

Algorithmique Appliquée

BTS SIO SISR

Procédures et fonctions



CHAMBRE DE COMMERCE
ET D'INDUSTRIE

1^{er} ACCÉLÉRATEUR DES ENTREPRISES



Loïc Yvonnet



Plan

- Procédures : définition et appel
- Arguments
- Valeurs par défaut
- Variables locales et globales
- Fonctions
- Spécifications et contrat
- Nombre variable d'arguments
- Retour de plusieurs résultats
- Un mot sur la récursivité
- Fonctions d'ordre supérieur
- Fonctions lambda
- Programmation impérative et fonctionnelle
- Un mot sur les méthodes

Procédures : définition et appel

Comment réutiliser ce code ?

```
a0 = 16

s = a0 / 2
epsilon = 0.001

while abs(s ** 2 - a0) >= epsilon:
    P = s ** 2 - a0
    P_prime = 2 * s
    s = s - P / P_prime
```

- ➔ On souhaite pouvoir appeler ce code pour n'importe quelle valeur de a_0 .

Comment combiner des algorithmes ?

```
a0, b0, epsilon = 16, -27, 0.001

# Racine carrée
sa = a0 / 2
while abs(sa ** 2 - a0) >= epsilon:
    P = sa ** 2 - a0
    P_prime = 2 * sa
    sa = sa - P / P_prime

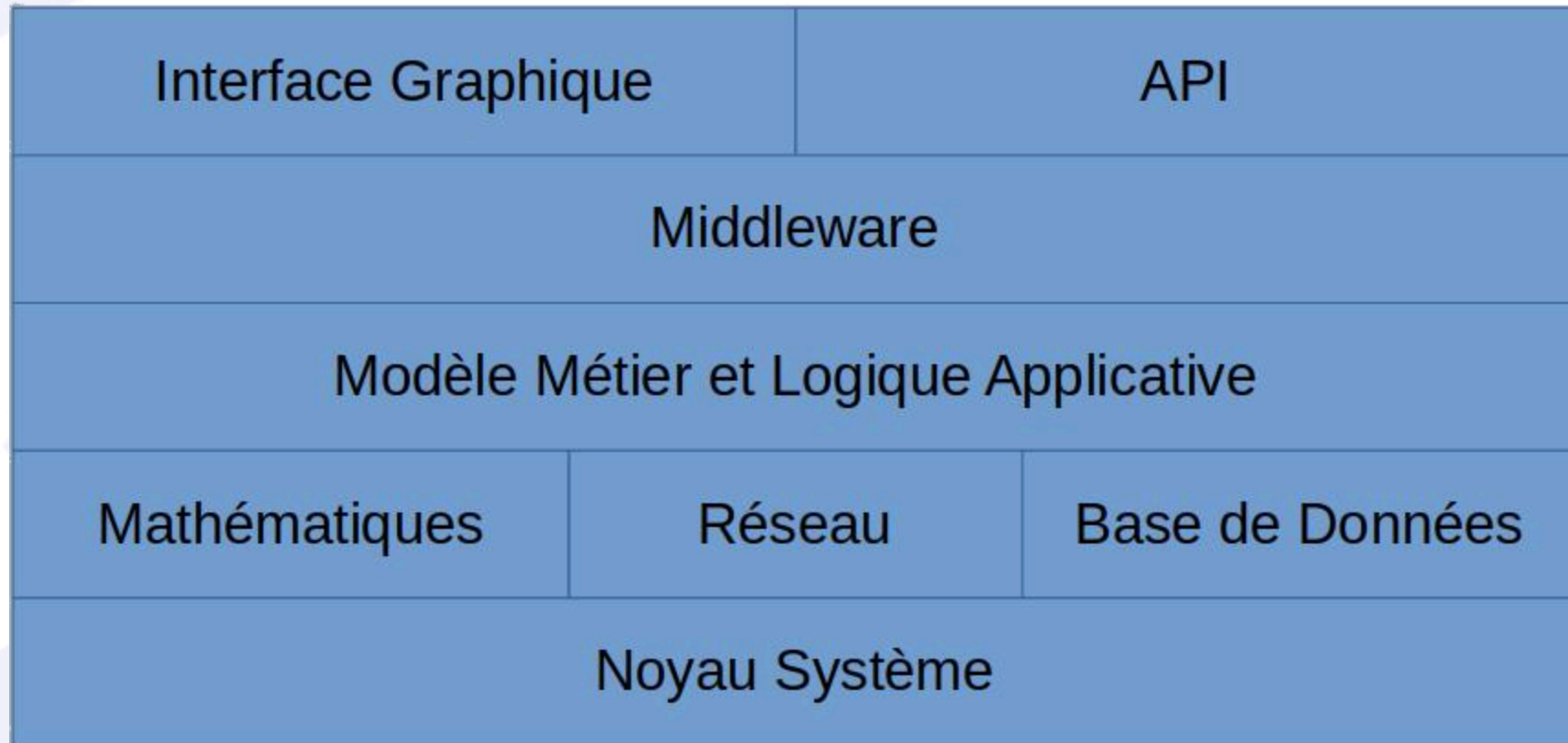
# Racine cubique
positif = True
if b0 < 0:
    positif, b0 = False, -b0
sb = b0 / 2
while abs(sb ** 3 - b0) >= epsilon:
    P = sb ** 3 - b0
    P_prime = 3 * sb ** 2
    sb = sb - P / P_prime
if not positif:
    sb = -sb

s = sa + sb
```

Réutilisabilité

- On souhaite pouvoir **réutiliser** des algorithmes.
- On souhaite pouvoir **appeler** et **paramétrer** les appels à nos algorithmes.
- On souhaite pouvoir **combiner** facilement nos algorithmes.

