Algorithmique et Programmation 1

Lundi 7 octobre 2024

L1 Mathématiques - L1 Informatique
Semestre 1

CC0 : Comment et quoi ?

Horaire

- **Jeudi 10 octobre 10h45-11h15** (tiers-temps : 11h25) en A1
- Les retardataires ne seront pas admis
- Correction de 11h45 à 12h45 (en A1)

CC0 : Comment et quoi ?

Horaire

- **Jeudi 10 octobre 10h45-11h15** (tiers-temps : 11h25) en A1
- Les retardataires ne seront pas admis
- Correction de 11h45 à 12h45 (en A1)

Contenu

- Tout ce qu'on a vu, y compris la séance d'aujourd'hui
- Les slides et les notes de cours
- Pas de pseudo-code
- QCM

CC0 : Comment et quoi ?

Horaire

- **Jeudi 10 octobre 10h45-11h15** (tiers-temps : 11h25) en A1
- Les retardataires ne seront pas admis
- Correction de 11h45 à 12h45 (en A1)

Contenu

- Tout ce qu'on a vu, y compris la séance d'aujourd'hui
- Les slides et les notes de cours
- Pas de pseudo-code
- QCM

Sur la triche

- L'examen est **individuel**
- Tolérance zéro aux CC 1 et 2 (et 0.000000000001 au CC0)

Approfondissements : les fonctions

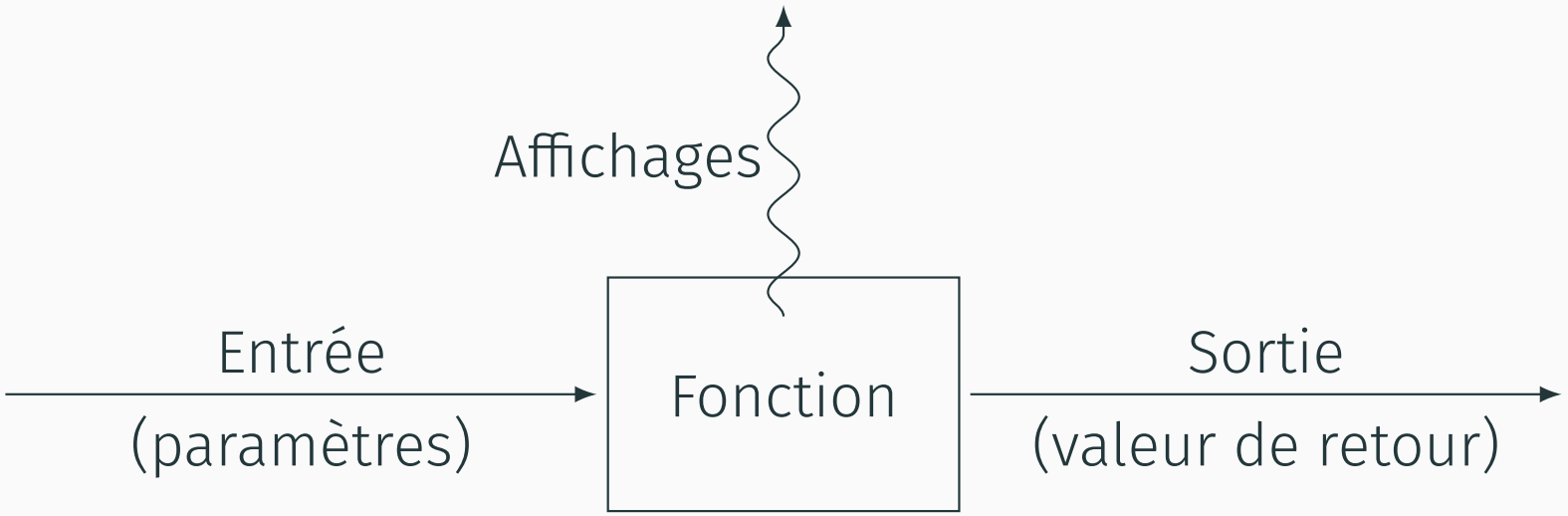
Fonction

En informatique, une fonction est :

