Python Programming Language – lists, tuples, dictionaries

Assoc. Prof. Krzysztof Małecki

kmalecki.zut.edu.pl
p. 308 (WI2)

Sorting

arranging data in the order resulting from a certain criterion

- one of the elementary algorithmic activities
- there are many algorithms with different complexity and performance
- a grateful topic for research and reflection

Sorting

1, 13, 0, 9, 5

- rising (non-decreasing) → 0, 1, 5, 9, 13
- decreasing (non-growing) → 13, 9, 5, 1, 0
- natural
  - File.txt
  - File1.txt
  - File2.txt
  - File3.txt
  - File10.txt
  - File24.txt
- ...

A trivial task

- sorting of two numbers in non-decreasing way

```
a=int(input())
b=int(input())
#
# ?
#
```

A trivial task

- sorting of two numbers in non-decreasing way

```
a=int(input())
b=int(input())
if a<b:
    print(a,b)
else:
    print(b,a)
```

A bit more interesting task

- sorting of three numbers in non-decreasing way

```
a=int(input())
b=int(input())
c=int(input())
#
# ?
#
```

A bit more interesting task

- sorting of three numbers in non-decreasing way

```
a=int(input())
b=int(input())
c=int(input())
if a <= b and a <= c:
    print(a, end=" ")
    if b > c:   print(c,b)
    else:       print(b,c)
elif b <= a and b <= c:
    print(b, end=" ")
    if a > c:   print(c,a)
    else:       print(a,c)
elif c <= a and c <= b:
    print(c, end=" ")
    if(b > a):  print(a,b)
    else:       print(b,a)
else:  print(a,b,c)
```

You can write like here if there is only one statement behind `if` ... but it is not the recommended style because the code is not readable

A difficult task

- sorting of four numbers in non-decreasing way

```
a=int(input())
b=int(input())
c=int(input())
d=int(input())
#
# ?
#
```

An almost impossible task

- sorting of five numbers in non-decreasing way

```
a=int(input())
b=int(input())
...
e=int(input())
#
# ???!!!
#
```

What is the conclusion?

- sorting and other activities of a similar nature **cannot** be performed on single variables

- you need **a container** that can store **more than one value** at a time

A list

What is a list?

- a **list** is a set of data
- in other languages known as a **table**
- it is a variable of variables
- an ordinary variable → one value (e.g. number)
- a list → any number (including zero) of **numbered data**
- the amount of data in the list may change over time

```
variable = 12
list = [1,3,7,15]
```

A list

- list items are **numbered**
- the first element of the list is always **numbered by 0**
- the last element - one less than the number of items in the list
- the number of the item in the list → **index**

A list

```
a = list[0]
b = list[a]
c = list[2 * a + 1]
d = list[ list[0] ]
e = list[ int(input()) ]
```

index can be a literal, variable, or even
any complex expression!

Some examples

```
list = [1, 10, 4, 5, 7]
el = list[0]
list[1] = el

el = list[10]
```

it cannot be successful - the list does not have an element with this index

```
Traceback (most recent call last):
  File "z.py", line 2, in <module>
    el = lista[10]
IndexError: list index out of range
```

Some examples

```
list = [1, 10, 4, 5, 7]
list[10] = 1
```

it will also fail - for the same reason as before

```
Traceback (most recent call last):
  File "z.py", line 2, in <module>
    lista[10] = 1
IndexError: list assignment index out of range
```

Some examples

...but

```
list = [1, 10, 4, 5, 7]
el = list[-1]
```

Surprise - it works!
indexing with negative numbers means **indexing from the end**, that is:

`list[-1]` – the last item
`list[-2]` – one before the last element
etc.

