Abrégé v1.2.7

Abrégé Dense Python 3.1

Abrégé nécessairement incomplet pour tenir sur une feuille, voir sur http://docs.python.org/py3k.

Symbolique de l'Abrégé

♂ instructions optionnelles, ↻ instruction répétables, ⚷ valeur immutable (non modifiable), --› conteneur ordonné (⇝ non ordonné), `constante`, *variable*, `type`, `fonction` & `.méthode`, *paramètre*, *[,paramètre optionnel]*, `mot_clé`, `litéral`, `module`, *fichier*.

Introspection & Aide

`help`(*[objet ou "sujet"]*)
`id`(*objet*) `dir`(*[objet]*) `vars`(*[objet]*)
`locals`() `globals`()

Accès Qualifiés

Séparateur `.` entre un **espace de noms** et un **nom** dans cet espace. Espaces de noms : objet, classe, fonction, module, package…. Exemples :

```
math.sin(math.pi)        f.__doc__
MaClasse.nbObjets()
point.x                  rectangle.largeur()
```

Types de Base

non défini⚷ : `None`
Booléen ⚷: `bool` `True` / `False`
`bool`(*x*) ➙ `False` si *x* nul ou vide
Entier ⚷: `int` `0` `165` `-57`
binaire:`0b101` octal:`0o700` hexa:`0xf3e`
`int`(*x[,base]*) `.bit_length`()
Flottant ⚷: `float` `0.0` `-13.2e-4`
`float`(*x*) `.as_integer_ratio`()
Complexe ⚷: `complex` `0j` `-1.2e4+9.4j`
`complex`(*re[,img]*) `.real` `.imag`
`.conjugate`()
Chaîne ⚷-->: `str` `''` `'toto'` `"toto"`
`"""multiligne toto"""`
`str`(*x*) `repr`(*x*)

Identificateurs, Variables & Affectation

Identificateurs : `[a-zA-Z_]` suivi d'un ou plusieurs `[a-zA-Z0-9_]`, accents et caractères alphabétiques non latins autorisés (mais à éviter).

nom `=` *expression*
nom1`,`*nom2*…`,`*nomN* `=` *séquence*
séquence contenant N éléments
nom1 `=` *nom2*… `=` *nomX* `=` *expression*
☞ éclatement séquence: *premier*`,`*`*suite`=*séquence*
☞ incrémentation : *nom*`=`*nom*`+`*expression*
⬌ affectation augmentée : *nom*`+=`*expression*
☞ suppression : `del` *nom*

Conventions Identificateurs

Détails dans PEP 8 "Style Guide for Python"

`UNE_CONSTANTE`	majuscules
`unevarlocale`	minuscules sans _
`une_var_globale`	minuscules avec _
`une_fonction`	minuscules avec _
`une_methode`	minuscules avec _
`UneClasse`	titré
`UneExceptionError`	titré avec Error à la fin
`unmodule`	minuscules plutôt sans _
`unpackage`	minuscules plutôt sans _

Éviter `l O I` (l min, o maj, i maj) seuls.

`_xxx`	usage interne
`__xxx`	transformé _Classe__xxx
`__xxx__`	nom spécial réservé

Opérations Logiques

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*)
☞ combinables : `12<`*x*`<=34`

Maths

`-`*a* *a*`+`*b* *a*`-`*b* *a*`*`*b* *a*`/`*b* a^b➙*a*`**`*b* `(`*expr*`)`
division euclidienne 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]*)
☞ fonctions/données suivantes dans le 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*)…
☞ fonctions suivantes dans le module `random`
`seed`(*[x]*) `random`() `randint`(*a,b*)
`randrange`(*[déb],fin[,pas]*) `uniform`(*a,b*)
`choice`(*seq*) `shuffle`(*x[,rnd]*) `sample`(*pop,k*)

Manipulations de bits

(sur les entiers) *a*`<<`*b* *a*`>>`*b* *a*`&`*b* *a*`|`*b* *a*`^`*b*

Chaîne

Échappements : `\`
`\\` ➙ `\` `\'` ➙ `'` `\"` ➙ `"`
`\n` ➙ retour à la ligne `\t` ➙ tabulation
`\N{`*nom*`}` ➙ unicode *nom*
`\x`*hh* ➙ *hh* hexa `\0`*oo* ➙ *oo* octal
`\u`*hhhh* et `\U`*hhhhhhhh* ➙ unicode hexa *hhhh*
☞ préfixe `r`, désactivation du `\` : `r"\n"` ➙ `\n`
Formatage : `"{`*modèle*`}".format(`*données…*`)`

