Codage de l'information

Laurent Noé-Léopold Weinberg
paternité : Francesco De Comité

Licence 1 Mathématiques Informatique Semestre 1
Université de Lille
Faculté des Sciences et Technologies

27 novembre 2025

Python et Codage

Un peu de Python

Les opérations bits à bits et de décalages sont disponibles en Python. Il s'agit d'une version généralisée aux entiers relatifs sans limitation.

Tableau de correspondance

nom	notation mathématique	en python
ou bit à bit	$a \vee b$	`a	b`
et bit à bit	$a \wedge b$	`a & b`
ou exclusif bit à bit	$a \oplus b$	`a ^ b`
la négation bit à bit	$\neg a$	`~ a`
décalage itéré gauche	$a \ll n$	`a << n`
décalage itéré droite	$a \gg n$	`a >> n`

Le type bytes

Il s'agit d'un type *iterable* pour représenter les suites d'octets.
les éléments de types `bytes` possèdent une longueur, peuvent être indicés.
Les éléments qui constituent un bytes sont des octets (entiers compris entre 0 et 255.)
Les valeurs litérales des bytes sont représentées comme une chaine de caractères précédées d'un b.
Comme les chaînes, les bytes ne sont pas mutables.
La fonction `bytes` permet de convertir une chaine en bytes à condition de préciser l'encodage en second paramètre. elle permet aussi de convertir un itérable dont les éléments sont des entiers compris entre 0 et 255 au sens large en bytes...

Le type bytes : exemples

exemples

```
>>> b=b"abcd"
>>> len(b)
4
>>> type(b)
<class 'bytes'>
>>> b[0]
97
>>> b[1]
98
>>> b[3]
100
>>> b[4]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
IndexError: index out of range
```

La forme with

Il existe une instruction python qui permet de garantir la fermeture automatique des fichiers
il s'agit de la forme `with`
la syntaxe est :

```
with open(nom,mode,) as identificateur_de_canal:
    bloc_d_instructions
#ici le fichier a été fermé par python
```

Les opérations de lecture ou d'écriture dans le fichier se font à l'intérieur du bloc d'instructions

littéraux

Bases prédéfinies

En python, dans le code sources des programmes, en plus du décimal usuel, on peut écrire les valeurs entières respectivement en base 2, 8 ou 16. La syntaxe est très simple :

- on écrit un 0 car tous les littéraux entiers en Python doivent commencer par un chiffre,
- puis on écrit respectivement, la lettre `b`, `o` ou `x` ,
- enfin on écrit la liste des chiffres du nombre dans la base respective.

remarque

Les chiffres de 10 à 15 en hexadécimal peuvent être écrit indifféremment en haut de casse ou en bas de casse.

Exemple

```
>>> 0xFF
255
```

```
>>> 0o77
63
```

```
>>> 0b111
7
```

fonctions de conversion (début...)

la fonction prédéfinie `bin`

La fonction `bin` permet d'obtenir la chaine de caractère qui est la représentation binaire de l'entier passé en paramètre.

`bin(x: int) -> str`

Le résultat a la forme d'un littéral binaire en Python

exemple

```
>>> bin(42)
'0b101010'
>>> bin(255)
'0b11111111'
```

fonctions de conversion (...suite...)

la fonction prédéfinie `oct`

La fonction `oct` permet d'obtenir la chaine de caractère qui est la représentation octale de l'entier passé en paramètre.

`oct(x: int) -> str`

Le résultat a la forme d'un littéral octal en Python

exemple

```
>>> oct(42)
'0o52'
>>> oct(255)
'0o377'
```

fonctions de conversion (...et fin)

la fonction prédéfinie `hex`

La fonction `hex` permet d'obtenir la chaine de caractère qui est la représentation hexadécimale de l'entier passé en paramètre.