Example: the sum of all list items – ver. #1

```
list = [0, 3, 12, 8, 2]
sum = 0
for i in range(5):
      sum = sum + list[i]
print(sum)
```

Why it works?

for – variant #3

```
for x in list:
    code1
else:
    code2
```

- variable *x* will take on all the values of the list items in turn
- *code2* will be executed when the values from the **list** are finished

Example: the sum of all list items – ver. #3

```
list = [0, 3, 12, 8, 2]
sum = 0
for el in list:
      sum += el
print(sum)
```

a list and print()

```
list = [0, 3, 12, 8, 2]
print(list)
```

```
[0, 3, 12, 8, 2]
```

not in

```
value not in list
```

answers the question:

is the value not in the list? (`True` – `False`)

Examples

```
list = [0, 3, 12, 8, 2]
print(5 in list)
print(5 not in list)
print(12 in list)
```

```
False
True
True
```

A list fragment

```
list [ from where : where to ]
```

- creates **a new list** from a fragment of **an existing list**

- note - we have the same trap as in the function `range()`
  - `where from` – the index of the first list item that will **be** in the slice
  - `where to` – the index of the first list item that will **not be** in the slice

- and ... we can use negative indices

Example

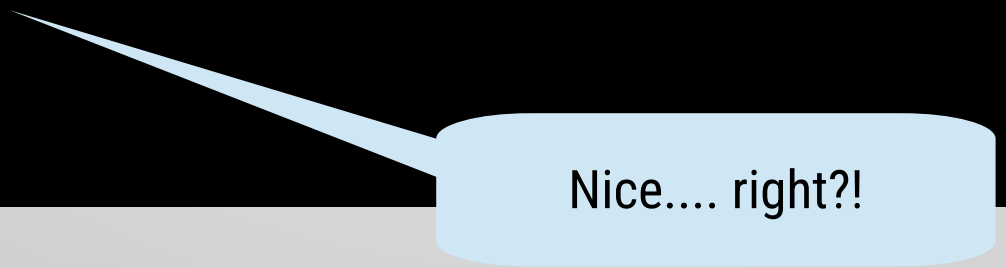
```
list = [10, 8, 6, 4, 2]
new_list = list[1:3]
print(new_list)
```

```
[8, 6]
```

Example

```
list = [10, 8, 6, 4, 2]
new_list = list[1:-1]
print(new_list)
```

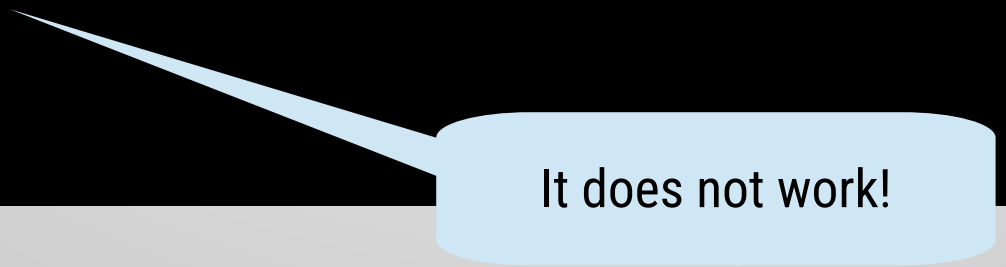
```
[8, 6, 4]
```



Example

```
list = [10, 8, 6, 4, 2]
new_list = list[-1:1]
print(new_list)
```

```
[]
```



Example

```
list = [10, 8, 6, 4, 2]
new_list = list[:3]
print(new_list)
new_list = list[3:]
print(new_list)
```

```
[10, 8, 6]
[4, 2]
```

and now…
everything is clear!
☺

Remember

```
list[ : where to ]
```

is equivalent to:

```
list[ 0 : where to ]
```

len()

this function determines the length of the list (the number of its elements)

```
len(x)
```

the argument: a list

the result: the length of the list

e.g.:

```
print(len([ ]))  →  0
```

This is important, too

```
list[ from where : ]
```

is equivalent to:

```
list[ from where : len(list) ]
```

```
list[ : ]
```

is equivalent to:

```
list[ 0 : len(list) ]
```

So…

```
list = [10, 8, 6, 4, 2]
new_list = list[:]
print(new_list)
```

```
[10, 8, 6, 4, 2]
```

but… remember

```
new_list = list[ : ]
```

is something completely different than :

```
new_list = list
```

Example – look carefully

```
list = [1,3,5]
list1 = list[:]
list2 = list
list[1] = 0
print(list)
print(list1)
print(list2)
```

```
[1, 0, 5]
[1, 3, 5]
[1, 0, 5]
```

