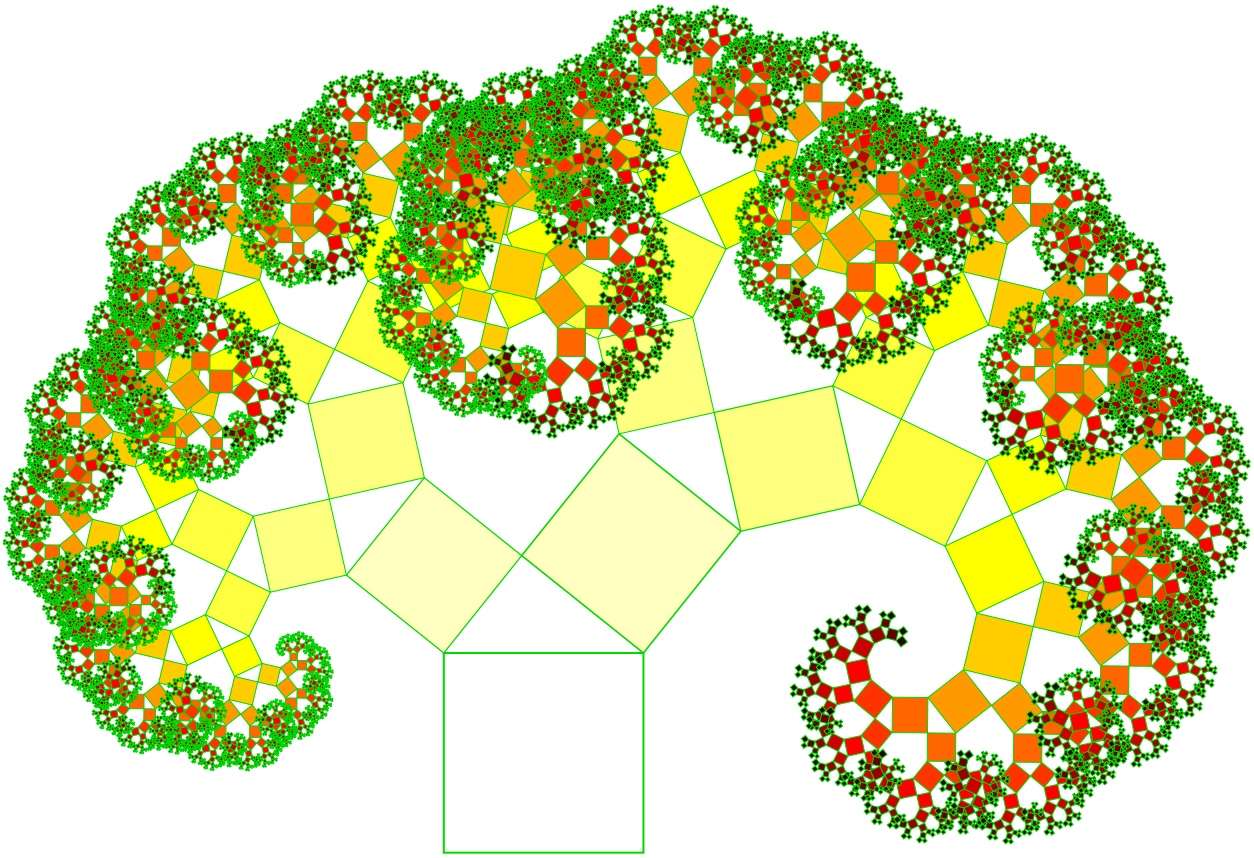
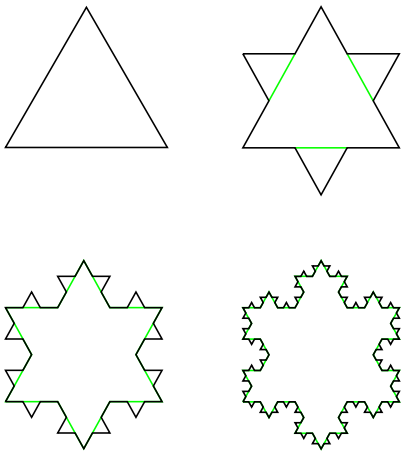
Initiation à l'algorithmique avec Python 3

Franck CHAMBON

23 janvier 2011



Ci-dessus, l'arbre de PYTHAGORE, ci dessous le flocon de VON KOCH



```
from turtle import *
T=[1,-2,1,0]
def koch(x,n):
    if n==0:
        forward(x)
    else:
        for i in T:
            koch(x/3, n-1)
            left(60*i)
for _ in range(3):
    koch(300, 5)
    right(120)
mainloop()
```

Table des matières

Initiation à Python 3

I] Avant propos

Python est un langage de programmation, très répandu et idéal pour débuter. Le langage est sous *licence libre* ; l'utilisation est et restera *gratuite*. Python est multiplateforme ainsi vous pourrez l'utiliser avec Windows, Mac/OS, ou GNU/Linux. *Il existe deux branches actives :* Python 2.x et Python 3.x.

