Programmation 2

Slides modified from https://github.com/UGE-IGM/courspython

Chapitre 4 - Fonctions

Dans ce chapitre vous allez :

1. prendre conscience de toutes les fonctions prédéfinies du langage Python que nous avons déjà utilisées ;
2. apprendre ce qu'est une fonction ;
3. apprendre à effectuer un appel de fonction en Python ;
4. apprendre à écrire correctement une fonction en Python réalisant une tache unique prédéfinie ;
5. apprendre à transformer un bout de programme en une fonction ;
6. apprendre à comprendre la différence entre la saisie d'un utilisateur et le passage d'un paramètre à une fonction ;
7. apprendre à comprendre la différence entre une valeur de retour d'une fonction et un affichage ;
8. apprendre à décrire l'espace de nom global d'un programme ;
9. apprendre à décrire les espaces de nom locaux lors de l'exécution d'une fonction ;

10. comprendre de manière détaillée comment fonctionne un appel de fonction ;
11. comprendre l'intérêt d'écrire de la documentation ;
12. apprendre à écrire des doctests pour tester vos fonctions ;
13. apprendre à lancer une série de doctests automatiquement.

I Introduction

Dans le cours sur les boucles, nous avons écrit le code suivant permettant d'effectuer une saisie par l'utilisateur d'un entier positif :

```
nb = int(input(’Donnez moi en entier positif ou nul : ’))
```

```
while nb < 0: print('Erreur de saisie.') nb = int(input('Donnez moi en entier positif ou nul : '))
```

Nous avons depuis réécrit ce code de nombreuses fois, et aimerions ne pas avoir à l'écrire à chaque fois...
En programmation, l'objectif des fonctions est justement de pouvoir :

- créer de nouvelles fonctionnalités ;
- réutiliser à souhait ces fonctionnalités. Plus précisemment, en programmation, une fonction est un morceau de programme, portant en général un **nom**, acceptant zéro, un ou plusieurs **paramètres** et produisant un **résultat** (un calcul, un affichage, etc.). Des exceptions existent, mais la forme la plus courante d'une fonction est donc proche de celle d'une fonction mathématique.

On notera immédiatement que l'utilisation des fonctions améliore les aspects suivants du code :

- Lisibilité : séparer une partie du programme (par exemple un gros calcul compliqué)
- Modularité : réutiliser le même code plusieurs fois (évite de recopier

le code)

- Généricité : changer la valeur des paramètres (même calcul mais avec différentes valeurs de départ)

II Fonctions prédéfinies et bibliothèque standard

En Python, il existe un grand nombre de fonctions prédéfinies, que nous avons déjà utilisées, par exemple :

- `int(obj)` : 1 paramètre, 1 résultat. Reçoit en paramètre un objet (par exemple `str` ou `float`), essaie de le transformer en entier et renvoie l'entier obtenu.

In [1]:
```
int("34")
```

Out[1]:

34

Ici, l'interpréteur a directement affiché la valeur de l'expression `int("34")`. On peut aussi l'affecter à une variable qu'on réutilise ensuite :

In [2]:
```
n = int("34")
n + 5
```

Out[2]:

39

- `len(obj)` : 1 paramètre, 1 résultat. Reçoit un objet (par exemple de type `str`) et renvoie sa longueur.

In [3]:
```
len("34")
```

Out[3]:

2

Il y en a beaucoup d'autres, comme `print, input, float, str....`

Ces fonctions sont appelées *prédéfinies* (ou *built-in*). Il n'est bien sûr pas question de les connaître toutes par coeur. La totalité des fonctions prédéfinies de Python est **documentée**, en particulier sur **cette page**.

Il existe également de nombreux *modules* officiels (par exemple le module `random`), c'est-à-dire des bibliothèques de fonctions, de types et d'objets, dont on trouvera la description **ici**. Voici un exemple de fonction du module `random` :

- `randint(mini, maxi)` : 2 paramètres, 1 résultat. Reçoit deux nombres, et renvoie un entier aléatoire compris entre ces deux nombres (inclus).

In [10]:
```
from random import randint

randint(1, 34)
```

Out[10]:

7

Vérifier vos connaissances

A ce stade, vous devez être capable :

- de citer une dizaine de fonctions prédéfinies (ou *built-in*) du langage Python ;
- de citer toutes les fonctions prédéfinies utilisées dans un programme.

EXERCICE 1 :

Dans le code suivant, citer toutes les fonctions utilisées.

