Python 3.2 Reference Card

Summary necessarily incomplete to keep on one paper sheet, see http://docs.python.org/py3k.

Cheat Sheet Symbols

☍ optional instructions , ↻ repeatable instructions, ⚷ immutable data, ⇢ ordered container (⇝ non ordered), `constant`, *variable*, `type`, `function` & `.method`, *parameter*, *[,optional parameter]*, `keyword`, `literal`, *`module`*, *file*.

Introspection & Help

`help`(*[objet or "subject"]*)
`id`(*objet*) `dir`(*[object]*) `vars`(*[object]*)
`locals`() `globals`()

Qualified Access

Separator `.` between a **namespace** and a **name** in that space. Namespaces : object, class, function, module, package…. Examples :

`math``.sin`(*`math`*`.pi`) `f.__doc__`
`MyClasse.nbObjects`()
point`.x` *rectangle*`.width`()

Base Types

undefined ⚷ : `None`
Boolean ⚷: `bool` `True` / `False`
`bool`(*x*) ➙ `False` if *x* nul or empty
Integer ⚷: `int` `0` `165` `-57`
binary:`0b101` octal:`0o700` hexa:`0xf3e`
`int`(*x[,base]*) `.bit_length`()
Floating ⚷: `float` `0.0` `-13.2e-4`
`float`(*x*) `.as_integer_ratio`()
Complex ⚷: `complex` `0j` `-1.2e4+9.4j`
`complex`(*re[,img]*) `.real` `.imag`
`.conjugate`()
String ⚷⇢: `str` `''` `'toto'` `"toto"` `"""multiline toto"""`
`str`(*x*) `repr`(*x*)

Identifiers, Variables & Assignment

Identifiers : `[a-zA-Z_]` followed by or one or multiple `[a-zA-Z0-9_]`, accent and non-latin alphabetical chars allowed (but should be avoided).
name `=` *expression*
name1`,`*name2*…`,`*nameN* `=` *sequence*
sequence containing N items
name1 `=` *name2*… `=` *nameX* `=` *expression*
unpacking sequence: *first*`,`*`*`*remain*`=`*sequence*
increment : *name*`=`*name*`+`*expression*
augmented assignment : *name*`+=`*expression* (with other operators too)
deletion : `del` *nom*

Identifiers Conventions

Details in PEP 8 "Style Guide for Python"

`A_CONSTANT`	uppercase
`alocalvar`	lowercase without _
`a_global_var`	lowercase with _
`a_function`	lowercase with _
`a_method`	lowercase with _
`AClass`	title
`AnExceptionError`	title with Error at end
`amodule`	lowercase rather without _
`apackage`	lowercase rather without _

Avoid `l O I` (l min, o maj, i maj) alone.

`_xxx`	internal usage
`__xxx`	modified _Class__xxx
`__xxx__`	spécial reserved name

Logical Operations

a`<`*b* *a*`<=`*b* *a*`>=`*b* *a*`>`*b* a=b➙*a*`==`*b* a≠b➙*a*`!=`*b*
`not` *a* *a* `and` *b* *a* `or` *b* (*expr*)
combined : `12<`*x*`<=34`

Maths

`-`*a* *a*`+`*b* *a*`-`*b* *a*`*`*b* *a*`/`*b* a`**`➙*a*`**`*b* (*expr*)
euclidian division a=b.q+r ➙ *q*=*a*`//`*b* et *r*=*a*`%`*b* et *q,r*=`divmod`(*a,b*)
|x|➙`abs`(*x*) x^y^z➙`pow`(*x,y[,z]*) `round`(*x[,n]*)
following functions/data in module *`math`*
e *pi* `ceil`(x) `floor`(*x*) `trunc`(*x*)
e^x^➙`exp`(*x*) `log`(*x*) √➙`sqrt`(*x*)
`cos`(*x*) `sin`(*x*) `tan`(*x*) `acos`(*x*) `asin`(*x*)
`atan`(*x*) `atan2`(*x,y*) `hypot`(*x,y*)
`cosh`(*x*) `sinh`(*x*)…
following functions in module *`random`*
`seed`(*[x]*) `random`() `randint`(*a,b*)
`randrange`(*[start],end[,step]*) `uniform`(*a,b*)
`choice`(*seq*) `shuffle`(*x[,rnd]*) `sample`(*pop,k*)

