ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Real-Time Systems and programming for Automation M

3. Iterables in Python

Notice

The course material includes material taken from Prof. Chesani and Prof. Martini courses (with their consent) which have been edited to suit the needs of this course.

The slides are authorized for personal use only.

Any other use, redistribution, and any for profit sale of the slides (in any form) requires the consent of the copyright owners.

Strings - Recap

- Strings are sequences that are
  - Non-modifiable
  - Finite
  - Ordered
  - Contain alpha-numeric characters from a fixed alphabet

- Operations:
  - concatenation (`+`)
  - repetition (`*`)
  - length (`len(…)`)
  - subscription and slicing (`[…]`)

in

- The **`in`** operator can be used as a membership **test** in a Boolean condition between one collection of elements and another collection or a single elements

```
if ('H' in "HELLO"): print('Eureka’)

if ('HELL' in "HELLO"): print('Eureka')
```

Subscription and Slicing

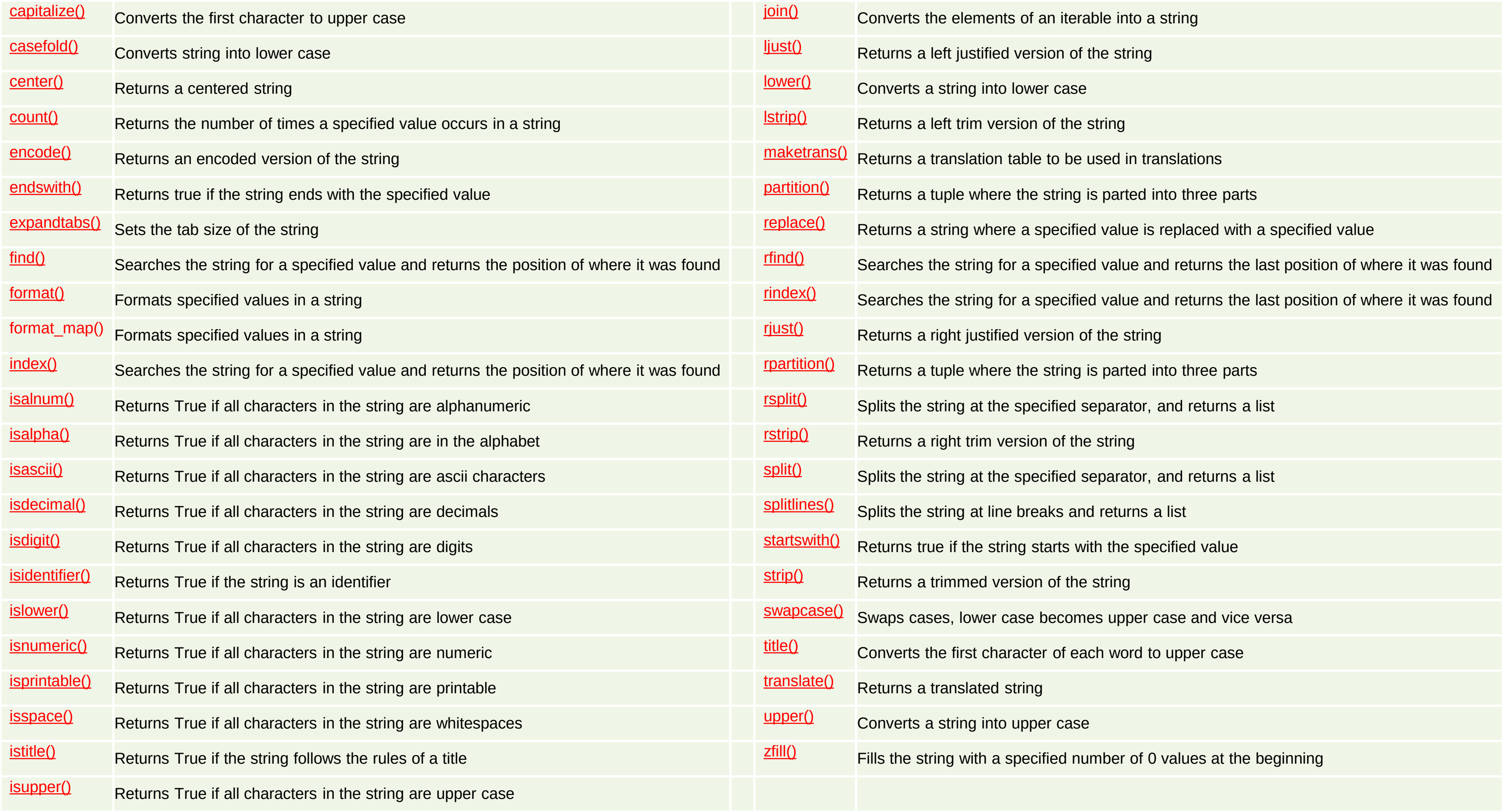
- **Subscription/Slicing**: given an iterable, select a subset substring based on indexes
  `[index]`   `[s_idx:e_idx]`   `[s_idx:e_idx:step]`

