j7

J7: Objectifs

- classe
- objet
- champs
- méthodes
- __init__, __str__, __repr__
- dir()
- héritage
- surcharge
- *name mangling*
- MRO
- EAFP
- Duck Typing
- Associations d'objets
- Itérateurs

Introduction

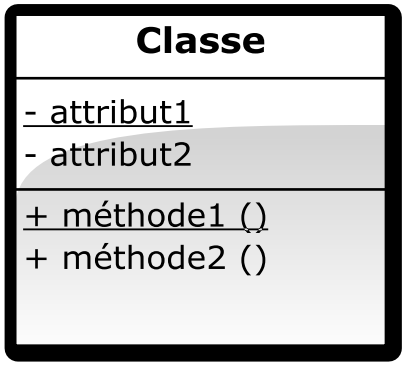
- Dans les langages orientés objet, la notion d'objet supporte
  - l'encapsulation
    - un interface par lequel s'adresser à l'objet
    - un espace de noms privé
    - des fonctions et des données
  - l'héritage
    - une notion de classe qui regroupe des objets semblables
    - des relations de filiations entre classes
    - une transmission des propriétés des classes mères vers leurs classes filles
  - le polymorphisme
    - les mêmes opérations, les mêmes méthodes pour des objet de classes différentes

- En Python, la notion d'objet est centrale car
  - toute variable, tout constante, tout littéral référence un objet
  - De même, les types sont des classes

class

- Une classe est un type de données
- Caractérisé par un nom
- Et par des attributs
  - les informations contenues
    - attribut ou champ ou membre
    - *attribute, field, member*
  - les opérations permises
    - méthodes
- On distingue les attributs de la classe de ceux des instances (self)
  - méthodes aussi
  - Bizarrerie :
    - self n'est pas un mot-clé du langage.
- Notation graphique UML

```
class Classe:
    """ un exemple de classe """
    attribut1 = 12
    def méthode1()
        …
    def méthode2(self)
        self.attribut2 = 0.0
```



Classeur Jupyter Classes.ipynb

@classmethod

- Une méthode de classe peut être signalée par un décorateur
  - elle doit alors avoir la classe comme premier argument
  - elle s'applique alors aussi bien aux instances qu'à la classe (flexibilité)
  - elle peut servir à un constructeur alternatif
- A mon avis, c'est recommandé surtout pour la clarté du code
  - puisqu'il y a moyen de faire tout cela en écrivant le nom de la classe devant le .
  - le décorateur traite l'argument cls

```
class Car:
    __n = 0

    def __init__(self, brand):
        Car.__n +=1

    @classmethod
    def get_internal(cls):
        return f'# Cars created: {cls.__n}'

    @classmethod
    def mkOne(cls, brand, model):
        """ Alternate constructor """
        c = cls(value)
        Car.__n -=1 #leave count unchanged
        return c

print(Car.get_internal())

c1 = Car("Peugeot")
print(Car.get_internal())

c2 = Car("Ford")
print(c2.get_internal())

c3 = Car.mkOne("Mitsubishi", "Outlander")
print(Car.get_internal())
```

@staticmethod

- Une méthode statique n'a besoin ni d'une instance, ni de la classe.
- On la regroupe dans la classe pour des raisons de clarté
  - Méthode utilitaire en lien avec ce qui se fait dans la classe
  - On peut l'appeler aussi bien depuis l'instance, que depuis la classe (flexibilité)
- Elle ne dispose pas des variables self ni cls et ne peut donc modifier l'état d'une instance ou de la classe
  - Elle peut cependant accéder aux variables de classes en écrivant en toutes lettres le nom de la classe avec la notation pointée

```
class Person:
    invalidAgeExists = False

    def __init__(self, name, age):
        self.name = name
        self.age = age

    @staticmethod
    def validate(age):
        if age>0 and age<150:
           return True
        else:
           Person.invalidAgeExists = True
           return False

p1 = Person("Donald",12)

print(Person.validate(160))
```

Objet

- Un objet est une instance d'une classe
- Pour instancier la classe

```
x = Classe()
```

- On accède alors à ses attributs et à ses méthodes par la notation pointée

```
x.attribut2
x.méthode2()
```

- Observer que l'on peut ajouter de nouveaux attributs

```
x.attribut3 = "Nouveau"
```

