Partie 11 :
Programmation Orientée Objet
Aspects Élémentaires

Syllabus & Exercices

11. POO : Aspects Élémentaires

- 11-01 : Introduction
  - Module
  - Module Math
  - Module Turtle
  - Méthodes
  - Help
- 11-02 : Objet simple
  - Objet
  - Conception et modélisation d'une classe
  - Méthode : constructeur, `__init__`
  - Manipulation d'un objet
  - `self`
  - Références
  - Méthode : destructeur
  - Égalité et copie d'un objet
  - Méthode : `__str__`
- 11-03 : attribut public ou privé
  - Attributs privés vs publics
  - Méthodes accesseurs
  - `@property & @my_attr.setter`
- 11-04 : variable et méthode de classe
  - Variable de classe (statique)
  - Méthode de classe
  - `@classmethod`

🏛 → Orienté Objet → 11-POO-basic

= BUROTIX ()

Chapitre 11-01 : introduction

= BUROTIX ()

Modules

Un **module** est un fichier Python contenant
un ensemble de définitions et d'instructions

La **librairie standard Python** offre un **vaste choix de modules**

texte, calcul, fichiers, réseau, web, graphique, système, …

```
>>> help("modules")
```

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

On peut **installer des modules** supplémentaires

pip, Thonny package manager

Pour utiliser le module `turtle` dans un programme :

```
import turtle
```

Module math

```
import math
print(math.pi)
                    3.141592653589793
print(math.cos(math.pi/3))
                    0.5000000000000001
print(math.cos(math.radians(60)))
                    0.5000000000000001
print(math.sqrt(2.0))
                    1.4142135623730951
print(math.log(math.e))
                    1.0
```

Accéder au module **math**

help(math)

```
>>> help(math)
 Help on module math:
NAME
   math
DESCRIPTION
   This module is always available.  It provides access to the
   mathematical functions defined by the C standard.
FUNCTIONS
   acos(...)
      acos(x)
      Return the arc cosine (measured in radians) of x.
acosh(...)
      acosh(x)
      Return the inverse hyperbolic cosine of x.
[ ..... ]
DATA
   e = 2.718281828459045
   inf = inf
   nan = nan
   pi = 3.141592653589793
   tau = 6.283185307179586
```

```
>>> help(math.sqrt)
Help on built-in function sqrt in module
math:
sqrt(...)
   sqrt(x)
   Return the square root of x.
```

```
>>> help(math.log)
Help on built-in function log in module math:
log(...)
   log(x[, base])
   Return the logarithm of x to the given
base.
   If the base not specified, returns the
natural logarithm (base e) of x.
```

Arrondi

- **math.ceil** : arrondi vers le haut

```
print(math.ceil(2.1))     3
print(math.ceil(2.9))     3
```

- **math.floor** : arrondi vers le bas

```
print(math.floor(2.1))    2
print(math.floor(2.9))    2
```

- **round** : arrondi au plus proche

```
print(round(2.1))         2
print(round(2.9))         3
```

Graphiques tortue

```
import turtle
tortue = turtle.Turtle()
tortue.color("blue")
for i in range(4):
    tortue.forward(50)
    tortue.right(90)
```

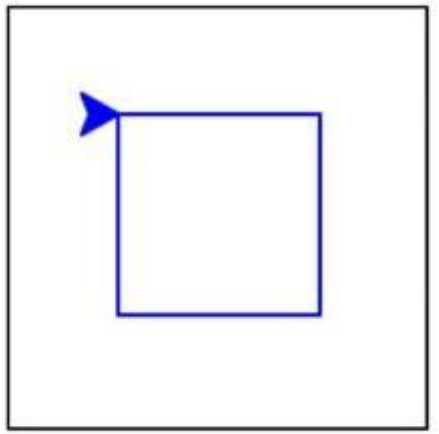
Module des graphiques tortue

Crée une nouvelle tortue (**objet**)

appelle une fonctionalité de la tortue (**méthode**)

idem.

idem.



Objets

Une tortue est un **objet** (de type `turtle.Turtle`)

Une **instance** de `turtle.Turtle`

Un **objet** est une structure qui associe

- des **données** (**attributs**)
  - position, orientation de la tortue
  - position haut/bas, couleur, largeur de la plume
  - …
- des **fonctionnalités** (**méthodes**)
  - avancer, reculer, tourner à gauche/droite
  - lever/baisser la plume, changer de couleur
  - remplir un contour
  - …

Méthodes

```
tortue.forward(50)
```

Exécute (invoque) la **méthode** forward
de **l'objet** tortue

Méthode = **fonction liée à un objet**

L'objet est un paramètre de la fonction :

```
def forward(self, distance): …
```

Méthodes de `Turtle`

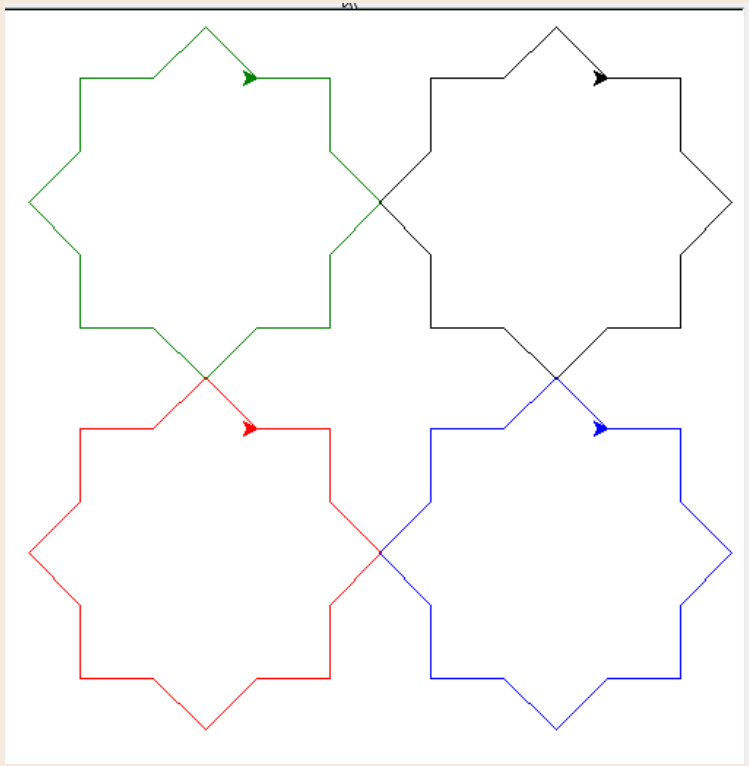
```
tortue.forward(10), tortue.backward(10)
tortue.left(45), tortue.right(45)
tortue.circle(20)
tortue.color("blue")
tortue.pendown(), tortue.penup()
tortue.begin_fill(), tortue.end_fill()
```

... et beaucoup d'autres

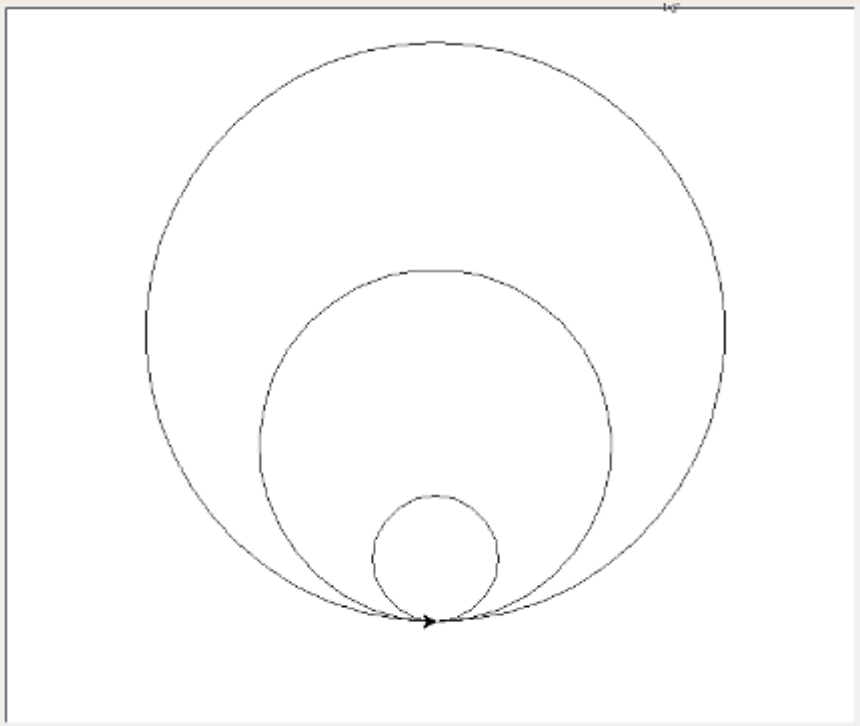
Voir `help(turtle)`

Exo 11-01-01 11-01-02 : la tortue

1. Dessinez des diamants



2. Dessinez des cercles concentriques

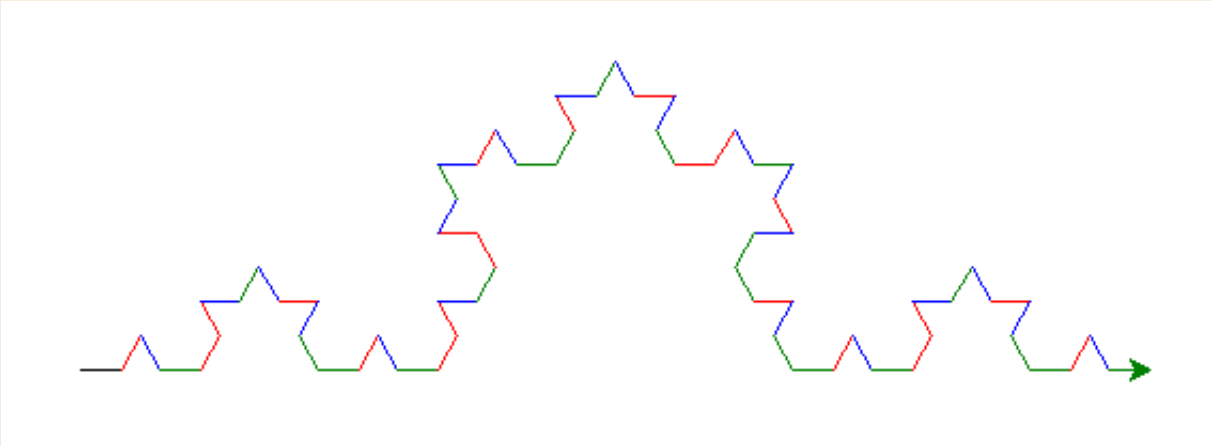


= BUROTIX ()

Exo 11-01-05 11-01-06 : les fractales de Koch

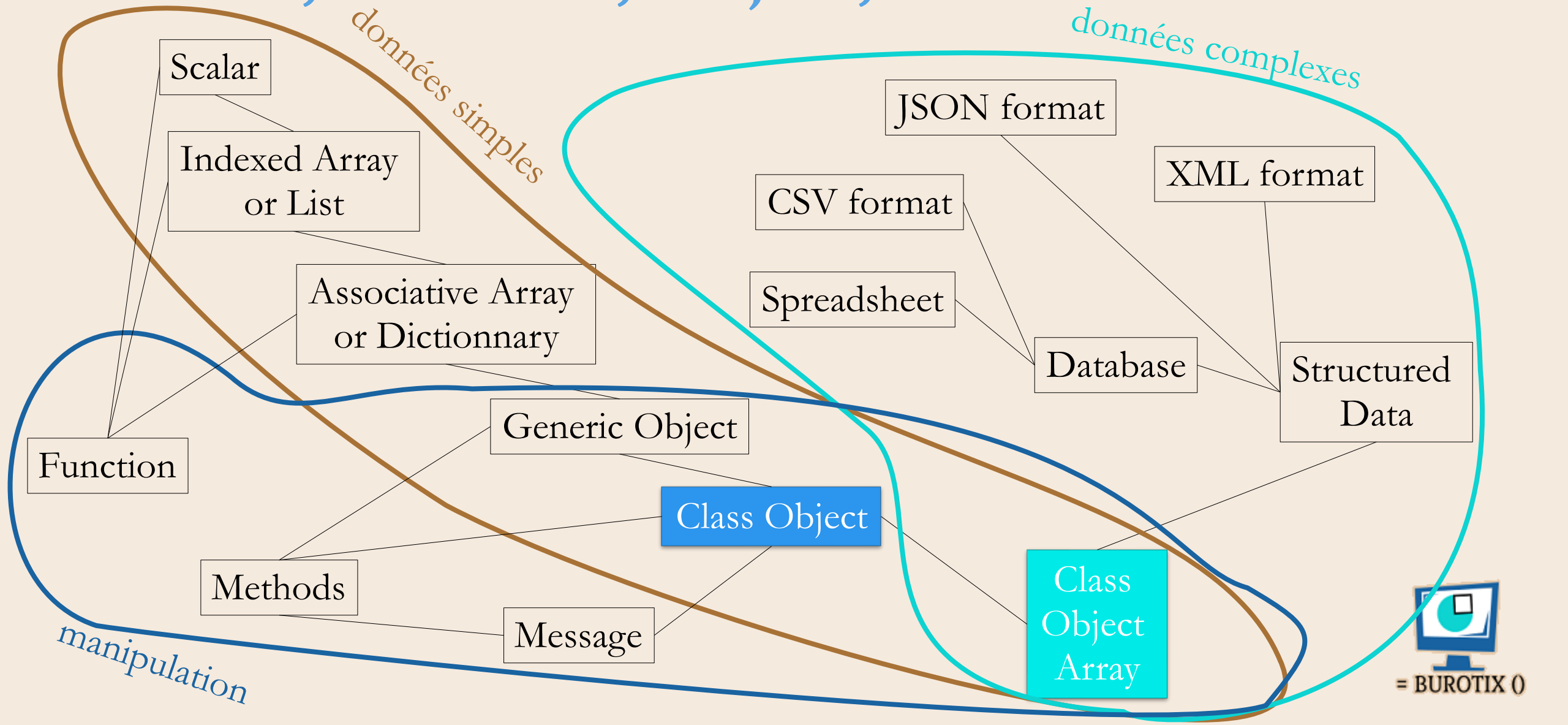
- Dessinez les fractales de Koch en utilisant la Tortue (exo 05)

- Le même code, conçu en POO (exo 06)



= BUROTIX ()

Scalar, Function, Object, Database



= BUROTIX ()

Chapitre 11-02 : l'objet simple

Qu'est-ce qu'un **objet** ?