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

Opérations
s`*`*n* (répétition) *s1*`+`*s2* (concaténation)
`.split(`*[sep[,n]]*`)` `.join(`*iterable*`)`
`.partition(`*sep*`)` `.replace(`*old,new[,n]*`)`
`.find(`*s[, déb[,fin]]*`)` `.count(`*s[, déb[,fin]]*`)`
`.index(`*s[, déb[,fin]]*`)` `.isdigit`() & Co
`.lower`() `.upper`() `.strip(`*[chars]*`)`
`.startswith(`*s[,déb[,fin]]*`)`
`.encode(`*[enc[, err]]*`)`
`ord(`*c*`)` `chr(`*i*`)`

Expression Conditionnelle

Évaluée comme une valeur.
expr1 `if` *condition* `else` *expr2*

Contrôle de Flux

☞ blocs d'instructions délimités par l'indentation (idem fonctions, classes, méthodes). Convention 4 espaces - régler l'éditeur.

Alternative Si

```
if condition1:
    # bloc exécuté si condition1 est vraie
elif condition2: ♂↻
    # bloc exécuté si condition2 est vraie
else: ♂
    # bloc exécuté si toutes conditions fausses
```

Boucle Parcours De Séquence

```
for var in itérable:
    # bloc exécuté avec var valant tour à tour
    # chacune des valeurs de itérable
else: ♂
    # exécuté après, sauf si sortie du for par break
```

☞ *var* à plusieurs variables: `for` *x,y,z* `in`…
☞ *var* index,valeur: `for` *i,v* `in enumerate`(…)
☞ *itérable : voir Conteneurs & Itérables*

Boucle Tant Que

```
while condition:
    # bloc exécuté tant que condition est vraie
else: ♂
    # exécuté après, sauf si sortie du while par break
```

Rupture De Boucle : `break`
Sortie immédiate de la boucle, sans passer par le bloc else.
Saut De Boucle : `continue`
Saut immédiat en début de bloc de la boucle pour exécuter l'itération suivante.

Traitement D'erreurs: Exceptions

```
try:
    # bloc exécuté dans les cas normaux
except exc as e: ↻
    # bloc exécuté si une erreur de type exc est
    # détectée
else:
    # bloc exécuté en cas de sortie normale du try
finally:
    # bloc exécuté dans tous les cas
```

☞ *exc* pour n types : `except` (*exc1*`,`*exc2*…`,`*excn*)
☞ `as` *e* optionnel, récupère l'exception
⚠ détecter des exceptions précises (ex. ValueError) et non génériques (ex. Exception).

Levée D'exception (situation d'erreur)
`raise` *exc* `(`*[args]*`)`
☞ `raise`, sans valeur, dans bloc except, pour propager l'exception et ne pas la bloquer.
Quelques classes d'exceptions : `Exception` - `ArithmeticError` - `ZeroDivisionError` - `IndexError` - `KeyError` - `AttributeError` - `IOError` - `ImportError` - `NameError` - `SyntaxError` - `TypeError`…

Contexte Géré

```
with garde() as v:
    # Bloc exécuté dans un contexte géré
```

☞ `as` *v* optionnel
☞ Typiquement: `with open`(…) `as` *f*:

Définition et Appel de Fonction

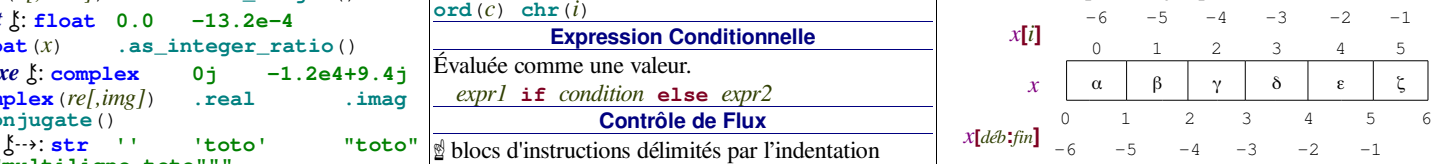
```
def nomfct(x,y=4,*args,**kwargs):
    # le bloc de la fonction ou à défaut pass
    return ret_expression ♂
```

x: paramètre simple
y: paramètre avec valeur par défaut
args: paramètres variables par ordre (`tuple`)
kwargs: paramètres variables nommés (`dict`)
ret_expression: `tuple` ➙ retour de plusieurs valeurs

Appel
res `=` *nomfct*`(`*expr*`,`*param*`=`*expr*`,*`*tuple*`,**`*dict*`)`

Séquences & Indexation

☞ pour tout conteneur ordonné à accès direct.
*i*e ***Élément*** : *x*`[`*i*`]`
Tranche (slice) : *x*`[`*déb*`:`*fin*`]` *x*`[`*déb*`:`*fin*`:`*pas*`]`
☞ *i*, *déb*, *fin*, *pas* entiers positifs ou négatifs
☞ *déb*/*fin* manquant ➙ jusqu'au bout