In [ ]:
```
somme = ""
        saisie = None #variable en fait pas réellement initialisée
while saisie != "stop":
    saisie = input("Donnez moi un entier : ")
    if saisie != "stop":
         somme += saisie
print("Vous avez tapez " + str(len(somme)) + " caractères")
```

III Fonctions définies par l'utilisateur

1) Définition d'une fonction

En informatique, une fonction est une portion de code représentant un sous-programme qui effectue une tâche ou un calcul relativement indépendant du reste du programme.
Une fonction a une entrée, les **paramètres** qu'on lui donne ; elle exécute un traitement sur ces arguments ; enfin, elle retourne généralement une valeur, **la valeur de retour**.
Le principal intérêt des fonctions est de définir une portion de code réutilisable à différents endroits dans le programme. En ce sens, l'utilisation de fonctions permet de :

- condenser le code du programme, sans avoir à recopier des parties simillaires (Cf. le problème sur les dates...)
- rendre plus lisible un programme, en identifier des briques

élémentaires

2) Syntaxe Python pour définir une fonction

Pour définir une nouvelle fonction ayant `n` paramètres, on utilise la syntaxe suivante :

```
# ligne suivante : en-tête de fonction
def nom_fonction(param_1, ..., param_n):
    """ Partie concernant la documentation de la fonction """
    # corps de la fonction
    # utilisant param_1 à param_n
    ...
    # peut renvoyer un résultat :
    return resultat
```

Ici, les paramètres de la fonction sont `param_1`, `param_2`, ..., `param_n`. La valeur de retour est `resultat`.

Exemple 1 :

In [1]:
```
def affiche(msg):
    """Affiche le msg passé en paramètre encadré par des étoiles"""
    print("*" * (4 + len(msg)))
    print("* " + msg + " *")
    print("*" * (4 + len(msg)))
```

In [2]:
```
affiche("Bonjour maman")
```

```
******************
* Bonjour maman *
******************
```

3) Appel de fonctions

Une fois définie, la fonction `nom_fonction` peut être utilisée dans le code (on parle d'un **appel**) en indiquant entre parenthèses ses paramètres séparés par des virgules :

```
# définition de fonction
def nom_fonction(param_1, ..., param_n):
    """Documentation de la fonction""" # cf plus loin dans le cours
    Code
    ...

# reste du programme
...

# appel de la fonction :
une_var = nom_fonction(param_1, ..., param_n) #si la fonction a une
valeur de retour (cf plus bas)
    #ou
nom_fonction(param_1, ..., param_n) #si la fonction n'a pas de valeur de
retour
```

Exemple 1 :

Nous venons de définir la fonction *affiche(msg)* permettant d'afficher un message entouré par des étoiles. Voyons comment l'utiliser :

In [13]:
```
def affiche(msg):
    """Affiche le msg passé en paramètre encadré par des étoiles"""
    print("*" * (4 + len(msg)))
    print("* " + msg + " *")
    print("*" * (4 + len(msg)))

affiche("Bonjour à tous")
```

```
******************
* Bonjour à tous *
******************
```

Vérifier vos connaissances

A ce stade, vous devez être capable :

- d'expliquer à quelqu'un ce qu'est une fonction en informatique ;
- de définir une fonction dans un programme que vous écrivez ;
- de commencer à transformer un bout de programme en une fonction ;
- de faire un appel de fonction simple.

4) Valeur de retour d'une fonction

On a vu que la fonction `len` permet de renvoyer la longueur d'une chaine de caractères.
De manière générale, une fonction renvoye un résultat. Cela se fait en utilisant le mot clé `return`. L'exécution d'un `return` provoque la sortie de la fonction : on ne peux exécuter qu'un seul `return` par bloc.

In [ ]:
```
def carre(x):
    """Fonction élévant au carré son paramètre

    :param x: int
    :return value: int
    """
    return x * x
```

In [ ]:
```
x = 2
x_2 = carre(x)
print("x^2 = ", x_2)
```

Néanmoins, il n'est pas nécessaire d'avoir un `return` dans une fonction. Si l'exécution arrive à la dernière instruction du corps de la fonction sans rencontrer d'instruction `return expr`, alors la valeur de retour par défaut est `None`. On a le même comportement si on écrit un `return` seul. On parlera alors de **fonction sans valeur de retour**. Par exemple :

In [15]:
```
def affiche(msg):
    """Affiche le msg passé en paramètre encadré par des étoiles"""
    print("*" * (4 + len(msg)))
    print("* " + msg + " *")
    print("*" * (4 + len(msg)))

a = affiche("Bonjour à tous")
print("\nValeur de retour de la fonction affiche : ", a)
```

```
******************
* Bonjour à tous *
******************

Valeur de retour de la fonction affiche :  None
```

Exemple 1 : Traduction d'un programme en fonction

Dans le cours sur les boucles, nous avons écris le programme suivant permettant de calculer le plus grand entier inférieur à la racine carrée de l'entier `N = 142` :

```
N = 142
n = 0
while n*n <= N:
    n += 1
print(n - 1)
```

Pour modifier le comportement du programme, il suffit de modifier la variable `N`. On peut donc écrire une fonction `racine_carre_entiere` prenant un paramètre `N` comme suit :

In [ ]:
```
def racine_carre_entiere(N):
    n = 0
    while n*n <= N:
        n += 1
    return n - 1
```

Exemple 2 : une fonction sans paramètre simulant du hasard

Bien sur, une fonction peut ne pas avoir de paramètre. En général, une telle fonction a toujours le même comportement.

