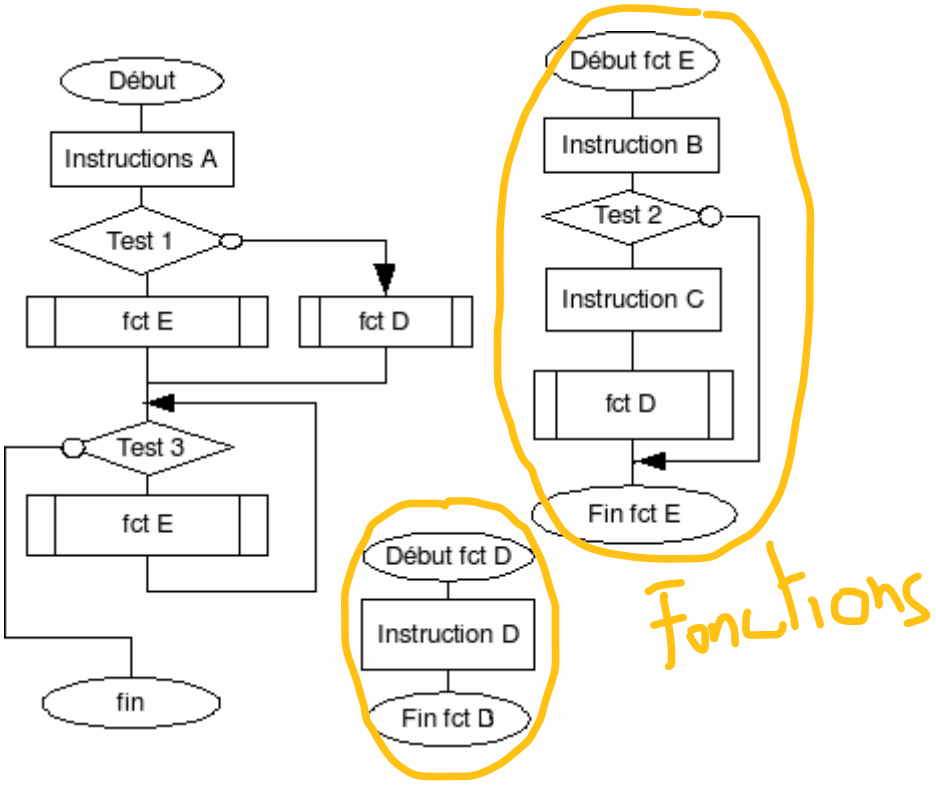
CORRIGE INSTRUCTIONS COMPOSEES EN PYTHON (2) : FONCTIONS.

Me signaler toute erreur éventuelle !



- *Logiciels et sites internet* : Editeur et console Python (Thonny, VS Code etc.) ; pythontutor.com, franceioi.org.
- *Pré-requis pour prendre un bon départ :*

	☹	😐	☺	☺☺
Variables : initialisation, affectation, auto-affectation, incrémentation, etc.				
Boucles For				
Tests.				
Boucles While.				

Ce cours Python fonctionne en pédagogie inversée. Ce livret est donc un post-cours complémentaire aux exos de France IOI, et doit être fait juste après les chapitres correspondants sur France IOI :

Exercices France IOI	*Cours livret*
Niveau 2 Chapitre 4	Tous les chapitres de ce livret.

NOM et prénom : …… *Première spécialité NSI*

I. PROBLEMATIQUE : REUTILISATION D'INSTRUCTIONS.

➢ Soit le programme ci-contre. On veut réutiliser à la ligne 7 le bloc d'instructions débutant à la ligne 3.

1	instruction
2	instruction
3	Bloc d'instructions
4	
5	
6	instruction
7	
8	

- 1ère idée : Recopier bêtement ce bloc !

  Problème : Si le bloc fait 10 000 lignes ? Et si plus tard on veut encore réutiliser ce bloc ? Encore recopier 10 000 lignes ? Impensable !

- 2ème idée : Si ce bloc était adjacent à la ligne 8, cela ferait 2 blocs identiques enchaînés qu'on aurait pu simplement remplacer par une boucle For.

