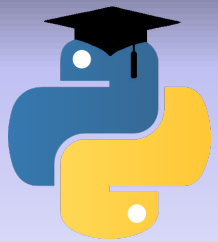
UV2.1. Langage et algorithmique - CR1. Programmation Orientée Objet (POO)

A. Malek TOUMI

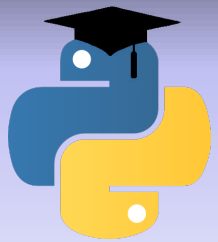
toumiab@ensta-bretagne.fr

2020/2021

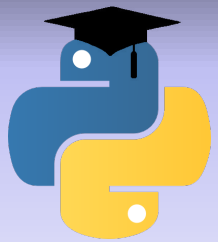
ENSTA Bretagne



Sommaire



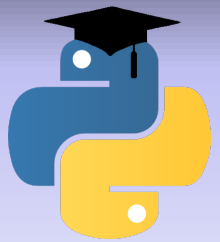
Sommaire



Exemple : rationnels

- Définition d'une Classe : Constructeur et Variables d'instance
- + méthodes d'instance : égalité, multiplication, représentation, ...

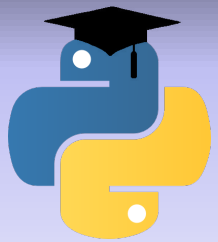
```
class Rationnel (object):
    def __init__(self, num, den = 1):
        self.__num = num
        self.__den = den
```



Exemple : rationnels

- Définition d'une Classe : Constructeur et Variables d'instance
- + méthodes d'instance : égalité, multiplication, représentation, ...

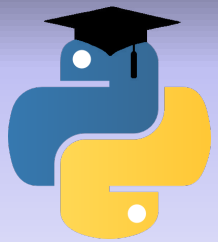
```
class Rationnel (object):
    def __init__(self, num, den = 1):
        self.__num = num
        self.__den = den

    def __eq__(self, other):
        return self.__num * other.__den == self.__den
        * other.__num
```



Exemple : rationnels

- Définition d'une Classe : Constructeur et Variables d'instance
- + méthodes d'instance : égalité, multiplication, représentation, ...

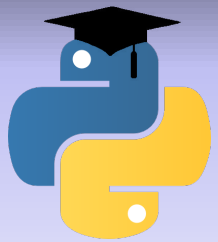
```
class Rationnel (object):
    def __init__(self, num, den = 1):
        self.__num = num
        self.__den = den

    def __mult__(self, other):
        n = self.__num * other.__num
        d = self.__den * other.__den
        return Rationnel(n, d)
```



Exemple : rationnels

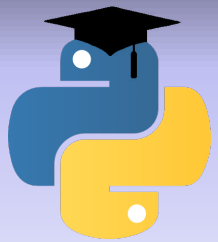
- Définition d'une Classe : Constructeur et Variables d'instance
- + méthodes d'instance : égalité, multiplication, représentation, ...

```
class Rationnel (object):
    def __init__(self, num, den = 1):
        self.__num = num
        self.__den = den

    def __str__(self):
        return '{0}/{1}'.format(self.__num, self.__den)
```



Exemple : Classe/Type `Rationnel`

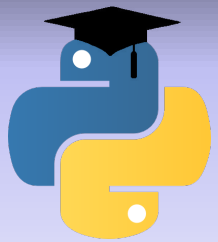
Rationnel
__num __den
__init__ __eq__ __add__ __mul__ __str__ get_numerateur get_denominateur pgcd simplifie



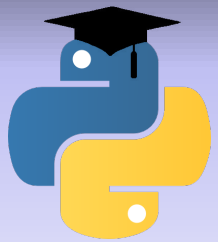
Définition

Remarques (Quelques remarques importantes)

- Tous les attributs et méthodes des classes Python sont `publics` au sens de java et C++,
- Le constructeur d'une classe est une méthode spéciale qui s'appelle *__init__(self)*.
- Les attributs (variables d'instance) sont *généralement* créés et initialisés dans le constructeur
  (exemple : *self.__nomVar*, *self._nomVar*, *self.nomVar*)
- Toutes les méthodes d'instance prennent une variable self comme premier argument. Cette variable est une référence à l'objet manipulé.



Sommaire



Concepts

Objectif :

- Augmenter la réutilisabilité

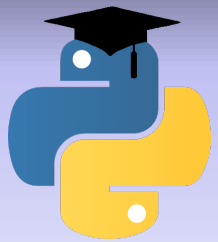
$\Rightarrow$ modularité du code, notion de composants

Concepts mis en place :

- encapsulation : les décorateurs ; les getteurs/setteurs
- héritage (UE 2.1)
- polymorphisme (UE 2.1)

Définition (Encapsulation)

Mécanisme permettent de cacher l'implantation d'un TAD. Ce méchanisme nous fournit une interface pour accéder aux données (lecture/écriture de contenu).



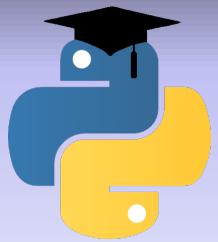
Notion de classe et d'instance

- Classe $\simeq$ type
- Instancier la classe $\simeq$ créer une variable de ce type
- Objet ou instance : variable créée dont le type est une classe

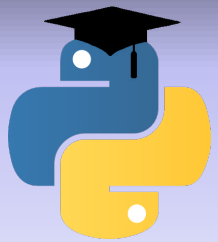
Classe $\rightsquigarrow$ plan
Instance $\rightsquigarrow$ réalisation

Important

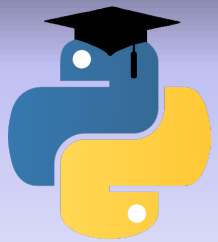
Un programme est composé d'instances (objets) et non de classes



Sommaire



Sommaire



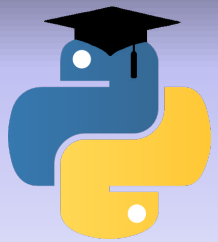
Définition

Définition (Héritage)

Mécanisme spécifique aux langages orientés objet qui permet à une classe B (appelée sous-classe ou classe fille) d'hériter de toutes les propriétés d'une classe A (appelée super-classe ou classe mère).

