Découverte de Python

Bruno Grenet

M1 MEEF Mathématiques

Ce TP est fortement inspiré [1] de l'ouvrage **Apprendre à programmer avec Python 3** de Gérard Swinnen, publié aux éditions Eyrolles, et disponible librement en téléchargement à l'adresse `http://inforef.be/swi/python.htm`. Cet ouvrage est destiné aux lycéens et aux enseignants du secondaire.

1 Configuration de `Pyzo`

Pour les oraux du CAPES, vous aurez à utiliser le logiciel `Pyzo` pour écrire votre code. On utilisera donc ce logiciel dans les TP pour s'y habituer. C'est un logiciel libre (et gratuit) que vous pouvez facilement installer sur votre propre ordinateur en visitant le site `http://www.pyzo.org`.

1. Lancer `Pyzo` :
   — ouvrir le menu en haut (ou en bas) à gauche ;
   — taper `pyzo` ;
   — cliquer sur `Exécuter pyzo` ;
2. À la première fenêtre, cliquer sur le lien shell config dans la partie en haut à droite.
3. Rentrer les réglages suivants avant de valider avec `Done` :
   — choisir `/usr/bin/python3.5 [v3.5.2]` dans la liste déroulante `exe` ;
   — cocher la case `Use system default`.

La figure en page suivante vous montre les étapes successives. Dans la troisième fenêtre, la partie gauche est l'**éditeur de texte** et la partie en haut droite est l'**interpréteur Python**. Les deux fenêtres en bas à droite (`Source structure`) et (`File Browser`) peuvent être fermées.

1. Cet ouvrage m'a été conseillé par Mickaël Montassier, qui en a également tiré un TP d'introduction à Python consultable à l'adresse `http://www.lirmm.fr/~montassier/hdin101/etudiant.pdf`. Il constitue la seconde source d'inspiration du présent TP.

