Partie III

La programmation orientée objets



Informatique 1
Introduction à la programmation

Mission 10 : INTRODUCTION

La programmation orientée objets : le polymorphisme

Kim Mens

Le polymorphisme

Le mot « **polymorphisme** » vient du grec et signifie « *qui peut prendre plusieurs formes* »

En programmation orientée objets, on parle du **polymorphisme** quand des méthodes de mêmes noms ont des implémentations différentes sur des classes différentes.

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

class CompteCourant :
    def __str__(self) :
        return "compte courant"

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

Le polymorphisme est la capacité pour des **objets de classes différentes** de **réagir différemment à la même méthode**, parce que chacune redéfinit cette méthode à sa manière.

Le polymorphisme ad-hoc

Le mot « **polymorphisme** » vient du grec et signifie « *qui peut prendre plusieurs formes* »

En programmation orientée objets, on parle du **polymorphisme** quand des méthodes de mêmes noms ont des implémentations différentes sur des classes différentes.

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

class CompteCourant :
    def __str__(self) :
        return "compte courant"

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

On parle du « **polymorphisme ad-hoc** » si les classes ne sont pas liées par héritage

Le polymorphisme « ad-hoc »

```
class Duck :

  def __init__(self,nom):
    self.nom = nom

  def talk(self):
    return "Quack, quack!"

  def fly(self):
    return "Flap, flap!"

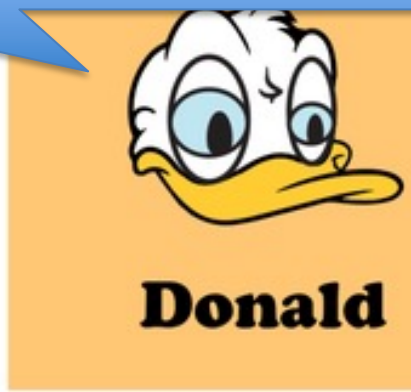
  def __str__(self):
    return self.talk() + \
           " " + self.fly()
```

```
class Interview :

  def __init__(self,thing):
    self.t = thing

  def __str__(self):
    return self.t.nom + \
           ': "' + \
           str(self.t) + '"'
```

« Duck Typing » en Python:
"If it walks like a duck and quacks like a duck, it must be a duck"




```
>>> donald = Duck("Donald Duck")
>>> interview = Interview(donald)
>>> print(interview)
Donald Duck: "Quack, quack! Flap, flap!"
```

Le typage canard
“duck typing”

```
class Person :

  def __init__(self,nom) :
    self.nom = nom

  def talk(self) :
    return "I won the elections

  def fly(self) :
    return "I'm flying."

  def __str__(self):
    return "I'm " + self.nom + \
           ". " + self.talk() + \
           " " + self.fly()
```

```
class Interview :

  def __init__(self,thing):
    self.t = thing

  def __str__(self):
    return self.t.nom + \
           ': "' + \
           str(self.t) + '"'
```

```
>>> donald = Person("Donald Trump")
>>> interview = Interview(donald)
>>> print(interview)
Donald Trump: "I'm Donald Trump. I won
the elections. I'm flying."
```



Le polymorphisme par héritage

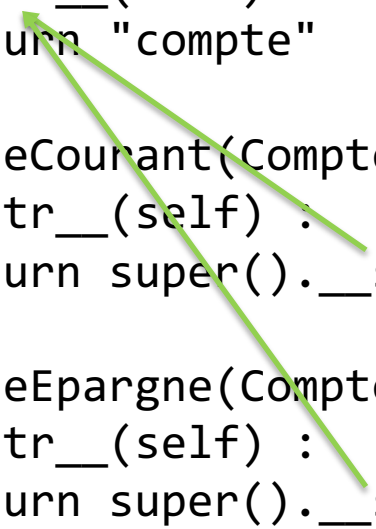
On parle du « **polymorphisme par héritage** » si les classes sont liées par héritage

En programmation orientée objets, on parle du **polymorphisme** quand des méthodes de mêmes noms ont des implémentations différentes sur des classes différentes.

