Déboguer un programme

La phase de débogage d'un programme, qui consiste à rechercher les erreurs de programmation, est très gourmande en temps. Ceci est particulièrement vrai avec Python dont le typage dynamique repousse la découverte des fautes au moment de l'exécution.

1. Le traceback de Python

Lorsque l'interpréteur Python rencontre un problème, une **exception** est levée. Si elle n'est pas capturée, cette exception provoque l'arrêt du programme et l'affichage d'un message appelé traceback. Ce message permet de connaître la nature et le contexte de l'incident.

```
>>> t = [1, 1, 2, 5, 14, 42, 1321]
>>> t[12]
Traceback (most recent call last):
  File "<stdins>", line 1, in <module>
IndexError: list index out of range
```

Le tableau suivant donne quelques exceptions courantes, observables en utilisant les structures de base de Python.

NameError	accès à une variable inexistante
IndexError	accès à un indice invalide d'un tableau
KeyError	accès à une clé inexistante d'un dictionnaire
ZeroDivisionError	division par zéro
TypeError	opération appliquée à des valeurs incompatibles
ValueError	Argument de valeur incompatible mais de bon type

2. Signaler un problème avec une exception

Il est possible de déclencher directement toutes ces exceptions (on dit « lever une exception ») avec l'opération **raise** de Python.

```
>>> raise IndexError('indice trop grand')
Taceback (most recent call last) :
  File "<stdins>", line 1, in <module>
IndexError : indice trop grand
```

Cette opération s'écrit en faisant suivre le mot-clé **raise** du nom de l'exception à lever, lui-même suivi entre parenthèses d'une chaîne de caractères donnant des informations sur l'erreur signalée.
Le message affiché lorsqu'une exception interrompt un programme et donne un aperçu de l'état de la pile d'appels au moment où l'exception à été levée.

Interfaces et exceptions

Les exceptions peuvent être utilisées en particulier dans les fonctions formant l'interface d'un module, pour signaler à un utilisateur du module toute utilisation incorrecte de ces fonctions. L'interface mentionnera dans ce cas quelles exceptions spécifiques sont levées et dans quelles conditions.

Rattraper une exception

Les exceptions levées par un programme peuvent avoir plusieurs causes. Certaines traduisent, des erreurs du programme : celles qui sont imprévues et leurs conséquences ne sont par définition pas maîtrisées. Dans ces conditions, interrompre l'exécution du programme est légitime. D'autres exceptions, en revanche, s'inscrivent dans le fonctionnement normal du programme : elles correspondent à des situations connues, exceptionnelles mais possibles.
Pour mettre en place ce comportement alternatif, il faut rattraper cette exception, c'est-à-dire l'intercepter avant que l'exécution du programme ne soit définitivement abandonnée, avec la construction suivante.

```
try :
    x = int (input ('Entrer un jour'))

except ValueError :
    print ('Entrer un entier valide')
```

Le mot-clé **try** suivi du symbole **:** (deux-points) introduit un premier bloc de code, puis le mot-clé **except** suivi du nom de l'exception et du symbole **:** précède un deuxième bloc de code. On qualifiera le premier bloc de normal et le second d'alternatif.

L'exécution d'une telle instruction procède comme suit. On exécute d'abord le bloc de code normal. Si l'exécution de ce bloc s'achève normalement, sans lever d'exception, alors le bloc alternatif est ignoré et on passe directement aux instructions suivantes. Si à l'inverse une exception est levée dans l'exécution du bloc normal, alors l'exécution de ce bloc est immédiatement interrompue et le nom de l'exception levée est comparé avec le nom précisé à la ligne **except**. Si les noms correspondent, l'exception est rattrapée et on exécute le bloc de code alternatif avant de passer à la suite. Sinon, l'exception est propagée et le programme s'interrompt.

Les requêtes SQL

Le but de ce cours est de décrire la structure d'une requête SQL afin de sélectionner et mettre à jour des données. La sélection va consister en l'écriture de requêtes SQL permettant de trouver toutes les données de la base vérifiant un certain critère.
Le premier rôle du programmeur de bases de données va donc être de traduire des questions que l'on se pose sur les données du langage naturel au langage SQL, afin que le SGBD puisse y répondre.