Dans l'exemple suivant, on crée une fonction qui simule un lancer de dé. Elle n'aura donc a priori pas le même comportement pour deux exécutions distinctes.

In [25]:
```
from random import randint

def lance_de() :
    """Simule un lancé de dé à 6 faces"""
    return randint(1,6)

compteur = 1
while lance_de() != 6:
    compteur = compteur + 1
print('Obtenu un 6 en', compteur, 'jets de dé.')
```

```
Obtenu un 6 en 6 jets de dé.
```

Remarque :
Les nombres générés aléatoirement par Python ne sont pas réellement aléatoire : ils proviennent d'un **générateur pseudo-aléatoire**. Cela signifie que le hasard est simulé à partir d'une unique valeur, appelée la **graine** (**seed** en anglais).
Si cette graine n'est pas précisé explicitement, elle est calculé à partir de l'heure actuelle pour le système. C'est notre cas ici : cela explique que la fonction `lance_de()` ne produit pas toujours la même série de valeur de retour.

Par opposition, en initialisant la graine au moment de l'`import`, la série de valeur de `lance_de` sera identique.

In [ ]:
```
from random import randint
from random import seed

#Première génération aléatoire de trois nombres
graine = 30
print ("Trois nombres aléatoires crée à partir de la graine", graine)
seed(graine)
print ("Premier nombre :", randint(1, 6))
print ("Second nombre :", randint(1, 6))
print ("Troisième nombre :", randint(1, 6))

print() #pour sauter une ligne

#Seconde génération aléatoire de trois nombres
graine = 30
print ("Trois nombres aléatoires crée à partir de la graine", graine)
seed(graine)
print ("Premier nombre :", randint(1, 6))
print ("Second nombre :", randint(1, 6))
print ("Troisième nombre :", randint(1, 6))
```

Exemple 3 : une fonction ayant des paramètres et une valeur de retour

On souhaite re-écrire une fonction déterminant le maximum entre deux valeurs (souvent de type `int`, mais pouvant aussi être de type `string` par exemple).

Raisonnablement, celle-ci doit avoir deux paramètres. Elle devra aussi renvoyer une valeur (l'un des paramètres)

In [26]:
```
# fonction ayant deux paramètres
def maximum(a, b):
    """Renvoie le maximum des deux paramètres"""
    if a > b:
        return a
    else:
        return b

nb1 = int(input("Saisissez un nombre : "))
nb2 = int(input("Saisissez un nombre : "))

# On appelle la fonction et on garde le résultat dans c :
m = maximum(nb1, nb2)
print("Le max de", nb1, "et", nb2, "est", m)

# On peut aussi utiliser directement le résultat :
print("Le max de", nb1, "et", nb2, "est", maximum(nb1, nb2))
```

```
Saisissez un nombre : 3
Saisissez un nombre : 9
Le max de 3 et 9 est 9
Le max de 3 et 9 est 9
```

Vérifier vos connaissances

A ce stade, vous devez être capable :

- de transformer un bout de programme déjà écrit en une fonction ;
- d'écrire correctement une fonction en Python réalisant une tache prédéfinie.

Exercice 3 :

Au chapitre précédent, nous avons écrit le code suivant pour réaliser la saisie controllée par l'utilisateur d'un entier positif.

```
saisie = -1 #variable en fait pas réellement initialisée
while saisie < 0:
    saisie = int(input("Donnez moi un entier positif : "))
```

Utiliser ce code pour écrire une fonction `saisie_nb_positif()` demandant à l'utilisateur un entier positif, recommençant tant que le nombre saisie n'est pas positif et retournant finalement l'entier saisi.

Exercice 4 :

- Écrire une fonction `min(a, b)` renvoyant le plus petit des deux objets `a` et `b`.
- Écrire une fonction `max(a, b)` renvoyant le plus grand des deux objets `a` et `b`.
- Affecter successivement les valeurs `3` et `2`, puis `"toto"` et `"titi"` aux variables `a` et `b`. Que vaut alors `min(a, b)` et `max(a, b)` ?

Exercice 5 :

Écrire une fonction `factorielle(n)` renvoyant l'entier `n! = 1 * 2 * ... * n`.