  - If only one index is used, only one item is obtained
  - If the `:` is used, multiple items can be obtained
    - If the start or the end indexes are not used, the subset respectively start at the beginning and ends at the end
    - The item at starting index is included
    - The item at ending index is excluded
  - indexes start at zero (hence, e.g. the third element is at index 2)
  - negative indexes are counted from the end
  - the step can be used to skip elements
    - negative step allows to go backwards

Strings – Methods

https://www.w3schools.com/python/python_ref_string.asp

- String are not modifiable: each method will produce a NEW string object



capitalize()	Converts the first character to upper case	join()	Converts the elements of an iterable into a string
casefold()	Converts string into lower case	ljust()	Returns a left justified version of the string
center()	Returns a centered string	lower()	Converts a string into lower case
count()	Returns the number of times a specified value occurs in a string	lstrip()	Returns a left trim version of the string
encode()	Returns an encoded version of the string	maketrans()	Returns a translation table to be used in translations
endswith()	Returns true if the string ends with the specified value	partition()	Returns a tuple where the string is parted into three parts
expandtabs()	Sets the tab size of the string	replace()	Returns a string where a specified value is replaced with a specified value
find()	Searches the string for a specified value and returns the position of where it was found	rfind()	Searches the string for a specified value and returns the last position of where it was found
format()	Formats specified values in a string	rindex()	Searches the string for a specified value and returns the last position of where it was found
format_map()	Formats specified values in a string	rjust()	Returns a right justified version of the string
index()	Searches the string for a specified value and returns the position of where it was found	rpartition()	Returns a tuple where the string is parted into three parts
isalnum()	Returns True if all characters in the string are alphanumeric	rsplit()	Splits the string at the specified separator, and returns a list
isalpha()	Returns True if all characters in the string are in the alphabet	rstrip()	Returns a right trim version of the string
isascii()	Returns True if all characters in the string are ascii characters	split()	Splits the string at the specified separator, and returns a list
isdecimal()	Returns True if all characters in the string are decimals	splitlines()	Splits the string at line breaks and returns a list
isdigit()	Returns True if all characters in the string are digits	startswith()	Returns true if the string starts with the specified value
isidentifier()	Returns True if the string is an identifier	strip()	Returns a trimmed version of the string
islower()	Returns True if all characters in the string are lower case	swapcase()	Swaps cases, lower case becomes upper case and vice versa
isnumeric()	Returns True if all characters in the string are numeric	title()	Converts the first character of each word to upper case
isprintable()	Returns True if all characters in the string are printable	translate()	Returns a translated string
isspace()	Returns True if all characters in the string are whitespaces	upper()	Converts a string into upper case
istitle()	Returns True if the string follows the rules of a title	zfill()	Fills the string with a specified number of 0 values at the beginning
isupper()	Returns True if all characters in the string are upper case		

Tuples

- **Tuples** are structured type composed by group of values
- They are collections of elements that are
  - Non-modifiable
  - Ordered
  - Can contain duplicate values
  - Can contain heterogenous elements (different data types)
  - Elements are separated by **commas**

different types of data can be stored in a tuple

- Similar to strings, but are not limited to characters

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Tuples - Creation

- Tuples are created and assigned to a variable:
  - by specifying a sequence of comma-separated values

    ```
    t=1,2,'hello’,3
    ```

  - Possibly using round brackets

    ```
    t=(1,2,'hello',3)
    ```

- There exists also the empty tuple: `t = ( )`
- Tuples with one member must be defined as: `t = 7,`
  - The use of the comma is mandatory

Tuples – Overloading + and *

- Since they are non modifiable, every operation creates a new tuple
- The `+` operator is overloaded with the meaning of concatenation

```
a = (1,2,3)
b = (4,5,6)
c = a + b # c=(1,2,3,4,5,6)
```

