Common functions

`print``(x, x, x, ..., sep='␣', end='\n')`: `sep` is the separator character between the values to be displayed (default is space), `end` is the terminating character (default is newline)
`input``(s)`: returns a string containing information entered from the keyboard (without `'\n'`). `'s'` is the displayed message to the terminal.
`range``(i, j, k)`: generates a sequence of integers starting from `i` (default `i` is `0`), up to `j` (`j` is excluded from the sequence), with step `k` (default `1`).

from pprint import pprint ↘

`pprint(...)` almost like **`print`**, but prettier with nested/complex data structures

For containers `cont`:

`len``(cont)`: returns the number of elements.
`x` **`in`** `cont`: returns `True` if the element `x` is included in `cont`, `False` otherwise.
`sum``(cont)`: returns the sum of all values in `cont`.
`max``(cont)` / **`min`**`(cont)`: returns the maximum/minimum value in `cont`.
`cont.clear()`: deletes all elements (if container is mutable).
`sorted``(cont)`: returns a sorted list containing the elements of `cont` (see note on section on sorting complex data).

For sequences `seq`:

`seq.count(x)`: returns how many times `x` is present in `seq`.
`seq[i]`: returns the element with the index `i` (`i<`**`len`**`(seq)`, otherwise `IndexError`). If `i<0`, it starts counting from the end of the `seq`.
`seq[i:j]`: returns a sub-sequence with consecutive elements from `seq`, starting from the element with index `i` (default=0) and ending with the element with index `j-1`. (default=**`len`**`(seq)`).
`seq[i:j:k]`: uses `k` as "step" to select the elements of the new sub-sequence. If `k<0` and `i>j` it starts counting from the end of the `seq`.

Mathematical

`abs``(a)` $= |a|$
`round``(a)`, **`round`**`(a, n)`: round the number `a` to the nearest integer, round the number `a` to the nearest float value with `n` decimal digits.
`floor(a)`/`ceil(a)`: $\lfloor a \rfloor$ / $\lceil a \rceil$.
`trunc(a)`: truncate the fractional part.

import math ↘

`math.sin(a)`, `cos(a)`, `tan(a)`, `exp(a)`, `log(a)`, `sqrt(a)`. May raise `ValueError`
`math.isclose(a, b)`: `True` if a is almost equal to b.

import random ↘

`random.random()`: returns a random float number in the interval `[0,1)`.
`random.uniform(a, b)`: returns a random float number in the interval `[a,b]`.
`random.randint(i, j)`: returns a random float number in the interval `[i,j]`.
`random.choice(seq)`: returns a randomly selected element of `seq`.
`random.shuffle(seq)`: randomly shuffles the elements of `seq`.

Strings

`str(x)`: converts `x` into a string.
`int(s)`: converts `s` into an integer. Exception: `ValueError`.
`float(s)`: converts `s` into a float. Exception: `ValueError`.
`ord(s)`: returns the Unicode point (an integer) of the rune (character) `s` (`len(s) == 1`).
`chr(i)`: returns the rune (character) that corresponds to the Unicode point `i`. Exception: `ValueError`.
`s+s1`: creates a new string by concatenating two existing ones. Note: `s*2` is equivalent to `s + s`, `s*3` to `s + s + s`, etc.
`s.lower()` / `s.upper()`: returns the lowercase/uppercase version of string `s`.
`s.replace(s1, s2)` / `s.replace(s1, s2, n)`: returns a copy of the string where each occurrence of `s1` in `s` have been substituted with `s2`. If `n` is provided, it replaces at most `n` occurrences of `s1`.
`s.strip(s)`: returns a copy of `s` where leading and trailing whitespaces (spaces, tabs, newlines, ...) have been removed. `s.lstrip(s)` / `s.rstrip(s)`: do the same, but only for leading (left) or trailing (right) whitespace characters.
`s1 in s`: returns `True` if `s` contains the sub-string `s1`.
`s.count(s1)`: returns the number of non-overlapping occurrences of `s1` in `s`.
`s.startswith(s1)` / `s.endswith(s1)`: returns `True` if `s` starts/ends with `s1`.
`s.find(s1)` / `s.find(s1, i, j)`: returns the index of the first occurrence of `s1` in `s`, or `-1` if not found. The optional arguments `i` and `j`, restrict the search in `[i:j]`.
`s1 in s`: returns `True` if `s` contains `s1` as sub-string, otherwise `False`.
`s.count(s1)`: returns the number of occurrences of `s1` in `s`.
`s.startswith(s1)` / `s.endswith(s1)`: returns `True` if `s` begins/ends with `s1`, otherwise `False`.
`s.find(s1)` / `s.find(s1, i, j)`: returns the first index of `s` when an occurrence of `s1` begins, or `-1` if not found. If `i` and `j` are present, searches for `s1` in `s[i:j]`.
`s.index(s1)` / `s.index(s1, i, j)`: similar to `find`, but if `s1` not found raises `ValueError`.
`s.isalnum()`: returns `True` if `s` contains only letters or digits (`[a-zA-Z0-9]`) and has at least one element (`len(s)>=1`), otherwise `False`.
`s.isalpha()`: returns `True` if `s` contains only alphabetic characters (`[a-zA-Z]`) and has at least one element, otherwise `False`.
`s.isdigit()`: returns `True` if `s` contains only digits (`[0-9]`) and has at least one element, otherwise `False`.
`s.islower()` / `s.isupper()`: returns `True` if `s` contains only lowercase/uppercase (`[a-z]/[A-Z]`) characters and has at least one element, otherwise `False`.
`s.isspace()`: returns `True` if `s` contains only whitespace characters i.e., spaces, tabs, newline (`['␣','\t','\n']`) and has at least one element, otherwise `False`.

