



UNIVERSITÉ
CÔTE D'AZUR

Bases de l'informatique 1

Cours 4. Codages et représentations

Olivier Baldellon

Courriel : `prénom.nom@univ-cotedazur.fr`

Page professionnelle : <https://upinfo.univ-cotedazur.fr/~obaldellon/>

LICENCE I — FACULTÉ DES SCIENCES ET INGÉNIERIE DE NICE — UNIVERSITÉ CÔTE D'AZUR

 Partie I. Représentation

 Partie II. Couleurs et tuples

 Partie III. Compléments

 Partie IV. Représentation des caractères

 Partie V. Cryptologie

 Partie VI. Table des matières

- ▶ On a vu que les nombres sont représentés en binaire en informatique
 - On a vu des algorithmes de conversion en TD.
 - Ne marche que pour les nombres positifs.
- ▶ Comment représenter les nombres négatifs ?
 - Que signifie $x = -17$?
 - C'est l'unique nombre x vérifiant $x + 17 = 0$
 - Cool... mais en quoi cette information est importante ?

- ▶ La mémoire d'une machine étant limité
 - ▶ Les nombres sont toujours bornées
 - ▶ Sauf en Python (mais c'est pourquoi Python est lent)

- ▶ Travaillons sur 8 bits et calculons $255 + 2$

1	1	1	1	1	1	1	1
---	---	---	---	---	---	---	---

 : 255

0	0	0	0	0	0	1	0
---	---	---	---	---	---	---	---

 : 2

1

0	0	0	0	0	0	0	1
---	---	---	---	---	---	---	---

 : $257 = 1$

- ▶ Il y a dépassement et le résultat est absurde.
 - ▶ On travaille modulo 256 (en effet $257 \% 256 == 1$)
 - ▶ $255 + 2 = 1$
 - ▶ Mais en fait c'est génial!!! on vient de démontrer que $255 = -1$:)

- ▶ Soit on considère que les nombres sont positifs
 - ▶ Sur 8 bits on code les entiers de 0 à 255
 - ▶ Attention à ne pas dépasser
- ▶ Soit on autorise les négatifs
 - ▶ Sur 8 bits on codera les positifs de 0 à 127
 - ▶ les nombres commençant à 1 seront vu comme négatifs.
- ▶ Ajouter un négatif est équivalent à ajouter un nombre positif
 - ▶ même algo machine
 - ▶ même vitesse
 - ▶ En travaillant %256 toutes les règles de calculs restent valides

- ▶ Prenons un nombre entre 0 et 127 : 81 et calculons sa négation

0	1	0	1	0	0	0	1
---	---	---	---	---	---	---	---

 : 81

- ▶ Calculons le complément à deux
on remplace 1 par 0 et réciproquement

1	0	1	0	1	1	1	0
---	---	---	---	---	---	---	---

- ▶ Si on ajoute les deux on obtient

1	1	1	1	1	1	1	1
---	---	---	---	---	---	---	---

- ▶ Si on ajoute ensuite 1 on obtient $256=0$!

1

0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---

- ▶ -81 est obtenu en trois étapes :
 - ▶ On calcule 81 en binaire
 - ▶ On calcule le complément à deux
 - ▶ On ajoute 1

1	0	1	0	1	1	1	1
---	---	---	---	---	---	---	---

 Partie I. Représentation

 Partie II. Couleurs et tuples

 Partie III. Compléments

 Partie IV. Représentation des caractères

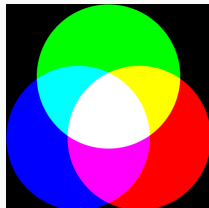
 Partie V. Cryptologie

 Partie VI. Table des matières

► En informatique, toutes les couleurs sont codées comme combinaison de trois couleurs : le rouge (R), le vert (G) et le bleu (B).

► Chaque couleur va de 0 à 255.

- (255, 0, 0) : rouge
- (0, 255, 0) : vert
- (0, 0, 255) : bleu



► On obtient les autres couleurs à partir des précédentes.

