15-110 Principles of Computing – F19

Lecture 15: Iteration 2

Teacher:
Gianni A. Di Caro

جامعة كارنيجي ميلون في قطر
Carnegie Mellon University Qatar

So far about Python ...

- Basic elements of a program:
  - Literal objects
  - Variables objects
  - Function objects
  - Commands
  - Expressions
  - Operators

- Utility functions (built-in):
  - `print(arg1, arg2, …)`
  - `type(obj)`
  - `id(obj)`
  - `int(obj)`
  - `float(obj)`
  - `bool(obj)`
  - `str(obj)`
  - `input(msg)`
  - `len(non_scalar_obj)`
  - `sorted(seq)`
  - `min(seq), max(seq)`
  - `range(start, end, step)`

- Object properties
  - Literal vs. Variable
  - Type
  - Scalar vs. Non-scalar
  - Immutable vs. Mutable
  - Aliasing vs. Cloning

- Conditional flow control
  - ```
    if cond_true:
        do_something
    ```
  - ```
    if cond_true:
        do_something
    else:
        do_something_else
    ```
  - ```
    if cond1_true:
        do_something_1
    elif cond2_true:
        do_something_2
    else:
        do_something_else
    ```

- Flow control: repeated actions
  - ```
    for x in seq:
        do_something
    ```

- Data types:
  - `int`
  - `float`
  - `bool`
  - `str`
  - `None`
  - `tuple`
  - `list`

- Relational operators
  - `>`
  - `<`
  - `>=`
  - `<=`
  - `==`
  - `!=`

- Logical operators
  - `and`
  - `or`
  - `not`

- Operators:
  - `=`
  - `+`
  - `+=`
  - `-`
  - `/`
  - `*`
  - `*=`
  - `//`
  - `%`
  - `**`
  - `[ ]`
  - `[ : ]`
  - `[ : : ]`

- String methods
- List methods

Recap: `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`

Recap: `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

  - **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(10, 1):
    print('Counter:',i)
```
→ Nothing

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

`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 using range

```
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
```

initialization

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

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)
```


combining `len()` with `range()`

`for` loops: examples for manipulating a data list

- 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

`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

```
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]`

…

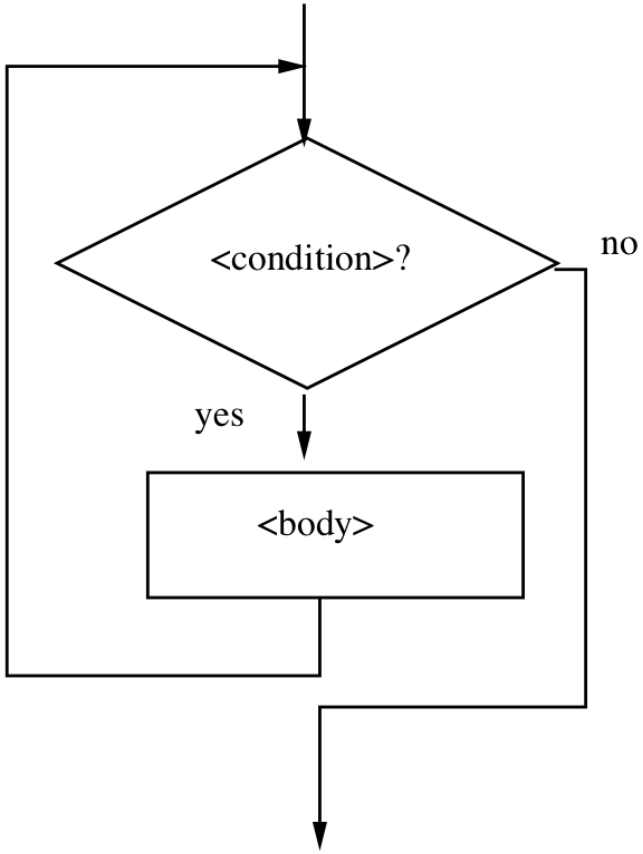
Iteration 7
`n = 3`

Sequence to go:
`[]`

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

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

```
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*:

```
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

```
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

```
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!

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

```
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: use range()

- 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)
```

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

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

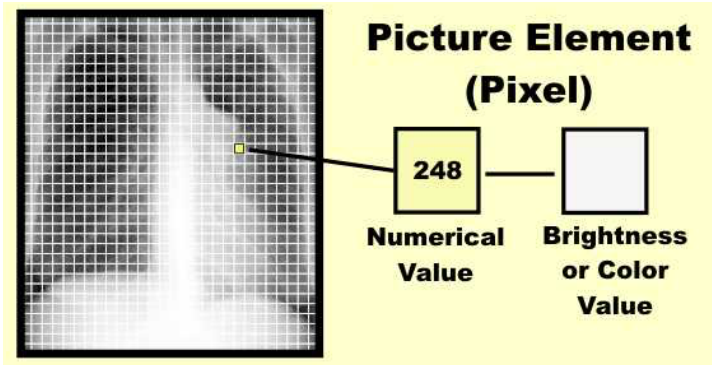
3 x 3 matrix

- 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

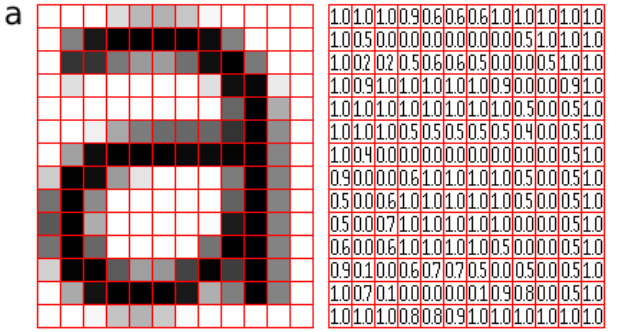
- Exemplary use of matrices in computing: digital image processing!

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

- **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!

Check you knowledge: Iterating over (all) the elements of a list

- **Modify** or **use/extract** all values (or all values that satisfy a given condition) of a large list according to a given pattern that depends on *item* values
  - Scale all values by a factor 0.5 (e.g., price discount rate)

```
articles = [['book', 15], ['toy', 25], ['cookies', 8], ...]
articles ← [['book', 7.5], ['toy', 12.5], ['cookies', 4], ...]
```

  - Extract all items that are older than one five days (e.g., food articles)

```
articles = [['cheese', 10], ['milk', 2], ['butter', 8], ...]
expiring ← [['cheese', 10], ['butter', 8], ...]
```

  - Find items satisfying a condition and perform an incremental operation (e.g., sum money invested in edge funds)

```
investments = [['EF1', 100000], ['B1', 50000], ['EF4', 2000], ...
capital_in_EF ← 100000 + 2000 + ...
```

Check your knowledge: Iterating over (all) the elements of a list

- **Initialize** a large list according to a given pattern that might depend on *index* values

  - Set up a list of n (e.g., 1000) elements such that the element at position i has value i

    ```
    sequential_numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, ..., 999]
    ```

  - Set up a list of n (e.g., 1000) 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]
    ```

  - Set up a list of n (e.g., 256) elements such that each element is a unique string of 0 and 1

    ```
    binary = ['00000000', '00000001', '00000010', ..., '11111111']
    ```