EXERCICE 6 :

Ecrire une fonction `pgcd(a, b)` renvoyant le plus grand commun diviseur des entiers `a` et `b`. *On rapelle que l'algorithme pour calculer un p.g.c.d. a été vu au cours du TD 3 sur les boucles*

5) Composition de fonctions

Bien sur, lorsque l'on écrit une fonction, il est possible d'utiliser une autre fonction. Nous l'avons déjà fait dans la fonction `affiche` en utilisant la print prédéfinie `print`, ou encore dans la `trace_polygone` avec les fonctions `forward` et `left` du module `turtle`. On peut aussi utiliser nos propres fonctions, comme le montre l'exemple suivant :

In [ ]:
```python
def pgcd(a, b):
    """Fonction calculant le pgcd de deux entiers

    :param a: int
    :param b: int
    :return value: int
    """
    while a % b != 0:
        r = a % b
        a = b
        b = r
    return b

def premiers_entre_eux(a, b):
    """Fonction testant si deux entiers sont premiers entre eux

    :param a: int
    :param b: int
    :return value: bool
    """
    return pgcd(a, b) == 1

# programme principal :
m = int(input("Saisissez un nombre : "))
n = int(input("Saisissez un nombre : "))
o = int(input("Saisissez un nombre : "))
```

```
if (premiers_entre_eux(m, n)
    and premiers_entre_eux(n, o)
    and premiers_entre_eux(o, m)):
    print("tous premiers entre eux")
```

Vérifier vos connaissances

A ce stade, vous devez être capable :

- de comprendre la différence entre la saisie d'un utilisateur et le passage d'un paramètre à une fonction ;
- d'isoler dans une fonction les saisies utilisateurs réalisées dans un programme ;
- d'écrire des fonctions réalisant une seule tache.

2) Confusion entre retour et affichage

Les programmeurs débutants confondent aussi très souvent la notion de **valeur de retour** avec la notion de **valeur affichée**.
Imaginons que nous sommes en train de travailler sur un projet d'exploration de Neptune pour savoir si cette planète serait habitable par l'homme. Il nous faudrait donc d'écrire un programme utilisé par un robot explorateur analysant la composition de l'atmosphère sur place.
En particulier, il faudrait une fonction prenant comme argument l'analyse de l'atmosphère et renvoyant "Viable pour les Hommes" ou "Pas viable pour les Hommes".
Que se passerait-il si cela avait été programmé comme cela ?

In [27]:
```
# ATTENTION CECI EST INCORRECT, A NE PAS REPRODUIRE !!!

def analyse(atmosphere) :
    if atmosphere == "CO2":
        print("Viable pour les Hommes")
    else:
        print("Pas viable pour les Hommes")
```

NE SURTOUT PAS PROGRAMMER COMME ÇA :

- la valeur calculée n'est pas réutilisable : elle est uniquement affichée à l'écran.
- aucun commentaire n'a été écrit par le programmeur.

Evidemment, le programmeur sur Terre voit son écran. Il obtient donc d'une certaine manière la réponse à sa question. Mais, en réalité, un tel programme ne serait pas d'une grande aide pour les hommes restés sur Terre... Car l'affichage ne se fera pas face au programmeur, mais à plus de 1300 millions de kilomètres...
Autrement dit, le robot serait très fier d'afficher sur son torse contenant un petit écran le résultat demandé. *Mais, comment ferait-on sur Terre pour pouvoir expoloiter cette donnée ?*
Voici évidemment la bonne manière d'écrire ce programme :

In [ ]:
```
def analyse(atmosphere) :
    """Détermine les résultats finaux de l'analyse de l'atmosphère

    :return value: String
    """
    if atmosphere == "CO2":
        return "Viable"
    else:
        return "Pas viable"
```

Cette fois, il est possible de récupérer sur Terre le résultat via une fonction de transmission de donnée entre le robot et notre centre d'étude spatiale !

Moralité : Une fonction peut néanmoins contenir un appel à `print` s'il s'agit d'une fonction d'affichage *(et uniquement dans ce cas !)*
Convention : Une fonction d'affichage aura un nom commencant par `affiche`, par exemple `affiche_triangle`

In [ ]:
```
def affiche_triangle(n):
        """Affiche un triangle d'étoile de hauteur n
    :param n: int
    """
    for i in range(n):
        print("*" * (i + 1))
```

In [ ]:
```
affiche_triangle(6)
```

Vérifier vos connaissances

A ce stade, vous devez être capable :

- de comprendre la différence entre `return` et un affichage dans une fonction.

