R208

Analyse et traitement de données structurées

Manuel Munier

UPPA STEE - IUT des Pays de l'Adour - Département RT
manuel.munier@univ-pau.fr
https://munier.perso.univ-pau.fr/teaching/butrt-r208/

2022-2023

Plan du cours

Plan du cours

Ce que dit le PPN

- Objectifs du module
  - ▷ Structure d'un programme : arborescence de fichiers, modules et packages
  - ▷ Structure complexe de données : listes 2D, tableaux associatifs/dictionnaires
  - ▷ POO $\rightsquigarrow$ notion de classes : instance, attributs, méthodes
  - ▷ Manipulation de fichiers avancée
- Compétences visées (entre autres)
  - → Lire, exécuter, corriger et modifier un programme
  - → Traduire un algorithme, dans un langage et pour un environnement donné
  - → Choisir les mécanismes de gestion de données adaptés au développement de l'outil
- Pré-requis
  - ▷ R107 : Fondamentaux de la programmation

Ce que l'on va (essayer) de faire...

- Contenu de ce module
  - ✓ structures de données
  - ✓ algorithmique avancée
  - ✓ introduction à la programmation orientée objet
  - ✓ lecture/écriture de fichiers
- **Ce que nous ne ferons pas ! ! !**
  - ✗ Algorithmique : boucles, tests, tableaux à 1 dimension, fonctions,...
  - ✗ Ligne de commande : gestion fichiers & répertoires,...

  ⇒ `if (lacunes>0) then do_révisions([R107]);`

Pourquoi s'intéresser aux données ?

- Sur de petits projets
  - peu de données $\rightsquigarrow$ de "simples" tableaux suffisent
  - peu de composants logiciels $\rightsquigarrow$ échanges $\equiv$ passage de paramètres & return dans les fonctions
  - et parfois... on structure "un peu" $\Rightarrow$ tuples, tableaux à plusieurs dimensions, etc.
- Mais dans la vraie vie...
  - **beaucoup** plus de données
  - des données **structurées** : enregistrements, listes, dictionnaires, objets, etc.
  - différentes **technologies** selon les usages
  - traitements plus **complexes**

Plan du cours

Déroulement

- Très peu de temps (2 cours d'1h30 ☹), donc :
  - pas le temps de faire de la théorie ⇒ apprentissage par la pratique, sur des exemples
  - on va se concentrer sur les fondamentaux :
    - **structures des données** ⇝ enregistrements (aka tuples), tableaux à plusieurs dimensions, listes, dictionnaires,...
    - algorithmique associée, et en particulier la notion de récursivité
    - **programmation orientée objet** ⇝ objets, classes, attributs, méthodes,...
    - lecture et écriture dans des fichiers

Plan du cours

Introduction

- TODO...

Tuples

- Objectif : regrouper plusieurs valeurs dans une seule variable
  - ⇝ simplifier la gestion des données
  - ⇝ diminuer le nombre de variables et de paramètres dans les fonctions
- NB : dans d'autres langages on parle aussi d'enregistrements
- Exemples :
  - `date = (2022,3,1)`
  - `cours = ("R208","Structures de données",3,4.5,9)`
  - `personne = ("John","Doe",(1970,4,1),"male",("371","rue du Ruisseau","40000","Mont de Marsan"))`

Tuples

- Python fournit plusieurs outils du langage sur les tuples

composition

```
jour  = 1
mois  = 3
annee = 2022
date  = (annee,mois,jour)
```

décomposition

```
date = (2022,3,1)
annee,mois,jour = date
```

comparaison

```
date1 = (2022,3,1)
date2 = (2022,2,28)
if (date1 < date2):
    print(date1,"avant",date2)
else:
    print(date1,"après",date2)
```

Tuples

- Python fournit plusieurs outils du langage sur les tuples

⚠ l'opérateur + ne réalise pas l'addition champs par champs mais "fusionne" les tuples
⇝ comme avec des listes...