Modification (si séquence modifiable)
x`[`*i*`]=`*expression* *x*`[`*déb*`:`*fin*`]=`*itérable*
`del` *x*`[`*i*`]` `del` *x*`[`*déb*`:`*fin*`]`

Conteneurs & Itérables

Un *itérable* est un objet qui permet d'accéder aux valeurs qu'il contient l'une après l'autre, cela inclus : conteneurs, vues sur dictionnaires, objets itérables, fonctions générateurs…

Générateurs (calcul des valeurs lorsque nécessaire)
`range`(*[déb,]fin[,pas]*)
`(` *expr* `for` *var* `in` *iter* ↻ `if` *cond* ♂ `)`

Opérations Génériques
v `in` *conteneur* *v* `not in` *conteneur*
`len`(*conteneur*) `enumerate`(*iter[,déb]*)
`iter`(*o[,sent]*) `all`(*iter*) `any`(*iter*)
`filter`(*fct,iter*) `map`(*fct,iter,…*)
`max`(*iter*) `min`(*iter*) `sum`(*iter[,déb]*)
`reversed`(*seq*) `sorted`(*iter[,k][,rev]*)

Chaîne ⚷--› : (séquence de caractères)
☞ cf. types `bytes` et `bytearray` pour manipuler directement des octets (+notation `b"octets"`).

Liste --› : `list` `[]` `[1,'toto',3.14]`
`list`(*iterable*) `.append`(*x*) `.count`(*x*)
`.extend`(*iterable*) `.index`(*x[,i[,j]]*)
`.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,`

Ensemble ⇝ : `set` `{1,'toto',42}`
`set`(*iterable*)

Ensemble Figé ⚷⇝ : `frozenset`
`frozenset`(*iterable*)
☞ opérateurs et méthodes pour réaliser des opérations ensemblistes (∈ ∉ ∩ ∪ ⊂ ⊄…).

Dictionnaire (tableau associatif, 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` retournent des "vues" sur le dictionnaire, itérables et supportant certaines opérations ensemblistes.

Entrées/Sorties & Fichiers

`print`(`"x=",`*x[,y…][,sep=…][,end=…][,file=…]*`)`

```
v = input("Age ? ")
```
☞ faire un transtypage explicite en `int` ou `float` si besoin : `int(input("..."))`

Fichier : *f*=`open`(*nom[,mode][,encoding=...]*)

mode : `'r'` lecture (défaut) `'w'` écriture `'a'` ajout `'+'` lecture écriture `'b'` mode binaire...

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

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

`for` *line* `in` *f* :...

dans le module ***os*** (voir aussi ***os.path***):

`getcwd`() `chdir`(*chemin*) `listdir`(*chemin*)

Paramètres ligne de commande dans ***sys***.*argv*

Modules & Packages

Module : fichier script extension *.py* (et modules compilés en C). Fichier *toto.py* ➡ module ***toto***.

Package : répertoire avec fichier *__init__.py*. Contient des fichiers modules.

Recherchés dans le PYTHONPATH, voir liste ***sys***.*path*.

Modèle De Module :

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
"""Documentation module - cf PEP257"""
# Fichier: monmodule.py
# Auteur: Joe Student
# Import d'autres modules, fonctions…
import math
from random import seed,uniform
# Définitions constantes et globales
MAXIMUM = 4
lstFichiers = []
# Définitions fonctions et classes
def f(x):
    """Documentation fonction"""
    …
class Convertisseur(object):
    """Documentation classe"""
    nb_conv = 0 # var de classe
    def __init__(self,a,b):
        """Documentation init"""
        self.v_a = a # var d'instance
        …
    def action(self,y):
        """Documentation méthode"""
        …
# Auto-test du module
if __name__ == '__main__':
    if f(2) != 4: # problème
    …
```

Import De Modules / De Noms

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

Pour limiter l'effet `*`, définir dans ***monmodule*** :

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

Import via package :

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

Définition de Classe

Méthodes spéciales, noms réservées `__xxxx__`.

```
class NomClasse([claparent]) :
    # le bloc de la classe
    variable_de_classe = expression
    def __init__(self[,params...]) :
        # le bloc de l'initialiseur
        self.variable_d_instance = expression
    def __del__(self) :
        # le bloc du destructeur
    @staticmethod       # @ ↔ "décorateur"
    def fct([,params...]) :
        # méthode statique (appelable sans objet)
```

Tests D'appartenance

`isinstance`(*obj*,*classe*)

`isssubclass`(*sousclasse*,*parente*)

Création d'Objets

Utilisation de la classe comme une fonction, paramètres passés à l'initialiseur `__init__`.

obj = `NomClasse`(*params...*)

Méthodes spéciales Conversion