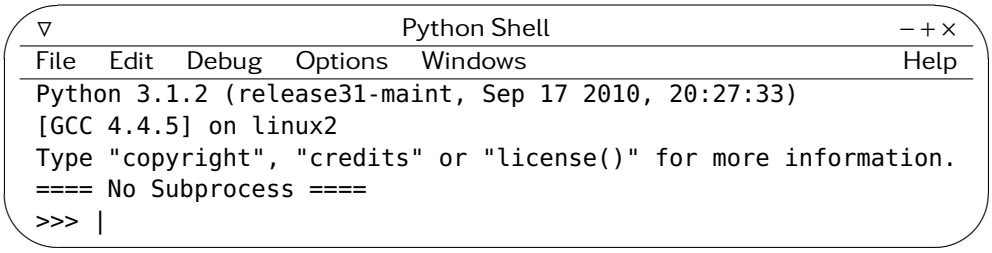
Pour s'initier à l'algorithmique nous choisissons Python 3

I-a) Installation de Python

Installation sous GNU/Linux
Avec votre gestionnaire de paquet, installez le paquet **idle3**, c'est tout. *Félicitation.*

Installation sous Windows
Sur le site www.python.org, vous trouverez le Windows Installer, fichier de 14Mo environ, c'est très peu. *Installation par défaut.*

Pour ouvrir Python, lancer le programme `IDLE3`. Vous devriez obtenir presque ceci :



I-b) Amélioration de l'ergonomie

Le choix par défaut de la fonte et de sa taille est mauvais, cliquer sur options, puis Configure IDLE... .
Choisir dans la liste des fontes : `DejaVu Sans Mono`, une belle fonte mécane, **Mono** est important.
Avec une taille de 16, pour commencer, vous pourrez réduire ensuite.

Ensuite, dans l'onglet General, section Autosave Preferences choisir **No Prompt**, pour At Start of Run (F5)
Cela vous permettra de sauver/compiler/lancer vos programmes en une seule touche : F5 .

II] Premières opérations

II-a) Les quatre opérations et les puissances

```
>>> 5+3          # Un commentaire, il débute avec sharp # (AltGr+3) au clavier
8

>>> 35 - 2*10    # Espaces optionnelles. Priorités respectées
15

>>> (35-2)*10
370

>>> 23/4     # !!! attention avec Python2, le résultat était tronqué à 5 !!!
5.75

>>> 23:4     # Erreur de syntaxe, ":" n'est pas l'opérateur de la division !!!
SyntaxError: invalid syntax

>>> 23//4    # Si vous voulez le résultat entier tronqué
5
```

```
>>> 5**2    # 5 au carré, attention dans d'autres langages, on note 5^2
25

>>> 2*5**3  # Les puissances sont prioritaires, on obtient alors 2*(5**3)=2*125=250
250         #    ( et non 10**3=1000  )

>>> 55.6 + 21.13  # Dans le monde anglo-saxon, le séparateur décimal est le point .
76.73             # On ne doit pas utiliser la virgule pour un nombre décimal

>>> 55,6 + 21,13     # Une erreur sémantique, (vous donnez un autre sens)
(55 , 27 , 13)       # sauf si vous désirez la liste (55 , 6+21 , 13)
```

Résumé des opérations de bases

Avec Python 3 :

+	addition
-	soustraction
*	multiplication
/	division
**	puissance
//	division entière
%	reste de cette division

Particularités

Les priorités sont respectées
Les espaces autour des opérations : une option
Le séparateur décimal est le point

Divisions entières

```
>>> 147//9  # réponse : le quotient
16
>>> 147%9   # réponse : le reste
3
```

Ainsi : 147 divisé par 9 donne 16, et il reste 3.

II-b) L'écriture scientifique

Avec Python, on peut entrer des nombres en écriture scientifique, suivant le modèle :
Pour $1{,}3\times10^5$, on entre :

```
>>> 1.3e5   # réponse :
130000.0
```

D'accord, mais pourquoi `.0` ?
C'est que Python connait différents **type**s de nombres, il y a les nombres entiers (***int**egers*), et les nombres réels à virgule flottante (***float**ing point numbers*).
Une écriture scientifique sera toujours de `type float`, donc Python l'indique avec « `.0` »
On reviendra sur la notion de `type`.

Mini-problème

La masse d'un atome de carbone est environ 2×10^{-26} kg, celle de l'oxygène est environ $2{,}7\times10^{-26}$ kg.
Quelle est la masse du dioxyde de carbone (CO_2) ?

```
>>> 2e-26 + 2*2.7e-26 #Réponse :
7.4e-26
```

La molécule de CO_2 a une masse de $7{,}4\times10^{-26}$ kg.