From strings to lists and vice versa:

`s.split(sep, maxsplit=n)`: returns a list of sub-strings obtained by breaking `s` at each occurrence of the string `sep` (separator). If `sep` if omitted, by default it breaks the string on spaces. If `maxsplit` is specified, at most `n` separations will be done, starting from the left (the final list will have at most `n+1` elements).
`s.rsplit(sep, maxsplit=n)`: similar to `split`, but the breaking of string `s` starts from the right.
`s.splitlines()`: similar to `split`, but uses as as separator the newline `'\n'` and then divides `s` into a list where each element is a line of text in `s`.
`s.join(l)`: returns a single string containing all elements of `l` (which must be a list of strings) separated by the separator `s`.

Formatted string literals `f'{x:fmt}'`

x is any variable or expression. fmt are format codes, which may contain:
< ^ >: for selecting left, center or right alignment
,: to group digits with a comma (e.g., 1,234,567)
width: for indicating how many characters in total the value must occupy. Default: the minimum number required. .precision: for indicating the number of decimal digits (if float) or maximum number of characters (if not numeric). format: `s` string; `d` decimal integer; ; `b` binary integer; `x` hexadecimal integer; `o` octal integer; `f` decimal floating point; `e` decimal floating point in scientific notation; `g` automagically choose between `d`, `f`, and `e`.
0: to pad with zeros instead of spaces (e.g., 42:010d) — Use a space before the size to force a space before the field (e.g., 42: 2d)
Example: `f'{n:5d}␣{a:7.2f}␣{s:>10s}'`

Lists

`list()`: creates and returns a new empty list. **`list(cont)`**: returns a new list containing all elements of container `cont`.
`[x, ..., x]`: creates and returns a new list with the supplied elements.
`l.copy()` or **`list(l)`**: returns a new list, which is a shallow copy of the list `l`.
`l + l1`: returns a new list by concatenating the elements of `l` and `l1`. `l * 2` is equivalent to `l + l`, `l * 3` is equivalent to `l + l + l`, etc.
`l1 == l2`: returns `True` whether the two lists contain the same elements in the same order, otherwise `False`.
`l.pop()`: removes the last element from the list and returns it. `l.pop(i)`: removes the element at the position `i` and returns it. The following elements are moved back by one place.
`l.insert(i, x)`: inserts `x` in the position `i` in list `l`. The following elements are moved forward by one place.
`l.append(x)`: appends `x` at the end of the list `l`.
`l.extend(l1)`: extends the list `l` by appending to it all elements of list `l1`.
`l.count(x)`: returns the number of occurrences of element `x` in list `l`
`l.index(x)`: returns the index of the first occurrence of element `x` in the list `l`. If the element is not present in the list, it raises `ValueError`. `l.index(x, i, j)`: returns the index of the first occurrence of the element `x` in the list `l[i:j]` (the element in position `j` is not included in the search). The position is calculated from the beginning of the list. If not found, it raises `ValueError`.
`l.remove(x)`: removes the element with the value `x` from the list and move all elements that follow it back by one place. If the element `x` is not in the list it raises `ValueError`.
`l.reverse()`: changes the list `l` by reversing the order of its elements.
`l.sort(reverse=False)`: Sorts in place the elements of the list. See the notes for **`sorted`** (see note on section on sorting complex data).
`enumerate``(l)`: returns a list of tuples of `[(index1, value1), (index2, value2), ...]`, that allows you to iterate simultaneously on indices and values of the list `l`.