Couches d'abstraction





Procédure

- Une **procédure** est une suite d'instructions.
- Il est possible d'appeler plusieurs fois une procédure.

Syntaxe d'une procédure

```
def procedure():  
    print("Première procédure")
```

Appel d'une procédure (invocation)

```
def procedure():  
    print("Première procédure")
```

```
procedure()  
procedure()  
procedure()
```



```
Première procédure  
Première procédure  
Première procédure
```

Exemple avec racine cubique

```
def racine_cubique_27():
    """Affiche la racine cubique de 27."""
    b0 = 27
    epsilon = 0.001
    positif = True
    if b0 < 0:
        positif = False
        b0 = -b0
    sb = b0 / 2
    while abs(sb ** 3 - b0) >= epsilon:
        P = sb ** 3 - b0
        P_prime = 3 * sb ** 2
        sb = sb - P / P_prime
    if not positif:
        sb = -sb
    print(sb)

racine_cubique_27()
```



3.0000000081210202



Arguments

Passage d'un argument

```
def carre(x):  
    print(f"{x ** 2}")
```

```
carre(2)  
carre(3)
```



```
4  
9
```

Argument pour le calcul de la racine cubique

```
def racine_cubique(x):  
    epsilon = 0.001  
    positif = True  
    if x < 0:  
        positif = False  
        x = -x  
    s = x / 2  
    while abs(s ** 3 - x) >= epsilon:  
        P = s ** 3 - x  
        P_prime = 3 * s ** 2  
        s = s - P / P_prime  
    if not positif:  
        s = -s  
    print(s)
```

```
racine_cubique(27)  
racine_cubique(-27)
```



```
3.0000000081210202  
-3.0000000081210202
```

Plusieurs arguments

```
def somme(a, b)  
    resultat = a + b  
    print(resultat)
```

```
somme(2, 3)  
somme(3, 4)
```



```
5  
7
```

Passage par valeur et passage par référence (1/2)

- Certains langages (comme C++ ou C#) font la distinction entre le passage d'arguments **par valeur** et le passage **par référence**.
- **En Python, seul le passage par valeur existe.**
- En C#, par défaut, les arguments sont également passés par valeur, comme en Python.
- Le passage par référence permet de modifier le paramètre d'entrée.

Passage par valeur et passage par référence en C# (2/2)

```
static void passage_par_valeur(int a)
{
    a = a + 1;
    Console.WriteLine(a); // affiche 1
}

static void passage_par_reference(ref int a)
{
    a = a + 1;
    Console.WriteLine(a); // affiche 1
}

static void Main()
{
    x = 0;

    passage_par_valeur(x); // passe une copie de x
    Console.WriteLine(x); // affiche 0

    passage_par_reference(ref x); // passe une référence vers x
    Console.WriteLine(x); // affiche 1
}
```

Valeurs par défaut

Intérêt

- Une valeur par défaut pour un argument permet de **simplifier l'appel** dans les cas classiques.
- Cela donne à l'appelant **plus de possibilités** de paramétrage dans les cas particuliers.

Puissance

```
def puissance(x, exposant=2):  
    print(f"{x ** exposant}")
```

```
puissance(2)  
puissance(2, 3)
```



```
4  
8
```

Racine cubique avec ε

```
def racine_cubique(x, epsilon=0.0000001):
    positif = True
    if x < 0:
        positif = False
        x = -x
    s = x / 2
    while abs(s ** 3 - x) >= epsilon:
        P = s ** 3 - x
        P_prime = 3 * s ** 2
        s = s - P / P_prime
    if not positif:
        s = -s

    print(s)

racine_cubique(27)
racine_cubique(27, 1)
```



```
3.0000000000000002
3.0004936436555805
```

Tous les arguments sont éligibles

```
def bonjour(prenom="Amélie", nom="Poulain"):  
    print(f"Bonjour {prenom} {nom}")
```

```
bonjour()  
bonjour(nom="Teng")
```



```
Bonjour Amélie Poulain  
Bonjour Amélie Teng
```

Attention

```
def bonjour(prenom="Amélie", nom):  
    print(f"Bonjour {prenom} {nom}")
```



```
def bonjour(prenom="Amélie", nom):  
    print(f"Bonjour {prenom} {nom}")  
^  
SyntaxError: non-default argument follows default argument
```

Variables locales et globales

Scope

Portée des variables

```
def f(a):  
    print(a)  
  
def g(a, b):  
    f(a)  
    f(b)  
  
def h():  
    a = 3  
    b = a + 2  
    g(b, a)
```

```
h()
```



```
5  
3
```

Variable globale

```
a = 5

def f(a=10):
    print(a)

f()
f(a)
```



```
10
5
```

Accès en lecture à une variable globale

```
a = 3

def f():
    for _ in range(a):
        print(a * "*")

f()
```



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

Accès en écriture à une variable globale

```
a = 3

def f():
    a -= 1
    for _ in range(a):
        print(a * "*")

f()
```



```
a -= 1
```

UnboundLocalError: local variable 'a' referenced before assignment 28

Mot clé `global`

```
a = 3

def f():
    global a
    a -= 1
    for _ in range(a):
        print(a * "*")

f()
```



```
**
**
```



Fonctions

Généralisation

- Une procédure est une **fonction** qui ne retourne pas de résultat.
- Une procédure est donc une fonction.
- De manière générale, on parle toujours de fonctions.

