```python
#!coding: utf-8
from __future__ import print_function, division

######################
# Base CINQ : Objets #
######################

# un objet permet de grouper plusieurs variables dans un... objet
# par exemple on peut grouper la "vie" et le "mana" dans un objet "personnage"

# D'abord on définit la classe de l'objet
# pour l'instant elle est vide mais utilisable
# une classe va définir le "modèle" d'une série d'objet
# par exemple tous les personnages seront des... personnages

class Personnage:
    pass # vide

# Créer un objet... Appeler sa classe !
# Vu que la classe est vide, il n'y a pas de paramètres

bob = Personnage()
print(bob) # "objet de la classe Personnage à l'adresse mémoire 0x..."
# bob est un personnage, oui oui
# on peut lui mettre des variables !
bob.vie = 60
bob.max_vie = 100
bob.mana = 10

# d'autres persos

alice = Personnage()
alice.vie = 40
alice.max_vie = 70
alice.mana = 50

# imaginez une liste de personnage !
les_personnages = [alice, bob]

p = Personnage()
p.vie = 20
p.max_vie = 20
p.mana = 30

les_personnages.append(p)

p = Personnage()
p.vie = 25
p.max_vie = 25
p.mana = 20

les_personnages.append(p)

# lancez cet exemple sur python tutor, et observez la dynamique des flèches lors du deuxième p
=
# plus d'infos sur ces flèches dans la section "En savoir plus"

# essayez de créer une liste de 100 personnages différents en moins de 10 lignes de code
# remarquez qu'un personnage peut être différent d'un autre même s'il a les mêmes stats

########################
# Base SIX : Fonctions #
########################

# Certaines structures de code reviennent souvent
# Par exemple dans un exercice précédent, on a calculé le maximum d'une liste

ma_liste = [1,2,7,2]

m = ma_liste[0]
```

```python
i = 0
while i < len(ma_liste):
    if ma_liste[i] > m:
        m = ma_liste[i]
    i = i + 1
print(m)

# Ou encore, on aimerait donner 20 points de vie à un objet Personnage
# sans dépasser son maximum de vie

bob.vie = bob.vie + 20
if bob.vie > bob.max_vie:
    bob.vie = bob.max_vie

# cependant, on aimerait bien mettre ça dans une "boite"
# une "boite à calculer le maximum d'une liste"
# et pouvoir la réutiliser autant de fois que l'on veut

# nous allons donc faire une FONCTION

def calculer_maximum(la_liste):
    """
    Calcule le maximum d'une liste

    1 Paramètre d'entrée :
        - la_liste : la liste
    1 Paramètre de retour :
        - le maximum
    """
    m = la_liste[0]
    i = 0
    while i < len(la_liste):
        if la_liste[i] > m:
            m = la_liste[i]
        i = i + 1

    return m

# on a fait une fonction, avec un paramètre (la liste), et une valeur de retour (le maximum)
# et un petit texte descriptif décrivant ce que fait la fonction (docstring)
# maintenant on peut l'appeler

une_belle_liste = [1,2,7,2]
a = calculer_maximum(une_belle_liste)

une_autre_liste = [8,0,1,6]
b = calculer_maximum(une_autre_liste)

# pour appeler une fonction on...
# écrit son nom,
# ouvre la parenthèse,
# met les paramètres séparés par des virgules,
# ferme la parenthèse

# et la valeur de retour peut être stockée dans une variable...
b = calculer_maximum(une_autre_liste)

# ou utilisée dans une expression
c = calculer_maximum(une_belle_liste) + calculer_maximum(une_autre_liste)