  Problème : Il n'y a aucune raison que le bloc soit adjacent à la ligne où on veut le réutiliser ! Donc on ne peut quasiment jamais utiliser une boucle dans ce cas.

- 3ème idée : Utiliser une instruction spéciale faisant revenir à la ligne 3.

  Problème : Si cette instruction existe parfois dans certains langages (instruction JMP (jump) en Assembleur ; instruction « Go To » dans des langages primitifs comme Basic ou Fortran ; ou même en langage C !), elle a quasiment disparu des langages structurés modernes (Python, Java, Ruby etc.) à cause des problèmes inextricables de lisibilité du programme qu'elle engendre :

  Ex : On met un go to 3 en ligne 7. Que faut-il alors mettre à la ligne 5 pour que le programme reprenne en ligne 8 sans exécuter l'instruction 6 ? Un test.

1	instruction
2	k = 0
3	Bloc
4	k = k + 1
5	if k = 2 go to 8
6	instruction
7	go to 3
8	instruction

  Quels nouveaux problèmes se posent alors et comment les résoudre ? Il faut mettre un compteur k pour voir si c'est la 1ère fois ou 2ème fois qu'on rentre dans le bloc. Bien compliqué pour réaccéder à un bloc !

- 4ème idée : Améliorer la 3ème idée !

  Les problèmes de lisibilité viennent du fait que le bloc est situé *en plein* dans le flux d'exécution du programme et engendre ainsi un enchevêtrement de Go To difficiles à démêler. Comment y remédier ?

  En externalisant tout simplement le bloc !

➢ On se retrouve donc avec un programme et un bloc externe d'instructions.

Une autre façon de voir les choses est de s'inspirer d'une usine et d'un sous-traitant :

Usine principale	↔	Sous-traitant
↕		↕
Programme	↔	Bloc externe

Usine principale	
1	instruction
2	instruction
3	
4	instruction
5	instruction

Sous-traitant	
1	Bloc d'instructions
2	
3	

De quoi a besoin l'usine principale par rapport à son sous-traitant ?

- Appeler son sous-traitant.
- Lui envoyer des choses.
- Recevoir des choses en retour.

En Python et dans bien d'autres langages, un bloc d'instructions externalisé ne s'appelle pas un sous-traitant mais une fonction.

II. FONCTIONS : DEFINITION, SYNTAXE.

Définition : En Algorithmique-Programmation, **une fonction est un sous-programme** pouvant être utilisé par un autre programme.

Syntaxe	*Explications sur la syntaxe*
❶ **def** nom_fonction (paramètre(s)) : """ Doc strings ❷ """ Corps de la fonction ❸ return (valeurs)	❶ **def** nom_fonction (paramètre(s)) : • Entête obligatoire permettant de définir la fonction. • Commence obligatoirement par le mot réservé def et se termine obligatoirement par « : ». • nom_fonction : nom **sans espaces, explicite (verbe),** indiquant ce que fait la fonction. Ex : calcul_moyenne, tri_tableau etc. C'est par ce nom qu'on appelle la fonction à partir du programme principal. • (paramètre(s)) : **parenthèses obligatoires, même vides !** Il peut y avoir 0 ou 1 ou plusieurs paramètres d'entrée. Le ou les paramètres sont des variables qui réceptionnent s'il y en a les informations en entrée de la fonction. ❷ """Doc strings""" : « obligatoires », indentés. Commentaires sur une ou plusieurs lignes, entre triples quotes ", documentant la fonction : que fait-elle ? y-a-t-il des pré-conditions sur les arguments en entrée ? des post-conditions sur les valeurs en sortie ? etc. ❸ Corps de la fonction : • Bloc d'instructions obligatoirement ………………… à droite. • En Python, au moins 1 instruction ! • return (valeurs) : instruction parfois facultative. Permet la transmission en sortie d'une ou plusieurs informations du sous-programme vers le programme appelant.
Attention !	
• Pas de nom vague et non explicite comme f, fonction etc. ! • Oubli des parenthèses après le nom. • Paramètres séparés par des virgules. • Oubli des « : ». • Indentation des docs strings et du corps de la fonction. • Oubli des doc strings.	