1. Sélection de données

Sélection de colonnes avec la clause SELECT

La commande **SELECT** permet de sélectionner des colonnes d'une ou plusieurs tables données en paramètre. Comme souvent, l'ensemble des colonnes peut être indiqué par une étoile *. La syntaxe est :

```
SELECT colonne(s) FROM nom_table(s)
```

Exemple :
Prenons pour exemple la table TABLE_NOTES dont le schéma relationnel et les enregistrements sont les suivants :

nom	maths	anglais	info
Joe	19	17	15
Alan	14	15	17
Zoe	18	16	20

Les requêtes SQL à base d'instruction SELECT suivantes renvoient les tables :

```
SELECT * FROM TABLES_NOTES ;
```

→ Renvoie →

nom	maths	anglais	info
Joe	19	17	15
Alan	14	15	17
Zoe	18	16	20

```
SELECT maths, nom FROM TABLE_NOTES ;
```

→ Renvoie →

maths	nom
19	Joe
14	Alan
18	Zoe

Sélection de lignes avec la clause WHERE

La commande **WHERE** filtre des enregistrements à partir d'un critère de sélection. Cette instruction vient en complément de l'instruction **SELECT**. La syntaxe est :

```
SELECT colonne(s) FROM nom_table(s) WHERE critère_de_sélection
```

Exemples :

```
SELECT * FROM TABLES_NOTES

WHERE maths>15 OR info>18 ;
```

→ Renvoie →

nom	maths	anglais	info
Joe	19	17	15
Zoe	18	16	20
Zoe	18	16	20

```
SELECT maths, nom FROM TABLES_NOTES
WHERE info>15 ;
```

→ Renvoie →

maths	nom
19	Zoe
18	Alan

Suppression des données redondantes

Le premier exemple du tableau précédent fait apparaître des valeurs de retour redondantes (nom : Zoe). La commande **DISTINCT** associée à **SELECT** permet de supprimer ces doublons.

```
SELECT DISTINCT nom FROM TABLES_NOTES

WHERE maths>15 OR info>18 ;
```

→ Renvoie →

nom
Joe
Zoe

Tri de donnée avec ORDER BY

Le tri par ordre alphabétique des données de retour sur une colonne de référence peut être effectué avec l'instruction **ORDER BY**. Le mot clé **ASC** stipule un tri croissant et **DESC** un tri décroissant.

```
SELECT * FROM TABLES_NOTES

ORDER BY anglais ASC ;
```

→ Renvoie →

nom	maths	anglais	info
Alan	14	15	17
Zoe	18	16	20
Joe	19	17	15

Les fonctions de groupes

Les fonctions de groupes permettent d'obtenir des informations sur un ensemble de lignes travaillant sur les colonnes et non pas sur les lignes comme avec **WHERE**.

- ✗ **AVG** : calcule la moyenne d'une colonnes
- ✗ **SUM** : calcule la somme d'une colonnes
- ✗ **MIN**, **MAX** : calculent le minimum et le maximum d'une colonnes
- ✗ **COUNT** : donne le nombre de lignes d'une colonnes

```
SELECT AVG(maths), MIN(maths),
MAX(maths) FROM TABLES_NOTES ;
```

→ Renvoie →

17	14	19

Renommage avec la clause AS

Les attributs des données renvoyées peuvent être renommées avec **AS** :

```
SELECT nom AS name FROM
TABLES_NOTES ;
```

→ Renvoie →

name
Alan
Zoe
Joe

Les jointures avec `JOIN`

Une jointure permet d'associer dans une même requête plusieurs tables en les liant à partir d'un attribut commun. Par exemple, supposons que nous ayons une table contenant des mentions en fonctions de la note obtenue, sur le principe suivant :

mention	note
Bien	14
Presque parfait	19
Inespéré	18

Extrait de la table Table_mentions

Associer directement une mention à chaque élève en fonction de son résultat en mathématiques s'écrit :

```
SELECT Table_notes.nom,
Table_mentions.mention FROM Table_notes JOIN
Table_mentions ON
Table_notes.maths = Table_mentions.note ;
```

→ Renvoie →

nom	mention
Alan	Bien
Zoe	Inespéré
Joe	Presque parfait

2.Modification d'une table

Mise à jour d'une table avec `INSERT INTO`

La mise à jour d'une table avec un nouvel enregistrement s'effectue en entrant le tuple de nouvelles valeur dans la table avec la commande **`INSERT INTO`** :

```
INSERT INTO table (attr1, attr2, ...) VALUES (val1, val2 …)
```