Bits Manipulations

(with integers) *a*`<<`*b* *a*`>>`*b* *a*`&`*b* *a*`|`*b* *a*`^`*b*

String

Escape : `\`
`\\` ➙ `\` `\'` ➙ `'` `\"` ➙ `"`
`\n` ➙ new line `\t` ➙ tab
`\N{`*name*`}` ➙ unicode *name*
`\x`*hh* ➙ *hh* hexa `\0`*oo* ➙ *oo* octal
`\u`*hhhh* et `\U`*hhhhhhhh* ➙ unicode hexa *hhhh*
prefix `r`, disable `\` : `r"\n"` ➙ `\n`
Formating : `"{`*model*`}".format`(*data…*)

```
"{} {}".format(3,2)
"{1} {0} {0}".format(3,9)
"{x} {y}".format(y=2,x=5)
"{0!r} {0!s}".format("text\n")
"{0:b}{0:o}{0}{0:x}".format(100)
"{0:0.2f}{0:0.3g}{0:.1e}".format(1.45)
```

Operations
s`*`*n* (repeat) *s1*`+`*s2* (concatenate) `*= +=`
`.split`(*[sep[,n]]*) `.join`(*iterable*)
`.splitlines`(*[keepend]*) `.partition`(*sep*)
`.replace`(*old,new[,n]*) `.find`(*s[, start[,end]]*)
`.count`(*s[, start[,end]]*) `.index`(*s[, start[,end]]*)
`.isdigit`() & Co `.lower`() `.upper`()
`.strip`(*[chars]*)
`.startswith`(*s[,start[,end]]*)
`.endsswith`(*s[,start[,end]]*)
`.encode`(*[enc[, err]]*)
`ord`(*c*) `chr`(*i*)

Conditional Expression

Evaluated as a value.
expr1 `if` *condition* `else` *expr2*

Flow Control

statements blocs delimited by indentation (idem functions, classes, methods). Convention 4 spaces - tune editor.

Alternative If

```
if condition1:
    # block executed if condition1 is true
elif condition2: ☍↻
    # block executed if condition2 is true
else: ☍
    # block executed if all conditions are false
```

Loop Over Sequence

```
for var in iterable:
    # block executed with var being successively
    # each of the values in iterable
else: ☍
    # executed after, except if exit for loop by break
```

var with multiple variables : `for` *x,y,z* `in`…
var index,valuer : `for` *i,v* `in` `enumerate`(…)
iterable : see Containers & Iterables

Loop While

```
while condition:
    # block executed while condition is true
else: ☍
    # executed after, except if exit while loop by break
```

Loop Break : `break`
Immediate exit of the loop, without going through else block.
Loop Jump : `continue`
Immediate jump at loop start with next iteration.

Errors Processing: Exceptions

```
try:
    # block executed in normal case
except exc as e: ↻
    # block executed if an error of type exc is detected
else:
    # block executed in case of normal exit from try
finally:
    # block executed in all case
```

exc for n types : `except` (*exc1*`,`*exc2*…`,`*excn*)
`as` *e* optional, fetch exception
⚠ detect specific exceptions (ex. ValueError) and not generic ones (ex. Exception).
Raising Exeptions (error situation)
`raise` *exc*(*[args]*)
`raise` ➙ ⚠ propagate exception
Some exception classes : `Exception` - `ArithmeticError` - `ZeroDivisionError` - `IndexError` - `KeyError` - `AttributeError` - `IOError` - `ImportError` - `NameError` - `SyntaxError` - `TypeError` - `NotImplementedError`…
Managed Context

```
with managed() as v ☍:
    # Block executed in a managed context
```

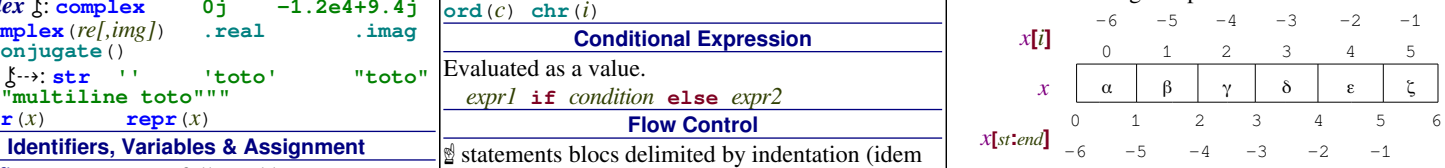
Function Definition and Call

```
def fname(x,y=4,*args,**kwargs):
    # function block or, if no code, pass
    return ret_expression ☍
```

x: simple parameter
y: parameter with default value
args: variable parameters by order (`tuple`)
kwargs: named variable parameters (`dict`)
ret_expression: `tuple` ➙ return multiple values
Call
res = *fname*(*expr*, *param*=*expr*, **tuple*, ***dict*)
Anonymous Functions
`lambda` *x,y*`:` *expression*