Retourner un résultat

```
def somme(a, b):  
    """Renvoie la somme de a + b."""  
    return a + b  
  
resultat = somme(1, 2)
```



3

Définition formelle (1/3)

```
def nom_de_la_fonction(liste_de_parametres):  
    corps_de_la_fonction
```

Définition formelle (2/3)

- Lorsque l'on appelle (ou *invoque*) une fonction :
 - les expressions qui forment les paramètres sont évaluées.
 - les paramètres formels de la fonction sont liés aux valeurs de ces expressions (passage par valeur).
 - le point d'exécution est déplacé depuis le point d'invocation à la première instruction du corps de la fonction.
 - *cf. diapositive suivante...*

Définition formelle (3/3)

- Lorsque l'on appelle (ou *invoque*) une fonction :
 - *cf. diapositive précédente...*
 - le corps de la fonction est exécuté :
 - jusqu'à une instruction `return`, auquel cas la valeur de la fonction devient la valeur de cette expression `return`,
 - ou alors jusqu'à ce qu'il n'y ait plus d'instruction à exécuter, auquel cas `None` est retourné.

Fonction racine carrée

```
def racine_carree(x, epsilon=0.000001):  
    """Renvoie la racine carrée de x à epsilon près."""  
    s = x / 2  
    while abs(s ** 2 - x) >= epsilon:  
        P = s ** 2 - x  
        P_prime = 2 * s  
        s = s - P / P_prime  
  
    return s  
  
resultat = racine_carree(16)
```



4.0000000000000004

Fonction racine cubique

```
def racine_cubique(x, epsilon=0.000001):  
    """Renvoie la racine cubique de x à epsilon près."""  
    positif = True  
    if x < 0:  
        positif = False  
        x = -x  
    s = x / 2  
    while abs(s ** 3 - x) >= epsilon:  
        P = s ** 3 - x  
        P_prime = 3 * s ** 2  
        s = s - P / P_prime  
    if not positif:  
        s = -s  
    return s  
  
resultat = racine_cubique(-27)
```



-3.0000000000000002

Retour au problème initial

```
r2 = racine_carree(16)  
r3 = racine_cubique(-27)  
resultat = somme(r2, r3)
```



```
1.000000000000000022
```

Plusieurs retours

```
def converti_nombre(x):  
    """Converti le nombre x textuel sous un format entier."""  
    if x == "zéro":  
        return 0  
    elif x == "un":  
        return 1  
    elif x == "deux":  
        return 2  
    else:  
        return "non défini"  
  
resultat = converti_nombre("un")
```



1

Retour de plusieurs résultats

```
def ajoute_soustrait_et_multiplie(a, b):  
    """Renvoie a + b, a - b et a * b."""  
    somme = a + b  
    difference = a - b  
    produit = a * b  
  
    return somme, difference, produit  
  
ajout, diff, prod = ajoute_soustrait_et_multiplie(3, 2)  
print(f"somme : {ajout}; différence : {diff} ; produit : {prod}")
```



```
somme : 5; différence : 1 ; produit : 6
```


Spécifications et contrat

Intérêt de la documentation des spécifications

- Vous avez étudié, lors du dernier cours, l'algorithme de Newtown-Raphson.
- Cet algorithme n'est pas trivial.
- De manière générale, il faut considérer qu'**aucun algorithme n'est trivial**.
- Il faut **documenter son code**.
- En particulier, il faut documenter chacune de ses fonctions.

Les docstrings

- Chaque langage de programmation a ses bonnes pratiques de documentation.
- En Python, on documente le contrat de nos fonctions en utilisant une **docstring**.
- Une **docstring** commence et termine par un triple double-guillemet sur plusieurs lignes.

```
"""La première ligne décrit de manière concise le but de la fonction.
```

```
On laisse ensuite une ligne vide avant de rentrer plus dans les détails.  
Ensuite, on documente chaque entrée, puis chaque sortie.
```

```
"""
```

Exemple avec la fonction

somme

```
def somme(a, b):  
    """Retourne la somme des arguments.  
  
    a - entier, flottant ou chaîne de caractères.  
    b - entier, flottant ou chaîne de caractères.  
    Retourne la somme a + b.  
    """  
    return a + b
```

Exemple avec la fonction \sqrt{x}

```
def racine_carree(x, epsilon=0.000001):  
    """Renvoie la racine carrée de x à epsilon près.  
  
    Calcule la racine carrée d'un nombre x positif en employant  
    l'algorithme de Newtown-Raphson.  
    x - nombre flottant positif ou nul.  
    epsilon - nombre flottant strictement positif.  
    Retourne une valeur proche de la racine carrée de x, à plus  
    ou moins epsilon près.  
    """  
    s = x / 2  
    while abs(s ** 2 - x) >= epsilon:  
        P = s ** 2 - x  
        P_prime = 2 * s  
        s = s - P / P_prime  
  
    return s
```

Aide (1/2)

```
help(round)
```



```
round(number, ndigits=None)
```

Round a number to a given precision in decimal digits.

The return value is an integer if `ndigits` is omitted or `None`. Otherwise the return value has the same type as the number. `ndigits` may be negative.

