ALGO 02 Fonctions et procédures

I. Introduction

▸ Intro

Nous avons déjà manipulé des fonctions et procédures. Par exemple en ALGO python, on a travaillé avec PY `eval`, PY `import`, ou PY `print`.

Ce sont des *vraies* fonctions, au sens algorithmique, telles qu'elles seront définies et étudiées dans ce chapitre.

▸ Remarques

— Les ALGO `fonctions` permettent notamment de décomposer un programme complexe en sous-programmes plus simples et d'améliorer ainsi la **lisibilité** d'un algorithme.

Ces sous-programmes peuvent en outre être utilisés plusieurs fois au sein d'un même programme, évitant ainsi des répétitions de code.

— Un autre intérêt des ALGO `fonctions` est qu'elles sont **réutilisables** dans d'autres programmes et par d'autres programmeurs.

▸ Illustration

L'image ci-dessous montre une petite partie du code ALGO python de la fonction PY `randint` qui sert à générer un entier aléatoire (en fait pseudo-aléatoire) :

```
## -------------------- integer methods  -------------------

def randrange(self, start, stop=None, step=_ONE):
    """Choose a random item from range(stop) or range(start, stop[, step]).

    Roughly equivalent to ``choice(range(start, stop, step))`` but
    supports arbitrarily large ranges and is optimized for common cases.

    """

    # This code is a bit messy to make it fast for the
    # common case while still doing adequate error checking.
    try:
        istart = _index(start)
    except TypeError:
        istart = int(start)
        if istart != start:
            _warn('randrange() will raise TypeError in the future',
                  DeprecationWarning, 2)
            raise ValueError("non-integer arg 1 for randrange()")
        _warn('non-integer arguments to randrange() have been deprecated '
              'since Python 3.10 and will be removed in a subsequent '
              'version',
              DeprecationWarning, 2)
    if stop is None:
        # We don't check for "step != 1" because it hasn't been
        # type checked and converted to an integer yet.
        if step is not _ONE:
            raise TypeError('Missing a non-None stop argument')
        if istart > 0:
            return self._randbelow(istart)
        raise ValueError("empty range for randrange()")

    # stop argument supplied.
    try:
        istop = _index(stop)
    except TypeError:
        istop = int(stop)
        if istop != stop:
            _warn('randrange() will raise TypeError in the future',
                  DeprecationWarning, 2)
            raise ValueError("non-integer stop for randrange()")
        _warn('non-integer arguments to randrange() have been deprecated '
              'since Python 3.10 and will be removed in a subsequent '
              'version',
              DeprecationWarning, 2)
    width = istop - istart
    try:
        istep = _index(step)
    except TypeError:
        istep = int(step)
        if istep != step:
            _warn('randrange() will raise TypeError in the future',
                  DeprecationWarning, 2)
            raise ValueError("non-integer step for randrange()")
        _warn('non-integer arguments to randrange() have been deprecated '
              'since Python 3.10 and will be removed in a subsequent '
              'version',
              DeprecationWarning, 2)
    # Fast path.
    if istep == 1:
        if width > 0:
            return istart + self._randbelow(width)
        raise ValueError("empty range for randrange() (%d, %d, %d)" % (istart, istop, width))

    # Non-unit step argument supplied.
    if istep > 0:
        n = (width + istep - 1) // istep
    elif istep < 0:
        n = (width + istep + 1) // istep
    else:
        raise ValueError("zero step for randrange()")
    if n <= 0:
        raise ValueError("empty range for randrange()")
    return istart + istep * self._randbelow(n)

def randint(self, a, b):
    """Return random integer in range [a, b], including both end points.
    """

    return self.randrange(a, b+1)
```

Au lieu de devoir réécrire toutes ces instructions à chaque fois que l'on désire obtenir un entier pseudo-aléatoire par exemple entre 1 et 6, il suffit d'écrire PY `randint(1,6)`, c'est-à-dire de faire **appel** à la fonction PY `randint` avec en **paramètres** 1 et 6.

▸ À savoir

Tout comme les variables, les fonctions portent un nom. Les règles de nommage d'une fonction sont les mêmes que les règles de nommage d'une variable.

Licence Ouverte Etalab 2.0

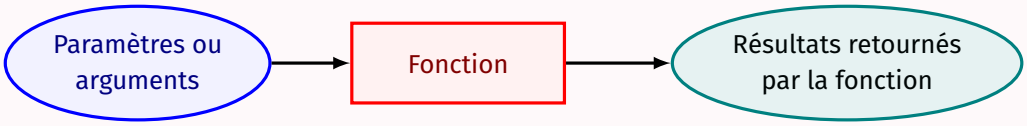
II. Définitions et structures

II.1. Définitions et premiers exemples

▸ Définitions 2.1

- Une fonction est un morceau de code pouvant recevoir en **entrée** une ou plusieurs valeurs à partir desquelles elle retourne une ou plusieurs valeurs en sortie.
- Les valeurs que la fonction reçoit en entrée s'appellent les **paramètres** ou les arguments de la fonction.
- Les valeurs **retournées** par la fonction sont les résultats de celle-ci.

