FORMATION PYTHON POUR L'ALGORITHMIQUE AU LYCÉE
Yannick Le Bastard – Hubert Raymondaud
Mars 2019

Table des matières

1. Quelques environnements de programmation

Un environnement de programmation est un ensemble d'outils facilitant le travail du programmeur. Ces outils sont souvent réunis dans une application, souvent dédiée à un langage.

Les trois principaux outils sont :

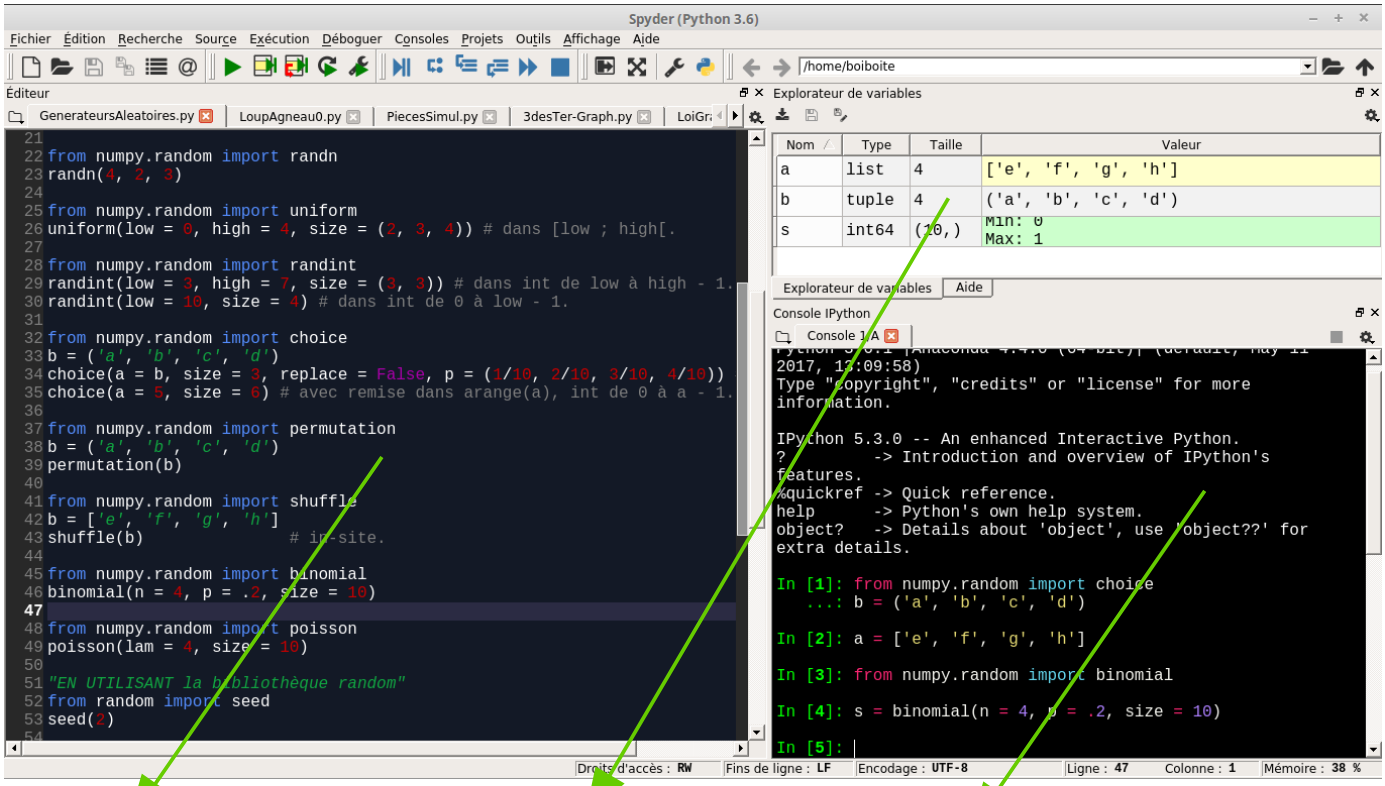
- un éditeur de texte à coloration syntaxique, complétion, indentation et aide en ligne automatiques, facilitant la saisie, la lecture et la correction du code informatique,
- une console ou interpréteur de commande ou shell, permettant d'exécuter les lignes de codes des programmes et d'afficher les résultats,
- un explorateur de variable ou workspace, permettant d'afficher touts les objets ("variables", "fonctions", "modules", ...) présents en mémoire,
- une fenêtre permettant l'affichage de séries de graphiques.

Voici quelques exemples d'environnement de programmation :

Pyzo, Spyder, dédiés à Python, RStudio dédié à **R**

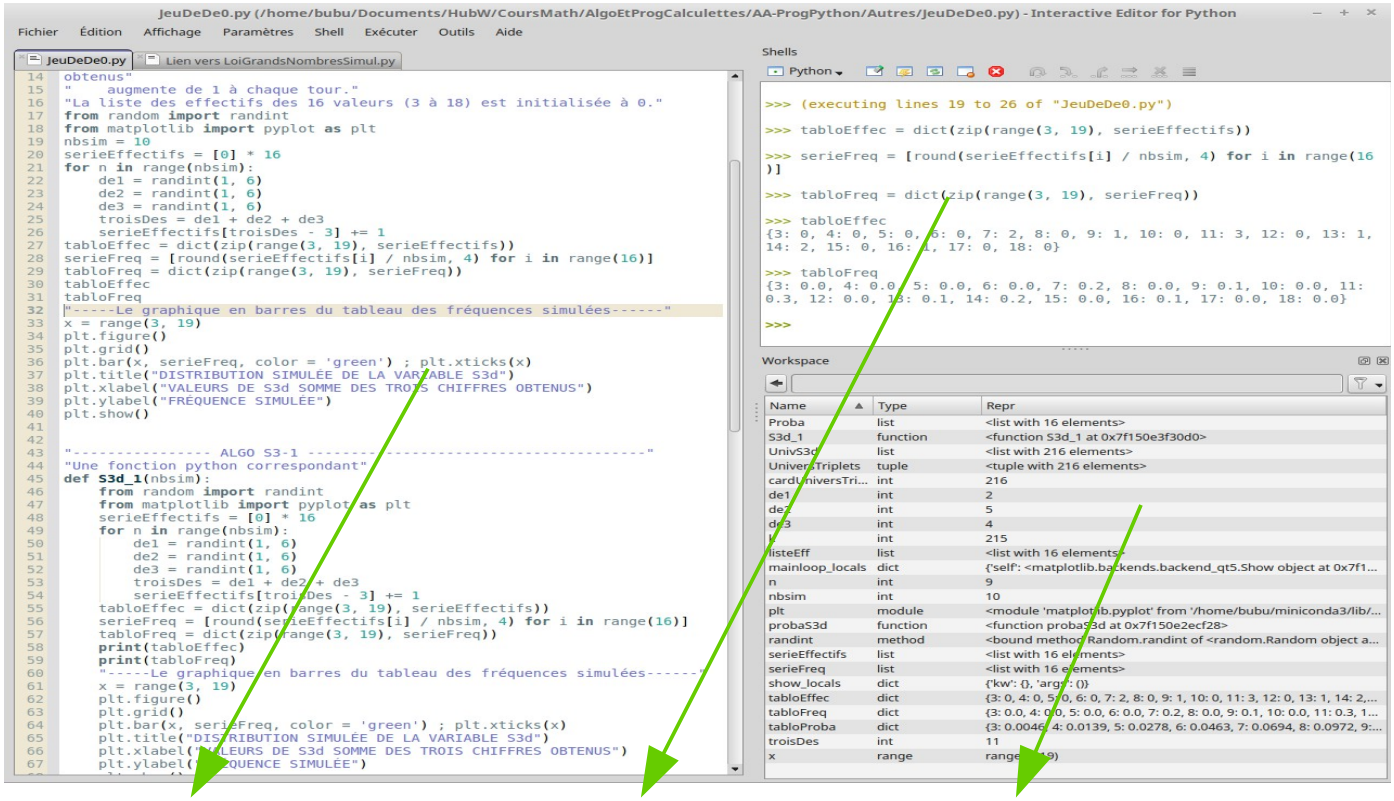
Sublime text2, Gedit, sont seulement des éditeurs de texte généralistes fournissant, entres autres, la coloration syntaxique et l'indentation automatique.

Ci-dessous une copie d'écran de Spyder 3 :



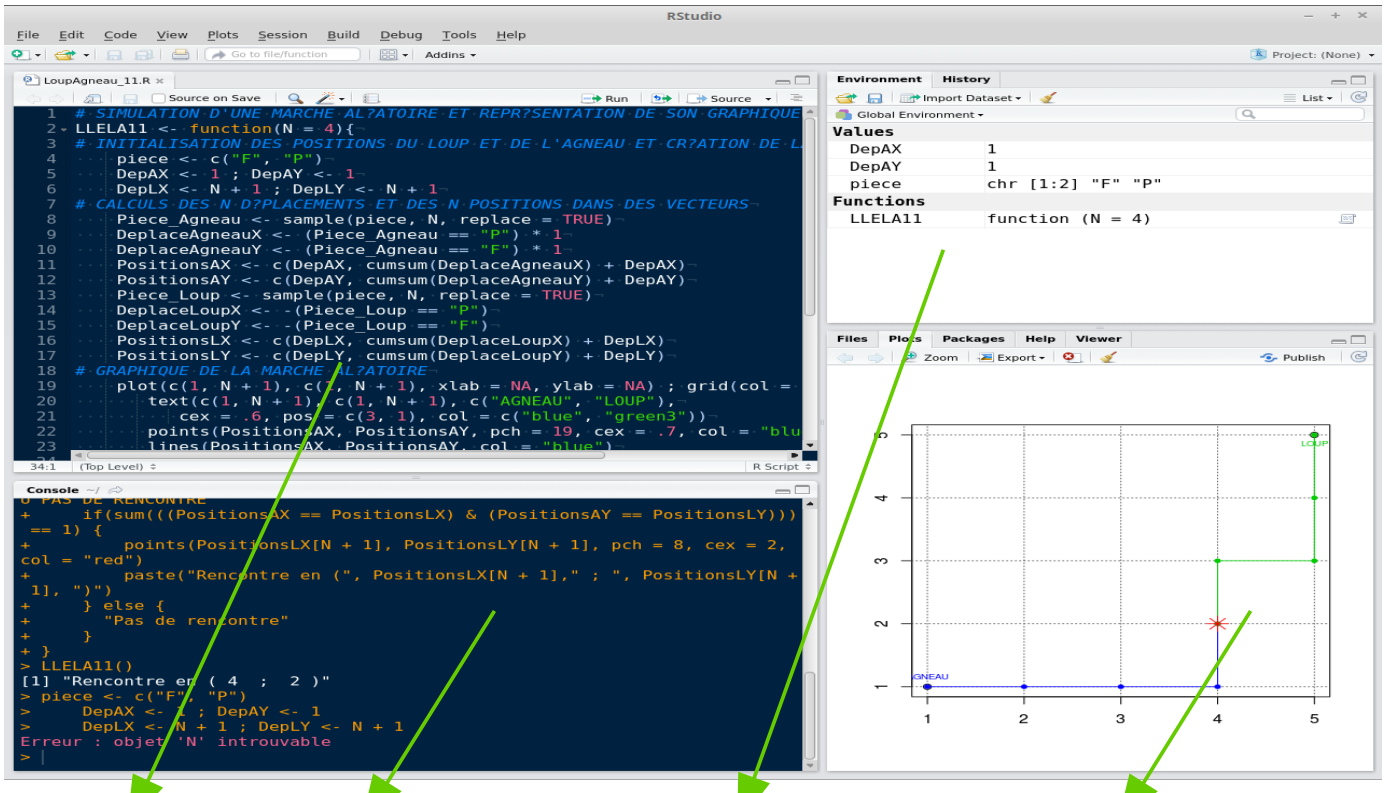
l'éditeur de texte l'explorateur de variables la console ou shell

Ci-dessous une copie d'écran de Pyzo 4 :



l'éditeur de texte la console ou shell l'explorateur de variables

Ci-dessous une copie d'écran de RStudio :



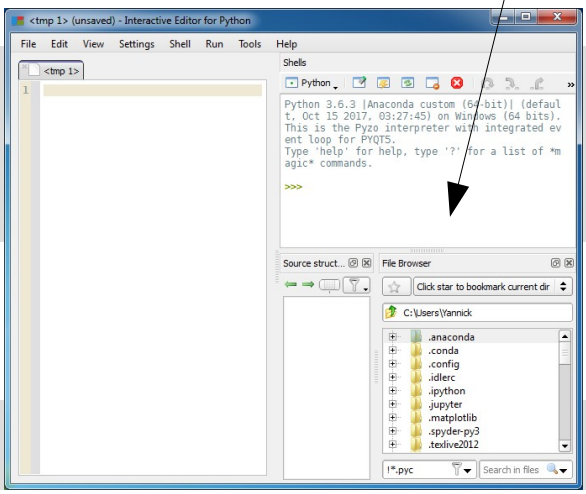
l'éditeur de texte la console ou shell l'explorateur de variables la fenêtre graphique

La fenêtre graphique est particulièrement utile lorsque l'on veut comparer les graphiques illustrant plusieurs simulations.