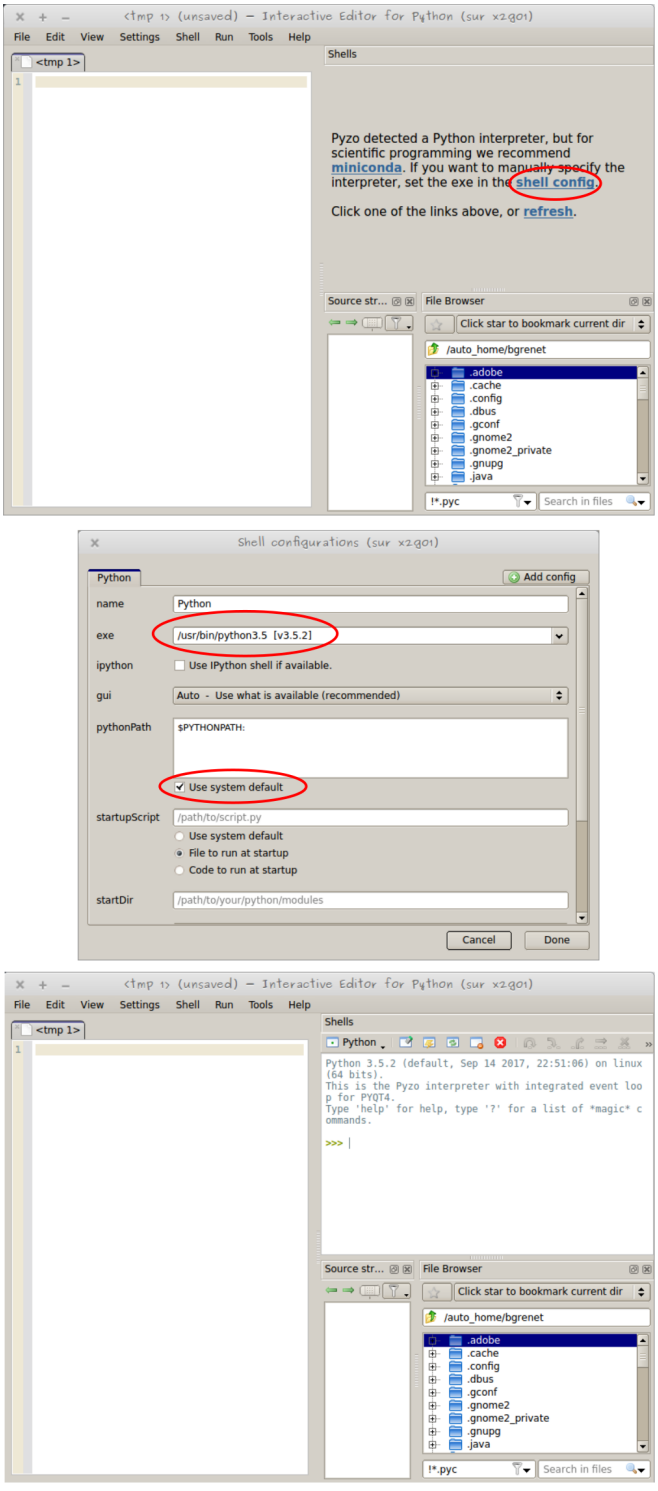


FIGURE 1 – Les 3 étapes de la configuration

2 Premier pas en Python

2.1 La calculatrice Python

On commence par jouer avec l'interpréteur Python. L'invite de commande (`>>>`) permet de taper des instructions. Taper les instructions suivantes *dans l'interpréteur*, en essayant d'anticiper la réponse de Python.

```
>>> 12 + 25
>>> 37*43
>>> -2.2 + 2.3 * 5.1
>>> (-2 + 2) * 5
>>> 4**2
>>> 2**3
>>> 20 / 3
>>> 20 // 3
>>> 20.0 // 3
>>> 20 % 3
>>> abs(-4)
```

✎ Donner la signification des opérateurs `+`, `*`, `-`, `**`, `/`, `//` et `%` d'un point de vue mathématique.

Pour des opérations plus complexes, on peut utiliser la bibliothèque `math`, de la manière suivante :

```
>>> from math import *  # Importer les fonctions de la bibliothèque math
>>> cos(2*pi/3)
>>> (sqrt(5)-1)/4
```

Python manipule deux *types* de nombres : les entiers (type `int`) et les flottants (type `float`). Ces derniers sont des approximations de nombres réels. La division *standard* (avec `/`) de deux entiers renvoie toujours un flottant (une approximation du rationnel correspondant), tandis que la division entière (avec `//`) fournit le quotient dans la division euclidienne.

Note : le comportement était différent en Python 2, mais nous n'en parlerons pas ici (les plus curieux peuvent chercher sur internet !).

2.2 Variables et affectations

Pour des calculs plus complexes, on utilise volontiers les variables. En Python, le nom d'une variable commence toujours par une lettre, et peut contenir des lettres, des chiffres, et le caractère _. Et c'est tout ! L'affectation se fait avec le signe `=`. Par exemple `x = 3*2` *affecte* à la variable `x` le résultat du calcul `3*2`. Une affectation n'affiche rien à l'écran (il n'y a pas de valeur de retour).

Prévoir ce qu'il se passe dans les lignes suivantes, et exactement ce qui va s'afficher après chaque ligne, avant de vérifier :

```
>>> n = 7
>>> n * 2
>>> m = n + 2
>>> m
>>> m = m + 1
>>> m
>>> n = m
>>> m = m - 3
>>> n
```

On peut bien sûr affecter tout autre chose que des entiers à une variable !

Quelques affectations spéciales en Python :

```
>>> x, y = 5, 6.3
>>> x
>>> y
>>> u = v = 12
>>> u, v
>>> x += 3
>>> x
```

✎ Écrire une suite d'instruction pour calculer les deux racines réelles du polynôme $X^2 - X - 1$: on dénotera par a, b et c les coefficients du polynôme, par D le discriminant, et par $r0$ et $r1$ les racines.