Aide (2/2)

```
help(racine_carree)
```



```
racine_carree(x, epsilon=1e-06)
```

Renvoie la racine carrée de x à epsilon près.

Calcule la racine carrée d'un nombre x positif en employant l'algorithme de Newton-Raphson.

x - nombre flottant positif ou nul.

ϵ - nombre flottant strictement positif.

Retourne une valeur proche de la racine carrée de x , à plus ou moins epsilon près.

TD : Fonctions géométriques simples



TD : Fonctions géométriques simples

Lien vers le sujet de TD.

Nombre variable d'arguments

Fonction `print`

```
arg1 = "La fonction print"  
arg2 = "peut prendre"  
arg3 = "N"  
arg4 = "arguments"  
print(arg1, arg2, arg3, arg4)
```



La fonction `print` peut prendre N arguments.

Fonction `max`

```
max_pair = max(1, 5)
max_serie = max(4, 8, 0, -1, 4, 5)
print(f"{max_pair}\n{max_serie}")
```



```
5
8
```

Intérêt

- La capacité à passer une liste variable d'arguments offre de la **flexibilité** pour l'appelant.
- La sémantique au niveau de l'appelant est claire.
- Il vous est possible de définir vos propres fonctions à nombre variable d'arguments positionnels.

Opérateur * de déballage (*unpacking operator*).

```
def moyenne(*arguments):  
    """Renvoie la moyenne des arguments.  
  
    arguments - doit comporter au moins une valeur et toutes  
                les valeurs sont numériques.  
    Retourne la moyenne de ces arguments.  
    """  
    total = 0  
    for argument in arguments:  
        total += argument  
  
    return total / len(arguments)  
  
resultat = moyenne(1, 2, 3, 4, 5, 6, 7, 8, 9)  
print(resultat)
```



Autre exemple

```
def log(message, *valeurs):  
    """Affiche dans la sortie standard le message et la liste des valeurs.  
  
    message - chaîne de caractères à afficher.  
    valeurs - liste d'arguments variable de valeurs à afficher.  
    """  
    if not valeurs:  
        print(message)  
    else:  
        valeurs_str = str(valeurs[0])  
        for valeur in valeurs[1:]:  
            valeurs_str += ", " + str(valeur)  
        print(f"{message} : {valeurs_str}")  
  
log("Bonjour")  
log("Mes valeurs", 7, 42, 3.14)
```



```
Bonjour  
Mes valeurs : 7, 42, 3.14
```

Arguments positionnels et nommés (1/2)

- Il est possible d'appeler une fonction avec les arguments dans l'ordre de leur déclaration. Il s'agit d'**arguments positionnels**.
- Il est également possible d'appeler une fonction en **spécifiant les noms des arguments et leurs valeurs**. Il s'agit d'**arguments nommés**.

Arguments positionnels et nommés (2/2)

```
def debit(diff_poids, diff_temps, periode=1, unites_par_kg=1):  
    return ((diff_poids * unites_par_kg) / diff_temps) * periode
```

- Les 2 premiers arguments sont positionnels, et les 2 derniers sont nommés.
- Il est possible de définir des fonctions avec un nombre variable d'arguments nommés.

Nombre variable d'arguments nommés

```
def affiche_parametres(**kwargs):  
    """Affiche simplement les paramètres d'entrée."""  
    for cle, valeur in kwargs.items():  
        print(f"{cle} : {valeur}")  
  
affiche_parametres(a=1, b=3, c=5)  
affiche_parametres(prenom="Louise", nom="Clark")
```



```
a : 1  
b : 3  
c : 5  
prenom : Louise  
nom : Clark
```

Retour de plusieurs résultats

Intérêt

- Nous avons vu que le passage d'arguments se fait par valeur en Python.
- Il est donc nécessaire de retourner les résultats, et il peut y en avoir plusieurs.
- Nous avons vu des exemples avec les vecteurs lors du TP précédent.

Un autre exemple de retour de plusieurs résultats

```
def echange(premier, second):  
    return second, premier
```

```
un = 1  
deux = 2  
un, deux = echange(un, deux)  
print(f"{un}, {deux}")
```



2, 1

Inintérêt d'un sous-ensemble de résultats

- Il peut arriver que certains résultats ne soient pas pertinents dans notre contexte d'appel.
- Une convention en Python consiste à utiliser `_` (underscore) pour une variable dont la valeur ne nous intéresse pas.

```
def quelques_elements():  
    return 1, 2, 3  
  
un, _, trois = quelques_elements()  
print(f"un : {un} ; trois : {trois}")
```



```
un : 1 ; trois : 3
```

Opérateur de déballage pour retours de fonction

Vous pouvez utiliser l'unpacking operator lorsqu'une fonction retourne un grand nombre de résultats.

```
def longue_liste():  
    return 1, 2, 3, 4, 5, 6, 7, 8, 9  
  
un, *entre, neuf = longue_liste()  
print(f"un : {un} ; neuf : {neuf} ; nombre d'autres : {len(entre)}")
```



```
un : 1 ; neuf : 9 ; nombre d'autres : 7
```

Autre exemple

```
def longue_liste():  
    return 1, 2, 3, 4, 5, 6, 7, 8, 9  
  
un, deux, *autres = longue_liste()  
print(f"un : {un} ; deux : {deux} ; nombre d'autres : {len(autres)}")
```