2. Utiliser la console interactive (shell en anglais).

2.1. Exécuter des lignes de commandes simples dans la console

Travailler directement dans l'interpréteur de commande (ou shell ou console) permet de tester des commandes. Elles sont saisies après le "prompt" >>> et peuvent être exécutées en appuyant sur la touche "entrée". Ce n'est pas le mode habituel pour exécuter des programmes car ça n'est pas pratique.



Testez dans cette fenêtre les commandes qui suivent :

```
>>> 5+9
>>> 8*6
>>> 4/3
>>> (5+7*3)/2
```

La priorité des opérations est-elle respectée ?

Saisir ensuite :

```
>>>3**2
>>>3**3
```

Quelle fonction a la commande ** ?

2.2. Objets Python et affectation

Pour exécuter les certaines opérations arithmétiques, le langage interprété Python utilise des *variables*. Ce sont des *objets* informatiques que le langage sait manipuler à l'aide de certains opérateurs, tout comme on utilise des objets mathématiques qui interviennent dans des opérations mathématiques.

Mais attention, une *variable* informatique est une référence à un emplacement en mémoire vive, dont le contenu peut évoluer pendant le déroulement du programme, mais qui ne peut prendre qu'une seule "valeur" à un moment donné. Une variable informatique peut aussi contenir une chaîne de caractères. Elle est, en tout cela, mais pas uniquement, différente d'une variable en mathématique ou d'une variable en statistique.

L'affectation d'une valeur à une variable s'effectue à l'aide du symbole égale « = ». Là encore il faut bien faire la différence avec la relation mathématique =, intervenant entre 2 objets, assertion vraie ou fausse et l'affectation informatique d'une référence à une "valeur". On place une "valeur" dans une "case" ayant une "adresse". Certaines calculatrices utilisent le symbole →, le langage **R** peut utiliser les symboles =, →, ←.

Testez, dans l'interpréteur de commandes, les instructions suivantes :

```
>>>a = 5
>>>b = 3.14                #Le séparateur décimal en Python est le point.
>>>type(a)                 #Que s'affiche-t-il ?
>>>type(b)                 #Que s'affiche-t-il ?
>>>c = "Salut"
>>>type(c)                 #Que s'affiche-t-il ?
>>>d = [3, 5, 2.7, "coucou"]
>>>type(d)                 #Que s'affiche-t-il ?
>>>e = (3, 5, 2.7, "coucou")
>>>type(e)                 #Que s'affiche-t-il ?
>>>f = True
>>>type(f)                 #Que s'affiche-t-il ?
>>>g = range(8)
>>>type(g)                 #Que s'affiche-t-il ?
```

Afin de faciliter la lecture des lignes d'instruction, il est d'usage de mettre des espaces de chaque coté des opérateurs.

Il est important de noter que les variables peuvent être de types différentes, le typage se faisant automatiquement lors de l'affectation d'une adresse à un contenu. L'interpréteur Python sait reconnaître certains type de variables. Il faut parfois l'aider en saisissant des "signes" lui permettant cette reconnaissant. C'est ce qu'on appelle le typage automatique des variables.

Donner quelques exemples de types des variables que vous venez de rencontrer :

-
-
-
-

-
-
-
-

D'autres types existent, dont nous rencontrerons quelques exemples par la suite. Remarquez que l'objet de type "list" peut être constituée de variables de types différents. Certaines opérations ne peuvent pas se faire sur tous les types de variables ou sur des variables de types différents. Testez en observant l'explorateur de variables :

```
>>>c = a + b
>>>c
>>>c = c * 3
>>>type(c)          #On a ré-affecté à la variable c une autre valeur
>>>c = "à nouveau du texte !"
>>>type(c)          #On peut recommencer autant de fois que l'on veut une ré-affectation
>>>a + c            #Que s'affiche-t-il ?
>>>c * 3            #Que s'affiche-t-il ?
```

Pour d'avantage d'informations à propos des types de variables et de leur utilisation, vous pouvez consulter l'aide de Python, très utile et fourmillant d'exemples. Nous donnons également en annexe un complément utile sur la notion d'affectation. En ce qui concerne les variables, nous utiliserons ici essentiellement les types *int*, *float*, *str, tuple* et *list*.

- **Exercice 2–1** : Signalons au passage l'opérateur **modulo**, représenté par le symbole **%**. Cet opérateur fournit le reste de la division entière d'un nombre par un autre. Essayez par exemple :

  ```
  >>>10 % 3
  >>>10 % 5
  ```

  Cet opérateur vous sera très utile plus loin, notamment pour tester si un nombre **a** est divisible par un nombre **b**. Il suffira en effet de vérifier que **a % b** donne un résultat égal à zéro.

  Vérifier à l'aide de la commande modulo si le nombre 33294 est divisible par 3 puis par 17.

- **Exercice 2–2** (un grand classique) : Saisir dans le prompt de commandes

  ```
  >>>a = 5
  >>>b = 2
  ```

  Puis saisir des instructions (en nombre minimum) qui permettent d'échanger les valeurs de a et de b. Vérifiez après coup.

2.3. Compléments utiles sur la notion d'affectation

2.3.1. L'affectation parallèle

```
>>>a, b = 3, 4                    #J'affecte simultanément à a la valeur 3 et à b la valeur 4
```

Ici, la ligne de code ci-dessus parvient exactement au même résultat que les deux lignes de codes :

```
>>>a = 3
>>>b = 4
```

♦♦♦ **ATTENTION !** Il convient cependant de bien comprendre comment fonctionne l'affectation parallèle par rapport à l'affectation simple.

- **Exercice résolu 2–3** : on souhaite échanger les valeurs de 2 variables a et b.
  - Résolution avec l'affectation simple

```
>>>a = 3
>>>b = 4
                                 #Fausse bonne idée : échanger directement les valeurs de a et b
>>>a = b                        #a prend la valeur 4
>>>b = a                        #La valeur 4 étant à présent affectée à la variable a, b la prend aussi !
                                 #Au final, a et b valent 4
                                 #On va avoir besoin d'une variable supplémentaire pour stocker la valeur
                                 #initiale de a. Corrigeons le script...
>>>a = 3
>>>b = 4
>>>c = a                        #J'affecte à c la valeur initiale de a, c'est-à-dire 3
>>>a = b                        #a prend la valeur de b, c'est-à-dire 4
>>>b = c                        #b prend la valeur de c, c'est-à-dire la valeur initiale de a : 3
```

  - Résolution avec l'affectation simple

```
>>>a, b = 3, 4                  #J'affecte simultanément à a la valeur 3 et à b la valeur 4
>>>a, b = b, a                  #ET LÀ, GROSSE DIFFÉRENCE : les expressions de la partie droite sont
                                 #d'abord toutes évaluées avant qu'aucune affectation ne se fasse.
                                 #Comme b vaut 3 et a vaut 4, le tour est joué !
```

- **Exercice 2–4** : Sans utiliser Python, quelles valeurs vont être affectées à a et à b au final après les lignes de codes suivantes ?

```
>>> a, b = 6, 10                 >>> a, b = 6, 10
>>> a = b                        >>> a, b = b, a - b
>>> b = a – b
```

2.3.2. L'affectation parallèle

Exemple :

```
a = b = 5                        #On affecte simultanément à a et à b la valeur 5. Raccourci très pratique
```

2.3.3. Les surprises de l'affectation, deux exemples

Observez ce qui se passe à l'aide de l'explorateur de variables. [2, 3] est une liste, (2, 3) est un tuple.

```
>>>a, b = 6, 10 #donne le même résultat que >>> a, b = (6, 10) # et que >>>(a, b) = (6, 10) # par contre
>>>a = 2, 3 # et >>>a = (2, 3) # donnent le même résultat, différent du précédent.
```

>>>c = [2, 6, 7]	>>>d[2] = 5	>>>e[0] = 8
>>>c[1] = 0	>>>d	>>>e
>>>c	>>>c	>>>c
>>>**d = c**	>>>**e = c[:]**	

2.3.4. Pluralités problématiques, exceptions syntaxiques, quatre exemples

J'appelle pluralité problématique, essentiellement du point de vue pédagogique tant pour les enseignants que pour les élèves, le fait que le même symbole corresponde à plusieurs actions très différentes quand ils sont utilisés dans le langage Python. C'est malheureusement assez fréquent dans ce langage, et on rencontre même ce problème dans les bibliothèques dont plusieurs modules portant le même nom font des choses différentes !

2.3.4.1. Pluralités problématiques

► **[...]** crée des listes OU pointe des éléments d'une liste pour lecture ou affectation, deux actions très différentes pour un même symbole.

```
a = ['prob', 3, 2.5] #création d'une liste.
a[0] #pointe l'élément d'indice 0 et l'affiche.
```

```
a[2] = 'affect' #pointe la case d'indice 2 de la liste
pour lui affecter une chaîne de caractères.
```

► **:** délimite un bloc OU agrège des références dans des listes. À ne pas confondre avec les **:** de Texas ou de **R**.

```
suite_u = [20] ; n = 10
u = suite_u[0]
for i in range(n + 1):
    if u % 2 == 0:
        u = int(u / 2)
    else:
        u = int(3 * u + 1)
    suite_u.append(u)
print(suite_u)
On trouve :
[20, 10, 5, 16, 8, 4, 2, 1, 4, 2, 1, 4]
```

```
suite_u[2:5] #Affiche les éléments d'indices 2 à 4.
suite_u[:5] #Affiche les éléments d'indices 0 à 4.
suite_u[6:] #Affiche les éléments d'indices 6 au dernier.
suite_u[:] #Affiche tous les éléments.
```

► **(...)** crée un tuple OU délimite les paramètres d'une fonction OU ne fait rien ...

```
a, b = ('prob', 3, 2.5), (8, 7) #création de 2 tuples.
a, b = (9, 7) #ne fait rien
```

```
print(suite_u) #délimite les paramètres d'une
fonction.
```

► Nous verrons les exemples concernant les modules dans le chapitre 9.

