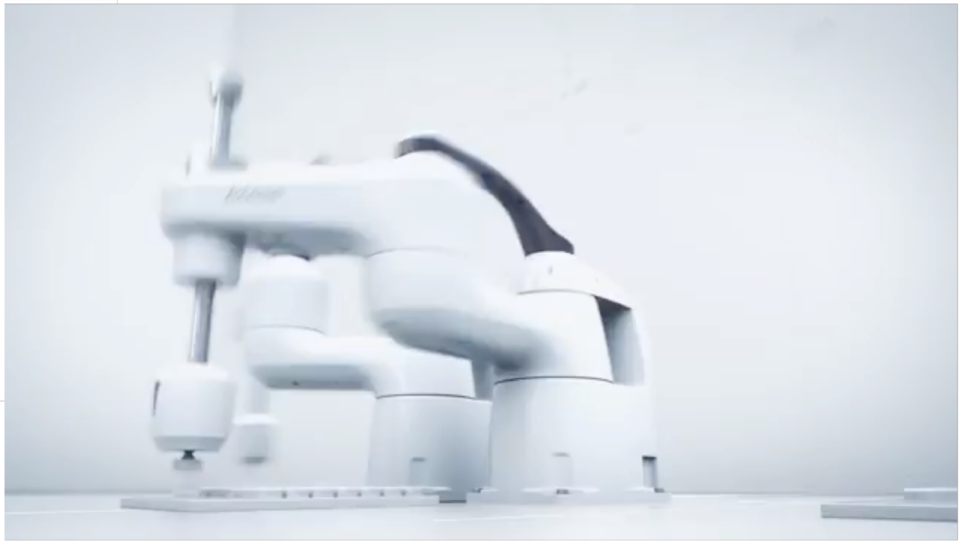
15-110 Principles of Computing – S19

Lecture 8:
Iteration 1

Teacher:
Gianni A. Di Caro

Repeating actions



Average value of N=10 numbers
1, 3, 5, 7, 4, 10, 12, -1, 8, 5

```
sum = 0
    add number to sum     } Repeat for all numbers
average = sum / N
```

Constructs and Operators for iterations in python:

for loop

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

Definite loop

while loop

```
while condition_is_true:
    actions
```

Indefinite loop

```
break

continue
```

Definite loops: for construct

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

✓ Each time the action *can* be executed on a different input parameter

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

scope of for loop (indent)

sequence: tuple, list

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

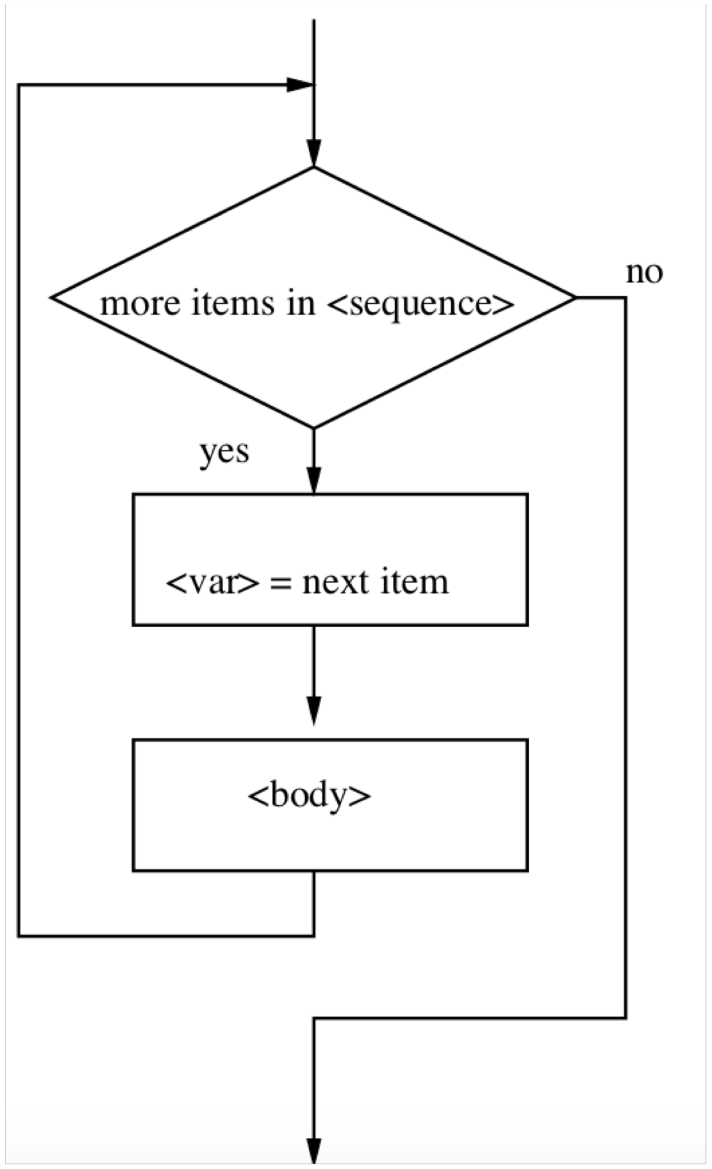
variable: *loop index*: each time the variable is set to the value of the next item in the sequence

