

Partie III

La programmation orientée objets



# Informatique 1

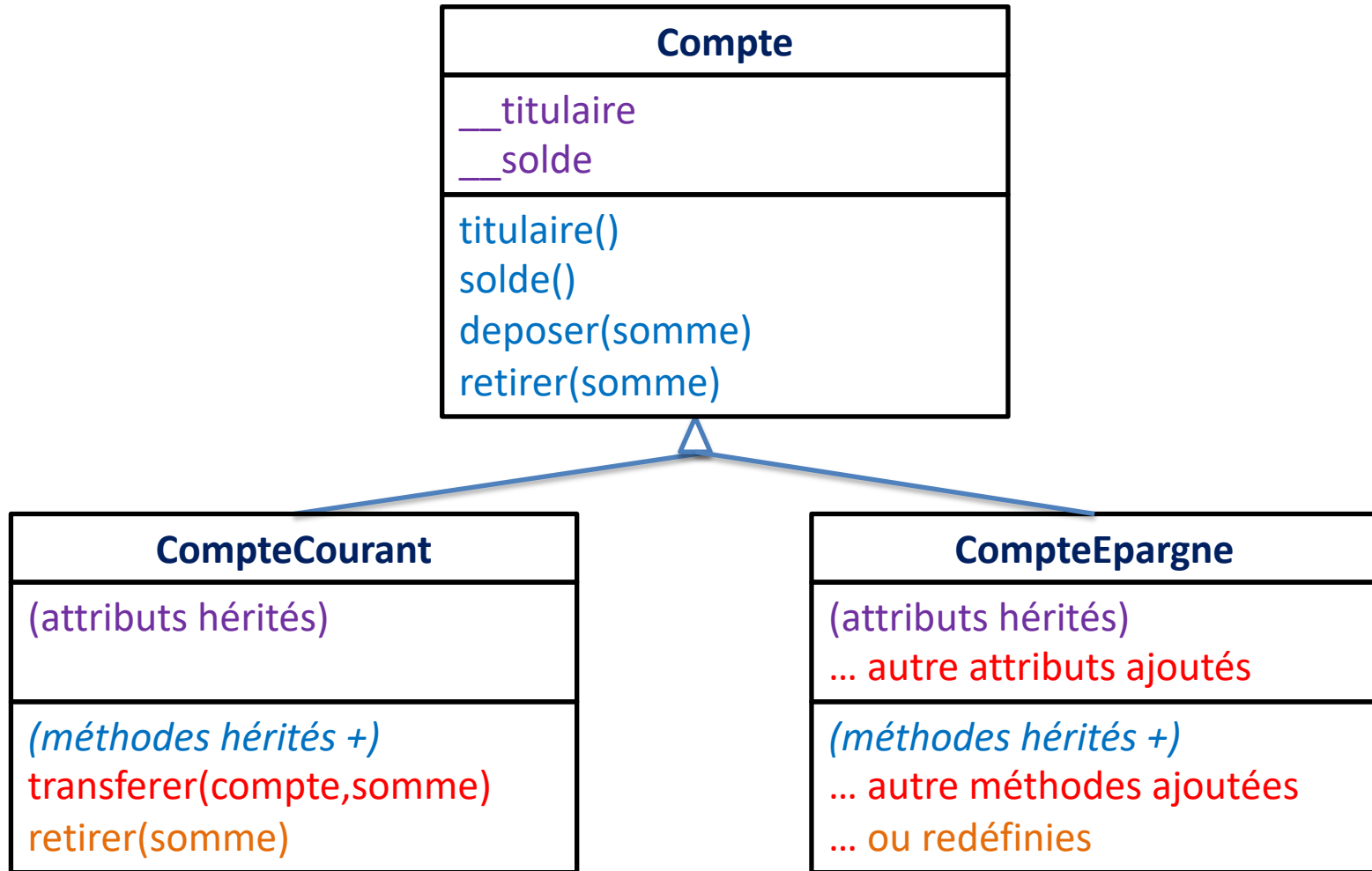
## Introduction à la programmation

### **Mission 9 : RESTRUCTURATION**

Héritage, variables d'instances privées, variables de classes privées,  
méthodes de classe, tests unitaires, la portée des variables