`hex(x: int) -> str`

le résultat a la forme d'un littéral hexadécimal en Python

exemple

```
>>> hex(42)
'0x2a'
>>> hex(255)
'0xff'
```

unicodedata

Le module `unicodedata` permet de consulter la base de donnée des caractères unicodes.
Voici une sélection de quelques fonctions de ce module.

pour aller plus loin

Les étudiants interessés peuvent lire la documentation de ce module ici : RTFM

unicodedata.name(chr:str [, default:str])->str

Renvoie le nom assigné au caractère chr comme une chaîne de caractères. Si aucun nom n'est défini, default est renvoyé, ou, si ce dernier n'est pas renseigné l'exception ValueError est déclenchée..

exemple

```
>>> unicodedata.name('a')
'LATIN SMALL LETTER A'
>>> unicodedata.name('1')
'DIGIT ONE'
>>> unicodedata.name("@")
'COMMERCIAL AT'
>>> unicodedata.name(chr(0x10F000))
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: no such name
>>> unicodedata.name(chr(0x10F000),'sans nom')
'sans nom'
```

lookup

unicodedata.lookup(name :str)->str

Renvoie le caractère de nom `name` donné. S'il n'y a pas de caractères portant ce nom, une exception `KeyError` est déclenchée.

exemple

```
>>> import unicodedata
>>> unicodedata.lookup("LATIN CAPITAL LETTER A")
'A'
>>> unicodedata.lookup("LATIN SMALL LETTER A")
'a'
>>> unicodedata.lookup("BIDULE")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: "undefined character name 'BIDULE'"
>>> unicodedata.lookup("LATIN SMALL LETTER A WITH TILDE")
'ã'
```

category

unicodedata.category(chr: str) -> str

Renvoie la catégorie générale assignée au caractère `chr` comme une chaîne de caractères.

exemple

```
>>> unicodedata.category('a')
'Ll'
>>> unicodedata.category('A')
'Lu'
>>> unicodedata.category('1')
'Nd'
>>> unicodedata.category('@')
'Po'
```

numeric

`unicodedata.numeric(chr :string [,default: float])-> float`

Renvoie la valeur numérique assignée au caractère `chr` comme un flottant. Si aucune valeur de ce type n'est définie, `default` est renvoyé, ou, si ce dernier n'est pas renseigné, l'exception `ValueError` est déclenchée.

exemple

```
>>> unicodedata.numeric(chr(0x216F))
1000.0
>>> unicodedata.name(chr(0x216F))
'ROMAN NUMERAL ONE THOUSAND'
```

introduction

cette semaine en ODI

si vous êtes allés jusqu'au bout des travaux, vous avez écrit vos premier scripts `shell`.

et Python dans tous ça ?

Python permet aussi d'écrire de nouvelles commande à utiliser dans le terminal.

quelques pas au dehors de `thonny-l1-test`...

Il suffit de peu de chose pour transformer un script python en programme utilisable en ligne de commande.

Introduction (suite)

le plan

- le `shebang`
- les droits d'exécution
- la variable `path`
- les paramètres de la ligne de commande

shebang

définition

On appelle *shebang* le commentaire suivi d'un point d'exclamation qui permet de préciser au terminal quel interprète de commande il doit utiliser pour exécuter un script. *shebang* est un acronyme anglais formé à partir de *shell* qui est utilisé pour désigner ce qui est l'interface entre l'intérieur (le système) l'exterieur (l'utilisateur) et *bang* qui désigne le point d'exclamation.

Pour python3

```
#!/usr/bin/env python3
```

droit d'exécution

définition

le droit `x` pour les fichiers signifie que le fichier peut être éxécuter par le système. Comme pour les autres droits, il se décline en droit pour l'utilisateur, pour les membres du groupe et pour le reste du monde.

la commande `chmod`

la commande `chmod` permet de changer les droits d'accés à un fichier ou à un dossier.
On l'utilise en précisant comme paramètres les nouveaux droits et le nom du fichier.
Il y a plusieurs manières de préciser les droits.

droit d'exécution

exemple de droits

```
mi@mbarak:~/exemples$ ls -l
total 0
-rw-rw-rw- 1 mi traval 0 nov.  29 01:03 tata
-rw--w---- 1 mi traval 0 nov.  29 01:03 titi
-rwx------ 1 mi traval 0 nov.  29 01:03 toto
```

- tata peut être lu et écrit par tout le monde. En octal ce mode est 666,
- titi peut être lu et écrit par l'utilisateur `mi` les membres du groupe `traval` peuvent écrire le fichier et c'est tout. En octal ce mode est 620,
- seul l'utilisateur `mi` peut lire, écrire et exécuter le fichier `toto`. En octal ce mode est 700.

