Tutoriel d'utilisation de Python

Introduction

Python est un langage interprété portable, dynamique, extensible et gratuit. Lorsque la programmation a fait son apparition au collège, l'outil **Scratch** s'est beaucoup développé car il permet d'avoir une approche visuelle de la programmation en utilisant des blocs.

Au lycée, le langage Python s'est très largement répandu car il est facilement accessible, notamment grâce à une syntaxe simple. De plus, les nombreux modules qu'il propose permettent d'écrire des programmes très complexes.

Sauvegarder votre code

a) Sélectionner entièrement votre code dans la console et le copier.
b) Ouvrir un éditeur de texte (Bloc-notes, Sublime text, etc.) et coller le code.
c) Enregistrer le travail en le nommant par exemple code.py (l'extension .py est obligatoire pour l'enregistrer en tant que fichier Python).

Petite aide

Pour les utilisateurs de Scratch, il existe une petite application en ligne - **Pyblock** - qui permet d'écrire un programme en utilisant les blocs (avec moins d'options qu'avec Scratch) et l'application se charge d'écrire le code Python dans un autre onglet. Le code est ensuite compilable :

http://mathematiques-medias.discipline.ac-lille.fr/PyBlock/

Rappel du programme 2018 avec exemples

Variable et instructions élémentaires :

- choisir ou déterminer le type d'une variable (entier, flottant ou chaîne de caractères) ;
- concevoir et écrire des affectations à des variables ;
- écrire une formule permettant un calcul combinant des variables.

Python ne demande pas nécessairement de définir le type des variables utilisées mais cela est possible, par exemple, lors d'une demande de saisie pour l'utilisateur. Une bonne pratique est de nommer les variables en fonction de ce qu'elles représentent et pas simplement en utilisant des lettres. De plus, les noms de variables sont sensibles à la casse (majuscule, minuscule).

Le programme suivant qui nous sert d'exemple permet de calculer l'aire d'un rectangle en spécifiant sa longueur et sa largeur. La fonction *print* permet d'afficher la valeur finale de l'*Aire.*

Code 1 sans déterminer le type des variables	Code 2 en explicitant le type des variables

`Longueur = 4`
`largeur = 3`
`Aire = Longueur * largeur`
`print(Aire)`
	`Longueur = int(input("Longueur ?"))`
	`largeur = int(input("Largeur ?"))`
	`Aire = Longueur * largeur`
	`print(Aire)`

Les trois premières lignes sont des affectations de variables. La 3e ligne est l'écriture d'une formule permettant un calcul combinant des variables.	Les deux premières lignes vont permettre de demander à l'utilisateur de saisir des valeurs au lancement du programme. Ces valeurs sont nécessairement des entiers (*int*). Pour avoir des nombres décimaux, on écrira *float*. On notera l'importance des guillemets pour écrire du texte dans l'instruction *input*.

Boucle et itérateur, instruction conditionnelle :

- programmer une instruction conditionnelle ;
- programmer une boucle bornée ;
- programmer une boucle non bornée.

Le premier point concerne les conditions "*Si <...> alors <...> sinon <...>*" ; le point suivant aborde les boucles bornées "*Pour <var> allant de <a> à <b> faire <...>*" et le dernier point traite des boucles non bornées "*Tant que <condition> faire <...>*".

Instruction conditionnelle :	Exemple : On place deux points sur la droite numérique réelle. Connaissant leur abscisse, on cherche à connaître la distance entre ces deux points.	`AbscissePoint1 = int(input("Abscisse du 1er point ?"))`
		`AbscissePoint2 = int(input("Abscisse du 2e point ?"))`
		
		`if AbscissePoint1 > AbscissePoint2:`
		`  Distance = AbscissePoint1 - AbscissePoint2`
		`else:`
		`  Distance = AbscissePoint2 - AbscissePoint1`
		
		`print(Distance)`

Attention : les deux-points à la fin de la ligne de l'instruction *if* et de la ligne de l'instruction *else* sont obligatoires ainsi que les indentations effectuées, soit automatiquement lors du retour à la ligne, soit à partir de la barre espace ou de la touche tab. Il n'y a pas de bonne ou de mauvaise méthode, l'important est de choisir une syntaxe et de s'y tenir. En revanche, les lignes laissées vides ne sont pas obligatoires mais permettent de gagner en lisibilité.

