

entier, flottant, booléen, chaîne Types de base

int 783 0 -192
float 9.23 0.0 -1.7e-6
bool True False
str "Un\nDeux" 'L\âme'

retour à la ligne
multiligne { "X\tY\tZ", "1\t2\t3" }
non modifiable, séquence ordonnée de caractères
échappé
tabulation

Types Conteneurs

■ séquences ordonnées, accès index rapide, valeurs répétables
list [1,5,9] ["x",11,8.9] ["mot"] []
tuple (1,5,9) 11,"y",7.4 ("mot",) ()
non modifiable
expression juste avec des virgules
■ sans ordre a priori, clé unique, accès par clé rapide ; clés = types de base ou tuples
dict {"clé": "valeur"}
dictionnaire couples clé/valeur
ensemble
set {"clé1", "clé2"} {1,9,3,0} set()

pour noms de variables, fonctions, modules, classes... Identificateurs

a..zA..Z_ suivi de a..zA..Z_0..9
□ accents possibles mais à éviter
□ mots clés du langage interdits
□ distinction casse min/MAJ
◎ a toto x7 y_max BigOne
◎ 8y and

Affectation de variables

x = 1.2+8+sin(0)
↑ valeur ou expression de calcul
nom de variable (identificateur)
y,z,r = 9.2,-7.6,"bad"
noms de variables conteneur de plusieurs valeurs (ici un tuple)
x+=3 incrémentation
x-=2 décrémentation
x=None valeur constante « non défini »

type(expression) Conversions

int ("15") on peut spécifier la base du nombre entier en 2nd paramètre
int (15.56) troncature de la partie décimale (round(15.56) pour entier arrondi)
float ("-11.24e8")
str (78.3) et pour avoir la représentation littérale → repr("Texte")
voir au verso le formatage de chaînes, qui permet un contrôle fin
bool → utiliser des comparateurs (avec ==, !=, <, >, ...), résultat logique booléen
list ("abc") → utilise chaque élément de la séquence en paramètre → ['a', 'b', 'c']
dict ((3, "trois"), (1, "un")) → {1: 'un', 3: 'trois'}
set ("un", "deux") → utilise chaque élément de la séquence en paramètre → {'un', 'deux'}
:"".join(['toto', '12', 'pswd']) → 'toto:12:pswd'
chaîne de jointure séquence de chaînes
"des mots espacés".split() → ['des', 'mots', 'espacés']
"1,4,8,2".split(",") → ['1', '4', '8', '2']
chaîne de séparation

pour les listes, tuples, chaînes de caractères... Indexation des séquences

index négatif	-6	-5	-4	-3	-2	-1
index positif	0	1	2	3	4	5

lst = [11, 67, "abc", 3.14, 42, 1968]
tranche positive 0 1 2 3 4 5 6
tranche négative -6 -5 -4 -3 -2 -1

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]

len(lst) → 6
accès individuel aux éléments par [index]
lst[1] → 67 **lst**[0] → 11 le premier
lst[-2] → 42 **lst**[-1] → 1968 le dernier
accès à des sous-séquences par [tranche début : tranche fin : pas]
lst[1:3] → [67, "abc"]
lst[-3:-1] → [3.14, 42]
lst[:3] → [11, 67, "abc"]
lst[4:] → [42, 1968]

Indication de tranche manquante → à partir du début / jusqu'à la fin.

Sur les séquences modifiables, utilisable pour suppression **del lst[3:5]** et modification par affectation **lst[1:4] = ['hop', 9]**

Logique booléenne

Comparateurs: < > <= >= == !=
≤ ≥ = ≠
a and b et logique
les deux en même temps ou logique
a or b l'un ou l'autre ou les deux non logique
not a
True valeur constante vrai
False valeur constante faux

Blocs d'instructions

instruction parente:
bloc d'instructions 1...
instruction parente:
bloc d'instructions 2...
instruction suivante après bloc 1

bloc d'instructions exécuté uniquement si une condition est vraie Instruction conditionnelle

if expression logique:
→ bloc d'instructions
combinable avec des sinon si, sinon si... et un seul sinon final, exemple:
if x==42:
bloc si expression logique x==42 vraie
print("vérité vraie")
elif x>0:
bloc sinon si expression logique x>0 vraie
print("positivons")
elif bTermine:
bloc sinon si variable booléenne bTermine vraie
print("ah, c'est fini")
else:
bloc sinon des autres cas restants
print("ça veut pas")