- Fonctionnalités

Les mêmes pour tous les objets d'un même type

le vieux Nokia de Kim Mens

- Identité
- Etat

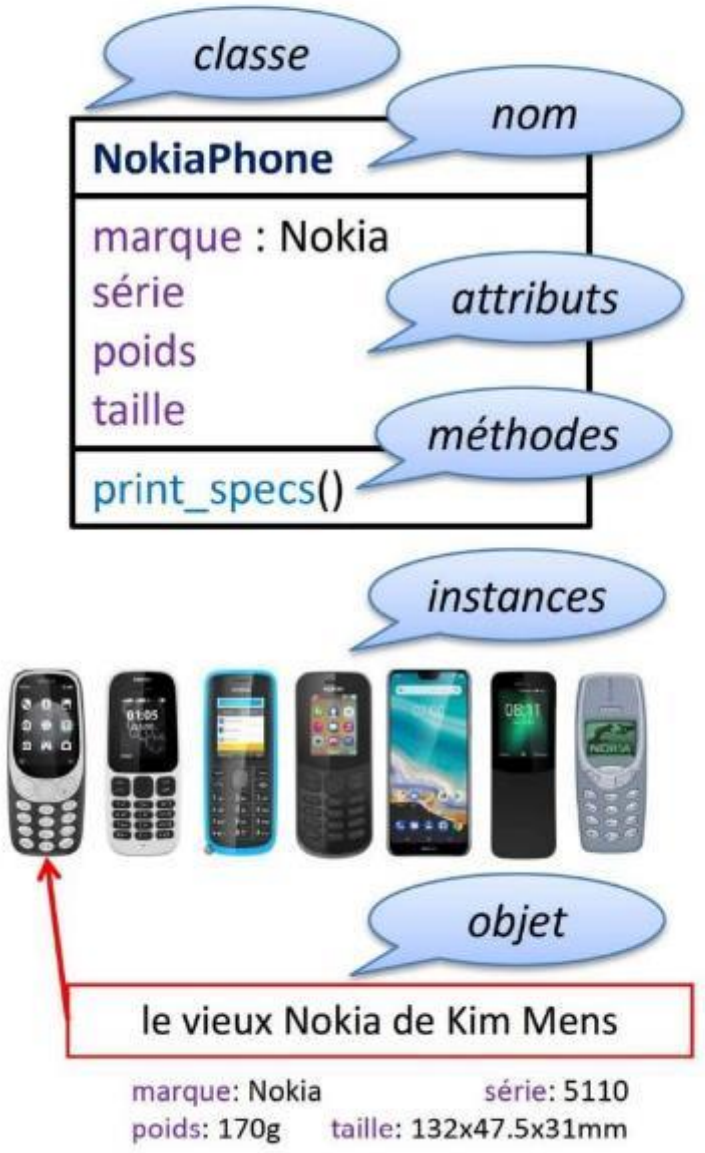
Spécifique à un objet particulier

BUROTIX ()

Qu'est-ce qu'une **classe** ?

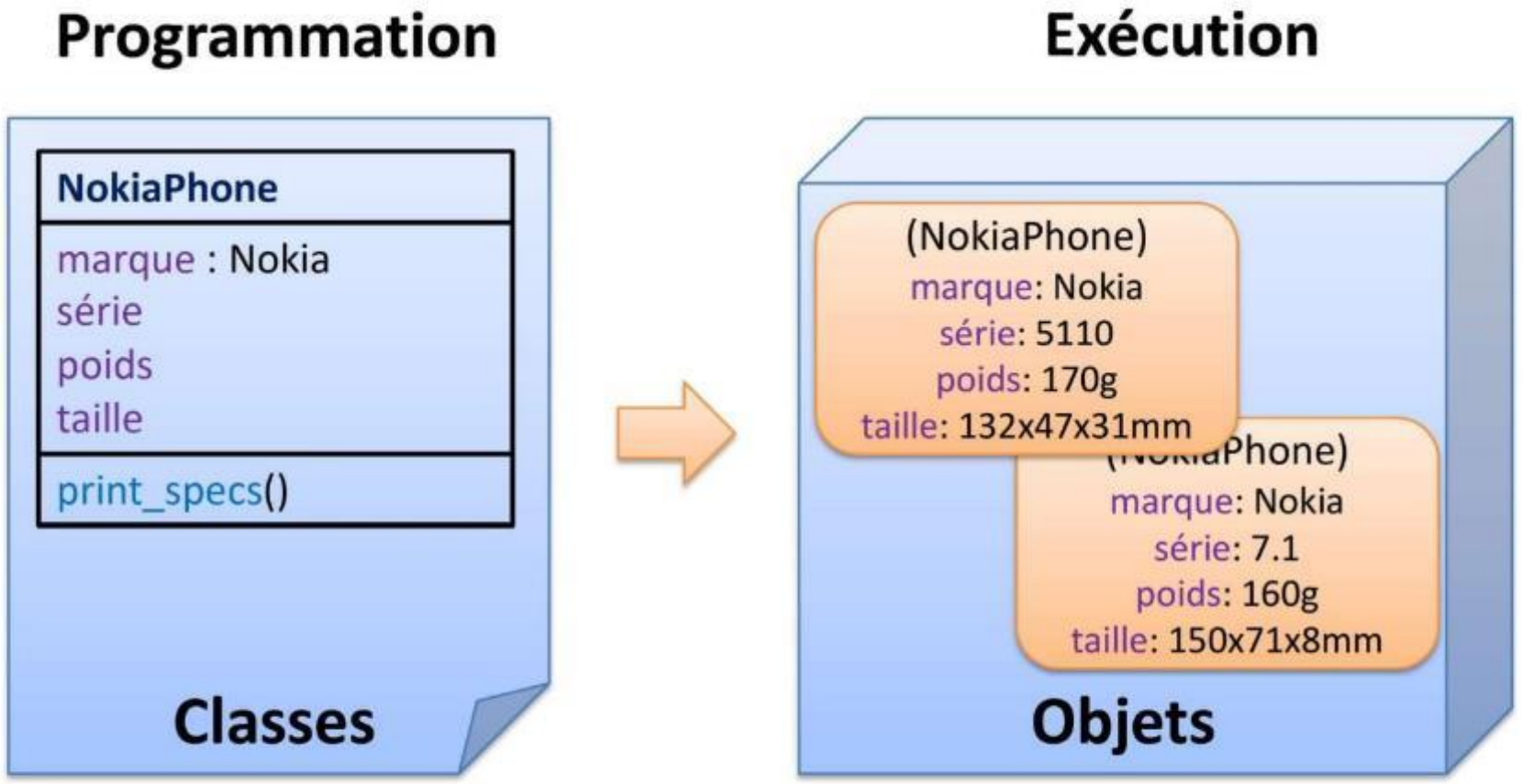
- Décrit les caractéristiques communes de tous les objets d'un même type
  - **Attributs** et valeurs initiales
  - **Méthodes** (fonctionnalité)

- Peut être vue comme une « usine » pour créer des objets de cette classe
  - **Instances**



BUROTIX ()

La programmation orientée objets



Une classe Python

classe

```
class NokiaPhone :

  def __init__(self,s,p,t) :
    self.marque = "Nokia"
    self.serie = s
    self.poids = p
    self.taille = t

  def print_specs(self) :
    print(self.marque + " " + str(self.serie))
    print("Poids:  " + str(self.poids) + " g")
    print("Taille: " + self.taille + " mm")
```

attributs ou
variables d'instance

méthode d'instance

BUROTIX ()

Création d'un objet

classe

```
class NokiaPhone :

  def __init__(self,s,p,t) :
    self.marque = "Nokia"
    self.serie = s
    self.poids = p
    self.taille = t

  …
```

NokiaPhone
marque : Nokia série poids taille
print_specs()

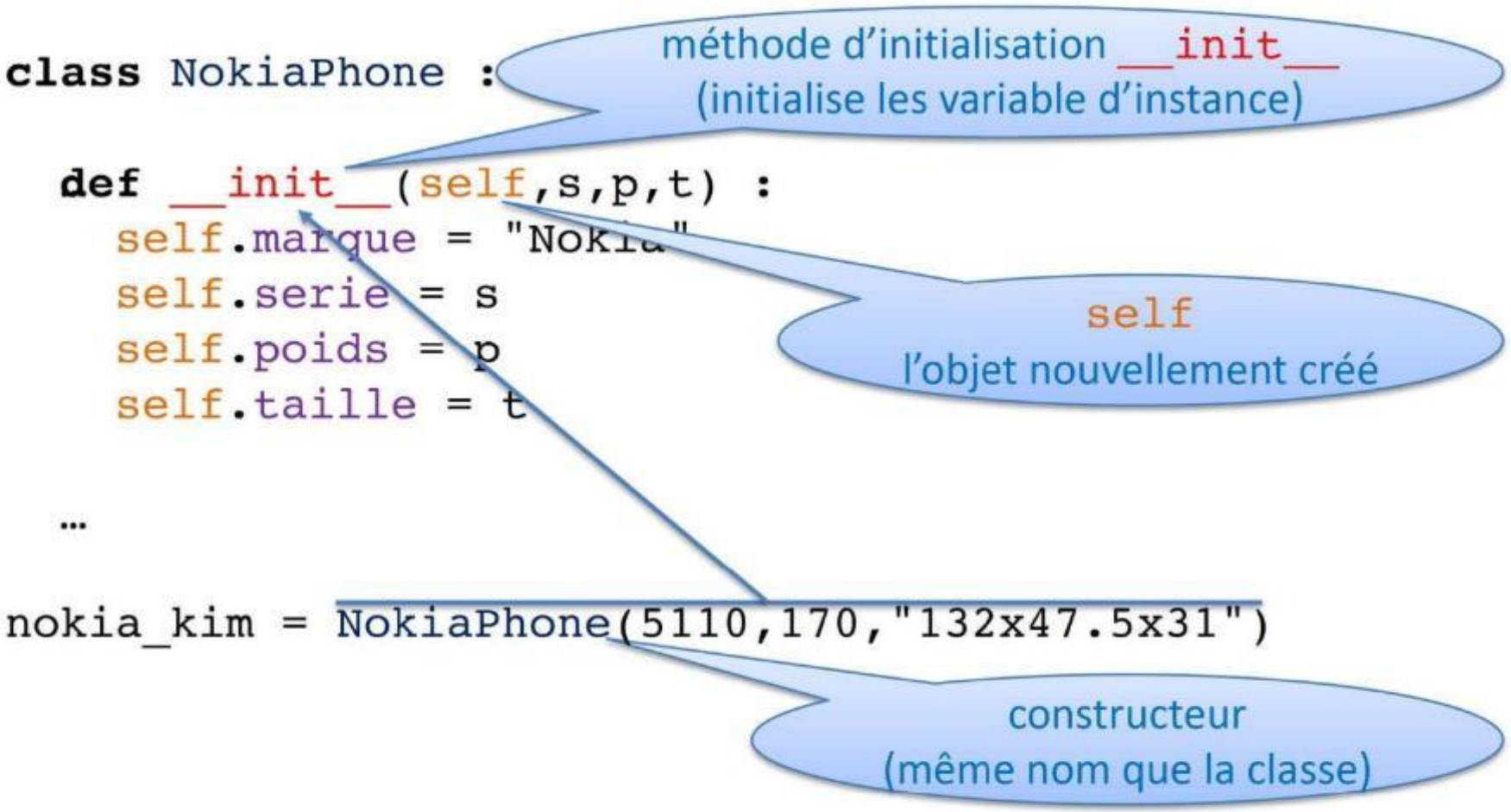
objet

```
nokia_kim = NokiaPhone(5110,170,"132x47.5x31")
```

(NokiaPhone)
marque: Nokia
série: 5110
poids: 170g
taille: 132x47x31mm

BUROTIX ()

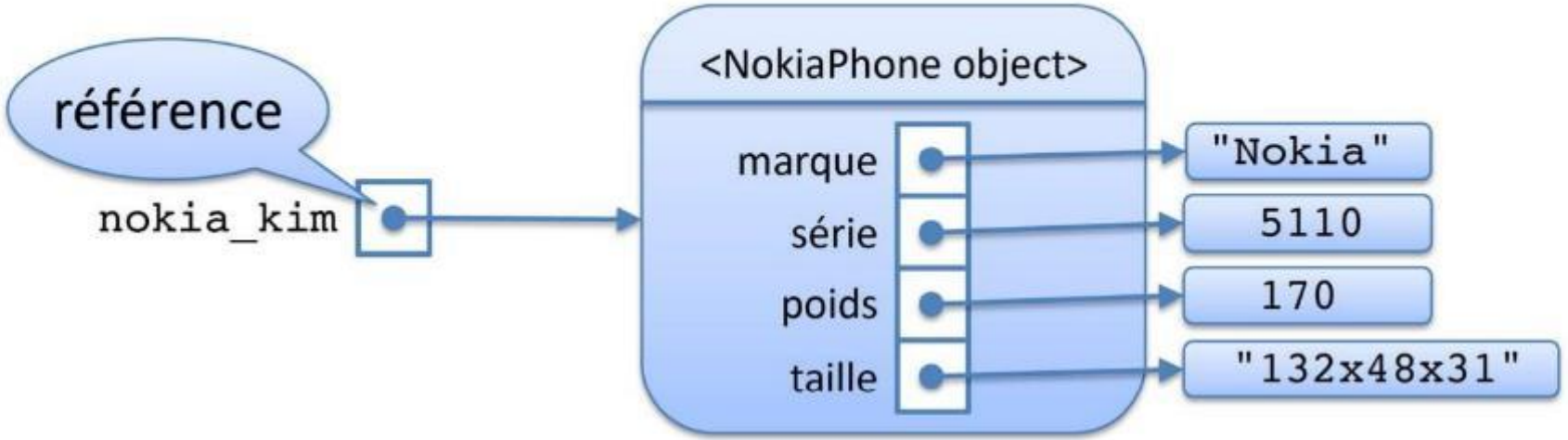
Création d'un objet



BUROTIX ()

Manipuler des objets

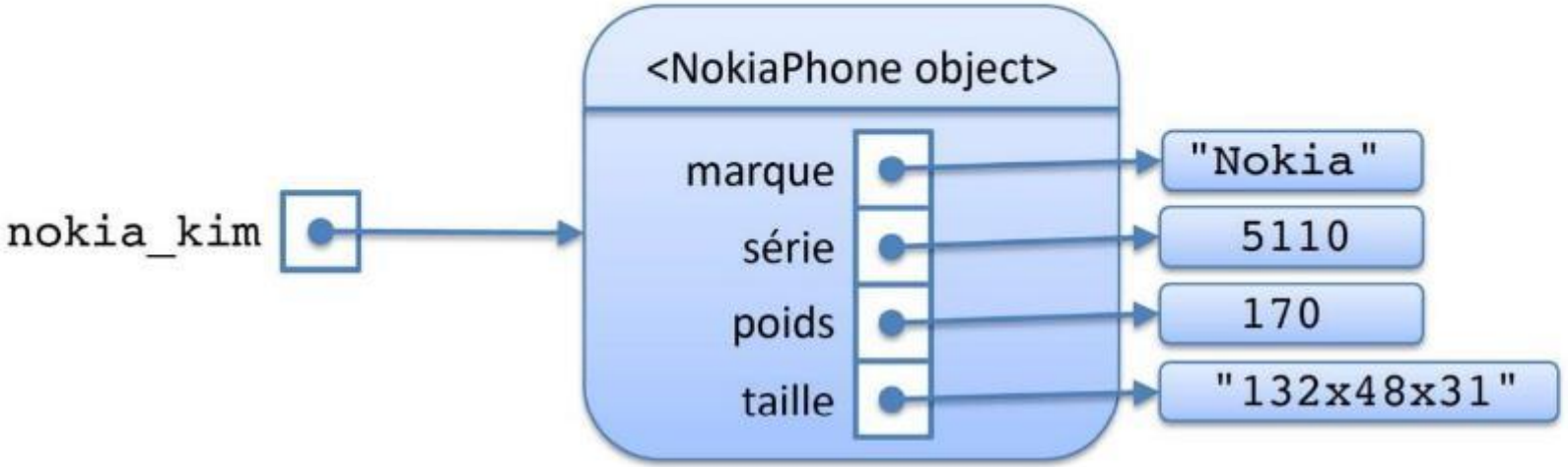
```
>>> nokia_kim = NokiaPhone(5110,170,"132x48x31")
>>> nokia_kim
<__main__.NokiaPhone object at 0x1042686a0>
```