Par exemple, l'ajout d'une nouvelle appréciation s'écrit :

```
INSERT INTO Table_mentions (note, mention) VALUES (20 , 'Au top') ;
```

Modification d'un enregistrement avec `UPDATE`

La modification de valeurs d'un enregistrement existant s'effectue avec la commande UPDATE :

```
UPDATE table SET (attr1 = val1, attr2 = val2…)
```

Par exemple la requête suivante change la note de mathématiques de Joe avec 15 :

```
UPDATE Table_notes SET maths=15 WHERE Nom='Joe' ;
```

Supprimer un enregistrement

La suppression d'une ligne s'effectue avec la commande DELETE :

```
DELETE FROM table WHERE condition
```

La requête suivante supprime tous les enregistrements qui ont une note de maths ≤ 14

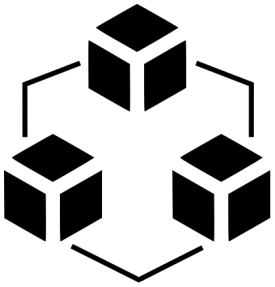
```sql
DELETE FROM Table_notes WHERE maths <= 14 ;
```

Types de données en SQL

Les domaines abstraits du modèle relationnel correspondent à des types de données en langage SQL. Le tableau suivant récapitule les principaux types utilisés :

Nom du type	Description
Numériques	
SAMLLINT	Entier 16 bits signé
INTEGER	Entier 32 bits signé
BIGINT	Entier 64 bits signé
Decimal(t, f)	Décimal signé de t chiffres dont f après la virgule
REAL	Flottant 32 bits (valeur approchée)
DOUBLE PRECISION	Flottant 64 bits (valeur approchée)
Chaîne de caractères	
CHAR(*n*)	Chaîne de n caractères (espaces dans caractères manquants)
VARCHAR(*n*)	Chaîne d'au plus n caractères
TEXT	Chaîne de taille quelconque
Date	
DATE	Date au format 'AAAA-MM-JJ'
TIME	Heure au format 'hh:mm,ss'
DATETIME	Instant au format 'AAAA-MM-JJ hh:mm:ss'

La programmation orientée objet (Poo)



La méthode classique de résolution des problème informatique passe généralement par l'analyse descendante qui consiste à décomposer un problème en sous problème jusqu'à descendre à des actions primitives.
On décompose ainsi un programme en un ensemble de sous-programmes appelés procédures qui coopèrent pour la résolution d'un problème. Cette méthode de résolution appelée programmation impérative, comporte de nombreux inconvénients. En effet les procédures et fonctions sont généralement des outils qui produisent et/ou modifient des données.
L'évolution d'une application développée suivant ce modèle n'est pas évidente car la moindre modification des structures de données d'un programme conduit à la révision de toutes les procédures manipulant ces données. De plus pour de très grosses applications, le développement peut être très long.

La POO résulte de la prise de conscience des problèmes posés par l'industrie du logiciel ces dernières années en termes de complexité accrue et de stabilité dégradée des logiciels développés. En effet la faiblesse de la programmation impérative tient essentiellement du fait que ce mode de programmation manipule une grande quantité de variables qu'il devient difficile d'identifier lorsque le programme devient très important.

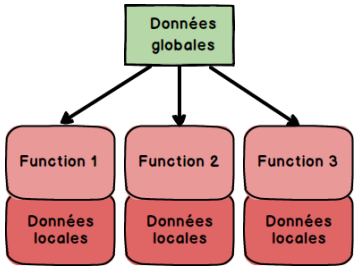


Figure 1: Programmation impérative

La Poo s'inspire du monde réel en faisant interagir des objets plus ou moins complexes à l'aide de messages. Ce paradigme de programmation laisse de côté la notion de variable au profit d'objet ayant chacun ses propres caractéristiques et dialoguant entre eux à l'aide de messages.

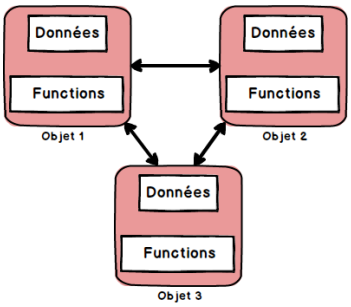


Figure 2: Programmation objet

Repères historiques
La programmation orientée objet a fait ses débuts dans les années 1960 avec le langage Lisp. Cependant elle a été été formellement définie avec le langage Simula (vers 1970) puis SmallTalk. Elle est maintenant incontournable dans les langages récents tels que Java ou Python.

