

The Python logo, consisting of two interlocking snakes, one blue and one yellow, is positioned in the background. The text "Pour démarrer avec Python" is overlaid on the logo in a large, black, serif font.

Pour démarrer avec Python

Version du 23 avril 2019

Sommaire

I	Qu'est-ce qu'un algorithme?	1
II	Les langages	1
III	Particularités avec Python	2
IV	Le tracé de figures avec le module turtle de Python	2
V	L'affectation	4
VI	Demander une valeur à l'utilisateur	5
VII	L'affichage	5
VIII	Effectuer des calculs	5
IX	L'instruction conditionnelle	6
X	La boucle conditionnelle	6
XI	La boucle itérative	7
XII	Les fonctions algorithmiques	8
A	Utilisation simple	8
B	Utilisation plus évoluée	8
XIII	Simulation du hasard	9
XIV	L'utilisation de listes	9
XV	Les graphiques de fonctions mathématiques	11

I Qu'est-ce qu'un algorithme ?

Définition :

Un algorithme est une liste finie de processus élémentaires, appelés instructions élémentaires, amenant à la résolution d'un problème.

Exemples :

- ★ Une recette de cuisine
- ★ Un trajet élaboré par un GPS
- ★ Le plan de montage d'un meuble

II Les langages

Un algorithme peut être décrit en langage « naturel », mais on utilise dans la plupart des cas un langage plus précis adapté aux instructions utilisées : on parle alors de langage de programmation.

Cette année, nous utiliserons principalement...

- ★ le langage « naturel »,
- ★ le langage de programmation de votre calculatrice,
- ★ et le langage de programmation Python.

De façon générale, on peut considérer trois étapes dans un algorithme :

1. L'entrée des données

Dans cette étape figure la lecture des données qui seront traitées au cours de l'algorithme. Ces données peuvent être saisies au clavier ou bien être lues dans un fichier annexe.

2. Le traitement des données

C'est le cœur du programme. Il est constitué d'une suite d'instructions, parmi lesquelles les différentes opérations sur les données, qui permettent de résoudre le problème.

3. La sortie des résultats

C'est le résultat obtenu qui peut être affiché à l'écran ou enregistré dans un fichier.

III Particularités avec Python

Python est un langage de programmation très utilisé dans le domaine scientifique. Il est complété par différentes bibliothèques qui permettent de faire des calculs, de tracer des représentations graphiques ou de générer des nombres aléatoires.

Exemples :

★ La bibliothèque `math` regroupe des fonctions mathématiques telles que racine carrée `sqrt()` et les fonctions trigonométriques `sin()`, `cos()`, `tan()`. Pour l'importer, on tape la commande :

```
from math import *
```

★ La bibliothèque `random` regroupe des fonctions permettant de traiter des problèmes liés aux statistiques et probabilités. Pour l'importer, on tape la commande :

```
from random import *
```

★ La bibliothèque `pylab` regroupe les fonctions permettant le tracé de courbes représentatives de fonctions. Pour l'importer, on tape la commande :

```
from pylab import *
```

Les phrases précédées d'un dièse (#) sont des commentaires qui n'ont pas d'action sur le programme.

Elles peuvent être intéressantes pour expliquer ce qui est attendu du programme.

Exemple :

```
0 from turtle import *
1 Screen() # Affiche une fenêtre
2 forward(100) # Trace un trait de 100 pixels vers la droite
3 circle(50,180) # Trace un 1/2 cercle de centre situe a 50 px a
4 # gauche de la flèche (180 degrés pour faire un demi-tour)
```

IV Le tracé de figures avec le module turtle de Python

Dans l'exemple du **III**, on a déjà vu quelques instructions élémentaires du module « turtle ».

Voici d'autres commandes qui peuvent être utiles :

```
0 up() # Sous-entendu : pen up : pas de trace
1 forward(100) # Forward signifie : avance
2 right(90) # Tourne la tortue de 90 degrés vers sa droite
3 down() # Sous-entendu : pen down : on écrit !
4 back(100) # Recule de 100 pixels
```

Exemple :

```

0 from turtle import *
1 shape("turtle") # Enfin une tortue qui ressemble a une tortue!
2 circle(50,180)
3 forward(100)
4 up()
5 forward(100)
6 right(90)
7 down()
8 back(100)
9 mainloop() # Permet de garder la fenêtre ouverte

```

Le programme ci-dessus donne la figure suivante :



Il existe bien d'autres commandes comme :

★ `left(nombre)` ou `right(nombre)` : tourner la tortue de nombre degré vers sa gauche ou sa droite.