Exemple et traduction

def calc_rect (a, b) : """Doc strings""" aire = a × b peri = 2*a + 2*b return (aire, peri)	• Fonction nommée calc_rect avec 2 paramètres d'entrée (a, b) qui représentant la longueur et la largeur d'un rectangle. • En sortie, la fonction renverra un couple de 2 valeurs (aire , péri) qui représentent l'aire et le périmètre du rectangle.

Remarques

• **Jamais le même nom pour une fonction et une variable !** Sinon TypeError assuré !

• Les fonctions en Informatique ont un sens plus large qu'en Maths : il faut plutôt les comprendre comme fonctionnalités que comme formules de calcul.

III. COMMUNICATION PROGRAMME ↔ SOUS-PROGRAMME.

A. Com. programme → sous-programme : appel d'une fonction.

Programme	
1	instruction
2	instruction
3	nom_fonction( )
4	instruction
5	instruction

1	def nom_fonction ( ) :
2	Bloc d'instructions
3	
4	

- Pour faire exécuter dans un programme le bloc externe représenté par la fonction, il suffit d'écrire le nom_fonction ( ) à l'endroit voulu dans le programme.

On dit alors qu'**on appelle la fonction**.

- Tout comme avec les variables, on ne peut faire appel à une fonction que si logiquement elle a été déclarée avant son appel !

Par convention, on place donc au début du fichier toutes les fonctions et seulement après le reste du programme (voir ci-contre).

Pour accentuer la séparation, on place une ligne **# Programme principal**.

1	def nom_fonction ( ) :
2	Bloc d'instructions
3	
4	
5	
6	# Programme principal
7	instruction
8	nom_fonction **( )**
9	instruction

- Ex : A quelle ligne commence le programme principal ? 6.
  A quelle ligne est appelée la fonction ? 8.
  Combien de lignes d'instruction comporte la fonction ? 3.

- Si la fonction est écrite dans un autre fichier, on placera au début du fichier contenant le programme principal l'instruction suivante : from nom_du_fichier import nom_fonction.

B. Communication programme → sous-programme : arguments.

➢ Dans l'exemple au-dessus, appel de la fonction ⇒ communication Programme vers Sous-programme.

Mais pas de transmission d'informations car les parenthèses après nom_fonction (ligne8) étaient ……

Lors de l'appel d'une fonction, la transmission d'informations du programme principal vers la fonction se fait de la manière ci-contre :

1	def nom_fonction (param1, param2, etc.) :
2	Bloc d'instructions
3	
4	
5	
6	# Programme principal
7	instruction
8	nom_fonction (arg1, arg2, etc.)
9	instruction

❷ ❶

❶ Les informations arg1, arg2 etc. transmises entre parenthèses ( ) s'appellent les arguments.

En Python, ces arguments sont des simples valeurs ou des valeurs de variables ou des valeurs d'expressions.

❷ Ces arguments sont alors affectés aux paramètres entre parenthèses dans l'entête, dans le même ordre.

Puis ces paramètres peuvent alors être utilisés comme variables dans le bloc d'instructions de la fonction.

➢ 13 erreurs ou oublis dans le script ci-contre. Où ça ?

1	instruction
2	Def faire_Bisous (ki , kombien ; ouh) :
3	Bloc d'instructions
4	
5	
6	# Programme principal
7	instruction
8	faire_bisous (5 ; 'Gilles et John' , 'cou')

La fonction n'est pas déclarée dès le début du programme.

Bloc d'instructions non indenté dans la fonction.

Les premiers arguments ne sont pas dans le bon ordre.

Il manque un argument.

Plein de petites erreurs : virgules, ( ), « : », majuscules etc.