Un réservoir de voiture contient 37 kg de carburant, et la combustion génère environ $1{,}6\times10^{27}$ molécules de CO_2.
À chaque plein de carburant, quelle masse de CO_2 a été envoyée dans l'atmosphère ?

```
>>> 1.6e27*7.4e-26
118.4 # kg àchaque plein !!!
```

III] Premières affectations

Python possède des « mémoires », que l'on appelle des *variables*.

III-a) L'affectation

```
>>> a=12  # Ceci est une affectation, on "met" 12 dans 'a'.    a ← 12

>>> b=5+a # Ceci est un calcul (5+12 qui donne 17), suivi de l'affectation b ← 17

>>> a=9   # Ceci est une réaffectation, a ← 9, maintenant a ≠ 12
```

Conseils

1. Bien que l'affectation multiple soit possible, on déconseille cela aux débutants
   ```
   >>> a = b = 7        # qui donne  a ← 7 et b ← 7
   ```
2. Bien que l'affectation parallèle soit possible, on déconseille cela aux débutants
   ```
   >>> c, d = 8, 9      # qui donne  c ← 8 et d ← 9
   ```
   Mais bientôt, on vous conseillera de l'utiliser !!!

III-b) L'utilisation

Une fois qu'une valeur est « stockée » dans une variable, on peut l'utiliser.

```
>>> a=174.26     # Une affectation simple
>>> a            # Un appel
174.26      ⟵ la réponse, quelle surprise
>>> a**2 - 3*a  # 'a' au carré, moins le triple de 'a'
29843.7676      ⟵ le calcul de a² – 3a
```

III-c) Les noms de variables

On ne peut pas choisir n'importe quel nom pour une variable. De manière synthétique, on a :

Obligatoire : Débuter par une lettre et n'utiliser que les caractères : `a→z` , `A→Z` , `0→9` et `_`
Conseil : Débuter par une minuscule.
Exemples : `nombre_d0r` , `a15` sont des noms de variable valides.
Remarques : `niter, nIter, NITER` sont trois variables différentes.
On dit que Python est sensible à la casse ; les majuscules comptent.
Avec Python 3, les lettres accentuées sont autorisées pour les noms de variables.

III-d) Les 33 mots réservés

Interdiction de les utiliser en nom de variable

```
and     as      assert  break   class   continue  def
del     elif    else    except  False   finally   for
from    global  if      import  in      is        lambda
None    nonlocal not    or      pass    raise     return
True    try     while   with    yield
```

Chacun de ces mots a une utilisation que nous découvrirons, il ne faut pas les mélanger avec les variables.

Exemple : détruire (***del****ete*) une variable

```
>>> del nombre_d0r #Détruit l'instance nombre_d0r
```

Remarque : `True` et `False` (vrai et faux) sont les seuls mots réservés commençant par une majuscule.

III-e) Python, un langage à typage dynamique

On a déjà vu des types de variables : `int` pour entier (*integer*) et `float` pour nombre à virgule flottante (*floating point number*). Il existe d'autres types, comme `string` pour chaîne de caractères.
On peut changer le type d'une variable en la réaffectant, on dit que Python est un langage à typage dynamique.

```
>>> a=7     # Une affectation qui donne à 'a' le type 'int' pour integer (entier)
>>> type(a)   # Pour connaître le type d'une variable
<class 'int'>  # Réponse : 'int'
>>> a="bonjour" # Une réaffectation qui modifie le type de 'a' en 'str'
     # 'str' pour string (chaîne de caractères)
     # C'est interdit avec d'autres langages (ex. C++) dont le typage est statique
```

On en profite pour signaler qu'une méthode pratique pour entrer une chaîne de caractères est de l'insérer entre une paire de " (guillemet droit double).

```
>>> phrase1 = "Exemple de chaîne de caractères"#  C'est la méthode que vous devez retenir.
>>> phrase2 = 'Un autre exemple'              # Certes le guillemet droit simple est accepté, mais
```

```
FAUX : >>> phrase3 = 'L'erreur'        # c'est le même caractère que l'apostrophe dactylographique.
''        # Une chaîne vide,
""     # une autre chaîne vide.
"""Du "texte" incluant des " et ', étonnant?"""     # Situation extrême : paire de triple double
```

III-f) Affectation ≠ égalité

En pseudo-langage algorithmique, l'affectation se note ←, comme $a \leftarrow 7$.
En informatique, l'affectation se note souvent = , ainsi

```
>>> a=7 # Affectation classique,    a ← 7
>>> a=a+8 #Affectation légale,    a ← a + 8 = 7 + 8 = 15
```

En mathématiques, $a = a + 8$ est une équation qui n'a pas de solution réelle, elle signifierait $0 = 8$.
Ainsi, on veillera à ne pas confondre le = d'affectation en informatique, et le = en mathématiques.