Boucle bornée :	Exemple : La suite (u_n) est définie pour tout entier naturel n par $u_n = 3n - 4$.
	On souhaite calculer les 5 premiers termes puis les termes u_{12} à u_{16}.
	
	Le symbole # permet d'insérer des commentaires qui n'influencent pas le code.
		`#Calcul des 5 premiers termes`
		`for n in range(5):`
		`  u=3*n - 4`
		`  print(u)`
		
		`# Calcul des 5 termes de u_12 à u_16`
		`for n in range(12,17):`
		`  u=3*n - 4`
		`  print(u)`

Attention : une fois encore, les deux-points à la fin de la ligne de l'instruction *for* sont obligatoires.

L'instruction *range(n)* donne les n premiers entiers donc l'intervalle d'entiers entre 0 et $n - 1$.
L'instruction *range(a, b)* donne les $b - a$ premiers entiers suivant a en commençant par a.
Ainsi l'instruction *range(12, 17)* donne les 5 entiers suivants : 12 ; 13 ; 14 ; 15 et 16.

Multiple de 3 entre 15 et 28 :	Décompte de 4 en 4 en partant de 19 :
`for n in range(15,28,3):` `  print(n)`	`for n in range(19,0,-4):` `  print(n)`

L'instruction *range(a, b, c)* permet d'aller de a vers b en suivant un pas de c (où c peut être négatif).

Boucle non bornée :	Exemple : La suite (u_n) est définie pour tout entier naturel n par $u_n = 3n - 4$.
	On souhaite connaître le premier terme à partir duquel u_n est supérieur ou égal à 100 et son rang.
		`n = 0`
		`u = -4`
		
		`while u < 100:`
		`  n = n + 1`
		`  u = 3*n - 4`
		
		`print("u =", u, "pour n =", n)`

On remarque ici que les nombres n et u ont dû être initialisés à leur valeur de départ. On pensera aux deux-points à la fin de la ligne de l'instruction *while*.
En bonus, une utilisation de la commande *print* qui mélange à la fois du texte et l'affichage de valeur de variables.

Notion de fonction :

- programmer des fonctions simples, ayant un petit nombre d'arguments.

Reprenons notre exemple de calcul d'aire d'un rectangle mais écrivons cette fois notre programme avec une fonction :

```
# définition de la fonction
def Aire(Longueur, largeur):
    aire = Longueur * largeur
    print(aire)

# utilisation de la fonction
Aire(4,6)
Aire(4.5, 7)
```

On définit ici une fonction prenant deux arguments. Le corps du programme va travailler sur ces variables et, parce qu'on l'a codé ainsi, nous affichera la valeur calculée (ici, on a appliqué la fonction deux fois donc deux valeurs sont affichées).

Quelques exemples

Remarque : pour certains exemples, on a dû importer la librairie math *pour pouvoir effectuer certains calculs (racine carrée, puissance, exponentielle, etc.).*

<table>
<tr><td colspan="2">Racine d'un trinôme de la forme $ax^2 + bx + c$ en traitant tous les cas possibles (on admettra cependant qu'il s'agit bien d'un trinôme et donc $a \neq 0$).</td></tr>
<tr><td>Python en utilisant une saisie (input) de la part de l'utilisateur</td><td>

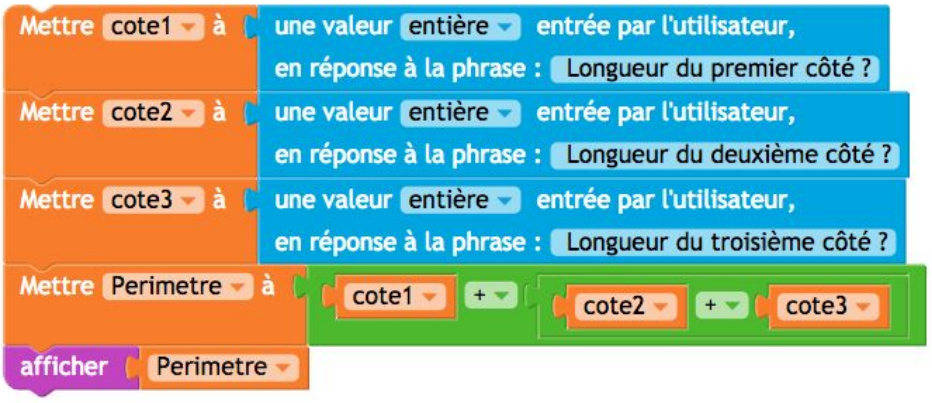
```
from math import*
a = float(input("Coefficient a ? "))
b = float(input("Coefficient b ? "))
c = float(input("Coefficient c ? "))

Delta = b**2 - 4 * a * c

if Delta < 0:
    print("Le trinôme n'a pas de racine réelle.")
elif Delta == 0:
    x0 = -b / (2 * a)
    print("Le trinôme a une seule racine :", x0, ".")
else:
    x1 = (-b - sqrt(Delta))/(2 * a)
    x2 = (-b + sqrt(Delta))/(2 * a)
    print("Le trinôme a deux racines :", x1, "et", x2, ".")
```

