# Recap and language details for Python exercises

3 - dictionaries and comprehension

# Dictionaries

# Definition and Initialization

Dictionaries:

- **mutable**.
- unlike lists, where the elements have a numeric key, here **the key can be of any type *immutable***.
- **elements are NOT sorted** (or, from Python 3.7, sorted by input - in any case, we will always use them as if they were NOT sorted).
- All the exercises assume dictionaries are NOT sorted (so they are correct also in previous Python versions) → do not use the "ordering" of dictionary as a tool for solving exercises

To initialize create an empty dictionary:

```
D = {}
```

# Definition and Assignment

It is a set of pairs key:value (called *item*)

```
D = {key:value, key2:value2, ...}
```

**key** can be string type, whole, tuple, etc... any immutable type

**value** can be of any type

For example:

```
D =  { 'vowels' : [ 'a', 'e', 'i', 'o', 'u' ], 'consonants' : [ 'b', 'c' ], 'punctuation' : [':', ';'] }
```

**keys** in this case they are: `'vowels'`, `'consonants'` , `'punctuation'`

**values** referenced by the respective keys are `['a','e','i','o','u']`, `['b','c']`, `[',',';']`

# Dictionary operations

**Access** `D['punctuation']` →`[',',';']` If the key is not present, error.

**Edit** `D['vowels'] = ['a','e','i','o','u', 'A','E','I','O' ,'U']` → modifies the key already present with the values to the right of the equals

**Assignment** `D['alphabet']=['a','b', ... ]` → insert in dictionary $D$ a new item: *chiave* associated with values on the right hand side of the =

**Deletion** `del D['consonants']` → deletes from $D$ the key and the associated values

**Dimention** `len(D)` → counts the number of items (pairs key:value) in the dictionary $D$

# Methods and iteration

**keys** `D.keys()` → returns the *dynamic view* of the keys

N.B! It is not (yet) a list, but we can generate it with `list (D.keys())` or even `list (D)`

**values** `D.values()` → returns the dynamic view of the values.

**item** `D.items()` → returns the dynamic view of the items...

**iteration** `for k in d` itera on the keys...

**presence** `k in d` returns `True` if `k in D.keys()`

**access** For convenience, `D.get(k,default)` returns the index element in `k` in `d`! (that is `D[k]`), if it exists (`k in d`), or the value `default`. (if we do not pass the second parameter, `default=None`).

**update** `D1.update(D2)` extends D1 with the elements of D2, updating D1 with the values of D2 if there are equal keys in D1 and D2

# Comprehension

# List Comprehension

The *list comprehension*, inspired by the mathematical notation of set construction, is a useful way to quickly create lists whose content respects simple rules.

For example the analogue of

$$\{x^3 \mid x \in [0..4]\}$$

could be

```python
cubes = [x**3 for x in range(5)].
print(cubes)
```

```
>>>
[0, 1, 8, 27, 64]
>>>
```

# List Comprehension with if

A comprehension list can contain a **if** to filter only those values that meet a certain condition into the resulting list.

For example, the analog of

$$\{i^2 \mid i \in [0..9] \wedge i \text{ is even}\}$$

could be

```python
evensquared = [i**2 for i in range(10) if i % 2 == 0]
print(evensquared)
```

```
>>>
[0, 4, 16, 36, 64]
>>>
```

# Example

Return the list of the vowels in a string.

```python
def vowels(s):
    RES = []
    for c in s:
        if c in "aeiouAEIOU":
            RES.append(c)
    return RES
```

```python
def vowels_c(s):
    return [c for c in s if c in "aeiouAEIOU"]
```

# List Comprehension: general syntax i

A comprehension list consists of brackets (squares) containing an *expression* followed by a `for` loop, then **zero or more** `fors` and/or `ifs`.

