	**Langages et programmation**	**NSI T^{ale}**
	Paradigmes de programmation	Cours

*

1. Définition

Paradigme :
«Ensemble de méthodes et modèles reconnu par une communauté scientifique permettant à cette communauté de résoudre une famille de problèmes donnés» (Thomas S. Kuhn)

Il existe de nombreux paradigmes de programmation dont certains que nous avons déjà abordés. En voici une liste non exhaustive :

- Programmation descriptive / déclarative
- Programmation impérative
- Programmation procédurale
- Programmation fonctionnelle
- Programmation orientée objet
- Programmation événementielle
- Programmation parallèle
- Programmation logique
- DSL (Domain Specific Language)

2. Exemple

Exercice

Écrire un programme en Python qui permet de calculer la somme d'une liste de nombres.

Exemple de solution -approche «impérative»

```
# Récupère la liste de nombres via le prompt
liste = input("Entrez la liste de nombres : \n").split()

# initialise la variable "somme" qui contiendra le résultat
somme = 0

# boucle sur la liste
for nb in liste:
    somme = somme + float(nb)

# affiche la somme
print(somme)
```

Exemple de solution -approche «procédurale»

```
# définit une fonction pour effectuer le calcul
def somme_liste(liste):
    # initialise la variable "somme" qui contiendra le résultat
    somme = 0
    # boucle sur la liste
    for nb in liste:
        somme = somme + float(nb)
    return somme

# Récupère la liste de nombres via le prompt
liste = input("Entrez la liste de nombre : \n").split()

# affiche la somme
print(somme_liste(liste))

# facile de réutiliser pour tester par exemple
assert(somme_liste([]) == 0)
assert(somme_liste([0, 3.5, 6]) == 9.5)
```

Exemple de solution -approche plus «fonctionnelle»

```
def somme_liste(liste):
    if len(liste) == 0:
        return 0
    return float(liste[0]) + somme_liste(liste[1:])

print(somme_liste(input("Entrez la liste de nombre : \n").split()))
```

Exemple de solution -approche «objet»

```
# On décrit les caractéristiques communes de tous
# les objets d'une même famille à l'aide d'une classe
class Calculateur:
    # Initialisation de l'objet référencé par "self"
    # avec l'attribut "liste" qui est encapsulé dans l'objet
    def __init__(self, liste):
        self.liste = liste

    # L'objet calcule la somme
    # à partir de sa liste
    def somme(self):
        valeur = 0
        for nb in self.liste:
            valeur = valeur + float(nb)
        return valeur

# Récupère la liste de nombres via le prompt
uneListe = input("Entrez la liste : \n").split()

# Créer une instance de Calculateur en l'initialisant
# avec la liste récupérée précédemment
unCalculateur = Calculateur(uneListe)

# On demande au sommeur de faire le calcul
somme = unCalculateur.somme()

# On affiche
print(somme)
```

Exemple de solution -approche Objet avec fonctionnel

```
# On décrit les caractéristiques communes de tous
# les objets d'une même famille à l'aide d'une classe
class Calculateur:
    # Initialisation de l'objet référencé par "self"
    # avec l'attribut "liste" qui est encapsulé dans l'objet
    def __init__(self, liste):
        self.liste = liste

    # L'objet calcule la somme
    # à partir de sa liste
    def somme(self):
        if len(self.liste) == 0: return 0
        return float(self.liste[0]) +
                        Calculateur(self.liste[1:]).somme()

# Programme principal
print(Calculateur(
        input("Entrez la liste de nombre : \n").split()).somme())
      )
```

3. Le paradigme impératif

3.1. Caractéristiques

Programme sous forme de séquence d'instructions devant être exécutées par la machine

3 grandes familles d'instructions

- Instructions d'affectation de variable
- Instructions d'entrées/sorties
- Instructions de contrôle

3.2. Illustration

```
# Récupère la liste de nombres via le prompt
liste = input("Entrez la liste de nombres : \n").split()

# initialise la variable "somme" qui contiendra le résultat
somme = 0

# boucle sur la liste
for nb in liste:
    somme = somme + float(nb)

# affiche la somme
print(somme)
```

Instructions d'entrées/sorties
Instructions d'affectation de variable
Instructions de contrôle

3.3. Langages supportant le paradigme impératif

- Assembleur
- C, C++
- Perl, PHP
- Python
- Java, C#
- Javascript
- Kotlin, Swift
- et une quantité difficilement dénombrable d'autres langages

3.4. Processeurs et modèle impératif

Les processeurs traditionnels reposent sur une architecture ne supportant que le modèle impératif.
La programmation impérative représente en ce sens le niveau d'abstraction le moins élevé par rapport au fonctionnement de la machine.