addition

```
v1 = (1,3)
v2 = (2,4)
v3 = v1+v2
print(v3)
```

affichage

```
(1, 3, 2, 4)
```

Tuples

- Python fournit plusieurs outils du langage sur les tuples
  - on peut aussi manipuler les tuples "comme des tableaux"...

opérateur [ ]

```
date = (2022,3,1)
mois = date[1]
print(mois)
```

Tableaux

- Vous savez déjà utiliser des tableaux "simples" $\rightsquigarrow$ module R107
- Mais il est aussi possible de mettre des tableaux dans des tableaux
  - $\Rightarrow$ tableaux multi-dimensionnels
  - $\triangleright$ exemples :
    - tables (comme en SQL)
    - matrices 2D, 3D,...
- NB : **avantages de Python** sur d'autres langages plus classiques (ex : C, Java)
  - $\Rightarrow$ les valeurs des cellules peuvent être de types différents
  - $\Rightarrow$ les lignes peuvent avoir des longueurs différentes

Tableaux

- Exemple de tableau à 2 dimensions

programme

```
tab = [ [1,2,3], [4,5,6], [7,8,9] ]

for i in range(3):
    s = ""
    for j in range(3):
        s = s + str(tab[i][j]) + " "
    print(s)
```

affichage

```
1 2 3
4 5 6
7 8 9
```

Tableaux

- La fonction `len()` permet de connaître la taille d'un tableau

programme

```
tab = [ [1,2,3],
        [4,5,6,7,8],
        [9],
        [10,11,12,13] ]

print("Il y a %d lignes" % len(tab))

for i in range(len(tab)):
    print("Ligne %d -> %d cellules" % (i, len(tab[i])))
```

affichage

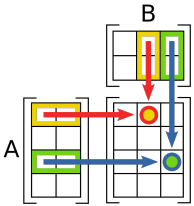
```
Il y a 4 lignes
Ligne 0 -> 3 cellules
Ligne 1 -> 5 cellules
Ligne 2 -> 1 cellules
Ligne 3 -> 4 cellules
```

Tableaux

- Exercice : produit matriciel (cf. Wikipédia)
  - Si $A = (a_{ij})$ est une matrice de type (m, n) et $B = (b_{ij})$ est une matrice de type (n, p), alors leur produit, noté $AB = (c_{ij})$ est une matrice de type (m, p) donnée par :

$$\forall i, j : c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} = a_{i1} b_{1j} + a_{i2} b_{2j} + \cdots + a_{in} b_{nj}$$

  - La figure suivante montre comment calculer les coefficients c_{12} et c_{33} de la matrice produit AB si A est une matrice de type $(4, 2)$, et B est une matrice de type $(2, 3)$.



$$c_{12} = \sum_{r=1}^{2} a_{1r} b_{r2} = a_{11} b_{12} + a_{12} b_{22}$$

$$c_{33} = \sum_{r=1}^{2} a_{3r} b_{r3} = a_{31} b_{13} + a_{32} b_{23}$$

Tableaux

produit matriciel (cf. Wikipédia)

```
def affiche(m):
    for i in range(len(m)):
        s = ""
        for j in range(len(m[i])):
            s = s + str(m[i][j]) + " "
        print(s)
    print()

def produit(a,b):
    c = []
    for i in range(len(a)):
        ligne = []
        for j in range(len(b[0])):
            temp = 0
            for k in range(len(b)):
                temp += a[i][k] * b[k][j]
            ligne.append(temp)
        c.append(ligne)
    return c
```

produit matriciel (suite)

```
m1 = [[1,0],[2,-1]]
m2 = [[3,4],[-2,-3]]
affiche(produit(m1,m2))

m3 = [[1,2,0],[4,3,-1]]
m4 = [[5,1],[2,3],[3,4]]
affiche(produit(m3,m4))

affiche(produit(m4,m3))
```

Dictionnaires

- Une source d'information sur le net → les dictionnaires python
- Objectif : associer des valeurs à des clés
  - ⇝ "comme un tableau", mais on accède aux valeurs à partir de leur clé et non de leur position ⇒ la recherche d'une valeur est faite (efficacement) par Python
  - ⇝ "comme une liste", mais chaque valeur doit avoir une clé
- Avantages
  - ▷ meilleure structuration des données
  - ▷ approche similaire aux formats JSON & co.