- Pour tester que deux variables désignent le même objet
  - et non deux objets différents qui auraient la même valeur

```
if x is y :
```

__init__(self)

- Définir la méthode init
  - Elle s'exécute
    - A chaque fois qu'une classe est instanciée
    - Pour initialiser les variables d'instance

```
class Trinome:
    """ Trinôme OO"""
    def __init__(self, va=0.0, vb=0.0,
        vc=0.0):
        self.a=float(va)
        self.b=float(vb)
        self.c=float(vc)
```

- On n'a pas besoin de l'appeler explicitement

```
t = Trinome(2.34, 6, -1)
```

__str__(self)

- Définir la méthode str

```
class Trinome:
    """ Trinôme OO"""
    def __str__(self):
        """ Display the equation """
        return "%5.2f x² + %5.2f x + %5.2f = 0"\
        %(self.a, self.b, self.c)
```

- Implicitement appelée à chaque fois que l'on a besoin de convertir un objet en str

```
t = Trinome(2.34, 6, -1)
print(t)
# 2.34 x² + 6.00 x + -1.00 = 0
```

__repr__(self)

- __repr__ est déclenchée par repr

```
class Trinome:
    """ Trinôme OO"""
    def __repr__(self):
        """ Display the equation """
        return f"Trinome({self.a},
                {self.b}, {self.c})"
```

- Une représentation textuelle qui permet de recréer un objet identique

```
>>> t = Trinome (1, 2, 3)

Trinome(1,2,3)

>>> eval(repr(t))
```

dir()

class	delattr	dict	dir	doc	eq
format	ge	getattribute	gt	hash	init
le	lt	module	ne	new	reduce
reduce_ex	repr	setattr	sizeof	str	subclasshook
weakref					

- Voici le résultat de dir(x). Ces __id__ sont ceux prédéfinis par Python pour un objet de base x instancié à partir d'une classe de définition vide.
  - Les underscores devant et derrière les noms des méthodes ont été supprimés pour la lisibilité

Classeur Jupyter Dir.ipynb

__new__

- appelée à la création d'un objet
- avant son initialisation

```
object.__new__(cls, *args, **kwargs)
```

- en général, fait appel à

```
x = super().__new__(cls, *args,
    **kwargs)
```

- pour retourner l'objet nouvellement créé

```
return x
```

name mangling

- Python part du principe que nous sommes entre "adultes consentants" *consenting adults*
  - rien n'est caché
    - Contrairement à Java
      - (Private, Protected, Public)

mangling : défigurer, détruire, dénaturer

un camouflage !

- Pour occulter un nom, on peut
  - ne rien mettre :
    - la lecture des commentaires invite à ne pas utiliser
  - mettre un préfixe _:
    - invitation à considérer que c'est interne
  - mettre un préfixe __ :
    - Python va alors modifier le nom interne (le camoufler) pour que ce soit moins simple à utiliser
      - mais on peut toujours y arriver si on veut

Classeur Jupyter Mangling.ipynb

Usage de l'*underscore* _ dans les noms

- _id désigne une variable privée
- id_ pour éviter une collision avec les mots réservés ou les fonctions prédéfinies de Python
- __id déclenche le *name mangling* co
- __id__ désigne une méthode "magique" de Python
- séparateur de chiffres (3.6+)
- peut désigner la fonction gettext dans le contexte de l'internationalisation

```
_delta
```

```
map_
```

```
__delta  #_Trinome__delta
```

```
__len__
```

```
1_000_000
```

```
_('Un texte internationalisable')
```

Retour sur @property

- Proxy pour accéder à des membres cachés
  - name mangling
  - encapsulation
- Contrôler les accesseurs (getters) et les mutateurs (setters et deleters)
  - Utile lorsqu'ils embarquent un peu d'intelligence
    - validation