Il faut toutefois disposer d'un symbole pour vérifier une égalité.
En informatique le « test d'égalité » se note souvent ==

```
>>> 10==8+2 # Un test d'égalité, qui répond True (vrai)
True
>>> 11==8+2 # Réponse : False (faux)
False
```

III-g) Structurer ses instructions

L'ordre dans lequel on effectue les affectations est important.

```
1 >>> a=7  # a ← 7, 'a' est affecté à 7
2 >>> b=8  # b ← 8, 'b' est affecté à 8
3 >>> a=b  # a ← b, 'a' est affecté à 'b' qui est égal à 8, donc a = 8
4 >>> b=a  # b ← a, 'b' est affecté à 'a' qui est désormais égal à 8, donc b = 8
5 >>> a, b # Pour afficher les valeurs de 'a' et 'b'
6 (8,8)    # ⟵ La réponse. Réessayer en intervertissant les lignes 3 et 4 !!!
```

III-h) Utilisation en mathématiques au Collège

Python est capable de répondre à de nombreux problèmes numériques au Collège.

Exemple 1 Calculer $A = 3x(5x-2)+6x(5+3x)$, avec $x = 2$, $x = 12$ et $x = 103$.
Solution avec Python :

```
>>> x=2                          # Pour le premier calcul
>>> 3*x*(5*x-2)+6*x*(5+3*x)   # Les multiplications doivent être explicites
180         # La réponse pour x=2
```

Pour les autres calculs, on modifie x, puis on refait le calcul. Il est possible de faire un copier-coller.

Exemple 2 Tester l'égalité $3x + 7 = 12x - 9$ pour $x = 5$, $x = 3$ et $x = 3{,}5$

```
>>> x=5              # x ← 5, pour le premier test avec x=5
>>> 3*x+7==12*x-9    # == pour le test d'égalité
False   # La réponse est 'faux'
```

Pour les autres tests, on modifie x, on peut copier-coller le test.

C'est juste et rapide, mais il n'y a pas de justifications...
Pour une liste de valeurs plus longue, nous ferons un programme...

Contre-exemple : Pour travailler avec les fractions, on ne peut pas avoir directement les résultats exacts, il faut charger un *module* fractions. Utilisation de **import**. De manière générale, pour travailler avec des valeurs exactes, on utilisera un CAS (*Computer Algebra System*), comme SAGE ou XCas. Utilisation à partir du lycée.

IV] Les Fonctions

IV-a) Première fonction mathématique

Avant de faire de la programmation, montrons comment ajouter des fonctionnalités à Python.

Python n'est qu'un noyau, auquel on peut ajouter des programmes qui ont été écrits par des tiers.
Il y a, par exemple, un module **math** qui a été écrit pour apporter de nombreuses fonctions mathématiques usuelles, comme la racine carrée.

En informatique, la racine carrée (*square root*) est souvent notée `sqrt`. Pour demander à Python de charger (`import`) la fonction `sqrt` à partir (`from`) du module `math`, on entre :

```
>>> from math import sqrt
```

À partir de maintenant, on peut faire ces calculs :

```
>>> sqrt(169)
13
>>> sqrt(9876543210)
99380.79900061178
```

Si votre calculatrice bloque avec des nombres trop grands, pensez à Python, et vous aurez peut-être la réponse.

IV-b) Premiers affichages

Pour afficher (*print*) un résultat, en mélangeant texte et nombre, suivre ce modèle pour commencer :

```
>>> x = 12 # Affectation    x ← 12
>>> calculA = 3*x*(5*x-2)+6*x*(5+3*x)   # Calcul de A, et affectation dans 'calculA'
>>> print("Pour x =", x, ", on a : A =", calculA)
Pour x = 12 , on a : A = 5040
```

`print` est une fonction qui peut prendre plusieurs arguments, séparés par des virgules ; on peut faire se succéder des chaînes de caractères et des nombres. Pendant la saisie, une aide contextuelle apparaît.

Avec Python 3, `print` est une fonction comme les autres, avec la version 2, elle avait un rôle spécial. Nous verrons les possibilités avancées de `print` plus tard.

IDLE3 possède une coloration syntaxique qui vous permet de bien distinguer les chaînes de caractères du reste, ainsi que les fonctions usuelles et les mots réservés.