Dictionnaires

- Exemple

programme

```
dico = {}
dico["R207"] = "Sources de données"
dico["R208"] = "Analyse et traitement de données structurées"
dico["R209"] = "Initiation au développement Web"
dico["SAE23"] = "Mettre en place une solution informatique pour l'entreprise"

print(dico)
print(dico.get("R208"))
```

- Affichage

```
{'R207': 'Sources de données', 'R208': 'Analyse et traitement de données structurées', 'R209': 'Initiation au développement Web
Analyse et traitement de données structurées
```

Dictionnaires

- Python fournit plusieurs outils du langage sur les dictionnaires
  - ▷ récupérer les clés d'un dictionnaire (`.keys()`)

```
for cle in dico.keys():
    print cle
```

  - ▷ récupérer les valeurs d'un dictionnaire (`.values()`)

```
for valeur in dico.values():
    print(valeur)
```

  - ▷ récupérer les entrées d'un dictionnaire (`.items()`)

```
for cle,valeur in dico.items():
    print(cle,valeur)
```

Dictionnaires

- Python fournit plusieurs outils du langage sur les dictionnaires
  - vérifier si une clé est présente dans un dictionnaire (`.has_key()`)
    ```
    print dico.has_key("R208"):
    ```
  - supprimer une entrée (`del`)
    ```
    del dico["R208"]
    ```

Dictionnaires

- On peut aller encore plus loin...
  - ▷ bien d'autres outils du langage Python sur les tableaux, les listes, les tuples, les dictionnaires
  - ▷ combiner / emboîter ces types de données "élémentaires" pour construire des structures de données bien plus "intéressantes" ☺

⇒ cette étape de structuration des données dans un projet est **très importante** en génie logiciel

NB étape d'autant plus indispensable que Python est un langage non typé, c'est-à-dire que l'on peut mélanger des données de types différents dans ces structures de données

Récursivité (intro)

- Définition(s) sur Wikipédia
  - la définition de certaines **structures de données**, comme les listes ou les arbres, est récursive : elle mentionne le type de données en train d'être défini
  - une **fonction** ou plus généralement un algorithme peut contenir un ou des appels à lui-même, auquel cas il est dit récursif
- On parle aussi de **structures de données dynamiques** car on peut connecter / déconnecter des nœuds à la volée : listes, arbres, graphes

Récursivité (intro)

- Exemple de fonction récursive : factorielle
- Définition mathématique $n! = \begin{cases} 1 & \text{si } n = 0 \\ n * (n-1)! & \text{si } n > 0 \end{cases}$

version itérative

```
n = int(input("N = "))

f = 1
for i in range(n):
    f *= (i+1)

print("N! = %d" % f)
```

version récursive

```
def factorielle(a):
    if (a==0):
        return 1
    else:
        return a * factorielle(a-1)

n = int(input("N = "))
print("N! = %d" % factorielle(n))
```

Plan du cours

Introduction : motivations

- Il est parfois nécessaire de **regrouper** plusieurs données dans une seule variable
  - 3 entiers `jour`, `mois`, `année` pour représenter une date
  - `nom`, `prénom`, `date_naissance`,... pour une personne
  - `ref`, `libellé`, `prix`, `catégorie`,... pour un article
  - `point` (lui même composé de `X` et `Y`), `longueur`, `largeur` pour un rectangle

⇒ On définit pour cela des structures de données
  - ⇝ tuple en Python
  - ⇝ `record` en Pascal
  - ⇝ `struct` en langage C

Introduction : motivations

Pb n°1 Si les données sont mieux structurées, cela reste néanmoins très "statique"