3) Un dernier exemple

Corrigeons maintenant le programme qu'un étudiant a écrit l'an dernier :

In [ ]:
```
# ATTENTION CECI EST INCORRECT, A NE PAS REPRODUIRE !!!
    def pgcd(a, b):
a = int(input())  # NON !
b = int(input())  # NON !
while a % b != 0:
    r = a % b
    a = b
    b = r
print("le pgcd est", b)  # NON !
```

NE SURTOUT PAS PROGRAMMER COMME ÇA :

- pour avoir des programmes facilement lisibles et maintenable dans le temps, une fonction se devrait d'effectuer une seule tache ; or ici, `pgcd` réalise trois taches : la saisie de `a` et `b`, le calcul du pgcd, puis l'affichage de la valeur du pgcd.
- les paramètres de la fonction, `a` et `b`, sont écrasés par une saisie de l'utilisateur : ils sont donc totalement inutiles.
- aucun commentaire n'a été écrit par le programmeur.

Voici la manière correcte d'écrire un programme calculant un pgcd en utilisant une fonction réalisant le calcul :

In [ ]:
```
# VERSION CORRECTE !!!

# Fonction(s) préliminaire(s)
def pgcd(a, b):
    """Fonction calculant le pgcd de deux entiers

    :param a: int
    :param b: int
    :return value: int
    """
    while a % b != 0:
        r = a % b
        a = b
        b = r
    return b

#Début du programme principal
a = int(input("Saisissez un nombre : "))
b = int(input("Saisissez un nombre : "))

print(pgcd(a, b))
```

Le véritable objectif de la fonction `pgcd` est de **calculer** le pgcd. On a donc retiré du corps de la fonction tout ce qui n'était pas lié au calcul (saisie des nombres + affichage du résultat). On a aussi rajouté un peu de documentation de la fonction.

Exercice 7 :

- Comment écrire un programme vérifiant si trois entiers sont premiers entre eux à l'aide de cette fonction ?
- Combien ce programme ferait-il de saisies ?
- Qu'afficherait ce programme ?

IV Sémantique d'un appel de fonctions et espaces de noms

Au moment d'un appel de fonction, un espace de noms local est créé. Il associe à chacun des paramètres la valeur de l'expression correspondante dans l'appel. Les variables locales sont également créées dans ces espace de noms.

1) Notion d'espace de noms

Un espace de noms est un ensemble de noms (de variables, de fonctions, ...) définies à un certain point d'un programme. L'ensemble de tous les noms connus à un point du programme peut être constitué de plusieurs espaces de noms superposés (du plus bas au plus haut) :

- espace de nom global contenant les variables (fonctions, classes...) définies directement dans le programme principal ;
- empilement des espaces de noms locaux des appels de fonction en cours, dans l'ordre chronologique, contenant chacun les paramètres de l'appel correspondant ainsi que les variables définies dans le corps de cette fonction (variables locales).

L'empilement des espaces de noms locaux obéit à une politique de **pile** : le sommet correspond au dernier appel en date. Quand cet appel se termine, son espace de nom est supprimé de la pile et l'exécution de l'appel précédent reprend. Quand un nouvel appel est effectué, l'exécution de la fonction en cours s'interrompt, un nouvel espace de noms local est créé et l'exécution de la fonction appelée commence. En Python, il est possible d'accéder en lecture à n'importe quel nom défini dans un des espaces de noms empilés. Si plusieurs espaces contiennent le même nom, c'est l'espace de nom le plus récent qui est sélectionné. Pour accéder en écriture à une variable qui n'est pas dans l'espace de nom le plus récent, il faut utiliser les mots-clés `global` ou `non-local` (à utiliser avec précaution).

2) Déroulement détaillé d'un appel

Considérons l'appel suivant :

```
def ma_fonction(p_1, ..., p_n):
    ...
    return expr

# Appel de fonction (à l'intérieur d'une expression)
... ma_fonction(e_1, ..., e_n) ...
```

Voici la succession des étapes de l'appel :

- Création d'un espace de noms local contenant `p_1` à `p_n` au sommet de la pile d'appels
- Chaque expression `e_i` est évaluée en une valeur `v_i` et affectée à la variable `p_i`
- Exécution du corps de la fonction dans l'espace de noms local
- Si la fonction exécute l'instruction `return expr` ou atteint la fin de son bloc d'instructions, l'espace de noms local est détruit et l'expression appelante `ma_fonction(e_1, ..., e_n)` prend la valeur de `expr` (respectivement `None`)
- Reprise du programme principal dans l'espace global

Les noms `p_1` à `p_n` sont parfois appelés **paramètres formels**, les valeurs `v_1` à `v_n` **paramètres effectifs**. On appelle parfois aussi **paramètres** (tout court) les paramètres formels et **arguments** les paramètres effectifs.