**Kim Mens** – Siegfried Nijssen – Charles Pecheur

# Héritage



# type()

```
class Compte :  
    ...
```

```
class CompteCourant :  
    ...
```

```
class CompteEpargne :  
    ...
```

```
compte_kim = CompteCourant()  
compte_tom = CompteEpargne()
```

```
>>> type(compte_kim)  
<class '__main__.CompteCourant'>
```

**type()** retourne la classe de l'objet passé en paramètre

# type()

```
class Compte :
```

```
...
```

```
class CompteCourant :
```

```
...
```

```
class CompteEpargne :
```

```
...
```

```
compte_kim = CompteCourant()
```

```
compte_tom = CompteEpargne()
```

```
type(compte_kim) == CompteCourant
```

```
#True
```

```
type(compte_tom) == CompteEpargne
```

```
#True
```

**type()** retourne la classe de l'objet passé en paramètre

```
type(compte_kim) == Compte) #False
```

```
type(compte_tom) == Compte) #False
```

# isinstance()

**isinstance()** vérifie si un objet (premier paramètre) est une instance d'une classe (deuxième paramètre) ou d'une classe fille de cette classe

```
class Compte :
```

```
...
```

```
class CompteCourant :
```

```
...
```

```
class CompteEpargne :
```

```
...
```

```
compte_kim = CompteCourant()
```

```
compte_tom = CompteEpargne()
```

```
isinstance(compte_kim,CompteCourant)
```

```
#True
```

```
isinstance(compte_tom,CompteEpargne)
```

```
#True
```

```
isinstance(compte_kim,Compte) #False
```

```
isinstance(compte_tom,Compte) #False
```

# type() vs. isinstance ()

même comportement si pas d'héritage

```
class Compte :
    ...
class CompteCourant :
    ...
class CompteEpargne :
    ...

compte_kim = CompteCourant()
compte_tom = CompteEpargne()

type(compte_kim) == CompteCourant #True
type(compte_tom) == CompteEpargne #True

isinstance(compte_kim,CompteCourant) #True
isinstance(compte_tom,CompteEpargne) #True

type(compte_kim) == Compte #False
type(compte_tom) == Compte #False
isinstance(compte_kim,Compte) #False
isinstance(compte_tom,Compte) #False
```

# type() vs. isinstance ()

mais ... **isinstance** tient compte de l'héritage

```
class Compte :
    ...
class ComptesCourant(Compte) :
    ...
class ComptesEpargne(Compte) :
    ...

compte_kim = ComptesCourant()
compte_tom = ComptesEpargne()

type(compte_kim) == ComptesCourant #True
type(compte_tom) == ComptesEpargne #True

isinstance(compte_kim, ComptesCourant) #True
isinstance(compte_tom, ComptesEpargne) #True

type(compte_kim) == Compte #False
type(compte_tom) == Compte #False
isinstance(compte_kim, Compte) #True
isinstance(compte_tom, Compte) #True
```

# “if” vs. polymorphisme

à éviter: des “if” sur les types des classes filles

```
class Compte :
    def __str__(self) :
        if type(self) == CompteCourant :
            s = "compte courant"
        elif type(self) == CompteEpargne :
            s = "compte d'épargne"
        else :
            s = "compte"
        return s
```

```
class CompteCourant(Compte) :
    ...
class CompteEpargne(Compte) :
    ...
```



```
compte      = Compte()
compte_kim  = CompteCourant()
compte_tom  = CompteEpargne()
print(compte)
# compte
print(compte_kim)
# compte courant
print(compte_tom)
# compte d'épargne
```



# if vs. “polymorphisme”

mieux: éviter des if grâce au “polymorphisme”

```
class Compte :  
    def __str__(self) :  
        return "compte"  
  
class CompteCourant(Compte) :  
    def __str__(self) :  
        return "compte courant"  
  
class CompteEpargne(Compte) :  
    def __str__(self) :  
        return "compte d'épargne"
```



```
compte      = Compte()  
compte_kim  = CompteCourant()  
compte_tom  = CompteEpargne()  
print(compte)  
# compte  
print(compte_kim)  
# compte courant  
print(compte_tom)  
# compte d'épargne
```

# if vs. “polymorphisme”

encore mieux: avec utilisation de super()

```
class Compte :
    def __str__(self) :
        return "compte"

class CompteCourant(Compte) :
    def __str__(self) :
        return super().__str__() \
            + " courant"

class CompteEpargne(Compte) :
    def __str__(self) :
        return super().__str__() \
            + " d'épargne"
```



```
compte = Compte()
compte_kim = CompteCourant()
compte_tom = CompteEpargne()
print(compte)
# compte
print(compte_kim)
# compte courant
print(compte_tom)
# compte d'épargne
```