Maths

Opérateurs: + - * / // % **
x ÷ ↑ a^b
÷ entière reste ÷
(1+5.3)*2 → 12.6
abs (-3.2) → 3.2
round (3.57, 1) → 3.6
angles en radians
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)

bloc d'instructions exécuté tant que la condition est vraie

Instruction boucle conditionnelle

while expression logique :

→ bloc d'instructions

s = 0
i = 1 initialisations avant la boucle

condition avec au moins une valeur variable (ici **i**)

while i <= 100:

bloc exécuté tant que $i \leq 100$

s = s + i2**
i = i + 1 faire varier la variable de condition !

print("somme:", s) résultat de calcul après la boucle

attention aux boucles sans fin !

Contrôle de boucle

break sortie immédiate

continue itération suivante

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

bloc d'instructions exécuté pour chaque élément d'un conteneur ou d'un itérateur

Instruction boucle itérative

for variable in séquence :

→ bloc d'instructions

Parcours des valeurs de la séquence

s = "Du texte" initialisations avant la boucle

cpt = 0

variable de boucle, valeur gérée par l'instruction **for**

for c in s:

if c == "e":

cpt = cpt + 1

Comptage du nombre de **e** dans la chaîne.

print("trouvé", cpt, "e")

boucle sur dict/set = boucle sur séquence des clés

utilisation des tranches pour parcourir un sous-ensemble de la séquence

Parcours des **index** de la séquence

□ changement de l'élément à la position

□ accès aux éléments autour de la position (avant/après)

lst = [11, 18, 9, 12, 23, 4, 17]

perdu = []

for idx in range(len(lst)):

val = lst[idx]

if val > 15:

perdu.append(val)

lst[idx] = 15

Bornage des valeurs supérieures à 15, mémorisation des valeurs perdues.

print("modif:", lst, "-modif:", perdu)

Parcours simultané **index** et **valeur** de la séquence:

for idx, val in enumerate(lst):

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

Affichage / Saisie

éléments à afficher : valeurs littérales, variables, expressions

Options de **print**:

□ **sep=" "** (séparateur d'éléments, défaut espace)

□ **end="\n"** (fin d'affichage, défaut fin de ligne)

□ **file=f** (print vers fichier, défaut sortie standard)

s = input("Directives: ")

input retourne toujours une chaîne, la convertir vers le type désiré (cf encadré Conversions au recto).

len(c) → nb d'éléments

min(c) **max(c)** **sum(c)**

Opérations sur conteneurs

Note: Pour dictionnaires et ensembles, ces opérations travaillent sur les clés.

sorted(c) → copie triée

val in c → booléen, opérateur **in** de test de présence (not **in** d'absence)

enumerate(c) → itérateur sur (index, valeur)

Spécifique aux conteneurs de séquences (listes, tuples, chaînes) :

reversed(c) → itérateur inversé **c*5** → duplication **c+c2** → concaténation

c.index(val) → position **c.count(val)** → nb d'occurrences

modification de la liste originale

lst.append(item)

ajout d'un élément à la fin

lst.extend(seq)

ajout d'une séquence d'éléments à la fin

lst.insert(idx, val)

insertion d'un élément à une position

lst.remove(val)

suppression d'un élément à partir de sa valeur

lst.pop(idx)

suppression de l'élément à une position et retour de la valeur

lst.sort() **lst.reverse()**

tri / inversion de la liste sur place

Opérations sur dictionnaires

d[clé]=valeur **d.clear()**

d[clé]→valeur **del d[clé]**

d.update(d2) mise à jour/ajout

d.keys() des couples

d.values() vues sur les clés,

d.items() valeurs, couples

d.pop(clé)

Opérations sur ensembles

Opérateurs:

| → union (caractère barre verticale)

& → intersection

- ^ → différence/diff symétrique

< <= > >= → relations d'inclusion

s.update(s2)

s.add(clé) **s.remove(clé)**

s.discard(clé)

stockage de données sur disque, et relecture

Fichiers

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

↑ variable ↑ nom du fichier ↑ mode d'ouverture ↑ encodage des

fichier pour sur le disque □ 'r' lecture (read) caractères pour les

les opérations (+chemin...) □ 'w' écriture (write) fichiers textes:

cf fonctions des modules **os** et **os.path** □ 'a' ajout (append)... utf8 ascii latin1 ...

en écriture

f.write("coucou")

↑ chaîne vide si fin de fichier

↑ en lecture

↑ si nb de caractères

↑ pas précisé, lit tout

↑ le fichier

f.close() ne pas oublier de fermer le fichier après son utilisation !