- les données d'un côté
- les traitements (fonctions) en "vrac" dans le programme

Pb n°2 Du coup, les fonctions doivent connaître la structure (interne) de ces données pour pouvoir les manipuler

- `lendemain(date)`
- `ajouter(date,nb_jours)`
- `surfaceRect(rectangle)`, `surfaceCercle(cercle)`,...
- `moyenne(tableau,nb_cases_remplies)`
- `ajouter(tableau,nb_cases_remplies,valeur)`

Introduction : concept d'objet

Définition d'un objet

L'idée consiste à **encapsuler** au sein d'une même entité à la fois

→ les données (appelées **attributs**)

→ les traitements (appelés **méthodes**)

- Les méthodes ont accès à tous les attributs de l'objet sur lequel elles s'exécutent (en plus de leurs paramètres)
- Exemples :
  - `date.lendemain()`
  - `date.ajouter(nb_jours)`
  - `rectangle.surface()`, `cercle.surface()`,...
  - `tableau.moyenne()`, `tableau.ajouter(valeur)`

Introduction : concept d'objet

Abstraction de données

Seules les méthodes d'un objet ont accès aux attributs de cet objet.

- Les attributs sont "masqués"
  - ⇒ ils ne sont pas accessibles de l'extérieur
  - ⇒ l'objet ne peut être manipulé qu'au travers de ses méthodes

⇒ Modularité
- les objets peuvent être testés individuellement
- un changement sur les attributs n'impacte pas l'extérieur (à condition que les signatures des méthodes restent identiques)
- les algorithmes sont plus "clairs" ☺

Objets en Python

- En POO il est donc nécessaire de définir la structure des objets que l'on va utiliser
  - ▷ **attributs** ⇝ quelles sont les informations "portées" par chaque objet
  - ▷ **méthodes** ⇝ "fonctions" attachées à chaque objet
  - ▷ **constructeurs** ⇝ fonctions spéciales permettant de créer un objet

⇒ En POO, une **classe** est un type d'objet

- **Problème :** Contrairement à Java, C++ ou Pascal, le langage Python est **non typé** (comme JavaScript ou PHP d'ailleurs)

Objets en Python

- La définition de classes en Python n'est qu'une "surcouche syntaxique" ☹
  - ▷ on peut parler de programmation par prototypage

⇒ Inconvénients
- ▷ pas "POO only" : on peut mixer objets et fonctions classiques (même pb en C++) ⇒ peut perturber les débutants en OO
- ▷ quiconque peut ajouter des attributs et des méthodes à des objets, sans pour autant "mettre à jour" la classe ⇝ pas très rigoureux...

⇒ Avantages
- ▷ peut apporter plus de souplesse... quand on sait exactement ce que l'on fait !

Objets en Python

- Par contre, autant prendre tout de suite de "bonnes habitudes" en POO...
  - pour programmer "proprement" en OO (cf. GL)
  - pour bien comprendre l'intérêt des approches OO
    - programmation par assemblage de composants
    - notion de services $\rightsquigarrow$ SOA
    - dissociation interface et implémentation des services
  - pourvoir faire de l'OO dans d'autres langages, parfois plus stricts...

Objets en Python : exemple n°1 (Point)

- Nous voulons définir des objets "points" composés de 2 coordonnées x et y

v1 Version "basique" : comme un simple tuple... pas vraiment POO !!!

- rien dans la classe ⇝ pas d'encapsulation
- les attributs sont ajoutés par le main ⇝ pas d'abstraction

classe Point

```
class Point:
        "Définition d'un point"

def main():
        pt = Point()
        pt.x = 10.5
        pt.y = 5.4
        print("pt=(%2.1f,%2.1f)" % (pt.x,pt.y))

# Programme principal
main()
```

affichage

```
pt=(10.5,5.4)
```

Objets en Python : exemple n°1 (Point)