▸ Illustration



▸ Exemple 2.1

La fonction ALGO python ci-dessous prend en paramètre un nombre à trois chiffres et retourne les trois chiffres de ce nombre dans l'ordre.

</> Code Python

```python
def decompose(nombre):
    chiffre1 = nombre // 100
    reste = nombre % 100
    chiffre2 = reste // 10
    chiffre3 = reste % 10
    return chiffre1, chiffre2, chiffre3
```

La fonction ci-dessus, nommée `decompose`, a les caractéristiques suivantes :

- elle a un unique paramètre nommé `nombre` ;
- elle retourne trois résultats qui sont nommés `chiffre1`, `chiffre2` et `chiffre3`.

II.2. Création

▸ Algorithme 2.1

Pour définir une `fonction`, on utilise la syntaxe suivante :

- En `pseudo-code` :

</> PseudoCode

```
Fonction nom_fonction(parametre_1, parametre_2, ...) :
Variables locales : ...
Debut Fonction
    instructions
    Retourner : valeur_1, valeur_2 ,...
Fin Fonction
```

- En ALGO python :

</> Code Python

```python
def nom_fonction(parametre_1, parametre_2 , ...) :
    instructions
    return valeur_1, valeur_2 ,...
```

▸ Exemples 2.2

En pseudo-code :

PseudoCode

```
Fonction cube(x) :
Variables locales : x, res
Debut Fonction :
    res ← x × x × x
    Retourner res
Fin Fonction
```

En python :

Code Python

```
def cube(x) :
    res = x * x * x
    return res
```

Utilisation (suite de l'algorithme) :

PseudoCode

```
a ← cube(3)
Afficher : a
Afficher : cube(4)
Afficher : cube(5)
```

Utilisation (suite de l'algorithme) :

Code Python

```
a = cube(3)
print(a)
print(cube(4))
print(cube(5))
```

La console permet de vérifier le bon fonctionnement de cette fonction (les `print` ne sont pas obligatoires si les lignes d'affichage sont saisies dans la console directement !) :

```
Début de la console Python
>>> a = cube(3)
>>> a
27
>>> cube(4)
64
>>> cube(5)
125
Fin de la console Python
```

▸ Remarques

- Les parenthèses après le nom de la fonction sont obligatoires. Si une fonction n'a pas de paramètre, on ne met rien entre les parenthèses.
- Le symbole `:` après les parenthèse est également obligatoire.
- L'instruction `Retourner` ou `return` précède la liste des valeurs retournées par la fonction.
  Attention, les instructions – algorithmiquement parlant – après le `return` ne sont pas effectuées.

▸ Définition 2.2

En python, les instructions qui sont dans la définition de la fonction (ou dans n'importe quelle autre structure de programmation) commencent par un espace horizontal appelé **indentation**.
L'indentation s'obtient grâce à la touche tabulation, via les touches ⇥ ou ⇤ (on peut également l'obtenir en appuyant quatre fois sur la touche Espace. Elle est **obligatoire** en Python.

▸ Remarques

L'**indentation** fait partie du code, elle sert à indiquer le début et la fin d'une *structure* : les instructions indentées sont dans la structure, celles qui ne sont pas indentées sont hors de la structure.
Les règles d'indentation sont donc à respecter scrupuleusement sous peine d'erreur(s) !!

II.3. Les procédures

▸ **Définition 2.3**

Il existe des structures en tout point similaires à des fonctions, mais ne retournant aucun résultat ou comprenant des « Entrées/Sorties » (`input`/`print`).
Dans ce cas, on parle de procédures, qui se construisent de la même façon que les fonctions en python (et en `pseudo-code`), mais elles sont autorisées à contenir d'autres éléments que `return`.

▸ **Remarque**

Une procédure pourra être utile dans le cas où une interaction avec l'utilisateur sera *nécessaire*, avec notamment des *questions* posées, et des *conclusions* affichées.

▸ **Algorithme 2.2**

La fonction `cube` précédente peut-être modifiée en procédure, mais dans ce cas on perd une information *fondamentale*, le `type` du résultat !