Fermeture automatique Pythonnesque : **with open(...) as f:**

très courant : boucle itérative de lecture des lignes d'un fichier texte :

for ligne in f:

→ bloc de traitement de la ligne

très utilisé pour les boucles itératives **for**

Génération de séquences d'entiers

par défaut 0 non compris

range([début,] fin [,pas])

range(5) → 0 1 2 3 4

range(3, 8) → 3 4 5 6 7

range(2, 12, 3) → 2 5 8 11

range retourne un « générateur », faire une conversion

en liste pour voir les valeurs, par exemple:

print(list(range(4)))

nom de la fonction (identificateur)

Définition de fonction

paramètres nommés

def nomfct(p_x, p_y, p_z):

"""documentation"""

→ # bloc instructions, calcul de res, etc.

return res ← valeur résultat de l'appel.

si pas de résultat calculé à

retourner : **return None**

que dans le bloc et pendant l'appel à la fonction (« boîte noire »)

r = nomfct(3, i+2, 2*i) Appel de fonction

↑ un argument par paramètre

↑ récupération du résultat retourné (si nécessaire)

directives de formatage

Formatage de chaînes

valeurs à formater

"modele{ } { } { }".format(x, y, r) → **str**

"{sélection:formatage!conversion}"

□ Sélection : **"{:+2.3f}".format(45.7273)**

2 → **"+45.727"**

x **"{1:>10s}".format(8, "toto")**

0.nom → **"toto"**

4[clé] **"{!r}".format("L'ame")**

0[2] → **"L'ame"**

□ Formatage : **car-rempl alignement signe larg.mini-précision-larg.max type**

<>^ = + - espace 0 au début pour remplissage avec des 0

entiers : **b** binaire, **c** caractère, **d** décimal (défaut), **o** octal, **x** ou **X** hexa...

flottant : **e** ou **E** exponentielle, **f** ou **F** point fixe, **g** ou **G** approprié (défaut)

pourcentage

chaîne : **s** ...

□ Conversion : **s** (texte lisible) ou **r** (représentation littérale)

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()`
 Boucle sur lignes : `for line in f : ...`
 Contexte géré (close) : `with open(...) as 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 `__init__.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)
issubclass(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
def __repr__(self) :
    # retourne chaîne de représentation
def __bytes__(self) :
    # retourne objet chaîne d'octets
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) :
int(x) → def __int__(self) :
float(x) → def __float__(self) :
round(x, n) → def __round__(self, 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 __getattr__(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) :
issubclass(sousclasse, cls)
→ def __subclasscheck__(cls, sousclasse) :
```

Générateurs

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

Fonction générateur, contient une instruction