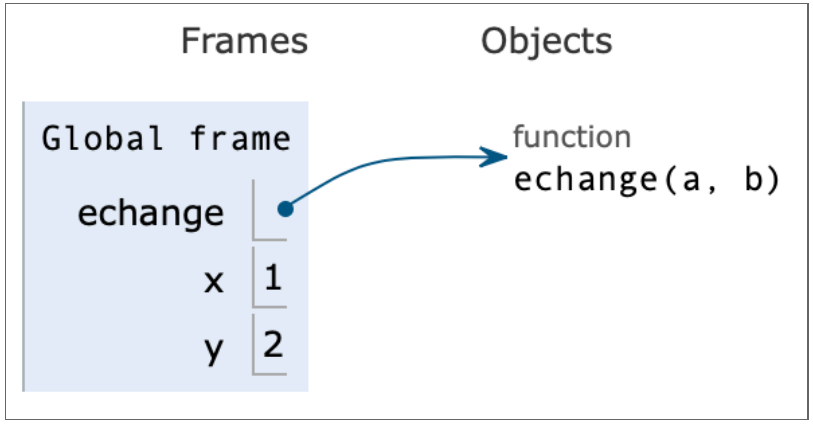
3) Exemple détaillé

Essayons d'écrire une fonction permettant d'intervertir les valeurs de deux variables :

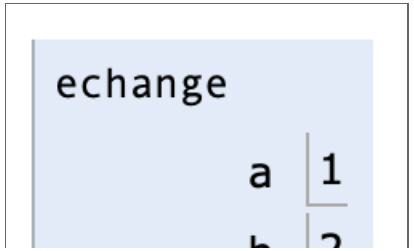
In [28]:
```
def echange(a, b):
    """ Fonction censé intervertir les valeurs de deux variables"""
    temp = a
    a = b
    b = temp

x = 1
y = 2
echange(x, y)
print(x, y)
print(temp)
```

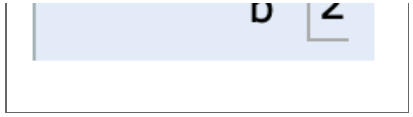
```
1 2

---------------------------------------------------------------------
-------
NameError                                 Traceback (most recent cal
l last)
~\AppData\Local\Temp/ipykernel_3132/4107649134.py in <module>
      9 echange(x, y)
     10 print(x, y)
---> 11 print(temp)

NameError: name 'temp' is not defined
```

A la ligne 1, on place dans l'espace de nom global une variable `echange`. Aux lignes 7 et 8, on y place deux variables `x` et `y`. A ce stade, on a donc :

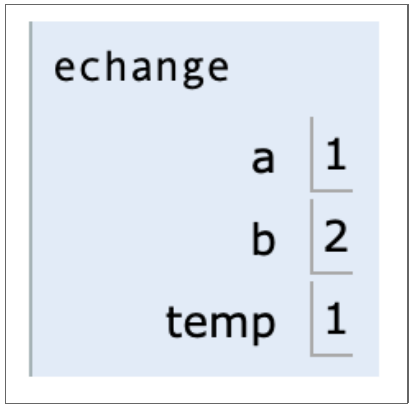


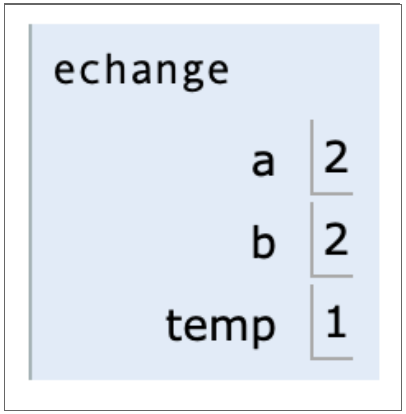
Ensuite, on arrive à l'appel de fonctions `echange(x, y)`. A ce niveau la, un nouvel espace de nom est crée. On y évalue `x` et stocke le résultat dans la variable `a` ; de même, `y` est évaluée et le résultat est stocké dans la variable `b`.

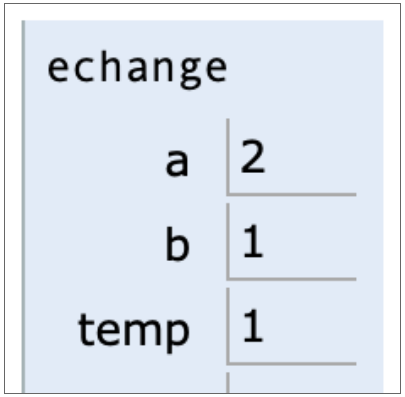




Désormais, la variable `temp` est crée, puis la variable `a` est mise à jour et enfin la variable `b` est mise à jour.