3.5. Limites

- Pas ou peu de modularité
- Difficilement testable automatiquement
- Le développement de programmes complexes est quasi-impossible
- Une même variable peut être réaffectée
  - Les changements d'état finissent toujours par introduire des effets de bords dans les programmes complexes

Besoin de niveaux d'abstraction plus élevés pour

- plus de modularité
- plus de facilité à concevoir des programmes complexes
- plus de garantie de sûreté, robustesse de ces programmes complexes

4. Le paradigme procédural

4.1. Caractéristiques

De l'impératif augmenté pour plus de modularité

Introduction de la notion de fonction ou de procédure pour

- Factoriser le code réutilisable
- Rendre le programme plus modulaire
  - Le découper en programmes, sous-programmes

Possibilité d'écrire un programme à partir de plusieurs fichiers sources
Possibilité de tester unitairement les fonctions

```
#
# Contenu fichier "somme_liste.py"
#
# Effectue la somme des éléments d'une liste
def somme_liste(liste):
    somme = 0
    # boucle sur la liste
    for nb in liste:
        somme = somme + float(nb)
    return somme

#
# Contenu fichier "test_somme_liste.py"
#
# Importation de la fonction
from illustrations.somme_liste import somme_liste
assert(somme_liste([]) == 0)
assert(somme_liste([0]) == 0)
assert(somme_liste([1, 3]) == 4)
```

```
#
# Contenu fichier "app_somme_liste.py"
#
# Importation de la fonction
from illustrations.somme_liste import somme_liste
# Récupère la liste de nombres via le prompt
liste = input("Liste ? \n").split()
# affiche la somme
print(somme_liste(liste))
```

4.2. Langages supportant le paradigme procédural

- ~~Assembleur~~
- C, C++
- Perl, PHP
- Python
- Java, C#
- Javascript
- Kotlin, Swift
- et une quantité difficilement dénombrable d'autres langages

4.3. Limites

- ~~Pas ou peu de modularité~~
- ~~Difficilement testable automatiquement~~
- ~~Le développement de programmes complexes est quasi-impossible~~
- Tous les programmes complexes ne se conçoivent pas facilement avec une approche procédurale
- Une même variable peut être réaffectée
  - Les changements d'état finissent toujours par introduire des effets de bords dans les programmes complexes

Besoin de niveaux d'abstraction plus élevés pour

- ~~plus de modularité~~
- plus de facilité à concevoir des programmes complexes
- plus de garantie de sûreté, robustesse de ces programmes complexes

5. Le paradigme fonctionnel

5.1. Caractéristiques

Du procédural augmenté pour plus d'abstraction, de sûreté et de robustesse via les principes suivants :

- Fonctions «pures»
- Immutabilité
- Fonctions de premier ordre

5.2. Fonctions "pures"

Les fonctions ne produisent pas d'effets de bord

- Elles ne modifient pas l'état du programme.

Avec la même entrée, une fonction «pure» produira toujours la même sortie.

Illustration fonction non «pure»

```
#
# Contenu fichier "fonctions_impures.py"
#
# Multiplie par 2 les nombres de la liste - version "impure"
def multiplie_par_2(liste):
    for i in range(0,len(liste)):
        liste[i] = 2 * liste[i]
    return liste

uneListe = [0,1,2,3]
assert multiplie_par_2(uneListe) == [0,2,4,6], "erreur appel 1"
assert multiplie_par_2(uneListe) == [0,2,4,6], "erreur appel 2"
```

Erreur au deuxième appel !

Illustration fonction «pure»

```
#
# Contenu fichier "fonctions_impures.py"
#
# Multiplie par 2 les nombres de la liste - version "pure"
def multiplie_par_2(liste):
    newListe = []
    for i in range(0,len(liste)):
        newListe.append(2 * liste[i])
    return newListe

uneListe = [0,1,2,3]
assert multiplie_par_2(uneListe) == [0,2,4,6], "erreur appel 1"
assert multiplie_par_2(uneListe) == [0,2,4,6], "erreur appel 2"
```

Aucune erreur !