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

Enrolled loop

`n = 6`	`n = 3`	`n = 5`	`n = 7`
`sum += n`	`sum += n`	`sum += n`	`sum += n`
Iteration 1 (sum is 6)	**Iteration 2** (sum is 9)	**Iteration 3** (sum is 14)	**Iteration 4** (sum is 21)

Definite loops: for construct



for loops: use of loop variable

- Loop actions can make a **direct use of the variable value** (it plays the role of a *changing input parameter*)

```
factorial = 1
for n in [1,2,3,4]:
   factorial *= n
print(“Factorial of”, n, “is”,  factorial)

colors = [‘r’, ‘g’, ‘y’, ‘b’, ‘g’, ‘bk’]
count_green = 0
for c in colors:
   if c == ‘g’:
      print(“A green item”)
      count_green += 1
      print(“Found”, count_green, “green items”)
print(“Last color checked was”, c)
```

Loop index variable:

✓ the variable created by the for loop <u>doesn’t disappear after the loop is done</u>

✓ it will contain the <u>last value</u> used in the for loop

`for` loops: getting a range of numbers, `range( )` function

- Sometimes, a loop **doesn’t need to to make a direct use of the variable value**, we might only need to specify how many iterations should be performed: we only need an **iteration counter**

```
eight_bits_num = 127
binary_rep = ""
div_by_two = eight_bits_num
for i in [1, 2, 3, 4, 5, 6, 7, 8]:
    bit = div_by_two % 2
    binary_rep = str(bit) + binary_rep
    div_by_two = div_by_two // 2
print('Binary representation of', eight_bits_num, 'is', binary_rep)
```

```
for i in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]:
    print(‘Hello!’)
```

only need to say how many times the action should be iterated (10, 8)

- More in general, it might be useful to **automatically generate lists of numbers (integer**), that can be also used as input parameters during iterations → `range(start, end, step)` function
- **Counted loops**

`for` loops: getting a range of numbers, `range()` function

- `range(start, end, step)` generates the integer numbers in the specified range
  - `start, end, step` must be **integer**
  - range is *exclusive*: the **last number is not generated**

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

✓ `range(s, n, ss)` generates the integers between `s` and `n-1` with a step of `ss`

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

✓ `range(s, n)` is equivalent to `range(s, n, 1)`

✓ generates the successive integers between `s` and `n-1`

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

✓ `range(n)` is equivalent to `range(0, n, 1)`

✓ generates the successive integers between `0` and `n-1`

`for` loops: getting a range of numbers, `range( )` function

- `range(s, n, ss),` **rules for the arguments:**
  - **Increasing ranges**: `n > s, ss` must be *positive*

```
for i in range(1,5,1):
    print(“Counter:”,i)    →  1, 2, 3, 4
```

```
for i in range(1,5,-1):
    print(“Counter:”,i)    →  Nothing
```

```
for i in range(-10):
    print(“Counter:”,i)    →  Nothing
```

```
for i in range(10, 1):
    print(“Counter:”,i)    →  Nothing
```