la commande chmod

exemple de modification de droits

une manière d'utiliser le chmod est de préciser les droits sous forme d'un nombre octal.

```
mi@mbarak:~/exemples$ chmod 755 toto
mi@mbarak:~/exemples$ ls -l
total 0
-rw-rw-rw- 1 mi traval 0 nov.  29 01:03 tata
-rw--w---- 1 mi traval 0 nov.  29 01:03 titi
-rwxr-xr-x 1 mi traval 0 nov.  29 01:03 toto
```

maintenant tout le monde peut lire et exécuter le fichier toto.

variables d'environnement

Qu'est ce que c'est ?

L'interprète permet de définir des variables de commandes, qui peuvent avoir des effets sur son fonctionnement et sur le fonctionnement des autres programme.

la variable `path`

cette variable sert à préciser la liste des dossiers dans lesquels le système va chercher les exécutables. Il s'agit d'une liste de chemin. Cela permet de ne pas mettre le chemin des commandes.

récuperer les paramètres de la ligne de commande

sys.argv

Le module `sys` définit variable `argv` qui est une liste de chaînes de caractères et qui contient les paramètres de la ligne de commande.

exemple

```
#!/usr/bin/env python3
import sys

print("le nombre de parmètres est :", len(sys.argv))

for i in range(len(sys.argv)):
    print("le paramètre numéro",i,"est",sys.argv[i])
```

```
$ ./ldc.py toto titi tata
le nombre de parmètres est : 4
le paramètre numéro 0 est ./ldc.py
le paramètre numéro 1 est toto
le paramètre numéro 2 est titi
le paramètre numéro 3 est tata
```

deux études de cas

On va fabriquer deux outils classiques.

- un visualisateur hexadécimal
- un utilitaire qui assure le codage et le décodage en base64

le visualisateur hexadécimal

qu'est ce que c'est ?

Il s'agit d'un outil qui permet d'afficher le contenu d'un fichier. l'affichage produit contient trois zones :

- la zone de gauche qui contient le décalage ou encore l'adresse dans le fichier (on parle aussi d'offset). Il s'agit d'un nombre en hexadécimal qui correspond au numéro du premier octet dans cette ligne.
- une zone centrale qui contient une liste de seize octets exprimés en hexadécimal et séparés par des espaces
- enfin une zone de droite qui contient une visualisation des octets en tant que caractère ASCII. Biensur seul les caractères imprimables qui font partie du code ASCII sont affichés : les autres sont remplacé par des points.

Sur la dernière ligne la longueur du fichier est affichée.

hexdump -C

Voici un exemple de la commande `hexdump` on utilise l'option `-C` qui permet l'affichage classique.

```
mi@mbarak:~/codage/cours$ hexdump -C cours6.pdf | head -n 5 ; \
                          echo ... ; \
                          hexdump -C cours6.pdf |tail -n 5
00000000  25 50 44 46 2d 31 2e 35  0a 25 e4 f0 ed f8 0a 36  |%PDF-1.5.%.....6|
00000010  20 30 20 6f 62 6a 0a 3c  3c 2f 54 79 70 65 2f 58  | 0 obj.<</Type/X|
00000020  4f 62 6a 65 63 74 2f 53  75 62 74 79 70 65 2f 46  |Object/Subtype/F|
00000030  6f 72 6d 2f 46 6f 72 6d  54 79 70 65 20 31 2f 42  |orm/FormType 1/B|
00000040  42 6f 78 5b 30 20 30 20  37 32 35 2e 36 36 39 20  |Box[0 0 725.669 |
  ...
00040180  63 ff fa 96 9d 23 ee b9  dd 89 fe 0f 46 7c 8d e4  |c....#......F|..|
00040190  0a 65 6e 64 73 74 72 65  61 6d 0a 65 6e 64 6f 62  |.endstream.endob|
000401a0  6a 0a 73 74 61 72 74 78  72 65 66 0a 32 36 30 33  |j.startxref.2603|
000401b0  33 35 0a 25 25 45 4f 46  0a                       |35.%%EOF.|
000401b9
```

cahier des charges

On va procéder à une réimplantation du visualisateur.