BUROTIX ()

Manipuler des objets

```
>>> nokia_kim = NokiaPhone(5110,170,"132x47.5x31")
```




```
>>> nokia_kim.marque
'Nokia'
>>> nokia_kim.poids
170
```

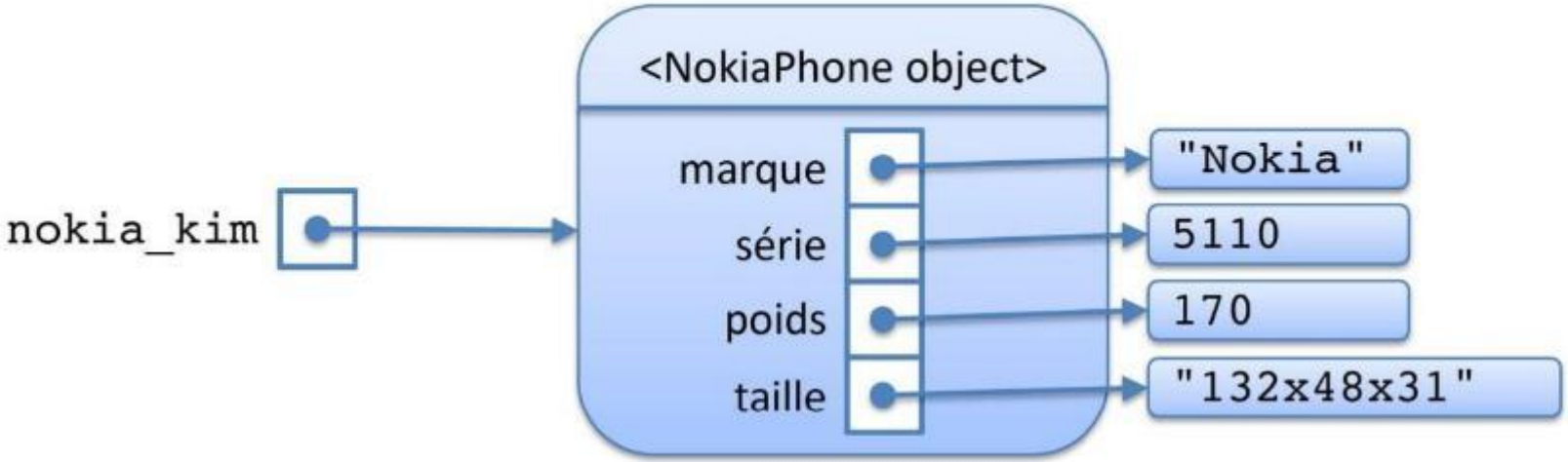
```
>>> nokia_kim.print_specs()
```

```
Nokia 5110
Poids:  170 g
Taille: 132x48x31 mm
```

BUROTIX ()

Manipuler des objets

```
>>> nokia_kim = NokiaPhone(5110,170,"132x47.5x31")
>>> nokia_kim.print_specs
<bound method NokiaPhone.print_specs of …>
```

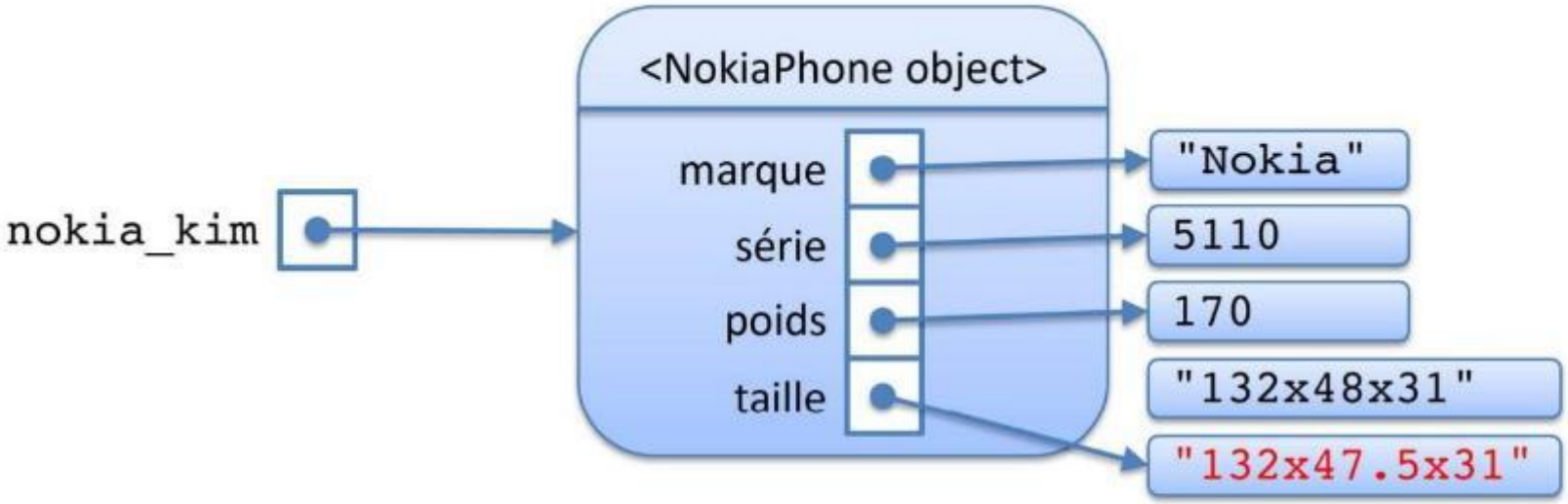



```
>>> nokia_kim.taille = "132x47.5x31"
```

BUROTIX ()

Manipuler des objets

```
>>> nokia_kim = NokiaPhone(5110,170,"132x47.5x31")
>>> nokia_kim.print_specs
<bound method NokiaPhone.print_specs of …>
```




```
>>> nokia_kim.taille = "132x47.5x31"
>>> nokia_kim.print_specs()
```

Nokia 5110
Poids: 170 g
Taille: 132x47.5x31 mm

BUROTIX ()

Appel d'une méthode

```
class NokiaPhone :

  def __init__(self,s,p,t) :
    self.marque = "Nokia"
    self.serie = s
    self.poids = p
    self.taille = t

  def print_specs(self) :
    print(self.marque + " " + str(self.serie))
    print("Poids:  " + str(self.poids) + " g")
    print("Taille: " + self.taille + " mm")

nokia_kim = NokiaPhone(5110,170,"132x48x31")
nokia_kim.print_specs()
```

self
le récepteur du message

```
Nokia 5110
Poids:  170 g
Taille: 132x48x31 mm
```

BUROTIX ()

Attention à ne pas oublier `self`

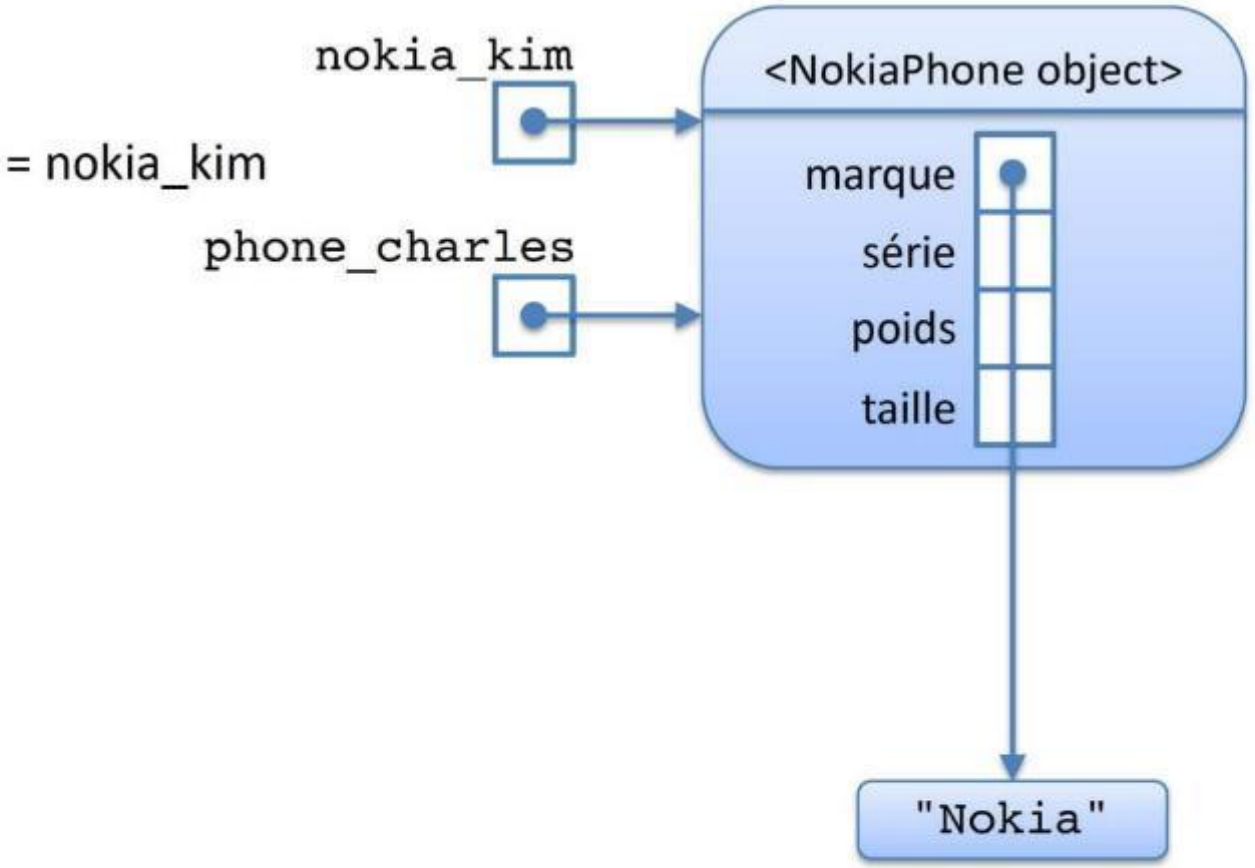
```
class NokiaPhone :

  def __init__(self,s,p,t) :
    self.marque = "Nokia"
    self.serie = s
    self.poids = p
    self.taille = t

  def print_specs(self) :
    print(self.marque + " " + str(self.serie))
    print("Poids:  " + str(self.poids) + " g")
    print("Taille: " + self.taille + " mm")

nokia_kim = NokiaPhone(5110,170,"132x48x31")
nokia_kim.print_specs()
```

sauf lors de l'appel même...

BUROTIX ()

Références

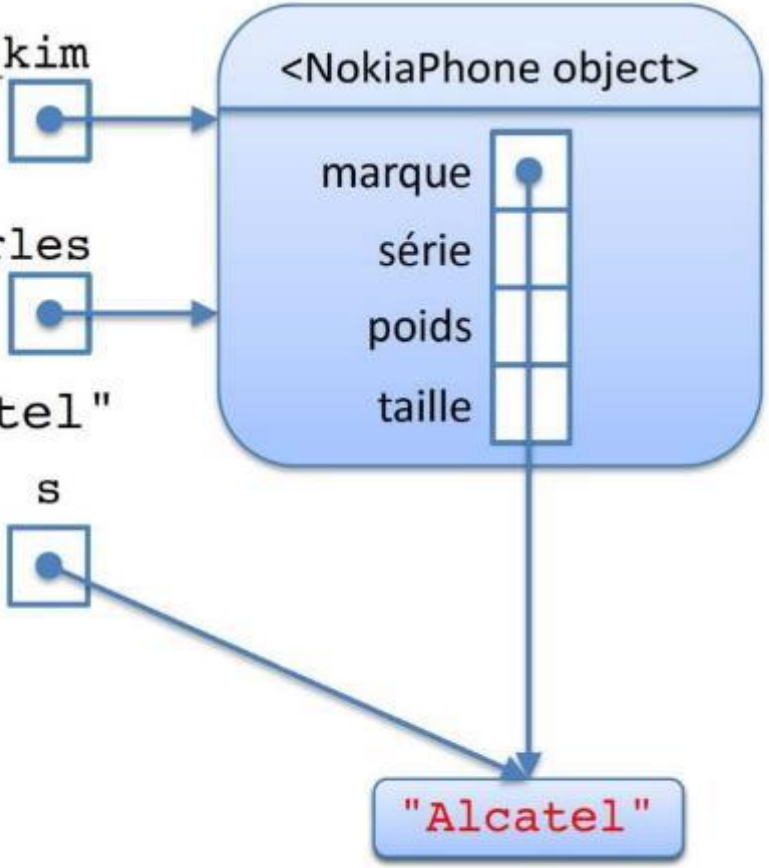
```
nokia_kim = NokiaPhone(5110,170,"132x48x31")

phone_charles = nokia_kim
```



BUROTIX ()