# Variables d'instance « *privées* » ?

```
class Compte :  
    def __init__(self, titulaire) :  
        self.__titulaire = titulaire  
        self.__solde = 0  
    def solde(self) :  
        return self.__nom
```

```
>>> a = Compte("kim")
```

```
print(a.solde())
```

```
AttributeError: 'Compte' object  
has no attribute '__Compte__nom'
```

```
print(a.__Compte__titulaire)
```

```
kim
```

*Que signifie  
cette erreur ?*

*Pourquoi  
ça marche ?*

# « Name Mangling »

En Python, les variables privées ne le sont pas vraiment

Python utilise la technique du “private name mangling”

Si le nom d'un identifiant dans une classe commence par ‘\_\_’  
(et ne se termine pas par ‘\_\_’)

alors l'identifiant est transformé en un nom moins compréhensible

Par exemple :

L'identifiant `__nom` dans la classe `Compte` est transformé en  
`__Compte__nom`

L'identifiant `__titulaire` de la classe `Compte` devient  
`__Compte__titulaire`

# « Name Mangling »

Ce mécanisme de “private name mangling” explique l’erreur :

```
AttributeError: 'Compte' object has no attribute '_Compte__nom'
```

Et pourquoi les variables ne sont pas vraiment privées :

```
c._Compte__titulaire
```

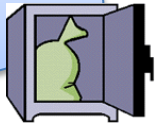
donne accès à la variable privée `__titulaire`



# Variable de classe privée

```
class Compte :  
    __taux_interet = 0.02  
    ...
```

variable de classe  
privée



```
>>> Compte.__taux_interet  
AttributeError: 'Compte' class  
has no attribute 'taux_interet'
```

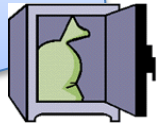
```
>>> a = Compte("Kim")  
>>> a.__taux_interet  
AttributeError: 'Compte' object  
has no attribute '__taux_interet'
```

Comment y accéder ?

# Variable de classe privée

```
class Compte :  
    __taux_interet = 0.02  
    ..
```

variable de classe  
privée



```
>>> Compte.__taux_interet  
AttributeError: 'Compte' class  
has no attribute 'taux_interet'
```

Comment y accéder ?

```
>>> a = Compte("Kim")  
>>> a.__taux_interet  
AttributeError: 'Compte' object  
has no attribute '__taux_interet'
```



```
>>> Compte._Compte__taux_interet  
0.02
```

« private name mangling »

Nouveau

# Méthode de classe

```
class Compte :  
    __taux_interet = 0.02  
  
    @classmethod  
    def taux_interet(cls):  
        return cls.__taux_interet  
  
    @classmethod  
    def set_taux_interet(cls, nouveau_taux):  
        cls.__taux_interet = nouveau_taux  
  
    ...
```

Comment y accéder ? ...

Par une méthode de classe

Reçoit la classe comme paramètre implicite

Envoyé à la classe !

```
Compte.taux_interet()
```

```
# 0.02
```

```
Compte.set_taux_interet(0.04)
```

```
Compte.taux_interet()
```

```
# 0.04
```

Méthode mutateur

Méthode accesseur



# Méthode de classe

```
class Compte :  
    __taux_interet = 0.02  
  
    @classmethod  
    def taux_interet(cls):  
        return cls.__taux_interet  
  
    @classmethod  
    def set_taux_interet(cls, nouveau_taux):  
        cls.__taux_interet = nouveau_taux  
  
    ...
```