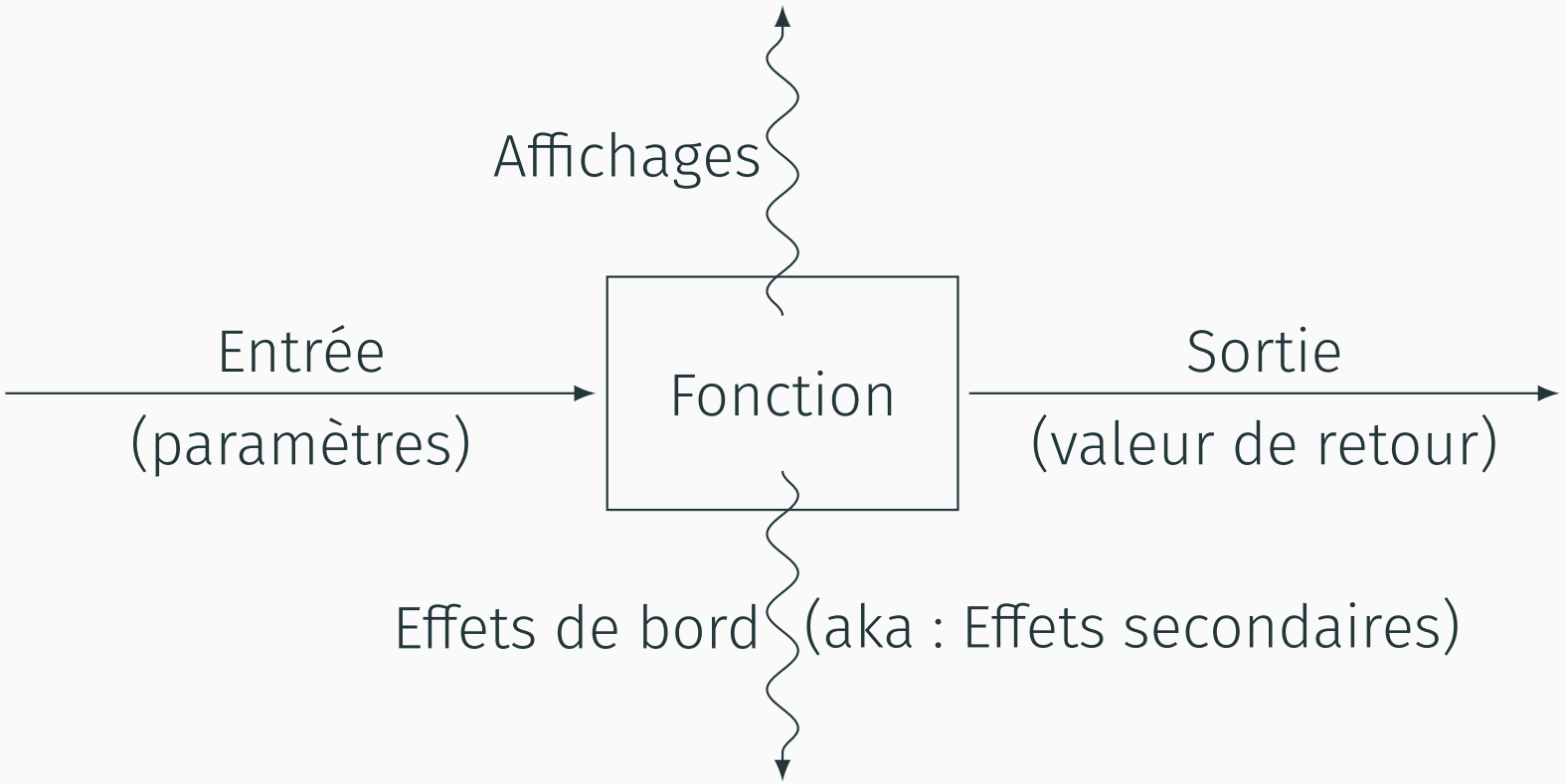
- Un morceau de programme
- Portant en général *un nom*
- Prenant un ou plusieurs *paramètres* (ou zéro)
- Renvoyant un résultat (la plupart du temps)

Anatomie d’une fonction

Anatomie d'une fonction



Anatomie d'une fonction



Exemples

Fonction à paramètres et résultats

Calculer le *minimum* de deux entiers (dans Thonny) :

Fonction à paramètres et résultats

Calculer le *minimum* de deux entiers (dans Thonny) :

```
def minimum(a,b):
    if a < b:
        return a
    else:
        return b
```

Fonction à paramètres et résultats

Calculer le *minimum* d'une liste non-vide (dans Thonny) :

Fonction à paramètres et résultats

Calculer le *minimum* d'une liste non-vide (dans Thonny) :

```
def minimum(lst):
    m = lst[0] # /!\ Erreur si lst est vide
    for elt in lst:
        if elt < m:
        m = elt
    return m
```