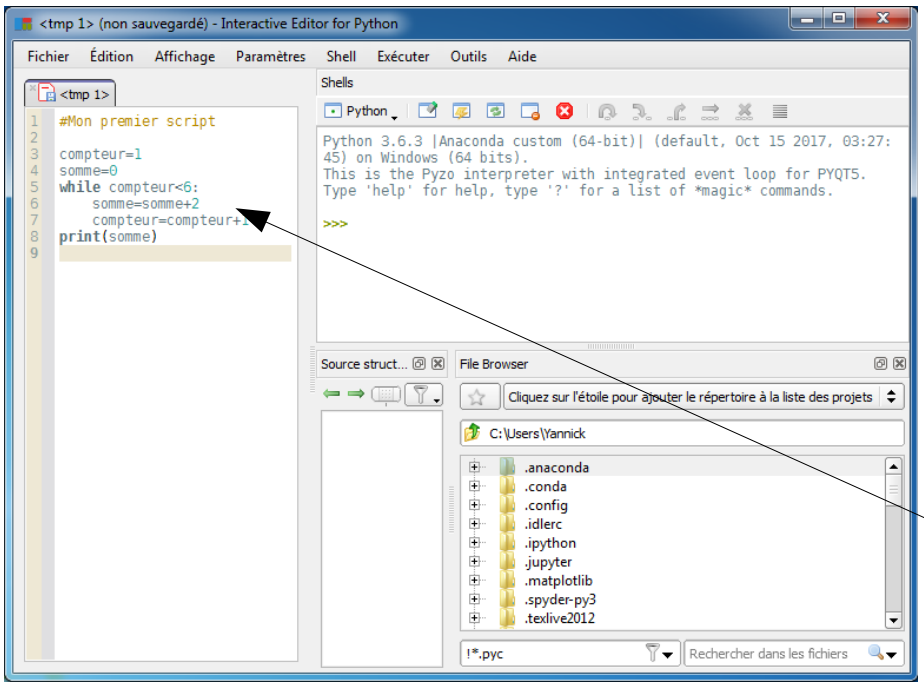
2.3.4.1. Exceptions syntaxiques, un exemple

► Python 3 un langage fonctionnel dans lequel les instructions sont des fonctions (au sens informatique). Une instruction est donc suivie, normalement obligatoirement, de parenthèses dans lesquelles figurent les paramètres ou rien s'il n'y a pas de paramètres. Il y a malheureusement des exceptions. Par exemple l'instruction return qui ne donne pas le même résultat selon qu'elle est suivie de rien, de parenthèse ou de crochets ...

```
print('obligatoire') #parenthèses obligatoires.
suite_u.append() #parenthèses sans arguments.
```

```
return a, b #retourne un tuple (a, b).
return(a, b) #retourne un tuple (a, b).
return [a, b] #retourne une liste [a, b].
```

3. Créer un script, utiliser les boucles while (tant que) et for (pour)



Si le mode interactif est très pratique pour et tester, explorer des commandes, découvrir les propriétés des différents types de variables, Dès qu'il y a plusieurs lignes de commandes, il est plus pratique de créer des scripts que l'on peut enregistrer pour d'éventuelles réutilisations. Un script est tout simplement du texte contenant des lignes de code structurées selon quelques règles précises que nous allons voir ci-dessous. Il faut donc saisir le texte dans la fenêtre de gauche qui est l'éditeur.

3.1. Applications avec une boucle while

Exemple 3.1–1 :

En Python	En langage naturel	Avec CASIO	Avec TI

compteur = 1
somme = 0
while compteur < 6**:**
  somme = somme + 2
  compteur = compteur + 1
print(somme)
	compteur ← 1
	somme ← 0
	Tant que compteur < 6 faire
	  somme ← somme + 2
	  compteur ← compteur + 1
	Fin Tant que
	Afficher somme
		1 → C
		0 → S
		While C<6
		S + 2 → S
		C + 1 → C
		WhileEnd
		S◢
			1 → C
			0 → S
			While C<6
			S + 2 → S
			C + 1 → C
			End
			Disp S

Quelques règles pour l'écriture des scripts :

- les **double-points « : »** sont **obligatoires,** ils précèdent un nouveau bloc d'instructions qui doit obligatoirement être indenté.
- L'**indentation** du bloc d'instruction suit le double point **« : »**
- L'utilisation de cette boucle conditionnelle **while** a un intérêt pédagogique car nous voyons l'évolution du compteur à chaque tour. Ceci ne transparaît pas aussi clairement avec une boucle du type **Pour** lors d'une première approche. On peut alors faire remplir un tableau d'évolution des variables à titre d'exemple.

Enregistrez ce script via **Ctrl+S** (ou via le menu fichier) sous le nom exemple3-1-1 (qui prendra l'extension **.py**) et exécutez-le via **Ctrl+F5** ou via la commande exécuter du menu Exécuter.

Quel est le résultat affiché ?

■ **Exercice 3–1** :

1. Modifier le script pour calculer la somme $S=2+4+6+...+100$ et donner le résultat.
2. Modifier le script pour calculer le produit $P=2\times4\times6\times...\times18$ et donner le résultat.
3. Modifier le script pour calculer la somme $S=\frac{1}{3}+\frac{1}{5}+\frac{1}{7}+...+\frac{1}{101}$ et donner le résultat.

3.2. Applications avec une boucle for

Lorsque le nombre d'itérations est connu à l'avance, le mieux est d'utiliser une boucle **for** (**pour**). Cette dernière se conjugue en Python avec la **fonction prédéfinie range(...)**, que nous allons d'abord tester dans l'interpréteur de commandes (fenêtre de droite) en lien avec une liste.

Saisir :

```
>>>range(10)                    #rien ne se passe a priori, et pourtant...
>>>[i for i in range(10)]        #création d'une liste définie par compréhension.
>>>list(range(10))              #création d'une liste, permet de visualiser l'effet de range(...).
>>>tuple(i for i in range(10))   #création d'un tuple défini par compréhension.
>>>tuple(range(10))              #création d'un tuple, permet de visualiser l'effet de range(...).
```

Quel résultat renvoie la fonction range(N), où N est un entier naturel non nul ?

Saisir :

```
>>>[i for i in range(1, 11)]
>>>[i for i in range(2, 21)]
>>>[i for i in range(1, 21, 2)]
>>>[i for i in range(1, 25, 3)]
```

Quel résultat renvoie la fonction **range(a, b, k)** où a, b et k sont des entiers naturels ?

On donne ci-dessous un script alternatif à celui fourni en exemple 3.1–1 :

```
somme = 0
for compteur in range(1, 6):
    somme = somme + 2
print(somme)
```

Remarquez que le bloc d'instruction se réduit alors à une seule ligne.

- **Exercice 3–2** :

  1) Reprendre les questions 1 à 3 de l'exercice 3–1, en utilisant une boucle for et la fonction range().

  2) Créer un script qui calcule et affiche $S=\frac{1}{2\times4}+\frac{1}{2\times4\times6}+...+\frac{1}{2\times4\times6\times...\times20}$.

  3) Créer un script qui calcule et affiche $P=\prod_{i=1}^{10}\left(\frac{1}{1+i^2}\right)$.

- **Exercice 3–3** : Programmer la suite récurrente linéaire d'ordre 2 définie par $u_0=2$, $u_1=3$ et pour tout entier naturel n $u_{n+2}=\frac{1}{2}u_{n+1}-4u_n$. Vous afficherez tous les termes de u_0 à u_{20}.

- **Exercice 3–4** (***Flash première***) : Écrire un programme qui utilise une boucle for et qui calcule la somme des 30 premiers termes consécutifs d'une suite géométrique de premier terme $u_0=-15$ et de raison $r=2$.

- **Exercice 3–5** : Que renvoie l'expression

  ```
  >>>[i for i in range(20, –1, –2)] ?
  ```

  En déduire un exemple pour les boucles à pas dégressifs.

  Idée d'application : résolution de systèmes triangulaires simples par remontée.

Remarque **:** Nous verrons plus loin qu'il est préférable d'éviter, tant que possible, les boucles. Des outils, dont ceux de la bibliothèque numpy nous permettent parfois de nous en passer : *cf* l'exercice suivant.

- **Exercice 3–6** : Reprendre l'exemple 3.1–1 et les exercices 3–2 à 3–5 en utilisant les tuples ou les listes, mais sans faire appel à la bibliothèque numpy.

4. Input – Output (entrées et sorties de données), un résumé

4.1. input : entrée des données au clavier

La commande de base est l'instruction **input(...)** qui attend que l'utilisateur saisisse une donnée à l'écran. Attention, en Python (version 3.6), tout ce qui est saisi sera automatiquement interprété comme une variable de type str (chaîne de caractères). Si nous avons à saisir un nombre (entier ou flottant, il faudra composer avec une autre instruction).

Dans la programmation des langages fonctionnels, les entrées se font uniquement par l'intermédiaire des paramètres des fonctions informatiques. Il n'y a plus quasiment plus besoins de "input".

On retiendra ceci :

Pour saisir une chaîne de caractères :	**Pour saisir un entier :**	**Pour saisir un nombre flottant :**
c = **input(**"Saisir votre texte "**)**	d = int(**input(**"Saisir un entier "**)**)	e = float(**input(**"Saisir un nombre réel "**)**)

Remarques **:** ● la commande input(...) permet de faire afficher un message de votre choix écrit entre guillemets.
● Sinon utiliser int(input(...)), float(input(...)). Pour les listes saisir élément par élément.

4.2. print : sorties de données à l'affichage

Selon que l'on veuille simplement écrire la valeur d'une variable issue d'un calcul, du texte ou les deux, la syntaxe sera la suivante :

Pour écrire la valeur d'une variable saisie ou traitée auparavant :	**Pour écrire du texte :**	**Pour mixer les deux :**
print(variable**)**	**print(**"Texte"**)** #Le texte est entre guillemets.	**print(**"Texte 1", variable 1, "Texte 2", variable 2, ...**)**

4.3. Opérateurs de comparaison :

Ces derniers sont indispensables dans la création de scripts.

Opérateur	Signification	Syntaxe Python
<	Strictement inférieur à	<
≤	Inférieur ou égal à	<=
>	Strictement supérieur à	>
≥	Supérieur ou égal à	>=
=	Égal à	==
≠	Différent de	!=

- **Exercice 4–1 :** écrire un script qui étant donné un triangle ABC rectangle en A, demande à l'utilisateur de saisir les mesures des côtés AB et AC et renvoie la valeur de l'hypoténuse BC. (on invoquera la fonction racine carrée dès la première ligne via l'instruction : from math import sqrt)
- **Exercice 4–2** : écrire un script qui étant donné un triangle équilatéral ABC, demande à l'utilisateur de saisir la mesure d'un côté, et renvoie la mesure de sa hauteur.
- **Exercice 4–3** : écrire un script qui donne la table de multiplication par N, où N est un entier naturel saisi par l'utilisateur.

  Il s'affichera par exemple pour N = 4 :

  0 fois 4 égale 0

  1 fois 4 égale 4

  …

  10 fois 4 égale 40

■ **Exercice 4–4** : écrire un script qui demande à l'utilisateur :

1) de saisir un entier naturel N non nul,

2) de saisir un autre entier naturel L non nul,

3) et qui affiche la table de multiplication de L par N sous la forme d'un tableau de taille $(N+1)\times(L+1)$ (**éléments en gras**) de la manière suivante :

Par exemple, si N = 4 et L = 3, il s'affichera :

N / L	0	1	2	3	4
0	**0**	**0**	**0**	**0**	**0**
1	**0**	**1**	**2**	**3**	**4**
2	**0**	**2**	**4**	**6**	**8**
3	**0**	**3**	**6**	**9**	**12**

Indication : la commande **\n** permet un retour à la ligne ; la commande **end = ""** permet de rester sur la même ligne.

5. Instructions conditionnelles

Nous en avons déjà rencontré une à travers la boucle conditionnelle **while**. En ce qui concerne l'instruction conditionnelle classique **if – else (Si – Sinon)**, on peut donner le tableau récapitulatif suivant :

Langage naturel	En Python	Avec CASIO	Avec TI
Si *condition vraie* **faire** *instruction1* **Sinon faire** *instruction 2* ***Fin Si***	**if** *condition vraie***:** *instruction 1* **else :** *instruction 2*	**If** *condition vraie* **Then** *instruction 1* **Else** *instruction 2* **IfEnd**.	**:If** *condition vraie* **:Then** *instruction 1* **:Else** *instruction 2* **:End**

Remarques :

- Si l'alternative est de ne rien faire, une simple instruction **if** suffit,
- Dans le cas de plusieurs conditions disjointes deux à deux, on dispose de l'instruction **elif**, contraction de else + if **(Sinon si)** :

Si (condition 1 vérifiée**)** faire Bloc d'instructions n°1 …etc. **Sinon si** (condition N-1 vérifiée) faire Bloc d'instructions n° N-1 **Sinon** faire Bloc d'instruction n° N ***Fin Si***	**if** condition 1 vérifiée**:** Bloc d'instructions n°1 …etc. **elif** condition N-1 vérifiée**:** Bloc d'instructions n° N-1 **else:** Bloc d'instruction n° N

- **Exercice 5–1** : Écrire un script qui demande à l'utilisateur de saisir une note N entre 0 et 20 et qui renvoie le message « reçu » si $N \geq 10$ et recalé sinon.
- **Exercice 5–2** : Convertir une note scolaire N quelconque (sur 20), entrée par l'utilisateur sous forme de points (par exemple 12,75), en une note sous forme de lettre standardisée suivant le code suivant :

Note	Appréciation
N >= 16	A
16 > N >=12	B
12 > N >= 10	C
10 > N >= 8	D
N < 8	E

- **Exercice 5–3** : Déterminer si une année (dont le millésime est saisi par l'utilisateur) est bissextile ou non. (Une année A est bissextile si A est divisible par 4. Elle ne l'est cependant pas si A est un multiple de 100, à moins que A ne soit multiple de 400). Le ET du tableur s'écrit « and » en Python, et le OU s'écrit « or ».
- **Exercice 5–4 *(*Flash seconde*)** :

  Écrivez un programme qui demande à l'utilisateur de saisir les coordonnées de quatre points A, B, C et D et qui détermine précisément la nature du quadrilatère ABCD (trapèze, parallélogramme, rectangle, losange, carré ou quelconque)

  Indication : Faire d'abord un schéma des situations comme dans l'exercice précédent.
- **Exercice 5–5 (*un grand classique qui plaît*)** : « Le jeu du devin » : la commande randint(a,b) renvoie un entier (pseudo-)aléatoire entre a et b inclus. On l'appelle via la commande from random import randint.

