15-110 Principles of Computing – F19

Lecture 13:
Tuples, Lists 4

Teacher:
Gianni A. Di Caro

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)`

- 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
    ```

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

- Relational operators
  - `>`
  - `<`
  - `>=`
  - `<=`
  - `==`
  - `!=`
  - Logical operators
    - `and`
    - `or`
    - `not`

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

- String methods
- List methods
    ```

Useful operations: reverse() method

- `reverse()` : Changes (*in-place*) the list `l` (not applicable to tuples!) putting the elements in the reverse order compared to the original list

```
numbers = [1, 4, 2, -7, 0, 6]
numbers.reverse()
```

→ `numbers` list is now: [6, 0, -7, 2, 4, 1]

- Other way to obtain the same macroscopic result using `[]` operator:

```
numbers = [1, 4, 2, -7, 0, 6]
numbers = numbers[::-1]
```

→ `numbers` list is now: [6, 0, -7, 2, 4, 1]

- Watch out: a list with a *new* identity is being created (but the *macroscopic* effect is the same)

```
numbers = [1, 4, 2, -7, 0, 6]
print(id(numbers))                  → 4729970376
numbers = numbers[::-1]
print(id(numbers))                  → 4729921992
```

→ identity values (*memory addresses of list's content*) will be different on different executions / computers

Useful operations: sort () method

- `sort(key, reverse)` : Changes (*in-place*) the list `l` (not applicable to tuples!) with the elements sorted according to the (optional) criterion `key` for comparing the items ; the (optional) parameter `reverse`, if set to `True`, provides the result in *descending* order

```
numbers = [1, 4, 2, -7, 0, 6]
```

`numbers.sort()` → items are all integer, no criterion is required, `numbers` is sorted ascending:

[-7, 0, 1, 2, 4, 6]

```
cars = ['toyota', 'BMW', 'nissan']
```

`cars.sort(reverse=True)` → items are str, no criterion is required, `cars` is sorted descending:

['toyota', 'nissan', 'BMW']

```
mix = ['toyota', 'BMW', 'nissan', 3]
```

`cars.sort()` → error! items have mixed types, a comparison criterion is required

Useful operations: sort () method

- **Note:** A *list of lists/tuples of primitive types* is sorted according to the first element(s) of each list/tuple

```
my_tuples = [(1,2), (5,7,8), (-1,), (0,9,1,3)]
my_tuples.sort()
```

→ `[(-1,), (0, 9, 1, 3), (1, 2), (5, 7, 8)]`

- If the first element is the same, sorting is based on the second (and so on)

```
my_tuples = [(-1,2), (5,7,8), (-1,3), (0,9,1,3)]
my_tuples.sort()
```

→ `[(-1,2), (-1,3), (0, 9, 1, 3), (5, 7, 8)]`

- *Ties* do not matter since the items become indistinguishable

```
my_tuples = [(-1,2), (5,7,8), (-1,2), (0,9,1,3)]
my_tuples.sort()
```

→ `[(-1,2), (-1,2), (0, 9, 1, 3), (5, 7, 8)]`

sorted() function: copy *and* sort

```
l = [2,4,1]
l.sort()
b = l
print(l, b)                 → [1, 2, 4] [1, 2, 4]
print(id(l), id(b))         → 4584102856 4584102856
```

`sort()` *method* operates in-place, it changes the list `a` by sorting it

- Function `sorted(seq)`: works for any sequence (list, tuple, string) and returns a **list which is sorted copy of the original sequence,** the original object is *not modified*

```
l = [2,4,1]
b = sorted(l)
print(l, b)                 → [2, 4, 1] [1, 2, 4]
print(id(l), id(b))         → 4989597000 4584103560
```

`sorted()` *function* makes a copy of the object and returns it sorted

```
print(sorted('classroom'))  → ['a', 'c', 'l', 'm', 'o', 'o', 'r', 's', 's']
```

- Function `sorted(seq, key, reverse)`: same optional arguments as `sort()` method

Useful operations: max(), min() functions

- `min(t, key)` : Returns the **item** of the list/tuple `t` with **minimum value**
  - Without a key (optional criterion for comparison), it can be applied only to homogeneous lists/tuples (all elements of the same type)
  - Return type depends on the type of the items

`prime_numbers = [1, 3, 5, 7, 11]` → generates an error (how to compare different items?)