`del` instruction (it is not a function!)

removes an item from the list (if it exists)

```
del list[x]
```

the argument: the element `x` from the `list`

the effect: the lack of an item

e.g.:

```
del list[0]
```

Some examples of the instruction `del`

```
list = [10, 8, 6, 4, 2]
del list[1]
print(list)
```

```
???
```

Some examples of the instruction `del`

```
list = [10, 8, 6, 4, 2]
del list[1]
print(list)
```

```
[10, 6, 4, 2]
```

Some examples of the instruction `del`

```
list = [10, 8, 6, 4, 2]
del list[1:3]
print(list)
```

```
???
```

Some examples of the instruction `del`

```
list = [10, 8, 6, 4, 2]
del list[1:3]
print(list)
```

```
[10, 4, 2]
```

Some examples of the instruction `del`

```
list = [10, 8, 6, 4, 2]
del list[:]
print(list)
```

```
???
```

Some examples of the instruction `del`

```
list = [10, 8, 6, 4, 2]
del list[:]
print(list)
```

```
[ ]
```

Some examples of the instruction `del`

```
list = [10, 8, 6, 4, 2]
del list
print(list)
```

```
???
```

Some examples of `del` instruction

```
list = [10, 8, 6, 4, 2]
del list
print(list)
```

```
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    print(list)
NameError: name 'list' is not defined
```

Remember

```
del list[:]
```

it removes **all elements of the list** (it removes all what is in the container)

```
del list
```

it removes **the list** (it removes the container)

Function vs Method

- **Method** – a specific kind of **function**, called in a specific way

- this is how we induce a function: `function(argument)`
- this is how we induce a method: `data.method(argument)`

- **function** → takes an argument and uses it to calculate the result, which is returned
- **method** → takes an argument and uses it to perform actions to modify the data

Function vs Method

- to induce **a function**, we must be sure that **the function exists**
- to induce **a method**, we must be sure that **some data has such a method**

Back to the lists

each list has an `append()` method

```
list.append(element)
```

which adds the `element` to the end of the `list`

Example

```
list = [10, 8, 6, 4, 2]
list.append(0)
print(list)
```

```
[10, 8, 6, 4, 2, 0]
```

Example

```
list = [ ]
for x in range(5):
      list.append(x*x)
print(list)
```

```
???
```

Example

```
list = [ ]
for x in range(5):
      list.append(x*x)
print(list)
```

```
[0, 1, 4, 9, 16]
```

Remember

- each list has an `insert()` method

```
list.insert(where, element)
```

- adds the `element` to the position `where` of the `list`

Example

```
list = [10, 8, 6, 4, 2]
list.insert(0, 0)
print(list)
```

```
[0, 10, 8, 6, 4, 2]
```

and now let's insert sth in the middle of the list

```
list = [10, 8, 6, 4, 2]
list.insert(3, 111)
print(list)
```

```
[10, 8, 6, 111, 4, 2]
```

and it is possible, too

```
list = [10, 8, 6, 4, 2]
list.insert(len(list), 111)
print(list)
```

```
[10, 8, 6, 4, 2, 111]
```

and THIS is possible, too…

```
list = [10, 8, 6, 4, 2]
list.insert(1000, 111)
print(list)
```

```
[10, 8, 6, 4, 2, 111]
```

and THIS…

```
list = [10, 8, 6, 4, 2]
list.insert(-1000, 111)
print(list)
```

```
[111, 10, 8, 6, 4, 2]
```

