

Cours n°3

Python : Fonctions

mercredi 7 octobre

deux problèmes

→ pour changer les valeurs: il fallait à chaque fois modifier les variables et relancer l'évaluation du fichier avec RUN (ou méthode équivalente avec d'autres outils)

→ même si les variables sont données par l'utilisateur grâce à input, le lancement du programme ne pouvait se faire qu'avec "RUN"

exemple: compter le nombre de 'e' dans différentes chaînes

plutôt que de faire ch=" ertyr" puis RUN puis ch="bonjour" puis RUN.....

on voudrait pouvoir compter n'importe quel caractère dans n'importe quelle chaîne sans repasser par la case run!



On peut “bricoler” des solutions..... mais nous allons voir ce qu'est une fonction, comment écrire de nouvelles fonctions et gérer nos programmes en les utilisant.

On va pouvoir écrire une instruction du style
`compte(x,ch)`
à appeler quand on en a besoin dans l'interpréteur ou dans un autre programme

Créer une fonction

Syntaxe:

```
def nom_fonction(arg1, arg2,...):  
    bloc d'instructions
```

arg1, arg2... sont appelés les paramètres (ou arguments) de la fonction

une première fonction

```
def aire_rect (long, larg):  
    print (long*larg)
```

```
>>> aire_rect(10,4)  
40
```

```
>>> aire_rect (10,10)  
100
```

Exemple de fonction

Pour compter les caractères c dans la chaîne ch

```
def compte(c, ch):  
    cp=0  
    for x in ch:  
        if c==x:  
            cp+=1  
    print(cp)
```

Avec une utilisation très simple:

```
>>> compte('o', 'python est un langage formidable')  
2  
>>> compte('e', "c'est bientôt le printemps")  
4
```

Fonction sans argument

On peut aussi faire une fonction sans argument si on veut demander les arguments à l'utilisateur:

```
def prog():  
    c=input("caractère à compter ")  
    ch=input ("dans quelle chaîne ? ")  
    compte(c, ch)
```

```
>>> prog()  
caractère à compter o  
dans quelle chaîne ? j'aime trop python  
2
```


Fonction sans argument

On peut alors lancer le programme autant de fois que l'on veut avec `prog()` à chaque fois

Attention remarque importante:
Il faut bien différencier `prog` et `prog()`:
`prog` est une fonction
`prog()` effectue l'appel de cette fonction

```
>>> prog
<function prog at 0x22b2b78>
>>> prog()
caractère à compter o
dans quelle chaîne ? bonjour
2
```


remarques

- Une fonction peut contenir un nombre quelconque de lignes, contenir une ou plusieurs boucles ou non, des tests ou non....
- Il n'y a pas de déclaration de type
- attention à bien mettre le nombre d'arguments prévus et n'oubliez pas de lire les messages d'erreur!!! ce n'est pas forcément le programme qui est faux, cela peut être l'appel

Attention : valeurs “retournées”

Reprenons toujours l'exemple de l'aire de notre rectangle: il semblerait logique de pouvoir ajouter des aires ou des valeurs à une aire déjà calculée:

```
>>> x=aire_rect(23,12)
```

```
276
```

```
>>> x+3
```

Traceback (most recent call last):

File "<pyshell#28>", line 1, in -toplevel-

x+3

TypeError: unsupported operand type(s) for +:
'NoneType' and 'int'

On obtient un message d'erreur

Que s'est-il passé?

En fait, x n'a pas de valeur car dans la fonction que l'on a écrite, la dernière instruction est un affichage avec print:

```
>>> x= aire(10,12)
```

```
>>> print x
```

```
None
```

```
>>> x
```

x a donc la valeur **none**

(qui correspond en gros à “pas de valeur”)

→ si on ne met pas print?

c'est pire pas de résultat (qui est pourtant calculé au passage!)

```
def aire2 (larg, long):
```

```
    larg*long
```

```
>>> aire (5,8)
```

```
>>>
```

→ Il faut utiliser ***return*** pour “renvoyer” une valeur

```
def aire (larg, long):
```

```
    return larg*long
```

```
>>> aire (5,8)
```

```
40
```

```
>>> x= aire (5,8)
```

```
>>> x
```

```
40
```

return / print

- print : réalise l'affichage et renvoie la valeur none (peut se mettre à n'importe quel moment d'un programme et plusieurs fois)
- return permet de renvoyer une valeur à la fin d'une fonction
- Attention: **remarque très importante:**
return provoque automatiquement une sortie de la boucle en cours

exemple

Revenons sur la somme des cubes

```
def sommeCube(n):  
    s=0  
    for k in range(n+1):  
        s+=k**3  
    return s    # même programme sauf print(s)
```