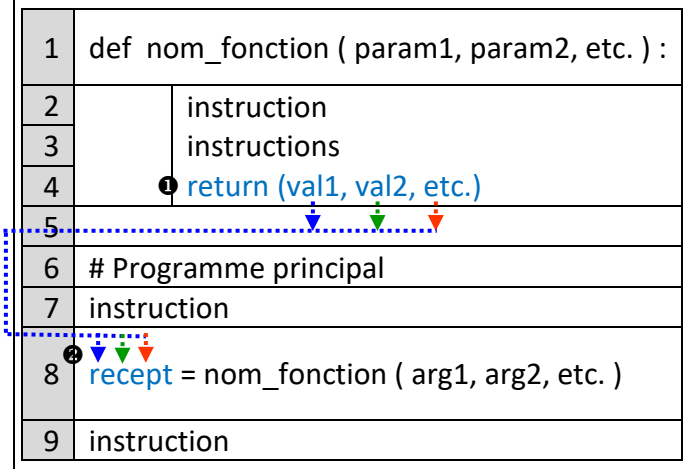
Grâce à la transmission d'arguments, des informations peuvent passer du programme principal → fonction.

Evidemment, la possibilité dans le sens inverse fonction → programme principal existe et elle se fait grâce à l'instruction return.

C. Communication sous-programme → programme : instruction return.

Définition : L'instruction **return** permet, si besoin, la sortie d'informations vers le programme appelant.

Schéma	*Explications*



❶ return (val1 , val2, etc.) :

• return : instruction de renvoi vers l'appel de la fct.

En général, il y a un seul return placé en fin de bloc.

• val1, val2 : la ou les valeurs à retourner.

❷ recept : Variables de réception dans pgm principal.

• Réception des valeurs retournées dans la ou les variables auxquelles nom_fonction( ) est affectée.

• Attention, s'il n'y a pas de variables de réception, les valeurs retournées ne sont pas stockées !

Valeur(s) renvoyée(s) par la fonction suivant les différentes possibilités de return.

Instruction :	pas de return	return vide (inutile !)	return (val) ou return val	return (val1, val2, etc.) ou return val1, val2, etc.	return [val1, val2, etc.]
La fonction renvoie :	None (rien)	None (rien)	val	**le tuple** (val1, val2, etc.) (liste constante)	la liste [val1, val2, etc.] (liste non constante)

D'autres façons d'utiliser return

Return non en fin de bloc

Return agit comme un break qui stoppe l'exécution du bloc et renvoie ce qu'il faut.

Plusieurs return dans le bloc

Le bloc continue à être exécuté jusqu'à ce qu'un return soit exécuté.

```
def essai1 (a, b) :
    print (a)
    return
    print (b)
y = essai1 (1 , 2)
```

Que s'affiche-t-il ? 'a'

Que vaut y ? None car le return est vide.

```
def essai2 (a, b) :
    if a < 2 :
        return (a < 1)
    if b = = 2 :
        return (b)
y = essai2 (1 , 2)
```

Quel test est exécuté ?
Le 1er pas le 2ème.

Que vaut y ? y est le résultat de l'expression booléenne 1 < 1 c-à-d False.

Que renvoient les fonctions prédéfinies (natives, built in en anglais) de Python ?

None.

Pour celles qui font une action mais ne renvoient aucune valeur au programme : print( ) en particulier.

Une valeur.

Pour celles qui renvoient qq chose au programme : input( ) ; int( ) ; str( ) ; range( ) ; cos( ) etc.

D. Résumé de la communication Programme ↔ Sous-programme :

❶ Appel de la fonction par nom_fonction ( ).

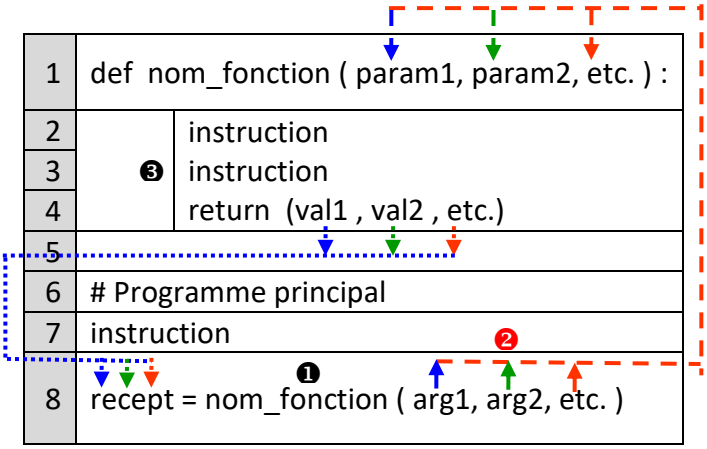
❷ • Transmission des arguments entre ( ).