Classeur Jupyter Property.ipynb

i18n/l10n

- i18n = internationalisation
- l10n = localisation

sont des abréviations utilisées dans certains contextes

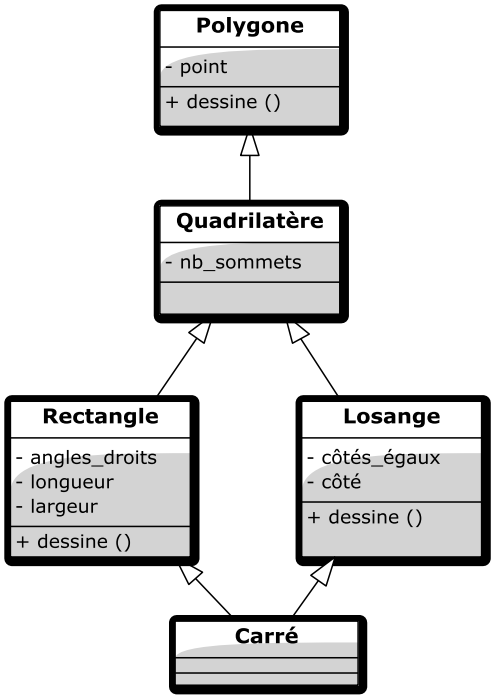
Héritage

- Hériter, *to inherit*
- La définition d'une classe peut mentionner une ou des classes parentes
  - classe fille, sous-classe
  - classe parente, super-classe

  Modélise la relation "est-un" *is-a*

- La classe fille hérite des attributs et des méthodes de la classe parente
  - héritage multiple, *multiple inheritance*

- Référencer un attribut ou une méthode de la classe parente dans une classe fille
  - super(laClasseFille, self).x
  - super().x




```
class Polygone: …
class Quadrilatère(Polygone):…
class Rectangle(Quadrilatère):…
class Losange(Quadrilatère):…
class Carré(Rectangle, Losange):…
```

Classeur Jupyter Classes.ipynb

Surcharge

- Sucharger une méthode

  to override a method

  – Nous l'avons déjà fait avec

```
__init__
__str__
__repr__
```

- Surcharger des opérateurs

  – utiliser les opérateurs du langage
  binaires + - / // % *
  unaires + - abs()
  comparaisons == != < > <= >=

```
__add__, __sub__, __mul__, …
__neg__, __pos__, __abs__
__eq__, __ne__, __lt__, __gt__, …
```

  – les opérateurs de conteneurs
  len, x[k], x[k]=

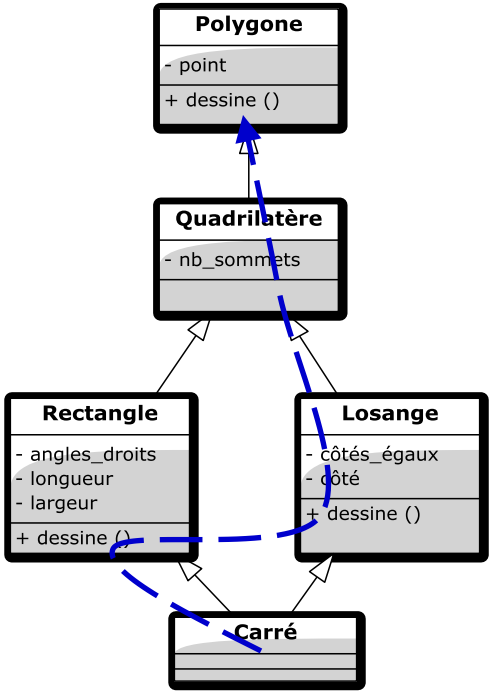
```
__len__, __getitem__, __setitem__
```

Method Resolution Order

- Ordre de résolution des méthodes
  - attributs
- La recherche s'effectue dans l'ordre dans
  - La classe
  - La classe parente

    ou les classes parentes dans l'ordre d'apparition, en cas d'héritage multiple
  - La classe parente que l'on indique explicitement
- En cas d'ambiguité : exception

Classeur Jupyter MRO.ipynb




```
c=Carré()
c.dessine()

class Carré(Rectangle,Losange):
  def dessine(self):
    # Il y a le choix entre…
    super().dessine() #Rectangle
    Losange.dessine(self) #Ou Losange
```

it is **E**asier to **A**sk for **F**orgiveness than **P**ermission

- EAFP
  - C'est plus facile de s'excuser après que de demander la permission avant

- Plutôt que de tester si une méthode est présente

```
if hasattr(x,'add'): #non Pythonic !
    …
else:
    …
```

- On l'utilise, mais on gère l'erreur

```
try:                        #Pythonic !
    x.add()
except AttributeError:
    …
```

Duck typing

If it quacks, it's a duck

- Python dit que "si cela caquète, c'est que c'est un canard !"

