Programmation 101

Eyyo futur(e) IC!
Plusieurs documents de soutien et de préparation en algèbre linéaire, analyse et physique existent déjà, dont le polycoaché de la section GM qui devrait être facile à trouver si ce n'est pas déjà fait. Un conseil: vaut mieux les bosser en priorité (au moins la partie algèbre linéaire pour les bachelier(e)s français(es)…).

Ce PDF est une initiation à la programmation destiné aux parfait(e)s débutant(e)s, couvrant, **en Python**, langage à la syntaxe plus intuitive que l'infâme Java qui vous sera enseigné, **les concepts fondamentaux de programmation vus les 6 ou 7 premières semaines de cours CS-107.** Travaillé sérieusement, le tout ne devrait prendre que quelques jours, vous permettant de vous familiariser avec un écran d'IDE (ie le logiciel de développement) et de vous faire un début d'intuition que vous peaufinerez en TP de prog.

Pourquoi ce PDF de 40 pages à peine quand la moitié du cours d'Introduction à la programmation traitera plus en profondeur de tout ça et ce même pas dans le même langage ?
D'abord, apprendre en même temps des éléments de syntaxe et de sémantique Java et des concepts plus généraux de programmation, on risque de tout mélanger. Connaître ne serait-ce qu'un peu un autre langage permet de comparer pour mieux comprendre.
Ensuite, en programmation comme dans les autres cours (surtout en algèbre...), si les premières semaines ne sont pas assimilées à fond, vous risquez de souffrir de sérieuses lacunes sur la fin du semestre.
Enfin, un avis personnel : le cours insiste rapidement sur des concepts certes critiques mais que je trouve un peu excessifs pour des débutant(e)s, et on peut s'y perdre assez rapidement entre ce qui sert tout le temps et ce qu'on doit simplement garder en mémoire.

Une petite note: j'avais écrit ce document, à l'origine, pour des amis en CPGE, d'où le titre original de la page ci-dessous.

On se revoit à la toute fin du doc page 50, d'ici là, bonne lecture et bon travail! ;)

Python pour CPGE Scientifiques

Ce qui suit n'est pas un cours en lui-même, c'est un plan d'apprentissage de Python avec les idées clés + des renvois vers des cours (un en particulier) vachement clairs et efficaces. Je vais essayer de pas faire trop long mais tout est important (même la préface ici lul).

Cette fiche n'abordera 'que' la partie programmation du programme d'informatique des deux premières années de CPGE scientifiques (hors option info en MPSI et MP où c'est du Caml qui est utilisé – mais la difficulté c'est le premier langage, après ça vient tout seul), qui est en fait assez light: 2 tiers de ce qu'on a vu en spé ISN. Je couvrirai deux parties 'hors-programme' à la fin.
En cours vous verrez beaucoup d'autres notions liées à l'informatique, pas mal de baratin mais aussi pas mal d'applications directes de programmation et de théories/techniques algorithmiques importantes, en relation avec des maths.

Ma référence pour les programmes de prépa, le site de LLG (en espérant que ça n'ait pas trop changé depuis 2017): https://info-llg.fr/?a=accueil
PCSI et PTSI ont exactement le même tronc commun que MPSI et MP. Sur ce site, il y a des cours et transparents pour le programme d'info, assez brouillons (surtout pour les bases de programmation), mais qui donnent une idée du reste du programme.

La référence pour apprendre Python, *Apprendre à programmer en Python 3* de Gérard Swinnen. Je vous conseille de l'acheter – la 3ème édition -, plus simple pour suivre, mais sinon voilà une version PDF: https://inforef.be/swi/download/apprendre_python3_5.pdf
Je mettrai le chapitre et le paragraphe quand je cite le bouquin parce que les pages sont décalées entre le manuel et le PDF (mais c'est le mm contenu), check le sommaire.
Même si c'est la référence je suis pas entièrement d'accord sur certains points, dont l'ordre d'apprentissage, d'où cette fiche. Je ne compte pas non plus suivre l'ordre officiel de prépa hyper brouillon.

Bon, avant de commencer, si vous avez des problèmes/questions, que ce soit pendant l'été ou pendant l'année (voire les 2 années lul), sur la programmation ou sur d'autres aspects du programme d'info, n'hésitez-pas à m'envoyer un mp là où vous voulez (pour ceux qui n'ont pas mon Discord: Dicedead #8849).

Une dernière chose: si vous travaillez tout ça pendant l'été, il est possible que lors des exercices en "TP" vous vous sentiez restreints car justement il faudra faire avec les connaissances restreintes données en cours, sauf si le prof est ok et vous laisse faire à votre guise – peu probable en prépa quand mm. Le but ici c'est surtout d'apprendre à programmer, pas juste d'adhérer au programme.

Sommaire:

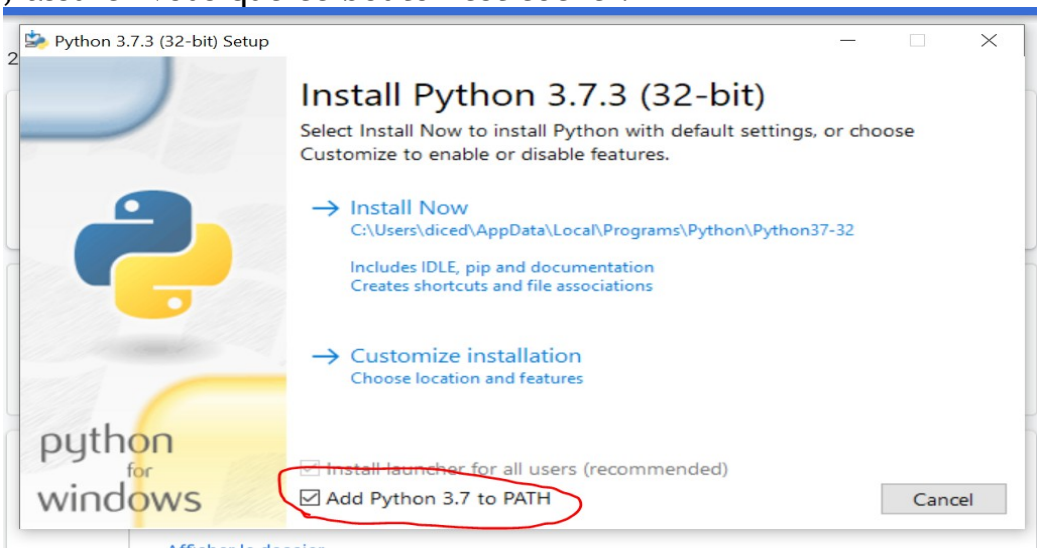
*Il y aura probablement des points, des outils que je n'expliquerai en détail qu'après dans la partie A, voire dans l'hors-programme. Je vais essayer de réduire ces sauts le plus possible car ils sont le moteur de la confusion mais parfois ce ne sera pas possible. En espérant que vous ne vous rendrez pas compte avant l'explication…
Donnez la priorité aux idées principales que j'écris ici avant le Swinnen, vous comprendrez mieux – mais bossez le Swinnen quand mm.

0) Installer Python

Windows :

1) Méthode lazy qui ira parfaitement pour la prépa:
https://edupython.tuxfamily.org/ et appuyez sur la disquette bleue, version 2.6, puis laissez vous guider, y a pas de piège. EduPython est sympa car assez fonctionnel et simple à utiliser + les étiquettes d'aide.
On aura aussi besoin de Thonny: https://thonny.org/ , Download en haut à droite.

2) Méthode normale, plus flexible mais qui sera inutile à la plupart d'entre vous:
-Téléchargez Python de https://www.python.org/ftp/python/3.7.3/python-3.7.3.exe
-ATTENTION SINON VOUS ALLEZ VOUS ENERVER POUR RIEN LUL: quand vous faites le setup, assurez vous que ce bouton est coché :



Puis cliquez sur Install Now sans crainte.
-Téléchargez un "interpréteur Python", je recommande Visual Studio Code https://code.visualstudio.com/ auquel il faudra ajouter les extensions Code Runner, MagicPython, Python et Save Typing (le carré sur la barre à gauche de l'appli), puis quand vous tapez un code, mettez bien 'MagicPython' au lieu de Plain Text en bas à droite. (ou alors PyScripter, mais alors faut télécharger Python et PyScripter en 64 bit)
-Il faudra quand mm télécharger Thonny (voir ci-dessus)

Mac et Linux :

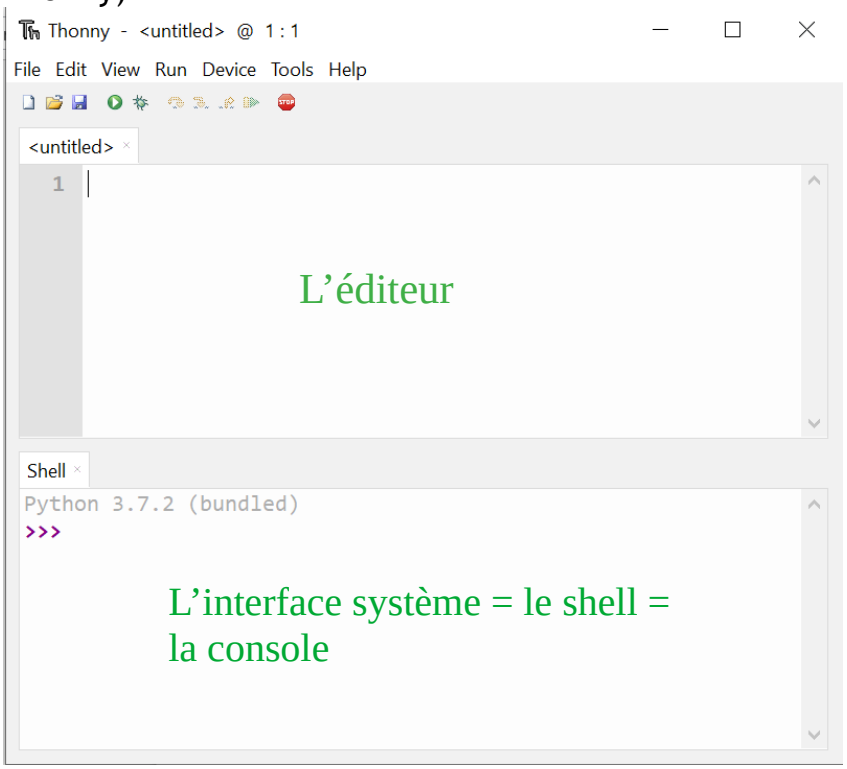
Faites direct la méthode normale ^^, tant pis pour EduPython. Visual Studio Code pour plateformes autres que Windows: https://code.visualstudio.com/#alt-downloads . Vous utiliserez Thonny au départ, passez à VSC si vous voulez après (vous comprendrez pourquoi).

A) Au programme officiel, tronc commun

0. a/Qu'est-ce que Python ?

Je vous épargne les détails de langage de 'haut-niveau' et 'bas-niveau' et 'compilateur' que vous verrez en prépa. En bref, Python est un langage de programmation qui a un codage assez intuitif et naturel, ce qui a permis son essor dans l'application aux sciences. L'énorme avantage d'apprendre à programmer en Python, c'est qu'on n'a pas les énormes complexités syntaxiques des autres langages.

Aussi, en pratique, Python tel qu'il se présente au programmeur = l'environnement Python, c'est l'ensemble de 2 choses: (c'est là qu'il faut ouvrir EduPython ou Thonny)



La différence étant que l'éditeur permet d'écrire des 'programmes' ou 'algorithmes' complets, sur plusieurs lignes, quand le shell ne permet d'écrire qu'une ligne à la fois comme 1+6 (en théorie on peut faire des blocs entiers mais ne le faites dans le shell que si vous êtes masochistes et voulez refaire votre programme à chaque fois au lieu de le sauvegarder). Appuyez sur enter et il affichera beeh 7. Dans l'éditeur on aurait eu à sauvegarder le fichier au préalable, vous comprendrez alors que par souci de rapidité on travaillera essentiellement sur le shell au début – inexistant dans Visual Studio Code.

b/ Calculs basiques

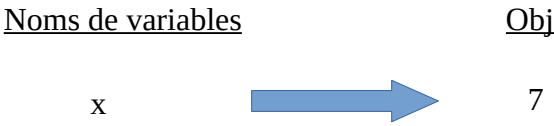
Dans Swinnen: 2. Premiers pas, Calculer avec Python

+ → addition, * → multiplication, - → soustraction, / → division, // → quotient de la division euclidienne, % → son reste, ** → la puissance. On verra le reste bien après (racines, exponentielles, etc.).

1. Variables: affectations et typages

Dans Swinnen: 2. Premiers pas, Affectation ou assignation

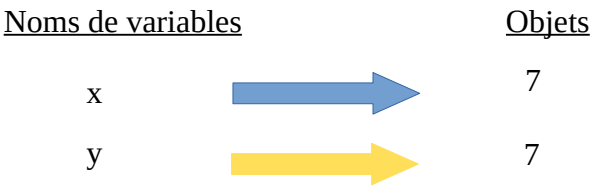
Ecrivons dans le shell: x=7 (ou x =7, ou x = 7).
En maths, on dirait que la variable entière x est de valeur 7, ie: x est égal à 7.
Dans la plupart des langages de programmation dont Python, le signe '=' ne signifie pas "est égal à", par contre la signification diffère d'un langage à l'autre.
En Python, rigoureusement: le signe '=' est un pointeur, qui, à l'objet 7, affecte la variable, le nom, la désignation x. On va dans le sens: 7 → x.
On peut dire que x 'pointe vers, désigne' l'objet 7, et c'est d'ailleurs la terminologie que j'utiliserais sûrement plus tard dans la fiche. M. Leveque nous faisait souvent ce schéma, qui sera très important dans la suite:



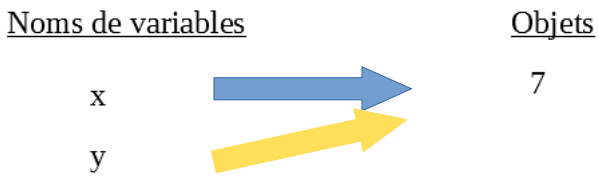
où pour chaque instruction comportant un nom de variable (= instanciation), Python tentera de pointer cette variable vers un objet en mémoire, ici, 7.
Mais sachez qu'en fait, pour l'instruction: x=7, Python fait désigner 7 par x; comprendre: Python lit les affectations du côté droit de '=' vers la gauche.
En bref, en langage humain, c'est exactement comme si x était le mot 'arbre' et 7 l'objet en lui-même qu'on appelle en français arbre. Si on vous demande de planter un arbre, vous ne pouvez pas planter le mot 'arbre' mais seulement l'objet en lui-même, d'où une inégalité de nature entre x et 7, l'arbre et l'objet.
On y reviendra dans 2 minutes.

Swinnen, 2. Premiers pas, Affectations multiples
Cela dit, on peut désigner l'objet qu'on appelle en français 'arbre' par un autre mot, 'tree' en anglais par ex. Pareil en programmation: y=7. Par contre, ça reste le même arbre, reprenons le schéma:

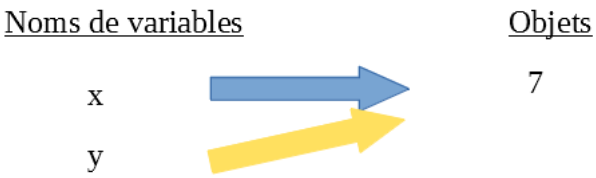
Il ne se passe PAS ça:



Mais bien ça:



Et, bon, on peut aussi taper: x = y = 7 pour obtenir le même résultat.
Plus intéressant: on tape x=7 sur une première ligne puis y=x. En aucun cas on ne pointe y vers la variable x: on pointe y vers l'objet pointé par x, on retombe en fait aussi sur ce schéma:



Ces premiers paragraphes sont assez denses mais si vous les comprenez bien, on part sur de bonnes bases pour le reste.