• Affectation des arguments aux paramètres.

❸ Exécution du corps de la fonction.

❹ • Transmission retour des valeurs par return ( ).

• Stockage des valeurs retour dans la, les variables auxquelles nom_fonction( ) est affectée (ici recept mais cela aurait pu être à la place la liste [a , b , etc.]).



IV. VARIABLES GLOBALES, VARIABLES LOCALES.

Evidemment, tout comme le programme principal, un sous-programme peut contenir des variables.

A. Variables globales ; variables locales : définitions.

Définitions	*Exemple*	*Commentaires*

❶ On appelle **variable globale toute variable contenue dans le programme principal.**
❷ On appelle **variable locale toute variable déclarée, affectée, assignée dans un sous-programme.**
	def exemple0 (a , b) :
	k = 1 + f
	somme = a + k
	# pgm principal
	c = f = 2
	y = exemple0 (1 , 4)
		❷ variables locales :
		a , b , k et somme.
		❶ variables globales :
		c , f et y.
		Que vaut y ? None (pas de return !)

Remarques

- Les paramètres étant initialisés dans l'entête de la fonction grâce aux arguments transmis, **les paramètres sont donc des variables locales !**
- En pratique : **dans une fonction, toute variable à gauche d'un signe « = » est une variable locale.**

Application : Dans les programmes suivants, quelles sont les variables locales, globales et non définies ?

def exemple1 (c) :
h = k = 1
k = a + k + c
# pgm principal 	
exemple1 (34)
	def exemple2 (c) :
	h = k = 1
	k = a + k + c
	# pgm principal
	a = 2
	exemple2 (34)
		def exemple3 ( ) :
		k = 1
		h = k
		# pgm principal
		k = 2
		y = exemple3 ( )
			def exemple4 ( ) :
			h = k
			k = 1
			# pgm principal
			k = 2
			y = exemple4 ( )

locales : c, h, k globales : non définies : a	locales : c, h, k globales : a non définies :	locales : k , h globales : k, y non définies :	locales : h, k globales : k, y non définies : k, h, y

Les 3 dernières fonctions montrent l'une des principales difficultés liées aux fonctions : que se passe-t-il si par malheur des variables ont le même nom dans le programme principal et dans le sous-programme ?!

B. Conflit de noms entre variables locale et globale ?

➢ Lorsqu'une fonction est appelée, un espace mémoire lui est dynamiquement réservé, en dehors de l'espace mémoire alloué au programme principal (voir schéma ci-dessous).

Donc une variable k dans le programme principal et une variable qui aurait le même nom k dans le sous-programme ne partagent pas la même cellule mémoire et sont donc complètement indépendantes l'une de l'autre bien qu'elles aient le même nom. En fait, c'est comme si la variable locale s'appelait k_loc.

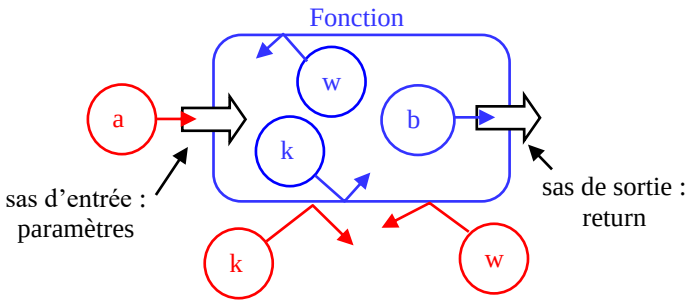
Espace mémoire du programme principal			Espace mémoire de la fonction		
	adresse 10 nom = k			adresse 15 nom = k	

➢ Il n'y a donc pas de conflit de noms ! Les changements sur une variable globale k du programme principal ne seront pas répercutés vicieusement sur la variable locale qui aurait le même nom k dans la fonction. Et vice versa ! Ouf !

