$\mathcal{MPSI}$ 2

LYCEE CHARLEMAGNE
2020/2021

JEUDI 5 NOVEMBRE

I.P.T.

Il existe en informatique trois types de boucles

Impératives	Conditionnelles	

`for k in range(n) :`
`....instruction`
`....instruction`
`....instruction`
	`while condition :`
	`....instruction`
	`....instruction`
	`....instruction`
		`repeat instruction`
		`......instruction`
		`......instruction`
		`until condition valide`

		n'existe pas en Python

La boucle impérative peut se baser sur

une liste qui existe déjà	un `range` qu'on définit

`Somme = 0`
`for Note in DS :`
`....Somme += Note`
	`F = 1`
	`for k in range(n) :`
	`....F *= k+1`

Vous aurez souvent l'habitude d'utiliser des `range`, mais c'est quand même parfois bien lourd.

Par exemple, si `DS` est une liste de notes, pourquoi utiliser

```
S = 0
for k in range(len(DS)) :
....S += DS[k]
```

pour calculer la somme des notes ?

Ce n'est pas faux, mais ça force l'ordinateur à faire un travail redondant (créer une liste d'index `k` allant de `0` à la longueur de la liste, parcourir cette liste d'index et pour chacun d'entre eux, aller chercher l'élément de `DS` d'index `k`. Autant dire « aligne les notes côte à côte et additionne les »... Et donc, autant utiliser la syntaxe indique dans l'exemple au dessus.

Pour les `range`, vous pouvez utiliser

un `range` classique	`range(n)`	les entiers de 0 inclus à n exclu	n éléments
	`range(5)`	0, 1, 2, 3, 4	
un `range` avec début et fin	`range(p, q)`	les entiers de p (inclus à exclu)	q-p éléments
	`range(-3, 3)`	-3, -2, -1, 0, 1, 2	
	`range(3, -3)`	rien, c'est logique	
un `range` avec un pas (step)	`range(p, q, r)`	les entiers entre p et q en avançant de r en r	$\left[\frac{q-p}{r}\right]$ termes
	`range(3, 19, 4)`	3, 7, 11, 15, 19	
	`range(7, 33, 4)`	7, 11, 15, 19, 23, 27, 31	
	`range(14, 6, -1)`	14, 13, 12, 11, 10, 9, 8, 7	$\left[\frac{6-14}{-1}\right]$ =?

Avec le pas de -1, on remonte en arrière. Attention aux extrémités.
Il pourrait être tentant de l'utiliser pour parcourir une liste à l'envers, mais qu'allez vous utiliser pour lire à l'envers la liste L :

`for k in range(len(L)) :` `....print(L[-k])`	`for k in range(len(L),0,-1) :` `....print(L[k])`
`for k in range(len(L)) :` `....print(L[-k-1])`	`for k in range(len(L)-1,-1,-1) :` `....print(L[k])`

Les deux scripts suivants servent à calculer des produits d'entiers impairs :

```
def Produit(n) :
....P = 1
....for k in range(n) :
........P *= 2*k+1
....return(P)
```

```
Produit(1) donne 1
Produit(4) donne 1.3.5.7
Produit(13) donne 1.3.5.7.9.11.13.15.17.19.21.23.25
```

```
def Produit(n) :
....P = 1
....for k in range(1,n,2) :
........P *= k
....return(P)
```

```
Produit(1) donne 1
Produit(4) donne 1.3
Produit(13) donne 1.3.5.7.9.11
```

Si la liste est vide, le programme saute et n'exécute rien. C'est le cas avec `range(0)`. Et c'est fort heureux...

```
def Factorielle(n) :
....F = 1
....for k in range(n) :
........F *= k+1
....return(F)
```

```
def Binomial(n, k) :
....B = 1
....for i in range(k) :
........B = B*(n-k)/(k+1)
....return(B)
```

```
def Puissance(a,n) :
....P = 1
....for k in range(n) :
........P *= a
....return(P)
```

```
def Puissance(a,n) :
....P = a
....for k in range(n-1) :
........P *= a
....return(P)
```

Lesquels de ces programmes sont cohérents, y compris pour les « petites valeurs » ?