Utilité :

- Factoriser du code
- Gérer des différents de manière similaire
- Augmenter la réutilisabilité



Exemple

- Gestion de figures géométriques
- Différents types de figures : cercles et rectangles
- Propriétés communes (position du centre de la figure)
- Propriétés spécifiques (rayon du cercle, longueur et largeur du rectangle)

$\Rightarrow$ trois classes

Exemple

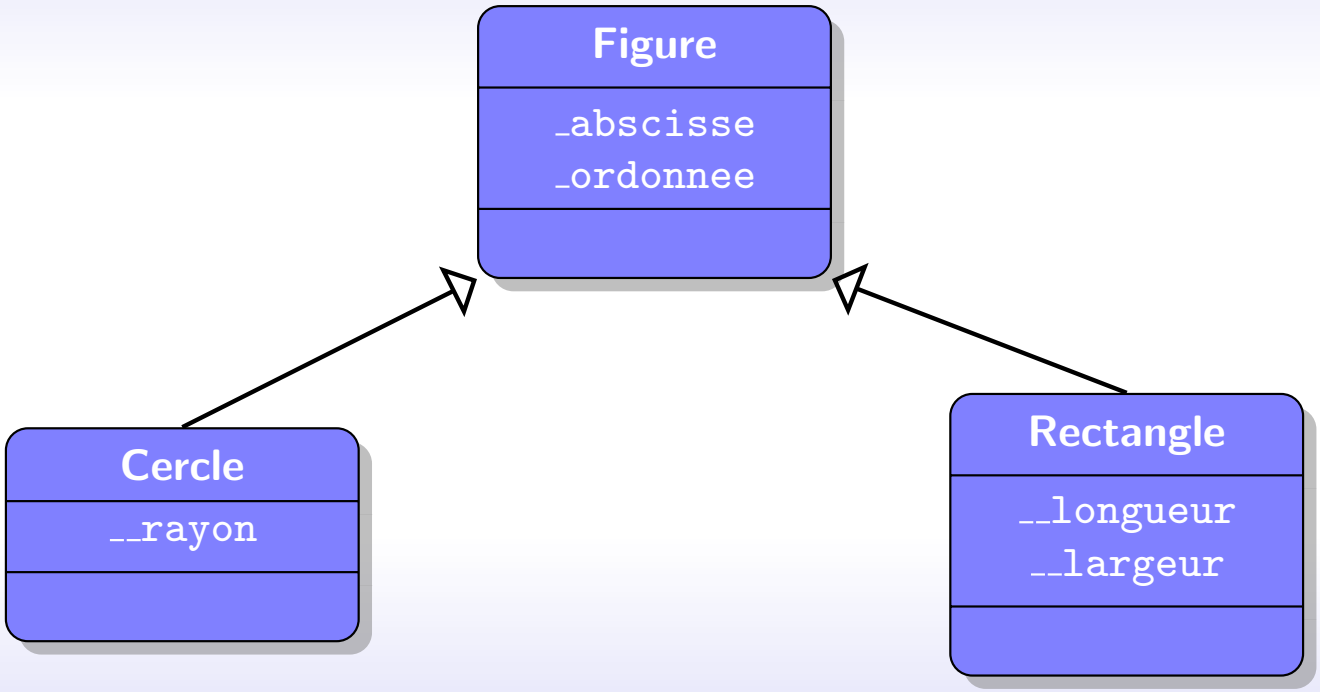
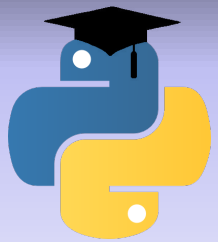


Figure
_abscisse
_ordonnee
Cercle
__rayon
Rectangle
__longueur
__largeur



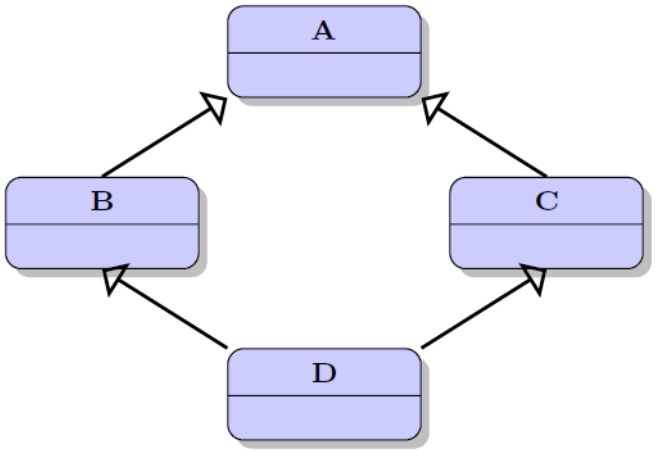
Héritage multiple

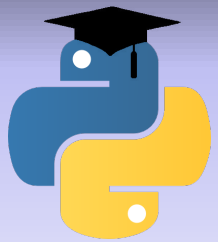
Remarque (Héritage multiple)

Python, à l'encontre de certains langages, permet d'hériter de plusieurs classes.

Héritage multiple

Exemple : D hérite de B et C, B et C héritent de A.





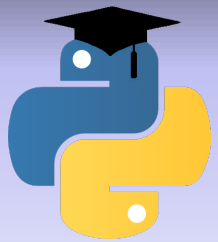
Héritage multiple

Remarque (Héritage multiple)

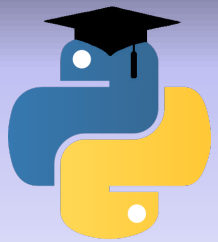
Python, à l'encontre de certains langages, permet d'hériter de plusieurs classes.