- (255, 255, 0) (R+G) donne le jaune
- (255, 0, 255) (R+B) donne le magenta
- (0, 255, 255) (B+G) donne le cyan

► Les trois couleurs ensembles donnent le blanc (255, 255, 255).

► L'absence de couleur donne le noir (0, 0, 0).

- ▶ En règle générale, on ne peut pas ajouter des couleurs.
 - ▶ Chaque composante doit être inférieure à 255 : ~~(355, 300, 455)~~
 - ▶ D'ailleurs le + ne fait pas l'addition !

```
>>> (255,100,200) + (100, 200, 255)
(255, 100, 200, 100, 200, 255)
```

SHELL

- ▶ Écrivons un programme qui fait la moyenne entre deux couleurs.

```
def mélange(c1,c2):
    (r1,g1,b1) = c1
    (r2,g2,b2) = c2
    r3=(r1+r2)//2
    g3=(g1+g2)//2
    b3=(b1+b2)//2
    return (r3,g3,b3)
```

SCRIPT

```
>>> rouge=(255,0,0)
>>> jaune=(255,255,0)
>>> orange=mélange(rouge,jaune)
>>> orange
(255, 127, 0)
```

SHELL

- ▶ Écrire un programme qui mélange deux couleurs selon une proportion p

- ▶ $\text{mélange_p}(0.75, c1, c2) = 0.75 \times c1 + 0.25 \times c2$

- ▶ $\text{mélange_p}(p, c1, c2) = p \times c1 + (1-p) \times c2$

- ▶ Remarque : on doit avoir :

$$\text{mélange_p}(0.5, c1, c2) == \text{mélange}(c1, c2)$$

- ▶ Le gris est un mélange entre blanc et noir.

- ▶ (10,10,10) gris très foncé
 - ▶ (200,200,200) gris très clair

- ▶ Pour avoir des couleurs plus claires ou plus foncées :

- ▶ Bleu foncé : mélange de bleu et de noir (0,0,100)
 - ▶ Jaune clair : mélange de jaune et blanc (255,255,100)
 - ▶ Marron : orange foncé

- ▶ On code les couleurs sur 8 bits.
- ▶ Un nombre sur 8 bits s'appelle un octet (entre 0 et 255).
- ▶ Un octet se représente facilement avec deux chiffres de la base 16.
 - ▶ 0 : "00"
 - ▶ 9 : "09"
 - ▶ 10 : "0A"
 - ▶ 15 : "0F"
 - ▶ 16 : "10"
 - ▶ 255 : "FF" ($15 \times 16 + 15$)
- ▶ Le système RGB est souvent représenté sous forme de chaîne :
 - ▶ "#FF0000" : Rouge
 - ▶ "#00FF00" : Vert
 - ▶ "#0000FF" : Bleu
 - ▶ "#FFFFFF" : Blanc
 - ▶ "#RRGGBB" : de manière générale : c'est la notation HTML

- 🍃 Partie I. Représentation
- 🍃 Partie II. Couleurs et tuples
- 🍃 Partie III. Compléments
- 🍃 Partie IV. Représentation des caractères
- 🍃 Partie V. Cryptologie
- 🍃 Partie VI. Table des matières

Python permet d'extraire une tranche (*slice*) d'une chaîne de caractères, repérée par ses positions extrêmes.

```
>>> texte = '123 nous allons au bois'
>>> texte[4]
'n'
>>> texte[12]
'o'
>>> texte[4:12] # de texte[4] à texte[11] !
'nous all'
>>> texte[4:]
'nous allons au bois'
>>> texte[:12]
'123 nous all'
```

SHELL

► `texte[a:b]` signifie la sous-chaîne d'indice $i \in [a, b - 1]$

- On peut même préciser un pas.

texte = 'abcABC123abcABCfff'

```
>>> texte[::3]
```

SHELL

```
'aA1aAf'
```

```
>>> texte[4:12:3]
```

```
'B2b'
```

```
>>> texte[2::3]
```

```
'cC3cCf'
```

```
def extraire(chaîne,début,fin,pas):
```

SCRIPT

```
    tranche = ''
```

```
    for i in range(début,fin,pas):
```

```
        tranche = tranche + chaîne[i]
```

```
    return tranche
```

```
>>> extraire(texte,0,len(texte),3)
```