Références

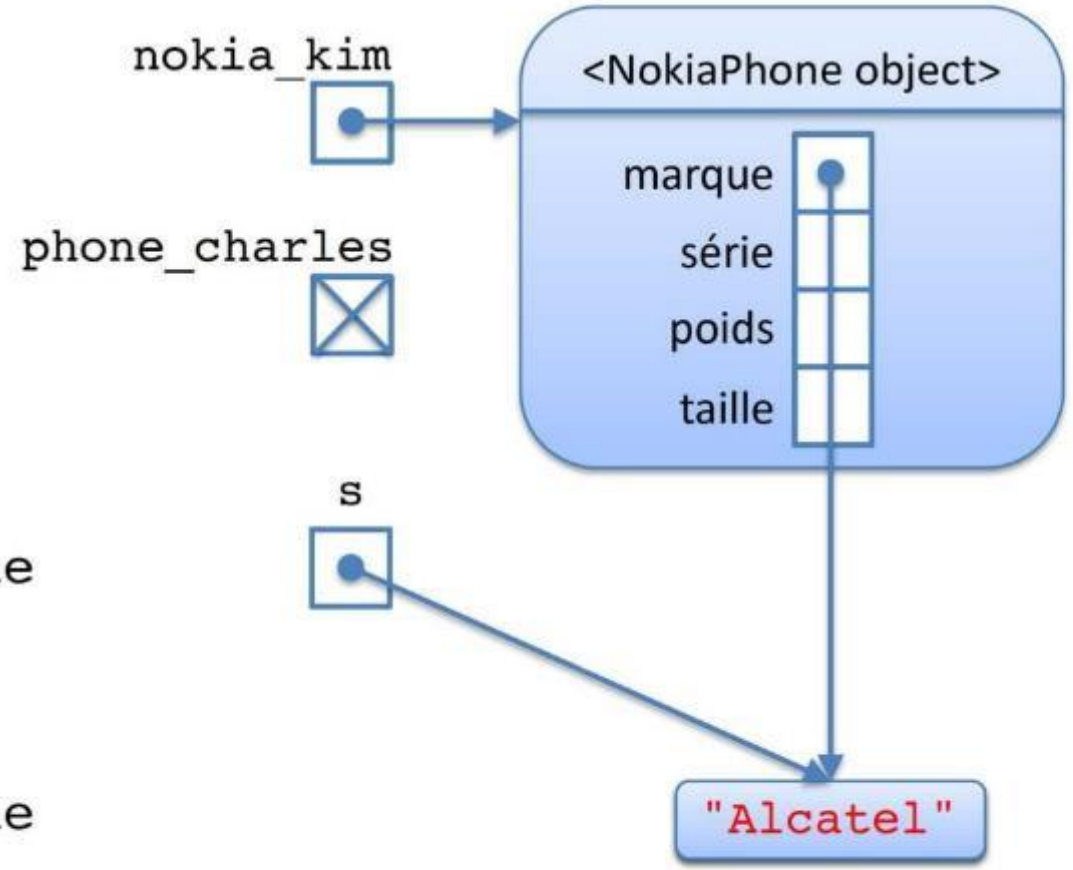
```
nokia_kim = NokiaPhone(5110,170,"132x48x31")

phone_charles = nokia_kim

phone_charles.marque = "Alcatel"

s = nokia_kim.marque

phone_charles = None
```



BUROTIX ()

Références

```
nokia_kim = NokiaPhone(5110,170,"132x48x31")
```




```
s = nokia_kim.marque
```

```
phone_charles = None
```

BUROTIX ()

Références

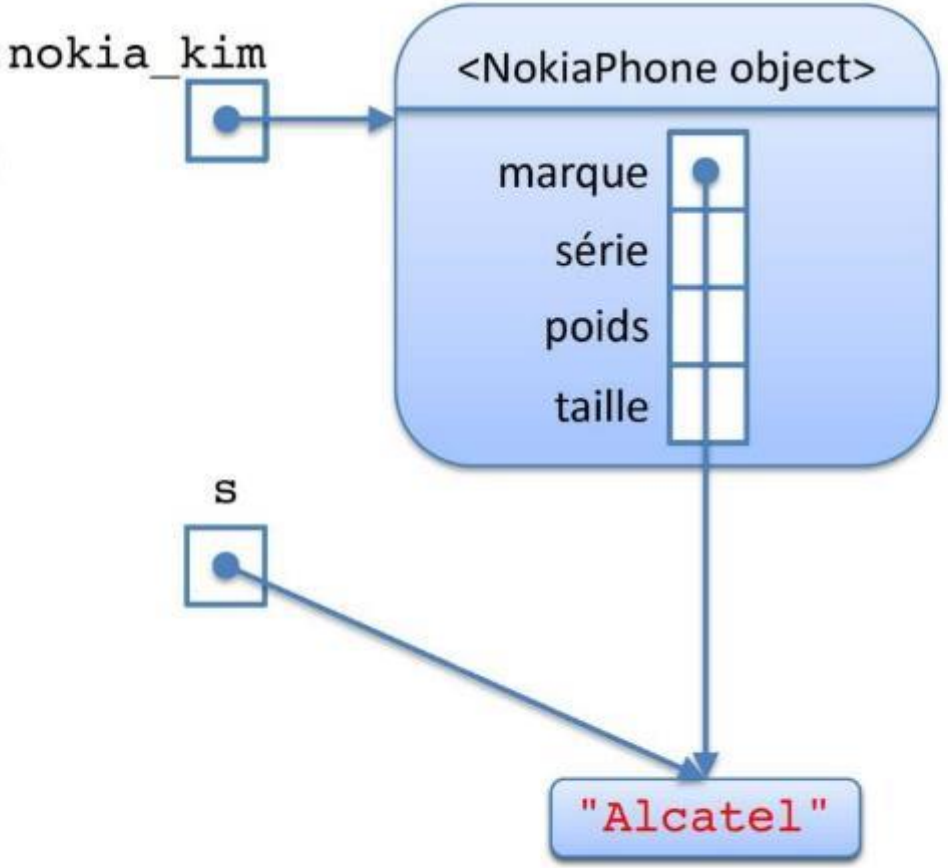
```
nokia_kim = NokiaPhone(5110,170,"132x48x31")
```

```
nokia_kim.print_specs()
```

```
Alcatel 5110
Poids:  170 g
Taille: 132x47.5x31 mm
```

```
s = nokia_kim.marque
```

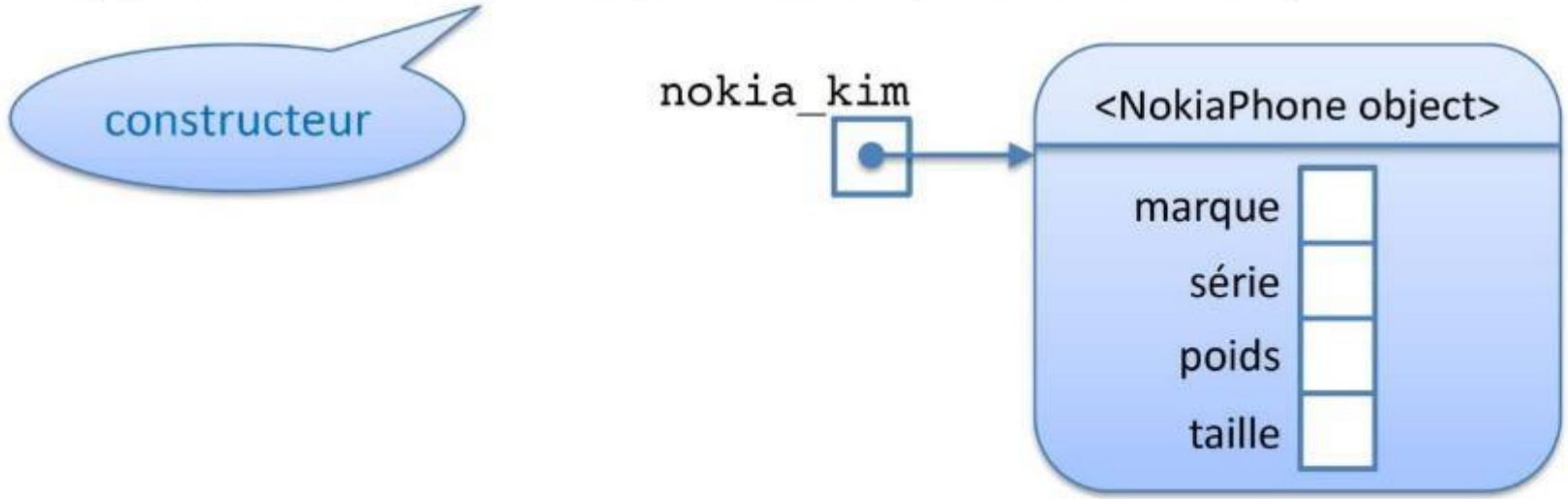
```
phone_charles = None
```



BUROTIX ()

Destructeurs ?

```
nokia_kim = NokiaPhone(5110,170,"132x48x31")
```

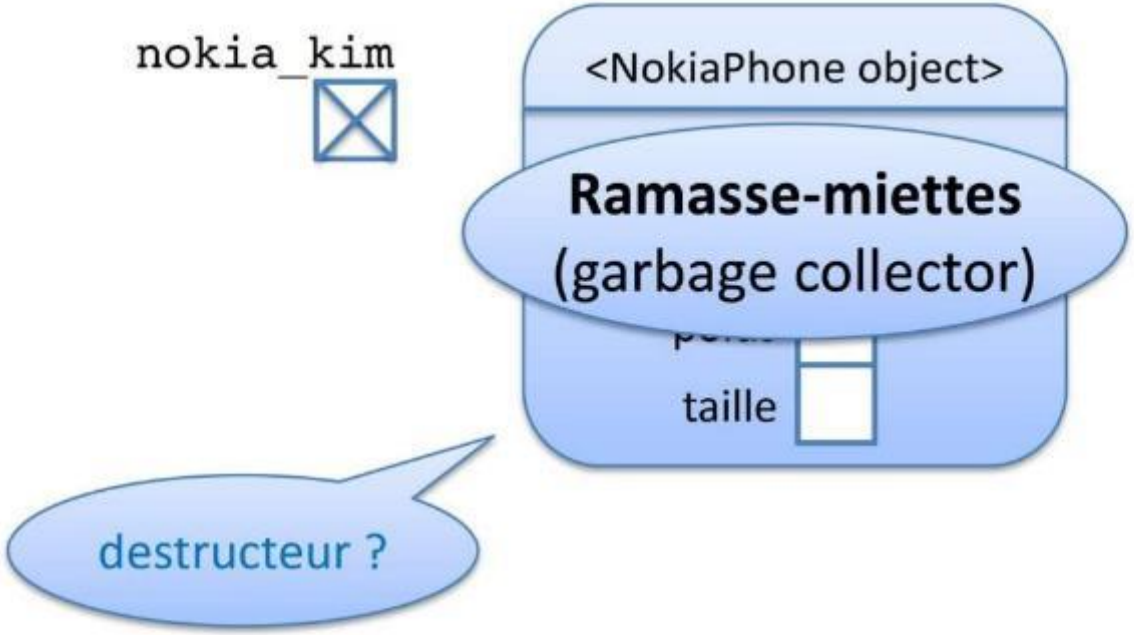



```
nokia_kim = None
```

BUROTIX ()

Destructeurs ?

```
nokia_kim = NokiaPhone(5110,170,"132x48x31")
```

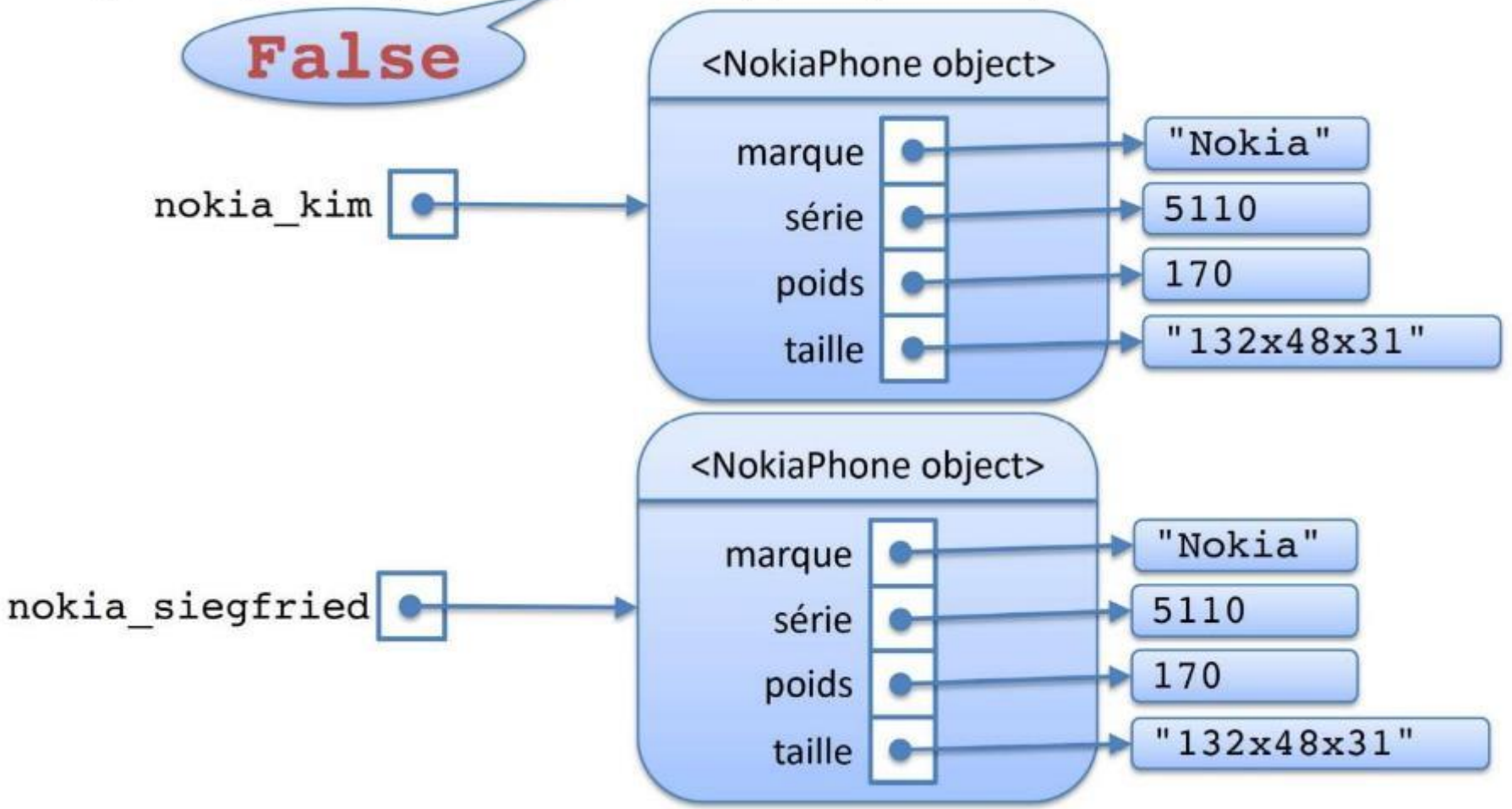



```
nokia_kim = None
```

BUROTIX ()