1. Des objets communicants

Dans la programmation orientée objet, les différents **objets** utilisés peuvent être construits indépendamment les uns des autres (par exemple par des programmeurs différents) sans qu'il n'y ait de risque d'interférence.
Ce résultat est obtenu grâce au concept d'**encapsulation** : la fonctionnalité interne de l'objet et les variables qu'il utilise pour effectuer son travail, sont en quelque sorte enfermés dans l'objet.
Les autres objets et d'une façon général, le mode extérieur ne peuvent y accéder qu'à travers des procédures bien définies, c'est ce que l'on appelle l'**interface** de l'objet. Ces objets sont comme un iceberg qui ne laisse apparaître que son sommet au dessus de l'eau - l'interface - mais qui cache sous l'eau une partie importante et inaccessible - le traitement des données – (voir fig 3).

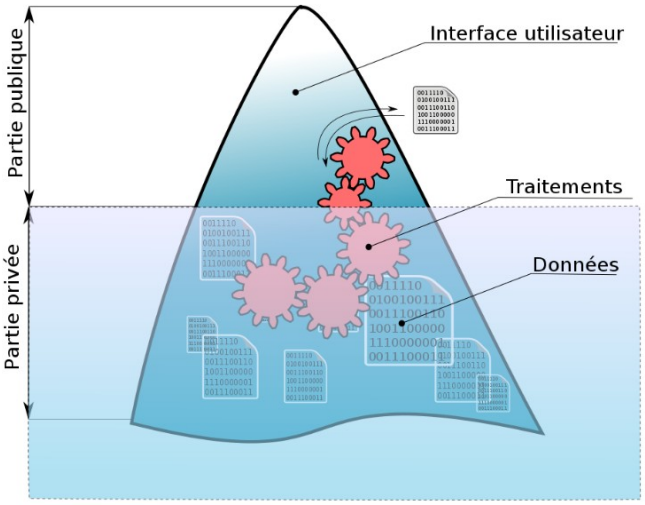


Figure 3: Métaphore de l'iceberg

Il ne faut pas opposer la programmation impérative à l'orienté objet car, au final toute programmation des traitements reste tributaire des mécanismes de programmation procédurale et structurée. On y rencontre des variables, des arguments, des boucles, des arguments de fonction, des instructions conditionnelles, tout ce que l'on trouve classiquement dans les boîtes à outils impératives.

2.Les classes

Comme dans la vie réelle, les objets d'une même famille ont des caractéristiques similaires. Par exemple toutes les voitures ont 4 roues, des portières, une peinture de finition... Dans la programmation objet, une classe est la description générale d'une famille d'objet. La classe est comme un moule à objets, elle définit les données communes (**attributs**) et les actions (**méthodes**) que les objets de la classe pourront réaliser.

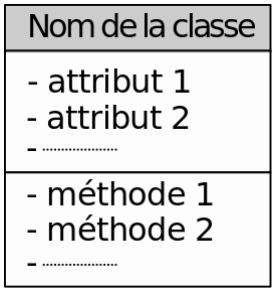


Figure 4: Représentation d'une classe

Un objet est donc un paramétrage particulier d'une classe, ainsi tous les objets d'une même classe possèdent les mêmes méthodes et attributs. Par contre les valeurs des attributs changeront d'un objet à un autre.
La représentation d'une classe peut s'effectuer en suivant le modèle de la figure 4. Ce diagramme est appelé diagramme de classe.

3.Création d'une classe

L'encapsulation désigne le principe de regrouper des données brutes dans un objet avec un ensemble de méthodes permettant de les lire ou de les manipuler. Le but étant de :

- ➔ masquer leur complexité
- ➔ Proposer une interface simple d'utilisation
- ➔ Permettre de les modifier indépendamment du reste du programme

Les attributs et les méthodes des classes peuvent être déclarés en accès :

- ➔ **Public** (**+**) : les autres objets peuvent y accéder et les modifier dans le cas des attributs.
- ➔ **Privé** (**-**) : Les autres objets n'ont pas le droit d'y accéder directement.