Prenons un autre cas, on veut affecter l'objet 5 à x et l'objet 3 à y. On peut bien sûr taper: x = 5, puis sur une autre ligne, y=3 (ce qui est recommandé mm pour les pros, pour plus de clarté) mais on peut aussi taper: x,y=5,3. Rappel: comme sur la calculatrice, la virgule n'est qu'un séparateur, et pour les décimaux, on utilise des points.
On peut également utiliser cette astuce pour permuter les pointeurs. Situation: on a déjà tapé x,y=5,3 , et on désire que y pointe vers l'objet pointé par x, donc 5, et que x pointe vers l'objet pointé par y, donc 3.

On peut écrire: x,y = y,x. C'est là que l'ordre de lecture Python est important, voilà le déroulé étape par étape:
1) Python voit x à droite, soit l'ancien x qui pointe vers 5, et l'ancien y qui pointe vers 3.
2) Il fait pointer l'objet 5 vers le nouvel y et l'objet 3 vers le nouvel x "en même temps" (bien sûr, c'est impossible, mais sauf si vous allez faire de la recherche en informatique vous n'avez pas besoin de savoir si exactement que ça).

Quelques consignes pour le nommage des variables: *Swinnen, 2. Premiers pas, Noms de variables et mots réservés*

Bon, vous avez sûrement remarqué que je prends des pincettes pour parler des objets 5, 3 et 7.

Swinnen, 2. Premiers pas, Typage des variables

Le paragraphe qui suit est propre à Python.
Ces objets 5, 3 et 7 sont tous de classe 'integer', entier en anglais (comme entier relatif). Il existe beaucoup d'autres classes bien sûr, les plus basiques et à connaître dès maintenant:
-les integers, ex: 0, 5, 3 et 7, qui ont la particularité d'avoir une précision infinie (grâce au codage binaire bien précis que chacun prend, on en parle en mp si vous

voulez)
-les floats, ie les décimaux: 6.432, 3.13333333333333, 2.5, qui ne sont précis qu'à une quinzaine de chiffres significatifs même quand ils sont finis comme 2.5
-les strings, ie les chaînes de caractère: "LUL", 'i', '5', ''. Tout ce qui est encadré par "" ou '' est une chaîne de caractère, ainsi, 5 est un integer, mais "5" est un string
-les listes: [4, "spam", "lul", x], on en reparle bientôt
-etc., on en verra bien d'autres dans la suite

Chacune comporte ses propres *méthodes*, terme qu'on définira précisément après. Un exemple de ces 'méthodes' (pas vraiment, on verra au chapitre suivant): +, quand on tape 5+5. Le résultat est 10, on dit que le renvoi de la somme d'integers 5+5 est l'integer 10 – renvoi que nous définirons plus précisément plus tard également. Certaines méthodes diffèrent d'une classe à l'autre, ou ne sont simplement pas utilisables. Par ex, la méthode / qui divise des floats et integers renvoie une erreur quand on l'applique à des strings, essayez: 'lul' / 'lul', voire même 5 / 'lul'.
Mais, 'lul' + 'lul' renvoie un autre string: 'lullul': la méthode + (aussi appelée l'opérateur +, mais c'est un cas particulier) s'applique aux strings pour effectuer la 'concaténation' = rassembler deux strings en 1. De même, 'lul'*3 renvoie 'lullullul'. Cependant, 'lul'+3 renvoie une erreur.

Se pose alors la question: dans l'instruction x=7, est-ce que x est donc de classe integer ? La réponse est non rigoureusement oui: x *hérite* de la classe de l'objet 7, qui se trouve être integer. Sauf si vous allez vous aventurer dans l'hors-programme, vous pouvez dire que: x est de classe integer. Vous pouvez le vérifier en tapant type(x).
Ainsi, si x=7, alors x*2 + 5 renvoie 19.

Cependant, x peut de changer de classe. Tapez x=7, puis x= 'lul', et vérifiez son type: string. C'est ce qu'on appelle le typage dynamique.
En maths et dans la plupart des langages de programmation, le typage statique: 'soit k un entier > 0' fixe la nature de k en tant qu'entier strictement positif. En Python, une variable beh k par ex peut désigner n'importe quoi n'importe quand.

Revenons aux affectations.
Swinnen, 4. Instructions répétitives, Réaffectation
Considérons ce bout de code:

```
>>> a = 5
>>> b = a
>>> a = 2
>>> print('a =',a,',','b =',b)
 a = 2 , b = 5
```

Ce n’est qu’une illustration de plus de ce que j’ai dit plus haut, mais à bien retenir:
Après a = 5 et b = a, voici l’état du schéma habituel:

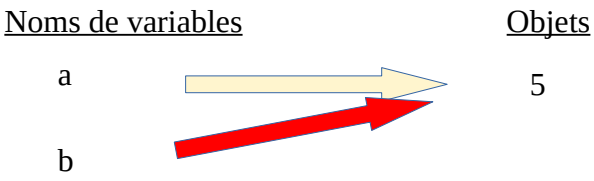
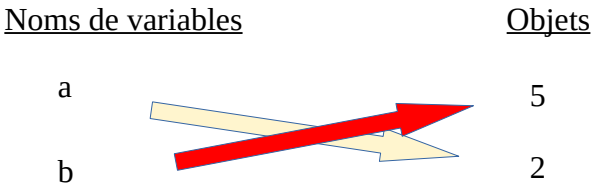


Schéma identique à celui de x et y plus haut. Cependant, à l’instruction a = 2, a pointe vers l’integer 2, mais b qui pointe bien vers 5 et non vers a à l’instruction b = a, pointe encore l’integer 5



Plus rigoureusement, en conclusion de cette partie 1:

L’affectation d’une variable à un objet fait pointer cet objet à la variable, l’objet étant d’une certaine classe qu’il transmet par héritage à la variable.

Préparation à la suite:
Physiquement, cet objet est une “référence”, une adresse dans la mémoire, un emplacement qu’a attribué Python à cet objet.
Pour voir ces emplacements, tapez id(7) par ex, ou, dans Thonny, allez dans View > Heap. Encore mieux, tapez f=8.3 puis essayez id sur f et 8.3, vous verrez que les deux n’ont pas la même adresse, on fait alors mieux encore la distinction entre noms de variables et objets pointés.

2. Fonctions part 1

(Pensez à fermer le heap si vous êtes encore dans Thonny)

Swinnen, 7. Fonctions originales: Définir une fonction, Fonction simple sans paramètres, Fonction avec paramètre, Fonction avec plusieurs paramètres, Vraies fonctions et procédures

La plupart des futurs prépas dans la classe vont en MPSI -désolé Pierre-, j'ai donc décidé de laisser de côté l'aspect algorithmique de Python qui leur sera plutôt intuitif pour me concentrer sur le vocabulaire Python. Et les fonctions beh c'est central bien sûr.

Outre les méthodes, j'ai déjà mis 'print()' quelque part. C'est en fait l'équivalent de la fonction 'afficher' de la calculatrice, à la différence près qu'elle est plus puissante.
Tapez x=5 puis print(x), le résultat est peu surprenant. Maintenant, tapez print(x*2). On peut donc faire des calculs dans print… En fait, on peut imbriquer toutes sortes de fonctions dans print, comme print(type(5)).

Essayez à présent print(print(5)), en tentant de prévoir le résultat, même si vous aurez forcément faux (pas sadique ça).
Si vous avez deviné le 5, yes, comme en maths: h(f(x)) → la fonction interne, f, s'exécute en premier, puis vient la fonction externe.
Si vous avez deviné le None, c'est que vous avez bien suivi en ISN et merci de lire ça en confirmant que je raconte pas trop d'anneries. Sinon, les autres: print est ce qu'on appelle une procédure, car son 'return' vaut None.

Explications: passez à l'éditeur, on va créer une fonction. Print() existait déjà dans la librairie standard de Python, le built-in, mais on peut très bien concevoir nos propres fonctions. Pour cela, on tape def, puis le nom de la fonction, puis, entre parenthèses, ses ou son paramètre -si elle en a. Ex: la fonction constante f(x) = 5 n'a pas besoin de paramètre. La voici avec:

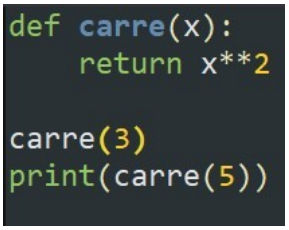
```
def constante_5(r):
    print(5)

constante_5('lul')
```

Lancez le script, le résultat est 5 – et ce, d'ailleurs, malgré le fait qu'on n'ait pas spécifier de nombre mais un string en argument dans l'appel de la fonction: constante_5('lul'). La voici sans:

```
def constante_5():
    print(5)

constante_5()
```

Remarquons que même s'il n'y a pas de paramètre (à la ligne de def) donc pas d'argument (à la ligne finale) dans les parenthèses, ces-dernières restent nécessaires. Par contre, il faut nécessairement qu'il y ait autant de paramètres que d'arguments sauf dans le cas de paramètres constructeurs qu'on verra dans la seconde partie sur les fonctions.

Ecrivons la fonction carré (qui associe à tout réel x son carré) et testons-la:



Lancez le script: que 25. Où est passé le 9 ?
return se substitue en fait à carre(x), donc l'écriture return x**2 signifie en fait: carre(x) = x**2. Ainsi, carre(3) ne fait que calculer le carré de 3, à aucun moment on ne demande à Python de l'afficher. Mais on le fait bien pour carre(5) en l'insérant dans un print.

En fait, toute fonction possède un return que nous pouvons tester en l'insérant dans print. Essayons de print(constante_5()): 5 puis None. 5, comme déjà expliqué, vient de constante_5 exécutée en premier. None est le return implicite de constante_5: car nous ne l'avons pas spécifié dans la définition de constante_5, une instruction s'est automatiquement ajoutée bien qu'invisible: return None, None étant l'unique objet de classe None, le vide en Python (Null, void dans d'autres langages).
Revenons à print(print(5)). Pareil: le return de print, telle que définie dans el built-in Python, vaut None. Et ces fonctions à return None sont appelées des procédures, ezpz.
Notons qu'on peut poser autant de paramètres qu'on veut dans la définition d'une fonction, il y aura l'occasion de s'exercer plus tard.

Et les méthodes ?
Les méthodes sont des fonctions, parfois plus précisément des procédures, propres à une unique classe. Ex: tapons en console, préférablement EduPython: spam = "lul lul lul", nous créons donc (rappel) l'objet string 'lul lul lul' en mémoire auquel pointe la variable spam.
Tapez: spam. , puis faites défiler la liste qui s'affiche (sur EduPython). Cette liste est composée de fonctions qui ne s'appliquent qu'aux strings: donc des méthodes. Testez spam.upper() par ex. +, dans le premier chapitre, n'est pas vraiment une méthode car il s'applique aux integers, aux floats et aux strings de manières différentes. Ce n'est pas non plus une fonction, c'est un opérateur, bref il n'a mm pas de type, passons…
Les méthodes nous intéresserons particulièrement dans les deux chapitres suivants.

Juste une petite précision, un détail pour l'instant qui se révèlera critique plus tard: remarquez que la mise en page d'une fonction en Python est importante: 'def' est un mot-clé qui signale à Python le début de son écriture, puis chaque ligne entrant dans la définition de la fonction.

```
1   def lul():
2       """De plus, vous pouvez utiliser 3
3       guillemets pour 'documenter' la fonction
4       = dire à quoi elle sert"""
5       print('lul'*3)
6
7   print('lol') #et '#' signale que le reste de
8                 #la ligne est un commentaire, n'imp
9                 #où dans le code
```

Le docstring (entre 3 guillemets) est ici techniquement dans la fonction.

3. Point sur les listes part 1

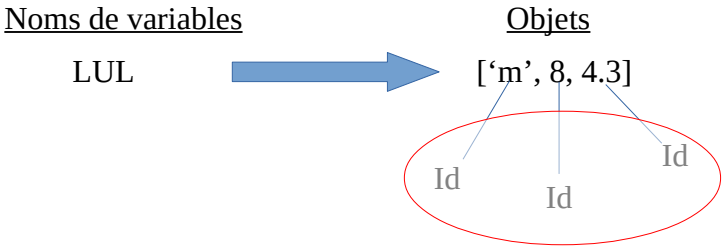
Swinnen:

5. Principaux types de données, Les listes (première approche)

10. Approfondir les structures de données, Le point sur les listes jusqu'à Opérations sur les listes incluses voire Test d'appartenance si vous voulez brûler des étapes

Les listes sont un facteur majeur entrant dans la spécificité de Python par rapport aux autres langages. Elles sont le fondement de presque tout programme Python, grâce à leur capacité infinie de rassemblement de données et de leur traitement.

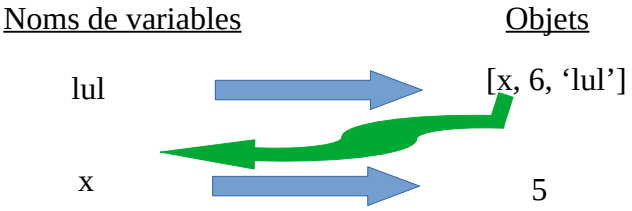
Une liste Python est une chaîne à laquelle on peut accrocher n'importe quoi, dont d'autres chaînes, et qu'on peut allonger ou raccourcir à souhait. En pratique, une liste est contenue entre deux crochets: [] est une liste vide, ['m'] est une liste contenant le string m, ['m',8, 4.3] est une liste contenant 3 éléments de classes différentes. L'instruction LUL = ['m', 8, 4.3] crée la liste en mémoire comme ci-après et la fait pointer vers LUL.



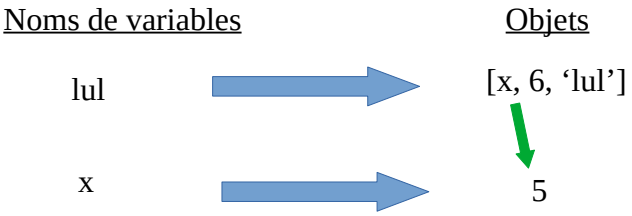
Il est important de comprendre qu'une liste ne stocke pas les objets mais les références, les adresses de ceux-ci en mémoire. Considérons ce code:

```
>>> x=5
>>> lul=[x,6,"lul"]
>>> print (lul)
[5, 6, 'lul']
```

La liste lul contient, entre autres éléments, la ‘variable’ x, qui référence en fait un 5 en mémoire. On serait tenté de faire ce schéma:



avec x dans la liste tendant vers x dans les variables. Mais, en accord avec le référencement induit par les variables et par les listes, Python va bien chercher l’objet référencé par x:



Ce qui explique le résultat suivant:

```
>>> x=5
>>> lul=[x,6,"lul"]
>>> print (lul)
[5, 6, 'lul']
>>> x=950149
>>> print(lul)
[5, 6, 'lul']
>>>
```

lul reste identique malgré le changement de pointeur de x car lul contenait en fait la référence désignée par x au départ, 5, et a ‘oublié’ x ensuite. Nous reviendrons à cette idée plus tard au point 2 sur les listes.

In the meantime, étudions l’organisation de ces listes et comment les trier.
Prenons par ex les listes [‘spam’, 4, 90] et [100,‘Pelletier’,0]. Chaque élément dans chacune de ces listes a un index (ou indice) précis qui définit sa place de gauche à droite, en commençant par 0:

Ordre tel que lu:	1er	2ème	3ème
	[‘spam’,	4,	90]
Index:	0	1	2

On peut ainsi trouver par son index tout élément d'une liste en procédant ainsi:

```
>>> test=['spam',4,90]
>>> test[0]
'spam'
```

Et ce n'est pas tout: on peut retrouver des parties de listes avec des 'slicings':

```
>>> test2=['spam',4,90,'ieif',5432]
>>> test[0:2]
['spam', 4]
```

L'intervalle [0:2] est en fait l'équivalent, en maths, de l'intervalle [0;2[comprenant uniquement des entiers, donc l'intervalle [0;1]. Notons aussi que le slicing renvoie une autre liste contenant les éléments convoqués.

