Résumé Première NSI

1. Constructions élémentaires

Types de variables

`a = True`	type bool (booléen)
`b = False`	type bool
Variable de type int (entier) Sa valeur a plusieurs écritures possibles : `c = 2`	- Ecriture en base 10.
`d = 0b1111`	- Ecriture en base 2. Le zéro pour commencer indique qu'on donne l'écriture dans une autre base que la base 10. Le *b* signifie que l'écriture est en *binaire*
`r = 2 + 0b1111`	r vaut 17
`e = 0xf`	- Ecriture en base 16. Le zéro pour commencer indique qu'on donne l'écriture dans une autre base que la base 10. Le *x* signifie que l'écriture est en *hexadécimal*.
`s = 4 + 0xf`	s vaut 19
Pour afficher la chaine ' 0b….' correspondant à n	
`bin(2)`	vaut '0b10'
`bin(0xf)`	vaut '0b1111'
Pour afficher la chaine ' 0x….' correspondant à n	
`hex(15)`	vaut '0xf'
`hex(0b1110)`	vaut '0xe'
Pour convertir une chaine en un entier :	
`int('0b1111', 2)` vaut 15 `int('15')` vaut 15 `int('0xf', 16)` vaut 15	
`f = 3.14`	type float (à virgule flottante)
`g = "Bonjour"`	type str (chaîne de caractères)
`h = ["un", 2]`	type list (liste)

Opérations sur int et float

`a // b`	Quotient dans la division entière.
`a % b`	Reste dans la division entière.
`a**b`	a exposant b.
`abs(a)`	Valeur absolue.
`round(a, n)`	Arrondi à n décimales.

Affectation

Affecter, c'est associer une valeur à une variable.

1. Evaluation du membre à droite du signe =
2. Affectation de cette valeur à la variable dont le nom est à gauche.

`score = 10`	Affecte à la variable *score* la valeur 10.
`x = 1.2 * sin(y)`	Evalue *y*, puis *sin(y)* puis le produit par *1,2* et affecte le résultat à la variable *x*.
`a, b = 2, 3`	Signifie $a \leftarrow 2$ et $b \leftarrow 3$
`a = a + 1`	Signifie $a \leftarrow a + 1$
`b = b / 2`	Signifie $b \leftarrow b / 2$
`a += 1`	Raccourci pour `a = a + 1`
`b /= 2`	Raccourci pour `b = b / 2`

Conversion de type

	entier	flottant	chaine	liste
`int(a)`	√	√	√	X
`float(a)`	√	√	√	X
`str(a)`	√	√	√	√
`list(a)`	X	X	√	√

Exemples

`int(-3.14)`	Vaut -3 (troncature à l'entier)
`int('3')`	Vaut 3
`int('0b1111', 2)`	Vaut 15
`int('0xf', 16)`	Vaut 15
`float(3)` `float('3')`	Vaut 3.0 Vaut 3.0

`str(137)`
`str(3.14)`
`str([1, 2, 3])`
`str([1, 2, 3]).strip('[]')`
	Vaut '137'
	Vaut '3.14'
	Vaut '[1, 2, 3]'
	Vaut '1, 2, 3' # .strip *retire* les caractères spécifiés seulement s'ils sont trouvés *en début ou en fin de chaîne*.

`ch = ''.join(['1', '2', '3'])` `ch = 's'.join(['1', '2', '3'])`	ch vaut '123' ch vaut '1s2s3'
`list('Bel été')`	Vaut ['B', 'e', 'l', ' ', 'é', 't', 'é']
`list(str(137))`	Vaut ['1', '3', '7']

Boucle bornée (boucle for)
On dit aussi boucle inconditionnelle

Boucle for classique

```
for i in range(100):
    print(i)
```

Affiche :
0
…
99

Boucle for classique

```
for caractere in range('Bel'):
    print(caractere)
```

Affiche :
B
e
l

Boucle for classique

```
for i in range(1, 20, 5):
    print(i)
```

Affiche :
1
6
11
16

Boucle for énumérative

```
ma_liste = [1, 2, 1, 3, 1, 4]
for nombre in ma_liste:
    print(i)
```

Affiche :
1
2
1….