L'avantage des boucles impératives est qu'on évite les risques de boucles folles, comme avec un `while` dans lequel la variable sur laquelle porte la condition n'est pas modifie.
Avec la boucle `for`, on sait à l'avance combien de fois l'instruction sera effectuée.
En quoi les boucles conditionnelles ci dessous ne sont elles pas à effectuer :

```
def CompteChiffres(n) :
....C = 0
....while n > 0 :
........C +=1
....n = n//10
....return(C)
```

```
def CompteChiffres(n) :
....C = 0
....P = 1
....while P < n :
........P *= 10
........C += 1
....return(C)
```

Que pensez vous des programmes contenant une instruction

```
while True :
....instruction
....instruction
```

.
.

On peut s'interroger ce qu'il advient quand on modifie la variable qui sert à parcourir la boucle.
Essayez les scripts ci-dessous

```
for k in range(10) :
....print(k)
....k = k+1
```

```
for k in range(10) :
....k = k+1
....print(k)
```

```
for k in range(10) :
....print(k)
....k = 3*k+1
....print(k)
```

On voit que le range est insensible à ce que l'on fait à la variable `k`. Quand on revient à la première ligne, on prend le `k` suivant dans la liste établie dès le début, c'est tout.

Il va de soi que c'est quand même une drôle d'ide de modifier la variable qui fait la boucle. Mais informatiquement, on peut envisager sans passer pour un fou

```
for k in range(n) :
....k=2*k+1
....instructions...
```

pour créer une « liste » de nombres impairs. A éviter quand même aux concours, préférez créer une variable auxiliaire

```
for k in range(n) :
....impair = 2*k+1
....instructions...
```

On peut même concevoir de modifier une variable qui sert à définir le `range`.
La chose qui suit est étrange

```
n=10
for k in range(n) :
....print(k)
....n *= 3
```

mais...
On doit aller de 0 à n, mais n s'éloigne au fur et à mesure. Est ce prudent ?
Oui. Car le `range(n)` du début crée automatiquement le t-uple `0, 1, 2, 3, 4, 5, 6, 7, 8, 9`[1] puis travaille avec. `k` va donc prendre dix valeurs et c'est tout. L'entier `n` est modifié en cours de route, mais pas le range, prédéfini.
En revanche, à la fin, on a bien `n=590490`.
Et on a `k=9`, dernière valeur prise par `k` dans la boucle.
En revanche, avec

```
n, k=10, 0
while k<n :
....print(k)
```

quelle est la valeur de k à la fin ?
Et allez vous lancer

```
n, k =10, 0
while k<n :
....k += 1
....n *= 10
```

J'ai écrit plus haut qu'avec une boucle impérative, on savait tout de suite combien de fois elle allait être exécutée.
Ce n'est pas tout à fait exact.
Il est possible que dans une procédure, à cause d'un `return` (ou grâce à un `return`, selon votre point de vue), vous sortiez de la boucle en cours de route.
Imaginons qu'on vous dise « *il y a dans la liste* `L` *au moins un réel négatif, trouvez le* ».
Allez vous parcourir la liste en entier ? Non, dès que vous allez trouver un nombre négatif, vous allez vous arrêter et me le donner.
C'est pourquoi vous utiliserez

```
def Negatif(L) :
....for Nombre in L :
....if Nombre<0 :
........return(Nombre)
```

A priori, en lisant `for Element in L`, Python dit « je vais prendre un à un les éléments de `L`, les appeler `Nombre`, et faire des tests avec chaque `Nombre` ».
On est donc parti pour `len(L)` boucles.
Toutefois, dès que le test `Nombre<0` est valide, on sort de la boucle et on retourne `Nombre`. Et on n'étudie pas les suivants. La case mémoire crée par Python avec tous les éléments appelés `Nombre` est perdue.

Attention, le programme

```
def Negatif(L) :
....for Nombre in L :
....if Nombre<0 :
........print(Nombre)
```

aurait un effet différent.
Pour sa part, il afficherait tous les nombres, effectuant ses `len(L)` boucles, avec conscience. Et à la fin, il ne retournerait rien.