Considérons à présent ces résultats, déso mais faut travailler par induction là:

a)
b)
c)

```
>>> test2=['spam',4,90,'ieif',5432]
>>> test[0:2]
['spam', 4]
>>> test[:1]
['spam']
>>> b=test[:1]
>>> print(b)
['spam']
>>> test[:]
['spam', 4, 90]
>>>
```

a) Deux choses:
*on peut ommettre le 0 en première position, Python comprendra implicitement qu'il y en a un
*même si le slicing ne contient qu'un seul élément, il renvoie quand mm une liste contenant la référence vers cet élément unique

b) Nous pouvons récupérer le renvoi (d'ailleurs aussi appelé "l'effet", mais bref) du slicing par une variable (et non juste les éléments qu'il contient)

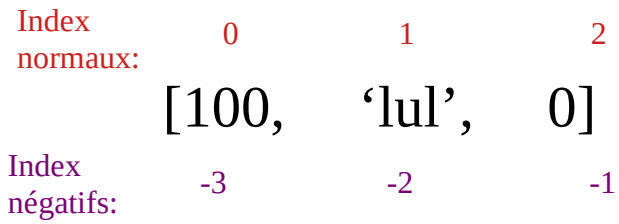
c) Comme pour le 0, nous pouvons laisser vide à droite pour aller jusqu'à la fin. Ainsi, nous pouvons très rapidement réaliser la 'copie' de cette liste. Nous verrons ce mécanisme en détail dans la part 2 sur les listes.

On peut faire encore plus avec les slicings et les indexs. Prenons notre seconde liste:

a)
b)
c)
d)

```
>>> Dre_Dre = [100,'lul',0]
>>> Dre_Dre[-1]
0
>>> Dre_Dre[-1:]
[0]
>>> Dre_Dre[:-1]
[100, 'lul']
>>> Dre_Dre[::-1]
[0, 'lul', 100]
>>>
```

a) On peut prendre les index à reculons:



Ainsi, on peut trouver l’élément en fin de liste sans avoir à connaître son vrai index, donc sans avoir à connaître la longueur de la liste. Aussi, on commence à -1 et pas 0 cette fois-ci.

b) Ce slicing indique de commencer par le dernier élément inclus, d’où le résultat.

c) Celui-ci indique de terminer par le dernier élément exclus. On peut alors mêler index positifs (0 implicite) et négatifs dans le mm slicing.

d) Nani??! La liste à l’envers et deux fois deux points??

Ajouter un troisième nombre, après deux autres points, indique le pas du slicing, qui est de 1 par défaut. Mettre un pas négatif signifie qu’on commence par la fin de la liste, d’où le résultat ci-dessus.

```
>>> LUL=[1,2,3,4,5,6,7,8]
>>> LUL[:50:2]
[1, 3, 5, 7]
>>>
```

Remarquons aussi que nous pouvons dépasser les bornes des index de la liste sans générer d’erreur (le 50).

Au sujet de la longueur de la liste, on n'a pas à compter les éléments un à un: il existe une fonction du built-in qui s'en charge:

```
>>> Dre_Dre = [100,'lul',0]
>>> len(Dre_Dre)
3
```

len renvoie ici 3, le nombre d'éléments dans la liste, soit le dernier index, 2, +1.

Il existe beaucoup d'autres opérateurs et méthodes propres aux listes, comme:

```
>>> Dre_Dre + test
[100, 'lul', 0, 'spam', 4, 90]
>>>
```

Consultez les autres en définissant une variable pointant vers une liste (ex: test) puis en tapant test., sur EduPython, et le déroulé des méthodes apparaîtra (sinon, google). Une autre méthode, mais que je recommande pas trop mm si c'est mieux que rien: tapez help(test) en console, ça vous donnera la liste complète + les explications de chaque méthode.
Les plus importantes sont, en partant de Dre_Dre ci-dessus:

*append, pour ajouter un seul élément en fin de liste:

```
>>> Dre_Dre.append(7)
>>> Dre_Dre
[100, 'lul', 0, 7]
>>>
```

*extend, pour en ajouter plusieurs:
(compris dans une liste, typiquement – on verra les autres options dans une min)

```
>>> Dre_Dre.extend([5,'spam'])
>>> Dre_Dre
[100, 'lul', 0, 7, 5, 'spam']
>>>
```

*pop: enlève l'élément à l'index spécifié, récupérable dans une variable ou pas, au choix:

```
>>>
>>> Dre_Dre = [100,'lul',0]
>>> GG = Dre_Dre.pop()
>>> GG
0
>>>
```

Quand aucun index n'est spécifié, pop prend le dernier par défaut.

*insert: insère UN élément à l'index spécifié.

```
>>> Dre_Dre = [100,'lul',0]
>>> Dre_Dre.insert(1,'lullul')
>>> Dre_Dre
[100, 'lullul', 'lul', 0]
>>>
```

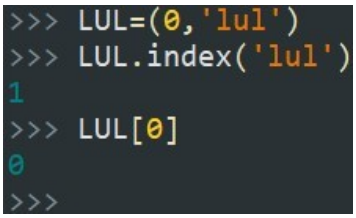
Notez que Dre_Dre reste la même liste, on ne crée pas une copie de liste à chaque fois qu'on utilise une méthode: on dit que les listes sont mutables en place. Ainsi, on peut changer 'en place' tout élément à tout index, voire plusieurs en mm temps:

```
>>> Dre_Dre = [100,'lul',0]
>>> Dre_Dre[0] = 5
>>> Dre_Dre
[5, 'lul', 0]
>>> Dre_Dre[1:50] = [4 for i in range(49)]
>>> print(Dre_Dre)
[5, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4
>>>
```

Je vous garantis qu'il y a 49 '4', mais les ranges c'est pour plus tard.

Il existe d'autres classes s'apparentant aux 'listes', formant avec elles le groupe des *itérables*, mais elles ne sont pas toutes au programme. Leur point commun: elles ne sont pas mutables en place, c'est-à-dire que lorsqu'on les a définies, on doit les copier afin de les modifier. L'avantage est qu'on ne risque pas de les modifier par erreur dans un programme de 4000 lignes, car, on verra dans la suite que l'utilisation des listes doit être avisée, elles sont sujettes à des modifications difficiles à repérer et parfois même comprendre.

L'une de ces classes secondaires (la seule au programme de prépa d'ailleurs): les tuples, qui se présentent avec des parenthèses au lieu de crochets.



Index (existant également pour les listes et, on le verra très bientôt, les strings) renvoie l'index de la première occurrence de l'élément entré. Remarquons aussi que l'index d'un tuple est entre crochets. Cependant, l'instruction LUL[0] = 'a' renvoie une erreur car le tuple est, on dit, *non mutable,* non modifiable en place. On peut aussi effectuer des slicings de tuples, mais, encore une fois, pas pour les muter.

Vous pouvez également checker les dictionnaires dans le Swinnen, mais vous n'en aurez normalement pas besoin (surtout pas en prépa).

Swinnen, 10. Approfondir les structures de données: Les tuples, Les dictionnaires

4. Point sur les strings (ou str)

Swinnen: 10.Approfondir les structures de données, Le point sur les chaînes de caractères

Profitez de ce chapitre et du suivant car ils sont, à mon avis, les plus relax: notions plutôt intuitives ou déjà vues mais ré-appliquées.

Un str est délimité, comme déjà dit, par deux paires de guillemets ou deux apostrophes. Cette image regroupe les deux moyens pour utiliser une apostrophe ou des guillemets dans le str même:

```
>>> 'c\'est l\'été'
"c'est l'été"
```

Les autres caractères spéciaux sont \t pour faire une tabulation (qlqs espaces) et \n pour le retour de ligne, annulables en posant \ devant:

```
>>> print('tabulation:\tretour à la ligne\nannulation de tabulation\\tlul')
tabulation:	retour à la ligne
annulation de tabulation\tlul
```

\ permet aussi de faire un retour de ligne si l'on juge que la ligne est trop longue sans qu'elle soit comptabilisée comme nouvelle ligne d'instruction, et ceci vaut pour tout ce qui n'est pas entre parenthèses ou crochets dans Python – dans des 'conteneurs' [] ou () on peut retourner à la ligne sans pb. Mais pour un str:

```
>>> test='lul \
... lul'
>>> print(test)
lul lul
>>>
```

Et bon, tant qu'on est sur la stylisation, une remarque sur print; considérons ce code écrit dans l'éditeur:

```
print('lul1')
print('lul2')
print('=========')
print('lul1',end='uuu')
print('lul2')
print('=========')
print('lul1',end='')
print('lul2')
```

Son output:

```
lul1
lul2
=========
lul1uuulul2
=========
lul1lul2
>>>
```

print met donc par défaut un retour de ligne à la fin de l'élément à afficher, fin qu'on peut changer avec end='T' où T est l'élément qu'on veut en fin.
Ces petites techniques se révèlent parfois cruellement nécessaires.

Parlons d'opérations et propriétés des str.
A l'image des tuples et des listes, les str comportent des index, lettre par lettre, et sont sliçables:

```
>>> stuff='spam'
>>> stuff[0]
's'
>>> stuff[::-1]
'maps'
```

En revanche, ils sont, comme les tuples, non mutables: stuff[0]='p' renvoie une erreur.

Les str sont concaténables: 'spam' + 'lul' donne le str 'spamlul'.

Ils comportent aussi certaines méthodes et fonctions, comme:

*len: ex: len(stuff) renvoie 4, comme pour les listes

*replace: remplace le premier élément dans le str par le second (les deux peuvent être plus longs qu'un seul carractère):

```
>>> frag='lul lul lil'
>>> frag.replace('l','0')
'0u0 0u0 0i0'
>>>
```

*split: la plus utilisée pour les str; récupérable dans une variable et renvoie une liste contenant des str provenant du str père, séparés par un élément entré dans split – par défaut, l'espace:

```
>>> frag='lul lul lil'
>>> frag.replace('l','0')
'0u0 0u0 0i0'
>>> frag.split()
['lul', 'lul', 'lil']
>>>
```

Ce petit approfondissement sur les str permet surtout de préparer à des exercices du Swinnen pour le prochain chapitre. En pratique, le travail sur chaînes de caractères est très important dans la manipulation d'images et de fichiers textes.

5. Boucles conditionnelles

Swinnen:
3. Contrôle du flux d'éxecution, dans sa totalité
4. Instructions répétitives: de Répétitions en boucle – L'instruction while jusqu'aux (exclus) Problèmes éventuels liés aux caractères accentués – mais avec les exos à la fin du chapitre

On va enfin écrire des programmes, des algos. Des petits, mais des programmes, algos quand mm.
D'ailleurs:

Un algorithme est un ensemble d'étapes (d'ordre souvent mathématique) permettant d'atteindre un but (souvent mathématique aussi) en exécutant un nombre fini de fois un nombre fini d'instructions.
La variante à la déf qu'on pourrait vous apporter en prépa – si on vous en apporte une d'ailleurs: en *répétant* un nombre fini de fois (etc). Après réfléxion, j'aime mieux la deuxième, car la première se rapproche un tout petit peu de celle d'un prog. Mais vous l'aurez compris: un algo sert fondamentalement à trouver un résultat chiffré à partir de données qu'on lui balance. On dit qu'il se "pense", car il résout forcément un problème.

Un programme se distingue par sa destination première à l'utilisateur humain: un jeu vidéo est un programme par ex, où encore Excel. Tant que l'humain veut l'utiliser, le programme tourne. Bien sûr, un programme peut être composé d'un ou plusieurs algorithmes, mais pas exclusivement: il a aussi une interface graphique, etc. On dit qu'il se "tape" pour son aspect plus technique et moins intellectuel (pourtant, d'expérience, faire un bon programme demande trois fois plus de rigueur et de temps que faire un bon algo).

C'est pas un secret qu'en prépa vous ferez quasi-exclusivement des algorithmes.

Vous connaissez déjà ceux écrits calculatrice. Vous serez alors contents d'apprendre que les algos en Python sont similaires mais encore plus intuitifs à écrire. Avant de voir ça, je dois vour parler de deux classes.

D'abord, les *ranges*. range(a,b) est la séquence allant de a, a+1 etc. jusqu'à b-1, comme pour les index. Donc range(2,5) est 2, 3, 4. On peut aussi ajouter un pas: range(a,b,c), c étant 1 par défaut quand il n'est pas spécifié, mais sincèrement, personne ne le fait. Ils sont utiles pour créer des itérables (listes/tuples) hyper efficacement:

```
>>> print([5 for i in range(0,50)])
[5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5,
```

Cette méthode est nommée l'instanciation de liste en compréhension. Ils sont aussi très utiles pour les boucles et peuvent prendre des variables en argument.

```
>>> x=5
>>> range(x)
range(0, 5)
>>>
```

Notons au passage que la valeur par défaut du terme de droite est 0.

Ensuite, les *bools*, ou booléens. Ils n'ont que deux valeurs: True ou False. Typiquement, ce sont les tests de comparaison, à retenir:

```
x == y      # x est égal à y (et pas juste =)
x != y      # x est différent de y
x > y       # x est plus grand que y
x < y       # x est plus petit que y
x >= y      # x est plus grand que, ou égal à y
x <= y      # x est plus petit que, ou égal à y
```

Essayez type(5>3) en console, la réponse sera class bool.
C'est aussi le test d'appartenance 'in', propres aux itérables et str, comme:

```
>>> x=5
>>> OK=[5,4,'lul']
>>> x in OK
True
>>>
```

Il existe aussi des tests d'identité, avec is et is not:

```
>>> a=5
>>> b=5
>>> a is b
True
>>> a=['lul']
>>> b=['lul']
>>> a is b
False
>>>
```

En effet, is et is not testent l'identité des adresses en mémoire. Hors, pour les integers:

```
>>> a=5
>>> b=5
>>> id(a)
500884352
>>> id(b)
500884352
```

a et b, deux variables différentes, pointent vers le même integer 5 qui a donc une adresse fixe en mémoire. Alors que pour les listes:

```
>>> a=['lul']
>>> b=['lul']
>>> id(a)
49160360
>>> id(b)
49115784
```

A chaque instanciation de la liste ['lul'], une nouvelle liste ['lul'] a été créée en mémoire, d'où les 2 adresses différentes. Nous reverrons ça au prochain chapitre.

Enfin, certains return de fonction sont des booléens aussi, comme la méthode isdigit() qui s'applique aux str (True si le str est un nombre: "123".isidigit() est True, mais "12 3".isdigit() est False).

Mais en fait, tout objet, dans Python a une valeur booléenne. Essayez bool(3>5), bool(5), bool('lul'), etc. Tous auront une valeur True ou False. Hormis les tests de comparaison, d'appartenance et d'identité à réponse fausse (du type 3 < 5), les objets qui rendent une valeur False dans Python sont:
0, None, [], "", () (et {} pour les dictionnaires vides), ainsi que ce qui se ramène à ces objets (comme 4-4 se rapportant à 0).
Tous les autres, 'lul', 90, [()], ((())), [""], 43.5, etc. sont de valeur True.

Pourquoi est-ce important ?

