UNIVERSITY OF TWENTE.

Lists and tuples

Data containers and how to work with them

1

Today's objectives

After this lecture, you will be able to:

1. Lists:
   - Understand the syntax of lists
   - Differentiate between a list and a tuple
   - Apply basic operations to lists
2. Tuples
   - Understand how to work with tuples

Contents

Lists as data structures

- Operations: membership, indexing, slicing, adding, multiplication, iteration, nested lists, member deletion
- Methods: sorting lists
- Special: tuples
  lists as matrices
  clones

Tuples as data structures

- Differences between Lists and Tuples

UNIVERSITY OF TWENTE.

3

Built-in Python data structures

Python provides a number of built-in container (or collection) types that can be used to group together objects.

Sequences

- Types: strings, lists, tuples
- Operations: indexing, slicing, adding, multiplying, iteration & membership testing

Dictionaries

- Map keys to values through an index over the keys
- Suitable for unstructured data that has a look-up nature

Sets

- Unordered collection of elements without look-up nature

UNIVERSITY OF TWENTE.

4

Lists

The most versatile container is the *list*, which can be written as a sequence of comma-separated values (items) together set between square brackets […]

List items can be of different types:

list_num = [1, 2, 4, 8, 16, 32.0, 64.0, 128, 256, 512.0]

list_str = ["dear", "students", "this", "is", "a", "list", "of", "strings"]

list_mixed = ["dear", 32.0, "this", "is", "a", 512.0, "!", 2016]

UNIVERSITY OF TWENTE.

5

Operations: Indexing

If *fruits* is a list, then *fruits[3]* is the 4th member in the list, and *fruits[0]* the first.

```
                0          1            2          3           4
fruits = [  "orange",  "watermelon",  "lemon", "coconut", "pineapple",
            "banana", "pomegranate", "kiwi",   "grapes",  "apricot" ]
                5          6            7          8           9

               -10         -9           -8         -7          -6
fruits = [  "orange",  "watermelon",  "lemon", "coconut", "pineapple",
            "banana", "pomegranate", "kiwi",   "grapes",  "apricot" ]
               -5          -4           -3         -2          -1
```

We can refer to elements in the list forwards and backwards, so *fruits[4]* is the same as *fruits[-6]* (at position 6 from the end)

UNIVERSITY OF TWENTE.

6

Operations: Slicing

Slicing is done with the : operator

A slice [n:m] is a segment of the sequence.
- From the *n-th* item (including) until the *m-th* item (excluding)
- Look at it as [start:stop]

```
fruits = ["orange", "watermelon", "lemon", "coconut", "pineapple",
          "banana", "pomegranate", "kiwi", "grapes", "apricot" ]

print(fruits[4:8])                    # Think of index to point between the elements
["pineapple", "banana", "pomegranate", "kiwi"]

print(fruits[-3:])
["kiwi", "grapes", "apricot"]

print(fruits[-3:-1])
["kiwi", "grapes"]

print(fruits[:4])
["orange", "watermelon", "lemon", "coconut"]
```

UNIVERSITY OF TWENTE.

8

Operations: Slicing

We can slice, concatenate and multiply lists.

```
fruits = ["orange", "watermelon", "lemon", "coconut", "pineapple",
          "banana", "pomegranate", "kiwi", "grapes", "apricot" ]


print(fruits[1:-3])
["watermelon", "lemon", "coconut", "pineapple", "banana", "pomegranate"]

print(fruits[:2] + ["peach", 99*10])
["orange", "watermelon", "peach", 990]

print(2 * fruits[3:6] + ["cherry"])
["coconut", "pineapple", "banana", "coconut", "pineapple", "banana", "cherry"]
```

UNIVERSITY OF TWENTE.

9

Operations: Slicing with a step

Special case of slicing, using [start:stop:step]

```
fruits = ["orange", "watermelon", "lemon", "coconut", "pineapple",
          "banana", "pomegranate", "kiwi", "grapes", "apricot" ]


# Start in position 1, stop at end, and get every second
print(fruits[1::2])
["watermelon", "coconut", "banana", "kiwi", "apricot"]

# Start at the beginning, stop at end, make steps backwards (inversion)
print(fruits[::-1])
["apricot", "grapes", "kiwi", …, "watermelon", "orange"]

# Start in position 2, end in position 8, make steps backwards (empty)
print(fruits[2:8:-1])
[]
```

UNIVERSITY OF TWENTE.

10

Operations: Membership

Is an element in the list? Test is conducted with **in** keyword

```
list_str = ["dear", "students", "this", "is", "a", "list", "of", "str"]

print("dear" in list_str)

True

print("Joe" in list_str)

False
```

UNIVERSITY OF TWENTE.