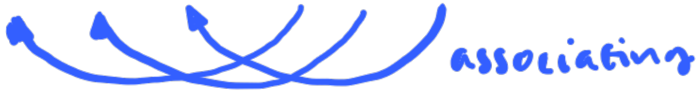
- The `*` operator is overloaded with the meaning of replication

```
a = (1,2,3)
b = a * 3 # b = =(1,2,3,1,2,3,1,2,3)
```

Tuples – Overloading =

- The = operator is overloaded with a more complex semantic
  - The left value can be a sequence of variable
  - The right value can be a collection of the same length
  - Elements on the right will be assigned to a variable on the left

```
a, b, c = (1,2,3) # a=1, b=2, c=3
```



- This can be used to swap two variables in a single instruction

```
a, b = b, a
```

a= 2, b=3

a, b = b, a

a=3 b=2

Tuples – other operators

- Other operators can be used with tuples are
  - Subscription/slicing `[…]`
  - Length `len()`
  - `in`

Nested Tuples

- Tuples can be nested:
  - a tuple can contain any element, among them another tuple:

```
t = (2,3,4)
t1 = (1, (2,3,4), 5)
t2 = (1, t, 5) # (1, (2,3,4), 5)
```

Keep in mind that we don't create t_2 = (1,2,3,4,5)

- Double subscription syntax can be used:

```
print(t1[1][1]) # prints 3
```

Sequences' Methods

- Sequences such as strings and tuples are Objects
  - Being objects, they have predefined **methods**

- **count**(element): returns the number of occurences of an element in a tuple

```
      occ = (2,3,4,2,3,2,2,2).count(2) # occ = 5
```

to search an element inside the tuple

- **index**(el, s, e): return the index of the first occurrence of element el; if not present, returns an error; s and e are the starting and ending indexes for the search: they can be omitted

```
      t1 = (2,3,4,5,6).index(6)

      t2 = (2,3,4,5,6).index(6,0,2)
```

Lists

- Lists are collections of elements that are:
  - Modifiable
  - Ordered
  - Can contain duplicated values
  - Can contain heterogenous elements

- Similar to tuples, but they can be modified

Lists - Creation

- Lists are created and assigned to a variable:
  - by specifying a sequence of comma-separated values
  - enclosed by mandatory square brackets

    ```
    l=[1,2,'hello',3]
    ```

- There exists also the empty list: `l = [ ]`

Lists - Operations

- All the operation that can be done on tuples can be done on lists
  - Subscription/Slicing `[…]`
  - Length `len`
  - `in`
  - Overload of `+`, `*`, and **`=`**
  - It is possible to create nested lists

- It is possible to change elements in the list

```
l1 = [1,2,3]
l1[2] = 4    # l1 = [1,2,4]
```

- It is possible to delete and element of the list with the **`del`** command

```
del l1[1] # l1 = [1,4]
```

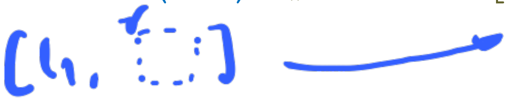
Lists – Methods to add elements

- **`append(el)`** : add a single new element at the end of the list, returns `None`

```
l1 = [1,4]
l1.append(5)  # l1 = [1,4,5]
```

- **`extend(iterable)`** : add all the elements inside iterable at the end of the list, returns `None`

```
l2 = [6,7,8]
l1.extend(l2) # l1 = [1,4,5,6,7,8]
```



- **`insert(index, el)`**: insert the element el at the specified index; the list grows in length of one unit, returns `None`

```
l1.insert(2,99) # l1= [1,4,99,5,6,7,8]
```

Lists – Methods to remove elements

- **`clear()`** remove all the elements of a list, returns `None`
- **`pop(index)`** removes and returns the element at `index`; if `index` is omitted, it removes the last element → return the removed element to value
- **`remove(el)`** removes the first occurrence of element `el`

L = [1,2,3,7,2,3]

L.remove[3]