Attention

L'héritage multiple entraîne des problèmes théoriques lorsqu'on hérite plusieurs fois du même parent.
$\Longrightarrow$ linéarisation de l'ordre de l'héritage



Sommaire



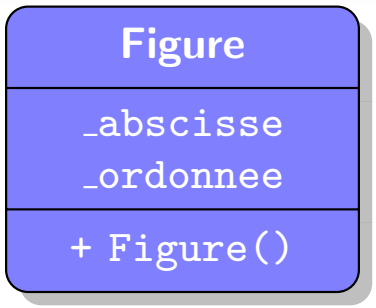
Héritage en Python

- En Python : l'héritage est défini entre les parenthèses
- Le constructeur est la méthode nommée *__init__*
- Visibilité des variables d'instance : (__) $\Longrightarrow$ décorateurs
- Si aucun héritage n'est précisé : hérite de **object**

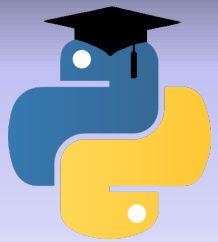
Exemple (Figures)

```
class Figure (object):
   def __init__(self,x, y):
      self._abscisse = x
      self._ordonnee = y

f1 = Figure(10,2) # création de l'objet Figure f1
f2 = Figure(0, 4) # création de l'objet Figure f2
```

Héritage en Python




```
f = Figure(10,11) # Création de l'objet Figure f
f1 = Figure (1,5) # Création de l'objet Figure f1
```



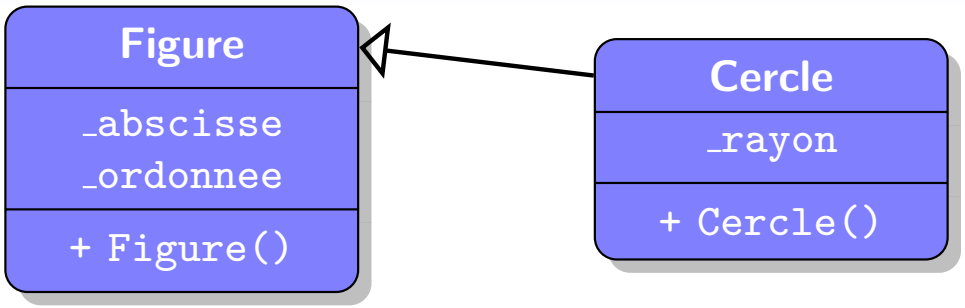
Héritage en Python

- En Python : l'héritage est défini entre les parenthèses
- Le constructeur est la méthode nommée *__init__*
- Visibilité des variables d'instance : (__) $\Longrightarrow$ décorateurs
- Si aucun héritage n'est précisé : hérite de **object**

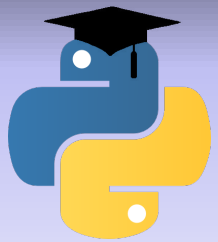
Exemple (Figures)

```
class Figure (object):
   def __init__(self,x, y):
      self._abscisse = x
      self._ordonnee = y
class Cercle (Figure):
   def __init__(self,x=0, y=0, r=0):
      super().__init__(x,y) # définit et initialise
      self._rayon = r       # _abscisse et _ordonnee
```

Héritage en Python



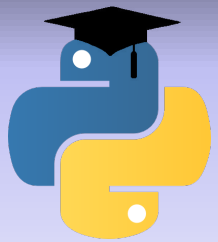

```
f= Figure(10,11) # Création de l'objet Figure f
c= Cercle (1,5,3) # Création de l'objet Cercle c
```



Appel du constructeur de la super-classe

Exemple (Figures)

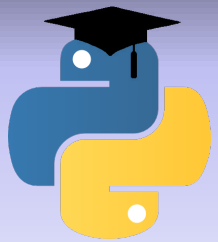
```
class Figure(object):
   def __init__(self,a, b):
      self._abscisse = a
      self._ordonnee = b
class Cercle (Figure):
   def __init__(self,x=0, y=0, r=0):
      super().__init__(x,y) # définit et initialise
      self._rayon = r       # _abscisse et _ordonnee

c = Cercle(10, 3, 2) # création de l'objet c
```



Appel du constructeur de la super-classe

Exemple (Figures)

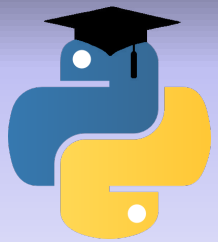
```
class Figure(object):
   def __init__(self,a, b):
      self._abscisse = a
      self._ordonnee = b
class Cercle (Figure):
   def __init__(self,x=0, y=0, r=0):
      super().__init__(x,y) # définit et initialise
      self._rayon = r       # _abscisse et _ordonnee

c = Cercle(10, 3, 2) # création de l'objet c
```



Appel du constructeur de la super-classe

Exemple (Figures)

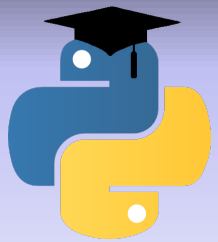
```
class Figure(object):
   def __init__(self,a, b):
      self._abscisse = a
      self._ordonnee = b
class Cercle (Figure):
   def __init__(self,x=0, y=0, r=0):
      super().__init__(x,y) # définit et initialise
      self._rayon = r       # _abscisse et _ordonnee

c = Cercle(10, 3, 2) # création de l'objet c
```



Appel du constructeur de la super-classe

Exemple (Figures)

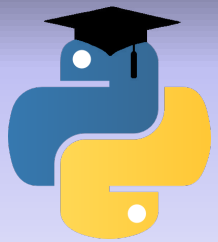
```
class Figure(object):
   def __init__(self,a, b):
      self._abscisse = a
      self._ordonnee = b
class Cercle (Figure):
   def __init__(self,x=0, y=0, r=0):
      super().__init__(x,y) # définit et initialise
      self._rayon = r       # _abscisse et _ordonnee

c = Cercle(10, 3, 2) # création de l'objet c
```