11

List boundaries

```
fruits = [   "orange", "watermelon", "lemon", "coconut", "pineapple", "banana",
             "pomegranate", "kiwi", "grapes", "apricot" ]

print(fruits[27])
IndexError: list out of range

print(fruits[-15])
IndexError: list out of range

print(fruits[20:30])
[]

print(fruits[-20:20])
["orange", "watermelon", "lemon", …, "apricot"]
```

UNIVERSITY OF TWENTE.

12

Lists can be changed

Unlike strings, which are immutable, it is possible to change individual elements of a list:

```
fruits = [   "orange", "watermelon", "lemon", "coconut", "pineapple", "banana",
             "pomegranate", "kiwi", "grapes", "apricot" ]

# Let's change an element
fruits[1] = "PEAR"
print(fruits)
["orange", "PEAR", "lemon", "coconut",…, "grapes", "apricot" ]

# You can even change more than one element at a time
fruits[2:4] = ["cherry", "mango"]
print(fruits)
["orange", "PEAR", "cherry", "mango", "pineapple", …, "apricot"]
```

UNIVERSITY OF TWENTE.

13

List length and empty lists

Length of lists can be obtained using function len()

```
fruits = [   "orange", "watermelon", "lemon", "coconut", "pineapple", "banana",
             "pomegranate", "kiwi", "grapes", "apricot" ]

print(len(fruits))
10

# function len() can only be applied to sequences
print(len(fruits[0]))

# … and numbers are NOT sequences
print(len(9999))
TypeError: object of type 'int' has no len()
```

Empty lists can be initialized in two ways:

1) fruits = []
2) fruits = list()

For both, we have *len(fruits) = 0*

UNIVERSITY OF TWENTE.

14

Nested lists

It is possible to make lists that have lists as members

```
fruits = ["orange", "watermelon", "lemon", "coconut", "pineapple",
          "banana", "pomegranate", "kiwi", "grapes", "apricot" ]
food = ["bread", "milk", "rice", fruits, "olive oil", "lentils"]
print(len(food))
6
print(len(food[3]))                    # This is the position of the fruit list
10
print(len(food[3][0]))                 # This is the position of "orange" (a string)
6

# You can keep adding elements, regardless of the nesting
food = food + ["beans"]                # We add "beans to the food list"
print(len(food))
7
food[3] = food[3] + ["lime"]           # We add "lime" to the fruits list
print(len(food))
7
print(len(food[3]))
11
```

UNIVERSITY OF TWENTE.

15

List methods

Python has different types of functions:

1. Built-in functions: *len(), sum(), min(), max()*

2. Custom functions: *calculate_least_squares(), connect_to_my_database(), read_my_CSV_file()*

3. Functions imported from a module: *math.cos(), datetime.datetime(), operator.itemgetter()*

4. Methods
   - Functions associated with an object class (such as *list*)
   - Called like this: *object.method(arguments)*
   - Example: *mylist.append(x)* *fruits.append("strawberries")*

UNIVERSITY OF TWENTE.

16

List methods

Two types of list methods:

- Void functions (in-place operation)

```
.append(x)      ➔ fruits.append("blueberries")
.extend(L)      ➔ fruits.extend(["cranberries","mango"])
.insert(i, x)   ➔ fruits.insert(0, "avocado")
.remove(x)      ➔ fruits.remove("banana")
.sort()         ➔ fruits.sort()
.reverse()      ➔ fruits.reverse()
.clear()        ➔ fruits.clear()          # or just: fruits = []
```

- Functions with returned value**:**

```
.index(x)       ➔ fruits.index("avocado")   # returns 0
.count(x)       ➔ fruits.count("avocado")   # returns 1
.pop(i)         ➔ fruits.pop(0)             # returns "avocado"
```

UNIVERSITY OF TWENTE.

17

List methods

Method sort() vs. function sorted()

- *sort()*
  - Sorts in place
  - Object is modified
  - Returns *None*
- *sorted(list)*
  - Returns a sorted list
  - Old list remains unsorted
- In practice:
  - *sort(fruits)*
  - *sorted_fruits = sorted(fruits)*

UNIVERSITY OF TWENTE.

18

Deleting elements

Deletion of a list element is done with **del** keyword, using its position

Used to remove individual elements or slices

```
fruits = ["orange", "watermelon", "lemon", "coconut", "pineapple","banana",
          "pomegranate", "kiwi", "grapes", "apricot" ]

# First we delete an element
del fruits[0]                    # We can also remove it by fruits.remove("orange")
print(fruits)
[ "watermelon", "lemon", "coconut", … , "grapes", "apricot" ]


# Now we delete a slice
del fruits[1:5]
print(fruits)
[ "watermelon", "pomegranate", "kiwi", "grapes", "apricot"]
```

UNIVERSITY OF TWENTE.