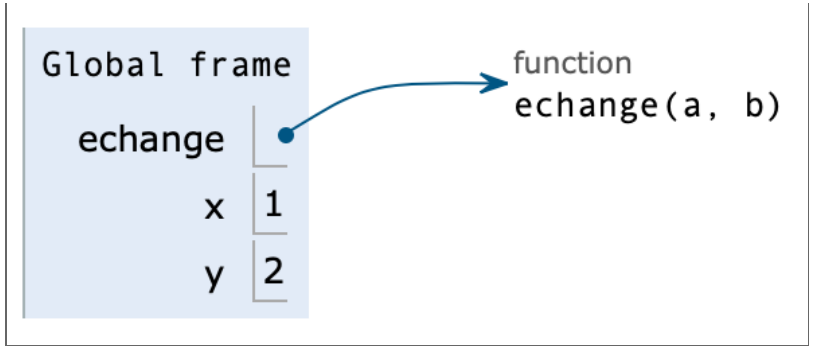






Création de `temp` Mise à jour de `a` Mise à jour de `b`

Au cours de l'exécution de la fonction `echange`, avons-nous modifié les données dans l'espace de noms global ? **Non, bien sur !** Nous avons interverti le contenu des variables locales `a` et `b` présente dans l'espace de nom local.
A la fin de l'éxécution du programme, l'espace de nom global vaut donc :



En particulier, l'appel `print(tmp)` provoquera une erreur, car la variable `tmp` n'est pas dans l'espace de nom global !

Vérifier vos connaissances

A ce stade, vous devez être capable :

- de décrire l'espace de nom global d'un programme ;
- de décrire les espaces de nom locaux lors de l'exécution d'une fonction ;
- de comprendre de manière détaillée comment fonctionne un appel de fonction ;
- de décrire l'exécution pas à pas d'une fonction.

Exercice 8 : Décrire l'exécution pas à pas du programme suivant (avec état de la mémoire).

In [ ]:
```
def maximum(a, b):
    """Renvoie le maximum des deux paramètres"""
    if a > b:
        return a
    else:
        return b

nb1 = int(input("Saisissez un nombre : "))
nb2 = int(input("Saisissez un nombre : "))
print("Le max de", nb1, "et", nb2, "est", maximum(nb1, nb2))
```

Remarque : *On pourra s'auto-corriger grâce à* ***Python Tutor***.

V Documentation et test de fonctions

1) Chaînes de documentation (*docstring*)

Il est important d'indiquer par un commentaire à quoi sert une fonction que l'on a programmé. Cela permet de :

- transmettre simplement à ces collègues comment fonctionne une fonction, quels sont ses paramètres, sa valeur de retour, etc ;
- se souvenir simplement ce que l'on a déjà programmé et de comment cela fonctionne.

In [ ]:
```python
def triple(nombre):
    '''
    Fonction calculant le triple d'un nombre.

    :param nombre: int ou float
    :return value: int ou float (même type que celui d'entrée)
    '''
    return nombre * 3
```

On peut accéder à la chaîne de documentation d'une fonction en tapant `help(nom de la fonction)` dans l'interpréteur :

In [ ]:
```
help(triple)
```

Cela fonctionne aussi pour les fonctions prédéfinies ou issues de modules :

In [ ]:
```
from random import randint
help(randint)
```

Vérifier vos connaissances

A ce stade, vous devez être capable :

- de comprendre l'intérêt d'écrire de la documentation ;
- d'écrire des docstrings pour décrire vos fonctions.

Exercice 9

Ecrire des docstrings pour les fonctions que vous avez écrites aux exercices 3 à 6.

2) Tests intégrés à la documentation (*doctest*)

POURQUOI FAIRE DES TESTS ?

Comme tout morceau de programme, chaque fonction doit être *testée* immédiatement pour s'assurer qu'elle fonctionne correctement.

In [ ]:
```
triple(3)
```

In [ ]:
```
triple(9.0)
```

Plutôt que de perdre ces tests, ou de devoir les réaliser à chaque fois que l'on modifie le programme, il est utile de les intégrer à la documentation de la fonction pour pouvoir s'y référer plus tard :

- Cela donne des exemples d'utilisation de la fonction ;
- Si l'on change le code de la fonction, cela permet aussi de vérifier que son comportement reste correct.

LA SYNTAXE POUR ÉCRIRE DES DOCTESTS

In [ ]:
```
def triple(nombre):
    '''
    Fonction calculant le triple d'un nombre.

    :param nombre: int ou float
    :return value: int ou float (même type que celui d'entrée)

    >>> triple(3)
    9
    >>> triple(9.0)
    27.0
    '''
    return 3 * nombre
```

In [ ]:
```
def racine(nombre):
    '''
    Fonction calculant la racine carrée d'un nombre.

    :param nombre: int ou float
    :return value: float

    >>> racine(0)
    0.0
    >>> racine(1)
    1.0
    >>> racine(4)
    2.0
    '''
    return nombre ** (1/2)
```

Comment vérifier que les doctests sont validés ?