Sequences & Indexation

for any direct access ordered container.
i^th^ ***Item*** : *x*`[`*i*`]`
Slice : *x*`[`*start*`:`*end*`]` *x*`[`*start*`:`*end*`:`*step*`]`
i, *start*, *end*, *step* integers positive or negative
start/*end* missing ➙ up to start/end



Modification (if sequence is modifiable)
x`[`*i*`]=`*expression* *x*`[`*start*`:`*end*`]=`*iterable*
`del` *x*`[`*i*`]` `del` *x*`[`*start*`:`*end*`]`

Containers & Iterables

An *iterable* provide values one after the other. Ex : containers, dictionary views, iterable objets, generator functions…
Generators (calculate values when needed)
`range`(*[start,]end[,step]*)
`(` *expr* `for` *var* `in` *iter* ↻ `if` *cond* ☍ `)`
Generic Operations
v `in` *containeur* *v* `not in` *containeur*
`len`(*containeur*) `enumerate`(*iter[,start]*)
`iter`(*o[,sent]*) `all`(*iter*) `any`(*iter*)
`filter`(*fct,iter*) `map`(*fct,iter,…*)
`max`(*iter*) `min`(*iter*) `sum`(*iter[,start]*)
`reversed`(*seq*) `sorted`(*iter[,k][,rev]*)
On sequences : `.count`(*x*) `.index`(*x[,i[,j]]*)
String ⚷⇢ : (sequence of chars)
cf. types `bytes`, `bytearray`, `memoryview` to directly manipulate bytes (+notation `b"bytes"`).
List ⇢ : `list` `[]` `[1,'toto',3.14]`
`list`(*iterable*) `.append`(*x*)
`.extend`(*iterable*) `.insert`(*i,x*) `.pop`(*[i]*)
`.remove`(*x*) `.reverse`() `.sort`()
`[` *expr* `for` *var* `in` *iter* ↻ `if` *cond* ☍ `]`
Tuple ⚷⇢ : `tuple` `()` `(9,'x',36)` `(1,)`
`tuple`(*iterable*) `9,'x',36` `1,`
Set ⇝ : `set` `{1,'toto',42}`
`set`(*iterable*) ⚷⇝ ➙ `frozenset`(*iterable*)
`.add`(*x*) `.remove`(*x*) `.discard`(*x*)
`.copy`() `.clear`() `.pop`()
∪➙`|`, ∩➙`&`, diff➙`-`, sym.diff➙`^`, ⊂…➙`<`…
`|= &= -= ^=` …
Dictionnary (associative array, map) ⇝ : `dict`
`{}` `{1:'one',2:'two'}`
`dict`(*iterable*) `dict`(*a*`=`2`,`*b*`=`4)
`dict.fromkeys`(*seq[,val]*)
d`[`*k*`]=`*expr* *d*`[`*k*`]` `del` *d*`[`*k*`]`
`.update`(*iter*) `.keys`() `.values`()
`.items`() `.pop`(*k[,def]*) `.popitem`()
`.get`(*k[,def]*) `.setdefault`(*k[,def]*)
`.clear`() `.copy`()
`items`, `keys`, `values` iterable "views".

Input/Output & Files

`print`("x=",*x[,y…][,sep=…][,end=…][,file=…]*)
`input`("Age ? ") ➙ `str`

☞ explicit cast to `int` or `float` if needed.