Boucle for énumérative avec index

```
ma_liste = [1, 2, 1, 3, 1, 4]
liste_avec_index = enumerate(ma_liste)
for i in liste_avec_index:
    print(i)
```

Affiche :
(0, 1)
(1, 2)
(2, 1)
(3, 3)…

Boucle for avec range inversé

```
for i in reversed(range(100)):
    print(i)
```

Affiche :
99
98
…
0

Affichage à l'écran

```
a = 16
print("Vous avez", a, "ans.")
```

Le message affiché est :

```
Vous avez 16 ans.
```

Autre méthode à partir de Python 3.6 :

```
a = "la variable 1"
b = 3
print(f"A cet endroit on a {a} et là on a {b}.")
```

Le message affiché est :

```
A cet endroit on a la variable 1 et là on a 3.
```

Boucle non bornée (boucle while)
On dit aussi boucle conditionnelle

On répète un bloc d'instructions tant qu'une condition est vraie

```
u = 5
while u < 10:
    u = 2 * u - 1
print(u)
```

Affiche 17 qui est la première valeur de u supérieure ou égale à 10.

Instruction conditionnelle (if)

Exécuter une ou de actions suivant qu'une proposition a la valeur True ou pas.
Différentes variantes existent :

`if condition:` `    instructions`	

`if condition:`
`    instructions`
`else:`
`    instructions`
	Le else n'est pas obligatoire

`if condition_1:`
`    instructions`
`elif condition_2:`
`    instructions`
`else:`
`    instructions`
	Le else n'est pas obligatoire

Exemple :

```
def mention(note):
    if note < 12:
        return "Passable"
    elif note < 14:
        return "Assez bien"
    elif note < 16:
        return "Bien"
    else:
        return "Très bien"
```

mention(14.5) retourne 'Bien'

Fonctions

```
def ma_fonction(arguments):
    """
    Description de ce que fait la fonction

    Parametres nommes
        argument 1 : de type …
        Le rôle de l'argument 1

        argument 2 : de type …
        Le rôle de l'argument 2.

    Retourne
        valeur 1 : de type …
        Le rôle de la valeur 1.
    """
    Le corps de la fonction est indenté.
    Le résultat est retourné avec return.
```

Exemple

```
def produit(a, b):
    """
    Calcule le produit d'entiers

    Parametres nommes
    -----------------
        a, b : de type int
        Les operandes du produit.

    Retourne
    --------
        p : de type int
        Le resultat du produit.

    """"

    p = a * b
    return p
```

Appel de la fonction :

```
mon_produit = produit(3, 5)
```

Import de librairies (modules)

`from math import *` # importe toutes les fonctions du module math
Principales fonctions : `sqrt, cos, pi …`

`from random import *` # importe toutes les fonctions du module random (probabilités).

```
randint(a, b) # Un entier aléatoire dans [a ; b]
choice(liste) # Un entier aléatoire dans liste
schuffle(liste) # Mélange aléatoirement liste
```

`from time import time` # importe time qui renvoie le nombre de secondes depuis le 1/1/1970.

```
start_time = time()
… traitement
end_time = time()
duree_traitement = end_time – start_time
```

Spécification de fonction et test

La spécification d'une fonction comporte les informations suivantes :
- La tâche effectuée
- Les contraintes que doivent respecter les paramètres (autrement dit les variables d'entrée)
- Ce qu'on peut attendre des résultats retournés.

Elle est résumée dans la « docstring » écrite au début du corps de la fonction (voir l'exemple de la fonction produit)

Le test d'une fonction f se fait de manière variée :
- Tester f manuellement sur quelques exemples.
- Ecrire une fonction test_f

```
def test_f:
    for a in range(…):
        for b in range(…):
            if not (invariant de boucle de f ):
                message = "Echec pour a = ", str(a)
                return False, message
    return True
```

- Utiliser ses connaissances Ex : copie de listes.
- Tester les cas limites Ex : avec une liste vide.
- Utiliser le site de visualisation *pythontutor.com*

2. Variables de type booléen et de type entier

Evaluation d'une proposition

`<, >, <=, >=` Pour comparer.
`==` Pour tester l'égalité.
`!=` Pour tester la différence.
`in` Pour tester l'appartenance à *liste* ou *chaîne*.

`and, or, not` Opérateurs logiques.

`&, |, ~, ^` Pour appliquer les opérateurs et, ou, not, xor sur plusieurs bits à la fois.