# lancez cet exemple dans python tutor !

# Les objets et fonctions vont bien ensemble,
# quand une fonction a accès à un objet ellepeut modifier ses attributs

def donner_potion(perso, montant):
    """ Donne "montant" pv à un perso "perso" sans dépasser son maximum de vie """
    perso.vie = perso.vie + montant
    if perso.vie > perso.max_vie:
        perso.vie = perso.max_vie

# on appelle la fonction avec nos personnages de la base CINQ

donner_potion(bob, 25) # 25 points de vie pour bob
```

```python
donner_potion(alice, 100) # 100 points de vie pour alice

# quand une fonction est appelée, python...
# - crée un bloc dans la mémoire "de gauche"
# - copie les paramètres comme si on avait fait "="
# - exécute le code de la fonction, qui peut faire tout ce qu'il veut
# - supprime le bloc quand on est à la fin

# remarquez que la variable "i" et "m" sont "locales" à la fonction
# elles sont crées dans la fonction et détruites à la fin

# EN UN MOT : une fonction, c'est une "boi-boite"
# elle a des paramètres d'entrée... ou pas
# ... fait un calcul ...
# et sort des valeurs de retour ... ou pas

def f(x,y):
    return x + 2 * y

# super dessin de la "boi-boite"
#          +---+
# x,y ---> | f | ---> z
#          +---+

# les fonctions peuvent avoir
# 0 ou plus paramètres d'entrées ("paramètres" ou "arguments")
# 0 ou plus paramètres de sortie ("valeur(s) de retour")

# Voici plusieurs exemples de fonctions

# 0 paramètre d'entrée, 0 de retour
def afficher_description():
    print("+------------------+")
    print("| Hello !          |")
    print("| Je suis Robert ! |")
    print("+------------------+")

afficher_description()

# 1 paramètre d'entrée, 0 de retour
def afficher_etoiles(nombre):
    for i in range(nombre):
        print("*")

afficher_etoiles(5)

# 2 paramètres d'entrée, 2 de retour
def minute_suivante(h,m):
    if m < 59:
        nouveau_h = h
        nouveau_m = m + 1
    else:
        nouveau_h = h + 1
        nouveau_m = 0
    return nouveau_h, nouveau_m

a,b = minute_suivante(13,30)
c,d = minute_suivante(12,59)

# pas d'entrée, un retour
def generate():
    liste = []
    for i in range(5):
        liste.append(i * i)
    return liste

l = generate()

# remarquez qu'on peut mettre autant de return que l'on veut
# quand python lit un return, il arrête tout de suite la fonction et renvoie (ou non) une
valeur

def yo(x):
    if x < 0:
```

```
        return -1

    i = 0
    s = 0
    while i < x:
        s += i
        i = i + 1
    return s

# ici, si x est négatif, la fonction s'arrête et renvoie -1

#########################
# Pour en en savoir plus #
#########################

### fonctions ###

## simplifications automatiques

# avant:
def f():
    A
    if COND:
        return X
    else:
        return Y

# après (if fonctionnel)
def f():
    A
    return X if CONDITION else Y

# avant
def f():
    ...
    if CONDITION:
        return True
    else:
        return False

# après (bool)
def f():
    ...
    return CONDITION

# avant
def f():
    if CONDITION:
        A
        return X
    else:
        B
        return Y

# après (return)
def f():
    if CONDITION:
        A
        return X
    B
    return Y
# conseil: faire ceci si il y a peut d'instructions dans "A" et beaucoup dans "B"

# avant
def f():
    ...
    for X in L:
        if CONDITION:
            return True
    return False

# après : (voir fichier progra_functionals)
def f():
    ...
```

```
    return any(CONDITION for X in L)
# lire ça comme "il existe un X dans L qui vérifie CONDITION"
# exemple: any(x == 2 for x in L) : il existe un "2" dans L
# ... ou "il existe un X dans L tel que CONDITION"
# ... any(x == 2 for x in L): il existe un x dans L tel que x == 2
# ... ∃ x ∈ L: x == 2

# conseil: passer à la ligne en fonction de la longueur dans le "any" !
# ... si c'est return False, puis return True : return not any(...)

# avant
def f():
    ...
    for X in L:
        if not CONDITION:
            return False
    return True

# après : (voir fichier progra_functionals)
def f():
    ...