Example
how to find the largest item in a list?

```
list = [17,3,11,5,1,9,7,15,13]
#
# ?
#

print(max)
```

Example
how to find the largest item in a list?

```
list = [17,3,11,5,1,9,7,15,13]
max = list[0]
for i in range(1,len(list)):
      if list[i] > max:
            max = list[i]

print(max)
```

Example
how to check if a certain value is on the list and in what position?

```
list = [1,2,3,4,5,6,7,8,9,10]
wanted = 5
#
# ?
#
```

Example
how to check if a certain value is on the list and in what position?

```
list = [1,2,3,4,5,6,7,8,9,10]
wanted = 5
itIs = False
for i in range(len(list)):
      if list[i] == wanted:
            itIs = True
            break
if itIs:
      print("It is on position: ", i)
else:
      print("It is not in the list.")
```

`randint()` function from `random` module

"Randomizes" an integer from the given range

```
randint(min,max)
```

the argument: `min` – the lowest number expected

`max` – the largest number expected

the result: a pseudo-random number in the given range

the effect: none

e.g.:

```
print(random.randint(0,10)) → ???
```

Example
how to draw numbers for a lottery?

```
numbers = []
how_much = 6
max = 49
#
# ?
#
print(numbers)
```

Example
how to draw numbers for a lottery?

```
from random import randint

numbers = []
how_much = 6
max = 49
while len(numbers) < how_much:
    number = randint(1,max)
    if number in numbers:
        continue
    numbers.append(number)
print(sorted(numbers))
```

Example
how to check lottery results?

```
lottery_numbers = [5, 11, 9, 42, 3, 49]
my_numbers = [3, 7, 11, 42, 34, 49]
#
# ?
#
print(how_many)
```

Example
how to check lottery results?

```
lottery_numbers = [5, 11, 9, 42, 3, 49]
my_numbers = [3, 7, 11, 42, 34, 49]
how_many = 0
for number in my_numbers:
    if number in lottery_numbers:
        how_many += 1

print("You hit: ",how_many, "numbers.")
```

Back to the lists

each list has a `remove()` method

```
list.remove(element)
```

which removes the `element` from the `list`

Example

```
list = [10, 8, 6, 4, 2]
list.remove(8)
print(list)
```

```
[10, 6, 4, 2]
```

lists

each list has a `reverse()` method

```
list.reverse()
```

which reverses the order of values in the `list`

Example

```
list = [10, 8, 6, 4, 2]
list.reverse()
print(list)
```

```
[2, 4, 6, 8, 10]
```

lists

each list has a `pop()` method

```
list.pop(position)
```

which returns the value from the indicated position, while removing this value (element) from the `list`

Example

```
list = [2, 4, 6, 8, 10]
list.pop(0)
print(list)
```

```
2
[4, 6, 8, 10]
```

lists

each list has an `extend()` method

```
list.extend(newList)
```

which adds the values from the `newList` to the `list`

Example

```
list = [2, 4, 6, 8, 10]
newList = [12, 14]
list.extend(newList)
print(list)
```

```
[2, 4, 6, 8, 10, 12, 14]
```

lists

each list has an `index()` method

```
list.index(element)
```

which returns the position item for the `element` in the `list`

Example

```
list = [2, 4, 6, 8, 10]
list.index(4)
print(list)
```

```
1
```

lists

each list has a `sort()` method

```
list.sort()
list.sort(reverse=True)
```

which sorts the `list` in ascending order

or in descending order `(reverse=True)`

Example

```
list = [2, 4, 6, 8, 10]
list.sort(reverse=True)
print(list)
list.sort()
print(list)
```

```
[10, 8, 6, 4, 2]
[2, 4, 6, 8, 10]
```