</td></tr>
</table>

Python en utilisant une fonction	```from math import*
	
	# Définition de la fonction
	def Racines(a,b,c):
	    Delta = b**2 - 4 * a * c
	    if Delta < 0:
	        print("Le trinôme n'a pas de racine réelle.")
	    elif Delta == 0:
	        x0 = -b / (2 * a)
	        print("Le trinôme a une seule racine :", x0, ".")
	    else:
	        x1 = (-b - sqrt(Delta))/(2 * a)
	        x2 = (-b + sqrt(Delta))/(2 * a)
	        print("Le trinôme a deux racines :", x1, "et", x2, ".")
	
	# Utilisation de la fonction sur 3 exemples :
	Racines(1, 2, 1)
	Racines(1, -1, -6)
	Racines(1 , 2 , 2)```

Calculer le périmètre d'un triangle dont les longueurs des côtés sont entières	
Pyblock	
Python en utilisant une saisie (*input*) de la part de l'utilisateur	```cote1 = int(input(" Longueur du premier côté ?")) cote2 = int(input(" Longueur du deuxième côté ?")) cote3 = int(input(" Longueur du troisième côté ?")) Perimetre = cote1 + cote2 + cote3 print(Perimetre)```

Python en utilisant une fonction	```def perimetre(cote1, cote2, cote3):
	    Perimetre = cote1 + cote2 + cote3
	    print(Perimetre)
	
	perimetre(3,4,5)```

Exemple avec un algorithme :

<table>
<tr><th colspan="2">BAC S 2017</th></tr>
<tr><td>Algorithme ancienne version</td><td>
On considère l'algorithme suivant :

Variables λ est un réel positif
S est un réel strictement compris entre 0 et 1
Initialisation Saisir S
λ prend la valeur 0
Traitement Tant que $1-\frac{\lambda+1}{e^{\lambda}}<S$ faire
λ prend la valeur $\lambda+1$
Fin Tant que
Sortie Afficher λ

a. Quelle valeur affiche cet algorithme si on saisit la valeur $S = 0{,}8$?
b. Quel est le rôle de cet algorithme ?

Remarque : la valeur de λ est ici affichée en fin d'algorithme.
</td></tr>
<tr><td>Algorithme nouvelle version (2018)</td><td>
On considère l'algorithme suivant, où la variable S désigne un réel de l'intervalle]0,1[.

$\lambda \leftarrow 0$
Tant que $1-\frac{\lambda+1}{e^{\lambda}}<S$
$\lambda \leftarrow \lambda+1$
Fin Tant que

a. Si la variable S contient la valeur 0,8 avant l'exécution de cet algorithme, que contient la variable λ à la fin de son exécution ?
b. Quel est le rôle de cet algorithme ?

Remarque : dans ce cas, la valeur de λ n'est pas affichée en fin d'algorithme.
</td></tr>
<tr><td>Pyblock</td><td>


Remarque : nouvelle version : le programme n'affiche pas la valeur de λ même si celle-ci est sauvegardée.
</td></tr>
<tr><td>Python</td><td>

```
from math import*

Lambda = 0
S = float(input(" Entrez une valeur strictement entre 0 et 1 :"))
while 1 - (Lambda + 1) / exp(Lambda) < S:
  Lambda = Lambda + 1
print(Lambda)
```

Remarque : le *print(Lambda)* permet d'afficher la valeur de λ comme dans l'algorithme ancienne version.
</td></tr>
</table>