  1) Programmer un script qui choisit au hasard un entier (pseudo-)aléatoire entre 1 et 100, et demande à l'utilisateur de le deviner. Tant que ce ce dernier n'a pas trouvé ce nombre, il recommence : s'il saisit un nombre plus grand (resp.) plus petit, l'ordinateur renverra le message « plus petit » (resp. « plus grand »). En cas de victoire, le message sera « gagné » avec le nombre d'essais utilisés.

  2) Modifier le programme précédent pour que le nombre d'essais soit limité à 10.

- **Exercice 5–6** : Écrivez un programme qui affiche en ligne les 15 premiers termes de la table de multiplication par 7, en signalant au passage (à l'aide d'un astérisque) ceux qui sont des multiples de 3. Exemple : 0* 7 14 21* 28 35 42* 49 ...

- **Exercice 5–7** : Convertir un entier en nombre Romain.

  On demande à l'utilisateur de choisir un entier compris entre 1 et 3999. L'ordinateur le convertit ensuite en nombres romains.

  Rappel : En nombres romains, on a :

un	deux	trois	quatre	cinq	six	sept
I	II	III	IV	V	VI	VII
huit	neuf	dix	cinquante	cent	cinq cents	mille
VIII	IX	X	L	C	D	M

  Vous chercherez sur Internet le principe d'écriture d'un nombre en nombre romain. Testez-le ensuite sur les exemples qui suivent :

  a) 2397 b) 3912 c) 3812 d) 756

6. Un peu de dessin avec le module turtle

Prenons quelques instants de détente en utilisant le module Turtle de Python (référence à la tortue du LOGO). C'est par ailleurs un excellent moyen de faire travailler les élèves sur la notion de boucle, de coordonnées, avec le double avantage de visualiser directement le résultat et de pouvoir comparer ce-dernier avec celui à acquérir. Des erreurs d'indentation classiques, très formatrices, sont alors aisées à corriger, et sont un excellent moyen d'acquérir de la rigueur.

On appelle le module turtle via la commande « from turtle import* ».

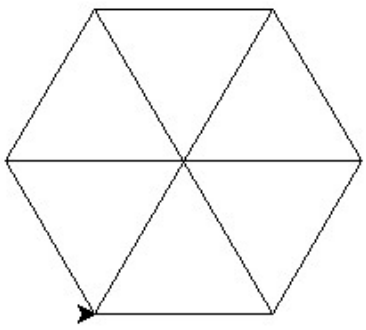
Les principales commandes qui nous seront utiles sont :

from turtle import *	Permet d'importer les commandes pour les tracés.
reset()	On efface tout et on recommence
goto(x,y)	Aller à l'endroit(x,y)
forward(x)	Avancer de la distance x
backward(x)	Reculer de la distance x
up()	Relever le crayon (avancer sans dessiner).
down()	Abaisser le crayon pour recommencer à dessiner.
left(x)	Tourner à gauche d'un angle en degrés égal à x.
right(x)	Tourner à droite d'un angle en degrés égal à x.
color('couleur')	colorie les traits du dessin en la couleur choisie.
begin_fill() … end_fill()	remplit le dessin.

- **Exercice 6–1** : Par défaut, le premier déplacement s'effectue horizontalement, vers la droite. Sans vous servir de l'ordinateur, devinez ce que font les scripts qui suivent :

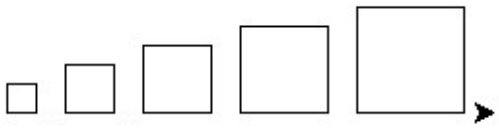
```
from turtle import *
for i in range(6):
    forward(100)
    left(360/6)                #petit indice : à la place d'écrire left(60)

from turtle import *
for i in range(5):
    forward(50)
    left(72)

from turtle import *
for i in range(12):
    forward(80)
    left(30)
```

- **Exercice 6–2** : écrivez un script qui renvoie les dessins suivants :
  1. Un carré de côté 70,
  2. Un heptagone de côté 100,
  3. La figure (de côté 100) qui suit. Avec juste deux boucles pour la construire.



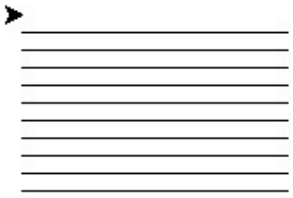
4. Cinq carrés de côté 15 dont les bases sont espacées de 15 :

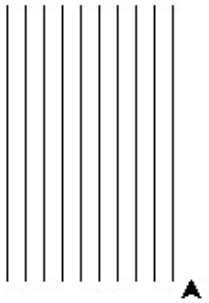
5. Les mêmes carrés, avec le même espacement, mais dont les côtés s'accroissent de 10.



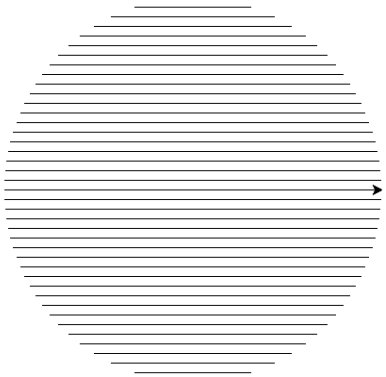
6. Un carré de carrés rouges (tous de taille 15 et espacés de 15)

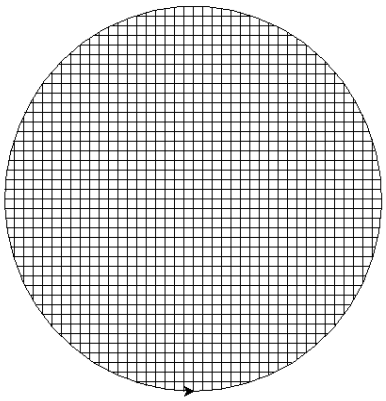
7. Les quadrillages suivants (longueur des traits de 150, espacement de 10) :



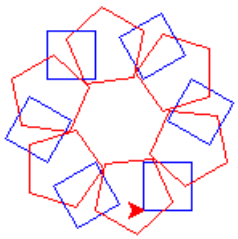


8. Dessiner les figures suivantes (le diamètre du cercle est de 150) et il y a 40 bandes horizontales pour la première, à laquelle on rajoute 40 bandes verticales pour la seconde. La commande circle(r) trace un cercle de rayon r à partir de son « pôle sud ».

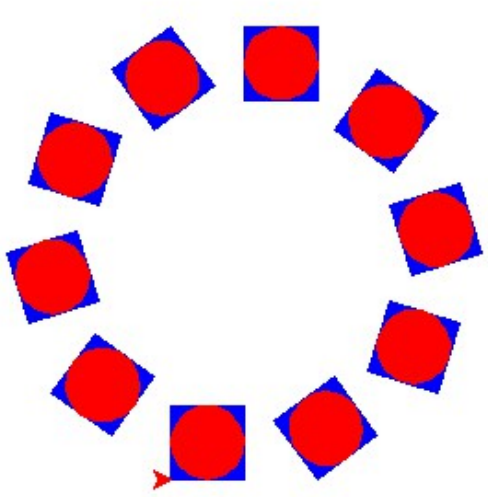




9. Le motif qui suit :



10. Et ce motif (le carré a pour côté 40) : si vous bloquez, attendez la section 8 !



10. Rajoutez un carré vert inscrit dans le cercle rouge passant par les 4 points de tangence.

7. Chaînes de caractères et listes

Ce sont deux types de variables particulièrement utiles et utilisés. Si le second a sa place (ténue) dans les calculatrices, le premier peut être exploité efficacement en Python. La rapidité de calcul de ce dernier est cependant sans commune mesure avec nos outils de poche.

Nous n'en donnerons ici qu'un bref aperçu.

7.1. Les chaînes de caractères (type str – string)

7.1.1. Définition et exemples

Sous Python, une donnée de type **string** est une suite quelconque de caractères (texte ou même nombres) délimitée soit par des apostrophes (simple quotes), soit par des guillemets (double quotes).

Exemple 7.1–1 : Tester sous Pyzo les commandes suivantes

```
>>> phrase1 = 'Voilà un écrit '
>>> phrase2 = 'bien intéressant'
>>> print(phrase1, phrase2)
Voilà un écrit bien intéressant
```

À l'intérieur d'une chaîne de caractères, l'antislash « \ » permet d'insérer un certain nombre de codes spéciaux (sauts à la ligne, guillemets, etc.). On ne donne que l'exemple des sauts à la ligne.

```
>>> phrase3 = "Comment s'appelle-t-il ?\n Ce rustre ?"
>>> print(phrase3)
Comment s'appelle-t-il ?
Ce rustre ?
```

Python considère les chaînes de caractères comme une ***collection ordonnée*** d'éléments. Pour accéder à un élément de la chaîne, on saisit le nom de la variable et on lui accole entre crochets le numéro de l'élément considéré. On parle aussi de liste indicée. **Attention, la numérotation (l'indice) commence à zéro !**

Exemple 7.1–2 : Tester sous Pyzo les commandes suivantes

```
>>> mot = 'erg4ju6op'
>>> print(mot[0])
e
>>> print(mot[3])
4
```

7.1.2. Opérations élémentaires sur les chaînes de caractères

7.1.2.1. La concaténation

Cette opération consiste à mettre bout à bout les éléments constitutifs de plusieurs chaînes de caractères. Elle se symbolise à l'aide de l'opération +

Exemple 7.1–3 : Tester sous Pyzo les commandes suivantes

```
>>> mot1="Mais c'est "          #Le caractère spécial apostrophe est dans les double quotes
>>> mot2="merveilleux !"
>>> print(mot1+mot2)            #On concatène mot1 et mot2
Mais c'est merveilleux !
>>> print(mot2+mot1)
merveilleux !Mais c'est         #L'ordre est important.
```

7.1.2.2. Déterminer la longueur d'une chaîne

Cette opération s'effectue grâce à la fonction **len(...)**.

Exemple 7.1–4 : Tester sous Pyzo les commandes suivantes

```
>>> mot = "adseghbm55f"
>>> len(mot)
11
```

7.1.2.3. Convertir une chaîne de caractères en nombre

Exemple 7.1–5 : Tester sous Pyzo les commandes suivantes

```
>>> nb1 = '54'
>>> nb1 + 5
Traceback (most recent call last):
  File "<pyshell#36>", line 1, in <module>
    nb + 5
TypeError: Can't convert 'int' object to str implicitly
>>> int(nb1) + 5        #On convertit la chaîne nb1 en entier grâce à la fonction int()
59
>>> nb2 = '20.36'
>>> float(nb2) + 4      #On convertit la chaîne nb2 en flottant grâce à la fonction float()
24.36
```

Remarque **:** La chaîne qui ne contient aucun élément se note "" (rien entre les guillemets)

- **Exercice 7.1–1 :** Écrire un programme qui :
  1) demande à l'utilisateur de saisir une chaîne de caractères
  2) compte le nombre d'occurrences du caractère « a » dans cette chaîne.
  3) Si « a » n'apparaît pas, le programme le rajoute à la fin de la chaîne et affiche la chaîne.

- **Exercice 7.1–2** : Écrire un programme qui recopie une chaîne (dans une nouvelle variable) en l'inversant. Ainsi par exemple, « bouchon » deviendra « nohcuob ».