  - **Decreasing ranges**: `s > n, ss` must be *negative*

```
for i in range(5,1,-1):
    print(“Counter:”,i)    →  5, 4, 3, 2
```

```
for i in range(5,1,1):
    print(“Counter:”,i)    →  Nothing
```

`for` loops: getting a range of numbers, `range()` function

- Watch out: `range()` **doesn't generates all numbers at once, i.e., it doesn't return a list with the numbers!**
- Range is a **generator**

```
numbers = range(5)
print(len(numbers))          → 5
print(numbers[2])            → 2
print((2 in numbers))        → True
for i in numbers:            → 0, 1, 2, 3, 4
    print("Counter:", i)
```

it *looks* like a list, the behavior is the expected one ...

```
numbers = range(5)
print(numbers)               → range(0, 5)
```

- It's a sequence because the object supports membership testing, indexing, slicing and has a length, just like a list or a tuple.
- Unlike a list or a tuple, it doesn't actually contain all integers in the sequence in memory, making it *virtual*, elements are returned on demand

```
numbers = list(range(5))
print(numbers)               → 0, 1, 2, 3, 4
```

Counting and summing `for` loops, `sum( )` function

- A typical use of loops is for **counting items** (e.g., that satisfy certain conditions)

```
counter = 0
for i in data_list:
    if i < 0:
        counter += 1
```

`counter:` accumulator

- Another typical use of loops is for **summing up item values** (e.g., that satisfy certain conditions)

```
sum = 0
for i in data_list:
    sum += i
```

- `sum(l)` function does the same thing: sums up the element of list

```
sum = 0
for i in data_list:
    if i[1] > 0:
      sum += i[0]
```

```
data_list = [(3,2), (-1, -3), (0,-2), (1,1), (4,6)]
```

`for` loops: examples for creating a data list

- Set up a list of n elements such that the element at position i has value i

```
my_list = list(range(10))
```

Without using range():

```
my_list = [0] * 10
counter = 0
for i in my_list:
    my_list[i] = counter
    counter += 1
```

Loop ***counter*** **variable**

initialization

increment

`for` loops: examples for creating a data list

- Set up a list of n elements such that the element at position i has value $\sum_{k=0}^{i} k$

```
incremental_sum = [0, 1, 3, 6, 10, 15, 21, ..., 4999500]
```

seed the computation

```
n = 10
incremental_sums = [0]*n
incremental_sums[0] = 0
for i in range(1, n):
    incremental_sums[i] = incremental_sums[i-1] + i
```

Or, if we know Gauss formula: $s[n] = \frac{n(n+1)}{2}$

```
n = 10
gauss_sums = [0]*n
for i in range(n):
    gauss_sums[i] = (i*(i+1))//2
```

for loops: examples for manipulating a data list

- Scale all values of a list by a factor depending on the position in the list 0 (e.g., price discount rate depending on recency of the data)

```
my_data = [1, 5, 2, 9, 8, 11]
for i in range(len(my_data)):
   my_data[i] *= (0.9**i)
```

$\longleftarrow$ combining len() with range()

- Extract all items (with their index) that satisfy a condition (e.g., higher than a reference value)

```
my_data = [1, 5, 2, 9, 8, 11]
extracted_data = []
for i in range(len(my_data)):
   if my_data[i] > 4.5:
      extracted_data.append((my_data[i],i))
print(extracted_data)
```

→ [(5, 1), (9,3), (8,4), (11,5)]

`for` loops: examples for manipulating a data list

- Range over characters (based on their UTF-8 numeric code)

```
def character_range(char_start, char_end):
    char_list = []
    for char in range(ord(char_start), ord(char_end)+1):
        char_list.append(char)
    return(char_list)

for letter in character_range('a', 'z'):
    print( chr(letter) )
```

→ a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z,

`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:”, n)
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