Mémento v1.2.2

Python 3 Cheat Sheet

Official Python documentation on http://docs.python.org/py3k

Base Types

integer, float, boolean, string

```
  int 783    0    -192
float 9.23   0.0   -1.7e-6        (e-6 = 10^-6)
 bool True   False
  str "One\nTwo"   'I\'m'
```

- `\n` : new line
- `\'` : ' escaped
- immutable, ordered sequence of chars
- multiline: `"""X\tY\tZ`
`1\t2\t3"""` (`\t` : tab char)

Container Types

- ordered sequence, fast index access, repeatable values

```
  list [1,5,9]   ["x",11,8.9]   ["word"]   []
 tuple (1,5,9)   11,"y",7.4     ("word",)  ()
```

immutable → `str` as an ordered sequence of chars

expression with just comas

- no *a priori* order, unique key, fast key access ; keys = base types or tuples

```
  dict {"key":"value"}                              {}
  dict {1:"one",3:"three",2:"two",3.14:"π"}
```

dictionary

key/value associations

```
  set {"key1","key2"}   {1,9,3,0}   set()
```

Identifiers

for variables, functions, modules, classes… names

`a..zA..Z_` followed by `a..zA..Z_0..9`

- diacritics allowed but should be avoided
- language keywords forbidden
- lower/UPPER case discrimination

☺ `a toto x7 y_max BigOne`
☹ ~~`8y and`~~

Variables assignment

```
x = 1.2+8+sin(0)
y,z,r = 9.2,-7.6,"bad"
x+=3
x-=2
x=None
```

- `x` : variable name (identifier)
- `1.2+8+sin(0)` : value or computed expression
- `y,z,r` : variables names
- `9.2,-7.6,"bad"` : container with several values (here a tuple)
- `x+=3` / `x-=2` : increment / decrement
- `x=None` : « undefined » constant value

Conversions

`type(`*expression*`)`

```
int("15")          can specify integer number base in 2nd parameter
int(15.56)         truncate decimal part (round(15.56) for rounded integer)
float("-11.24e8")
str(78.3)          and for litteral representation → repr("Text")
```

see other side for string formating allowing finer control

`bool` → use comparators (with `==`, `!=`, `<`, `>`, …), logical boolean result

```
list("abc")              → ['a','b','c']   (use each element from sequence)
dict([(3,"three"),(1,"one")]) → {1:'one',3:'three'}
set(["one","two"])       → {'one','two'}   (use each element from sequence)
":".join(['toto','12','pswd']) → 'toto:12:pswd'
"words with  spaces".split()   → ['words','with','spaces']
"1,4,8,2".split(",")           → ['1','4','8','2']
```

- `":"` : joining string
- `['toto','12','pswd']` : sequence of strings
- `","` : splitting string

Sequences indexing

for lists, tuples, strings, …

```
negative index   -6   -5   -4    -3    -2    -1
positive index    0    1    2     3     4     5
          lst=[11,  67,  "abc",  3.14,  42,  1968]
positive slice    0    1    2     3     4     5    6
negative slice   -6   -5   -4    -3    -2    -1
```

`len(lst)` → `6`

individual access to items via `[`*index*`]`

`lst[1]`→`67` `lst[0]`→`11` *first one*
`lst[-2]`→`42` `lst[-1]`→`1968` *last one*

access to sub-sequences via `[`*start slice*`:`*end slice*`:`*step*`]`

```
lst[:-1]→[11,67,"abc",3.14,42]
lst[1:-1]→[67,"abc",3.14,42]
lst[::2]→[11,"abc",42]
lst[:]→[11,67,"abc",3.14,42,1968]
lst[1:3]→[67,"abc"]
lst[-3:-1]→[3.14,42]
lst[:3]→[11,67,"abc"]
lst[4:]→[42,1968]
```

Missing slice indication → from start / up to end.

On mutable sequences, usable to remove `del lst[3:5]` *and to modify with assignment* `lst[1:4]=['hop',9]`

Boolean Logic

Comparators: `<` `>` `<=` `>=` `==` `!=` (≤ ≥ = ≠)

- `a and b` logical and — *both simultaneously*
- `a or b` logical or — *one or other or both*
- `not a` logical not
- `True` true constant value
- `False` false constant value

Statements Blocks

```
parent statement:
    statements block 1…
    ⋮
parent statement:
    statements block 2…
    ⋮
next statement after block 1
```

indentation !

Conditional Statement

statements block executed only if a condition is true

```
if logical expression:
    statements block
```

can go with several elif, elif... and only one final else, example :

```
if x==42:
    # block if logical expression x==42 is true
    print("real truth")
elif x>0:
    # else block if logical expression x>0 is true
    print("be positive")
elif bFinished:
    # else block if boolean variable bFinished is true
    print("how, finished")
else:
    # else block for other cases
    print("when it's not")
```

Maths

floating point numbers… approximated values! *angles in radians*

Operators: `+` `-` `*` `/` `//` `%` `**` (× ÷ integer ÷ ÷ remainder a^b)

```
(1+5.3)*2→12.6
abs(-3.2)→3.2
round(3.57,1)→3.6
from math import sin,pi…
sin(pi/4)→0.707…
cos(2*pi/3)→-0.4999…
acos(0.5)→1.0471…
sqrt(81)→9.0        √
log(e**2)→2.0       etc. (cf doc)
```

Conditional loop statement

statements block executed as long as condition is true

```
while logical expression:
    statements block
```

```
s = 0
i = 1
```
} *initializations **before** the loop*

condition with at least one variable value (here `i`)

```
while i <= 100:
    # statement executed as long as i ≤ 100
    s = s + i**2
    i = i + 1
```
} *make condition variable change*

$$s = \sum_{i=1}^{i=100} i^2$$

```
print("sum:",s)
```
} *computed result after the loop*

be careful of inifinite loops !