```
yield yield expression
yield from séquence
variable = (yield expression) transmission de
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()
```


Symbolique de l'Abrégé	
♣ instructions optionnelles, ♢ instruction répétables, ♠ valeur immuable (non modifiable), → conteneur ordonné (↔ non ordonné), constante, variable, type, fonction & méthode, paramètre, [.paramètre optionnel], mot_clé, littéral, 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" 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 incréméntation : nom+=expression affectation augmentée : nom+=expression (avec les autres opérateurs aussi) suppression : del nom	
Conventions Identificateurs	
Détails dans PEP 8 "Style Guide for Python" UNE_CONSTANTE majuscules une_var_locale 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 not a and b a or b (expr) combinables : 12<x<34	
Maths	
-a +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) lx →abs(x) x%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([deb],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 → nouvelle ligne \\t → tabulation \\N(nom) → unicode nom \\xhh → hh hexa \\ooo → oo octal \\uhhhh et \\Uhhhhhhhh → 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: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) .splitlines([keepend]) .partition(sep) .replace(old,new[,n]) .find(s[,deb[,fin]]) .count(s[,deb[,fin]]) .index(s[,deb[,fin]]) .isdigit() & Co .lower() .upper() .strip([chars]) .startswith(s[,deb[,fin]]) .endswith(s[,start[,end]]) .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 Δ détecter des exceptions précises (ex. ValueError) et non génériques (ex. Exception). Levée D'exception (situation d'erreur) raise exc([args]) raise → Δ propager l'exception Quelques classes d'exceptions : Exception - ArithmeticError - ZeroDivisionError -	

IndexError - KeyError - AttributeError - IOError - ImportError - NameError - SyntaxError - TypeError - NotImplementedError...	
Contexte Géré	
with garde() as v : # Bloc exécuté dans un contexte géré	
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) Fonctions Anonymes lambda x,y: expression Séquences & Indexation pour tout conteneur ordonné à accès direct. ic Élément : x[i] Tranche (slice) : x[deb:fin] x[deb:fin:pas] i, deb, fin, pas entiers positifs ou négatifs deb/fin manquant → jusqu'au bout x[i] x x[deb:fin] Modification (si séquence modifiable) x[i]=expression x[deb:fin]=itérable del x[i] del x[deb:fin] Conteneurs & Itérables Un itérable fournit les valeurs l'une après l'autre. Ex : conteneurs, vues sur dictionnaires, objets itérables, fonctions générateurs... Générateurs (calcul des valeurs lorsque nécessaire) range([deb[,fin[,pas]]) (expr for var in iter if cond &) Opérations Génériques v in conteneur v not in conteneur len(conteneur) enumerate(iter[,deb]) iter(of_sen) all(iter) any(iter) filter(fct,iter) map(fct,iter,...) max(iter) min(iter) sum(iter[,deb]) reversed(seq) sorted(iter[,k[,rev]]) Sur séquences : .count(x) .index(x[,i[,j]]) Chaîne : (séquence de caractères) cf. types bytes, bytearray, memoryview pour manipuler des octets (+notation b"octets"). Liste → 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, Ensemble : set {1, 'toto', 42} set(iterable) : frozenset(iterable) .add(x) .remove(x) .discard(x) .copy() .clear() .pop() U → J, O → G, diff → ^, C... → <... J = G = ^ = ... 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 "vues" itérables Entrées/Sorties & Fichiers print("x=", x[,y...][,sep=...][,end=...][,file=...]) input("Age ? ") → str transtypage explicite en int ou float si besoin.	

Fonctions prédéfinies

- ▷ `abs(x)` : valeur absolue de `x`
- ▷ `int(x)` : valeur `x` convertie en entier
- ▷ `float(x)` : valeur `x` convertie en réel
- ▷ `str(x)` : valeur `x` (int ou float), convertie en str
- ▷ `list(x)` : valeur `x` convertie en liste
- ▷ `tuple(x)` : valeur `x` convertie en tuple
- ▷ `dict(x)` : séquence de couples `x` convertie en dictionnaire
- ▷ `set(x)` : `x` converti en ensemble
- ▷ `help(x)` : aide sur `x`
- ▷ `dir(x)` : liste des attributs de `x`
- ▷ `type(x)` : type de `x`
- ▷ `print(...)` : imprime
- ▷ `input(x)` : imprime le string `x` et lit le string qui est introduit
- ▷ `round(x [,ndigits])` : valeur arrondie du float `x` à `ndigits` chiffres (par défaut 0)
- ▷ `range([start], stop, [step])` : retourne une suite d'entiers
- ▷ `sorted(s)` : retourne une liste avec les éléments de `s` triés

Gather, scatter et keyword arguments

- ▷ `def fun(*args)` : *gather* des arguments en un tuple `args`
- ▷ `fun(*s)` : *scatter* de la séquence `s` lors de l'appel

Opérations et méthodes sur les séquences (str, list, tuples)

- ▷ `len(s)` : longueur de la séquence `s`
- ▷ `min(s), max(s)` : élément minimum, maximum
- ▷ `sum(s)` : (ne fonctionne pas pour les string) : somme de tous les éléments (valeur numérique)
- ▷ `s.index(value, [start, [stop]])` : premier indice de valeur dans `s[start:stop]`
- ▷ `s.count(sub [,start [,end]])` : nombre d'occurrences sans chevauchement de `sub` dans `s[start:end]`
- ▷ `enumerate(s)` : construit une séquence de couples dont le *i*ème élément (à partir de 0) vaut le couple (*i*, `s[i]`)
- ▷ `zip(a,b), zip(a,b,c), ...` : construit une séquence de couples, resp. triples, ..., dont le *i*ème élément reprend le *i*ème élément de chaque séquence `a`, `b`, `c`

Méthodes sur les chaînes de caractères (str)

- ▷ `s.lower(), s.upper()` : string avec caractères en minuscules respectivement en majuscules
- ▷ `s.islower(), s.isdigit(), s.isalnum(), s.isalpha(), s.isupper()` : vrai si `s` n'est pas vide et n'a (respectivement) que des minuscules, des chiffres, des car. alphanumériques, alphabétiques, majuscules
- ▷ `s.find(sub [,start [,end]])` : premier indice de `s` où le sous string `sub` est trouvé dans `s[start:end]`
- ▷ `s.replace(old, new[, co])` : copie de `s` en remplaçant toutes les (ou les `co` premières) occurrences de `old` par `new`.