2.3 Listes, chaînes de caractères et uplets

Les listes Python (type `list`) se comportent comme des tableaux[2]. Une liste contient un nombre arbitraire d'éléments, de n'importe quels types. Une liste s'écrit `[a, b, c, d]`,

2. Et non des listes chaînées, pour ceux qui savent ce que c'est.

où `a`, `b`, `c` et `d` sont ici les quatre éléments de la liste. On obtient la longueur d'une liste `L` avec `len(L)` et on accède à ces éléments avec `L[i]` où `i` peut aller de `0` à `len(L)-1`. On peut également modifier une liste avec une affectation : `L[i] = b`.

Observer et prédire le comportement des commandes suivantes :

```
>>> L = [1, 2, 3, 4]
>>> L
>>> len(L)
>>> L[1]
>>> L[2] = 0
>>> L
>>> M = [5, 6.0]
>>> L + M
>>> T = [L, M]
>>> T
>>> len(T)
>>> T[1][1]
```

✎ Que fait `+` sur les listes ?

On peut également extraire une sous-liste ave l'écriture suivante : `L[a:b]` renvoie la sous-liste `[L[a], L[a+1], ... L[b-1]]`. On peut omettre `a` pour commencer au premier élément de `L` et `b` pour finir au dernier élément. De plus, on peut ajouter une valeur de *pas* : `L[a:b:p]` va de a inclus à b exclu, par pas de p.

```
>>> L = [1, 2, 3, 4, 5]
>>> L[2:4]
>>> L[:3]
>>> L[2:]
>>> L[1:5:2]
>>> L[3:0:-1]
```

✎ Trouver une manière très simple, avec l'écriture ci-dessus, d'obtenir la liste `L` dans l'ordre inverse (pour la liste de l'exemple ci-dessus, on doit obtenir la liste `[5,4,3,2,1]`).

Les *uplets* (type `tuple`) s'écrivent sous la forme `(a,b,c,d)` et s'utilisent de la même manière que les listes à une exception près : il n'est pas possible de modifier un élément d'un uplet. On les utilisent particulièrement quand un algorithme doit renvoyer plusieurs valeurs.

Les chaînes de caractères 'utilisent (essentiellement) comme des listes de caractères. Une chaîne est délimitée soit par des guillemets doubles (`"..."`) soit par des guillemets simples (`'...'`), au choix.

Observer et prédire le comportement des commandes suivantes :

```
>>> "Élu par cette crapule"
>>> 'Esope reste ici et se repose.'
>>> "Ceci n'est pas une pipe"
>>> 'Il ne faut pas "abuser" des guillemets.'
>>> ''
>>> len('chien')
>>> len("")
>>> s = "deux" + "trois"
>>> s
>>> s[3]
>>> str(789 + 875)
>>> str([1,2,3])
```

✎ Que fait `str` sur un objet Python quelconque ?

2.4 Affichage

Jusqu'à présent, on a vu que l'interpréteur Python affichait tout seul le résultat des calculs. On peut également demander un affichage particulier avec la fonction `print`[3].

Observer et prédire :

```
print(4*5)
print("Bonjour !")
print(4, 5, 'rien')
a = 12
b = 5
print(a, 'moins', b, 'égale', a-b)
```

✎ Normalement, votre interpréteur a toujours en mémoire les valeurs des racines du polynômes $X^2 - X - 1$. Afficher à l'écran le texte :

`"Les racines de X^2-X-1 sont ... et ..."`

(où les `...` sont remplacés par les valeurs calculées précédemmment).

3. Une deuxième différence entre Python 3 et Python 2 concerne cette fonction. Allez voir sur internet si vous voulez en savoir plus.

3 Un peu d'algorithmique

Pour aller plus loin, nous allons avoir besoin de définir des fonctions et des *structures de contrôle* (boucles, branchements conditionnels, etc.). On va enfin utiliser la partie gauche de la fenêtre, l'éditeur de texte.