lists

each list has a `clear()` method

```
list.clear()
```

which clears items from the `list`, leaving it empty

Let’s try to copy the list

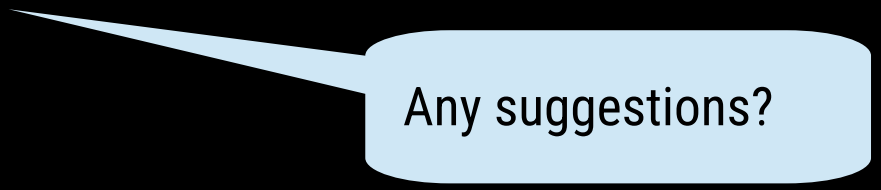
```
list = [2, 4, 6, 8, 10]
listCopy = list
print(list)
print(listCopy)
```

```
[2, 4, 6, 8, 10]
[2, 4, 6, 8, 10]
```

Let’s try to copy the list

```
list = [2, 4, 6, 8, 10]
listCopy = list
list.clear()
print(list)
print(listCopy)
```

```
???
???
```



Let’s try to copy the list

```
list = [2, 4, 6, 8, 10]
listCopy = list
list.clear()
print(list)
print(listCopy)
```

listCopy is just another name for the **list**
and it points to the same place in memory !!!

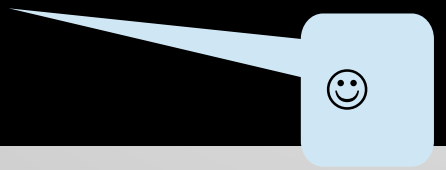
```
[]
[]
```

????

How to copy the list?

```
list = [2, 4, 6, 8, 10]
listCopy = list.copy()
listCopy.clear()
print(list)
print(listCopy)
```

```
[2, 4, 6, 8, 10]
[]
```



lists

each list has a `count()` method

```
list.count(element)
```

which counts how many times the `element` is in the `list`

Example

```
list = [2, 4, 6, 8, 10, 4]
list.count(4)
print(list)
```

```
2
```

A tuple

A tuple

- **a tuple** is a collection of ordered objects, which can be of any type (such as a text string, a list, a dictionary, other tuple, etc.).
- a tuple is similar to a list, but ...
- two important differences:
  - a tuple is **immutable**. Therefore, once you create a tuple, you cannot change its content or even its size.
  - a tuple is written using the normal brackets (), not the square brackets [] as it was in the case of the list.

Example

```
tuple = (1, 2, 3, 4)
print(len(tuple))
print(tuple + (5, 6))       #tuple concatenation
print(tuple * 2)            #tuple multiplication
print(tuple)
```

```
4
(1, 2, 3, 4, 5, 6)
(1, 2, 3, 4, 1, 2, 3, 4)
(1, 2, 3, 4)
```

Remember

- **tuples** are immutable sequences and they have fewer built-in methods than lists, e.g. there are checking or counting methods:
  - tuple.index(2) #returns the index of the value 2
  - tuple.count(4) #counts the occurrences of element 4 in the tuple
  - min(tuple), max(tuple), sum(tuple), ...

- **but there are no modifying methods:**
  - append(), remove(), extend(), pop(), ...

When to use them?

- **tuples** are much faster to compute than **lists**.
- **tuples** are used when we are dealing with elements that should not be susceptible to modification, such as: a list of days of the week or dates in the calendar.
- **tuples** ensure the safety that none of its elements, as well as itself, will not change during the operations performed. The **lists** do not provide this.
- we use **tuples** to store different types of data (*heterogeneous datatypes*), and **lists** for similar types of data (*homogeneous datatypes*).
- **tuples** can be used in the so-called dictionaries as their keys (patience ... it will be soon).