    return all(CONDITION for X in L)
# lire ça comme "tous les X de L vérifient CONDITION"
# exemple: all(x > 0 for x in L) : tous les nombres de l sont > 0
# ... ou "quel que soit X, il vérifie CONDITION
# ... all(x > 0 for x in L)
# ... ou "pour tous les éléments de X, il vérifie CONDITION"
# ... all(x > 0 for x in L) : pour tous les élements x de L on a x > 0
# ... ∀ x ∈ L: x > 0

## paramètres par défaut

def f(x,y=5):
    return x + y

# attention, ne mettre que des objets "immuable" : ni list ni objets
# un bon défaut est "None" qui est une valeur spéciale

print(f(1))
print(f(1,2))

## appel utilisant les noms des paramètres

print(f(1, 2))
print(f(1, y=1))
print(f(x=5, y=2))

## récursivité

# méditez là dessus avec python tutor..

def f(x):
    if x > 0:
        f(x-1)
        print("Hello", x)
        f(x-1)

f(4)

## variables locales

g = 5
p = 2
def f(x):
    print(g)
    y = x + 1
    p = 5
    print(p)

# LECTURE quand on lit une variable (print x par exemple), python fait ceci :
# - Regarder si la variable existe dans la Fonction en cours (dans le bloc à gauche, en bas sur
pythontutor)
# - Si elle existe, il donne la valeur (contenu de la case), sinon :
# - Il regarde dans l'espace GLOBAL (global frame dans pythontutor), et s'il ne la trouve
```

```
pas...
# - lance une NameError (variable non trouvée)

# ÉCRITURE : quand on écrit dans une variable (variable = ), python fait ceci :
# - Regarder si la variable existe dans la Fonction en cours (dans le bloc à gauche, en bas sur
pythontutor)
# - Si elle existe, il écrit la valeur (modifie le contenu de la case), sinon :
# - Il Crée la variable DANS LA FONCTION, on peut donc avoir une variable globale et locale
avec le même nom

# Quand une fonction se termine (return ou arrivée à la fin du bloc), les variables sont
détruites

# même si c'est une mauvaise pratique, il est possible de dire qu'on parle d'une variable
globale
# cela modifie donc la dynamique lors de l'écriture

g = 5
def f():
    global g
    g = 2

f()
print(g)

# comment éviter de marquer une variable comme "globale" ?
# en RENVOYANT une info

def f():
    return 2

g = f()

# ou en groupant des variables dans des... OBJETS !

### Objets ###

## Méthodes
# On peut mettre des fonctions dans les classes...
# On appelle ça des "méthodes"
# À la place de le faire dans l'espace global, on le fait dans la classe
# Ces fonctions reçoivent TOUJOURS un premier paramètre "self"
# qui représente l'objet manipulé
# par rapport à notre fonction donner_potion(perso, montant)
# le "self", c'est "perso"

class Personnage:
    def boire_potion(self, montant):
        """ Donne montant pv à un perso sans dépasser son maximum de vie """
        self.vie = self.vie + montant
        if self.vie > self.max_vie:
            self.vie = self.max_vie

    def crier(self):
        """ Affiche un cri de guerre à l'écran """
        print("Bouh!")

bob = Personnage()
bob.vie = 60
bob.max_vie = 100
bob.mana = 10

alice = Personnage()
alice.vie = 40
alice.max_vie = 70
alice.mana = 50

bob.boire_potion(25) # on appelle les méthodes comme ceci :)
alice.boire_potion(100)
alice.crier()

## Contructeur

# On voit des parties du code précédent qui peuvent être répétée
```

```python
# Ce serait pratique de faire une fonction qui "initialise" les attributs

def init_personnage(perso, a, b, c):
    perso.vie = a
    perso.max_vie = b
    perso.mana = c

bob = Personnage()
init_personnage(bob, 60, 100, 10)
alice = Personnage()
init_personnage(alice, 60, 100, 10)

# Cette fonctionnalité existe : Le constructeur est une fonction qui est appelée
# à la création de tout objet
# On donne les paramètres à la création

class Personnage:
    def __init__(self, a, b, c):
        self.vie = a
        self.max_vie = b
        self.mana = b

bob = Personnage(60, 100, 10)
alice = Personnage(40, 70, 50)

### Héritage ###