- ▷ `s.format(...)` : copie de `s` après formatage
- ▷ `s.capitalize()` : copie de `s` avec première lettre en majuscule
- ▷ `s.strip()` : copie de `s` en retirant les blancs en début et fin
- ▷ `s.join(t)` : crée un str qui est le résultat de la concaténation des éléments de la séquence de str `t` chacun séparé par le str `s`
- ▷ `s.split([sep [,maxsplit]])` : renvoie une liste d'éléments séparés dans `s` par le caractère `sep` (par défaut blanc); au maximum `maxsplit` séparations sont faites (par défaut l'infini)

Opérateurs et méthodes sur les listes

- ▷ `s.append(v)` : ajoute un élément valant `v` à la fin de la liste
- ▷ `s.extend(s2)` : rajoute à `s` tous les éléments de la liste `s2`
- ▷ `s.insert(i,v)` : insère l'objet `v` à l'indice `i`
- ▷ `s.pop([i])` : supprime l'élément d'indice `i` de la liste (par défaut le dernier) et retourne la valeur de l'élément supprimé
- ▷ `s.remove(v)` : supprime la première valeur `v` dans `s`
- ▷ `s.reverse()` : renverse l'ordre des éléments de la liste, le premier et le dernier élément échangent leurs places, ...
- ▷ `s.sort(key=None, reverse=False)` : trie `s` en place
- ▷ `s.copy()` : *shallow* copie superficielle de `s`
- ▷ `del s[i], del s[i:j]` : supprime un ou des éléments de `s`

Méthodes sur les dictionnaires (dict)

- ▷ `d.clear()` : supprime tous les éléments de `d`
- ▷ `d.copy()` : *shallow* copie de `d`
- ▷ `{}.fromkeys(s,v)` : crée un dict avec les clés de `s` et la valeur `v`
- ▷ `d.get(k [,v])` : renvoie la valeur `d[k]` si elle existe `v` sinon
- ▷ `d.items()` : liste des items (*k,v*) de `d`
- ▷ `d.keys()` : liste des clés
- ▷ `d.pop(k [,v])` : enlève `d[k]` s'il existe et renvoie sa valeur ou `v` sinon
- ▷ `d.popitem()` : supprime un item arbitraire (*k,v*) et retourne l'item sous forme de tuple
- ▷ `d.setdefault(k [,v])` : `d[k]` si elle existe sinon `v` et rajoute `d[k]=v`
- ▷ `d.update(s)` : `s` est une liste de tuples que l'on rajoute à `d`
- ▷ `d.values()` : liste des valeurs de `d`
- ▷ `del d[k]` : supprime l'élément de clé `k` de `d`

Méthodes sur les ensembles (set)

- ▷ `s = set(v)` : initialise `s` : un set contenant les valeurs de `v`
- ▷ `s.add(v)` : ajoute l'élément `v` au set `s` (ne fait rien s'il y est déjà)
- ▷ `s.clear()` et `s.copy()` : idem dictionnaires
- ▷ `s.remove(v)` : supprime l'élément `v` du set (erreur si `v` n'est pas présent dans `s`)
- ▷ `s.discard(v)` : si `v` existe dans `s`, le supprime
- ▷ `s.pop()` : supprime et renvoie un élément arbitraire de `s`

Modules

- ▷ `math` : accès aux constantes et fonctions mathématiques (`pi`, `sin()`, `sqrt(x)`, `exp(x)`, `floor(x)` (valeur plancher), `ceil(x)` (valeur plafond), ...) : exemple : `math.ceil(x)`
- ▷ `copy` : `copy(s)`, `deepcopy(s)` : *shallow* et *deepcopy* de `s`

Méthodes sur les fichiers

- ▷ `f = open('fichier')` : ouvre 'fichier' en lecture (autre paramètres possibles : 'w'(en écriture), 'a'(en écriture avec ajout), `encoding='utf-8'` : encodage UTF-8)
- ▷ `with open('fichier') as f` : ouvre 'fichier' pour traitement à l'intérieur du `with`
- ▷ `for ligne in open('fichier')` : ouvre et traite chaque ligne de 'fichier' et le ferme à la fin du `for`
- ▷ `f.read()` : retourne le contenu du fichier texte `f`
- ▷ `f.readline()` : lit une ligne
- ▷ `f.readlines()` : renvoie la liste des lignes de `f`
- ▷ `f.write(s)` : écrit la chaîne de caractères `s` dans le fichier `f`
- ▷ `f.close()` : ferme `f`



Traduction d'un problème en programme

1. Analysez le problème

- ▷ Identifiez clairement ce que sont les **données fournies**, ainsi que les **résultats et types** attendus à l'issue du traitement.
- ▷ Formalisez une **démarche générale de résolution** par une séquence d'opérations simples.
- ▷ Vérifiez que vous **envisagez tous les cas** de figures (en particuliers les cas "limites").

2. Découpez votre problème en fonctions

- ▷ Chaque fonction doit réaliser **une tâche** clairement identifiée.
- ▷ Limitez les fonctions à **25 lignes** maximum, sauf dans des cas exceptionnels.
- ▷ Éviter la **redondance** dans le code (copier/coller). Si cela arrive, c'est qu'il manque soit une fonction, soit une boucle, soit que des tests conditionnels peuvent être regroupés.
- ▷ N'utilisez **pas de variables globales**.
- ▷ Veillez à ce que tous les paramètres et variables d'une fonction soient **utilisés** dans cette fonction.
- ▷ Pour une fonction qui renvoie un résultat, organisez le code pour qu'il ne contienne qu'un **seul return**, placé comme dernière instruction de la fonction. Pour une fonction qui ne renvoie pas de résultat, ne mettez pas de **return** (il y en aura un implicitement à la fin de l'exécution de la fonction).
- ▷ Ne modifiez pas les **paramètres**. □ Exemple: Si vous recevez une borne inférieure `first` et une supérieure `last` et que vous devez itérer de la première à la dernière, n'incrémentez pas `first` dans la boucle, car la signification n'en serait plus claire; créez plutôt une variable locale `pos` initialisée à `first`.
- ▷ Sauf si la fonction a comme but de modifier la (structure de) données reçue en paramètre; dans ce cas la fonction ne renvoie pas de valeur.

3. Testez le code au fur et à mesure du développement

- ▷ Créez des **scénarios de test**, pour lesquels vous choisissez les données fournies et vous vérifiez que le résultat de la fonction est conforme à ce que vous attendez.
- ▷ Vérifiez les cas particuliers et les **conditions aux limites**. □ Exemples: Pour le calcul d'une racine carrée, que se passe-t-il lorsque le paramètre est un nombre négatif ?

Programmation

1. Style de programmation

- ▷ N'utilisez pas les instructions `break` ou `continue`
- ▷ Utilisez la forme raccourcie `if(is_leap_year(2008))` plutôt que la forme équivalente `if(is_leap_year(2008)==true)`
- ▷ Utilisez la forme `return <expression booléenne>` plutôt que la forme équivalente