Appel du constructeur de la super-classe

Exemple (Figures)

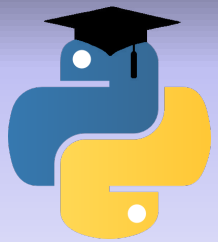
```
class Figure(object):
   def __init__(self,a, b):
      self._abscisse = a
      self._ordonnee = b
class Cercle (Figure):
   def __init__(self,x=0, y=0, r=0):
      super().__init__(x,y) # définit et initialise
      self._rayon = r       # _abscisse et _ordonnee

c = Cercle(10, 3, 2) # création de l'objet c
```



Appel du constructeur de la super-classe

Exemple (Figures)

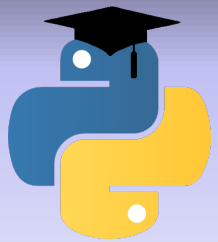
```
class Figure(object):
   def __init__(self,a, b):
      self._abscisse = a
      self._ordonnee = b
class Cercle (Figure):
   def __init__(self,x=0, y=0, r=0):
      super().__init__(x,y) # définit et initialise
      self._rayon = r       # _abscisse et _ordonnee

c = Cercle(10, 3, 2) # création de l'objet c
```



Appel du constructeur de la super-classe

Exemple (Figures)

```
class Figure(object):
   def __init__(self,a, b):
      self._abscisse = a
      self._ordonnee = b
class Cercle (Figure):
   def __init__(self,x=0, y=0, r=0):
      super().__init__(x,y) # définit et initialise
      self._rayon = r       # _abscisse et _ordonnee

c = Cercle(10, 3, 2) # création de l'objet c
```

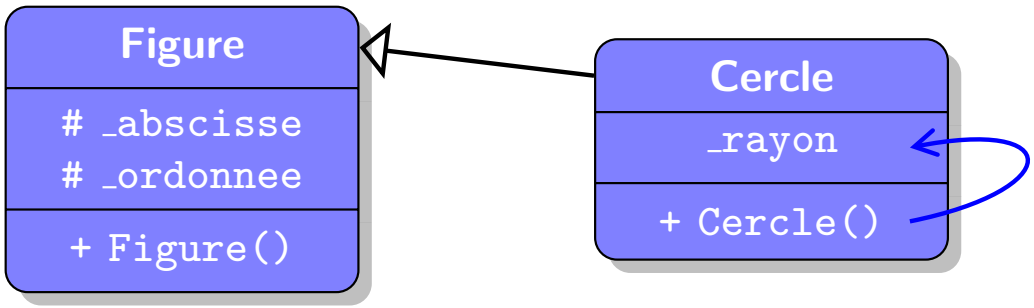


Appel du constructeur de la super-classe

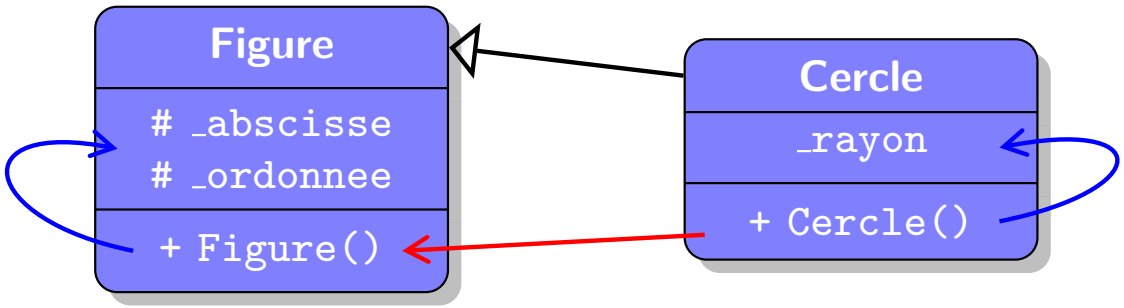
Exemple (Figures)

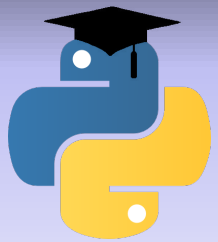
```
class Figure(object):
   def __init__(self,a, b):
      self._abscisse = a
      self._ordonnee = b
class Cercle (Figure):
   def __init__(self,x=0, y=0, r=0):
      super().__init__(x,y) # définit et initialise
      self._rayon = r       # _abscisse et _ordonnee

c = Cercle(10, 3, 2) # création de l'objet c
```

Appel du constructeur de la super-classe

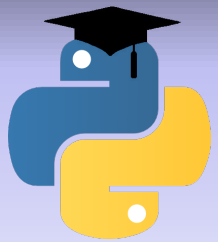


Appel du constructeur de la super-classe

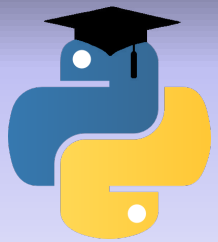




Sommaire



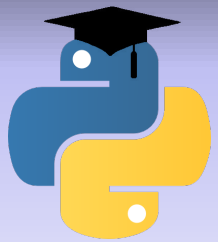
Sommaire



Définition

Définition (Polymorphisme)

Mécanisme selon lequel un même message peut être envoyé vers des objets de types différents, chaque objet réagissant de façon originale.



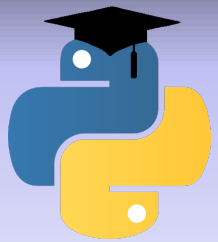
Définition

Définition (Polymorphisme)

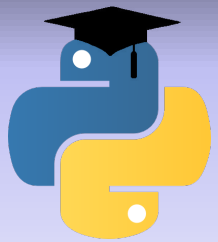
Mécanisme selon lequel un même message peut être envoyé vers des objets de types différents, chaque objet réagissant de façon originale.

