Partie III

La programmation orientée objets

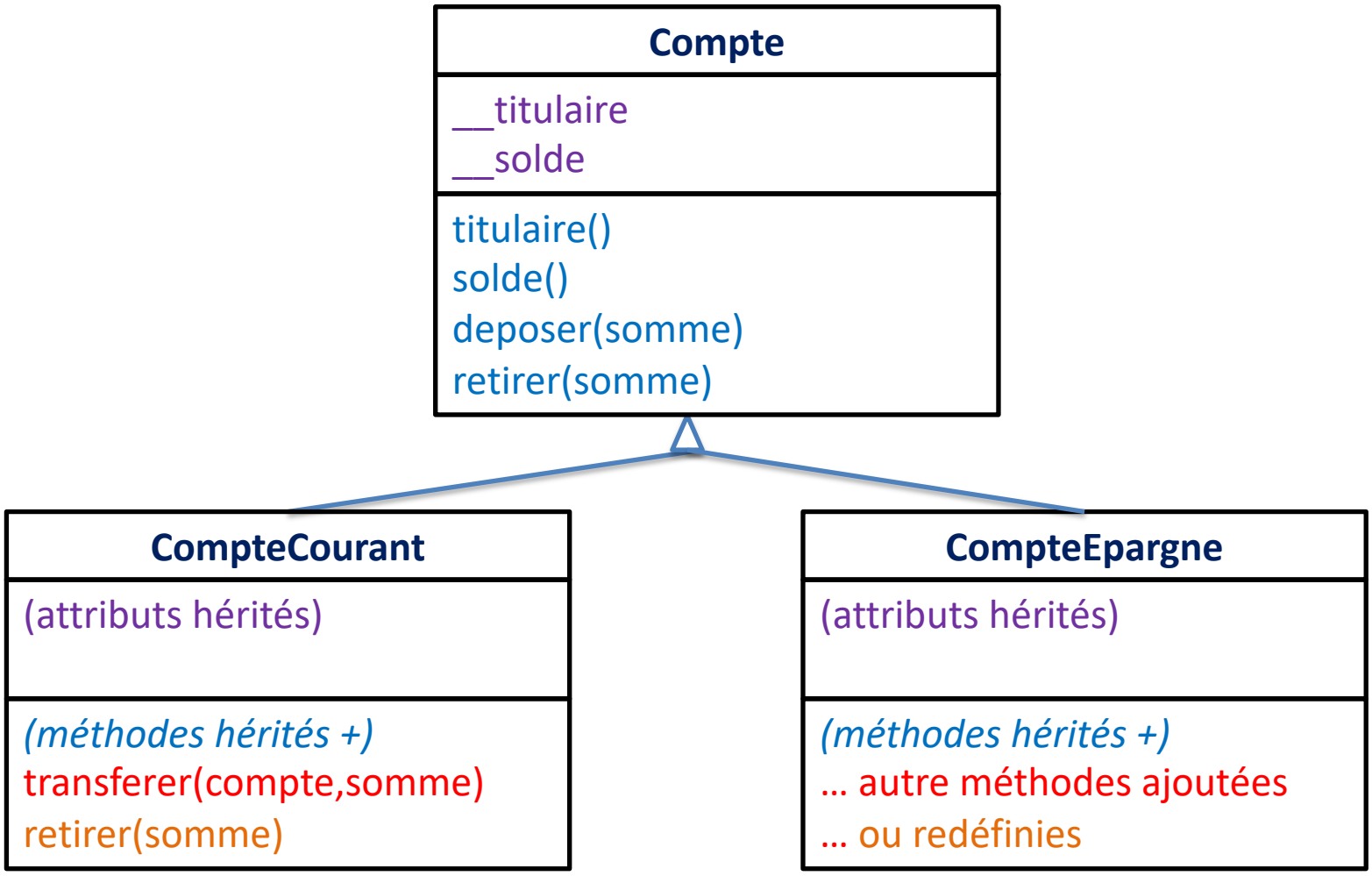
Informatique 1
Introduction à la programmation