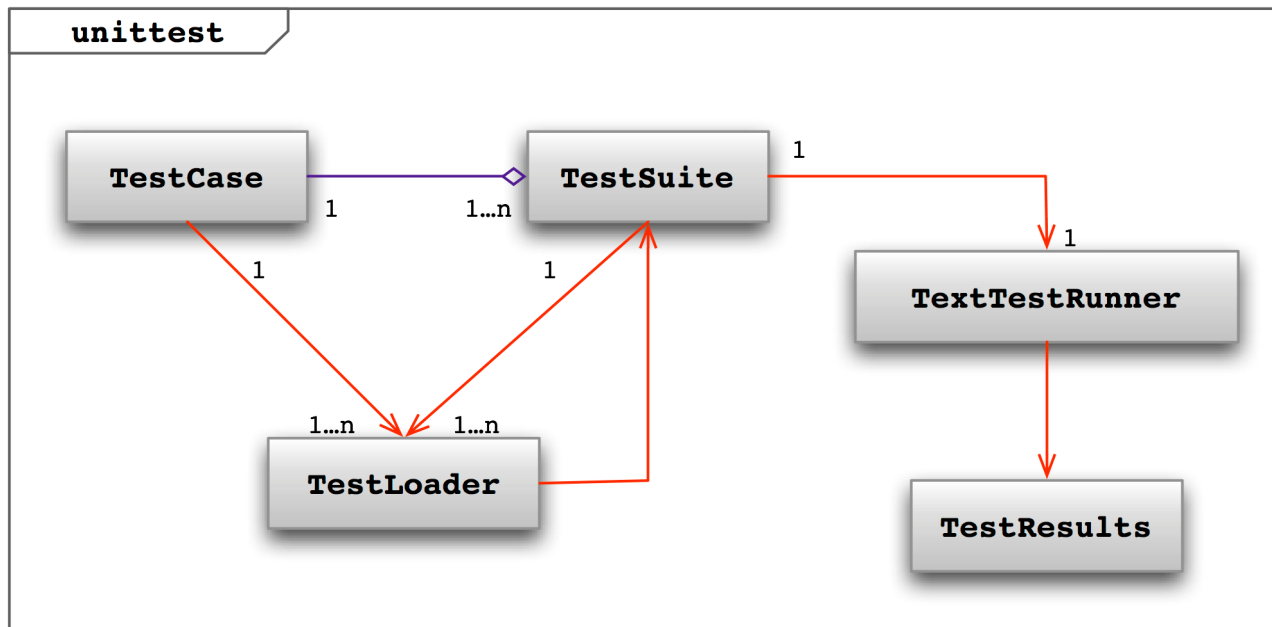
```
a = Compte("Kim")  
compte_kim.taux_interet()  
# 0.02
```

Si pas une méthode d'instance,  
Python va l'appeler  
comme méthode de classe

Nouveau

# Les tests unitaires avec unittest

<https://docs.python.org/3/library/unittest.html>



# Quelques fonctions à tester

```
import random
```

*Comment tester que ces fonctions correspondent à leur spécification?*

```
lst = ["chat", "chien", "renard", "serpent", "cheval"]  
random.choice(lst)  
→ 'renard'
```

*random.choice: cette fonction retourne un élément au hasard de la séquence précisée en paramètre.*

```
lst = [1, 2, 3, 4, 5, 6, 7, 8, 9]  
random.shuffle(lst)  
lst  
→ [1, 6, 7, 4, 3, 5, 9, 8, 2]
```

*random.shuffle: cette fonction mélange une liste; la liste d'origine est modifiée.*

```
random.sample(range(1000), 10)  
→ [389, 406, 890, 955, 837, 401, 971, 716, 954, 862]
```

*random.sample: cette fonction prend une séquence et un nombre en paramètres. Elle retourne une nouvelle séquence contenant autant d'éléments que le nombre indiqué, sélectionnés aléatoirement dans la séquence d'origine.*

# Structure d'un test unitaire

`random.choice`: cette fonction retourne un élément au hasard de la séquence précisée en paramètre.

```
import random
```

1. Importer *unittest*

```
import unittest
```

2. Ecrire une classe test

3. Doit hériter de `TestCase`

```
class RandomTest(unittest.TestCase):
```

```
    """Classe de test utilisé pour tester
    les fonctions du module 'random'."""
```

4. Une méthode par fonctionnalité à tester

```
def test_choice(self):
```

```
    """Test de fonctionnement de la fonction
    random.choice."""
```

```
    l = list(range(10))
```

```
    e = random.choice(l)
```

```
    # Vérifie que 'e' est dans 'l'
```

```
    self.assertIn(e, l)
```

*On teste que l'élément choisi se trouve bien dans la liste originale*

5. Utiliser des méthodes d'assertion `assert...()`

*`assertIn` vérifie qu'un élément se trouve bien dans une liste donnée*

# Exécuter un test unitaire

**Approche 1:** Exécuter le fichier Python directement

```
import unittest

class RandomTest(unittest.TestCase):
    """Classe de test utilisé pour tester
    les fonctions du module 'random'."""
    ...

# Ajoutez cette ligne :
if __name__ == '__main__':
    unittest.main()
```



```
>>> %Run RandomTest.py
```

```
.
```

```
-----
Ran 1 test in 0.000s
```

```
OK
```