Objectif :

- Rendre une certaine complexité de programmation transparente
- Faciliter la conception de programmes

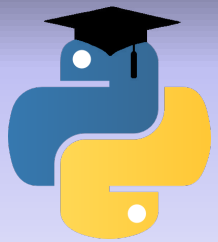


Sommaire



Principe

- Plusieurs méthodes avec le même nom dans une hiérarchie de classes
- Méthode appelée : correspond à l'objet instancié
- Avantage : manipuler toutes les instances d'une hiérarchie de classes de manière uniforme

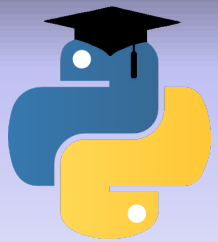


Principe

- Plusieurs méthodes avec le même nom dans une hiérarchie de classes
- Méthode appelée : correspond à l'objet instancié
- Avantage : manipuler toutes les instances d'une hiérarchie de classes de manière uniforme

Exemple (Jeux d'échecs)

Méthode **deplacer()** pour les pièces : Roi, Reine, Fou, Cavalier Tour et Pion (héritant chacun du type Pièce).

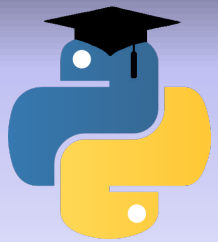


Principe

- Plusieurs méthodes avec le même nom dans une hiérarchie de classes
- Méthode appelée : correspond à l'objet instancié
- Avantage : manipuler toutes les instances d'une hiérarchie de classes de manière uniforme

Exemple ()

```
# Déclaration d'une liste de pièces
liste_pieces = [Pion(), Cavalier()]
# Appliquer la méthode déplacer à chaque pièce
for piece in liste_pieces :
    piece.deplacer()
```

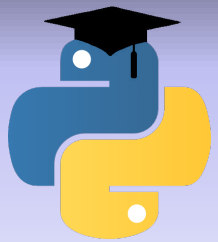


Principe

- Plusieurs méthodes avec le même nom dans une hiérarchie de classes
- Méthode appelée : correspond à l'objet instancié
- Avantage : manipuler toutes les instances d'une hiérarchie de classes de manière uniforme

Exemple (Figures 2D, cercles, rectangles)

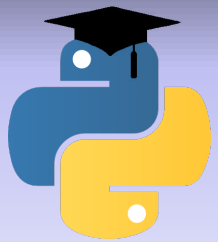
Méthode **aire()** : fournit la valeur l'aire pour Cercle et Rectangle.



Exemple

Exemple

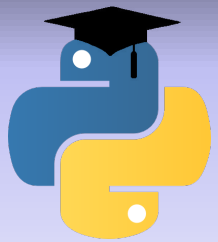
```
import numpy as np
class Figure (object):
      ...
class Cercle (Figure):
      ...
      def aire():
            return np.pi * self._rayon**2
class Rectangle (Figure):
      ... # avec 2 var d'instance self._h, et self._l
      def aire():
            return self._h * self._l
```



Utilisation des figures

Exemple

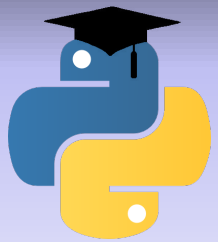
```
liste_figures = []
liste_figures.append(Cercle(10, 11, 2))
```



Utilisation des figures

Exemple

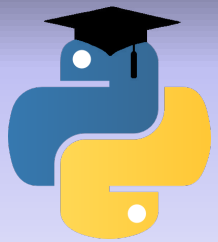
```
liste_figures = []
liste_figures.append(Cercle(10, 11, 2))
liste_figures.append(Rectangle(10,5, 3, 4))
....
```



Utilisation des figures

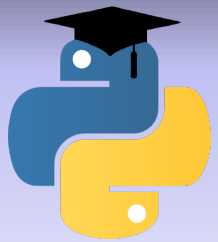
Exemple

```
liste_figures = []
liste_figures.append(Cercle(10, 11, 2))
liste_figures.append(Rectangle(10,5, 3, 4))
....

for i in range(len(liste_figures)):
    print(liste_figures[i].aire())
```



Exemple de surcharge : méthode __str__

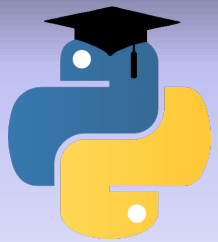
- Méthode appelée pour convertir un objet en chaîne de caractères



Exemple de surcharge : méthode __str__

- Méthode appelée pour convertir un objet en chaîne de caractères

$\Rightarrow$ afficher directement les objets



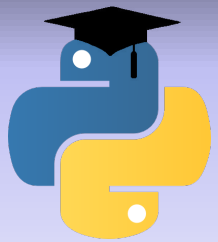
Exemple de surcharge : méthode `__str__`

- Méthode appelée pour convertir un objet en chaîne de caractères

⇒ afficher directement les objets

- Prédéfinie dans la classe `object`

Exemple vu (le type Intervalle) : l'affichage d'un objet `Intervalle`



Exemple de surcharge : méthode __str__

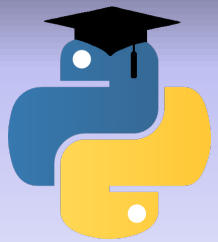
- Méthode appelée pour convertir un objet en chaîne de caractères

⇒ afficher directement les objets

- Prédéfinie dans la classe `object`

Exemple vu (le type Intervalle) : l'affichage d'un objet `Intervalle`
Exemple d'affichage d'une figure **f** :

```
f = Figure(3,5)
print(f)
```



Exemple de surcharge : méthode __str__

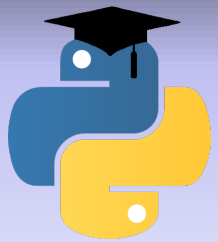
- Méthode appelée pour convertir un objet en chaîne de caractères

⇒ afficher directement les objets

- Prédéfinie dans la classe `object`

Exemple vu (le type Intervalle) : l'affichage d'un objet `Intervalle`
Exemple d'affichage d'une figure **f** :

```
f = Figure(3,5)
print(f)
<__main__.Figure object at 0x000000000575DBA8>
```