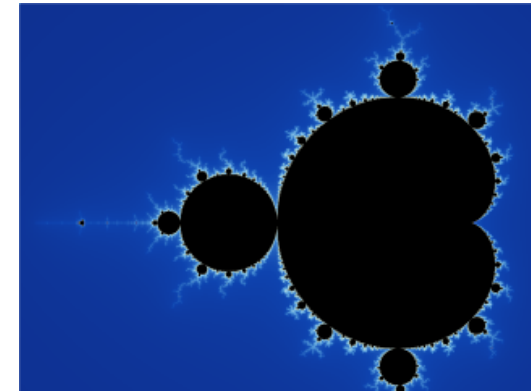
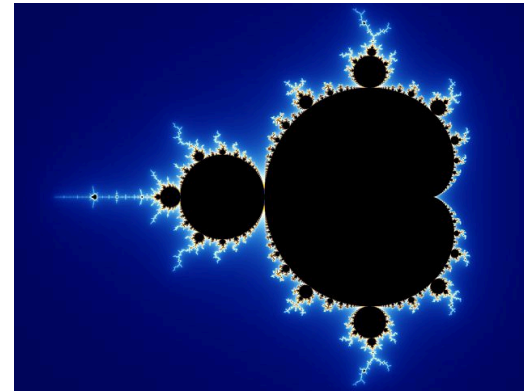


```
un : 1 ; deux : 2 ; nombre d'autres : 7
```


Un mot sur la récursivité

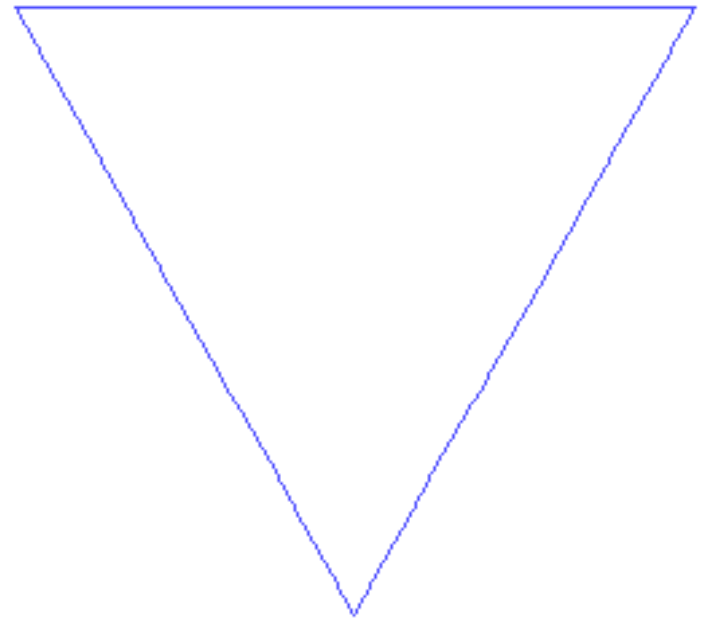
Fractal

- Un fractal est un motif géométrique dont la forme se répète indéfiniment à différentes échelles.
- L'exemple ci-contre est la séquence de Mandelbrot.



Fractal et récursivité

- Un fractal est récursif dans le sens où il se répète lui-même.
- Le flocon de Von Koch ci-contre part d'un triangle et lui applique plusieurs fois la même fonction.
- On dit que cette fonction est appliquée de manière récursive.



Accronymes récursifs

- Ces acronymes sont récursifs :
 - **GNU** is **N**ot **U**nix
 - **WINE** Is **N**ot an **E**mulator
 - **CURL** **U**RL **R**equest **L**ibrary
 - **PHP** **H**ypertext **P**reprocessor
 - **GRPC** **R**emote **P**rocedure **C**all
 - **YAML** **A**in't **M**arkup **L**anguage
- On les dit récursifs car ils se répètent indéfiniment.

Définition : fonction récursive

- Une fonction est **récursive** si elle s'appelle elle-même.
- La récursion peut être **directe** si la fonction s'appelle elle-même directement.
- La récursion peut être **indirecte** si la fonction appelle une séquence de fonctions qui finit par appeler la fonction initiale.
- La récursion doit avoir une **condition de fin**. Sinon, son exécution prendrait un temps infini.

Suites mathématiques

- En mathématiques, on peut définir la plupart des suites arithmético-géométriques de la manière suivante :

$$\forall \{a, b, b_0, c, N\} \in \mathbb{N}^5, \begin{cases} f(0) = b_0 \\ f(N) = a f(N-1)^c + b \end{cases}$$

- Il s'agit d'une **définition réursive** car l'évaluation de la fonction au rang N dépend des valeurs des rangs inférieurs.

En Python

```
def f(N, a=1, b=0, c=1, b0=0):  
    """Calcule la Nième valeur de la suite arithmético-géométrique f.  
  
    f est définie telle que  $f(0) = b_0$ , et  $f(N) = a \cdot f(N-1)^c + b$  sinon.  
    N - nombre entier strictement positif.  
    a - nombre entier utilisé comme multiplicateur géométrique.  
    b - nombre entier utilisé comme raison arithmétique.  
    c - nombre entier comme puissance.  
    b0 - nombre entier constituant le début de la suite.  
    Retourne la Nième valeur de la suite.  
    """  
    if N == 0:  
        return b0  
    else:  
        return a * (f(N - 1, a, b, c, b0) ** c) + b
```