Exemple : `left(180)` ou `right(180)` font effectuer un demi-tour à la tortue.

★ `setx(nombre)` : définit l'abscisse de la tortue (son point de départ étant l'origine du repère $O(0;0)$).

Exemple : `setx(100)`

★ `sety(nombre)` : définit l'ordonnée de la tortue (son point de départ étant toujours l'origine $O(0;0)$).

Exemple : `sety(100)`

★ `goto(x,y)` : change la position de la tortue (x désigne l'abscisse, y l'ordonnée).

Exemple : `goto(25,75)`

★ `setheading(nombre)` : définit l'orientation de la tortue. nombre représente un angle (en degrés par défaut : 0 pour l'est, 90 pour le nord, 180 pour l'ouest, 270 pour le sud, etc.).

Exemple : `setheading(30)`

★ `circle(rayon, angle)` : trace un cercle dans le sens contraire aux aiguilles d'une montre. Le point de départ est situé à la position de la tortue, le centre est situé à une distance donnée par rayon à gauche de la tortue. La portion du cercle tracée dépend de angle (mettre 360 — ou rien du tout! — pour un cercle complet). Si la valeur de rayon est négative, le cercle est tracé dans le sens des aiguilles d'une montre, avec un centre situé à droite de la tortue!

★ `dot(taille, couleur)` : trace un disque dont le diamètre est gouverné par le paramètre optionnel `taille` et la couleur définie par une chaîne de caractères désignant une couleur par son nom (liste des noms de couleurs : <http://wiki.tcl.tk/37701>).

Exemple : `dot(30, "pale green")`

★ `stamp()` : dessine le symbole actuel de la tortue. Ce symbole peut être changé par la commande `shape(nom)`, où `nom` est une chaîne de caractères à choisir parmi "arrow", "turtle", "circle", "square", "triangle" et "classic".

Exemple : `shape("turtle") ; stamp()`

★ `width(taille)` : gouverne la largeur du trait (lorsque le crayon est abaissé).

Exemple : `width(10)`

★ `color(couleur)` : définit la couleur du trait (les couleurs sont définies comme précédemment).

Exemple : `color("DarkOrange1")`

★ `fillcolor(couleur)` : définit la couleur de remplissage.

Exemple : `fillcolor("honeydew3")`

★ `bgcolor(couleur)` : définit la couleur du fond.

★ `begin_fill()` : à exécuter **avant** de tracer une forme dont on veut colorier l'intérieur.

Exemple : `begin_fill() ; goto(75,25) ; stamp() ; goto(0,0) ; stamp() ; forward(100) ; stamp() ; goto(25,75) ; stamp()`

★ `end_fill()` : à appeler **après** avoir tracé une forme dont on voulait colorier l'intérieur.

Exemple : `end_fill()`

★ `mainloop()` : à appeler dans un script, pour empêcher la fenêtre de se fermer toute seule une fois l'ensemble des tracés achevés.

Exemple : `mainloop()`

L'affectation


Un programme informatique contient des instructions qui utilisent des variables prenant diverses valeurs (valeurs numériques, booléens Vrai-Faux ou chaînes de caractères). Le langage Python (comme celui des calculatrices TI) ne requiert pas de déclaration de variables.

L'affectation des variables se fait avec le symbole « = ».

Exemples :

`a = 2` : `a` prend la valeur 2.

`b = 1.5` : `b` prend la valeur 1,5.

 Le séparateur décimal est un point et non pas une virgule.

`c = 2%3` : `c` prend la valeur égale au reste de la division euclidienne de 2 par 3.

`d = 2/3` : `d` prend la valeur du quotient décimal de 2 par 3.

`e = 2*3` : `e` prend la valeur du produit de 2 par 3.

`f = 2**3` : `f` prend la valeur de 2 puissance 3 (double signe `*` pour les puissances).

`somme, n=0, 100` : `somme` prend la valeur 0 et `n` la valeur 100.

VI Demander une valeur à l'utilisateur

La commande `a=int(input("Entrez le valeur de a : "))` affiche le message entre guillemets et stocke la réponse de l'utilisateur (qui doit être un entier à cause de l'instruction `int`) dans la variable `a`.

En langage Python, on doit donc indiquer le type de variable attendu pour la réponse de l'utilisateur.

Voici des types de variables possibles :

- ★ `int` pour les entiers
- ★ `float` pour les nombres décimaux
- ★ `str` pour les chaînes de caractères

VII L'affichage

La commande « `print(...)` » permet l'affichage du contenu entre parenthèses selon les règles décrites ci-dessous :

`print(a)` : Affiche la valeur de la variable `a`.