Voici trois exemples de blocs (ou arbres) conditionnels, avec des opérateurs familiers: if, else, while, and, or, not, et d'autres moins. Je vous conseillerais de les ouvrir dans une seconde fenêtre pour pouvoir tous les lire en mm temps que l'explication.

a)

```
a=4
b=0
if a>=4:
    print('lul')
    if not a<2 and b == 0:
        print('START B')
        b=2+a
        while b:
            print(b)
            b=b-1
        print('FINI POUR B')
        c='lul'
elif a<2:
    print(b)
else: #Donc si a==2 ou a==3 FAUX
    print(a)
```

Output / Sortie:

```
>>>
lul
START B
6
5
4
3
2
1
FINI POUR B
>>>
```

b) Je vous conseille de le tester en le collant en éditeur:

```
def nombresSVP():
    """Fonction qui demande deux nombres"""
    a=input('Nombre 1 ?')
    b=input('Nombre 2 ?')
    try:
        a=int(a)
        b=int(b)
    except:
        print('NOMBRE merdeeeeuuhh')
        nombresSVP()
```

```
def nombresSVP():
    """Fonction qui demande deux nombres"""
    a=input('Nombre 1 ?')
    b=input('Nombre 2 ?')
    try:
        a=int(a)
        b=int(b)
    except:
        print('NOMBRE merdeeeeuuhh')
        nombresSVP()

nombresSVP()
```

c) nombresSVP()

```
a=[1,2,3,4]
b=[1,2,3,4,5]
while 508:
    if a==[]:
        break
    elif a[-1]%2 == 0 ^ b[-1]%2 != 0:
        for i in range(0,5,2):
            b.append(i)
    elif a[-1]%2 == 0 or b[-1]%2 >= 0 :
        print('lul')

    a.pop()

print(b)
```

Output / Sortie:

```
lul
lul
lul
[1, 2, 3, 4, 5, 0, 2, 4]
```

Repérez déjà ce qui vous semble, au moins, familier dans ces codes, puis les nouveautés. Ensuite, passons en revue chacun d'entre eux:

a) Premier exemple de l'importance de l'indentation: plus de 'FinSi' ou de 'FinTantQue', etc. C'est l'indentation qui détermine quelles instructions rentrent dans un bloc, et sont donc à considérer et/ou à répéter.(*Swinnen: 3. Contrôle du flux d'exécution, Les limites des instructions et des blocs sont définies par la mise en page, jusau'à la fin du chapitre*)

Lisons le programme bloc par bloc, en commençant par le bloc principal – ce qui doit devenir un réflexe qui remplace la lecture de haut en bas, si ce n'est pas déjà fait.

Celui-ci est ouvert par une instruction 'si' (if) qui commence à la ligne 3: if a >= 4. On aurait pu laisser le if seul, comme nous le ferons ensuite, sans préciser d'instruction pour les cas où a < 4, mais on a choisi de définir deux autres conséquences distinctes pour deux autres ensembles auquel peut appartenir a. D'où la nécessité de préciser au moins 1 des ensembles, dans le elif (else-if / sinon, si) à la ligne 13: elif a < 2. L'autre ensemble aurait pu n'être qu'un seul nombre, ou ne pas recevoir d'instructions, mais nous voulions quand mm en ajouter: le else (sinon) à l'avant-dernière ligne, qui regroupe alors tous les autres cas de figure, soit toutes les valeurs de a dans [2;4[(ensemble de réels et non d'entiers comme j'ai écrit en commentaire) – comme dans la calculette, pas besoin de préciser cet ensemble.

Les blocs définis par elif et else sont peu intéressants bien-sûr, voyons celui commencé par if not a < 2 and b == 0.
Deux choses: d'abord, le 'et' et le 'ou' utilisés en maths sont les mm qu'en info; (Si A et B) alors C requiert que les deux soient vérifiés pour avoir C, mais (Si A ou B) alors C requiert seulement qu'au moins un des deux soit vérifié pour avoir C.
Aussi, le 'not' joue le même rôle que la barre en maths (A et son complémentaire A barre), et ne s'applique par défaut que sur la première proposition ou opération le suivant, exemple:

```
a=5
b=5
if not a == 5 or b == 5: #output: 'lul'
    print('lul')


a=5
b=5
if not (a==5 or b==5): #output: Rien
    print('lul')
```

Dans le premier cas, a == 5 donc not a == 5 n'est pas vérifié, mais b == 5 d'où l'output 'lul'. La deuxième version montre que l'utilisation de parenthèses (que ce soit pour not, and, or, is, in, etc.) permet de préciser où, ici, mettre la barre, psq a==5 or b==5 est vraie donc not (a==5 or b==5) est faux, d'où l'absence d'output.

Une dernière chose intéressante pour ce programme a, et pas la moindre: la boucle while b. Qué? 'Tant que 6?' (psq 6 est la première valeur de b printée).
En fait, après un if ou un while, Python lit le reste de la ligne comme un booléen. Donc à la troisième ligne, il lisait if bool(a>=4), le bool étant True car 4>=4. Pour while b, il lit while bool(b), donc while bool(6) au départ. Rappelez-vous, tout objet Python sauf des exceptions (les plus importantes étant 0 et la liste vide []) et des comparaisons fausses a une valeur booléenne True, donc bool(6) a une valeur True. Donc tant que bool(b) est True, while fait son boulot, qui est de print(b) puis de lui enlever 1 à sa valeur, et ce jusqu'à ce qu'il atteigne 0 où bool(b) devient donc False car bool(0) vaut False, et la boucle s'arrête. Dans ce genre de boucle, assurez-vous que vous avez bien programmé une fin car sinon elle tournerait à l'infini et ferait crasher votre ordi.

Le c='lul' est juste là pour montrer qu'on peut définir de complètement nouvelles variables au sein même d'un bloc conditionnel, sans avoir à les mentionner auparavant. Si la condition n'est jamais vraie, alors la variable c pointant vers 'lul' ne sera pas créée.

b) C'est un bout de code très, très commun dans les programmes (pas dans les algos): l'utilisateur pourrait essayer de troller le logiciel et marquer un str là où on attend de lui un nombre. Rappel: les décimaux c'est avec un point
Quand vous lancez le programme s'ouvrent successivement 2 fenêtres vous demandant un nombre, grâce à la méthode input. En effet, je dis méthode, car son return est le string de ce que vous tapez dedans: si vous tapez 5 pour le nombre 1, x pointera vers le string '5', si vous tapez "lul" (avec les guillemets), x pointera vers ' "lul" ', 4.66 devient "4.66", etc.
Ensuite, pour convertir le str en nombre, on utilise la fonction int(), qui a pour return la valeur entière, si elle existe, du nombre écrit en string ou du float introduit. Deux choses au passage:
-Faire pointer une variable vers une fonction la fait pointer vers son return, ce

que nous verrons plus en détail dans fonctions part 2;
-int() sur un float return sa troncature à l'unité.

Seulement, int('lul') par ex renvoie une erreur fatale car int ne s'applique pas aux strings, et toute erreur fatale possible est un bug majeur dans un programme – il existe des erreurs bégnignes que Python se contentera de signaler mais fera avec, mais vous n'en verrez normalement pas en prépa. On évite ce cas de figure avec le try and except.
Try commence un bloc où Python tentera d'exécuter chaque instruction. Si tout marche bien – ici, s'il peut convertir les 2 variables str en integers – il ignorera le bloc except. S'il détecte une erreur fatale, il passe au bloc except: l'objectif de celui-ci est de remédier à ces erreurs, ici: en 'faisant gentiment remarquer son erreur à l'utilisateur' en console et en le soumettant aux deux fenêtres jusqu'à ce qu'il y inscrive deux nombres.
A titre d'information, vous pouvez spécifier le type d'erreur après except, et donc, pour chaque type d'erreur, introduire une conséquence différente avec un autre except. Ici, l'erreur typique d'inscrire 'lul' et donc de faire faire int('lul') à Python est une ValueError (comme en témoigne le test en console). On pouvait alors écrire: except ValueError: puis continuer le bloc.

Aussi, la présence d'except n'est pas obligatoire avec try: si le try échoue et qu'il n'y a pas de bloc except, Python se contentera de passer ce bloc try et de ne pas l'exécuter. Un moins de forcer ce passage ce passage est le mot-clé *pass*:

Correction: non, un bloc except est nécessaire quand on utilise try, bien qu'il puisse ne contenir que 'pass'. Mais un bloc if n'a pas forcément besoin de bloc else en réponse.

```
def nombresSVP():
    """Fonction qui demande deux nombres"""
    a=input('Nombre 1 ?')
    b=input('Nombre 2 ?')
    try:
        a=int(a)
        b=int(b)
    except:
        if a=='lul' or b=='lul':
            print('ok')
            pass
        else:
            print('NOMBRE merdeeeuuhh')
            nombresSVP()

nombresSVP()
```

J'ai utilisé pass ici pour donner un moyen à l'utilisateur de faire passer 'lul' et de faire afficher un message d'abandon par Python 'ok'. Je pouvais me passer de pass avec le même résultat ainsi:

```
def nombresSVP():
    """Fonction qui demande deux nombres"""
    a=input('Nombre 1 ?')
    b=input('Nombre 2 ?')
    try:
        a=int(a)
        b=int(b)
    except:
        if a!='lul' and b!='lul':
            print('NOMBRE merdeeeuuhh')
            nombresSVP()
        else:
            print('ok')

nombresSVP()
```

(non ce n'est pas exactement une loi de Morgan mais je pourrais me tromper, fais chaud)

Au passage… nombresSVP est une procédure.

c) Trois notions et demi ici. D'abord, au lieu de stopper la boucle while quand une éventuelle variable atteint 0 ou [], on a while 508, et 508 != (différent) 0 donc bool(508) est True, ainsi, la boucle while devrait tourner à l'infini… sauf que nous choisissons de la *break* quand une certaine condition est remplie (bon, ici, il s'est trouvé que c'était a==[] qu'on aurait pu employer directement dans while, mais bon bref). Break est un mot-clé Python qui stoppe la boucle interne, donc si on avait emboité plusieurs while comme ceci:

```
while:
    while:
        while:

            break

            while:
                while:
```

Le troisième bloc while en partant du haut serait stoppé, et donc ses 2 blocs while enfants aussi en bas.

En second lieu, l'ordre des tests est ultra-important ici: le pop() réalisé sur a à la fin implique que la liste a ne doit pas être vide si on arrive à cette ligne, sous-peine d'erreur fatale (IndexError). On s'assure donc qu'elle ne l'est pas avant, ici, dès le début de la boucle, en la stoppant quand a est vide (if a==[]).

En troisième lieu, la ligne: for i in range(0,5,2). C'est l'équivalent Python du 'pour k allant de A à B', sauf que nous n'avons pas eu à définir de variable i au préalable, et bon, j'ai trouvé un moyen d'utiliser les sauts dans un range… Aussi, nous n'avons pas forcément à utiliser de ranges:

```
for i in 'lul':
    print(i)
```

```
>>>
l
u
l
>>>
```

```
for i in [5,'spam',6]:
    print(i)
```

```
>>>
5
spam
6
>>>
```

Ici, i parcourt le str et la liste d'index en index, cette technique s'appelle littéralement le parcours de liste / de chaîne de caractère et est à la base de beaucoup de programmes et algos. Il existe des algorithmes particuliers que vous verrez en prépa (plutôt 2ème année) tels les algorithmes de tri, qui font du double parcours de liste. Voici l'algorithme de tri par sélection (ou extraction), un des plus simples que vous aurez au programme car il ne nécessite qu'un niveau plutôt faible en programmation, et vous êtes déjà en mesure de le saisir

complètement (même si c'est pas si facile):

```
a = [5,1,4,20,3,1]
print('Ancienne liste =',a)

for i in range (0,len(a)):
    for j in range (i+1,len(a)):
        if a[i] > a[j]:
            a[i],a[j]=a[j],a[i] #permutation
            print(a)

print('Nouvelle liste =',a)
```

Testez le, regardez les étapes en console, pour différentes listes a si vous voulez. On apprendra plus tard comment générer une liste aléatoire. Pour comprendre son fonctionnement, partez des étapes montrées en console; essayez ensuite de trier une liste en tri sélection à la main sur papier et corrigez-vous à l'aide de l'algo. Les étapes que vous aurez suivies à la main sont identiques à celles programmées, tentez alors d'expliquer la programmation de l'algo. J'en parlerais après ce paragraphe pour pas que vous ayez la réponse sous le nez lul.

En dernier lieu, le ^ à la ligne 6 qui est le 'ou exclusif', le 'ou' français, le xor anglais: A xor B (en Python: A ^ B) n'est vraie que si soit A soit B est vérifiée. Les 2 vérifiées donne A ^ B fausse, idem pour A et B non vérifiées toutes deux.
Spoiler alert: on ne l'utilise quasiment jamais.

Dans ce chapitre 5, j'ai essayé de vous donner tous les outils logiques nécessaires à l'élaboration de vos premiers algos / programmes + quelques clés et idées pour que vous écriviez des programmes de manière pensée et non technique – j'espère au moins. Vous pouvez commencer à sérieusement travailler les exercices des 6 premiers chapitres du Swinnen – du moins je recommande pour avoir ça en tête et passer à la suite sur de bonnes bases.
En fait, on a fait l'essentiel du programme d'info, mais il vous manque encore des notions précises à la fois pour programmer et pour mieux comprendre, qui feront l'objet des 3 prochains chapitres.
Et concernant l'algo de sélection: Vous avez sûrement compris qu'on part de l'index 0 de la liste puis on cherche, parmi les prochains termes, le premier qui est plus petit, puis on permute leurs index, et ainsi de suite, et ce jusqu'à ce qu'on parcourt tout le reste de la liste, de l'index 1 jusqu'au dernier, sans trouver de plus petit. On passe alors à l'index 1, et ainsi de suite. Ce qui veut dire qu'on parcourt la liste de deux manières: une première du début jusqu'à la fin puis du deuxième index jusqu'a la fin et etc (for i in range(len(a)), et puis une seconde manière: de l'index que l'on cherche à minimiser (i) + 1, le prochain index, soit i+1, jusqu'à la fin (for j in range(i+1,len(a)). La suite est facile: si l'élément de la liste à un index j > i est inférieur (if a[j]<a[i] ou vice-versa), on les permute. Reprend alors la boucle. Si c'est encore flou, c'est normal, c'était ptet trop tôt pour ça, c'est juste un avant-goût de la prépa.

6. Point sur les listes part 2

Swinnen: 10. Approfondir les structures de données, Copie d'une liste, là seulement en guise d'intro. Pas trop compris pourquoi le Swinnen n'en parle presque pas

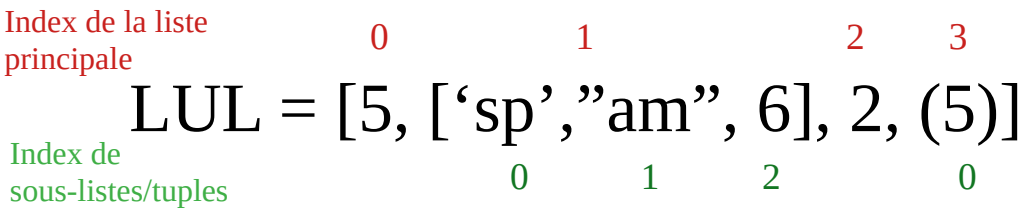
En guise d'intro de la part 1, j'avais dit qu'on peut mettre n'imp quoi dans une liste, même des listes, qu'on qualifie alors de sous-liste. Je ne l'ai pas fait car ça supposait une compréhension du système de pointeurs vers des adresses dans les listes (que je vous invite à revoir si vous avez des doutes).

Bref, une liste peut contenir des sous-listes voire des sous-sous-listes, style:

m=[5, 6, ['lul', 'spam', True], ('jesuisuntuple'("qui contient un autre tuple")), ["liste 2", ["liste 3"]], 6, None]

et, bon, les tuples aussi mais ça on s'en fout

Comment les indexer ? Ex:



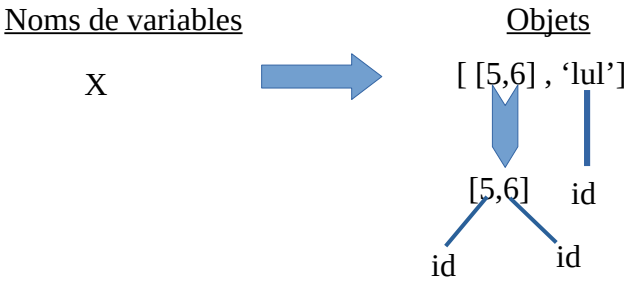
Ainsi, LUL[1] est la liste ['sp','am',6]. Soit X LUL[1]. X étant une liste, X a donc des index, donc X[0] est 'sp'. On peut donc remplacer: LUL[1][0] est 'sp', comme LUL[3] est le tuple (5) mais LUL[3][0] est l'integer 5. On ajoute alors deux crochets pour accéder à ces sous-listes, et on peut effectuer du slicing dans les derniers crochets comme expliqué précedemment, et dans les premiers aussi. Je vous laisse juger des effets par contre:

```
>>> LUL=[[5],[6]]
>>> LUL[:2][0]
[5]
>>> LUL[:2][0][0]
5
```

Mais personne ne fait ça non ironiquement lul. En plus c'est de la mauvaise gestion de liste, mieux vaut juste extraire la sous-liste désirée (x=LUL[0]) pour travailler dessus.