Si vous tenez à avoir tous les nombres négatifs, vous devez créer une liste :

1. oui, un range est un n-uplet, et pas une liste ; il n'y a pas de crochets autour

```
def Negatif(L) :
....LesNegatifs = []
....for Nombre in L :
....if Nombre<0 :
........LesNegatifs.append(Nombre)
....return(LesNegatifs)
```

Question : que fera

```
def Negatif(L) :
....LesNegatifs = []
....for Nombre in L :
....if Nombre<0 :
........LesNegatifs.append(Nombre)
........return(LesNegatifs)
```

.
.
.
.
.
.

Un classique, plus ou moins bien traité par les élèves.
On a une liste de notes, et on cherche la meilleure note,
et l'index de l'élève qui a eu la meilleure note
ou même la liste des élève ayant réalisé ce record.
Par exemple, avec `L = [13, 15, 9, 10, 9, 16, 13, 10]`, Python doit répondre `[5]` (puisque `L[5]` est le maximum).
`L = [13, 15, 9, 10, 9, 15, 13, 10]`, Python doit répondre `[1, 5]` (puisque `L[1] et L[5]` sont égaux au maximum).
Approche classique : on cherche déjà le maximum par un parcours de la liste (ou alors par la fonction max)
on parcourt la liste et on dit qui a atteint ce maximum

```
Maxi = L[0] #initialisation d'un record
for Note in L : #parcours de la liste
....if Note > Maxi : #si on trouve mieux
........Maxi = Note #on mémorise

TheBest = [] #initialisation de la liste des meilleurs
for k in range(len(L)) : #parcours de la liste (encore)
....if L[k] == Maxi : #est ce un qui a atteint le maximum ?
........TheBest.append(k)
```

et pour les flemmards :

```
Maxi = max(L) #instruction directe

TheBest = [] #initialisation de la liste des meilleurs
for k in range(len(L)) : #parcours de la liste (encore)
....if L[k] == Maxi : #est ce un qui a atteint le maximum ?
........TheBest.append(k)
```

Mais il y a mieux. On attend que vous fassiez les deux en même temps.

```
Maxi = L[0] #initialisation d'un record
TheBest = [0] #et du gagnant...provisoire
for k in range(len(L)) : #parcours de la liste
....if L[k] > Maxi : #si on trouve mieux
........Maxi = Note #on mémorise le nouveau record
........TheBest = [k] #et on retient qui est ce nouveau vainqueur
....elif L[k] == Maxi : #si il a fait aussi bien
........TheBest.append(k) #on l'ajoute à la liste des gagnants
```

Variante :

```
Maxi = L[0] #initialisation d'un record
TheBest = [0] #et du gagnant...provisoire
for k in range(len(L)) : #parcours de la liste
....if L[k] > Maxi : #si on trouve mieux
........Maxi = Note #on mémorise le nouveau record
........TheBest = [] #on efface tout
....if L[k] == Maxi : #si il a fait aussi bien.. que lui même eventuellement
........TheBest.append(k) #on l'ajoute à la liste des gagnants
```

Exercice : que fait cette procédure ?

```
def initif(n) :
....k = 0
....while True :
........k +=1
........M = k*n
........while M%10==0 :
............M=M//10
........while M%10 == 1 :
............M=M//10
........if M==0 :
............return(k)
```

Passons maintenant à deux petits mots utilisés par Python (mais non utilises aux concours) : `break` et `continue`.

Une boucle `for` est de la forme

.
.
.
.
.
.

```
for element in quelquechose :
....instruction
....instruction'
....encore
....instruction"
suite du programme
```