IV-c) Premières créations de procédures

Nous reprenons les exemples de III-h) pour automatiser des instructions.
Exemple 1 Calculer $A = 3x(5x-2)+6x(5+3x)$, avec $x = 2$, $x = 12$ et $x = 103$.

```
>>> def afficheA(x):
...     print("L'expression A = 3x(5x-2) + 6x(5+3x)  dépend de x.")
...     print("Pour x =", x, ", A = ", 3*x*(5*x-2)+6*x*(5+3*x))
                                          # /!\ ligne vide
>>> afficheA(2)
L'expression A = 3x(5x-2) + 6x(5+3x)  dépend de x.
Pour x = 2 , A = 180
>>> afficheA(12)
L'expression A = 3x(5x-2) + 6x(5+3x)  dépend de x.
Pour x = 12 , A = 5040
```

Explications ligne par ligne :

1. On définit (`def`), une procédure `afficheA`, qui prendra un argument `x`, on finit par `:` (mais que fait `afficheA` ?)
2. Le curseur a été **indenté** (*décalé*) automatiquement.
   `afficheA` commence par écrire une simple chaîne de caractères, sans aucun calcul.
3. Ensuite, avec la même indentation, `afficheA` affiche des bouts de phrases avec le résultat du calcul demandé.
4. Une ligne vide implique que la définition de `afficheA` est finie.
5. On utilise `afficheA` avec un argument, on peut la réutiliser autant de fois qu'on veut.

Exemple 2 Tester l'égalité $3x+7=12x-9$ pour $x=5$, $x=3$ et $x=3{,}5$

```
>>> def exemple2(x):
...     if 3*x+7==12*x-9 :
...         print("L'égalité 3x+7=12x-9 est vérifiée pour x = ", x)
...     else
...         print("L'égalité 3x+7=12x-9 n'est pas vérifiée pour x = ", x)
...
>>> exemple2(5)    # Puis 3, puis 3.5
L'égalité 3x+7=12x-9 n'est pas vérifiée pour x = 5  # La réponse
```

Explications ligne par ligne :

1. On définit `exemple2` (il faut choisir des noms de circonstance), qui prend un argument `x`. Ne pas oublier `:`
2. On débute le test SI (*if*). **if** <le test> `:` Ne pas oublier de finir par `:`
3. Ici, si <le test> est vrai (*True*), alors on affiche que l'égalité est vérifiée, avec la valeur de x.
4. Sinon (*else*)
5. Cette ligne de code n'est exécutée que si <le test> à échoué. On affiche que l'égalité n'est pas vérifiée.
6. Ligne vide pour finir la définition de `exemple2`.
7. Utilisation de `exemple2`, avec l'argument `5`
8. La réaction de la procédure `exemple2`.

Le décalage dans les structures `def`, `if` et `else` est obligatoire. On l'appelle l'indentation. Par tradition, on utilise 4 espaces par niveau. Python interprète alors correctement leur début et fin.

D'autres langages utilisent des balises comme `begin` et `end`, ou `{` et `}` ce qui rend le code moins lisible et plus long. L'indentation est une bonne pratique, en Python elle est obligatoire.

IV-d) Première création de fonction

Nous sommes heureux d'avoir pu automatiser les exemples 1 et 2, nous avons eu un affichage correct, mais nous ne pouvons pas réutiliser automatiquement les résultats obtenus ; ils n'ont pas été stockés.
En fabriquant une fonction (*une procédure qui renvoie une valeur*), on peut réutiliser le résultat.

Exemple 1 : définition de la fonction calculA : $x \mapsto 3x(5x-2)+6x(5+3x)$

```
>>> def calculA(x):
...     return 3*x*(5*x-2)+6*x*(5+3*x)

>>> calculA(103)%10  # le reste de la division par 10, donne le chiffre des unités.
9
```

Explications ligne par ligne :

1. On définit (**def**) la fonction `calculA` par `:`
2. Indentation. On renvoie (*return*) le résultat du calcul
3. Ligne vide ; fin de la définition de `calculA`
4. Utilisation : calcul pour $x=103$, on demande le reste de la division par 10
5. Le résultat est affiché, c'est le dernier chiffre de calculA(103).

Premiers exercices

Avant d'aller plus loin, il faut s'exercer à manipuler les mémoires.
Affectation, réaffectation, fonction sont au programme.

1. Lire des instructions et comprendre le mécanisme.
2. Savoir appliquer manuellement des instructions.
3. Créer ses propres instructions.

Faire les exercices Python 3 sur la feuille n° 1 et la feuille n° 2.

V] L'environnement de travail

Pour un exercice rapide comme précédemment, on utilise une console (*shell*) Python.
Notre travail ne sera pas enregistré sur l'ordinateur.

Pour un projet plus long, que l'on souhaite retravailler, on utilise un IDE (*Integrated Development Environment*), pour débuter avec Python, l'IDE d'origine est parfait. On peut éditer (écrire/modifier) un programme dans un fichier et l'exécuter (le lancer — *run*).

