Fonctions et variables

1 - Fonctions : les bases

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 Définition de fonction (rappel)

Voir Python 1(4.7) si besoin.

[DEF] **Définir une fonction** : stocker des instructions pour les utiliser plus tard.

Le mot-clé **def** indique le début de la définition. Le mot-clé **`return`** indique la sortie à renvoyer.

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

1.4 Appel de fonction (rappel)

[DEF] **Lancer un appel de fonction** : utiliser une fonction en lui fournissant des entrées puis attendre pour récupérer la réponse qu'elle fournira.

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

Déroulé

L1(définition) L4(appel)-L1-L2-L4(retour)

- L1 : définition 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`.

1.5 Fonction plus complexe (rappel)

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

2 - Fonctions : paramètres, arguments

2.1 return : mécanisme

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

- Il évalue l'expression située derrière lui ;
- Il quitte définitivement cet appel ;
- Il renvoie la sortie à l'endroit d'où a été lancé l'appel initialement.

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
```

2.2 Paramètre et argument

[DEF] **Argument** : nom donné à une entrée qu'on fournit à la fonction lors d'un appel. On parle également de paramètre effectif.

[DEF] **Paramètre** : variable locale présente sur la première ligne de la définition. Lors de l'appel, les paramètres servent à stocker les arguments reçus. Un paramètre n'a d'existence que pendant l'exécution d'un appel. Il n'existe ni avant l'appel, ni après l'appel puisqu'il est une variable locale.

```
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.

3 - Documentation : expliquer comment utiliser

3.1 COMMENTAIRES(rappel)

Expliquer le fonctionnement

[DEF] Les **commentaires** sont **destinés à un lecteur humain** et ils visent à **à fournir des explications sur le fonctionnement des parties les plus difficiles** du code interne.

Voir Python Programme 7(1.2)

3.2 DOCUMENTATION

Expliquer l(utilisation

[DEF] La **documentation** explique à l'utilisateur comment **lancer un appel** à la fonction, sans provoquer d'erreur.

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.

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.
```

3.3 DOCUMENTATION rapide

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

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

[DEF] La **spécification** d'une fonction regroupe le **prototype** et une **explication** textuelle ou mathématique de ce qu'on a le droit de demander à la fonction et ce qu'elle va alors répondre.

En Python, la phrase doit être un string multiligne situé directement sous la déclaration (donc avec trois guillemets d'ouverture et de fermeture).

On l'appelle DOCSTRING, pour Documentation String.

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

3.4 DOCUMENTATION longue

On utilise un DOCTRING 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
```

3.5 contrat de confiance

[DEF] **précondition** : condition supplémentaire sur l'entrée que l'utilisateur doit respecter (en plus du simple type) lorsqu'il lance un appel à cette fonction.

[DEF] **postcondition** : condition supplémentaire sur la sortie que le développeur s'engage à respecter (en plus du simple type de la sortie). Par contre, le créateur de la fonction n'engage sa responsabilité que si les types et préconditions sur les entrées sont bien validées.

En informatique, le contrat de confiance entre les utilisateur et concepteurs est basé sur ces principes :

- SI l'utilisateur respecte les préconditions, ALORS le concepteur garantit les postconditions.
- SI l'utilisateur ne respecte pas les préconditions, ALORS le concepteur ne garantit rien sur la réponse.

On résume cela par cette simple équation :

$PRECONDITONS \rightarrow POSTCONDITON$

4 - Return, ou pas Return, sortez !

4.1 Sortie immédiate avec return

Dès que la fonction rencontre un **return** :

— elle évalue l'expression fournie ;
— elle renvoie sa réponse à l'endroit d'où l'appel a été fait ;
— son espace local des noms est détruit définitivement.

4.2 return None

[DEF] **None** correspond à une absence d'informations. Elle est la seule valeur possible du type natif **NoneType**.

Lors d'un transtypage vers bool, la valeur None est considérée comme le cas nul ou vide.

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

A savoir Une fonction Python renvoie au moins la valeur **None** lorsqu'elle se termine sans avoir rencontré un autre **return**.

[DEF] Dans certains langages de programmation, une **procédure** est le nom des routines qui agissent mais ne renvoient réellement aucune valeur. Python ne possède pas de véritables procédures mais, par abus de langage, on appelle parfois "procédure" une fonction Python qui renvoie uniquement et toujours **None**.

print et return

HYPER IMPORTANT : attention à la différence entre les fonctions qui affichent et les fonctions qui renvoient une valeur.

Si on vous demande de créer une **fonction qui renvoie**, il faut utiliser un **return**. On pourra stocker la valeur.

```
def fois2(v:int) -> int:
    """Renvoie le double de la valeur envoyée"""
    return v * 2

rep = fois2(10)
```

Si on vous demande de créer une **fonction qui affiche**, il faut utiliser **print()**. On ne pourra pas récupérer une valeur via cette fonction : elle ne fait qu'afficher à l'écran.

```
def fois2(v:int) -> int:
    """Affiche le double de la valeur envoyée"""
    print(v * 2)

fois2(10)
```

L'erreur terrible

NE FAITES JAMAIS CELA : vous laissez entendre que rep contient parfois quelque chose. C'est faux.

```
def fois2(v:int) -> int:
    """??? le double de la valeur envoyée"""
    print(v * 2)

rep = fois2(10)
```

NE FAITES JAMAIS CELA : vous laissez entendre que rep contient parfois quelque chose. C'est faux.

5 - Compléments sur les paramètres - hors programme

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

5.1 Paramètres par défaut

5.2 Paramètres nommés ou positionnels

5.3 Documentation des types construits

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


CC BY NC SA