Fonction sans valeur de retour

Dessiner un carré fait du caractère `caractere` (dans Thonny) :

```
*****
*****
*****
*****
*****
```

Fonction sans valeur de retour

Dessiner un carré fait du caractère `caractere` (dans Thonny) :

```
*****
*****
*****
*****
*****
```

```
def dessine_carre(n, caractere):
    i = 0
    while i < n:
        j = 0
        while j < n:
            print(caractere, end = '')
            j += 1
        print('\n', end = '')
        i += 1
    return
```

Fonction sans valeur de retour

Dessiner un carré fait du caractère `caractere` (dans Thonny) :

```
*****
*****
*****
*****
*****
```

```
def dessine_carre(n, caractere):
    for i in range(0, n):
        for j in range(0, n):
            print(caractere, end = '')
        print('\n', end = '')
    return
```

Composition de fonctions

En maths

$$\forall x \in X, g \circ f(x) = g(f(x))$$

En informatique

On peut appeler une fonction dans une fonction !
(et ainsi de suite)

Dessiner un carré : variante

```
*****
*****
*****
*****
*****
```

Carré de côté **n** = **n** lignes de longueur **n**

On peut donc décomposer ainsi :

-
```
def dessine_ligne(n,car):
    """Dessine *une ligne* de longueur n
       composée du caractère car."""
```
-
```
def dessine_carre(n,car):
    """Dessine *un carré* de longueur n
       composé du caractère car."""
```

➔ Ce que ça donne dans Thonny

Dessiner un carré : variante

```
def dessine_ligne(n, caractere):
 """dessine une ligne de longueur n
    composée du caractère car"""
    for j in range(0, n):
        print(caractere, end = '')
    print('\n', end = '')
    return
```

Dessiner un carré : variante

```
def dessine_carre(n, caractere):
 """dessine un carré de longueur n
    composé du caractère car"""
    for i in range(0, n):
        dessine_ligne(n,caractere)
    return
```

Erreur fréquente : confusion paramètre / saisie retour / affichage

Paramètre / saisie

```
def minimum(a,b):
    a = int(input()) # NON !
    b = int(input()) # NON !
    if a <= b:
        return a
    else:
        return b
```

Erreur fréquente : confusion paramètre / saisie retour / affichage

Paramètre / saisie

```
def minimum(a,b):
    a = int(input()) # NON !
    b = int(input()) # NON !
    if a <= b:
        return a
    else:
        return b
```

Retour / affichage

```
def maximum(a,b):
    if a >= b:
        print(a) # NON !
    else:
        print(b) # NON !
```

Erreur fréquente : confusion paramètre / saisie retour / affichage

Paramètre / saisie

```
def minimum(a,b):
    a = int(input()) # NON !
    b = int(input()) # NON !
    if a <= b:
        return a
    else:
        return b
```

Dans Thonny

Retour / affichage

```
def maximum(a,b):
    if a >= b:
        print(a) # NON !
    else:
        print(b) # NON !
```

Fonctions et espaces de nom

- Paramètres et variables d'une fonction : indépendantes des autres variables du programme
- N'existent plus une fois l'exécution de la fonction terminée
- On les appelle des variables *locales*

$\Rightarrow$ On peut renommer les variables d'une fonction.

$\rightarrow$ Démonstration sur Thonny (`minimum.py`).

(Contre-)exemple : intervertir des variables

Dans Thonny : `echange.py`

(Contre-)exemple : intervertir des variables

Dans Thonny : `echange.py`

- changer les valeurs de `a` et `b` dans la fonction *n'a pas d'effet* sur `x` et `y` dans le programme principal !
- la variable `temp` n'existe plus après l'exécution de la fonction

Sous le capot : sémantique d'un appel

Espace de noms

Ensemble de noms (variables, fonctions) défini à un certain point d'un programme

➔ lors d'un appel de fonction, création d'un espace de nom *local* :

- Paramètres $\rightarrow$ leur valeur lors de l'appel
- Variables locales

Exemple

`minimum_liste.py` dans Python Tutor et Thonny.