`print('a')` : Affiche la valeur de la lettre `a` (l'utilisation des guillemets entraîne la copie du message entre guillemets).

`print('a=', a)` : Affiche `a=` puis affiche la valeur de la variable `a`.

VIII Effectuer des calculs

★ 4 opérations classiques : `+` `-` `*` `/` Bibliothèque `math`

★ `a` exposant `b` : `a**b`

★ Racine carrée de `a` : `sqrt(a)`

★ Division entière de `a` par `b` : `a//b`

★ Exponentielle de `a` : `exp(a)`

★ Reste de la division euclidienne de `a` par `b` : `a%b`

★ Logarithme népérien de `a` : `log(a)`

et `cos`, `sin`, `tan...` (angles en radians)

Nombres complexes

★ Définir `z`, nombre complexe : `z=complex(3,2)`

En particulier, définir `i` : `i=complex(0,1)`

★ Valeur absolue, module de `z` : `abs(z)`

★ Partie réelle, partie imaginaire de `z` : `z.real` et `z.imag`

★ Argument de `z` (avec bibliothèque `cmath`) : `phase(z)`

Bibliothèque `random`

★ Nombre aléatoire dans `[0;1[` : `random()`

★ Nombre aléatoire entier entre `a` et `b` inclus : `randint(a,b)`

IX L'instruction conditionnelle

```
0 if condition1 : # Ne pas oublier les deux points.
1     instruction1 # Noter 1 indentation (espace)
2 elif condition2 : # Ce test est facultatif selon le problème
3     instruction2
4 else : # Dernière condition.
5     instruction
```

Exemple : Déterminer la plus petite de deux valeurs a et b entrées par l'utilisateur.

```
0 def pluspetit(a,b):
1     if a<b :
2         return(a)
3     else :
4         return(b)
```

Remarques :

★ Les comparaisons possibles sont :

== : égal à
!= : différent de
> : strictement supérieur à
>= : supérieur ou égal à
< : strictement inférieur à
<= : inférieur ou égal à

★ Il est possible d'affiner une condition avec les mots clé AND qui signifie « ET » et OR qui signifie « OU ».

On veut par exemple tester si une valeur est plus grande que 5 mais aussi plus petite que 10 :

if v>5 and v<10 :

On peut également écrire if 5<v<10 :

Simplification d'écriture

On peut remplacer un test du type if x!=0: par if x:.

Du coup, le test if x=0: peut être remplacé par if not x:

X La boucle conditionnelle

On demande à l'utilisateur de choisir un nombre A (le seuil) et on cherche le premier entier n tel que $2^n > A$ (2^n dépasse le seuil).


```

0 a=float(input("Quel est le seuil choisi? "))
1 n=0
2 u=1
3 while u<a:
4     n=n+1
5     u=u*2
6 print("Seuil dépassé a partir de n=", n)

```

XI La boucle itérative

On cherche à calculer la somme des nombres de la forme a^k pour k allant de 0 à n , où n est un entier naturel non nul choisi par l'utilisateur.

```

0 print("Nous allons calculer la somme des a^k")
1 a=int(input("Quel est l'entier a choisi? "))
2 n=int(input("Quel est l'entier n choisi? "))
3 S=0
4 for k in range(n+1):
5     S=S+a**k
6 print("La somme est", S)

```

Remarques :

★ Python propose une instruction « *for variable in liste :* » qui permet d'exécuter un bloc d'instructions (dont l'ouverture est signalée par le symbole :) en donnant successivement à la variable les différentes valeurs de la liste.

★ Pour retrouver le comportement du bloc répéter 6 fois de Scratch, il suffira d'utiliser l'instruction « *for i in range(6) :* » puisque `range(6)` permet d'itérer sur la liste `[0, 1, 2, 3, 4, 5]`.

★ Plus généralement `range(a, b)` permet d'itérer sur la liste des entiers compris entre `a` (inclus) et `b` (exclu).

Exemple :

Le programme suivant permet de calculer la moyenne des 100 premiers nombres entiers impairs :

```

0 somme, n = 0, 100
1 for x in range(n):
2     somme = somme + 2*x+1
3     moyenne = somme/n

```

XII Les fonctions algorithmiques

A Utilisation simple

Le langage Python utilise fréquemment des **fonctions** pour effectuer les traitements attendus.

Une **fonction** est un programme qui utilise un ou plusieurs paramètres de différents types (numérique, booléen ou chaîne de caractères).