- **Exercice 7.1–3** : Écrire un programme qui :
  1) demande à l'utilisateur de saisir une chaîne de caractères sous la forme de lettres minuscules et convertit toutes les minuscules en majuscules.
  2) Et vice-versa !
  3) effectue ces deux opérations dans une chaîne de caractères de casses mélangées.
  Indication **:** cf l'aide de Python ou le Web pour la numérotation des caractères en **ASCII**

- **Exercice 7.1–4** : L'utilisateur doit saisir une chaîne d'ADN valide et une séquence d'ADN valide (« valide » signifie qu'elles ne sont pas vides et sont formées exclusivement d'une combinaison arbitraire de "A", "T", "G" ou "C"). Pour l'approche fonctionnelle, se référer au paragraphe 7.
  1) Écrire une fonction valide qui renvoie True si la saisie est valide, False sinon.
  2) Écrire une fonction saisie qui effectue une saisie valide et renvoie la valeur saisie sous forme d'une chaîne de caractères.
  3) Écrire une fonction proportion qui reçoit deux arguments, la chaîne et la séquence et qui retourne la proportion de séquence dans la chaîne (c'est-à-dire son nombre d'occurrences).

  Le programme principal appelle la fonction saisie pour la chaîne et pour la séquence et affiche le résultat.

  Exemple d'affichage :

  Il y a 13.33 % de "CA" dans votre chaîne.

  Indication **:** vous rechercherez dans l'aide Python la *méthode* **count** pour les chaînes de caractères. La syntaxe est **chaine.count(**sous_chaine**)**.

7.2. Les listes (type list et type tuple)

7.2.1. Définition et exemples

Sous Python, on peut définir une liste comme une collection d'éléments (éventuellement de types divers) séparés par des virgules, l'ensemble étant encadré par des crochets.

Exemple 7.2–1 : Tester sous Pyzo les commandes suivantes :

```
>>> maliste = ['abc', 'hello', 20, 3.14, 'coucou']
>>> print(maliste)
['abc', 'hello', 20, 3.14, 'coucou']
```

Remarque **:** Comme les chaînes de caractères, les éléments d'une liste sont numérotés en commençant à zéro.

Exemple 7.2–2 : Tester sous Pyzo les commandes suivantes

```
>>> maliste = ['abc', 'hello', 20, 3.14, 'coucou']
>>> maliste[0]
'abc'
>>> maliste[2]
20
```

À retenir : l'indice d'une liste de *n* éléments commence à 0 et se termine à *n*-1

7.2.2. Opérations de base sur les listes

7.2.2.1. Concaténation et duplication

Comme pour les chaînes de caractères, l'opérateur + sert pour concaténer des listes et * pour les dupliquer.

Exemple 7.2–3 : Tester sous Pyzo les commandes suivantes

```
>>> liste1 = [325, 'bonjour', 65, 69]
>>> liste2 = ['ha', 'ben', 564]
>>> liste1 + liste2                    #concaténation
[325, 'bonjour', 65, 69, 'ha', 'ben', 564]
>>> liste2 * 3                         #répétition
['ha', 'ben', 564, 'ha', 'ben', 564, 'ha', 'ben', 564]
```

Python est un langage « orienté objet ». On dispose de méthodes spécifiques permettant d'effectuer certaines actions sur des objets. Nous considérerons ici qu'une liste est un objet.

7.2.2.2. Ajouter un élément à la fin d'une liste

Exemple 7.2–4 : Tester sous Pyzo les commandes suivantes

```
>>> maliste = [201, 45.8, 12]
>>> maliste.append(35)
>>> maliste
[201, 45.8, 12, 35]
```

On a appliqué la méthode **append(...)** à l'objet maliste avec l'argument 35. La syntaxe est :
liste.append(objet**)**

Remarque **:** la liste vide se note **[]**. La méthode **append(**...**)** est particulièrement adaptée pour construire une liste à partir d'une boucle.

7.2.2.3. Ajouter un élément à l'intérieur d'une liste

Exemple 7.2–5 : Tester sous Pyzo les commandes suivantes

```
>>> maliste = [201,45.8,12,35,89]
>>> maliste.insert(3,100)
>>> maliste
```

```
[201, 45.8, 12, 100, 35, 89]
```

La commande **liste.insert(**indice, objet**)** insère l'objet dans la liste *avant* l'indice précisé.

7.2.2.4. Supprimer un élément dans une liste

Cette opération s'effectue grâce à la commande **del liste[**indice**]** ou **liste.remove(**élément**)**

Exemple 7.2–6 : Tester sous Pyzo les commandes suivantes

```
>>> maliste = [55,201,89,33]
>>> del maliste[1]
>>> maliste
[55, 89, 33]
```

La fonction **len(...)** renvoie également le nombre d'éléments d'une liste.

Exemple 7.2–7 :

```
>>> maliste = [55, 120, 40, -8]
>>> len(maliste)
4
```

7.2.2.5. Sommer les éléments d'une liste

Exemple 7.2–8 :

```
>>> maliste = [2, 1.35, 4]
>>> sum(maliste)
7.35
```

7.2.2.6. Min et Max d'une liste

Exemple 7.2–9 :

```
>>>liste = [0, 2, 4, 12, 3, 5]
>>>min(liste)
0
>>>max(liste)
12
```

7.2.2.7. Trier les éléments d'une liste (dans l'ordre croissant)

Exemple 7.2–10 :

```
>>> liste = [0,2,4,12,3,5]
>>> liste.sort()
>>> liste
[0, 2, 3, 4, 5, 12]
```

Remarque **:** On a appliqué la méthode **sort()** à l'objet liste. La syntaxe est : **liste.sort()**

7.2.2.8. Inverser les éléments d'une liste

Exemple 7.2–11 :

```
>>> maliste = [1, 2, 3, 4]
>>> maliste.reverse()
>>> maliste
[4,3,2,1]
```

Remarque **:** On a appliqué la méthode **reverse()** à l'objet maliste. La syntaxe est : **maliste.reverse()**

7.2.2.9. Le slicing (on s'en paye une tranche !)

Exemple 7.2–12 :

```
>>> liste = [1, 2, 3, 4, 5, 6, 7]
>>> liste[:2]
[1, 2]
>>> liste[2:]
[3, 4, 5, 6, 7]
>>> liste[3:6]
[4, 5, 6]
```

- **Exercice 7.2–5** : soit une liste L de longueur $N>3$. On se donne deux entiers $0\leq a<b\leq N$. Expliquez ce que font les opérations suivantes :

1) L[2 :], L[3 :]	L[-2 :], L[-3 :]	L[-b :]
2) L[: 2], L[: 3]	L[: -2], L[: -3]	L[: -b]
3) L[1 : 3], L[a : b]	L[: : 2], L[: : -2]	L[2 : N : 3]

7.2.2.10. Les listes définies en compréhension

Voici un moyen très puissant de définir une liste. Ses éléments sont définis par une propriété commune qu'il convient de bien définir.

Exemple 7.2–13 :

L'exemple qui suit crée la liste des 10 premiers carrés d'entiers

```
>>> carres = [i ** 2 for i in range(1, 11)] #attention au décalage de fin avec la fonction range()
>>> carres
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Exemple 7.2–14 :

Essayons par exemple de définir la liste de tous les entiers pairs compris entre deux bornes, par exemple 51 et 106. Il est possible d'insérer une instruction conditionnelle « if ».

```
>>> pairs = [i for i in range(51, 107) if i%2 == 0]
>>> pairs
[52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74, 76, 78, 80, 82, 84, 86, 88, 90, 92, 94, 96, 98, 100, 102, 104, 106]
```

- **Exercice 7.2–6** : écrire une commande *d'une ligne seulement* permettant de calculer :
  1) $S_{15}=\sum_{k=1}^{15}\frac{1}{(2k)^2}$ et $T_{15}=\sum_{k=1}^{15}\frac{1}{(2k+1)^2}$
  2) $S'=\sum_{k=4}^{10}\frac{1}{(2k)^2}$ à partir de S_{15} en utilisant le slicing. (deux lignes ici)
  3) $\cos(\pi/7)+\cos(3\pi/7)+\cos(5\pi/7)$
  4) La norme euclidienne sur $\mathbb{R}^N$: $\|(a_1,\dots,a_N)\|=\left(\sum_{k=1}^{N}|a_k|^2\right)^{1/2}$. On prendra par exemple $(a_1,\dots,a_N)=(\sqrt{7},-4,3,2)$.
- **Exercice 7.2–7** : la commande **from random import randint** permet d'importer la fonction **randint(**a,b**)** qui génère un entier aléatoire compris entre les bornes entières a et b *incluses*.
  1) En utilisant cette commande, créer une liste de cinq entiers aléatoires compris entre 1 et 6,
  2) Trier la liste dans l'ordre décroissant, **avec**, puis **sans** les méthodes sort() et reverse(),
  3) Créer un script qui compte le nombre d'ex-aequo et les transfère dans une nouvelle liste.
  4) Créer un script qui sépare en deux listes : entiers pairs et entiers impairs la liste créée à la question 1.

!!! Mise en garde importante : les listes sont des objets modifiables

Exemple 7.2–14 : Tester sous Pyzo les commandes suivantes :

```
>>> x = ['Monsieur', 'Seguin', 'devient', 'chevre', 35]
>>> y = x                #Attention y est seulement un alias de x
>>> y[1] = 'Fery'        #change x[1] en 'Fery'
>>> print(x)
['Monsieur', 'Fery', 'devient', 'chevre', 35]
>>> y = x[:]             #Cette fois-ci, on effectue une copie de x
>>> y[1] = 'Renard'      #modifie y mais pas x
>>> print(x)
['Monsieur', 'Fery', 'devient', 'chevre', 35]
>>> print(y)
['Monsieur', 'Renard', 'devient', 'chevre', 35]
```

- **Exercice 7.2–8** : créer un script qui génère une liste aléatoire de 10 lettres minuscules : 5 consonnes et 5 voyelles, puis qui renvoie une nouvelle liste alternant consoles et voyelles dans leur ordre d'apparition. Par exemple : la liste [a, o, e, f, i, t, r, y, z, z] sera transformée en [a, f, o, t, e, r, i, z, y, z].
  Indication **:** la commande **shuffle(**liste**)** permet de mélanger les éléments d'une liste donnée.

- **Exercice 7.2–9** :
  1) créer un script qui génère 1000 tirages de pile ou face avec une pièce équilibrée et qui calcule la fréquence de piles et de faces obtenus sur ces 1000 essais.
  2) même chose si la probabilité de faire pile est égale à 0,75.

- **Exercice 7.2–10** : créer un script qui génère un brin d'ADN aléatoire de taille 10 et renvoie sa séquence complémentaire.

7.2.3. tuples et opérations de base sur les tuples

Les tuples sont des listes non modifiables. Elles prennent moins de place en mémoire. Toutes les opérations sur les listes excepté celles qui les modifient, peuvent être utilisées sur les tuples.

Exemple 7.2–15 : Que font les commandes suivantes ? Vérifier en les exécutant.