3.1 Les fonctions

Pour définir une fonction, on utilise le mot-clé `def`.

1. Entrer la ligne suivante dans l'éditeur de texte (y compris les « deux-points »), puis appuyer sur `<Entrée>`.

   ```
   def carre(x):
   ```

   Le curseur passe à la ligne, *décalé de 4 espaces vers la droite* : ces quatre espaces s'appellent de l'*indentation*. Contrairement à de nombreux langages, l'indentation est (très) importante en Python !
2. Continuer en tapant à la ligne suivante : `return x*x` (en gardant l'indentation).
3. Appuyer maintenant sur `<Ctrl> + <Entrée>` et observer l'interpréteur.
4. Dans l'interpréteur, taper `carre(12)` (et observer, encore...).

Une fonction en Python est définie avec le mot clé `def` suivi du nom de la fonction (ici `carre`), puis entre parenthèses les paramètres de la fonction séparés par des virgules (ici un seul paramètre : `x`), puis (*très important !*) le symbole « : ». On passe ensuite à la ligne et tout le *corps de la fonction* doit être indenté par rapport la première ligne. Après éventuellement plusieurs calculs, la fonction renvoie un résultat avec le mot clé `return`.

Dans `Pyzo`, appuyer sur `<Ctrl> + <Entrée>` permet d'exécuter le code écrit dans l'éditeur de texte du côté de l'interpréteur. En réalité, on peut exécuter différentes parties du code avec différents raccourcis clavier ou en utilisant les boutons du menu `Run` (allez jeter un œil !). Pour définir différentes *cellules* (`cell` dans le menu), on insère dans le fichier une ligne contenant au moins deux fois le caractère `#` (qui indique les commentaires en Python).

Recopier la fonction `mystere` ci-dessous dans l'éditeur de texte, dans une nouvelle cellule, et essayer de prévoir ce qu'elle calcule. Vérifier ensuite en exécutant le code dans l'interpréteur et en appliquant la fonction à différentes valeurs.

```
def mystere(u, v):
    u += v
```

```
    v = u
    u -= v
    return u, v
```

✎ Écrire une fonction `discriminant` qui étant donné trois variables a, b et c, renvoie le discriminant du polynôme $aX^2 + bX + c$.

✎ Écrire une fonction `aire_du_cercle` qui étant donné d renvoie l'aire d'un cercle de diamètre d.

Avant de continuer, pensez à sauvegarder ce que vous avez écrit dans l'éditeur de texte : soit en allant dans `File > Save` (ou `Save as`), soit en appuyant sur `<Ctrl> + S`. Sauvegardez le fichier avec le nom que vous voulez, où vous le souhaitez sur l'ordinateur, mais **soyez sûr d'être capable de retrouver votre fichier plus tard !** À partir de maintenant, pensez à **sauvegarder régulièrement votre travail**.

3.2 Le branchement conditionnel

Le branchement conditionnel prend la forme suivante en Python :

```
if <condition 1>:
    <instructions si condition 1 vérifiée>
elif <condition 2>:
    <instructions si condition 1 non vérifiée mais condition 2 vérifiée>

...

else:
    <instructions si aucune condition n'est vérifiée>
```

On remarque l'indentation, comme pour les fonctions. S'il n'y a qu'une condition, il n'y aura pas de bloc `elif <condition 2>:`. De même, le bloc `else:` est facultatif : ne pas en mettre revient à dire « sinon ne rien faire ».

Pour les `<conditions>` on utilise des *tests* qui renvoie un *booléen* (`True` ou `False`). Par exemple : `x == y` (oui, deux signes =!) teste si $x = y$, `x >= y` si $x \geq y$, etc. Pour combiner deux conditions, on peut utiliser les opérateurs booléens `and` (et), `or` (ou) et `not` (non).