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

```
...
```

# Exécuter un test unitaire

## Approche 2a: Via la console de commandes

Last login: Tue Nov 27 11:49:17 on console

```
mac-kim:~ kimmens$ cd /Users/kim/Enseignement/MesCours/Info1/theory-mission-11
```

```
mac-kim:theory-mission-11 kimmens$ python -m unittest RandomTest.py
```

```
.
```

```
Ran 1 test in 0.000s
```

Nombre de tests exécutés

```
OK
```

Récapitulatif:  
OK = tout va bien

Un caractère par test exécuté.

"." = le test s'est validé

"F" = le test n'a pas obtenu le bon résultat

"E" = le test a rencontré une erreur

# Exécuter un test unitaire

**Approche 2b:** Via la console de commandes

```
Last login: Tue Nov 27 11:49:17 on console
mac-kim:~ kimmens$ cd /Users/kim/Enseignement/MesCours/Info1/theory-mission-11
mac-kim:theory-mission-11 kimmens$ python -m unittest -v RandomTest.py
test_choice (RandomTest.RandomTest)
Test de fonctionnement de la fonction random.choice. ... ok
```

-v = verbeux

```
-----
Ran 1 test in 0.000s
```

OK

# Un test qui échoue

```
import random
```

```
import unittest
```

```
class RandomTest(unittest.TestCase):
```

```
    """Classe de test utilisé pour tester  
    les fonctions du module 'random'."""
```

```
    def test_choice(self):
```

```
        """Test le fonctionnement de la fonction  
        random.choice."""
```

```
        l = list(range(10))
```

```
        e = random.choice(l)
```

```
        self.assertIn(e, ('a', 'b', 'c'))
```

Ce test ne sera pas valide

```
if __name__ == '__main__':  
    unittest.main()
```



# Un test qui échoue

```
import random
```

```
import unittest
```

```
class RandomTest(unittest.TestCase):
```

```
    """Classe de test utilisant
       les fonctions du module random"""
```

```
    def test_choice(self):
```

```
        """Test le fonctionnement
           de la fonction random.choice."""
```

```
        l = list(range(10))
```

```
        e = random.choice(l)
```

```
        self.assertIn(e, ('a', 'b', 'c'))
```

```
if __name__ == '__main__':
    unittest.main()
```

```
>>> %Run RandomTest.py
```

```
F
```

```
=====
FAIL: test_choice (__main__.RandomTest)
```

```
Test de fonctionnement de la fonction
random.choice.
```

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

```
File
```

```
"/Users/kim/Enseignement/MesCours/Info1/theory-
mission-11/RandomTest.py", line 12, in test_choice
```

```
    self.assertIn(e, ('a', 'b', 'c'))
```

```
AssertionError: 1 not found in ('a', 'b', 'c')
```

```
-----
Ran 1 test in 0.001s
```

```
FAILED (failures=1)
```

# Test de la fonction random.shuffle

`random.shuffle`: cette fonction mélange une liste; la liste d'origine est modifiée.

```
import random
```

```
import unittest
```

```
class RandomTest(unittest.TestCase):
```

```
    """Classe de test utilisé pour tester  
    les fonctions du module 'random'."""
```

```
# Autres méthodes de test
```

```
def test_shuffle(self):
```

```
    """Test le fonctionnement de la  
    fonction random.shuffle."""
```

```
    l = list(range(10))
```

```
    random.shuffle(l)
```

```
    l.sort()
```

```
    self.assertEqual(l, list(range(10)))
```

*On teste que si on trie la liste mélangée on obtient de nouveau la liste originale*

*assertEqual vérifie que deux valeurs sont identiques*

# Test de la fonction random.sample

```
import random
```

```
import unittest
```

```
class RandomTest(unittest.TestCase):
```

```
    # Autres méthodes de test
```

```
    def test_sample(self):
```

```
        """Test le fonctionnement de la
           fonction random.sample."""
```

```
        l = list(range(10))
```

```
        extrait = random.sample(l, 5)
```

```
        for e in extrait:
```

```
            self.assertIn(e, l)
```

```
            self.assertRaises(ValueError, random.sample, l, 20)
```

*assertRaises vérifie qu'un certain type d'exception est levé si on appelle une certaine fonction avec certains paramètres.*

