Fonctions et variables

1 - Fonctions, paramètres, arguments

1.1 Fonction : notion (rappel)

ENTREE(S) ⇒ Fonction ⇒ SORTIE

1.2 Fonction native (rappel)

C'est une fonction présente de base dans Python.

1.3 Fonction personnelle (rappel)

Voir Python 1.4.7 au besoin.

A - Déclarer une fonction : stocker des instructions.

```
1  def double(x) :          # Déclaration
2      return 2 * x
3
4  n = double(12)           # Appel
```

Le mot-clé **return** indique ce que la fonction va renvoyer.

B - Lancer un appel de fonction

Sur l'exemple, l'appel se fait LIGNE 4.

Déroulé

L1(déclaration) L4(appel)-L1-L2-L4(reponse)

- L1 : déclaration d'une fonction `double()` qui va recevoir une donnée qu'on placera dans une variable **x**.
- L4 (partie droite) : appel à `double()` en lui envoyant 12.
- L1 : On place 12 dans **x**.
- L2 : la fonction répond 24
- L4 (partie gauche) : on stocke 24 dans **n**.

1.4 Fonction plus complexe (rappel)

Une fonction peut contenir des boucles, des instructions conditionnelles, ect...

1.5 Une déclaration pas à pas

```
def                  # mot-clé def
def nouvelle         # nom de la fonction
def nouvelle()       # rajout parenthèses
def nouvelle(note)   # rajout des paramètres
def nouvelle(note):  # n'oubliez pas ce :
    les instructions # les instructions tabulées
    return truc      # on renvoie la réponse
```

1.6 return : on peut utiliser directement une expression

Lorsque l'interpréteur rencontre le mot-clé return, il réalise deux choses :

- Il évalue l'expression située derrière lui.
- Il renvoie la réponse à l'endroit d'où l'appel a été lancé.

Deux choix de conception : répondre indirectement ou directement.

```
1  def fois2_via_variable(x) :
2      resultat = x * 2
3      return resultat
4
5  def fois2_via_expression(x) :
6      return x * 2
```

1.7 Paramètre et argument

Paramètre : variable locale servant à récupérer l'une des entrées. Son nom figure dans la déclaration de la fonction. On parle également de paramètre formel.

Argument : l'une des entrées qu'on envoie lors d'un appel. On parle également de paramètre effectif.

```
1  # d et u sont les paramètres déclarés
2  def valeur(d, u) :
3      resultat = d*10 + u
4      return resultat
5
6  # 5 et 3 sont les arguments de l'appel^^I
7  valeur(5, 3)
```

(p)aramètre <-> (d)éclaration.

(A)rgument <-> (A)ppel.

2 - Documentation : expliquer comment utiliser

2.1 COMMENTAIRES(rappel)

Les commentaires sont **destinés à un lecteur humain** et ils visent à **rendre le code interne facile à comprendre**.

Voir Python Programme : 7.1.2

2.2 DOCUMENTATION

La documentation doit expliquer **comment utiliser la fonction sans provoquer d'erreur**.

En Python, pour trouver la documentation, il suffit d'utiliser une fonction native : la fonction `help()`.

```
>>> help(len)
  Help on built-in function len in module builtins:

len(obj, /)
    Return the number of items in a container.
```

La documentation contient des informations sur

— Le but de cette fonction
— le type et le nombre d'arguments à lui envoyer ;
— le type et le contenu de la réponse.