Lists – Other Methods

- `index()` and `count()`

- **`copy(list)`** returns a copy of list
- **`reverse()`** invert the order of the list
- **`sort(key=…, reverse=…)`** order the list, using the function specified through the key. Key and reverse can be both omitted.

```
l1=[6,5,4,3,2,1]
l1.sort()
l1.sort(reverse=True)

l2 = ['Federico', 'Marco', 'Gianni’]
l2.sort()  → [ F, G, M ]
l2.sort(key=len)  → [ M, G, F ]
l2.sort(key=len, reverse=True)  → [ F, G, M ]
```

Sort by string length

Lists – Use with for

- As for the other iterables, lists can be used with the **for** construct

```
l1 = [1,2,3,4,5]
for i in l1: print(i)
```

- Lists are modifiable! What happens if the instructions inside the **for** modify the list?

```
for i in l1:
   if (i==3): l1.pop(i)
```

DON'T EVER MODIFY THE LIST USED FOR ITERATIONS!!!

- The behaviour will become difficult to predict
  - May become an infinite loop raise errors, or simply create mistakes
  - IF YOU ITERATE WITH FOR ON A LIST, DON'T MODIFY IT
- Solution: create a copy of the list or use a **while** loop

```
while i < len(my_list):
    if (i==3): l1.pop(i)
    i += 1 # increment
```

```
l2=l1.copy()
for i in l2:
   if (i==3): l1.pop(i)
```

← copy the list

Aliases

- If multiple variables reference the same object, they are called **aliases**
  - Many names for the same object

```
l1 = [1,2,3,4,5]
l2 = l1
l3 = l2
l4 = l1  # all 4 lists reference the same object
```

- If the object can be modified (like lists!) any change done to the object through the any of those variables will have repercussion on all the other variables

```
l1.pop(0) # l1=[2,3,4,5]
print(l2) # l2=[2,3,4,5]
print(l3) # l3=[2,3,4,5]
print(l4) # l4=[2,3,4,5]
```

- BE CAREFUL!

Sets

- Sets are collections of elements that are:
  - Modifiable
  - Unordered
  - Can not contain duplicated values
  - Can contain heterogenous elements
  - Can contain only unmodifiable elements
    - Can contain numbers, strings, tuples, bool
    - Can not contain lists or sets

because sets are modifiable

Sets - Creation

- Lists are created and assigned to a variable:
  - by specifying a sequence of comma-separated values
  - enclosed by mandatory curly brackets

```
s={1,2,'hello’,3}
```

- Alternatively, it is possible to use the function `set(iterable)`

- There exists also the empty set: `s = set()`
  - NOT `s={}` DON'T!

- It is possible to create an immutable set using

```
s = frozenset(iterable)
```

Sets can't contain lists because we don't want duplicates in it.

Sets – Intersection, Union, Subsets

- Intersection and union are supported by overloading
  - `&` for intersection
  - `|` for union

```
s1={1,2,3,4}; s2={3,4,5,6}
s3 = s1 & s2 # s3={3,4}
s4 = s1 | s2 # s4={1,2,3,4,5,6}
```

- Comparison operators (>=, <=, >, <) are overloaded
  - New meaning: subset of
    - `s1 < s2` => “is s1 subset of s2?”
    - `s1 >= s2` => “is s2 a subset or the same set of s1?”

```
s1 < s4 # True
s1 < s3 # False
```

NOT SYMETRICAL

Sets – Differences

- Asymmetrical difference: overload of `-`
  - `s1 - s2` => all elements of s1 that are not in s2
  - Changing the order of operands changes the result

```
s1={1,2,3,4}; s2={3,4,5,6}
s1 - s2 # {1,2}
s2 - s1 # {5,6}
```

NOT SYMETRICAL

- Symmetrical difference: overload of `^`
  - `s1 ^ s2` => all elements that are NOT in both sets
  - Changing the order of operands DOES NOT changes the result

```
s1 ^ s2 # {1,2,5,6}
s2 ^ s1 # {1,2,5,6}
```

SYMETRICAL

Sets – Operations and Methods

- `len` and `in` operators are supported

- It is possible to remove elements from the set with methods
  - **`.remove(`**`item`**`)`** if the element is not present raises and error
  - **`.discard(`**`item`**`)`** if the element is not present nothing happens

- It is possible to add elements from the set with methods
  - **`.add(el)`** : add a single new element to the set, returns `None`
  - **`.update(iterable)`** : add all the elements inside iterable to the set, returns `None`

```
s1 = {1,2,3,4}
l1 = [3,4,3,4,5,6]
s1.update(l1)  # s1 = [1,2,3,4,5,6]
```