Voyons ce qui se passe en mémoire pour les sous-listes. Vu que ce sont toujours des listes, elles doivent toujours pointer vers des adresses. Cependant, vu qu'on y pointe via une liste principale, on pourrait croire qu'elles ont leur propre adresse.

Ce qui est vrai, prenons X = [[5, 6], 'lul']:



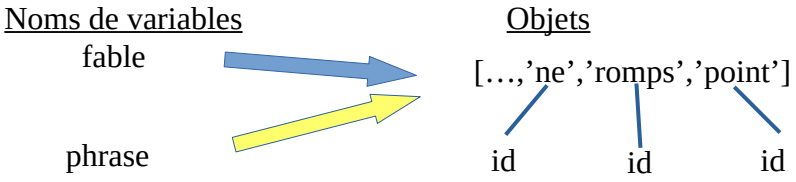
En pointant X vers cette liste, on crée cette liste en mémoire. Cette liste contient des éléments dont 'lul' qui doit être créée en mémoire. Idem pour [5,6], qui le sera aussi, mais c'est une liste, alors 5 et 6 devront être créés en mémoire aussi. Nous reviendrons à cette idée plus tard.

Si vous ne l'avez pas encore fait, lisez à présent le parag Copie d'une liste dans le Swinnen (mentionné au début du chapitre).

Pour expliquer plus rigoureusement ce qui s'y passe, voici l'état de nos pointeurs après ces lignes:

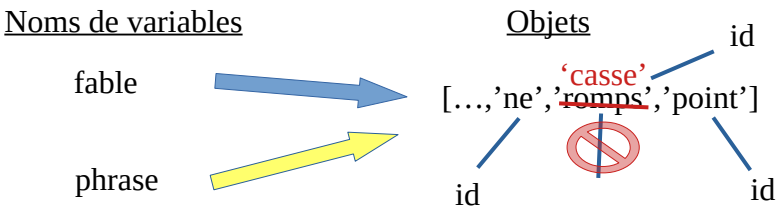
```
>>> stuff='Je plie mais ne romps point'
>>> fable=stuff.split()
>>> print(fable)
['Je', 'plie', 'mais', 'ne', 'romps', 'point']
>>> phrase = fable
>>>
```

(rappelez-vous, la méthode split)



La liste a été sauvegardée en mémoire avec fable=stuff.split() puis s'est ajouté le pointeur de phrase vers cette même liste. Cette liste est donc une *référence partagée* par ces 2 variables. Et comme l'instruction fable[4]='casse' agit sur l'objet, donc la référence, la liste est changée. Et comme la référence de cette liste est partagée, elle change pour les 2. Ce changement sur l'une des variables impactant l'autre est appelé un *effet de bord*, et c'est un grand danger de Python, quand on traite bcp de listes et bcp de variables.

Voici le schéma résumant le changement, qui s'opère dans l'adresse de l'index 4 de la liste:



Deux questions se posent:
a- pourquoi ça ne fait pas ça avec les integers qu'on avait vus au début ?
b- et surtout comment alors copier une liste ?

a- En effet:

```
>>> x = 7
>>> y = x
>>> y
7
>>> x is y
True
>>> x = x + 5
>>> x
12
>>> y
7
>>>
```

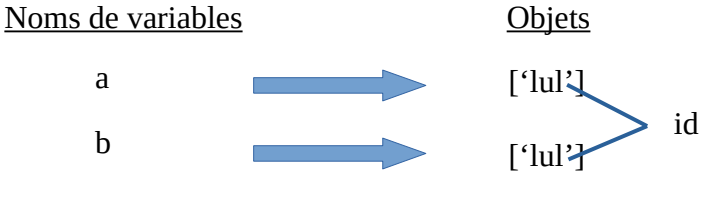
Rappelez-vous des 'vraies' étapes de pointage. On dit que 'x pointe vers 7'. En fait, on va de droite à gauche: on fait pointer 7 vers x. De même à l'instruction x = x + 5: Python lit d'abord x + 5, reconnaissant l'objet pointé par x, 7. Python déduit donc que 7+5 = 12, et doit donc faire pointer 12 vers une variable, ici, x. A aucun moment on change l'objet 7 – qui est d'ailleurs immuable, les seuls objets mutables étant les listes -, on ne fait que repointer x vers un autre integer qui se trouve être de valeur 7 + 5. Ainsi, y ne change pas de pointeur. C'est pourquoi on ne parle pas de référence partagée pour les integers même si x et y ont la même adresse (vérifié par le test 'is').

Dans l'exemple de phrase et fable, on change bien la liste, objet mutable. Elle change donc pour toutes les variables y faisant référence. (Indice pour la suite: elle peut être une sous-liste d'une liste principale)

b- Pour copier, il existe 4 méthodes.
La première est de recopier la liste à la main:
Cela produit deux listes identiques avec deux emplacements différents en mémoire, d'où le False, voici donc le schéma:

```
>>> a=['lul']
>>> b=['lul']
>>> a is b
False
>>>
```

(Une seule adresse mais différents pointeurs pour gagner de la place en mémoire)



Mais on n'a pas un ordi à disposition pour rien.

La seconde est la méthode un peu longue:

```
a=['lul',1]
b=[]
for i in a:
    b.append(i)
print('a =',a,',','b =',b)
print(a is b)
```

Output:

```
>>>
a = ['lul', 1] , b = ['lul', 1]
False
>>>
```

On ajoute les index de a un à un à b avec la méthode append (qui ajoute l'élément entre parenthèses à la fin de la liste. Résultat identique à la réécriture à la main non ?

La troisième est la plus efficace:

```
>>> a = ['lul']
>>> b=a[:]
>>> print(b)
['lul']
>>> a is b
False
>>>
```

D'où:

```
>>> a=['lul']
>>> b=a[:]
>>> a[0]=1
>>> print(a)
[1]
>>> print(b)
['lul']
>>>
```

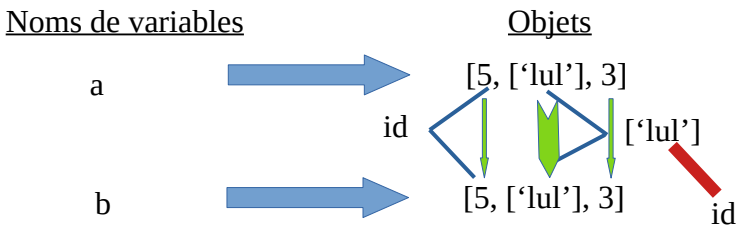
Le slicing, qui permet la copie de a en une seule ligne en créant une seconde liste en mémoire, et donc pas d'effet de bord, tout ça en une seule ligne. Je peux vous assurer que c'est absolument identique en résultat à la deuxième méthode. A la première aussi ?

Naaah la vie est pas si belle que ça quand même.

```
>>> a=[5,['lul'],3]
>>> b=a[:]
>>> print('b =',b)
b = [5, ['lul'], 3]
>>> a[0]=1
>>> a[1][0]=999
>>> print('a =',a)
a = [1, [999], 3]
>>> print('b =',b)
b = [5, [999], 3]
>>>
```

Ce que nous avons réalisé en 2 et 3 est nommé une *shallow-copy*, littéralement une copie superficielle. Elle est suffisante pour copier les éléments de la liste principale non enfermés dans une sous-liste sans effet de bord, voilà pourquoi – schéma à ce stade:

```
>>> a=[5,['lul'],3]
>>> b=a[:]
```



En vert: les copies d'index. Car en effet, le slicing et append font la même chose: copier des index entiers de la liste mère vers la liste fille, l'index étant un élément dans la mémoire – elles pointent alors vers le même élément mais avec deux pointeurs différents. Seulement, pour les sous-listes, elles ne pointent que vers la globalité de la sous-liste et non ses propres index, d'où le *pointeur commun aux 2 listes, en rouge*.
Fatalement, a[1][0]=999 change l'objet pointé par a, utilisant le même pointeur que b -> effet de bord.

La quatrième méthode résout définitivement cet effet de bord. Vous pensez peut être aux slicings de folie, mais que faites-vous si je vous pose un nombre indéterminé de sous-listes ?
La réponse est: faut tricher. On va *importer* une fonction qui fera le sale boulot.

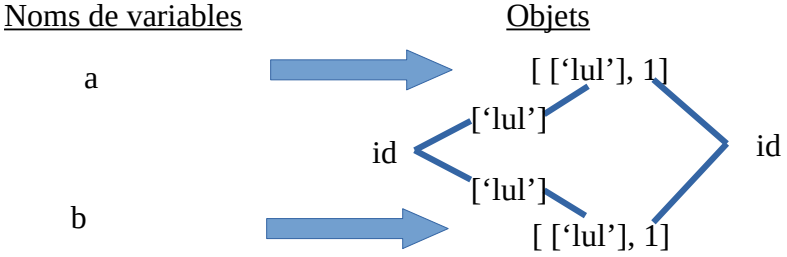
```
from copy import deepcopy

a=[['lul'],1]
b=deepcopy(a)
a[0][0]=999
print('a =',a,',','b =',b)
```

Output:

```
a = [[999], 1] , b = [['lul'], 1]
```

On parlera de cette importation dans une minute, promis. Mais en ce qui concerne la conséquence, voici le schéma à la ligne b=deepcopy(a), équivalent à celui qu'on aurait dessiné avec bcp de slicings ou en tapant b à la main:

Aucun pointeur partagé donc aucune référence partagée, les sous-listes sont recréées en mémoire => pas d'effet de bord



On peut s'assurer de la création de 2 sous-listes différentes en mémoire en ajoutant au code ci-dessus la ligne: print(a[0] is b[0]) => False, alors que pour une shallow-copy:

```
>>> a=[['lul']]
>>> b=a[:]
>>> print(a[0] is b[0])
True
```

Cependant, ne nous mettons pas à ne faire que des deepcopies. L'alias (c'est à dire le fable = phrase qui change les index superficiels), la shallow-copy et la deepcopy ont tous une utilité à un moment donné. Quand user de chacun est une habitude qui viendra avec le temps et les heures de programmation.

Dans cette part 2 sur les listes, nous avons observé les mécanismes de copie de liste afin de mieux comprendre les mécanismes de pointeurs et adresses mémoires de Python, et aussi pour savoir travailler avec des sous-listes. Celles-ci seront indispensables pour les tris fusion et bulle de prépa, tous deux beaucoup plus rapides que le tri par sélection/extraction. En outre, elles exploitent à fond la capacité infinie de concentration et d'organisation des données dans des listes.

7. Modules (maths en Python)

Swinnen:
6. Fonctions prédéfinies, Importer un module de fonctions
9. Manipuler des fichiers, Les deux formes d'importation

Le titre clickbait n'est pas pour rien. Ce chapitre est hyper simple (le prochain ne le sera pas autant, alors là pas du tout), et constituera les toutes premières de code de la quasi-totalité des algos et programmes que vous ferez en prépa.

Concernant les modules: un module est un fichier de type .py, c'est le type de fichier que vous créez quand vous sauvegardez votre code écrit en éditeur. Ces modules contiennent des variables (ex: listes particulières, valeurs particulières comme π ou la constante de Neper,...) mais aussi des fonctions et des classes. Vous pouvez utiliser ces fonctions, classes et variables définies dans un module dans un autre en les *important*. Ici, Python est très pratique car il vient avec des modules prédéfinis ultra-utiles, qui n'attendent qu'à être importés. Ils ne sont pas présents d'office dans tout programme car il y en a trop, ce qui rendrait l'exécution du programme trop demandeuse en ressources de l'ordi. Il y a une deuxième raison plus concrète qu'on verra tt à l'heure.

Avant de parler de deepcopy, parlons de l'aléatoire dans Python – promis au chapitre euuhh... 5 peut être ? Cet aléatoire est compris dans le module random. Les deux méthodes qui y sont le plus utilisées sont randint et randrange, qui ont le même but: randint(a,b) return un integer aléatoire dans l'intervalle [a;b],

randrange(a,b) return un integer aléatoire dans l'intervalle [a;b-1]. Et bon on peut ajouter une virgule, ex randint(a,b,c) tel que c est le pas. 1 est la valeur par défaut de c dans ces méthodes, mais il faut nécessairement spécifier a et b.
Je dis bien méthodes et pas fonctions: a, b et c doivent tous être des integers.

Pour les importer, il existe trois méthodes:

a) Celle que vous utiliserez le plus souvent:

```
from random import randint, randrange
```

from (nom du module) import (les fonctions, variables et classes désirées, séparées par des virgules), qui présente l'avantage d'importer un nombre limité de fonctions mais un désavantage que j'expliquerais dans b. Vous pouvez alors utiliser ces importations directement:

```
from random import randint, randrange
x=randint(0,6)
LUL=[randint(3,randint(4,20)) for i in range(randrange(1,7))]
print(x,LUL)
```

Voici quelques outputs:

```
>>>
5 [3]
>>>
*** Console de processus distant Réinitialisée ***
>>>
2 [11]
>>>
*** Console de processus distant Réinitialisée ***
>>>
0 [3, 4, 6]
>>>
```

Attention dans la création d'une liste en compréhension (comme LUL ici, avec boucle for): i doit parcourir un itérable, çàd une liste, un tuple, un str, un range... mais pas un integer. Ainsi, for i in randrange(a,b) et for i in randint(a,b) sont des erreurs de programmation car, je le rappelle, randint et randrange – malgré le nom, mal choisi – renvoient toutes deux des integers, pas des itérables.

b) Celle que vous utiliserez quand vous n'êtes pas sûrs de vous:

```
from random import *
```

L'astérisque signifie tout importer du module précisé. Randint et randrange sont alors utilisables de la même manière que montré ci-dessus, ainsi que tout ce qu'il y a dans random – nous verrons comment voir ce que le module contient dans c.

Cette méthode pose 2 problèmes. Le premier est celui du ralentissement et des ressources utilisées, dû au chargement de toutes les fonctions, classes et variables du module – qui sont généralement très, très nombreuses. Le second – et principal – et celui du conflit dans les noms. Une fonction ou variable dans un

module peut très bien avoir le même nom qu'une fonction ou variable dans un autre, dont celui où vous écrivez votre script. Comme Python lit séquentiellement (de haut en bas mais en respectant les blocs – on en parle dans le c), il n'y aura pas de message d'erreur: la dernière fonction/variable/classe d'un nom donné par rapport à la ligne lue sera celle prise en compte.

```
def spam():
    print('lul')

spam()

def spam():
    print('LUL')

spam()
```

Output:

```
>>>
lul
LUL
>>>
```

Nous verrons ce qui se passe précisément en mémoire au chapitre suivant – moins simple qu'en apparence. Mais pour le moment, dire que spam() est repointée vers la fonction {print('LUL')} est parfaitement satisfaisant.
Bien sûr, vous pouvez utiliser cette réaffectation à votre avantage pour créer des scripts plus compacts, mais ça m'étonnerait que vous le fassiez en prépa.

c) La solution à nos deux problèmes:

```
import random

x = random.randint(5,8)
```

import random ne fait que signaler à Python une adresse d'un module. Il ne regardera le contenu de random qu'à son appel, dans x = random.randint, cette instruction signifiant qu'il doit aller chercher la fonction randint dans random. Mieux, il ne lira que la fonction randint; explications.

```
1) def lul(x):
2)     if x > 5:
3)         return 'lul'
4)     else:
5)         return 'LUL'
6)
7) print(lul(4))
8) print(lul(9))
```

On pourrait croire que Python lit strictement de haut en bas et en respectant les retours, donc: 1 – 2 – 3 - 4 – 5 pour lire la fonction, puis 7, puis 1 – 2 – 3 – 4 – 5, puis 8, puis 1 – 2 – 3 - 4 – 5. En réalité, Python est plus intelligent que ça. Il ne lira que ce qui l'intéresse à un moment donné, ce qui veut dire qu'il ne lira que la ligne de def d'une fonction avant son appel, et ce n'est qu'à son appel qu'il regardera son contenu. De même, si dans un bloc if/else, une condition n'est pas satisfaite, il ne lira pas ses conséquences. L'ordre réel de lecture de ce script est

donc:
1 pour reconnaître l'existence de la fonction (= la créer et la pointer en mémoire, on verra ça au chapitre suivant), puis 7, où il lit l'appel, puis 1 – 2 – 4 – 5, puisque 4 < 5 donc la conséquence return 'lul' n'est pas lue, puis 8 où il y a un second appel, et pour finir, 1 – 2 – 3, pas besoin d'aller plus loin (9 > 5).