from operator import itemgetter ↘

`l.sort(key=itemgetter('k'))`: sort a list of dicts on the value associated to key `k`.
`l.sort(key=itemgetter(n))`: sort a list of lists or tuples on the sub-element value with index `n` is max/min. May be useful when the list `l` has been returned by **`enumerate()`** or **`dict`**`.items()`.
`max/min``(l, key=itemgetter('k'))` : in a list of dicts, returns the element those value with key `k` è max/min.
`max/min``(l, key=itemgetter(n))`: in a list of lists or tuples, returns the element those sub element with index `n` is max/min. May be useful when the list `l` has been returned by **`enumerate()`** or **`dict`**`.items()`.

Note: `reverse` and `key` can be used together.
Note: It is always possible to define a local function that gets the object to be sorted and returns the associated values, and then use `key=my_local_function`.

Tuple

`tuple()`: creates and returns a new empty list. **`tuple(cont)`**: returns a new tuple containing all elements of container `cont`.
`(x, ..., x)`: creates and returns a new tuple with the supplied elements. Note: use `(x,)` for a 1-element tuple.
Tuples support all lists' functions and methods that do not modify the container. E.g., `u[i]`, `u+u1`, `x in u`, `u.index(x)`, `sorted(u)`, `enumerate(u)`.

Sets

`set()`: returns a new empty set. **`set(cont)`**: returns a new set that contains all elements of `cont` (without duplicates).
`{x, x, ..., x}`: returns a new set containing the indicated elements (without duplicates).
`t.copy()` or **`set(t)`**: returns a shallow copy of the set `t`.

`t.add(x)`: adds the new element `x` to set `t`. If the element already exists, nothing happens.
`t.discard(x)`: removes the element `x` from set `t`. If the element is not in the set, nothing happens.
`t.remove(x)`: similar to `discard`, but if the element is not in the set raises `KeyError`.
`t == t1`: checks if the set `t` is equivalent with set `t1`.
`t.issubset(t1)` or `t<=t1`: checks if `t` $\subseteq$ `t1`.
`t.issuperset(t1)` or `t>=t1`: checks if `t` $\supseteq$ `t1`.
`t.isdisjoint(t1)`: returns `True` if the intersection of `t` and `t1` is zero.
`t.union(t1)` or `t|t1`: returns a new set equal to `t` $\cup$ `t1`.
`t.intersection(t1)` or `t&t1`: returns a new set equal to `t` $\cap$ `t1`.
`t.difference(t1)` or `t-t1`: returns a new set with elements present in `t` but not in `t1`.
`t.symmetric_difference(t1)` or `t^t1`: returns a new set that contains elements that are present in only one of the sets and not in both (operator x-or).

Dicts

`dict()` or `{}`: create and return a new empty dictionary. `{k:x, ..., k:x}`: create and return a new dictionary containing the specified key/value pairs.
`dict(d)` or `d.copy()`: returns a shallow copy of the dictionary `d`.
`k in d`: returns `True` if the key `k` exists in the dictionary `d`, otherwise `False`.
`d[k] = x`: set the new key/value pair in the dictionary `d`.
`d[k]`: returns the value associated with the key `k` if present in `d`, otherwise raises `KeyError`.
`d.get(k, x)`: returns the value associated with the key `k`, if present in `d`, otherwise it returns the default value `x`.
`d.pop(k)`: removes from `d` the key `k` and the value associated with it; if not present raises `KeyError`. Returns the deleted value.
`d.items()`: returns a sequence of tuples `(key, value)` of all elements of `d`, in order of insertion.
`d.values()`: returns a sequence containing the values of `d`.
`d.keys()`: returns a sequence containing the keys of `d`, in order of insertion.
`sorted(d)`: returns a sorted list of the keys of the dictionary `d` (see note on section on sorting complex data).

File

`f = open(s, mode)`: opens the file named `s`. `mode`: `'r'` reading (default), `'w'` writing. Returns a "file object" `f`. If something fail, raises the exception: `OSError`.

`f.close()`: closes the open file `f`.

`with open(s, mode) as f`: this statement wraps the opening of the file named `s` with mode `mode` in a block. It creates a "file object" `f` to be used within the block. When the code exits the `with` compound statement the file is automagically closed.