```
>>> x = ('Monsieur', 'Seguin', 'devient', 'chevre', 35)
>>> x[:2]
>>> x[2:]
>>> x[1:3]
>>> x + x
>>> 3 * x
>>> x.sort()
>>> x.reverse()
>>> x.append(5)
>>> x.insert(2, 'inser')
>>> del x[4]
>>> x.remove('chevre')
>>> del(x)
```

8. La notion de fonction en langage de programmation

8.1. Syntaxe et mise en œuvre

De même qu'un organisme vivant (donc une structure biologiquement complexe) est constitué d'un assemblage de nombreuses cellules, un programme « compliqué » se structure en un assemblage de plus petits sous-programmes, chacun ayant un rôle bien défini. Nous allons découvrir comment créer ces petits sous-programmes et les utiliser pour en construire un autre plus compliqué.

L'utilité de ces sous-programmes, que l'on appellera **fonctions** est primordiale lorsque l'on veut réaliser plusieurs fois la même opération au sein d'un même programme.

ATTENTION : Python ne distingue pas fonctions et procédures comme en Turbo Pascal.

La syntaxe Python pour définir une fonction est la suivante :

```
def nom_de_la_fonction(liste de paramètres):
    Blocs d'instructions
```

***Remarques* :**

- Vous pouvez donner à votre fonction n'importe quel nom sauf ceux réservés au langage comme while, if, etc. et sans utiliser de caractères spéciaux.
- **L'indentation est obligatoire** après le mot clef « **def** » qui introduit le corps de la fonction et là encore celui-ci est terminé par un double-point **:**
- Enfin les parenthèses peuvent ou non contenir un ou plusieurs arguments.

Les nouveaux programmes privilégient cette approche fonctionnelle, ancrée dans le domaine de l'informatique. Aussi convient-il, sans rentrer dans les détails, de s'y intéresser.

Pour le moment nous allons utiliser Python en mode interactif.

Exemple 8.1–1 :

Recopiez le script qui suit dans l'interpréteur de commande de Pyzo :

```
>>> def carre():                #Pas de paramètres en argument ici
        for i in range(4):       #Remarquez l'indentation du bloc d'instructions
        forward(50)
        left(90)
                                 #Tapez deux fois Entrée : La première fois valide la ligne en cours.
                                 #La deuxième valide la fonction, elle est alors disponible en mémoire.
>>> from turtle import *
>>> carre()                     #Exécute la fonction en réalisant toutes les instructions dans l'ordre du script.
                                 #Que s'affiche-t-il à l'écran ?
```

***Remarques* :**

- L'intérieur des parenthèses est vide. Dans cette fonction, nous n'avons utilisé aucun paramètre, c'est-à-dire que l'utilisateur ne se donne pas le choix de faire varier par exemple la longueur du côté du carré.
- Bien remarquer l'indentation après l'instruction **def** suivie de deux points. Elle est obligatoire.

On aimerait pouvoir dessiner des carrés de tailles et de couleurs variables. Encore faut-il les choisir ! Pour cela, on mettra ces données en paramètre. À la suite de ce que vous avez saisi précédemment, écrire le script ci-dessous :

Exemple 8.1–2 :

```
>>> def carre(taille, couleur):
        color(couleur)
        for i in range(4):
            forward(taille)
            left(90)            #Appuyez deux fois sur la touche Entrée. Le prompt >>> réapparaît.
Saisir alors :
>>> from turtle import *
>>> carre(40, 'pink')
                                #Que s'affiche-t-il à l'écran ?
Saisir ensuite :
>>> carre(80, 'blue')
                                #Que s'affiche-t-il à l'écran ?
```

***Remarques* :**

- Le fait d'avoir mis la taille et la couleur du carré en paramètres (à l'intérieur des parenthèses) permet de tracer le carré avec les informations fournies. Il suffit de remplacer la taille et la couleur par les valeurs choisies en appelant la fonction.
- On aimerait néanmoins demander à l'utilisateur de saisir la taille et la couleur à l'écran pour qu'ensuite la fonction soit appelée et dessine le carré. On séparera donc en deux la construction de la fonction et le programme principal.

Tout programme en Python un peu complexe prendra nécessairement la forme suivante :

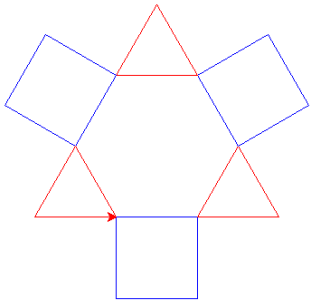
Liste de fonctions
#il peut n'y en avoir qu'une.
Programme principal (le 'main' en anglais)
#on utilise une fonction utilisant les fonctions précédentes.

■ **Exercice 8.1–1** :

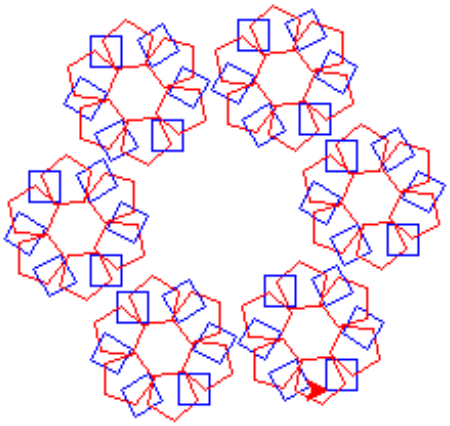
1) Créer une fonction polygone(taille,N) qui dessine un polygone régulier à N côtés dont la mesure du côté est taille, puis l'utiliser pour dessiner la figure suivante :

Les polygones, ont leurs côtés de mesure 20 et leurs bases sont espacées de 30.

2) La figure suivante (les polygones ont tous un côté de 80) :



3) Le motif de motifs :



Mais les fonctions sont tout aussi utiles, et c'est même leur but, pour décomposer un problème complexe en une multitudes de problèmes simples. Reprenons par exemple le calcul de $S=\frac{1}{2\times4}+\frac{1}{2\times4\times6}+...+\frac{1}{2\times4\times6\times...\times20}$ rencontré à la question 2 de l'exercice **3–2** (Partie 3).

Les dénominateurs successifs sont les produits des nombres pairs de 2 à x, x nombre pair variant de 4 à 20 (y pour généraliser). La fonction denominateur va calculer cette somme. La fonction principale siprod() calculera la somme des inverses de tous ces produits.

Exemple 8.1–3 :

```
def denominateur(x):                #Pour calculer 2*4, 2*4*6, ..., 2*4*6*...*x, x entier pair de 4 à 20.
    p = 2
    for i in range(4, x + 1, 2) :   #Le pas est de 2
        p = p * i
    return p
#Programme principal Algorithme 1, utilise denominateur(x)
def siprod1(y = 20):
    sip = 0
    for x in range(4, y + 1, 2):    #Attention à l'argument de fin de range
        sip = sip + 1 / (denominateur(x))
    return sip
```

***Remarque* :** pour les accros des listes, on peut remplacer le programme principal par :

```
#Programme principal Algorithme 2, utilise denominateur(x)
def siprod2(y = 20) :
    sip = sum([1 / (denominateur(x)) for x in range(4, y + 1, 2)])
    return sip
```

Les méthodes associées aux listes sont des outils très efficaces.

- **Exercice 8.1–2** : écrire un programme utilisant une fonction exactement :
  a) Qui demande à l'utilisateur de saisir un entier N.
  b) Et qui calcule $u_N=\sum_{k=0}^{N}\frac{1}{k\,!}$.

  Soit (v_n) la suite définie pour tout entier naturel n par $v_n=u_n+\frac{1}{n.n\,!}$. On admet que les suites u et v sont adjacentes. Leur limite commune et bien connue est **e**. En utilisant ces deux suites, déterminer **e** avec au moins 9 chiffres significatifs.

■ **Exercice 8.1–3** : écrire un programme utilisant des fonctions :

1) Créer une fonction tri_C(N) qui étant donné une chaîne de caractères (sans espace et non accentués, en minuscule pour simplifier) de N caractères renvoie la chaîne ordonnée alphabétiquement. Par exemple 'abart' devient 'aabrt'.

Indication **:** penser au codage ASCII des caractères et à la méthode sort()

2) Créer une fonction tri_L() qui étant donné une liste de quatre mots de 5 lettres minuscules (de type 'str') ordonne les mots de cette liste dans l'ordre alphabétique. Par exemple si l'utilisateur saisit :
liste = ['hello','paulo','anaïs','salut']
le script renverra :
['anaïs','hello','paulo','salut']
L'appliquer aux exemples suivants :
a) liste1 = ['bravo','james','costa','brava']
b) liste2 = ['alexi','anais','anana','alexy']

3) Généraliser au cas où la liste comporte T caractères, majuscules et minuscules mélangées, de tailles différentes.

Indication **:** Pour simplifier 'ALERTE'.lower() renvoie 'alerte' et 'alerte'.upper() renvoie 'ALERTE' . Et ça marche aussi quand on a des casses mélangées. Par exemple 'AlertE'.upper() renverra 'ALERTE'.

Disons enfin quelques mots sur la portée des variables...

8.2. Variables locales et globales

Lorsque l'on définit une fonction, il est nécessaire de connaître la portée des variables.

Exemple 8.2–1 : Définissons une fonction en mode interactif sous Pyzo.

```
>>>def fonction1():
        x = 3
        print("Dans cette fonction x est égal à ", x)
>>> fonction1()
Dans cette fonction x est égal à 3
>>> x                                    #Message d'erreur
Traceback (most recent call last):
 File "<pyshell#14>", line 1, in <module>
   x
NameError: name 'x' is not defined
```

Remarque **:** la variable x n'existe pas en dehors de la fonction où elle a été définie. On dit que x est une une **variable locale.**

Par contre, une valeur déclarée à la racine du module principal est visible partout.

Exemple 8.2–2 : Tester les commandes suivantes sous Pyzo

```
>>> def fonction2():
        print(x)
>>> x = 4
>>> fonction2()
4
```

```
>>> x = 5
>>> fonction2()
5
```

On dit que x est une **variable globale.** Cette variable est visible dans tout le module.

††† ATTENTION aux types modifiables comme les listes †††

►La règle LGI

Nous avons vu au paragraphe précédent les notions de **variables locales** ou **globales**. Il existe aussi un troisième type de variables : les **variables internes**. Exemple : la fonction len() qui renvoie la taille d'une liste ou d'une chaîne de caractères et qui existe dès qu'on lance Python.

Python traite les variables par ordre de priorité :

1) D'abord, il regarde si la variable considérée est une variable locale,
2) Ensuite, si elle n'existe pas localement, il regarde si c'est une variable globale,
3) Enfin, il regarde si c'est une variable interne.

Exemple 8.2–3 :

```
>>> def fonction():
        x = 10                              #x est une variable locale
        print('Dans la fonction x vaut ', x)
>>> x = 20                                  #x est une variable globale
>>> fonction()
Dans la fonction x vaut 10
>>>print('Dans le module principal x vaut ', x)
Dans le module principal x vaut 20
```

Remarque **:** Notez bien ce qui s'est passé. x a pris en priorité la valeur qui lui était définie localement par rapport à celle qui lui était définie globalement.

Il est possible de forcer une variable à prendre sa valeur globale dans une fonction avec le mot clé : **global**

Exemple 8.2–4 :

```
>>> def double():
        global x
        x = x * 2
>>> x = 3
>>> double()
>>> x
6
```

Remarque **:** cette notion a son importance lorsque dans une fonction qui renvoie une ou plusieurs valeurs, on souhaite que ces dernières servent dans le programme principal sous un nom de variable. On doit donc les déclarer en tant que variables globales. Mieux vaut cependant l'éviter au maximum. Et c'est généralement possible.

Pour optimiser l'espace mémoire il est d'usage de faire les appels de bibliothèque à l'intérieur des fonctions, et de n'appeler que les fonctions utilisées dans les algorithmes. Elles sont ainsi affectées en variables locales, juste le temps d'utilisation de la fonction. Il faut éviter d'importer toute la bibliothèque quand on n'utilise qu'une seule de ses fonctions.

9. Les bibliothèques numpy et matplotlib et les graphiques

Python dispose de nombreuses bibliothèques répondant à des problèmes d'ordres divers :

1. Calcul scientifique (random, numpy et scipy)
2. Tracés de graphiques (matplotlib)
3. Calcul symbolique (sympy)

Nous ne détaillerons brièvement dans cette section que les items 1 et 2 (numpy et matplotlib), et encore de manière très partielle.

D'ailleurs le procédé d'appel d'une bibliothèque a déjà été utilisé avec le module turtle. Retenons les procédés usuels :

```
import ma_bibliotheque
# appel à une fonction de ma_bibliotheque :
ma_bibliotheque.la_fonction(...)

import ma_bibliotheque as bibli                # raccourci
# appel à une fonction de ma_bibliotheque :
bibli.la_fonction(...)
```

Moins précis (car la bibliothèque d'origine des fonctions n'est pas précisée à leur appel) :

```
from ma_bibliotheque import la_fonction
# appel à une fonction de ma_bibliotheque :
la_fonction(...)

from ma_bibliotheque import *
# appel à la_fonction
la_fonction(...)
```

9.1. matplotlib et numpy

9.1.1. Quelques commandes matplotlib

Nous allons étudier sur quelques exemples simples l'utilisation de matplotlib pour tracer des courbes de fonctions usuelles. Voici la commande à entrer pour matplotib avec pour alias plt :