```
def __str__(self) :
    # retourne chaîne d'affichage, utilisé par print(obj),
    # str(obj) ou "{!s}".format(obj)
def __repr__(self) :
    # retourne chaîne de représentation
    # utilisé par repr(obj) ou "{!r}".format(obj)
def __bool__(self) :
    # retourne un booléen
def __format__(self,spécif_format) :
    # retourne chaîne suivant le format spécifié
```

Méthodes spéciales Comparaisons

Retournent `True`, `False` ou `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) :`

Méthodes spéciales Opérations

Retournent un nouvel objet de la classe, intégrant le résultat de l'opération, ou `NotImplemented` si ne peuvent travailler avec l'argument *y* donné.

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) :`

Méthodes suivantes appelées ensuite avec *y* si *x* ne supporte pas l'opération désirée.

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) :`

Méthodes spéciales Affectation augmentée

Modifient l'objet *self* auquel elles s'appliquent.

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) :`

Méthodes spéciales Conversion numérique

Retournent la valeur convertie.

x ➡ *self*

`complex`(*x*) ➡ `def __complex__(self,x) :`
`int`(*x*) ➡ `def __int__(self,x) :`
`float`(*x*) ➡ `def __float__(self,x) :`
`round`(*x*,*n*) ➡ `def __round__(self,x,n) :`
`def __index__(self) :`
    # retourne un entier utilisable comme index

Méthodes spéciales Accès aux attributs

Accès par *obj*.*nom*. Exception `AttributeError` si attribut non trouvé.

obj ➡ *self*

```
def __getattr__(self,nom) :
    # appelé si nom non trouvé en attribut existant,
def __getattribute__(self,nom) :
    # appelé dans tous les cas d'accès à nom
def __setattr__(self,nom,valeur) :
def __delattr__(self,nom) :
def __dir__(self) : # retourne une liste
```

Accesseurs

Property

```
class C(object) :
  def getx(self) : ...
  def setx(self,valeur) : ...
  def delx(self) : ...
  x = property(getx,setx,delx,"docx")
  # Plus simple, accesseurs à y, avec des décorateurs
  @property
  def y(self) :                 # lecture
    """docy"""
  @y.setter
  def y(self,valeur) :          # modification
  @y.deleter
  def y(self) :                 # suppression
```

Protocole Descripteurs

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

Méthode spéciale Appel de fonction

Utilisation d'un objet comme une fonction (callable) :

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

Méthode spéciale Hachage

Pour stockage efficace dans `dict` et `set`.

`hash`(*o*) ➡ `def __hash__(self) :`

Définir à `None` si objet non *hachable*.

Méthodes spéciales Conteneur

o ➡ *self*

`len`(*o*) ➡ `def __len__(self) :`
o[*clé*] ➡ `def __getitem__(self,clé) :`
o[*clé*]=*v* ➡ `def __setitem__(self,clé,v) :`
`del` o[*clé*] ➡ `def __delitem__(self,clé) :`
`for` *i* `in` *o*: ➡ `def __iter__(self) :`
    # retourne un nouvel itérateur sur le conteneur
`reversed`(*o*) ➡ `def __reversed__(self) :`
x `in` o ➡ `def __contains__(self,x) :`

Pour la notation `[`*déb*`:`*fin*`:`*pas*`]`, un objet de type `slice` est donné comme valeur de *clé* aux méthodes conteneur.

Tranche : `slice`(*déb*,*fin*,*pas*)
.*start* .*stop* .*step* `.indices`(*longueur*)

Méthodes spéciales Itérateurs

```
def __iter__(self) :# retourne self
def __next__(self) :# retourne l'élément suivant
```

Si plus d'élément, levée exception `StopIteration`.

Méthodes spéciales Contexte Géré

Utilisées pour le `with`.

```
def __enter__(self) :
    # appelée à l'entrée dans le contexte géré
    # valeur utilisée pour le as du contexte
def __exit__(self,etype,eval,tb) :
    # appelée à la sortie du contexte géré
```

Méthodes spéciale Métaclasses

```
__prepare__ = callable
def __new__(cls[,params...]) :
    # allocation et retour d'un nouvel objet cls
```

`isinstance`(*o*,*cls*) ➡ `def __instancecheck__(cls,o):`

`isssubclass`(*sousclasse*,*cls*) ➡ `def __subclasscheck__(cls,sousclasse):`

Générateurs

Calcul des valeurs lorsque nécessaire (ex.: `range`).

Fonction générateur, contient une instruction :
`yield` *expression* qui fait une pause dans la fonction et retourne la valeur, l'exécution reprend quand on a besoin de la valeur suivante.

Instruction *variable* = (`yield` *expression*) pour pouvoir transmettre des valeurs au générateur.

Si plus de valeur, levée exception `StopIteration`.

Contrôle Fonction Générateur

générateur`.__next__()`
générateur`.send`(*valeur*)
générateur`.throw`(*type*[,*valeur*[,*traceback*]])
générateur`.close()`