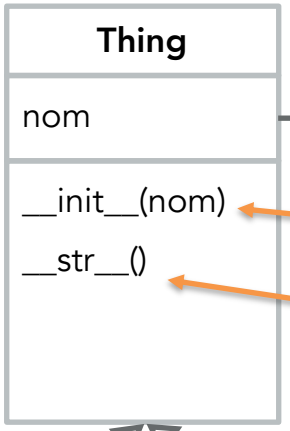
```
class Compte :
    def __str__(self) :
        return "compte"

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

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



Le polymorphisme par héritage



1

```
class Interview :

  def __init__(self,thing):
    self.t = thing

  def __str__(self):
    return self.t.nom + \
           ': "' + \
           str(self.t) + '"'
```

Duck



Donald

Person



Donald

Le polymorphisme par héritage

```
class Thing :

  def __init__(self,nom):
    self.nom = nom

  def __str__(self):
    return self.talk() + \
           " " + self.fly()
```

```
class Duck(Thing) :

  def talk(self) :
    return "Quack, quack!"

  def fly(self) :
    return "Flap, flap!"
```

Ajout de méthodes

```
class Person(Thing) :

 def talk(self) :
    return "I won the elections."
 def fly(self) :
    return "I'm flying..."

 def __str__(self):
    return "I'm " + self.nom \
           + ". " + \
           super().__str__()
```

Redéfinition de méthode

Le polymorphisme par héritage

```
class Thing :

  def __init__(self,nom):
    self.nom = nom

  def __str__(self):
    return self.talk() + \
           " " + self.fly()
```

```
class Interview :

  def __init__(self,thing):
    self.t = thing

  def __str__(self):
    return self.t.nom + \
           ': "' + \
           str(self.t) + '"'
```

```
class Duck(Thing) :

  def talk(self) :
    return "Quack, quack!"

  def fly(self) :
    return "Flap, flap!"
```

```
>>> donald = Duck("Donald Duck")
>>> interview = Interview(donald)
>>> print(interview)
Donald Duck: "Quack, quack! Flap,
flap!"
```

Le polymorphisme par héritage

```
class Thing :

  def __init__(self,nom):
    self.nom = nom

  def __str__(self):
    return self.talk() + \
           " " + self.fly()
```

```
class Interview :

  def __init__(self,thing):
    self.t = thing

  def __str__(self):
    return self.t.nom + \
           ': "' + \
           str(self.t) + '"'
```

```
class Person(Thing) :

 def talk(self) :
    return "I won the elections."
  def fly(self) :
    return "I'm flying..."

  def __str__(self):
    return "I'm " + self.nom \
           + ". " + \
           super().__str__()
```

```
>>> donald = Person("Donald Trump")
>>> interview = Interview(donald)
>>> print(interview)
Donald Trump: "I'm Donald Trump. I
won the elections. I'm flying."
```

La liaison dynamique

```
class Thing :

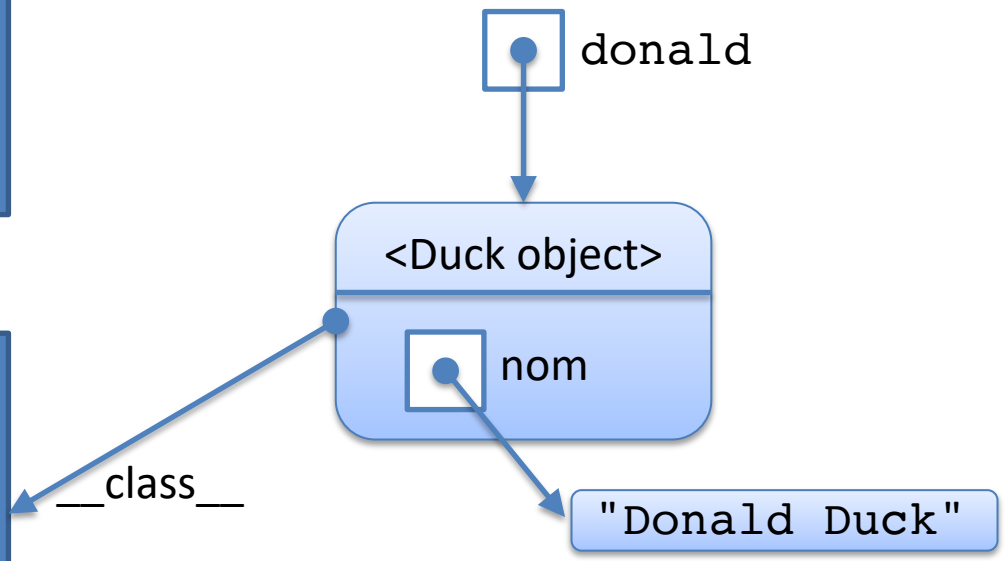
  def __init__(self,nom):
    self.nom = nom