Les premières étapes à effectuer :

1. Ouvrir une **n**ouvelle feuille avec l'éditeur, avec Ctrl + N .
2. Écrire un programme en python. (à suivre)
3. **S**auvegarder (*Save*) le code, avec Ctrl + S .
   Choisir l'extension `.py`. Est-elle automatique ?
4. Vérifier la syntaxe, avec Alt + X .
5. Lancer (*Run*) votre programme avec F5

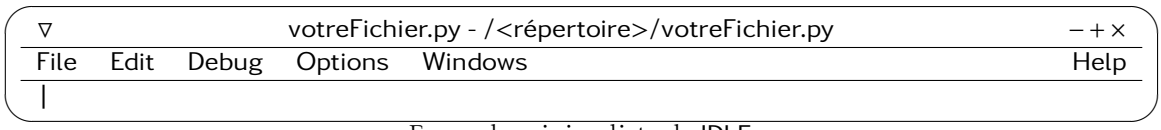
Remarque : Pour sauvegarder votre fichier, choisir un répertoire dans lequel vous avez des droits d'écriture. Avec une installation en réseau, ce n'est pas le cas par défaut.

pendant la construction

Alt + X pour vérifier le code

F5 pour sauver et exécuter le programme.

Alt + ⇆ pour alterner entre vos fenêtres.



Exemple minimaliste de IDLE.

La décoration variera en fonction de votre gestionnaire de fenêtre.
Il est conseillé de laisser l'extension par défaut `.py` à vos fichiers.
Ils pourront ensuite être exécutés facilement dans un terminal.

V-a) Premier programme : *Hello World*

Tradition oblige, notre premier programme saluera le monde.

Ouvrir un nouvelle fenêtre IDLE — Ctrl + N
Sauvegarder sous le nom `HelloWorld(.py)` — Ctrl + S
Taper le code suivant :

```
1 print("Hello World!")
2 nom=input("Comment t'appelles-tu, toi ? >")
3 print("Bien le bonjour",nom)
```

Résultat dans le shell :

```
Hello World!
Comment t'appelles-tu, toi ? >Cémoi
Bien le bonjour Cémoi
```

Explications :

1. On affiche une première ligne, facile !
2. On demande une entrée (*input*), on affecte le résultat dans `nom`.
3. On affiche les deux chaînes de caractères à la suite.

Utilisation d'`input`

`input` est une fonction qui renvoie toujours une chaîne de caractères. Ainsi :

```
1 nombre=input("Donne un nombre >")
2 print("Son carré est :",nombre*nombre)
```

provoquera une erreur ; on ne multiplie pas les chaînes de caractères entre elles.

Changer le type d'une variable

Les types déjà rencontrés :
`'str'` pour les chaînes de caractères
`'int'` pour les entiers
`'float'` pour les nombres à virgule flottante.

```
1 réponse=input("Donne un nombre >")
2 nombre=float(réponse)
3 print("Son carré est :",nombre*nombre)
```

fonctionnera correctement.

`float(.)`, `int(.)`, `str(.)` transforment dans la mesure du possible leur argument dans le type choisi.

VI] Les tableaux

VI-a) Définition, première utilisation

```
>>> tablo = [100, 101, 102, 103]
>>> len(tablo)
4    # La réponse
>>> tablo[0]
100  # La réponse
>>> tablo[len(tablo)-1]
103  # La réponse
>>> tablo[len(tablo)]
IndexError: list index out of range
```

Les indices vont de `0` à `len(.)-1`

Explication ligne par ligne

1. Un tableau : valeurs entre crochets, séparées par des virgules.
2. `len(.)` renvoie la longueur (*length*) du tableau.
3. Ici 4 valeurs pour `tablo`.
4. On demande la valeur d'indice 0 du tableau.
5. 100 est la première (indice 0) valeur du tableau.
6. On demande la valeur d'indice $4-1=3$ du tableau.
7. 103 est la dernière (indice 3) des 4 valeurs du tableau.
8. On demande la valeur d'indice 4 du tableau.
9. Une erreur d'indice est annoncée, `tablo[4]` n'existe pas.

VI-b) Tableaux automatiques : `range`

Python 3 dispose de tableaux pré-remplis virtuellement, ils ne prennent presque aucune place en mémoire.

`range(100)` est un tableau virtuel à 100 éléments, de 0 à $99 < \mathbf{100}$
`range(100)`$\leftrightsquigarrow$ `[0,1,2,... 99]`.

`range(5,100)` est un tableau virtuel à $100-5=95$ éléments, de 5 à $99 < \mathbf{100}$
`range(5,100)`$\leftrightsquigarrow$ `[5,6,7,... 99]`.

`range(0,100,7)` est un tableau virtuel à 14 éléments, de 0 à $14\times 7 = 98 < \mathbf{100}$
`range(0,100,7)`$\leftrightsquigarrow$ `[0,7,17,... 98]`.