Exemple de surcharge : méthode __str__

- Méthode appelée pour convertir un objet en chaîne de caractères

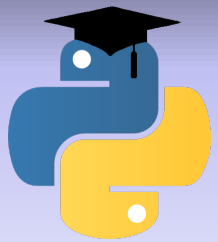
⇒ afficher directement les objets

- Prédéfinie dans la classe `object`

Exemple vu (le type Intervalle) : l'affichage d'un objet `Intervalle`
Exemple d'affichage d'une figure **f** :

```
f = Figure(3,5)
print(f)
```

```
def __str__(self):
    return "Fig. Pos = ({0},{1})".format(self._abscisse,
self._ordonnee)
```



Exemple de surcharge : méthode __str__

- Méthode appelée pour convertir un objet en chaîne de caractères

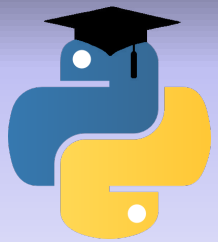
⇒ afficher directement les objets

- Prédéfinie dans la classe `object`

Exemple vu (le type Intervalle) : l'affichage d'un objet `Intervalle`
Exemple d'affichage d'une figure **f** :

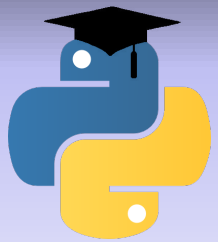
```
f = Figure(3,5)
print(f)
Fig. Pos = (3,5)
```

```
def __str__(self):
    return "Fig. Pos = ({0},{1})".format(self._abscisse,
self._ordonnee)
```

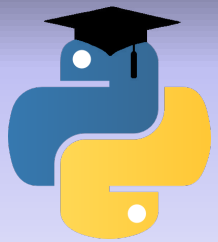


Quelques méthodes à surcharger

Méthode de la classe à surcharger	Utilisation
object.__str__(self)	`print(self)`
object.__repr__(self)	`self`
object.__add__(self, other)	`self + other`
object.__sub__(self, other)	`self - other`
object.__mul__(self, other)	`self * other`
object.__and__(self, other)	`self and other`
object.__or__(self, other)	`self or other`
object.__len__(self)	`len(self)`
object.__getitem__(self,i)	`x = self[i]`
object.__setitem__(self, i)	`self[i]= y`

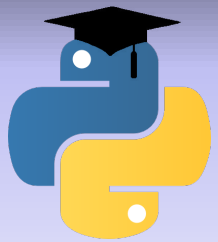


Sommaire



Principe

- Variables et méthodes propriétés de la classe, pas de l'instance
- $\Rightarrow$ un seul exemplaire par classe
- Accessible depuis les classes et les instances
- Utilisation le décorateur `@classmethod` avec comme premier argument **cls** et non pas `self`

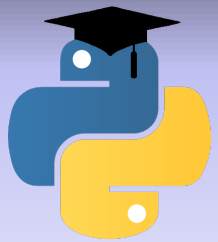


Principe

- Variables et méthodes propriétés de la classe, pas de l'instance
- $\Rightarrow$ un seul exemplaire par classe
- Accessible depuis les classes et les instances
- Utilisation le décorateur `@classmethod` avec comme premier argument **cls** et non pas `self`

Remarque

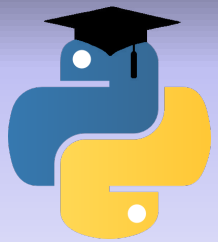
On peut utiliser la fonction built-in `classmethod(methodeClasse)` pour définir la méthode methodeClass() comme méthode de classe.



Exemple

Exemple

Comment compter les figures créées dans la classe **Figure** (et ses classes filles) ou attribuer un numéro unique à chaque objet ?

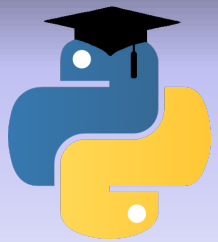


Exemple

Exemple

Comment compter les figures créées dans la classe **Figure** (et ses classes filles) ou attribuer un numéro unique à chaque objet ?

```
class Figure (object):
    nb_figure = 0 # variable de classe : compteur
    def __init__(self, x, y):
        self._abscisse = x
        self._ordonnee = y
```

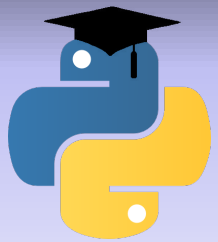


Exemple

Exemple

Comment compter les figures créées dans la classe **Figure** (et ses classes filles) ou attribuer un numéro unique à chaque objet ?

```
class Figure (object):
   nb_figure = 0 # variable de classe : compteur
   def __init__(self, x, y):
       self._abscisse = x
       self._ordonnee = y
       Figure.nb_figures += 1
   @classmethod
   def get_nbfigures(cls):
         return Figure.nb_figures
```

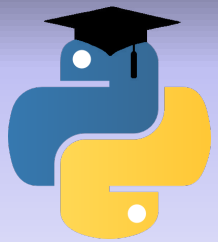


Exemple

Exemple

Comment compter les figures créées dans la classe **Figure** (et ses classes filles) ou attribuer un numéro unique à chaque objet ?

```
class Figure (object):
   nb_figure = 0 # variable de classe : compteur
   def __init__(self, x, y):
       self._abscisse = x
       self._ordonnee = y
       Figure.nb_figures += 1
   @classmethod
   def get_nbfigures(cls):
         return Figure.nb_figures
```

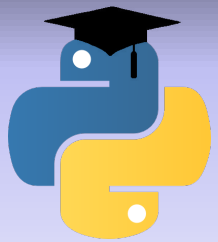


Exemple

Exemple

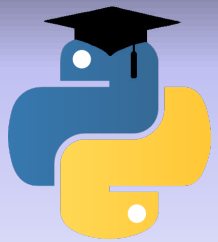
Comment compter les figures créées dans la classe **Figure** (et ses classes filles) ou attribuer un numéro unique à chaque objet ?

```
class Figure (object):
   nb_figure = 0 # variable de classe : compteur
   def __init__(self, x, y):
       self._abscisse = x
       self._ordonnee = y
       Figure.nb_figures += 1
   @classmethod
   def get_nbfigures(cls):
         return Figure.nb_figures
```



