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

  - **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(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

```
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)

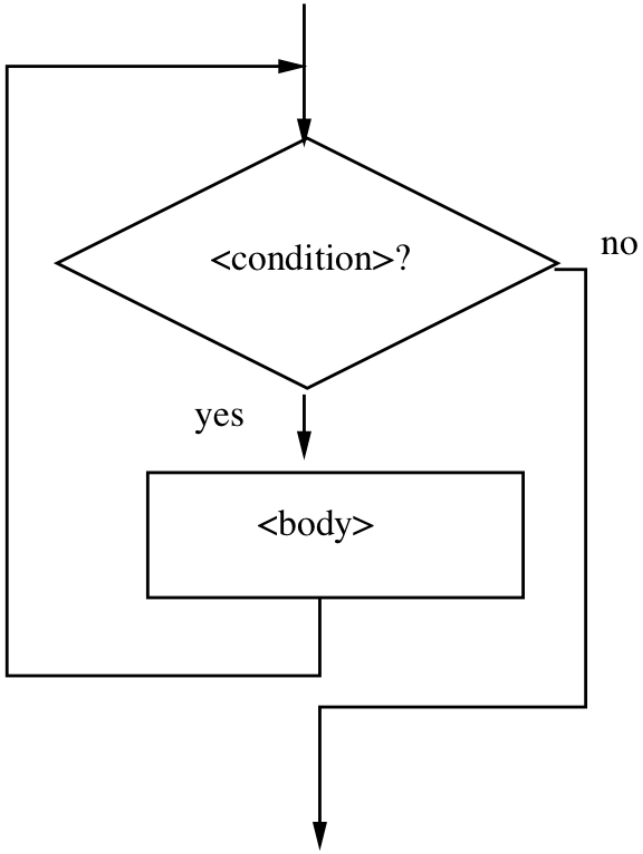
```
def character_range(char_start, char_end):
    char_list = []
    for char in range(ord(char_start), ord(char_end)+1):
        char_list.append(char)
    return(char_list)

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,

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 <u>maximum number of iterations that would be required</u> (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 <u>loop will never end</u>, 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