Prédire ce que fait le code suivant :

```
>>> if 2 + 2 == 5 or (3 < 4 and not 4 < 5):
...     print("Condition vérifiée !")
... else:
...     print("Condition non vérifiée !")
```

✎ Écrire un programme qui prend en paramètre trois variables a, b et c et renvoie le nombre de racines réelles du polynôme $aX^2 + bX + c$. *On pourra réutiliser la fonction* `discriminant` *définie précédemment, en écrivant par exemple quelque chose comme* `d = discriminant(a, b, c)`. Penser à tester la fonction…

3.3 Boucle `while`

Le but d'une boucle est de répéter plusieurs fois une même instruction. Par exemple le code suivant répète les `<instructions>` **tant que** la `<condition>` est vérifiée (noter l'indentation !) :

```
while <condition>:
    <instructions>
```

Prédire ce que fait le code suivant :

```
>>> i = 0
>>> while i < 10:
...     print(i)
...     i += 1
...
>>> i
```

✎ La suite de Syracuse est définie comme suit : on choisit un entier de départ, puis on itère le processus suivant : si l'entier est pair, on le divise par 2 ; sinon on le multiplie par 3 et on ajoute 1. Autrement dit, c'est la suite définie par une valeur u_0 et par la récurrence

$$u_{n+1} = \begin{cases} u_n/2 & \text{si } u_n \text{ est pair,} \\ 3u_n + 1 & \text{sinon.} \end{cases}$$

— Écrire un programme qui prend en entrée une valeur de départ u_0 et un entier n, et calcule le $n^{\text{ème}}$ terme u_n de la suite de Syracuse.
— Il est conjecturé que partant de n'importe quel entier u_0, la suite de Syracuse finit par boucler sur le cycle $1 \to 4 \to 2 \to 1 \to \cdots$. Calculer, étant donné un entier u_0, la valeur n minimale telle que $u_n = 1$.

3.4 Boucle `for`

Il existe en Python un autre type de boucle : les boucles `for`. Le code suivant répète les `<instructions>` avec la variable `x` prenant toutes les valeurs possible de la `<liste-de-valeurs>` :

```
for x in <liste-de-valeurs>:
    <instructions>
```

Prédire ce que fait le code suivant :

```
>>> for x in [18, 5, 6, 2, 3, 17]:
...     if x % 2 == 0:
...         print(x)
...
```

Un cas particulier d'utilisation est la construction `range` : `range(a, b, p)` renvoie la liste[4] `[a, a+p, a+2*p, ..., a+k*p]` telle que $a + kp < b$. Si on omet le *pas* `p`, il vaut `1` par défaut et `range(a,b)` renvoie la liste `[a, a+1, ..., b-1]`. On peut également omettre `a`, qui vaut alors 0. Prédire le comportement du code suivant :

```
>>> for i in range(3, 7, 2):
...     print(i)
...
>>> for i in range(5):
...     print(i)
...
>>> for i in range(13, 2):
...     print(i)
...
>>> for i in range(13, 2, -1):
...     print(i)
...
```

✎ Écrire, à l'aide d'une boucle `for`, un programme qui étant donné un entier n, calcule la somme $\sum_{i=0}^{n} i$. Écrire l'équivalent avec une boucle `while`.

La boucle `for` permet aussi de parcourir les éléments d'un uplet ou d'une chaîne de caractère :

```
>>> for x in "algorithme":
...     print(x)
...
>>> for n in (1, 3, 6, 4, 12):
```

4. En fait, il s'agit d'un itérateur, mais je ne rentre pas dans le détail.

```
...     if n % 3 == 1:
...         print(n)
...
```

3.5 Algorithmes récursifs

On peut également en Python écrire des programmes qui s'appellent eux-mêmes : il suffit pour cela, dans un programme nommé `programme` de faire appel à `... programme` ! Le programme suivant est par exemple une version récursive du calcul de la somme des n premiers entiers. Essayez de vous comprendre comment il marche avant de le tester.

```
def somme(n):
    if n == 0:
        return 0
    else:
        return n + somme(n-1)
```

✎ Réécrire de manière récursive l'algorithme qui, étant donné un entier u_0, calcule le plus petit n tel que $u_n = 1$ dans la suite de Syracuse. *Indication : le nombre d'étapes pour atteindre* 1 *depuis* u_0 *est un de plus que le nombre d'étapes pour atteindre* 1 *depuis* u_1.