  def __str__(self):
    return self.talk() + \
           " " + self.fly()
```

```
class Duck(Thing) :

  def talk(self) :
    return "Quack, quack!"

  def fly(self) :
    return "Flap, flap!"
```




```
>>> donald = Duck("Donald Duck")
```

La résolution de méthodes

```
class Thing :
  def __init__(self,nom):
    self.nom = nom
  def __str__(self):
    return self.talk() + \
           " " + self.fly()
```



```
class Duck(Thing) :
  def talk(self) :
    return "Quack, quack!"
  def fly(self) :
    return "Flap, flap!"
```

La résolution de méthodes :

- **On cherche la méthode à exécuter d'abord dans la classe de l'objet auquel le message a été envoyé.**
- Si pas trouvé on cherche dans la classe mère, et ainsi de suite.
- Si la méthode n'existe pas dans cette hiérarchie, on obtiendra une erreur du type AttributeError

```
message = __str__()
self = donald
```

```
>>> donald = Duck("Donald Duck")

>>> donald.__str__()
```

La liaison dynamique

```
class Thing :
  def __init__(self,nom):
    self.nom = nom
  def __str__(self):
    return self.talk() + \
           " " + self.fly()
```

```
class Duck(Thing) :
  def talk(self) :
    return "Quack, quack!"
  def fly(self) :
    return "Flap, flap!"
```

```
message = __str__()
self = donald
```

Même si on trouve la méthode dans la classe mère

- self refère toujours à l'objet auquel le message initial a été envoyé
- ceci permet d'exécuter la méthode comme si elle était définie dans la classe fille

```
>>> donald = Duck("Donald Duck")

>>> donald.__str__()
```

La liaison dynamique

```
class Thing :
  def __init__(self,nom):
    self.nom = nom
  def __str__(self):
    return self.talk() + \
           " " + self.fly()
```

on exécute la méthode pour
`self = donald`

```
return self.talk() + \
           " " + self.fly()
```

```
class Duck(Thing) :
  def talk(self) :
    return "Quack, quack!"
  def fly(self) :
    return "Flap, flap!"
```

```
>>> donald = Duck("Donald Duck”)

>>> donald.__str__()
```

La liaison dynamique

```
class Thing :

  def __init__(self,nom):
    self.nom = nom

  def __str__(self):
    return self.talk() + \
           " " + self.fly()
```

```
class Duck(Thing) :

  def talk(self) :
    return "Quack, quack!"

  def fly(self) :
    return "Flap, flap!"
```

```
return donald.talk() + \
           " " + donald.fly()
```

```
message = talk()
self = donald
return "Quack, quack!"
```

```
>>> donald = Duck("Donald Duck")

>>> donald.__str__()
```

La liaison dynamique

```
class Thing :
  def __init__(self,nom):
    self.nom = nom
  def __str__(self):
    return self.talk() + \
           " " + self.fly()
```

```
class Duck(Thing) :
  def talk(self) :
    return "Quack, quack!"
  def fly(self) :
    return "Flap, flap!"
```

```
return "Quack, quack!" + \
           " " + donald.fly()
```

```
message = fly()
self = donald
return "Flap, flap!"
```

```
>>> donald = Duck("Donald Duck")

>>> donald.__str__()
```

La liaison dynamique

```
class Thing :
  def __init__(self,nom):
    self.nom = nom
  def __str__(self):
    return self.talk() + \
           " " + self.fly()
```



```
class Duck(Thing) :
  def talk(self) :
    return "Quack, quack!"
  def fly(self) :
    return "Flap, flap!"
```

```
return "Quack, quack! " + \
           " " + "Flap, flap!"
```

```
>>> donald = Duck("Donald Duck")
>>> donald.__str__()
"Quack, quack! Flap, flap!"
```

La sémantique de super()