Pour maintenir les caractéristiques d'encapsulation, il est d'usage de déclarer les différents attributs en accès privé, ceci afin de contrôler les modification des variables de la classe. Pour accéder aux valeurs des attributs de la classe, il faut alors proposer dans l'interface des méthodes renvoyant les valeurs lorsqu'on les invoque (**accesseurs** / **getters**) et d'autres pouvant les modifier (**mutateurs** / **setters**).
Chaque classe doit définir une méthode particulière appelée **constructeur**. Ce constructeur est une méthode invoquée lors de la création d'un objet, elle permet d'initialiser l'objet.

Prenons pour exemple la numérisation d'un avare (personne proche de son argent), dans sa définition qui est présentée figure 5 nous pouvons dire que :

- ➔ la classe possède deux attributs privés (nom et fortune)
- ➔ Le constructeur permet d'initialiser les attributs de la classe
- ➔ La classe possède un accesseur à l'attribut fortune (`compter()`)

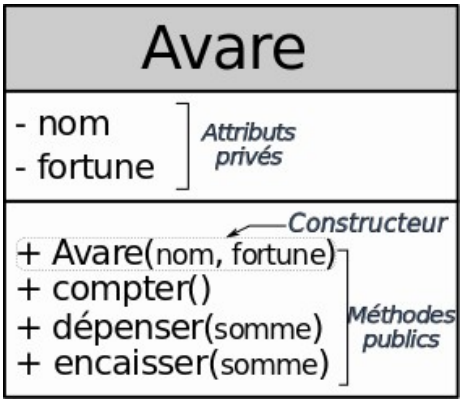


Figure 5: Classe "Avare"

4.Manipulation d'un objet

Une classe est un concept muni de caractéristiques et de propriétés. Pour bénéficier de ces caractéristiques/propriétés, il faut invoquer une instance de cette classe afin de créer un objet.

Création d'un objet

La création objet s'effectue en invoquant le constructeur de la classe. Dans l'exemple de la classe Avare précédente, la création de deux objets « Picsou » et « Gates » de fortune respective 1 000 000$ et 100 000 000 000$ s'effectue de cette façon :

```
>>> balthazar = Avare("Picsou",1000000)
>>> bill = Avare("Gates",100000000000)
>>> bill.compter()
100000000000
>>> balthazar.compter()
1000000
```

L'évolution de la fortune de ces objets s'effectue en appelant les méthodes correspondantes :

```
>>> balthazar.encaisser(100000)
>>> balthazar.compter()
1100000
>>> bill.encaisser(100000)
>>> bill.compter()
100000100000
```

5.Implémentation d'une classe sous Python

On déclare la classe à l'aide du mot clé **`class`** :

```
1  class Avare :
2      ''' Classe Avare :
3      Caractérisé par sa fortune et son nom
4      '''
```

Les lignes 2 à 5 permettent de documenter la classe à l'aide de docstring. Cette documentation est obtenue avec la commande **`__doc__`**

Le constructeur

Le constructeur permet d'initialiser l'objet. Ce constructeur est définit avec **`__init__`**

```
6      def __init__(self, name, fortune) :
7          self.__nom = name
8          self.__fortune = fortune
```

La variable **`self`**, dans les méthodes d'un objet désigne l'objet lui même auquel s'appliquera la méthodes.

Accès public/privé

La limitation d'une méthode ou d'un attribut à un accès privé est effectué en ajoutant un double underscore devant l'attribut/méthode. Par exemple l'attribut `self.__nom` est un attribut privé, pour le rendre public il faudrait écrire `self.nom`

Méthodes particulières

Le langage Python possède un certain nombre de méthodes particulières, chacune avec un nom fixé et entouré d'un double underscore comme pour le constructeur **__init__**. Ces méthodes sont appelée par certaines opérations de prédéfinies de Python.

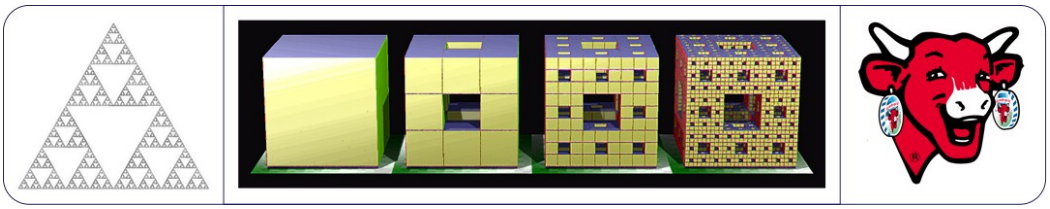
Méthode	appel	effet
`__str__(self)`	`str(obj)`	Renvoie une chaîne de caractère décrivant l'objet
`__rep__(self)`	Invocation de l'objet dans la console	Renvoie une chaîne de caractère décrivant l'objet

