 2

`n = 40`

Sequence to go:

`['*',20]`

Iteration 3

`n = '*'`

Sequence to go:

`[20,1,2,3]`

Modifying loop index variable and sequence during iteration?

- The loop index variable is just a variable, therefore it can (*you shouldn't*) be modified inside a loop
- Also the sequence, if modifiable (i.e., not a `range`()), can be changed (*you shouldn't*) during the iterations

```
numbers = [30, 40, '*', 20]
percent = []
for n in numbers:
    if n == '*' :
            numbers += [1,2,3]
            continue
    n /= 100
    frac = n
    percent.append(frac)
print('Total percent:', len(percent))
```

Iteration 1

`n = 30`

Sequence to go:

`[40,'*',20]`

Iteration 2

`n = 40`

Sequence to go:

`['*',20]`

Iteration 3

`n = '*'`

Sequence to go:

`[20,1,2,3]`

…

Modifying loop index variable and sequence during iteration?

- The loop index variable is just a variable, therefore it can (*you shouldn't*) be modified inside a loop
- Also the sequence, if modifiable (i.e., not a `range`()), can be changed (*you shouldn't*) during the iterations

```
numbers = [30, 40, '*', 20]
percent = []
for n in numbers:
    if n == '*' :
            numbers += [1,2,3]
            continue
    n /= 100
    frac = n
    percent.append(frac)
print('Total percent:', len(percent))
```

Iteration 1

`n = 30`

Sequence to go:

`[40,'*',20]`

Iteration 2

`n = 40`

Sequence to go:

`['*',20]`

Iteration 3

`n = '*'`

Sequence to go:

`[20,1,2,3]`

...

Iteration 7

`n = 3`

Sequence to go:

`[]`

Modifying loop index variable and sequence during iteration?

- The loop index variable is just a variable, therefore it can (*you shouldn't*) be modified inside a loop
- Also the sequence, if modifiable (i.e., not a `range`()), can be changed (*you shouldn't*) during the iterations

```
numbers = [30, 40, '*', 20]
percent = []
for n in numbers:
    if n == '*' :
            numbers += [1,2,3]
            continue
    n /= 100
    frac = n
    percent.append(frac)
print('Total percent:', len(percent))
```

Iteration 1
`n = 30`

Sequence to go:
`[40,'*',20]`

Iteration 2
`n = 40`

Sequence to go:
`['*',20]`

Iteration 3
`n = '*'`

Sequence to go:
`[20,1,2,3]`

...

Iteration 7
`n = 3`

Sequence to go:
`[]`

What happens with: `numbers[:] = []` ?

Nested loops

- **Loops can be nested** in arbitrary levels, that can be directly related or not to each other

Nested loops

- **Loops can be nested** in arbitrary levels, that can be directly related or not to each other

```
for s1 in seq1:
    for s2 in seq2:
        #do something with (s1, s2)
```

Two level nesting, each level is independently defined

Nested loops

- **Loops can be nested** in arbitrary levels, that can be directly related or not to each other

```
for s1 in seq1:
    for s2 in seq2:
        #do something with (s1, s2)
```

Two level nesting, each level is independently defined

```
for s1 in seq1:
    for s2 in s1:
        for s3 in s2:
            #do something with s3
```

Three level nesting, in this example each level is derived from the previous one

Nested loops

- ✓ Typical operation on databases

Nested loops

```
cars = [ ['Toyota', 'white', 2012, 15000],
         ['Toyota', 'black', 2011, 12000],
         ['Nissan', 'black', 2011, 10000],
         ['Toyota', 'black', 2015, 25000],
         ['BMW', 'blue', 2018, 50000],
         ['Toyota', 'white', 2018, 60000],
         ['Ferrari', 'red', 2016, 100000],
         ['Ferrari', 'blue', 2015, 85000]  ]
```

- ✓ Typical operation on databases

Nested loops

```
cars = [ ['Toyota', 'white', 2012, 15000],
         ['Toyota', 'black', 2011, 12000],
         ['Nissan', 'black', 2011, 10000],
         ['Toyota', 'black', 2015, 25000],
         ['BMW', 'blue', 2018, 50000],
         ['Toyota', 'white', 2018, 60000],
         ['Ferrari', 'red', 2016, 100000],
         ['Ferrari', 'blue', 2015, 85000]  ]

colors = [ 'white', 'red', 'blue']
```

- ✓ Typical operation on databases

Nested loops

```
cars = [ ['Toyota', 'white', 2012, 15000],
         ['Toyota', 'black', 2011, 12000],
         ['Nissan', 'black', 2011, 10000],
         ['Toyota', 'black', 2015, 25000],
         ['BMW', 'blue', 2018, 50000],
         ['Toyota', 'white', 2018, 60000],
         ['Ferrari', 'red', 2016, 100000],
         ['Ferrari', 'blue', 2015, 85000]  ]

colors = [ 'white', 'red', 'blue']
cars_of_specific_color = []
```

- ✓ Typical operation on databases

Nested loops

```
cars = [ ['Toyota', 'white', 2012, 15000],
         ['Toyota', 'black', 2011, 12000],
         ['Nissan', 'black', 2011, 10000],
         ['Toyota', 'black', 2015, 25000],
         ['BMW', 'blue', 2018, 50000],
         ['Toyota', 'white', 2018, 60000],
         ['Ferrari', 'red', 2016, 100000],
         ['Ferrari', 'blue', 2015, 85000]  ]

colors = [ 'white', 'red', 'blue']
cars_of_specific_color = []
for c in cars:
```