Egalité entre objets

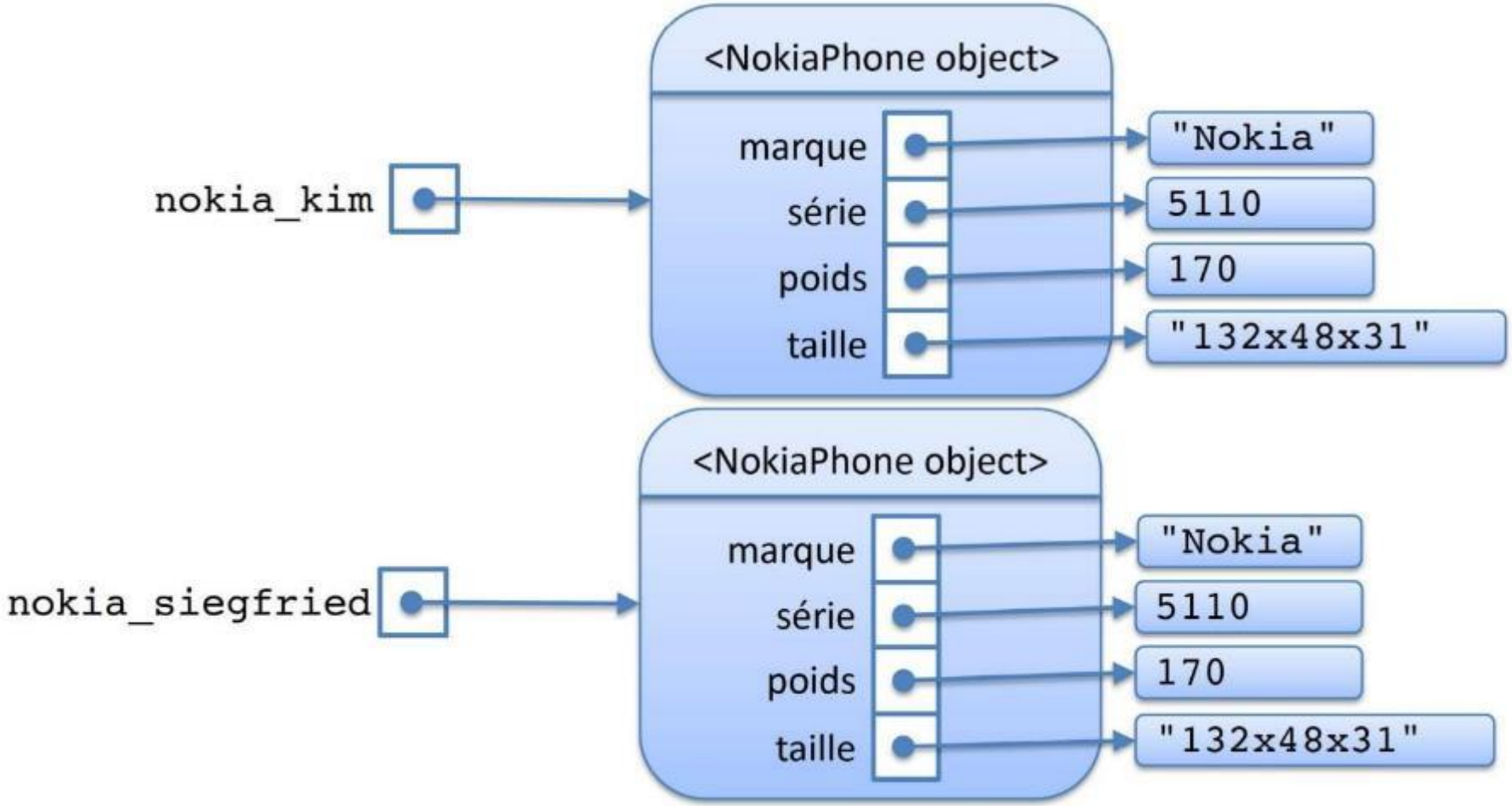
```
nokia_kim = NokiaPhone(5110,170,"132x48x31")
nokia_siegfried = NokiaPhone(5110,170,"132x48x31")
print(nokia_kim == nokia_siegfried)
```



BUROTIX ()

Copie d'un objet

```
import copy
nokia_kim = NokiaPhone(5110,170,"132x48x31")
nokia_siegfried = copy.copy(nokia_kim)
```



BUROTIX ()

La méthode magique __str__

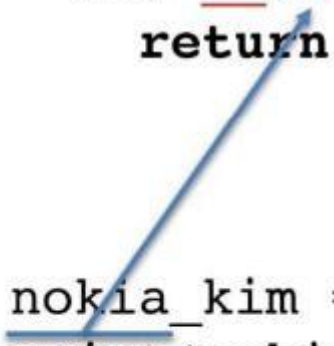
```
class NokiaPhone :

  def __init__(self,s,p,t) :
    self.marque = "Nokia"
    self.serie = s
    self.poids = p
    self.taille = t

  def print_specs(self) :
    print(self.marque + " " + str(self.serie))
    print("Poids:  " + str(self.poids) + " g")
    print("Taille: " + self.taille + " mm")

nokia_kim = NokiaPhone(5110,170,"132x48x31")
nokia_kim.print_specs()
```

```
Nokia 5110
Poids:  170 g
Taille: 132x48x31 mm
```

BUROTIX ()

La méthode magique __str__

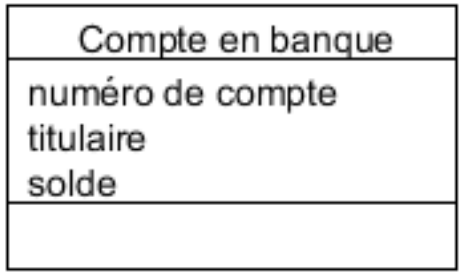
```
class NokiaPhone :

  def __init__(self,s,p,t) :
    self.marque = "Nokia"
    self.serie = s
    self.poids = p
    self.taille = t

  def __str__(self) :
    return self.marque + " " + str(self.serie) + "\n" \
           + "Poids:  " + str(self.poids) + " g" + "\n" \
           + "Taille: " + self.taille + " mm" + "\n"

nokia_kim = NokiaPhone(5110,170,"132x48x31")
print(nokia_kim)
```

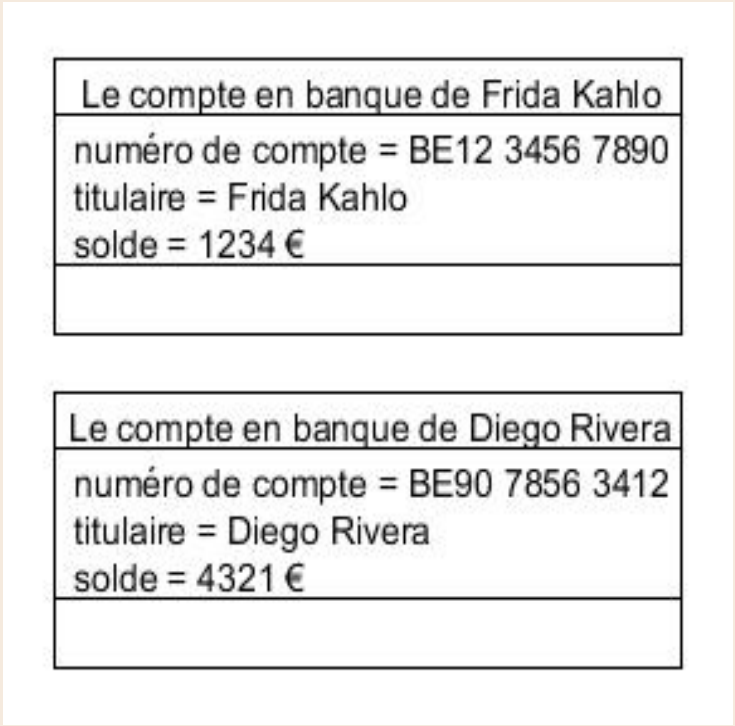


```
Nokia 5110
Poids:  170 g
Taille: 132x48x31 mm
```

BUROTIX ()

Classe vs. Objet

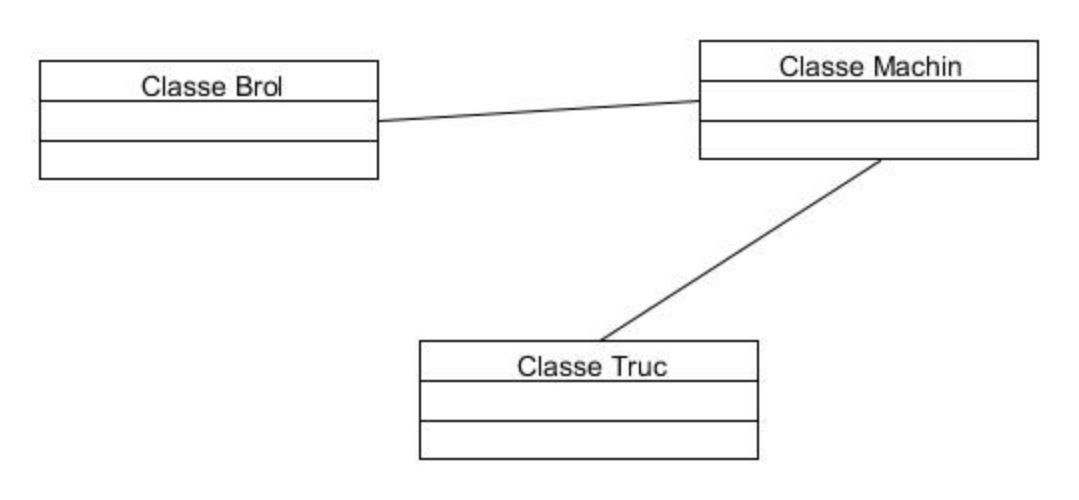




- Classe
  - type de l'objet
  - exemple : un compte en banque, dans un article de presse
- Objet
  - instance physique d'une classe
  - exemple : mon compte en banque en particulier
- Une classe définit un objet
- Une classe n'est pas un ensemble d'objets !

= BUROTIX ()

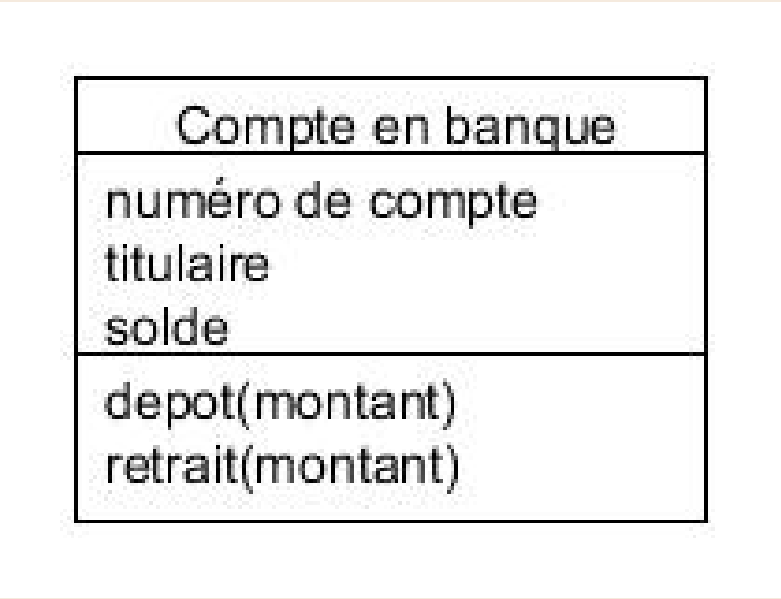
Une société de classes



- Une application OOP est une "société de classes"
  - chacune spécialisée dans ses tâches propres
  - collaborant entre elles par « messages »

= BUROTIX ()

Formalisme



- Une classe se définit par :
  - Un nom propre
  - Des attributs
  - Des méthodes agissant sur ces attributs
- Les objets, instances d'une même classe
  - Partagent un même comportement
  - Ne diffèrent entre eux que par la valeur des attributs.

= BUROTIX ()

Constructeur

- Le constructeur (méthode optionnellement définie) est exécuté au moment de la création de l'objet construction, qui permet de l'initialiser.

Exo 11-02-04 :
attribut de classe & attribut d'objet

- Que va imprimer le programme suivant ?

```
# la classe C …
class C:
    a = 0
    b = 0
    c = 0

    def test (self):
        a = 1
        C.b = 2
        self.c = 3
        print( a, C.b, C.c )
```

```
# le code principal …
O = C()
O.test()
print( C.a, O.a )
print( C.b, O.b )
print( C.c, O.c )
```

```
1 2 0
0 0
2 2
0 3
```

garbage collector : "vie et mort d'un objet"

- "ramasse-miettes" ou "garbage collector" (GC)
- gestionnaire automatique de la mémoire
  - responsable du recyclage de la mémoire préalablement allouée puis inutilisée
- En Python, le GC est automatique !
  - cas particuliers : module **gc**

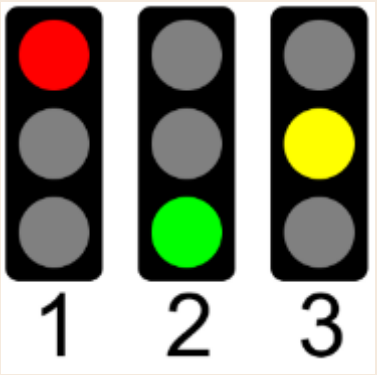
Exo 11-02-06 : garbage collector

```
class MaClasse:
    monAO = None
    def __init__(self, name):
        self.name = name
        print(…)
        self.monAO = MonAutreClasse()
    def __del__(self):
        print(…)
    def __str__(self):
        return …

class MonAutreClasse:
    def __init__(self):
        print(…)
    def __del__(self):
        print(…)
```

```
# MAIN
O1 = MaClasse("O1")
O2 = MaClasse("O2")
O3 = MaClasse("O3")
O4 = O3

O2 = None  # objet détruit
O3 = None  # référence de l'objet
détruit
# end of MAIN
# tous les objets sont détruits
```

= BUROTIX ()

Exo 11-02-11 : simulateur de trafic



- Programmez un feu de signalisation
  - classe `Light`
  - trois états, cf image
  - pas d'aspect graphique
  - pas de constructeur
- Output

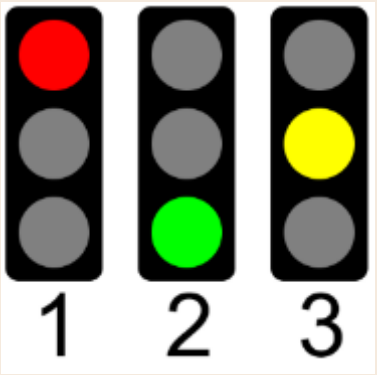
```
light color is 1
light color is 2
light color is 3
light color is 1
```

- Input

```
feu01 = Light()
feu01.color = 1
print(feu01)
feu01.change()
print(feu01)
feu01.change()
print(feu01)
feu01.change()
print(feu01)
```

= BUROTIX ()

Exo 11-02-12 : simulateur de trafic



- Programmez un feu de signalisation
  - classe **`Light`**
  - avec un constructeur
- Output

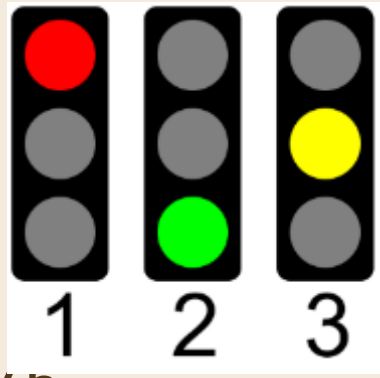
```
light color is 1
light color is 2
light color is 3
light color is 1
```

- Input

```
feu01 = Light()
feu01.color = 1
print(feu01)
feu01.change()
print(feu01)
feu01.change()
print(feu01)
feu01.change()
print(feu01)
```

= BUROTIX ()

Exo 11-02-14 : simulateur de trafic



- Programmez une voiture autonome
  - classe **`Car`**
  - contructeur, arguments :
    - nom de la voiture
    - vitesse de démarrage
      - zéro par défaut
  - changement de vitesse, méthodes :
    - increment()
    - decrement()
    - increment(n)
- Output

```
Peugeot: speed 0 km/h
Peugeot: speed 1 km/h
Peugeot: speed 11 km/h
Peugeot: speed 10 km/h
```

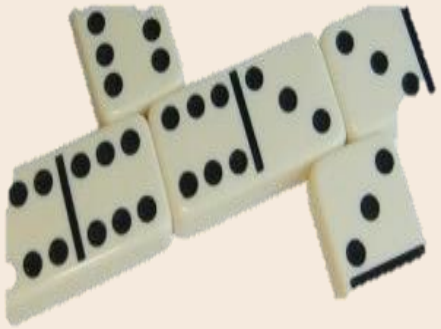
- Input

```
voiture_A = Car("Peugeot")
print(voiture_A)
voiture_A.increment()
print(voiture_A)
voiture_A.increment(10)
print(voiture_A)
voiture_A.decrement()
print(voiture_A)
```

= BUROTIX ()

Exo 11-02-21 : Dominos

- Définissez une classe Domino() pour instancier des objets simulant les pièces d'un jeu de dominos.
- Le constructeur initialisera les valeurs des points présents sur les deux faces A et B du domino (valeurs par défaut = 0).
- Méthodes
  - Affiche_points() : affiche les points présents sur les deux faces.
  - Somme_valeur() : renvoie la somme des points présents sur les 2 faces.