```
import matplotlib.pylab as plt
```

Pour x, y vecteurs de même dimension :

plt.plot(x, y)
affiche la courbe affine par morceaux reliant les points d'abscisses x et d'ordonnées y (nombreuses options possibles)

plt.hist
#trace un histogramme (spécifier normed = True).
#Deux options pour les colonnes :
#bins = nombre de colonnes ou bins = abscisses des séparations des colonnes

plt.bar
#trace un diagramme en bâtons

plt.scatter(x, y)
#affiche le nuage de points d'abscisse x et d'ordonnée y

plt.stem(x, y)
#affiche des barres verticales d'abscisse x et de hauteur y

plt.axis([xmin, xmax, ymin, ymax])
définit les intervalles couverts par la figure

plt.axis('scaled')
  # impose que les échelles en x et en y soient les mêmes
plt.show()
  #affiche les fenêtres créées dans le script. A insérer à la fin.
plt.figure()
  #crée une nouvelle fenêtre graphique
plt.title("mon titre")
  #donne un titre à une figure
plt.legend(loc = 'best')
  #affiche la légende d'un graphique (en position optimale)
plt.subplot
  #subdivise la fenêtre graphique de façon à y afficher plusieurs graphiques

Bien des options sont négociables dans l'affichage des graphiques voulus. Nous n'en donnerons que trois exemples. Voici comment importer la bibliothèque numpy via un alias :

9.1.2. utiliser numpy dans la construction des graphiques

```
import numpy as np
```

La bibliothèque numpy est par essence liée aux tableaux (array ou matrices) de nombres. Elle dispose cependant de nombreuses fonctions avancées en statistiques et probabilités, très utiles dans les classes du secondaire ou dans le supérieur (notamment en BTSA). En outre, sa rapidité d'exécution par rapport aux boucles classiques en Python est largement supérieure.

L'une des options les plus utilisées est la création d'un tableau ligne dont les éléments sont espacés régulièrement. C'était possible en Python classique grâce à la fonction range, mais cette dernière, seule, ne pouvait prendre que des pas entiers.

Cependant si l'on veut subdiviser l'intervalle [0;1] en 10 intervalles égaux :

```
>>>a = [i/10 for i in range(11)]
>>>a
[0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]
```

qui peut être remplacé avec numpy par la commande :

```
>>> a = np.linspace(0, 1, 11)          #0 et 1 début et fin de l'intervalle [0;1] , 11 est le nombre de points de
                                       #la subdivision régulière de [0;1]
>>> a
array([ 0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. ])
```

Remarque **:** a est du type np.ndarray (tableau) , objet fondamental de la bibliothèque numpy qui est dédiée au calcul matriciel. Nous n'en ferons qu'un usage au cas par cas dans ce stage. Mais le lecteur intéressé ou utilisateur d'algèbre linéaire (par exemple en post BTS-DUT) pourra consulter avec profit les tutoriels fourmillant sur la toile.

Voici deux exemples de tracés (avec quelques options). Ils sont proposés en lignes de commandes afin de pouvoir détailler l'effet de chaque option des commandes. Il faudra les inclure dans des fonctions pour en faciliter l'utilisation et en optimiser la mise en œuvre informatique.

Il faut aussi penser à n'appeler que les fonctions utilisées dans les algorithmes et effectuer les appels à l'intérieur des fonctions, qui seront ainsi affectées en variable locales.

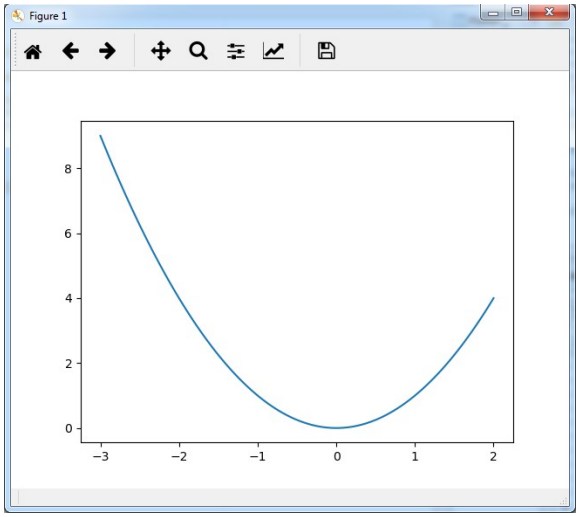
Exemple 9.1–1 : recopier le script suivant et l'exécuter. Vous devez obtenir la courbe ci-dessous.

graphe1.py (C:\Users\Yannick\Documents\Livre Yannick modélisation\graphe1.py) - Interactive Editor for Python

Fichier Édition Affichage Paramètres Shell Exécuter Outils Aide

graphe1.py

```
#Mon premier graphe en Python

#importation des bibliothèques numpy et matplotlib avec alias
import numpy as np
import matplotlib.pylab as plt  #ou import matplotlib.pyplot as plt

x=np.linspace(-3,2,100)        #Trace de la fonction sur [-3;2], 100 points
y=[i**2 for i in x]            #on trace la fonction carre
                               #x est du type array et y du type list !

plt.plot(x,y)
plt.show()
```



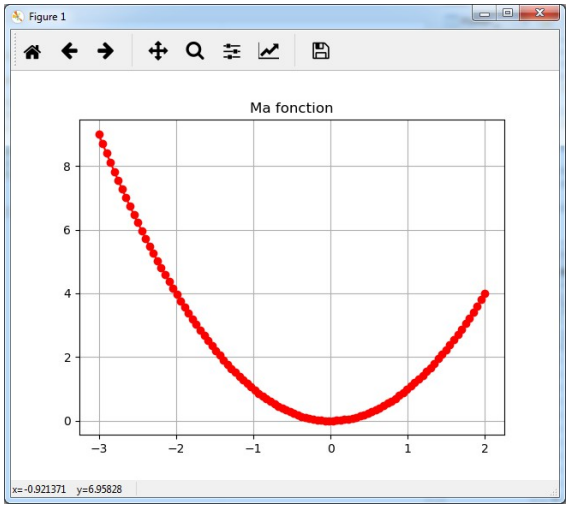
Nous pouvons bien sûr « customiser » ce graphe en affichant par exemple les 100 points ayant servi à tracer la fonction, en lui adjoignant un titre, des couleurs, etc. Vous testerez par exemple :

graphe1.py (C:\Users\Yannick\Documents\Livre Yannick modélisation\graphe1.py) - Interactive Editor for Python

Fichier Édition Affichage Paramètres Shell Exécuter Outils Aide

graphe1.py

```
#Mon premier graphe en Python

#importation des bibliothèques numpy et matplotlib avec alias
import numpy as np
import matplotlib.pylab as plt  #ou import matplotlib.pyplot as plt

x=np.linspace(-3,2,100)        #Trace de la fonction sur [-3;2], 100 points
y=[i**2 for i in x]            #on trace la fonction carre
                               #x est du type array et y du type list !

plt.plot(x,y,marker='o',color='red')  #les 100 points sont marques o et en rouge
plt.title('Ma fonction')              #le titre du graphe est Ma fonction
plt.grid(True)                        #on affiche une grille en pointilles
plt.show()
```



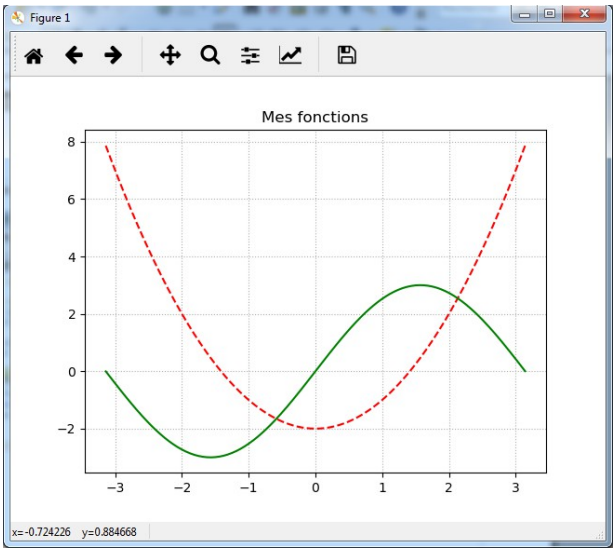
Exemple 9.1–2 : tracer les graphes de deux fonctions dans la même fenêtre

graphe1.py (C:\Users\Yannick\Documents\Livre Yannick modélisation\graphe1.py) - Interactive Editor for Python

Fichier Édition Affichage Paramètres Shell Exécuter Outils Aide

graphe1.py carre2.py dessin1.py exo9.py

```
#Mon premier graphe en Python

#importation des bibliothèques numpy et matplotlib avec alias
import numpy as np
import matplotlib.pylab as plt  #ou import matplotlib.pyplot as plt

x=np.linspace(-3.14,3.14,200)       #Trace de la fonction sur [-3;2], 100 points
y=[i**2-2 for i in x]             #on trace la fonction carre
                                  #x est du type array et y du type list !

plt.plot(x,y,color='red',linestyle='--')   #style de ligne en tirets pour la fonction1
plt.plot(x,3*np.sin(x),color='green')      #fonction 2 en vert
plt.title('Mes fonctions')                 #le titre du graphe est Mes fonctions
plt.grid(True,linestyle=':')               #on change l'aspect de la grille
plt.show()
```



Nous pouvons bien entendu légender chacune des courbes, mises dans une liste... créer une famille de courbes, etc. Et pour les utilisateurs de LATEX, exporter les figures au format png ou eps.

Examinons maintenant les utilisations de numpy via l'approche fréquentiste des probabilités ou utilisant les lois exactes. Ceci est tout à fait exploitable en classe.

9.2. random, numpy et les probabilités

random et numpy sont deux bibliothèques qui contiennent certains modules portant le même nom mais dont la syntaxe est différente. Il est donc utile de connaître ces doublons de façon à les différencier et comprendre et corriger les éventuelles erreurs de syntaxe ou de l'interpréteur.

9.2.1. utiliser numpy pour mettre en œuvre probabilités et simulations probabilistes

Nous ne prétendons pas rivaliser avec le logiciel spécialisé **R**, cependant, au niveau du secondaire, les outils offerts par numpy sont bien suffisants pour des présentations en classe ou même pour faire réfléchir (un peu) nos élèves ! Donnons quelques commandes essentielles :

```
import numpy.random as npr
```

Syntaxe générale :
my_sample = npr.ma_loi(paramètres, taille_du_tableau)

Les résultats des commandes numpy sont en général des objets de type "numpy.ndarray" (attention array n'est pas un type), c'est à dire des tableaux n-dimensionnels constitués de listes de même type et de même longueur. De tels tableaux peuvent se construire avec la commande **numpy.array(...)**.

9.2.1.1. Quelques outils de simulation de distributions de variables aléatoires continues

npr.random(d1, d2, ...)
#tableau de V.A. uniformes indépendantes sur [0;1]

npr.sample(d1, d2, ...)
#tableau de V.A. uniformes indépendantes sur [0;1]

npr.rand(d1, d2, ...)
#tableau de V.A. uniformes indépendantes sur [0;1]

npr.uniform(low = a, high = b, size = n)
#même chose sur [a;b]

***Remarque* :** size = n peut être remplacé par size = (d1, d2, ...), comme partout dans ce qui suit.

npr.randn(d1, d2, ...)
#tableau $d_1 \times ... d_N$ de V.A.I. de loi normale centrée réduite

9.2.1.2. Quelques outils de simulation de distributions de variables aléatoires discrètes finies

npr.randint(low = a, high = b, size = n)
#V.A. uniformes sur $[a\,;b] \cap \mathbb{N}$.

npr.choice([a1, ..., an], p = [p1, ..., pn], size = n)
#tirages indép. Dans [a1,...,an] de loi [p1,...,pn].

npr.permutation(mon_urne)
#permutation de mon_urne ; produit un numpy.ndarray.

npr.shuffle(mon_urne)
#permutation **in situ** de mon_urne ; conserve une liste.

npr.binomial(n, p, size = n)
#n valeurs simulées tirées d'une distribution binomiale de paramètres n et p.

npr.poisson(lam, size = n)
#n valeurs simulées tirées d'une distribution de Poisson de paramètres lam (lambda).

Beaucoup d'autres exemples sont disponibles sur :
http://docs.scipy.org/doc/numpy/reference/routines.random.html

9.2.2. random une bibliothèque qui date