Exemple

```
x = Figure (2, 3)
print(Figure.get_nbfigures())
```

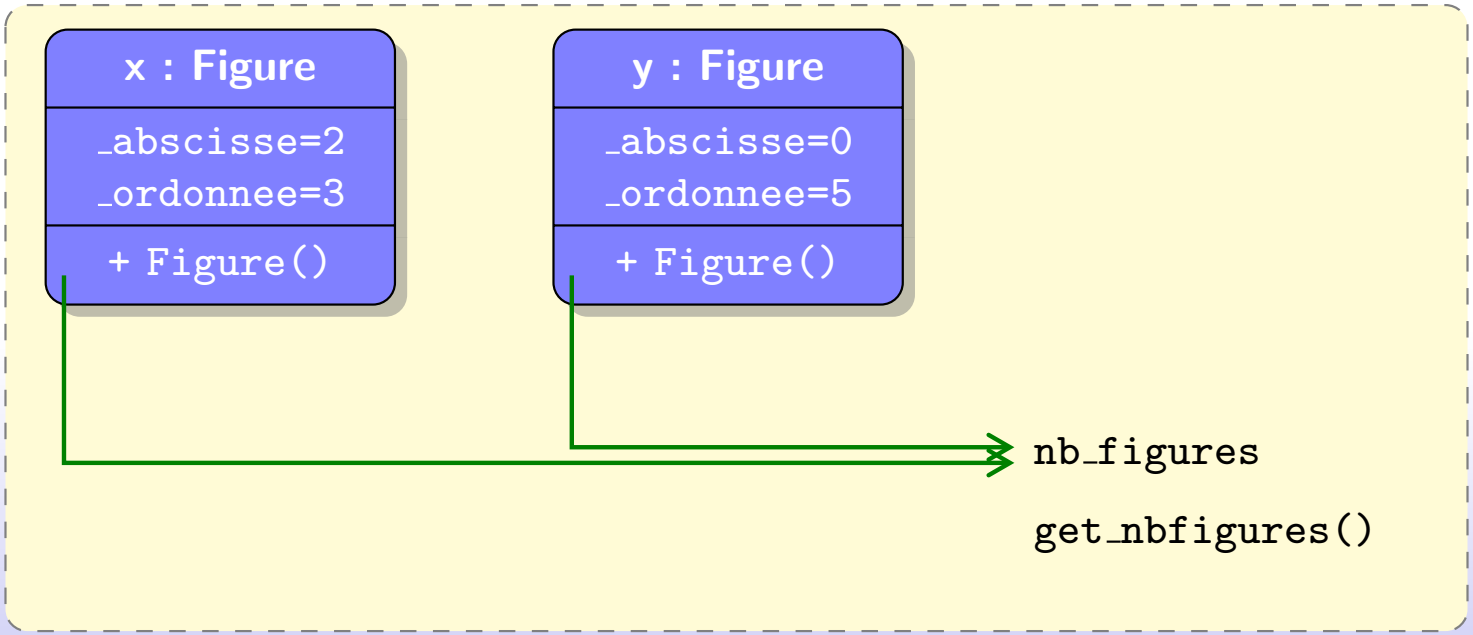


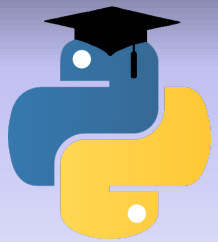
Exemple

```
x = Figure (2, 3)
print(Figure.get_nbfigures()) # => 1
y = Figure (0, 5)
print(Figure.get_nbfigures())
```

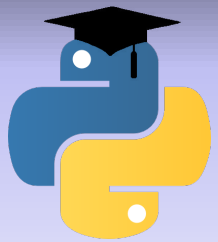
Exemple

```
x = Figure (2, 3)
print(Figure.get_nbfigures()) # => 1
y = Figure (0, 5)
print(Figure.get_nbfigures()) # => 2)
```

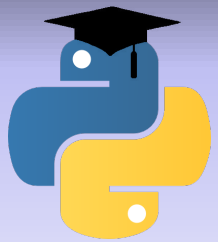




Sommaire



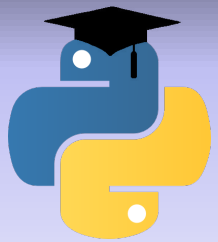
Sommaire



Méthodes abstraites

Définition (Méthode abstraite)

Une méthode est abstraite si sa déclaration est précédée par le décorateur **@abstractmethod**. Elle ne contient alors aucun code



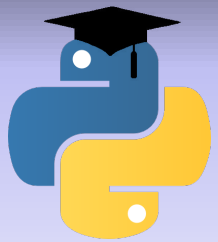
Méthodes abstraites

Définition (Méthode abstraite)

Une méthode est abstraite si sa déclaration est précédée par le décorateur **@abstractmethod**. Elle ne contient alors aucun code

Exemple (Dans la classe Figure)

```
from abc import abstractmethod
class Figure(object):
    ...
    @abstractmethod
    def aire(self):
        """La méthode aire est déclarée abstraite.
        """
        pass
```



Méthodes abstraites

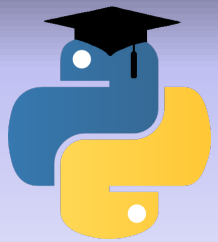
Définition (Méthode abstraite)

Une méthode est abstraite si sa déclaration est précédée par le décorateur **@abstractmethod**. Elle ne contient alors aucun code Il n'est pas possible de l'appeler ssi :

- Sa classe est définie comme **classe abstraite** $\Rightarrow$ définir dans les sous-classes.

