Mémento v1.0.5a

Python 3 Basic Reference Card

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
- unmodifiable, 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]   ["mot"]    []
tuple (1,5,9)    11,"y",7.4     ("mot",)   ()
```

tuple : unmodifiable ; 11,"y",7.4 : expression with just comas

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

```
 dict {"key":"value"}                                  {}
      {1:"one",3:"three",2:"two",3.14:"π"}
 set {"key1","key2"}   {1,9,3,0}   set()
```

dict : dictionnary

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
- min/MAJ case discrimination

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

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 verso for string formating allowing finer control

`bool` → use comparators (with `==`, `!=`, `<`, `>`, …), logial 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'
```

joining string ; sequence of strings

```
"words with  spaces".split() → ['words','with','spaces']
"1,4,8,2".split(",") → ['1','4','8','2']
```

splitting string

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 calculation 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

Sequences indexing

for lists, tuples, char 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

Boolean Logic

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

- `a and b` logical and
- `a or b` logical or (*twice simultaneously* / *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 else if, else if... and only one final else, example :

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

Maths

floating point numbers… approximated value! angles in radians

Operators: `+` `-` `*` `/` `//` (integer ÷) `%` (÷ remain) `**` (×, ÷, 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
```

another option to exit loop is if <condition> : break

```
s = 0
i = 1
while i <= 100:
    # statement executed as long as i ≤ 100
    s = s + i**2
    i = i + 1
print("sum:",s)
```

initialisations **before** the loop

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

make condition variable change

computed result after the loop

care to inifinite loops !

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

Iterative loop statement

statements block executed for each item of a sequence of values

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

Go over sequence's **values**

```
s = "Some text"
cpt = 0
for c in s:
    if c == "e":
        cpt = cpt + 1
print("found",cpt,"'e'")
```

initialisations **before** the loop

loop variable, value managed by instruction `for`

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, memorisation of lost values.

Display / Input

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

items to display: littéral 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 return a **string**, convert it to required type (cf boxed *Conversions* on recto).

Operations on sequences

```
len(seq) →items count
min(seq) max(seq) sum(seq)
sorted(seq) →sorted copy  reversed(seq) →reversed copy
enumerate(seq) →sequence (index,value) for for loops
```

lists special

```
lst.append(item) lst.extend(seq)
lst.index(val) lst.count(val) lst.pop(idx)
lst.sort() lst.remove(val) lst.insert(idx,val)
```

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 », convert it to list to see the values, example:

```
print(list(range(4)))
```

Files

storage of data on disk, and read back

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

- file **variable** for operations
- **name** of file on disk (+path…)
- opening **mode**
  - `'r'` read
  - `'w'` write
  - `'a'` append…
- **encoding** of chars for text files: `uft8` `ascii` `latin1` …

cf functions in modules `os` and `os.path`

writing

```
f.write("coucou")
```

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

reading

empty string if end of file

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

if char count not specified, read whole file

read next line

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

```
f.close()
```

dont miss to close file after use

very common: iterative loop reading lines of a text file

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

Function definition

function name (identifier)

named parametrs

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

Strings formating

```
"{:e}".format(123.728212)
→'1.237282e+02'
"{:f}".format(123.728212)
→'123.728212'
"{:g}".format(123.728212)
→'123.728'
```

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

formating directives

values to format

Conversion parameter:

- `s` → display string via `str()`
- `r` → representation string via `repr()`

Formating parameter:

- *filling*: 1 char (followed by alignment!)
- *alignment*: `<` left, `>` right, `^` center, `=` on sign
- *sign*: `+` for >0 and <0, `-` only for <0, *espace* for >0
- `#`: alternative representation
- *minwidth*: number, 0 at start for filling with 0
- *.precision*: decimal count for a float, max width
- *type*:
  - integers: `b` binary, `c` char, `d` decimal (default), `o` octal, `x` ou `X` hexadecimal…
  - float: `e` or `E` exponential, `f` or `F` fixed point, `g` ou `G` appropriate (default),
  - `%` pourcent

Selection parameter (apparition order by default):

- `2` → argument index 2 (the 3rd)
- `y` → argument named y

```
"…".format(x=3,y=2,z=12)
```

- `0.name` → attribute name of argument index 0
- `0[name]` → value for key name of argument index 0
- `0[2]` → value for index 2 of argument index 0