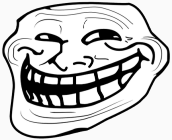
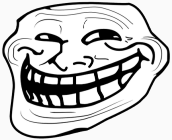
Récursivité

Chapitre 01

NSI2

6 septembre 2021





Pour comprendre la récursivité, il faut d'abord comprendre la récursivité.

Le formidable principe de la récursivité (la forme)

S’appeler et se terminer

On dit qu’une fonction est récursive lorsque dans sa définition on appelle (une ou plusieurs fois) cette même fonction.

S’appeler et se terminer

On dit qu’une fonction est récursive lorsque dans sa définition on appelle (une ou plusieurs fois) cette même fonction.

Cela pose un problème : si la fonction ne cesse de s’appeler, comment cela peut-il se terminer ?

Exemple 1

La fonction suivante est incorrecte :

```python
def f():
    return f()
```

Exemple 1

La fonction suivante est incorrecte :

```
def f():
    return f()
```

l'appel `f()` provoque *a priori* une boucle infinie.

Exemple 2

```python
def f(n : int):
    return f(n-1)
```

Exemple 2

```python
def f(n : int):
    return f(n-1)
```

Quelle que soit la valeur de `x`, l'appel `f(x)` provoque également *a priori* une boucle infinie :

Exemple 2

```
def f(n : int):
    return f(n-1)
```

Quelle que soit la valeur de `x`, l'appel `f(x)` provoque également *a priori* une boucle infinie :
`f(10)` appelle `f(9)` qui appelle `f(8)` *et cætera*.

Nécessité d’une condition d’arrêt

```python
def f(n : int) -> int:
    if n <= 0 :
        return 0
    else:
        return f(n - 1)
```

Nécessité d’une condition d’arrêt

```python
def f(n : int) -> int:
    if n <= 0 :
        return 0
    else:
        return f(n - 1)
```

Cette fonction n’est pas très utile…

Nécessité d'une condition d'arrêt

```python
def f(n : int) -> int:
    if n <= 0 :
        return 0
    else:
        return f(n - 1)
```

Dans le corps de cette fonction, on distingue 2 cas :

- le cas d'arrêt, dans lequel `n <= 0` et la fonction renvoie zéro;
- le cas récursif dans lequel la fonction s'appelle elle-même.

Nécessité d'une condition d'arrêt

```python
def f(n : int) -> int:
    if n <= 0 :
        return 0
    else:
        return f(n - 1)
```

Quel que soit l'entier `n` de départ (qu'on suppose grand), lorsqu'on évalue `f(n)` alors celle-ci évalue `f(n-1)` qui elle-même évalue `f(n-2)` *et cætera.*

Puisque les nombres `n-1` , `n-2` , `n-3` , . . . diminuent strictement, les appels récursifs vont finir par laisser place au cas d'arrêt de sorte que `f(n)` est bien défini.

Syntactic Sugar

« sucre syntaxique » d'un langage de programmation : ensemble des règles de syntaxe qui ont été ajoutées pour rendre le code plus facile à lire et à écrire.

Syntactic Sugar

« sucre syntaxique » d'un langage de programmation : ensemble des règles de syntaxe qui ont été ajoutées pour rendre le code plus facile à lire et à écrire.

```python
def f(n : int) -> int:
    return 0 if n <= 0 else f(n - 1)
```

Un premier exemple digne d'intérêt

Soit $n \in \mathbb{N}$. *Factorielle n* est le produit de tous les entiers non-nuls inférieurs ou égaux à n, et se note $n!$.

Un premier exemple digne d'intérêt

Soit $n \in \mathbb{N}$. *Factorielle n* est le produit de tous les entiers non-nuls inférieurs ou égaux à n, et se note $n!$.

- $1! = 1$;

Un premier exemple digne d'intérêt

Soit $n \in \mathbb{N}$. *Factorielle n* est le produit de tous les entiers non-nuls inférieurs ou égaux à n, et se note $n!$.

- $1! = 1$;
- $2! = 1 \times 2 = 2$;

Un premier exemple digne d'intérêt

Soit $n \in \mathbb{N}$. *Factorielle n* est le produit de tous les entiers non-nuls inférieurs ou égaux à n, et se note $n!$.

- $1! = 1$;
- $2! = 1 \times 2 = 2$;
- $10! = 1 \times 2 \times \ldots \times 10 = 3\,628\,800$;

Un premier exemple digne d'intérêt

Soit $n \in \mathbf{N}$. *Factorielle n* est le produit de tous les entiers non-nuls inférieurs ou égaux à n, et se note $n!$.

