# Python Cheat Sheet: Built-in types, common functions, methods and packages

Computer Sciences (07JCJ) — aa 2021/22 — Politecnico di Torino

## 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).

### For most 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.

`sorted(cont)`: returns a sorted list containing the elements of `cont` (see note on section on sorting complex data).

### For all 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`.

## Strings

`int(s)`: converts `s` into an integer. Exception: `ValueError`.

`float(s)`: converts `s` into a float. Exception: `ValueError`.

`str(x)`: converts `x` into string.

`ord(s)`: returns the Unicode point (an integer) of the character `s` (`len(s) == 1`).

`chr(i)`: returns the character (rune) that corresponds to the Unicode point `i`. Exception: `ValueError`.

`s+s1`: creates a new string by concatenating two existing ones.

`s*n`: creates a new string by concatenating `n` times the string `s` with itself.

`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 whitespace characters (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 `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 substrings 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 strings `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).

*Example*: `s` string, `d` decimal integer, `f` real number, `g` real number in scientific notation:

```
f"{n:5d}␣{a:7.2f}␣{s:>10s}"
```

## Mathematics

`abs(a)`: $|a|$

`round(a)`, `round(a, n)`: rounds `a` to the nearest integer or to the float with `n` decimal digits if `n` is specified.

`floor(a)`/`ceil(a)`: $\lfloor a \rfloor$ / $\lceil a \rceil$

`trunc(a)`: eliminates the fractional part of `a`.

### import math ↘

`math.sin(a)`, `cos(a)`, `tan(a)`, `exp(a)`, `log(a)`, `sqrt(a)`. They can raise `ValueError`.

`math.isclose(a, b, rel_tol, abs_tol)`: returns `True` if `|a - b|` is less or equal to `rel_tol` (relative tolerance) or `abs_tol` (absolute tolerance).

### import random ↘

`random.random()`: returns a random float number in the interval `[0,1)`.

`random.randint(i, j)`: returns a number integer between `i` and `j` (`j` is included).

`random.choice(seq)`: returns a randomly selected element of `seq`.

`random.shuffle(seq)`: randomly shuffles the elements of `seq`.

## Lists

`[]`: creates and returns a new empty list.

`[x, ..., x]`: returns a new list with the supplied elements.

`list(cont)`: returns a *new* list containing all elements of container `cont`.

`l * n`: returns a new list by replicating the elements of `l` exactly `n` times.

`l + l1`: returns a new list by concatenating the elements of `l` and `l1`.

`l == l1`: returns `True` if the two lists contain exactly 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.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.extend(l1)`: extends the list `l` by appending to it all elements of list `l1`.

`l.reverse()`: changes the list `l` by reversing the order of its elements.

`l.copy()` or `list(l)`: returns a new list, which is a (shallow) copy of the list `l`.

`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`.

## File

`f = open(s, mode)`: opens the file named `s`. `mode`: `'r'` reading, `'w'` writing. Returns a “file object” `f`. Exceptions: `FileNotFoundError` if the file does not exist, in general `OSError`.

`f.close()`: closes the (previously opened) 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)`: 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=''`.

## Sets

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

## Dictionaries

`dict()` or `{}`: a new empty dictionary.
`{k:x, ..., k:x}`: 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).

## import copy ↘

`copy.copy(x)`: returns a shallow copy of `x`. It constructs a new object and then inserts into it references to the objects found in the original (`x`).
`copy.deepcopy(x)`: returns a deep copy of x. It constructs a new object, then inserts a new replica of the objects of the original container (`x`).

## Sorting complex data

Optional parameters that can be used with `sort`, `sorted`, `max`, and `min`. Note: `itemgetter` must be imported as `from operator import itemgetter`.
`reverse=True`: reverse the comparison.
`key=lambda key: data[key]`: sort the *dictionary* `data` based on the value.
`key=lambda elem: elem['k']`: sort a list of dictionaries based on the value of the entry with key `k` of each dictionary (alternative: `key=itemgetter('k')`).
`key=lambda elem: elem[n]`: sort a list of *lists*, *tuples*, or other *sequences*, based on their $(n+1)$-th value (alternative: `key=itemgetter(n)`).

## Common Exceptions

`ValueError`: incorrect argument value (e.g., `math.sqrt(-1)`).
`IndexError`: access to out-of-bound element in a sequence (e.g., `l[len(l)]`).
`KeyError`: access to non-existing key in a collection (e.g., `dict()['foo']`).
`OSError`: general exception for trapping I/O errors, such as `FileNotFoundError`, `PermissionError`, and `FileExistsError`.

## Legend (types of accepted arguments/objects)

`s`, `s1`: string
`a`, `b`, `c`, `...`: integer or float
`i`, `j`, `k`, `n`: integer
`x`: any
`l`, `l1`: list
`d`: dictionary
`t`, `t1`: set
`seq`: sequence (list, tuple, string)
`cont`: container (list, tuple, string, set, dict)

*v0.5 @ 18/01/2022*