4 Exercices

Le but de cette partie est de s'entraîner à programmer des algorithmes simples, puis de plus en plus compliqués, pour d'une part apprivoiser Python et d'autre part l'algorithmique. La difficulté des exercices va croissant.

On pourra s'aider du récapitulatif ci-dessous des principales fonctions du langages vues dans les parties précédentes. Retournez voir dans les parties précédentes si vous ne savez plus ce que ça fait !

Types de base :

— entier (`int`) : `2`, `-5`, ... ;
— réel approché (`float`) : `2.3`, `-5.0`, ... ;
— liste (`list`) : `[]`, `[3, 5.3, [1, 2], 4]`, ... ;
— chaîne de caractère (`str`) : `''`, `'azerty'`, `"aujourd'hui"`, ...

Opérations de base :

— nombres : `+`, `*`, `-`, `/` et `//`[5], `%` ;

5. À ne pas confondre !

— listes et chaînes : `+` (attention à la signification), `len`, `L[i]`, `s[i]` ;
— tests : `==`, `!=` ($\neq$), `<=`, `<`, `>=`, `>`.

Affectation de variables :

— `x = 2`
— `x += 1`
— `x, y = [1], 2`
— `a = b = 'c'`

Structures de contrôle[6] **:**

— fonctions : `def <mafonction>:` ;
— conditionnelle : `if <condition>:`, `elif <condition>:`, `else:` ;
— boucles : `while <condition>:`, `for <variable> in <liste-de-valeurs>:`.

Essayez dans la mesure du possible d'écrire vos solutions d'exercice dans un fichier en utilisant l'éditeur de texte, pour ne pas perdre ce que vous avez fait ! Vous pouvez les *ranger* dans des cellules, de la manière suivante (par exemple) :

```
### Exercice 1 ###

<solution de l'exercice 1>>

### Exercice 2 ###

<solution de l'exercice 2>
```

4.1 Exercices de base

1. L'affectation `a, b, c, d = 3, 4, 5, 6` affecte des valeurs aux quatre variables `a`, `b`, `c` et `d`. Écrire une suite d'affectations *simples* (du type `x = 2`) pour obtenir le même résultat.
2. L'affectation `a, b = b, a` échange les valeurs de `a` et `b`. Écrire la suite d'affectations *simples* pour obtenir le même résultat.
3. Écrire un programme qui calcule le volume d'un parallélépipède rectangle lorsqu'il reçoit en paramètre la largeur, la hauteur et la profondeur.
4. Écrire un programme qui prend en entrée un nombre (entier) de secondes et le convertit en nombre d'années, de mois, de jours, d'heures, de minutes et de secondes.
5. Écrire un programme qui étant donné n, affiche n lignes contenant 1 étoile ($*$) pour la première, puis 2 pour la deuxième, ..., puis n pour la $n^{\text{ème}}$. *Exemple avec* $n = 4$ *:*

```
*
**
***
****
```