```
class Thing :

  def __init__(self,nom):
    self.nom = nom

  def __str__(self):
    return self.talk() + \
           " " + self.fly()
```

```
class Person(Thing) :

 def talk(self) :
    return "I won the elections."
  def fly(self) :
    return "I'm flying..."

  def __str__(self):
    return "I'm " + self.nom \
           + ". " + \
           super().__str__()
```

`super( ).__str__(…)` veut dire :

- ne cherchez pas l'implementation de la méthode `__str__(…)` pas dans cette classe
- mais cherchez-la dans la classe mère de cette classe
- (ou encore plus haut dans l'hiérarchie)

```
donald = Person("Donald Trump")

donald.__str__()
```

La sémantique de super()

```
class Thing :

  def __init__(self,nom):
    self.nom = nom

  def __str__(self):
    return self.talk() + \
           " " + self.fly()
```

```
class Person(Thing) :

 def talk(self) :
    return "I won the elections."
  def fly(self) :
    return "I'm flying..."

 def __str__(self):
    return "I'm " + self.nom \
           + ". " + \
           super().__str__()
```

```
message = __str__()
self = donald
```

```
return "I'm Donald Trump. "
       + super().__str__()
```

```
donald = Person("Donald Trump")

donald.__str__()
```

La sémantique de super()

```
class Thing :

  def __init__(self,nom):
    self.nom = nom

  def __str__(self):
    return self.talk() + \
           " " + self.fly()
```

on exécute la méthode pour
self = donald

```
message = __str__()
self = donald
```

```
return self.talk() + \
           " " + self.fly()
```

```
class Person(Thing) :

 def talk(self) :
    return "I won the elections."
  def fly(self) :
    return "I'm flying..."

  def __str__(self):
    return "I'm " + self.nom \
           + ". " + \
           super().__str__()
```

```
return "I'm Donald Trump. "
       + super().__str__()
```

```
donald = Person("Donald Trump")

donald.__str__()
```

La sémantique de super()

```
class Thing :

  def __init__(self,nom):
    self.nom = nom

  def __str__(self):
    return self.talk() + \
           " " + self.fly()
```

```
return donald.talk() + \
           " " + donald.fly()
```

"I won the elections." +
" " + "I'm flying..."

```
class Person(Thing) :

 def talk(self) :
    return "I won the elections."
 def fly(self) :
    return "I'm flying..."

