Fonctions et variables

8.1

Fonctions, paramètres, arguments

8.1.1 Fonction : notion (rappel)

ENTREE(S) ⇒ Fonction ⇒ SORTIE

8.1.2 Fonction native (rappel)

C'est une fonction présente de base dans Python.

```
>>> max(10, 0, 50, 40)
50
```

8.1.3 Fonction personnelle (rappel)

Voir Python 1.4.7 au besoin.

A - Déclarer une fonction : stocker des instructions.

On écrit dans l'ordre

- le mot-clé **def**.
- le nom de la fonction
- les variables d'entrée (parenthèses et virgules)
- le caractère **:**
- les instructions du bloc **tabulé**

```
1  def double(x) :
2      return 2 * x
```

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

B - Lancer un appel de fonction : fournir les entrées et attendre la réponse.

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

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

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**.

8.1.4 Fonction plus complexe (rappel)

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

8.1.5 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
```

8.1.6 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 valeur trouvée à l'entité qui a lancé l'appel initialement.

Deux choix de conception : répondre indirectement ou directement.

```
def fois2_via_variable(x) :
    resultat = x * 2
    return resultat

def fois2_via_expression(x) :
    return x * 2
```

8.1.7 Paramètre et argument

Paramètre : nom qu'on donne à une variable locale servant à récupérer l'une des entrées. Son nom figure donc dans la déclaration de la fonction. On parle également de paramètre formel. (p)aramètre <-> (d)éclaration.

Argument : nom qu'on donne à l'une des entrées qu'on envoie lors d'un appel de fonction. On parle également de paramètre effectif. (A)rgument <-> (A)ppel.

```
# d et u sont les paramètres déclarés
def valeur(d, u) :
    resultat = d*10 + u
    return resultat

# 5 et 3 sont les arguments de l'appel^^I
valeur(5, 3)
```

8.2

Documentation : expliquer comment utiliser

8.2.1 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

8.2.2 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 (les ENTREES) et
- le type et le contenu de la réponse (la SORTIE).

8.2.3 DOCUMENTATION rapide

On donne juste le prototype et une phrase.

Signature d'une fonction

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

Prototype d'une fonction Prototyper une fonction consiste à rajouter les noms des paramètres à la signature.

```
addition(nombre1:int, nombre2:int) -> int
```

Spécification Spécifier une fonction consiste à donner :

- Le PROTOTYPE qui fixe la SYNTAXE et le TYPAGE
- Une description nommée DOCSTRING qui fixe clairement la SEMANTIQUE

Si on fournit la spécification d'une fonction, on fournt donc toutes les informations nécessaires à sa bonne utilisation.

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

8.2.4 DOCUMENTATION longue

On fait la "même" chose mais en utilisant 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
```

8.3

Return, ou pas Return, sortez !

8.3.1 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.

8.3.2 `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 :

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

B - Attention au stockage de la réponse...

Il arrive souvent qu'on utilise mal de telles fonctions qui renvoient None si on a lu trop vite la documentation.

Le cas typique vient de 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
```

Sur le deuxième exemple, **t** contient maintenant None.

8.4

Compléments hors programme

Pour l'instant, on vous demande simplement de vous souvenir à quoi correspondent ces notions. Aller voir sur le site si vous n'en avez aucun souvenir. Lors des évaluations, vous n'aurez jamais à taper ce type de code mais simplement à le comprendre.

8.4.1 Paramètres par défaut

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

8.4.2 Paramètres nommés ou positionnels

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

8.4.3 Documentation des types structurés

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

8.5

Fonction et conteneur MUABLE

Une fonction peut ACCEDER à une variable globale

Exemple : la fonction va lire l'adresse de **d** et, une fois sur place, on lit la valeur de la clé.

```
def actualiser_vue() :
    ...
    afficher_vaisseau(d['x'], d['y'])
```

Une fonction ne peut PAS MODIFIER UNE VARIABLE GLOBALE

Le **x** global ne pourra pas modifier de cette façon :

```
x = 10

def mise_a_zero() :
    x = 0    # Juste un x local
```

Une fonction peut MODIFIER LE CONTENU d'un CONTENEUR MUABLE GLOBAL

Si la variable globale fait référence à un type structuré, la fonction peut donc accéder à la localisation de la structure. Une fois sur place, elle pourra donc MODIFIER LE CONTENU de la structure.

```
def doubler() :
    ...
    d['x'] = d['x'] * 2   # OK
```