Exemple (Dans la classe Figure)

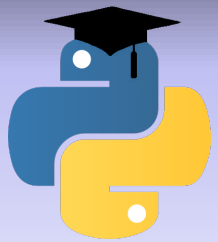
```
from abc import abstractmethod ABCMeta
class Figure(metaclass=ABCMeta):
    ...
    @abstractmethod
    def aire(self):
        """La méthode aire est déclarée abstraite.
        """
        pass
```



Héritage d'une classe abstraite

Héritage

- Une sous-classe d'une classe abstraite peut être concrète si elle redéfinit toutes les méthodes abstraites. Sinon, elle est considérée abstraite
- Exemple : **aire** est redéfinie dans **Cercle** et **Rectangle**



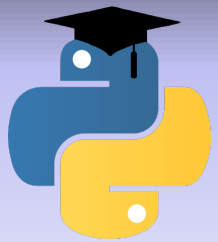
Héritage d'une classe abstraite

Héritage

- Une sous-classe d'une classe abstraite peut être concrète si elle redéfinit toutes les méthodes abstraites. Sinon, elle est considérée abstraite
- Exemple : **aire** est redéfinie dans **Cercle** et **Rectangle**

Intérêt :

- Déclarer explicitement qu'une classe n'est pas instanciable
- Définir les prototypes des méthodes dans la super-classe



Héritage d'une classe abstraite

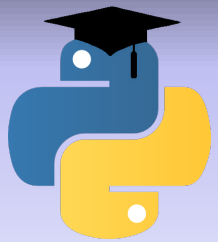
Héritage

- Une sous-classe d'une classe abstraite peut être concrète si elle redéfinit toutes les méthodes abstraites. Sinon, elle est considérée abstraite
- Exemple : **aire** est redéfinie dans **Cercle** et **Rectangle**

Intérêt :

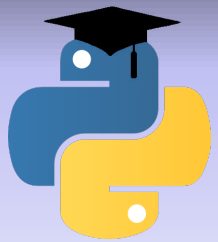
- Déclarer explicitement qu'une classe n'est pas instanciable
- Définir les prototypes des méthodes dans la super-classe

Exemple : Liste de **Figure**



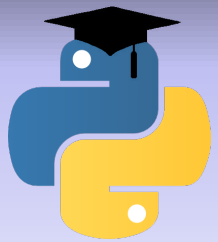
Exemple de classe abstraite

```
from abc import abstractmethod, ABCMeta
class Figure(metaclass = ABCMeta):
      @abstractmethod
      def aire(self):
            pass
class Cercle (Figure) :
      ...
      def aire(self):
            return np.pi * self._rayon**2
      ...
```



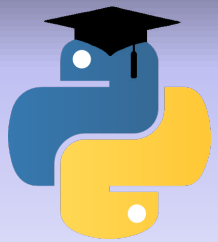
Exemple de classe abstraite

```
from abc import abstractmethod, ABCMeta
class Figure(metaclass = ABCMeta):
      @abstractmethod
      def aire(self):
            pass
class Cercle (Figure) :
      ...
      def aire(self):
            return np.pi * self._rayon**2
      ...
class Rectangle (Figure) :
      ...
      def aire(self):
            return self._h *self._l
      ...
```



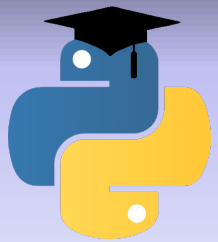
Exemple de classe abstraite

```
from abc import abstractmethod, ABCMeta
l_figures = []
# création d'un objet Cercle et ajout à la liste
l_figures.append(Cercle(10, 11, 1))
# création d'un objet Rectangle et ajout à la liste
l_figures.append(Rectangle(10,5, 3, 4))
# tentation d'instancier la classe abstraire Figure
```



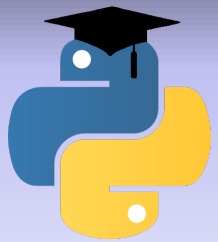
Exemple de classe abstraite

```
from abc import abstractmethod, ABCMeta
l_figures = []
# création d'un objet Cercle et ajout à la liste
l_figures.append(Cercle(10, 11, 1))
# création d'un objet Rectangle et ajout à la liste
l_figures.append(Rectangle(10,5, 3, 4))
# tentation d'instancier la classe abstraire Figure
l_figures.append(Figure(2,0))
```



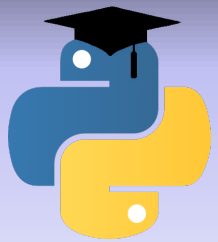
Exemple de classe abstraite

```
from abc import abstractmethod, ABCMeta
l_figures = []
# création d'un objet Cercle et ajout à la liste
l_figures.append(Cercle(10, 11, 1))
# création d'un objet Rectangle et ajout à la liste
l_figures.append(Rectangle(10,5, 3, 4))
# tentation d'instancier la classe abstraire Figure
l_figures.append(Figure(2,0)) # => Error
```



Exemple de classe abstraite

```
from abc import abstractmethod, ABCMeta
l_figures = []
# création d'un objet Cercle et ajout à la liste
l_figures.append(Cercle(10, 11, 1))
# création d'un objet Rectangle et ajout à la liste
l_figures.append(Rectangle(10,5, 3, 4))
# tentation d'instancier la classe abstraire Figure
l_figures.append(Figure(2,0)) # => Error
# affichage de l'aire des objets
```



Exemple de classe abstraite

```
from abc import abstractmethod, ABCMeta
l_figures = []
# création d'un objet Cercle et ajout à la liste
l_figures.append(Cercle(10, 11, 1))
# création d'un objet Rectangle et ajout à la liste
l_figures.append(Rectangle(10,5, 3, 4))
# tentation d'instancier la classe abstraire Figure
l_figures.append(Figure(2,0)) # => Error
# affichage de l'aire des objets
for f in l_figures :
      print(f.aire())
```