Espaces de noms lors de l'exécution

À un point du programme, l'ensemble des noms (variables, fonctions) connus est constitué de plusieurs espaces imbriqués (du plus ancien au plus récent) :

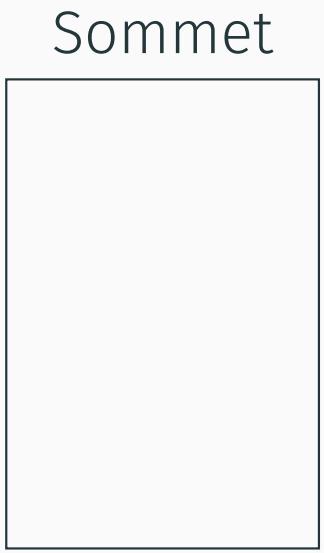
- espace de noms prédéfini (`print`, `input`, …)
- espace de noms global : définis dans le programme principal
- espaces de noms locaux imbriqués dans l'*ordre chronologique* des appels
  - paramètres de l'appel
  - variables locales à la fonction

Pile d'appels

L'empilement des espaces de noms obéit à une politique de *pile*

- sommet : appel en cours
- en-dessous : appels précédents
- (presque) tout en bas : espace de nom global

```
def f(x):
    return g(x) + 1
def g(x):
    return h(x) + 2
def h(x):
    return x + 3
```

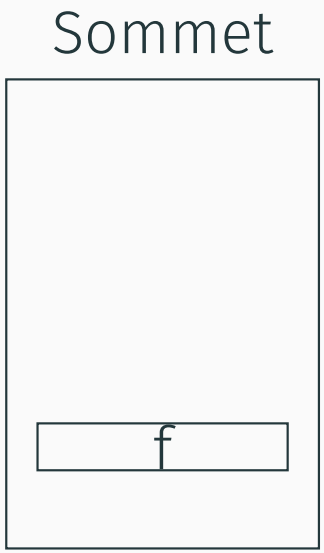


Pile d'appels

L'empilement des espaces de noms obéit à une politique de *pile*

- sommet : appel en cours
- en-dessous : appels précédents
- (presque) tout en bas : espace de nom global

```
def f(x):
    return g(x) + 1
def g(x):
    return h(x) + 2
def h(x):
    return x + 3
```

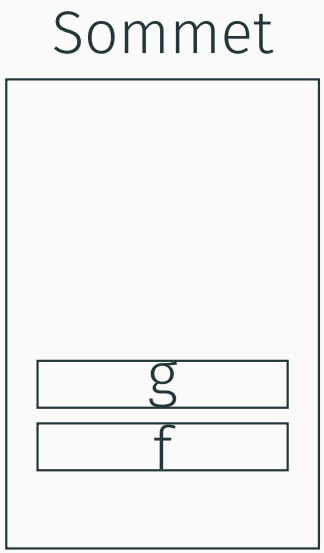


Pile d'appels

L'empilement des espaces de noms obéit à une politique de *pile*

- sommet : appel en cours
- en-dessous : appels précédents
- (presque) tout en bas : espace de nom global

```
def f(x):
    return g(x) + 1
def g(x):
    return h(x) + 2
def h(x):
    return x + 3
```

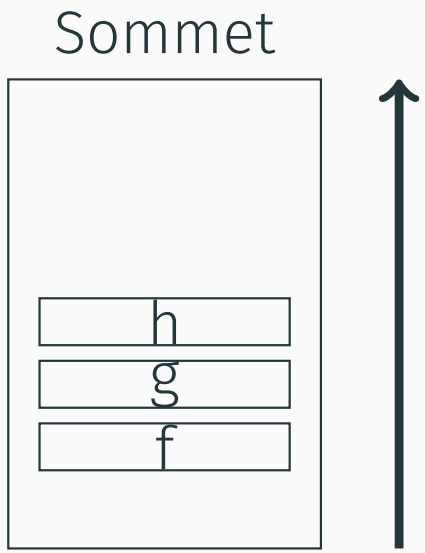


Pile d'appels

L'empilement des espaces de noms obéit à une politique de *pile*

- sommet : appel en cours
- en-dessous : appels précédents
- (presque) tout en bas : espace de nom global

```
def f(x):
    return g(x) + 1
def g(x):
    return h(x) + 2
def h(x):
    return x + 3
```

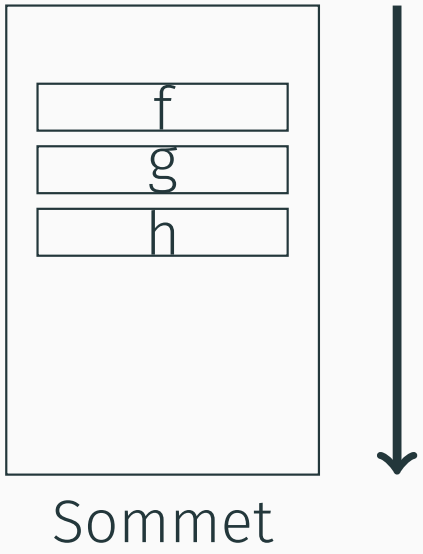


Pile d'appels

L'empilement des espaces de noms obéit à une politique de *pile*

- sommet : appel en cours
- en-dessous : appels précédents
- (presque) tout en bas : espace de nom global

```
def f(x):
    return g(x) + 1
def g(x):
    return h(x) + 2
def h(x):
    return x + 3
```



Attention ! Dans Python Tutor, le plus récent est en bas.