Il existe des outils qui permettent de lancer automatiquement tous les tests présents dans la documentation, et de vérifier qu'ils produisent exactement les résultats annoncés.
Par exemple, à la fin d'un programme, on peut écrire le code suivant pour lancer systématiquement tous les tests présents dans le fichier :

```
import doctest
doctest.testmod()
```

In [ ]:
```
def triple(nombre):
    '''
    Fonction calculant le triple d'un nombre.

    :param nombre: int ou float
    :return value: int ou float (même type que celui d'entrée)

    >>> triple(3)
    9
    >>> triple(9.0)
    27.0
    '''
    return 3 * nombre

def racine(nombre):
    '''
    Fonction calculant la racine carrée d'un nombre.

    :param nombre: int ou float
    :return value: float

    >>> racine(0)
    0.0
    >>> racine(1)
    1.0
    >>> racine(4)
    2.0
```

```
    '''
    return nombre ** (1/2)

import doctest
doctest.testmod()
```

Voici le retour de Python lorsqu'un test échoue :

In [ ]:
```
def racine(nombre):
    '''
    Fonction calculant la racine carrée d'un nombre.

    :param nombre: int ou float
    :return value: float

    >>> racine(0)
    0.0
    >>> racine(1)
    1.0
    >>> racine(2)
    1.414
    >>> racine(4)
    2.0
    '''
    return nombre ** (1/2)

import doctest
doctest.testmod()
```

Ici, le test de la valeur de retour de `racine(2)` échoue. La valeur n'est pas `1.414` : il semble qu'elle soit plutôt `1.4142135623730951`.

DES DOCTESTS ÉVOLUÉS 1 : TESTER L'ÉGALITÉ DE FLOATTANT

De temps en temps, on a besoin de tester que la valeur de retour d'une fonction renvoyant un floatant est correcte. Mais, tester l'égalité de deux floattants est difficile.
Par exemple, on a :

In [ ]:
```
0.1 + 0.2 == 0.3
```

Ce test, évalué à `False` par Python, devrait bien évidemment être évalué à `True`. Ceci est du à la représentation des nombres en binaires.
En raison de cet exemple, modifier le doctest de l'exemple précédent en

```
>>> racine(2)
1.4142135623730951
```

ne serait pas plus satisfaisant que de laisser le doctest précédent.
Voici comment faire un tel test :
Pour tester l'égalité de deux floattants `x` et `y`, on se donne alors une précision `epsilon` proche de `0`. Si nos deux floattants sont égaux à `epsilon` près, c'est-à-dire lorsque `abs(x - y) <= epsilon` vaut `True`, on considèrera que les floattants `x` et `y` sont égaux.

In [ ]:
```
epsilon = 10**-10
x = 0.1 + 0.2
y = 0.3
abs(x - y) <= epsilon
```

On peut alors transformer ces lignes de codes en doctest :

In [ ]:
```
def racine(nombre):
    '''
    Fonction calculant la racine carrée d'un nombre.

    :param nombre: int ou float
    :return value: float

    >>> racine(0)
    0.0
    >>> racine(1)
    1.0
    >>> epsilon = 10**-10
    >>> abs(racine(2) - 1.4142135623730951) < epsilon
    True
    >>> racine(4)
    2.0
    '''
    return nombre ** (1/2)

import doctest
doctest.testmod()
```

DES DOCTESTS ÉVOLUÉS 2 : TESTER UNE FONCTION AVEC DE L'ALÉATOIRE

Rappelons que pour reproduire le comportement d'une fonction simulant de l'aléatoire, il nous faut fixer la graine à l'aide d'une *graine*.

In [ ]:
```
from random import randint
from random import seed

def lance_de() :
    """Simule un lancé de dé à 6 faces

    >>> seed(104438493)
    >>> lance_de()
    6
    >>> lance_de()
    5
    >>> lance_de()
    3
    >>> lance_de()
    1
    >>> lance_de()
    4
    >>> lance_de()
    6
    """
    return randint(1,6)

def jusqu_a_un_6():
    """Simule des lancés de dés à 6 faces jusqu'à obtenir un 6

    :return value: le nombre de lancés réaliser pour obtenir un 6
```

```
    >>> seed(104438493)
    >>> jusqu_a_un_6()
    1
    >>> jusqu_a_un_6()
    5
    """
    compteur = 1
    while lance_de() != 6:
        compteur = compteur + 1
    return compteur

#print('Obtenu un 6 en', jusqu_a_un_6(), 'jets de dés.')

import doctest
doctest.testmod()
```

Vérifier vos connaissances

A ce stade, vous devez être capable :

- de comprendre l'intérêt d'écrire des doctests ;
- de savoir écrire un doctest simple ;
- de savoir écrire un doctest évolué ;
- de savoir lancer une série de doctests automatiquement.

Exercice 10 :

Ecrire des doctests pour les fonctions que vous avez écrites aux exercices 3 à 6.