La syntaxe générale est `range([start,]stop[, step])`
Il y a deux arguments optionnels, on ne peut pas choisir le deuxième sans le premier.

On retrouve très souvent `range(.)` dans les structures en boucle `for`.

VII] Les boucles

L'outil de base en programmation, pour répéter, est la boucle. Il y a les boucles décrivant un tableau : boucle **`for`**.
Il y a les boucles fonctionnant suivant une condition : boucle **`while`**.

VII-a) Boucles `for`

Pour $x \in [\![0;3]\!]$ calculer et afficher $5x^2+3x-1$ se traduit par **`for x in [0,1,2,3]:`** calculer...

Comme dans l'exemple réel :

```
>>> for x in [0,1,2,3]:
        print(5*x**2 + 3*x - 1)

-1
7
25
53
```

Ou avec `range(.)`

```
>>> for x in range(4): # de 0 à 3<4
        print(5*x**2 + 3*x - 1)

-1
7
25
53
```

Exemple : Tester ceci dans un fichier.

```
1 réponse = input("Donne un entier positif : ")
2 monEntier = int(réponse) # Pour obtenir un entier
3 for x in range(monEntier+1): # x, z, c ou truc ...
4     print("Si x =",x,"alors -x =",-x)
5     print("Si x =",x,"alors (-x)^2 =",(-x)**2)
6     print("")  # Une ligne vide sera affichée
7 print("Nous avons fait", monEntier+1, "boucles,")
8 print("nous le savions dès la ligne 2,")
9 print("ceci ne sera affiché qu'une fois")
```

Dans un programme, les boucles sont délimitées par des parties indentées.
La boucle *for* commence en ligne 3, et se termine en ligne 6.
L'indentation est une spécificité de Python, cela offre un code clair et court.

VII-b) Boucle `while`

Une boucle qui tourne **tant que** (*while*) une condition est remplie se construit suivant le modèle :

```
>>> while condition :
        instruction 1
        instruction 2...
```

```
1 ça_repart = True # ça_repart est un booléen, ici vrai
2 while ça_repart: # pas de cédille avec python2 !!!
3     réponse = input("Donne un nombre entre 10 et 20 :")
4     nombre = float(réponse)
5     ça_repart = (nombre<10 or nombre>20)
6 print("Il manque",20-nombre,"pour arriver à 20")
```

Explications ligne par ligne :

1. On crée une variable booléenne (vrai ou faux — `True or False`), ici nommée `ça_repart` affectée à `True`, ainsi la boucle pourra se faire au moins une fois.
2. Tant que (`while`) `ça_repart` est vrai (`True`), alors `:`
3. On demande un nombre, mais on obtient une chaîne de caractères.
4. On transforme cette chaîne en nombre à virgule (`float`).
5. On teste si le nombre est trop petit ou (`or`) trop grand, (on voulait un nombre entre 10 et 20)
   si oui alors `ça_repart` ← `True`, sinon `ça_repart` ← `False` (alors le nombre est bon).

   La boucle a fait un tour, si `ça_repart` est vrai, on recommence la boucle.
   Si `ça_repart` est faux, on passe à la suite du programme.
6. Fin de l'indentation, la boucle est finie, on écrit un petit message avec un calcul.

Avec tout ça on peut faire les exercices sur la feuille n° 3 et la feuille n° 4.

VIII] Le module `turtle`

Afin de travailler sur des algorithmes avec des boucles, un moyen intéressant est de construire des figures ; c'est plus visuel que des suites de nombres. On peut obtenir des figures simples mais aussi des fractales.
Python 3 possède un module de dessin adapté à l'apprentissage de l'algorithmique : la tortue (*turtle*).
On lui donne des instructions graphiques : avance ou recule, tourne à gauche ou à droite... et laisse une marque lors de son passage.

VIII-a) Premières instructions

Découverte du module `turtle`

```
up() # relève le crayon
forward(50) # avance de 50 pixels
left(60) # tourne à gauche de 60°
down() # repose le crayon
circle(+100, 37) # trace un arc de cercle
# vers la gauche (+), de rayon 100 pixels,
# et d'angle 37°.
circle(-50, 180) # arc à droite (-),
# rayon 50, demi-tour (180°)
pensize(5) # change l'épaisseur du
# crayon, ici 5 pixels
right(25) # tourne à droite de 25°
backward(30) # recule de 30 pixels
towards(0,0) # tourne en direction de 0.

mainloop() # pour finir avec la tortue
```

Rangés par catégorie :

- `left(x)` tourne à gauche de x degrés.
- `right(x)` tourne à droite de x degrés.
- `up()` `down()` lève ou pose le crayon.
- `forward(x)` avance de x pixels.
- `backward(x)` recule de x pixels.
- `circle(±R, x)`! trace un arc de x degrés à gauche (+) ou à droite (-) de rayon R.
- `circle(±R)` pour le cercle complet.
- `pensize(x)` ↦ l'épaisseur du trait à x pixels.
- `speed(x)` change la vitesse de la tortue, x de `1` (lent) à `10` (rapide).
  Ou `speed(0)` ultra rapide, pas d'animation.