File : *f*=`open`(*name[,mode][,encoding=…]*)

mode : `'r'` read (default) `'w'` write `'a'` append `'+'` read write `'b'` binary mode …

encoding : `'utf-8'` `'latin1'` `'ascii'`…

`.write`(*s*) `.read`(*[n]*) `.readline`()
`.flush`() `.close`() `.readlines`()

Loop in lines :`for` *line* `in` *f* :…

Managed context (close) : `with open`(…) `as` *f*:

🖄 in module ***os*** (see also ***os.path***):

`getcwd`() `chdir`(*path*) `listdir`(*path*)

Command line parameters in ***sys.argv***

Modules & Packages

Module : script file extension *.py* (and C compiled modules). File *toto.py* ➡ module `toto`.

Package : directory with file *__init__.py*. Contains module files.

Searched in the PYTHONPATH, see ***sys.path*** list.

Module Sample :

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
"""Module documentation - cf PEP257"""
# File: mymodule.py
# Author: Joe Student
# Import other modules, functions…
import math
from random import seed,uniform
# Definition of constants and globals
MAXIMUM = 4
lstFiles = []
# Definition of functions and classes
def f(x):
    """Function documentation"""
    …
class Converter(object):
    """Class documentation"""
    nb_conv = 0 # class var
    def __init__(self,a,b):
        """init documentation"""
        self.v_a = a # instance var
        …
    def action(self,y):
        """Method documentation"""
        …
# Module auto-test
if __name__ == '__main__':
    if f(2) != 4: # problem
    …
```

Modules / Names Imports

```
import mymodule
from mymodule import f,MAXIMUM
from mymodule import *
from mymodule import f as fct
```

To limit `*` effect, define in ***mymodule*** :

```
__all__ = [ "f", "MAXIMUM"]
```

Import via package :

```
from os.path import dirname
```

Class Definition

Special methods, reserved names `__xxxx__`.

```
class ClassName([superclass]) :
    # class block
    class_variable = expression
    def __init__(self[,params…]) :
        # initialization block
        self.instance_variable = expression
    def __del__(self) :
        # destruction block
    @staticmethod       # @↔"decorator"
    def fct([,params…]) :
        # static method (callable without object)
```

Membership Tests

```
isinstance(obj,class)
isssubclass(subclass,parentclass)
```

Objects Creation

Use the class as a function, parameters are passed to constructor `__init__`.

```
obj = ClasseName(params…)
```

Special Conversion Methods

```
def __str__(self) :
    # return display string
def __repr__(self) :
    # return representation string
def __bytes__(self) :
    # return bytes string object
def __bool__(self) :
    # return a boolean
def __format__(self,format_spec) :
    # return a string following specified format
```

Special Comparison Mehods

Return `True`, `False` or `NotImplemented`.

```
x<y  ➡ def __lt__(self,y) :
x<=y ➡ def __le__(self,y) :
x==y ➡ def __eq__(self,y) :
x!=y ➡ def __ne__(self,y) :
x>y  ➡ def __gt__(self,y) :
x>=y ➡ def __ge__(self,y) :
```

Special Operations Methods

Return a new object of the class, containing the operation result, or `NotImplemented` if cannot work with given *y* argument.

x ➡ *self*

```
x+y  ➡ def __add__(self,y) :
x-y  ➡ def __sub__(self,y) :
x*y  ➡ def __mul__(self,y) :
x/y  ➡ def __truediv__(self,y) :
x//y ➡ def __floordiv__(self,y) :
x%y  ➡ def __mod__(self,y) :
divmod(x,y) ➡ def __divmod__(self,y) :
x**y ➡ def __pow__(self,y) :
pow(x,y,z) ➡ def __pow__(self,y,z) :
x<<y ➡ def __lshift__(self,y) :
x>>y ➡ def __rshift__(self,y) :
x&y  ➡ def __and__(self,y) :
x|y  ➡ def __or__(self,y) :
x^y  ➡ def __xor__(self,y) :
-x   ➡ def __neg__(self) :
+x   ➡ def __pos__(self) :
abs(x) ➡ def __abs__(self) :
~x   ➡ def __invert__(self) :
```

Following methods called after, on *y* if *x* don't support required operation.

y ➡ *self*

```
x+y  ➡ def __radd__(self,x) :
x-y  ➡ def __rsub__(self,x) :
x*y  ➡ def __rmul__(self,x) :
x/y  ➡ def __rtruediv__(self,x) :
x//y ➡ def __rfloordiv__(self,x) :
x%y  ➡ def __rmod__(self,x) :
divmod(x,y) ➡ def __rdivmod__(self,x) :
x**y ➡ def __rpow__(self,x) :
x<<y ➡ def __rlshift__(self,x) :
x>>y ➡ def __rrshift__(self,x) :
x&y  ➡ def __rand__(self,x) :
x|y  ➡ def __ror__(self,x) :
x^y  ➡ def __rxor__(self,x) :
```

Special Augmented Assignment Methods

Modify *self* object on which they are applied.

x ➡ *self*