`f.readline()`: returns a string of characters read from file `f` up to `'\n'` (including `'\n'`). Returns `""` (empty string) if at the end of the file.

`f.read(num)`, `f.read()`: returns a string with at most `num` characters read from the file `f`. If no argumnet is used it returns the entire file as a single string.

`f.readlines()`: returns the file as a list of strings as elements, where each string is a line of the file.

`f.write(s)`: writes `s` to file `f`. Note: it does not automatically write a new line `'\n'`.

`print(..., file=f)`: similar to `print`, but writes to file `f` instead of the terminal.

import csv ↘

`csv.reader(f)`: returns a CSV reader object for the file `f` to iterate over with a `for` loop, which yields in each iteration a list whose elements are the fields of the next line of file `f`.

`csv.DictReader(f)`: returns a CSV dictionary reader object to iterate over with a `for` loop. The keys are the field names in the very first line of the file, unless specified using option `fieldnames=`.

`csv.writer(f)`: returns a CSV writer object for the file `f` opened for writing. Data can be written line by line using either the method `writerow(one_record)` or the method `writerows(all_records)`.

Option: use `delimiter='X'` to use 'X' instead of the default comma ',' as a field separator. Useful for some Italian CSV that uses semicolon instead of comma.

Note: CSV files should be opened using option `newline=''`.

import copy ↘

`copy.copy(x)`: returns a shallow copy of `x`. That is, constructs a new object and then inserts into it the references to the objects found in the original (`x`).

`copy.deepcopy(x)`: returns a deep copy of `x`. That is, constructs a new object, then inserts into it the deep-copies of the objects found in the original container (`x`).

Note: If `foo` and `bar` are lists: `foo=bar` is not a copy; `foo=bar[:]` is a shallow copy.

Main exceptions

`ValueError`: an operation or function receives an argument that has the right type but an inappropriate value (e.g., `math.sqrt(-1)`).

`IndexError`: a sequence subscript is out of range (e.g., `l[len(l)]`).

`KeyError`: a mapping (dictionary) key is not found in the set of existing keys.

`OSError` (o `IOError`): a system function returns a system-related error, including I/O failures such as `FileNotFoundError`, `FileExistsError`, `PermissionError`, or "disk full". Not to be used for illegal argument types or other incidental errors.

Keys (argument types)

`s`, `s1`: string

`a`, `b`, `c`, `...`: number

`i`, `j`, `k`, `n`: integer

`x`: anything
`l`, `l1`: list
`d`: dict
`t`, `t1`: set
`u`, `u1`: tuple
`seq`: sequence (list, tuple, string)
`cont`: container (list, tuple, string, set, dict)

	Container				
Operation	`str`	`list`	`tuple`	`set`	`dict`
Create	`"abc"` `’abc’`	`[a, b, c]`	`(a, b, c)`	`{a, b, c}`	`{a:x, b:y, c:z}`
Create empty	**`str`**`()` `""` `’’`	**`list`**`()` `[]`	**`tuple`**`()` `()`	**`set`**`()`	**`dict`**`()` `{}`

Access `i`-th item	`s[i]`	`l[i]`	`u[i]`		`d[k]`
					`d.get(k,default)`

Modify `i`-th item		`l[i]=x`			`d[k]=x`
Add one item (modify value)		`l.append(x)`		`t.add(x)`	`d[k]=x`
Add one item at position (modify value)		`l.insert(i,x)`			
Add one item (return new value)	`s+’x’`	`l+[x]`	`u+(x,)`		
Join two containers (modify value)		`l.extend(l1)`		`t.update(t1)`	

Join two containers (return new value)	`s+s1`	`l+l1`	`u+u1`	`t.union(t1)`
				`t|t1`	

Does it contain a value?	`x` **`in`** `s`	`x` **`in`** `l`	`x` **`in`** `u`	`x` **`in`** `s`	`k` **`in`** `d`
					(search keys)
					`x` **`in`** `d.values()`
					(search values)

Where is a value? (returns index)	`s.find(x)`
	`s.index(x)`
		`l.index(x)`	`u.index(x)`		

Delete an item, by index		`l.pop(i)`
		`l.pop()`
					`d.pop(k)`

Delete an item, by value		`l.remove(x)`		`t.remove(x)`
				`t.discard(x)`	

Sort (modify value)		`l.sort()`			
Sort (return new list)	**`sorted`**`(s)`	**`sorted`**`(l)`	**`sorted`**`(u)`	**`sorted`**`(t)`	**`sorted`**`(d)` (keys) **`sorted`**`(d.items())`