```
import random as rd
```

9.2.2.1. Distributions continues

rd.random()

un tirage dans une distribution uniforme [0;1].

rd.uniform(a, b)

#**un** tirage dans une distribution uniforme sur [a;b].

rd.gauss(mu, sigma)

#**un** tirage dans un distribution gaussienne de paramètres mu et sigma.

9.2.2.1. Tirages équiprobables et distributions discrètes

rd.randrange(start, stop, step)

#**un** tirage équiprobable dans range(start, stop, step).

rd.randint(a, b)

#**un** tirage équiprobable dans range(a, b +1).

rd.sample(population, k)

#tirage sans remise équiprobable de k éléments dans population. **N'a rien à voir avec le sample de numpy.**

rd.choice(seq)

#**un** tirage équiprobable dans seq ; même chose que rd.sample(seq, 1).

rd.shuffle(seq)

#permutation aléatoire **in-situ** de seq.

9.2.3. Exemples à réaliser soit avec random soit avec numpy

Exemple 9.2–1 : Simuler une variable aléatoire discrète infinie.
On lance trois dés à 6 faces équilibrées et on fait la somme **S** des 3 valeurs obtenues :

- si la somme obtenue est supérieure ou égale à 15 on gagne la partie,
- si elle est comprise entre 8 et 14, **on relance** les trois dés,
- sinon on perd la partie.
- Soit **Z** la v.a. prenant pour valeur 1 pour une partie gagnée et 0 pour une partie perdue
- Soit **Y** la v.a. prenant pour valeurs le nombre de lancers ayant mené à une partie gagnante.

1. Écrire une fonction une_partie() qui renvoie 1 si la partie simulée est gagnée et 0 sinon,
2. **a)** Écrire une fonction frequence(N) qui renvoie la fréquence de parties gagnées sur N simulations (N choisi par l'utilisateur),
   b) Écrire une fonction qui trace le diagramme en barres de la distribution simulée de **Z**.

Exercice 9.2–1 :

En reprenant les données de l'**exemple 9.2–1**, tracer le diagramme en bâton, ainsi que les fonctions de répartition, de la loi de probabilité simulée des variables aléatoires suivantes (on prendra N=100 000) :

a) Y : Temps d'attente (en nombre de lancers) du premier gain,

b) X : Nombre de parties gagnées sur N simulations.

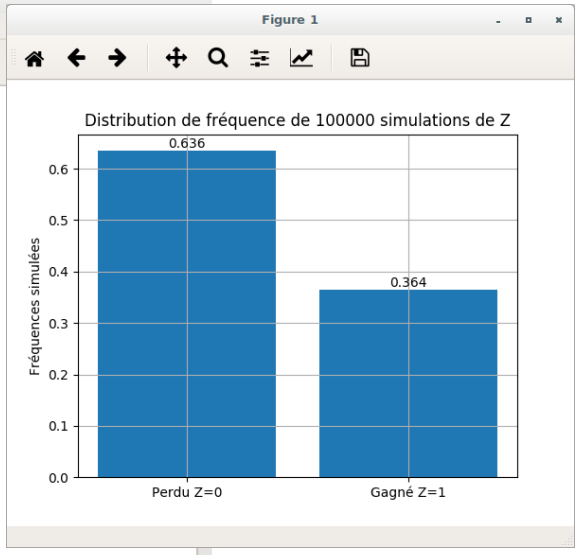
Exemple 9.2–1 : Proposition d'algorithme mettant en œuvre les commandes présentées et les recommandations vues pendant cette formation : décomposer la tâche en plusieurs parties codées par des fonctions, faire les appels aux librairies à l'intérieur des fonctions, utiliser les tuples plutôt que les nécessaires tuples suffisent, utiliser des listes plutôt que des ndarray lorsque les nécessaires listes suffisent...

3desTer-Graph.py (/home/carole/Documents/FiHub/IREM_APMEP/AnimStages/PythonAvecYann-Nimes-

Fichier Édition Affichage Paramètres Shell Exécuter Outils Aide

3desTer-Graph.py

```
"JEU SOMME DE 3 DÉS - DIAGRAMME EN BARRES DE LA DISTRIBUTION SIMULÉE DE Z"
"Z est une variable de Bernoulli prenant la valeur 0 pour une partie perdue,"
"  et 1 pour une partie gagnée"
"Exemple 9.2-1 Question 1"
"Utilise randint() de la bibliothèque random."
def une_partie():
    from random import randint
    # S = randint(1, 6) + randint(1, 6) + randint(1, 6)
    S = sum((randint(1, 6) for i in range(3)))
    while S > 7 and S < 15:
        S = sum((randint(1, 6) for i in range(3)))
    if S >= 15:
        Z = 1
    elif S <= 7:
        Z = 0
    return Z

"Exemple 9.2-1 Question 2"
"Fréquence simulés du nombre de parties gagnées (G) lors de N simulations."
"Utilise une_partie() et randint() de la bibliothèque random."
def frequence(N):
    G = 0
    for i in range(N):
        G = G + une_partie()
    return G / N

"Exemple 9.2-1 Question 2 Diagramme en barres de"
"    la distribution des fréquences simulées de Z."
"Utilise frequence(N), une_partie(), randint() de random et la bibliothèque matplotlib.pylab."
def bar3des (N = 5000):
    import matplotlib.pylab as plt
    ZZ = (0, 1)
    freqG = frequence(N)
    FG = (1 - freqG, freqG)
    titre = 'Distribution de fréquence de '+ str(N) + ' simulations de Z'
    plt.figure()
    plt.bar(ZZ, FG, tick_label = ('Perdu Z=0', 'Gagné Z=1'), align = 'center')
    plt.title(titre)
    plt.text(ZZ[0], FG[0], str(round(FG[0], 3)), ha = 'center', va = 'bottom')
    plt.text(ZZ[1], FG[1], str(round(FG[1], 3)), ha = 'center', va = 'bottom')
    #plt.xticks((0, 1), ('Perdu', 'Gagné'))
    plt.ylabel('Fréquences simulées')
    plt.grid()
    plt.show()
```



Exercice 9.2–2 : résoudre cet exercice de manière théorique (informatiquement!) ? à préciser ?

10. Séquences de mise en œuvre de l'algorithmique en TP de mathématiques de la seconde à la terminale.

10.1. Présentation générale

- Ces séquences sont construites pour mettre en œuvre l'algorithmique en mathématiques, telle qu'elle est prévue aux programmes de la seconde à la terminale.
- L'activité d'algorithmique telle qu'elle apparaît à travers les exercices du baccalauréat consiste à lire et interpréter un algorithme écrit en "langage naturel", à modifier ou compléter un algorithme donné pour qu'il réalise une tâche particulière, à corriger un algorithme qui contient une erreur, à choisir parmi plusieurs algorithmes celui que réalise une tâche demandée.
- Les activités proposées s'inscrivent toutes dans ce cadre, il n'est pas possible avec le peu d'heures consacrées à l'algorithmique, compte tenu de la difficulté du langage de programmation utilisé, d'aller au delà de ce type d'activité.
- Chaque séquence est présentée par une introduction détaillant son contenu puis est suivie par des fiches élèves comprenant les instructions pour réaliser le travail à effectuer.
- Les fichiers dont le nom se termine par ..._Elev.py sont les fichiers destinés à servir de support au travail des élèves afin d'éviter de perdre du temps sur la saisie des lignes de codes. Les fichiers dont le nom se termine par ...0.py sont les fichiers contenant l'intégralité des lignes de codes demandées lors des activités, plus parfois, quelques traces des recherches effectuées lors de l'élaboration des algorithmes. Ces fichiers sont à ouvrir avec un environnement de programmation permettant de numéroter les lignes de commandes, d'avoir un coloration syntaxique et l'indentation automatique, au minimum. (Spyder, Pyzo, Gedit, ...)
- La numérotation indiquée dans les introductions correspond à celle des fichiers enseignants dont les noms se terminent par ...0.py ; la numérotation indiquée dans les fiches élèves correspond à celle des fichiers dont les noms se terminent par ...0_Elev.py.
- Les activités sont de difficulté progressives au fur et à mesure de l'avancée dans les fiches. Les premières activités sont guidées par l'enseignant, les suivantes sont prévues pour être faites en autonomie.
- L'élève demande à l'enseignant de valider une activité avant de passer à la suivante.
- Les prolongement sont prévus pour les élèves ayant terminé en avance les activités prévues dans la séance et peuvent aller un peu au delà des exigence des programmes en proposant la recherche d'algorithmes nouveaux, basés cependant sur ceux vus précédemment.
- Le nombre de fiche à faire pendant la séance d'informatique est à adapter en fonction du niveau de la classe.
- Les fiches peuvent être rendues et notées à chaque séance.
- Concernant le code Python le choix est fait de limiter autant que faire se peut l'utilisation des bibliothèques spécialisées afin d'éviter l'effet "boite noire", vu que leur documentation est souvent sommaire et pas facile d'accès.
  La diversité des thèmes abordés dans les séquences montrent que l'on peut couvrir un grand éventail des notions du programme de mathématique, en les illustrant de façon rigoureuse et ludique, avec des outils de base de la programmation en Python.
  Le choix est fait, autant que faire se peut, et selon une saine pratique mathématique, lors de l'utilisation d'une bibliothèque, de n'importer que la ou les fonctions nécessaires et suffisantes pour le bon fonctionnement de l'algorithme. On a donc évité les from random import *...
- Les séquences sont :

  A) La notion de fonction en seconde générale.

  B) Un marche aléatoire : La rencontre du loup et de l'agneau (APMEP – Bulletins verts n° 515 p. 401-406 et n° 516 p. 637-639). Une version est présentée en langage **R**.

  C) Le paradoxe d'un duc de Toscane – Parier sur la somme des points de 3 dés.

D) Intégration numérique – Méthode des petits et des grands rectangles – Bac S Polynésie juin 2013 – Exercice 1 questions 2.a. et 2.b.

E) Un intervalle de fluctuation d'une variable binomiale, en première S.

F) Une autre marche aléatoire : Le robot TOM – Bac S Antilles Guyane septembre 2013 – Exercice 4 questions 1. et 2.

- Chacune des séquences est introduite en détaillant **un des algorithmes** de sa progression, qui n'est pas forcément le premier mais qui illustre un des aspects de la thématique abordée dans la séquence.
- Le tableau suivant organise le récapitulatif des différents fichiers utilisés.

Le fichiers contenant les introductions aux séquences est "SequencesPythonAuLycee.pdf". Le dossier compressé contenant tous les fichiers élèves et propositions de solutions est LesFichesPythonA-F_pdf.zip		
Fichiers de présentation et de consignes et fiches élèves	**Fichiers Python élèves (algorithmes à compléter ou à corriger)**	**Fichiers Python propositions de solutions**
A) Notions de fonction en seconde : Fonctions2nde-Docu-Elev.odt	FonctionAffinePosi2nde0_Elev.py FonctionGenePosi2nde0_Elev.py	FonctionAffinePosi2nde0.py FonctionGenePosi2nde0.py
B) Marche aléatoire (APMEP – Bulletins verts n° 515 et n° 516) : LeLoupEtLAgneau-Docu-Elev.odt	LoupAgneau0_Elev.py	LoupAgneau0.py
C) Paradoxe probabiliste au jeu de dés : DucToscane-Docu-Elev.odt	JeuDeDe0_Elev.py	JeuDeDe0.py
D) Intégration numérique méthode des rectangles (Bac S Polynésie juin 2013) : IntegrationNumeriqueDarboux-Docu-Elev.odt	Darboux0_Elev.py	Darboux0.py
E) L'intervalle de fluctuation d'une variable binomiale selon la méthode du document ressource de première S : IF-Bino-1ere-Docu-Elev.odt	BinoDistribEtIF_Elev.py	BinoDistribEtIF0_0.py
F) Une marche aléatoire dans un sujet de bac (Bac S Antilles Guyane septembre 2013) : Il n'y a pas encore de fiche élève ni de fichier Python élève.		RoboTom0.py

10.2. Prise en main des séquences

On peut, dans un premier temps, faire une lecture de l'algorithme de présentation des thématiques abordées dans chaque séquence (fichier **"SequencesPythonAuLycee.pdf"**). La prise en main des séquences pourra se faire en les réalisant avec les fiches et les fichiers élèves.