Elle est définie par son nom et ses paramètres : `fonction(paramètre1,paramètre2...)`.

Lorsqu'une fonction est appelée, elle renvoie un résultat numérique, un booléen ou une chaîne de caractères, c'est-à-dire que l'utilisateur lit sur l'écran un résultat.

La syntaxe d'une fonction est du type :

```
0 def fonction(paramètres) :  
1     instructions  
2     return(...)
```

Ces fonctions peuvent renvoyer un message, une(des) valeur(s) numérique(s) ou un booléen (Vrai ou Faux), comme dans l'exemple ci-dessous.

```
0 def affine(a) : # Ne pas oublier les deux points.  
1     y = 3*a-5 # Calcul.  
2     return(y) # Affichage du résultat.
```

Pour calculer l'image de 2 par la fonction `affine()` définie ci-dessus, on appelle cette fonction en tapant la commande : `affine(2)`. On obtient l'affichage : 1.

Remarque :

Une **procédure** est définie comme une **fonction**, mais elle ne comprend pas l'instruction `return`.

B Utilisation plus évoluée

1 Fonction récursive

Une fonction **récursive** est une fonction qui s'appelle elle-même.

L'usage de fonctions récursives fournit un code élégant, mais qui demande un certain effort intellectuel pour comprendre son fonctionnement.

De plus, il faut être attentif aux conditions utilisées : encore plus qu'avec un `while`, le danger d'une boucle infinie est présent.

Exemple : pgcd de deux entiers avec une fonction récursive

À comparer avec le pgcd classique fondé sur une boucle `while`.

```
0 def pgcd_recuratif(a,b):  
1     r = a%b # Reste dans la division euclidienne de a par b  
2     if r==0 :  
3         # Si le reste est nul : on renvoie b  
4         return b
```

```

5     else :# Sinon : on calcule le pgcd de b et r
6         return pgcd_recuratif(b,r)

```

2 Appliquer une fonction sur chaque item d'un élément itérable

On utilise une instruction du type `map(function, [])`

```

0 >>> def add_one(x): return x + 1
1 >>> list(map(add_one, [1,2,3]))
2 [2, 3, 4]

```

XIII Simulation du hasard

On cherche à simuler le lancer cubique à 6 faces dont les faces sont numérotées de 1 à 6 un nombre de fois N choisi par l'utilisateur et à calculer la fréquence d'apparition de la face 6 dans cette série. Pour simuler le choix au hasard d'un entier, on utilise la fonction `randint` de la bibliothèque `random`.

```

0 from random import *
1 n=int(input("Combien de lancers souhaitez-vous? "))
2 S=0
3 for k in range(n):
4     a=randint(1,6)# on tire au hasard un nombre entier
5     #parmi 1,2,3,4,5,6
6     if a==6:
7         S=S+1
8 print("La fréquence des 6 sur cet échantillon")
9 print("de ",n," lancers est ",S/n)

```

La fonction `choice([])` de la bibliothèque `random` retourne une valeur d'une liste aléatoirement.

```

0 >>> import random
1 >>> random.choice([1,2,3,4,5])
2 3
3 >>> random.choice([1,2,3,4,5])
4 2
5 >>> random.choice(['x','abcde','hello','yes!'])
6 'yes!'

```

XIV L'utilisation de listes

- ★ Initialiser une liste L vide : `L = []`
- ★ Initialiser une liste L contenant 6 termes égaux à 0 : `L = 6*[0]`

- ★ Affiche la valeur d'indice i de la liste L (le premier indice est 0) : $L[i]$
 - ★ Affecter la valeur a à la valeur d'indice i de la liste L (le premier indice est 0) : $L[i]=a$
 - ★ Ajouter l'élément x au bout de la liste L : $L.append(x)$
 - ★ Ajouter l'élément x au rang i de la liste L : $L.insert(i,x)$
 - ★ Supprimer la première occurrence de x dans la liste L : $L.remove(x)$
 - ★ Supprimer l'élément d'indice i dans la liste L : $L.pop(i)$ ou $del L[i]$
 - ★ Longueur d'une liste L : $len(L)$
 - ★ Trier une liste L et stocker le résultat dans L_triee : $L_triee = sorted(L)$
 - ★ Inverser l'ordre des éléments de la liste L : $L.reverse()$ ou $L[::-1]$
- ⚠ Il y a une différence notable entre les deux solutions :
- $L.reverse()$ modifie la liste L (ainsi que les autres listes qui lui sont égales), alors que $M=L[::-1]$ ne modifie pas la liste L .
- Au lieu de $M=L[::-1]$, on peut également utiliser $M=L[:]$ ou $M=L.copy()$
- ★ Compter le nombre d'occurrences de a dans la liste L : $L.count(a)$
 - ★ $L[::3]$ renvoie une liste formée des termes de L de rang multiple de 3.
 - ★ $L[::2]$ renvoie une liste formée des termes de L de rang multiple de 2.