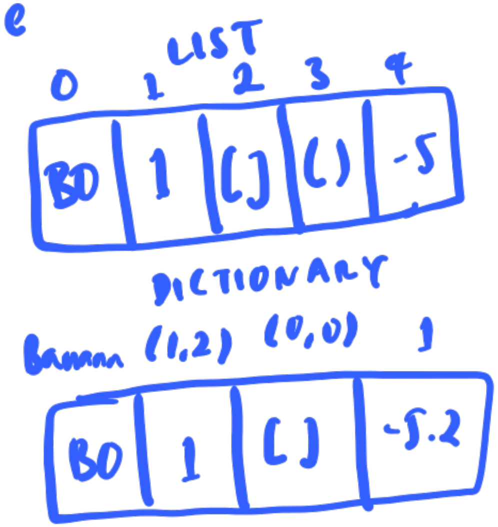
unique elements from s1

added

Dictionaries

- Dictionaries are collections of elements that are:
  - Modifiable
  - Ordered
  - Contain pairs `key:value`
  - Can not contain duplicate `keys`
  - Can contain duplicate `values`
  - Can contain heterogenous elements
  - Keys must me immutable objects
    - Numbers, strings, tuples

keys must be unique



- Similar to lists, but instead of indexes, each element is accessible through a `key`

Dictionaries - Creation

- Lists are created and assigned to a variable:
  - by specifying a sequence of pairs `key:value`
  - enclosed by mandatory curly brackets
    - Several notations are possible for the pairs

```
d = { 1:'Jan', 2:'Feb', 3:'Mar', 4:'Apr'}
d = { 1='Jan’, 2='Feb’, 3='Mar’, 4='Apr'}
d = { [1,'Jan’], [2,'Feb’], [3,'Mar’], [4,'Apr’]}
```

- Alternatively, it is possible to use the function

```
dict(1:'Jan', 2:'Feb')
```

- There exists also the empty dictionary: `d = {}`

Dictionaries – Operations and Methods

- `in` tests if an element is among the `keys`
- `len` returns the number of pairs

tests are only to check whether the object is among the key

- ~~It is possible to remove elements from the set~~
  - ~~`remove(item)` if the element is not present raises and error~~
  - ~~`discard(item)` if the element is not present nothing happens~~

- Main methods
  - `.keys()` : returns the keys as iterable
  - `.values()` : returns the values as iterable
  - `.items()` : returns the items

to check if the value is in the dictionary

Dictionaries – Access and Modification

- To access to a `value` knowing the `key`
  - **`d[key]`** : index-like notation, raise error if `key` is not present
  - **`d.get(key)`** : similar, returns None if `key` is not present

- **`d[key] = value`**: add a `value` to a `key`, index-like notation
  - If `key` was not present is created

- **`del d[key]`** : removes the pair associated to `key`
- **`pop(key)`**: returns the corresponding value and removes the pair

- **`d.setdefault(key, value)`**: returns the value associated to key, if present. If not present, add the couple key-value to the dictionary.
- **`.update(pairs)`**: update the dictionary with the value in pairs

Exercise

- Which of the following raise an error?
  1) D1= {1:10, 2:20} ✓
  2) D2={1:10, 'a':3.14, 3:3} ✓
  3) D3={1:[1,2,3], 3:{1:10,2:20,3:30}, 2:(1,2,3)} ✓

  4) D3[5]='pippo' → it can be operated since it creates a new key
  5) D3[1][0]=100
  6) D3[2][1]=20

  7) D4={(1,2):[1,2]}
  8) D5={[1,2]:(1,2)}
  9) D6={(1,[2]):23}

5.)

D3

key	value
1	[1,2,3]

3	1 10
	2 20
	3 30

2	(1,2,3)

D3[1][0]

↳ calls key 1 and change the list element in index 0

6.)

D3[2][1] = 20, error because it tries to change a tuple

7.)

D4

key	value
(1,2)	[1,2] ← list

it's doable

8.)

D5

key	value
[1,2]	(1,2)

not doable because the key is a list

9.)

D6

key	value
(1,[2])	23

tuple is not modifiable but the objects inside it is modifiable (e.g. lists)

{(,) : v} ← dict, modifiable, but keys are not

(1, [ ]) ← tuple, not modifiable

[2] ← list, modifiable

since the tuple has a list, it can't be used as a key.