`<<` Pour décaler à gauche plusieurs bits.
`>>` Pour décaler à droite plusieurs bits.

`3 <= 4` a la valeur True.
`'Paul' == 'Georges'` a la valeur False.
`e in 'Fred'` a la valeur True.
`1 not in [1, 2, 4]` a la valeur False.
(3 <= 4) and (5 > 4) a la valeur False
(3 <= 4) or (5 > 4) a la valeur True
Evaluation paresseuse :
- Si la proposition p a la valeur False alors *p and q* a la valeur False sans avoir évalué la valeur de q.
- Si la proposition p a la valeur True alors *p or q* a la valeur True sans avoir évalué la valeur de q.

3. Variables de type construit

Listes en compréhension

Exemples

```
liste = [2 * i for i in range(10)]
```

ou

```
liste = [i for i in range(20) if i%2 == 0]
```

Dans ces deux exemples, *liste* prend la valeur :

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Si on a `L = [2, 3, 5, 7]` alors `liste = [i for i in L if i%2 == 0]` a pour valeur `[2]`.

Concaténation : chaînes, listes ou tuples

La concaténation consiste à relier.

`"Bel " + " été "`	a pour valeur "Bel été"
`"Bel" * 3`	a pour valeur "BelBelBel"
`[1, 2] + [4, 5]`	a pour valeur [1, 2, 4, 5]
`[2, 3] * 3`	a pour valeur [2, 3, 2, 3, 2, 3]
`(1, 2) + (3, 4)`	a pour valeur (1, 2, 3, 4)
`(1, 2) * 3`	a pour valeur (1, 2, 1, 2, 1, 2)

Tuples

- Un tuple est une liste non mutable (c'est-à-dire non modifiable). On ne peut pas changer une valeur existante. mon_tuple[0] = 5 renvoie une erreur.
- On peut ajouter un élément supplémentaire à un tuple déjà existant.
- On peut convertir une liste en tuple et un tuple en liste.

`t0 = ()`	Un tuple vide
`t1 = (1,)` ou `t1 = 1,`	Un tuple à un élément. Attention : t = (1) est un entier et non un tuple.
`t2 = (3.14, 2)` ou `t2 = 3.14, 2`	Un tuple à deux éléments
`a, b = (2, 3)`	Signifie $a \leftarrow 2$ et $b \leftarrow 3$
`t2[0]` `t2[-1]`	Vaut 3,14 Vaut 2
`t1 + t2`	Vaut (1, 3.14, 2)
`2 * t2`	Vaut (3.14, 2, 3.14, 2)
`t2 + (1,)`	Vaut (3.14, 2, 1)
`L = [2, 3, 5, 7]` `t = tuple(L)`	t vaut (2, 3, 5, 7)
`t = (2, 3, 5, 7)` `ma_liste = list(t)`	ma_liste vaut [2, 3, 5, 7]

Soit *a* une liste, une chaine ou un tuple

`list(range(10))`	Création de [0, 1, ... ,9]
`tuple(range(10))`	Création de (0, 1, ... ,9)
`list(range(2,5))`	Création de [2, 3, 4]
`tuple(range(2,5))`	Création de (2, 3, 4)
`len(a)`	Vaut la longueur de a.
`a[0]`	Vaut l'élément d'indice 0.
`a[-1]`	Vaut le dernier élément.
`a[p : q]`	Vaut la liste, la chaine ou le tuple avec les éléments de a[p] à a[q – 1].
`a[p :]`	Vaut la liste, la chaine ou le tuple avec les éléments a[p] au dernier.
`a[: q]`	Vaut la liste, la chaine ou le tuple avec les éléments du premier à a[q – 1].