2.3 DOCUMENTATION rapide

Signature d'une fonction

```
addition(int, int) -> int
```

Prototype d'une fonction

```
addition(n1:int, n2:int) -> int
```

Spécification Spécifier une fonction consiste à donner :

— Le PROTOTYPE qui fixe les TYPES ;
— Une DOCSTRING (Documation string) qui complète les informations

La spécification fournit donc toutes les informations nécessaires à sa bonne utilisation.

```
1  def addition(n1 :int, n2 :int) -> int :
2      """Renvoie la somme de n1 et n2"""
3      return nbr1 + nbr2
```

2.4 DOCUMENTATION longue

On utilise un string multi-ligne plutôt que de noter les informations de type directement dans le prototype.

```
1  def vitesse(distance, duree) :
2      """Renvoie la vitesse connaissant la distance parcourue et la durée
3
4      :: param distance(int) :: la distance en m
5      :: param duree(int)    :: la durée NON NULLE en s
6      :: return (float)      :: la vitesse en m.s-1
7
8      """
9      return distance / duree
```

3 - Return, ou pas Return, sortez !

3.1 Sortie immédiate avec return

Dès que la fonction rencontre un `return` :

— elle évalue l'expression fournie puis
— elle renvoie sa réponse, puis
— son espace local des noms est détruit définitivement.

3.2 `return None`

A - Le cas du return absent

Une fonction Python renvoie au moins la valeur `None` lorsqu'elle se termine sans avoir rencontré return. On parle parfois de **procédure** pour désigner une fonction qui ne répond rien. Les procédures n'existent donc pas vraiment en Python.

La ligne 4 ci-dessous est donc totalement facultative :

```
1  def division(x :int, y :int) -> None :
2      mon_petit_calcul = x // y
3      return None
```

B - Attention au stockage de la réponse...

On peut mal utiliser les fonctions qui renvoient None. Le cas typique est la méthode `append()` qui permet

de rajouter une case à la fin d'un tableau Python : elle modifie le tableau et renvoie... **None**.

```
# Exemple de bonne utilisation

>>> t = [10, 20, 30]
>>> t.append(100)
>>> t
[10, 20, 30, 100]

# Exemple de mauvaise utilisation

>>> t = [10, 20, 30]
>>> t = t.append(100)  # TRES MAL
>>> t
# Pas de réponse car t contient None
```

Attention : si on utilise **append()**, il ne s'agit plus d'un tableau statique mais d'un **tableau dynamique**. Nous les utiliserons plus tard dans l'année.

C - None et les booléens

Lorsqu'on veut évaluer un contenu sous forme de booléen, **None** est considéré comme le cas nul ou vide.

```
>>> bool(None)
False
```

Mais attention : **None** n'est identique à **False**.

```
>>> a = None
>>> a == False
False

>>> bool(a) == False
True
```

4 - Compléments sur les paramètres - hors programme

Aucune connaissance à avoir à part connaître leurs existences. Aller voir sur le site pour les infos. Rien dans le DS.

4.1 Paramètres par défaut

```
def aleatoire(fin, debut=1):
```

Si on lance un appel **aleatoire(12)**, le paramètre **fin** référencera ce 12 mais **debut** ne récupère rien. Par défaut, il référencera donc 1.

Par contre, sur un appel **aleatoire(12, 3)**, on aura le fonctionnement normal : **fin** référence 12 et **debut** référence 3.

4.2 Paramètres nommés ou positionnels

```
def aleatoire(fin, debut):
```

On peut lancer un appel en fournissant l'ordre qu'on veut pourvu qu'on nomme les paramètres : l'appel **aleatoire(debut=3, fin=12)** est compris par Python qui placera les bonnes arguments dans les bons paramètres.

On peut empêcher l'utilisateur d'utiliser les paramètres nommées sur certains paramètres en plaçant ces paramètres devant un slash :

```
def aleatoire(fin, /, debut=1):
```

Ici, on empêche alors fin d'être utilisé de façon nommé : il faut nécessairement qu'il soit le premier lors de l'appel.

4.3 Documentation des types construits

```
def f(x:list[int]):
def f(x:dict[(str, int)]):
```

Première exemple : on signale qu'on veut un tableau d'entier.

Deuxième exemple : on signale qu'on veut un dictionnaire dont les clés sont des strings et les valeurs associées sont des entiers.