Conséquence: nous pouvons taper de la programmation complètement érronée dans les blocs conditionnels ou de fonctions: tant que Python n'a pas à les lire, il n'y aura pas de message d'erreur. Ex:

```
def lul(x):
    if x > 5:
        return 5/0
    else:
        return 'LUL'

print(lul(4))
```

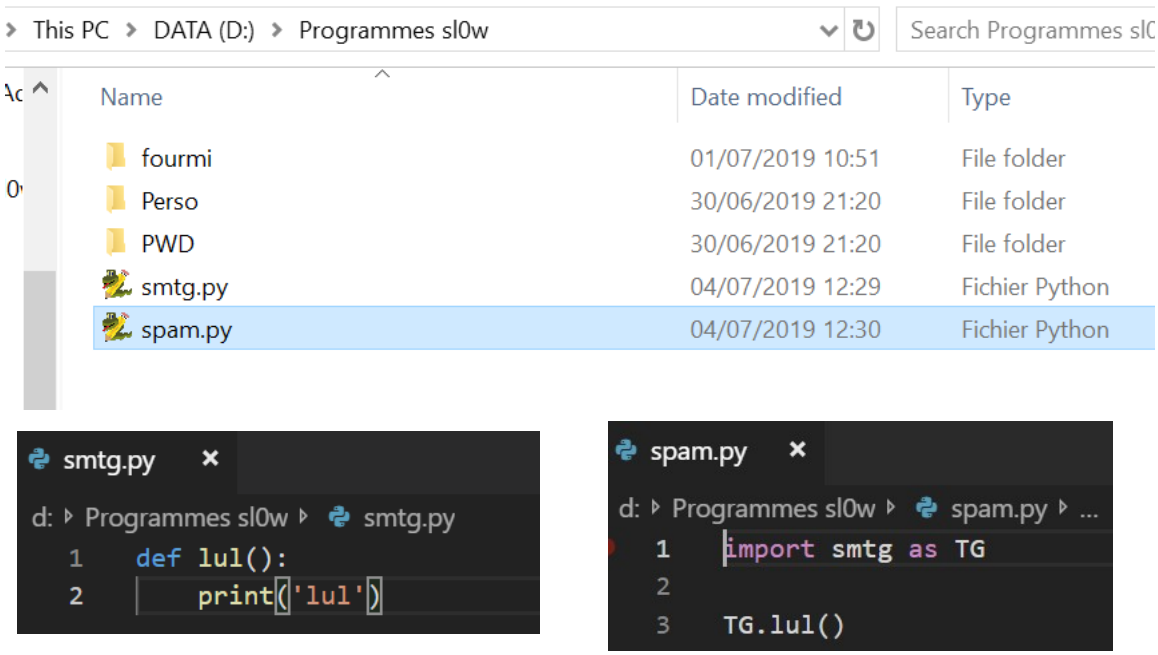
L'output est 'LUL'. Si return 5/0 avait été lue, l'output aurait été une ZeroDivisionError.
Une autre conséquence: nous pouvons taper autant de fonctions qu'on veut dans notre script; en grand nombre, elles ralentiront bien un peu le programme par la lecture des lignes de def mais tant que nous ne leur faisons pas appel elles n'auront pas d'impact sur la rapidité. Mais le problème de conflit de nommage demeure, ce qui nous ramène à cette troisième méthode d'importation.

Avec cette méthode, si 2 fonctions ont un nom identique dans deux modules différents, ex les fonctions de nom spam dans les modules lul et celsius, on pourra appeler chacune grâce à lul.spam et celsius.spam.
Si écrire le nom complet d'un module est trop long, vous pouvez le désigner par une abbréviation ainsi:

```
import random as rd

x = rd.randint(5,6)
```

Et pour voir le contenu d'un module, deux manières: dans EduPython ou Visual Code, importez le (avec la troisième méthode) puis tapez random. (avec le point) – si, bien sûr, random est le module voulu - et faites défiler la liste qui contiendra tout ce qui y est utilisable avec des descriptions. Sinon, importez le (3ème méth) et tapez help(random) par ex. Bien sûr la première méthode est plus efficace. Une troisième méthode serait de consulter directement le module en mémoire maaaiiiis bonne chance pour trouver ce qui vous intéresse.

Vous pouvez également importer les modules que vous créez vous même (rappel: un module est tout fichier .py ou .pyw, les types créés quand vous sauvegardez un programme codé en éditeur), mais c'est moins simple, sauf si le module d'où vous importez est dans le même dossier que celui où vous l'importez :



Sinon, s'ils sont plus éloignés, je vous souhaite bon courage avec les modules sys et os. On rentrerait dans de la programmation orientée système qui n'est pas le fort du Python, je vous l'accorde.
Bien sûr, les modules et les liens entre eux sont une question bien plus complexe que ce que j'ai exposé mais guère besoin de plus avant des études plutôt poussées en info.

Ah oui, j'avais promis des maths. Les modules math et numpy vous intéresseront particulièrement en prépa, math contenant des fonctions plus basiques telles la racine carrée (sqrt) et les fonctions trigonométriques (cos, sin, tan, cotan, etc., + radians et pi), et numpy des notions plus 'avancées' telles que les matrices, les complexes, etc. Vous les utiliserez pour résoudre des problèmes tels que l'approximation d'une intégrale en l'encadrant entre deux intégrales connues, tel qu'on a fait en AP. Au passage, l'aspect mathématique de ce type de calculs dominera toujours, et vous ne passerez à l'informatique qu'à la toute fin, ce qui montre bien qu'en info... vaut mieux savoir ce qu'on fait avant de faire cracher du sang à Python.

Whoops j'ai oublié deepcopy. Bref, vous avez compris ce que from copy import deepcopy fait, maintenant. Ce n'était pas un mauvais choix pour ce programme-là car je n'ai importé aucun autre module qui pouvait potentiellement avoir le même nom de fonction, ni utilisé de fonctions à nom proche. La deuxième méthode est en fait la seule à vraiment proscrire de vos scripts, à l'exception de l'importation d'un module entier dont vous connaissez parfaitement le contenu – préférablement, que vous avez créé.

Résumé : structure d'un programme Python type

```
# -*- coding:Utf8 -*-

###################################
# Programme Python type           #
# auteur : G.Swinnen, Liège, 2009 #
# licence : GPL                   #
###################################


###################################
# Importation de fonctions externes :

from math import sqrt


#################################
# Définition locale de fonctions :

def occurrences(car, ch):
    "Cette fonction renvoie le \
     nombre de caractères <car> \
     contenus dans la chaîne <ch>"

    nc = 0

    i = 0

    while i < len(ch):

        if ch[i] == car:
            nc = nc + 1

        i = i + 1

    return nc


#############################
# Corps principal du programme :

print("Veuillez entrer un nombre :")
nbr = eval(input())

print("Veuillez entrer une phrase :")
phr = input()
print("Entrez le caractère à compter :")
cch = input()

no = occurrences(cch, phr)
rc = sqrt(nbr**3)

print("La racine carrée du cube", end=' ')
print("du nombre fourni vaut", end=' ')
print(rc)

print("La phrase contient", end=' ')
print(no, "caractères", cch)
```

Un programme Python contient en général les blocs suivants, dans l'ordre :
- Quelques instructions d'initialisation (importation de fonctions et/ou de classes, définition éventuelle de variables globales).
- Les définitions locales de fonctions et/ou de classes.
- Le corps principal du programme.

Le programme peut utiliser un nombre quelconque de fonctions, lesquelles sont définies localement ou importées depuis des modules externes.
Vous pouvez vous-même définir de tels modules.

La définition d'une fonction comporte souvent une liste de PARAMÈTRES.
Ce sont toujours des VARIABLES, qui recevront leur valeur lorsque la fonction sera appelée.

Une boucle de répétition de type 'while' doit toujours inclure au moins quatre éléments :
- l'initialisation d'une variable 'compteur' ;
- l'instruction while proprement dite, dans laquelle on exprime la condition de répétition des instructions qui suivent ;
- le bloc d'instructions à répéter ;
- une instruction d'incrémentation du compteur.

La fonction "renvoie" toujours une valeur bien déterminée au programme appelant.
Si l'instruction 'return' n'est pas utilisée, ou si elle est utilisée sans argument, la fonction renvoie un objet vide : 'None'.

Le programme qui fait appel à une fonction lui transmet d'habitude une série d'ARGUMENTS, lesquels peuvent être des valeurs, des variables, ou même des expressions.

D'après Swinnen dans 7. Fonctions originales, petit document récapitulatif de la forme, sinon le fond, d'une bonne partie de ce que nous avons étudié jusqu'à présent.

8. Fonctions part 2

Swinnen:
7. Fonctions originales, Variables locales, variables globales; Typage des paramètres, Valeurs par défaut pour les paramètres, Arguments avec étiquettes
14. Et pour quelques widgets de plus..., Métaprogrammation – expressions lambda

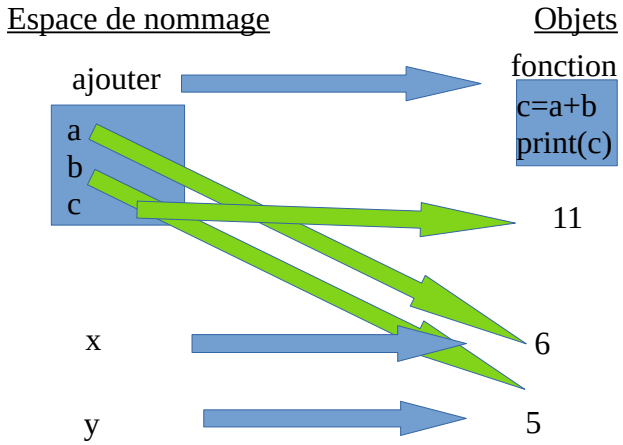
Je pense ne l'avoir pas encore précisé: une fonction est un objet de classe fonction, au même titre qu'un integer est un objet de classe integer. Ainsi, dans notre schéma, on devrait parler "d'espace de nommage" plutôt que de "noms de variables", car, comme nous le verrons plus tard, le nom d'une fonction est plus qu'une variable, puisqu'il entraîne une série d'instructions (d'où son nom de fonction).
Pour comprendre le mécanisme d'une fonction, je dois aussi vous révéler une vérité sur ces 'variables'. Elles ne sont pas toutes de nature égale.

```
def ajouter(a,b):
    return a+b

x=6
y=5
ajouter(6,5)
```

X et y, comme toute variable définie dans le corps principal du programme, sont des variables dites *globales*, telles que nous avons eu l'habitude de manipuler depuis le début. Elles ont la propriété de réaffectation, de typage dynamique (rappel: passer du pointage d'un integer à celui d'un str) + celles propres à leurs classes respectives.
A et b sont des variables dites *locales*, car elles sont locales à la fonction ajouter. Il se trouve que ce sont aussi les paramètres de cette fonction, mais on aurait pu écrire le code ainsi:

```
def ajouter(a,b):
    c=a+b
    print(c)

x=6
y=5
ajouter(6,5)
```

et là aussi, c est une variable locale, mais sans être un paramètre.
La principale propriété des variables locales est qu'elles sont supprimées après l'exécution de la fonction. Ajouter print(c) en dehors de la fonction dans ce code renvoie-donc une UnboundError, caractérisant l'absence d'une variable 'c' utilisable (globale, psq'en dehors d'une fonction). Voici le fameux schéma, durant l'exécution du programme dans la seconde photo:



Tous les pointeurs verts sont temporaires et sont supprimés après exécution de la fonction.
Remarquons au passage que la fonction ajouter est polymorphe: elle accepte plusieurs classes d'objet, comme les str, listes (concaténation) et floats.

Je m'acharne à dire 'classes' d'objets quand, en prépa, on vous parlera sûrement de 'type', ce qui est non rigoureux, bien que tentant à dire puisqu'on parle de 'typage' dynamique quand on devrait parler de classes dynamiques. Bref, ce n'est la faute qu'à la sémantique, la terminologie.

Parenthèses: il existe des typeurs dynamiques dans Python, que nous avons déjà rencontré: bool(), int(), mais aussi list(), tuple(), float()... Voilà les 5 principaux en tt cas. Ils convertissent, quand possible, un objet d'une classe vers la classe de leur nom, ex:

```
>>> a=('lul')
>>> b=list(a)
>>> b
['l', 'u', 'l']
>>>
```

Rappel: a est ici un tuple.

Revenons aux variables globales/locales. Puisque le pointeur de la variable locale est supprimé après exécution, utiliser le même nom pour une variable locale et une autre globale et parfaitement possible, bien que non recommendé car créateur de confusion. Mais il ne faut pas croire que la variable globale change avec la locale:

```
def change_b():
    b=5

b=6
change_b()
print(b)
```

Output:

```
>>>
6
>>>
```

Pour forcer ce changement, on peut utiliser le mot clé *global* dans la fonction qui signalera que le b dont on parle est la variable globale b:

```
def change_b():
    global b
    b=5

b=6
change_b()
print(b)
```

Output:

```
>>>
5
>>>
```

Je précise vite fait avant de passer à la suite qu'on peut imbriquer des définitions de fonctions:

```
def ajouter(a,b):
    def lul():
        def spam():
```

L'intérêt en prépa sera limité maaaais on ne sait jamais. On peut ainsi conditionner l'existence d'une fonction (en la mettant dans un bloc if, etc).
Parlons à présent de métaprogrammation.
Ce n'est pas une notion exigible en soi en prépa mais je doute que vos profs auront des scrupules à l'utiliser dans ses 2 niveaux, le + simple et le plus abstrait.

Commençons bien sûr par le plus simple: les valeurs par défaut dans les paramètres de fonction. Là, je ne peux rien ajouter au Swinnen, je vous renvoie donc à *7. Fonctions originales, Valeurs par défaut pour les paramètre et Arguments avec étiquettes*. On pourrait, à la limite, qualifier ces valeurs par défaut de variables *constructeurs*, bien que ce terme est plutôt réservé à la programmation orientée objet qu'on verra plus ou moins dans la partie hors-programme (msh hors-prog la elna Adam, mn shufa tene semester bas 3a Java – vous n'avez rien vu).
Ces valeurs par défaut sont bien pratiques mais question programmation elles n'apportent pas grand chose, elles ne résolvent pas un problème particulier.

Les expressions lambda si par contre, c'est le second niveau. Là aussi, je n'ai pas beaucoup plus à dire que le Swinnen. Normalement, vous n'y serez pas confrontés très souvent, mais assez pour justifier ce paragraphe.
Avant toute chose, comprenons qu'un nom de fonction n'est pas une variable. C'est une chaîne de caractère qui permet de retrouver une séquence d'exécution qui est l'objet 'fonction'. Par métonymie (en bref), on dira: un nom de fonction équivaut à une instruction.
Lambda permet de passer d'une instruction à une expression en utilisant la forme, par ex pour la fonction ajouter ci-dessus: lambda a = 6, b = 5 : ajouter(a,b), expression équivalente aux codes en photo ci-dessus. L'ennui, c'est que je n'ai pas d'exemple concret où un lambda est indispensable, à notre niveau de programmation – d'ailleurs en fait un lambda est toujours dispensable, bien que vraiment pratique. Bref, vous verrez peut être ça plus en détail en prépa ou pas. Perso, je les utilise vraiment rarement dans mes programmes.