```
if <expression booléenne>:
    res = true
else:
    res = false
return res
```
- ▷ N'exécutez pas plusieurs fois une fonction alors qu'une exécution suffit en retenant le résultat.
- ▷ Précisez le domaine de validité des paramètres.

Programmation (suite)

2. Quelques erreurs classiques

- ▷ Vous essayez d'utiliser une variable avant de l'avoir **initialisée**.
- ▷ L'**alignement des blocs** de code n'est pas respecté.
- ▷ Vous oubliez de fermer un fichier que vous avez ouvert.

Nommage de variables, fonctions, etc.

1. Utilisez une convention de nommage

`joined_lower` pour les **variables** (attributs),
et **fonctions** (méthodes)

`ALL_CAPS` pour les **constantes**

2. Choisissez bien les noms

- ▷ Donner des noms de variables qui expriment leur contenu, des noms de fonctions qui expriment ce qu'elles font (cf. règles de nommage ci-dessus).
- ▷ Évitez les noms trop proches les uns des autres.
- ▷ Utilisez aussi systématiquement que possible les mêmes genres de noms de variables.
 - Exemples: `i, j, k` pour des indices, `x, y, z` pour les coordonnées, `max_length` pour une variable, `is_even()` pour une fonction, etc.

Style et documentation du code

1. Soignez la clarté de votre code

... c'est la première source de documentation.

- ▷ Utilisez les **docstrings** dans chaque fonction pour :
 - brièvement décrire ce que fait la fonction, **pas comment elle le fait**, et préciser ses entrées et sorties.
 - Décrire les arguments des fonctions.
- ▷ Soignez les indentations (2 à 4 espaces chacune) et la gestion des espaces et des lignes blanches (deux lignes blanches avant et entre chacune des définitions de fonction globales; une ligne blanche pour mettre en évidence une nouvelle partie dans une fonction),
- ▷ Il faut commenter le code à bon escient et avec parcimonie. Évitez d'indiquer le **fonctionnement** du code dans les commentaires.
 - Exemples: Avant l'instruction `"for car in line:"`, ne pas indiquer qu'on va boucler sur tous les caractères de la ligne...
- ▷ Évitez de paraphraser le code. N'utilisez les commentaires que lorsque la fonction d'un bout de code est difficile à comprendre.

Structure d'un programme Python

1. Voir verso





Structure d'un programme Python

docstring initial