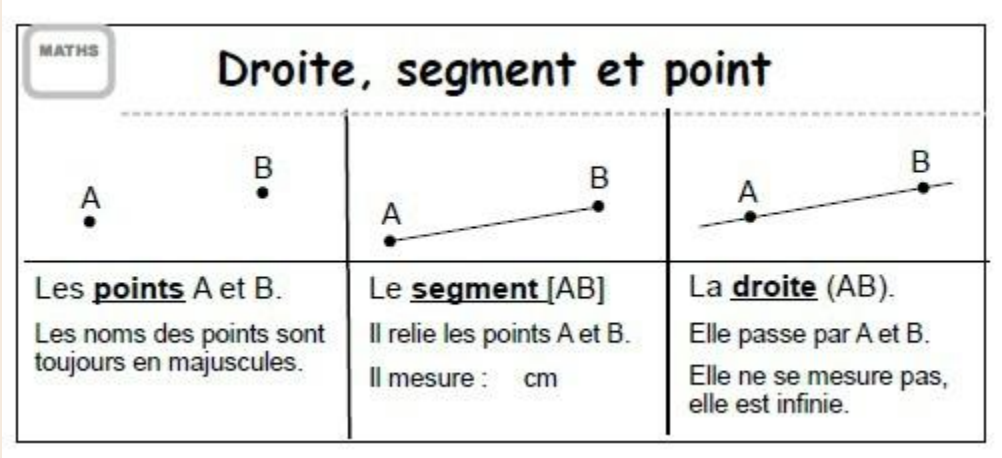


```
> d1 = Domino(2,6)
> d2 = Domino(4,3)
> d1.affiche_points()
face A : 2 / face B : 6
> d2.affiche_points()
face A : 4 / face B : 3
> print "total des points :",
d1.valeur() + d2.valeur()
15
> l_dominos = []
> for i in range(7):
      l_dominos.append(Domino(6, i))
> print l_dominos
```

= BUROTIX ()

Exo 11-02-22 : le point du plan

- Définissez une classe Point
  - Un point est représenté par son abscisse et son ordonnée
  - Constructeur : coordonnées (0, 0) par défaut
  - Méthode « distance » : calcule et renvoie la distance du point avec l'origine (0, 0)

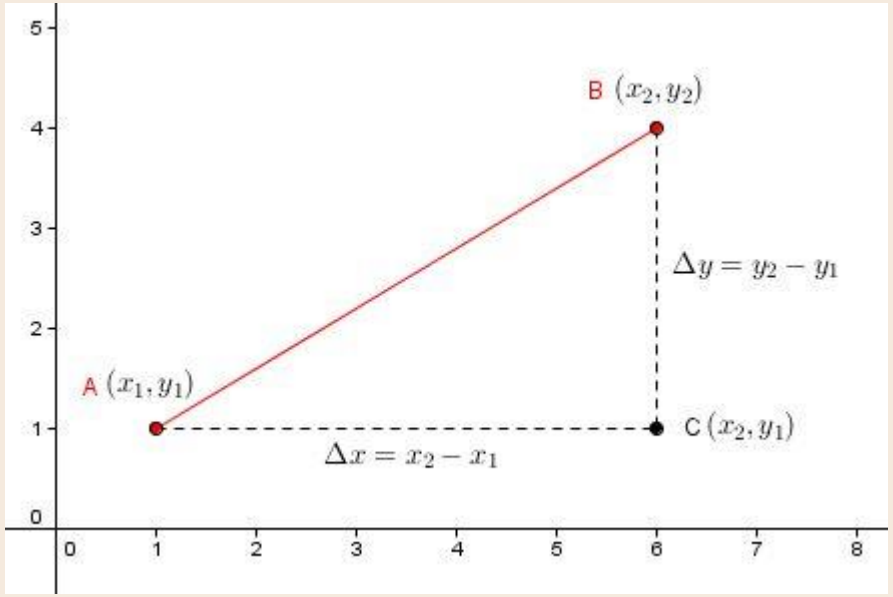


= BUROTIX ()

Exo 11-02-23 : le segment du plan

- Définissez une classe Segment
  - Un segment est défini par deux points.
- Méthodes
  - **`longueur_pythagore()`**
    - retourne la longueur pythagorienne du segment
    - $= \sqrt{(xT - xS)^2 + (yT - yS)^2}$
  - **`longueur_manhattan()`**
    - retourne la longueur manhattanienne du segment
    - = | xT – xS | + | yT – yS |

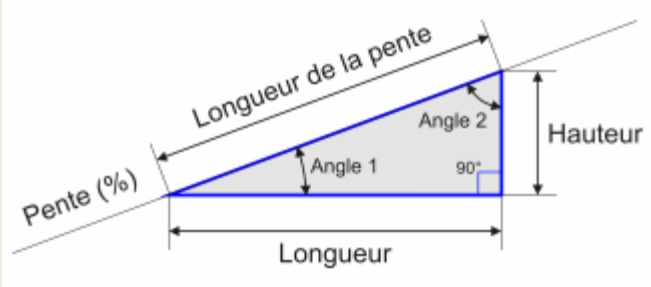
```
P1 = Point(3,4)
P2 = Point(6,0)
Mon_segment = Segment(P1,P2)
print(Mon_segment.longueur_pythagore())
// 5
print(Mon_segment.longueur_Manhattan())
// 7
```



= BUROTIX ()

Exo 11-02-23 : le segment du plan

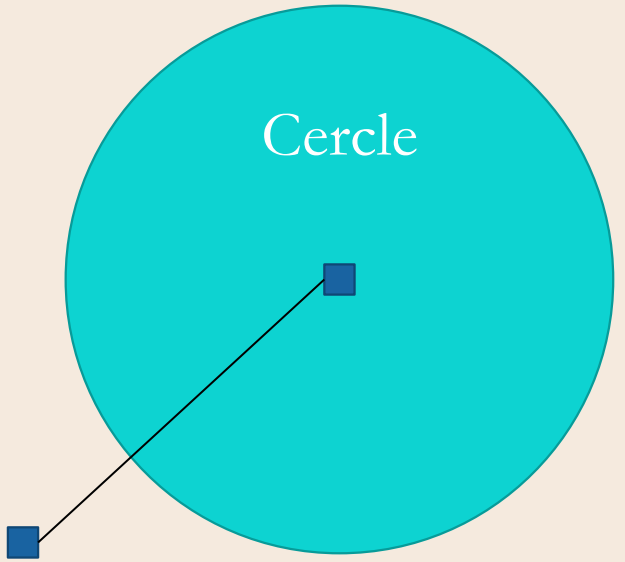
- Définissez une classe Segment (suite)
  - Comment grouper les deux calculs de longueur en une seule méthode ?
  - Méthode « pente » : calcule et retourne la pente (%) du segment



= BUROTIX ()

Exo 11-02-25 : le cercle

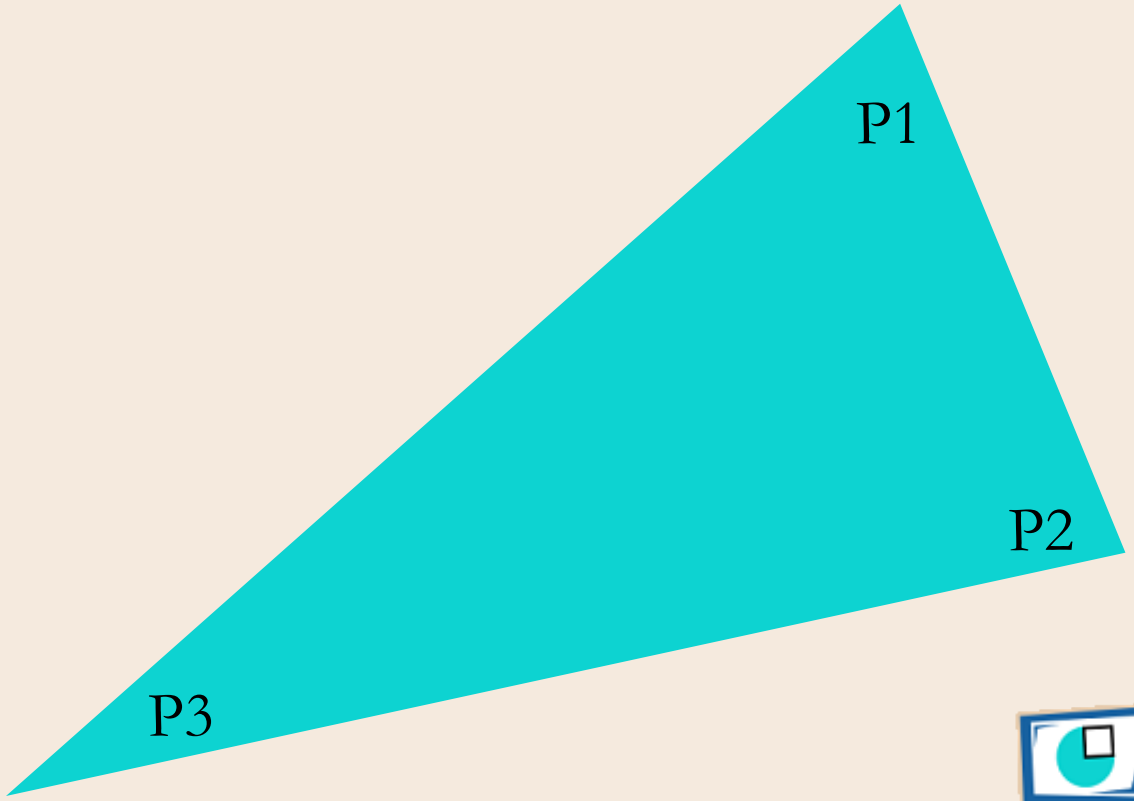
- Un cercle est défini par :
  - Un point `C` qui représente son centre
  - Son rayon `r`
- Méthodes de la classe `Cercle` :
  - Constructeur : On crée un cercle en précisant son centre `C` et son rayon `r`.
  - `getPerimetre()` : retourne le périmètre du cercle.
  - `getSurface()` : retourne la surface du cercle.
  - `isInside(Point p)` : retourne `True` si `p` appartient au cercle, càd si la longueur du `segment(p , r)` est inférieure au rayon.
  - `__str__()` : retourne chaîne de type "CERCLE(x,y,R)"
- Ce problème se basera sur les classes `Point` et `Segment` déjà définies.
  - On notera l'extrême concision de la classe Cercle ainsi obtenue.
- Réf : https://www.exelib.net/csharp-poo/la-classe-cercle.html



= BUROTIX ()

Exo 11-02-27 : le triangle

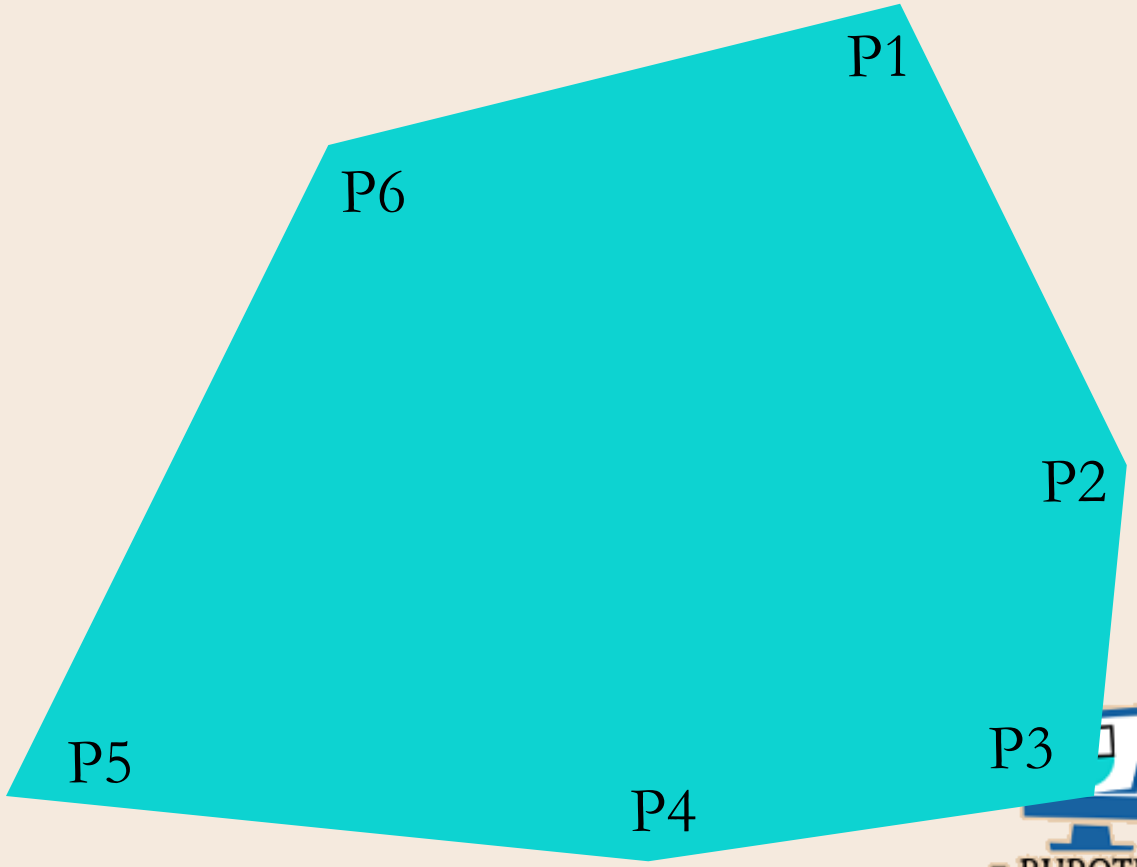
- Définissez une classe **`Triangle`**
  - modélisant un triangle quelconque à partir de trois points
  - avec une méthode **`perimeter()`**
  - en vous basant sur les classes **`Point`** et **`Segment`**
  - BONUS : écrivez la méthode **`surface()`** et modifiez en conséquence la classe **`Segment`**. Bonne chance …



= BUROTIX ()