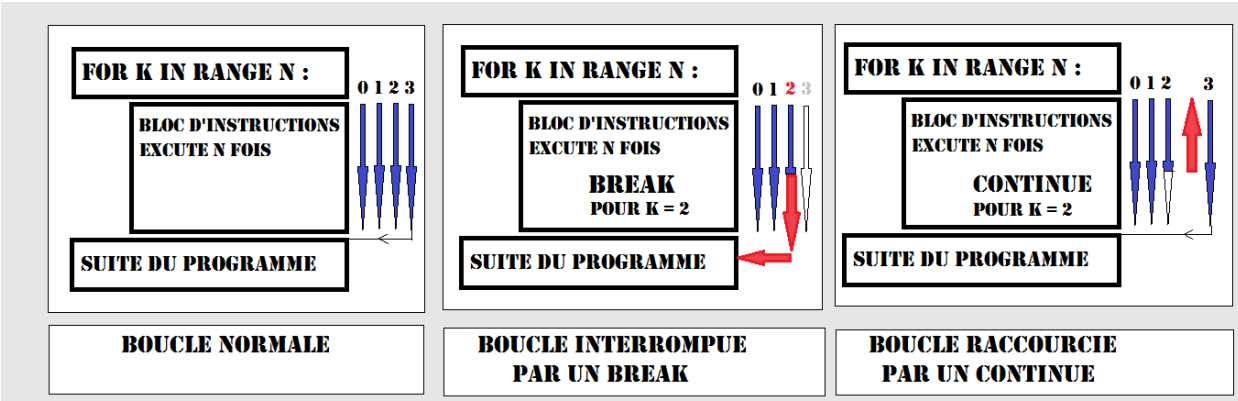
Mais on peut insérer un `break` dans la suite d'instructions (en général dans un test, sinon, c'est idiot, vous devriez comprendre pourquoi).

Quand le compilateur arrive sur le `break`, il stoppe la boucle, et passe immédiatement à la suite du programme.

`for k in range(10**1000) :`
`....print(k)`
`....break`
`....print('Ceci sera-t-il imprimé ?')`
`print('Fini')`
	`F = 1`
	`for k in range(1,10**10) :`
	`....F *= k`
	`....if '010101' in str(F) :`
	`........print(k)`
	`........break`
	`print('Fini')`
		`while True :`
		`....instruction`
		`....if condition :`
		`........break`
		`suite`

Pas grave, on sort dès `k=0` !
Mais quand même, l'impératif « `for k in range (10**1000)` » force quand même Python a créer en memoire le `range(10**1000)` qui prend beaucoup de place même si on n'exécute rien au final.
	Quand F (c'est à dire $k!$ va contenir le motif `'010101`, on va afficher k et sortir. Si il n'y a pas eu de solution avant $k = 10^{10}$, on va s'arrêter.
	On aurait pu faire une boucle `while`.
	Sauf qu'elle risquait de moulinner longtemps, ici on stoppe de toutes façons à 10^{10}.
		Ne serait ce pas finalement une boucle du type
		`repeat instruction`
		`.....instruction'`
		`until condition` » ?
		La question est « serait-elle accepté aux concours ?
		L'autre question « serait elle acceptée en option Info ? ».

Le mot clef continue rencontré lors de l'exécution de la boucle ne nous fait pas sortir de la boucle mais passer au contraire au terme suivant dans le range.




```
for k in range(9) :
....print(k)
....if k%3 == 1 :
........print('Coucou')
....print('Bonjour.')
print('Fini')
```

```
0
Bonjour
1
Coucou
Bonjour
2
Bonjour
3
Bonjour
4
Coucou
Bonjour
5
Bonjour
6
Bonjour
7
Coucou
Bonjour
8
Bonjour
Fini
```

```
for k in range(9) :
....print(k)
....if k%3 == 1 :
........print('Coucou')
........break
....print('Bonjour !')
print('Fini')
```

```
0
Bonjour !
1
Coucou
Fini
```

```
for k in range(9) :
....print(k)
....if k%3 == 1 :
........print('Coucou')
........continue
....print('Bonjour ?')
print('Fini')
```

```
0
Bonjour ?
1
Coucou
2
Bonjour ?
3
Bonjour ?
4
Coucou
5
Bonjour ?
6
Bonjour ?
7
Coucou
8
Bonjour ?
Fini
```

Un programme fait exactement ce qu'on lui dit de faire. Et pas ce qu'on se dit qu'il fera quand on lit le script trop vite sans réfléchir.

Question : et quand il y a des boucles imbriquées
`comme for k in range n :`
`....for in in range(k) :`
un `break` vous fera sortir de quelle boucle ?

```
for k in range(16) :
....for i in range(k) :
........print(k, i)
........if i*k == 10 :
............break
print('Fini')
```