```

1  """
2  Petit jeu de devinette (version 2)
3  Auteur: Thierry Massart
4  Date : 10 octobre 2018
5  Petit jeu de devinette d'un nombre entier tiré aléatoirement
6  par le programme dans un interval donné
7  Entrée : le nombre proposé par l'utilisateur
8  Résultat : affiche si le nombre proposé est celui tiré
9             aléatoirement
10 """

```

importation
des modules

```

11
12 import random # module le tirage des nombres aléatoires

```

définition des
constantes
globales

```

13
14 VALEUR_MIN = 0 # borne inférieure de l'intervalle
15 VALEUR_MAX = 5 # borne supérieure de l'intervalle

```

Définitions de
fonctions

```

16
17 def entree_utilisateur(borne_min, borne_max):

```

```

18     """
19     Lecture du nombre entier choisit par l'utilisateur
20     dans l'intervalle [borne_min, borne_max]
21     Entrées : bornes de l'intervalle
22     Résultat : choix de l'utilisateur
23     """

```

```

24
25     message = "Votre choix de valeur entre {0} et {1} : "
26     ok = False # drapeau : vrai quand le choix donné est valable
27     while not ok: # tant que le choix n'est pas bon
28         choix = int(input(message.format(borne_min, borne_max)))
29         ok = (borne_min <= choix and choix <= borne_max)
30         if not ok: # entrée hors de l'intervalle
31             print("Hors de l'intervalle ! Donnez une valeur valide")
32     return choix

```

```

33
34 def tirage(borne_min, borne_max):

```

```

35     """
36     Tirage aléatoire d'un entier dans [borne_min, borne_max]
37     """
38     return random.randint(borne_min, borne_max)

```

```

39
40 def affichage_resultat(secret, choix_utilisateur):

```

```

41     """
42     Affiche le résultat
43     """
44     if secret == choix_utilisateur:
45         print("gagné !")
46     else:
47         print("perdu ! La valeur était", secret)

```

Entête

docstring
de
la fonction

Code principal

```

51
52 mon_secret = tirage(VALEUR_MIN, VALEUR_MAX)
53 choix_util = entree_utilisateur(VALEUR_MIN, VALEUR_MAX)
54 affichage_resultat(mon_secret, choix_util)
55

```