- Pour Python,
  - il n'est pas nécessaire de vérifier le type de l'objet auquel on s'adresse, mais seulement qu'
  - il implémente les méthodes dont on a besoin.

Duck typing pratique

Don't !

```
def mul_list(m):
  if isinstance(m, list): ####
    acc = 1
    for i in m:
      acc *= i
    return acc
   else:
    raise TypeError("Arg should
                     be a list")
```

Do !

```
def mul_list(m):
  if hasattr(m, '__iter__') or \
     hasattr(m, '__getitem__'): ####
    acc = 1
    for i in m:
      acc *= i
    return acc
   else:
    raise TypeError("Arg should
                     be a sequence")
```

Polymorphisme

- Le polymorphisme est une caractéristique du langage qui permet d'utiliser les mêmes opérateurs / fonctions /méthodes pour des objets de classes différentes
  - L'opérateur + désigne l'addition des entiers, l'addition des nombres, mais aussi la concaténation des séquences.
  - len() est une fonction qui donne le nombre d'éléments d'un conteneur (chaîne, liste, tuple, ensemble ou dictionnaire)
  - Une classe fille peut utiliser/redéfinir une méthode de la classe mère (polymorphisme par héritage). On parle de "overriding".
    - "Overloading" désigne 2 méthodes de même nom dans une classe mais avec des signatures différentes. Non supporté par Python. Pour y arriver, on utilise les *args, **kwargs.
- Ecrire ainsi des algorithmes plus génériques

Classe abstraite

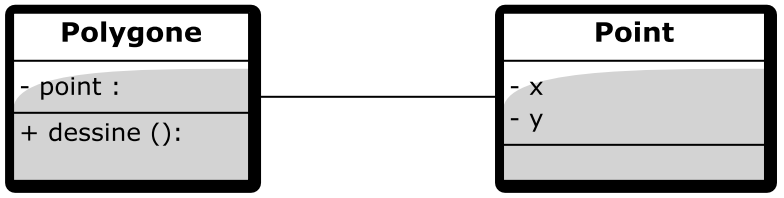
- *Abstract Base Class*, ABC
- Classes qui contiennent des méthodes non implémentées ("…pass")
- On définit des sous-classes, mais n'instancie pas une classe abstraite
- Il est nécessaire de surcharger toutes les méthodes non implémentées
- Le module abc définit la class abstraite ABC et des annotations telles que @abstractmethod qui personnalisent les messages d'erreurs à l'exécution lorsque l'on tente d'instancier une classe abstraite sans avoir surchargé les méthodes abstraites

```
import abc

class Animal(abc.ABC)
  @abc.abstractmethod
  def move(self):
    pass

class Dog(Animal):
  def bark(self):
    print("Waf")

class Fish(Animal):
  def move(self):
    swim()

a = Animal() # Fails !
d = Dog() # Fails !
f = Fish() # Succeeds !
```

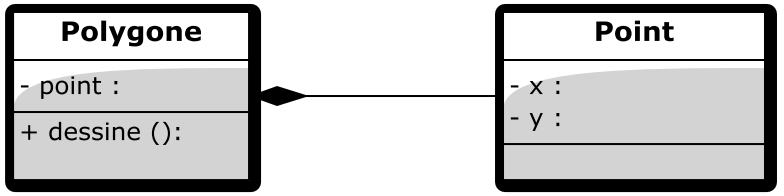
Composition

- La composition modélise la relation "a-un", *has-a*
  - une voiture a-un numéro de série
  - une voiture a-un moteur
  - une voiture a-des roues
- Un objet référence alors d'autres objets
- UML distingue graphiquement les sens de telles associations en prenant en compte :
  - associations d'objets dont l'un est dépendant de l'autre
  - agrégation lorsque les objets ont des existences indépendantes
  - arité de la relation

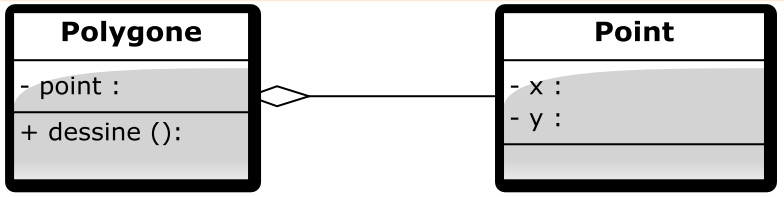
```
# Association
```




```
# Agrégation 1:1
```




```
# Agrégation  1:n
```



Object **O**riented **D**esign

- Une classe doit être fermée à la modification mais ouverte à l'extension
  - Ajouter des membres et des méthodes
  - Préserver un interface stable
  - Ne pas altérer le source de la classe
  - Mais étendre ses fonctionnalités dans des classes filles.