Et donc pas de problème pour additionner les résultats de deux appels

```
>>> sommeCube(3)+sommeCube(4)  
136
```

attention

Que se passe-t-il si on écrit:

```
def sommeCube(n):  
    s=0  
    for k in range(n+1):  
        s+=k**3  
    return s
```

Et que l'on évalue

```
>>> sommeCube(5)
```


attention

Le programme va renvoyer 0

En effet à la première itération $k=0$ donc $s=0$
Comme il y a un return le script renvoie s
(et donc la boucle s'arrête)

Si on veut que la boucle continue le reurn doit être
après la boucle

- Exemples pour un tuple:

faire la somme des éléments d'un tuple de nombres

```
def somme(t):
```

```
    s=0
```

```
    for x in t:
```

```
        s+=x
```

```
    return s
```

Quel va être l'appel?

- >>> x=(1,2,3)

```
>>> somme(x)
```

```
6
```

```
>>> somme((1,2,3))
```

```
6
```

Attention:

Que fait

```
>>> somme(1,2,3)
```

Que fait `somme(1,2,3)`? Une erreur....

```
>>> somme((1,2,3))
```

```
6
```

```
>>> somme(1,2,3)
```

```
Traceback (most recent call last):
```

```
  File "<pyshell#1>", line 1, in <module>
```

```
    somme(1,2,3)
```

```
TypeError: somme() takes exactly 1 positional argument (3 given)
```

```
>>> |
```

Return permet d'arrêter une boucle avant la fin de la séquence

C'est évidemment très utile de ne pas finir le parcours quand on a déjà atteint le résultat souhaité.

Par exemple, on veut vérifier qu'une chaîne ne contient que des 0 et des 1.

Inutile de parcourir toute la chaîne à tous les coups. Dès qu'on rencontre un caractère qui n'est ni "1" ni "0" alors on peut s'arrêter de parcourir la chaîne et il faut renvoyer False.

Si on a fini le parcours sans s'arrêter en cours de ce parcours, c'est forcément qu'il n'y a que des "0" et des "1" donc on peut renvoyer True.

On gagne forcément en efficacité!

D'où le programme:

```
def verifie(ch):  
    for x in ch:  
        if x!='1' and x!='0':  
            return False # le programme s'arrête  
    return True # parce que la boucle est allée jusqu'au bout
```

```
>>> verifie("012011111")  
False  
>>> verifie("01111001101")  
True
```

Et si on écrit:

```
def verifie(ch):  
    for x in ch:  
        if x!='1' and x!='0':  
            return False  
    return True
```

Rappel programme précédent:

```
def verifie(ch):  
    for x in ch:  
        if x!='1' and x!='0':  
            return False  
    return True
```

Seul le premier caractère sera testé.

Attention à la place du return.....

Et si on commence le programme par x==1:

```
def verifie(ch):  
    for x in ch:  
        if x=='1' or x=='0':  
            #ici il faudrait continuer la boucle  
        else:  
            return False # là on s'arrête  
    return True # si on ne s'est pas arrêté avant.
```

Pour “continuer” on dispose de l'instruction pass qui ne fait rien et qui peut être utile quand on a besoin d'écrire un bloc qui ne fait rien

D'où l'écriture:

```
def verifie(ch):  
    for x in ch:  
        if x=='1' or x=='0':  
            pass  
        else:  
            return False # là on s'arrête  
    return True # si on ne s'est pas arrêté avant.
```

modularité=utilisation de “fonctions auxilliaires”

Un grand intérêt des fonctions est de pouvoir découper les tâches en plus petites tâches: on peut définir des fonctions auxilliaires pour simplifier l'écriture d'autres fonctions.