Code Python

```python
def proc_cube() :
    x = eval(input("Saisir un nombre x : "))
    res = x * x * x
    print(f"Le cube de {x} est {res}")
```

La console permet de vérifier le (bon) fonctionnement de cette procédure, et de constater une différence fondamentale entre une fonction et une procédure :

```
Début de la console Python
>>> cube(4)
64
>>> type(cube(4))
<class 'int'>
>>>
>>> cube(3.75)
52.734375
>>> type(cube(3.75))
<class 'float'>
>>>
>>> proc_cube()
Saisir un nombre x : 4
Le cube de 4 est 64
>>> type(proc_cube())
Saisir un nombre x : 4
Le cube de 4 est 64
<class 'NoneType'>
Fin de la console Python
```

III. Appel d'une fonction ou d'une procédure

III.1. Introduction

▸ **Remarque**

Définir une fonction est un peu comme écrire une recette de cuisine : on fait la liste des ingrédients (paramètre d'entrée), puis on note les instructions à effectuer avec ces ingrédients pour arriver à un résultat final.
Mais une fois qu'on a fini d'écrire la recette de cuisine, on n'a toujours rien cuisiné : il faut que quelqu'un réunisse les ingrédients et effectue les instructions de la recette avec ceux-ci pour obtenir le plat.

Licence Ouverte Etalab 2.0

▸ Définitions 3.1

– Définir une fonction veut dire écrire les instructions constituant cette fonction (c'est-à-dire, en python, le `def` et tout ce qui est *indenté* ensuite).

– Pour exécuter les instructions constituant une fonction, il faut avoir recours à une instruction qui porte le nom d'appel à la fonction.

▸ Remarque

Attention : sans appel, aucune instruction n'est exécutée !!

III.2. Appel d'une fonction

▸ Algorithme 3.1

Il existe deux cas de figure pour l'appel d'une fonction :

– Premier cas : on veut utiliser le résultat de la fonction dans des instructions ultérieures. Dans ce cas, il est nécessaire de `stocker` le résultat de la fonction dans une `variable`.

En `pseudo-code` :

PseudoCode
```
var ← nom_fct(params)
```

En python :

Code Python
```
var = nom_fct(params)
```

– Deuxième cas : on veut juste afficher le résultat de la fonction (pour tester celle-ci par exemple).

En `pseudo-code` :

PseudoCode
```
Afficher : nom_fct(params)
```

En python :

Code Python
```
print(nom_fct(params))
```

▸ Exemple

On peut vérifier dans la console python :

```
🐍 Début de la console Python 🐍
>>> # appel de la fonction min (premier cas)
>>> a = min(5, 42)
>>> print(a)
5
>>>
>>> # appel de la fonction min (deuxième cas)
>>> print(min(5, 42))
5
>>> min(5, 42)
5
🐍 Fin de la console Python 🐍
```

III.3. Appel d'une procédure

▸ Algorithme 3.2

Pour appeler une procédure, on utilise la syntaxe suivante :

En `pseudo-code` :

PseudoCode
```
nom_proc(params)
```

En python :

Code Python
```
nom_proc(params)
```

▸ **Exemple 3.1**

On va définir et appeler une procédure `affiche_somme` en python :

Code Python

```python
# Définition :
def affiche_somme(x,y) :
    print(f"{x}+{y} vaut {x+y}")
```

Dans ce cas, il n'y a pas besoin de `print` dans un script ou dans la console :

```
Début de la console Python
>>> a = affiche_somme(45, 27)
45+27 vaut 72
>>> print(a)
None
>>>
>>> affiche_somme (45, 27)
45+27 vaut 72
>>> print(affiche_somme (45, 27))
45+27 vaut 72
None
Fin de la console Python
```

En cas d'affichage non voulu de `None`, vérifiez que vous n'avez pas fait appel à une procédure avec la méthode d'appel d'une fonction !!

III.4. Aide d'une fonction ou d'une procédure

▸ **Idée**

En python, on commence en général la fonction par une ligne de documentation écrite entre triples guillemets (les docstrings). Cette ligne n'est pas obligatoire, mais documenter ses fonctions est une bonne habitude à prendre.

Il suffit dans ce cas d'utiliser la commande `help` pour afficher l'aide située dans les docstrings.

▸ **Algorithme 3.3**

On peut reprendre la fonction `cube` et luis rajouter une aide :

Code Python

```python
def cube(x) :
    """Retourne le cube du réel x"""
    res = x**3
    return res
```

Et dans ce cas la console permettra d'afficher l'aide :

```
Début de la console Python
>>> help(cube)
Help on function cube:

cube(x)
    Retourne le cube du réel x
Fin de la console Python
```

IV. Variables locales, variables globales

IV.1. Variables locales

▸ **Définition 4.1**

Lorsque des variables sont définies à l'intérieur d'une procédure ou d'une fonction, elles ne sont accessibles que dans cette dernière. On dit que ce sont des **variables locales**.
Les paramètres d'une fonction sont également des variables locales.