Exo 11-02-28 : polygone quelconque

- Définissez une classe **`Polygone`**
  - modélisant un polygone quelconque à partir de N points
  - avec une méthode **`perimeter()`**
  - en vous basant sur les classes **`Point`** et **`Segment`**



= BUROTIX ()

Chapitre 11-03 : attributs publics et privés

Exemple: un compte en banque

```
class Compte :

  def __init__(self, titulaire):
    self.titulaire = titulaire
    self.solde = 0
```

Compte
titulaire solde

```
>>> a = Compte('kim')
>>> a.titulaire
'kim'
>>> a.solde
0
```

BUROTIX ()

Exemple: un compte en banque

```
class Compte :

  def __init__(self, titulaire):
    self.titulaire = titulaire
    self.solde = 0
```

à protéger

```
>>> a = Compte('kim')
>>> b = Compte('siegfried')
>>> a.solde += 1000
>>> b.solde -= 1000
```

à éviter

BUROTIX ()

Variables d'instance privés

```
class Compte :

  def __init__(self, titulaire) :
    self.__titulaire = titulaire
    self.__solde = 0
```

attributs privés

```
>>> a = Compte('kim')
>>> a.__titulaire
AttributeError: 'Compte' object
has no attribute '__titulaire'
>>> a.__solde
AttributeError: 'Compte' object
has no attribute '__solde'
```

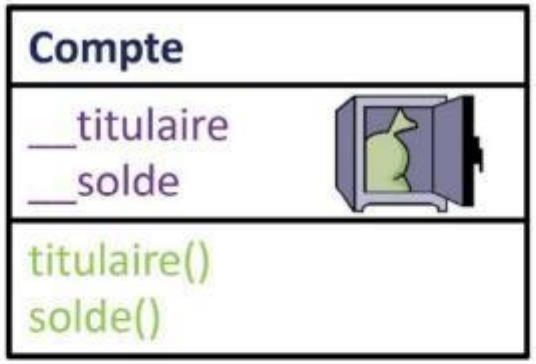
BUROTIX ()

Méthodes accesseurs

```
class Compte :

  def __init__(self, titulaire) :
    self.__titulaire = titulaire
    self.__solde = 0

  def titulaire(self):
    return self.__titulaire
  def solde(self):
    return self.__solde
```

méthodes accesseurs

```
>>> a = Compte('kim')
>>> a.titulaire()
'kim'
>>> a.solde()
0
```



BUROTIX ()

Méthode __str__

```
class Compte :

  def __init__(self, titulaire) :
    self.__titulaire = titulaire
    self.__solde = 0

  def titulaire(self):
    return self.__titulaire
  def solde(self):
    return self.__solde

  def __str__(self) :
    return "Compte de {} : solde = {}".
           format(self.__titulaire,self.__solde)
```

méthode __str__

```
>>> print(a)
Compte de Kim : solde = 0
```

BUROTIX ()

Méthode __str__

```
class Compte :

  def __init__(self, titulaire) :
    self.__titulaire = titulaire
    self.__solde = 0

  def titulaire(self):
    return self.__titu
  def solde(self):
    return self.__solde

  def __str__(self) :
    return "Compte de {} : solde = {}".
           format(self.titulaire(),self.solde())
```

MIEUX : Utiliser les accesseurs

Appel à self

méthodes accesseurs

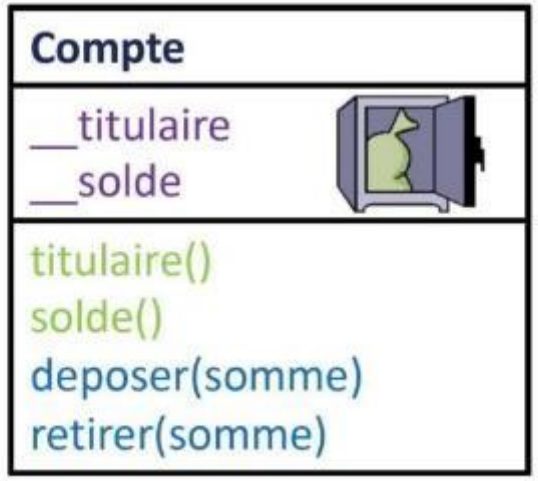
```
>>> print(a)
Compte de Kim : solde = 0
```

BUROTIX ()

Méthodes d'instance

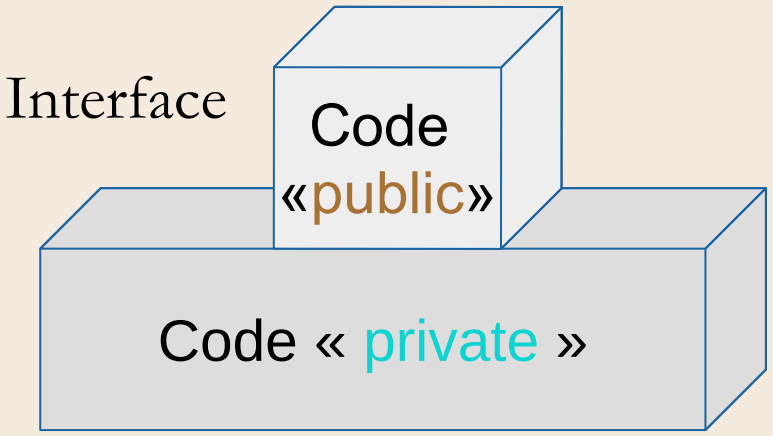
```
class Compte :

  …

  def deposer(self, somme):
    self.__solde += somme
    return self.solde()

  def retirer(self, somme):
    if self.solde() >= somme :
      self.__solde -= somme
      return self.solde()
    else :
      return "Solde insuffisant"
```

```
>>> a = Compte('kim')
>>> a.deposer(100)
100
>>> a.retirer(90)
10
>>> a.retirer(50)
Solde insuffisant
```



BUROTIX ()

Encapsulation & Interface



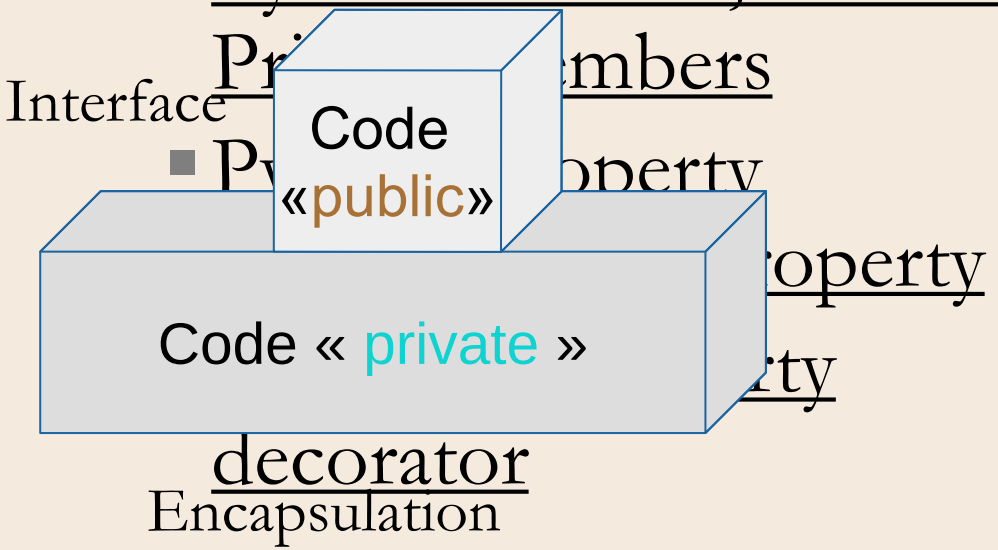
Encapsulation

- Tout attribut ou méthode peut être d'encapsulation :
  - **Private** : manipulable seulement au sein de la classe
  - **(Protected** : idem, mais aussi des classes héritées)
  - **Public** : manipulable de l'extérieur de la classe
- Développer des applications OOP complexes nécessite un couplage réduit entre classes
  - attribut et méthode définis « private » par défaut
  - si nécessaire, on définit une méthode comme « public »
  - attribut jamais défini « public »
  - on écrit des méthodes pour accéder à un attribut
    - get : pour lire sa valeur
    - set : pour imposer une nouvelle valeur
- Interface = ensemble des méthodes « public »

= BUROTIX ()

Encapsulation & Interface en Python

- Liens:
  - Python - Public, Protected, Pr… …mbers
  - Py… …operty
  - … …roperty
  - … …rty
  - decorator



= BUROTIX ()

Lien : Les décorateurs en Python

Encapsulation & Interface en Python

Approche classique (11-03-05_getset_classical.py)

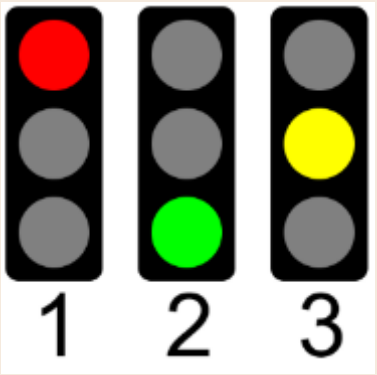
```
Class Thermometre:

    …
    def get_value_celsius(self):
        # value_celsius getter
        return self.__value_celsius

    def set_value_celsius(self, val):
        # value_celsius setter
        # aucune température
        # en dessous de -273.15
        if val < -273.15 : val = -273.15
        self.__value_celsius = val

therm_o = Thermometre()
print(therm_o.get_value_celsius())
therm_o.set_value_celsius(100)
```

Approche par décorateur en Python (11-03-05_getset_decorator.py)

```
Class Thermometre:

    …
    @property
    def value_celsius(self):
        # value_celsius getter
        return self.__value_celsius

    @value_celsius.setter
    def value_celsius(self, val):
        # value_celsius setter
        # aucune température
        # en dessous de -273.15
        if val < -273.15 : val = -273.15
        self.__value_celsius = val

therm_o = Thermometre()
print(therm_o.value_celsius)
therm_o.value_celsius = 100
```

= BUROTIX ()

Exo 11-03-12 : simulateur de trafic



- Programmez un feu de signalisation
  - classe `Light`
  - avec l'attribut `color` privé et ses getter / setter.
- Output

```
1
2
3
1
```

- Input

```
feu01 = Light()
print(str(feu01.color))
feu01.change()
print(str(feu01.color))
feu01.change()
print(str(feu01.color))
feu01.change()
print(str(feu01.color))
```

= BUROTIX ()

Exo 11-03-13 : one car on the road

- voiture autonome
  - Classe `Car`
  - On fixe la vitesse de la voiture, entre 0 et 50km/h.
  - On fixe la "duration"
  - Méthode `forward()` : la voiture avance.
  - Unités : km et heure.
  - Définissez un maximum d'attributs privés avec leurs getters / setters.
- Output

```
A: sp  0km/h, ti 0.0h, po   0.0km
A: sp 50km/h, ti 1.0h, po  50.0km
A: sp 50km/h, ti 1.5h, po  75.0km
A: sp 50km/h, ti 2.5h, po 125.0km
```

- Input

```
voiture_A = Car("A")
print(voiture_A)
voiture_A.speed = 50
voiture_A.duration = 1
voiture_A.forward()
print(voiture_A)
voiture_A.duration = 0.5
voiture_A.forward()
print(voiture_A)
voiture_A.duration = 1
voiture_A.forward()
print(voiture_A)
```

= BUROTIX ()

Exo 11-03-31 : gestion d'un stock

- Ecrivez une classe « Stock » qui gère le stock d'une entreprise.
- En pratique, cette classe gère :
  - stock de papier (bloc de 500 feuilles) : 20 actuellement
  - stock de crayons : 8 actuellement
  - tous ces attributs sont privés.
- Un « seuil de recommande » est défini, en dessous duquel une recommande est nécessaire :
  - Papier : 2 blocs
  - Crayons : 1
- Ecrivez les méthodes permettant les actions suivantes
  - Connaître la valeur du stock
  - Retirer un élément du stock, tant que celui-ci le permet.
  - Si la valeur du stock est inférieure ou égale au seuil, alors un message est envoyé à l'utilisateur lui demandant une recommande.

= BUROTIX ()

Exo 11-03-31 : gestion d'un stock

```
stock = Stock( papier=20, crayon=8 )
stock.seuil_de_recommande(papier=2, crayon=1 )
print(stock) # imprime état du stock - tout va bien
stock.retirer( papier=12, crayon=6 )
print(stock)
stock.retirer(papier=7 ) # message de recommande
print(stock)
stock.retirer(crayon=1) # message de recommande
print(stock)
stock.valeur() # message de recommande
print(stock)
```

= BUROTIX ()

Exo 11-03-32 : grand stock

- Ecrivez une classe « Stock » qui gère le stock d'une entreprise.
- Cette classe gère un ensemble de produits.
  - Modélisés par un dictionnaire
- Un « seuil de recommande » est défini pour chaque membre de l'ensemble de produits, en dessous duquel une recommande est nécessaire.
- Ecrivez les méthodes permettant les actions suivantes
  - Initialiser le stock
  - Initialiser le seuil de recommande
  - Connaître la valeur du stock
  - Retirer un élément du stock.
  - Si la valeur du stock est inférieure ou égale au seuil, alors un message est envoyé à l'utilisateur lui signalant la nécessité de procéder à une recommande.
  - Si le stock est vide, on ne peut plus retirer d'éléments.
  - En cas de livraison, réalimenter le stock.

= BUROTIX ()

Exo 11-03-41 : jouons avec les dates

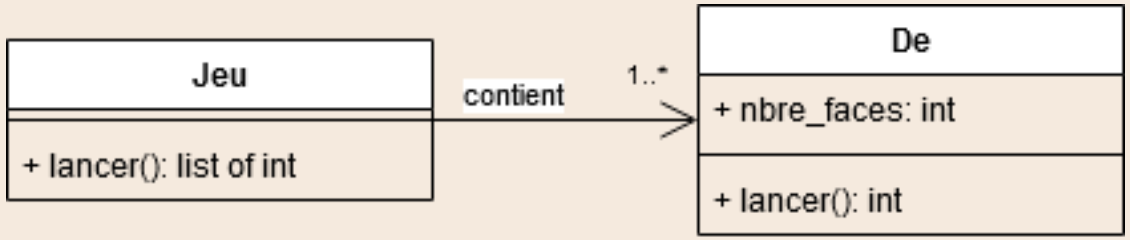
Date
+ jour: int + mois: int + annee: int

- cf diagramme de classe UML ci-contre
- implémenter cette classe en Python, en privatisant les attributs.
- constructeur :
  - prévoir un dispositif pour éviter les dates impossibles (du genre 32/14/2020)
    - Génération d'une erreur : instruction `raise`
- méthode `__str__()`
  - afficher la date sous la forme "25 janvier 2023"
  - noms des mois définis comme attribut de classe à l'aide d'une liste
- Méthode __repr__()
  - afficher la date sous la forme "20230125"
- méthode `__lt__()`
  - comparer deux dates
  - `d1 < d2` renvoie `True` ou `False`
- *Référence : https://info.blaisepascal.fr/nsi-exercices-poo*

= BUROTIX ()

Exo 11-03-43-a : jouons avec les dés

- Un « jeu de dés » contient plusieurs dés.
- Implémenter les deux classes
  - **Jeu**
  - **Dé**



- Instancier un jeu de 3 dés et afficher le résultat d'un lancer.
- *Référence : https://info.blaisepascal.fr/1t-poo-des-des*

= BUROTIX ()

Exo 11-03-43-b : jouons avec les dés

- Maintenant, les valeurs des dés restent mémorisées au sein même des objets `Dé` (comme si les dés du jeu étaient posés sur le plateau de jeu).
- La valeur de chaque dé est mémorisé au sein de chacun des objets `Dé`, dans un nouvel attribut valeur.