- ✓ Typical operation on databases

Nested loops

```
cars = [ ['Toyota', 'white', 2012, 15000],
         ['Toyota', 'black', 2011, 12000],
         ['Nissan', 'black', 2011, 10000],
         ['Toyota', 'black', 2015, 25000],
         ['BMW', 'blue', 2018, 50000],
         ['Toyota', 'white', 2018, 60000],
         ['Ferrari', 'red', 2016, 100000],
         ['Ferrari', 'blue', 2015, 85000]  ]

colors = [ 'white', 'red', 'blue']
cars_of_specific_color = []
for c in cars:
    for col in colors:
```

- ✓ Typical operation on databases

Nested loops

```
cars = [ ['Toyota', 'white', 2012, 15000],
         ['Toyota', 'black', 2011, 12000],
         ['Nissan', 'black', 2011, 10000],
         ['Toyota', 'black', 2015, 25000],
         ['BMW', 'blue', 2018, 50000],
         ['Toyota', 'white', 2018, 60000],
         ['Ferrari', 'red', 2016, 100000],
         ['Ferrari', 'blue', 2015, 85000]  ]

colors = [ 'white', 'red', 'blue']
cars_of_specific_color = []
for c in cars:
    for col in colors:
        if c[1] == col:
```

- ✓ Typical operation on databases

Nested loops

```
cars = [ ['Toyota', 'white', 2012, 15000],
         ['Toyota', 'black', 2011, 12000],
         ['Nissan', 'black', 2011, 10000],
         ['Toyota', 'black', 2015, 25000],
         ['BMW', 'blue', 2018, 50000],
         ['Toyota', 'white', 2018, 60000],
         ['Ferrari', 'red', 2016, 100000],
         ['Ferrari', 'blue', 2015, 85000]  ]

colors = [ 'white', 'red', 'blue']
cars_of_specific_color = []
for c in cars:
    for col in colors:
        if c[1] == col:
            cars_of_specific_color.append(c)
```

- ✓ Typical operation on databases

Nested loops

```
cars = [ ['Toyota', 'white', 2012, 15000],
         ['Toyota', 'black', 2011, 12000],
         ['Nissan', 'black', 2011, 10000],
         ['Toyota', 'black', 2015, 25000],
         ['BMW', 'blue', 2018, 50000],
         ['Toyota', 'white', 2018, 60000],
         ['Ferrari', 'red', 2016, 100000],
         ['Ferrari', 'blue', 2015, 85000]  ]

colors = [ 'white', 'red', 'blue']
cars_of_specific_color = []
for c in cars:
    for col in colors:
        if c[1] == col:
            cars_of_specific_color.append(c)
print('Found', len(cars_of_specific_color), 'cars of the desired colors:')
for c in cars_of_specific_color:
    print(c)
```

- ✓ Typical operation on databases

Nested loops: accessing data in lists of lists

- Finding the **max (min) in a list of lists**

Nested loops: accessing data in lists of lists

- Finding the **max (min) in a list of lists**

```
list1 = [ [ [ [110, 'r'], [22, 'g'], [3, 'b']  ]  ],
          [ [ [45, 'r'], [105, 'g'], [26, 'b'] ] ],
          [ [ [76, 'r'], [88, 'g'], [190, 'b'] ] ]
        ]
```

Nested loops: accessing data in lists of lists

- Finding the **max (min) in a list of lists**

```
list1 = [ [ [ [110, 'r'], [22, 'g'], [3, 'b']  ]  ],
          [ [ [45, 'r'], [105, 'g'], [26, 'b'] ] ],
          [ [ [76, 'r'], [88, 'g'], [190, 'b'] ] ]
        ]
print(max(list1))
```

Nested loops: accessing data in lists of lists

- Finding the **max (min) in a list of lists**

```
list1 = [ [ [ [110, 'r'], [22, 'g'], [3, 'b']  ]  ],
          [ [ [45, 'r'], [105, 'g'], [26, 'b'] ] ],
          [ [ [76, 'r'], [88, 'g'], [190, 'b'] ] ]
        ]
print(max(list1))
```

→ what will be printed here?

Nested loops: accessing data in lists of lists

- Finding the **max (min) in a list of lists**

```
list1 = [ [ [ [110, 'r'], [22, 'g'], [3, 'b']  ]  ],
          [ [ [45, 'r'], [105, 'g'], [26, 'b'] ] ],
          [ [ [76, 'r'], [88, 'g'], [190, 'b'] ] ]
        ]
print(max(list1))
rgb_max = -1
iteration_count = 0
```

→ what will be printed here?

Nested loops: accessing data in lists of lists

- Finding the **max (min) in a list of lists**

```
list1 = [ [ [ [110, 'r'], [22, 'g'], [3, 'b']  ]  ],
          [ [ [45, 'r'], [105, 'g'], [26, 'b'] ] ],
          [ [ [76, 'r'], [88, 'g'], [190, 'b'] ] ]
        ]
print(max(list1))
rgb_max = -1
iteration_count = 0
for s1 in list1:
```

→ what will be printed here?

Nested loops: accessing data in lists of lists

- Finding the **max (min) in a list of lists**

```
list1 = [ [ [ [110, 'r'], [22, 'g'], [3, 'b']  ]  ],
          [ [ [45, 'r'], [105, 'g'], [26, 'b'] ] ],
          [ [ [76, 'r'], [88, 'g'], [190, 'b'] ] ]
        ]
print(max(list1))
rgb_max = -1
iteration_count = 0
for s1 in list1:
    for s2 in s1:
```

→ what will be printed here?