Simplification de l'exemple précédent

Pour $a = 2$, $b = 1$, $b_0 = 0$, $c = 1$:

```
def f(N):  
    """Calcule la Nième valeur de la suite arithmético-géométrique f.  
  
    f est définie telle que  $f(0) = 0$ , et  $f(N) = 2 * f(N-1) + 1$  sinon.  
    Retourne la Nième valeur de la suite.  
    """  
    if N == 0:  
        return 0  
    else:  
        return 2 * f(N - 1) + 1  
  
resultat = f(3)  
print(resultat)
```



Instrumentation de l'exemple précédent

```
def f(N):  
    if N == 0:  
        print("f(0) = 0")  
        return 0  
    else:  
        precedent = f(N - 1)  
        actuel = 2 * precedent + 1  
        print(f"f({N}) = {actuel}")  
        return actuel  
  
resultat = f(3)
```



```
f(0) = 0  
f(1) = 1  
f(2) = 3  
f(3) = 7
```

Limites de la pile

```
def f(N):  
    if N == 0:  
        return 0  
    else:  
        precedent = f(N - 1)  
        actuel = 2 * precedent + 1  
        return actuel  
  
resultat = f(1000000)
```



```
RecursionError: maximum recursion depth exceeded  
Fatal Python error: _Py_CheckRecursiveCall: Cannot recover from stack overflow.
```

Notes concernant la récursivité

- Certains problèmes ont une **définition naturellement récursive**.
- Ces problèmes sont **plus faciles** à résoudre en utilisant la récursivité.
- Parfois, les solutions itératives équivalentes sont très difficiles à trouver.
- Les **solutions itératives** sont presque toujours **meilleures** car :
 - Elles n'engendrent pas de *stack overflow*.
 - Elles nécessitent souvent moins de mémoire car le contexte d'appel n'a pas à être sauvegardé.

Fonctions d'ordre supérieur

Fonctions en tant qu'objets

Intérêt

- Un **code propre** est écrit en fonction d'algorithmes et de structures de données.
- Lorsque l'on écrit des bibliothèques de fonctions, on souhaite que les fonctions fournies puissent être utilisées dans une grande variété de contextes.
- Les **fonctions d'ordre supérieur** sont un outil puissant pour **réutiliser** des algorithmes et les **généraliser**.

Définition

- Une **fonction d'ordre supérieur** est une fonction qui fait au moins l'une des 2 choses suivantes :
 - Prend une fonction comme argument.
 - Renvoie une fonction comme résultat.

Tout est objet

- En Python, tout est objet.
- En particulier, **une fonction est un objet.**

```
def foo():  
    pass  
  
foo_type = type(foo)  
print(f"{foo_type}")
```



```
<class 'function'>
```

Assignment d'une fonction à une variable

```
def f():  
    return 1  
  
def g():  
    return 2  
  
fonction = f    # la variable "fonction" est liée à f  
a = fonction() # f est appelée  
  
fonction = g    # la variable "fonction" est liée à g  
b = fonction() # g est appelée  
  
print(f"a = {a} ; b = {b}")
```



```
a = 1 ; b = 2
```


Généralisation de la dichotomie

```
def dichotomie(x, f, debut=0, fin=1000, epsilon=0.001):
```

```
    """Cacule la racine r telle que  $f(r) - x = 0$  par dichotomie.
```

```
    Généralisation de l'algorithme de dichotomie sur un intervalle [debut ; fin]
    avec une fonction d'évaluation f pour le calcul d'une racine r. La racine r
    doit être dans l'intervalle de recherche, sinon la condition de fin de
    l'algorithme n'est pas garantie.
```

```
    x - nombre flottant dont on recherche la racine  $|f(r) - x| < \text{epsilon}$ .
```

```
    f - fonction d'évaluation prenant et renvoyant un flottant. Cette fonction
    doit être dérivable sur l'intervalle [debut ; fin].
```

```
    debut - début de l'intervalle de recherche de r.
```

```
    fin - fin de l'intervalle de recherche de r.
```

```
    epsilon - erreur acceptable qui doit être strictement supérieure à 0.
```

```
    Renvoie la racine r telle que  $|f(r) - x| < \text{epsilon}$ .
```

```
    """
```

```
    r = (debut + fin) / 2
```

```
    while abs(f(r) - x) >= epsilon:
```

```
        if f(r) < x:
```

```
            debut = r
```

```
        else:
```

```
            fin = r
```

```
        r = (debut + fin) / 2
```

```
    return r
```

Recherche dans un intervalle avec dichotomie d'ordre supérieur

```
def affine(x):  
    """Renvoie la valeur en entrée."""  
    return x  
  
resultat = dichotomie(50, affine)  
print(resultat)
```



49.999237060546875

Racine carrée avec dichotomie d'ordre supérieur

```
def racine_carree(x, epsilon=0.001):  
    """Renvoie la racine carrée de x."""  
  
    def carre(x):  
        return x ** 2  
  
    debut = 0  
    fin = max(1, x)  
  
    return dichotomie(x, carre, debut, fin, epsilon)  
  
resultat = racine_carree(25)  
print(resultat)
```



4.9999237060546875

Générateur de fonction (1/3)

- Le principe d'un générateur de fonction : on **retourne une nouvelle fonction** en capturant les entrées.
- Exemple : série mathématique paramétrable.