- Remarque : chaque objet a ses propres attributs

classe Point

```
class Point:
        "Définition d'un point"

def main():
        pt = Point()
        pt.x = 10.5
        pt.y = 5.4
        print("pt=(%2.1f,%2.1f)" % (pt.x,pt.y))

        pt2 = Point()
        pt2.x = -3
        pt2.y = 12
        print("pt=(%2.1f,%2.1f)" % (pt.x,pt.y))
        print("pt2=(%2.1f,%2.1f)" % (pt2.x,pt2.y))

# Programme principal
main()
```

affichage

```
pt=(10.5,5.4)
pt=(10.5,5.4)
pt2=(-3.0,12.0)
```

Objets en Python : exemple n°1 (Point)

- Remarque : distinction entre variable et objet $\rightsquigarrow$ notion de référence
  - ici, les 2 variables pt2 et pt sont des références vers le même objet en mémoire !

classe Point

```
class Point:
        "Définition d'un point"

def main():
        pt = Point()
        pt.x = 10.5
        pt.y = 5.4
        print("pt=(%2.1f,%2.1f)" % (pt.x,pt.y))

        pt2 = pt
        pt2.x = -3
        pt2.y = 12
        print("pt=(%2.1f,%2.1f)" % (pt.x,pt.y))
        print("pt2=(%2.1f,%2.1f)" % (pt2.x,pt2.y))

# Programme principal
main()
```

affichage

```
pt=(10.5,5.4)
pt=(-3.0,12.0)
pt2=(-3.0,12.0)
```

Objets en Python : instanciation

- En POO il faut impérativement distinguer 2 choses :
  - l'**objet** lui même
    - ⇝ *une zone de la mémoire contenant les attributs, les méthodes, sa classe (~type de l'objet), etc...*
  - une **référence** sur cet objet
    - ⇝ *"l'adresse" de cette zone mémoire (~pointeur)*
    - ⇝ *plusieurs références (variables différentes) peuvent désigner le même objet (même "adresse" comme valeur)*

Instanciation

Une **instance** est un objet créé à partir d'une classe par un mécanisme appelé **instanciation**.

Objets en Python : exemple n°1 (Point)

- Une **méthode** est une "fonction" s'exécutant sur l'état d'un objet

v2 Ajout d'une méthode

- le 1er paramètre doit être `self` (réf. à l'objet lui-même)

classe Point

```
class Point:
        "Definition d'un point"
        def deplacer(self, dx, dy):
                self.x = self.x+dx
                self.y = self.y+dy

def main():
        pt = Point()
        pt.x = 10.5
        pt.y = 5.4
        print("pt=(%2.1f,%2.1f)" % (pt.x,pt.y))

        pt.deplacer(3,2)
        print("pt=(%2.1f,%2.1f)" % (pt.x,pt.y))

# Programme principal
main()
```

affichage

```
pt=(10.5,5.4)
pt=(13.5,7.4)
```

Constructeurs

- Modularité & tests (GL) $\rightsquigarrow$ une méthode prend un objet "stable" et le rend dans un état "stable"

Pb Quel est l'état [1] de l'objet lors de son instanciation ?

Constructeur

Un **constructeur** est une méthode particulière qui est invoquée automatiquement lors de l'instanciation d'un objet. Son objectif est d'initialiser tous les attributs de l'objet.

- En Python, un constructeur porte le nom `__init__()`

1. valeurs de ses attributs

Constructeurs : exemple n°1 (Point)

v3 Notion de constructeur pour initialiser "proprement" l'objet

- on constate que le constructeur a bien créé les 2 attributs x et y

classe Point

```
class Point:
        "Définition d'un point"
        def __init__(self):
                self.x = 0
                self.y = 0

        def deplacer(self, dx, dy):
                self.x = self.x+dx
                self.y = self.y+dy

def main():
        pt = Point()
        print("pt=(%2.1f,%2.1f)" % (pt.x,pt.y))

        pt.deplacer(3,2)
        print("pt=(%2.1f,%2.1f)" % (pt.x,pt.y))

# Programme principal
main()
```