Le corps d'une fonction est donc presque complètement hermétique au programme principal.

Les échanges avec la fonction se font logiquement :

- en entrée par ses paramètres.
- en sortie par l'instruction return.



➢ La transmission d'informations par le canal paramètres-return est une communication balisée et sécurisée. En dehors des sas d'entrée et de sortie, il n'y a pas de transmission incontrôlée d'informations.

C. Les dangers de la transmission directe :

A tort (ou à raison ?), les concepteurs de langages dont Python ont laissé la possibilité d'une transmission d'informations en dehors du canal sécurisé paramètres-return ! C'est la transmission directe.

<table>
<tr><th colspan="2">Transmission directe en entrée
sans passer par le sas d'entrée des paramètres.</th><th colspan="2">Transmission directe en sortie
sans passer par le sas de sortie return.</th></tr>
<tr><td>def exemple5 (b) :
  h = b + a
  return h
# pgm principal
a = 'to'
c = exemple5 ('to')

Que vaut c ? 'toto'.</td><td>Une variable globale peut transmettre directement sa valeur dans une fonction sans passer par les paramètres, tant qu'elle ne se transforme pas en variable locale !
C'est le cas ici pour la variable a.</td><td>def exemple6 (c) :
  global h
  h = c + 1
# pgm principal
exemple6 (9)
a = h

Que vaut a ? 10.</td><td><u>Seulement si la fonction est appelée</u>, la variable h devient globale (mot réservé global), et existera aussi dans le pgm principal. Sa valeur est donc directement accessible au programme principal sans passer par return.</td></tr>
<tr><td colspan="2"><u>Danger !</u> Si par mégarde, le programmeur rend la variable a locale en l'affectant à gauche d'un signe =, alors :
• au mieux il n'y aura plus échange entre la variable globale et la variable locale du même nom (fonction exemple3 p.6).
• au pire il y aura une erreur d'assignation : dans la fonction exemple4 p.6, la variable k est devenue locale à cause de k = 1. Mais elle est alors utilisée avant d'avoir été initialisée !</td><td colspan="2"><u>Danger !</u> Il y a déjà des variables globales, des variables locales, et des variables locales et globales qui auraient le même nom. Rajouter en plus des variables quasi-globales ne peut qu'augmenter la confusion !!
C'est pourquoi on réserve en général la déclaration global seulement aux constantes !
La déclaration global est à éviter absolument !</td></tr>
</table>

D. Variables globales, variables locales : les bonnes pratiques.

Soucis !	*Pourquoi ?*	*Remède*
Variables globale et locale ayant le même nom !	• Confusions entre les 2 variables qui ont le même nom et qui sont pourtant indépendantes ! • Illisibilité du programme.	**Rajouter le suffixe _loc à la variable locale** pour matérialiser l'indépendance entre la variable locale et la variable globale.
Transmission directe en entrée sans passer par les paramètres.	Si la variable dont on transmet la valeur sans passer par les paramètres devient locale, alors : • soit il y aura indépendance non voulue (fonction exemple3 p.6). • soit il y aura erreur (fonction exemple4 p.6).	*Transmettre en argument la valeur à la fonction et utiliser return en sortie.*
Transmission directe en sortie sans passer par return.	Confusions entre les variables vraiment globales (définie dans le pgm principal) et quasi-globales (global dans une fonction) ayant le même nom !	*Transmettre en argument la valeur à la fonction et utiliser return en sortie.*

E. Exercices sur la portée des variables : globales ou locales ?

❶ Qu'affichent ces 7 programmes ? Lequel est préférable ? mieux

def essai1 ( ) : x = 5 print(x)	def essai2( ) : x = 5 return x print(x)	def essai3( ) : x = 5 return x print(essai( ))	def essai4( ) : x = 5 return x essai4( ) print(x)	def essai5( ) : x = 5 return x x = essai5( ) print(x)	def essai6( ) : global x x = 5 print(x)	def essai7( ) : global x x = 5 essai7( ) print(x)
NameError : x n'est pas défini.	*NameError : x n'est pas défini.*	*5 : **print capture le return.***	*NameError : x n'est pas défini.*	*5*	*NameError : x non défini car essai non appelée.*	*5*