5.3. Immutabilité

Les données ne peuvent pas être modifiées après leur création.

Exemple : la création d'une liste de 3 éléments et son stockage dans une variable `my_list`.

- Si `my_list` est **immutable** (ou **immuable**), vous ne pourrez pas remplacer un des éléments de la liste par un autre. La création d'une nouvelle liste sera nécessaire.

Illustration immutabilité

```
#
# Contenu fichier "immutabilite.py"
#
# Multiplie par 2 les nombres de la liste - version "impure"
def multiplie_par_2(liste):
    for i in range(0,len(liste)):
        liste[i] = 2 * liste[i]
    return liste

uneListe = (0,1,2,3) # c'est un tuple : immuable en python
assert multiplie_par_2(uneListe) == [0,2,4,6], "erreur appel 1"
assert multiplie_par_2(uneListe) == [0,2,4,6], "erreur appel 2"
```

```
TypeError: 'tuple' object does not support item assignment
```

Travailler avec des structures immutables renforce l'utilisation de fonctions «pures»

Point d'attention

L'immutabilité et l'utilisation de fonction «pure» peut avoir un impact très négatif sur les ressources mémoires du programme puisque cela induit une création massive de nouveaux «objets».

L'implantation des types immuables dans le langage utilisé est critique pour éviter ces problèmes d'explosion mémoire.
Exemples :

- Kotlin a ré-implanté la plupart des classes collections dans une version immuable compatible avec une approche fonctionnelle «pure» et efficace.
- JavaScript propose des librairies avec des structures alternatives codées «immuables» proprement.

L'approche consiste principalement à réutiliser les données dès que possible :

- une nouvelle liste peut recycler les objets communs avec la liste initiale à partir de laquelle elle est créée.

5.4. Fonctions de premier ordre

Les fonctions peuvent accepter d'autres fonctions comme paramètres et les fonctions peuvent renvoyer de nouvelles fonctions comme résultat. Elles permettent l'abstraction au niveau des actions.

Illustration fonctions de premier ordre

```
# Compte les éléments de la liste qui satisfont
# la fonction de validation
def count(liste, functionValidation):
    res = 0
    if (len(liste) == 0):
        return res
    if (functionValidation(liste[0])):
        res += 1
    return res + count(liste[1:], functionValidation)
```

```python
# compte les nombres pairs dans la liste
def estPair(nombre):
    return nombre % 2 == 0

assert count([0, 1, 3, 6, 9], estPair) == 2

# compte les nombres pairs avec une fonction anonyme
assert count([0, 1, 3, 6, 9], lambda x: x % 2 == 0) == 2

# compte les couples qui ont leur premier élément
# différent du second
assert count([('A','A'),('G','G'),('T','C')],
                              lambda x:x[0] != x[1]) == 1
```

5.5. Langages supportant le paradigme fonctionnel

- ~~*Assembleur*~~
- ~~*C, C++*~~
- ~~*Perl, PHP*~~
- *Python*
- *Java, C#*
- *JavaScript*
- *Kotlin, Swift, Scala*
- ***Haskell, F#, Lisp, Caml, et autres langages purement fonctionnels***
- *et une quantité difficilement dénombrable d'autres langages*

5.6. Limites

- ~~Pas ou peu de modularité~~
- ~~Difficilement testable automatiquement~~
- ~~Le développement de programmes complexes est quasi-impossible~~
- Tous les programmes complexes ne se conçoivent pas facilement avec une approche fonctionnelle
- ~~Une même variable peut être réaffectée~~
  - ~~Les changements d'état finissent toujours par introduire des effets de bords dans les programmes complexes~~

Besoin d'un niveau d'abstraction ~~plus élevés~~ différent pour

- ~~plus de modularité~~
- plus de facilité à concevoir ~~des~~ certains programmes complexes
- ~~plus de garantie de sûreté, robustesse de ces programmes complexes~~

6. Le paradigme objet

6.1. Origines

- 1967 -Simula 67 : première implantation des classes
- 1969 -Simula 69
- 1980 -Smalltalk-80
  - Alan Kay
  - Premier langage objet à succès