`n = max(prime_numbers)` → `n` is an integer of value 11, the item of highest value

`c = max('red', 'green', 'blue')` → `c` is a string of value 'red', corresponding the item of highest code value (starting from 'r') in UTF-8

`l = max('a', 'c', 'C')` → `l` is a string of value 'c', that has the highest code value in UTF-8

`logical = max(True, False, True)` → `logical` is a boolean of value `True` (1)

`x = max(1, 3, True, 'red')` → generates an **error** (how to compare different items?)

Use of a *key* for item comparison in sort(), max()/min(),…

- `sort(key, reverse)` : Changes (in-place) the list `l` (not applicable to tuples!) with the elements sorted according to the (optional) criterion `key` for comparing the items ; the (optional) parameter `reverse`, if set to `True`, provides the result in *descending* order

- **key** parameter:
  - specifies a **function** to be called **on each list element** prior to making comparisons

    → a function that takes a single input parameter, `F(x)`
  - the **return value** of the function is the key used for *comparison purposes*
  - return value must be a **primitive type**, such that python knows how to make comparisons among keys

```
my_tuples = [(1,2), (5,7,8), (1,), (2,2,0,0)]
my_tuples.sort(key = len)
```

→ [(1,), (1, 2), (5, 7, 8), (2, 2, 0, 0)]

```
my_strings = ['hello', 'Good morning', 'I am', 'list example', 'zzTop']
my_strings.sort(key = str.lower)
```

→ ['Good morning', 'hello', 'I am', 'list example', 'zzTop']

Use of a key for item comparison in sort(), max()/min(),…

```
def cmp_on_second_element(item):
    return item[1]

my_tuples = [(1,2), (5,7,8), (1,0), (2,2,0,0)]
my_tuples.sort(key = cmp_on_second_element)
```

Write your own **custom function** for performing comparisons

```
def cmp_on_second_element(item):
    if len(item) >= 2
        return item[1]
    else
        return -1

my_tuples = [(1,2), (5,7,8), (1,), (2,2,0,0)]
my_tuples.sort(key = cmp_on_second_element)
```

What if the list/tuple item is not long enough?

Watch out: We must ensure that `item` is a list/tuple, otherwise `len(item)` would return an error in this example!

join(seq) string method: a string out a sequence

- `join(seq)` *string* method: given an *iterable* object type `seq` (e.g., `tuple, list, dict, set`) containing only string elements, `s.join(seq)` returns a string in which the elements of `seq` have been joined by `s` as separator

```
l = ['1','2','3','4']
sep = "-"
s = sep.join(l)
print(s)        →  1-2-3-4
```

```
l = ['1','2','3','4']
sep = ''
s = sep.join(l)
print(s)        →  1234
```

```
l = ['1','2','3','4']
s = ''.join(l)
print(s)        →  1234
```

```
l = ('1','2','3','4')
sep = ", "
s = sep.join(l)
print(s)        →  1, 2, 3, 4
```

```
l = ('This','is','a','story')
sep = " "
s = sep.join(l)
print(s)        →  This is a story
```

A string `s` is treated as a sequence of characters

```
s = '123'
sep = 'abc'
print(sep.join(s))  → 1abc2abc3
```

```
s = 'abc'
sep = '123'
print(sep.join(s))→ a123b123c
```

Iterating over (all) the elements of a list

- We might want to perform actions **on the entire list**, potentially on all items, or subsets of items

- **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']
    ```

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 week (e.g., food articles)

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

  - Find items satisfy 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 + ...
```

Iterating over (all) the elements of a list

- How do we perform these list-level operations? → **Iterators**
- Constructs to repeat actions without explicitly enumerating all the elements to act upon

Operators for iterations:

- ✓ `for i in sequence`
- ✓ `while condition_is_true`

Next time!