affichage

```
pt=(0.0,0.0)
pt=(3.0,2.0)
```

Constructeurs : exemple n°1 (Point)

v3.1 Utilisation de constructeurs paramétrés

- le constructeur a bien initialisé les 2 attributs x et y avec les 2 paramètres

classe Point

```
class Point:
        "Définition d'un point"
        def __init__(self,a,b):
                self.x = a
                self.y = b

        def deplacer(self, dx, dy):
                self.x = self.x+dx
                self.y = self.y+dy

def main():
        pt = Point(10.5,5.4)
        print("pt=(%2.1f,%2.1f)" % (pt.x,pt.y))

        pt.deplacer(3,2)
        print("pt=(%2.1f,%2.1f)" % (pt.x,pt.y))

# Programme principal
main()
```

affichage

```
pt=(10.5,5.4)
pt=(13.5,7.4)
```

Constructeurs : exemple n°1 (Point)

NB Contrairement à la quasi majorité des langages OO, Python ne supporte malheureusement pas la **surcharge de méthodes**

- en Python, la 2ème définition du constructeur va remplacer la 1ère ☹

classe Point ⇝ erreur

```
class Point:
        "Définition d'un point"
        def __init__(self):
                self.x = 0
                self.y = 0

        def __init__(self,a,b):
                self.x = a
                self.y = b

def main():
        pt1 = Point() # -> erreur car ce constructeur n'existe plus !
        print("pt=(%2.1f,%2.1f)" % (pt1.x,pt1.y))
        pt2 = Point(10.5,5.4)
        print("pt=(%2.1f,%2.1f)" % (pt2.x,pt2.y))

# Programme principal
main()
```

classe Point

```
class Point:
        "Définition d'un point"
        def __init__(self,a=0,b=0):
                self.x = a
                self.y = b

def main():
        pt1 = Point() # -> ok; utilise les valeurs par défaut
        print("pt=(%2.1f,%2.1f)" % (pt1.x,pt1.y))
        pt2 = Point(10.5,5.4)
        print("pt=(%2.1f,%2.1f)" % (pt2.x,pt2.y))

# Programme principal
main()
```

Encapsulation : exemple n°1 (Point)

v4 Encapsulation ≡ attributs protégés

- le nom des attributs est préfixé par __ ⇒ attributs privés ⇒ **méthodes d'accès**

classe Point

```
class Point:
        "Définition d'un point"
        def __init__(self,a,b):
                self.__x = a
                self.__y = b

def main():
        pt = Point(10.5,5.4)
        print("pt=(%2.1f,%2.1f)" % (pt.__x,pt.__y))

# Programme principal
main()
```

affichage ⇝ erreur d'accès

```
Traceback (most recent call last):
  File "test221.py", line 15, in <module>
    main()
  File "test221.py", line 11, in main
    print("pt=(%2.1f,%2.1f)" % (pt.__x,pt.__y))
AttributeError: 'Point' object has no attribute '__x'
```

classe Point

```
class Point:
        "Définition d'un point"
        def __init__(self,a,b):
                self.__x = a
                self.__y = b

        def afficher(self):
                print("pt=(%2.1f,%2.1f)" % (self.__x,self.__y))

def main():
        pt = Point(10.5,5.4)
        pt.afficher()

# Programme principal
main()
```

affichage

```
pt=(10.5,5.4)
```

Encapsulation

- Si les attributs sont privés, seul l'objet lui-même y a accès

⇒ Tout membre "extérieur" (autre objet, fonction, etc.) devra donc obligatoirement passer par les méthodes d'accès proposées par cet objet

  - ▷ **constructeurs** pour initialiser l'objet
  - ▷ **accesseurs** pour lire les attributs (ex : des méthodes `get_XXX()`)
  - ▷ **mutateurs** pour modifier les attributs (ex : des méthodes `set_XXX()`)

- Au passage, ces méthodes d'accès peuvent également...
  - filtrer, mettre en forme les informations
  - contrôler les valeurs "injectées", déclencher des calculs, etc.
  - mettre en place une politique d'accès : *read only*, *write only*, etc.