Exemple: pour chercher le plus grand parmi 4 nombres, on peut commencer par chercher le plus grand de deux nombres.

```
def max2 (x,y):  
    if x< y:  
        return y  
    else:  
        return x
```

```
def max4(x,y,z,t):  
    return max2(max2 (x,y), max2(z,t))
```

Exemple d'utilisation de "sous-fonctions"

Si on veut par exemple compter les voyelles d'un texte, il peut être très efficace de faire une fonction qui teste si un caractère est une voyelle:

```
def voyelle1(c):  
    if c=="a" or c=="e" or c=="i" or c=="o" or c=="y" or c=="u":  
        return True  
    else:  
        return False
```

Fonction qui peut s'écrire plus simplement:

```
def voyelle(c):  
    return c=="a" or c=="e" or c=="i" or c=="o" or c=="y" or  
c=="u"
```

Exemple d'utilisation de "sous-fonctions"

Ou si on connaît les tuples

```
def voyelle(c):  
    return c in ("a","e","i","o","y","u")
```

Remarque: y-a-t-il une différence si on utilise:

```
def voyelle(c):  
    return c in "aeiouy"
```

(La réponse est oui! Par exemple si c='ae';
mais dans le cadre d'un test lors d'une boucle
caractère par caractère cela ne change rien)

Exemple d'utilisation de "sous-fonctions"

D'où le programme:

```
def compte_voyelles(texte):  
    cp=0  
    for x in texte:  
        if voyelle(x):  
            cp=cp+1  
    return cp
```

```
>>> compte_voyelles("bonjour tout le monde")  
8
```

Si on veut écrire différemment la fonction voyelle, on n'aura pas besoin de modifier la fonction compte_voyelles

remarque

D'où le programme:

```
def compte_voyelles(texte):  
    cp=0  
    for x in texte:  
        if voyelle(x):  
            cp=cp+1  
    return cp
```

Dans ce programme , ne pas mettre

```
if voyelle(x)==True
```

variables globales/locales

Variables globales:

Ce sont les variables utilisées aux deux premiers cours. Elles sont définies en dehors des fonctions et les fonctions peuvent les utiliser:

```
def aire(r):  
    return pi*r**2 # il faut définir pi avant d'utiliser la fonction  
  
>>> pi=3.1415  
>>> aire (1)  
3.1415  
>>> aire(2)  
12.566
```


On utilise aussi des variables locales:

elles sont définies et accessibles uniquement à l'intérieur d'un programme

```
def sommeCube(n):  
    s=0  
    for k in range(n+1):  
        s+=k**3  
    return s
```

s est une variable locale au programme sommeCube.

On ne peut pas connaître sa valeur en dehors de l'intérieur du programme

```
>>> sommeCube(2)  
9  
>>> s
```

Traceback (most recent call last):

File "<pyshell#17>", line 1, in <module>

s

NameError: name 's' is not defined

Du coup, même si on a plusieurs fonctions utilisant des variables locales de même nom, il n'y aura pas de conflit

```
def sommeCube(n):  
    s=0  
    for k in range(n+1):  
        s+=k**3  
    return s
```

```
def sommeCarre(n):  
    s=0  
    for k in range(n+1):  
        s+=k**2  
    return s
```

```
>>> sommeCube(2)+sommeCarre(2)  
14
```

Pas de conflit entre les deux valeurs de s

fonctions prédéfinies

Il existe beaucoup de fonctions déjà définies.
On peut savoir ce que fait une fonction grâce à help

```
>>> help(pow)
Help on built-in function pow:
```

```
pow(...)
    pow(x, y[, z]) -> number
```

With two arguments, equivalent to $x**y$.
With three arguments, equivalent to
 $(x**y) \% z$,
but may be more efficient (e.g. for longs).

```
>>> pow(4,3,5)
4
```

les modules

En python, certaines fonctions font partie du “pack” de base qui est toujours disponible; on a déjà croisé len, input, print, pow.....

Beaucoup d'autres sont rangées dans des “modules” (ou bibliothèques) qui ne sont pas chargés au lancement de Python. Certains de ces modules sont dans la distribution de base de python, d'autres doivent être installés.

exemple: sin et cos ne sont pas présentes dans le pack de base contrairement à pow. La racine carrée non plus n'est pas dans le chargement de base. Mais ces fonctions sont dans le module math présent dans la distribution de base.

le module math

sin et cos sont dans le module math.

On peut “importer” une fonction:

```
>>> from math import cos
>>> cos(45)  #les angles sont en radians:
0.52532198881772973
>>> from math import pi
>>> pi
3.1415926535897931
>>> cos(pi)
-1.0
```

inconvenient: il faut recommencer pour chaque fonction ou variable importée (mais par contre on ne surcharge pas avec des trucs inutiles donc à faire pour 1 fonction).