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


Compte
__titulaire
__solde
titulaire()
solde()
deposer(somme)
retirer(somme)
CompteCourant
(attributs hérités)
(méthodes hérités +)
transferer(compte,somme)
retirer(somme)
CompteEpargne
(attributs hérités)
(méthodes hérités +)
… autre méthodes ajoutées
… ou redéfinies

type()

```
class Compte :
    …

class CompteCourant :
    …

class CompteEpargne :
    …

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

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

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'heritage

```
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 CompteCourant(Compte) :
    …

class CompteEpargne(Compte) :
    …

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) #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
```

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 ?

Méthode de classe

Nouveau

```
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 implicit

```
Compte.taux_interet()
# 0.02
Compte.set_taux_interet(0.04)
Compte.taux_interet()
# 0.04
```

Envoyé à la classe !

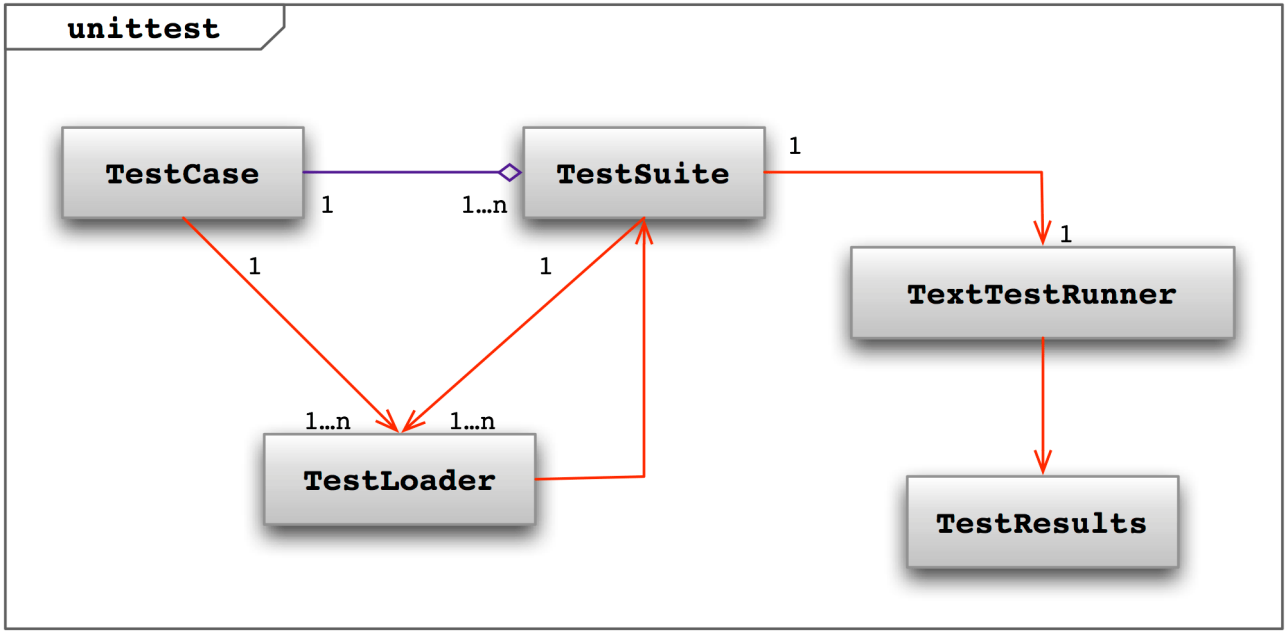
Méthode mutateur

Méthode accesseur

Les tests unitaires avec `unittest`

Nouveau

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



Quelques fonctions à tester

```
import random
```

```
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.

Comment tester que cette fonction correspond à sa specification?

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'."""

    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)
```

4. Uné méthode par fonctionnalité à tester

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

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

***assertIn** vérifie qu'un élement 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

OK
```

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

Nombre de tests exécutés

Récapitulatif:
OK = tout va bien

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

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

OK
```

-v = verbeux

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'))

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

Ce test ne sera pas valide

Un test qui échoue

```
import random
```

```
import unittest

class RandomTest(unittes
  """Classe de test util
     les fonctions du mo

  def test_choice(self):
    """Test le fonctionn
       random.choice."""
    l = list(range(10))
    e = random.choice(l)
    self.assertIn(e, ('a

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)
```

Les principales méthodes d'assertion

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