Avec ces instructions, on peut découvrir visuellement le fonctionnement des boucles.
Vous pouvez faire les exercices de la feuille n° 5.

VIII-b) Autres instructions pour `turtle`

Créons une fonction `segment(...)` qui enregistre les paramètres de la tortue, trace ailleurs un segment, et rétablit les paramètres.

```
def segment(x1, y1, x2, y2, ma_couleur):
    départ = pos() # enregistre la position de départ, pour la rétablir en suite
    orientation = heading() # enregistre l'orientation de la tortue
    posé = isdown() # dit si le crayon était posé ou non
    ancienne_couleur = pencolor() # renvoie la couleur du crayon
    # c'est parti
    up() # on relève
    setpos(x1, y1) # on va en (x1, y1)
    pencolor(ma_couleur) # couleur demandée
    down() # on pose le crayon
    dot()  # marque un petit disque
    setpos(x2, y2) # on va en (x2, y2)
    dot() # un autre
    # le segment est tracé !
    up() # on relève
    setpos(départ) # on retourne au point de départ
    setheading(orientation) # on oriente la tortue comme avant
    pencolor(ancienne_couleur)
    if posé:   # s'il était posé,
        down() # on repose le crayon


# Utilisation
circle(20, 127) # Un arc de cercle débuté
segment(44, 55, 144, 155, "red") # Un segment rouge est tracé
circle(20, 53) # l'arc de cercle se poursuit normalement.

mainloop()
```

À retenir : `pos` : position, `heading` : orientation, `set` : placer, `pen` : crayon, `color` : couleur.
`towards(M)` renvoie la direction du point M, ainsi `setheading(towards(M))` tourne dans la direction de M.

```
def carré_Plein(a , ma_couleur):
    fillcolor(ma_couleur)
    begin_fill()
    for _ in range(4):
        forward(a)
        left(90)
    end_fill()
    left(45)
    forward(1.4142*a)
    write("Carré")

# Utilisation
if not(isvisible()):
    showturtle()
carré_Plein(50, "blue")
hideturtle()

mainloop()
```

Les instructions placées entre `begin_fill()` et `end_fill()` définissent une région qui sera remplie avec la couleur choisie.

`fillcolor("blue")` prépare le remplissage en bleu.

Pour choisir une couleur on peut utiliser le système (r,g,b) : (*red*, *green*, *blue*).
`ma_couleur = (r, g, b)` où `r, g, b` sont des nombres entre `0` et `1`.
`ma_couleur = (0, 0, 0)` rien, c'est noir.
`ma_couleur = (1, 0, 0)` que du rouge.
`ma_couleur = (1, 1, 0)` rouge et vert : jaune
`ma_couleur = (1, 1, 1)` rouge, vert et bleu : blanc
C'est une méthode additive des couleurs, pas comme avec des tubes de peintures où les couleurs se soustraient.

`isvisible()` revoie **`True`** ou **`False`**.
`show-` (montre) et `hide-` (cache) la `-turtle` (tortue).

Pour plus de fonctions et de précisions, voir la documentation officielle du module `turtle`.

Vous pouvez faire les exercices de la feuille n° 6. (et la feuille n° 7 – difficile)

B Sur ce document

B1 : Licence option 1

DO WHAT THE FUCK YOU WANT TO PUBLIC LICENSE
Version 2, December 2004

Copyright © 2004 Sam Hocevar
14 rue de Plaisance, 75014 Paris, France
Everyone is permitted to copy and distribute verbatim or modified copies of this license document, and changing it is allowed as long as the name is changed.

DO WHAT THE FUCK YOU WANT TO PUBLIC LICENSE
TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. You just DO WHAT THE FUCK YOU WANT TO.

B2 : Licence option 2

Cet article est publié sous la licence Creative Commons-BY-NC-SA :

BY [Paternité] Vous devez citer le nom de l'auteur original

NC [Pas d'Utilisation Commerciale] Vous n'avez pas le droit d'utiliser cette création à des fins commerciales.

SA [Partage des Conditions Initiales à l'Identique] Si vous modifiez, transformez ou adaptez cette création, vous n'avez le droit de distribuer la création qui en résulte que sous un contrat identique à celui-ci.

B3 : Licence option 3

B4 : Auteur

Franck CHAMBON : <Franck.Chambon@ac-grenoble.fr>
Remerciements à IsaT, fidèle lectrice.