▸ **Algorithme 4.1**

Par exemple, la variable `res` présente dans la fonction `cube` est locale, elle n'existe *plus* une fois la fonction appelée.

```
Début de la console Python
>>> cube(11)
1331
>>> res
Traceback (most recent call last):
  File "<input>", line 1, in <module>
NameError: name 'res' is not defined
Fin de la console Python
```

IV.2. Variables globales

▸ **Définition 4.2**

Les variables définies à l'extérieur d'une procédure ou d'une fonction sont appelées **variables globales**.
Elles sont accessibles à l'intérieur d'une fonction si une variable locale de même nom n'existe pas.

▸ **Algorithme 4.2**

Par exemple, si on définit une variable `pi_approx`, on peut définir une fonction permettant de retourner le périmètre d'un cercle de rayon donné.

Code Python

```python
pi_approx = 3.14
def perimetre(R) :
    res = 2*R*pi_approx
    return res
```

On peut vérifier que la variable `pi_approx` est correctement utilisée dans la fonction.

```
Début de la console Python
>>> perimetre(4)
25.12
Fin de la console Python
```

▸ **Remarque**

Il est également possible de spécifier qu'une variable est globale, directement dans le corps d'une fonction, grâce à la commande `global`.

Code Python

```python
def aire(R) :
    global resultat_aire
    resultat_aire = 3.14*R**2
    return resultat_aire
```

```
Début de la console Python
>>> aire(10)
314.0
>>> resultat_aire
314.0
>>>
>>> aire(25)
1962.5
>>> resultat_aire
1962.5
Fin de la console Python
```

V. Fonctions prédéfinies

V.1. Définitions

▸ **Définitions 5.1**

En python, il existe un très grand nombre de fonctions prédéfinies. Il n'est pas pratique de charger toutes ces fonctions en mémoire. Une grosse partie d'entre elles sont donc stockées dans des fichiers appelés **modules** qu'il faut importer en mémoire avant utilisation.

V.2. Modules

▸ **Algorithme 5.1**

Pour charger un module en mémoire, on utilise l'instruction `import`.

▸ **Exemples 5.1**

Pour utiliser la fonction `randint` du module `random`, on peut utiliser l'instruction `import` de plusieurs façons différentes :

— Méthode 1 : en important uniquement la fonction voulue depuis (`from` en anglais) le module dans l'espace de noms principal :

```
Début de la console Python
>>> from random import randint
>>> # utilisation
>>> randint(0, 10) # ou print(randint(0, 10)) dans un script
0
Fin de la console Python
```

— Méthode 2 : en important toutes les fonctions du module dans l'espace de noms principal :

```
Début de la console Python
>>> from random import *
>>> # utilisation
>>> randint(0, 10) # ou print(randint(0, 10)) dans un script
3
Fin de la console Python
```

▸ Remarque

On peut également, au cas où plusieurs fonctions porteraient le même nom, charger un module avec un *préfixe* de chargement :

— Méthode 3 : en important toutes les fonctions du module dans un sous-espace de l'espace de noms principal ; et dans ce cas pour utiliser les fonctions du module, il faut alors spécifier l'espace de noms dans lequel elles ont été chargées ; et par défaut, ce nom est le nom du module :

```
🐍 Début de la console Python 🐍
>>> import random
>>> # utilisation
>>> random.randint(0, 10) # ou print(random.randint(0, 10)) dans un script
2
🐍 Fin de la console Python 🐍
```

— Méthode 4 : en important toutes les fonctions du module dans un sous-espace de l'espace de noms principal, pour lequel on choisit l'espace de noms :

```
🐍 Début de la console Python 🐍
>>> import random as bob
>>> # utilisation
>>> bob.randint(0, 10) # ou print(bob.randint(0, 10)) dans un script
10
🐍 Fin de la console Python 🐍
```

▸ Algorithme

Pour obtenir la liste des fonctions contenues dans un module, on utilise la commande help suivi du nom du module entre guillemets.
Par exemple, en *ligne de commandes* (ou dans la console), on aura la sortie (tout n'est pas affiché !) suivante :

```
Terminal
C:\Users\user>python
Python 3.11.5
Type "help", "copyright", "credits" or "license" for more information.
>>> help("random")
Help on module random:

NAME
    random - Random variable generators.

MODULE REFERENCE
    https://docs.python.org/3.11/library/random.html
    The following documentation is automatically generated from the Python
    source files. It may be incomplete, incorrect or include features that are
    considered implementation detail and may vary between Python
    implementations. When in doubt, consult the module reference at the
    location listed above.

DESCRIPTION
    bytes
    -----
        uniform bytes (values between 0 and 255)

    integers

-- Suite  --
```

▸ À savoir

Le module `random` et sa fonction `randint` seront régulièrement utilisés et sont à connaître pour l'examen !