6.2. Ambitions

Ne plus être amené à penser comme la machine

- Le fonctionnement des processeurs <=> séparation données/traitement

Penser la conception de logiciel de manière «humaine»

- «Décrire le problème dans les termes du problème» (**Bruce Eckel** – *Thinking In Java*)

- Le matériel n'impose pas de contraintes d'architecture

6.3. Les 5 principes en POO

Tout est objet

- Un objet encapsule des données et des opérations
- Un objet exécute une opération sur demande

Un programme est un ensemble d'objets se disant les uns aux autres quoi faire en s'envoyant des messages

- un message est une demande à un objet pour qu'il exécute une opération

Chaque objet dispose de son propre espace de mémoire composé d'autres objets

Chaque objet a un type

- une «classe» peut être associée à un «type»

Deux objets sont de même type quand ils peuvent recevoir les mêmes messages

- Héritage (Hors Programme NSI)
- Polymorphisme (Hors Programme NSI)

6.4. Les objets

Un objet est caractérisé par

- **Un état**
  - Décrit sous forme de valeurs d'attributs
  - Synonymes d'attribut : propriété, variable d'instance
- **Un comportement**
  - Décrit sous formes d'opérations
  - Une opération est déclenchée par l'envoi d'un message par un autre objet
- **Une identité**
  - L'identité est attribuée de manière implicite à la création d'un objet
  - Caractérise l'unicité d'un objet

6.5. Les classes

Démarche d'abstraction

Une classe contient la description commune d'un ensemble d'objets

- Liste des attributs ou propriétés
- Liste des opérations
- L'implémentation d'une opération est appelée méthode

Chaque objet est lié à une classe

- **Un objet est une instance d'une classe**

6.6. Illustration en Python

```python
# Classe voiture
class Voiture:
    def __init__(self):
        self.couleur = "bleue"
        self.masse = 630
        self.estDemarre = False

    def demarre(self):
        self.estDemarre = True

# Création de deux instances
k2000 = Voiture()
flashMacQueen = Voiture()

# Accès aux attributs
assert k2000.couleur == "bleue"

# Demande à flashMacQueen de démarrer
flashMacQueen.demarre()

assert flashMacQueen.estDemarre == True
assert k2000.estDemarre == False
```

```
# Change la couleur de FlashMacQueen
flashMacQueen.couleur = "rouge"
assert flashMacQueen.couleur == "rouge"
```

6.7. Focus sur «self» et «__init__»

La méthode «`__init__`» est une méthode responsable de l'initialisation d'un objet à sa construction

«`self`» est une référence sur l'objet en cours de construction

Dans l'exemple ci-dessous, chaque objet créé aura 3 attributs (`couleur`, `masse` et `estDemarre`) initialisés avec les valeurs fournies dans la méthode «`__init__`»

```
def __init__(self):
    self.couleur = "bleue"
    self.masse = 630
    self.estDemarre = False
```

6.8. Langages supportant le paradigme objet

- ~~*Assembleur*~~
- ~~*C,* **C++**~~
- ~~*Perl,* **PHP**~~
- *Python*
- *Java, C#*
- *JavaScript*
- *Kotlin, Swift, Scala*
- *et une quantité difficilement dénombrable d'autres langages*

6.9. Limites

Les langages objets les plus anciens sont bâtis sur une approche impérative

- Limites liées à l'impératif

L'explosion du développement d'applications concurrentes et distribuées a remis au goût du jour les langages fonctionnels

Les langages de dernière génération prennent le meilleur des deux mondes avec un premier langage mixant les 2 paradigmes avec succès : Scala

Aujourd'hui, il y a Kotlin, recommandé par Google pour les apps Android

Les langages objets restent généralistes

- Parfois non adaptés à des problèmes spécifiques

7. En résumé

Langage	Impératif	Procédural	Fonctionnel	Objet
Assembleur	X			
C	X	X		
C++	X	X		X
Perl	X	X		
PhP	X	X		X
Python	X	X	~	X
Java	X	X	~	X
C#	X	X	~	X
Javascript	X	X	X	X
Kotlin	X	X	X	X
Swift	X	X	X	X

Source : cours de Franck Silvestre - franck.silvestre@iut-rodez.fr