Une boucle impérative for peut être remplacée par une boucle conditionnelle while.
Il suffit d'initialiser la variable de parcours et de la faire augmenter pas à pas jusqu'à ce qu'elle atteigne/dépasse la valeur finale.
Méthode : pour bien savoir quoi mettre dans la condition while, pensez que c'est sa négation qui vous inté-

resse, et ensuite, travaillez avec les loi de Morgan (la négation du et est le ou, et vice versa...).
Attention toutefois à l'ordre des instructions. Complétez les scripts suivants pour qu'ils affichent tous les entiers 3, 4, 5, 6, 7, 8, 9, 10, 11.

```
k = ...
while k <= ....
....k += 1
....print(k)
print('Fini')
```

```
k = ...
while k <= ....
....print(k)
....k += 1
print('Fini')
```

```
k = ...
while k < ....
....print(k)
....k += 1
print('Fini')
```

Écrivez des scripts qui prennent en entrée n et k et retournent le nombre d'arrangements A_n^k égal à $\frac{n!}{(n-k)!}$
c'est à dire $n.(n-1)\dots(n-k+1)$ formé de k termes.
L'un en boucle impérative `for`, l'autre en boucle conditionnelle `while`.

En revanche, une boucle conditionnelle ne peut pas être remplacée par une boucle impérative, puisqu'on ne sait pas à l'avance combien de fois il faudra l'exécuter.
C'est le cas avec l'algorithme d'Euclide entre deux entiers, avec le découpage d'un entier en la suite de ses chiffres en base 10 (ou autre).

```
def Euclide(a, b) :
....while b != 0 :
........a, b = b, a%b
....return(a)
```
oui, c'est aussi bref que ça.

```
def Chiffres(n) :
....L = [ ]
....while n > 0 :
........L.append(n%10)
........n = n//10
....return(L)
```

```
def ChiffresSecure(n) :
....L = [ ]
....nn = n
....while nn > 0 :
........L.append(nn%10)
........nn = nn//10
....return(L)
```

On note quand même un risque dans l'algorithme d'Euclide. On confie au programme deux entiers a et b, mais ceux ci sont modifies tout au long de la procédure (jusqu'à ce qu'il n'en reste finalement plus qu'un qui soit non nul, le dernier reste non nul du cours de maths.
Mais une fois entré dans le def, a et b sont propres à la procédure. Ce sont deux nombres que la procédure utilise pour son propre travail. Mais ils restent invisible du reste du programme (variables locales). Et à la sortie, les deux nombres confies sont restes les mêmes.
Vérifions :

```
A, B = 543, 76
d = Euclide(A, B)
print(A, B, d)
```
573, 76, 1

```
A, B = 543, 76
d = Euclide(A, B)
print(a, b, d)
```
name 'a' is not defined

Intéressons nous au script `Chiffres()`. De la même façon, on lui confie un entier n, et celui ci fond petit à petit avec les n=n//10.
En toute rigueur, il faudrait travailler sur une variable tampon, comme le fait le programme `ChiffresSecure()` de la dernière colonne.
Sinon, que fait `Chiffres` ? Il crée une liste L qui va grossir peu à peu par des `append`.
Et tant que n est non nul, il prend son chiffre des unités `n%10`
le colle au bout de la liste `L`
on l'efface par `n=n//10` (division euclidienne)
Pour comprendre, observons pas à pas l'exécution pour n = 6574 par exemple :

ligne exécutée	test	L	n
`L = [ ]`		[ ]	6574
`while n > 0 :`	True	[ ]	6574
`....L.append(n%10)`		[4]	6574
`....n=n//10`		[4]	657
`while n>0 :`	True	[4]	657
`....L.append(n%10)`		[4, 7]	657
`....n=n//10`		[4, 7]	65
`while n>0 :`	True	[4, 7]	65
`....L.append(n%10)`		[4, 7, 5]	65
`....n=n//10`		[4, 7, 5]	6
`while n>0 :`	True	[4, 7, 5]	6
`....L.append(n%10)`		[4, 7, 5, 6]	6
`....n=n//10`		[4, 7, 5, 6]	0
`while n>0 :`	False	[4, 7, 5, 6]	0
`return(L)`		[4, 7, 5, 6]	

mais

ligne exécutée	test	L	n
`L = [ ]`		[ ]	6574
`while n > 0 :`	True	[ ]	6574
`....L.append(n%10)`		[4]	6574
`....n=n/10`		[4]	657.4
`while n>0 :`	True	[4]	657.4
`....L.append(n%10)`		[4, 7.4]	657.4
`....n=n/10`		[4, 7.4]	65.74
`while n>0 :`	True	[4, 7.4]	65.74
`....L.append(n%10)`		[4, 7.4, 5.74]	65.74
`....n=n/10`		[4, 7.4, 5.74]	6.574
`while n>0 :`	True	[4, 7.4, 5.74]	6.574
`....L.append(n%10)`		[4, 7.4, 5.74, 6.574]	6.574
`....n=n/10`		[4, 7.4, 5.74, 6.574]	0.6574
`while n>0 :`	True	[4, 7.4, 5.74, 6.574]	0.6574