# méditez là dessus
# les fonctions dans une classe sont liées aux... objets !

class Personnage:
    def __init__(self, vie, max_vie=None):
        self.vie = vie
        self.max_vie = 100 if max_vie == None else max_vie

    def boire_potion(self, x):
        self.vie = min(self.vie + x, self.max_vie)

class Guerrier(Personnage): # un Guerrier est un Personnage
    def crier(self): # mais qui fait "crier()" différemment
        print("Arrrrrrrrgggg")

class Magicien(Personnage):
    def crier(self):
        print("Wololo")

alice = Magicien(50)
bob = Guerrier(75)
guy = Personnage(20,80)
personnages = [alice, bob, guy] # différentes classes, mais tous des Personnages
for perso in personnages:
    perso.crier() # va appeler une fonction différente en fonction de la classe
    perso.boire_potion(10) # la fonction boire_potion existe toujours

# Pour réutiliser une fonction de la classe héritée, on fait comme ceci :

class MagicienDouble(Personnage):
    def crier(self):
        Personnage.crier(self) # on appelle la version de "crier" qui se trouve dans
"Personnage"
        print("Wouloulou")

# Comment ça se passe ?
# Python regarde si la classe de l'objet a la fonction appelée
# Si elle ne l'a pas, elle regarde dans sa classe parente
# Et finalement si aucune des classes parentes n'a la fonction, il lance une exception

# on peut savoir si un objet "est" d'une classe

print(isinstance(alice, Magicien)) # True
print(isinstance(bob, Magicien)) # False
print(isinstance(guy, Magicien)) # False
print(isinstance(guy, int)) # False
```

```python
print(isinstance(alice, Personnage)) # True car héritage

# on peut donner plusieurs classes pour éviter de faire un "or"
print(isinstance(alice, Magicien) or isinstance(alice, Guerrier)) # long
print(isinstance(alice, (Magicien, Guerrier))) # raccourci

# bien que peu utile en pratique, on peut accéder à la classe d'un objet
cls = alice.__class__
print(cls == Magicien) # True
print(cls == Personnage) # True
# toujours utiliser "isinstance" quand vous voulez tester l'appartenance
# si votre code fait quelque chose pour un Personnage, ça devrait être
# la même chose pour un Magicien

### Références ###

# méditez là dessus avec pythontutor

a = 5
b = a
a = 6
print(a)
print(b)

a = [1,2,3]
b = a
a[0] = 2
print(a)
print(b)

class Personnage:
    pass

a = Personnage()
b = a
a.vie = 1
print(a.vie)
print(b.vie)

# plot twist : les listes sont des objets, de la classe "list"
# quand on crée un objet (et donc une liste),
# on reçoit une "référence" (symbolisée par une "flèche").
# Contrairement aux objets, les int, float, string n'ont pas de flèche,
# leur valeur est écrite directement dans la case.
# Les objets et références sont dans deux zones de la mémoire différentes,
# symbolisées par deux colonnes dans pythontutor.

# Quand on COPIE une variable, on copie la CASE
# donc quand on copie un int, on copie le nombre
# quand on copie une "flèche", on fait une nouvelle flèche avec la même cible
# une COPIE se passe uniquement avec "=" ou un appel de fonction :)

# Ainsi, dans le code précédent, on a plusieurs exemples de copie avec =

def modifier_premier(liste):
    liste[0] = 0

ma_liste = [1,2,3]
modifier_premier(ma_liste)

def modifier_local(x):
    x = 2

p = 2
modifier_local(p)
```