Exemple

```

0  def syracuse(n):
1      L = [n]
2      while (n!=1):
3          if (n%2==0): # Test de parité
4              n = (n//2) # Division entière
5          else:
6              n = n*3+1
7              L.append(n)
8      return L
9
10 import matplotlib.pyplot as plt
11
12 def graph_syracuse(nombre):
13     Y_liste = syracuse(nombre) # On remplit la liste Y_liste
14     # avec les valeurs de la suite de Syracuse
15     X_liste = [] # Initialisation de la liste des abscisses
16     for i in range(1,len(Y_liste)+1): # Affectation des
17         #indices de la suite
18         X_liste.append(i)
19     plt.plot(X_liste,Y_liste,"b.") # Création du graphique et
20     #affichage

```

XV Les graphiques de fonctions mathématiques

La bibliothèque matplotlib et son module pyplot sont conçus de manière à tracer des graphiques point par point de façon rapide... à condition de se plier à leur pratique.

Soit à tracer la parabole d'équation $y = x^2$ sur l'intervalle $[-2;2]$, avec un pas de 0.001 : quatre milliers de points suffisent à faire une étude comparative de vitesse de tracé, à l'aide de la bibliothèque time.

Pour pouvoir tracer des courbes de fonctions mathématiques nous allons donc charger le module pyplot de la bibliothèque matplotlib.

Comme c'est un peu long, on peut modifier le nom avec l'instruction

```
import matplotlib.pyplot as plt.
```

L'instruction plot de matplotlib.pyplot est optimisée pour tracer des graphiques utilisant des listes.

Il faut raisonner en termes de « tableau de valeurs » :

★ X et Y sont deux listes de même longueur : par exemple [1,2,3] et [1,4,9].

★ `plt.plot(X,Y,".b")` trace l'ensemble des points $(X[i],Y[i])$: (1,1), (2,4) et (3,9).

L'essentiel du programme consiste donc à alimenter les deux listes X et Y, à l'aide de la commande append.

```
0 import matplotlib.pyplot as plt
1
2 def courbe(f,a,b):
3     X = []
4     Y = []
5     for k in range(a,b+1) :
6         X.append(k)
7         Y.append(f(k))
8     plt.plot(X,Y,"b-")#trace la courbe en bleu, les points
9     #etant relies
10    plt.show()
11
12
13 def courbe_pas(f,a,b,pas):
14     X = []
15     Y = []
16     while a <= b :
17         X.append(a)
18         Y.append(f(a))
19         a = a + pas
20    plt.plot(X,Y,"g-")#trace la courbe en vert, les points
21    #etant relies
22    plt.show()
```

```

23
24
25 import time #pour calculer le temps d'exécution du programme
26 def trace_courbe_fct(f,a,b,pas):
27     depart=time.perf_counter()
28     X = [] # Initialisation des listes
29     Y = []
30     while a<=b:
31         X.append(a) # Ajout des valeurs
32         Y.append(f(a)) # au "bout" de X et Y
33         a = a+pas
34     # Trace de l'ensemble du tableau de valeurs
35     plt.plot(X,Y,"r.")#trace la courbe en rouge, les points
36     #n'etant pas relies
37     plt.show()
38     fin=time.perf_counter()
39     return "Temps : " + str(fin-depart) + " s."

```

Quelques instructions

- ★ Placer un point de coordonnées (x;y), en bleu, gros : `plt.plot(x,y,"bo")`
- ★ Tracer un segment entre deux points A et B, rouge : `plt.plot([xA,xB],[yA,yB],"r")`
- ★ Tracer les points dont les coordonnées sont données par les listes X_liste et Y_liste, en noir, en reliant les points : `plt.plot(X_liste,Y_liste,"k-")`
- ★ Pour avoir un repère orthonormé : `plt.axis("equal")`
- ★ Pour une grille : `plt.grid()`
- ★ Afficher la fenêtre graphique : `plt.show()`

Couleurs et style

b	: bleu	-	: ligne continue
g	: vert	-	: tirets
r	: rouge	:	: pointillés
c	: cyan	.	: points
m	: magenta	o	: billes
y	: jaune	x	: croix
k	: noir	v	: triangles
w	: blanc	-.	: point-tirets