Remarques

- En POO, toute la "mécanique" est cachée dans les objets
  - organisation des attributs
  - algorithmes des méthodes
- Les objets "extérieurs" doivent utiliser les "services" proposés par les objets
  - → constructeurs pour instancier des objets
  - → méthodes (publiques) pour consulter / modifier les objets

⇒ "Propreté" d'un point de vue GL
- traitements locaux
- interfaces bien identifiées
- impact d'un changement minimisé

Ex *modification des spécifications, ajout de fonctionnalités, correction de bug,...*

Héritage

- L'héritage est un mécanisme qui nous permet de créer une nouvelle classe (**classe fille** ou **sous-classe**) qui est basée sur une classe existante (**classe mère** ou **super-classe**), en ajoutant de nouveaux attributs et méthodes en plus de la classe existante.
- Ce faisant, la classe fille **hérite** des attributs et des méthodes de la classe mère

syntaxe

```
class mere:
        # corps de la classe mère

class enfant(mere):
        # corps de la classe enfant
```

Héritage : exemple n°2 (Employe)

classes Personne et Employe

```
class Personne():
        # Constructeur
        def __init__(self, nom, prenom):
                self.nom    = nom
                self.prenom = prenom

        def afficher(self):
                print("Nom    : ",self.nom)
                print("Prénom : ",self.prenom)

class Employe(Personne):
        # Constructeur
        def __init__(self, nom, prenom, job):
                # appel du constructeur de la classe mère (Personne)
                Personne.__init__(self, nom, prenom)

                # ajout d'un attribut
                self.job = job

        def afficher(self):
                Personne.afficher(self)
                print("Job    : ",self.job)
```

programme

```
def main():
        # création d'une variable d'instance
        p=Personne("Doe", "John")
        # appel d'une fonction de la classe Personne via son instance
        p.afficher()

        # création d'une instance de la sous-classe
        e=Employe("Bond","James","agent secret")
        e.afficher()

# Programme principal
main()
```

affichage

```
Nom    : Doe
Prénom : John

Nom    : Bond
Prénom : James
Job    : agent secret
```

Héritage

- L'héritage est un concept **très important** en POO ! ! !
- Via l'héritage nous pouvons définir de nouvelles classe et...
  - ajouter de nouveaux attributs
  - ajouter de nouvelles méthodes
  - "masquer" des attributs de la classe mère
  - redéfinir des méthodes de la classe mère (**surcharge**)
    - Ex méthode `afficher()` de `Personne` redéfinie dans la sous-classe `Employe`
    - ⇒ **liaison dynamique** ≡ choix de la "bonne" méthode lors de l'exécution

Synthèse

- En POO, un programme est un ensemble de petites entités autonomes (les **objets**) qui interagissent et communiquent par messages (les **appels de méthodes**)
- L'accent est mis sur l'autonomie de ces entités et sur les traitements locaux

$\Rightarrow$ Programmer avec des objets nécessite un changement d'état d'esprit de la part du programmeur

Synthèse

- 3 critères de qualité en Génie Logiciel
  - **fiabilité** : un composant fonctionne dans tous les cas de figure
  - **extensibilité** : on peut ajouter de nouvelles fonctionnalités sans modifier l'existant
  - **réutilisabilité** : les composants peuvent être réutilisés (en partie ou en totalité) pour construire de nouvelles applications

$\Rightarrow$ 1 mot : **modularité**

Synthèse

- Cette modularité a un coût
  - → à partir du cahier des charges, il faut identifier les objets
- En amont de la phase de programmation ont ainsi été développées des méthodes d'analyse et/ou de conception orientées objet
  - Ex UML (*the Unified Modeling Language*)
- **Ne concerne pas le module R208 ☺**
  - ⇝ les sujets préciseront les objets/méthodes à programmer
  - ⇝ UML sera abordé "plus tard"...

Plan du cours

Introduction

- TODO...