- $1! = 1$;
- $2! = 1 \times 2 = 2$;
- $10! = 1 \times 2 \times \ldots \times 10 = 3\,628\,800$;
- $0! = 1$ par convention.

À l'aide d'une boucle for

```python
def factorielle( n : int) -> int:
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result
```

À l'aide d'une boucle for

```python
def factorielle( n : int) -> int:
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result
```

L'algorithme est dit itératif car il utilise une boucle.

De manière récursive

```
fonction factorielle(n : entier naturel) -> entier naturel
    si n = 0 alors
        renvoyer 1
    sinon
        renvoyer n * factorielle(n -1)
    fin si
```

De manière récursive

```
fonction factorielle(n : entier naturel) -> entier naturel
    si n = 0 alors
        renvoyer 1
    sinon
        renvoyer n * factorielle(n -1)
    fin si
```

D'une certaine manière, c'est plus « élégant ».

Exercice

1. Programmer la fonction `factorielle` en PYTHON de manière récursive, et tester cette fonction en vérifiant que `factorielle(10)` renvoie bien 3 628 800.
2. Mettre du *Syntactic Sugar* là-dedans !

Formidable, mais pourquoi (le fond) ?

Principe

- On veut écrire une fonction `f` qui résout un problème dépendant d'un entier naturel `n` ;

Principe

- On veut écrire une fonction `f` qui résout un problème dépendant d'un entier naturel `n` ;
- on examine le cas ou `n` vaut 0 (ou 1), correspondant à un problème très simple, que l'on sait résoudre ;

Principe

- On veut écrire une fonction `f` qui résout un problème dépendant d'un entier naturel `n` ;
- on examine le cas ou `n` vaut 0 (ou 1), correspondant à un problème très simple, que l'on sait résoudre ;
- on suppose que l'on sait résoudre le problème pour un entier `n-1`, à l'aide de la fonction `f`, on regarde alors les opérations à effectuer pour passer de ce problème au problème de taille `n` ;

Principe

- On veut écrire une fonction `f` qui résout un problème dépendant d'un entier naturel `n` ;
- on examine le cas ou `n` vaut 0 (ou 1), correspondant à un problème très simple, que l'on sait résoudre ;
- on suppose que l'on sait résoudre le problème pour un entier `n-1`, à l'aide de la fonction `f`, on regarde alors les opérations à effectuer pour passer de ce problème au problème de taille `n` ;
- on programme alors la fonction `f` de manière récursive.

L’exemple des poignées de main

Nous l’avons traité en activité préparatoire :

```python
def f(n : int) -> int
    return 0 if n == 0 else n - 1 + f(n - 1)
```

Un exemple de récursion double

La célèbre suite de Fibonacci, notée F, est définie ainsi :

Un exemple de récursion double

La célèbre suite de Fibonacci, notée F, est définie ainsi :

- Ses deux premiers termes F_0 et F_1 valent 1;

Un exemple de récursion double

La célèbre suite de Fibonacci, notée F, est définie ainsi :

- Ses deux premiers termes F_0 et F_1 valent 1;
- On construit chaque terme suivant en faisant la somme des deux précédents.

Un exemple de récursion double

De proche en proche, on calcule :

Un exemple de récursion double

De proche en proche, on calcule :

- $F_2 = 1 + 1 = 2$

Un exemple de récursion double

De proche en proche, on calcule :

- $F_2 = 1 + 1 = 2$
- $F_3 = 2 + 1 = 3$

Un exemple de récursion double

De proche en proche, on calcule :

- $F_2 = 1 + 1 = 2$
- $F_3 = 2 + 1 = 3$
- $F_4 = 3 + 2 = 5$

Un exemple de récursion double

De proche en proche, on calcule :

- $F_2 = 1 + 1 = 2$
- $F_3 = 2 + 1 = 3$
- $F_4 = 3 + 2 = 5$
- *et cætera*

Un exemple de récursion double

De proche en proche, on calcule :

- $F_2 = 1 + 1 = 2$
- $F_3 = 2 + 1 = 3$
- $F_4 = 3 + 2 = 5$
- *et cætera*

Et plus généralement

Un exemple de récursion double

De proche en proche, on calcule :

- $F_2 = 1 + 1 = 2$
- $F_3 = 2 + 1 = 3$
- $F_4 = 3 + 2 = 5$
- *et cætera*

Et plus généralement

$$F_n = \begin{cases} 1 & \text{si } n = 0 \text{ ou } n = 1 \\ F_{n-1} + F_{n-2} & \text{sinon} \end{cases}$$

Algorithme

```
fonction fibonacci(n : entier naturel) -> entier naturel
    si n < 2 alors
        renvoyer 1
    sinon
        renvoyer fibonacci(n - 1) + fibonacci(n - 2)
    fin si
```

Pour s’entraîner

Exercice

Programmer la fonction `fibonacci` en PYTHON et vérifier que `fibonacci(30)` vaut 1 346 269.