Générateur de fonction (2/3)

```
def suite(a=1, b=0, c=1, b0=0):  
    """Renvoie la suite f telle que  $f(0) = b_0$ , et  $f(N) = a \cdot f(N-1)^c + b$ .  
  
    a - nombre entier utilisé comme multiplicateur géométrique.  
    b - nombre entier utilisé comme raison arithmétique.  
    c - nombre entier comme puissance.  
    b0 - nombre entier constituant le début de la suite.  
    Retourne une fonction prenant un paramètre entier N et renvoyant f(N).  
    """  
  
    def f(N):  
        return b0 if N == 0 else a * (f(N - 1) ** c) + b  
  
    return f
```

Générateur de fonction (3/3)

```
suite_arithmetique = suite(b=2)
print(f"f(0) = {suite_arithmetique(0)}")
print(f"f(1) = {suite_arithmetique(1)}")
print(f"f(2) = {suite_arithmetique(2)}")
print(f"f(3) = {suite_arithmetique(3)}")
```



```
f(0) = 0
f(1) = 2
f(2) = 4
f(3) = 6
```

Fonctions lambda

Origine

- Le **calcul lambda**, noté λ -calcul est un système mathématique formel pour exprimer des calculs quelconques à partir des concepts de **fonction** et d'application.
- Il est inventé dans les années 1930 par Alonzo Church.
- Ce système est **Turing-complet**.
- Dans ce système, tout est fonction.

Fonctions lambda dans les langages impératifs

- Une fonction lambda :
 - n'a pas de nom - elle est **anonyme**.
 - est courte.
- Les fonctions lambda sont utilisés dans la **programmation d'ordre supérieur**.
- Ces fonctions offrent une syntaxe plus légère pour les fonctions internes.

Fonction lambda en Python

- Le mot clé `lambda` est suivi d'une liste de paramètres puis d'une expression.
- Cette expression est la valeur de retour de la fonction lambda.
- Exemple :

```
lambda x : x ** 2
```

Racine carrée avec dichotomie et lambda

```
def racine_carree(x, epsilon=0.001):  
    """Renvoie la racine carrée de x."""  
    return dichotomie(x,  
                      lambda x : x ** 2,  
                      debut=0,  
                      fin=max(1, x),  
                      epsilon=epsilon)  
  
resultat = racine_carree(25)  
print(resultat)
```



4.9999237060546875

Générateur de suite avec lambda (1/2)

```
def suite(a=1, b=0, b0=0):  
    """Renvoie la suite f telle que  $f(0) = b_0$ , et  $f(N) = a \cdot f(N-1) + b$ .  
  
    a - nombre entier utilisé comme multiplicateur géométrique.  
    b - nombre entier utilisé comme raison arithmétique.  
    b0 - nombre entier constituant le début de la suite.  
    Retourne une fonction prenant un paramètre entier N et renvoyant f(N).  
    """  
    if a == 1:  
        # Calcul du terme général dans le cas d'une suite arithmétique  
        return lambda N : b0 + N * b  
    else:  
        # Calcul du terme général dans le cas général  
        r = b / (1 - a)  
        return lambda N : (a ** N) * (b0 - r) + r
```

Générateur de suite avec lambda (2/2)

```
ma_suite = suite(a=2, b=1)
print(f"f(1) = {ma_suite(1)}")
print(f"f(2) = {ma_suite(2)}")
print(f"f(3) = {ma_suite(3)}")
```



```
f(1) = 1.0
f(2) = 3.0
f(3) = 7.0
```

TP : Fonctions d'ordre supérieur



TP : Fonctions d'ordre supérieur

[Lien vers le sujet de TP.](#)

Programmation impérative

vs

Programmation fonctionnelle

Notions de pureté et d'immuabilité

Contraintes des langages fonctionnels (1/2)

- Dans un langage fonctionnel :
 - Les variables sont **immuables**. Elles ne peuvent pas changer de valeur.
 - Les fonctions sont **pures** :
 - Les sorties ne dépendent que des entrées.
 - Il n'y a donc pas d'accès possible à une variable globale.
 - Il n'y a pas d'effet de bord : lecture depuis un fichier, entrée console, etc.

Contraintes des langages fonctionnels (2/2)

- **Python n'est pas un langage fonctionnel** : on peut changer les valeurs des variables et accéder à des variables globales.
- Il est possible d'écrire du code fonctionnel avec Python en appliquant les principes fonctionnels.



Quelques langages fonctionnels notables

- Haskell
- Scala
- F#
- Erlang

Programmation impérative

- Etat **global** et **effets de bords** parfois difficiles à maîtriser.
- Pour comprendre une fonction : il faut potentiellement comprendre l'état global du programme.
- Un code non maîtrisé peut devenir un "plat de spaghetti".



Solution fonctionnelle

- La programmation fonctionnelle tente de remédier à ces problèmes en ayant une **approche plus proche des mathématiques**.

Pourquoi tout le monde ne fait pas du fonctionnel ? (1/2)

- **Performances :**

- Les langages fonctionnels sont interprétés et comportent tous un *garbage collector*.
- Un langage fonctionnel ne peut pas être aussi performant que C++ ou Rust.
- Un langage fonctionnel peut être aussi performant que Python.

Pourquoi tout le monde ne fait pas du fonctionnel ? (2/2)

- **Complexité :**
 - Les contraintes supplémentaires et le rattachement à la théorie des catégories peut sembler plus complexe à appréhender.
 - Les langages impératifs (ou multi-paradigmes) comme Python sont plus anciens et ont une plus grande communauté d'utilisateurs.
- **Bibliothèques :** Il existe moins de bibliothèques disponibles avec les langages fonctionnels.