```
x+=y  ➡ def __iadd__(self,y) :
x-=y  ➡ def __isub__(self,y) :
x*=y  ➡ def __imul__(self,y) :
x/=y  ➡ def __itruediv__(self,y) :
x//=y ➡ def __ifloordiv__(self,y) :
x%=y  ➡ def __imod__(self,y) :
x**=y ➡ def __ipow__(self,y) :
x<<=y ➡ def __ilshift__(self,y) :
x>>=y ➡ def __irshift__(self,y) :
x&=y  ➡ def __iand__(self,y) :
x|=y  ➡ def __ior__(self,y) :
x^=y  ➡ def __ixor__(self,y) :
```

Special Numerical Conversion Methods

Return the converted value.

x ➡ *self*

```
complex(x) ➡ def __complex__(self) :
int(x)     ➡ def __int__(self) :
float(x)   ➡ def __float__(self) :
round(x,n) ➡ def __round__(self,n) :
def __index__(self) :
    # return an int usable as index
```

Special Attribute Access Methods

Access with *obj.name*. Exception `AttributeError` if attribute not found.

obj ➡ *self*

```
def __getattr__(self,name) :
    # called if name not found as existing attribute
def __getattribute__(self,name) :
    # called in all case of name access.
def __setattr__(self,name,valeur) :
def __delattr__(self,name) :
def __dir__(self) : # return a list
```

Accessors

Property

```
class C(object) :
    def getx(self) : …
    def setx(self,value) : …
    def delx(self) : …
    x = property(getx,setx,delx,"docx")
    # Simpler, accessor to y, with decorators
    @property
    def y(self) :    # read
        """docy"""
    @y.setter
    def y(self,valeur) :    # modification
    @y.deleter
    def y(self) :    # deletion
```

Descriptors Protocol

```
o.x ➡ def __get__(self,o,classe_de_o) :
o.x=v ➡ def __set__(self,o,v) :
del o.x ➡ def __delete__(self,o):
```

Special Function Call Method

Use an object as if it was a function (callable) :

```
o(params) ➡ def __call__(self[,params…]):
```

Hash Special Method

For efficient storage in `dict` and `set`.

```
hash(o) ➡ def __hash__(self) :
```

Define to `None` if object not *hashable*.

Special Container Methods

o ➡ *self*

```
len(o) ➡ def __len__(self) :
o[key] ➡ def __getitem__(self,key) :
o[key]=v ➡ def __setitem__(self,key,v) :
del o[key] ➡ def __delitem__(self,key) :
for i in o: ➡ def __iter__(self) :
    # return a new iterator on the container
reversed(o) ➡ def __reversed__(self) :
x in o ➡ def __contains__(self,x) :
```

For notation `[`*start*`:`*end*`:`*step*`]`, a `slice` object is given to container methods as value for *key* parameter.

Slice: `slice`(*start*,*end*,*step*)
.start *.stop* *.step* `.indices`(*lentgh*)

Special Iterator Methods

```
def __iter__(self) :# return self
def __next__(self) :# return next item
```

If no more item, raise exception `StopIteration`.

Special Managed Context Methods

Used for `with` statement.

```
def __enter__(self) :
    # called at entry in the managed context
    # return value used for context' as variable
def __exit__(self,etype,eval,tb) :
    # called at exit of managed context
```

Special Metaclass Methods

```
__prepare__ = callable
def __new__(cls[,params…]) :
    # allocation and return a new cls object
isinstance(o,cls)
    ➡ def __instancecheck__(cls,o):
isssubclass(subclass,cls)
    ➡ def __subclasscheck__(cls,subclass):
```

Generators

Calculate values when needed (ex.: `range`).

Generator functions, contains a statement `yield`.

```
yield expression
yield from séquence
```

variable = (`yield` *expression*) transmission of values to the generator.

If no more item, raise exception `StopIteration`.

Generator Function Control

```
generator.__next__()
generator.send(value)
generator.throw(type[,value[,traceback]])
generator.close()
```