Remarques

– La récursivité est un des concepts *fondamentaux* de l'informatique.

Remarques

– La récursivité est un des concepts *fondamentaux* de l'informatique.
  On dit que c'est un paradigme de programmation.

Remarques

- La récursivité est un des concepts *fondamentaux* de l'informatique.
  On dit que c'est un paradigme de programmation.
- Certains algorithmes se programment naturellement de manière récursive.

Remarques

- La récursivité est un des concepts *fondamentaux* de l'informatique.
  On dit que c'est un paradigme de programmation.
- Certains algorithmes se programment naturellement de manière récursive.
- Certaines structures de données se définissent également de manière récursive.

La « magie » a un coût

La pile d’appels

Que se passe-t-il réellement en machine lorsqu’on évalue la fonction récursive `factorielle(3)` ?

La pile d'appels

Que se passe-t-il réellement en machine lorsqu'on évalue la fonction récursive `factorielle(3)` ?

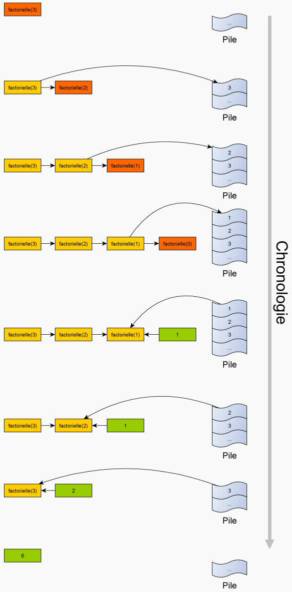
PYTHON possède une pile : c'est une structure de données simple, qui permet d'empiler et de dépiler des données un peu comme on empile des assiettes.

La pile d'appels

Que se passe-t-il réellement en machine lorsqu'on évalue la fonction récursive `factorielle(3)` ?

PYTHON possède une pile : c'est une structure de données simple, qui permet d'empiler et de dépiler des données un peu comme on empile des assiettes.

Lors de chaque appel récursif, une valeur est empilée en attendant le résultat de l'appel.


factorielle(3)
...
Pile
factorielle(3)
factorielle(2)
3
...
Pile
factorielle(3)
factorielle(2)
factorielle(1)
2
3
...
Pile
factorielle(3)
factorielle(2)
factorielle(1)
factorielle(0)
1
2
3
...
Pile
factorielle(3)
factorielle(2)
factorielle(1)
1
1
2
3
...
Pile
factorielle(3)
factorielle(2)
1
2
3
...
Pile
factorielle(3)
2
3
...
Pile
6
...
Pile
Chronologie

Taille de la pile

Lorsqu'il y a beaucoup d'appels récursifs *imbriqués*, la taille de la pile augmente.

Taille de la pile

Lorsqu'il y a beaucoup d'appels récursifs *imbriqués*, la taille de la pile augmente.
Pour éviter qu'elle sature, PYTHON fixe la limite des appels récursifs *imbriqués* à 999. Dès que l'on dépasse cette limite on obtient un message d'erreur :

Taille de la pile

Lorsqu'il y a beaucoup d'appels récursifs *imbriqués*, la taille de la pile augmente.
Pour éviter qu'elle sature, PYTHON fixe la limite des appels récursifs *imbriqués* à 999. Dès que l'on dépasse cette limite on obtient un message d'erreur :

```
RecursionError: maximum recursion depth exceeded in comparison
```

Fixer la taille de la pile

```
import sys

sys.setrecursionlimit(10_000)
```

Pour aller jusqu'à 10 000 appels récursifs au maximum (par exemple).

Pour s'entraîner

Exercice

Reprendre la fonction récursive `fibonacci` et utiliser le site Python Tutor pour visualiser la pile d'appels lors de l'évaluation de `fibonacci(5)`.

Quel commentaire peut-on faire ?