6. Écrire un programme qui étant donné n, renvoie la liste des n premiers termes de la table de multiplication par n. *Par exemple, si $n = 6$, l'algorithme doit renvoyer la liste* $[0, 6, 12, 18, 24, 30]$.
   Écrire un deuxième programme, qui au lieu de renvoyer la liste, l'affiche à l'écran en séparant les nombre par des espaces. *Pour $n = 6$, l'algorithme affiche* `0 6 12 18 24 30`.
7. Écrire un programme qui étant donné n et m, renvoie la liste des n premiers termes de la table de multiplication par n qui ne sont pas multiples de m. *Exemple avec $n = 6$ et $m = 4$:* $[6, 18, 30, 42, 54, 66]$.
8. Écrire un programme qui étant donné une liste d'entiers, teste si tous les éléments sont strictement positifs. *On renverra la valeur* `True` *si c'est le cas, et* `False` *sinon.*
9. Écrire un programme qui étant donné une liste d'entiers, calcule le minimum de cette liste. *Attention à tester si votre algorithme fonctionne aussi lorsque certains entiers de la liste sont négatifs !*
10. Écrire un programme qui étant donné une liste d'entiers, sépare cette liste en deux : les entiers pairs d'un côté, les entiers impairs de l'autre. Le programme renverra donc deux listes. *Pour renvoyer deux objets, on peut écrire* `return x, y` *(qui renvoie le couple (x, y)).*
11. Écrire un programme qui étant donné une chaîne de caractères et une lettre ℓ, teste la présence de la lettre ℓ dans la chaîne.
    Écrire un programme qui compte le nombre de ℓ dans la chaîne de caractères.
12. Écrire un programme qui retourne une chaîne de caractère : étant donné par exemple la chaîne `"bonjour"`, l'algorithme doit renvoyer la chaîne `"ruojnob"`.
13. Écrire un programme qui teste si une chaîne est un palindrome (peut se lire dans les deux sens, comme « radar »).
14. Le conte américain « Make Way for Ducklings » met en scène huit canetons portant les noms *Jack*, *Kack*, *Lack*, *Mack*, *Nack*, *Oack*, *Pack* et *Qack*. Trouver le moyen de créer le 8-uplets contenant ces 6 noms à l'aide d'une boucel `for` *la plus simple possible*.

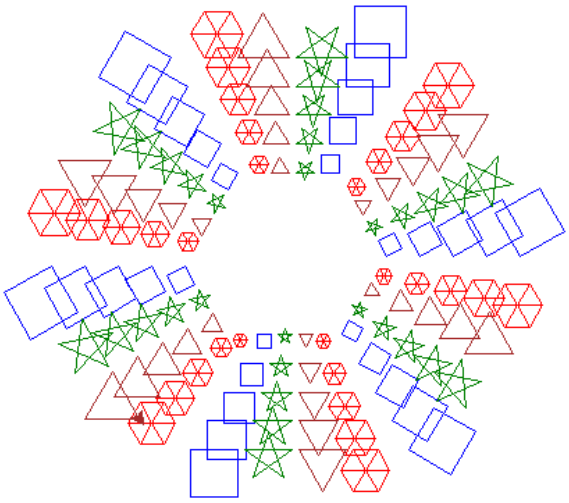
4.2 Exercices graphiques

Pour la suite des exercices, on va utiliser le module `turtle` qui permet de dessiner à l'aide d'une « tortue » virtuelle. Plutôt que de longues explications théoriques, essayez donc le code suivant :

```
>>> from turtle import * # On charge le module
>>> forward(120)         # Avancer de 120 pas en avant
>>> left(90)             # Tourner de 90° vers la gauche
>>> color('red')         # Prendre le style rouge
>>> forward(80)          # Avancer de 80 pas
```

Pour réinitialiser la fenêtre, on utilise `reset()`. On peut aussi lever le style avec `up()` (la tortue ne dessine plus en se déplaçant) et le rabaisser avec `down()`. Enfin on peut choisir la couleur avec `color("red")` (par exemple pour du rouge).

L'objectif est de réaliser le dessin ci-dessous[7] :



15. Écrire un programme qui prend en entrée un entier n et dessine un carré bleu de côté n.
16. Écrire un programme qui prend en entrée un entier n et dessine un triangle équilatéral marron de côté n.
17. Écrire un programme qui prend en entrée un entier n et dessine une étoile à 5 branches verte de côté n.
18. Écrire un programme qui prend en entrée un entier n et dessine un hexagone et ses trois diagonales, rouge, de côté n.
19. Écrire un programme qui produit une ligne avec un carré, une étoile, un triangle puis un hexagone.
20. Écrire un programme qui produit un hexagone donc chaque côté est une ligne de la question précédente.
21. Écrire le programme final qui produit la figure demandée.