On peut donc importer tout un module:

```
>>> import math
>>> cos
Traceback (most recent call last):
  File "<pyshell#61>", line 1, in -toplevel-
    cos
NameError: name 'cos' is not defined
```

En fait, avec ce principe d'importation: il faut mettre `math.f` pour utiliser la fonction `f` du module `math`.

```
>>> math.cos
<built-in function cos>
>>> math.pi
3.1415926535897931
>>> math.cos(math.pi)
-1.0
```

On peut renommer les fonctions que l'on vient d'importer:

```
>>> pi=math.pi
>>> cos = math.cos
>>> cos(pi/3)
0.50000000000000011
```


On peut aussi directement importer toutes les fonctions du module math, plutôt que d'importer le module lui-même:

```
>>> from math import * #l'étoile indique qu'il faut tout importer
>>> sqrt
<built-in function sqrt>
>>> cos
<built-in function cos>
>>> sin
<built-in function sin>
>> pi
3.1415926535897931
>>> cos(pi/3)
0.500000000000000011
```

L'avantage est que les noms sont “gardés”

Inconvénients: si on importe deux modules qui ont des noms en commun....

connaître le contenu d'un module

Pour connaître le contenu d'un module, on peut par exemple l'importer:

```
>>> import(math)
```

puis savoir “ce qu'il y a dedans” grâce à la fonction dir:

```
>>> dir(math)
['__doc__', '__file__', '__name__', 'acos', 'asin', 'atan', 'atan2', 'ceil',
'cos', 'cosh', 'degrees', 'e', 'exp', 'fabs', 'floor', 'fmod', 'frexp', 'hypot',
'ldexp', 'log', 'log10', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt',
'tan', 'tanh']
```

puis utiliser help..... ou aller lire la doc en ligne!!!

Le module random

Revenons un peu sur l'aléatoire dont nous avons eu besoin la semaine dernière: en python, pour gérer tout ce qui est aléatoire, on utilise les fonctions du module random

```
from random import *
```

→ `randint(a,b)`: renvoie aléatoirement un entier compris entre a et b (inclus tous les deux).

Attention: a et b doivent être entiers (négatifs éventuellement)

```
>>> randint(2,75)
```

```
63
```

→ `random()`: renvoie aléatoirement un réel compris entre 0 (inclus) et 1 (exclu)

```
>>> random()
```

Prendre un élément au hasard dans un tuple ou une chaîne de caractères?

```
from random import randint
```

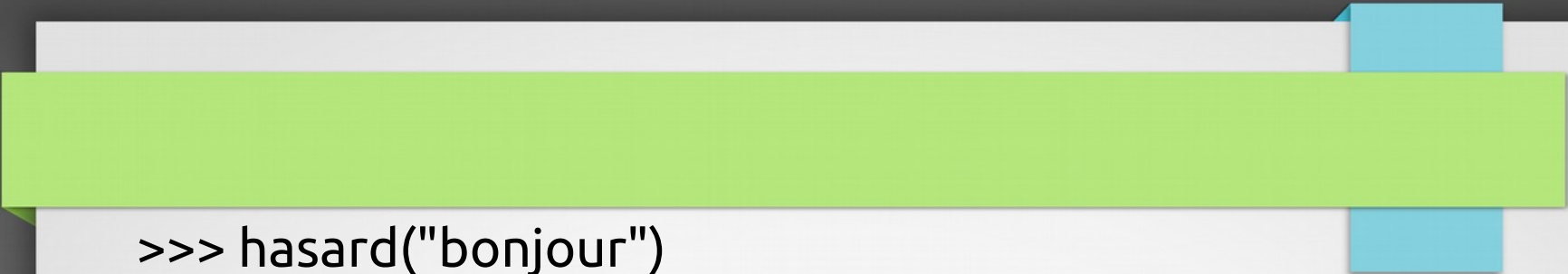
```
def hasard(x):
```

```
# x est une chaîne ou un tuple (pour l'instant)
```

```
    n=len(x)
```

```
    i=randint(0,n-1)
```

```
    return x[i]
```



```
>>> hasard("bonjour")
```

```
'u'
```

```
>>> hasard("bonjour")
```

```
'o'
```

```
>>> hasard(("pomme", "poire", "ananas", "banane", "citron"))
```

```
'pomme'
```

```
>>> hasard(("pomme", "poire", "ananas", "banane", "citron"))
```

```
'poire'
```

```
>>> hasard(("pomme", "poire", "ananas", "banane", "citron"))
```

```
'citron'
```