Example

```
t = ("elem1", 35.2, (1, 2, 3))
print(t[0])
print(t[0][2:-1])
print(t[2][1])
```

```
elem1
em
2
```

Conversion

list = list(tuple)

tuple = tuple(list)

```
tuple = ("elem1", 35.2, (1, 2, 3))
print(tuple)
list = list(tuple)
print(list)
tuple = tuple(list)
print(tuple)
```

```
('elem1', 35.2, (1, 2, 3))
['elem1', 35.2, (1, 2, 3)]
('elem1', 35.2, (1, 2, 3))
```

A dictionary

A dictionary

- **a dictionary** is a data structure similar to the list, with the difference that dictionaries do not work with indexes, but with a **key-value** pair

- it can be said that dictionaries are built of **key-value** pairs, which define the relationship between them

A dictionary

```
dictionary = {'key1':'value1', 'key2':'value2'}
```

special character of **dictionary**

: (colon) separates the **key** from the **value**

, (comma) separates each **key-value** instance

A dictionary

```
dictionary = {}
```

creation of an empty dictionary

```
dictionary['key1'] = 'value1'

dictionary['key2'] = 'value2'
```

adding a **key-value** pair to the dictionary

The effect:

```
{'key1': 'value1', 'key2': 'value2'}
```

Remember

- **dictionaries** have the advantage that:
  - **values** can contain any type of data (e.g. strings, numbers, lists, etc.)
  - **keys** must be sets of the same type of elements (e.g. strings, numbers, etc.)

- It is impossible to specify a list and numbers as a set of keys
  - Python will return an error then!!!

Example

```
dict1 = {'key1': 'string', 'key2': 123, 'key3': ['i1', 'i2', 'i3']}
```

```
dict1['key3']
```

```
dict1['key3'][0]
```

a dictionary which contains various data types

calling a dictionary element.
the effect: ['i1', 'i2', 'i3']

here we have a value:
'i1'

Example

```
dict1 = {'key1': 'string', 'key2': 123, 'key3': ['i1', 'i2', 'i3']}
dict1['key4'] = [11, 22, 33]
```

adding a new item to the dictionary

The effect:

```
{'key1': 'string', 'key2': 123, 'key3': ['i1', 'i2', 'i3'], 'key4': [11, 22, 33]}
```

```
dict1['key4'] = 'new value'
```

modification of dictionary's values

The effect:

```
{'key1': 'string', 'key2': 123, 'key3': ['i1', 'i2', 'i3'], 'key4': 'new value'}
```

Remember

- when you use a dictionary **key** you reference unambiguously because **none of two key names can be the same** in a given dictionary

- **key names** in the dictionary are case-sensitive

- assigning a value to an existing key automatically overwrites the old value

A dictionary

```
dict1 = {'key1': 'string', 'key2': 123, 'key3': ['i1', 'i2', 'i3'],
'key4': 'new value'}

dict1['key2'] = dict1['key2'] - 100
dict1['key1'] + ' new'
dict1['key3'] * 2

del dict1['key4']
#dict1.pop('key4')
```

modification of values by subtraction

adding a string to an existing value in the dictionary

duplicating a dictionary item

delete an element with the **del** command or using **pop** method

The effect:

```
{'key1': 'string new', 'key2': 23, 'key3': ['i1', 'i2', 'i3', 'i1',
'i2', 'i3']}
```

A dictionary

```
dict1 = {'k1': 'w1', 'k2': 'w2'}

len(dict1)
dict1.items()
dict1.keys()
dict1.values()
dict1.clear()
```

checking the number of **keys** in the dictionary

display all dictionary items: **key-value** pairs

display all dictionary **keys**

display all dictionary **values**

clearing the entire dictionary. **The effect:** {}

The effect:

```
2
dict_items([('k1', 'w1'), ('k2', 'w2')])
dict_keys(['k1', 'k2'])
dict_values(['w1', 'w2'])
```

A dictionary

```
dict1 = {'k1': 'w1', 'k2': 'w2'}

dict2 = {'k10': 10}          #we create the next dictionary
dict1.update(dict2)
```

dictionary update based on another dictionary

The effect:

```
{'k1': 'w1', 'k2': 'w2', 'k10': 10}
```

Thank you for your attention

see you at the next lecture