❷ Qu'affichent ces 7 programmes ? Lesquels changent la valeur de x ? Lequel est préférable ? mieux

def change1(x) : x = 5 x = 7 print(change1(x))	def change2(x) : x = 5 return x x = 7 print(change2(x))	def change3(x) : x = 5 return x x = 7 change3 (x) print(x)	def change4(x) : x = 5 return x x = 7 x = change4 (x) print(x)	def change5(x_loc) : x_loc = 5 return x_loc x = 7 x = change5 (x) print(x)	def change6( ) : global x x = 5 x = 7 print(x)	def change7( ) : global x x = 5 x = 7 change7 (x) print(x)
Rien ne s'affiche. x = 7.	*5 s'affiche. print capture le return. x = 7.*	*7 s'affiche. x = 7.*	*5 s'affiche. x = 5.*	*5 s'affiche. x = 5.*	*Pas d'affichage. Fonction non appelée, x = 7.*	*5 s'affiche. Fonction appelée, x = 5.*

❸ Les 8 scripts suivants affichent soit « 5 », soit « None », soit « NameError ». Qu'affiche chaque script ?

def somme1(x,y) :
  global res
  res = x+a
 	
a = 3
print (res)
	def somme2(x,y) :
	  global res
	  res = x+a
	
	a = 3
	somme2 (2,0)
	print (res)
		def somme3(x,y) :
		  global res
		  res = x+a
		
		a = 3
		print (somme3(2,0))
			def somme4(x,y) :
			  res = x+a
			  return res
			
			a = 3
			print (res)
				def somme5(x,y) :
				  res = x+a
				
				a = 3
				print (somme5(2,0))
					def somme6(x,y) :
					  res = x+a
					
					a = 3
					somme6 (2,0)
					print (res)

NameError. *res non défini malgré global, car fonction non appelée*	*5 s'affiche.* *a = 3* *res = 5*	*None.* *Somme3( ) ne renvoie rien même si res vaut 5.*	*NameError.* *res non définie dans le programme principal.*	*None.* *La fonction ne retourne rien (pas de return).*	*NameError.* *res non définie dans le programme principal.*

def somme7(x,y) :
  res = x+a
  return
 	
a = 3
print (somme7(2,0))
	def somme8(x,y) :
	  res = x+a
	  return res
	
	a = 3
	print (somme8(2,0))				

None. *La fonction ne retourne rien (return vide).*	*5 s'affiche.* *le print capture ce qui est retourné par la fonction.*				

V. QUELQUES CHOSES EN PLUS…

A. Pourquoi utiliser des fonctions ?

1. Factorisation du code :

   Les fonctions évitent d'écrire plusieurs fois les mêmes blocs de code (aux valeurs de variables près).

2. Clarté :

   Les fonctions structurent le programme en séparant les différentes fonctionnalités du programme principal et en les nommant.

3. Modularité :

   On peut créer des bibliothèques (librairies) de fonctions réutilisables par d'autres programmes.

4. Diviser pour régner :

   Lorsqu'on analyse un problème, sa découpe en sous-parties fait apparaître des fonctions.

5. Rapidité d'exécution :

   Les fonctions étant écrites en premier, elles sont compilées d'avance.