Ou en utilisant la fonction choice:

```
>>> from random import *
```

```
>>> help(choice)
```

Help on method choice in module random:

choice(seq) method of random.Random instance

Choose a random element from a non-empty sequence.

```
>>> choice(("pomme", "poire", "abricot", "kiwi", "banane"))
```

```
'abricot'
```

```
>>> choice(("pomme", "poire", "abricot", "kiwi", "banane"))
```

```
'kiwi'
```

```
>>> choice("abcdefghijklmnopqrstuvwxyz")
```

```
'o'
```


```
>>> choice("abcdefghijklmnopqrstuvwxyz")
```

```
'm'
```



Il existe encore d'autres fonctions dans le module random mais que nous verrons après avoir vu les listes....

Remarque: l'aléatoire du python est **reproductible**:
C'est à dire que l'on peut avec les mêmes commandes,
obtenir les mêmes valeurs si on fait la même initialisation (avec seed)



```
>>> seed(1)
```

```
>>> randint(1,600)
```

```
81
```

```
>>> randint(1,600)
```

```
509
```

```
>>> randint(1,600)
```

```
459
```

```
>>> randint(1,600)
```

```
154
```

```
>>> seed(1)
```

```
>>> randint(1,600)
```

```
81
```

```
>>> randint(1,600)
```

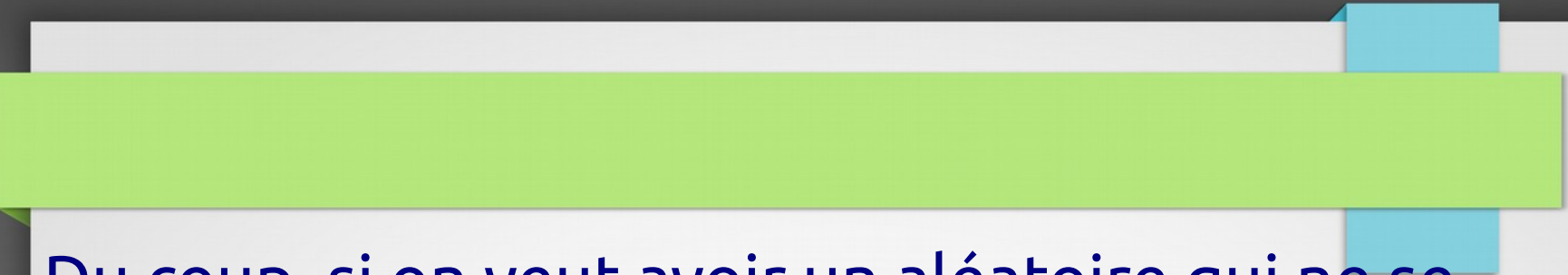
```
509
```

```
>>> randint(1,600)
```

```
459
```

```
>>> randint(1,600)
```

```
154
```

Du coup, si on veut avoir un aléatoire qui ne se reproduira pas à chaque démarrage d'un même jeu, on fera un seed avec un truc "imprévisible", par exemple avec un appel à un temps machine.

Ce temps machine peut être obtenu avec une fonction du module time.....

Autres modules? La liste est ENORME

- * pour des calculs: numpy, scipy ,sympy
- * visualisation: matplotlib
- * gestion temps et date: time, datetime
- * images: pil
- * interface graphique: tkinter, pygame
- * traitements de fichiers: os

et beaucoup d'autres....