Nested loops: accessing data in lists of lists

- Finding the **max (min) in a list of lists**

```
list1 = [ [ [ [110, 'r'], [22, 'g'], [3, 'b']  ]  ],
          [ [ [45, 'r'], [105, 'g'], [26, 'b'] ] ],
          [ [ [76, 'r'], [88, 'g'], [190, 'b'] ] ]
        ]
print(max(list1))
rgb_max = -1
iteration_count = 0
for s1 in list1:
    for s2 in s1:
        for s3 in s2:
```

→ what will be printed here?

Nested loops: accessing data in lists of lists

- Finding the **max (min) in a list of lists**

```
list1 = [ [ [ [110, 'r'], [22, 'g'], [3, 'b']  ]  ],
          [ [ [45, 'r'], [105, 'g'], [26, 'b'] ] ],
          [ [ [76, 'r'], [88, 'g'], [190, 'b'] ] ]
        ]
print(max(list1))
rgb_max = -1
iteration_count = 0
for s1 in list1:
    for s2 in s1:
        for s3 in s2:
            if s3[0] > rgb_max:
                rgb_max = s3[0]
```

→ what will be printed here?

Nested loops: accessing data in lists of lists

- Finding the **max (min) in a list of lists**

```
list1 = [ [ [ [110, 'r'], [22, 'g'], [3, 'b']  ]  ],
          [ [ [45, 'r'], [105, 'g'], [26, 'b'] ] ],
          [ [ [76, 'r'], [88, 'g'], [190, 'b'] ] ]
        ]
print(max(list1))
rgb_max = -1
iteration_count = 0
for s1 in list1:
    for s2 in s1:
        for s3 in s2:
            if s3[0] > rgb_max:
                rgb_max = s3[0]
            iteration_count += 1
```

→ what will be printed here?

Nested loops: accessing data in lists of lists

- Finding the **max (min) in a list of lists**

```
list1 = [ [ [ [110, 'r'], [22, 'g'], [3, 'b']  ]  ],
          [ [ [45, 'r'], [105, 'g'], [26, 'b'] ] ],
          [ [ [76, 'r'], [88, 'g'], [190, 'b'] ] ]
        ]
print(max(list1))
rgb_max = -1
iteration_count = 0
for s1 in list1:
    for s2 in s1:
        for s3 in s2:
            if s3[0] > rgb_max:
                rgb_max = s3[0]
            iteration_count += 1
print('max rgb:', rgb_max, _iteration_count)
```

→ what will be printed here?

Nested loops: accessing data in lists of lists

- Finding the **max (min) in a list of lists**

```
list1 = [ [ [ [110, 'r'], [22, 'g'], [3, 'b']  ]  ],
          [ [ [45, 'r'], [105, 'g'], [26, 'b'] ] ],
          [ [ [76, 'r'], [88, 'g'], [190, 'b'] ] ]
        ]
print(max(list1))
rgb_max = -1
iteration_count = 0
for s1 in list1:
    for s2 in s1:
        for s3 in s2:
            if s3[0] > rgb_max:
                rgb_max = s3[0]
            iteration_count += 1
print('max rgb:', rgb_max, _iteration_count)
```

→ what will be printed here?

***Complexity* of the computing:**

- Doing one `if` comparison + assignment `=` : how many times?
  length(list level 1) * length(list level 2) * length(list level 3)

Nested loops: creating and accessing matrix data structures

- **Matrix:** in linear algebra it is a *rectangular* array of numbers organized in m rows and n columns, where the rows are horizontal and the columns are vertical
- Each row and each column can be read as a *vector*, of dimension n and m respectively

$$M = \begin{bmatrix} 3 & 109 & 88 \\ 17 & 4 & 12 \end{bmatrix}$$

2 x 3 matrix

$$M = \begin{bmatrix} 0.4 & 100 \\ -3 & 247 \\ 0 & 25 \end{bmatrix}$$

3 x 2 matrix

$$M = \begin{bmatrix} 1 & 4 & 88 \\ 25.4 & -100 & 7 \\ 2 & 99 & 4.5 \end{bmatrix}$$

3 x 3 matrix

Nested loops: creating and accessing matrix data structures

- **Matrix:** in linear algebra it is a *rectangular* array of numbers organized in m rows and n columns, where the rows are horizontal and the columns are vertical
- Each row and each column can be read as a *vector*, of dimension n and m respectively

$$M = \begin{bmatrix} 3 & 109 & 88 \\ 17 & 4 & 12 \end{bmatrix}$$

2 x 3 matrix

$$M = \begin{bmatrix} 0.4 & 100 \\ -3 & 247 \\ 0 & 25 \end{bmatrix}$$

3 x 2 matrix

$$M = \begin{bmatrix} 1 & 4 & 88 \\ 25.4 & -100 & 7 \\ 2 & 99 & 4.5 \end{bmatrix}$$

3 x 3 matrix

- Given a matrix A, the notation m_{ij} or M_{ij} is commonly used to refer to the element in row i and column j
- In python, a matrix data structure can be implemented using lists/tuples, and it can be *convenient* to use something like `m[i][j]` to access the elements

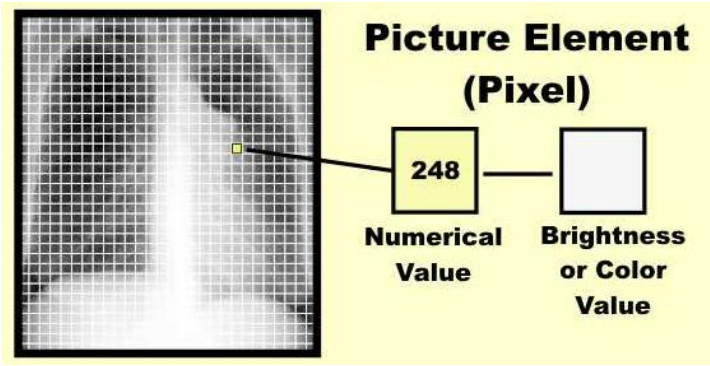
Nested loops: creating and accessing matrix data structures

- Exemplary use of matrices in computing: digital image processing!