Modification des listes

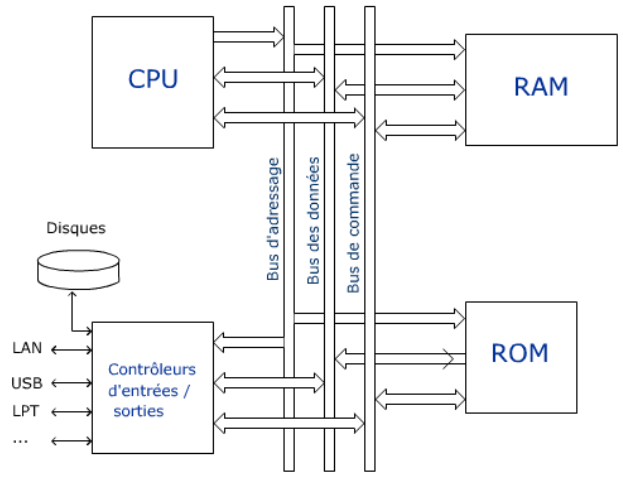
`a = []` ou `a = list()`	Crée une liste vide.
`a = a + [elt]` ou `a.append(elt)`	Ajoute l'élément *elt* à la fin de la liste.
`a.insert(i, elt)`	Insère elt à la position i.
`del a[k]`	Supprime l'élément a[k].
`a.remove(elt)`	Supprime la première occurrence de a[k] de a.
`a.pop(k)`	Supprime a[k] de a (et vaut `a[k]`).

`ma_chaine.split()`	`ma_chaine = "Bel été !"`
	`ma_chaine.split()`
	renvoie la liste
	`['Bel', 'été', '!']`

`"r".join(chaine)`	Ajoute un r entre chaque caractère de la chaîne.
`list(a)`	*list()* renvoie une copie de la liste *a pour a de dimension 1.*
`import copy` `copy.deepcopy(a)`	*copy.deepcopy()* renvoie une copie de la liste *a quelle que soit la dimension de a.*
`a.sort()`	Trie sur place la liste *a* par ordre croissant.
`a.sort(reverse = True)`	Trie sur place la liste *a* par ordre décroissant.
`sorted(a)`	Renvoie une copie de la liste triée par ordre croissant et laisse *a* intacte.
`sorted(a, reverse = True)`	Renvoie une copie de la liste triée par ordre décroissant et laisse *a* intacte.
`elt in a`	Vaut True si *elt* est dans la liste *a*, False sinon.
`a.count(ma_valeur)`	Vaut le nombre de *ma_valeur* dans la liste *a*
`a.index(ma_valeur)`	Vaut l'index de la première occurrence de ma_valeur.

4. Machines et systèmes d'exploitation

Schéma bloc d'un ordinateur



Demi – additionneur (c'est-à-dire sans retenue entrante)

retenue	0	
A		0
+ B		0
S		0

retenue	0	
A		0
+ B		1
S		1

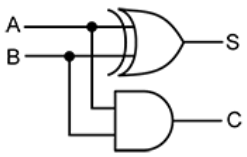
retenue	0	
A		1
+ B		0
S		1

retenue	1	
A		1
+ B		1
S		0

Table de vérité de la somme S et de la retenue C :

A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Ainsi, S = A *xor* B et C = A *and* B.



Désassemblage d'une docstring Python en bytecode

```
from dis import dis

dis("""
if x > 3:
   y = x + 1
else:
   y = x
""")
```

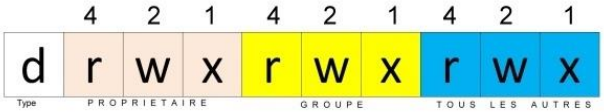
Retourne :

```
0   LOAD_NAME                0 (x)
2   LOAD_CONST               0 (3)
4   COMPARE_OP               4 (>)
6   POP_JUMP_IF_FALSE       18

8   LOAD_NAME                0 (x)
10 LOAD_CONST                1 (1)
12 BINARY_ADD
14 STORE_NAME                1 (y)
16 JUMP_FORWARD              4 (to 22)

18 LOAD_NAME                 0 (x)
20 STORE_NAME                1 (y)
22 LOAD_CONST                2 (None)
```

Système d'exploitation

Fonctions : Organiser les données en fichiers dans des dossiers. Il permet de lire, d'écrire (et de supprimer) et d'exécuter de fichiers.

Lignes de commande dans un terminal Linux :
mkdir : créer un dossier. Ex : mkdir seance_4
cd : change le dossier. Ex : cd seance_4
ls – l : Affiche les fichiers avec les droits d'accès



Ex : drwxr-xr-w Documents indique que Documents est un dossier (présence de d au début).
Le propriétaire (le "user") a les droits de lecture, d'écriture et d'exécution.
Les utilisateurs du même groupe que le propriétaire ont les droits de lecture et d'exécution.
Les autres ont les droits de lecture et d'exécution.

chmod : Change les droits.
Ex : chmod u+rwx, g+rx-w, o+r-wx fichier3
donne les droits : -rwxr-xr—sur le fichier3

5. Entiers relatifs, réels et caractères

Entiers relatifs

Contrairement aux entiers naturels (non signés), il est nécessaire de définir le nombre n de bits sur lesquels on code les entiers relatifs (c'est à dire signés) puisque le bit de poids fort est le bit de signe.