*random.sample: cette fonction prend une séquence et un nombre en paramètres. Elle retourne une nouvelle séquence contenant autant d'éléments que le nombre indiqué, sélectionnés aléatoirement dans la séquence d'origine.*

*On teste que chaque élément de la liste retournée appartient à la liste originale*

*On teste qu'une exception est levée si le nombre d'éléments sélectionnés est plus élevé que la taille de la liste.* <sup>31</sup>

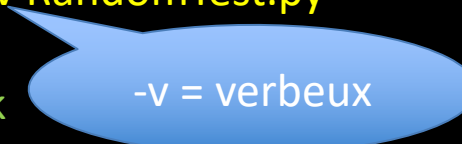
# Exécuter les tests unitaires

**Approche 2b:** Via la console de commandes

```
Last login: Tue Nov 27 11:49:17 on console
mac-kim:~ kimmens$ cd /Users/kim/Enseignement/MesCours/Info1/theory-mission-11
mac-kim:theory-mission-11 kimmens$ python -m unittest -v RandomTest.py
test_choice (RandomTest.RandomTest)
Test de fonctionnement de la fonction random.choice. ... ok
test_sample (RandomTest.RandomTest)
Test le fonctionnement de la fonction random.sample. ... ok
test_shuffle (RandomTest.RandomTest)
Test le fonctionnement de la fonction random.shuffle. ... ok

-----
Ran 3 tests in 0.001s

OK
```



# Les principales méthodes d'assertion

Méthode	Explications
<code>assertEqual(a, b)</code>	<code>a == b</code>
<code>assertNotEqual(a, b)</code>	<code>a != b</code>
<code>assertTrue(x)</code>	<code>x is True</code>
<code>assertFalse(x)</code>	<code>x is False</code>
<code>assertIs(a, b)</code>	<code>a is b</code>
<code>assertIsNot(a, b)</code>	<code>a is not b</code>
<code>assertIsNone(x)</code>	<code>x is None</code>
<code>assertIsNotNone(x)</code>	<code>x is not None</code>
<code>assertIn(a, b)</code>	<code>a in b</code>
<code>assertNotIn(a, b)</code>	<code>a not in b</code>
<code>assertIsInstance(a, b)</code>	<code>isinstance(a, b)</code>
<code>assertNotIsInstance(a, b)</code>	<code>not isinstance(a, b)</code>
<code>assertRaises(exception, fonction, *args, **kwargs)</code>	Vérifie que la fonction lève l'exception attendue.

# Initialisation des tests

```
class RandomTest(unittest.TestCase):  
  
    def test_choice(self):  
        """Test de fonctionnement de la fonction random.choice."""  
        l = list(range(10))  
        e = random.choice(l)  
        self.assertIn(e, l)  
  
    def test_shuffle(self):  
        """Test le fonctionnement de la fonction random.shuffle."""  
        l = list(range(10))  
        random.shuffle(l)  
        l.sort()  
        self.assertEqual(l, list(range(10)))  
  
    def test_sample(self):  
        """Test le fonctionnement de la fonction random.sample."""  
        l = list(range(10))  
        extrait = random.sample(l, 5)  
        for e in extrait:  
            self.assertIn(e, l)  
        self.assertRaises(ValueError, random.sample, l, 20)
```

*Comment éviter cette répétition?*

# Initialisation des tests

```
class RandomTest(unittest.TestCase):
```

```
def setUp(self):  
    """Initialisation des tests."""  
    self.l = list(range(10))
```

```
def test_choice(self):  
    """Test de fonctionnement de la  
    e = random.choice(self.l)  
    self.assertIn(e, self.l)
```

```
def test_shuffle(self):  
    """Test le fonctionnement de la fonction  
    random.shuffle(self.l)  
    self.l.sort()  
    self.assertEqual(self.l, list(range(10)))
```

```
def test_sample(self):  
    """Test le fonctionnement de la  
    extrait = random.sample(self.l,  
    for e in extrait:  
        self.assertIn(e, self.l)  
    self.assertRaises(ValueError, ran
```

*Utiliser la méthode `setUp`  
de `unittest.TestCase`*

La méthode `setUp` est appelée  
avant chaque méthode de test

Remplacer `l` par  
`self.l` car `l` est  
maintenant un attribut  
de la classe test

*Remarque: il existe également  
une méthode `tearDown` qui est  
appelée après chaque test*