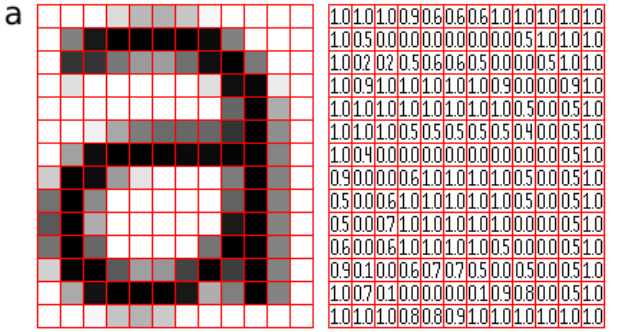
Nested loops: creating and accessing matrix data structures

- Exemplary use of matrices in computing: digital image processing!

  A digital image is basically represented as an $m \times n$ **matrix of pixel values**



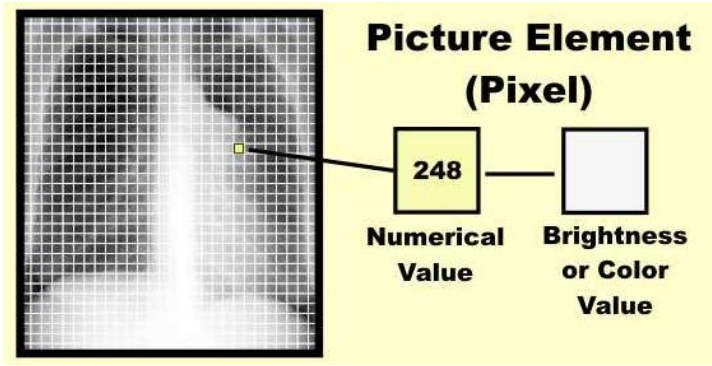
a



Nested loops: creating and accessing matrix data structures

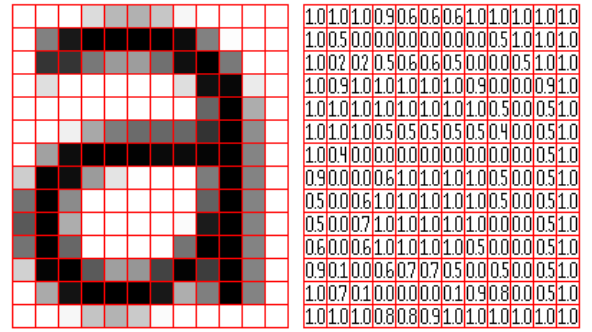
- Exemplary use of matrices in computing: digital image processing!

A digital image is basically represented as an $m \times n$ **matrix of pixel values**



Grayscale image: each pixel is encoded in one byte, such that it can take values in the integer range between 0 and 255

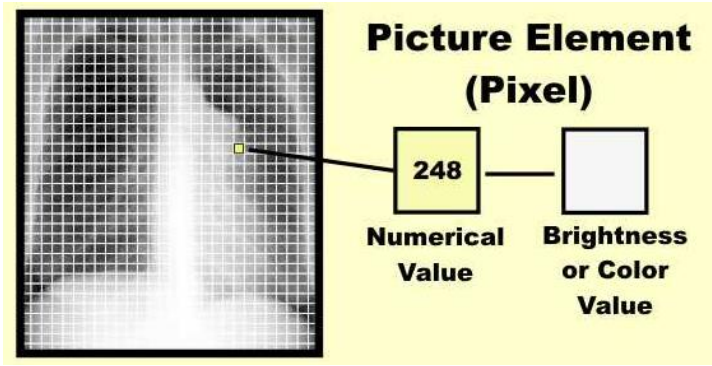
a



Nested loops: creating and accessing matrix data structures

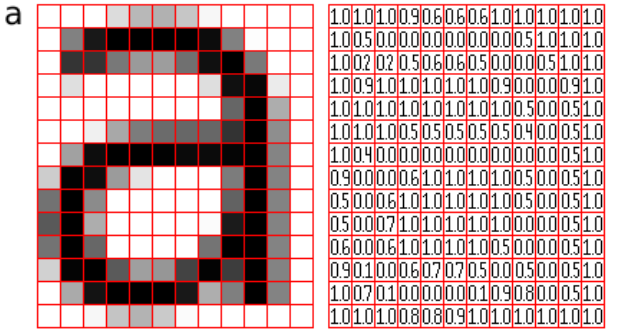
- Exemplary use of matrices in computing: digital image processing!

A digital image is basically represented as an $m \times n$ **matrix of pixel values**



Grayscale image: each pixel is encoded in one byte, such that it can take values in the integer range between 0 and 255

a



Color Chart	R	G	B	Color Name
	0	0	0	Black
	255	255	255	White
	224	224	224	Light Gray
	128	128	128	Gray
	64	64	64	Dark Gray
	255	0	0	Red
	255	96	208	Pink
	160	32	255	Purple
	80	208	255	Light Blue
	0	32	255	Blue
	96	255	128	Yellow-Green
	0	192	0	Green
	255	224	32	Yellow
	255	160	16	Orange
	160	128	96	Brown
	255	208	160	Pale Pink

RGB image: color images where each pixel has a triple of values (r,g,b), each encoded in one byte, that altogether encode the color

Nested loops: creating and accessing matrix data structures

Nested loops: creating and accessing matrix data structures

- **Create an image matrix** using lists (we will see different ways of doing this same task), `range()` is useful!