La colonne de droite est là pour vous montrer l'effet déplorable d'une erreur de frappe où la division est décimale et pas euclidienne..

```
def ChiffresRaté(n) :
....L = [ ]
....while n > 0 :
........L.append(n%10)
........n = n/10
....return(L)
```

```
def Frechi(n) :
....L = [ ]
....while n > 0 :
........L = [n%10]+L
........n = n//10
....return(L)
```

```
def Chiffre(n) :
....L = [ ]
....while n > 0 :
........n = n//10
........L.append(n%10)
....return(L)
```

Quelle est l'utilité du script du milieu ?
quel est le défaut du script de droite ?
quel est le risque de tous ces scripts si on entre un n négatif ? Comment pouvez vous résoudre ce problème ?

Maintenant,ce script est il propre à la base 10 ?
Si vous remplacez 10 par un antre entier, que constatez vous ? Que vous récupérez les restes de divisions euclidiennes successives.
La ligne n = n//base dira « dans n combien de fois base »
La ligne n%base dira « et il reste combien ? »

Reprenons 6574 et cette fois, remplaçons 10 par 7

division	reste	L	quotient
$6574 = 939 \times 7 + 1$	1	$[1]$	639
$939 = 134 \times 7 + 1$	1	$[1, 1]$	134
$134 = 19 \times 7 + 1$	1	$[1, 1, 1]$	13
$19 = 2 \times 7 + 5$	5	$[1, 1, 1, 5]$	2
$2 = 0 \times 7 + 2$	2	$[1, 1, 1, 5, 2]$	0

et on confirme : $6574 = 2 + 5 \times 7 + 1 \times 7^2 + 1 \times 7^3 + 1 \times 7^4$ (proprement $1 + 7 \times (1 + 7 \times (1 + 7 \times (5 + 7 \times (2))))$ que l'on développe [2]) et on écrit $6574 = \overline{25111}_{base\ 7}$.
Et on devrait écrire $\overline{6574}_{base\ 10} = \overline{25111}_{base\ 7}$. Mais on ne prend pas la peine de préciser que l'écriture usuelle des nombres est « en base 10 ».
On crée donc le programme et ses variantes

2. présentation dite de Hörner

`def Chiffres(n) :`
`....L = [ ]`
`....while n > 0 :`
`........L.append(n%10)`
`........n = n//10`
`....return(L)`
	`def Chiffres2(n, base) :`
	`....L = [ ]`
	`....while n > 0 :`
	`........L.append(n%base)`
	`........n = n//base`
	`....return(L)`
		`def Chiffre3(n, base =10) :`
		`....L = [ ]`
		`....while n > 0 :`
		`........n = n//base`
		`........L.append(n%base)`
		`....return(L)`

`Chiffres(6574)` donne `[4, 7, 5, 6]`
`Chiffres(7815)` donne `[5, 1, 8, 7]`
	`Chiffres2(6574, 7)` donne `[1, 1, 1, 5, 2]`
	`Chiffres2(815, 2)` donne `[1, 1, 1, 1, 0, 1, 0, 0, 1, 1]`
	`Chiffres2(815, 10)` donne `[5, 1, 8]`
		`Chiffres3(6574, 7)` donne `[1, 1, 1, 5, 2]`
		`Chiffres3(815, 2)` donne `[1, 1, 1, 1, 0, 1, 0, 0, 1, 1]`
		`Chiffres3(815)` donne `[5, 1, 8]`

Vous connaissez le premier, on l'utilise simplement en lui donnant l'entier à décomposer.

Le deuxième attend un entier et une base `def Chiffres2(n, base)` ; il décompose l'entier sur la base.