 def __str__(self):
    return "I'm " + self.nom \
           + ". " + \
           super().__str__()
```

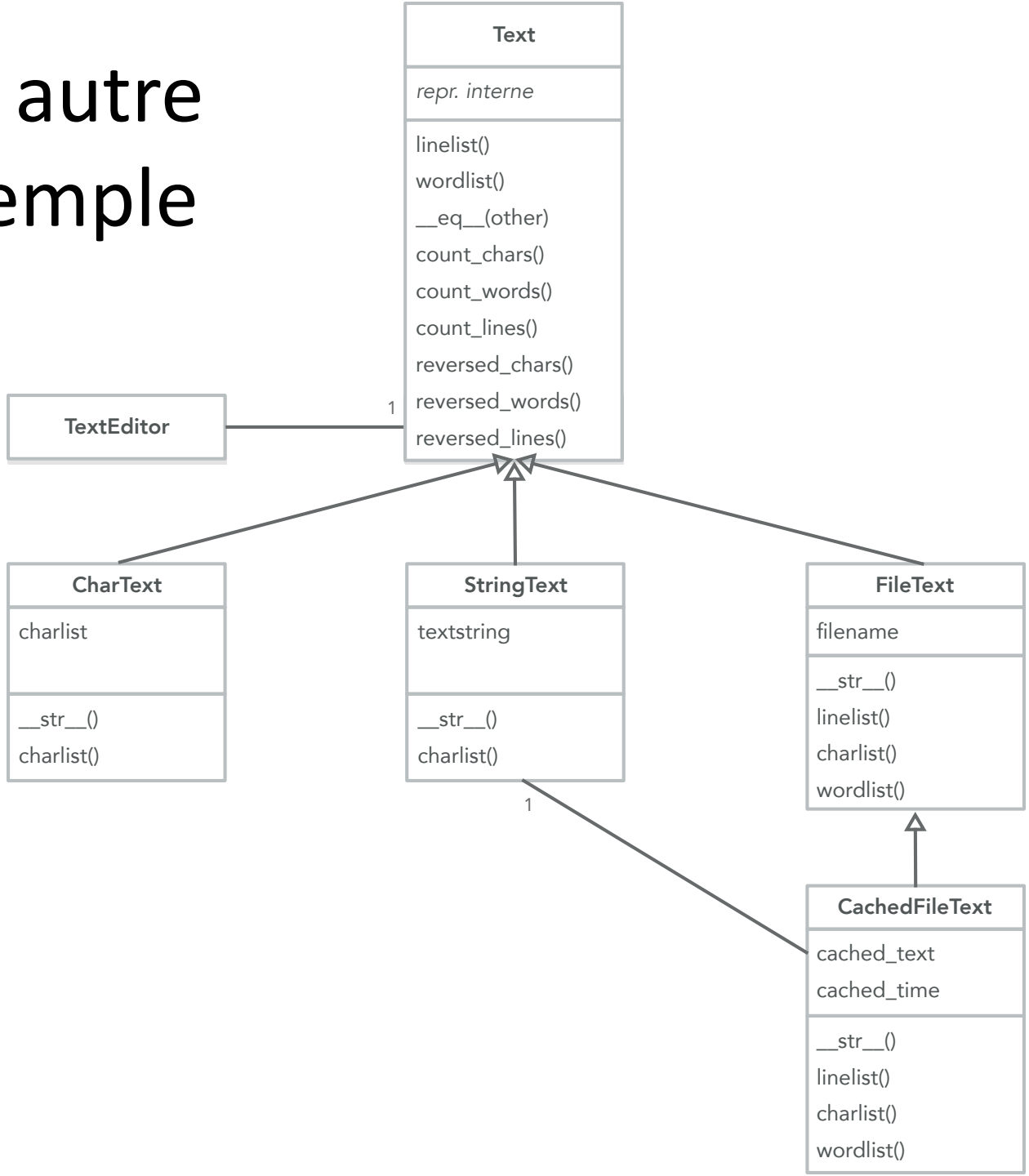
```
return "I'm Donald Trump. "
       + super().__str__()
```

```
donald = Person("Donald Trump")

donald.__str__()

"I'm Donald Trump. I won the
     elections. I'm flying..."
```

Un autre exemple



Mission 10

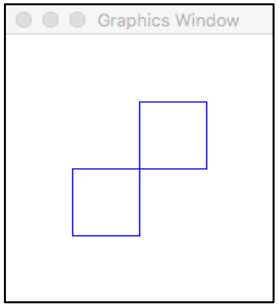
Objectifs :

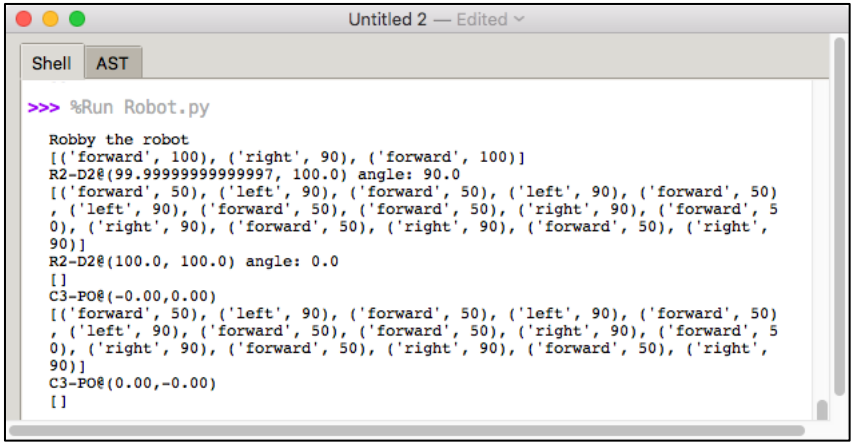
- Polymorphisme
- Tests unitaires

Problème :

- Deux implémentations alternatives d'un robot

XYRobot





TurtleBot