Les langages impératifs sont-ils condamnés à produire du mauvais code ?

- **Non.**
- Le langage de programmation C est impératif et il est très facile à utiliser de manière incorrecte.
- L'interpréteur **CPython** est implémenté dans le langage de programmation C.
- CPython est de très bonne qualité et fait environ 1 million de lignes de code.

Kernel Linux

- Le **kernel Linux** est implémenté dans le langage de programmation C.
- Le kernel Linux fait **plusieurs millions de ligne de code**.
- Le kernel Linux est très propre et suit un **standard de programmation** très strict.
- La qualité du code est dépendante de **l'équipe de développeurs** plus que du langage.

Un mot sur les méthodes

Avec l'exemple du type `str`

Langage Orienté-Objet

- Python est un langage de programmation **multi-paradigmes**.
- Python est un langage impératif et également **orienté objet**.
- La programmation orientée objet ne sera *pas* étudiée dans ce cours.
- En revanche, vous utilisez des objets qui sont fournis avec le langage.

Méthodes sur `str` (1/2)

```
chaine = "bonjour, monde"

majuscules = chaine.upper()
print(majuscules)

capitales = chaine.capitalize()
print(capitales)

index_o = chaine.find("j")
print(index_j)
```

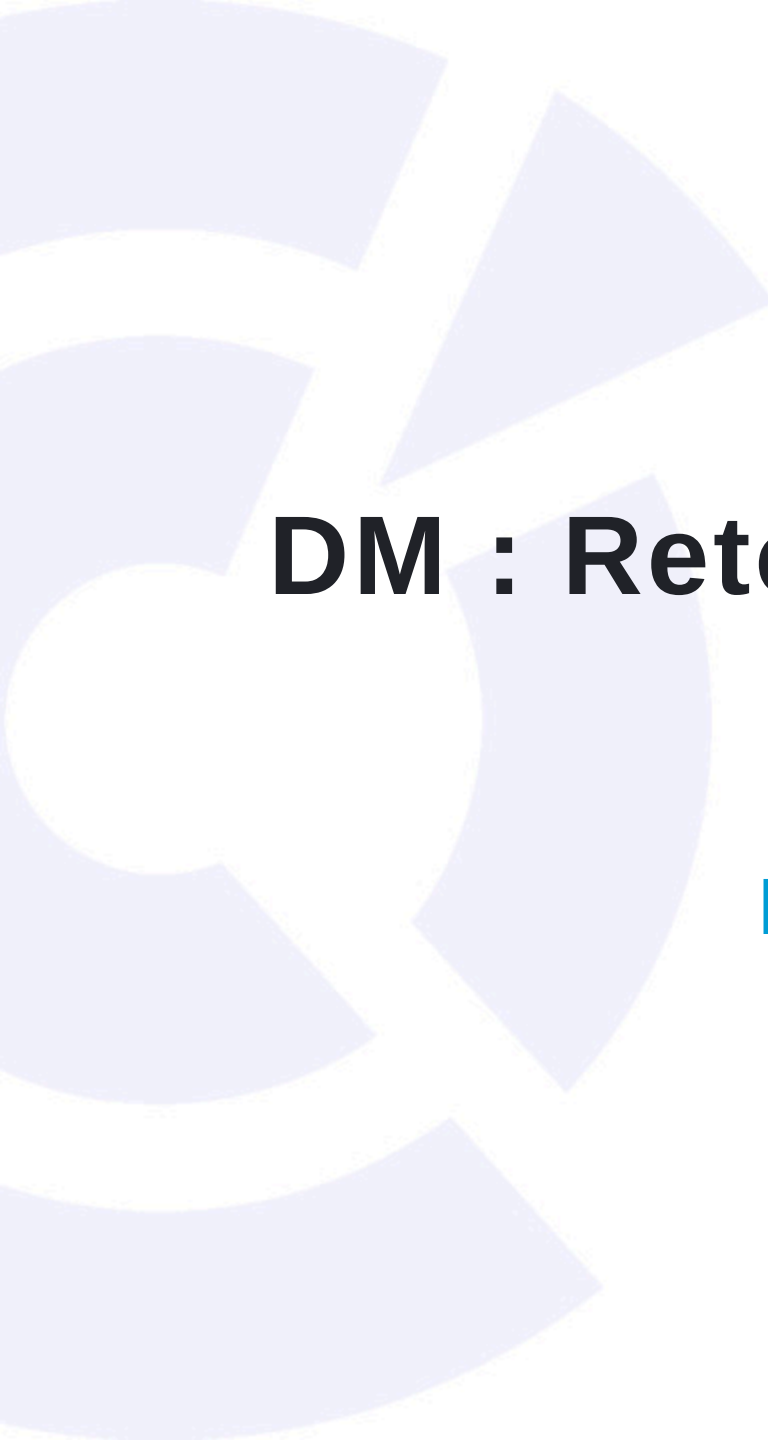


```
BONJOUR, MONDE
Bonjour, monde
3
```

Méthodes sur `str` (2/2)

- Les chaînes de caractères sont représentées par le type `str`.
- On a utilisé une syntaxe particulière pour appeler les fonctions `upper`, `capitalize` et `find` qui sont rattachées au type `str`.
- Ces fonctions rattachées à un type spécifiques s'appellent des **méthodes**.
- Chaque type peut définir son propre jeu de méthodes.

Devoir à la Maison 02



DM : Retours sur les fonctions et le débogage

[Lien vers le sujet de DM.](#)