The result will be a new list resulting from the evaluation of the expression in the context of the following `for` and `if`. For example, the following comprehension combines the elements of two lists if they are not the same:

```
>>> comb = [(x, y) for x in [1,2,3] for y in [3,1,4]
    if x != y]
>>> print(comb)
[(1,3), (1,4), (2,3), (2,1), (2,4), (3,1), (3,4)]
```

# List Comprehension: general syntax ii

and is equivalent to:

```python
comb = []
for x in [1,2,3]:
    for y in [3,1,4]:
        if x != y:
            comb.append((x, y))
```

and conceptually similar to

$$\{(x, y) \mid x \in [1, 2, 3] \wedge y \in [3, 1, 4] \wedge x \neq y\}$$

If the expression is a tuple, the parentheses are mandatory!

# List Comprehension: Examples i

```
>>> L = [-4, -2, 0, 2, 4]

# create a new list with values multiplied by 2
>>> [x*2 for x in L].
[-8, -4, 0, 4, 8]

# create a list with only non-negative values
>>> [x for x in L if x >= 0]
[0, 2, 4]

# apply a function to each element of a sequence
>>> [abs(x) for x in L].
[4, 2, 0, 2, 4]
```

# List Comprehension: Examples ii

```
# call a method on each element of a sequence
>>> fruit = ['BANANA', 'BLUEBERRY', 'PASSIONFRUIT']
>>> print([fruit.lower() for fruit in fruit])
['banana', 'blueberry', 'passionfruit']

# create a list of tuples (element, square) of the
    first 6 natural numbers
>>> [(x, x**2) for x in range(6)]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]

# remember: the tuples go in parentheses
>>> [x, x**2 for x in range(6)].
SyntaxError: invalid syntax
```

# List Comprehension: Examples iii

```
#print a list of all letters
>> from string import ascii_letters
>>> print([c for c in ascii_letters])
['a', 'b', 'c', ..., 'z', 'A', 'B', ..., 'Y', 'Z']

#Print the coordinates of the battleship board game...
>> from string import ascii_uppercase
>>> print([(ascii_uppercase[i], i+1) for i in range
    (10)])
('A', 1), ('B', 2), ('C', 3), ('D', 4), ('E', 5), ('F'
    , 6), ('G', 7), ('H', 8), ('I', 9), ('J', 10))]

#... alternative with zip
>>> print([(x,y) for x,y in zip(ascii_uppercase,range
    (1,11)])
```

# Nested List Comprehension

The sequence to which the rules of a comprehension apply may itself be a list comprehension

```
l = ['*'+s+'*' for s in [c.upper() for c in 'abcd']]
print(l)
```

```
['*A*', '*B*', '*C*', '*D*']
```

The initial expression of a list comprehension can be a list comprehension

```
double = [[c*2 for c in word] for word in ['Da', 'casa
    ']]
print(double)
```

```
[['DD', 'aa'], ['cc', 'aa', 'ss', 'aa']]
```

# Dictionary Comprehension

If we use curly brackets, and the expression is a couple `key:value`, we build a dictionary.

```
>>> {x: x**2 for x in (2, 4, 6)}
{2: 4, 4: 16, 6: 36}
```

```
>>> l = ['this', 'class', 'ago', 'confusion']
>>> {word: len(word) for word in l}
{'this': 6, 'class': 6, 'ago': 2, 'confusion': 10}.
```

```
>>> {c: c.upper() for c in 'letters'}
{'l': 'L', 'e': 'E', 't': 'T', 'r': 'R'}
```

# Generators (hints)

Putting instead the expressions in the notation seen so far in round brackets does **not** create tuples, but **generators**.

```
>>> p = (i for i in range(100) if i%2 == 0)
>> type(p)
<class 'generator'>
```

This notation can also be used within functions that expect sequences or *iterators*, for example if I want to calculate

$$\sum_{i=0}^{20} (i \text{ s.t. } i \text{ is not divided by } 6)$$

I can write

```
>>> sum(i for i in range(21) if i % 6 != 0)
154
```