Pour prendre l'opposé d'un nombre on prend son "complément à 2^n" ou simplement son "complément à 2". Si on travaille avec n = 5 bits seulement 4 bits sont utilisés pour le nombre.
Si le 5ème est 0 alors le nombre est positif.
Si le 5ème est 1 alors le nombre est négatif.

1. Exemples de codage d'entiers signés :

- Coder le nombre +9 avec 5 bits :
Réponse :
Le bit de signe est 0.
$9 = 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$
Donc +9 est codé sur 5 bits par 01001

- Coder le nombre -10 avec 5 bits :
Réponse :
On prend le complément à 2 de 10 qui s'écrit 01010
1. Le NOT de 01010 est 10101
2. Ajouter 1 : 10101 + 00001 donne 10110
3. Ignorer le 6ème bit s'il y en a un

Donc -10 est codé sur 5 bits par 10110

2. Exemples de décodage d'entiers signés codés sur 5 bits :

- Donner l'écriture décimale de 01001
Réponse :
Le bit de signe est 0 donc c'est un nombre positif.
On calcule sa valeur avec les autres bits :
$1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 9$
Donc le nombre signé codé par 01001 est +9.

- Donner l'écriture décimale de 10110
Réponse :
Le bit de signe est 1 donc c'est un nombre négatif.
On cherche son opposé en prenant le complément à 2 de 10110 :
1. Le NOT de 10110 est 01001
2. Ajouter 1 : 01001 + 00001 donne 01010
3. Ignorer le 6ème bit s'il y en a un

On obtient 01010 est positif et qui vaut
$1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 10$
Donc le nombre de départ est -10.

Nombres à virgule flottante

Les nombres de type *float* représentent les réels.
En 64 bits, un nombre de type *float* sera codé ainsi :

s	e	m'
signe	expo. décalé	mantisse tronquée
1	01111111101	10000000000000….0000
1 bit	11 bits	52 bits

Exemple : Quel nombre x représenté ?
s = 1 donc le nombre est négatif
$e = 0 \times 2^{10} + 1 \times 2^9 + \dots + 0 \times 2^1 + 1 \times 2^0 = 1021$ donc la puissance $p = 1021 - 1023 = \mathbf{-1}$
m' = ,10000….00 donc la mantisse m = 1,10000…00
Donc :
$x = -1{,}1 \times 2^{-1}$ en binaire
x = - 0,11 en binaire
En décimal, on a :
$x = -(0 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2})$
x = - (0,5 + 0,25)
x = -0,75
Sur 64 bits, on peut représenter par des nombres à virgule flottante les réels sur $[-1{,}7\ 10^{308} ; 1{,}7\ 10^{308}]$
La représentation exacte n'existe pas pour beaucoup de nombres. Par exemple pour 0,1.

Textes

- Les lettres, les signes de ponctuation, les chiffres sont encodés sous la forme d'entiers.
- Différentes tables de correspondance existent ASCII, ISO 8859-1 (ou latin-1), utf-8
Une table de caractères est un *charcater set* ou *charset*. Une correspondance est un *point de code*.
ord() et chr() donnent les points de code latin-1 :
`ord('é')` Type *int* et vaut 233 (écriture décimale)
`chr(233)` Type *str* et vaut é
`chr(0xe9)` Type *str* et vaut é (car on a $233_{10} = E9_{16}$)

***ma_chaine*.encode** retourne un objet du type *byte.* Sa valeur commence par b suivi de guillemets encadrant des caractères présent dans la table ASCII. Les caractères qui n'existent pas en ASCII sont remplacés par leur code hexadécimal en utf-8.
`'A'.encode('ascii')` Type *byte* et vaut b'A'
`'été'.encode('latin-1')` Vaut b'\xe9t\xe9'
`'été'.encode('utf-8')` Vaut b'\xc3\xa9t\xc3\xa9'