9. Manipuler des fichiers textes

Swinnen: 9. Manipuler des fichiers, en totalité SAUF L'instruction break et Gestion des exceptions, déjà traités – mais regardez-y les exemples

Eeeet c'est là que je découvre que le module os est -juste un brin- au programme de prépa.
Bon, ici non plus, je n'ai grand chose à dire de plus que le Swinnen. En fait, il n'y a pas grand chose un dire: vous ouvrez un fichier en lui attribuant une variable ainsi: LUL = open("MonFichier.txt","w"), où "MonFichier.txt" est le nom du fichier et "w" le write, qui écrase le fichier s'il en existait un de ce nom et le crée s'il n'existait pas; il existe aussi le mode "r", read, qui ne peut que lire un fichier sans le modifier, le mode "a", append, qui permet d'écrire à la fin d'un fichier, et le mode 'r+', read + write, qui combine la lecture et le mode write. A noter que dans cette configuration, 'MonFichier.txt' doit être dans le même dossier que le module. Une remarque: utilisez vraiment des fichiers txt Notepad, pas des fichiers odt.

On peut ensuite appliquer plusieurs méthodes à ce fichier: LUL.readlines() return les lignes du fichier dans une liste, LUL.readline(X) return un str à la ligne précisée; LUL.write(string) écrit en une seule ligne au début du fichier, LUL.writelines(itérable) prend en argument une liste pour écrire des lignes dans le fichier à partir du début – remplacez à partir du début par à la fin du fichier si vous êtes en mode append 'a'.

A la fin, n'oubliez surtout pas de LUL.close() le fichier afin de ne pas vous causer des problèmes inutiles.

Eeet c'est tout ce qui est propre à la manipulation de fichiers. Bien sûr, ici, l'essentiel réside dans les exercices que vous pouvez faire dessus, qui se ramènent le plus souvent à du traitement de listes et de str que de fichiers à vrai dire.

Je choisis délibérément de ne pas vraiment parler du traitement d'images car il sera différent d'une prépa à l'autre et que GIMP et autres logiciels interpréteurs d'images ne sont pas au programme. Ce que nous avons vu en PC est globalement juste: une image est un tableau de x lignes par y colonnes donc chaque case est un bit ou plusieurs. Pour les images en noir et blanc, chaque case contient 0 si elle symbolise un pixel noir, ou 1 si c'est un pixel blanc. Pour celles en couleur, pour du codage en RGB (red-green-blue), chaque case est un tuple de 3 octets, un pour chaque couleur, avec 2^8 valeurs pour chaque (allant de 0 à 255 inclus), puis, il y a synthèse additive. Ainsi un pixel jaune est (255,255,0), ou encore (140,140,0) psq plus on se rapproche de 255 plus claire est la couleur. Donc le blanc: (255,255,255), et un exemple de gris: (120,120,120). Il n'en reste pas moins qu'une image est en fait... un fichier texte, qui se manipule de la même manière (ou d'autres manières plus efficaces qui dépendront des logiciels dans votre prépa).

10. Récursivité

Rien d'utilisable dans le Swinnen cette fois !

Ce chapitre va vous rappeler les suites lul
On parle de récursivité quand une fonction s'appelle d'elle même, quelque part dans son bloc (souvent dans le return mais pas obligatoirement). En prépa, vous vous intéresserez plus aux maths derrière, mais il y a quelques régles de programmation de fonctions récursives qu'il faut connaître pour pas faire planter son ordi.
Voyons une programmation récursive de la factorielle:

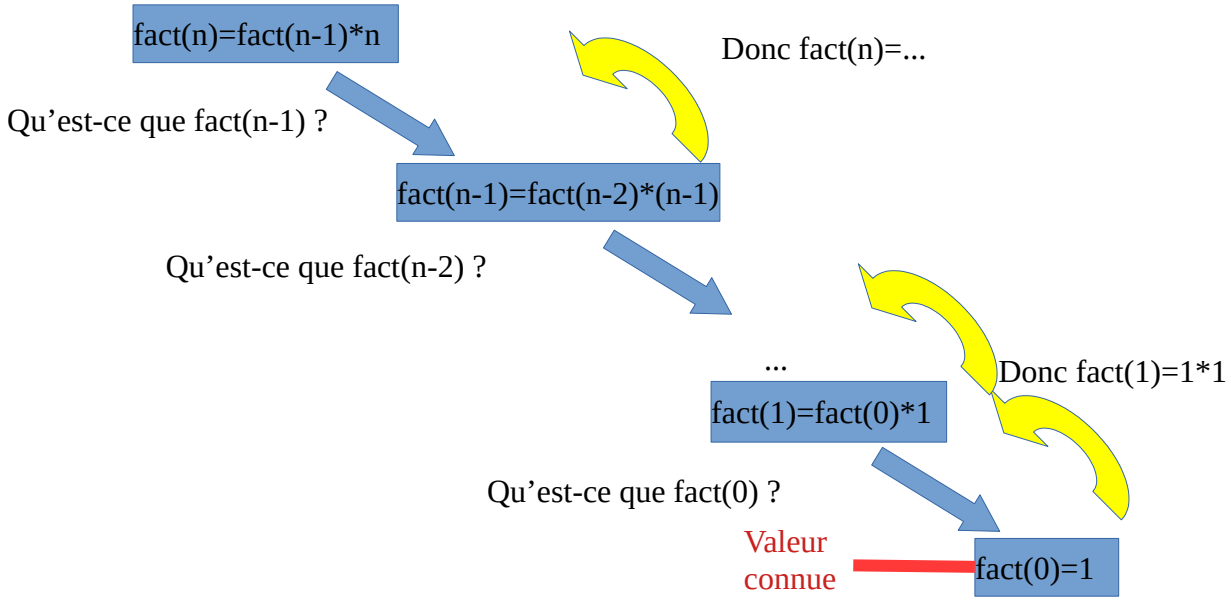
```
2)  def fact_recur(n):
        if n==0:
            return 1
1)      elif n>0:
            return fact_recur(n-1)*n
```

La factorielle de n est notée n ! où, par ex, si n=5, 5 ! = 5x4x3x2x1.
Ouais bon sinon n ! est u_n dans u_n = n x u_{n_1}, ou encore *n! = n x (n-1)! , avec 0! = 1.*

Deux concepts clés:
1) Comme j'ai dit, la fonction doit s'appeler d'elle-même. Ici, elle s'appelle avec un nouvel argument, n-1 au lieu de n, mais ce n'est pas forcément le cas.
2) Il faut nécessairement un stoppeur, une valeur donnée et connue, sinon la boucle se poursuit à l'infini et fait planter l'ordi. Ici, on a bien 0! = 1.

Ces 2 concepts sont les seuls qui régissent toute fonction récursive. Essayez de traduire des suites, des sommes, etc, en Python.

On pourrait se demander comment Python fait pour opérer ces calculs en ne connaissant la réponse qu'à un seul. Voici le schéma:



Ok, ben, voilà le programme de prépa concernant les notions de programmation Python. Je vous retrouve dans les mots de fin si l'hors-programme ne vous intéresse pas (ce sera rapide)

B) Hors-programme

C'est assez dommage que ces 2 'chapitres' soient hors-programme, mais compréhensible aussi. Le premier permet d'appliquer ses connaissances dans de vrais programmes plutôt que juste des algos, et de tripler le potentiel de Python, et le second permet une réelle compréhension des mécanismes de Python. C'est aussi dans ces deux chapitres que le Swinnen brille le plus, c'est pourquoi je le laisserai prendre la barre sans trop parler.

I. Interfaces graphiques en Python (tkinter)

Swinnen:
8. Utilisation de fenêtres et de graphismes, tout

Les interfaces graphiques sont en fait ce qui permet de faire des applications / logiciels sous Python. La plupart de ceux qu'on a utilisés en SVT, par ex, ont été (plutôt mal) codés en Python, ex: Edu Anatomist, Phylogène, peut être Anagène mais pas sûr.
Ce premier chapitre sur le module tkinter n'est qu'une initiation, vous verrez l'essentiel dans la seconde partie du prochain:

II. Programmation orientée objet, cœur de Python (les classes)

Swinnen, dans leur totalité:
11. Classes, objets, attributs
12. Classes, méthodes, héritage

Puis, classes + tkinter:
13. Classes et interfaces graphiques
14. Et pour quelques widgets de plus...
15. Analyse de programmes concrets, Jeu des bombardes

N'avez-vous jamais pensé "et si je pouvais définir ma propre classe ?"; beh voilà. Les chapitres 11 et 12 sont les mieux faits de tout le livre. Tout ce que j'ai à dire, c'est qu'il faudra vous habituer au concept d'un objet qui en contient d'autres, mais pas exactement comme dans les listes. Là, les *variables d'instance* n'existeront que pour la variable globale de classe que vous avez définie. Vous comprendrez en lisant le Swinnen.
Autre chose : n'ayez pas peur d'abuser de variables constructeurs dans __init__.

C) Mots de fin

Déjà, j'espère que ce doc vous a été utile. Perso ça m'a permis de revoir les trois quarts du programme de spé ISN (qui s'arrête en fait à celui des 2 ans de prépa).

En parlant d'informatique en prépa, faut que je vous apprenne une chose. Vous aurez peu d'heures de cours, et elle comptera pour peu dans les concours; par ex, pour le concours de la X, elle compte pour coeff 6 sur 140 pour les MP, et 10 sur 140 pour les MP option info – si je me rappelle bien lul. L'EPS est coeff 5.
Mais le temps que vous avez gagné si vous avez bossé ce doc pendant les vacances, c'est autant de temps de gagné pour les grosses matières. Par contre maintenant, faut faire des exercices du Swinnen, à fond, car ce doc, finalement, ne contient que peu de conseils de programmation même, et comporte plutôt des conseils pour apprendre à programmer, pas pareil.

J'espère que ça n'était pas trop chiant. L'informatique, à notre niveau de débutant, n'est encore qu'une technique, au mieux une application des sciences. Elle ne devient une science que bien après. En parlant de bien après: dans la plupart des écoles d'ingés, même si vous ne comptez pas vous spécialiser dans l'info, vous apprendrez d'autres langages, surtout Java et C... pour au final utiliser MathLab pour vos simulations.