B. Compléments sur les fonctions :

1. Vocabulaire souvent utilisé dans les autres cours sur les fonctions :

<table>
<tr><th>Vocabulaire</th><th>Explications</th></tr>
<tr><td>Programme principal.</td><td colspan="3">Toutes les instructions écrites en dehors de n’importe quelle fonction.</td></tr>
<tr><td>Déclarer une fonction.</td><td colspan="3">Ecrire son entête.
<u>Ex :</u> Comparer les entêtes des 2 fonctions somme, l’une écrite en C, l’autre en Python.
int calc_somme (int param1, int param2) ; def calc_somme (param1, param2) :
Différences ? En C, pas de mot réservé, « ; » à la place de « : ». typage des paramètres d’entrée et de la valeur de sortie.</td></tr>
<tr><td>Implémenter ou définir une fonction.</td><td colspan="3">Ecrire une fonction complètement (entête + corps).
Dans d’autres langages (exemple en C++), il est possible de déclarer une fonction sans l’implémenter (entête mais pas de corps), mais pas en Python !</td></tr>
<tr><td>Signature ou Prototype d’une fonction.</td><td colspan="3">Ensemble des caractéristiques (spécifications) d’une fonction : son nom, ce qu’elle fait, ses paramètres d’entrée et leur type, les valeurs retournées et leur type.
En général, le prototype se résume à l’entête + les doc strings.</td></tr>
<tr><td>Procédure.</td><td rowspan="2">Dans certains langages (Pascal) ou en Algorithmique, on entend parfois parler de procédure.
Une procédure est une fonction qui ne renvoie None (rien) au programme appelant.
Donc en Python, les fonctions sans return ou avec un return vide auraient pu s’appeler procédures.</td><td>def exemple7( ) :
somme = 5
return 20</td><td>def exemple8( ) :
somme = 5
return</td></tr>
<tr><td colspan="2">Laquelle est une procédure ?
exemple8( )</td></tr>
<tr><td>Sous fonction.</td><td colspan="3">Fonction définie à l’intérieur d’une autre fonction.</td></tr>
<tr><td>Paramètres formels, effectifs.</td><td colspan="3">Les paramètres formels sont les paramètres d’entrée d’une fonction, dans l’entête.
Les paramètres effectifs ou réels sont les arguments transmis lors de l’appel de fonction.</td></tr>
<tr><td>Portée d’une variable.</td><td colspan="3">Zone (espace mémoire) dans laquelle cette variable est utilisable.
Variable utilisable localement ? globalement ?</td></tr>
</table>

2. Fonction particulière : fonction main ( ) :

Les fonctions étant écrite en début de programme, les instructions sont plus rapidement exécutées lorsqu’elles sont à l’intérieur d’une fonction. D’où l’idée de mettre toutes les instructions du programme principal elles-mêmes dans le corps d’une fonction principale : la fonction **main ( )**.

Et le nouveau programme principal ne sera constitué que d’une instruction : l’appel de la fonction main ( ).

3. Affecter des valeurs par défaut à certains paramètres :

Lorsqu'on définit les paramètres d'une fonction, il est possible en Python de spécifier des valeurs par défaut à certains de ses paramètres (en C++ aussi par exemple mais pas en C).

Exemple	*Explication*	*Avantage*

def exemple7 (a , b , c = 3) :
somme_loc = a + b + c
return somme_loc
# pgm principal 	
somme1 = exemple7 (1, 3 , 6)
somme2 = exemple7 (1 , 3)
Combien vaut somme1 ? 10.
Combien vaut somme2 ? 7.
	La ou les valeurs par défaut sont directement affectées dans les paramètres **en fin de parenthèses**.
	Ces valeurs par défaut ne sont utilisées que si les arguments correspondants n'existent pas lors de l'appel de la fonction.
		Permet de plus sécuriser les appels de fonctions : en spécifiant des valeurs par défaut aux paramètres, il n'y aura pas d'erreur générée si par exemple le programmeur oublie un argument lors de l'appel d'une fonction dans le code.

VI. EXERCICES.

France IOI Niveau2 Chapitre 4.

Ai-je tout compris ? Fonctions.	☹	😐	☺	☺☺
Définition, syntaxe.				
Doc strings.				
Appel d'une fonction ; arguments ; paramètres.				
Instruction return ; réception des valeurs retournées dans une ou plus variables.				
Architecture globale de la communication Programme ↔ Sous-programme.				
Différences entre variables locales et globales.				
Dangers de la transmission directe.				
Utilité et avantages des fonctions.				
Fonction main( ).				
Valeurs par défaut pour un ou plusieurs paramètres.				