***mes_bytes*.decode** retourne un objet du type *str.*
`b'\xe9t\xe9'.decode('latin-1')` Type *str* et vaut été
`b'\xc3\xa9t\xc3\xa9'.decode('latin-1')` vaut Ã©tÃ©
`b'\xc3\xa9t\xc3\xa9'.decode('utf-8')` vaut été

Conversion du type *list* d'entiers vers le type *bytes* :
```
ma_liste_hex = [0xc3, 0xa9, 0x74, 0xc3, 0xa9]
mes_octets = bytes(ma_liste_hex) vaut
                                    b'\xc3\xa9t\xc3\xa9'
```

Conversion du type *bytes* vers le type *list* d'entiers :
```
mes_octets = b'\xc3\xa9t\xc3\xa9'
ma_liste = []
for elt in mes_octets :
    ma_liste.append(hex(elt))
```
ma_liste vaut ['0xc3', '0xa9', '0x74', '0xc3', '0xa9']

Ouverture et fermeture d'un fichier

Ouverture (et création) avec l'option 'w' (write)

```
mon_fichier = open('fichier.txt', 'w')
```

Crée le fichier fichier.txt dans le dossier où se trouve le fichier Python en cours (ou écrase le fichier 'fichier.txt' s'il existe déjà dans ce dossier).

Ecriture

```
mon_fichier.write("Une ligne\nEt je ferme")
```

Ecrit deux lignes dans fichier.txt.

\n est un saut de ligne.

Fermeture

```
mon_fichier.close()
```

Après une ouverture, il est indispensable de signaler au système d'exploitation qu'on ferme le fichier.

Ajout avec l'option 'a' (append)

```
mon_fichier = open('fichier.txt', 'a')
mon_fichier.write("\nEncore une ligne.")
mon_fichier.close()
```

Ajoute une nouvelle ligne à fichier.txt.
Si on ré ouvre avec l'option **'w'**, **on efface tout**.

Ajout de nombres

Pour enregistrer des valeurs de variables dans un fichier texte, celles-ci doivent être converties en chaines de caractères :

```
mon_fichier.write(str(a))
```

Ajout de tabulations

Pour ajouter des tabulations, il faut ajouter des chaines de caractères "\t"

```
mon_fichier.write(str(a) + "\t" + str(b))
```

Exemple de lecture de fichier

```
"""
Ouverture du fichier en mode lecture (option 'r')
Pour chaque ligne (séparée par un saut de ligne) :
    liste = les éléments séparés selon les tabulations
"""
mon_fichier = open('fichier.txt', 'r')
for ligne in mon_fichier:
    liste = ligne.rstrip().split("\t")
mon_fichier.close()
```

Remarques : .rstrip() sert à retirer les espaces à droite du dernier caractère de la chaine.
.split("\t") divise la ligne en mots selon les tabulations.

Vérifier le type d'encodage

Dans le terminal, saisir :
file(fichier.txt)
Si le fichier ne contient que des caractères du charset ASCII, alors il est du type ASCII.
Sinon il est automatiquement du type utf-8.
file * permet de voir le type de tous les fichiers présents dans le dossier.

Ecrire un fichier source en C et le compiler en un fichier exécutable.

Exemple : Dans un dossier langage_c, avec Geany, créer le fichier test_depassement.cpp

```
#include <iostream>
using namespace std;

int main(){
    short b;
b = 32000 + 1000;
cout<<"Valeur de b: "<<b<<endl;
return 0;
}
```

Enregistrer le fichier et quitter Geany. Dans le terminal se placer dans le dossier langage_c. Saisir
$ **sudo g++ test_depassement -o test**

Ceci crée un fichier **test** exécutable.
Pour l'exécuter, dans le terminal, saisir $ **./test**

On peut vérifier que **test_depassement** est un fichier source et que **test** est un fichier exécutable.
Il suffit pour cela, de saisir : $ **file ***

Charset ISO 8859-1 (ou latin -1) La première partie (en rouge) est commune avec la table ASCII

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{		}	~	
8																
9																
A		¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬	-	®	¯
B	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
C	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
D	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
E	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
F	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

Exemple : Le point de code 41 se lit à l'intersection de la ligne 4 et de la colonne 1. C'est la lettre 'A'.
Le nombre associé à la lettre 'A' est 41 en hexadécimal soit $4 \times 16^1 + 1 \times 16^0 = 65$ en décimal.