Loop control

`break` *immediate exit*

`continue` *next iteration*

Iterative loop statement

statements block executed for each item of a container or iterator

```
for variable in sequence:
    statements block
```

Go over sequence's **values**

```
s = "Some text"
cnt = 0
```
} *initializations **before** the loop*

loop variable, value managed by `for` statement

```
for c in s:
    if c == "e":
        cnt = cnt + 1
print("found",cnt,"'e'")
```

Count number of `e` in the string

loop on dict/set = loop on sequence of keys
use slices to go over a subset of the sequence

Go over sequence's **index**

- modify item at index
- access items around index (before/after)

```
lst = [11,18,9,12,23,4,17]
lost = []
for idx in range(len(lst)):
    val = lst[idx]
    if val > 15:
        lost.append(val)
        lst[idx] = 15
print("modif:",lst,"-lost:",lost)
```

Limit values greater than 15, memorization of lost values.

Go simultaneously over sequence's **index** and **values**:

```
for idx,val in enumerate(lst):
```

Display / Input

```
print("v=",3,"cm :",x,",",y+4)
```

items to display: litteral values, variables, expressions

`print` options:

- `sep=" "` (items separator, default space)
- `end="\n"` (end of print, default new line)
- `file=f` (print to file, default standard output)

```
s = input("Instructions:")
```

`input` always returns a **string**, convert it to required type (cf boxed *Conversions* on on ther side).

Operations on containers

`len(c)` → items count
`min(c)` `max(c)` `sum(c)`
`sorted(c)` → sorted *copy*
`val in c` → boolean, membersihp operator `in` (absence `not in`)
`enumerate(c)` → *iterator* on (index,value)

*Note: For dictionaries and set, these operations use **keys**.*

*Special for **sequence containeurs** (lists, tuples, strings) :*

`reversed(c)` → reverse *iterator* `c*5` → duplicate `c+c2` → concatenate
`c.index(val)` → position `c.count(val)` → events count

Operations on lists

modify original list

`lst.append(item)`	add item at end
`lst.extend(seq)`	add sequence of items at end
`lst.insert(idx,val)`	insert item at index
`lst.remove(val)`	remove first item with value
`lst.pop(idx)`	remove item at index and return its value
`lst.sort()` `lst.reverse()`	sort / reverse list *in place*

Operations on dictionaries

`d[key]=value` `d.clear()`
`d[key]→value` `del d[clé]`
`d.update(d2)` *update/add associations*
`d.keys()` `d.values()` `d.items()` *views on keys, values associations*
`d.pop(clé)`

Operations on sets

Operators:
`|` → union (vertical bar char)
`&` → intersection
`-` `^` → difference/symetric diff
`<` `<=` `>` `>=` → inclusion relations

`s.update(s2)`
`s.add(key)` `s.remove(key)`
`s.discard(key)`

Generator of int sequences

frequently used in `for` iterative loops

```
range([start,]stop[,step])
```

default 0, not included

```
range(5)   → 0 1 2 3 4
range(3,8) → 3 4 5 6 7
range(2,12,3) → 2 5 8 11
```

`range` returns a « generator », converts it to list to see the values, example:
`print(list(range(4)))`

Function definition

function name (identifier) — named parameters

```
def fctname(p_x,p_y,p_z):
    """documentation"""
    # statements block, res computation, etc.
    return res
```

`return res` ← result value of the call. if no computed result to return: `return None`

parameters and all of this bloc only exist *in* the block and *during* the function call (*"black box"*)

Function call

```
r = fctname(3,i+2,2*i)
```

one argument per parameter

retrieve returned result (if necessary)

Files

storing data on disk, and reading it back

```
f = open("fil.txt","w",encoding="utf8")
```

- file **variable** for operations
- **name** of file on disk (+path…) — cf functions in modules `os` and `os.path`
- opening **mode**: `'r'` read, `'w'` write, `'a'` append…
- **encoding** of chars for text files: `utf8` `ascii` `latin1` …

writing

```
f.write("hello")
```

*text file → read /write only **strings**, convert from/to required type.*

reading

```
s = f.read(4)
```

empty string if end of file — if char count not specified, read whole file

```
s = f.readline()
```

read next line

```
f.close()
```

don't forget to close file after use

Pythonic automatic close : `with open(…) as f:`

very common: iterative loop reading lines of a text file

```
for line in f :
    # line processing block
```

Strings formating

```
"model {} {} {}".format(x,y,r) → str
"{selection:formating!conversion}"
```

formating directives — values to format

- **Selection** :
  `2`
  `x`
  `0.nom`
  `4[key]`
  `0[2]`
- **Formating** :
  fillchar *alignment* *sign* *minwidth*.*precision~maxwidth* *type*
  `< > ^ =` `+ - space` `0` at start for filling with 0
  integer: `b` binary, `c` char, `d` decimal (default), `o` octal, `x` or `X` hexa…
  float: `e` or `E` exponential, `f` or `F` fixed point, `g` or `G` appropriate (default), `%` percent
  string : `s` …
- **Conversion** : `s` (readable text) or `r` (litteral representation)

Examples:

```
"{:+2.3f}".format(45.7273)
→'+45.727'
"{1:>10s}".format(8,"toto")
→'      toto'
"{!r}".format("I'm")
→'"I\'m"'
```