SHELL

```
'aA1aAf'
```

```
>>> extraire(texte,4,12,3)
```

```
'B2b'
```

```
>>> extraire(texte,2,len(texte),3)
```

```
'cC3cCf'
```

- ▶ Vous savez tous (?) écrire la fonction testant l'appartenance à une séquence.

```
def appartient(lettre, texte):  
    for i in range(len(texte)):  
        if lettre == texte[i]:  
            return True  
    return False
```

SCRIPT

- ▶ La fonction appartient n'est autre que l'opérateur `in` de Python.

```
>>> 'a' in 'Koala'  
True
```

SHELL

- ▶ Ce `in` n'a rien à voir avec celui de la boucle `for`
 - ▶ `for i in range(len(chaine))` : `i` varie dans `[0, len(chaine)-1]`
 - ▶ `c in chaine` : on teste si `c` $\in$ `chaine`

- Comptons le nombre de voyelles dans un texte.

```
def voyelles(chaine):  
    res=0  
    for i in range(len(chaine)):  
        c = chaine[i]  
        if c in 'aeiouy':  
            res=res+1  
    return res
```

SCRIPT

```
>>> voyelles('Python')  
2  
>>> voyelles("Ajourné") # Oups...  
2
```

SHELL

- ▶ Python est un langage dont les types sont des **classes**.

```
>>> type(23)
<class 'int'>
>>> type('Coucou')
<class 'str'>
```

SHELL

- ▶ Un membre d'une classe est un **objet**.
 - ▶ 23 est un objet de la classe `int`
 - ▶ `'Coucou'` est un objet de la classe `str`
- ▶ La programmation **orientée objet** est un style de programmation qui dépasse le cadre de ce cours, mais nous en verrons les rudiments.
- ▶ Une des principales différences est que des fonctions peuvent être attachées à un objet. On parle de **méthodes** et on utilise une syntaxe particulière.

```
>>> position('c', 'On appelle une fonction')
18
>>> 'On appelle une méthode'.find('c')
-1
```

SHELL

SHELL

```
>>> '123'.isdigit()           # que des chiffres ?
True
>>> 'Monde'.isalpha()         # que des lettres ?
True
>>> 'rayon+2'.islower()       # lettres minuscules ?
True
>>> 'RAYON+2'.isupper()       # lettres capitales ?
True
>>> 'RAYON+2'.lower()         # mettre en minuscules
'rayon+2'
>>> 'rayon+2'.upper()         # mettre en capitales
'RAYON+2'
>>> '  Olivier  '.strip()      # enlever les blancs aux bouts
'Olivier'
>>> 'abracadabra'.strip('ba')  # enlever aux bouts b et a
'racadabr'
>>> 'Zip chic'.replace('ip','o') # remplacement
'Zo chic'
```



Déconseillé aux moins de 18 ans.



- Dans certaines applications avancées, on peut évaluer une expression stockée dans une chaîne.

```
>>> eval('1+1')  
2
```

SHELL

- Application amusante

```
def évaluer(f,x):  
    """ Méthode de goret """  
    c=f.replace('x',str(x))  
    return eval(c)
```

SCRIPT

```
>>> f='2*x**2+x+1'  
>>> évaluer(f,5)  
56  
>>> évaluer(f,0)  
1  
>>> évaluer(f,'0 or méchant()#')  
Autodestruction de la machine
```

SHELL

- Peut poser des problèmes de sécurité.
 - En python `False==0`! (Oui, je sais, c'est très moche...)
 - On a pu exécuter le code malicieux voulu — ici la fonction `méchant()`.

- ▶ Dans cette UE, sauf mentions contraires, vous n'avez pas le droit d'utiliser
 - Les tranches (*slices*) (`chaîne[1:2:3]`)
 - Le mot clé `in` pour tester l'appartenance.
 - Les méthodes
- ▶ Vous devez savoir faire chacun de ces algo à la main.
- ▶ Après ce cours, faites ce que vous souhaitez.