L'implémentation de la classe Avare est donc :

```
1  class Avare :
2      ''' Classe Avare :
3      Caractérisé par sa fortune et son nom
4      '''
5      def __init__(self, name, fortune) :
6          self.__nom = name
7          self.__fortune = fortune
8
9      def __repr__(self) :
10         return 'Fortune de ' + self.__nom + ' : ' + str(self.__fortune) +' $'
11
12     def compter(self) :
13         return self.__fortune
14
15     def depenser(self,somme) :
16         self.__fortune -= 0.9*somme # Un avar ne depense jamais trop....
17
18     def encaisser(self,somme) :
19         self.__fortune += somme
```

Fonctions récursives

On dit qu'un programme est récursif si pour parvenir au résultat voulu il se réemploie lui-même. Cela peut s'illustrer par les images suivantes :



1. Principe de base :

Une fonction récursive simple s'écrit sous la forme :

```
def fonctionRecursive(args)
        if condition arret:
                return valeur
        fonctionRecursive(argument)
```

Pour écrire une fonction récursive, il faut :

- ➔ déterminer le type de données à renvoyer
- ➔ écrire la condition d'arrêt (appelé aussi cas de base) pour interdire une récursivité infinie
- ➔ écrire l'appel récursif

2.Intérêt de la récursivité

La récursivité fournit des algorithmes concis et élégants. Il s'agit à la fois d'un style de programmation mais également d'une technique pour définir des concepts et résoudre certains problèmes qu'il n'est parfois pas facile de traiter en programmant uniquement avec des boucles.
Il faut cependant ce méfier de la complexité, de plus il faut bien s'assurer de définir un cas d'arrêt pour interdire une récursivité infinie

3.Gestion de la mémoire lors d'un appel récursif

Pour chaque appel de fonction, les données (arguments de la fonction, place pour la valeur de retour, adresse de retour, les variables locales) sont stockés dans un espace réservé de la mémoire appelé la « pile ». Cette pile est une structure de données linéaire ayant pour maxime « dernier entré, premier sorti » (last in, first out – LIFO)

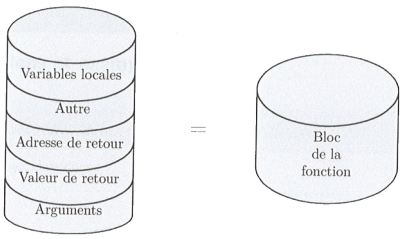
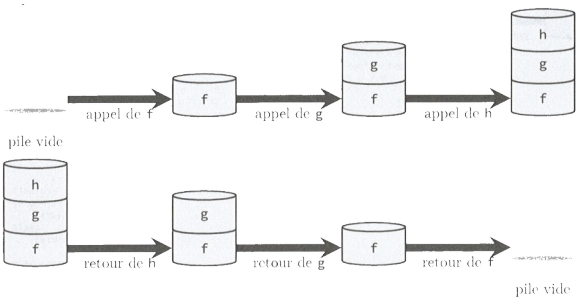


Figure 1: Pile LIFO

Gestion des appels de fonctions

Considérons l'appel des fonctions suivantes :

```
def h(x):
        return x+1
def g(x):
        return h(x)+2
def f(x):
        return g(x)+1
```



L'appel de f(5) entraine le mécanisme suivant dans la pile

Pile d'appel sous Python

Lors de l'appel d'une fonction récursive une pile est utilisée. En Python, la taille de la pile est limitée à 1000 par défaut. En cas de dépassement le message d'erreur suivant s'affiche :

```
RuntimError : maximum recursion depth exceeded
```

Types de récursivité

Il existe plusieurs types de récursivité :

- ➔ **récursivité simple** : un algorithme récursif est simple ou linéaire si chaque cas se résout en au plus un appel récursif. Le calcul de la factorielle en est un exemple.
- ➔ **récursivité multiple** : un algorithme récursif est multiple si l'un des cas qu'il distingue se résout avec plusieurs appels récursifs. Si "plusieurs=2" on parle de récursivité binaire.
- ➔ **récursivité terminale** : un algorithme est récursif terminal si l'appel récursif est la dernière instruction de la fonction. La valeur retournée est directement obtenue par un appel récursif.