Bref, merci de m'avoir suivi jusqu'au bout ~~et déso pour les éventuelles erreurs de français, j'ai dû écrire ça à des heures cheloues~~ et bon courage en prépa. Je vous laisse avec un programme que j'ai écrit sous l'impulsion d'Adrien sur la fourmi de Langton;
vidéo: https://www.youtube.com/watch?v=qZRYGxF6D3w
programme: (copiez collez ça – btw je ne recommande pas forcément d'imiter ce style de code en Java ni Python, mes habitudes ont changé depuis)

```
#Salim Najib, utf8, juillet 2019

"""La fourmi de Langton est un simple programme informatique
où une 'fourmi' reçoit deux instructions: si elle est sur une
case blanche, elle doit pivoter de 90° vers la droite puis
avancer tout en colorant la case initiale en noir; si elle
est sur une case noire, elle doit pivoter vers la gauche, puis
avancer, et colorer la case précédente en blanc.

Cependant, des comportements variés naissent de ces instructions
simples, avec, dans l'ordre:
1) des figures symétriques;
2) de l'asymétrie et du chaos;
3) "l'autoroute", un motif récurrent à l'infini.
Et cela a été vérifié dans nombre de configurations initiales,
sans encore de contre-exemple.

Ce programme est une simulation de la fourmi de Langton, permettant
d'étudier des situations initiales, des variations apportées
à la grille durant le travail de la fourmi et les altérations qui
feront varier le résultat."""

#IMPORTANT: THONNY N'EST PAS RECOMMANDE POUR CE PROG.
#Si néanmoins vous utilisez Thonny, passez en mode plein écran (au moins
#il y aura toutes les fonctionnalités).

#Moins important: d'habitude je mettrais des tests pour vérifier que Python
#reçoit bien un nombre de la part de l'utilisateur là où il demande un
#un nombre, mais là je compte un peu sur vous.

#Je voulais simplement vous montrer à quoi ressemblent des programmes plus complets
#que ceux du PDF - et si vous voulez vous amuser avec la fourmi, allez-y.

#J'exagère un peu ici mais tout bon programme doit être bien commenté (là j'en
#rajoute pour montrer des trucs bien sûr mais tout ce qui relève de notions
#communes Python n'ont normalement pas leur place. Dans les commentaires, vous
#devez plutôt expliquer le rôle de fonctions / classes / variables / boucles,etc.
```

```
#que vous jugez un minimum intéressantes (souvent presque toutes), la logique
#derrière, etc. Cela revient à montrer qu'on sait ce qu'on fait.

import tkinter as tk
from tkinter import ttk

class Ant(tk.Tk):
    """Définition de la fenêtre principale de la simulation."""
    def __init__(self, compteur=0, orient='nord',mode = 'first',
    X = 54, Y = 54, debug = [], side = 107, size = 5):
    #Dans le cas peu probable où la grille est trop grande pour votre écran:
    #réduisez side ci-dessus (80 par ex). Il faut au moins 56 cases pour arriver
    #à l'autoroute.
    #Supprimez les sauvegardes auparavant pour éviter les problèmes.

    #Paramétrage de la fenêtre + de qlqs variables
        tk.Tk.__init__(self) #Hors programme sur les classes.
        self.side = side #Nombre de carrés de la grille
        self.S = size #Taille des carrés

        self.compteur = compteur #0 sauf si grille chargée
        self.mode = mode #détermine s'il y a eu chargement ou non
        self.orient = orient #nord sauf grille chargée
        self.running = True #pause ou pas pause
        self.grilleaux = debug #... Longue histoire. Si vous avez étudié l'hors
        #programme et que vous ne comprenez pas l'utilité de ça, passez moi un message.

        #Instructions de routine: (hors-prog, tkinter)
        self.title('La fourmi de Langton - Main')
        self.configure(background='orange')
        self.geometry('%dx%d+%d+%d' %
          (self.side*self.S+128,self.side*self.S+3,
          self.winfo_screenwidth()/2-(self.side*self.S+100)/2,
          self.winfo_screenheight()/2-self.side*self.S/2-30))
        self.can = tk.Canvas(self,width=self.side*self.S,height=self.side*self.S,
          bg='orange',highlightbackground='orange')
        self.can.grid(row=1,column=2,rowspan=40)
        self.focus_force()

    #Création de la grille
        if self.mode == 'first': #Cas grille sans sauvegarde
            self.X = self.Y = self.side//2 + 1
            self.grille=[] #On crée une liste vide à laquelle on ajoute des éléments
            for i in range(self.side): #avec append, qui s'avèrent être des sous-
                self.grille.append([]) #listes vides auxquelles on ajoute des 1
                for j in range(self.side): #pour des cases blanches.
                    self.CreateCase(i,j,self.S,'white','no')
                    self.grille[i].append(1)
            #La grille se présente ainsi: liste[abscisseX][ordonnéeY],
            #il y a (nombre de cases par côté) self.side sous-listes d'abscisses
            #contenant self.side ordonnées: on a donc créé un tableau ou grille
            #dans laquelle nous pouvons situer les cases et la fourmi.

        elif self.mode != 'first': #Cas grille chargée
            self.X = X #on prend les valeurs X,Y chargées pour la localisation
            self.Y = Y #de la fourmi
            self.grille = self.grilleaux #alias, sans problème
            for i in range(self.side):
                for j in range(self.side):
                    if self.grille[i][j] > 0:
                        self.CreateCase(i,j,self.S,'white','no')
                    elif self.grille[i][j] == 0:
                        self.CreateCase(i,j,self.S,'black','no')
        #La grille était ici déjà pleine, ne restait plus qu'à créer une case noire
        #à l'emplacement de chaque 0 et une blanche pour les 1.

        #Case initiale de la fourmi
        self.ini=self.can.create_rectangle(self.X*self.S,
        self.Y*self.S,(self.X+1)*self.S,
        (self.Y+1)*self.S,fill='red') #centre, par défaut

    #Lecture des sauvegardes
        try:

            self.saves=[]
            self.fichier = open('SavesFourmi.txt','r')
            self.savesaux = self.fichier.readlines()
            try:
                for i in range(len(self.savesaux)):
                    if self.savesaux[i].split()[0] == 'Title':
                      self.saves.append(self.savesaux[i].split()[1:])
                #On parcourt le fichier txt de sauvegardes à la recherche de tous
                #les titres, qu'on place tous dans une liste qui sera affichée
                #dans un menu déroulant.
            except:
                pass

        except: #au cas où le fichier n'a pas été créé
            self.fichier=open('SavesFourmi.txt','w')

    #Boutons, étiquettes, etc.
        tk.Label(self,text='Nombre de mvmts?',bg='orange').grid(row=1,
          column=1)
        self.mvmts=tk.StringVar(self,value='15000')
        tk.Entry(self,textvariable=self.mvmts).grid(row=2,column=1)

        tk.Button(self,text='Lancer!',command=self.Go).grid(row=5,
          column=1)

        tk.Label(self,text='Compteur:',bg='orange').grid(row=7,
```

```
      column=1)
    self.compteurlab=tk.Label(self, text=str(self.compteur),bg='orange')
    self.compteurlab.grid(row=8,column=1)

    tk.Button(text='Effacer la grille',command=self.Reset).grid(row=10,
      column=1)

    tk.Button(text='Pause/Reprendre',command=self.Stop).grid(row=12,column=1)

        #Du ttkinter, oui 2 t. Puissiez-vous ne jamais avoir à en utiliser.
    self.style_combobox = ttk.Style()
    self.style_combobox.theme_create(self.style_combobox,parent='alt',
      settings = {'TCombobox':
      {'configure':
      {'fieldforeground': 'white','foreground': 'black',
       'fieldbackground': 'white','background': 'white'}}})
    self.style_combobox.theme_use(self.style_combobox)
    self.choix_orient = tk.StringVar()
    tk.Label(self,text='Orientation initiale:',bg='orange').grid(row=14,
      column=1)
    self.orient_combob = ttk.Combobox(self)
    self.orient_combob.config(textvariable=self.choix_orient,
    values=['nord','est','ouest','sud'])
    self.orient_combob.grid(row=15,column=1)
    self.orient_combob.current(0)

    tk.Button(text='Afficher la fourmi',command=self.Know).grid(row=17,column=1)
    self.releve_lab=tk.Label(text='Orientation actuelle:',bg='orange')
    self.releve_lab.grid(row=18,column=1)
    self.releve=tk.Label(text=str(self.orient),bg='orange')
    self.releve.grid(row=19,column=1)

    self.choix_nomsave = tk.StringVar()
    tk.Label(text='Nom sauvegarde?',bg='orange').grid(row=21,column=1)
    tk.Entry(textvariable=self.choix_nomsave).grid(row=22,column=1)

    self.savebutton=tk.Button(text='Sauvegarder la grille',command=self.Save)
    self.savebutton.grid(row=23,column=1)

    tk.Button(text='Charger une sauvegarde',command=
      lambda mod='load':self.Find(mod)).grid(row=25,column=1)

    tk.Button(text='Supprimer une sauvegarde',command=
      lambda mod='del':self.Find(mod)).grid(row=26,column=1)

    self.charge = tk.StringVar()
    self.saves_combob=ttk.Combobox(textvariable=self.charge,
      values=self.saves)
    self.saves_combob.grid(row=27,column=1)
    try:
        self.saves_combob.current(0)
    except:
        pass

    tk.Button(text='Aides et Astuces',command=self.Help).grid(row=37,
      column=1)

    tk.Label(text='Vitesse en ms?',bg='orange').grid(row=38,column=1)
    self.vit=tk.StringVar(self,value='2')
    tk.Entry(textvariable=self.vit).grid(row=39,column=1)

#Associations des clics de souris pour le changement de la grille
#(emplacement initial + cases noires/blanches ajoutées)
    self.can.bind('<Button-3>',self.ChangeLocIni)
    self.can.bind('<Button-1>',self.InvertColor)

    self.mainloop()

def CreateCase(self,X,Y,S,color,alter='yes'):
    """Méthode pour la création d'une case et l'altération, s'il y a lieu, de
    la liste grille."""
    if alter == 'yes': #alter permet d'utiliser cette méthode pour les cases
        self.grille[X].pop(Y) #rouges initiales + chargements.
    self.carre=self.can.create_rectangle(X*S,Y*S,(X+1)*S,(Y+1)*S,fill=color)
    if alter == 'yes':
        if color == 'white':
            self.grille[X].insert(Y,1)
        elif color == 'black':
            self.grille[X].insert(Y,0)

def ChangeLocIni(self,event):
    """Méthode pour choisir la case de départ."""
    if event.x > self.S*self.side or \
    event.y > self.S*self.side:
    #on ignore les clics hors grille
        pass
    else:
        #Suppression de la localisation précédente
        self.CreateCase(self.X,self.Y,self.S,'white')

        #Repérage et représentation de la case choisie
        self.X = int(event.x/self.S) #event.x et y sont les coordonnées du
        self.Y = int(event.y/self.S) #clic.
        self.CreateCase(self.X,self.Y,self.S,'red','no')

def InvertColor(self,event):
    """Méthode pour changer la couleur d'une case à la souris."""
    self.subX = int(event.x/self.S)
    self.subY = int(event.y/self.S)
```

```
        if self.grille[self.subX][self.subY] > 0:
            self.CreateCase(self.subX,self.subY,self.S,'black')

        else:
            self.CreateCase(self.subX,self.subY,self.S,'white')

    def Go(self):
        """Méthode de lancement de la silumation."""
        self.move=int(self.mvmts.get())
        self.XY = self.grille[self.X][self.Y] #XY est la position de la fourmi
        self.vitint=int(self.vit.get())       #sur la grille; (abs,ord).
        self.orient=self.choix_orient.get()
        #On récupère toutes les valeurs utiles, qu'elles soient chargées ou entrées
        self.Go2() #à la main.

    def Go2(self):
        """Méthode auxiliaire récursive de la simulation."""
        if self.move and self.running:
            try:
                if self.XY > 0: #case blanche
                    self.CreateCase(self.X,self.Y,self.S,'black')
                    if self.orient == 'sud':
                        self.X = self.X-1 #axe des X de gauche (0) à droite, comme
                        self.orient = 'ouest' #d'habitude (sans abscisses négatives)
                    elif self.orient == 'nord':
                        self.X = self.X+1
                        self.orient = 'est'
                    elif self.orient == 'ouest':
                        self.Y = self.Y-1 #axe des Y de haut (0) en bas, repère
                        self.orient = 'nord' #donc indirect (par défaut dans
                    elif self.orient == 'est': #tkinter).
                        self.Y = self.Y+1
                        self.orient = 'sud'

                else:
                    self.CreateCase(self.X,self.Y,self.S,'white')
                    if self.orient == 'nord':
                        self.X = self.X-1
                        self.orient = 'ouest'
                    elif self.orient == 'sud':
                        self.X = self.X+1
                        self.orient = 'est'
                    elif self.orient == 'ouest':
                        self.Y = self.Y+1
                        self.orient = 'sud'
                    elif self.orient == 'est':
                        self.Y = self.Y-1
                        self.orient = 'nord'
                #Si la fourmi regarde vers le haut ou vers le bas (de l'écran),
                #sa droite et sa gauche seront l'est ou l'ouest selon le cas.
                #Si elle regarde vers la droite ou la gauche de l'écran, ce seront
                #le nord ou le sud.

                self.XY = self.grille[self.X][self.Y]
                #Mise à jour de la position de la fourmi.

                self.move = self.move - 1
                self.compteur = self.compteur + 1
                self.compteurlab.configure(text=str(self.compteur),bg='orange')
                self.releve.configure(text=self.orient,bg='orange')
                self.after(self.vitint,self.Go2) #récursivité tkinter (hors-prog)

            except:
                pass

    def Reset(self):
        """Méthode pour effacer et redémarrer l'application."""
        self.destroy()
        Ant()

    def Stop(self):
        """Méthode pour mettre en pause/relancer la simulation."""
        if self.running == True:
            self.running = False
        elif self.running == False:
            self.running = True
            self.Go()

    def Help(self):
        """Méthode pour faire apparaître la page d'aides."""
        self.destroy()
        Aides()

    def Know(self):
        """Afficher la fourmi en permanence consomme des ressources et est peu
        utile. Cette méthode permet de connaître sa position et orientation
        quand on le désire."""
        self.running=True
        self.Stop()
        self.CreateCase(self.X,self.Y,self.S,'red','no')

    def Save(self):
        """Méthode pour sauvegarder une situation (grille/compteur/etc.)."""
        self.fichier.close()
        self.fichier = open('SavesFourmi.txt','a')
        self.temp=self.choix_nomsave.get()
        if not self.temp:
            self.temp = 'Le néant' #nom par défaut si rien n'est entré
        self.fichier.writelines('Title '+self.temp+'\n')
        for i in self.grille:
```

```
            self.fichier.writelines(str(i)+'@\n')
        self.fichier.writelines(str(self.compteur)+'@\n'+str(self.X)+'@\n'+\
          str(self.Y)+'@\n'+self.orient+'@\n')
        self.fichier.close()
        self.Reset()
    #On copie dans le fichier: le nom, la grille, le compteur, X, Y, l'orientation.

    def Find(self,mod):
        """Méthode qui trouve un nom de sauvegarde et agit en fonction."""
        for i in range(len(self.savesaux)):
            try:
                if self.savesaux[i] == 'Title '+self.charge.get()+'\n':

                    if mod == 'del': #delete
                        for j in range(i,i+self.side+5):
                            self.savesaux.pop(i)
                        #On enlève toute trace de la sauvegarde de la liste aux,
                        #puis on réécrit tout le document avec ce qui reste dans
                        #la liste. Il y a d'autres moyens mais bref.

                        self.fichier.close()
                        self.fichier=open('SavesFourmi.txt','w')
                        self.fichier.writelines(self.savesaux)
                        self.fichier.close()
                        self.Reset()

                    elif mod == 'load': #chargement
                        self.grilleaux=[]
                        #Vous devez pouvoir expliquer ce qui suit si vous avez
                        #bossé vos listes lul. Jusqu'à la ligne '####...'.
                        #Regardez aussi comment une sauvegarde se présente dans
                        #le fichier txt.
                        for j in range(i+1,i+self.side+1):
                            self.grilleaux.append(
                            self.BecomeList(self.savesaux[j].split('@')[0]))
                            #MERCI LE TYPAGE DYNAMIQUE
                        self.compteuraux = int(self.savesaux[i+self.side+1].split(
                        '@')[0])
                        self.Xaux = int(self.savesaux[i+self.side+2].split('@')[0])
                        self.Yaux = int(self.savesaux[i+self.side+3].split('@')[0])
                        self.orientaux = self.savesaux[i+self.side+4].split('@')[0]
                        ####...
                        self.fichier.close()
                        self.destroy()
                        Ant(self.compteur,self.orientaux,'NO',self.Xaux,self.Yaux,
                        self.grilleaux)

            except:
                pass

    def BecomeList(self,string):
        """Méthode qui convertit tous les nombres présents dans un liste enfermée
        dans un str."""
        liste_nombres=[]
        for i in string:
            try:
                liste_nombres.append(int(i)) #qui échouera pour tout ce qui n'est
            except: #pas un nombre, on n'aura alors que des nombres dans la liste.
                pass
        return liste_nombres
    #MERCI LE TYPAGE DYNAMIQUE

class Aides(tk.Tk):
    """Petite classe pour définir une fenêtre avec aides et astuces."""
    def __init__(self):
        tk.Tk.__init__(self)
        self.title('La fourmi de Langton - Aides')
        self.configure(background='orange')
        self.geometry('%dx%d+%d+%d' %
          (450,480,
          self.winfo_screenwidth()/2-450/2,
          self.winfo_screenheight()/2-480/2-20))
        self.focus_set()

        tk.Label(text='    ',bg='orange').pack()

        tk.Label(text='>Clic ou double clic gauche inverse la couleur d\'une case,\n\
 clic droit la choisit comme nouvelle position de la fourmi.',bg='orange').pack()

        tk.Label(text='    ',bg='orange').pack()

        tk.Label(text='>Vous pouvez cliquer avant et aussi PENDANT une\n\
 simulation. N\'oubliez pas le bouton de pause.',bg='orange').pack()

        tk.Label(text='    ',bg='orange').pack()

        tk.Label(text='>Une manière plutôt intéressante de procéder sans\n\
 perdre de temps est d\'aller à vitesse 0 jusqu\'à un\n\
 certain moment puis lancer de 10 en 10 par ex, dans le\n\
 \'Nombre de mvmts\'.',bg='orange').pack()

        tk.Label(text='    ',bg='orange').pack()

        tk.Label(text='>N\'oubliez pas de sauvegarder les situations intéressantes.',
        bg='orange').pack()

        tk.Label(text='    ',bg='orange').pack()

        tk.Label(text='>Sur une grille initialement totalement blanche, le\n\
 dernier motif symétrique est fait après 472 mouvements\n\
```

```
et l\'autoroute débute après 10.087 mouvements.',bg='orange').pack()

        tk.Label(text='    ',bg='orange').pack()

        tk.Label(text='>Attention quand le nombre de mvmts désiré est\n\
atteint: réappuyer sur Lancer continuera la simulation\n\
mais avec celle dans le champ \'Orientation initiale\',\n\
veillez donc à mettre celle "actuelle" à sa place, sauf\n\
bien sûr si vous cherchez à changer l\'orientation.\n\
De même quand vous continuez une simulation chargée.',bg='orange').pack()


        tk.Button(text='Retour',command=self.Quit).pack(side='bottom')

        self.mainloop()

    def Quit(self):
        self.destroy()
        Ant()

if __name__=='__main__':
    Ant()
```

Les vrais mots de fin (version remaniée EPFL):

D'abord remerci pour ton attention! J'espère que ça n'était pas trop chiant – et que ce sera utile. J'annonce tout de suite les principales différences entre Python et Java:
-le typage est dynamique en Python, statique/fort en Java; cela veut dire que pour initialiser une variable de type entier en Java on ne marquera pas que "x = 5" mais "int x = 5" (int comme integer); ça a des conséquences profondes concernant les tableaux/arrays Java, équivalents moins pratiques des listes Python
-ces tableaux, d'ailleurs, sont non mutables en Java: on peut modifier les index individuels d'un tableau mais pas le tableau lui-même (par ex sa longueur)
Pas de panique, vous aurez tout le loisir de revoir en détail et d'expérimenter avec ces nuances en semestre, tout en s'en rendant mieux compte en faisant la comparaison avec Python.
En parlant de Python, vous réutiliserez quasi-forcément ce langage en 2ème et 3ème années, voire au moins juste pour le fun.

En IC, la prog de BA1, c'est un équilibre à trouver entre pas la bosser du tout et trop s'y mettre aux dépens des autres cours car elle ne compte que pour le bloc 2 et n'aidera donc en rien à passer le semestre. La travailler en série + en cours + 2 heures en dehors, ça devrait largement suffire, sauf si les "mini" projets sont de retour au menu du semestre ; ce seront deux moments où il faudra s'activer en prog, mais surtout, n'oubliez pas les branches principales.
La difficulté principale de l'exam viendra du manque de temps. Ce sera plus un test de réflexes de programmation que de réflexion profonde, même s'il faudra faire attention, surtout sur les points de programmation orientée objet.

Bref, pour l'instant, profitez de ce qui vous reste d'été, car 14 semaines sans vacances, c'est un vrai choc. Les premières semaines seront plutôt lights, le temps de vous habituer et de trouver votre rythme de travail, mais dès la semaine 6, puis encore plus à partir de la 9, ça rigole plus. Je te souhaite plein de courage et d'apéros de coaching pour ce semestre :)

~Salim Najib salim.najib@epfl.ch