- 🍃 Partie I. Représentation
- 🍃 Partie II. Couleurs et tuples
- 🍃 Partie III. Compléments
- 🍃 **Partie IV. Représentation des caractères**
- 🍃 Partie V. Cryptologie
- 🍃 Partie VI. Table des matières

- Un chaîne peut être définie avec des guillemets simples ' ou double "

```
>>> 'Coucou' == "Coucou"  
True
```

SHELL

- Mais comment représenter les symboles guillemets ?

```
>>> texte = 'C'est une bonne question !'  
File "<console>", line 1  
    texte = 'C'est une bonne question !'  
                ^
```

SHELL

```
SyntaxError: unterminated string literal (detected at line  
1)
```

- Deux solutions

- On utilise un échappement \'
- On utilise des guillemets doubles "

```
>>> 'C\'est la bonne réponse'  
"C'est la bonne réponse"  
>>> "C'est aussi la bonne réponse"  
"C'est aussi la bonne réponse"
```

SCRIPT

- ▶ Un ordinateur ne comprend que les nombres.
- ▶ Un caractère est codé sur la machine comme un nombre.
- ▶ Les caractères américains sont numérotés de 0 à 127, c'est le code ASCII (AMERICAN STANDARD CODE FOR INFORMATION INTERCHANGE).

```
>>> ord('A')  
65  
>>> ord('a')  
97  
>>> ord('1')  
49  
>>> ord(' ')  
32
```

SHELL

```
>>> chr(65)  
'A'  
>>> chr(64)  
'@'  
>>> chr(10) # new line  
'\n'  
>>> chr(7) # bip !  
'\x07'
```

SHELL

- ▶ `"\n"`, `"\'"` et `'\"'` sont considérés chacun comme étant un caractère

SHELL

>>> `ascii()`

32	33 !	34 "	35 #	36 \$	37 %	38 &	39 '
40 (	41)	42 *	43 +	44 ,	45 -	46 .	47 /
48 0	49 1	50 2	51 3	52 4	53 5	54 6	55 7
56 8	57 9	58 :	59 ;	60 <	61 =	62 >	63 ?
64 @	65 A	66 B	67 C	68 D	69 E	70 F	71 G
72 H	73 I	74 J	75 K	76 L	77 M	78 N	79 O
80 P	81 Q	82 R	83 S	84 T	85 U	86 V	87 W
88 X	89 Y	90 Z	91 [	92 \	93]	94 ^	95 _
96 `	97 a	98 b	99 c	100 d	101 e	102 f	103 g
104 h	105 i	106 j	107 k	108 l	109 m	110 n	111 o
112 p	113 q	114 r	115 s	116 t	117 u	118 v	119 w
120 x	121 y	122 z	123 {	124	125 }	126 ~	

- L'affichage de la table est programmée en Python.
- Vivement le prochain TP : ce sera à vous de le faire !

- Il y a seulement 127 caractères ASCII (chacun codé sur 7 bits) ce qui n'utilise que la moitié des 256 octets disponibles (de 0 à 127).
- Il reste l'autre moitié (de 128 à 255) pour faire n'importe quoi !
- Chaque pays, chaque région du monde, a codé les caractères dont il avait besoin sur les 128 autres. Notre norme à nous était l'ISO-Latin-1

SHELL

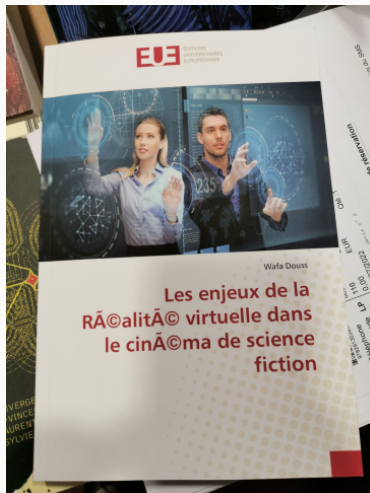
>>> iso_latin_1()

160	161	162	163	164	165	166	167
168	169	170	171	172	173	174	175
176	177	178	179	180	181	182	183
184	185	186	187	188	189	190	191
192	193	194	195	196	197	198	199
200	201	202	203	204	205	206	207
208	209	210	211	212	213	214	215
216	217	218	219	220	221	222	223
224	225	226	227	228	229	230	231
232	233	234	235	236	237	238	239
240	241	242	243	244	245	246	247
248	249	250	251	252	253		

- Ce fut un âge obscur avec de nombreux problèmes de codage.