Itérateur

- Pour faire une boucle for … in
  sans nécessairement se préoccuper de la structure de données sous-jacente
- L'entrée dans la boucle crée un objet de la classe itérateur
- Le passage dans la boucle engendre l'appel à next()
  - Jusqu'à l'exception qui commande l'arrêt
- Un générateur peut être utile pour réaliser une classe Itérateur.

```
for i in Iterateur():
```

```
class Iterateur:
    def __init__(self):
        pass

    def __iter__(self):
        …
        return self


    def __next__(self):
        …
        return xx #la valeur suivante
        …
        raise StopIteration
```

Classeur Jupyter Générateurs.ipynb

Que fait une boucle **for** ?

- Lorsque l'on écrit

```
for x in iterable:
  action
```

- C'est comme si on avait

```
try:
  while True:
    x = next(iterable)
    action
except StopIteration:
  pass
```

- Un itérable est une instance d'itérateur

with (retour sur…)

- expr représente un gestionnaire de contexte qui possède les méthodes
  - __enter__
  - __exit__
- action prend place entre enter et exit
- le gestionnaire de contexte garantit que exit est exécutée.
- utile pour acquérir puis libérer une ressource
  - Verrou
  - Transaction d'une bdd
  - et pas seulement ouverture/fermeture de fichier

```
with expr as var :
    action
```

```
class CtxtMgr:
  def __init__(self):
    pass

  def __enter__(self):
      ...
      return self

  def __exit__(self):
```

Lock

"*lock*", verrou

```
from threading import Lock
```

- usage d'un verrou
  - acquire est supposé non interruptible
  - utile lorsqu'il y a plusieurs fils d'exécution concurrents

```
lock.acquire()
try:
  # Traitement exclusif
  f(x,y)
finally:
  lock.release()
```

- La classe Lock est un gestionnaire de contexte

```
with lock:
  # Traitement exclusif
  f(x,y)
```

Module de classe 1/4

- Une bonne pratique quand on entreprend un développement de manière incrémentale (en commençant par un petit bout…) consiste à emballer son code dans une classe

```
class A:
  """ docstring de la classe """
  def __init__(self):

  def __str__(self):

  def __repr__(self):
```

- Définir cette classe, avec toutes ses méthodes indispensables pour instancier, afficher, mais aussi pour l'utiliser, pour le valider …

```
def main():
  a = A()
  print(a)


main()
```

- A la suite de la définition de la classe, inclure une fonction main() qui appelle les méthodes ci dessus

Module de classe 2/4

- La fonction main montre une utilisation de la classe, une démo
- Elle peut contenir un autotest
- L'exécuter vérifie le bon fonctionnement
- Une bonne pratique consiste à faciliter l'import de cette classe dans un projet
- Ne pas exécuter la fonction main si le module de classe est importé

```
def main():
  a = A()
  print(a)
  assert a.z == 12.34
  a.demo()
```

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

Module de classe 3/4

- La fonction main peut renvoyer un code de retour :
  - 0 : bien passé,
  - sinon : mal passé

```
def main():
  ...
  return 0
  ...
  return 1
```

- Une bonne pratique consiste à transmettre ce code de retour au shell

- La fonction exit de la bibliothèque sys transmet ce code retour

```
import sys
```

```
if __name__ == '__main__':
  status = main()
  sys.exit(status)
```

Module de classe 4/4

- Organisation modèle d'un module de classe

template

```
"""module docstring"""

# imports
# constants
# exception classes
# functions
# classes
# internal functions & classes

def main(…):
    …

if __name__ == '__main__':
    status = main()
    sys.exit(status)
```