19

Deleting elements

We can also clear a list with del.

del fruits[:]
print(fruits)
[ ]

And even delete (forget) variables

del fruits

From here on, the variable no longer exists (until you define it anew)

print(fruits)
NameError: name 'fruits' is not defined

UNIVERSITY OF TWENTE.

20

Lists as matrices

A *list of lists* can be used as a matrix:

m = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
print(m)
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]

We can print the elements of the matrix separately

print(m[1])
[4, 5, 6]
print(m[1][2])
6
print(m[2][0])
7
print(m[2][1])
8

list[row][column]

UNIVERSITY OF TWENTE.

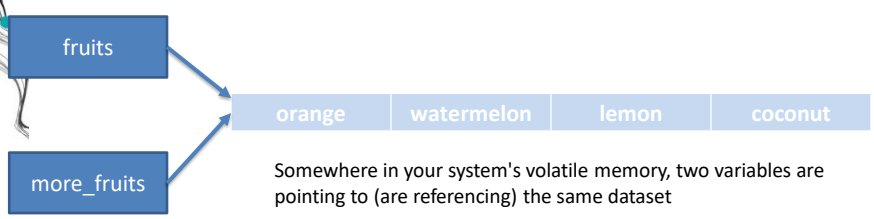
(Remember: Python starts indexing at zero!)

21

Lists are objects

A list variable is a reference to an object in Python
- Multiple variables can reference the same object
- The referenced object is "aliased"
- Each variable that points to that object is an alias

fruits = ["orange", "watermelon", "lemon", "coconut"]
more_fruits = fruits



Somewhere in your system's volatile memory, two variables are pointing to (are referencing) the same dataset

UNIVERSITY OF TWENTE.

22

Lists are objects

Watch this behaviour

fruits = ["orange", "watermelon", "lemon", "coconut"]
more_fruits = fruits

more_fruits.append("apple")

print(fruits)
["orange", "watermelon", "lemon", "coconut", "apple"]

print(more_fruits)
["orange", "watermelon", "lemon", "coconut", "apple"]

So, how to make two separate copies
of the same list? I.e., when not sharing memory.

UNIVERSITY OF TWENTE.

23

Cloning lists

We can clone the list
- Cloning a list forces Python to create a new object in memory
- The resulting copy is independent of the original list

We can do this in two ways:
- Using the slicing operator: ":"
- Using the list container: list()

```
fruits = ["orange", "watermelon", "lemon", "coconut"]
more_fruits = fruits[:]                          # or more_fruits = list(fruits)
more_fruits.append("apple")

print(fruits)
["orange", "watermelon", "lemon", "coconut"]

print(more_fruits)
["orange", "watermelon", "lemon", "coconut", "apple"]
```

UNIVERSITY OF TWENTE.

24

Cloning lists

But be careful

- Objects supporting in-place changes (e.g., lists, dictionaries, sets), can yield unexpected results.
- A change in a variable can impact other variables

UNIVERSITY OF TWENTE.

25

Tuples

A tuple is like a list, but it is **immutable**.

- Once it is defined, it cannot be changed
- It is also a sequence data type
- You can create a tuple in two ways:
  - Using parentheses ( )
  - Using the container constructor function tuple()

```
fruits = ("orange", "watermelon", "lemon", "coconut")          # this is a tuple, not a list
print(fruits[1])                                               # but we can index it, like a list
"watermelon"

fruits = tuple("orange", "watermelon", "lemon", "coconut")
print(fruits[2])
"lemon"
```

UNIVERSITY OF TWENTE.

26

Tuples

An attempt to modify a tuple will raise an error

```
fruits = tuple("orange", "watermelon", "lemon", "coconut")
fruits[2] = "mango"
TypeError: 'tuple' object does not support item assignment
```

You can convert seamlessly between lists and tuples

```
fruits = tuple("orange", "watermelon", "lemon", "coconut")
fruits = list(fruits)                                  # at this point, it is mutable
fruits = tuple(fruits)                                 # at this point no longer
```

A tuple of only one element is defined as follows with a trick:

```
fruits = ("orange",)                                   # note the comma!
                                  # without the comma the (…) would
                                  #just be interpreted as expression grouping
```

UNIVERSITY OF TWENTE.

27

Summary

A list in Python:

- Is a heterogeneous collection of items
- Shares common operations with other sequences (membership test, indexing, slicing, …)
- Has several handy methods (append, extend, insert, …)
- Can be sorted using method .sort() or function sorted()
- Can be cloned using slicing or the list container constructor
- Can be nested (e.g., to build matrices)
- Is mutable, while a tuple is immutable

UNIVERSITY OF TWENTE.

28