on s'autorise quelques variations :

- pas d'espace entre les représentation hexadécimale séparée par des espaces des 8 premiers et des 8 derniers octets d'une ligne octet
- pas de barre verticale pour encadrer les représentations textuelles des octets d'une ligne

avant de commencer

Remarques générales

On a essayé de tenir compte des consignes du cours de programmation

- toutes les fonctions sont
  - typées,
  - documentées,
  - et testées.
- les constantes sont en majuscules
- le choix du nom d'une fonction ou d'une variable a du sens,
- les identificateurs sont en bas de casse et on utilise le caractère blanc_souligné pour séparer les composants d'un identificateur
- on essaie de réutilser et de factoriser le code.

organisation du code

Le code est disponible sur `gitlab`. Un lien sera mis sur le semainier du portail

pour hexdump il y a deux fichiers.
L'un est un module nommé `outil_mi.py` qui contient plusieurs fonction qu'on réutilise aussi pour le second programme
l'autre est nommée `hexdump_mi.py` est exécutable.

examen du code

```
#!/usr/bin/env python3
# FIL - novembre 2023
# MI - codage
# Time-stamp: <2023-11-26 11:11:31 wl>
# -*- coding: utf_8 -*-

""" implantation d'hexdump -C """

import sys
import io #pour le type des canaux
from outils_mi import char_segment,\
                      joindre,\
                      allonge_eventuellement
_MODE_D_EMPLOI = '''
...
```

examen du code

```
...
def _main() -> None:
    """
␣␣␣␣procédure␣principale,
␣␣␣␣si␣un␣paramétre␣est␣précisé␣et␣que␣ce␣n'est␣pas
␣␣␣␣l'option␣pour␣obtenir,␣le␣fichier␣est␣affiché
␣␣␣␣"""
    if len(sys.argv)!=2 or sys.argv[1]=='-h':
        print(_MODE_D_EMPLOI)
    else:
        canal = open(sys.argv[1],'rb')
        dump(canal)
        canal.close()

if __name__ == '__main__':
    _main()
```

examen du code

```
...
def dump(canal: io.TextIOWrapper, size: int = 16) -> None:
    """
␣␣␣␣Effectue␣l'affichage␣d'un␣fichier␣préalablement␣ouvert
␣␣␣␣en␣lecture␣binaire␣passé␣grace␣au␣paramètre␣`canal`.

␣␣␣␣Le␣second␣paramètre␣permet␣de␣préciser␣le␣nombre␣d'octets
␣␣␣␣affichée␣par␣ligne

␣␣␣␣`dump`␣est␣une␣procédure,␣c'est-à-dire␣une␣fonction
␣␣␣␣qui␣produit␣un␣résultat␣connu␣à␣l'avance␣:␣c'est␣None
␣␣␣␣On␣n'appelle␣pas␣cette␣fonction␣pour␣son␣**résultat**
␣␣␣␣mais␣pour␣son␣**effet**.␣L'effet␣est␣ici␣un␣affichage.

␣␣␣␣Exemples:
␣␣␣␣>>>␣import␣io
␣␣␣␣>>>␣f␣=␣io.BytesIO(b"Timoleon Cunegonde \\x07\\x0d\\x0a")
␣␣␣␣>>>␣dump(f,4)
␣␣␣␣00000000␣54␣69␣6D␣6F␣␣Timo
␣␣␣␣00000004␣6C␣65␣6F␣6E␣␣leon
␣␣␣␣00000008␣20␣43␣75␣6E␣␣␣Cun
␣␣␣␣0000000C␣65␣67␣6F␣6E␣␣egon
␣␣␣␣00000010␣64␣65␣20␣07␣␣de␣.
␣␣␣␣00000014␣0D␣0A␣␣␣␣␣␣␣␣..
␣␣␣␣00000016
␣␣␣␣"""
...
```