Des fonctions particulières : les méthodes

- ce sont des fonctions prédéfinies un peu particulières
- elles sont associées à un “objet” (elles font partie de la définition de ce qu'on appelle la classe d'un objet voir beaucoup plus tard)
- leur utilisation est particulière: si truc est un objet (tuples, chaines, listes) pour lesquels une méthode machin existe alors l'utilisation sera:
`truc.machin(arguments)`

Exemples pour les strings

Si la chaîne `cp` est définie par:

```
cp= "python est un langage informatique puissant"  
cp1= "et très pratique"
```

`cp.count(sch)` compte le nombre d'ocurrences de la sous-chaîne `sch` dans `cp` (renvoie 0 si la sous chaîne n'est pas présente)

```
>>> cp.count("e")
```

```
3
```

```
>>> cp.count("un")
```

```
1
```

```
>>> cp.count("serpent")
```

```
0
```

```
>>> cp1.count("a")
```

```
1
```

`cp.upper ()` renvoie une copie de `cp` où toutes les lettres sont en écrites en majuscules

`cp.lower ()` renvoie une copie de `cp` où toutes les lettres sont écrites en minuscules

`cp.swapcase()` renvoie une copie de `cp` où toutes les majuscules sont converties en minuscules et réciproquement.

```
>>> cp.upper()  
'PYTHON EST UN LANGAGE INFORMATIQUE PUISSANT'
```

`cp.find(sch)` retourne l'index de début de la sous-chaîne `sch` dans `cp` (renvoie -1 si la sous chaîne n'est pas présente)

```
>>> cp.find("python")
```

```
0
```

```
>>> cp.find("est")
```

```
7
```

```
>>> cp.find("serpent")
```

```
-1
```

`cp.index(c)` retourne l'index de la première occurrence du caractère `c` dans la chaîne `cp` (avec une erreur si le caractère `c` n'est pas présent).

```
>>> cp.index('a')
```

```
15
```

```
>>> cp.index("6")
```

```
Traceback (most recent call last):
```

```
File "<pyshell#53>", line 1, in -toplevel-
```

```
cp.index("6")
```

```
ValueError: substring not found
```

Il existe bien d'autres méthodes encore.....

On peut les trouver dans les docs ou

```
>>> help(str)
```


Signalons par exemple

```
isdigit(...)  
S.isdigit() -> bool
```

Return True if all characters in S are digits
and there is at least one character in S, False otherwise.

```
>>> ch="23456"  
>>> ch1="123.34"  
>>> ch2="1o"  
>>> ch3="-2"  
>>> ch1.isdigit()  
False  
>>> ch.isdigit()  
True  
>>> ch2.isdigit()  
False  
>>> ch3.isdigit()  
False  
>>> int(ch3) # ch3 représente quand même un entier  
-2
```


Pour les tuples:

Il y a beaucoup moins de méthodes pour les tuples:
Suite à l'appel de `help(tuple)`, on obtiendra comme méthodes simplement:

```
count(...)
    T.count(value) -> integer -- return number of occurrences of value

index(...)
    T.index(value, [start, [stop]]) -> integer -- return first index of value.
    Raises ValueError if the value is not present.
```

```
>>> t=(2,3,4,5,2,3,3,3)
>>> t.count(3)
4
>>> t.index(3)
1
>>> tt=("p","y","t","h","o","n")
>>> tt.index("o")
4
```

Retour sur le jeu pierre feuille ciseau

On va introduire des fonctions pour pouvoir lancer le jeu quand on veut!!! puis faire quelques tests.

D'abord on transforme notre script en fonction:

La fonction pfc permet de faire une partie de plusieurs coups

```
from random import randint
```

```
def pfc():
```

```
    pts=0
```

```
    nb=int(input ("combien de tours de jeu?"))
```

```
    for i in range(nb):
```

```
        choix= input("votre choix (tapez pierre feuille ou ciseau)")
```

```
        nb_ordi= randint(0,2)
```

```
        choix_ordi=("pierre", "feuille", "ciseau")[nb_ordi]
```

```
        print ("l'ordinateur a choisi "+ choix_ordi)
```

```
        if choix==choix_ordi:
```

```
            print ("ex-aequo")
```

```
        elif (choix== "pierre" and choix_ordi== "ciseau") or (choix ==  
            "ciseau" and choix_ordi== "feuille") or (choix== "feuille"  
            and choix_ordi == "pierre"):
```

```
            print ("le joueur marque 1 point")
```

```
            pts+=1
```

```
        else:
```

```
            print ("le joueur perd un point")
```

```
            pts-=1
```

```
        print ("score actuel vous avez "+ str(pts)+ " point(s)")
```

```
    print ("score final " + str(pts))
```



Notre jeu fonctionne très bien mais a toujours un défaut: la faute de frappe.....

Il faut donc mettre en place un mécanisme de saisie avec tests.....
Il existe une gestion d'erreur en Python (voir plus tard) mais nous pouvons déjà faire des tests satisfaisants avec nos connaissances actuelles.....