Unicode ou Latin-1 ?

De l'importance des accents



- Unicode (≈ 1990) a proposé d'abandonner la limitation à 255 caractères, en traitant toutes les langues du monde (> 65000 caractères!).

```
grec = "Πυθαγόρας καὶ Πύθων"  
esperanto = "Ĉirkaŭaĵo en esperanto"  
emoji = "J'💖🐸"  
chinois = "\u6211\u662f\u6cd5\u56fd\u4eba"  
arabe="حفاث"  
sumérien ="𐎶𐎵 𐎶𐎥"  
  
print(grec,esperanto,chinois,sumérien,arabe,emoji,sep='\n\n')  
  
print()  
for i in range(0x0661,0x066a):  
    print(chr(i), end=" ")  
print()
```

Πυθαγόρας καὶ Πύθων
Ĉirkaŭaĵo en esperanto
我是法国人
𐎶𐎵 𐎶𐎥
حفاث
J'💖🐸
٩ ٨ ٧ ٦ ٥ ٤ ٣ ٢ ١

- Grec, Arabe, Chinois, Géorgien, Inuit, Cunéiforme, Emoji!
- Chaque caractère a un numéro unicode unique.

- ▶ La base 16 utilise 16 chiffres : **0123456789abcdef** ($a = 10, b = 11 \dots f = 15$)
- ▶ Chaque caractère unicode peut être représenté par son numéro en base 16.

SHELL

```
>>> a # a='    '
'  '
>>> ord('  ') # Num  ro unicode de '  ' en base 10
8364
>>> 2*16**3 + 10*16**1 + 12*16**0 # a=10 et c=12
8364
>>> chr(8364)
'  '
>>> chr(0x20ac)
'  '
```

- ▶ '  ' a pour num  ro unicode $8364_{10} = 20ac_{16}$ obtenue
 - ▶ avec `ord('  ')`
 - ▶ sur internet <http://www.unicode.org/>
- ▶ On peut le saisir sous diff  rents formats :

'    ' chr(8364) chr(0x20ac) ou simplement '  '

- ▶ Comme les caractères Unicode ne tiennent plus tous sur un seul octet, certains caractères sont représentés sur 2, 3, voire 4 octets. Il y a plusieurs conventions d'encodage possibles pour un même caractère.
- ▶ UTF-8 : le standard.
 - ▶ code les 127 premiers caractères comme ASCII
 - ▶ tout texte ASCII est un texte UTF-8 !
 - ▶ Les autres caractères sont sur plusieurs octets.
 - ▶ 'é' : deux octets. Caractères chinois : trois octets.
- ▶ UTF-16 : Windows (qui n'aime pas faire comme les autres)
- ▶ UTF-32 : Tous les caractères ont la même taille : 32 bits.
- ▶ Le numéro unicode ne correspond pas (exactement) au codage UTF-8 !
- ▶ Unicode est un annuaire de caractères ; UTF-8 est un codage machine.

- 🍃 Partie I. Représentation
- 🍃 Partie II. Couleurs et tuples
- 🍃 Partie III. Compléments
- 🍃 Partie IV. Représentation des caractères
- 🍃 **Partie V. Cryptologie**
- 🍃 Partie VI. Table des matières

- ▶ La **cryptographie** est l'art d'écrire et de lire des codes secrets.
 - ▶ Alice **chiffre** un message pour Bob.
 - ▶ Bob **déchiffre** le message d'Alice.
- ▶ La **cryptanalyse** est l'art de casser les codes secrets sans connaître la clé.
 - ▶ Eve, la méchante, **décrypte** la conversation entre Alice et Bob.
- ▶ La **cryptologie** réunit la cryptographie et la cryptanalyse.
- ▶ Remarque : crypter un message n'a aucun sens...
...sauf apparemment pour les journalistes.

Une des premières formes de chiffrements d'un message est le chiffrement par décalage circulaire, appelé code de César.