Exercice module de classe

- Mettre l'exercice précédent (IMC, convertisseur de devise, trinome, ou autre) dans une classe et proposer un module de classe incluant :
  - la définition de la classe
  - au moins un test
  - une démo de son bon fonctionnement

  prêt pour l'intégration dans un projet plus vaste

dataclasses

- Définir une classe avec
  - Les variables d'instance
  - les méthodes essentielles

```
import dataclasses

@dataclassses.dataclass
class Car:
    num: str
    brand: str
    model: str
```

- Ajouter des variables de classes

```
from typing import ClassVar
    …
    nbcars : ClassVar[int]=0
```

- Si besoin, compléter la méthode init qui a été fabriquée par @dataclass

```
def __post_init__(self):
    …
```

Module de classe revisité

- Reprenez votre module de classe et introduisez le décorateur Dataclass pour alléger votre code des parties répétitives que vous écrivez à chaque fois

Mixin

- Le Mixin est une classe :
  - Elle contient des méthodes /attributs à (ré)utiliser
  - Noter l'absence de __init__

```
class Mixin:
  def f(self):
    ...
  def g(self):
    ...
```

- Soit une classe B à laquelle on veut ajouter les méthodes f et g

```
class B(A):
  ...
```

- On utilise l'héritage multiple :
  - les instances de B ont les méthodes f et g.

```
class B(Mixin, A):
  ...
```

- Utile quand ont veut ajouter
  - beaucoup d'attributs/méthodes optionnels à une classe, ou bien
  - le même attribut/méthode à beaucoup de classes

Les mots-clés ajoutés

False	None	True	and	as	assert
async	await	break	class	continue	def
del	elif	else	except	finally	for
from	global	if	import	in	is
lambda	nonlocal	not	or	pass	raise
return	try	while	with	yield	

Exercice Polygones : la classe Point

- Définir une classe Point dont les membres sont : x,y, des flottants.
  - p = Point(10.5,26.3) p.x=10.5 et p.y=26.3

- Ajouter une méthode move qui à partir du point en cours fabrique un nouveau point situé dans la direction theta à une distance rho.
  - def move(self, rho, theta):
    """ retourne un nouveau point"""

- Ajouter une méthode milieu qui étant donné un deuxième point donne le point milieu du segment formé avec le point en cours
  - def milieu(self,p):
    """ retourne le point milieu de self et de p"""

- Ajouter une méthode distance qui donne la distance d'un point au point en cours
  - def distance(self, p):
    """ retourne un nombre positif qui est la distance de self au point p)

Rappel de trigonométrie ;-)

- Je suis en x,y
- Je bouge de rho dans la direction theta
- pour arriver en x1,y1

x 1 = x + rho * cos(theta)

y1 = y + rho * sin(theta)

Géométrie

- A(xa, ya)
- B(xb, yb)

- M(xm, ym) tel que M est le milieu de AB
  AM = MB

xm = (xa+xb)/2

ym = (ya+yb)/2

Géométrie

- A(xa, ya)
- B(xb, yb)

La distance du point A au point B est d(A,B)

$d(A,B) = \text{racine}((xb-xa)^2 + (yb-ya)^2)$

Exercice Polygones : la classe Polygone

- Ecrire une classe Polygone qui a un membre "sommets" constitué d'une liste de points et un membre "couleur" (chaîne de caractère: "blue", "black", par ex).
  - sommets = liste de points
  - couleur = chaine de caractères
- Surcharger (redéfinir), la méthode len pour qu'elle donne le nombre de sommets du polygone
- Ecrire la méthode périmètre qui donne le périmètre du polygone
- Ecrire une méthode dessine qui dessine le polygone

Exercice Polygones: la classe Rectangle

- Ecrire une classe Rectangle
- On convient que le Rectangle est défini par
  - Un sommet de départ
  - Un angle donnant la direction de la longueur, issue de ce sommet
  - la longueur,
  - la largeur
- Comment réutiliser au mieux les méthodes de la classe mère ?
- Quelles méthodes est il préférable d'écraser (override ?)

Exercice

- Créer un type IntList pour des listes qui ne peuvent contenir que des entiers
  - quelles méthodes devez-vous surcharger ?

Merci !

- Restons en contact :
  - Georges Georgoulis – ggeorgoulis@alteractifs.org – 06 12 68 40 06

  - Alteractifs

    Coopérative d'activité et d'entrepreneurs www.alteractifs.org