Pile d'appels

Quand un nouvel appel commence

- l'exécution de la fonction en cours s'interrompt
- un nouvel espace de noms local est créé
- l'exécution de la fonction appelée commence

Quand l'appel en cours se termine

- son espace de nom est supprimé de la pile
- l'exécution de l'appel précédent reprend

Portée des variables

Accès à la valeur d'une variable

- possible pour n'importe quel nom défini dans un des espaces de noms antérieurs
- si plusieurs espaces contiennent le même nom, c'est le plus récent qui est sélectionné

Affectation

- par défaut, uniquement aux variables locales
- pour une variable dans un espace de nom plus ancien, mots-clés **`global`** ou **`nonlocal`**
  (à utiliser avec précaution)

Signature d'une fonction

Considérons la fonction :

```
def f(p_1, ..., p_n):
...
return expr
```

Les noms `p_1` à `p_n` sont appelés paramètres (formels).

En général, ils d'un certain type. On peut le préciser en explicitant la *signature* de la fonction :

```
def f(p_1: type_1, ..., p_n: type_n) -> type_retour
...
return expr
```

Exemple

```
def minimum(a: int, b: int) -> int:
```

Déroulement détaillé d'un appel

Considérons maintenant l'appel `f(e_1, ..., e_n)` :

- les valeurs `v_1` à `v_n` sont appelées paramètres effectifs (ou arguments)
- création d'un espace de noms local contenant `p_1` à `p_n` au sommet de la pile d'appels
- chaque expression `e_i` est évaluée en une valeur `v_i` et affectée à la variable `p_i`
- exécution du corps de la fonction dans l'espace de noms local

Déroulement détaillé d'un appel

Considérons maintenant l'appel `f(e_1, ..., e_n)` :

- les valeurs `v_1` à `v_n` sont appelées paramètres effectifs (ou arguments)
- création d'un espace de noms local contenant `p_1` à `p_n` au sommet de la pile d'appels
- chaque expression `e_i` est évaluée en une valeur `v_i` et affectée à la variable `p_i`
- exécution du corps de la fonction dans l'espace de noms local
- si la fonction exécute l'instruction `return expr` ou atteint la fin de son bloc d'instructions :
  - l'espace de noms local est détruit
  - l'expression appelante `f(e_1, ..., e_n)` prend la valeur de `expr` (respectivement `None`)
  - reprise du programme principal dans l'espace global

Documentation et test de fonctions

Chaînes de documentation (`docstring`)

Bonne pratique : indiquer par un commentaire

- à quoi sert une fonction
- ce que représentent ses paramètres et leur type
- ce que représente sa valeur de retour
- d'éventuels effets ou causes secondaires

```
def triple(n):
    """
    Calcule le triple du nombre n (int ou float)
    ou la répétition trois fois de la chaîne n.
    """
    return n * 3
```

Documentation et test de fonctions

- On peut accéder à la chaîne de documentation d'une fonction en tapant `help(nom de la fonction)` dans l'interpréteur
- Cela fonctionne aussi pour les fonctions prédéfinies ou issues de modules

Tests intégrés à la documentation (`doctest`)

- Toute fonction *doit être testée immédiatement* pour s'assurer qu'elle fonctionne.
- On peut intégrer les tests à sa documentation.

```
def triple(n):
    """
    Calcule le triple du nombre n (int ou float)
    ou la répétition trois fois de la chaîne n.

    >>> triple(3)
    9
    >>> triple(9.0)
    27.0
    """
    return n * 3
```

Tests intégrés à la documentation (`doctest`)

Il existe des outils qui permettent de lancer automatiquement tous les tests présents dans la documentation, et de vérifier qu'ils produisent les résultats annoncés.

Par exemple, à la fin d'un programme, on peut écrire le code suivant pour lancer systématiquement tous les tests présents dans le fichier :

```
import doctest
doctest.testmod()
```