Nested loops: creating and accessing matrix data structures

- **Create an image matrix** using lists (we will see different ways of doing this same task), `range()` is useful!

```
rows, cols = 10, 8
img = [[]]*rows
print(img)
for r in range(rows):
    for c in range(cols):
        img[r] = [0]*cols
```

Nested loops: creating and accessing matrix data structures

- **Create an image matrix** using lists (we will see different ways of doing this same task), `range()` is useful!

```
rows, cols = 10, 8
img = [[]]*rows
print(img)
for r in range(rows):
    for c in range(cols):
        img[r] = [0]*cols
```

- So far it's I=initialized with all zero, let's give some more meaningful values to the entries:

Nested loops: creating and accessing matrix data structures

- **Create an image matrix** using lists (we will see different ways of doing this same task), `range()` is useful!

```
rows, cols = 10, 8
img = [[]]*rows
print(img)
for r in range(rows):
    for c in range(cols):
        img[r] = [0]*cols
```

- So far it's I=initialized with all zero, let's give some more meaningful values to the entries:

```
for r in range(rows):
    for c in range(cols):
        img[r][c] = (r * c) % 255
    print(img[r])
```

Nested loops: creating and accessing matrix data structures

- **Create an image matrix** using lists (we will see different ways of doing this same task), `range()` is useful!

```
rows, cols = 10, 8
img = [[]]*rows
print(img)
for r in range(rows):
    for c in range(cols):
        img[r] = [0]*cols
```

- So far it's I=initialized with all zero, let's give some more meaningful values to the entries:

```
for r in range(rows):
    for c in range(cols):
        img[r][c] = (r * c) % 255
    print(img[r])
```

- **Data smoothing / filtering**

Nested loops: creating and accessing matrix data structures

- **Create an image matrix** using lists (we will see different ways of doing this same task), `range()` is useful!

```
rows, cols = 10, 8
img = [[]]*rows
print(img)
for r in range(rows):
    for c in range(cols):
        img[r] = [0]*cols
```

- So far it's I=initialized with all zero, let's give some more meaningful values to the entries:

```
for r in range(rows):
    for c in range(cols):
        img[r][c] = (r * c) % 255
    print(img[r])
```

- **Data smoothing / filtering**

```
for r in range(rows):
    for c in range(1, cols-1):
        img[r][c] = int((2 * img[r][c-1] + img[r][c] + 2 * img[r][c+1]) / 3)
    print(img[r])
```

Nested loops: creating and accessing matrix data structures

- **Create an image matrix** using lists (we will see different ways of doing this same task), `range()` is useful!

```
rows, cols = 10, 8
img = [[]]*rows
print(img)
for r in range(rows):
    for c in range(cols):
        img[r] = [0]*cols
```

- So far it's I=initialized with all zero, let's give some more meaningful values to the entries:

```
for r in range(rows):
    for c in range(cols):
        img[r][c] = (r * c) % 255
    print(img[r])
```

- **Data smoothing / filtering**

```
for r in range(rows):
    for c in range(1, cols-1):
        img[r][c] = int((2 * img[r][c-1] + img[r][c] + 2 * img[r][c+1]) / 3)
    print(img[r])
```

WRONG!

Nested loops: creating and accessing matrix data structures

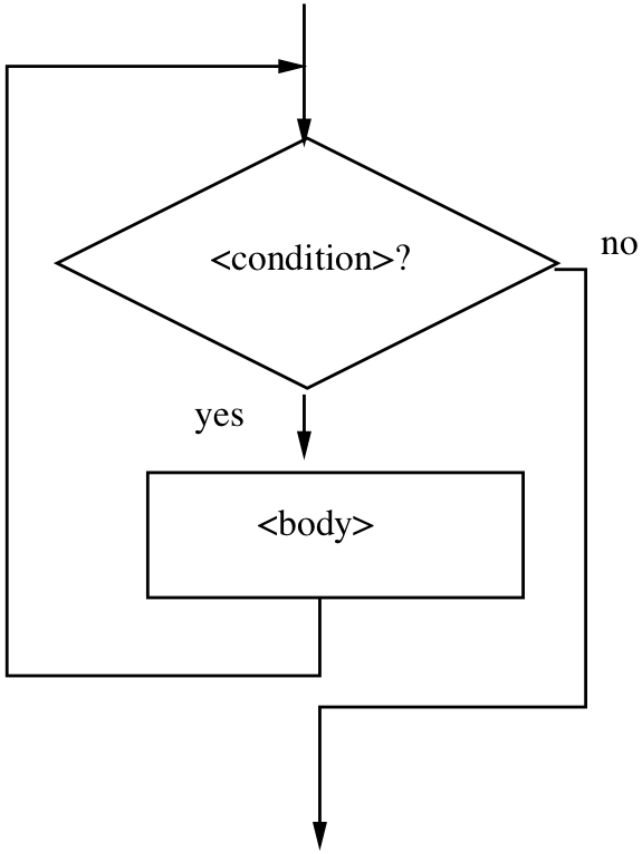
- Finding the **max (min) in a list of lists,** using indexes and `range()`

```
list1 = [ [ [ [110, 'r'], [22, 'g'], [3, 'b']  ]  ],
          [ [ [45, 'r'], [105, 'g'], [26, 'b'] ] ],
          [ [ [76, 'r'], [88, 'g'], [190, 'b'] ] ] ]

print(max(list1))
rgb_max = -1
for i1 in range(len(list1)):
    for i2 in range(len(list1[i1])) :
        for i3 in range(len(list1[i1][i2])):
            item = list1[i1][i2][i3]
            if item[0] > rgb_max:
                rgb_max = item[0]
print("max rgb:", rgb_max, count)
```