= BUROTIX ()

Exo 11-03-43-c : jouons avec les dés

- Implanter la méthode permettant de comparer le lancer de deux jeux de dés.
- Écrire le programme correspondant au scénario suivant :
  - Soit deux jeux, `j1` et `j2`
  - `j1` est lancé (et conservé)
  - `j2` est lancé de manière itérative
  - L'itération s'arrête quand `j2 > j1`

= BUROTIX ()

Partie 11-04 : Python, POO, les variables et méthodes de classe

Variable de classe

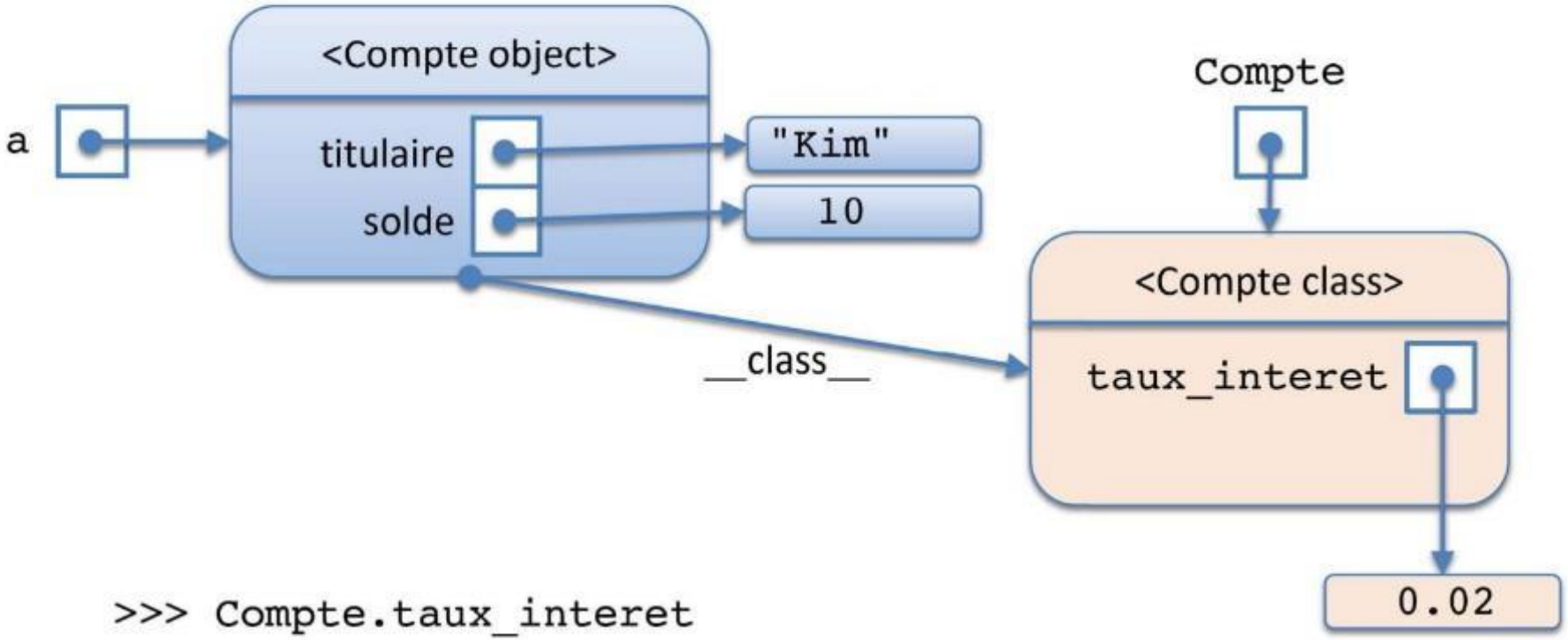
```
class Compte :

  taux_interet = 0.02
  …
```

une variable pour la classe

```
>>> Compte.taux_interet
0.02
>>> a = Compte("Kim")
>>> a.taux_interet
0.02
>>> b = Compte("Siegfried")
>>> b.taux_interet
0.02
>>> Compte.taux_interet = 0.04
>>> a.taux_interet
0.04
>>> b.taux_interet
0.04
```

même valeur pour toutes les instances de la classe

Variable de classe

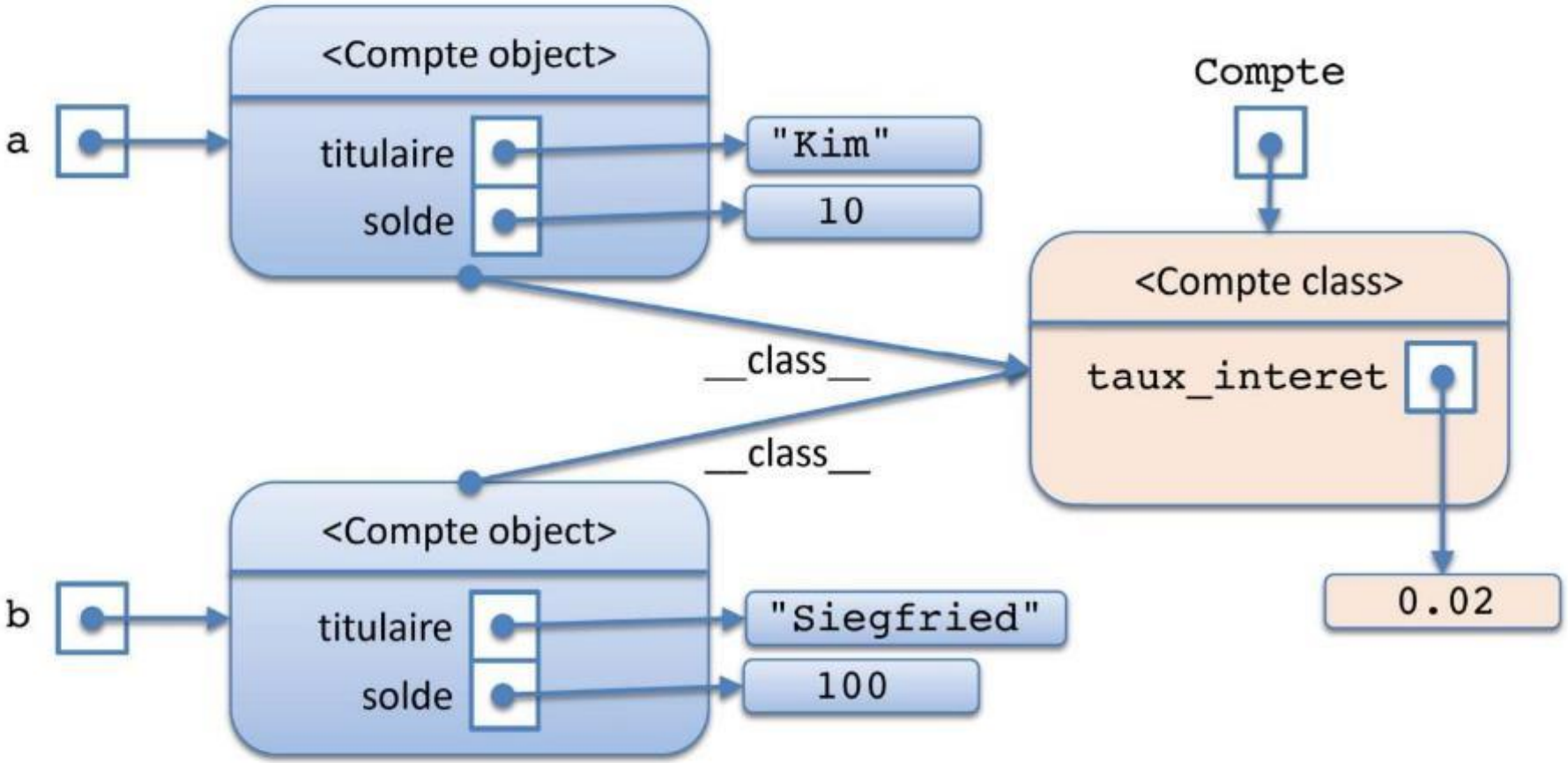



```
>>> Compte.taux_interet
0.02

>>> a.taux_interet
0.02
```

Si pas trouvé dans l'objet, Python va chercher dans la classe...

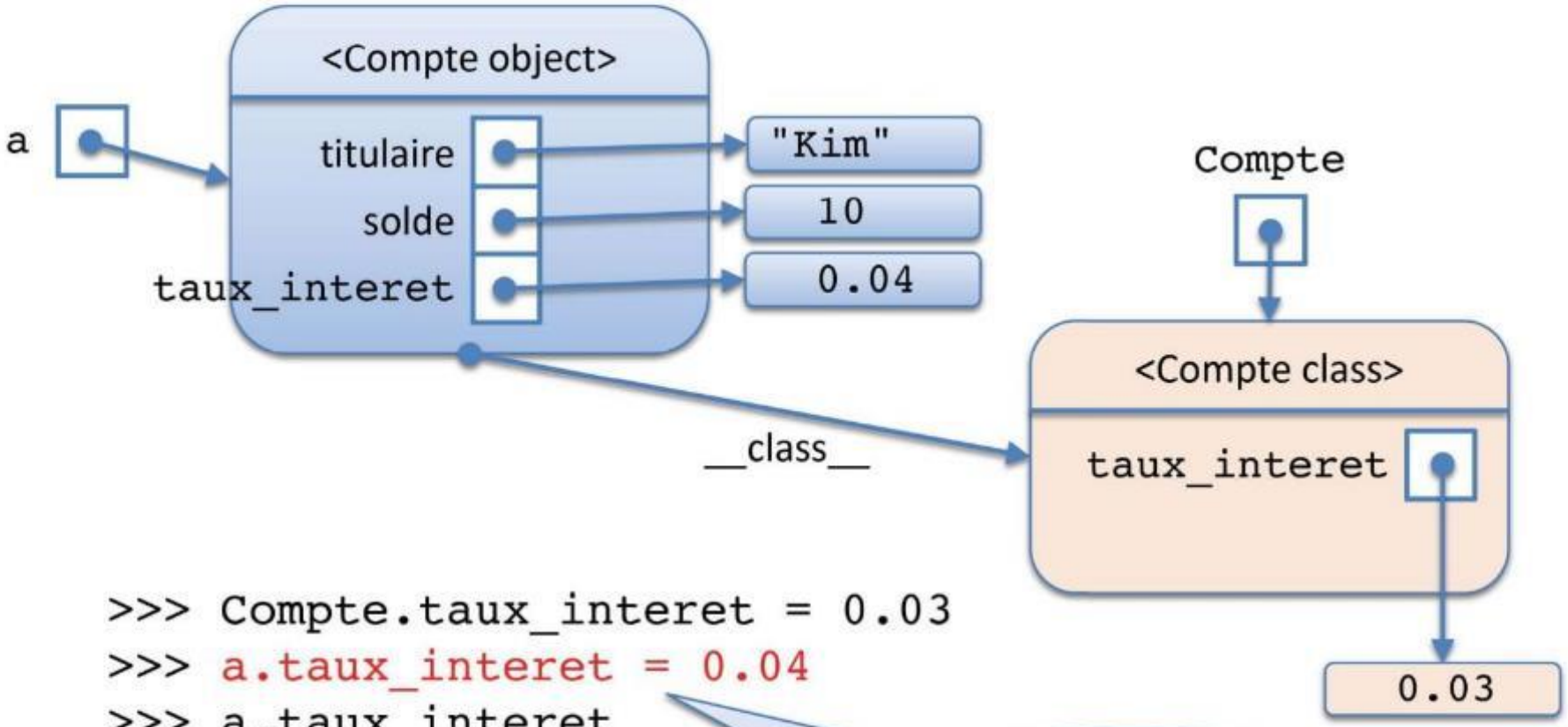
BUROTIX ()

Variable de classe



BUROTIX ()

Variable de classe




```
>>> Compte.taux_interet = 0.03
>>> a.taux_interet = 0.04
>>> a.taux_interet
0.04
>>> Compte.taux_interet
0.03
```

Attention!
Une variable d'instance peut avoir le même nom qu'une variable de classe

BUROTIX ()

Variable de classe privée

```
class Compte :

    __taux_interet = 0.02
    …
```

variable de classe privée

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

Comment y accéder ?

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
  …
```

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 !

BUROTIX ()

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

Implantation via décorateur @property

```
class Compte:
    __taux_interet = 0.02

    @classmethod
    @property
    def taux_interet(cls):
        return cls.__taux_interet

    @classmethod
    @taux_interet.setter
    def set_taux_interet(cls, novo_taux):
        cls.__taux_interet = novo_taux

taux = Compte.taux_interet              # OK
Compte.taux_interet = 0.04              # KO
Compte.set_taux_interet(0.04)# OK
```

- GETTER
  - On peut utiliser ensemble les décorateurs `@classmethod` et `@property`
  - On peut donc "getter" la valeur comme un attribut.
- SETTER
  - Le décorateur `@value.setter` n'est pas compatible avec `@classmethod`.
  - On est donc obligé d'utiliser une méthode propre `set_value(…)`

Exo 11-04-31 : calcul de prix avec TVA

- Créer la classe **`Article`** caractérisée par les attributs : **`référence, prix_ht, taux_tva`**.
- Méthodes :
  - Constructeur : params **`ref`** et **`prix_ht`**
  - **`.__str__()`** : affiche les informations de l'article
  - **`.calculer_prix_ttc()`** : calcule et retourne le prix TTC d'un article
    = PrixHT + (PrixHT * TauxTVA / 100)
- Créer un programme de test.
- Attribut de classe :
  - Le taux de TVA est en fait commun à tous les articles. Pour éviter toute redondance de cet attribut, il est déclaré comme "attribut de classe", partagé au niveau de la classe **`Article`** et non comme un attribut spécifique des objets instanciés.
- Optimisation :
  - Le prix TTC est calculé seulement s'il est demandé, et une seule fois.
- Réf : https://www.exelib.net/csharp-poo/la-classe-article.html

= BUROTIX ()

Documentations & Exos

- https://pynative.com/python-class-variables/
- Variable d'instance (d'objet)
  - Variable d'un objet à l'autre
  - Propre à l'objet, non partagée
- Variable de classe
  - Déclarée à l'intérieur de la classe
  - Déclarée en dehors de toute méthode d'instance ou de constructeur
  - Partagée par toutes les instances d'une classe

- **Exo 11-04-02**
  - variable de classe publique
- **Exo 11-04-04**
  - variable de classe privée
  - méthode de classe
  - décorateur @classmethod

= BUROTIX ()