7. Une variation avec un meilleur choix de couleurs est acceptée !

4.3 Programmes récursifs

22. Écrire un programme récursif qui calcule $n!$ lorsqu'il reçoit n en entrée.
23. Écrire un programme récursif qui teste si un entier est pair.
24. Écrire un programme récursif qui calcule le coefficient binomial $\binom{n}{k}$ en utilisant la formule du triangle de Pascal $\binom{n+1}{k+1} = \binom{n}{k} + \binom{n}{k+1}$.
25. Écrire un programme récursif qui prend en entrée un entier n et une base b et renvoie le nombre de chiffre de n en base b. *Indication : diviser un nombre par b diminue son nombre de chiffre de* 1.
26. Écrire un programme récursif qui calcule la somme des éléments d'une liste.
27. Écrire un programme récursif qui prend en entrée un entier n et deux flottants c et s tels que $c = \cos(\theta)$ et $s = \sin(\theta)$ pour un certain θ, et renvoie $\cos(nx)$ et $\sin(nx)$. *Le programme ne devra pas faire appel aux fonctions `cos` et `sin` de la bibliothèque `math` mais utiliser la formule* $\cos(n\theta) = \cos((n-1)\theta)\cos(\theta) - \sin((n-1)\theta)\sin(\theta)$ *et son équivalente sur* $\sin(n\theta)$.
28. Écrire un programme récursif qui prend en entrée une liste de *joueurs* (chaînes de caractère) et affiche la liste de tous les matches possibles entre ces joueurs (sous la forme `"Joueur1 rencontre Joueur2"`). *Indication : le premier joueur doit jouer un match contre chacun des autres joueurs, et ces autres joueurs doivent jouer tous les matches possibles entre eux.*
29. Le problème des tours de Hanoï consiste à déplacer n disques, qui ont tous un diamètre différent, d'une tour D (« départ ») à une tour A (« arrivée ») en s'aidant d'une tour I (« intermédiaire »), en respectant deux règles : on ne peut déplacer qu'un disque à la fois, et on ne peut poser un disque que sur un disque de diamètre supérieur (ou au pied d'une tour, si celle-ci est vide).

    Formellement, on suppose donc que la tour D contient au départ tous les disques, numérotés en fonction de leur diamètre de 1 à n (n étant le plus grand, au pied de la tour). On veut effectuer une suite de déplacements de telle sorte qu'à la fin, A contienne la pile des disques, dans le bon ordre.

    Écrire un programme récursif qui étant donné n, affiche à l'écran la suite des opérations à effectuer, sous la forme :

    `"Déplacer le disque i de la tour X à la tour Y"`.
30. Généraliser le programme précédent, dans un cas où on suppose que les disques peuvent se trouver initialement sur n'importe laquelle des tours (mais en respectant l'ordre d'empilement). En entrée, le programme prendra la liste des disques empilés sur chacune des trois tours.

4.4 Exercices 31 à 631

Le site `http://projecteuler.net` est une source très intéressante d'exercices de programmation, de nature mathématique. Il y a actuellement plus de six cent exercices, rangés par ordre de difficulté croissante. Les premiers sont relativement faciles (même s'ils peuvent déconcerter quand on les rencontre pour la première fois) : je vous encourage à essayer de les résoudre et à tester la validité de vos réponses sur le site. Les allergiques à l'anglais peuvent télécharger une traduction à l'adresse `https://github.com/RemiGascou/Project_Euler_FR/raw/master/Projet_Euler_FR.pdf`.

Je recopie ci-dessous le premier de ces exercices pour montrer le type de questions posées.

31. En listant tous les entiers naturels multiples de 3 ou 5 en dessous de 10, on obtient 3, 5, 6 et 9. La somme de ces multiples vaut 23. Trouver la somme de tous les multiples de 3 ou 5 en dessous de 1000.