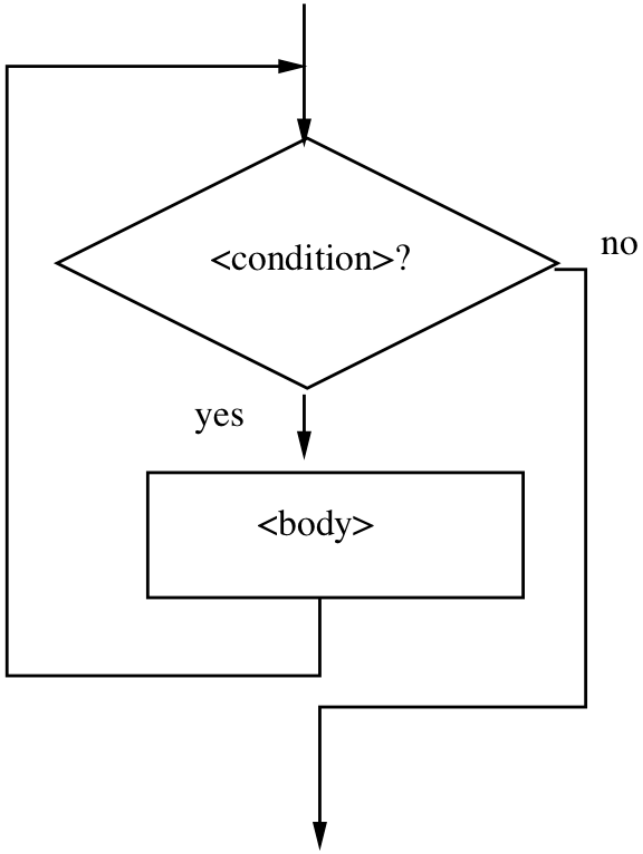
Indefinite (or conditional) iterations: `while` loops

- ✓ Repeat a set of actions an **unspecified number of times: keep doing until a certain condition is true**



Indefinite (or conditional) iterations: `while` loops

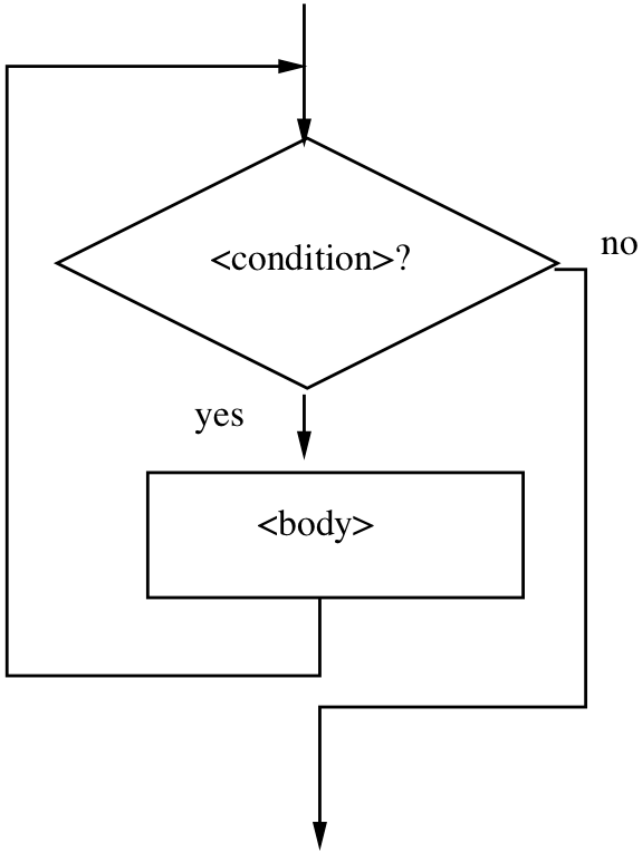
- ✓ Repeat a set of actions an **unspecified number of times: keep doing <u>until a certain condition is true</u>**




```
while condition_is_true:
    actions
```

Indefinite (or conditional) iterations: `while` loops

- ✓ Repeat a set of actions an **unspecified number of times: keep doing <u>until a certain condition is true</u>**

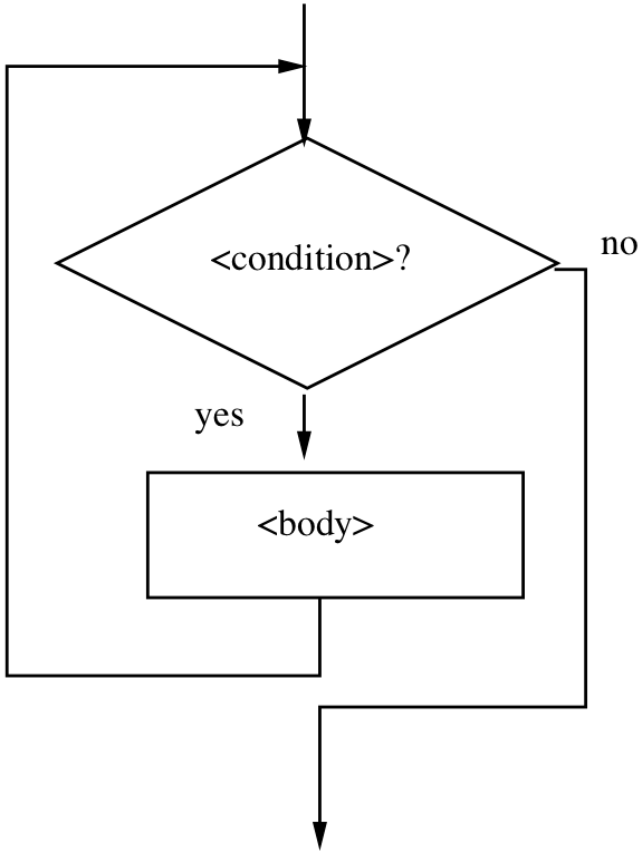



```
while condition_is_true:
    actions
```

```
i = 0
while i <= 10:
    print("Loop counter:", i)
    i += 1
```