Que faut-il tester ici?

Pour la première saisie:

que la chaîne saisie va pouvoir être traduite à l'aide de int

Nous allons d'abord tester si une chaîne de caractères est composée uniquement des caractères "1", "2",, "0"

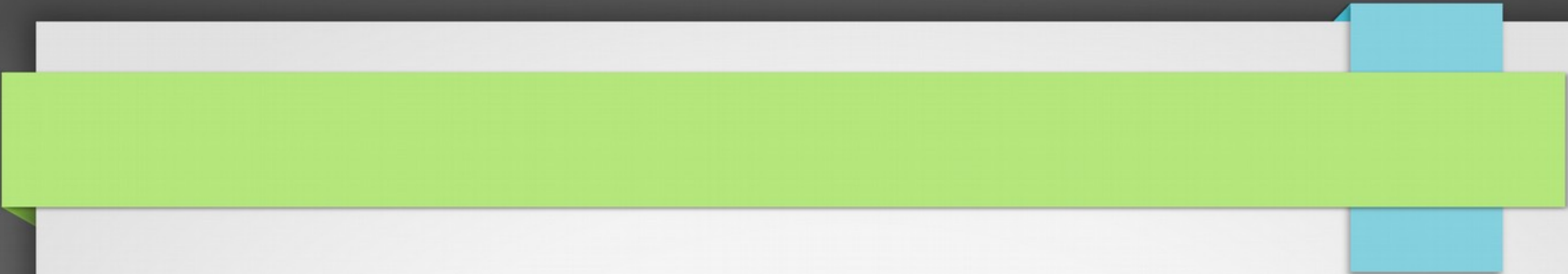
Pour tester si une saisie est bien un entier (on aurait pu utiliser `isdigit`)

```
# test pour savoir si un caractère est un chiffre
def chiffre(x):
    return x in "0123456789"
```

```
Ou def chiffre(x):
    return x in ('0','1','2','3','4','5','6','7','8','9')
```

```
# test pour savoir si la chaîne n'est composée que de chiffres
def nombreEntier(ch):
    for x in ch:
        if not(chiffre(x)):
            return False
    return True
```

```
# et la saisie avec tests:
def saisie(message):
    ch=input(message)
    if nombreEntier(ch): # on pourrait mettre ch.isdigit()
        return int(ch)
    else:
        print("erreur de saisie")
        return saisie(message) # on relance l'appel de la fonction
```



```
>>> pfc()
combien de tours de jeu? 1
erreur de saisie
combien de tours de jeu? 2
votre choix (tapez pierre feuille ou ciseau) pierre
l'ordinateur a choisi ciseau
le joueur marque 1 point
score actuel vous avez 1 point(s)
votre choix (tapez pierre feuille ou ciseau)feuille
l'ordinateur a choisi pierre
le joueur marque 1 point
score actuel vous avez 2 point(s)
score final 2
>>>
```


Deuxième source d'erreur : la saisie du coup

On va aussi modifier la saisie du coup joué par le joueur quand il choisit pierre feuille ou ciseau (ici pour l'instant cela ne plante pas mais cela fait un coup "bizarre")

Il suffit de remplacer le choix=input("tapez....") par choix= saisieCoup()

```
def saisieCoup():  
    c=input("votre choix (tapez pierre feuille ou ciseau)")  
    if c in ("pierre", "feuille", "ciseau"):  
        return c  
    else:  
        print("ce coup n'existe pas")  
        return saisieCoup()
```



```
def pfc():
    pts=0
    nb=saisie("combien de tours de jeu?")
    for i in range(nb):
        choix= saisieCoup()
        nb_ordi= randint(0,2)
        choix_ordi=("pierre", "feuille", "ciseau")[nb_ordi]
        print ("l'ordinateur a choisi "+ choix_ordi)
        if choix==choix_ordi:
            print ("ex-aequo")
        elif (choix== "pierre" and choix_ordi== "ciseau") or (choix == "ciseau"
and choix_ordi== "feuille") or (choix== "feuille" and choix_ordi == "pierre"):
            print ("le joueur marque 1 point")
            pts+=1
        else:
            print ("le joueur perd un point")
            pts-=1
        print ("score actuel vous avez "+ str(pts)+ " point(s)")
    print ("score final " + str(pts))
```