Le dernier attend un entier, et *si possible* une base `def Chiffres2(n, base=10)`.
Si on lui donne deux nombres, il décompose le premier sur la base égale au second.
Mais si on ne lui en donne qu'une ? Alors c'est n et par défaut, `base` prend la valeur `10`. C'est pratique.

En général, on appellera `Chiffres(n)` et on aura la liste des chiffres en base 10.
Et quand on voudra un cas particulier de base 2, 7, 8, 16 ou autre, on appellera `Chiffres3(n, 2)` ou `Chiffres3(n, 7)`...
Question perfide : que se passe-t-il si base est égal à 1, ou à 0, ou négatif ?

A retenir donc : une procédure (introduite par `def Nom() :`) prend en entrée suivant circonstances :

rien	`def Pause() :` `....sleep(1)`

une variable	`def Factorielle(n) :`
	`....F = 1`
	`....for k in range(2, n+1) :`
	`........F *= k`
	`....return(F)`

deux variables	`def Binomial(n, k) :`
	`....B = 1`
	`....for p in range(k) :`
	`........B = (B*(n-k))//(k+1)`
	`....return(B)`

une variable par défaut	`def Message(prenom='Juan') :`
	`....print('Bonne question '+prenom)`
	`....if prenom == 'Dylan' :`
	`........print(« Il faut que j'y réfléchisse »)`

une ou des variables par défaut	`def Chiffre3(n, base =10) :`
	`....L = [ ]`
	`....while n > 0 :`
	`........n = n//base`
	`........L.append(n%base)`
	`....return(L)`

Question : savoir si un entier *n* est premier : boucle while ou boucle for ?
On peut penser aux deux et même trois.

En boucle impérative for, on prend les entiers plus petits que n et on regarde si ils divisent n.
En fait, on compte les diviseurs.

```
def Premier(n) :
....NbDiv = 0
....for k in range(2, n) :
........NbDiv +=
int(n%k==0)
....if NbDiv>0 :
........return(False)
....else :
........return(True)
```

```
def Premier(n) :
....NbDiv = 0
....for k in range(2, n) :
........NbDiv +=
int(n%k==0)
....if NbDiv>0 :
........return(False)
....return(True)
```

```
def Premier(n) :
....NbDiv = 0
....for k in range(2, n) :
........NbDiv +=
int(n%k==0)
....return(NbDiv==0)
```

La première solution est celle qui sembla « naturelle » à bien des élèves. On teste si le nombre de diviseurs vaut 0 (en effet, on n'a pas mis 1 ni n dans la liste des diviseurs). Et on répond donc `True` ou `False`.
Dans la seconde, si on n'est pas sortit par le `False`, alors on peut répondre `True`. Le `else` est bien inutile.
dans la troisième (vraie programmation), on retourne directement le booléen à évaluer.

Mais on peut optimiser ce script en sachant que les diviseurs propres (donc autres que n) sont plus petits que $\sqrt{n}$.
Par exemple, pour n égal à 1000001, il suffit de chercher jusqu'à $k = 1000$. Si n a un diviseur k plus grand que $\sqrt{n}$, on a déjà du détecter l'autre diviseur $\frac{n}{k}$ plus petit que $\sqrt{n}$.
C'est donc du gaspillage d'aller chercher sur une liste beaucoup plus longue

Mais il faut avoir importé `sqrt` du module `math`.
Et on peut se poser la question du temps perdu à calculer cette racine carrée.
Et là, tout dépend du module `math` et de sa façon de pratiquer.

```
def Premier(n) :
....NbDiv = 0
....for k in range(2, int(sqrt(n))+2) :
........NbDiv += int(n%k==0)
....return(NbDiv==0)
```

Mais si on a besoin des N premiers nombres premiers ?
C'est un peu idiot, si on connaît la liste P des nombres premiers de 2 à 700 de s'interroger sur la primalité de 701 en testant tous les diviseurs possibles. En effet, si 701 n'est pas premier, alors il a un diviseur dans la liste P.
Donc, pour chaque nouveau nombre, on va juste se demander « a-t-il un diviseur dans la liste déjà construite ? ».
Cette fois, par exemple pour 701 on peut tester 699 diviseurs si on est lourd
26 diviseurs si on s'arrête à $\sqrt{701}$
quelques nombres premiers si on en a la liste