Indefinite (or conditional) iterations: `while` loops

- ✓ Repeat a set of actions an **unspecified number of times: keep doing <u>until a certain condition is true</u>**




```
while condition_is_true:
    actions
```

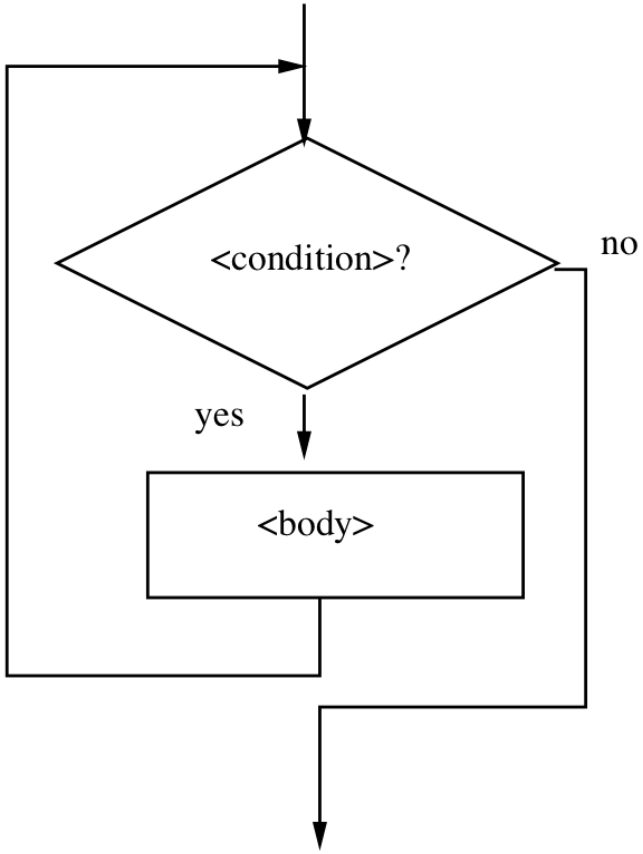
```
i = 0
while i <= 10:
    print("Loop counter:", i)
    i += 1
```

vs.

```
for i in range(10):
    print("Loop counter:", i)
```

Indefinite (or conditional) iterations: `while` loops

- ✓ Repeat a set of actions an **unspecified number of times: keep doing <u>until a certain condition is true</u>**




```
while condition_is_true:
    actions
```

```
i = 0
while i <= 10:
    print("Loop counter:", i)
    i += 1
```

vs.

```
for i in range(10):
    print("Loop counter:", i)
```

- ✓ More flexible and general than for loops, since we are not restricted to iterate over a sequence, but code can be less compact and more prone to errors …

Typical use of `while` loops

- ✓ **Sentinel loops:** keep processing data until a special value (a sentinel) that signals the end of the processing is reached

Typical use of `while` loops

- ✓ **Sentinel loops:** keep processing data until a special value (a sentinel) that signals the end of the processing is reached

```
i = 0
while i <= 10:
    print("Loop counter:", i)
    i += 1
```

Typical use of `while` loops

- ✓ **Sentinel loops:** keep processing data until a special value (a sentinel) that signals the end of the processing is reached

```
i = 0
while i <= 10:
    print("Loop counter:", i)
    i += 1
```

General computing pattern:

```
get the first data item
while item is not the sentinel:
    process the item
    get the next data item
```

Typical use of `while` loops

✓ **Sentinel loops:** keep processing data until a special value (a sentinel) that signals the end of the processing is reached

```
i = 0
while i <= 10:
    print("Loop counter:", i)
    i += 1
```

General computing pattern:

```
get the first data item
while item is not the sentinel:
    process the item
    get the next data item
```

- This type of while loops can be also implemented as `for` loops as long as we have a sound estimate of the maximum number of iterations that would be required (in the "worst" case), and then use `break` to exit the loop

Typical use of `while` loops

✓ **Sentinel loops:** keep processing data until a special value (a sentinel) that signals the end of the processing is reached

```
i = 0
while i <= 10:
    print("Loop counter:", i)
    i += 1
```

General computing pattern:

```
get the first data item
while item is not the sentinel:
    process the item
    get the next data item
```

- This type of while loops can be also implemented as `for` loops as long as we have a sound estimate of the maximum number of iterations that would be required (in the "worst" case), and then use `break` to exit the loop

```
val = 1
while val > 0.45:
    print("Value:", val)
    val *= 0.9
```

Typical use of `while` loops

✓ **Sentinel loops:** keep processing data until a special value (a sentinel) that signals the end of the processing is reached

```
i = 0
while i <= 10:
    print("Loop counter:", i)
    i += 1
```

General computing pattern:

```
get the first data item
while item is not the sentinel:
    process the item
    get the next data item
```

- This type of while loops can be also implemented as `for` loops as long as we have a sound estimate of the maximum number of iterations that would be required (in the "worst" case), and then use `break` to exit the loop

```
val = 1
while val > 0.45:
    print("Value:", val)
    val *= 0.9
```

```
max_iterations = 1000000
val = 1
for n in range(max_iterations):
    print('Value:', val)
    val *= 0.9
    if val <= 0.45:
        break
```