- ▶ César l'aurait utilisé pour ses correspondances durant la Guerre des Gaules.
- ▶ Principe : on transforme lettre par lettre le message.

Exemple : $A \rightarrow S, B \rightarrow T, C \rightarrow U, \dots, Z \rightarrow R$.

Le message ZEBRA devient RWTjS.

- ▶ Ici on décale de 18 lettres.
- ▶ $k = 18$ est la clé de chiffrement.
- ▶ On peut choisir n'importe quelle clé entre 1 et 25.
- ▶ César utilisait la clé $k = 3$.



- ▶ On ne code que les CAPITALES.
- ▶ Les étapes pour chiffrer UNE lettre.
 - ▶ On calcule l'index i de la lettre de l'alphabet avec `ord`.
 - ▶ On ajoute la clé à l'index, modulo 26 (la taille de l'alphabet) : $(i+k)\%26$
 - ▶ On calcule la lettre à cet index dans l'alphabet avec `chr`, en utilisant le code ASCII.
- ▶ Pour chiffrer un message de plusieurs lettres : on chiffre lettre à lettre en stockant le résultat dans une chaîne.

- ▶ Le déchiffrement est très simple :
 - ▶ il suffit de connaître la clé k utilisée pour le chiffrement
 - ▶ On décale alors de $-k$ (modulo 26) pour retrouver le message de départ
- ▶ Et si on ne connaît pas la clé k ? La cryptanalyse du code de César est possible...en testant toutes les clés (vivement le TP!).
- ▶ On parle de **cryptographie symétrique** (ou à clé secrète), la même clé permet de chiffrer et déchiffrer. Mais il faut échanger cette clé secrète à l'avance (et ne pas la révéler à tout le monde).
- ▶ Pour échanger la clé symétrique, les techniques modernes reposent sur la **cryptographie asymétrique** (ou à clé publique).
 - ▶ Bob a deux clés : sa clé privée et sa clé publique.
 - ▶ Alice utilise la **clé publique** de Bob pour lui écrire un message **chiffré**.
 - ▶ Bob utilise sa **clé privée** pour **déchiffrer**.
 - ▶ La clé publique ne permet pas de déchiffrer le message.

Merci pour votre attention

Questions



Cours 4 — Codages et représentations

Partie I. Représentation

Qu'est-ce qu'un nombre négatif?

Véritable représentation des entiers

Codons les négatifs

Calcul de la négations

Partie II. Couleurs et tuples

Le système RGB

Mélangeons les couleurs

🔧 Application

Couleur et hexadécimale

Partie III. Compléments

Extraction d'une sous-chaîne (1/2)

Extraction d'une sous-chaîne (2/2)

Tester l'appartenance (1/2)

Tester l'appartenance (2/2)

Des fonctions étranges

Exemples de méthodes de la classe str

Exécuter une chaîne

Règles de l'UE

Partie IV. Représentation des caractères

Guillemets simples et doubles

Le codage ASCII

La table ASCII

Autres caractères

Âges obscures

Unicode

Codage unicode

Des caractères aux codages machines

Partie V. Cryptologie

Définitions

Le code de César

César en Python

Déchiffrer un message

Partie VI. Table des matières

- ▶ © 2024 —Olivier Baldellon
- ▶ Ce document est publié sous licence **CC-BY Attribution 4.0** 
- Vous êtes autorisé à :
 - ▶ **Partager** —copier, distribuer et communiquer le matériel par tous moyens et sous tous formats pour toute utilisation, y compris commerciale.
 - ▶ **Adapter** —remixer, transformer et créer à partir du matériel pour toute utilisation, y compris commerciale.
- Selon les conditions suivantes :
 - ▶ **Attribution** —Vous devez créditer l'œuvre, intégrer un lien vers la licence et indiquer si des modifications ont été effectuées à l'œuvre. Vous devez indiquer ces informations par tous les moyens raisonnables, sans toutefois suggérer que l'Offrant vous soutient ou soutient la façon dont vous avez utilisé son œuvre.
- ▶ <https://creativecommons.org/licenses/by/4.0/deed.fr>