examen du code

```
...
def dump(canal: io.TextIOWrapper, size: int = 16) -> None:
    # documentation masquée pour concision
    addr = 0
    suite_d_octets   = canal.read(size)
    while len(suite_d_octets)>0:
        print(repr_adresse(addr),
              allonge_eventuellement(image_hexa(suite_d_octets),3*size),
              image_ascii(suite_d_octets))
        addr = addr + len(suite_d_octets)
        suite_d_octets   = canal.read(size)
    print(repr_adresse(addr))
...
```

examen du code

```
...
def repr_adresse(adresse: int) -> str :
    """
␣␣␣␣Renvoie␣la␣chaîne␣formée␣par␣la␣représentation␣hexadécimale
␣␣␣␣du␣nombre␣`adresse`␣passé␣en␣paramètre

␣␣␣␣précondition␣:␣0<=adresse<2**32
␣␣␣␣Exemples:
␣␣␣␣>>>␣repr_adresse(0)
␣␣␣␣'00000000'
␣␣␣␣>>>␣repr_adresse(255)
␣␣␣␣'000000FF'
␣␣␣␣"""
    assert adresse in range(0,1<<32),'l\'adresse␣est␣trop␣longue'
    res = ''
    for _i in range(8):
        res = CHIFFRES_HEXA[adresse & 0b1111] + res
        adresse = adresse >> 4
    return res
...
```

examen du code

```
...
def allonge_eventuellement(chaine: str, taille: int) -> str:
    """
␣␣␣␣renvoie␣une␣chaîne␣dont␣la␣longueur␣est␣etendue␣à␣la
␣␣␣␣taille␣passée␣en␣paramètre␣en␣ajoutant␣des␣espaces
␣␣␣␣à␣droite␣de␣`chaine`.

␣␣␣␣précondition␣:␣aucune
␣␣␣␣Exemples␣:
␣␣␣␣>>>␣allonge_eventuellement('toto',8)
␣␣␣␣'toto␣␣␣␣'
␣␣␣␣>>>␣len(allonge_eventuellement('toto',8))
␣␣␣␣8
␣␣␣␣"""
    if len(chaine)< taille:
        res = chaine + (taille - len(chaine)) * '␣'
    else:
        res = chaine
    return res
...
```

examen du code

```
...
CHIFFRES_HEXA=char_segment('0','9')+char_segment('A','F')

def image_hexa(suite_d_octets: bytes) -> str:
    """
␣␣␣␣Renvoie␣une␣chaîne␣contenant␣les␣valeurs␣hexadécimales
␣␣␣␣du␣bytes␣`suite_d_octets`␣passé␣en␣paramètre.␣Les␣valeurs␣sont
␣␣␣␣séparées␣de␣par␣une␣espace.

␣␣␣␣précondition␣:␣aucune
␣␣␣␣Exemples␣:
␣␣␣␣>>>␣image_hexa(b'AaBbCc01')
␣␣␣␣'41␣61␣42␣62␣43␣63␣30␣31'
␣␣␣␣>>>␣image_hexa(b'\\x01\\x02\\x03')
␣␣␣␣'01␣02␣03'
␣␣␣␣>>>␣image_hexa(b'')
␣␣␣␣''
␣␣␣␣"""
    liste=[]
    for octet in suite_d_octets:
        cps=CHIFFRES_HEXA[octet>>4]
        # chiffre plus significatif (poid fort)
        cms=CHIFFRES_HEXA[octet & 0b1111]
        # chiffre moins significatif (poid faible)
        liste.append(cps+cms)
    return joindre(liste,'␣')
...
```

examen du code

```
...
def joindre(liste_chaine: list[str], colle: str) -> str:
    """
␣␣␣␣concatène␣une␣liste␣de␣chaînes␣`liste_chaine`␣en␣plaçant
␣␣␣␣la␣chaîne␣`colle`␣entre␣chaque␣élément.

␣␣␣␣précondition␣:␣aucune
␣␣␣␣Exemples␣:
␣␣␣␣>>>␣joindre(['un',␣'deux',␣'trois'],␣',␣')
␣␣␣␣'un,␣deux,␣trois'
␣␣␣␣>>>␣joindre([],'**')
␣␣␣␣''
␣␣␣␣>>>␣joindre(['un',␣'deux',␣'trois'],'***')
␣␣␣␣'un***deux***trois'
␣␣␣␣"""
    if len(liste_chaine)==0:
        res = ''
    else:
        res=liste_chaine[0]
        for i in range(1,len(liste_chaine)):
            res = res + colle + liste_chaine[i]
    return res
```