Example, computing the square root

```
x = 9
g = 8.5
while abs(g * g - x) > 0.1:
    print('g', g)
    g = (g + x/g)/2
print('Square root of', x, 'is', g)
```

Typical use of `while` loops

- ✓ **Input loops:** keep processing data until more data is available from some input device (e.g., interactive user, file, sensor)
  - ➢ Not clear how many inputs, hard to safely implement with a for loop

Making an average sum *interactively using user inputs*:

Typical use of `while` loops

- ✓ **Input loops:** keep processing data until more data is available from some input device (e.g., interactive user, file, sensor)
  - ➢ Not clear how many inputs, hard to safely implement with a for loop

Making an average sum *interactively using user inputs*:

```
sum = 0.0
count = 0
moredata = "yes"
```

Typical use of `while` loops

- ✓ **Input loops:** keep processing data until more data is available from some input device (e.g., interactive user, file, sensor)
  - ➢ Not clear how many inputs, hard to safely implement with a for loop

Making an average sum *interactively using user inputs*:

```
sum = 0.0
count = 0
moredata = "yes"
while moredata[0] == "y":
```

Typical use of `while` loops

- ✓ **Input loops:** keep processing data until more data is available from some input device (e.g., interactive user, file, sensor)
  - ➢ Not clear how many inputs, hard to safely implement with a for loop

Making an average sum *interactively using user inputs*:

```
sum = 0.0
count = 0
moredata = "yes"
while moredata[0] == "y":
    x = eval(input("Enter a number >> "))
```

Typical use of `while` loops

- ✓ **Input loops:** keep processing data until more data is available from some input device (e.g., interactive user, file, sensor)
  - ➢ Not clear how many inputs, hard to safely implement with a for loop

Making an average sum *interactively using user inputs*:

```
sum = 0.0
count = 0
moredata = "yes"
while moredata[0] == "y":
    x = eval(input("Enter a number >> "))
    sum = sum + x
```

Typical use of `while` loops

- ✓ **Input loops:** keep processing data until more data is available from some input device (e.g., interactive user, file, sensor)
  - ➢ Not clear how many inputs, hard to safely implement with a for loop

Making an average sum *interactively using user inputs*:

```
sum = 0.0
count = 0
moredata = "yes"
while moredata[0] == "y":
    x = eval(input("Enter a number >> "))
    sum = sum + x
    count = count + 1
```

Typical use of `while` loops

- ✓ **Input loops:** keep processing data until more data is available from some input device (e.g., interactive user, file, sensor)
  - ➢ Not clear how many inputs, hard to safely implement with a for loop

Making an average sum *interactively using user inputs*:

```
sum = 0.0
count = 0
moredata = "yes"
while moredata[0] == "y":
    x = eval(input("Enter a number >> "))
    sum = sum + x
    count = count + 1
    moredata = input("Do you have more numbers (yes or no)? ")
```

Typical use of `while` loops

- ✓ **Input loops:** keep processing data until more data is available from some input device (e.g., interactive user, file, sensor)
  - ➢ Not clear how many inputs, hard to safely implement with a for loop

Making an average sum *interactively using user inputs*:

```
sum = 0.0
count = 0
moredata = "yes"
while moredata[0] == "y":
    x = eval(input("Enter a number >> "))
    sum = sum + x
    count = count + 1
    moredata = input("Do you have more numbers (yes or no)? ")
print("\nThe average of the numbers is", sum / count)
```

Never ending iterations with `while` loops

- ✓ If the condition is always true, the loop will never end, in principle

Never ending iterations with `while` loops

- ✓ If the condition is always true, the loop will never end, in principle

```
i = 0
while i <= 10:
    print(”Hello!”)
```

Never ending iterations with `while` loops

- ✓ If the condition is always true, the loop will never end, in principle

```
i = 0
while i <= 10:
    print(”Hello!”)
```

Watch out when you define while loops!

Never ending iterations with `while` loops

- ✓ If the condition is always true, the loop will never end, in principle

```
i = 0
while i <= 10:
    print(”Hello!”)
```

Watch out when you define while loops!

- ✓ If we want to keep **looping forever** (until the computer is shutdown …)

Never ending iterations with `while` loops

- ✓ If the condition is always true, the loop will never end, in principle

```
i = 0
while i <= 10:
    print(”Hello!”)
```

Watch out when you define while loops!

- ✓ If we want to keep **looping forever** (until the computer is shutdown …)

```
while True:
    print(”Hello!”)
```

Never ending iterations with `while` loops

✓ If the condition is always true, the loop will never end, in principle

```
i = 0
while i <= 10:
    print(”Hello!”)
```

Watch out when you define while loops!

✓ If we want to keep **looping forever** (until the computer is shutdown …)

```
while True:
    print(”Hello!”)
```

- Can we generate a never ending `for` loop?
  - No! We can keep extending the sequence, but eventually we reach either a memory or a number representation limit

Nested `while` loops

- ✓ Similar possibilities / (and more) issues as when using for loops

Nested `while` loops

- ✓ Similar possibilities / (and more) issues as when using for loops

```
i = 1
while i <= 10:
    j = 0
    while j < 5:
        j += i * (i/10)
        print(i,j)
    i += 1
```

Nested `while` loops

- ✓ Similar possibilities / (and more) issues as when using for loops

```
i = 1
while i <= 10:
    j = 0
    while j < 5:
        j += i * (i/10)
        print(i,j)
    i += 1
```

Watch out how you define, initialize, and modify sentinel variables!