examen du code

```
...
def image_ascii(suite_d_octets: bytes) -> str :
    """
␣␣␣␣Renvoie␣une␣chaîne␣de␣caractères␣correspondant␣à␣l'interprétation
␣␣␣␣du␣bytes␣comme␣une␣chaîne␣codée␣en␣ASCII.
␣␣␣␣Les␣caractères␣non␣imprimables␣sont␣remplacés␣par␣des␣.

␣␣␣␣précondition␣:␣aucune

␣␣␣␣>>>␣image_ascii(b'AaBbCc01')
␣␣␣␣'AaBbCc01'
␣␣␣␣>>>␣image_ascii(b'\\x01\\x02\\x03')
␣␣␣␣'...'
␣␣␣␣>>>␣image_ascii(b'')
␣␣␣␣''
␣␣␣␣"""
    liste=[]
    for octet in suite_d_octets:
        liste.append(image_char(octet))
    return joindre(liste,'')
...
```

Le codage base64

qu'est ce que c'est ?

Beaucoup de protocoles de communication reposent sur des échanges de texte.
Mais comment faire si on veut transférer des fichiers binaires ?

une première idée

un codage basé l'hexadécimal

Une première solution pourrait être de traduire chaque octet en sa représentation sur deux chiffres en hexadécimal.

mais bof !

Cela existe ! mais cela possède un inconvénient. Chaque octet est représenté par deux chiffres hexadécimaux. Par conséquent après ce codage, la taille du fichier à transmettre a doublé.

Regardons de plus près

une constatation

On gaspille un peu car le code ascii contient 128 caractères et là on n'utilise que 16.

comment faire mieux ?

l'idée est d'utiliser davantage de caractères.
Mais on souhaite que cela reste une puissance de deux pour que l'encodage et le décodage reste facile.

On ne peut pas prendre les 128 caractères car certains sont des caractères de contrôle. Il y a au moins 33 caractères de contrôle cela laisse environ 95 caractères disponibles.

base64

la plus grande puissance de deux inférieure à 95 est 64.
les nombres de 0 à 63 s'ecrivent en binaire avec 6 bits.
le codage de trois octets nécessite 24 bits. les 24 bits sont découpés en 4 paquets de 6 bits.
Chaque paquet de 6 bits est codé par un caractère en ascii.
Il subsiste un problème tous les fichiers à encoder n'ont pas nécessairement une taille en octet multiple de 3 ?

Le jeu de caractères utilisés

Il est composé des 26 lettres capitales de l'alphabet latin, suivi des 26 lettres bas de casse de l'alphabet latin, suivi des 10 chiffres. On en est à 26+26+10=62. Deux caractères supplémentaires sont nécessaires : le signe plus et la barre oblique.

déc.	bin.	car.	déc.	bin.	car.	déc.	bin.	car.	déc.	bin.	car.
00	000000	A	16	010000	Q	32	100000	g	48	110000	w
01	000001	B	17	010001	R	33	100001	h	49	110001	x
02	000010	C	18	010010	S	34	100010	i	50	110010	y
03	000011	D	19	010011	T	35	100011	j	51	110011	z
04	000100	E	20	010100	U	36	100100	k	52	110100	0
05	000101	F	21	010101	V	37	100101	l	53	110101	1
06	000110	G	22	010110	W	38	100110	m	54	110110	2
07	000111	H	23	010111	X	39	100111	n	55	110111	3
08	001000	I	24	011000	Y	40	101000	o	56	111000	4
09	001001	J	25	011001	Z	41	101001	p	57	111001	5
10	001010	K	26	011010	a	42	101010	q	58	111010	6
11	001011	L	27	011011	b	43	101011	r	59	111011	7
12	001100	M	28	011100	c	44	101100	s	60	111100	8
13	001101	N	29	011101	d	45	101101	t	61	111101	9
14	001110	O	30	011110	e	46	101110	u	62	111110	+
15	001111	P	31	011111	f	47	101111	v	63	111111	/